A NOVEL FRAMEWORK FOR TEST DATA GENERATI ON USING GENETIC ALGORI THIM

BINDHYACHAL KUMAR SINGH*, ARUN SOLANKI*, ASHISH KUMAR#

*School of Information and Communication Technology
Gautam Buddha University
Greater Noida, India
#Institute of Technology & Management
Meerut, India
bindhyachalsiwan@gmail.com asolanki@gbu.ac.in ashishtomar22@gmail.com

ABSTRACT

Software testing is one of the best methods to increase the confidence of the programmers in terms of correctness, quality and reliability of software. The testing effort is divided into three parts: test case generation, test execution, and test assessment. The goal of software testing is to design a set of minimal number of test data such that it reveals as many faults as possible. This paper proposed a framework for generating efficient test data for path testing using genetic algorithm (GA). The proposed framework accepts a program unit files and then generate a control flow diagram (CFG). The CFG is used for identification of paths in a detailed form with the help of path selection criteria. The identified path is used to select a target path using probability theory. Finally the selected path and execution of genetic algorithm are used to generate the final test data. The proposed framework also provides a facility for evaluation of generated test data to ensure their effectiveness in terms of coverage and efficiency. The proposed framework reduces time, effort and also increases the quality of generated test data. This paper is organized into three parts: part I discuss the applicability of genetic algorithm in software testing, part II describes the proposed framework, part III presents the experimental case study.

Keywords: *Test Data Generation, Cyclomatic complexity, Genetic Algorithms, Path testing, Fitness Function.*

I. INTRODUCTION

Todayøs, software quality and reliability are the main challenges for software products. Software testing is an important and valuable part of the software development life cycle. According to Myers õSoftware Testing is the process of executing a program with the intent of finding errorsö [*G. J. Myers, 1979*]. Testing remains the truly effective means to assure the quality of a software system of non-trivial complexity [*E. F. Miller, 2001*] as well as one of the most intricate and least understood areas in software engineering [*J. A. Whittaker, 2000*]. Software testing is the process of verification and validation of the software product. Effective software testing will contribute to the delivery of reliable and quality oriented software product, more satisfied users, lower maintenance cost, and more accurate and reliable result. The importance of testing can be understood by the fact that õaround 35% of the elapsed time and over 50% of the total cost are expending in testing programsö. Software is expected to work, meeting customerøs changing demands, first time and every time, consistently and predictably [*B. Beizer, 1990*]. Software needs to be tested properly and thoroughly. Such that anything that goes wrong can be detected and fixed in advance, before delivery to the user. However, even well tested software is not guaranteed to be bug-free. If the testing process could be automated, it would be reduce the cost of software development significantly. Automatic test data generation is a key problem in software testing and its implementation can not only significantly improve the effectiveness and efficiency but also reduce the high cost of software testing [*I. Premal, et al., 2011*].

Genetic algorithms have been used to find automatically a program s longest or shortest execution times. The critical point of the problem involved in the automation of software testing is the automation of software test-data generation [*B. K. Singh, 2012*]. Genetic algorithms that can automatically generate test data to selected path. This algorithm takes a selected path as a target and executes sequences of operators iteratively for test cases to evolve. The evolved test case can lead the program execution to achieve the target path for path testing [*I. Premal, et al., 2011*]. A dynamic test data generation approach which is based on a path wise test data generator to locate automatically the values of input variables for which a selected path is traversed. His steps are program control flow graph construction and test data generation. The path selection stage is not used because if unfeasible paths are selected, the result is a waste of computational effort examining these paths [*B.*

Korel., 1990]. Software testing is the important means that give a guarantee for quality and reliability for S/W. The techniques of genetic algorithm as the key algorithm to automatically generating the test data, and elaborates some specific problems need to solve in realization process: such as coding, the selection of fitness function and the improvement of hereditary operator [*W. Xibo et al., 2011*]. Random test data generation consists of generating test inputs at random, in the hope that they will exercise the desired software features. Often, the desired inputs must satisfy complex condition, and this makes a random approach seem unlikely to succeed. In contrast, combinatorial optimization techniques, such as those using genetic algorithms, are meant to solve problems involving the simultaneous satisfaction of many constraints [*C. C. Michael, et al., 2001*].

II. PROPOSED FRAMEWORK FOR TEST DATA GENERATION SYSTEM

This proposed framework accepts the program unit file and generate test data for path testing. This framework perform a sequence of operations like code parsing and identification of predicates, Control Flow Graph generation (CFG), calculation of cyclomatic complexity, GA execution, test data generation and finally evaluation of effectiveness of generated test data. The proposed framework applies a hybrid path selection criterion that combines the branch coverage, condition coverage, all-basic paths coverage, and full-predicate criteria. This framework also provides a facility for evaluation of effectiveness of generated test data theoretically.

The proposed framework is composed of seven main modules as shown in Fig. 1
Module 1: Code Parsing and Predicate Identification
Module 2: Cyclomatic Complexity Calculator
Module 3: Control Flow Graph Generator
Module 4: Path Selector using Path selection Criterion
Module 5: Genetic Algorithm Execution
Module 6: Test Data Generation
Module 7: Evaluation of effectiveness of Generated Test Data

Module 1: Code Parsing and Predicate Identification

This module accepts a program unit files (PUF) as an input. Accepted PUF is used to analyze text or code and made a sequence of tokens to determine the grammatical structure with respect to given PUF. Code parsing is also called syntactic analysis of a module. Code parser is one of the components that check the syntax and it uses separate lexical analyzer to create tokens from the sequence of inputs characters. Predicate identification is done using symbolic execution. A predicate node can be identified if it contains conditional test statement. It can also be identified using CFG in following manner. õA node contains two or more branch then it is predicate nodeö. But we can use this method when CFG is available priory. The output of this module is a set of identified predicate nodes from accepted PUF.

Module 2: Control Flow Graph Generator

Control flow graph describes the logical structures of the program unit. It builds from a set of nodes and edges. It is a directed graph that shows the all possibility for the flow of control through the PUF file from the source to sink node. There are many flow of control through the program that guaranteed the various paths. This module generates control flow graph by manually or automated tools (eclipse CFG generator) and provide a logical structure of PUF in the form of graphical representation. The non executable statements like comments, variables and its declaration are not considered for CFG. All decision making statements and looping statements are represented by separate nodes. The output of this module is a control flow graph.

Module 3: Cyclomatic Complexity Calculator

Cyclomatic complexity (CC) is software metric that was developed by Thomas J. McCabe in 1976. It is used to measure the no. of linearly independent path through a programøs source code. It also determines the minimum number of test cases for a program unit files which covers each path of CFG [*R. Mall, 2009*]. A cyclomatic complexity can be calculated using either CFG or predicate nodes in a PUF.

If we use CFG for calculation of CC then there are three parameters required. First is No. of nodes (N), No. of edges (E) and No. of unconnected graph (P). It is denoted by V (G) [*R. Mall, 2009*].

$$V(G) = E-N+2P$$

If we use predicate nodes then firstly count the no. of predicates (D) in a given source code and after that use following formula [*R. Mall, 2009*]

$$V(G) = D+1$$

Another way can be used for calculation of cyclomatic complexity is count the no. of bounded region (B) and use following formula [*R. Mall, 2009*]

$$V(G) = B+1$$

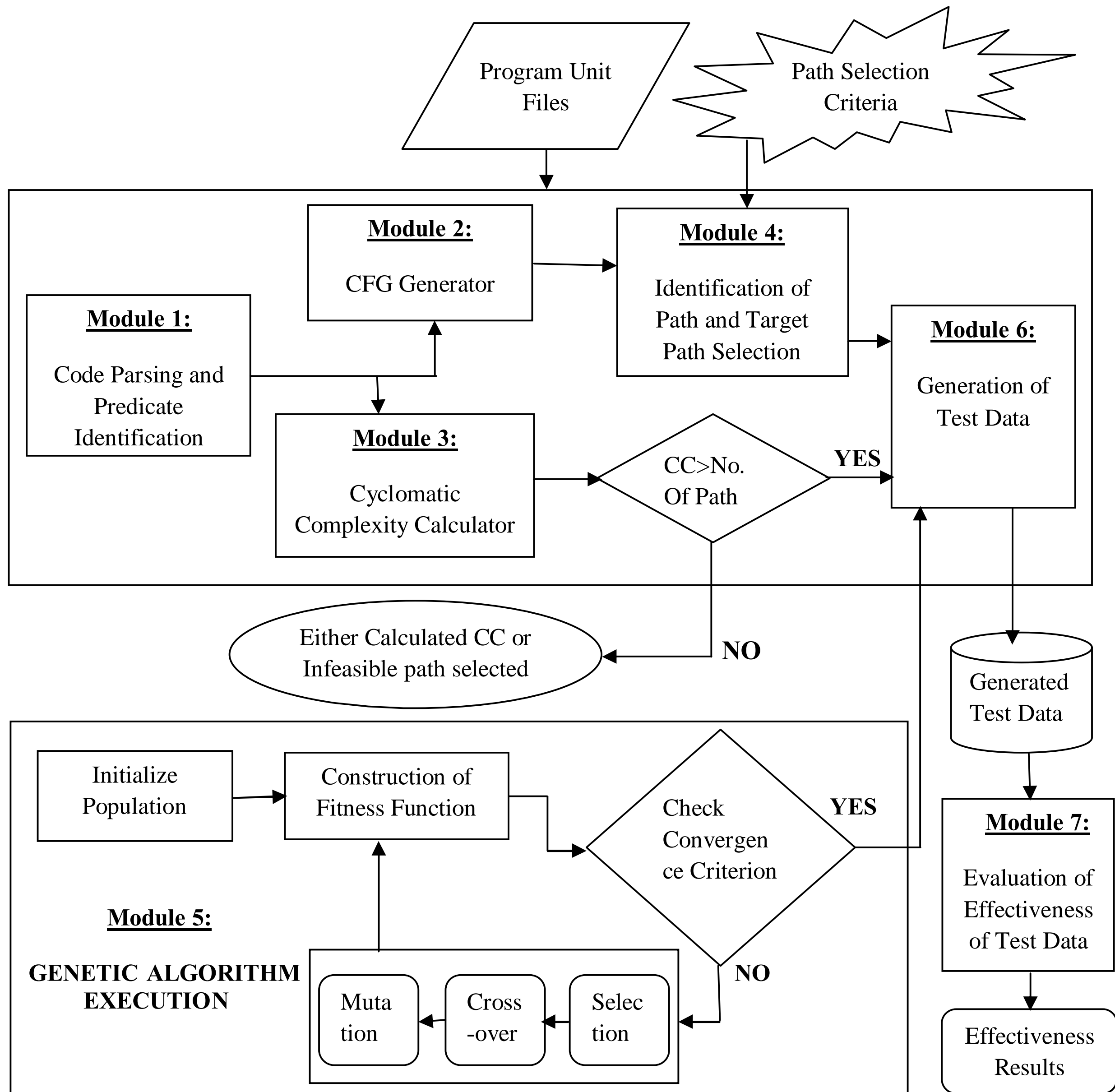

Fig. 1: Framework for Test Data Generation System

In this case system uses the predicate nodes for calculation of CC. Predicates in java code may be if, else, while, try, catch, for, switch, break, continue etc. Firstly system read all code using file reader, code analyzer and string tokenizer and counts the no. of predicates in a java PUF. A CC is the upper bound for branch coverage to determine the no. of test cases that are necessary. Besides, it is a lower bound for path coverage to determine the

no. of test case which executes all feasible paths [*N. B. Pakinam et al., 2011*]. The output of this module is an integer that indicates the cyclomatic complexity of given PUF.

Module 4: Path Selector using Path selection Criterion

The main aim of this module is to identify all possible feasible paths from CFG and after that select a target path. This module takes a CFG of given PUF as an input and apply path selection criterion for selection of paths. Path selection criterion may be selected all paths, statement coverage, condition coverage, branch coverage etc. The greatest advantage of a path selection is to ensure that all program constructs are executed at least once. Another advantage is to avoid redundancy in a selected path. This process is more complex than others. If selection of path is not effective then it leads to a wrong results and duplicate test cases. After selection of all paths, we need to select a target path. Target path is a path that ensures the execution of all paths if it is executed. According to probability theory, the target path is selected and selected target path will become convergence criterion [*I. Premal, et al., 2011*].

Module 5: Genetic Algorithm Execution

Genetic Algorithm starts with a set of initial individuals as the first generation, which is sampled random from the problem domain. This algorithm is developed to perform a series of operations that transform the present generation into a new fitter generation [*Y. Chen1 et al., 2008*]. Each individual in each generation is evaluated with a fitness function. There are three primary operators are used to perform GA operations. First is selection operator which is used to choose chromosome from population for mating. An individual with a higher ranking is given has a greater probability for reproduction. The fitter individuals are allowed a better survival chance from one generation to the next. Second is crossover operator which is practically a method for sharing information between two chromosomes. In our case single crossover is used. Third operator is mutation which is used to alter one or more values of the allele in the chromosome in order to increase structural variability [J. *H. Holland, 1975*]. The key term in a GA execution is the evaluation of fitness function for every individual. In each iteration of GA generates a generation of individuals and for every generation of individuals it generates a test data. The output of this module is fitness function for every path and next generation results. Twelve combination of operator can be applied to make more effective and efficient GA based algorithm for test data generation.

Table 1: Combination of Operators

OPERATORS / S. No.	SELECTION						CROSSOVER	
	RWS	**RS**	**SSS**	**ES**	**SUS**	**SRS**	**SP**	**MP**
1.	Y	-	-	-	-	-	Y	-
2.	Y	-	-	-	-	-	-	Y
3.	-	Y	-	-	-	-	Y	-
4.		Y	-	-	-	-	-	Y
5.	-	-	Y	-	-	-	Y	-
6.	-	-	Y	-	-	-	-	Y
7.	-	-	-	Y	-	-	Y	-
8.	-	-	-	Y	-	-	-	Y
9.	-	-	-	-	Y	-	Y	-
10.	-	-	-	-	Y	-	-	Y
11.	-	-	-	-	-	Y	Y	-
12.	-	-	-	-	-	Y	-	Y

RWS: Roulette Wheel Selection **RS**: Rank Selection **SSS**: Steady-State Selection **ES**: Elitism Selection **SUS**: Stochastic Universal Sampling **SRS**: Stratified Random Sampling **SP**: Single Point **MP**: Multi-Point

Module 6: Test Data Generation

This module generates test data using module 5 results and identified path. GA is used for test data generation because the greatest merit of GA in path testing is its simplicity. The main goal of this module is to generate a set of test data for every path.

Module 7: Evaluation of effectiveness of Generated Test Data

This module is used for check the effectiveness of generated test data in terms of defect found, percentage of execution coverage. It compares generated test data with previous results which is based on GA, Symbolic execution and random test data generator. Test data effectiveness can be calculated as: [*R. P. Mahapatra et al., 2008*].

$$\text{TDG effectiveness} = (N_{TD} / N_{Total})*100$$

Finally the proposed framework model will be useful got generation of test data for path testing using GA. It can always generate better results and helpful to reduce the time required for lengthy testing activity.

III. EXPERIMENTAL CASE STUDY

A. Input: Program Unit Files (Triangle Problem)

Triangle classification problem (TCP) has been widely used in the area of software testing. It determines the type of triangle if three input sides can form a triangle.

```
int triangletype(int a, int b, int c)
{
System.out.println("Enter three Sides");
a=Integer.parseInt(in.readLine());
b=Integer.parseInt(in.readLine());
c=Integer.parseInt(in.readLine());
if ((a>0 && b>0 && c>0) && (a+b>c && a+c>b && b+c>a))
{
if ((a==b) && (b==c))
{
System.out.println("EQUILATERAL TRIANGLE");
}
else if((a!=b) && (b!=c) && (a!=c))
{
System.out.println("SCALENE TRIANGLE");
}
else
{
System.out.println("ISOCELES TRIANGLE");
}
}
else
{
System.out.println("NOT A TRIANGLE");
}
}
```

B. Control flow Graph:

A control flow graph {fig. 2} of a given TCP contains different paths from entry node to exit node. A path that begins with the entry node and ends with the exit node is called a complete path. Otherwise it is called incomplete path or a path segment. Complete path may be feasible or infeasible. Let S = {S1, S2, S3.......Sn} and T = {T1, T2, T3.....Tn} be two paths then ST = {S1, S2, S3.....Sn, T1, T2, T3....Tn} denotes the concatenation of S and T. Let first(S) and last(S) denotes the first node S1 the last node Sn of path S

respectively. We can say that two paths connect if (last(S), first (T)) þ E, where E is a set of all edges [*J. Edvardsson, 1999*].

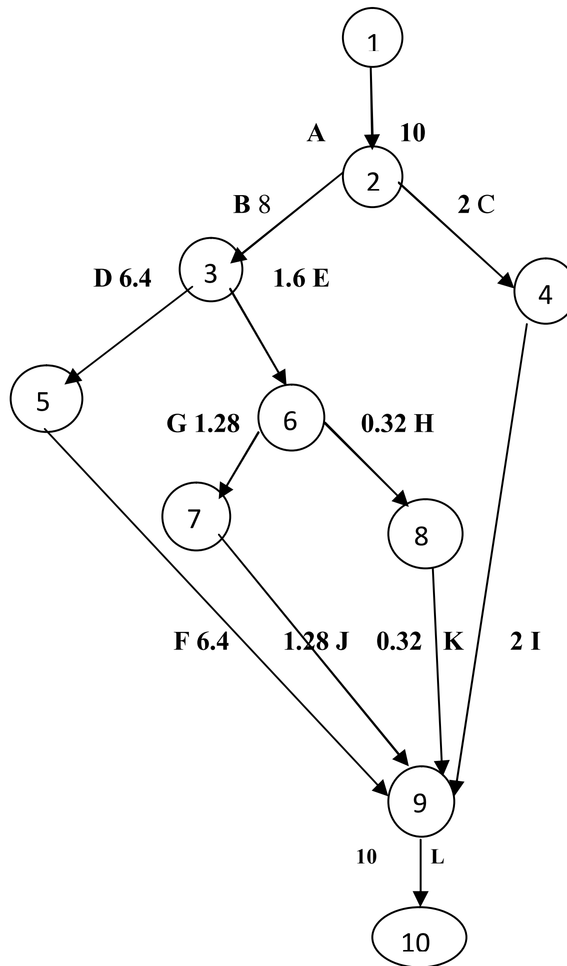

Fig. 2: A Control Flow Graph for TCP

C. Cyclomatic Complexity:

Calculated cyclomatic complexity indicates upper bound of a linearly independent path in a program. It also gives the minimum no. of test cases required to achieve path coverage.

No. of Nodes = 10
No. of Edges = 12

So,
Cyclomatic complexity [V (G)] = 12-10+2 = 4

It means at least 4 test cases required to achieve full path coverage and also at least 4 linearly independent path exist in PUF.

D. Path Identification and Weight Assigning:

We can use base lining method to identify path from CFG. In this method choose a baseline and then flip it at decision nodes. In above CFG letøs choose baseline path is

Path1 (P1): 1-2-4-9-10 (Baseline Path) Pure False Condition
Edges for path 1 [E1]: {A, C, I, L}

Path2 (P2): 1-2-3-5-9-10 (node 2 is flipped) Semi False Condition
Edges for path 2 [E2]: {A, B, D, F, L}

Path3 (P3): 1-2-3-6-7-9-10 (node 3 is flipped) Semi False Condition
Edges for path 2 [E3]: {A, B, E, G, J, L}

Path4 (P4): 1-2-3-6-8-9-10 (node 6 is flipped) Semi False Condition
Edges for path 2 [E4]: {A, B, E, H, K, L}

Now, weight for every edge is assigned in following manner.
õThe weight of true edge will be 80% and for false edge will be 20% of a total weight. Total weight of a node is sum of weight of all incoming edges at a particular node.ö

Weights for edges are given following:

Table 2: Weight for every edges

S. NO.	EDGES	WEIGHT
1.	A	10
2.	B	8
3.	C	2
4.	D	6.4
5.	E	1.6
6.	F	6.4
7.	G	1.28
8.	H	0.32
9.	I	2
10.	J	1.28
11.	K	0.32
12	L	10

E. Target Path Selection:

Target path is a path that guaranteed the execution of all paths and achieves full path coverage. If this path is executed then we consider that all paths had executed. For selection of target path, Firstly need to make a condition matrix which has MxN dimension where M is the total no. of feasible path and N is the total no. of decision node for a particular path.

Table 3: Condition Matrix

PATH NO.	DECISION NODE		
	2	**3**	**6**
P1	F	-	-
P2	T	T	-
P3	T	F	T
P4	T	F	F

P1: Path 1 **P2**: Path 2 **P3**: Path 3 **P4**: Path 4 **T**: True **F**: False

From table 3,

P1 (F) So, P (P1) = 1/1 = 1

P2 (TT) So, P (P2) = 0/2 = 0

P3 (TFT) So, P (P3) = 1/3 = 0.333

P4 (TFF) So, P (P4) = 2/3 = 0.667

According to probability theory, Target path will be Path 1 because it has maximum probability. If a case exist in which two maximum probabilities occurs then choose path as a target path which have more false condition.

F. Designing of Fitness Function:

After that fitness function for designed for every path. Fitness function can be calculated as

Fitness Function = Sum of weight of all edges that involves in execution for a path

For Path1: 1-2-4-9-10
Edges for path 1 [E1]: {A, C, I, L}
Fitness Function = 10+2+2+10 = 24

Path2: 1-2-3-5-9-10
Edges for path 2 [E2]: {A, B, D, F, L}
Fitness Function = 10+8+6.4+6.4+10 = 40.8

Path3: 1-2-3-6-7-9-10
Edges for path 3 [E3]: {A, B, E, G, J, L}
Fitness Function = 10+8+1.6+1.28+1.28+10 = 32.16

Path4: 1-2-3-6-8-9-10 (node 6 is flipped) Semi False Condition
Edges for path 4 [E4]: {A, B, E, H, K, L}
Fitness Function = 10+8+1.6+0.32+0.32+10 = 30.24

It is clear from above calculated fitness function that path 1 and path 2 have minimum and maximum fitness function respectively.

Table 4: Path Execution Probability and Its Fitness Function

Path	Fitness Function	P_i	C_p
P1	24	0.189	0.189
P2	40.8	0.321	0.510
P3	32.16	0.253	0.763
P4	30.24	0.237	1

$\mathbf{P_{i:}}$ Probability of Path execution $\mathbf{C_{p:}}$ Cumulative Probability

P_i can be calculated as: $P_i = F(X_i) / \hat{U} (f(X))$

According to table 4, Fitness = {30.24, 24, 40.8, 32.16} and P_i = {0.238, 0.189, 0.321, 0.252}

After that, apply single crossover operator and mutation operator to generate new test data or new population.

G. Generation of Test Data

Now, System accepts initial random population (Table 5) as an input and performs following operation to generate test data.

i. Conversion of Input Data into Binary String

TC1 can be written as: 001101000110
TC2 can be written as: 010010010010

Table 5: Initial Random Population

TestCase	Input Data (a, b, c)	Expected Outcome
TC1	3, 4, 6	Scalene Triangle
TC2	4, 9, 2	Not A Triangle

ii. Apply Crossover Operator

Parent 1(TC 1)

0	0	1	1	0	1	0	0	0	1	1	0

Single Point Crossover at K = 8

Parent 2 (TC 2)

0	1	0	0	1	0	0	1	0	0	1	0

Offspring 1

0	0	1	1	0	1	0	0	0	0	1	0

Offspring 2

0	1	0	0	1	0	0	1	0	1	1	0

So, after crossover new population becomes {(3, 4, 2), (4, 9, 6)}

iii. Apply Mutation Operator

After applying crossover, mutation operator is used to alter one or more values of the allele in the chromosome in order to increase structural variability.

Mutation Probability = 0.83

After 1st generation new population becomes {(3, **5**, **5**), (4, 9, **7**)}

Table 6: New Population and Its Fitness Function

New Test Case	Input Data (a, b, c)	Path	Fitness Function	Expected Outcome
NTC1	3, 5, 5	P4	30.24	Isoceles Triangle
NTC2	4, 9, 7	P3	32.16	Scalene Triangle

Thus system can generate test data after every generation and make testing process more efficient.

IV. CONCLUSION

In this paper, the genetic algorithms are used to generate test data for path testing. The greatest merit of using the genetic algorithm in program testing is its simplicity. The quality of test data produced by genetic algorithms is higher than the quality of test data produced by random way because the algorithm can direct the generation of test data to the desirable range fast. The proposed framework perform a sequence of operations like code parsing and identification of predicates, Control Flow Graph generation (CFG), and calculation of cyclomatic complexity, GA execution, test data generation and finally evaluation of effectiveness of generated test data. The proposed framework can generate highly efficient test data with the minimum number of steps saving time, effort and yet increasing the quality of generated test data thus improving the overall testing process performance.

REFERENCES

- B. Beizer, (1990) õSoftware Testing Techniquesö, *2nd. Edition, Van Nostrand Reinhold, USA*.
- B. Korel, (1990) õAutomated software test data generationö *IEEE Transactions on Software Engineering*, *VOL. 16*, *No. 8, pp. 870-879*.
- B. K. Singh, (2012) õAutomatic Efficient Test Data Generation Based on Genetic Algorithm for Path Testingö EARDA International Journal of Research in Engineering & Applied Sciences (IJREAS), Vol. 2, Issue 2, ISSN: 2249-3905, pp. 1460-1472, India.
- C. C. Michael, G. E. McGraw, M. A. Schatz, (2001) õGenerating Software Test Data by Evolutionö, *IEEE Transactions on Software Engineering, vol.27*, *No. 12,* pp. 1085-1110.
- E. F. Miller, (2001) õIntroduction to Software Testing Technology,ö *Tutorial: Software Testing & Validation Techniques, Second Edition, IEEE Catalogue No. EHO 180-0, pp. 4-16*.
- G. J. Myers, (1979) õThe Art of Software Testingö, *1st Edition, John Wiley and Sons, NY, USA*.
- I. Premal, B. Nirpal, K. V. Kale, (2011) õUsing Genetic Algorithm for Automated Efficient Software Test Case generation for Path Testingö, *IEEE Int. J. Advanced Networking and Applications*, *vol. 2, pp. 911-915*.
- J. A. Whittaker, (2000) õWhat is Software Testing? And Why Is It So Hard?ö *IEEE Software, pp. 70-79*.
- J. Edvardsson, (1999) öA survey on automatic test data generationö *In Proceedings of the Second Conference on Computer Science and Engineering in Linkoping (ECSEL), pp. 21-28*.
- J.H. Holland, (2008) õAdaptation in natural and artificial systemö The University of Michigan Press.
- R. Mall, (2009) õFundamentals of Software Engineeringö, 3rd *Edition, PHI, New Delhi,India*.
- R. P. Mahapatra, J. Singh, (2008)öImproving the Effectiveness of Software Testing through Test Case Reductionö *In Proceedings of World Academy of Science, Engineering and Technology (WASET), pp. 345-350*.
- N. B. Pakinam, L. B. Nagwa, M. Hashem and M. F. Tolba, (2011) õA Proposed Test Case Generation Technique Based on Activity Diagramsö *International Journal of Engineering & Technology IJET-IJENS Vol: 11 No: 03, pp. 37-57*.
- W. Xibo, S. N. Shenyang, (2011) õAutomatic Test Data Generation for Path Testing Using Genetic Algorithmsö, *IEEE Third International Conference on Measuring Technology and Mechatronics Automation (ICMTMA), pp. 596-599*.
- Y. Chen1, Y. Zhong, T. Shi1 and L. Jingyong, (2008) õComparison of Two Fitness Functions for GA-based Path-Oriented Test Data Generationö, *IEEE Fifth International Conference on Natural Computation, pp. 566-570*.