ISSN:0975-9646

Maximize the Code Coverage for Test Suit by Genetic Algorithm

[1]Mohd Athar, [2]Israr Ahmad

[1]Research Scholar, Shri Venkateshwara University, Gajraula, India

[2]Jamia Millia Islamia (Central University), New Delhi, India

Abstract— Software testing is important but it possesses some fundamental challenges. It poses two essentially arduous jobs; selecting test cases and assessing test results. Optimization problems can be unbridled by genetic algorithm (GA) which can be regarded as computer model of biological evolution. It works on principle of evolution, where superior chromosomes (having greater fitness value) are chosen for mutation and crossover operations. Evolution continues until the optimized solution is achieved. Good results are found astoundingly speedily when GA is implemented. Generating optimized test suit (TS) is meta-heuristic problem which can be resolved by GA. The only objective of programming is not to determine the algorithm to accomplish a result, but relevance and correctness of the result also requires to be ascertained. Genetic Algorithm, which is a meta-heuristic algorithm, is employed for optimizing path testing to achieve total code coverage.

Keywords— Genetic Algorithm, Software Testing, Software Under Test, Code Coverage, Test Suit.

I. Introduction

Software testing is a main method for improving the quality and increasing the reliability of software now and thereafter the long-term period future. It is a kind of complex, labor-intensive, and time consuming work; it accounts for approximately 50% of the cost of a software system development. Increasing the degree of automation and the efficiency of software testing certainly can reduce the cost of software design, decrease the time period of software development, and increase the quality of software significantly. The critical point of the problem involved in automation of software testing is of particular relevance of automated software test data generation. Test data generation in software testing is the process of identifying a set of program input data, which satisfies a given testing criterion. For solving this difficult problem, random, symbolic, and dynamic test data generation techniques have been used in the past. Recently, genetic algorithms have been applied successfully to generate test data, Khamis [2007].

Software testing is significant because failure in computer software may have severe aftermaths. Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or software under test (SUT).

Software testing can be stated as the process of validating and verifying that a computer program/application/product:

- ✓ meets the requirements that guided its design and development,
- ✓ works as expected,
- ✓ can be implemented with the same characteristics,
- ✓ and satisfies the needs of stakeholders.

Software testing has various different strategies. This explicates and gives overview of key difference between various approaches in it. Testing techniques are test cases design method. Test cases are developed using various testing techniques to achieve more effective testing of application. Following are the testing techniques, Black-box and White-box testing. Program is viewed as “black box” in black-box testing approach. In this, test cases are grounded on system specifications. White-box study internal structure of program, i.e., it utilizes control structure of the procedural design to obtain test cases. White-box testing examines internal structures or workings of an application without looking its functionality. In this, inner composition of SUT is studied. WBT is a complementary approach to BBT.

In Unit Testing (UT), individual units/components of a software/system are tested. The purpose is to validate that each unit of the software performs as designed. A single module of SUT is taken and run singly in isolation from remaining product. The intent of UT is, set apart each component of the SUT and establishes that the individual components have no error. It is comparatively less problematic to rectify the single module, as size of code is little, so errors are located easily. In Integration Testing (IT), individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units. In System Testing (ST), a complete, integrated software/system is tested. The purpose of this test is to evaluate the system’s compliance with the specified requirements. In Acceptance Testing (AT), a software/ system are tested for acceptability. The purpose of this test is to evaluate the system’s compliance with the business requirements and assess whether it is acceptable for delivery. Whenever a change in a software application is made it is quite possible that other areas within the application have been affected by this change. The intent of Regression Testing (RT) is to ensure that a change, such as a bug fix did not result in another fault being uncovered in the application. Regression Testing is, in fact, just a type of testing that can be performed at any of the above four main levels of testing.

Program testing and fault detection can be assisted significantly by testing tools. Testing tools can be put in two classes, static & dynamic.

Static testing involves verification. Static Analyzers probes programs thoroughly and automatically. These are employed on particular language, i.e., these are language dependent. Code Inspectors scrutinizes program to vouch that it hold on minimum quality criteria. Code Inspection activity is found in some COBOL tools (like AORIS librarian system).

Dynamic testing tools involve validation. These are performing analysis of programs on executing them. Coverage Analyzers finds degree of coverage. One of its e.g. CodeCover tool. CodeCover Tool is a well-known Eclipse plug-in, employed as white box coverage tool. This tool is very apposite to assure weather TS is giving full code coverage or not. Output Comparators checks weather anticipated and obtained outputs are same or not. JUnit is such a tool. JUnit Tool is Unit Testing framework for Java. It is applied for testing of single component, IT and ST. Features of JUnit are.

- ✓ test fixtures for sharing regular test data
- ✓ affirmations for testing expected results
- ✓ for running tests provides test runners

Static Analyzers and Code Inspectors are static testing tools while Coverage Analyzers and Output Comparators are dynamic testing tools.

II. MOTIVATION

Software testing is a principal technique which is employed for bettering quality attributes of software under test (SUT), particularly reliability and correctness but is also regarded to be tedious. Genetic Algorithms (GAs) have been used to automate the generation of test data for software developed in various languages. The test data were derived from the structure of program with the objective to traverse all the branches in the software. The input variables are represented in Gray code and as an image of the machine memory. The power of using genetic algorithms lies in their ability to handle input data which may be of complex structure, and predicates which may be complicated and unknown functions of the input variables. Thus, the problem of test data generation is treated entirely as an optimization problem. The Genetic Algorithms gives most improvements over random testing when these sub domains are small. Experiments show that Genetic Algorithms required less central processing unit (CPU) time in general reaching a global solution than random testing. The greatest advantage is when the density of global solutions is small compared to entire input search domain.

III. PROBLEM STATEMENT

The furtherance of basic knowledge required to develop new techniques for automatic testing. The main objective is to automate generation of test suit (TS) for each module of SUT by applying GA that could give 100% code coverage.

The performance of Genetic Algorithms in automatically generating test data for small procedures will be assessed and analyzed. A library of Genetic Algorithms will be to apply to large systems. The efficiency of Genetic Algorithms in generating test data will be compared to random testing with regard to the number of test data sets generated and the CPU time required.

IV. APPROACH

Our intent is to optimize TS which could give 100 % code coverage. This optimization which is grounded on total code coverage needs that inner composition of program is well-known. Inner composition of program can be discovered by path testing in which a set of test-paths are selected in a program. The different independent paths in the program could be determined through control flow graph (CFG). An independent path is that path in CFG that has one novel set of processing statements or novel conditions. Test cases carrying the information of the path covered by them are grouped together to form initial population of chromosomes and GA is applied. In the end, TS is obtained for each module that gives hundred per-cent code coverage. The main objective is to develop a test system to exercise all the branches of the software under test.

In order to generate the required test data for branch testing genetic algorithms and random testing are used. These two testing techniques will be compared by means of the percentage of coverage which each of them can achieve and by the number of test data.

V. METHODOLOGY

It delves into minutia of approach that is complied to reach the motive of optimizing software testing using genetic algorithm (GA). Generating test suite (TS) that guarantees full coverage of statements in program, is complex task. There are also odds that more than one test case in TS are checking same path. This redundancy is not appreciated. It is imperative to have optimized test data sets. In this section, GA is employed for optimizing path testing.

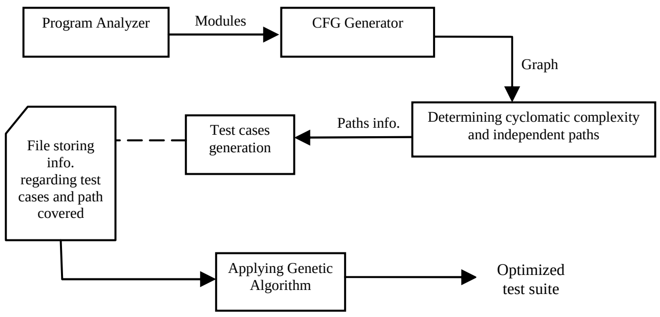

Program Analyzer
Modules
CFG Generator
Graph
Determining cyclomatic complexity and independent paths
Paths info.
Test cases generation
File storing info. regarding test cases and path covered
Applying Genetic Algorithm
Optimized test suite

Figure 1: Block Diagram of Methodology

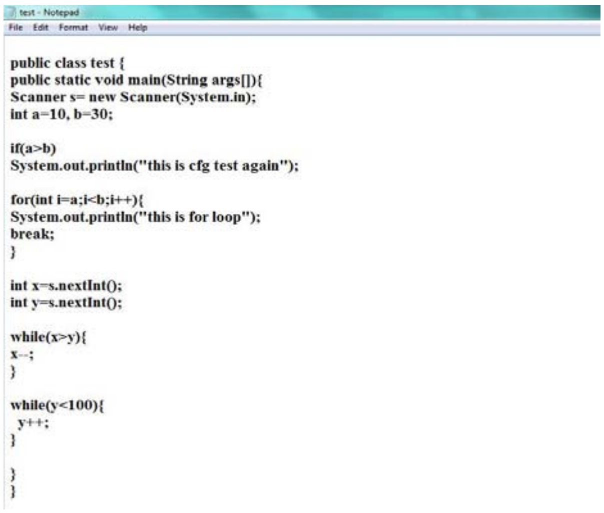
Figure 1, illustrates approach applied in this thesis to accomplish the objective. Program analyzer analyzes the java program and discovers all the modules in it. Control Flow Graph (CFG) generator generates the CFG for each module. CFG is used to find cyclomatic complexity (CC) and total independent paths. Test cases are generated and paths followed by them are found. The data regarding test cases and path followed are put in a file. This file is utilized when GA is employed. Each of the blocks is explicated fully in this chapter.

Methodology is divided into two approaches:

- ✓ Testing
- ✓ Applying Genetic Algorithm

VI. Results And Analysis

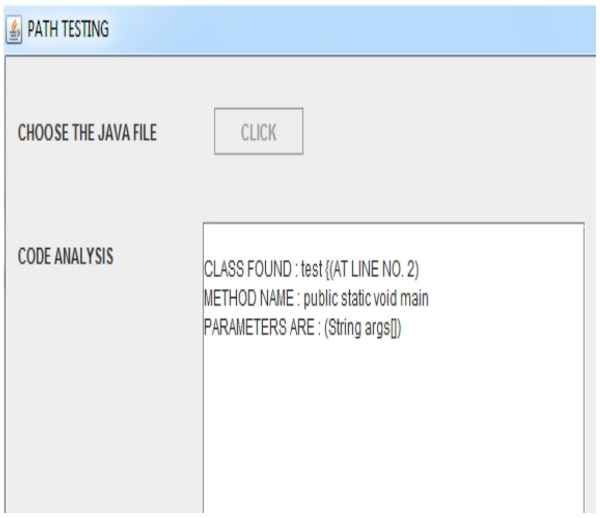
Some Sample problems are taken which are java program. Since WBT is concerned with code structure not functionality, the module is doing simple task of displaying some statements.



```
public class test {
public static void main(String args[]){
Scanner s= new Scanner(System.in);
int a=10, b=30;

if(a>b)
System.out.println("this is cfg test again");

for(int i=a;i<b;i++){
System.out.println("this is for loop");
break;
}

int x=s.nextInt();
int y=s.nextInt();

while(x>y){
x--;
}

while(y<100){
 y++;
}

}
}
```

Figure 2: Sample Problem 1

In figure 2, program is analyzed to discover the modules in it.



```
CLASS FOUND : test {(AT LINE NO. 2)
METHOD NAME : public static void main
PARAMETERS ARE : (String args[])
```

Figure 3: Output of Code Analyzer in TextArea

As depicted in figure 3, path of java file is given by clicking on “click” button. Class and methods information are displayed in “*TextArea*”. Information includes “class name”, “methods in class”, “parameters of methods”. *Code Analyzer* also writes output in a “*.txt” file, which is used to fetch line numbers at which method definition exists.

Modules find by *code analyzer* is used by CFG generator to build CFG. *CFG generator* fetches the line number from where module begins from text file generated by *code analyzer*.

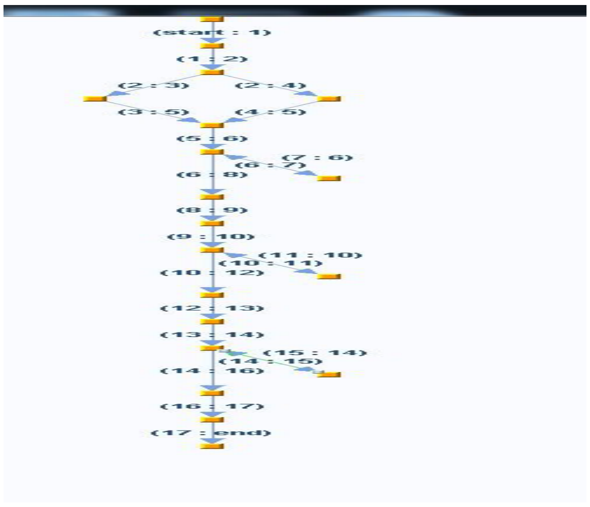
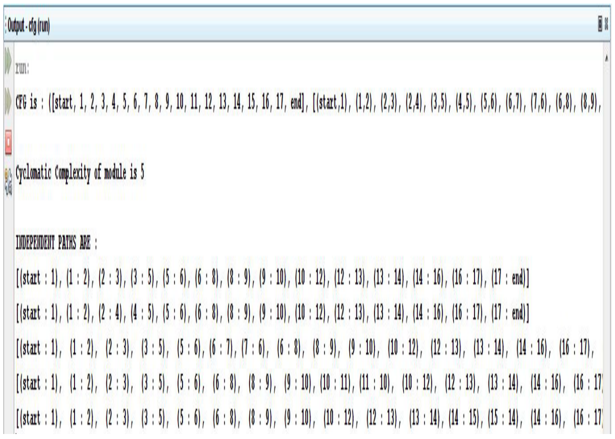


Figure 4: CFG of sample problem 1

Figure 4 shows the CFG of *main* module of sample problem. Orange buttons are vertices of CFG and arrows are edges of CFG. Arrows are labeled like “1:2” showing the flow of control from vertex “1” to vertex “2”.



```
run:
CFG is : ([start, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, end], [(start,1), (1,2), (2,3), (2,4), (3,5), (4,5), (5,6), (6,7), (7,6), (6,8), (8,9),

Cyclomatic Complexity of module is 5

INDEPENDENT PATHS ARE :
[(start : 1), (1 : 2), (2 : 3), (3 : 5), (5 : 6), (6 : 8), (8 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 16), (16 : 17), (17 : end)]
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 8), (8 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 16), (16 : 17), (17 : end)]
[(start : 1),  (1 : 2),  (2 : 3),  (3 : 5),  (5 : 6), (6 : 7), (7 : 6),  (6 : 8),  (8 : 9),  (9 : 10),  (10 : 12),  (12 : 13),  (13 : 14),  (14 : 16),  (16 : 17),
[(start : 1),  (1 : 2),  (2 : 3),  (3 : 5),  (5 : 6),  (6 : 8),  (8 : 9),  (9 : 10), (10 : 11), (11 : 10),  (10 : 12),  (12 : 13),  (13 : 14),  (14 : 16),  (16 : 17
[(start : 1),  (1 : 2),  (2 : 3),  (3 : 5),  (5 : 6),  (6 : 8),  (8 : 9),  (9 : 10),  (10 : 12),  (12 : 13),  (13 : 14), (14 : 15), (15 : 14),  (14 : 16),  (16 : 17
```

Figure 5: CC and Independent Paths of sample problem 1

As depicted in Figure 5 cyclomatic complexity of “main” module is 5 and all the independent paths are displayed.

After applying GA, following results were obtained.

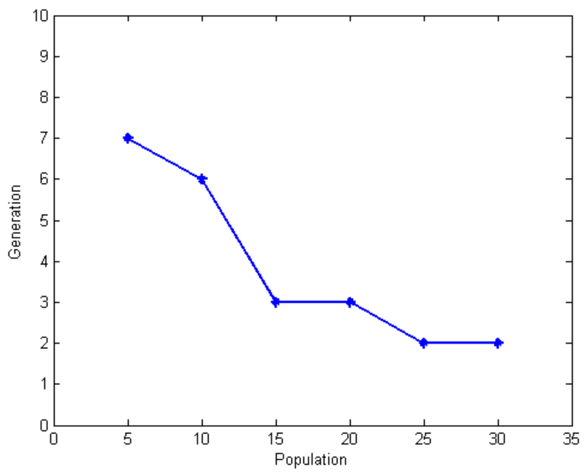


Figure 6: Initial Population vs. Generation Graph for sample problem 1

Figure 6 shows graph of initial population taken and number of generations taken to get an optimized solution. Here initial chromosomes size is 5 and number of test cases provided are 14.

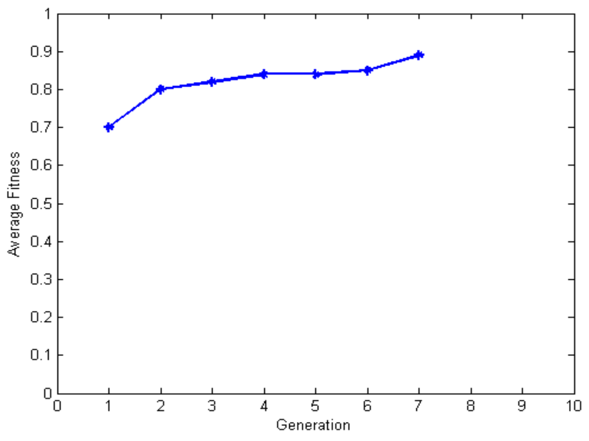
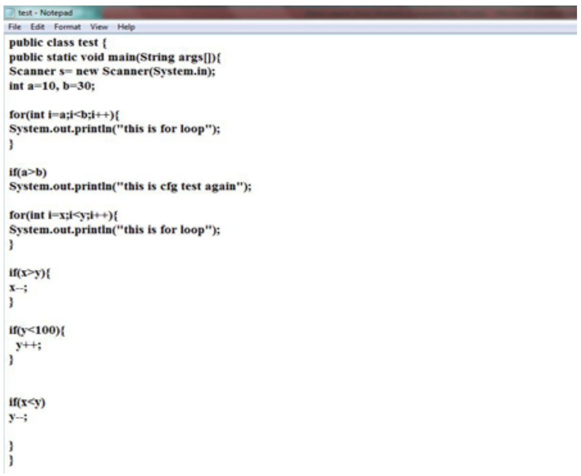


Figure 7: Generation vs. Average Fitness Graph for sample problem 1

Figure 7 shows graph between generation and average fitness of population. Initial chromosome size is 5 and test cases provided are 14 and population size is 5. With every passing generation, average fitness of population is improving.

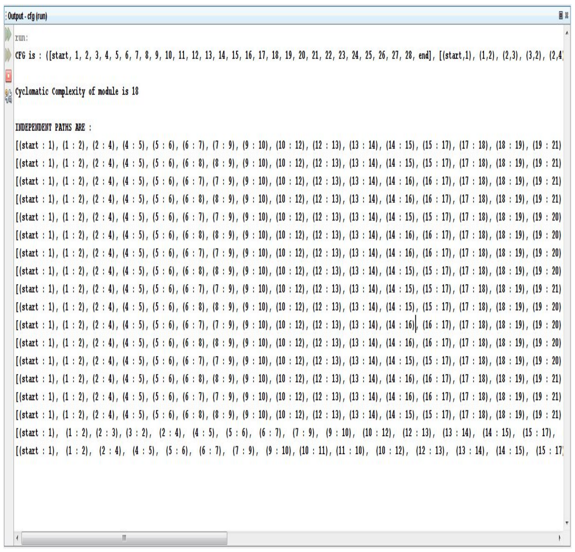
Taking Sample Problem 2:



```
public class test {
public static void main(String args[]){
Scanner s= new Scanner(System.in);
int a=10, b=30;

for(int i=a;i<b;i++){
System.out.println("this is for loop");
}

if(a>b)
System.out.println("this is cfg test again");

for(int i=x;i<y;i++){
System.out.println("this is for loop");
}

if(x>y){
x--;
}

if(y<100){
  y++;
}


if(x<y)
y--;

}
}
```

Figure 8: Sample Problem 2

Figure 9: CFG of sample problem 2



```
run:
CFG is : ([start, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, end], [(start,1), (1,2), (2,3), (3,2), (2,4

Cyclomatic Complexity of module is 18

INDEPENDENT PATHS ARE :
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 7), (7 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 15), (15 : 17), (17 : 18), (18 : 19), (19 : 21)
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 8), (8 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 15), (15 : 17), (17 : 18), (18 : 19), (19 : 21)
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 7), (7 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 16), (16 : 17), (17 : 18), (18 : 19), (19 : 21)
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 8), (8 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 16), (16 : 17), (17 : 18), (18 : 19), (19 : 21)
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 7), (7 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 15), (15 : 17), (17 : 18), (18 : 19), (19 : 20)
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 8), (8 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 16), (16 : 17), (17 : 18), (18 : 19), (19 : 20)
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 7), (7 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 16), (16 : 17), (17 : 18), (18 : 19), (19 : 20)
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 8), (8 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 15), (15 : 17), (17 : 18), (18 : 19), (19 : 20)
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 7), (7 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 15), (15 : 17), (17 : 18), (18 : 19), (19 : 21)
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 8), (8 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 15), (15 : 17), (17 : 18), (18 : 19), (19 : 20)
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 7), (7 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 16), (16 : 17), (17 : 18), (18 : 19), (19 : 20)
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 8), (8 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 16), (16 : 17), (17 : 18), (18 : 19), (19 : 20)
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 7), (7 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 15), (15 : 17), (17 : 18), (18 : 19), (19 : 20)
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 8), (8 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 16), (16 : 17), (17 : 18), (18 : 19), (19 : 21)
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 7), (7 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 16), (16 : 17), (17 : 18), (18 : 19), (19 : 21)
[(start : 1), (1 : 2), (2 : 4), (4 : 5), (5 : 6), (6 : 8), (8 : 9), (9 : 10), (10 : 12), (12 : 13), (13 : 14), (14 : 15), (15 : 17), (17 : 18), (18 : 19), (19 : 21)
[(start : 1),  (1 : 2), (2 : 3), (3 : 2),  (2 : 4),  (4 : 5),  (5 : 6),  (6 : 7),  (7 : 9),  (9 : 10),  (10 : 12),  (12 : 13),  (13 : 14),  (14 : 15),  (15 : 17),
[(start : 1),  (1 : 2),  (2 : 4),  (4 : 5),  (5 : 6),  (6 : 7),  (7 : 9),  (9 : 10), (10 : 11), (11 : 10),  (10 : 12),  (12 : 13),  (13 : 14),  (14 : 15),  (15 : 17
```

Figure 10: CC and Independent Paths of sample problem 2

After applying GA, following results were obtained.

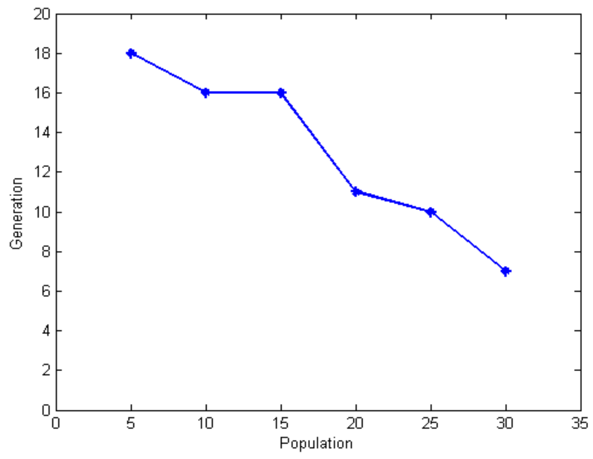


Figure 11: Initial Population vs. Generation Graph for sample problem 2

Figure 11 shows graph of initial population taken and number of generations taken to get an optimized solution. Here initial chromosomes size is 18 and number of test cases provided are 28.

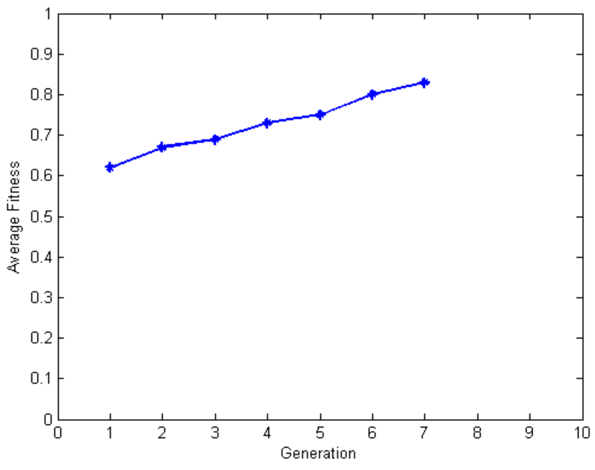


Figure 12: Generation vs. Average Fitness Graph for sample problem 2

Figure 12 shows graph between generation and average fitness of population. Initial chromosome size is 18 and test cases provided are 28 and population size is 30. With every passing generation, average fitness of population is improving.

The conclusion derived by taking different problems is depicted in figure 13.

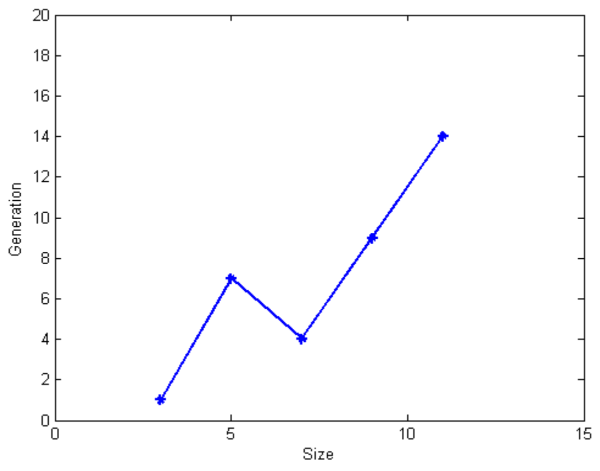


Figure 13: Chromosome Size vs. Generation Graph

Figure 13 depicts graph between initial size of chromosomes (x-axis) and number of generations (y-axis) taken to get an optimized solution. Initial population is fixed to 5. Chromosome's size is unswervingly proportional to intricacy of module. So, as the intricacy of module increases, it takes more generations to obtain an optimized solution.

VII. Conclusion And Future Work

In this work, optimization of software testing is achieved by employing GA and the process is automated. It results in formulation of test suite for a module that gives 100 % code coverage. The process of code analysis is to find all modules in a program, generation of CFG, finding cyclomatic complexity, determination of all independent paths and GA steps are automated. GA is employed on a set of different software programs and analyses are done on results obtained which decide performance of GA.

In this work, test cases are created manually and paths followed by them are manually determined. Roulette Wheel Selection (RWS) selection operator is employed for selecting parents and single point crossover is employed as crossover operator. In future, test case generation from operational profile and path followed by them in CFG can be automated. Other selection operators and crossover operator can be applied and comparison can be drawn between performances of different operators.

In this work very basic fitness function is used. In future, fitness function can be formulated based on *Average Percentage of Condition Coverage* (APCC).

Acknowledgment

This paper could not be written to its fullest without the guidance of Dr. Avdhesh Gupta, who served as my co-supervisor.

References And Bibliography

[1] Afzal, Wasif, and Richard Torkar: "On the application of genetic programming for software engineering predictive modeling: A systematic review" Expert Systems with Applications, vol. 38, no. 9, pp. 11984-11997, 2011.

[2] Ali, Shaukat, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege: "A systematic review of the application and empirical investigation of search-based test case generation" Software Engineering, IEEE Transactions, vol. 36, no. 6, pp. 742-762, 2010.

[3] Askarunisa, A., L. Shanmugapriya, and N. Ramaraj: "Cost and Coverage Metrics for Measuring the Effectiveness of Test Case Prioritization Techniques" INFOCOMP Journal of Computer Science, pp. 1-10, 2009.

[4] Bhasin, Harsh, Harish Kumar, and Vikas Singh: "Orthogonal Testing Using Genetic Algorithms" International Journal of Computer Science and Information Technology, vol. 4, no. 2. pp. 374-377, 2013.

[5] Dustin, Elfriede, Jeff Rashka, and John Paul: "Automated software testing" introduction, management, and performance. Addison-Wesley Professional, 1999.

[6] El-Far, Ibrahim K., and James A. Whittaker: "Model-Based Software Testing" Encyclopedia of Software Engineering, 2001.

[7] Fraser, Gordon, and Andreas Zeller.: "Mutation-driven generation of unit tests and oracles" Software Engineering, IEEE Transactions, vol. 38, no. 2, pp. 278-292, 2012.

[8] Gold Robert: "Control flow graphs and code coverage" pp. 739-749, 2010.

[9] Jin, Yaochu: "A comprehensive survey of fitness approximation in evolutionary computation" Soft computing, vol. 9, no. 1, pp. 3-12, 2005.

[10] Khamis, Abdelaziz M., 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, pp. 383-401, 2007.

[11] Konak, Abdullah, David W. Coit, and Alice E. Smith. "Multi-objective optimization using genetic algorithms: A tutorial." Reliability Engineering & System Safety, vol. 91, no. 9, pp. 992-1007, 2006.

[12] Mahdavi, M., Mohammad Fesanghary, and E. Damangir: "An improved harmony search algorithm for solving optimization problems" Applied mathematics and computation, vol. 188, no. 2, pp. 1567-1579, 2007.

[13] Mantere, Timo, and Jarmo T. Alander: "Evolutionary software engineering, a review" Applied Soft Computing, vol. 5, no. 3, pp. 315-331, 2005.

[14] Offutt, Jeff, Shaoying Liu, Aynur Abdurazik, and Paul Ammann: "Generating test data from state-based specifications" Software Testing, Verification and Reliability, vol. 13, no. 1, pp. 25-53, 2003.

[15] Pargas, Roy P., Mary Jean Harrold, and Robert R. Peck: "Test-data generation using genetic algorithms" Software Testing Verification and Reliability, vol. 9, no. 4, pp. 263-282, 1999.

[16] Roper, M.: "Software testing" International software quality assurance Series, 1994.

[17] Sebesta, Robert W.: "Concepts of programming languages" vol. 4, Addison Wesley, 2002.

[18] Srivastava, Praveen Ranjan, and Tai-hoon Kim: "Application of genetic algorithm in software testing" International Journal of software Engineering and its Applications, vol. 3, no. 4, pp. 87-96, 2009.

[19] Xu, B. W., X. Y. Xie, Liang Shi, and C. H. Nie: "Application of genetic algorithms in software testing" Proc. of the Advances in Machine Learning Application in Software Engineering. IGI Global 317, 2007.

[20] Yu, Yuen Tak, and Man Fai Lau: "Fault-based test suite prioritization for specification-based testing" Information and Software Technology, vol. 54, no. 2, pp. 179-202, 2012.