IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 18, Issue 4, Ver. V (Jul.-Aug. 2016), PP 97-107
www.iosrjournals.org

An Efficient Test Data Generation Approach for Unit Testing

Anil Kumar Gupta, Fayaz Ahmad Khan
Department of Computer Science and Applications, Barkatullah University, Bhopal (M.P)

Abstract: *To ensure the delivery of high-quality software, software testing plays the vital role. One of the major time-consuming and expensive activities in software testing is the generation of test data. Test data generation activity has a strong impact on the effectiveness and efficiency of the whole testing process. In order to reduce the cost and time involved in the process of test data generation, researchers and practitioners have tried to automate it. In literature, many such techniques have been developed and the most commonly used are; Random testing, Symbolic execution and evolutionary testing. In this work, an enhanced and efficient Random test data generation approach is proposed and investigated on a suite of programs and its efficiency is compared with the Genetic algorithm which is an evolutionary approach. The inconsistency of random approach is that it is not capable of generating a specific set or combination of test cases for the program input variables and in search of these effective test cases multiple populations needs to be created that will increase the burden of size . So, in order to remove these inconsistencies from the test suite, it is seeded with a more effective set of test cases through our proposed approach in order to make test suite more granular and limit its size by not generating more populations in search of these effective test cases. In addition to the proposed approach, the classification of test adequacy criteria and issues with random, symbolic execution and genetic algorithm based test data generation techniques are also provided and highlighted.*

Keywords: *Software Testing, Test Data Generation, Random Testing, Symbolic Execution, and Genetic Algorithm*

I. Introduction

Software testing is technically and economically imperative for high quality software production. In software production half of the expenses have been projected to be due to testing. Thus it is imperative to reduce the cost and enhance the effectiveness of manual software testing by automating the whole software testing process. In literature many techniques and tools have been proposed and developed for automatic the testing process. One of the important issues in automating software testing process is the automatic generation of test data [1, 2, 3]. Test data generation is the most time consuming task in software testing and one that impacts its effectiveness and efficiency. Test data generation in software testing is the process of locating and selecting input data that satisfies the given criterion [4]. Various artifacts of the Software can be considered to generate test data like requirements, design models, code; the input/output data space and information obtained from program execution.

The aim of automating the testing process is to reduce the cost and human effort devoted in manual software testing. If the testing process could be automated, the cost of software development would also be reduced significantly. Among many testing activities, test case generation is one of the most demanding task and also most critical one. Normally, the large domain of input values for variables is a problem and choice of exhaustive testing is impractical due to time and resource constraints. The only way is to use, a part of the input domain to execute the software under test. The question to be answered is which values and how many should be selected to maximize the chance of detecting faults. To generate test data, automated test case production methods should be applied. Bertolino in [1] addresses the need for 100 percent automatic testing. But, an automated testing strategy in addition to automatic test data generation activity must address other activities like: the generation of test requirements, oracle generation, selection of test cases and test case prioritization.

II. Steps In Our Proposed Approach For Automatic Unit Testing

1. Subject programs for experimentation.
2. Generate the Control Flow Graph in order to know the differnt components present in them.
3. Set Test adequacy Criteria as stopping condition and for coverage measurement. Here in this study path coverage (Basic paths or independent paths) and branch coverage are set as test adequacy criteria; because, it impossible to do exhaustive path testing due to infinite number of paths if the program has loops).
4. Find all the basic path sequences in the program using Mc-Cabe Cyclomatic Complexity Metric.
5. Use the proposed Enhanced Random algorithm with additional seeded test cases as a Search Space.
6. Execute the Program until stopping condition or coverage criteria is satisfied.

III. Control Flow Graph Construction

Control flow graph is a graphical representation of the source code of a program. The statements or expressions of a program are denoted with nodes and flow of control by edges in the program graph. The program control flow graph (CFG) helps us to understand the internal structure of the program which may provide the basis for designing the test cases. A control flow graph of a program P is a directed graph **G**= (**N, A, S, E**), where
N= Set of Nodes. , **A**= A binary relation on NX N, known as set of edges.
S= Start or Unique Entry, **E**= Exit or unique exit.

In this study, Control Flow Graph Factory is used as an Eclipse plug-in for generating control flow graphs and exporting them. The control flow graph and basic block graphs of programs in **Figure 1** and **Figure 2** are shown by **Figure 3** and **Figure 4.** The Control flow graph factory is a tool from Dr. Garbage Tools, a suite of Eclipse Plug-ins. Dr. Garbage tools are specialized in debugging and development.

IV. Test Adequacy Criteria

Two important questions arise with respect to software testing: “How is the design and selection of test data performed?” and “How one may be able to decide when a product was sufficiently tested?” During testing, testing criteria for test suite selection and evaluation are crucial to the success of the testing activity. A test adequacy criterion tells us that how test cases should be selected in order to increase the fault detection ability of the selected test suite. A testing criterion is used by the tester to subdivide or compress the input/ output domains and provides a systematic way to select a finite number of test cases to compose a test suite. The main objective is to create the smallest test suite which will take less time for execution and for which the output indicates the largest set of faults.

V. Classification Of Adequacy Criteria

There are two categories (or Classes) of test data adequacy criteria [3]. These two categories are considered as two dimensions of the space of software test adequacy criteria. One, classification based on the information used to specify testing requirements and hence includes:
i. Specification Based.
ii. Program Based.
iii. Combined Specification and Program Based.
iv. Interface Based.

Another classification is based on the underlying testing approach and thus includes:
i. Structural Testing:
ii. Fault-based Testing.
iii. Error Based.

The two below mentioned adequacy criteria’s (Program Based) are most commonly used:
1. Control flow Based which includes :
a) Statement Coverage b) Branch Coverage c) Path coverage
d) Cyclomatic –Number Criteria e) Multiple Condition Coverage.

2. Data Flow Based Criteria’s
a) All definition Criterion. b) All Uses Criterion.
c) Interactions between variables (The Ntafos required K- touples Criteria).
d) Combinations of Definitions (Laski-Korel Criteria) and few others.

For the current study Path Coverage criteria and Branch Adequacy criteria are set as testing criteria. Path testing is a complex problem and its challenging section is to generate test cases that cover selected paths. Specifically, the path testing problem is an NP-complete problem [3]. That is why several heuristic approaches have been developed and investigated for path coverage. In addition to this, instead of simply traversing every path, it only traverses independent paths of code under test. An independent path is any path that tests a decision independently of other decisions. To find the number of independent paths form both programs in **Figure 1** and **Figure 2**, the Cyclomatic complexity metric is employed and discussed below.

VI. Cyclomatic Complexity Metric

It is a measurement metric developed by Thomas McCabe [6] and is also named as Mc-Cabe Cyclomatic Complexity. It is used to measure the logical complexity of the code, by determining the number of linearly-independent paths of a structured program. It is used for two related purposes in testing methodology. First, it gives the number of recommended test cases for software under test. Second, it is used during all phases of the software lifecycle, beginning with design, to keep software reliable, testable, and manageable.
The following equation is used for computing Cyclomatic complexity of the program control flow graph (CFG).
V (G) = e - n + 2 Where,
V (G) = Number of independent paths in a CFG, **e** = Number of edges present in the graph
n = Number of nodes of the graph
The other formula or equation for the determination of Cyclomatic Complexity is known as Predicate Nodes (Decision Nodes) formula:
V (G) = Number of Predicate Nodes + 1.
For the calculation of the Cyclomatic complexity of both the programs depicted in **Figure 1** and **Figure 2**, any one of the above formula can be used;
The Cyclomatic complexity of Program in **Fig.1**:
V (G1) = (**Number of Predicate Nodes in CFG (1) + 1) =** 3+1 **= 4.**
The Cyclomatic complexity of Program in **Fig.2**:
V (G2) = (**Number of Predicate Nodes in CFG (2) + 1) =** 10+1**= 11.**
Based on Cyclomatic complexity measure, test data generation techniques are used for generation of input values. The most commonly and widely used test data generation techniques are discussed below.

VII. Test Data Generation Techniques

The goal of software testing is to uncover as many as faults by examining the code with a potent set of test cases. To automatically generate such a potent set of test cases in order to fulfill the desired or set adequacy criteria; is an intellectually demanding and a very difficult task. It also has a very strong impact on the effectiveness and efficiency of the whole testing process [1, 2, 3]. In literature, many such techniques have been developed and investigated. The most commonly used ones are Random test Data Generation [7], Symbolic Execution [8] and Search Based test data Generation [32] techniques.

VIII. Random Based Test Data Generation Techniques.

Random testing is one of the simplest, fundamental and most popular test data generation methods. In random testing technique, the software is executed with non- associated volatile test data from the specified input domain [7]. Random testing was introduced by Hanford [9], who reported a tool known as syntax machine that randomly generate data for testing PL/I compilers. Random testing approach with respect to other approaches discussed below is considered as economical, simple, unbiased, and also requires less computational effort [10].

Hamlet and Taylor in [11] mentioned that random testing is better than other techniques in terms of finding faults and there are not enough differences between partition and random testing. While as Deason in [14], commented that random number generators are unproductive and thus does not provide the necessary coverage of the program. Myers in [13] strengthened this statement and mentioned that random testing is almost certainly the poorest methodology in software testing. However, Duran and Ntafos in [12] affirmed that many errors can be easily found, but the problem is to determine whether a test run failed. So, automatic output checking is essential if large numbers of tests are to be carried.

DeMillo in [15] declared that the capability of random data is very much dependent on the interval from which the data is selected. In order to accommodate all the values including values out of the specified range, the interval plays a vital role. Data from poorly chosen intervals are much worse than those from well-chosen intervals. The authors in [12] also agreed that the change of range for random testing has a great effect on the efficiency of the testing technique. The disadvantage of random testing is that, it is inadequate to generate equality values of input variables. Bertolino in [16] stated that random testing is more stressing to the program under test than handpicked test data. But it is also mentioned in [16], that random inputs may never exercise both branches of a predicate when it is required to tests for equality between them. In this study as we are applying a random generator tool for the generation of test cases from a fixed interval for testing the sample programs. The initial automated generated test cases are depicted in Figure **5**.

IX. Symbolic Execution Based Test Data Generation Techniques.

Symbolic execution is a white box automatic test data generation technique. Symbolic executions uses symbolic values as program inputs instead of actual variables and represent these values as symbolic expressions of those inputs [17]. Basically, a symbolic executed program includes the symbolic values of the variables, a path constraint and a program counter. The path constraint is a Boolean formula and an accumulation of constraints that the inputs must fulfill in order to execute the path. The role of program counter is to identify the next statement to be executed. A major challenge with symbolic execution is that it needs to understand each and every statement in order to collect the path constraints. Thus the effectiveness of symbolic execution of real world programs is still limited due to the three fundamental problems like Path explosion, Path divergence and the solution of Complex constraints. However, due enormous computational power of today's computers, the barrier of applying symbolic execution is lower and apart from its application to test data generation [18,19,20], the other uses of symbolic execution include generation of security exploits [21], regression testing [22] and data base testing [23]. Hence due to its wide application, a number of tools have been developed and are available which includes: Symbolic Pathfinder, JCUTE [24], JFuzz [25] and LCT for Java [26], CUTE [27], Klee [28], S2E [29], Crest target C language [30], and PEX [31], for .NET language.

X. Evolutionary Test Data Generation Techniques.

Local search techniques like Hill Climbing becoming trapped in local optima, so global search techniques like Genetic Algorithms have been considered and applied in software test data generation. Genetic algorithms are characterized by an iterative procedure and thus work in parallel on a number of potential solutions. Miller and David Spooner in [32], generates test data consisting of floating-point inputs and is completely a unique technique compared to the existing techniques developed at the time. In [32], the cost function or fitness function is fundamental and is used to guide the optimization process towards the required direction. They provide the means to evaluate individuals, thus allowing a search to move towards better individuals in the hope of finding a solution. Inputs, which execute the desired path, were assigned lower cost values and those inputs with higher cost values were discarded. In 1992, Xanthakis in [33] applied GA for automatic test case generation. After that, there has been an explosion of work in applying Search-based optimization techniques to a huge number of software testing problems like, functional testing [34], integration testing [35], mutation testing [36], regression testing [37], test prioritization [38,39]. Among other optimization algorithms, genetic algorithms have been the most widely applied search technique. Pargas et al. [40] used Genetic Algorithm to search for test data to satisfy all-nodes and all branches criteria. Michael et al. [41] used GAs for automatic test-data generation to satisfy condition-decision test-coverage criterion. They proposed a Genetic Algorithm Data Generation Tool (GADGET) to generate test cases for large C and C++ programs.

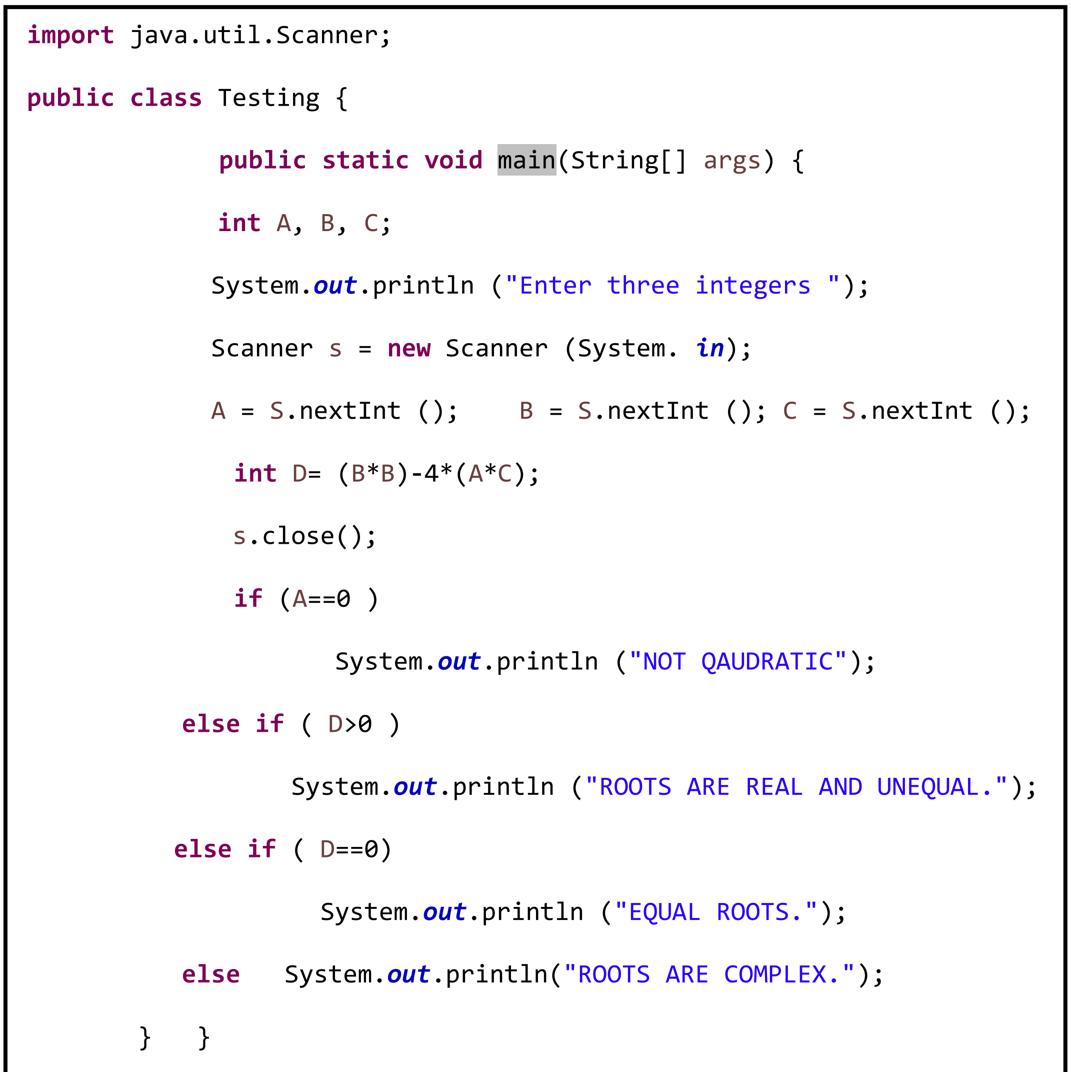
XI. Issues Found While Applying Genetic Algorithm

In applying Genetic Algorithms, the authors Pargas et al in [40] and Benoit Baudry et al in [42] found the following issues:

I. Representation of the population. The candidate solutions for the problem at hand must be encoded in order to be manipulated by the search algorithm.
II. The fitness function is domain specific, and needs to be defined for a new problem.
III. Risk of suboptimal solution, delayed convergence and strike up at local optima.
IV. The experiments with genetic algorithms were not satisfactory. The mutation rate has to be increased consistently when compared to usual application of genetic algorithms.
V. Moreover, due to the slow convergence, the results are not stable and one population can be more efficient from the following, due to a non-explicit memorization.

XII. The Proposed Approach And Its Implementation

In this present preliminary study, our aim is to explain the experimental investigation into software testing using the two well-known referenced programs shown in **Figure 1** and **Figure 2**. This study is also an extension to the two existing studies [43] [44], where the performance of random approach and the genetic algorithm are compared. In both of the studies, random approach achieves lower code coverage compared to the genetic algorithm in some cases. Because random approach lakes the ability to generate a specific combination of input values for testing the program and in a triangle classification problem it is due to predicate branch, where a triangle is classified as Equilateral. The other possibilities could be due to inconsistency in the random approach that it could not be able to generate other values like, boundary values both upper and lower bound, equal pair of values for each variables, some combination of these pairs, and other values. The initial test cases depicted in **Figure 5**, are generated using generatedata.com tool, which is free data generation tool for database testing and software testing. The tool is widely used by many vendors for the testing of data base and software applications. In this study, the range of each input values (A, B, and C) is set as (-110 to +110) and initially 100 rows of different combinations of each values are generated. It is clearly visible that the generated test suite is inefficient to achieve the specified adequacy Criteria's due to the absence of effective test cases and it contains a considerable number of redundant test cases which satisfied the same requirement multiple times. So, in order to enhance the efficiency of the test suite, the proposed technique in **Figure 6**, will seed all the remaining effective test cases automatically. The different number of effective test cases include (10,10,10), (1,2,3), (3,2,1),(1,3,2),(2,3,1),(0,0,0),(-1,-2,-3) and many more.Thus due to the addition of more effective test cases through our proposed approach, 100% branch coverage and independent path coverage is realized for both the programs and is shown in **Figure 7** and **Figure 9.** From the **Figure 7** and **Figure 9**, it is also observed that in addition to the specified adequacy criteria it also satisfied the other adequacy criteria's. To address the redundancy issue form the test suite, we have applied many clustering algorithms like K-Means and Hierarchical approach in our previous studies [45] [46]. In [44], the code coverage performance of GA and Random approach is shown in **Figure 11**. After the implementation of our proposed technique, the code coverage outperformed GA and the results are shown in **Figure 12**. The size of the test suite compared to the study in [43] is very small and will take less time to execute the whole test suite. The comparison of the size of test suites generated with our proposed approach and approach followed in [43] is shown in **Figure 13**. The reason for the size reduction is that, all the traditional random approaches generate multiple populations of test cases in search of few effective test cases but with our proposed approach these test cases are effectively seeded without a need to generate more populations. The future scope of this study will be the application of our proposed approach to all of the remaining programs in study [44] and its comparison with other search based techniques.



```
import java.util.Scanner;

public class Testing {

             public static void main(String[] args) {

             int A, B, C;

            System.out.println ("Enter three integers ");

            Scanner s = new Scanner (System. in);

            A = S.nextInt ();    B = S.nextInt (); C = S.nextInt ();

              int D= (B*B)-4*(A*C);

              s.close();

              if (A==0 )

                     System.out.println ("NOT QAUDRATIC");

          else if ( D>0 )

                  System.out.println ("ROOTS ARE REAL AND UNEQUAL.");

         else if ( D==0)

                    System.out.println ("EQUAL ROOTS.");

          else   System.out.println("ROOTS ARE COMPLEX.");

       }   }
```

Figure 1 Quadratic Triangle.

```
import java.util.Scanner;
public class Traingle {
       public static void main(String[] args) {
       Scanner S =new Scanner(System. in);
       int A,B,C;
       System.out.println ("ENTER THREE SIDES OF TRIANGLE");
       A= S.nextInt (); B= S.nextInt (); C= S.nextInt ();
       int triang;
       if (A <= 0 || B <= 0 || C <= 0)
       System.out.println ("illegal Input Values for sides");
               Else {  triang = 0;
               if (A == B)  triang = triang + 1;
               if (A == C)  triang = triang + 2;
               if (B == C)  triang = triang + 3;
               if (triang == 0)
               if (A + B <= C ||B + C <= A ||A + C <= B)
               {
                System.out.println ("illegal Not a traingle");
                }
              else
                System.out.println ("Scanlene Traingle");
               if (triang > 3)
                 System.out.println ("equilateral Traingle");
               else if (triang == 1 && A + B > C)
                 System.out.println ("isosceles");
               else if (triang == 2 && A + C > B)
                System.out.println ("isosceles");
               else if (triang == 3 && ((B + C) > A))
                 System.out.println ("isosceles");
                }

}}
```

Figure 2 Triangle Classification Problem

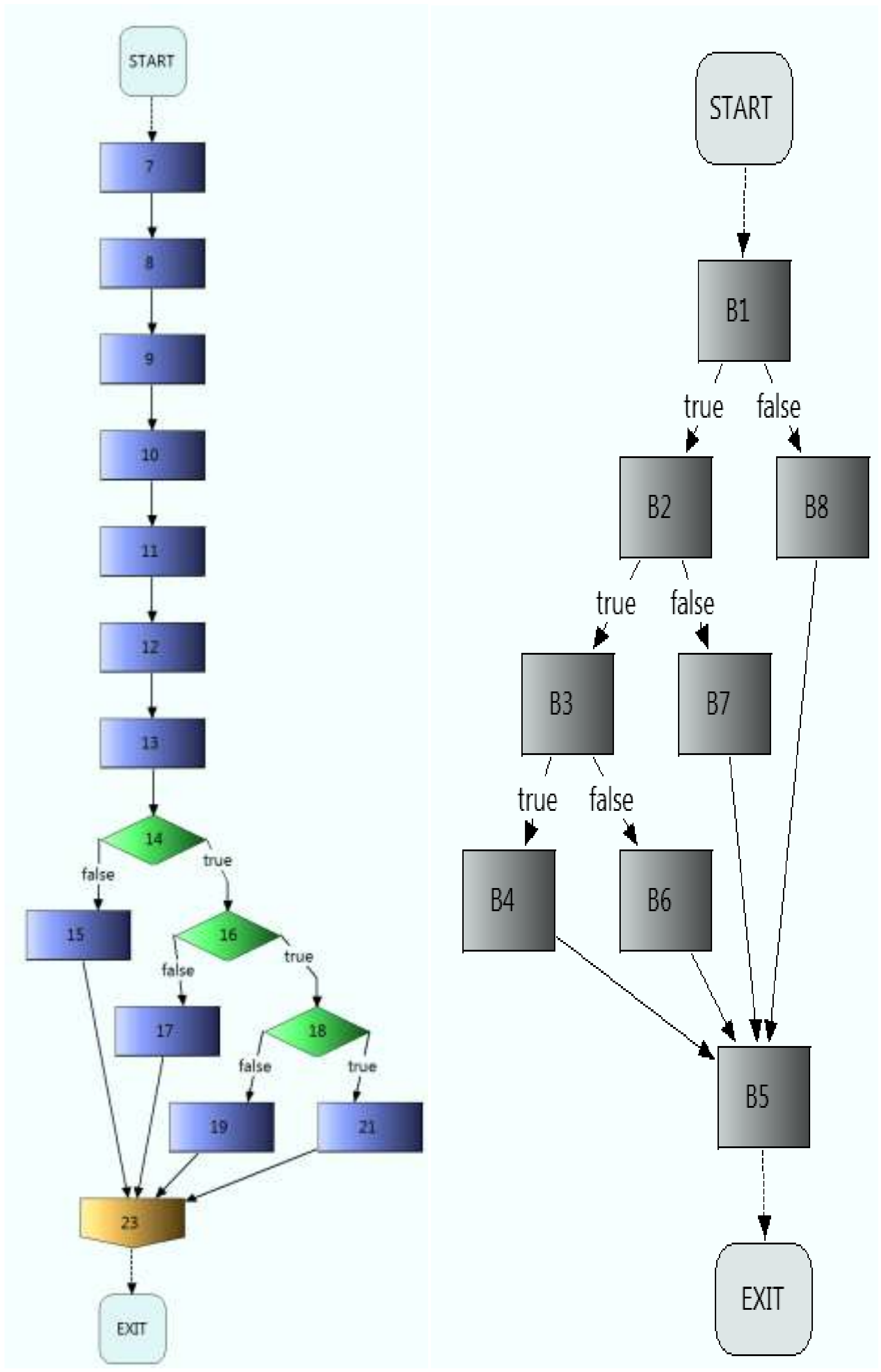


Figure 3. Source Code Graph and Basic Block Graph of the Program shown in Figure 1

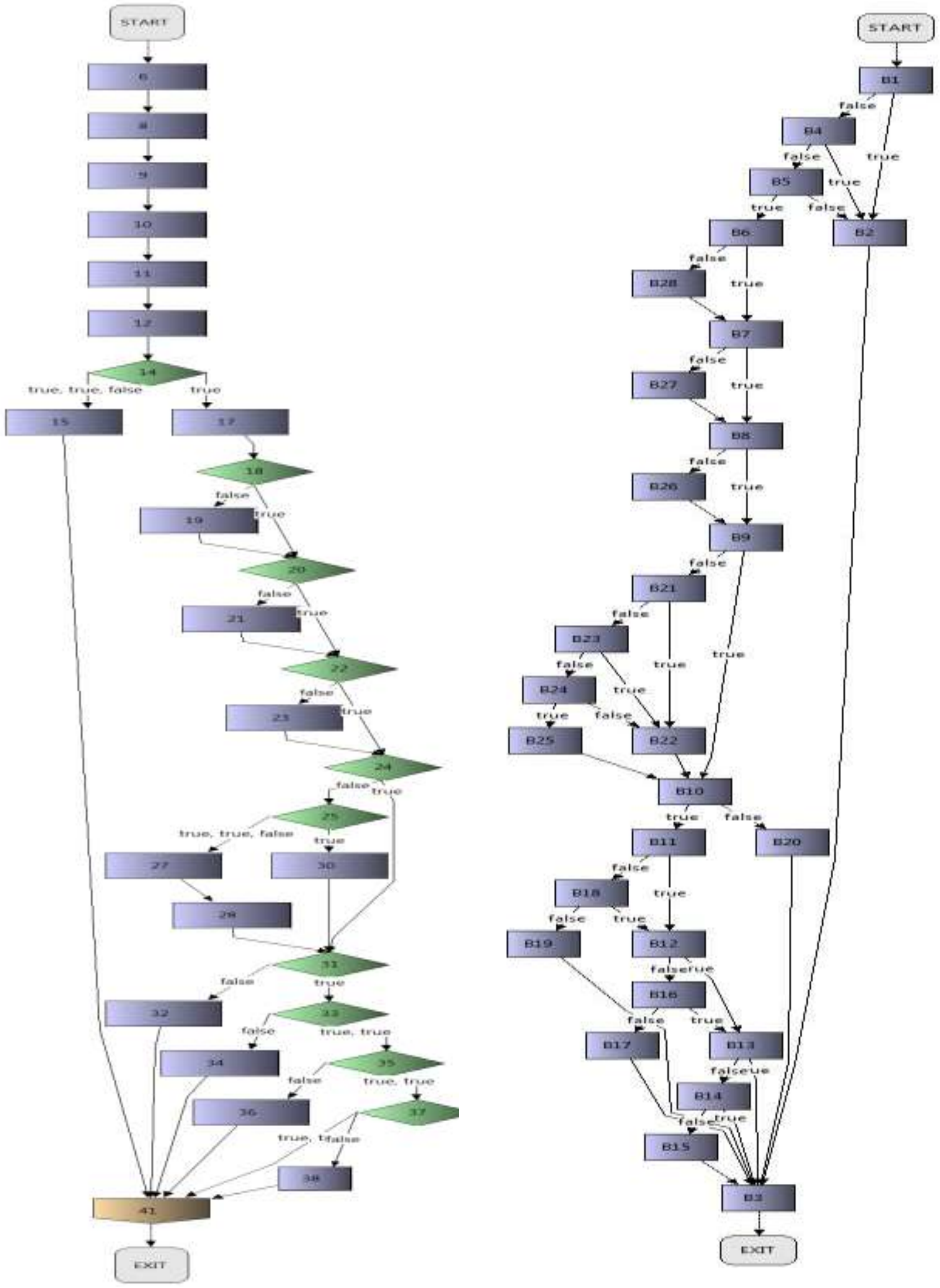
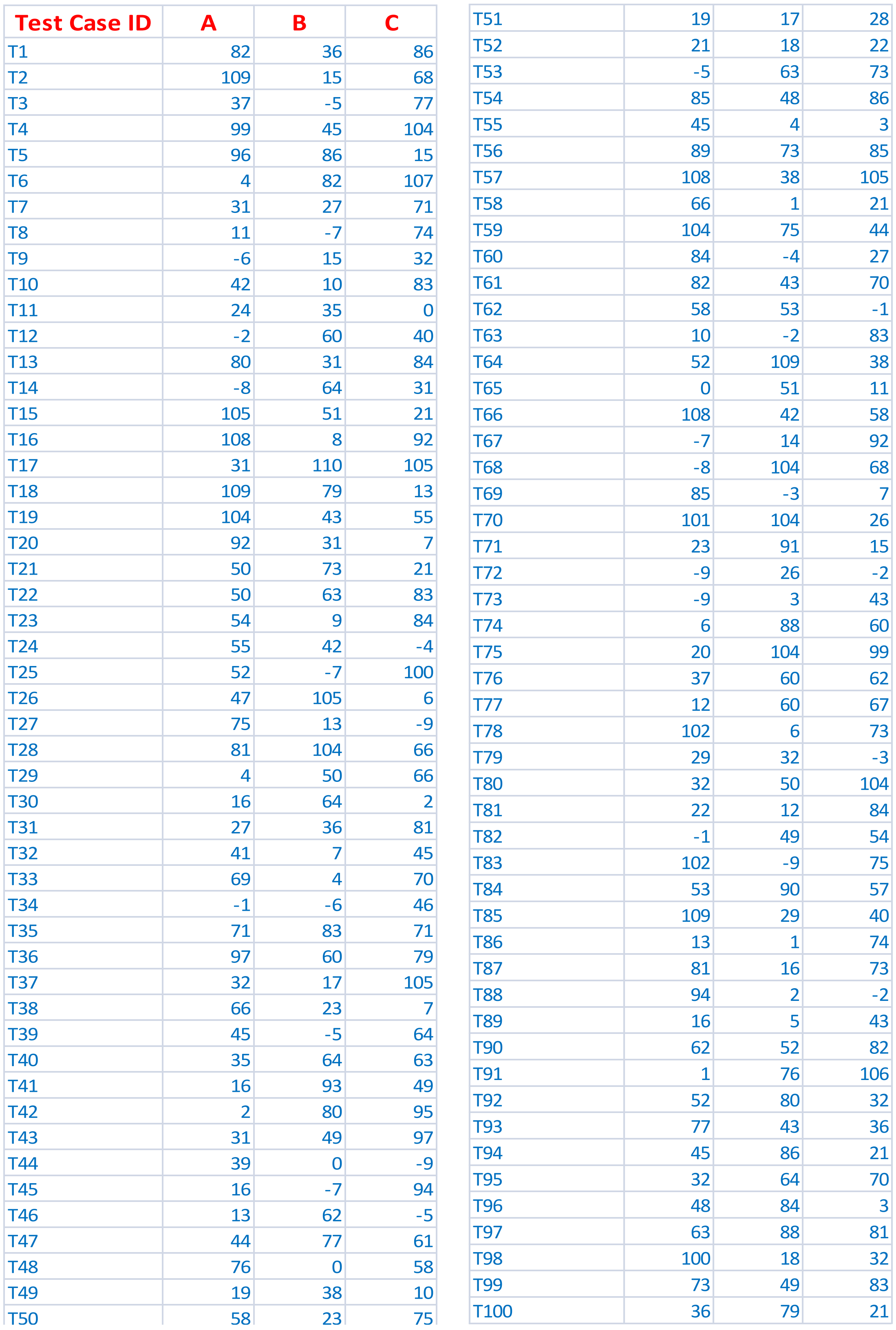


Figure 4. Source Code Graph and Basic Block Graph of Program Shown in Figure.2

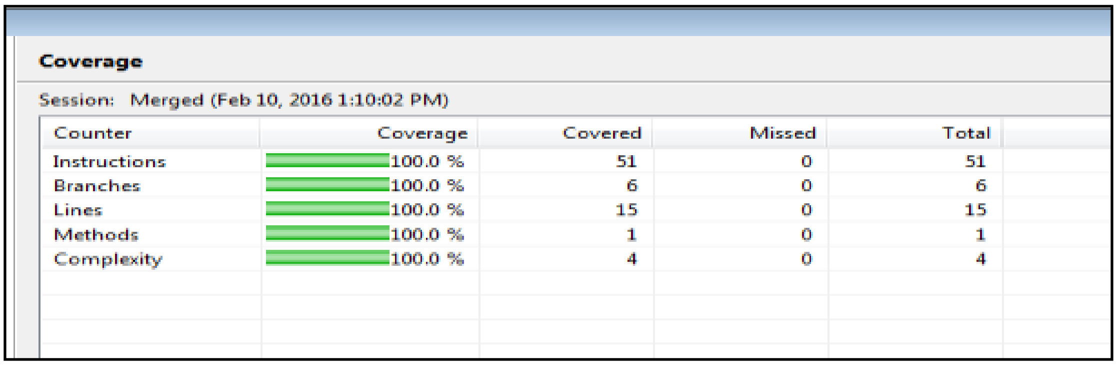


Test Case ID	A	B	C
T1	82	36	86
T2	109	15	68
T3	37	-5	77
T4	99	45	104
T5	96	86	15
T6	4	82	107
T7	31	27	71
T8	11	-7	74
T9	-6	15	32
T10	42	10	83
T11	24	35	0
T12	-2	60	40
T13	80	31	84
T14	-8	64	31
T15	105	51	21
T16	108	8	92
T17	31	110	105
T18	109	79	13
T19	104	43	55
T20	92	31	7
T21	50	73	21
T22	50	63	83
T23	54	9	84
T24	55	42	-4
T25	52	-7	100
T26	47	105	6
T27	75	13	-9
T28	81	104	66
T29	4	50	66
T30	16	64	2
T31	27	36	81
T32	41	7	45
T33	69	4	70
T34	-1	-6	46
T35	71	83	71
T36	97	60	79
T37	32	17	105
T38	66	23	7
T39	45	-5	64
T40	35	64	63
T41	16	93	49
T42	2	80	95
T43	31	49	97
T44	39	0	-9
T45	16	-7	94
T46	13	62	-5
T47	44	77	61
T48	76	0	58
T49	19	38	10
T50	58	23	75
T51	19	17	28
T52	21	18	22
T53	-5	63	73
T54	85	48	86
T55	45	4	3
T56	89	73	85
T57	108	38	105
T58	66	1	21
T59	104	75	44
T60	84	-4	27
T61	82	43	70
T62	58	53	-1
T63	10	-2	83
T64	52	109	38
T65	0	51	11
T66	108	42	58
T67	-7	14	92
T68	-8	104	68
T69	85	-3	7
T70	101	104	26
T71	23	91	15
T72	-9	26	-2
T73	-9	3	43
T74	6	88	60
T75	20	104	99
T76	37	60	62
T77	12	60	67
T78	102	6	73
T79	29	32	-3
T80	32	50	104
T81	22	12	84
T82	-1	49	54
T83	102	-9	75
T84	53	90	57
T85	109	29	40
T86	13	1	74
T87	81	16	73
T88	94	2	-2
T89	16	5	43
T90	62	52	82
T91	1	76	106
T92	52	80	32
T93	77	43	36
T94	45	86	21
T95	32	64	70
T96	48	84	3
T97	63	88	81
T98	100	18	32
T99	73	49	83
T100	36	79	21

Figure 5. Randomly Generated Test Cases using Generatedata.com tool for Both Programs in Figure 1 & Figure 2.

```
Input:   {CFG, Test Adequacy Criteria and TS (Test cases generated with
Generatedata.com   tool) of unit under Test}

Begin
    Execute (the Unit under Test with TS)
    If (TS is Adequate and Adequacy Criteria Met)
        Stop and Display The Coverage Information of The Unit Under Test.
   Else
       Update (TS) with More Effective set of test Cases.
   End
```

Figure 6. Proposed Approach

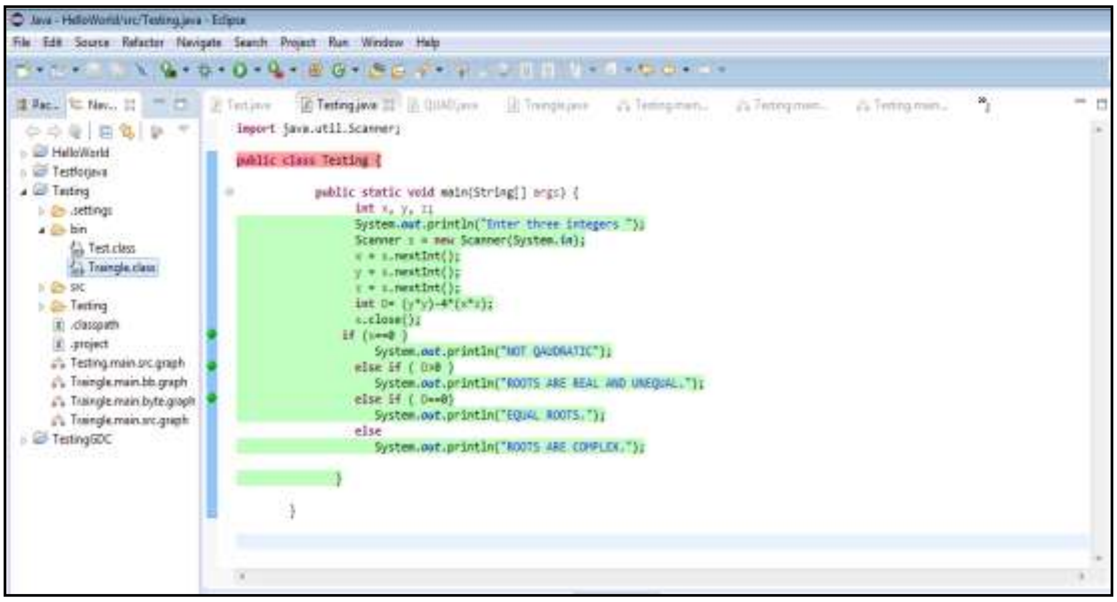


Coverage

Session: Merged (Feb 10, 2016 1:10:02 PM)

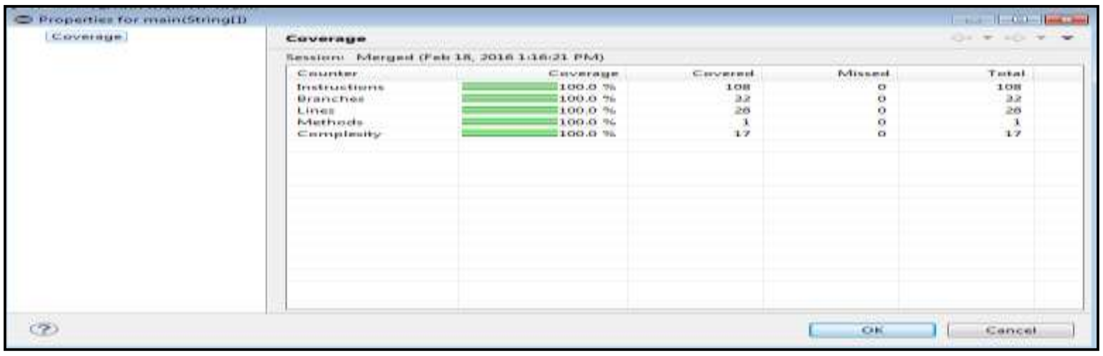
Counter	Coverage	Covered	Missed	Total
Instructions	100.0 %	51	0	51
Branches	100.0 %	6	0	6
Lines	100.0 %	15	0	15
Methods	100.0 %	1	0	1
Complexity	100.0 %	4	0	4

Figure 7. Code Coverage of the individual components of Fig.3



```
import java.util.Scanner;

public class Testing {

            public static void main(String[] args) {
                int x, y, z;
                System.out.println("Enter three integers ");
                Scanner s = new Scanner(System.in);
                x = s.nextInt();
                y = s.nextInt();
                z = s.nextInt();
                int D= (y*y)-4*(x*z);
                s.close();
              if (x==0 )
                   System.out.println("NOT QAUDRATIC");
                else if ( D>0 )
                   System.out.println("ROOTS ARE REAL AND UNEQUAL.");
                else if ( D==0)
                   System.out.println("EQUAL ROOTS.");
                else
                   System.out.println("ROOTS ARE COMPLEX.");

        }

    }
```

Figure 8. Code Coverage (Light Green area) of the Whole Unit

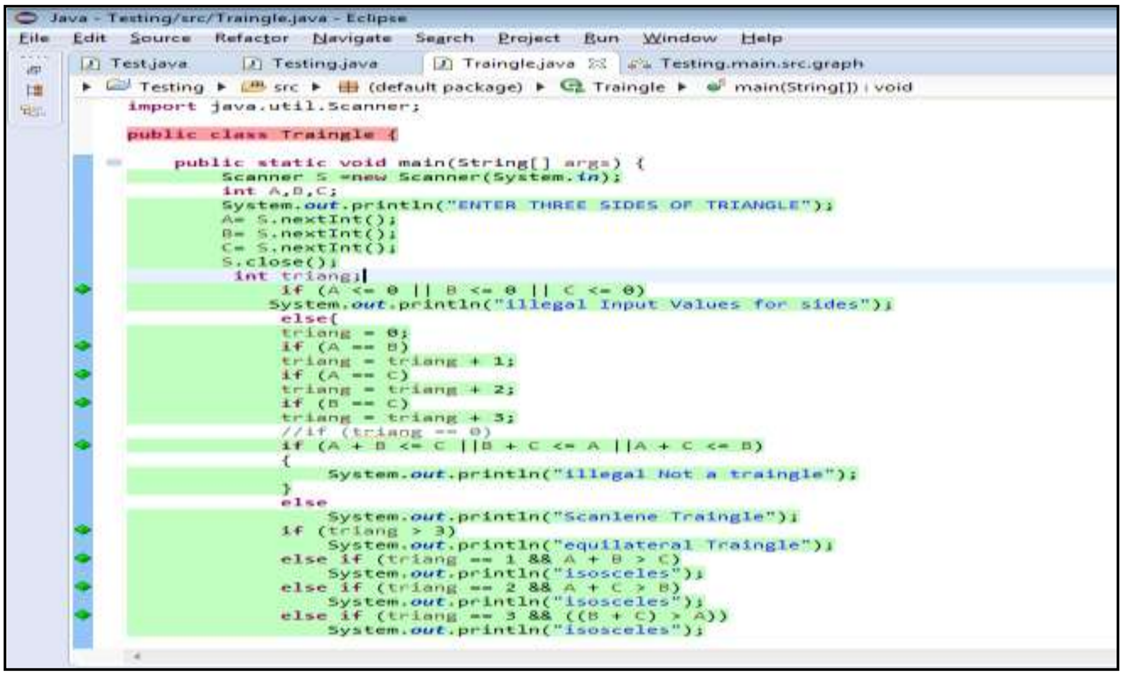


Coverage

Session: Merged (Feb 18, 2016 1:16:21 PM)

Counter	Coverage	Covered	Missed	Total
Instructions	100.0 %	108	0	108
Branches	100.0 %	32	0	32
Lines	100.0 %	28	0	28
Methods	100.0 %	1	0	1
Complexity	100.0 %	17	0	17

Figure 9. Code Coverage of the individual Components of Figure 4


Java - Testing/src/Traingle.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Test.java
Testing.java
Traingle.java
Testing.main.src.graph
Testing ▸ src ▸ (default package) ▸ Traingle ▸ main(String[]) : void

```
import java.util.Scanner;

public class Traingle {

    public static void main(String[] args) {
        Scanner S =new Scanner(System.in);
        int A,B,C;
        System.out.println("ENTER THREE SIDES OF TRIANGLE");
        A= S.nextInt();
        B= S.nextInt();
        C= S.nextInt();
        S.close();
         int triang;
            if (A <= 0 || B <= 0 || C <= 0)
           System.out.println("illegal Input Values for sides");
            else{
            triang = 0;
            if (A == B)
            triang = triang + 1;
            if (A == C)
            triang = triang + 2;
            if (B == C)
            triang = triang + 3;
            //if (triang == 0)
            if (A + B <= C ||B + C <= A ||A + C <= B)
            {
                System.out.println("illegal Not a traingle");
            }
            else
                System.out.println("Scanlene Traingle");
            if (triang > 3)
                System.out.println("equilateral Traingle");
            else if (triang == 1 && A + B > C)
                System.out.println("isosceles");
            else if (triang == 2 && A + C > B)
                System.out.println("isosceles");
            else if (triang == 3 && ((B + C) > A))
                System.out.println("isosceles");
```

Figure 10. Code Coverage of the whole Unit.

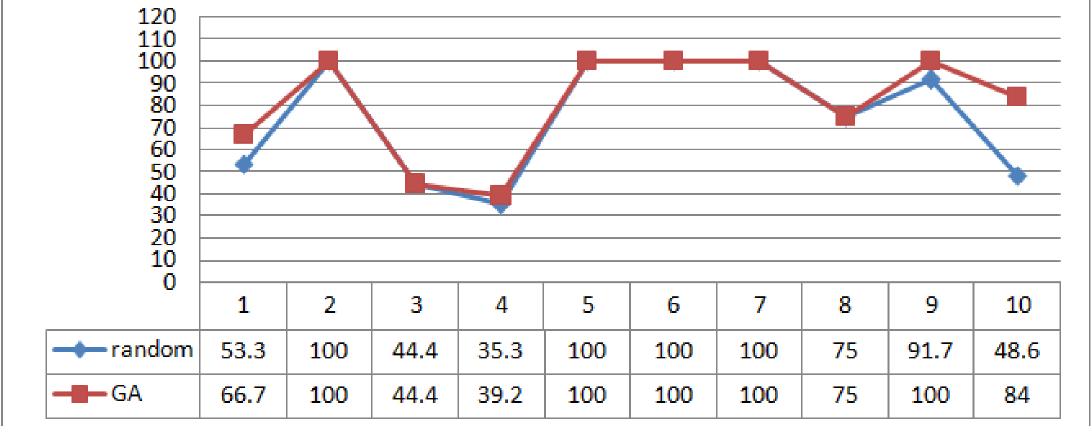


Figure. 11. Coverage Comparision of Random and Genetic Algorithm in [44]

	1	2	3	4	5	6	7	8	9	10
random	53.3	100	44.4	35.3	100	100	100	75	91.7	48.6
GA	66.7	100	44.4	39.2	100	100	100	75	100	84

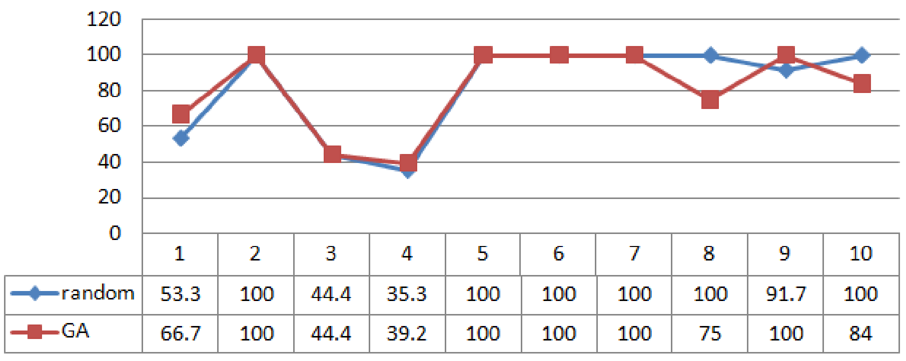


Figure.12. Comaprision after implementation of the proposed Appraoch

	1	2	3	4	5	6	7	8	9	10
random	53.3	100	44.4	35.3	100	100	100	100	91.7	100
GA	66.7	100	44.4	39.2	100	100	100	75	100	84

Range of Input domain	Traditional Random Approach	Genetic algorithm	Proposed Approach
-100 to 100	7354	1373	100
-100 to 200	25536	1975	300
-100 to 400	92348	2642	500

Figure 13. Test Case size Comparison with test cases in [43] and our technique.

Conclusion and future Study

Test data generation activity in software testing is the process of identifying the program inputs which satisfy the required adequacy criteria. To automatically generate such a potent set of test cases in order to fulfill the adequacy criteria is an intellectually demanding and a very difficult task and it strongly impacts its efficiency and effectiveness. This paper investigated the most prominent techniques used in automatic test data generation including symbolic execution, random and search based. In this study it is found that random approach is a simple process of test data generation and inexpensive compared to symbolic execution and Genetic algorithm. Random approach requires a random number generator and a small amount of software support. It was also found that there are also disadvantages in applying random testing. First, there is no assurance that full coverage can be attained and secondly, it is considered expensive in terms of human resources and it may mean examining the output from thousands of tests. So in order to combat the issues in random approach, we propose an efficient approach in which, we have seeded the test suite with those specific test cases which are not generated by our random generator tool. The future scope of the study will be the application of proposed approach to a large collection of programs and its comparison to other search based techniques.

References

[1]. Bertolino, A., 2007. Software testing research: achievements, challenges, dreams. In: Proceedings of the 1st Workshop on Future of Software Engineering (FOSE'07) at ICSE 2007, pp. 85–103.

[2]. Pezzè, M., Young, M., Software Testing and Analysis, Process, Principles and, Techniques. Wiley. 2007.

[3]. Zhu, H., Hall, P.A.V., May, J.H.R., 1997. Software unit test coverage and adequacy. ACM Computing Surveys 29 (4), 366–427.

[4]. Korel, B. (1990). Automated software test data generation. IEEE Transactions on software engineering, 16(8), 870-879

[5]. Korel, B. 1992. Dynamic method for software test data and generation, J. of Software Testing, Verification, and Reliability 2: 203–213.

[6]. McCabe, T.. Structured Testing. Washington, DC, US Government Printing Office. 1982.

[7]. Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. Normalized restricted random testing. In Reliable Software Technologies Ada-Europe 2003, pages 368–381. Springer, 2003.

[8]. J.C., 1975. A new approach to program testing. In: Programming Methodology. LNCS Vol. 23, pp. 278-290.

[9]. Kenneth V. Hanford. Automatic generation of test cases. IBM Systems Journal, 9(4):242–257, 1970.

[10]. Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. ARTOO: Adaptive random testing for object-oriented software. In Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on, pages 71–80. IEEE, 2008.

[11] Hamlet D. and Taylor R.: Partition testing does not inspire Confidence, IEEE Trans. On Software Engineering, Vol. 16, No. 12, pp. 1402-1411, December 1990

[12] Duran J. W. and Ntafos S. C.: 'An Evaluation of Random Testing', IEEE Transactions on Software Engineering, Vol. SE-10, No. 4, pp. 438-444, July 1984

[13] Myers G.J. The Art of Software Testing, John Wiley and Sons Inc, 1979.

[14] Deason W. H., Brown D. B., Chang K.H. and Cross J. H.: A rule-based software test data generator, IEEE Transactions on Knowledge and Data Engineering, Vol. 3, No. 1, pp.108-117, March 1991

[15] DeMillo R. A., Lipton R. J. and Sayward F. G.: 'Hints on test data selection: Help for the practicing programmer', IEEE Trans. on Computer, Vol. 11, Part 4, pp. 34-41, April 1978

[16] Bertolino, A. An overview of automated software testing, Journal Systems Software, Vol. 15, pp. 133-138, 1991

[17] King, J.C., 1975. A new approach to program testing. In: Programming Methodology. LNCS, Vol. 23, pp. 278-290.

[18] Cadar, C., Dunbar, D., Engler, D.R., 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the Symposium on Operating Systems Design and Implementation, pp. 209–224.

[19] Godefroid, P., Levin, M.Y., Molnar, D.A., 2008. Automated white box fuzz testing. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08).

[20] Khurshid, S., Pasareanu, C., Visser, W., 2003. Generalized symbolic execution for model checking and testing. In: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03), pp. 553–568.

[21] Brumley, D., Poosankam, P., Song, D.X., 0002, J.Z., 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 143–157.

[22] Santelices, R.A., Chittimalli, P.K., Apiwattanapong, T., Orso, A., Harrold, M.J.,2008. Test-suite augmentation for evolving software. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08), pp. 218–227.

[23] Grechanik, M., Csallner, C., Fu, C., Xie, Q., 2010. Is data privacy always good for software testing? In: Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering (ISSRE'10), pp. 368–377.

[24] Pasareanu, C.S., Rungta, N., 2010. Symbolic Path Finder: Symbolic execution of Java byte code. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10), pp. 179–180.

[25] Sen, K., Agha, G., 2006. CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: Proceedings of the 18th International Conference on Computer Aided Verification. (CAV'06), pp. 419–423.

[26] Jayaraman, K., Harvison, D., Ganeshan, V., Kiezun, A., 2009. A concolic white box fuzzer for Java. In: Proceedings of the 1st NASA Formal Methods Symposium, pp. 121–125.

[27] Kähkönen, K., Launiainen, T., Saarikivi, O., Kauttio, J., Heljanko, K., Niemelä, I., 2011. LCT: an open source concolic testing tool for Java programs. In: Proceedings of the 6th Workshop on Byte code Semantics, Verification, Analysis and Transformation (Bycode'11), pp. 75–80.

[28] Sen, K., Marinov, D., Agha, G., 2005. CUTE: A concolic unit testing engine for C. In: Proceedings of the 2005 Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 263–272.

[29] Cadar, C., Dunbar, D., Engler, D.R., 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the Symposium on Operating Systems Design and Implementation, pp. 209–224.

[30] Chipounov, V., Kuznetsov, V., Candea, G., 2011. S2e: a platform for in-vivo multi-path analysis of software systems. In: Proceedings of the 16th International Con- ference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11), pp. 265–278.

[31] Tillmann, N., de Halleux, J., 2008. Pex-White box test generation for .NET, In: Proceedings of the 2nd International Conference on Tests and Proofs (TAP'08), pp. 134–153.

[32] W. Miller and D. Spooner, Automatic generation of floating point test data, IEEE Transactions on Software Engineering, vol. 2, no. 3, pp. 223–226, 1976.

[33] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios, "Application of genetic algorithms to software testing (Application des algorithmes g´en´etiques au test des logiciels)," in 5th International Conference on Software Engineering and its Applications, Toulouse, France, 1992, pp. 625–636.

[34] Oliver Bühler , Joachim Wegener, Evolutionary functional testing, Computers and Operations Research, v.35 n.10, p.3144-3160, October, 2008.

[35] L. C. Briand, J. Feng, and Y. Labiche, Using genetic algorithms and coupling measures to devise optimal integration test orders, in 14th IEEE Software Engineering and Knowledge Engineering (SEKE), Ischia, Italy, 2002, pp. 43–50.

[36] Y. Jia and M. Harman, Constructing subtle faults using higher order mutation testing, in 8th International Working Conference on Source Code Analysis and Manipulation (SCAM 2008. Beijing, China: IEEE Computer Society, 2008.

[37] Z. Li, M. Harman, and R. M. Hierons, Search algorithms for regression test case prioritization, vol. 33, no. 4, pp. 225–237, 2007.

[38] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, Time aware test suite prioritization, in International Symposium on Software Testing and Analysis (ISSTA 06). Portland, Maine, USA: ACM Press, 2006, pp. 1–12.

[39] S. Yoo and M. Harman, Pareto efficient multi-objective test case selection, in International Symposium on Software Testing and Analysis (ISSTA'07). ACM Press, July 2007, pp.140–150.

[40] R. P. Pargas, M. J. Harrold and R. R. Peck, Test data generation using genetic algorithms, Software Testing Verification and Reliability, Vol. 9, pp. 263-282, 1999.

[41] C. C. Michael, G. E. McGraw and M. A. Schatz, Generating software test data by evolution, IEEE Transactions on Software Engineering, Vol. 27, No.12, pp. 1085-1110, 2001.

[42] Baudry B., Fleurey F., Le Traon Y. and Jézéquel J.M. (2005), An Original Approach for Automatic Test Cases Optimization: A Bacteriologic Algorithm'. I.E.E.E. Software, Vol. 22, No. 2, pp.76-82

[43] H.-H. Sthamer, The automatic generation of software test data using genetic algorithms, PhD Thesis, University of Glamorgan, Pontypridd, Wales, Great Britain, 1996.

[44] Michael, Christoph C., et al. Genetic algorithms for dynamic test data generation. Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference. IEEE, 1997.

[45] FA Khan, AK Gupta, and DJ Bora. An Efficient Technique to Test Suite Minimization using Hierarchical Clustering Approach. IJESE, Volume-3 Issue-11, September 2015.

[46] FA Khan, AK Gupta, and DJ Bora. Profiling of Test Cases with Clustering Methodology, International Journal of Computer Applications, 106.14 (2014).