AVISAR – An Automated Framework for Test Case Selection & Prioritization using GA for OOS.

Prashant Vats, Manju Mandot

***Abstract:** In this paper we have presented an automated unified approach called AVISAR for the testing of the Object-oriented Systems (OOS) by Test Case Prioritization (TCP) & their selection using Genetic Algorithm for the OOS. The testing of OOS has become a more challenging task as nowadays it has been widely accepted as a paradigm for large-scale system designing. In this research paper we have also studied the Genetic algorithms in relation to their applications for providing solutions to the various aspects of the OO Testing. As a result after implementing the tool AVISAR using GA's it has proven to be useful in providing effective solutions to resolve the issues related to the OO Testing domain. Thus it can be used for reducing the efforts of the users for testing by efficient selection of effective test cases.*

***Keywords:** Object Oriented Testing, Genetic Algorithm (GA), Fitness Function, Test Case Prioritization (TCP), Jenetics.*

I. INTRODUCTION

The main objective of testing Object-oriented software is to automate the process of generating test cases for Object-oriented software by means of an evolutionary strategy such as the "Genetic Algorithm" (GA). The use of GA in the prioritization of test cases is useful for software evaluators to accelerate the process and achieve a reduction in the software developer's testing efforts. The use of the GA-based rolling test approach tackles the control flow accessibility problem in Object Oriented Software's flow path tests and helps the effective generation of Object Oriented test data. It also aids the test developer in achieving the test objective and generates unit test cases in the conceptual framework in comparison of the randomized test approach. Using GA, he improvises the investigation to generate test cases from the event test sets that are structured using the message that passes from one object to the subsequent and presents us with superior code coverage compared to random tests. GA's key elements are:

a. A symbol of a possible solution to the problem is the chromosome and each of the component of chromosomes is a gene;

b. The encoded chromosomal component in a specific population;

c. For the components of chromosome, a fitness function which is utilized for the allocating of scores for all the chromosomal components;

d. To select parents as per their fitness, n number of operator are selected;

e. Through genetic sequence recombining, a crossover operator generates new offspring for the chromosomes;

f. A mutation operator by which mutation in the chromosomes differentiates the population and presents new offspring's.

In this paper we have presented an automated unified approach called **AVISAR** [18] for the testing of the Object-oriented Systems (OOS). In the First section of the paper we have discussed about the Object Oriented Testing. In Second section of the paper we have reviewed the work on OOT using GA. In the Third Section we have discussed about the Test case selection and their prioritization using Genetic algorithm.

Revised Manuscript Received on April 30, 2020.

* Correspondence Author

Prof. Manju Mandot*, Professor and Director of Directorate of Jan Shikshan and Extension, J.R.N. Rajasthan Vidyapeeth (Deemed University) , India.

Mr. Prashant Vats, Ph.D. in CSE from Banas anasthali Vidyapith, Rajasthan, India.

© The Authors. Published by Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP). This is an open access article under the CC-BY-NC-NDlicense (http://creativecommons.org/licenses/by-nc-nd/4.0/)

II. RELATED WORK ON APPLYING GENETIC ALGORITHMS IN TESTING OF OOS.

Mh. Husain, et al. [1] projected a GA in the assessment of OOMs through the use of tree diagrams. This is done in integration level testing. It entails UML based tests. Solve the optimization problem and increase the efficiency of a system. Better memory management and code reuse is also facilitated by this model.

The authors **Mukesh K. R., et al. [2]** for classes in OOS, presented a technique to produce test cases by means of a GA programming technique. It is executed in the unit level test. It implies evolutionary tests. The benefit of this technique is that it employs the demonstration of the instruction tree in test cases. Java is used for its implementation.

Nicholas A. K., et al., [3] presented a technique to the test order problem of class integration in aspect-oriented programs for GA-based aspect-oriented systems. This is done in integration level test. It entails failure-based testing. Integrating information about the relationships between aspects and classes, in addition to information about the methods affected by the aspects are its advantages; the production of an integration order can prevent change in test sets for classes affected indirectly. However, this completion is performed in small scale projects. Eclipse, AjMetrics, and Java plug-in are used for its implementation.

Satish K. D., et al. [4] proposed a software failure prediction model based on GA with OOMs. It is carried out on unit class level tests. It entails failure-based testing. The anticipated technique is useful for predicting defect-prone classes and is adaptable to OOSs. It is implemented in Java.

Tao X., et al. [5] proposed an Evacon framework that integrates symbolic execution (used for required method arguments generation) and evolutionary testing (used for required method sequences search). This is done at the integrated class level. These are structural tests with Evacon. Test Generation based on Search employs GA algorithms to discover arguments and method chains for the SUT in Evacon's framework. Tests created using the Evacon framework can reach greater branch coverage in comparison of tests that were generated through evolutionary tests or while random tests are performed at the same time, or symbolic execution. But simultaneously, more work is required to reach other types of coverage criteria, like mutation testing and data flow coverage. Java is used for its implementation.

Jie F., et al. [6] intended to reduce the complexity of the stumbling in integration tests using the collective measurement of coupling between classes and GA. This is done in integration level testing. These are tests based on the coupling. The proposed approach can be very functional if immediate outcome are needed and simple cost functions are utilized. However, the case study is carried out with no understanding of the real global minimum, since it cannot be developed with such complexity problems. Java compilers are used for its implementation.

M. Prasanna, et al. [7], in OOSs using GA, presented a model-based technique to the automated test cases generation, analyzing the dynamic conduct of objects as a result of external and internal stimuli. It entails integrated unit and class level tests. It implies mutation tests. Its benefit is that it is valuable when it is necessary to produce test cases following the designing stage is completed and bugs can be identified in an earlier phase of the "software development life cycle" (SDLC). However it cannot scale to bigger systems. Rational Rose is used for its implementation.

A.V.K. Shanthi, et al. [8] to generate test cases, proposed a technique by means of data mining models based on GA. This is done at the integrated class level. It entails mutation tests. The benefit of this approach is that the evolutionary genetic algorithm was intended and executed to show all feasible valid test cases from a class diagram.

Ritu Sibal, et al. [9] presented a system to prioritize test case scenarios, and using GA, identified critical path groups. It is carried out in the unit level test. It implies evolutionary tests. Its benefit is that it addresses the issue of changes in requirements, prioritizing the nodes of a state dependency chart and a control flow chart by means of the "stack-based memory allocation" technique and FI metrics. Rational Rose is the language used for its implementation.

Andrea A., et al. [10] presented an EVOSUITE search-based method to optimize complete test suites using GA to meet a coverage criterion. It is carried out in unit level tests. It implies evidence based on research. The benefit of this technique is that it treats dependencies between predicates and without applying impending exploitation restrictions, also deals with the duration of the test case dynamically. But it cannot be applied to procedural software. Java 10 is used for its implementation.

Alexander, C., et al. [11] proposed the use of attributes and methods located in the OO system classes to achieve using GA, a single objective optimization. This is done at the integrated class level. It entails tests based on couples. Its benefit is that it can be used in the early phases of the process of design or analysis, when incomplete information concerning attribute or method dependencies is on hand, but also in later phases, when source code or even a detailed class diagram is accessible to propose possible enhancements. However, it is designed for smaller systems. You can also investigate to apply it on a large scale. Eclipse and JGAP GA algorithm structures are used for its implementation.

Franck, F., et al. [12] proposed analyzing the GA algorithms application to adapt to the test optimization problem. It is done in the unit test. It involves tests based on mutations. Its advantage is that rather than a test case set, it generates test cases and from one generation to the subsequent, memorizes efficient test cases. But its disadvantage is that the test cases quality improves very gradually and does not attain very high values. .Net Frame is used for its implementation together with the Pylon library.

S. Kanmani, et al. [13] presented a technique called GA-based Class Based Elitist that will be used to test OOS. This is done at the integrated class level. Evolutionary testing is implied by this. Elitism facilitates solutions to an OOP to improve in due course. Significantly accelerates GA performance. It can be used for real-time applications and also for other similar software engineering problems. Java is used for the implementation of Elitism framework.

Kerstin B., et al. [15] introduced an evolutionary test setting with genetic algorithm, which for the majority of structural test methods, performs the generation of fully automatic test data to test actual software modules. It is carried out in the unit level test. It implies evolutionary tests. Its benefit is that it allows the full automation of designing of the test case for several test purposes. Because the method is completely automatic, it can be tested with different input situations with a large number. It also helps improve quality and reduce development costs. However, more work is required to ensure an efficient general test, to evaluate each individual set of tests. It is created with the "Genetic and Evolutionary Algorithms Toolbox" for usage with the Mat lab.

Yoonsik C., et al. [16] presented a specification-based physical fitness function that uses genetic algorithm for class-oriented libraries' evolutionary testing and object-oriented boards. This is done in integrated tests at the class level. It is about evolutionary testing. In this technique, it does not need a complete analysis of the program and functions even in the presence of dynamic dispatch and method substitutions. Java is used for its implementation.

III. SELECTION OF TEST CASES AND THEIR PRIORITIZATION USING GENETIC ALGORITHMS.

In the proposed OOT tool AVISAR, we have used the Genetic algorithm for selection of test cases and their prioritization. This technique is multi-objective optimization techniques which use execution time of the test suit, code coverage and the severity of the test case as the ordering criteria. The objective of the algorithm is to generate test cases sequences which maximize the test suite's ability in terms of code coverage and execution time. The test cases that test the software's important functionalities are identified as severe, according to customer requirement. The severe test case must have to be taken in the test suit for retesting. Once the severe test case is identified and included in the prioritized test suite, the remaining test cases should be generated by our GA-based algorithm. The steps involved in the execution of a GA are: 1. Generate Population. 2. Find the fitness function of the proposed genetic population. 3.Apply the selection, crossover, & mutation process to determine the survival of the fittest one. 4.Evaluate the chromosome & reproduce it.

Let's take an illustrative example to understand it better. We have a pool of 100 test cases and test cases < T5, T45, T23, T75, T53> are identified as severe. The rest 95 test cases in the test suite are optimally reordered by our GA-based algorithm. It is represented in figure 1.

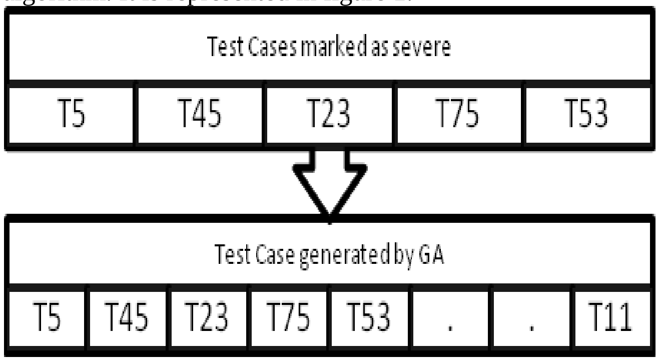


Fig. 3.1. Test Case Prioritization Process using Genetic Algorithm

As timely delivery of the SUT is impartment, it makes the execution time the major criteria for ordering. Our algorithm first considers execution time of the test cases and secondly it considers code coverage of the test case. The inputs to the algorithm are:

Test suite – T
Number of severe test case- S
Stopping criteria – S_{max}
Crossover probability - C_p
Mutation probability - M_p
Initial Population – P

Algorithm 1 *GA-pri*

```
Input: Program with initial population P
Output: Prioritized test cases
1. a <- 1
2. Loop
3. Pa <- null;
4. Loop
5. $P_a$ <- $P_a$ U t        where t is test case selected from T randomly
6. while | Pa |=n          {n=T-S}
7. a <- a+1
8. while a=n
9. e <- 1
10. Loop
11. b <- 1
12. Loop
13. Fb <- CalculateFitness($P_i$)
14. b <- b+1
15. while b <= n
16. b <- 1
17. Loop
18. $P_b$ <- ChooseParent($P_{[random()]}$)
19. $_{Pb+1}$ <- ChooseParent($P_{[random()]}$)
20. $C_1$, $C_2$ <- Crossover($C_p$, $P_b$, $P_{b+1}$)
21. C1 <- Mutation($M_p$, $C_1$)
22. C2 <- Mutation($M_p$, $C_2$ )
23. while b <= n
24. e <- e+1
25. while e <= $S_{max}$
26. $\delta$ min <- ChooseTuplewithminFitness(P,F)
27. return {Set of severe test case} U $\delta$min
```

3.1 Generating a Chromosome population

Initially a GA chromosome function's population is selected & encoded in a random fashion. Each Chromosome has denoted the possible answer to a given problem in order to arrange the test cases in a chromosomal order & our motive is to optimize that genetic sequence.

For e.g. Let us say we have got the following test sequence for a given set of N test cases when N=1 to N onwards. Given N = 10, So we may obtain the following genetic sequence.

T1->T3->T4->T6->T12->T5->T9->T11->T8->T13.

3.2 Evaluation of the fitness of Generated prioritized test case population.

The fitness is calculated by taking time of execution of the test cases and their code coverage. Two type of fitness constitutes the cover all fitness of the chromosome. Our first fitness component which we call primary fitness (F_p), is based on measurement of the final test suite's test adequacy. Primary fitness is computed by taking into account the code coverage of the entire chromosome.

$F_p = \textit{Code Coverage}\ (Ts_i)\ X\ W\ i <= n$ ……………(3.1)

Where Tsi is the code coverage for the test suite, n is number of the initial population. The wait w takes enough time to make the primary fitness dominant in the fitness value. The secondary fitness, the second component is calculated first considering the execution time of the individual test cases and their code coverage. For a single population, F_{cost} will be:

$F_{cost} = \sum_{i=1}^{n} Time(Ts_i)\ X\ \textit{Code Coverage}\ (Ts_i) \quad n<=k$ ……………………..(3.2)

Where n=no of the initial population, k=Test suit size-No of identified severe test case.

After calculating the F_{cost} for all test cases, considering the highest coverage of one of the test cases in the process and the execution time of each individual test case we calculate the maximum possible cost F_{max}.

$F_{max} = Max(\text{Code Coverage}\ (Ts_i))\ X \sum_{i=1}^{k} Time(Ts_i)\ i<=k$ …………….(3.3)

Then the secondary fitness

$F_s = \frac{Fcost\ j}{Fmax\ j}$ j<=n ……………(3.4)

It is possible that any two test cases may cover same or part of the same statements. Which make it impossible, to sum up, the coverage of individual test case. That is why secondary fitness is introduced. The secondary fitness considers incremental code coverage of test cases.

To understand incremental coverage better let's take an example, if a chromosome of size 3 is considered with test case <T6, T65, T39> and their execution time are < 6, 3, 2 > seconds.

Let first <T6> selected, in the second iteration T65 is added to it <T6, T65 >, and similarly in next iteration T39 is added <T6, T65, T39>. There combine coverage is 0.1%, 0.3%, and 0.6% coverage of the program. Using equation 7.1 the primary fitness $F_p = 0.6 * 100 = 60$. TCP is all about early fault detection, the incremental coverage is considered. The secondary fitness is calculated by calculation F_{cost} and F_{max}.

$F_{cost} = (6 \times 0.1) + (3 \times 0.3) + (2 \times 0.6) = 2.7$

F_{cost} is subject to comparison with its possible highest value possible, which is called the optimal secondary fitness; F_s. When the first test covers all code for that test suite, the fitness of a prioritization would be optimal secondary fitness. In our illustration for example T6 covered 100% of all statements covered by T then

$$F_{max} = .6 \times (6+3+2) = 6.6.$$

Now secondary fitness using equation 4.4

$$F_s = F_{cost} \div F_{max}$$
$$= 2.7 \div 6.6$$
$$= 0.41$$

The fitness of a chromosome is a summation of primary fitness and secondary fitness. For our example the fitness is

$$F = F_p + F_s$$
$$= 60 + 0.41$$
$$= 60.41$$

3.3 Apply selection of Test cases for Individual genes.

In general, the selection of Chromosomes will be dependent onto the fitness values of the gene. The possible chromosome with a higher or lower value would be chosen as a base for our problem definition.

3.4 Applying the Crossover & Mutation over a chosen gene.

The parents of gene will be chosen & combined in a random manner. This process of generation of genetic chromosomes into a random order is called a crossover. There would be two types of crossover in genes.

1. Single point based crossover.
2. Multiple point based crossover.

For e.g. given two sequences of test cases that has possibility of detection of fault in an Object oriented code. We have got two parents.

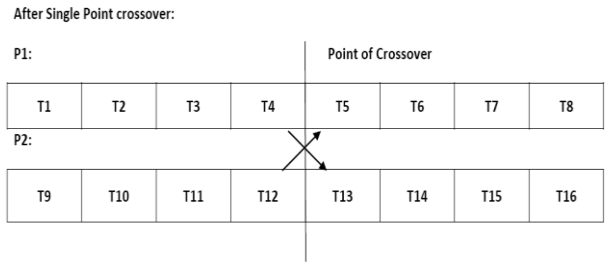


Fig. 3.2. Example of Crossover

Crossover occurs at the crossover point of 4 marked with dashed line; the resultant genetic offspring would be as follows:

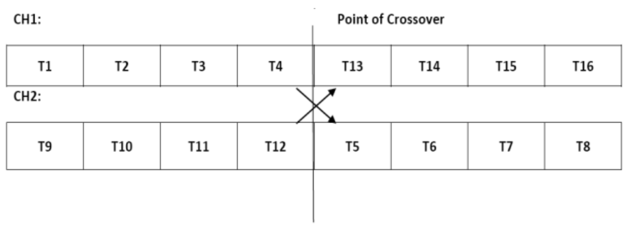


Fig. 3.3. Resultant children chromosomes after crossover

For C1, we will write the first portion of P1 as its in original form and after putting a constraint that for the second part of P2 we will not include test case into C1.

For the mutation onto any two genes we, have to select them randomly along with their chromosome & then swap them with each other.

For e.g. when T3 & T7 get selected randomly during mutation performed onto a chromosome.

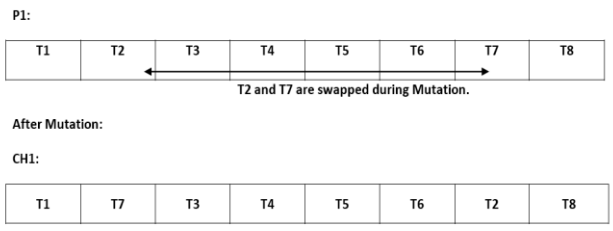


Fig. 3.4. Example of Mutation

3.5 Termination criteria.

The termination criteria could be selected in as many forms a may be defined like as:

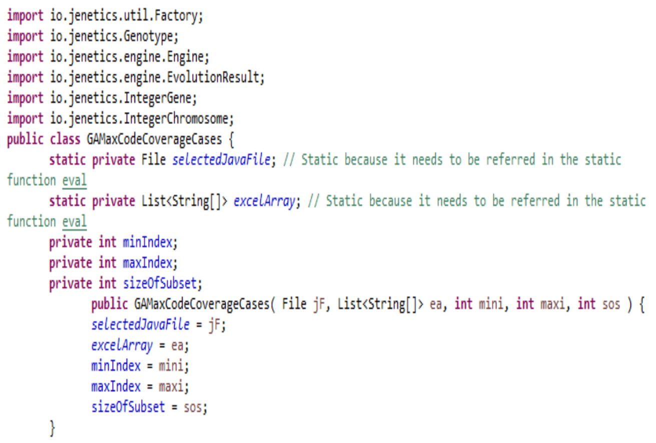
1. Attainment of predefined value.
2. Number of generations of Chromosome.

A series of non-similar variance in fitness values for each possible generation of a Chromosome.

IV. EXPERIMENTAL DESIGN.

During the designing and developing of the tool AVISAR, for our analysis we have implemented the coding in Java using the Jenetics library, Java Parser Classes and the Java Code Coverage, Jacoco for implementing our testing of Java programs using the Genetic Algorithms.

4.1 Jenetics Library- Jenetics is an advanced library based on **genetic evolutionary programming**, which was written in modern Java. It has been designed with a clear separation of the various algorithm concepts, and. g. genotype, Gene, chromosome, fitness function, phenotype, and population. Jenetics allows you to maximize or minimize a particular physical function without changing it. The library, unlike other GA implementations, uses the evolution *stream* concept (EvolutionStream) to perform the evolution steps. Because the Java Stream interface is implemented by EvolutionStream, it works smoothly with the rest of the Java Stream API.

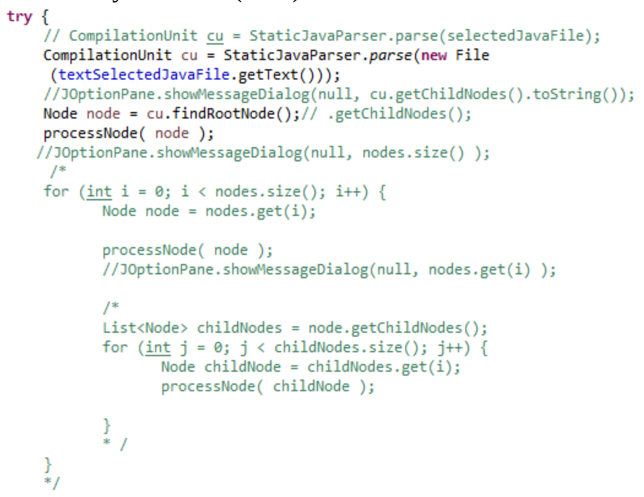


```
import io.jenetics.util.Factory;
import io.jenetics.Genotype;
import io.jenetics.engine.Engine;
import io.jenetics.engine.EvolutionResult;
import io.jenetics.IntegerGene;
import io.jenetics.IntegerChromosome;
public class GAMaxCodeCoverageCases {
        static private File selectedJavaFile; // Static because it needs to be referred in the static
function eval
        static private List<String[]> excelArray; // Static because it needs to be referred in the static
function eval
        private int minIndex;
        private int maxIndex;
        private int sizeOfSubset;
                public GAMaxCodeCoverageCases( File jF, List<String[]> ea, int mini, int maxi, int sos ) {
                selectedJavaFile = jF;
                excelArray = ea;
                minIndex = mini;
                maxIndex = maxi;
                sizeOfSubset = sos;
        }
```

Fig. 4.1. To show the use of Jenetics library.

4.2 Java Parser. The JavaParser library allows you to interact with Java source code as a Java object representation in a Java environment.

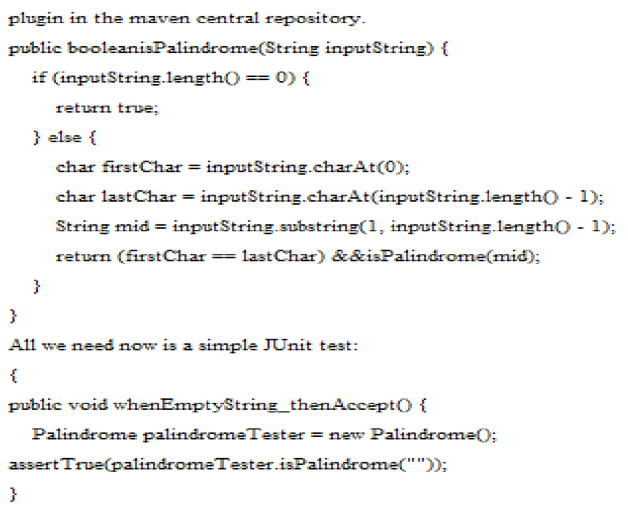
More formally we refer to this object representation as an Abstract Syntax Tree (AST).



```
try {
    // CompilationUnit cu = StaticJavaParser.parse(selectedJavaFile);
    CompilationUnit cu = StaticJavaParser.parse(new File
     (textSelectedJavaFile.getText()));
    //JOptionPane.showMessageDialog(null, cu.getChildNodes().toString());
    Node node = cu.findRootNode();// .getChildNodes();
    processNode( node );
   //JOptionPane.showMessageDialog(null, nodes.size() );
     /*
    for (int i = 0; i < nodes.size(); i++) {
            Node node = nodes.get(i);

            processNode( node );
            //JOptionPane.showMessageDialog(null, nodes.get(i) );

            /*
            List<Node> childNodes = node.getChildNodes();
            for (int j = 0; j < childNodes.size(); j++) {
                    Node childNode = childNodes.get(i);
                    processNode( childNode );

            }
            * /
    }
    */
```

Fig. 4.2. To show the use of Java Parser.

4.3 Jacoco Code Coverage - Code coverage is a software metric used to measure how many lines of our code are executed during automated tests running the test using JUnit will automatically set in motion the JaCoCo agent, thus, it will create a coverage report in binary format in the target directory –target/jacoco.exec.



```
plugin in the maven central repository.
public booleanisPalindrome(String inputString) {
    if (inputString.length() == 0) {
        return true;
    } else {
        char firstChar = inputString.charAt(0);
        char lastChar = inputString.charAt(inputString.length() - 1);
        String mid = inputString.substring(1, inputString.length() - 1);
        return (firstChar == lastChar) &&isPalindrome(mid);
    }
}
All we need now is a simple JUnit test:
{
public void whenEmptyString_thenAccept() {
    Palindrome palindromeTester = new Palindrome();
assertTrue(palindromeTester.isPalindrome(""));
}
```

Fig. 4.3. To show the use of Java Code Coverage.

4.4 Java Modules used for coding- Following are the Modules of our coding that we have used in implementing for the test case prioritization using Genetic Algorithm in Java.

Module 1.

final Factory<Genotype<IntegerGene>>gtf = Genotype.of(
    IntegerChromosome.of(minIndex, maxIndex,
sizeOfSubset)
);

Module 2.

"Class Genotype<G extends Gene<?,G>>
java.lang.Object
org.jenetics.Genotype<G>
All Implemented Interfaces:
Serializable, Iterable<Chromosome<G>>, Factory<Genotype<G>>, Verifiable
public final class Genotype<G extends Gene<?,G>>
extends Object
implements Factory<Genotype<G>>,
Iterable<Chromosome<G>>, Verifiable, Serializable

Module 3.

io.jenetics.engine
Class Engine<G extends Gene<?,G>,C extends
Comparable<? super C>>
java.lang.Object
io.jenetics.engine.Engine<G,C>
All Implemented Interfaces:
EvolutionStreamable<G,C>,
Function<EvolutionStart<G,C>,EvolutionResult<G,C>>
EvolutionStreamable<G,C>,
Function<EvolutionStart<G,C>,EvolutionResult<G,C>>
public final class Engine<G extends Gene<?,G>,C extends
Comparable<? super C>>
extends Object
implements
Function<EvolutionStart<G,C>,EvolutionResult<G,C>>,
EvolutionStreamable<G,C>

Module 4.

io.jenetics.engine
Class EvolutionResult<G extends Gene<?,G>,C extends
Comparable<? super C>>
java.lang.Object
io.jenetics.engine.EvolutionResult<G,C>
Type Parameters:
G - the gene type
C - the fitness type
All Implemented Interfaces:
Serializable, Comparable<EvolutionResult<G,C>>
public final class EvolutionResult<G extends Gene<?,G>,C
extends Comparable<? super C>>
extends Object
implements Comparable<EvolutionResult<G,C>>,
Serializable

Module 5.

```
<plugin>
<groupId>org.jacoco</groupId>
<artifactId>jacoco-maven-plugin</artifactId>
<version>0.7.7.201606060606</version>
<executions>
<execution>
<goals>
<goal>prepare-agent</goal>
</goals>
</execution>
<execution>
<id>report</id>
<phase>prepare-package</phase>
<goals>
<goal>report</goal>
</goals>
</execution>
</executions>
</plugin>
```

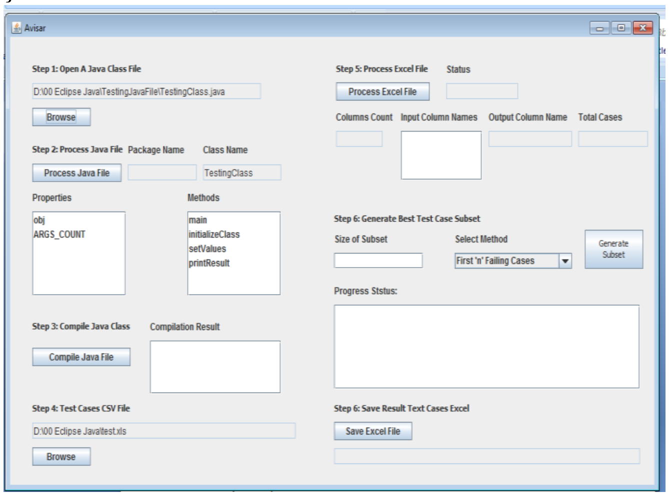
The link provided here-before will always lead you to the latest version of the plugin in the maven central repository.

```
public booleanisPalindrome(String inputString) {
    if (inputString.length() == 0) {
```

```
 return true;
    } else {
        char firstChar = inputString.charAt(0);
        char lastChar = inputString.charAt(inputString.length() -
1);
        String mid = inputString.substring(1,
inputString.length() - 1);
        return (firstChar == lastChar) &&isPalindrome(mid);
    }
}
```

All we need now is a simple JUnit test:

```
{
public void whenEmptyString_thenAccept() {
    Palindrome palindromeTester = new Palindrome();
assertTrue(palindromeTester.isPalindrome(""));
}.
```



Avisar

Step 1: Open A Java Class File: D:\00 Eclipse Java\Testing\JavaFile\TestingClass.java

Step 2: Process Java File — Package Name: — Class Name: TestingClass

Properties: obj, ARGS_COUNT — Methods: main, initializeClass, setValues, printResult

Step 3: Compile Java Class — Compilation Result:

Step 4: Test Cases CSV File: D:\00 Eclipse Java\test.xls

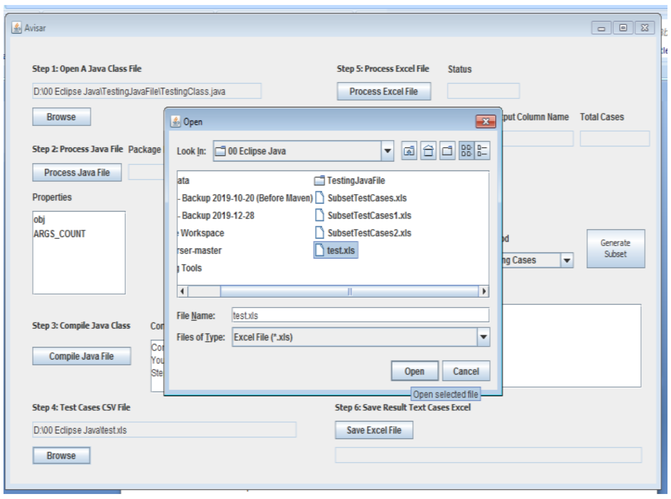
Step 5: Process Excel File — Status: — Columns Count: — Input Column Names: — Output Column Name: — Total Cases:

Step 6: Generate Best Test Case Subset — Size of Subset: — Select Method: First 'n' Failing Cases

Progress Ststus:

Step 6: Save Result Text Cases Excel

Fig. 4.4. To show the interface of the tool AVISAR for implementation of the proposed work.



Avisar

Step 1: Open A Java Class File: D:\00 Eclipse Java\Testing\JavaFile\TestingClass.java

Step 2: Process Java File — Package

Properties: obj, ARGS_COUNT

Step 3: Compile Java Class

Step 4: Test Cases CSV File: D:\00 Eclipse Java\test.xls

Step 5: Process Excel File — Status: — Output Column Name — Total Cases

Step 6: Save Result Text Cases Excel

Open — Look In: 00 Eclipse Java

ata; Backup 2019-10-20 (Before Maven); Backup 2019-12-28; Workspace; rser-master; Tools; TestingJavaFile; SubsetTestCases.xls; SubsetTestCases1.xls; SubsetTestCases2.xls; test.xls

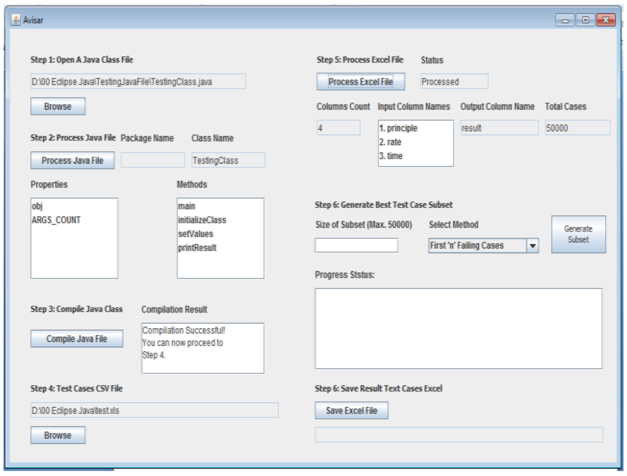
File Name: test.xls

Files of Type: Excel File (*.xls)

Fig. 4.5. To show the selection of any Object oriented Source code to be tested using AVISAR

4.5 GA used for TCP & their Selection: Following is the GA that we have used for the TCP of the selected test cases that ensures the maximum fault detection in the tool AVISAR.

```
public List<Integer> startGA()
{
        int i; // For traversing the genes in the result
        IntegerChromosome ich; // To store chromosome of
result
        IntegerGene ig; // To store gene of chromosome of result
        Integer caseNumber; // To store individual case numbers
in the gene of chromosome of result. // 1. Define the suitable
genotype (factory) for the problem
        // A factory is just a chromosome generator
        // We are here just letting the factory know how to
generate the chromosome
        //final Factory<Genotype<BitGene>> gtf =
Genotype.of( BitChromosome.of(10,0.5) );
        final Factory<Genotype<IntegerGene>> gtf =
Genotype.of(
            //IntegerChromosome.of( minIndex, maxIndex ),
            IntegerChromosome.of( minIndex, maxIndex,
sizeOfSubset )
            /* ,sizeOfSubset */
        );
        // 3. Creation of the execution environment
        //final Engine<BitGene, Integer> engine =
Engine.builder(HelloWorld::eval,gtf).build();
        final Engine<IntegerGene, Integer> engine =
Engine.builder(GAMaxFailingCases::eval,gtf).build();
        // 4. Starting of the execution (evolution) and collection
of the result
        //final Genotype<BitGene> result =
engine.stream().limit(100).collect(EvolutionResult.toBestGe
notype());final Genotype<IntegerGene> result =
engine.stream().limit(10).collect(EvolutionResult.toBestGen
otype());
        // This may stay the same
        System.out.println("Result : \n\t" + result);
        System.out.println( "Count of failed cases in Result are:
" + String.valueOf(eval(result)) );
            // generating list of test cases by case number
        List<Integer> subsetCases = new ArrayList<Integer>();
        ich =
result.getChromosome().as(IntegerChromosome.class);
        for( i=0; i<=(ich.length()-1); i++ ) {
            ig = ich.getGene(i);
            caseNumber = ig.intValue();
            subsetCases.add(caseNumber);}
```



Avisar

Step 1: Open A Java Class File: D:\00 Eclipse Java\Testing\JavaFile\TestingClass.java

Step 2: Process Java File — Package Name: — Class Name: TestingClass

Properties: obj, ARGS_COUNT — Methods: main, initializeClass, setValues, printResult

Step 3: Compile Java Class — Compilation Result: Compilation Successful! You can now proceed to Step 4.

Step 4: Test Cases CSV File: D:\00 Eclipse Java\test.xls

Step 5: Process Excel File — Status: Processed

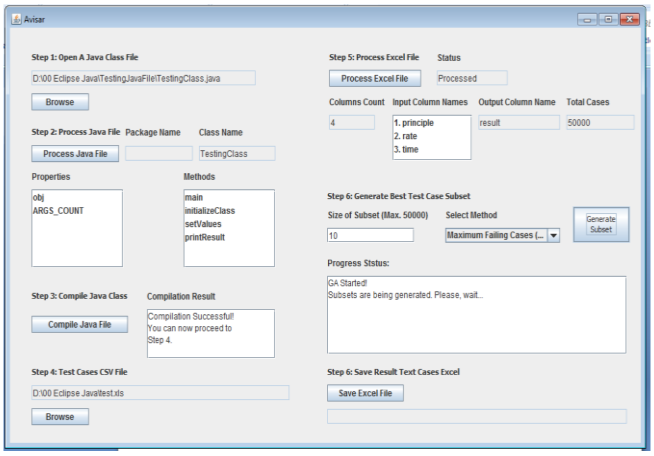
Columns Count: 4 — Input Column Names: 1. principle, 2. rate, 3. time — Output Column Name: result — Total Cases: 50000

Step 6: Generate Best Test Case Subset — Size of Subset (Max. 50000): — Select Method: First 'n' Failing Cases

Progress Ststus:

Step 6: Save Result Text Cases Excel

Fig. 4.6.To show the execution of test cases using Genetic Algorithm



Avisar

Step 1: Open A Java Class File
D:\00 Eclipse Java\TestingJavaFile\TestingClass.java
Browse

Step 2: Process Java File | Package Name | Class Name
Process Java File | | TestingClass

Properties: obj, ARGS_COUNT
Methods: main, initializeClass, setValues, printResult

Step 3: Compile Java Class | Compilation Result
Compile Java File
Compilation Successful! You can now proceed to Step 4.

Step 4: Test Cases CSV File
D:\00 Eclipse Java\test.xls
Browse

Step 5: Process Excel File | Status
Process Excel File | Processed

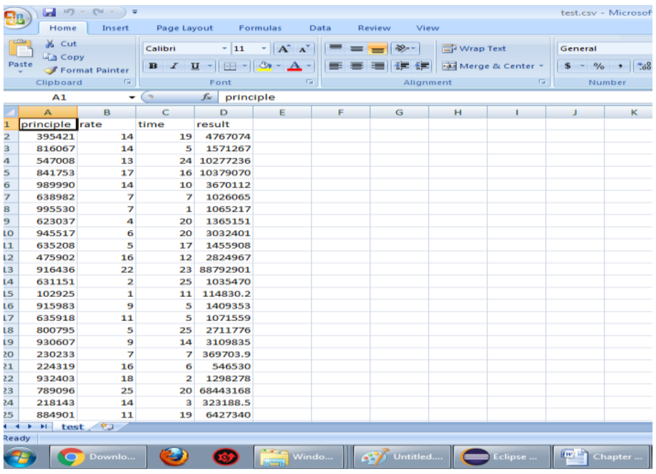
Columns Count: 4
Input Column Names: 1. principle 2. rate 3. time
Output Column Name: result
Total Cases: 50000

Step 6: Generate Best Test Case Subset
Size of Subset (Max. 50000): 10
Select Method: Maximum Failing Cases (...
Generate Subset

Progress Ststus:
GA Started!
Subsets are being generated. Please, wait...

Step 6: Save Result Text Cases Excel
Save Excel File

Fig. 4.7. To show the immediate optimized result of test cases using the GA.



A1 principle

	A	B	C	D
1	principle	rate	time	result
2	395421	14	19	4767074
3	816067	14	5	1571267
4	547008	13	24	10277236
5	841753	17	16	10379070
6	989990	14	10	3670112
7	638982	7	7	1026065
8	995530	7	1	1065217
9	623037	4	20	1365151
10	945517	6	20	3032401
11	635208	5	17	1455908
12	475902	16	12	2824967
13	916436	22	23	88792901
14	631151	2	25	1035470
15	102925	1	11	114830.2
16	915983	9	5	1409353
17	635918	11	5	1071559
18	800795	5	25	2711776
19	930607	9	14	3109835
20	230233	7	7	369703.9
21	224319	16	6	546530
22	932403	18	2	1298278
23	789096	25	20	68443168
24	218143	14	3	323188.5
25	884901	11	19	6427340

Fig.4.8. To show the Test cases being designed for OOP under Test.

V. EXPERIMENTAL RESULT ANALYSIS.

During our implementation and testing of our codes for the test case prioritization using AVISAR, we have obtained the following Fig. 5.1 for the method visibility, Polymorphism, Inheritance, and Encapsulation for estimation of effort for smaller codes implemented using four modules in Java. Fig. 5.2 & 5.3 also illustrated the results obtained. Fig. 5,2 is a Graph to show execution of test cases with GA and without using GA in AVISAR. Fig. 5,3 shows the Graph to show the code coverage during execution of Test cases using GA & without GA.

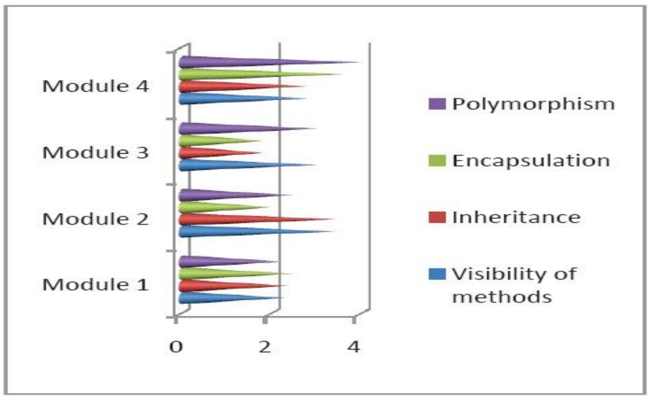


Fig. 5.1 To show the overall performances of OOD estimator

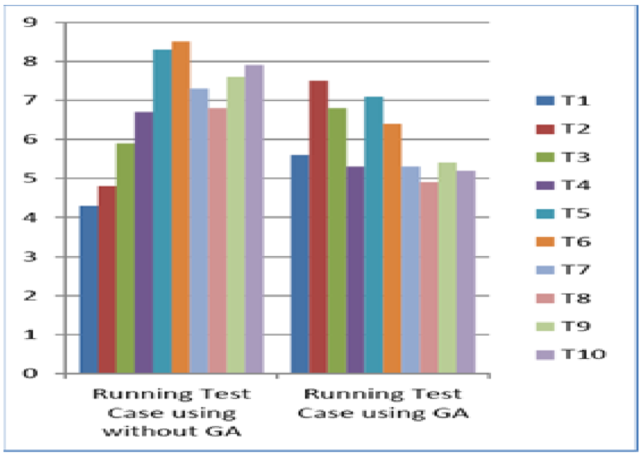


Fig. 5.2 Graph to show execution of test cases with GA and without using GA.

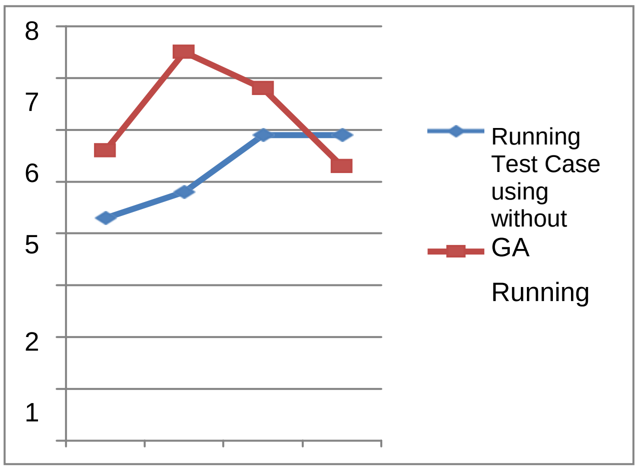


Fig 5.3 Graph to show the code coverage during execution of Test cases using GA & without GA.

VI. CONCLUSIONS AND FUTURE WORK.

Applications of the Genetic Algorithm technique to the problem of Test cases selection and prioritization leads to the solutions which are near optimal. The implemented tool AVISAR for the Test cases selection and prioritization using Genetic algorithm has also provided us encouraged results with reduced time complexity and thus finding its application in the industry. Also as the future work this tool can be further extended in implemented form to test large chunks of codes finding its application in the software industry.

REFERENCES:

1. Manoj Kumar, Mh. Husain," An Efficient Algorithm for Evaluation of Object-Oriented Models", pub. in International Journal of Computer Applications Volume 24- No.8, pg. no: 11-15, June 2011.
2. Nirmal K. G., Mukesh K. R.," Using Genetic Algorithm for Unit Testing Of Object Oriented Software", pub. in IJSSST, Vol. 10, No. 3, pg. no: 99-104, 2009.
3. Romain Delamare, Nicholas A. Kraft," A Genetic Algorithm for Computing Class Integration Test Orders for Aspect-Oriented Systems ", pub. in IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pg. no: 804 - 813, 2012.
4. Parvinder S. Sandhu, Satish Kumar Dhiman," A Genetic Algorithm Based Classification Approach for Finding Fault Prone Classes", pub. in proceedings of World Academy of Science, Engineering and Technology, Vol. 60, pg. no: 485- 488, 2009.

5. Kobi Inkumsah, Tao Xie," Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution", pub. in proceedings of 23rd IEEE ACM International Conference on Automated Software Engineering, pg. no. 297 - 306, 2008.
6. Lionel C. Briand, Jie Feng," Experimenting with Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders", pub. in Carleton University, Technical Report SCE-02-03, Version 3, October 2002.
7. M. Prasanna, K.R. Chandran," Automatic Test Case Generation for UML Object diagrams using Genetic Algorithm", pub. in Int. J. Advance. Soft Comput. Appl., Vol. I, No. I, pg. 19-32, July 2009.
8. A. V. K. Shanthai, G. Mohan kumar," Automated Test Cases Generation for Object Oriented Software", pub. in Indian Journal of Computer Science and Engineering, Vol. 2, No. 4, pg.543-546, 2011.
9. Sangeeta Sabharwal, Ritu Sibal, "Applying Genetic Algorithm for Prioritization of Test Case Scenarios Derived from UML Diagrams", pub. in IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 3, No. 2, pg. no: 433-444, May 2011.
10. Gordon Fraser, Andrea Arcuri," Evolutionary Generation of Whole Test Suites", pub. in proceedings of 11th IEEE International Conference on Quality Software (QSIC), pg. no. 31 - 40, 2011.
11. Margaritis B., Alexander c.," Placement of Entities in Object-oriented Systems by means of a Single-objective Genetic Algorithm", pub. in proceedings of Fifth IEEE International Conference on Software Engineering Advances (ICSEA), pg. no: 70-75, 2010.
12. Benoit Baudry, Franck Fleurey," From genetic to bacteriological algorithms for mutation-based testing", pub. in Journal of SOFTWARE TESTING, VERIFICATION AND RELIABILITY, Vol. 15, pg. no: 73-96, 2005.
13. P. Maragathavalli, S. Kanmani, "Evolutionary Testing Approach for Solving Path- Oriented Multivariate Problems", pub. in ACEEE Int. J. on Information Technology, Vol. 3, No. I, pg. no: 54 57, March 2013.
14. Joachim Wegener, Kerstin Buhr," Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing", pub. in proceedings of ACM GECCO '02 Proceedings of the Genetic and Evolutionary Computation, pg. no: 1233-1240, 2002.
15. Yoonsik Cheon, Myoung Kim," A Fitness Function for Modular Evolutionary Testing of Object-Oriented Programs", pub. in Genetic and Evolutionary Computation Conference, Seattle, WA, USA, July 8-12, 2006, pages 1952-1954, ACM Press, 2006.
16. Vats, P., "AVISAR - a three tier architectural framework for the testing of Object Oriented Programs" pub. In Second IEEE International Innovative Applications of Computational Intelligence on Power, Energy and Controls with their Impact on Humanity (CIPECH), 2016.

AUTHORS PROFILE

Prof. Manju Mandot is a Professor and Director of Directorate of Jan Shikshan and Extension, J.R.N. Rajasthan Vidyapeeth (Deemed University). She has 27 years of teaching experience. Her research interest includes image processing, E- governance, women empowerment with technology. She is esteemed member of Computer Society of India

Mr. Prashant Vats is working in the field of CSE & IT as an Assistant Professor from past 11 years. He is pursuing Ph.D. in CSE from Banas anasthali Vidyapith, Rajasthan..