Dharmik
Trivedi
101235680

EvoSuite

1 INTRODUCTION

Scope of project

The scope of this project includes the generation of test cases of objects from artifacts infrastructure repository, running of the generated cases, and recording the outputs. To conduct a case study for the objects and study the relevant research material to examine the uses and limitations of EvoSuite. Then, establish a conclusion and connect the findings with the theories.

Goal of project

The goal of the project is to precisely understand EvoSuite, to conduct a case study and learn whether EvoSuite helps to reveal the faults or not. In addition, to record the quantitative and qualitative results and conclude how and why questions about EvoSuite for selected subject softwares.

Structure of the report

STATE OF THE ART OVERVIEW

Methodology

To create automatic test generation, the maven plugin is used with evosuite. The Maven version used for this project is Apache Maven 3.8.4. The versions of java are: java version "1.8.0_202" Java(TM) SE Runtime Environment (build 1.8.0_202-b08), Java HotSpot(TM) 64-Bit Server VM (build 25.202-b08, mixed mode). All the commands are fired in the command prompt for the test case generation and run.

The bin path of jdk is added to environment variable PATH. Eclipse IDE is used to create a Maven project and command prompt is used to fire the commands for test case build, generation and run. The output is observed by refreshing the Maven projects on package explorer.

To assess the coverage report, JaCoCo java library is configured with evosuite. Eclipse IDE is chosen instead of IntelliJ because of prior experience of working.

EvoSuite

EvoSuite produces the test cases for the java class in the java code automatically. It gives an analysis of the testing properties of the particular java class. To search faults in specific software, test cases are needed to execute the software properly. In addition, the oracle checks whether the observed output is achieved or not at the end of test case. EvoSuite, to generate the automatic test cases applies a novel hybrid method. It also gives about the overall coverage. EvoSuite, by default targets not only lines of code but attempts to satisfy a range of different testing criteria, including things like mutation testing. Some of the testing goals described by these criteria are infeasible, which means that there exist no tests that satisfy; some other goals are just so difficult to cover that EvoSuite cannot easily produce the tests. The assertions produced by EvoSuite can be minimized to avoid the coverage of non-mandatory statements.

Important Properties of EvoSuite

1 Generation of JUnit 4 tests for the selected classes

2 Optimization of different coverage criteria, like lines, branches, outputs and mutation testing
3 Tests are minimized: only the ones contributing to achieve coverage are retained
4 Generation of JUnit asserts to capture the current behavior of the tested classes
5 Tests run in a sandbox to prevent potentially dangerous operations
Virtual file system
6 Virtual network and virtual file system

How EvoSuite is used?

EvoSuite is a tool that is specifically designed to make the generation of the test cases of java code easier and precise. When the java code is created, there are lots of chances that it produces errors if not being tested. So, testing should be a primary goal to render robust software. In order to do that it is advised to perform precise testing. Considering the language of code is Java, testing needs to be done to check whether the system behaves correctly for expected and unexpected for diffenent set off inputs. Here, the use of EvoSuite comes. EvoSuite helps to create a test case. It assists in deciding whether a test outcome is successful or not by creating test cases. It could be used to solve two things. 1) comparing the actual and expected output 2) deciding different inputs for test case. For example. Consider the java code below.

```
1  package intro_java;
2  import java.util.Scanner;
3
4  public class programmingInJava {
5
6      public static void main(String[] args) {
7          Scanner sc= new Scanner(System.in);
8          System.out.print("Enter first number- ");
9          int a= sc.nextInt();
10         System.out.print("Enter second number- ");
11         int b= sc.nextInt();
12         System.out.print("Enter third number- ");
13         int c= sc.nextInt();
14         int d=a+b+c;
15         System.out.println("Total= " +d);
16
17     }
18
19 }
```

Here, the code takes four integers, performs addition, and returns the result. What EvoSuite does is that it generate different test cases for different inputs and run the code. for instance, test@1 a=1, b=100, c=-201 test@2 a=200, b= -2098, c= 576. The comparison is done for the actual and expected value. The information about the code's coverage specification is returned (branch coverage[not applicable here], line coverage, overall coverage, total number of generated test cases). Generally, oracle is selected by

the user. EvoSuite tries different inputs on the test cases and checks how the class code behaves. As per the code coverage, we determine how many parts of the code are tested.

What happens, when we use EvoSuite to generate a large software that has so many classes, is that many of the test cases generated by EvoSuite fail. In such a situation, we analyze the reason for the test case failure, explore what made EvoSuite to skip a particular class without being tested (if so), we may use a code coverage library such as JaCoCo to crosscheck whether the coverage details given by EvoSuite is accurate or not. If not, then why? For example, how many methods, classes, blocks, and lines are missed?

Why do we use EvoSuite?

We use EvoSuite to determine the overall coverage of a code written in java language. Over the years, a lot of new changes and libraries are added to EvoSuite which make the generation of test cases more precise and accurate.

To detect errors in software, one needs test cases that execute the software as expected, and oracles that check the correctness of the observed behavior when running these test cases. [5] EvoSuite helps here because it automatically produces test cases with assertions for classes written in Java language. This is achieved by EvoSuite which applies a novel hybrid approach that generates and optimizes whole test suites towards satisfying a coverage criterion. For the produced test suites, EvoSuite suggests possible oracles by adding small and effective sets of assertions that concisely summarize the current behavior; these assertions allow the developer to detect deviations from expected behavior, and to capture the current behavior in order to protect against future defects breaking this behavior. [5]

EvoSuite uses evolutionary search to automatically generate test suites for Java classes [2]. It takes as input the name of a target class as well as the class path describing where the compiled bytecode of the class as well as its dependencies are located. A basic static analysis extracts information about the relevant classes as well as their constructors, methods, and fields. The bytecode is instrumented while classes are loaded, such that EvoSuite can produce execution traces for test executions, and to avoid test flakiness by replacing non-deterministic calls with deterministic, mocked versions. EvoSuite then applies meta-heuristic search algorithms to automatically produce a set of JUnit test cases aimed at maximizing code coverage. [2]

Originally, EvoSuite was implemented to optimize entire test suites with respect to their overall code coverage. The latest search algorithm, which is now used by default, is the Dynamic Many-Objective Sorting Algorithm (DynaMOSA) search algorithm [3] which operates at the test case level. The genetic encoding of test cases in EVOSUITE consists of variable-length sequences of Java statements (e.g., primitive statements as well as calls on constructors or methods). Standard evolutionary search operators (e.g., selection, crossover mutation) are adapted for this representation. EVOSUITE supports multiple different coverage criteria that can be optimized simultaneously. The core fitness

functions in EVOSUITE are based on traditional heuristics for code coverage, such as the branch distance and the approach level. Further fitness functions are based on mutation testing as well as other basic criteria [4]

Maven Plugin

Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time.

Important Properties of Maven:
1 Makes the build process easy
2 Provides a uniform build system
3 Provides quality project information
4 Encourages better development practices

JaCoCo

JaCoCo is a free code coverage library for Java. In the further section of the report, the configuration of JaCoCo is explained.

2 DESIGN OF THE CASE STUDY

Steps to design the case study

The approach which has taken, is Maven plugin, to use EvoSuite. The steps are following:

1 The software-artifact infrastructure repository (which is available at https://sir.csc.ncsu.edu/portal/index.php) comes with an .tar.gz extension. The extraction of software is done at the local machine using WinRAR on a suitable path.

2 A Maven project in Eclipse IDE is created and the folder/folders are added (packages) (which has all the java files from which you to want to generate the test cases) of software to src/main/java path of the project in eclipse.

3 In order to set up Maven project to run the EvoSuite tests the,configure the Maven plugin by adding some changes to pom.xml file (which is created as a default one, when generating a Maven project in eclipse IDE)

4 The jUnit dependency (to run the generated test cases in the future) before the </project> tag in the pom.xml is added (dependency is taken from https://www.evosuite.org/documentation/maven-plugin/).

```
    <artifactId>evosuite-arrayPartition</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
```

5 In order to tell the Maven to use the EvoSuite runtime library, one more dependency (which is EvoSuite-standalone-runtime dependency)is added into the pom.xml file.

```
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.evosuite</groupId>
            <artifactId>evosuite-standalone-runtime</artifactId>
            <version>1.0.6</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

6 Now, the Maven needs to know that how to use EvoSuite as a part of built (to generate a class file/to convert the code into the byte code). For that, <bulid> dependency below </dependency> section of pom.xml is added.

```
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.evosuite.plugins</groupId>
                <artifactId>evosuite-maven-plugin</artifactId>
                <version>1.0.6</version>
            </plugin>
        </plugins>
    </build>
```

7 To let the EvoSuite plugin available at Maven central, <pluginReposatories> by specifying the EvoSuite's URL in pom.xml is added.

```
        </plugins>
    </build>

    <pluginRepositories>
        <pluginRepository>
            <id>EvoSuite</id>
            <name>EvoSuite Repository</name>
            <url>http://www.evosuite.org/m2</url>
        </pluginRepository>
    </pluginRepositories>

</project>
```

8 The changes made at pom.xml are saved.

9 At the location of Maven project in command prompt, *mvn EvoSuite:generate* command (which downloads the EvoSuite and its dependencies when generating the tests for the first time)is used.

10 Now, *mvn EvoSuite:export* command is entered to integrate the tests with source tree.

11 By typing *mvn test*, the generated test cases (location of the generated test: src/test/EvoSuite) are run.

12 It can also be viewed at srs/test/java module in the project in eclipse.

It should be noted that an advantage of using Maven plugin is that we do not need to install EvoSuite on a local machine as Maven will take of this automatically.

The system under test

In this project, the system under the test is the software which are taken from the artifact infrastructure repository for case studies. It includes Array-Partition, Binary Heap, Vector, Binary Search Tree, Disjoint set, Doubly-Linked List, Stack, and Sorting. The classes of the source java files of a subject software are presented to EvoSuite for test case compilation, generation, and run.

Faulty data

It is expected that some of the subjects of this repository come along with existing faults. It will be important to note that whether EvoSuite helps reveal the faults or not.

The subject softwares for the case studies are:
1 Array-Partition
2 Binary Heap
3 Vector
4 Binary Search Tree
5 Disjoint set
6 Doubly-Linked List
7 Stack
8 Sorting

3 ANALYSIS OF THE RESULTS

In this section, the results obtained from case studies are described and analyzed.

Test results

1 Array-Partition

This class divides an array into upper and lower halves, using the first element (a[0]) as a pivot. It does this by moving all elements whose value is larger than the pivot to after all the elements whose value is smaller than the pivot. Ideally, this pivot will be the median value, and after partitioning the array would be evenly divided between low and high values. It should also be noted that the algorithm uses an in-place partitioning scheme to save memory.

As the name indicates, this class is implemented with an array rather than a linked list or tree of any kind. This implementation has several advantages and disadvantages, and a future study may wish to consider the same algorithm with different underlying data structures. The invariant is realized through a basic assert and function.

Metrics:

Lines of code for implementation	13
Lines of code for invariants	22
Methods for implementation	1
Methods for invariants	2

Quantitative results:

Total number of classes in the project	1
Number of classes in the project that are testable	1
Overall coverage	0.94125
Tests run	1
Failures	0
Errors	0

Table 1 Coverage details of Array-Partition subject software

The above table means the following:

1 Total number of classes in the project is the classes that are presented to EvoSuite to let the test case generation be done. This figure does not mean that EvoSuite has generated that much number of tests.

2 Number of classes in the project that are testable tells about how many classes are actually executed by EvoSuite. Sometimes, it may happen that the class presented to EvoSuite to generate a test case is an empty or partial empty class for any particular reason. In that case, EvoSuite may or may not generate the test cases for it. So that particular class is eliminated from the consideration in this property.

3 Overall coverage: This is the most crucial term because it renders about the average of all the coverage. The overall coverage considers line coverage, branch coverage, mutation score (as per the behavior of class).

4 Tests run, describe the number of test cases run by EvoSuite

5 Failure describes how many test cases generated by the EvoSuite failed. Again, this is important as it gives the opportunity to investigate the reason why the test case is failed

6 The number of errors presented in the code is presented by the property *errors.*

Now, we can generate a coverage report to check the whether the coverage given by EvoSuite is true or not, we can use JaCoCo which is a code coverage library for java.

The following steps is taken to configure of JaCoCo with EvoSuite:

1 To set up Jacoco offline with instrumentation, we need to add the jacoco.agent dependency in </dependency> ; jacoco-Maven-plugin and Maven-surefire-plugin in the <build>. Therefore, Jacoco become able to operate on the test cases which are generated by the EvoSuite.

2 Important thing is we need to add the path for generated coverage report (by Jacoco) manually.

3 Now after saving the changes, the command *mvn clean test jacoco:restore-instrumented-classes jacoco:report* is fired at the location of the project in command prompt.

4 The report is generated on the jacoco-ut/intelement/ArrayPartition.java.html at eclipse IDE or on the given path in pom.xml file.

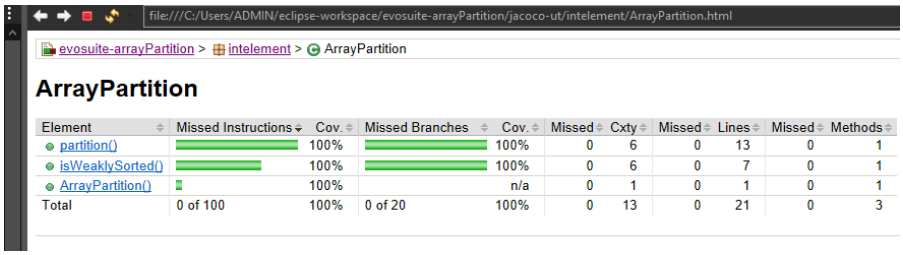


Fig. 1: coverage report of array-partition subject software in JaCoCo

Qualitative results:

In this section the why and how questions is discussed for the EvoSuite test case generation.

-Although the missed branches (0/20) and missed instructions (0/100) are null, the overall coverage is not 1.0000. It is 0.94125. That means that EvoSuite has not covered the entire source code. Perhaps, some branches or some lines are missed.

- All the test cases generated by the EvoSuites are run. None of the test cases failed. So, either the range of possible inputs can be increased or change in the oracle can be made to check the reliability of the code. It may also conclude that the code is good to apply further.

2 Binary Search Tree

Binary trees, or trees where any node may have no more than two children, have a special property in that the average depth for any node is O(√N). Binary search trees, however, have an average depth of O(log N). This makes them an important data structure as they allow rapid access to any node with little memory overhead.

Implementations of binary search trees are typically required to satisfy the following constraints:

- Each node must have a value (in this implementation values must be unique)
- A total order is defined on these values.
- The left subtree of a node contains only values less than the node's value.
- The right subtree of a node contains only values greater than or equal to the node's value.

Metrics:

Lines of code for implementation	130
Lines of code for invariants	28
Methods for implementation	23
Methods for invariants	4

Quantitative results:

Total number of classes in the project	5
Number of classes in the project that are testable	5
Overall coverage	0.6144999999999999
Tests run	binaryNode.java: 2 binarySearchTree.java: 18 range.java: 12
Failures	binaryNode.java: 0 binarySearchTree.java: 0 range.java: 0
Errors	binaryNode.java: 0 binarySearchTree.java: 0 range.java: 0

Table 2 Coverage details of Binary Search Tree subject software

3 Disjoint Set

A partition of some set X is a collection of non-empty subsets whose union is X, but any two of the subsets are disjoint. For example, let X = {1,2,3,4,5}. Then, {{1,3,5},{2,4}} is a partition of X.

There are two operations that must be supported by a disjoint set implementation:
-Find – This takes an element from the set X and returns the subset that contains the element. In the above example, Find(3) would return the subset {1,3,5}. This algorithm is also commonly referred to as a "union-find" algorithm.
-Union – This takes two different subsets and merges them into one subset.

Any disjoint-set implementation must satisfy the following properties:
-Tree nodes must be the elements of the set X.
-Every node (except the root) must have a pointer to the parent.
-Every tree forms a set, and the root of a tree is representative of the entire tree.

Metrics:
Original

Lines of code for implementation	35
Lines of code for invariants	39
Methods for implementation	5
Methods for invariants	3

Fast

Lines of code for implementation	38
Lines of code for invariants	39
Methods for implementation	4
Methods for invariants	3

Quantitative Results :

Total number of classes in the project	2
Number of classes in the project that are testable	2

Overall coverage	0.844375
Tests run	disjSetsFast.java: 17 disjSets.java: 13
Failures	0
Errors	0

Table 3 Coverage details of DisJoint Set subject software

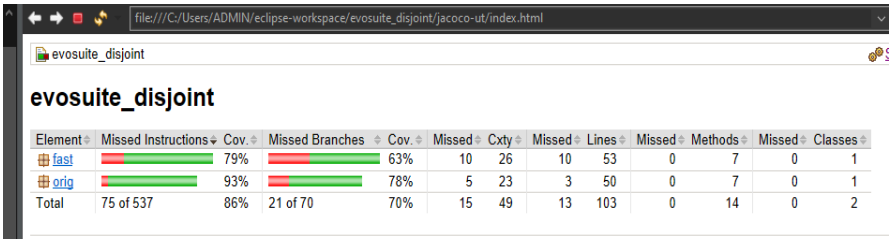


Fig. 3: coverage report of DisJoint Set subject software in JaCoCo

Qualitative results:

-Here, we can see that some part of fast and orig java files is not covers as 75 instructions are missed. In addition, out of 70, 21 branches are missed.

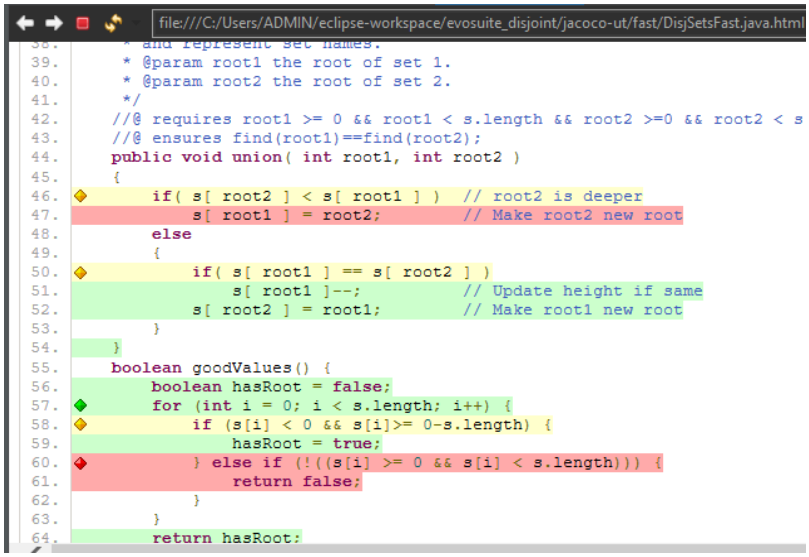


Fig. DisjSetsFast.java.html file generated by JaCoCo

(Note: Green: covered, yellow: partially covered, red: never executed [1])

Here, EvoSuite has missed the line 47.(Also, the yellow diamond (line 46) indicates that the branches are partially executed for if statement. So, the next line (47) may be missed by it)

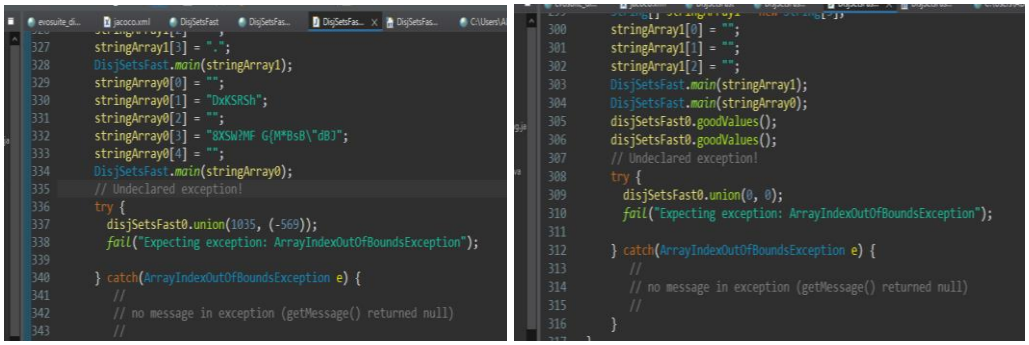


Fig. : test case from DisjSetsFast_ESTest.java

-Why EvoSuite missed that line?

For the union() method, the input values are : (0,0), (1035,-569). So, for both of the cases, the condition s(root2) > s(root1) does not become true. So, the line 47 is never get executed. This is the reason why EvoSuite does not cover that line & thereby the line coverage (and further overall coverage) decreases.

4 BinaryHeap

The included version of the binary heap is implemented as a min heap for integers. The implementation is straightforward and has only one custom exception (overflow). A new heap can be constructed with either a user-defined size or a preset default. Insertion consists of appending the new element to the bottom of the heap, and then percolating it up until the heap data structure is sound.

Only one field is used in conjunction with the invariants, and that is currentSize. Current size is an integer which stores the count of elements in the array.

Metrics:

Lines of code for implementation	72
Lines of code for invariants	20
Methods for implementation	11
Methods for invariants	2

Quantitative Results :

Total number of classes in the project	9
Number of classes in the project that are testable	2
Overall coverage	0.745625
Tests run	binaryHeap.java: 8 overflow.java: 1
Failures	0
Errors	0

Table 4 Coverage details of binaryHeap subject software

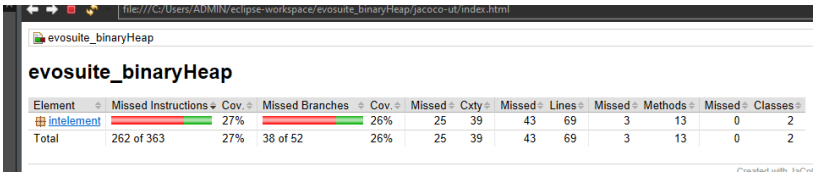


Fig. 4: coverage report of binaryHeap subject software in JaCoCo

Qualitative results:

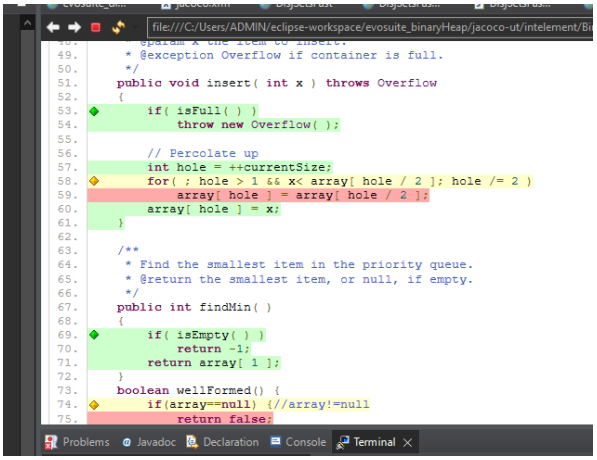
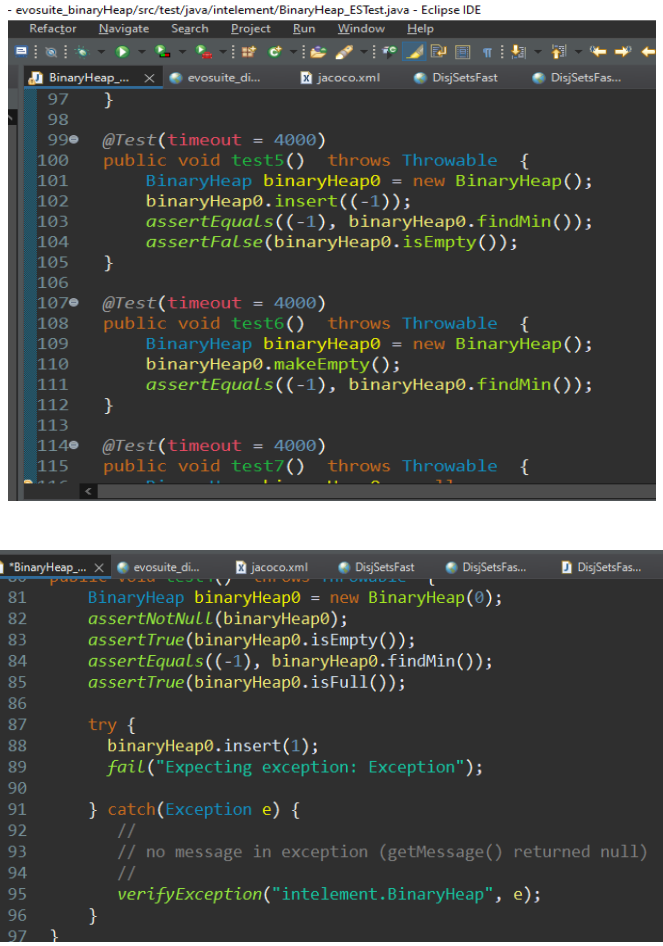


Fig. : BinaryHeap.java.html file generated by JaCoCo

-Why line 59 is not executed by EvoSuite?



Fig,: BinaryHeap_ESTest.java file showing test cases of insert()

Because, first of all, the line 58 is partially covered. Not all the Boolean conditions are covered by the test case inputs. In addition, the test cases' input 1 and -1 (from above two figures) does not satisfy the condition in line 58.

5 DoublyLinkedList

Normal (singly-linked) lists allow traversal of the list in only one direction, however the implementation discussed here allows for traversal in both a backwards and forwards fashion. Errors in lists can be subtle and unnoticed since pointers can get mishandled or assigned incorrectly. The impact of these errors can be significant, however, and require much debugging time and effort. The difficulty of detection and impact these errors can have justifies further analysis.

Implementations of doubly-linked lists are typically required to satisfy the following constraints:

-Any node A that is linked to by node B should also have a link back to B.

-A node should never have links whose target is unknown – nodes at the beginning and end of the list should point to sentinel nodes, null nodes, or each other.

Metrics:

Lines of code for implementation	277
Lines of code for invariants	95
Methods for implementation	32
Methods for invariants	10

Quantitative Results :

Total number of classes in the project	2
Number of classes in the project that are testable	2
Overall coverage	0.371875
Tests run	57
Failures	0
Errors	0

Table 5 Coverage details of doublyLinkedList subject software

6 Sorting

Sorting is a problem that is well suited to computers. Sorting steps can be laid out explicitly, and the task is highly repetitive. A significant number of sorting algorithms have been developed, and there are substantial differences between them. The majority of sorting algorithms are comparison based, which limits their fastest worst-case sorting time to $O(n \cdot \log(n))$ where n is the number of elements to be sorted.

Metrics

Lines of code for implementation	130
Lines of code for invariants	29
Methods for implementation	15
Methods for invariants	3

Quantitative Results :

Total number of classes in the project	1
Number of classes in the project that are testable	1
Overall coverage	0.93
Tests run	28
Failures	0
Errors	0

Table 6 Coverage details of sorting subject software

7 Stack

The stack is perhaps the most fundamental of all data structures used in computer science. It has been adapted to store everything from simple numbers to user-defined data types, to the entire state of running programs.

Implementations of stacks are typically required to satisfy the following constraints:
-You cannot pop elements from an empty stack.
-Random element access is not allowed – only the “top” element can be read or removed, and new elements can only go on top.

There are a number of fundamental operations that can be performed on any stack:
-Empty() – This will destroy the contents of the stack, leaving it empty.

-Push(Element) – This puts a new element on top of the stack.
-Pop() – This removes the top element from the stack and returns it to the user.
-Top() – This returns the top element to the user, but does not affect the stack.
-IsEmpty() – This returns true if the stack is empty, otherwise it returns false.

Metrics:

Lines of code for implementation	114
Lines of code for invariants	45
Methods for implementation	24
Methods for invariants	28

Quantitative Results :

Total number of classes in the project	4
Number of classes in the project that are testable	4
Overall coverage	0.78025
Tests run	stackAr.java: 16 overflow.java: 1 underflow.java: 1 listNode.java: 12 disjSets.java: 13
Failures	0
Errors	0

Table 7 Coverage details of stack subject software

8 Vector

Vectors are similar to arrays – one the most common and certainly one of the simplest data structures used in computer science. Arrays are hampered, however, by their fixed size. A vector data structure solves this problem, by increasing the array's size automatically when it becomes full and another insert is requested. Vectors are used frequently in both research and production environments. As they are versatile, fast, and space-efficient, they are a smart choice for a wide range of applications. Vectors are simple data structures, so correctness is not especially challenging, but it is extremely important.

Metrics

Lines of code for implementation	254
Lines of code for invariants	60
Methods for implementation	49
Methods for invariants	06

Quantitative Results :

Total number of classes in the project	1
Number of classes in the project that are testable	1
Overall coverage	0.67375
Tests run	53
Failures	0
Errors	31

Table 8 Coverage details of vector subject software

Qualitative results:

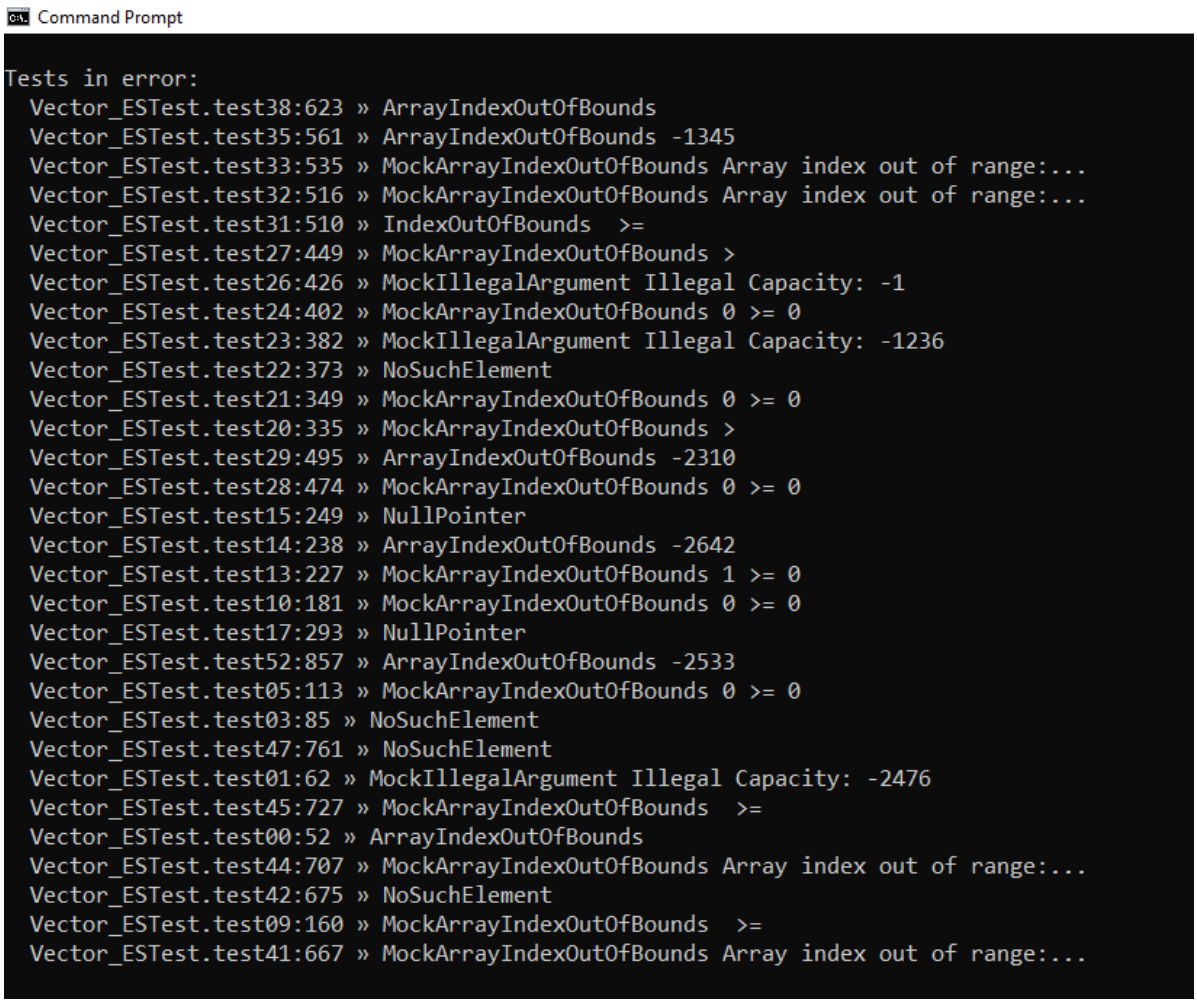


fig. : Errors produced in Vector_ESTest.java

The input values taken by the methods listed here produce the error.

Consider the 2nd error in the above fig and below snippet.

Test code:

```
@Test(timeout = 4000)
 public void test35()  throws Throwable  {
    Vector<Vector<Object>> vector0 = new Vector<Vector<Object>>();
    vector0.clone();
    Vector<Object> vector1 = vector0.get((-1345));
    // Undeclared exception!
    try {
      vector1.containsAll(vector0);
      fail("Expecting exception: ArrayIndexOutOfBoundsException");
```

```
    } catch(ArrayIndexOutOfBoundsException e) {
      //
      // -1345
      //
      verifyException("util.Vector", e);
    }
  }
```

Method:

```
public synchronized E get(int index) {
        if (index >= elementCount)
            throw new ArrayIndexOutOfBoundsException(index);


        return (E) elementData[index];
```

-It is very obvious that for input -1345 in the test code@35 the condition (index>= elementCount) is true thus, it produces an error.

-In a similar way, other error can be discussed.

Summary of the test results

It is found from the various results that absolute overall coverage is very difficult to achieve, almost impossible in case of multiple classes. Secondly, EvoSuite may or may not cover all the code while running the generated test cases. It depends upon the assertion of EvoSuite and its range. It is advisable to have wide range of test inputs. Again, this comes from the assertion of EvoSuite. We can increase the default 60 second time for the test generation in order to generate more test cases. Better the code coverage, better the reliability of the code.

4 LESSONS LEARNED AND OPEN ISSUES

Practical and technical difficulties

EvoSuite is an easy tool to use. However, the configuration of it sometimes creates difficulties.

The following difficulties are faced:

1) I tried to generate all the test cases from command prompt. I dd t. However, I was not able to run the generated test cases on command prompt. I was facing difficulties to tell the compiler to find the Maven dependency's location on my machine even though I specified the path in the environment variable. I tried to compile the code after entering the following command:

set CLASSPATH=target/classes:EvoSuite-standalone-runtime-1.0.6.jar;EvoSuite-tests;target/dependency/junit-4.12.jar;target/dependency/hamcrest-core-1.3.jar

I am getting an incorrect classpath error. This is still an open issue.

2) I am not able to generate coverage report of the subject software accept arrayPartition, binaryHeap and disjoint. I was able to build the subject source file for stack and doublyLinkedList by jacoco but the coverage report was not created. Therefore, I was unable to find the qualitative analysis for binarySearchTree, sorting, and stack subject softwares.
-doublyLinkedList
Error: *[77,8] duplicate class: DoubleLinkedList* was the error
-Sorting
Error: *incompatible types: java.lang.Comparable<java.lang.Object>[] cannot be converted to java.lang.Comparable<java.lang.Object>*
-binarySearchTree
Error: package edu.ksu.cis.projects.spex.java does not exist
These are also open errors.

3) I faced technical difficulty to configure the pom.xml file with EvoSuite to generate test cases. I forgot to add the EvoSuite-standalone-runtime-1.0.6.jar dependency. After adding that, I was able to generate test cases from Maven plugin.

4) If I get more time to use the Maven plugin, I would like to try it for other subject softwares using IntelliJ IDE.

5 REFERENCES

[1] To find colorcode used in JaCoCo, (which color represents what) https://www.jacoco.org/userdoc/annotations.html

[2] G. Fraser and A. Arcuri, “EvoSuite: Automatic test suite generation for object-oriented software.” in ACM Symposium on the Foundations of Software Engineering (FSE), 2011, pp. 416–419

[3] A. Panichella, F. M. Kifetew, and P. Tonella, “Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets,” IEEE Transactions on Software Engineering, vol. 44, no. 2, pp. 122–158, Feb 2018

[4] S. Vogl, S. Schweikl, G. Fraser, A. Arcuri, J. Campos, and A. Panichella, “EVOSUITE at the SBST 2021 Tool Competition,” in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, 2021, pp. 28-29.

[5] G. Fraser and A. Arcuri, “EvoSuite: automatic test suite generation for object-oriented software,” in Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, New York, NY, USA, 2011, pp. 416-419.

Appendix A Appendices

-Appendix are uploaded in .zip file along with the report.