2017

Genetic Algorithms as a Viable Method of Obtaining Branch Coverage

Jason Ross Frier

GENETIC ALGORITHMS AS A VIABLE METHOD OF OBTAINING BRANCH COVERAGE

by

Jason Frier

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

April, 2017

The thesis "Genetic Algorithms as a Viable Method of Obtaining Branch Coverage" submitted by Jason Frier in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

_____ _____
Robert Roggio
Thesis Advisor and Committee Chairperson

_____ _____
Sandeep Reddivari

_____ _____
Karthikeyan Umapathy

Accepted for the School of Computing:

_____ _____
Sherif Elfayoumy
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

_____ _____
Mark Tumeo
Dean of the College

Accepted for the University:

_____ _____
John Kantner
Dean of the Graduate School

ACKNOWLEDGEMENT

I wish to especially thank my thesis advisor, Dr. Robert Roggio, for his patience and dedication while assisting me with the development of my thesis.

CONTENTS

FIGURES

TABLES

ABSTRACT

Finding a way to automate the generation of test data is a crucial aspect of software testing. Testing comprises 50% of all software development costs [Korel90]. Finding a way to automate testing would greatly reduce cost and labor involved in the task of software testing. One of the ways to automate software testing is to automate the generation of test data inputs. For example, in statement coverage, creating test cases that will cover all of the conditions required when testing that program would be costly and time-consuming if undertaken manually. Therefore, a way must be found that allows the automation of creating test data inputs to satisfy all test requirements for a given test.

One such way of automating test data generation is the use of genetic algorithms. Genetic algorithms use the creation of generations of test inputs, and then choose the most fit test inputs, or those test inputs that are most likely to satisfy the test requirement, as the test inputs that will be passed to the next generation of inputs. In this way, the solution to the test requirement problem can be found in an evolutionary fashion. Current research suggests that comparison of genetic algorithms with random test input generation produces varied results. While results of these studies show promise for the future use of genetic algorithms as an answer to the issue of discovering test inputs that will satisfy branch coverage, what is needed is additional experimental research that will validate the performance of genetic algorithms in a test environment.

This thesis makes use of the EvoSuite plugin tool, which is a software plugin for the IntelliJ IDEA Integrated Development Environment that runs using a genetic algorithm as its main component. The EvoSuite tool is run against 22 Java classes, and the EvoSuite tool will automatically generate unit tests and will also execute those unit tests while simultaneously measuring branch coverage of the unit tests against the Java classes under test.

The results of this thesis' experimental research are that, just as the literature indicates, the EvoSuite tool performed with varied results. In particular, Fraser's study of the EvoSuite tool as an Eclipse plugin was accurate in depicting how the EvoSuite tool would come to perform as an IntelliJ plugin, namely that the EvoSuite tool would perform poorly for a large number of classes tested.

Chapter 1

INTRODUCTION

This thesis examines current research that attempts to find a suitable method for generating test data for software tests that covers a large percent, i.e close to 100 percent, of branches in the source code when executed in the testing environment. In other words, what is being studied is the concept of the coverage measure of branch coverage, and also a suitable option for generating test data that will achieve the fullest amount of coverage possible when executed in the testing environment.

The problem of achieving high levels (close to 100 percent) of successful branch coverage is not a new problem [Korel90]. From the beginning of software development developers have faced the problem of making sure that a significant portion of their code is executed when it is tested. From a developer's point of view, it does not make sense to have a program that passes all of its tests if the tests are not executing a majority of the source code. So arose the concept of branch coverage. One of the criteria of branch coverage is the idea that every branch within the source code should be executed. In other words, if there is an IF..ELSE statement, both the true and false branches should be executed in a testing environment to make sure that the code nested within those branches is not faulty and will not cause the program to behave incorrectly.

At this point, it is important to discuss why this research will focus on branch coverage as opposed to focusing on the other two traditional coverage measures, path and statement coverage. Path coverage deals with the execution of every possible path through the code. Statement coverage deals with typical imperative statements in the code and whether each one has been executed, such as the simple statements of assertion, a goto, return, and call, as well as the selection and iteration statements such as if statements, for loops, and while loops. It is important to note that branch coverage typically deals with whether IF statements have been executed, and so achieving branch coverage in an IF statement is in a way achieving a partial statement coverage. The reason that this thesis focuses on branch coverage solely instead of the other two coverage measures is because there is a significant amount of available research on the issue of achieving branch coverage using test input generators [Alshraideh11, Fraser11, Fraser13, Fraser14, Gupta00, Khor04, Korel90, Mahajan12, Michael01, Pargas99, Rohil08]. Perhaps the most important reason, however, for studying branch coverage at the expense of the other two main coverage measures is that it has been shown mathematically that the number of paths through a non-trivial program is infinite [Chang01]. Because of this, it is infeasible to study path coverage alone.

Along with this problem of achieving suitable branch coverage is the problem of generating test inputs that will execute all of the targets (branches) in a program's source code. Given this problem, there are several options for generating test inputs. One method is to do the generation of these test inputs manually. Another method for generating test inputs is to automate the process. However, if one chooses automation of

test input generation, there two primary choices: random test input generation (e.g. pathwise test data generators and data specification generators), and the use of tools driven by a genetic algorithm. Random test data generators and genetic algorithm tools are not necessarily the only two options available to developers in testing code, but they do represent two significant options as pertains to the available research.

The reason this is an issue is that it has been shown (and will be shown in this thesis) in contemporary research that random test data generators are not suitable for automatically generating test data to achieve significant branch coverage [Michael01]. Branch coverage can be defined in this context as the ability to generate a sufficient number of test inputs that when run through software being tested, will execute every source code branch in the program at least once. Branch coverage includes every true/false predicate that exists within the code.

Automating the generation of test inputs for software tests has been shown in the research to be more efficient and more successful at producing quality test inputs that will achieve branch coverage than manually generated test inputs [Fraser13]. So the issue becomes centered on what is the best method for automating the generation of test inputs. Research suggests that random test generators can achieve branch coverage for simple programs. Simple programs can be defined as those that contain an average of 30 lines of code and contain branches that are not complex, meaning, for example, they do not contain complicated nested conditional statements. The important thing to note, however, is that the specific number of lines of code is not as significant as the

complexity of the conditional statements contained in the code. Research also suggests, as programs grow more complex, random test input generators are less likely to be able to achieve high levels of branch coverage [Michael01].

According to Mahajan, C. Michael, Pargas, and Khor [Khor04, Mahajan12, Michael01, Pargas99], genetic algorithms are an answer to the problem of random test input generators. A genetic algorithm is an algorithm that is modeled after the theory of natural selection and evolution. They have increasingly been found within test data generation tools over the last twenty years. According to Fraser, Khor, and Mahajan [Fraser14, Khor04, Mahajan12] the main problem is that current research shows that genetic algorithm test input generators are inconsistent. Sometimes they perform remarkably well. Other times, they leave something to be desired. The current literature indicates genetic algorithm test input generators are pitted against randomized test input generators with the goal of proving that the genetic algorithm tools are superior. However, results indicate that the goals are often not realized. Chapter 2, Literature Search, addresses this issue in depth.

Again, it is important to emphasize why branch coverage is being isolated in regards to the problem of test input generation at the expense of the other two primary coverage measures, namely path and statement coverage. The amount of research literature available dealing with branch coverage and the problem of test input generation and automation is promising when it comes to the role that genetic algorithms play. Statement coverage is inherent in branch coverage, since branch coverage deals with IF

statements, which are a part of statement coverage. So providing branch coverage is in a way providing a measure of statement coverage, although not a complete 100% statement coverage measure. This presents the tester with a dilemma. For any non-trivial program, total path coverage will be impossible. However, achieving a high degree of branch coverage is in a way achieving a partial path coverage. So achieving a high degree of branch coverage can satisfy basic requirements for statement and path coverage, while also testing for the specifics of whether branch predicates will be covered. This makes branch coverage a suitable focus for the software tester.

Another question that can be asked is this: What is the best way to achieve branch coverage? Research suggests that using an automated tool for test input generation that is driven by a genetic algorithm can be an ideal solution to the problem of generating coverage measures, specifically branch coverage [Pargas99, Michael01, Khor04]. However, current research is varied as to the performance of genetic algorithms versus random test input generators. Herein lies the problem. What is needed is additional experimental research to validate the performance of a genetic algorithm in achieving branch coverage for Java classes. This paper attempts to address this problem by offering additional experimental research.

Chapter 2

LITERATURE REVIEW

2.1 Pargas' study

In Pargas' study, a test input generator named GenerateData was developed which contained a level of intelligence that allowed it, using the fitness function of the genetic algorithm, to determine which test cases came near to covering the target branch. The fitness function is an integral component of the genetic algorithm. Simply stated, it is a function that allows the genetic algorithm to determine how close it has come to achieving the desired goal. These test cases that came closest to covering the target branch were deemed more fit. The intelligence of GenerateData was generated from the control dependence graph of the given program. A control dependence graph utilizes the control-flow graph of a program to determine dependencies within the code. A visual representation of a program's control dependence graph can be seen below [Pargas99].

Here's an example below of a simple program (See Figure 1) and its control dependence graph (see Figure 2).

```
      Program Example
         integer i, j, k
1.       read i, j, k
2.       if (i < j)
3.          if (j < k)
4.             i = k;
            else
5.             k = i;
            endif
         endif
6.       print i, j, k
      end Example
```

Figure 1: Simple Program

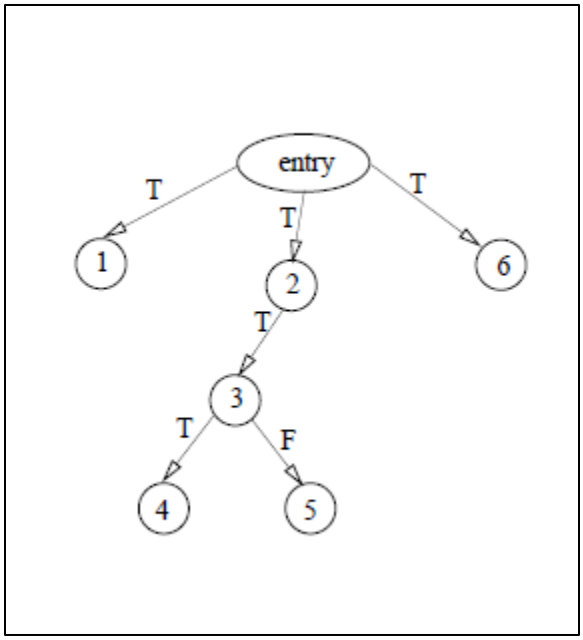

Figure 2: Dependence Graph

The GenerateData tool was more likely to use mutation than recombination to produce new generations of test inputs. Mutation is a process genetic algorithms use to maintain

diversity in the population of test inputs. Mutation occurs when one test input is altered from its initial state [Rohil08]. Recombination is a method genetic algorithms employ to create new generations of inputs, by taking elements of two parent test inputs and modifying them to create what are termed child test inputs. This can occur in the form of adding, deleting, or modifying constructors or methods in the test input [Rohil08]. The reason that mutation is used so frequently in GenerateData is so that the new populations of test inputs produced by GenerateData will be diverse. Mutation occurs in GenerateData by taking input variables of test cases and replacing them with values that are randomly generated. GenerateData functioned by using a nested while loop to iterate until either the target test requirement was satisfied or the loop timed out [Pargas99].

An implementation of GenerateData called TGen was tested against a random test input generator called Random. While Pargas tested TGen on six programs, branch coverage was only used as a metric for one program, Tritype c., which contained 61 lines of code. It took TGen an average of 132 iterations to achieve 100% branch coverage on the Tritype.c program, while it took the Random tool an average of 1100 iterations to achieve 100% branch coverage. For this program, the genetic algorithm tool TGen was deemed more efficient. (Tritype.c was a program in which triangle type had to be determined from the lengths of the triangle sides given. It is important to note that TGen was written in programming language C).

2.2 C. Michael's study

C. Michael discussed the GADGET tool. GADGET stands for Genetic Algorithm Data Generation Tool. GADGET, like Pargas' tool GenerateData, contains a level of intelligence within the algorithm. GADGET is able to identify the proximity of test inputs to achieving coverage for a given branch. All of this information is collected by the algorithm as it executes against the program under test. The GADGET tool keeps track of which conditions haven't been covered, and generates test inputs that will likely cover those target branches [Michael01].

One of the limitations of the GADGET tool was that it took a period of time from twenty minutes to several hours to execute the program b737 using a Sun Sparc-10 workstation. (Program b737 contained 75 conditional statements and 2,046 lines of code [Michael01]). This long execution time can be viewed as a limitation of the tool because it means that it will be more expensive to operate GADGET than it will be to generate test inputs by hand. However, because of the difficulty of creating test inputs by hand, GADGET is seen as a valuable tool, despite the cost incurred [Michael01].

GADGET was pitted against a random test input generator by testing the two on the following programs: binary search, bubble sort, number of days between two dates, Euclidean GCD, insertion sort, median computation, quadratic formula, Warshall's algorithm, and classification of a triangle. All of the programs tested contained approximately 30 lines of code and contained simple decisions with no complex decision

making such as commonly found in nested conditional statements. This made all of the programs under test roughly the same in terms of complexity. In every one of these programs tested, the GADGET tool performed superior to the random test input generator [Michael01].

In a second study, the GADGET tool was used on b737, which was a real-world C program with 69 decision points, 75 conditions, and 2,046 lines of code (excluding comments). This program was created using a CASE tool. The best runs of the GADGET tool reached a performance level of 93% branch coverage, while test generation by random means only achieved 55% coverage. The conditions contained in the b737 program fall into four categories. The first group contains conditions that were never covered by GADGET. The second group contains conditions that were covered by GADGET, but only while GADGET was trying to cover another condition, i.e. GADGET covered these conditions by luck. The third group contains conditions that were meant to be covered by GADGET. The fourth group contains conditions that were covered by the randomly seeded inputs that made up the initial population of inputs used by GADGET. The GADGET tool failed to cover Boolean variable conditions, meaning that when evaluated the condition could only have a value of true or false. The GADGET tool also failed to cover several nested conditions. This can be seen as another limitation of the GADGET tool, as GADGET had difficulty with nested conditions and Boolean variables [Michael01].

It is clear from C. Michael's study that the GADGET tool performed well on both simple programs and larger programs with more lines of code and more branches. However, the GADGET tool did have some limitations, namely that it didn't cover Boolean variable conditions or nested conditions well. One of the reasons why GADGET performed well is that it often covered one branch while it was actually trying to cover another, different branch. This is known as serendipitous coverage [Michael01].

Suppose you are trying to reach the True branch of the condition labeled c. The GA will only use inputs that can reach c to start with. Unfit test inputs won't reproduce, so only fit test inputs who can reach c will be produced. So each time a new input reaches c, it will take the FALSE branch (until the True branch can be satisfied). So therefore condition d is reached each time a new input tries to satisfy c, and d may have new branches that may be exercised. The result is that while the GA was trying to reach the True branch of c, it discovered new branches of condition d, and the test input may end up satisfying one of those branches even though it was looking for an input to satisfy the True branch of condition c all along (see Figure 3) [Michael01].

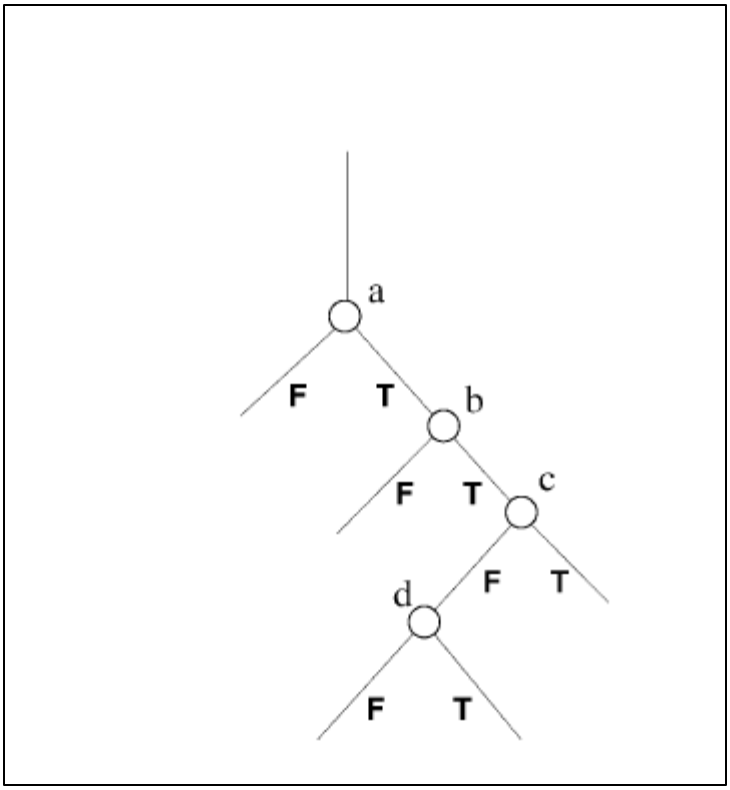

Figure 3: Branch Graph

2.3 Khor's Study

Khor proposed using an automated test input generator that implemented a genetic algorithm, and he called this tool Genet. Genet, like Pargas' and C. Michael's tools, contained a level of intelligence that allowed it to keep track of uncovered branches in the program under test. Genet contains a decision table that, as each test input executes, keeps track of all of the uncovered branches in the given program. If, at the end of the execution of all test inputs, there are still uncovered branches in the given program and the maximum number of generations has not been reached, the genetic algorithm driving the Genet tool proceeds to the development of Concepts (see Figure 4) [Khor04].

```
INPUT(X, Y, Z)
BEGIN
  IF (predicate1) { PRINT "1T"  // T2,T3,T4,T6
    IF (predicate2) PRINT "2T"  // T3, T6
    ELSE PRINT "2F" }           // T2,T4
  ELSE { PRINT "1F"             // T1,T5
    IF (predicate3) PRINT "3T"
    ELSE { PRINT "3F"           // T1,T5
      IF (predicate6) PRINT "6T"
      ELSE PRINT "6F"           // T1,T5
      RETURN } }
  IF (predicate4) { PRINT "4T"
    IF (predicate5) PRINT "5T"
    ELSE PRINT "5F" }
  ELSE PRINT "4F"               // T2,T3,T4,T6
END
```

Figure 1 Example program

Figure 4: Example of Concepts in the Program

Concepts are created by the genetic algorithm in Genet as a way to group together test inputs that are more likely to cover the target branches in the program under test. After Concepts are created, each chromosome is given a score. A chromosome is defined as a test input in Khor's study [Khor04]. This score is derived from how many times a chromosome can be found in the winning Concept. The winning Concept is the group of test inputs that are most likely to cover the target branch. Chromosomes that have the best score are therefore the chromosomes that are most likely to cover the target branch. These chromosomes are designated the most fit within the available gene pool. These chromosomes that are deemed the most fit are placed into a new pool. Mutation and recombination are applied to the test inputs in this new pool, and a new generation of test inputs are created from this pool of the most fit test inputs from the previous generation (see Tables 1 and 2) [Khor04].

C	Extent	Intent	Super C	Rank
0	–	–	1, 2, 3	0
1	T3, T6	2T	4	0
2	T1, T5	1F, 3F, 6F	5	2
3	T2, T4	2F	4	0
4	T2, T3, T4, T6	1T, 4F	5	1
5	T1,T2,T3,T4,T5,T6	–	–	0

Table 1: Concepts for Initial Generation

Tests	Branch
T11, T7	5F
T1, T5	6F
T9	6T
T1,T5, T9	3F
T10	3T
T1, T10, T5, T9	1F
T2, T4, T11	2F
T2, T3, T4, T6, T10	4F
T8	5T
T3, T6, T7, T8	2T
T7, T8, T11	4T
T2, T3, T4, T6, T7, T8, T11	1T

Table 2: Concepts for Final Populations

The performance of Genet was compared with Randy, a tool that randomly generated test inputs. They were pitted against each other in the testing of two different programs. The first program tested was called Clip, which contained 11 branches. Randy performed more efficiently because it required less computation. The second program tested was Tax. Tax contained 17 branches. Genet performed more efficiently against Tax because it covered nested predicates more efficiently and required fewer tests [Khor04]. Khor found that the complexity of the program under test was not a good indicator as to

whether Genet would perform better than the random test input generator. This research demonstrates the varied nature of results when it comes to the performance of genetic algorithms against random test input generation tools.

2.4 Fraser's Study

Fraser studied the issue of whether automatically generating test data will actually result in test cases that uncover more faults than with manual test generation in three different research papers [Fraser11, Fraser13, Fraser14]. The context for his experiments was in using the EvoSuite tool, which is an Eclipse plugin. The EvoSuite was used on Java classes seeded with faults, to address the following questions: How does an automated test generation tool impact branch coverage, and how does the automated test generation tool impact the tester's ability to find faults in the code?

To begin with, in Fraser's paper "*Does automated white-box test generation really help software testers?*" he examined how EvoSuite automatically produces JUnit test suites for a given Java class. EvoSuite supports branch coverage criteria, and it employs a genetic algorithm to generate candidate test suites using a fitness function [Fraser13]. It was determined that classes chosen to test should be non-trivial; in other words, they should contain no fewer than 100 lines of code. Classes also should not have I/O dependencies [Fraser13].

The subjects of the experiment were told to either generate their test cases manually in Eclipse or to use the EvoSuite plugin [Fraser13]. The results of the experiment shows that the EvoSuite tool accounted for a minimum branch coverage of 80%, compared to 35.71% branch coverage generated by manual testing [Fraser13]. However, in regards to finding faults, there was no scenario where the use of EvoSuite resulted in an increased ability to detect faults [Fraser13].

The problem here is that automating test generation using a genetic algorithm was shown to result in more effective coverage than a manual tester, however results did not indicate an ability to outperform manual testing when it came to finding faults in the code. Therefore, one must ask the question: Is the automation of test data really resulting in a better testing environment?

Fraser discussed in his paper "*EvoSuite: Automatic Test Suite Generation for Object-Oriented Software*", how EvoSuite generates whole suites of test inputs. This generation of whole test suites goes against the standard white-box testing procedure of generating test inputs for individual coverage goals. Whole test suite generation eliminates some of the faults that can be found with individual test input generation, namely that the search of the algorithm will not be negatively affected by the "order, difficulty, or infeasibility of the individual coverage goals" [Fraser11].

In practice, many coverage goals of a program under test are either infeasible, or some goals are more difficult to cover than other goals. In a situation such as these, testers may

be either lucky or unlucky when generating test inputs to cover those goals. For this reason, EvoSuite tries to eliminate “luck” by utilizing whole test suite generation [Fraser11].

Each EvoSuite test suite is made up of individual test cases. Crossover may be applied to these test cases by recombination and mutation. Recombination occurs when test cases are swapped based on a crossover position that is randomly chosen, which eliminates the usual difficulties experienced when crossover is done on method sequences. Method sequences can be defined at the natural progression of method calls within a program [Fraser11]. Mutation, which occurs when individual statements and parameters are added, deleted, or changed, can result in the addition of new test cases or the mutation of individual ones [Fraser11].

When determining the fitness of individual test cases, EvoSuite looks at each individual branch and determines a branch distance, which is the distance of how far each test case is from whether a branch will have a value of true or false. If a value of 55 will cause a branch to evaluate to true, and an input x==15, then the branch distance is |55-15| = 40 for the branch to evaluate to true. A fitness value of zero implies all branches in the Unit Under Test (UUT) are covered [Fraser11].

EvoSuite generates test suites for one class at a time, trying to maximize branch coverage for each class under test. EvoSuite has no restrictions when it comes to types of arrays, objects, or datatypes that it handles. EvoSuite also handles the string class in a unique

way, by allowing the EvoSuite search to evolve strings so that requirements on strings are satisfied. It does this by using its own helper methods in place of typical calls to string comparison methods [Fraser11].

EvoSuite detects faults in the programs that it tests by seeding mutants into the program (mutants can be defined here as artificial defects) [Fraser11]. If these mutants are detected by the test case (this happens if an assertion fails in the code) then these assertions are used by EvoSuite to develop a set of assertions that will likely detect all faults in the unit under test (UUT). However, if a mutant is not detected, then this means that a new test case must be created, and new tests run until the mutant is detected. The theory is that by developing a set of assertions that can detect all mutants seeded into the program, such a set of assertions may detect, potentially, any faults that may occur in the program under test [Fraser11].

In Fraser's paper "*A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite*", Fraser discusses that EvoSuite can only achieve optimal levels of branch coverage on particular classes. EvoSuite utilizes a built in security manager that will detect any negative dependencies on the environment. These include interactions the program under test may have with files, databases, and networks. The security manager will also detect dangerous code and will prohibit EvoSuite from achieving high coverage levels [Fraser14].

Fraser ran an experiment where he selected 100 random SourceForge Java projects, as well as the ten most popular Java programs on SourceForge. He called this test bed SF110. It contained 23,886 classes, 6.6 million lines of code, and required approx. 995 days to execute the code under test. Fraser found that the average branch coverage per project was 67%, where the project with the lowest average coverage amongst its classes was still 20% (See Figure 5).

Name	Grouping	Size	Min	Median	CI	Mean
All 110	Class	23105	0.00	0.94	[0.93, 0.95]	0.71
	Project	110	0.21	0.69	[0.66, 0.74]	0.67
Rand100	Class	11087	0.00	1.00	[1.00, 1.00]	0.78
	Project	100	0.21	0.69	[0.67, 0.75]	0.68
Top10	Class	12018	0.00	0.82	[0.80, 0.83]	0.64
	Project	10	0.32	0.55	[0.39, 0.61]	0.57
Industrial	Class	3970	0.00	1.00	[1.00, 1.00]	0.77
	Project	7	0.51	0.74	[0.62, 0.81]	0.75
CarFast	Class	1392	0.66	0.79	[0.79, 0.80]	0.81
	Project	11	0.76	0.91	[0.88, 1.02]	0.87

Figure 5: Fraser's Results for SF110

In Fraser's study, large groups of classes achieved dismal coverage (10% or less) and large groups of simpler classes achieved high coverage (90% or greater). Classes within the lowest coverage intervals (10%, 20%, and 30%) tended to contain large numbers of branches (more than 70). Most classes fell within either the 10% or fewer coverage interval, or the 90% or greater coverage interval [Fraser14].

Fraser was troubled that so many of his classes under test achieved such low coverage scores. One hypothesized answer that he gave as to why this may occur is that EvoSuite has a built in security manager that will not allow dangerous code to execute. The security manager also will not let programs properly test and execute if there are environmental dependencies (network, files, databases) that interact with the program under test in a dangerous manner. To support this hypothesis, Fraser found that for his experimental research, when classes raised no exceptions, EvoSuite performed well, achieving an average branch coverage of 84%. For classes that did raise exceptions, and where some level of permission was needed, EvoSuite performed much poorer. Fraser also hypothesized that since EvoSuite is not built to handle multithreaded code, in instances where the program code spawns threads, EvoSuite will not perform well [Fraser14].

2.5 Mahajan's Study

Mahajan's study focused on comparing the performance of a genetic algorithm-driven test generation tool with the performance of a random test data generation tool on classic problems such as the quadratic equation roots classification problem, triangle classification problem, the date difference problem etc. Mahajan ran his genetic algorithm tool on the problems listed above to find out the number of generations that were required to achieve 100% coverage. Study findings were varied. For some programs run with the two tools, the GA tool achieved a greater percentage of coverage in fewer generations. For other programs, the results of the two tools were similar. Yet for other programs, the random test generator performed more efficiently, with greater

coverage percentage than the GA tool (see Table 3). So it is difficult to draw strong conclusions from Mahajan's study [Mahajan12].

Prog. No.	No. of Vars	Pop Size	No. Of Gen		Def-Use Coverage %	
			GA	Random	GA	Random
1	3	8	5	8	100	100
2	4	8	8	7	100	95
3	4	7	10	13	87	80
4	3	6	6	9	100	100
5	3	9	13	17	100	100
6	5	8	7	8	100	91
7	4	8	6	6	100	85
8	4	10	11	11	94	80
9	3	10	6	7	100	95
10	5	6	11	10	100	100
11	5	5	13	13	83	95
12	4	5	12	16	100	100
13	3	9	16	17	91	85

Table 3: Mahajan's Results

2.6 Rohil's study

Rohil's study examined the performance of a genetic algorithm based software tool called gp against a random tool used to generate test cases during the generation of object oriented test cases. The classes that were tested in the experiment included NodeList, NodeIterator, ParserUtils, IteratorImpl, SimpleNodeIterator, and others that were taken from an open source project called HTMLParser. The results of the percentage of coverage achieved by the two approaches are captured in Figure 6 [Rohil08].

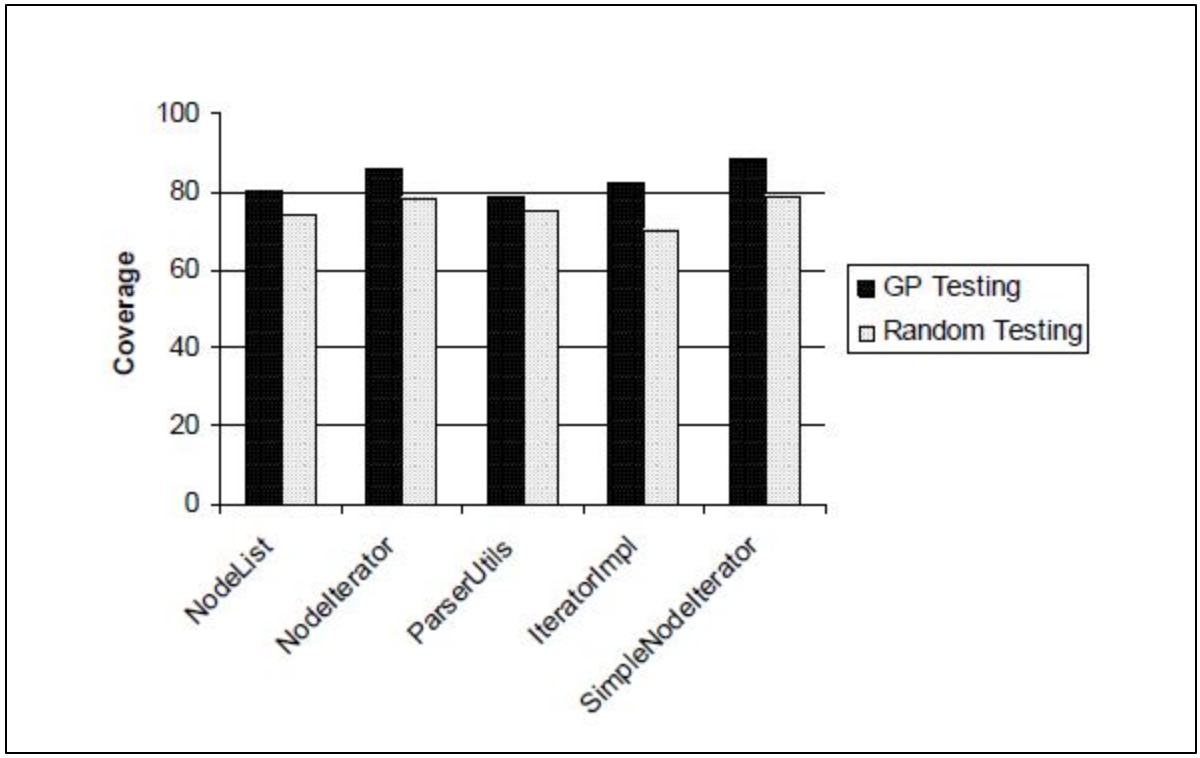

Figure 6: Rohil's Results

Rohil's study, unfortunately, did not contain a description of the classes that were tested in terms of lines of code and complexity (number of branches); as a result, it is difficult to make solid conclusions when it comes to whether the genetic algorithm tool gp that he tested performed well on smaller, less complex classes or if it performed well on larger, more complex classes. However, Rohil's study is relevant to this thesis research because it illustrates variation in the performance of his gp tool when compared against the performance of the random tool. By looking at Figure 6, it is clear that the performance of the two tools was almost the same for the ParserUtils class; however, there was a significant gap in the performance on the SimpleNodeIterator class. For this class the gp tool performed noticeably better than the random tool. This, in effect, is the reason that Rohil's study is relevant to this thesis research, because it does show variability in the

performance of the genetic algorithm tool, and the prediction of variability in the performance of genetic algorithm tools is the foundation of this thesis.

2.7 Branch Coverage

Branch coverage is a concept that falls under the white-box testing strategy known as code coverage. Branch coverage is the idea that testers strive to execute most of a program's branches at least once during testing. This includes TRUE/FALSE decisions for IF…ELSE statements, Loops, etc. For example, consider the following code block:

```
IF (X>Y){
  System.out.println ("Hello!")
}ELSE {
  System.out.println("Goodbye!")
};
```

In this case, a test case (5,4), where X=5 and Y=4, would execute the first part of the IF…ELSE statement, and the result would be that the word "Hello!" would be printed. However, this also means that only 50% of the IF…ELSE statement would be executed, leaving the other part of the branch untested. This could potentially be dangerous for the state of the program, given that the ELSE branch of the statement could contain something that when executed, could cause the program to abnormally terminate. So it is important to ensure that when running tests, that all of the branches are executed at least once. So in this case, two cases would be needed to run the program correctly: (5,4) and (4,6) [Software15].

Chapter 3

METHODOLOGY

Khor, C. Michael, Fraser, and Mahajan all present studies which show that genetic algorithms can perform as well as, and in some cases better than, random test case generators when it comes to code coverage, specifically branch coverage [Fraser11, Fraser13, Fraser14, Khor04, Mahajan12, Michael01]. Most of the research has been focused on comparing genetic algorithms to randomized test input generators for testing branch coverage. Further, most of the available research is undertaken from the perspective of showing that genetic algorithms are superior to randomized test data generation [Khor04, Mahajan12, Michael01, Pargas99]. But in attempting to unify the results of all this research into a single "state of the art" assertion, there is uncertainty.

Showing the superiority of genetic algorithms to randomized test input generation was the goal of Khor's research. The aim of Khor's research was to use Genet, an automated test input generator, to develop test cases that were more likely to cover target branches within the program under test. Basically, Genet identified target branches within the code and then set aside test cases that were more likely to cover these branches, and developed new populations of test cases from these original test cases that were likely to cover the target branch. It used a genetic algorithm to do this.

However, in Khor's research, the results were varied. In one trial run, the genetic algorithm tool performed less efficiently than the randomized test case generator, and in another test run the genetic algorithm tool performed more efficiently. The goal of Mahajan's study was similar. A genetic algorithm test case generator was implemented to demonstrate that genetic algorithms could provide a greater degree of branch coverage than random test case generators. However, Mahajan's study backs up Khor's results by showing that in certain test runs, the random test case generator performed equally as well as, or in one case more ideal than, the genetic algorithm tool in regards to percentage of coverage achieved and total number of generations required to achieve the results.

3.1 Background on the Experimental Environment and Unit Testing

EvoSuite is a software tool that can be utilized as a plugin for the IntelliJ IDEA Integrated Development Environment. EvoSuite utilizes a genetic algorithm, internally, to produce test suites to achieve branch coverage [Fraser13]. EvoSuite automates the production of JUnit test suites for a specified Java class. JUnit is a unit testing framework for Java that allows the user to write repeatable unit tests. EvoSuite can be implemented in the IntelliJ IDEA Integrated Development Environment (IDE) as a plug-in. To begin with, an integrated development environment is a software application that allows computer programmers to write, build, and debug source code. An IDE usually contains a source code editor, debugger, and build automation tools. A plugin is additional functionality, such as a software tool, that can be installed into an IDE that will function seamless as an integrated part of the IDE, but that does not initially come with

the software. In the IntelliJ IDE, there is a plug-in repository within the IDE that allows users to search for plugins and then install them (See Figure 7).

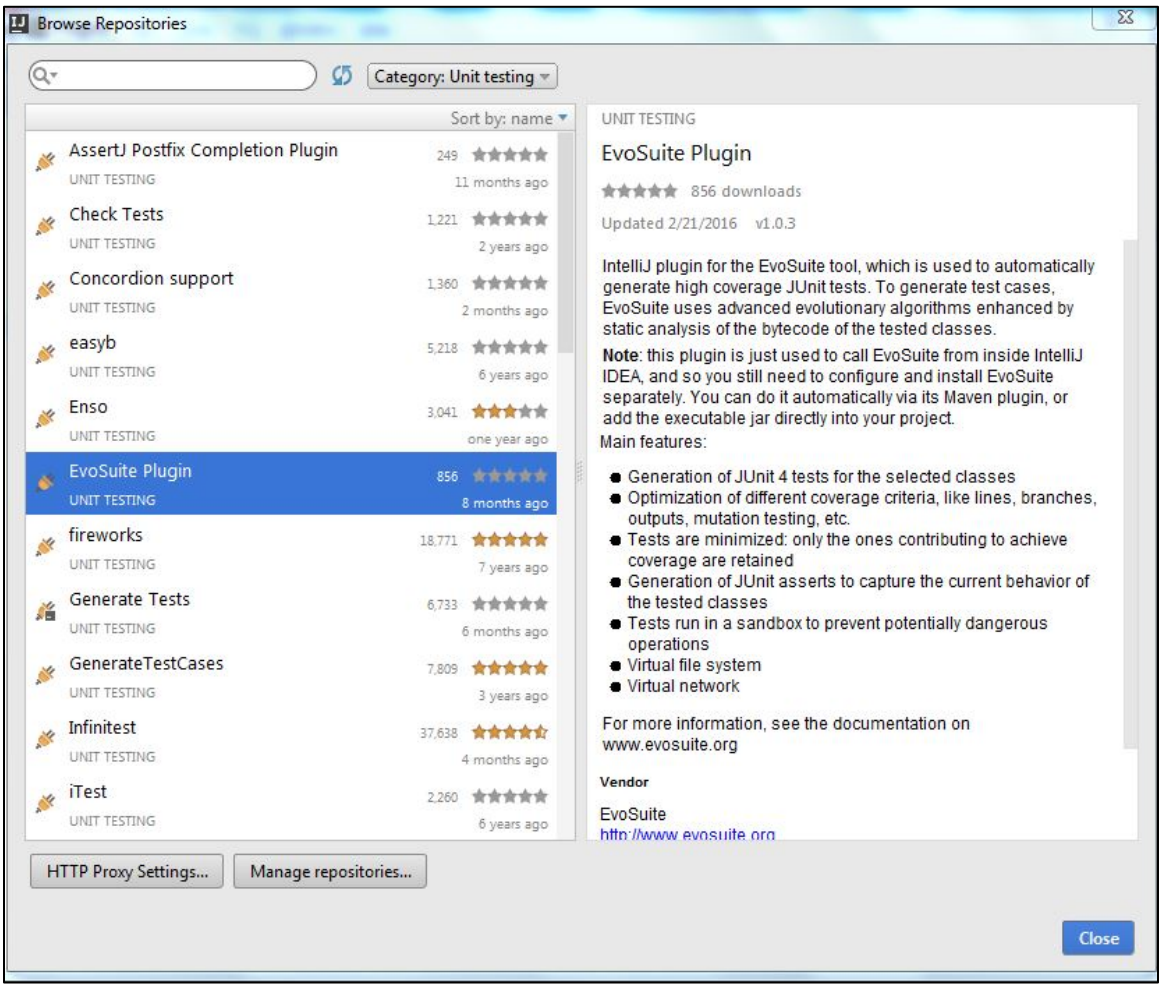

Figure 7: EvoSuite Plugin Repository

Figure 7 shows the IntelliJ plugin repository. It is from this screen that users can select different plugins to install. Clearly, the EvoSuite plugin has been selected from the menu.

Once the EvoSuite plugin has been installed, EvoSuite can be easily run against a Java class by right clicking on the SRC icon in the lefthand tool bar, and clicking “Run EvoSuite” (See Figure 8).

Figure 8: EvoSuite Running

Figure 8 depicts the view that users have at the bottom of the IntelliJ IDE main screen when EvoSuite is run. This display pane will also inform users when EvoSuite has completed running.

EvoSuite will automatically generate unit tests for that particular Java class in the IntelliJ IDEA IDE, and will subsequently run those unit tests and measure the percentage of branch coverage achieved by the EvoSuite plug-in [Fraser13].

IntelliJ IDEA was developed by JetBrains. JetBrains is an international software development company based in Prague, Czech Republic. They offer many IDEs not only for Java but also for Ruby, PHP, Python, etc. One of the key features of IntelliJ IDEA is to allow users to install plug-ins via its plug-in repository. It also supports the use of JUnit (See Figure 9).

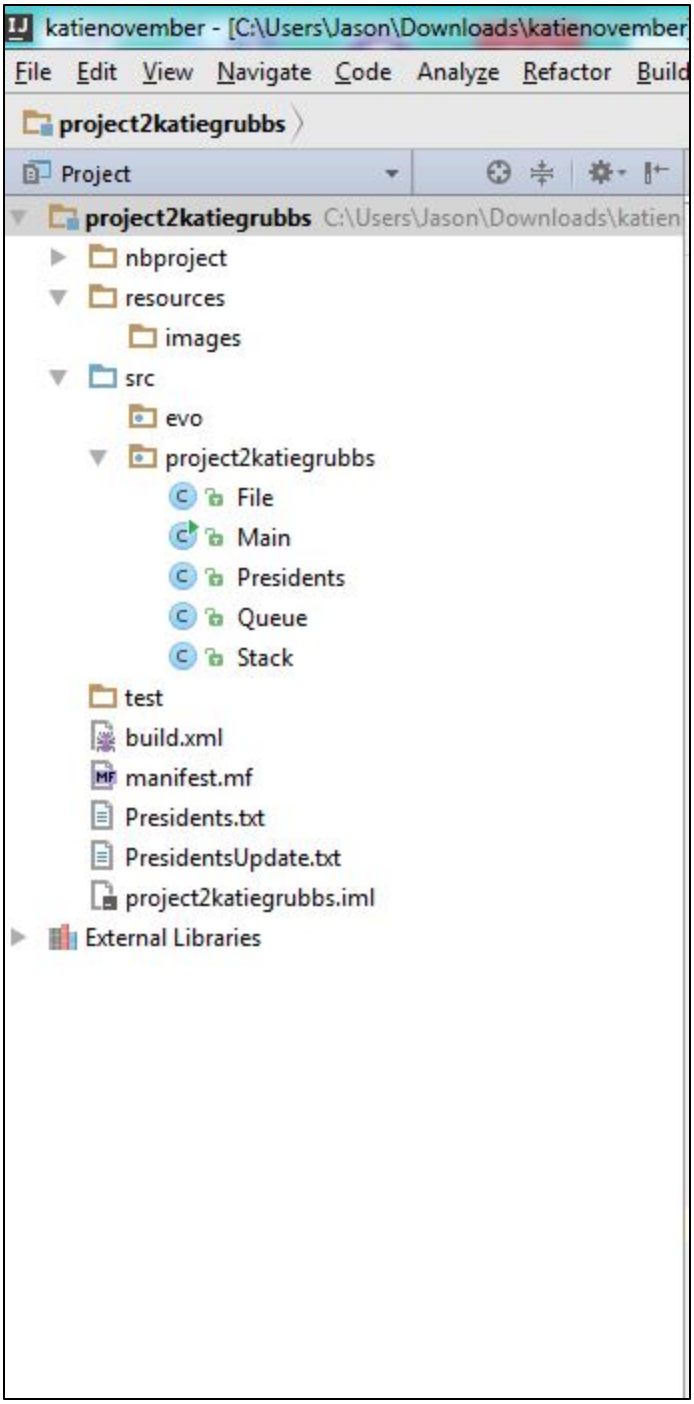

Figure 9: IntelliJ IDE Main Screen

Figure 9 depicts the main IntelliJ IDE workbench screen. What is seen in Figure 9 is the left hand side of the workbench, which is a pane which allows users to right click on the "src" folder. When this is done, a drop down menu will be displayed that will allow the user to select "Run EvoSuite" for that particular group of classes.

3.2 What is a Unit Test

When testing usually the smallest part of an application for which a program developer is responsible, the process is termed unit testing. A unit can be defined as the smallest portion of the program that can be tested. A unit test is a test performed on a unit to see if it is functioning as desired. Unit testing is often a precursor to measuring branch coverage. Branch coverage is usually measured as a metric against the execution of a unit test, i.e., when unit tests are run, what percentage of the branches in the unit were executed during the unit test.

3.3 Experimental Design

Fraser's studies, as elaborated on within this paper's literature review, presented much useful information on the EvoSuite tool. EvoSuite is used within this thesis' experimental research. Twenty-two Java classes will be selected from several Java programs of varying size (in terms of LOC), complexity (in terms of number of branches) and purpose. These classes come from programs selected from www.planet-source-code.com, SourceForge, and a data structures class from the University of North Florida. The main criteria used when looking for the target classes is size (classes should be non-

trivial, which according to Fraser is more than 100 lines of code); complex (Fraser defines more than 70 branches as being complex); and should come from a variety of different programs in terms of function. The classes selected for this study come from a Java program dealing with parsing data from input files, creating objects, and inserting those objects into queues; a Java Monopoly board game; a Java program for making music beats; a Java program for a virtual notice board; a Java program used to create menus; and a Java Checkers game.

The goal of this research is to test EvoSuite on a variety of classes, report the results in terms of tables and graphs, and compare the performance of the EvoSuite tool as used in the IntelliJ IDEA with the performance of Fraser's EvoSuite Eclipse plugin. While the scope of Fraser's research is beyond the scope of this thesis, this author is confident that it will be possible to draw parallels between the performance of Fraser's Eclipse plugin and the performance of the IntelliJ plugin used in this thesis.

The main contribution of this thesis research is to test out the EvoSuite tool as an IntelliJ IDEA plugin. Fraser's paper concentrated on the use of EvoSuite as a plugin for Eclipse, but nothing has been written about the use the IntelliJ plugin. Also, another contribution of this thesis is the observance of the performance of EvoSuite when the IntelliJ plugin paramenters are adjusted. In IntelliJ, the EvoSuite plugin will be tested with three sets of parameters: the default parameters, in which there is one core, the memory per core is 2,000 MB, and the time per class is three minutes; a parameter setting where two cores are used, there is 5,000 MB per core in memory, and the time per class is five minutes;

and finally, the parameter setting where four cores is used, there is 10,000 MB memory per core, and the time per class is seven minutes. The performance of EvoSuite on the twenty-two selected classes with these three parameter settings will be observed, results will be documented, and conclusions will be drawn in relation to the performance of Fraser's EvoSuite research.

3.4 How does a Genetic Algorithm Generate Unit Test Cases

The Genetic Algorithm used in EvoSuite must be able to generate test programs, in this case unit test cases to be run and measured for code coverage. Rohil's study provides some insight into how this works. The EvoSuite tool is built with a Genetic Algorithm. Genetic Algorithms are influence by the broader concept of Genetic Programming, which itself is built around the concept of hierarchically organized trees [Rohil08]. This requires the use of specialized genetic operators for crossover and mutation. (See Glossary for definitions) The trees used in this type of programming, which is the foundation of the EvoSuite tool, require that such trees must contain data that tells the tool not only the methods (including target and parameter objects) that the tool must call but also the order that those methods should be called in [Rohil08].

When chromosomes (individual test inputs generated by the genetic algorithm) are being constructed, genes are utilized to encode each component of each statement in the code [Rohil08]. The way this works is that each gene is assigned a corresponding number. When these genes are decoded, the integer values of the genes are used to identify the

methods and objects used for the method call [Rohil08]. All of the genes constitute a chromosome, which in genetic programming will make up the tree structure as the different nodes of the tree.

For example, consider the following code from Rohil's paper. The following is a test cluster of two classes: the Host class and the Connection class. Figure 10 shows a tree representation of the following code snippet [Rohil08]:

```
class Host
{
  public Host(Connection con);
  public void connect();
  public void configure(Connection con);
  public Connection getConnection();
  public boolean testPort(Connection con);
  public void disconnect();
}
```

The following is the code for the class Connection:

```
class Connection
{
  public Connection (int port);
  public int getPort();
}
```

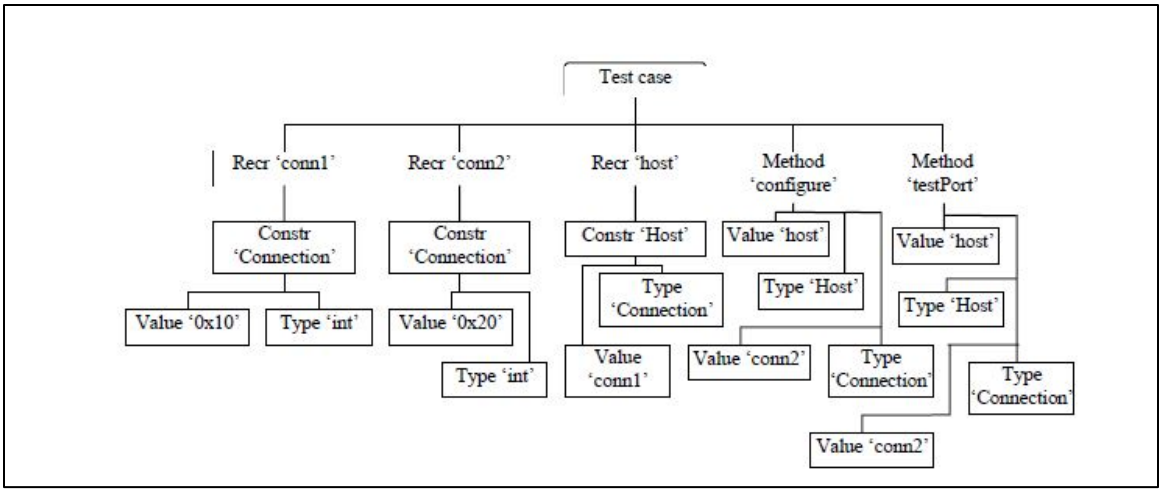

Figure 10: Genetic Programming Tree Representation of a Chromosome

The following is an illustration of a test input that is produced from the Genetic Programming representation of the chromosome [Rohil08]:

```
Connection conn1=new Connection(0x10);
Connection conn2=new Connection(0x20);
Host host=new Host(conn1);
host.configure(conn2);
host.testPort(conn2);
```

The explanation for the above tree representation of a chromosome and the corresponding code for the test case is that in genetic programming, methods and constructors in a class constitute genes in a chromosome. A chromosome can be made up of many genes, that is, many methods and constructors. This is how a chromosome is encoded from a programming language. Methods and constructors can take in variables as arguments in the method or constructor. In the above code, the conn1 object takes in 0X10 as the argument for the Connection constructor, and so forth. The same goes for the methods

host.configure, which takes in conn2 as an argument. These methods and constructors make up the genes in the chromosome being encoded [Rohil08].

For mutation, when a genetic value is altered from its original state, the gene pool will be updated with new gene values. It is because of these new gene values being entered into the gene pool that the genetic algorithm is able to achieve better solutions than were possible previously [Rohil08]. For each component of the test case statement, three mutation modifiers are defined. These three mutation modifiers work on the basic building block of the chromosome. They are as follows: create a new building block, modify an existing building block, or delete a building block. For example, these three modifiers can be applied to a constructor to create, modify, or delete a constructor in the Java class, or they can be applied to methods to add, modify, or delete methods. This is how new generations of test case inputs are created.

Chapter 4

RESULTS

EvoSuite was used as an IntelliJ plugin and run against twenty-two Java classes from a variety of programs. These twenty-two classes contained varying levels of complexity (number of branches) and size (LOC). The twenty-two classes were tested under three different parameter settings within the EvoSuite IntelliJ plugin: the first parameter setting was the default setting, in which one core was used, 2,000 MB per core were used, and three minutes per class; the second parameter setting, in which two cores, 5,000 MB of memory per core, and five minutes per class was used; and the third parameter setting, in which four cores, 10,000 MB of memory per core, and seven minutes per class was used. This is one of the main differences in the research presented within this paper and the research presented in Fraser's study, namely that Fraser used EvoSuite as an Eclipse plugin. There is nothing in the available research on EvoSuite that offers any suggestions as to how EvoSuite will perform in the IntelliJ IDEA Integrated Development Environment. Fraser's research mentioned nothing about the parameters found in the IntelliJ EvoSuite plugin, which is another reason that this thesis paper's research is a unique contribution.

The results of this thesis experiment show the variation in the performance of the EvoSuite tool, which supports the thesis of this paper, namely that the performance of genetic algorithm tools can vary in a test environment. For more than half of the classes

tested with the EvoSuite tool, EvoSuite achieved a branch coverage percentage of less than 10%, which is in line with the findings of Fraser in his research. Adjusting the parameters of the EvoSuite tool as an IntelliJ plugin resulted in, for the most part, an increase in the number of seconds required for EvoSuite's search to complete. In other words, as the number of cores, MB per core, and minutes per class increased, then the number of seconds required to complete EvoSuite's search increased as well, although this is not a definitive statement. The same cannot be said for the number of generations completed across the three test runs. As the parameters increased, the number of generations did not necessarily increase at the same proportion. EvoSuite achieved the best branch coverage on the Queue and Property classes. The Property class contained 71 branches and the Queue class contained 93 branches. This level of complexity falls in the middle, in terms of levels of complexity of the classes tested. The class with the highest number of branches contained 227 branches, while the smallest class, in terms of complexity, contained 15 branches. EvoSuite performed very poorly on the two most complex classes, achieving just 0% and 3% coverage on the two classes with 227 branches and 219 branches, respectively. There were several classes that contained less than 50 branches and which EvoSuite performed poorly, achieving less than 10% coverage. As a result, it is clear that there is variability in the performance of EvoSuite as a genetic algorithm tool, and therefore the thesis of this paper is supported; however, the experimental research performed in this thesis also validates the findings of Fraser, who found that EvoSuite performed poorly on a large number of classes tested in his research.

4.1 Test run 1:

Table 4 shows the results for test run 1. Test run 1 was performed with one core, 2,000 MB per core, and three minutes allotted for each class. In test run 1, EvoSuite performed poorly on the most complex classes, while it performed best on classes with middle-of-the-pack complexity. EvoSuite also performed poorly on classes with complexity at the lower end of the spectrum.

Name of Class	Seconds needed for search to complete	Generations created	Lines of Code	Number of Branches	Branches Covered	Coverage percentage
Gameboard	685	15720	432	227	1	0
ImageMover	140	1144	56	32	15	47
Movement	103	904	51	25	7	28
AssetMaintDialog	284	4729	214	90	1	1
Player	703	689	292	170	105	62
TradeDialog	104	1194	109	32	1	3
OwnedControl	103	1236	96	38	1	3
Property	232	912	96	71	63	89
PropertyInfoDialog	241	4205	139	69	1	1
Casa	41	391	36	15	9	60
Damas	32	645	93	18	2	11
Tabuleiro	211	24	230	216	90	42
Beatbox	261	1565	290	77	16	21
Menubuilder	154	1634	238	219	6	3
Queue	289	1196	110	93	87	94
Edit_Chief_Acct	111	659	183	35	2	6
Edit_Faculty_Acct	112	535	183	35	2	6
Edit_Offic_Acct	111	751	183	35	2	6
Notice_Board	164	3511	207	46	2	4
Update_Chief_Not	140	1013	257	45	2	4
Update_Facul_Not	140	980	257	45	2	4
Update_Offic_Not	140	920	257	45	2	4

Table 4: Test Run 1

4.2 Test run 2:

Table 5 shows the results for test run 2. Test run 2 was performed with two cores, 5,000 MB per core, and five minutes allotted for each class. In test run 2, the number of seconds required for EvoSuite to complete its search increased from test run 1, as did the number of generations created by EvoSuite, in comparison to test run 1. In test run 2, EvoSuite

performed poorly on the most complex classes, while it performed best on classes with middle-of-the-pack complexity. EvoSuite also performed poorly on classes with complexity at the lower end of the spectrum.

Name of Class	Seconds needed for search to complete	Generations created	Lines of Code	Number of Branches	Branches Covered	Coverage percentage
Gameboard	1311	19860	432	227	1	0
ImageMover	241	362	56	32	15	47
Movement	176	491	51	25	7	28
AssetMaintDialog	699	13121	214	90	1	1
Player	1340	2169	292	170	112	66
TradeDialog	175	950	109	32	1	3
OwnedControl	175	1196	96	38	1	3
Property	488	601	96	71	67	94
PropertyInfoDialog	515	6028	139	69	1	1
Casa	132	893	36	15	9	60
Damas	115	3001	93	18	2	11
Tabuleiro	253	18	230	216	92	43
Beatbox	307	2188	290	77	16	21
Menubuilder	157	841	238	219	6	3
Queue	96	307	110	93	86	92
Edit_Chief_Acct	190	767	183	35	2	6
Edit_Faculty_Acct	191	1000	183	35	2	6
Edit_Offic_Acct	190	789	183	35	2	6
Notice_Board	296	3797	207	46	2	4
Update_Chief_Not	250	1283	257	45	2	4
Update_Facul_Not	250	1438	257	45	2	4
Update_Offic_Not	250	1313	257	45	2	4

Table 5: Test Run 2

4.3. Test Run 3

Table 6 shows the results for test run 3. Test run 3 was performed with four cores, 10,000 MB per core, and seven minutes allotted for each class. In test run 3, the seconds needed for EvoSuite to complete its search increased, for the most part. However, in general, the number of generations created decreased from test run 2. In test run 3, EvoSuite performed poorly on the most complex classes, while it performed best on classes with middle-of-the-pack complexity. EvoSuite also performed poorly on classes with complexity at the lower end of the spectrum.

Name of Class	Seconds needed for search to complete	Generations created	Lines of Code	Number of Branches	Branches Covered	Coverage percentage
Gameboard	2316	7426	432	227	1	0
ImageMover	375	1668	56	32	13	41
Movement	38	0	51	25	5	20
AssetMaintDialog	856	3152	214	90	1	1
Player	2360	447	292	170	104	61
TradeDialog	261	2550	109	32	1	3
OwnedControl	265	2114	96	38	1	3
Property	924	1759	96	71	67	94
PropertyInfoDialog	995	9811	139	69	1	1
Casa	186	1097	36	15	9	60
Damas	187	2631	93	18	2	11
Tabuleiro	191	7	230	216	66	31
Beatbox	61	179	290	77	13	17
Menubuilder	66	602	238	219	6	3
Queue	66	192	110	93	86	92
Edit_Chief_Acct	297	923	183	35	2	6
Edit_Faculty_Acct	313	489	183	35	1	3
Edit_Offic_Acct	292	148	183	35	2	6
Notice_Board	502	3214	207	46	2	4
Update_Chief_Not	374	482	257	45	2	4
Update_Facul_Not	369	47	257	45	2	4
Update_Offic_Not	369	298	257	45	2	4

Table 6: Test Run 3

4.4 Test run 1 branch coverage

Figure 11 shows the branch coverage for test run 1.

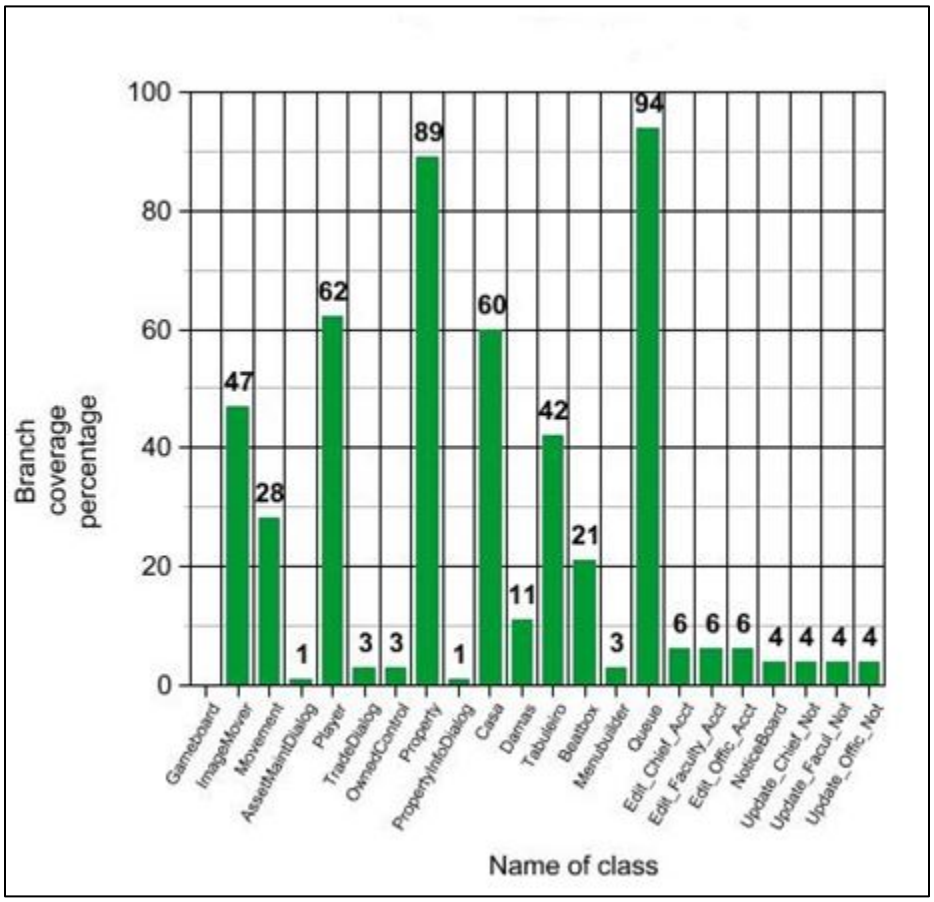

Figure 11: Test Run 1

4.5 Test run 2 branch coverage

Figure 12 shows the branch coverage for test run 2.

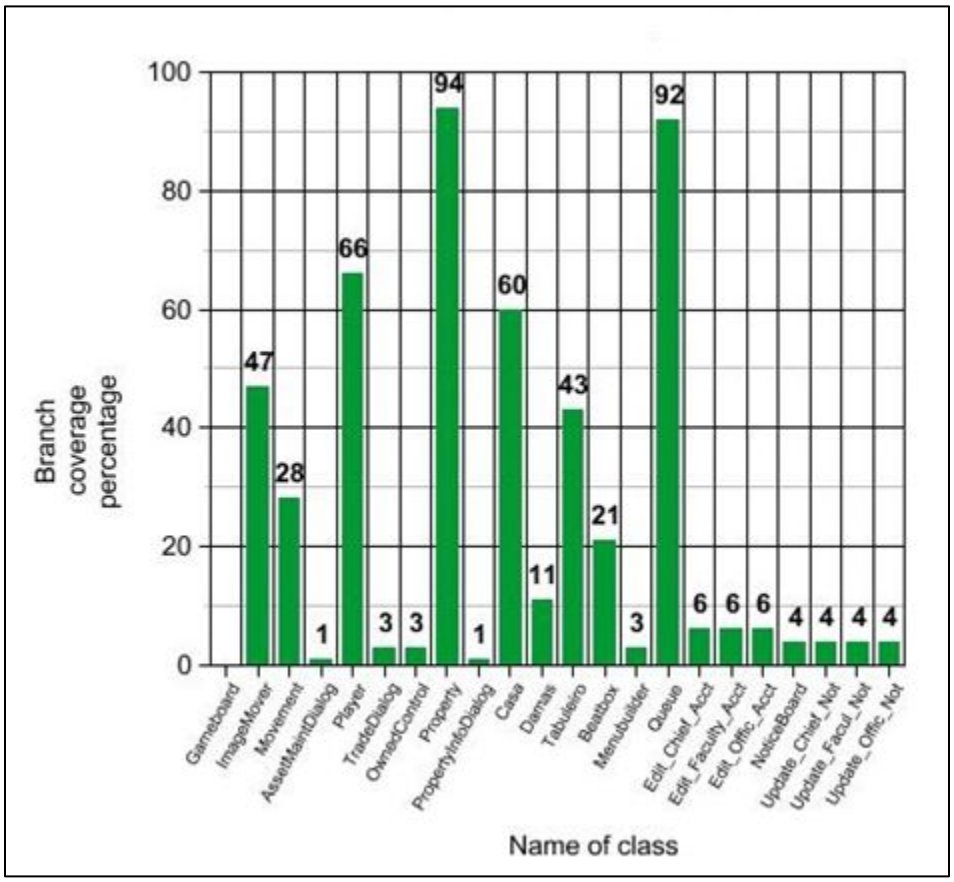

Figure 12: Test Run 2

4.6 Test run 3 branch coverage

Figure 13 shows the branch coverage for test run 3.

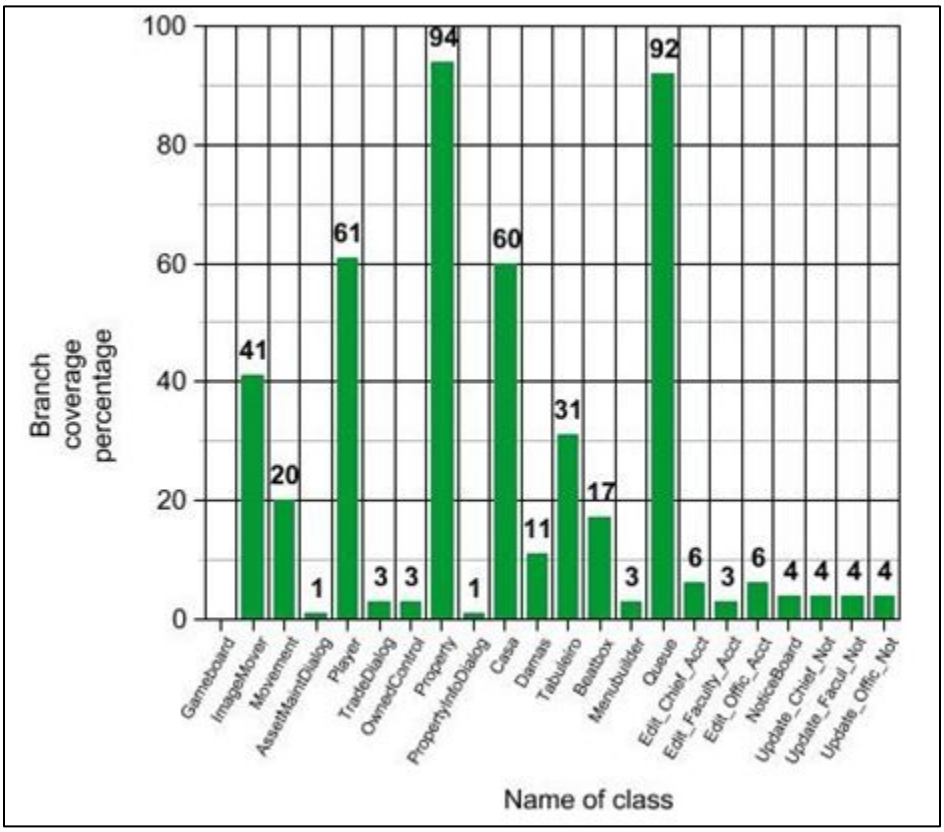

Figure 13: Test Run 3

4.7 Number of Branches

Figure 14 shows the number of branches for each class.

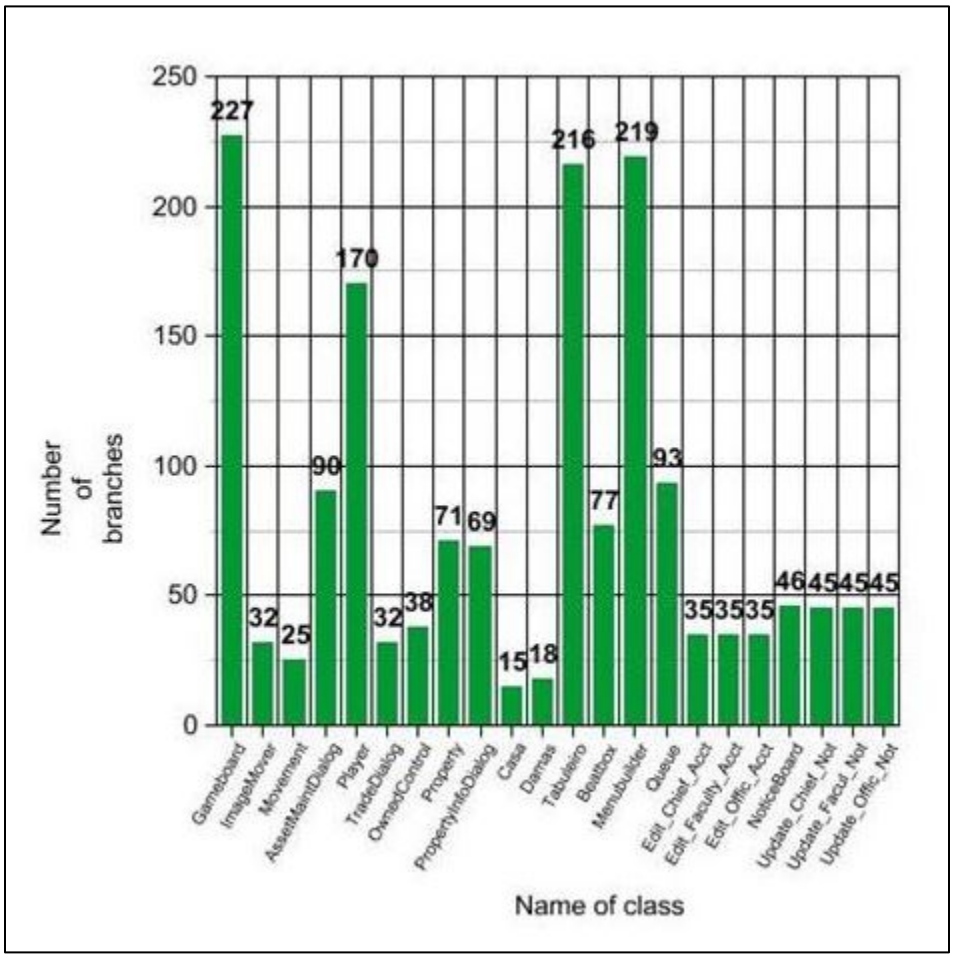

Figure 14: Number of Branches

4.8 Lines of Code

Figure 15 shows the lines of code for each class.

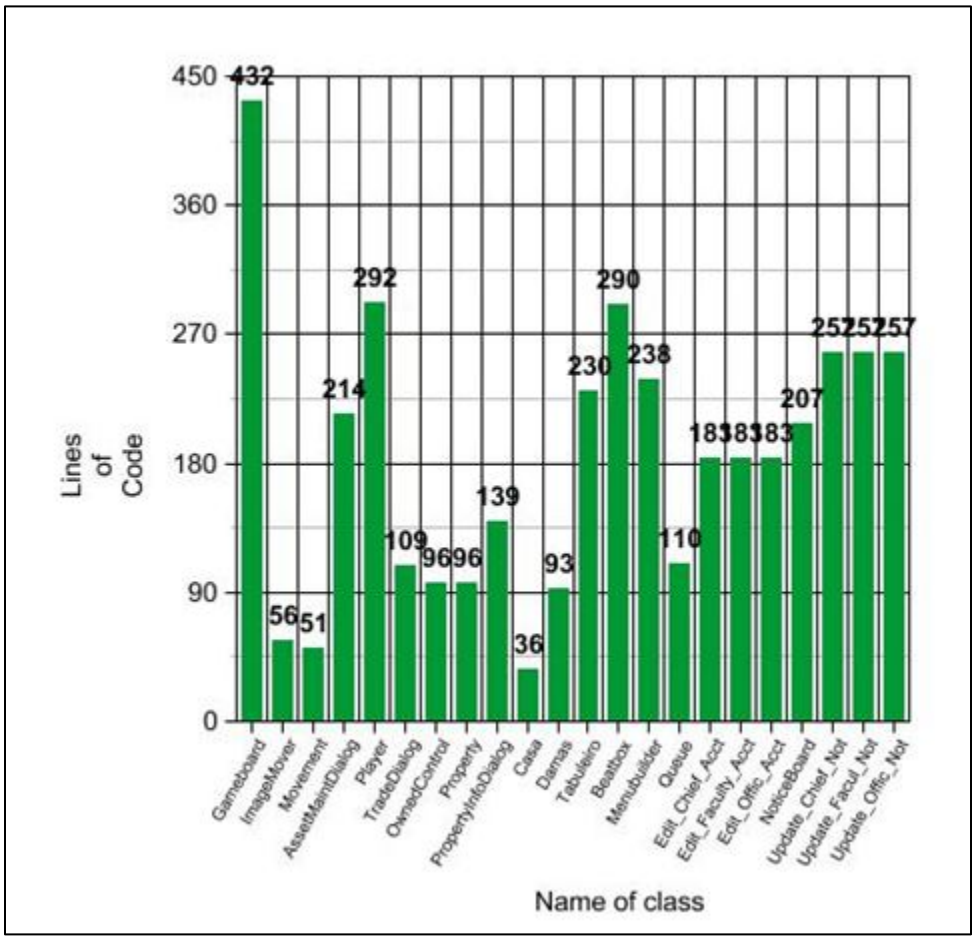

Figure 15: Lines of Code

4.9 Summary of Coverage

Table 7 lists a summary of the coverages for EvoSuite across all three test runs. Table 7 also includes data on lines of code and number of branches.

Name of class	Lines of Code	Number of Branches	Test Run 1 Coverage %	Test Run 2 Coverage %	Test Run 3 Coverage %
Gameboard	432	227	0	0	0
ImageMover	56	32	47	47	41
Movement	51	25	28	28	20
AssetMainDialog	214	90	1	1	1
Player	292	170	62	66	61
TradeDialog	109	32	3	3	3
OwnedControl	96	38	3	3	3
Property	96	71	89	94	94
PropertyInfoDialog	139	69	1	1	1
Casa	36	15	60	60	60
Damas	93	18	11	11	11
Tabuleiro	230	216	42	43	31
Beatbox	290	77	21	21	17
MenuBuilder	238	219	3	3	3
Queue	110	93	94	92	92
Edit_Chief_Acct	183	35	6	6	6
Edit_Faculty_Acct	183	35	6	6	3
Edit_Office_Acct	183	35	6	6	6
NoticeBoard	207	46	4	4	4
Update_Chief_Not	257	45	4	4	4
Update_Faculty_Not	257	45	4	4	4
Update_Office_Not	257	45	4	4	4

Table 7: Coverage Results for All Three Test Runs

It is clear from the results of this study (see Table 7) that EvoSuite performed well in a couple of instances, but performed poorly in several other instances. In fact, in half of the classes tested, EvoSuite achieved 10% or less coverage. Many of these classes in which EvoSuite performed poorly contained significant LOC/Branches. This is in line with the results that Fraser found in his study, namely that EvoSuite performed poorly for a high number of classes that he tested.

One thing to note is that many of the classes with significant lines of code had low branch coverage scores. Of the last seven classes tested, all had LOC equal to or greater than 183, but had fewer numbers of branches than some of the other classes. These classes had low branch coverage scores. The Property class, on the other hand, had 71 branches, 96 LOC, and scored coverage of 89, 94, and 94 % across three test runs. The Property class was close to Fraser's threshold of 100 lines of code or more for constituting a non-trivial class. The Queue class had 110 LOC, 93 branches, and a coverage score of 94, 92, and 92% across three test runs. Then there is the Gameboard class, which had 432 LOC, 227 branches, and achieved a coverage score of 0%. Overall, EvoSuite achieved very low branch coverage levels for the most complex classes (200+ branches), while it performed at its best on classes with middle-of-the-road complexity (50-100 branches).

It is also important to note that across test runs, as the amount of time allotted for each class increased in the parameter settings, the number of generations created by the EvoSuite tool did not necessarily increase proportionally. For the Gameboard class, from test run 2 to test run 3, the number of seconds it took for the search to complete almost doubled, however, the number of generations produced by EvoSuite decreased dramatically, even though EvoSuite was running on double the amount of cores and had two additional minutes per class allotted on test run 3.

4.10 Addressing the Outliers

One of the hypothesized reasons that Fraser gave for why EvoSuite might have performed so poorly in his study (and which rings true for these thesis findings as well) is

that EvoSuite has a built-in security manager that allows it to prevent certain classes from executing if the code in those classes is deemed unsafe. This is particularly relevant for cases in which the environment of the program under test (external files, networks, databases) cause the security manager inside of EvoSuite to prevent the tool from achieving a high level of branch coverage. Another reason that Fraser hypothesized might be responsible for EvoSuite performing so poorly in his study is that EvoSuite is not built to handle multi-threaded code. If there are places in the classes under test where code is spawning threads, EvoSuite will not perform well in terms of achieving high levels of branch coverage.

Chapter 5

CONCLUSIONS

Fraser's study introduced the genetic algorithm tool EvoSuite. This thesis made use of the EvoSuite tool to perform experimental research on Java classes. There were two reasons for using the EvoSuite tool in this author's thesis research. The most obvious one is that it was the only genetic algorithm tool found during a thorough web search for genetic algorithm tools that offered the option of measuring branch coverage of test inputs. The second reason is that Fraser's research, cited in this thesis, made use of EvoSuite, so there was already a research precedent set for using the tool. The EvoSuite tool automatically generated unit tests for the Java classes that were used for his studies, and then ran those unit tests and measured the level of branch coverage achieved during the tests.

Fraser's research focused on running EvoSuite as an Eclipse plugin and measuring the levels of branch coverage achieved. In one of his studies, his test bed was called SF110 and consisted of 110 Java programs from SourceForge, including the ten most popular Java programs at the time of his research. His test bed included 23,886 classes, 6.6 million lines of code, and required 995 days to run. At the end of his study, he found that most of the classes tested fell into one of two coverage intervals: either 10% and less, or 90% and more. He found that a significant number of classes that he tested achieved low branch coverage levels, and that EvoSuite performed poorly on these classes. He also

found that there were a significant number of classes in which EvoSuite performed well, with over 90% branch coverage achieved, but that these classes were usually simpler in regards to the number of conditional statements contained in the class. For example, classes in the 10, 20, and 30% coverage intervals contained 70 or more branches, while those classes in the 90% coverage interval contained much fewer branches.

Fraser hypothesized that the poor performance of EvoSuite on classes in his study that had high numbers of branches was due to the internal, built-in security manager found in EvoSuite. He suggested that this built-in security manager was detecting unsafe code and was prohibiting that code from executing. He found that in classes where no exceptions were raised, the EvoSuite tool performed well, achieving over 85% branch coverage, but that in classes where permissions had to be granted, the EvoSuite tool performed less efficiently in terms of the number of branches covered. Another reason that Fraser hypothesized was that since EvoSuite cannot handle multi-threaded code, there must have been some places in the classes under test where threads were being spawned, as this would negatively affect the performance of EvoSuite in terms of providing high levels of branch coverage.

This thesis sought to use EvoSuite not as an Eclipse plugin, but as an IntelliJ IDEA plugin. In this way, this thesis research paper makes a unique contribution to the field, because no one has published any literature on the use of the EvoSuite plugin for IntelliJ. The experimental design of this thesis sought to test EvoSuite against twenty-two classes, obtained from various Java programs from SourceForge, PlanetSourceCode, and a data

structures class from the University of North Florida. These twenty-two classes to be tested had varying levels of complexity in terms of branches contained in each class, as well as various sizes in terms of LOC. The default parameters in the IntelliJ EvoSuite plugin would be tested in three ways: the default parameters would be used for test run one, and then for test runs two and three, the parameters would be altered to observe any noticeable effect on performance.

This study found that the EvoSuite tool performed as expected, in a varying degree of efficiency. For some of the large, complex classes such as the Gameboard class (432 LOC, 227 branches) the EvoSuite tool performed extremely poor, covering only one branch out of 227 for all three test runs. For this reason it achieved a 0% level of branch coverage. However, for the Player class (292 LOC, 170 branches) the EvoSuite tool achieved 62%, 66%, and 61% for all three test runs, numbers which Fraser regards as being decent. There were several classes which achieved coverage levels of less than 10% across all three test runs, which is noticeably similar to the performance in Fraser's study. There were far fewer classes which achieved a high level of performance across all three test runs. The two that performed very well were the Property class (96 LOC, 71 branches) which achieved coverage levels of 89%, 94%, and 94% across all three test runs; and the Queue class (110 LOC, 93 branches) which achieved coverage levels of 94%, 92%, and 92%. In this way, the IntelliJ EvoSuite plugin mirrored the performance of the Eclipse plugin in Fraser's study, albeit on a much smaller scale. It's also important to note that the adjustment of the parameters in the IntelliJ plugin had little to no effect on the level of branch coverage achieved for any given class. Exceptions are when EvoSuite

IntelliJ plugin achieved 94% coverage on the first test run on the Queue class, and 92% coverage on the second and third test runs. The Beatbox class had a coverage score of 21% on the first and second test runs, and a score of 17% on the third run. One of the biggest drops amongst runs was the Tabuleiro class, which had a coverage score of 42% on the first run, 43% on the second run, and 31% on the third run. This is interesting because on the third test run, the parameters were extended from one core to four cores, from 2,000 MB per core to 10,000 MB per core, and a class time from three minutes to seven minutes. It's interesting that this actually resulted in poorer performance for the tool.

It is also important to note that across test runs, as the amount of time allotted for each class increased in the parameter settings, the number of generations created by the EvoSuite tool did not necessarily increase proportionally. For the Gameboard class, from test run 2 to test run 3, the number of seconds it took for the search to complete almost doubled, however, the number of generations produced by EvoSuite decreased dramatically, even though EvoSuite was running on double the amount of cores and had two additional minutes per class allotted on test run 3.

It is also important to note that the alteration of the parameters for the IntelliJ EvoSuite plugin to not significantly affect the performance of the tool. For most classes, the amount of branch coverage achieved across the three test runs, each with different parameters, stayed consistently the same. In a couple of instances, the branch coverage

level actually dropped on the second and third runs, even though the parameters for those runs had been adjusted in such a way so that performance would improve.

We can conclude, therefore, that the performance of EvoSuite is varied, with a strong tendency towards poorer performance in the program tested. There are instances where EvoSuite performed well. However, Fraser's study was verified by the performance of the EvoSuite tool within the IntelliJ IDE as a plugin. This thesis made a unique contribution by studying the EvoSuite tool as an IntelliJ plugin. Additional research may be needed to further verify results. Therefore, based on the literature reviewed in this thesis, as well as the experimental research performed by this author using the EvoSuite tool, this author concludes that not only is the EvoSuite tool not a reliable tool to use for generating test data for branch coverage, but that genetic algorithm-based test data generators as a whole are too varied in their performance to be considered a reliable option for generating test data for the objective of achieving branch coverage.

5.1 Future Research

It is clear that genetic algorithms can be useful, however, given the research data collected and the experimental research performed in this paper using EvoSuite, it becomes clear that genetic algorithm-based test generation tools are not a perfect solution to the problem of achieving branch coverage. This author recommends further research. Particularly, this author recommends testing the EvoSuite tool using the command line. All of the available literature from Fraser on his use of the EvoSuite tool focuses on the EvoSuite Eclipse plugin, and this thesis paper used the IntelliJ plugin. However, nothing

has been written about the use of EvoSuite from the command line. Perhaps eliminating a GUI and communicating directly with the computer's operating system will affect how EvoSuite performs. In fact, EvoSuite was designed with the command line in mind. This author also recommends more in-depth research into how EvoSuite's built-in security manager affects the performance of the EvoSuite tool. Specifically, how exactly does the security manager determine which code is harmful to execute? What is the exact way in which the operating environment can affect EvoSuite's performance? Is there any way to overcome the problem of the security manager not allowing EvoSuite to achieve high branch coverage? Further research is needed to answer these questions. The author also recommends further research using a comparison of randomized test data generation tools against EvoSuite, which this thesis did not do. This author also suggests exploring related methodologies for solving problems, including evolutionary programming and gene expression programming, as solutions to the issue of achieving branch coverage. Finally, this author suggests using other forms of genetic algorithm-based test input generators not employed in this paper. This includes running tests with partitioned (or multiple population) GAs to study the performance of a genetic algorithm that has many smaller populations in for each generation of tests, instead of just one large population

REFERENCES

Print Publications

[Alshraideh11]
Alshraideh, M., B. A. Mahafzah and S. Al-Sharaeh, "A Multiple-Population Genetic Algorithm for Branch Coverage Test Data Generation," Software Quality Journal 19,3 (2011), pp.489-513.

[Chang01]
Chang, S.K. Handbook of Software Engineering and Knowledge Engineering: Fundamentals. World Scientific Publishing Co. Inc., Singapore, 2001.

[Fraser11]
Fraser, G. and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software," Proceedings of the 19th ACM SIGSOFT symposium, September 5-9, 2011, Szeged, Hungary (2011), pp. 416-419

[Fraser13]
Fraser, G., M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated white-box test generation really help software testers?," In Proceedings of the 2013 International Symposium on Software Testing and Analysis, July 15-20, 2013 Lugano, Switzerland, ACM. (2013), pp. 291-301

[Fraser14]
Fraser, G. "A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite." ACM Transactions on Software Engineering and Methodology 24,2 (2014), pp. 8-48

[Gupta00]
Gupta, N., A. P. Mathur, and M. L. Soffa, "Generating Test Data For Branch Coverage," Proceedings of the 15th IEEE international conference on Automated software engineering, September 11-15, 2000, Grenoble, France (2000) p.219

[Khor04]
Khor, S. and P. Grogono, "Using a Genetic Algorithm and Formal Concept Analysis to Generate Branch Coverage Test Data Automatically," Proceedings of the 19th IEEE international conference on Automated software engineering, September 20-24, 2004, Washington, D.C. (2004) pp. 346-349

[Korel90]
Korel, B., "Automated Software Test Data Generation," IEEE Transactions on Software Engineering 16,8 (1990), pp. 870-879.

[Mahajan12]
Mahajan, M., S. Kumar, and R. Porwal. "Applying Genetic Algorithm to Increase the Efficiency of a Data Flow-based Test Data Generation Approach," ACM SIGSOFT Software Engineering Notes 37,5 (2012), pp. 1-5

[Michael01]
Michael, C., G. McGraw and M. Schatz,"Generating software test data by evolution," IEEE Transactions on Software Engineering 27,12 (2001) pp.1085-1110

[Pargas99]
Pargas, R.P., M.J. Harrold and R. R. Peck, "Test-Data Generation Using Genetic Algorithms," Journal of Software Testing, Verification and Reliability 9 (1999), pp. 263-282

[Rohil08]
Rohil, M.K. and N. K. Gupta."Using Genetic Algorithm for Unit Testing Of Object Oriented Software," IJSSST 10,3 (2008), pp. 97-102

[Zhao00]
Zhao, J. , "Dependence Analysis of Java Bytecode," Proceeding: COMPSAC '00 24th International Computer Software and Applications Conference, October 25-28, 2000, Washington, D.C. (2000) pp. 486-491

Electronic Sources

[Software15]
Software Testing Mentor, "Decision Coverage or Branch Coverage," http://www.softwaretestingmentor.com/test-design-techniques/decision-coverage/, 2015, last accessed September 20, 2016

APPENDIX A

Glossary

Child	The test input that is created as a result of recombination or mutation
Chromosome	An individual test input that's a part of a genetic algorithm population. For example, a test input that is designed to cover a true branch nested deep within the code
Crossover	The act of exchanging genes between parents to produce a new offspring
Fitness Function	An indicator of how fit a parent test input is for creating a new offspring. This is usually measured in terms of whether an input can come close to covering a target branch
Genes	The individual components of a test input that are used during recombination
Genetic Algorithm	An algorithm that functions using the basic science of natural selection, including recombination, mutation, crossover, etc.
Mutation	The act of taking an element from a parent test input and changing it so that it produces a new child offspring that is not identical in genetic makeup to the parent test input
Parent	A test input that is chosen for recombination
Population of Chromosomes	A group of test inputs that may be initially seeded via random methods or may be generated by the genetic algorithm
Recombination	The act of taking elements from two parent test inputs and combining them to create a child offspring test input.

VITA

Jason Frier is a Software Engineering graduate student in the School of Computing at the University of North Florida. Jason anticipates graduating with his degree in April, 2017. Dr. Robert Roggio is serving as Jason's thesis advisor. Jason's academic work had included use with Java, C#, ASP.NET, HTML, PHP, and JavaScript, as well as the testing tools EvoSuite and the Integrated Development Environments Eclipse, Visual Studio, and IntelliJ. Jason is currently working outside of the software industry. Jason currently has interests in software testing, including white box and black box testing.