2006

APPLICATIONS OF GENETIC ALGORITHMS TO RANDOMIZED UNIT TESTING

Felix Chun Hang Li
Western University

APPLICATIONS OF GENETIC ALGORITHMS TO RANDOMIZED UNIT TESTING

(Spine Title: Genetic Algorithms for Randomized Unit Testing)

(Thesis Format: Monograph)

by

Felix Chun Hang Li

Graduate Program in Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
December, 2006

THE UNIVERSITY OF WESTERN ONTARIO
FACULTY OF GRADUATE STUDIES

CERTIFICATE OF EXAMINATION

Supervisor

James H. Andrews

Supervisory Committee

Examiners

Duncan Murdoch

John Barron

Charles X. Ling

The thesis by
Felix Chun Hang Li
entitled

APPLICATIONS OF GENETIC ALGORITHMS TO RANDOMIZED UNIT TESTING

is accepted in partial fulfillment of the
requirements for the degree of
Master of Science

Date

Chair of Thesis Examining Board

Abstract

Software testing plays a critical role in the software development lifecycle. A well-developed test strategy can effectively evaluate the correctness of a piece of software and find bugs. One of these test strategies is randomized unit testing. Randomized unit testing allows a tester to randomly generate a sequence of method calls that can cause faulty behaviours in a program (i.e. a failing test case). This thesis focuses on using Genetic Algorithms (GAs) to help make randomized unit testing more useful and easier to use. We use GAs in failing test case minimization, which can facilitate the debugging process for software testers. We also use GAs to help finding optimal input values for randomized test case generation, which acts as a foundation that can enhance the randomized test case generation process.

Keywords: Software Testing, Unit Testing, Randomized Testing, Test Case Minimization, Genetic Algorithm

Acknowledgments

Lots of thanks go to Dr. Jamie Andrews, who gave me the opportunity to perform researches with him. Being able to learn from him is one of the most valuable experiences in my life. Jamie has given great help on the thesis topic, application development and experiment setup and analysis. It is impossible to have finished this thesis without his help.

Special thanks to Akbar Siami Namin from our research group who has suggested the idea of random assignment of gene values for the Genetic Algorithm application that finds optimal input values for randomized test case generation. Taking his suggestion has improved the time efficiency and effectiveness of the application.

Sincere thanks to Robert Mercer who reviewed my thesis proposal and suggested incorporation of the normalization of chromosomes in the Genetic Algorithm application which improved the quality of the solutions.

The rest of the thanks go to my family, Simon Shu Chuen Li, Margaret Oi Kwan Lee, Elvis Chun Yeung Li and my girlfriend, Hilda Hiu Ying Pang. The spiritual and financial support that they have given me has always been tremendous.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

The software development lifecycle is the process that software must go through during its development time. The cycle consists of the following major phases: requirement analysis and specification, design, implementation, testing, and maintenance. The overall quality of the developed software depends heavily on the quality of the execution of each phase in the lifecycle. This thesis focuses on building improvements within the software testing phase. We will discuss problems that software testers encounter in the course of testing a piece of software. These problems are major concerns to a type of software testing methodology called randomized unit testing. We have developed solutions to these problems using algorithms drawn from a particular class of algorithms that use techniques inspired by evolutionary biology, namely Genetic Algorithms (GAs).

1.2 Testing and Unit Testing

Software testing plays a critical part in the software development lifecycle. It is used to evaluate correctness, completeness and quality of a piece of software and facilitates making any improvements which are deemed to be necessary. Although we can never be sure that a program is completely free of problems (bugs), we can create a well developed testing strategy that can increase the chance of finding a fault if one exists.

There are many different approaches to software testing [9]. Depending on the type of the software implementation, different testing approaches yield different results and have different objectives. Black-box testing focuses on verifying the correctness of the functionality of the software; white-box testing validates the correctness and completeness of the logic represented by the actual source code. System testing is a thorough testing of the entire system while regression testing attempts to confirm the correctness of the functionality of the existing software after new features have been integrated. Finally, unit testing helps to validate the correctness and completeness of a particular module of the source code. Unit testing provides a foundation to system testing since it checks for correctness of each module that is included in the software package, making sure that they are working according to specification when they are running separately. After that, integration testing can be carried out in which modules are brought together to confirm validity. Finally, system testing can also be carried out based on the assumption that all basic underlying modules are free of errors.

1.3 Code Coverage

Code coverage is the measurement of the portion of the code that has been run (covered) when the program executes. When code coverage is applied in software testing, it is an excellent indication of the completeness of the test that is being carried out. The more code that is covered by running a test, the more thoroughly the code is being tested, and the higher the possibility of finding a bug if it exists.

Coverage can be measured in various ways. Line coverage measures the total number of lines covered in the code; decision coverage checks to see whether a true and false decision has been made at each decision point; path coverage can help indicate whether every possible route through the code has been executed. While there are many different types of coverage measurement, some of them are considered to be stronger than others. For example, path coverage is considered to be stronger than line coverage because covering all possible paths of code execution also means covering all lines in the code; but covering all lines in the code does not necessarily mean covering all paths of code execution.

The degree of completeness of each type of coverage measurement depends on the feasibility of developing test scripts that achieve them given a limited amount of resources. Systems which can cause critical problems when they fail tend to use stronger coverage measurements to test the system and achieve very high code coverage; examples are flight control systems, life support systems and nuclear power plant control systems. Throughout the research, we have made use of code coverage measurement together with GAs to develop our solutions. Cobertura is the coverage tool that we

have decided to use.

1.3.1 Cobertura

Cobertura is an open-source Java coverage tool. It reports the amount of line coverage and branch coverage after executions of a program. After the Java bytecode (i.e. class files) has been instrumented and the program executed, Cobertura can generate reports in HTML or XML formats. The line and branch coverage for each source file whose corresponding class files have been instrumented, will be shown in the report. Details about the number of times each line has been executed is also available for further examination. This gives users an idea of the part of the code that is executed more often which is possibly more complex and critical to the unit under test, and the part of the code that is executed fewer times, possibly meaning that it is less important or dead code (i.e. code that can never be reached).

1.4 Randomized Unit Testing

This thesis focuses on a technique called *randomized unit testing*. Running a test case in randomized unit testing consists of executing a sequence of calls to methods in the unit we would like to test. It requires randomization in both selecting the methods to be called and selecting the parameters to be passed into the particular chosen method call. The results obtained by executing the test case can be compared

to expected results after each method call is made. Any discrepancies between the obtained results and expected results can identify a possible fault within the unit under test. When it is utilized properly, we have found that randomized unit testing is efficient and easy to carry out.

1.4.1 RUTE-J

RUTE-J, a Randomized Unit Testing Engine for Java [2], is an automated software testing tool that utilizes the randomized unit testing methodology in order to search for faults embedded in the unit under test. To use RUTE-J, the tester creates a subclass of a class called TestFragmentCollection. Each method in this subclass is referred to as a "test fragment" and starts with a prefix tf_ followed by a method name. Each of these test fragments calls the method that the tester would like to test in the original unit under test. Java assertions allow testers to confirm correctness at different points of the code and can be inserted into the test fragments. RUTE-J makes use of this TestFragmentCollection subclass that the tester has created to randomly execute the test fragments within this subclass. At the same time, the parameters required by a randomly chosen test fragment are also chosen randomly during the fragment's execution. The assertions in the test fragments can then help to identify unexpected results generated by executing the sequence of test fragments. Once an unexpected result has been identified, RUTE-J quits the test case generation process and reports to the tester the failing test case that causes the unexpected result.

In order to test the unit more efficiently, RUTE-J provides the ability for testers to

define the frequency that each test fragment is chosen for a test case. Since methods within a unit vary in their importance according to the usage of the unit, setting the frequency of executing each test fragment properly can increase the chance of finding a fault within a unit if it exists. For example, a method which is comparatively more important than the other methods will require more executions to cover all code logics made within it; on the other hand, a comparatively less important method require fewer executions within a test case to cover all code logics made within it. RUTE-J also allows testers to choose the test case length(number of method calls) and the number of test cases to generate. This allows the tester to balance thoroughness and efficiency in the randomized unit testing process.

1.5 Genetic Algorithms

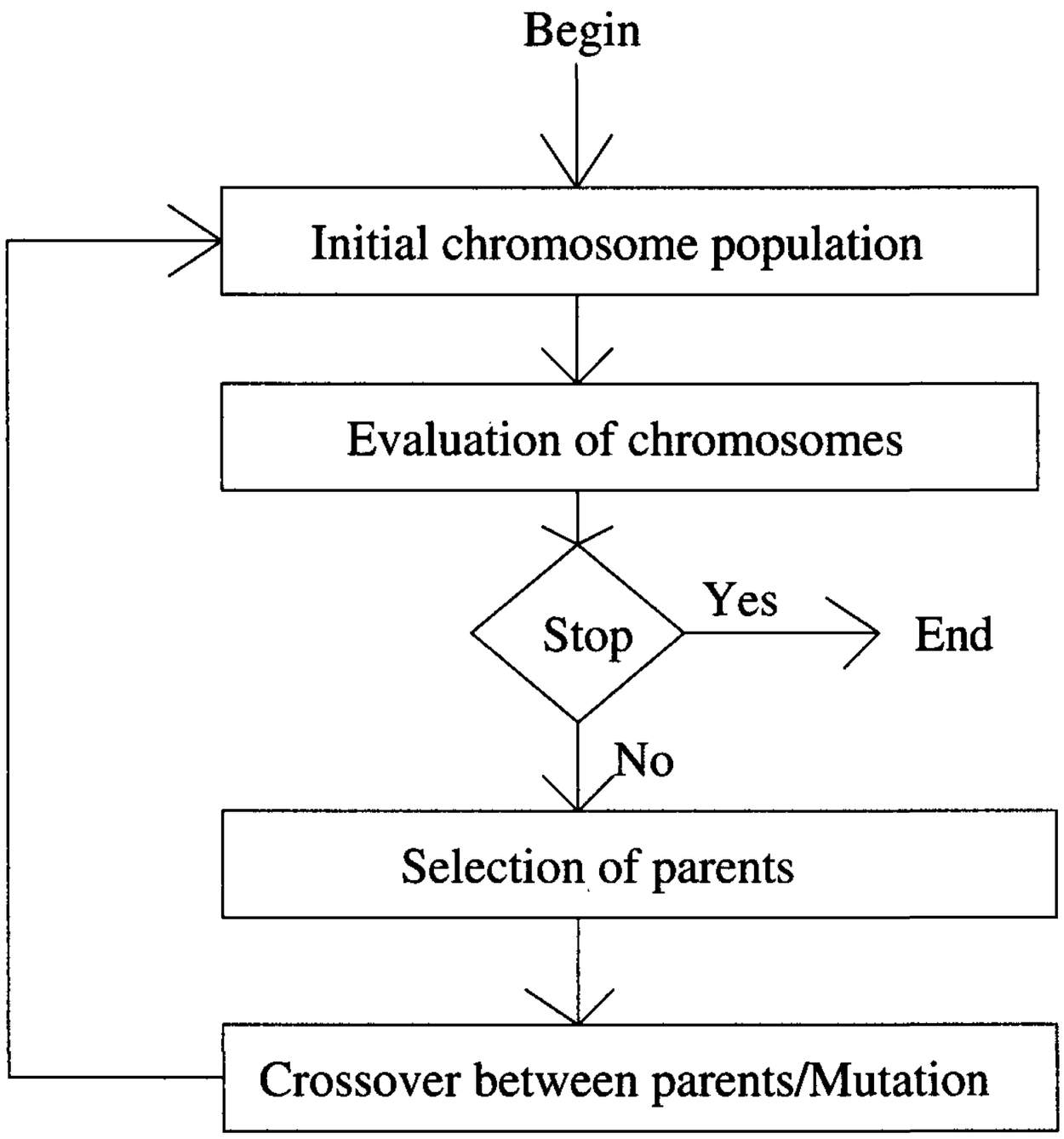
Genetic Algorithms (GAs) are a class of algorithms that is typically used to find maximum or minimum solutions to problems. They are popular because of their randomness and robustness [7]. GAs work with probabilistic transition rules rather than deterministic rules, such that multiple solutions can be identified and global minima and maxima are more likely to be reached than they are by random search.

A GA consists of a population of chromosomes in which each chromosome is an array of genes, similar to a human chromosome representation. A summary of the GA process is shown in Figure 1.1. The process begins by initializing a population of chromosomes such that each chromosome is a representation of a solution to the

problem that is being solved. Next, the chromosomes are evaluated by a fitness function to see how good a solution each one of them is representing. The GA then looks for either a higher or lower score result from the fitness function depending on the problem that it is trying to solve. After that, the GA process can then decide to either stop or continue. If the process has chosen to continue, the chromosomes which tend to have a better representation of a good solution will be chosen as "parents" and they can perform an operation called crossover that gives GAs an advantage over random search. The left-hand side of figure 1.2 describes the operation of crossover, in which a pair of selected parent chromosomes exchange gene values at a randomly chosen position. The operation gathers the solutions represented by the pair of parent chromosomes to simulate multiple children chromosomes in the hope that they will represent better solutions to the problem than their parents. In the mean time, mutation operations can also be carried out in the chromosomes representing good solutions. The right-hand side of figure 1.2 also describes the operation of mutation which involves a random change of gene values at random positions within a single selected chromosome. With mutation operations, GAs maintain a possibility of reaching the global minima or maxima by randomly assigning randomized values to the genes.

Finally, a population of new chromosomes has been generated by the crossover and mutation operations, and the GA process iterates until a stopping condition is reached. At the end, the chromosome with the highest score evaluated by the fitness function will represent the best solution found to the problem being solved. Since GAs work with probabilistic transition rules, running GAs multiple times on the same problem can possibly yield different solutions.

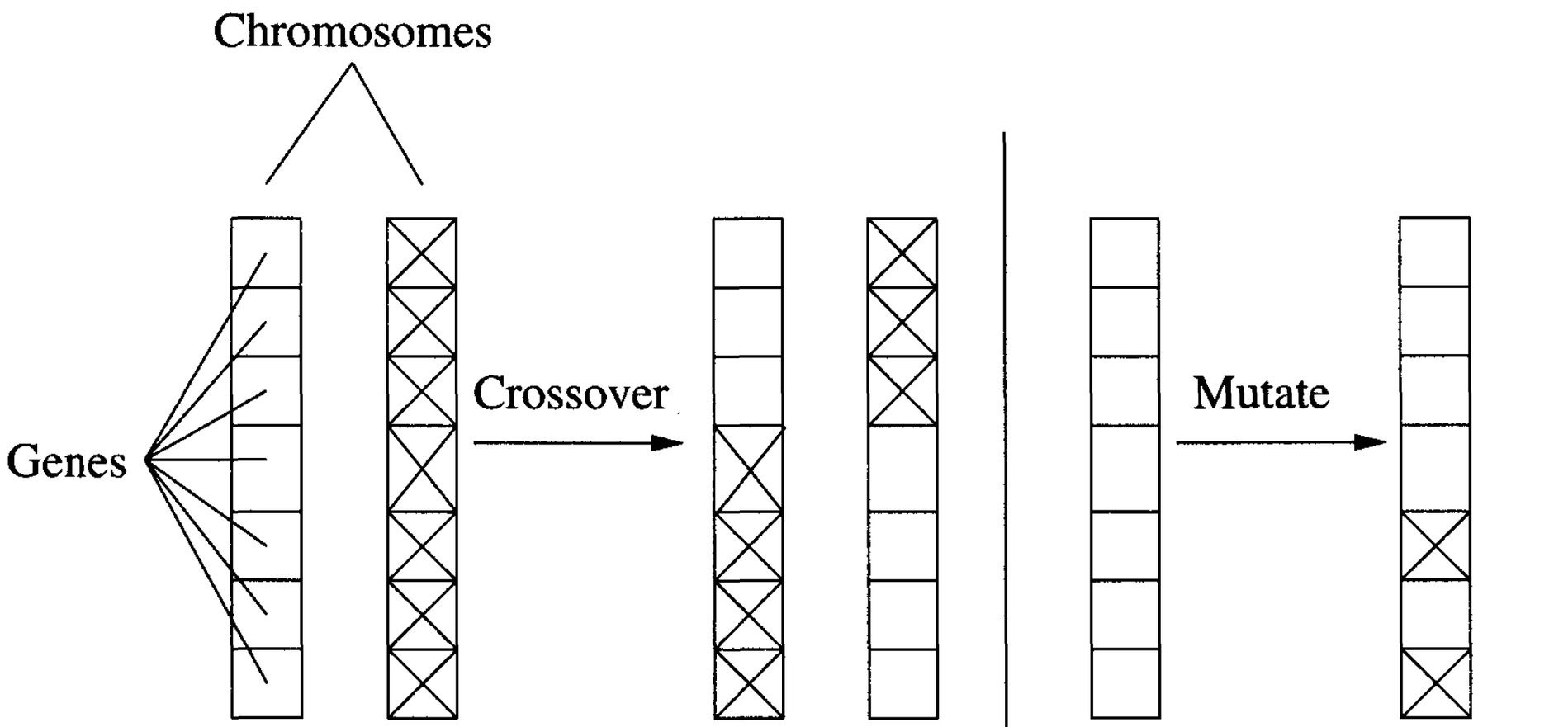
Figure 1.1 Summary of the GA process



1.6 Thesis Focus

Because of the fact that randomized testing performs a non-deterministic strategy in finding possible faults in a test case, the failing test cases generated by randomized testing can be lengthy and cause great difficulty for testers in pinpointing only the necessary method calls that contribute to the faulty behaviour. As a result, failing test case minimization is necessary. RUTE-J makes use of the Delta Debugging algorithm [14] to minimize failing test cases. This algorithm can deterministically generate a test case which is locally minimal in the sense that removing any method calls will not be able to produce the original faulty behaviour. We will discuss applying Genetic

Figure 1.2 Crossover and Mutation of chromosomes



Algorithms to achieve efficient failing test case minimization. Using a deterministic test case minimization algorithm like the Delta Debugging algorithm, only a single identical minimized test case can be obtained; but by using a GA for failing test case minimization, we may be able to obtain multiple minimized failing test cases, each of which indicates different faulty behaviours of the program. Therefore, software testers can obtain more information from a single failing test case where one or more faulty behaviours might be embedded within.

Our evaluation results have shown that using GAs to solve the failing test case minimization problem gives significantly shorter test cases in some cases and significantly longer test cases in other cases when compared with the Delta Debugging minimization approach. With that being said, GAs are able to give us multiple minimized test cases, which provide testers more information about the problems within the code.

We will also talk about the search for optimal input values for randomized test case generation. Randomized test case generation can randomly generate test cases based on some certain input values such as the frequency of method calls and the length of the test cases to be generated. Choosing these input values for randomized testing tools like RUTE-J is sometimes considered problematic since testers find it difficult to choose values that fit their testing purposes. Testers need to estimate the optimal input values such that a failing test case can be generated efficiently if it exists. We have therefore been motivated to ask whether using the properties of GAs and integrating with coverage measurement can help them in achieving this goal.

Using GAs for solving this problem has proven to give testers insight about the relative importance of the method calls within a unit being tested. At the same time, it provides us clear guidance on the input values needed by randomized test case generation.

1.7 Thesis Organization

Introduction and other relevant background information has been highlighted in chapter 1. We shall talk about some related work that has been done concerning both randomized unit testing and GAs in chapter 2. We shall discuss the approach in solving the failing test case minimization problem in chapter 3 and the idea of finding optimal input values for randomized test case generation in chapter 4. Both

of these chapters contain a set of statistical analyses which evaluate the efficiency and effectiveness of their corresponding proposed strategies. Chapter 5 provides a conclusion of materials discussed in the thesis and possible future work that can be implemented.

Chapter 2

Related Work

In this chapter, we give some basic definitions, and then discuss related work in randomized testing, genetic algorithms and test case minimization.

2.1 Definitions

Here we define some terms that will be useful through the rest of this thesis.

A **failing test case** is a test case which will cause the unit under test to fail, either by making it crash (Bus Error, arithmetic exceptions etc.), or by causing a running error (infinite loop or unexpected results).

A **passing test case** is a test case which does not cause the unit under test to fail.

In some situations, test case input can be broken up into individual units, such as lines of input, commands or characters. In these cases, we define a **minimized test case** as a failing test case such that, removing any single line will make it a passing test case.

2.2 Randomized Testing

Randomized testing is the simple strategy of generating randomized input and feeding it to the software under test. It is mentioned as early as Myers in 1979 [13]. Past research on randomized testing included that of Claessen and Hughes on QuickCheck [5]. QuickCheck is a testing tool that utilizes randomized testing to test Haskell programs. Using formal specifications, QuickCheck allows testers to define certain properties of the functions under test that should be expected and check whether the properties hold after running some test cases. The tool also has the ability to automatically generate test cases based on random inputs or based on custom defined test data generators.

Miller et al. [12] has also proven the effectiveness of random testing to end users. By simply randomly generating strings of characters using a program called fuzz, they found that a surprisingly large number of UNIX utility programs either terminate abnormally, loop infinitely or terminate without a clear description of what has happened, totaling to more than 24% of the basic UNIX utility programs.

Randomized unit testing is a specific type of randomized testing which automates the

testing by randomly generating sequences of function calls. Andrews [1] focused on coverage-checked random unit testing (CRUT), which applies randomized unit testing methodology to a given unit under test, continuously testing it until predefined coverage goals are achieved. Andrews concluded that CRUT has proven to be efficient in finding faults within the code and it can act as a complement to other types of structural and functional testing methodologies.

2.3 RUTE-J

The Randomized Unit Testing Engine for Java (RUTE-J) [2] was developed to randomly generate sequences of function calls based on tester preferences. It allows the tester to specify the length of the test case and the number of test cases to be generated and also to specify a time limit for continuously generating test cases. The engine stops generating test cases when the stopping condition has been satisfied, or it can stop once a fault is found, providing the failing test case and a log file as one of the outputs.

2.3.1 Using RUTE-J

To generate a test case in RUTE-J, the tester will first need the TestFragmentCollection of the unit that they are interested to test. Then the command to initiate the application can be as follows:

```
java TextDriver TestFragmentCollection
```

By doing this, the default process of randomly generating test cases is selected, which will generate one test case of length 50 test fragment calls. To set the desired length of test case to be generated to 500 and the number of runs desired to 10, the user can use the following command:

```
java TextDriver -repeat 500:10 TestFragmentCollection
```

There are also other methods for controlling the number of test cases and the corresponding length, such as continuously incrementing the length of test cases until a certain length limit is reached. Another well developed functionality of RUTE-J is test case minimization. If the user gives the command:

```
java TextDriver -repeat 500:10 -min TestFragmentCollection
```

then once a failing test case is found, RUTE-J will automatically minimize the test case using Zeller and Hildebrandt's algorithm. (see Section 2.5)

In terms of setting the weights for the test fragments, it is possible to do so by adding a few lines to the TestFragmentCollection class that the tester needs to create. The setWeight() method can help the tester to set the appropriate weight for the method specified. In case the tester is not interested or does not have the intention to set appropriate weights to the test fragments, all test fragments have a default weight of 100, meaning they are assumed to have equal importance in the subject unit.

RUTE-J also provides a graphical user interface for the ease of usage. It enables users to choose their desired method of generating test cases. Users can also view the generated or failed test cases, and minimize them if necessary. Saving and loading test cases is also possible, which helps users better organize their test plans. A feature that has not yet been implemented in the text interface in RUTE-J is the "break" button in the GUI version. This is necessary when the process of randomly generating test cases has forced the unit under test into an infinite loop. The break button allows the stoppage of the process without crashing the entire application.

2.3.2 Failing Test Cases

Consider the TreeMap class of the standard Java library java.util.TreeMap, which implements a red-black tree data structure for storing keys and associated data. Red-black trees are binary trees such that every node is colored either red or black, every red node that is not a leaf has only black children and every path from the root to a leaf contains the same number of black nodes. These node colouring properties ensure that a red-black tree is balanced and thus efficient, but the code for maintaining these properties is complex. If TreeMap had a fault in it, the fault might be revealed by a sequence of method calls as shown in Figure 2.1. It consists of calling a sequence of functions involving the size() method, put() method and getFirst() method.

Since RUTE-J automatically stops generating function calls when a test case fails, this failing test case implies that, by calling the last put(20, 30) method, the test case fails. Keep in mind that it does not necessarily mean that every function call plays a role in contributing to the fault. Some of the function calls in the test case might not

Figure 2.1 An example of a failing test case using the TreeMap unit.

treemap.size()
treemap.put(3, 10)
treemap.put(5, 7)
treemap.getFirst()
treemap.put(20, 30)

be needed in reproducing the fault. We will also use this example for later chapters in order to explain the ideas more clearly.

As an example of the effectiveness of RUTE-J, it has successfully found a previously undiscovered fault in the IMoney example, the main example distributed with the JUnit Java unit testing framework [3]. The IMoney example includes the Money class and the MoneyBag class which both implement the IMoney interface. They help to represent money in different currencies and contain methods such as for adding and subtracting currencies. Andrews et al. [2] found that the following test case, minimized from a longer failing test case of about 50 method calls, is able to force a failure in IMoney:

- add 2 0 5
- negate 5 7
- appendTo 7 2
- checkValue 2

The test case description refers to the TestFragmentInfo that has been written for

IMoney. The IMoney TestFragmentInfo maintains an array of IMoney objects. In the test case, first the index 2 element and the index 0 element are added, and the result is put in the fifth position in the array. Then the result is negated and put in the seventh position, and then the object stored at the seventh position is appended to the one stored at second position (it is unclear what appendTo() method does but it merges the monetary value for all currencies together in most cases). Finally the value of element stored in position 2 is checked. The expectation after executing the test case is to have a Money object stored in position 2, but the actual outcome is a MoneyBag object which creates a discrepancy and causes the test case to fail.

2.4 Genetic Algorithms

GAs can be applied to solve a variety of problems, generally to find maximum or minimum solutions. GAs are particularly applicable for solving scheduling and timetabling problems, but they have been also applied to testing in the past.

2.4.1 Comparision to other search algorithms

Random search is a simple algorithm that is used to look for a solution to a given problem. It explores the possible solution space by randomly selecting solutions and evaluating their fitness. GAs perform better than random search in the sense that each evolution tries to pass down its their good solutions to the next evolution in an

attempt to find even better solutions. This is made possible by performing crossovers and mutations on selected chromosomes. Since GAs help in giving a direction in retrieving a solution to the given problem, they have a higher probability of converging to a solution faster than random search.

Hill-climbing is a search algorithm which attempts to maximize a function. It tries to obtain the best solution by looking for solutions in the immediate neighbour set. If there is a better solution in the neighbour solution set, then that neighbour is chosen and the algorithm will continue the process of searching for better solutions from the new neighbour set; if there is no better solution in the surrounding neighbour set, then the solution is found and the current node will be the solution that is returned. The process of hill-climbing might take a shorter time than using GAs, but GAs have a higher probability of finding global maxima and minima; therefore, we have decided to use GAs for that reason.

Simulated annealing is also an algorithm for searching for an optimal solution to a given problem. It begins with finding a single random solution within the search space and assigns it as the best solution so far. Then it randomly tries other solutions in the space which are better and can replace the best solution so far. Whenever there is a better solution to replace the current best solution, the algorithm will restrict the part of the search space in which it looks for solutions. The best approach for achieving the global maxima or minima is to lengthen the time between replacements of the current best solution, allowing the search of a larger problem space before the solution converges too quickly. The same applies to GAs as well, but GAs contain a population of chromosomes to represent different solutions, together with the

crossover and mutation operations, so GAs always retain a possibility of reaching the global maxima and minima. At the same time, with a large population of chromosomes, the problem space can be explored faster, causing a shorter convergence time than for simulated annealing.

2.4.2 Previous work on GAs and testing

Guo et al.[8] applied GAs to the problem of generating test data based on finite state machine models of the software under test. Finite state machines have characteristic unique Input/Output sequences (UIOs) and it is known that computing UIOs is NP-hard. Guo et al. have found that using GAs to construct UIOs outperforms using random search to do so; and with the help of these UIOs, it can be determined whether the I/O behavior of a particular finite state machine conforms to another given finite state machine, thus accomplishing a black box test on the system. Guo et al. define a "do not care" value when mutation occurs in the genes, which can help improve the diversity of the chromosomes in the GA.

Michael et al. [11] and Berndt et al. [4] applied GAs to generate data for numeric input parameters. Michael et al. developed a test data generator known as GADGET (the Genetic Algorithm Data GEneration Tool), designed to work on large programs written in C and C++. GADGET uses condition-decision coverage as a criterion for the GA evaluation process. It tries to cover as many branches as possible within a given program. This is possible since GADGET maintains a coverage table and helps to determine whether extra input data generation is needed. Michael et al. compare

their findings on GAs to that of gradient descent, which aims to achieve local maxima and minima. They have found out that using GA to generate test data achieves higher coverage than gradient descent and random search.

Berndt et al. use GAs to help develop test cases that can help uncover as many faults as possible. The distinguishing idea is the dynamic changes on the fitness function based on historic results of the chromosomes. The triangle program written in Java (a standard small example in the software testing literature) is used for analysis and only numeric test data is considered since Berndt et al. focus on testing boundary conditions on the triangles.

This thesis focuses on two main problems; one is minimizing failing test cases, the other is finding optimal input values for randomized test case generation. In terms of minimizing a test case, we have a different focus with the previous research on software testing with GAs. Rather than generating a test case or set of test cases, we try to minimize a test case if it is found to reveal faults in a unit. We have developed an application based on Java, and we have defined a static fitness function to evaluate the goodness of a solution. The development of TestFragmentInfo by the tester can achieve conformance testing based on a comparison between expected and actual variable values, instead of comparison between two finite state machines.

In terms of finding optimal input values for randomized test case generation, we have developed an application in Java that has a similar scope as that of Michael et al. Instead of using condition-decision as a evaluation criteria, we have used line coverage

for chromosome evaluation in GA since this is the coverage criterion available to us in the coverage tool we used. While the other research attempts to develop single test cases for covering specific coverage elements using GAs, our presented solution aims to provide a better environment for randomized unit test case generation using GAs. While analyzing the effectiveness of our solutions, we have evaluated three units from the Java standard library which are widely used in the public. The units that were chosen to participate in the analysis process vary in their data structures, methods and their intended usages. We believe that using the three units under test gives better support for our findings and conclusions.

2.4.3 JDEAL

JDEAL [6] is a Java language library of Evolutionary Algorithms. GAs are a particular class of Evolutionary Algorithms. Using JDEAL, programmers can reuse and extend the library components to fit their own purposes. The library provides a flexible environment to adjust variables within the GAs, such as population size, crossover and mutation rate, and define the fitness function for evaluating the chromosomes. JDEAL also provides a default value that the developers have found to be most effective for some of these variables, if the programmer prefers not to adjust these values. JDEAL has facilitated our research and development and we have found it to be very useful.

2.5 Failing Test Case Minimization

In randomized unit testing and in other situations, failing test case inputs can be of various lengths. Generally, a longer failing test case is less useful for debugging than a shorter one, because a longer test case generally excutes more code that is irrelevant to the actual fault. Therefore it is desirable to try to decrease the size of a failing test case as much as possible. Recently, research has explored automatic techniques for doing this.

Zeller and Hildebrandt [14] defined the Delta Debugging Algorithm which deterministically minimizes a given failing test case. The algorithm applies an approach that is similar to the divide and conquer method, in which the test case will be divided into smaller granules depending on the failing or passing of the previous larger granule. The algorithm involves four major operations, test granule, test complement granule, increase granularity and reduce test case.

An example of minimizing a test case is shown in Figure 2.2. Assume we have a test case of length 8. The algorithm starts with the original failing test case assigned to be the first granule. The granule is tested and fails and it is then divided equally into two granules. The granules are tested (steps 1-2) and are not able to recreate the failure, so the granules are divided equally into half again, increasing granularity. Once again, the granules and their complements will be tested. None of the smaller granules causes the software to fail, so the complement granules (test cases formed by leaving out a granule) are tested (steps 7-8). The second complement (step 8) causes the software to fail, so we have found a smaller failing test case.

The algorithm proceeds by continuing to test granules and complements at granule size 2, sometimes finding a shorter failing test case, until no granule or complement fails. It then increases the granularity (reduces the granule size) and repeats. When no granule or complement fails and the granule size is 1, the latest failing test case is returned.

Yong and Andrews [10] have studied Zeller and Hildebrandt's [14] test case minimization algorithm in the context of randomized unit testing. The study has shown that the algorithm provides an efficient way of reducing irrelevant method calls within a failing test case. The results have shown an average reduction of 71% to 93% in the length of the failing test cases.

While the scope of the problem we are solving is similar to that of Zeller and Hildebrant, we have used a different approach for minimizing failing test cases. We use GAs to minimize test cases non-deterministically, which means that the resulting minimized test case may differ each time. Zeller and Hildebrandt's algorithm is deterministic and attempts to divide test cases systematically until a minimized test case is found; thus every run will result in the same minimized test case. At the same time, since GAs perform crossover and mutation to achieve wider varieties of solutions and maintain a diversified solution space, they have a better chance to find global minima, although that can not be guaranteed. Using GAs to solve this problem also has the possibility of finding more than one local minimum.

Figure 2.2 An example of minimizing a test case using the Delta Debugging algorithm

Step	Test Case								Result	Description
1	1	2	3	4	.	.	.	.	unknown	Testing and Testing Complement
2	.	.	.	.	5	6	7	8	unknown	Increase granularity
3	1	2	.	.	.	.	.	.	unknown	Testing
4	.	.	3	4	.	.	.	.	pass	Testing
5	.	.	.	.	5	6	.	.	pass	Testing
6	.	.	.	.	.	.	7	8	unknown	Testing
7	.	.	3	4	5	6	7	8	unknown	Testing Complement
8	1	2	.	.	5	6	7	8	fail	Testing Complement and Reduce
9	1	2	.	.	.	.	.	.	unknown	Testing
10	.	.	.	.	5	6	.	.	pass	Testing
11	.	.	.	.	.	.	7	8	unknown	Testing
12	.	.	.	.	5	6	7	8	unknown	Testing Complement
13	1	2	.	.	.	.	7	8	fail	Testing Complement and Reduce
14	1	2	.	.	.	.	.	.	unknown	Testing
15	.	.	.	.	.	.	7	8	unknown	Increase granularity
16	1	.	.	.	.	.	.	.	unknown	Testing
17	.	2	.	.	.	.	.	.	pass	Testing
18	.	.	.	.	.	.	7	.	unknown	Testing
19	.	2	.	.	.	.	.	8	unknown	Testing
20	.	2	.	.	.	.	7	8	unknown	Testing Complement
21	1	.	.	.	.	.	7	8	fail	Testing Complement and Reduce
22	1	.	.	.	.	.	7	8	unknown	Testing
23	.	.	.	.	.	.	7	.	unknown	Testing
24	.	.	.	.	.	.	.	8	unknown	Testing
25	.	.	.	.	.	.	7	8	unknown	Testing Complement
26	1	.	.	.	.	.	.	8	unknown	Testing Complement
27	1	.	.	.	.	.	7	.	unknown	Testing Complement
	1	.	.	.	.	.	7	8		Finished

Chapter 3

Test Case Minimization

In this chapter, we look into details about using genetic algorithms to help minimize failing test cases. With a minimized failing test case, a tester can efficiently identify the problematic sequence of method calls that generates faulty behaviour from the code. We describe the design and implementation of our GA solution, and empirically compare that solution with another existing failing test case minimization algorithm, using an experiment.

3.1 Background and Motivation

As mentioned in the Introduction section, randomized unit testing often generates lengthy failing test cases, making it difficult for a software tester to trace the fault. Although calling the very last method produces the failure in the test case, it does not

necessarily mean that the last method contains the fault contributing to the failure. It can be equally likely that the previous sequence of method calls have been incorrectly implemented, but only the last method call makes the fault apparent. There can be one erroneous method call but there can also be more than one erroneous method call within the failing test case. The longer the test case is, simply analyzing the entire failing test case and looking for the root of the problem is going to be a more time consuming process. Being able to minimize the failing test cases will make a tester's life easier; at the same time, using a GA to minimize a failing test case, we expect to give testers more information about the fault(s) than any deterministic algorithm can achieve. This is made possible since a GA works in a probabilistic way and has the ability to reach global minima.

3.2 Algorithm Design

In order to utilize GAs in solving the failing test case minimization problem, we encode the problem using GA terminology. With a given failing test case, each method call is represented by a single gene and, thus, a test case is represented by a chromosome. A simple bit string chromosome is used; that is, each gene is a single bit, corresponding to a method call of the original test case, and identifies whether that method call should appear in the minimized test case (Figure 3.1).

For example, suppose TreeMap (See Section 2.3.2) is the unit that we would like to test, and the methods that we are interested in testing are size(), put(int, int) and

Figure 3.1 Representation of method calls in an original failing test case using a chromosome.

Original failing test case	Chromosome	Reduced test case
treemap.size()	0	
treemap.put(3, 10)	1	treemap.put(3, 10)
treemap.put(5, 7)	1	treemap.put(5, 7)
treemap.getFirst()	0	
treemap.put(20, 30)	1	treemap.put(20, 30)

getFirst(). Suppose further that our failing test case is the one shown in column 1 of Figure 3.1. During the process of running GAs, a chromosome might become the sequence of bits represented in the middle column. With that particular chromosome, the actual sequence of method calls represented would be the test case shown in the right hand column. The new sequence of method calls represented by the chromosome might not cause a failure, and might not be the smallest possible; the overall GA handles this by its evaluation of the chromosomes.

3.2.1 Initial Population

As we have introduced in Chapter 1, a GA requires initializing a population of chromosomes before running. We initialize each chromosome in the entire population to contain a value of 1 in all genes (i.e. all methods will be called). By using this approach, we can start the minimization process from the original failing test case provided by the tester, and together with crossover and mutation operations, the possibility of reducing the length of the failing test case is increased. We considered randomly initializing the value in each gene to either 1 or 0 in the entire chromosome

population. However, doing so would require a random search for a failing test case before it can be minimized and the process would become more inefficient. The initial population of chromosomes has a size of 400 instead of the JDEAL default value of 200 because we would like to see a set of more diversified results at the end of running the GA.

3.2.2 Evaluation

As we have discussed in chapter 1, it is necessary to define a fitness function for the evaluation of chromosomes when running GAs. The fitness function will calculate a score for each chromosome, which represents the goodness of the chromosome and its associated solution to the problem. The fitness function that we have used gives a score of 0 for a chromosome which represents a non-failing test case, and 1000/(number of 1s in the chromosome) for a chromosome which represents a failing test case. Using this fitness function, we encourage shorter test case length, but only for the test cases which fail.

In order to determine whether the test case represented by a chromosome fails, the TCRunner class from the Rute-J implementation is used to automate the testing process for each chromosome to achieve this goal. The class contains a method called runOld() which runs a saved test case and checks if it fails. By reading each gene value in a given chromosome, we are able to recreate a test case object to be given to TCRunner. TCRunner will then respond either true, indicating that the test case passes, or false, indicating that the test case fails. The resulting boolean value can

then help in calculating the fitness function that we have defined earlier.

It is important to note that, since the evaluation process for each chromosome involves creating a test case and feeding it into the TCRunner class and finally calculating the fitness score, the evaluation process contributes the most to the overall running time of the GA compared to the rest of the processes such as initialization, selection, crossover and mutation.

3.2.3 Crossover and Mutation

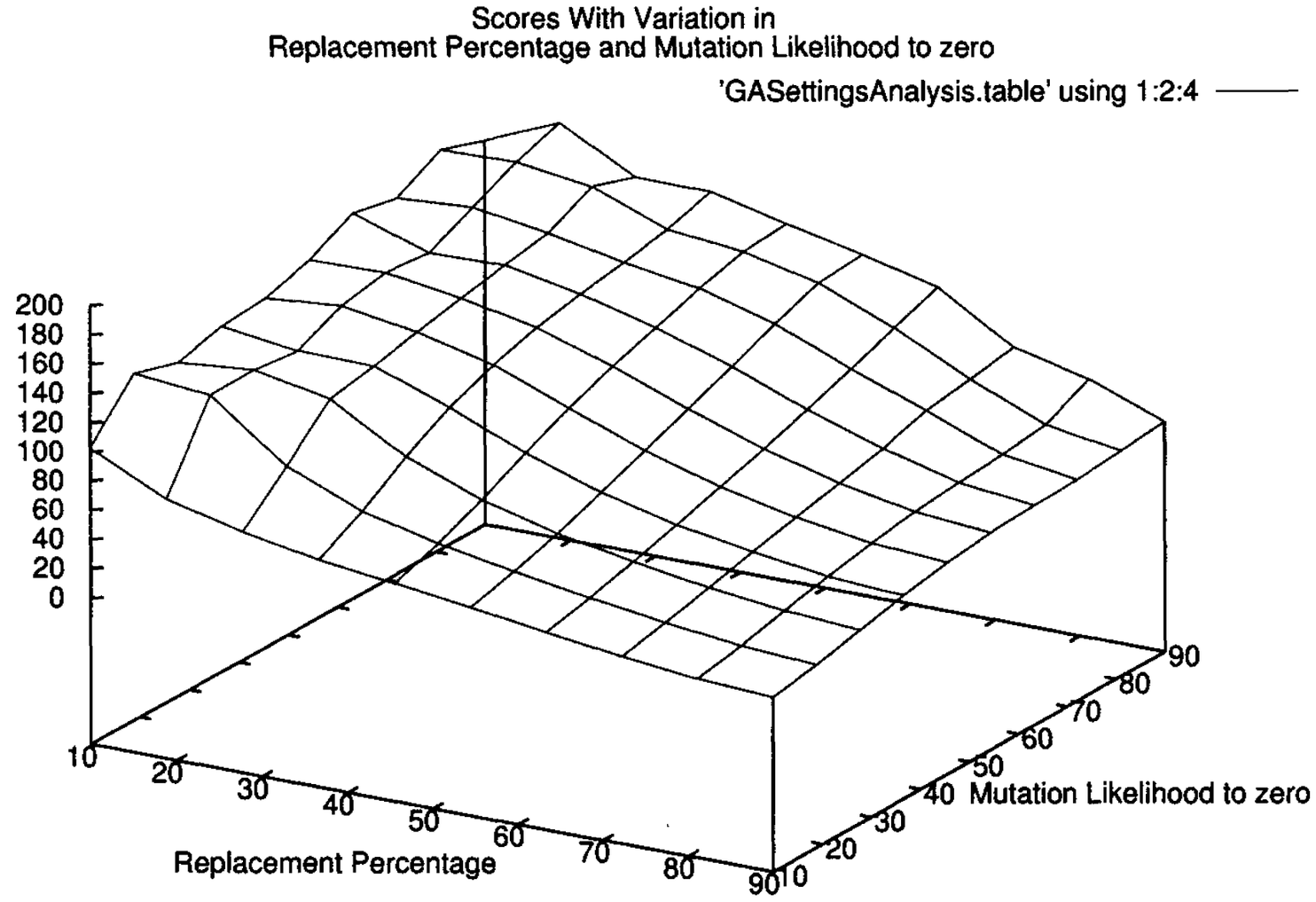
Crossover is the switching of a certain part of a sequence of genes between two parent chromosomes that would result in two new child chromosomes. The parents are chosen because of their high fitness value, meaning that they represent comparatively shorter failing test cases. The length of the child chromosomes will still remain the same as their parents', but they have a high probability of representing even shorter failing test cases than their parents since they inherit the goodness from both parents. We have set the crossover rate at 0.9, the default crossover rate given by JDEAL, which means that there is a 90% chance that the selected parents will perform the crossover. If it happens that the parents fall into the remaining 10%, their children will be the exact clones of their corresponding parents, and remain in the population for the next evaluation.

Mutations that happen in each evolution in the GA as well will assign randomly a 1

or 0 to the target gene. The mutation operation can be carried out by only one single chromosome. The mutation rate is set to be 0.01 as given by JDEAL, so that each gene in the selected chromosome will have a 1% chance to mutate. That being said, we have also created a new mutation strategy for solving our problem in an efficient manner. Doing so gives us the ability to assign the frequency of a gene to mutate to a 0 if a mutation is going to occur. By assigning 0.8 to the rate of mutation likelihood to zero, we are saying that if a mutation is going to occur to a particular gene in a chromosome, there is an 80% chance that this gene will contain a value of 0 after the mutation operation; and obviously, there is a 20% chance that it will turn to 1 after the mutation occurs. The reason for doing so is that we are trying to shorten the given failing test case, so there is definitely a higher need of shortening the test case by assigning more 0s than lengthening it by assigning more 1s to a chromosome; and at the same time, we do not want to entirely ignore the advantage of mutating a gene into 1, since by doing so, we maintain a possibility of finding the global minimal solution, which can be an even shorter failing test case.

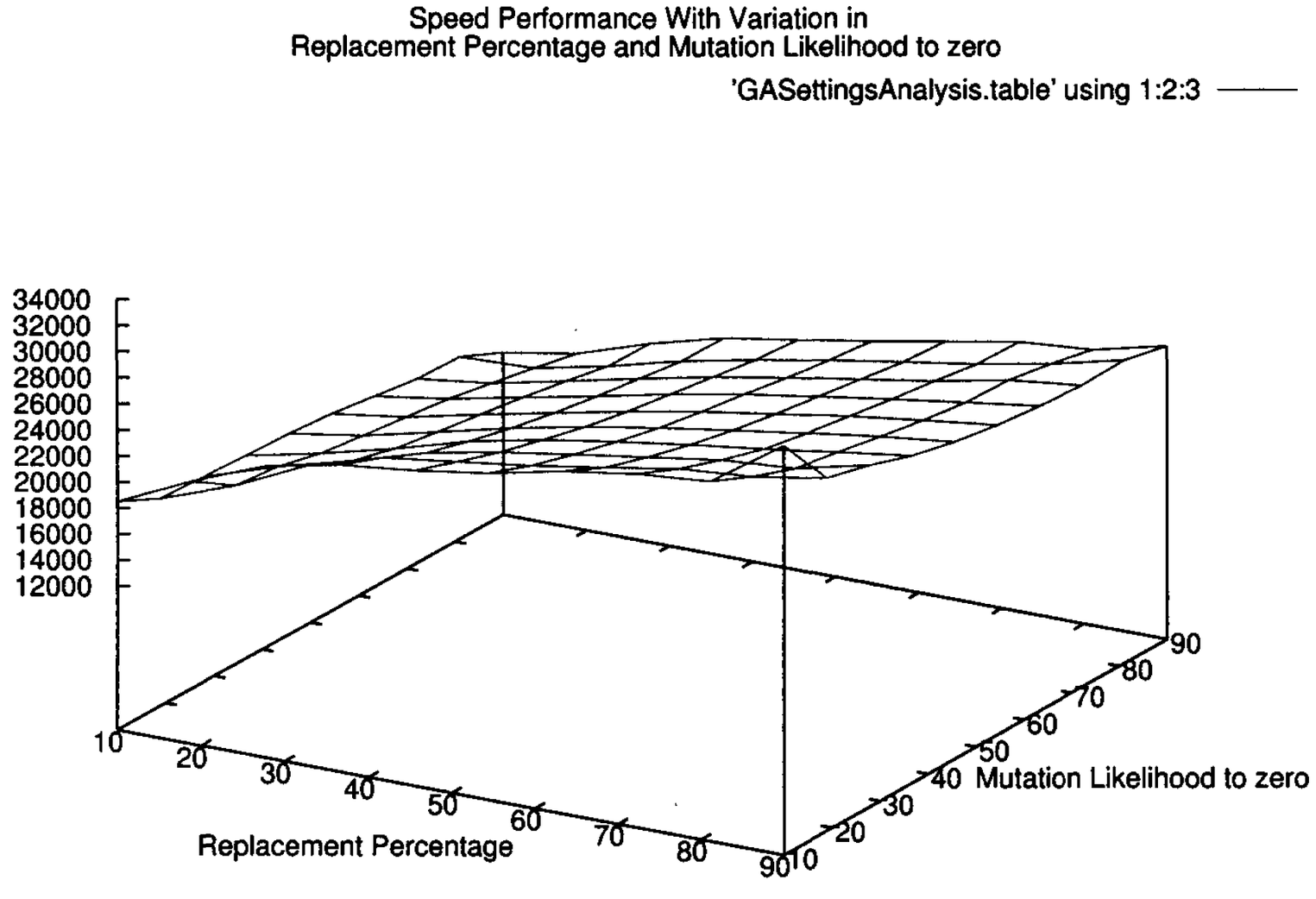
To find our chosen 0.8 rate of mutation likelihood to zero, we performed a detailed analysis. We believe that in addition to the impact that the rate of mutation likelihood to zero has on the qualities of the results generated, the replacement percentage has an impact as well. The replacement percentage determines the percentage of chromosomes that are to be carried on to the next evolution. We paired up the two variables and executed the GA application with 81 combinations of them, ranging from 10% to 90% in a step of 10% each. We ran GA using each combination of the two variables once and measured the effectiveness and efficiency of each combination based on the highest fitness score achieved in the evaluation process and the overall time taken.

Figure 3.2 Scores distribution with different combinations of replacement percentage and mutation likelihood to zero



As shown in Figure 3.2, we have a reason to believe that 80% is the optimal rate of mutation likelihood to zero which will give us the most efficient result when solving the test case minimization problem. The replacement percentage seems to work the best at 25% for the problem. In terms of overall time efficiency for running the algorithm (Figure 3.3), the result does not display much variation among different combinations of the two variables.

Figure 3.3 Speed distribution in milliseconds with different combinations of replacement percentage and mutation likelihood to zero



3.2.4 Running the Algorithm

During the course of running GAs to solve our problem, the process will go through a series of evolutions. Each evolution will have a higher possibility of finding some shorter test cases that still cause failures using the mutation and crossover operations. In our studies, we have set the GA process to continue for 10000 evolutions; based on numerous executions of the GA on different failing test cases, we have found that most minimized test cases converge before 10000 evolutions. The chromosome with the best score at the end of 10000 evolutions represents the minimized test case.

3.3 Experiment Design

We have paired up the two failing test case minimization strategies, our GA and Zeller and Hildebrandt's algorithm, in order to evaluate and compare their effectiveness and efficiency. There are two important result variables that we are interested in obtaining from the experiment: the length of the minimized failing test cases obtained by the two algorithms given the same failing test case, and the time required for running the two algorithms given the same failing test case. Having obtained these data, we can perform a paired *t*-test statistical analysis to see if there is a significant difference between the two algorithms in terms of the length of the minimized test case and the time required to obtain it.

3.3.1 Subject Unit Preparation

Before we can start generating the necessary data for analysis, we have to prepare TestFragmentCollections for some subject units in order to run the application. We have chosen three subject units for the experiment: the TreeMap unit, the HashMap unit and the BitSet unit. These units are chosen from the Java standard library `java.util` and have been widely used in real life practices. The three units vary in their implementations and the complexities of their data structuring algorithms.

Interestingly, in the course of creating the TestFragmentCollection for the three subject units, we were able to discover a fault within the BitSet unit. The unit is an implementation of a vector of bits. These bits contain boolean values and can be set

or cleared. The method set(int, int) is implemented to set a range of bits, bounded by the parameters, to "true". The two parameters represent the starting index and the ending index of the range of bits to be set. The parameters should both be non-negative, and the ending index should be greater than the starting index for any bits to actually be set. There were several special cases that the method was able to handle. It was able to handle gracefully when the integer parameters are negative; it was also able to handle the situation where the range has a smaller ending index than the starting index; unfortunately, it was not able to handle the situation where the starting index and the ending index are the same. We later found that this defect had been reported and appeared on Java's defect database for BitSet. Though the defect has been fixed in a later version, the version of BitSet that we were using still contained the problem. As a result, we had to fix the error before moving on to the next step of our experiment setup.

3.3.2 Mutant Generation

After the TestFragmentCollection for the subjects were ready, we generated a set of mutants corresponding to each subject. Mutants are variations of the original source code such that a certain part of the code is being altered intentionally. For instance, by replacing "<" by "<=" or replacing a constant with 0. Most mutants are expected to be faulty, i.e. to fail on some test cases. We need to generate mutants in order to generate failing test cases using RUTE-J.

Andrews [2] had previously generated mutants of TreeMap. Using Andrews' muta-

tion generator, we successfully created mutants for the HashMap unit and the BitSet unit and obtained compilable mutants of TreeMap from the development of RUTE-J. There are a total of 174 compilable mutants for the TreeMap unit; there are 513 mutants generated for the HashMap unit and 358 of them are compilable; and within 2640 mutants for the BitSet unit, 2429 are compilable. We only consider compilable mutants in our experiment since Java will not be able to execute any of the source code if it is not yet compiled.

After obtaining all compilable mutants for the three subject units, we use RUTE-J to help us in generating a failing test case for each mutant if possible, since not every mutant necessarily fails and we are only interested in those that fail.

3.3.3 Test Case Minimization

After generating a failing test case for a mutant, we immediately minimize it using Zeller and Hildebrandt's algorithm. We record the minimized test case and its length, and move on to minimize the same failing test case using GAs. Since GAs can produce different solutions in different runs, we perform GA minimization on the failing test case for 10 runs; on the other hand, it is not necessary to perform Zeller and Hildebrandt's algorithm 10 times as well, since the algorithm is deterministic and will give the same solution for every run. We also record the minimization time required for the two algorithms so that we can evaluate whether there is a significant difference in the efficiency of the algorithms.

We encountered certain difficulties while automating the script to run the experiment. Sometimes, when a minimized failing test case is converging on a very short length, say a length of 1 or 2, GA minimization has a high possibility to turn the 1s in the chromosome into 0s by mutation or crossover. This is reasonable since the resulting chromosome may be evaluated with a higher score. However, this caused errors when we tried to execute empty test cases using TCRunner. With this given situation, we therefore give a score of 0 immediately to chromosomes with all 0s without executing them in TCRunner.

The most difficult problem that we encountered in running the script is infinite loops in three possible places. Infinite loops can occur while automatically generating failing test cases, and they can occur while minimizing the test cases, using either Zeller and Hildebrandt's algorithm or the GA. The infinite loops are not the result of a bug in our algorithms, but rather of the mutants. Some mutants fail by crashing or giving incorrect output, and some fail by going into infinite loops.

Although there is a capability for RUTE-J to automatically kill off infinite loops when running Zeller and Hildebrandt's algorithm, it increases the minimization time tremendously by using an extra thread to kill off the loops. It is not feasible to do so since it results in an unfair comparison of the time required to minimize test cases. As a result, we have used some shell scripts that can help to keep track of the time used by the three processes. When a process exceeds a certain time limit, it will be killed and the process will re-run until it does not encounter an infinite loop. There were a small number of mutants that continuously cause infinite looping when generating a failing test case; since the number of those mutants is small, we did not include

them in the final results. For each of the other mutants, we were able to obtain a failing test case, a minimized test case using Zeller and Hildebrandt's algorithm, and 10 minimized test cases using GAs.

3.4 Results and Analysis

After retrieving all the results, we performed a paired t-test analysis on the length of the minimized test case obtained by running GA and running Zeller's algorithm. A t-test assesses whether the means of two groups are statistically different from each other. The null hypothesis was to say that the average lengths of the minimized test cases obtained by running the two algorithms are the same. As shown in Figure 3.4, the second and third column are the averages for the minimized test case length; column 4 is the p-value (significance) resulting from the t-test, where $p<0.10$ indicates moderate support for rejecting the null hypothesis, and $p<0.05$ indicates strong support. Boldface numbers indicate any averages that are statistically significantly lower. For the HashMap and TreeMap subject units, we were unable to obtain any significant differences between minimizing test cases using GA or Zeller's algorithm. But for the BitSet subject unit, the result shows that using Zeller's algorithm to minimize failing test cases obtained significantly shorter test cases than using GA to minimize.

In cases where the test case can be minimized to 1 by Zeller and Hildebrandt's algorithm, the GA cannot do better than that, but the mutants resulting in such failing

test cases might be criticized as unrealistic. To account for this, we also did a paired t-test analysis by using only mutants where the minimized test case length was greater than one. If the minimized failing test case obtained by running Zeller's algorithm is one, there is really no other way to generate a shorter failing test case. Interestingly, the result shows that the TreeMap subject unit, using the GA to minimize failing test cases gave us a significantly shorter failing test cases than using Zeller's algorithm, as shown in Figure 3.5.

As shown in Figure 3.6 and as expected, using GA to minimize failing test case takes significantly more time than using Zeller and Hildebrandt's algorithm. The most important contribution to the length of time taken using GA minimization is the evaluation process. It involves creating a new test case, executing it using RUTE-J, and finally calculating the fitness score. Moreover, the evaluation is performed on all chromosomes in every evolution. Although the evaluation time for the chromosome tends to decrease when the test case lengths become shorter, GA minimization still takes a significantly longer time than Zeller and Hildebrandt's algorithm as a whole.

From the result, it is difficult to conclude that GA minimization algorithm will achieve a significantly shorter minimized test case than Zeller and Hildebrandt's algorithm. There is a reason to believe that different levels of code complexity might affect the result, and the code complexity of the TreeMap unit might be more favourable in running GA minimization of test cases. To support the theory that the complexity of the code does have a direct effect on the length of the minimized test cases, it is likely that we will need to perform a more in-depth study and acquire a wider range of subject units for evaluation.

Figure 3.4 Length of minimized test case - All test cases

	ZH	GA	p-value
BitSet	**2.0850**	2.0934	<0.0001
HashMap	1.8246	1.8246	∞*
TreeMap	3.7874	3.7649	0.2484

Figure 3.5 Length of minimized test case - only test cases minimized to size > 1 by Zeller and Hildebrandt's algorithm

	ZH	GA	p-value
BitSet	**2.2508**	2.2636	<0.0001
HashMap	1.8246	1.8246	∞*
TreeMap	4.1090	**4.0603**	0.0184

At the same time, minimizing failing test cases using GA gives us more information on certain occasions. During the course of performing the analysis, longer failing test cases tend to have different minimized test cases in several runs. That is reasonable, since longer failing test cases contain more combinations of TestFragment sequences which possibly identify a fault that can be caused by different sequences of method calls, or multiple faults embedded within the code. In either of the cases, GA minimization can possibly give testers more information about faults within a failing test case. Figure 3.7 shows the percentage of failing test cases which the GA minimized to more than one test fragment for the three subject units.

* All results were identical for the subject unit.

Figure 3.6 Time taken to minimize - All test cases

	ZH	GA	p-value
BitSet	**1005.5585**	7289.4523	0
HashMap	**1006.1930**	4028.3175	<0.0001
TreeMap	**1029.9357**	6688.0485	<0.0001

Figure 3.7 Percentage of test cases minimized to more than 1 test fragment using GAs

	Number of test cases	Number of test cases yielding >1 minimized test cases	corresponding %
BitSet	1486	85	5.72
HashMap	57	0	0
TreeMap	174	43	24.71

Chapter 4

Finding Optimal Values for Randomized Test Case Generation

In this chapter, we shall examine the problem of finding optimal values for randomized test case generation. With the ability to obtain these values, testers can develop an insight into the relative importance of the methods to be tested. Testers can apply this knowledge to efficiently generate test cases that pose a higher probability of finding any existing bugs. We will explain the design of the solution and evaluate the overall design through experiments.

4.1 Background and Motivation

Randomized unit testing can be an efficient and effective way of discovering a bug if it exists. To randomly generate test cases, the tester is required to provide some important variables such as the length of the test case to be generated, the number of times to generate a test case, and the frequency of each method call to be selected within a test case. The weights of the method calls are a critical factor contributing to the thoroughness of a test; this is because there are methods within a unit that are comparatively more important than the others.

For example, for a given data structure, methods for adding elements into the data structure may be more important than checking the size of the structure; this may be because the adding methods execute more code and have a greater number of possible paths through the code. At the same time, methods for adding an element to a queue may be considered less important than methods for adding an element to a tree structure because the code for adding an element to a queue less complex. As a result, especially for software testers, finding the appropriate weights for the method calls can be a difficult task since methods vary their importance among different program functions and sizes. We believe that we can utilize GAs together with a coverage measurement tool to help testers solve this problem.

4.2 Algorithm Design

The general approach is to evaluate different combinations of test case lengths, number of runs, and method call weights, and check for their corresponding code coverage using GAs. We believe that the higher the code coverage, the more thorough the test case is testing. Although high code coverage might not guarantee thoroughness, it is a good first approximation that can be measured automatically.

Once again, using GAs to solve a problem requires us to encode the problem into GA terminology. Unlike failing test case minimization, we use integer chromosomes to represent the necessary input values for randomized test case generation, namely the length of each test case to be generated, the number of runs (i.e. number of times to generate a test case), and the weights of each method call (i.e. frequency of the methods to be generated within a test case). We will use the first gene of each chromosome to represent the length of the test case to be generated; the second gene will represent the number of runs; finally, the rest of the genes will represent the relative weights of the different method calls in the unit under test. Figure 4.1 shows a simple example of the representation of a chromosome.

The representation of the weights of the method calls takes the form as shown in Figure 4.1 because doing so is compatible with the RUTE-J application. We wanted to utilize some of the functionality in RUTE-J by which a test case can be generated randomly. Because our goal is now high code coverage, we do not require the test case to be evaluated to be passing or failing since it is irrelevant to solving our problem. Therefore, we have created a TCRunnerCov class which contains slight modifications

Figure 4.1 Representation of the length of test case to be generated, number of runs, and method calls' weights using an integer chromosome.

Chromosome number	Chromosome value	Meaning
1	500	Generate a test case with 500 method calls
2	10	Generate 10 of these test cases
3	40	The weight of the first method call is 40
4	80	The weight of the second method call is 80
5	30	The weight of the third method call is 30

to the original RUTE-J TCRunner class. We will describe TCRunnerCov in detail when we talk about evaluating chromosomes using this class.

The weights of method calls in Figure 4.1 can be understood as follows. If we were to generate a total of 150 method calls in a test case (150 = 40 + 80 + 30), 40 of them will be calling the first method; 80 of them will be calling the second method; and 30 of them will be calling the third method. This also implies that the second method will be called twice as often as the first method and indicates that it may be relatively more important to achieving high coverage of the unit being tested; on the other hand, the third method call is relatively less important for the unit under test.

4.2.1 Initial Population

We initialize the chromosome by randomly assigning integer values into each gene. The first gene in each chromosome will contain a randomly generated integer ranging from 1 to 1000, and the second gene will contain a randomly generated integer between 1 to 50. These numbers reflect past experience with randomized unit testing.

We have found that test cases of 1000 method calls are the longest we typically need for testing units of the size that we consider, and that 50 runs is usually enough to expose any faults that exist. The weights will contain any integer between 0 to 1000. Since the weights are relative, the actual upper bound is irrelevant, but we want to leave open the possibility that some method has a very low relative weight.

There is an advantage of randomly assigning a start-off value for each gene over assigning an exact value for every chromosome. With all chromosomes assigning the same integer values in each gene at start off, there is going to be little variance in the result even with crossover and mutation operations. This is because the result would have converged before having an opportunity to explore the entire solution space to look for possible global maxima. On the other hand, assigning all genes in all chromosomes randomly within an appropriate boundary gives a better opportunity for GA to look for possible global maxima. As a result, we have chosen to randomly assign the integer values in all genes within every chromosome.

4.2.2 Evaluation

The evaluation process for the chromosome is the most critical part in solving our problem since we have to make use of a code coverage measurement tool, Cobertura as described in section 1.3.1, together with part of the RUTE-J capability to calculate the fitness scores. We first collect the integer values in a chromosome. Using these values, we feed them into a class called TCRunnerCov that is a slight modification of the RUTE-J TCRunner class. TCRunner is originally used in RUTE-J to randomly generate a sequence of methods and necessary parameters, execute them and evaluate

whether the method calls result in a failure in the test case. With TCRunnerCov, we only require it to randomly generate test cases based on the length of test case required, number of runs and the weights of the method calls, represented by the chromosome. We need not evaluate whether the test case failed or not.

However, TCRunnerCov uses Cobertura to return the total number of lines covered in the source code by executing a particular test case. It does this by calling a static Cobertura method that returns the coverage information, and extracting the relevant information about the class under test. With the coverage information in hand, we calculate the fitness score for evaluation as represented by the following equation.

fitness = (lines of code covered × 1000) - (length of test case to be generated × number of runs)

In this equation, the term (length of test case to be generated × number of runs) represents the total number of method calls executed; as a result, we are encouraging fewer method calls to be executed while still achieving high code coverage. To be precise, we are willing to execute 1000 more method calls to achieve one more line of code to be covered. After evaluation, the coverage information has to be cleared from Cobertura in order to provide accurate coverage information for the next chromosome's evaluation. Within the algorithm, time is mostly used in the evaluation process.

4.2.3 Crossover and Normalization

The crossover operation in this application is similar to the one in test case minimization. Parent chromosomes are chosen based on their fitness score calculated during evaluation. There is a 90% chance that the parents will perform a crossover and generate child chromosomes. But there is one special thing to be noted, as follows.

The problem here is that while the method weights within chromosomes are meaningful relative to each other, they might not be meaningful relative to weights in another chromosome. This is a problem when crossover is being performed on a chosen pair of parents, since the parents contain the relative weights of the method calls, from the third gene to the last gene. These relative weights of the method calls are only "relative" before the crossover is being performed; after the crossover is performed, the child chromosomes will contain weights for the method calls that are not relative to each other anymore.

For example, in Figure 4.2, there are two chromosomes chosen as parents and are ready to perform crossover to generate children chromosomes; one has found out that giving 10 times less weight to the second method call can achieve a better solution, the other believes that giving 10 times less weight to the fourth method call can achieve a better solution. However, the weights in one chromosome are much higher than in the other. If crossover is to be performed between these parents, say by switching their last two genes, it can result in two possible children chromosomes as shown in the figure. Unfortunately, both of the children do not represent a better solution than their parents since they did not inherit the good properties from their parents. One

child chromosome is now considering that the last method call should be called more often than any other methods, which neither of the parents represent; as a result, rather than using the effectiveness of GA to converge on a certain solution, we would be performing something similar to a random search on the optimal values.

4.2.3.1 Normalization of Chromosomes

Because of the problem mentioned above, normalization is needed. Normalization can help us make the parent chromosomes contain weights that are meaningful relative to each other after crossovers. If the weights are relative to each other across the two parents, than the weights for their children will be relative to each other as well. To achieve this, we apply the following equation to re-adjust the gene values representing the weights of method calls before crossover is performed:

gene integer value = (gene integer value $\times$ 1000) / (sum of the weights of all method calls)

Figure 4.3 shows us a pair of possible child chromosomes generated by the same parent chromosomes but where normalization is performed before crossover. With normalization of the parent chromosomes, the child chromosomes can now inherit the properties from their parents: One child chromosome is now considering a possibility that both the second and fourth method calls to be called less often than the others. As a result, normalization has helped to enhance the efficiency and effectiveness of

Figure 4.2 Problems with crossover on method calls' weights when two "good" parent chromosomes produce bad children chromosomes

Parent Chromosomes	20	2	20	20	20
Parent Chromosomes	400	400	400	40	400
Possible Children Chromosomes	20	2	20	40	400
Possible Children Chromosomes	400	400	400	20	20

Figure 4.3 Normalizing the genes representing the method calls weights gives better children chromosomes

Parent Chromosomes	20	2	20	20	20
Parent Chromosomes	400	400	400	40	400
Normalized Parent Chromosomes	243	24	243	243	243
Normalized Parent Chromosomes	243	243	243	24	243
Possible Children Chromosomes	243	24	243	24	243
Possible Children Chromosomes	243	243	243	243	243

the crossover operation in solving our problem.

4.3 Mutation

In terms of the mutation operation, we maintain the default JDEAL mutation rate of 0.01, so that any selected genes will have 1% chance to mutate. But the mutation method is more complex in this problem compared to that for minimizing failing test cases. We are now dealing with a range of integers across each gene; moreover, the genes in different positions represent different input values to the randomized testing process. As a result, we apply different mutation methods to genes in different posi-

Figure 4.4 Various combinations of mutation method for acheiving optimal input values for randomized test case generation

Combination	First Gene	Second Gene	Third Gene to Last Gene
1	+[-200,200]	+[-5,5]	+[-200,200]
2	+[-200,200]	+[-5,5]	+[-500,500]
3	+[-200,200]	+[-10,10]	+[-200,200]
4	+[-200,200]	+[-10,10]	+[-500,500]
5	+[-500,500]	+[-5,5]	+[-200,200]
6	+[-500,500]	+[-5,5]	+[-500,500]
7	+[-500,500]	+[-10,10]	+[-200,200]
8	+[-500,500]	+[-10,10]	+[-500,500]

tions.

We have decided to randomly pick an integer ranging from -500 to 500 and add it to the first gene if it decides to mutate; we randomly pick an integer ranging from -10 to 10 and add it to the second gene if it decides to mutate; and for the rest of the genes representing the weights of the method calls, they will mutate by adding a randomly chosen integer ranging from (-1000 $\div$ number of method calls) to (1000 $\div$ number of method calls). Although these numbers are arbitrary, we did perform some exploratory studies to identify some good values. We performed 10 runs of the GA using each of the 8 combinations of mutation amounts shown in Figure 4.4. The combination we picked (combination 8) was the one that led to the fastest covergence of the GA.

4.3.1 Running the Algorithm

Before starting to run the algorithm, some preparation is needed in order to allow the GA to work properly. Since we are using integer chromosomes, we are required to put an upper and a lower limit on the range of possible values they can take; otherwise, mutation can produce undesired integer values such as negative integers. Although there is no exact boundary that is proven to be the best for solving this problem, there are certain values the genes should not take. For example, the first and the second gene can not contain a value less than zero since it does not make any sense to have a test case with a negative length, or to generate a test case a negative number of times. At the same time, the weights that we are trying to achieve are relative between method calls, and there is little chance that the weights will go above a certain limit. Therefore, we have set the following boundary for the genes after some exploratory research.

- [1,5000] for the first gene
- [1,2000] for the rest of the genes

The upper bounds for the genes act as a security line such that the results generated would be more reasonable; that being said, it is unlikely for the genes to go off the upper bound since the evaluation process of GA takes care of them nicely by driving out unreasonable chromosomes.

During each evolution step taken by running the algorithm, the chromosomes will represent a better combination of the three kinds of input values, namely length of test case to be generated, the number of test cases to generate and the method call weights, such that they will be able to cover more lines in the original source code. The number of evolutions necessary to generate the solution is 5000. Again, we have performed sample runs on various units under test and have concluded that most solutions would have converged before 5000. Needless to say, the number of evolutions needed to converge to a reasonable solution varies across units and the number of method calls involved. The more method calls we are interested in finding the optimal weights of, the longer the chromosomes, hence longer evaluation time, and more crossover and mutation operations. But we have found that 5000 evolutions is usually enough time for the solutions to converge.

4.4 Results of GA on Subject Units

We applied the resulting GA to the units studied in section 3.3.1, namely, the TreeMap unit, the HashMap unit and the BitSet unit. We are interested to see the resulting method weights in response to the execution of our GA. We, therefore, have performed 30 runs of the GA on the three subject units, and have obtained the results as follows.

Figure 4.5 shows a list of TreeMap method calls which have participated in the analysis. The first column lists the name of the method, the second column is the average weight of the corresponding method, and the third column is the standard deviation

Figure 4.5 The averages and standard deviation of the weights of the TreeMap methods

Method	Average weight	Standard Deviation
clear()	18.067	18.358
containsKey(int)	90.267	69.753
containsValue(int)	92.567	60.648
size()	105.933	71.840
firstKey()	111.300	64.512
get(int)	121.300	74.071
lastKey()	125.233	70.537
remove(int)	165.200	51.377
put(int, int)	166.333	59.704

of the weight obtained. The figure shows that the clear() method has a very low average weight compare to the other method calls weight, and at the same time, remove() and put() have a very high average weights. This is reasonable since the clear() method clears all data in the TreeMap data structure. By calling clear(), the TreeMap has to start from scratch and build up a more complex structure, which can possibly help execute some other complex lines of code that require switching and deleting branches within the TreeMap. As a result, the clear() method achieved the lowest average weight. On the other hand, remove() and put() methods are weighted highest in average which is also very reasonable as well. Removing and putting objects into a TreeMap require switching and deleting tree branches, re-calculating and re-structuring the TreeMap structure. As a result, the two methods can help execute complex code that other methods might not be able to do so. Other methods within the TreeMap unit seem to achieve about the same average weights, that is probably because they do not pose a huge effect on the data structure after execution.

The average weights and standard deviations for the HashMap methods are shown in

Figure 4.6 The averages and standard deviation of the weights of the HashMap methods

Method	Average weight	Standard Deviation
clear()	21.367	19.009
containsValue(int)	117.600	78.993
size()	124.600	71.354
remove(int)	125.367	64.312
containsKey(int)	129.733	67.592
isEmpty()	140.567	75.505
get(int)	149.133	91.768
put(int, int)	188.200	57.089

Figure 4.6. Once again, it is obvious that clear() has the lowest average weight among all the methods weights we are interested in analyzing in HashMap, which is possibily caused by the same reason as it is in TreeMap. Since the clear() method clears all data in the structure, this requires the structure to be rebuilt again using other methods such as put() and remove() that can possibly help executing other complex lines of code. Therefore, executing too many repetitions of the clear() method is not desired. The average weight for the put() method is very high as well; this is because in order to put an object into HashMap, it needs to calculate the proper index before adding the actual object, and calculating the proper index is complex and requires the execution of more methods and more lines of code. It is unclear to us why the remove() method does not have a high average weight compare to the rest of the methods. We expected to achieve a higher average weight for remove() since it requires calculation of the proper index and searching for the object to be removed. It is possible that the large gap between the average weight of remove() and put() in HashMap comes from the fact that remove() does not need to take care of the expansion of the hash table size but put() does, making put() weights more than remove().

Figure 4.7 The averages and standard deviation of the weights of the BitSet methods

Method	Average weight	Standard Deviation
size()	52.833	34.059
length()	53.467	32.513
set(int, int)	54.467	35.760
isEmpty()	54.500	37.212
set(int)	55.000	35.639
clear(int)	55.933	29.299
get(int)	56.967	27.408
nextSetBit(int)	61.500	29.044
cardinality()	62.600	31.848
set(int, boolean)	65.533	33.371
set(int, int, boolean)	65.533	31.926
nextClearBit(int)	66.267	28.900
clear()	69.000	30.253
clear(int, int)	69.433	37.733
flip(int, int)	72.333	28.620
flip(int)	76.800	27.290

The result for BitSet is shown in Figure 4.7. BitSet implements a vector of bits that grows as needed. The result as shown in the figure is different from other two sets of result obtained from the TreeMap and HashMap. The average methods weights do not vary much in BitSet. This is the case since BitSet might be a simpler data structure compared to the TreeMap and HashMap structures. Most of the methods in BitSet require setting bits to either a 1 or a 0. Sometimes it requires a range of bits to be set such as methods set(int, int), clear(int, int) and flip(int, int), which might be considered a little more complex if they require an expansion of the vector size as well. Some methods have the same effect on the data structure; if set(int, int, boolean) is setting a range of bits to be false (so the third argument is "false"), then the method will have the same effect as executing the method clear(int, int) if the given ranges are the same. Method flip(int, int) and flip(int) have the highest average

method weights possibly because it needs to check the value of the bits before setting them to their opposite values. Since the effect of the methods on the BitSet structure do not vary much, we believe that is the reason why the average methods weights do not vary much as well.

4.5 Empirical Evaluation of Effectiveness

In this section, we present the results of an experiment done to evaluate the effectiveness of the GA in finding optimal input values.

4.5.1 Motivation

We don't know for sure whether the GA is doing anything really effective. It could be just randomly exploring. In particular, it might be that the test case length and number of runs are the main contributors to the coverage, and the weights are simply random. If this is the case, then the weights that it arrives at might not give any better result than equal weights.

We therefore set up an experiment where the null hypothesis was that there is no difference between the weights arrived at by the GA and equal weights.

4.5.2 Experiment Design

One run of the experiment consisted of the following. We ran the GA on a subject unit, and recorded the length, number of runs, and coverage obtained by the solution chromosome. We then ran RUTE-J on the unit using the length and number of runs obtained from the GA, but using equal weights for all methods. We measured the coverage obtained by executing test cases generated by both the solution chromosome and RUTE-J.

We performed thirty runs for each subject unit, and performed a paired t-test, where the null hypothesis was that the mean coverage from the GA result is the same as that obtained from RUTE-J. The results, summarized in Figure 4.8, show that we can reject the null hypothesis for all units studied with at least 99.99% confidence, indicating that the mean coverage obtained by the solution chromosomes is consistently higher than that obtained by RUTE-J.

4.5.3 Discussion

The results support the conclusion that the GA is doing something effective. It is not only finding an optimal run length and number of runs, but method weights that work with those values in order to achieve high coverage. It also supports the conclusion that the GA is a useful tool that automatically provides something that testers would have to otherwise guess at by trial and error.

Figure 4.8 Coverage achieved by running the GA application with optimal weights applied and running RUTE-J with equal weights applied

	Optimal weights applied in GA	Equal weights applied in RUTE-J	p-value
BitSet	**228**	222.567	<0.0001
HashMap	**112**	74.900	<0.0001
TreeMap	**259**	234.133	<0.0001

Chapter 5

Conclusion

5.1 Conclusion

In this thesis, we have examined closely randomized unit testing and two of the associated problems. We have described the necessity for minimizing failing test cases for randomized unit testing. A GA application has been developed for minimizing failing test cases and the details of the implementation have been discussed. After comparing the effectiveness and efficiency of the GA application with an existing minimization algorithm created by Zeller and Hildebrandt, we have found out that both algorithms are able to achieve significantly shorter average test case length in different situations.

Although we can not conclude that one algorithm can achieve significantly shorter test cases, we can conclude two other points. We can be sure that Zeller and Hildebrandt's algorithm can minimize test cases with a significantly shorter time than

GAs. The speed efficiency for both algorithms has been evaluated by performing the minimization on three subject units of different data structures and code complexity, namely the TreeMap unit, the HashMap unit and the BitSet unit from the Java standard library. By performing several experiments on these units, we can also conclude that the GA is able to achieve different minimized test cases in different runs, but Zeller and Hildebrandt's algorithm gives the same minimized test case every time. The GA has the ability to provide even more minimized test cases when the original failing test case is lengthy. This property of GA can provide testers more information on the fault(s) embedded within the test case, since there can be more than one fault within a failing test case, and there can be more than one way to reveal a single fault.

We have also discussed our research on finding optimal values for randomized test case generation. We have presented our GA application development and discussed certain problems that we have encountered during the implementation. With the help of a coverage tool, the GA application is successful in finding the most appropriate test case length and number of test cases to be generated. More importantly, the optimal weights obtained from the GA has shown to be effective in helping more code to be covered, which greatly helps testers to identify methods that are relatively more important than the others. Our experiment demonstrated that the weights obtained from the GA are not merely just random numbers but, in fact, they play an important role in helping to achieve high coverage.

5.2 Future Work

In this thesis, we have presented a solution to finding optimal input values for randomized test case generation using GAs. With that information handy, software testers can now understand better about the importance of different method calls that they are interested in testing. Testers can also estimate the approximate number of method calls needed to generate in order to test the unit thoroughly.

We are looking forward to seeing improvements be made tn the application. One of the things we can consider is the integration of this application into RUTE-J. At this present time, RUTE-J does not have the functionality to suggest to testers the optimal length of test case to be generated, the number of runs needed and the weights of each method call. With the help of GAs, it is now possible to prompt testers whether they would like to analyze the unit under test before randomly generating test cases.

Obviously, if our application is to be integrated into RUTE-J, there are certainly places that need adjustment. For example, the GA produces a set of optimal values based on the number of lines being covered in the original source code. In terms of the number of method calls needed, covering all lines in the original source code does not necessarily mean that it will be able to find a bug if it exists; but it gives testers an insight of approximately how many method calls needed to cover all lines in the source code, so that the testers can adjust the input values accordingly. For instance, if they would like to test the unit so that all lines of code will be covered approximately 5 times, then they can use the optimal length of test case to be generated and the optimal number of runs needed, and multiply them by 5 before actually feeding them

into RUTE-J. Another way of improving the accuracy of the method calls needed is that we can also consider taking into account stronger coverage measures such as condition coverage in the evaluation process in GAs. This is because using condition coverage can help GAs to cover more possibilities that the code execution paths can take. The length of the test case to be generated and the number of runs needed will increase accordingly, to try to cover all conditions at each evaluation point.

One drawback of our application is that it takes a long time to evaluate the chromosomes, since it needs to execute randomly generated method calls and also retrieve the corresponding line coverage. It is going to be interesting to see whether adjusting GA parameters such as population size, crossover and mutation methods can help in reducing the overall time needed. Further research on this topic is necessary.

References

[1] James H. Andrews. A case study of coverage-checked random data structure testing. *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 316–319, Sep 2004.

[2] James H. Andrews, Susmita Haldar, Yong Lei, and Felix Chun H. Li. Randomized unit testing: Tool support and best practices. Technical Report 663, University of Western Ontario, 2006.

[3] Kent Beck. *Test-Driven Development By Example.* Addison-Wesley Publishing Company, Inc., Boston, 2003.

[4] Donald J. Berndt, John W. Fisher, Lee Johnson, J. Pinglikar, and Alison Watkins. Breeding software test cases with genetic algorithms. In *36th Annual Hawaii International Conference on System Sciences (HICSS'03)*, page 338, 2003.

[5] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM Press.

[6] João Costa, Pedro Silva, and Nuno Lopes. JDEAL Java distributed evolutionary algorithms library version 1.0: Getting started. LaSEEB Instituto Superior Técnico, Universidade Técnica de Lisboa, Portugal.

[7] David E. Goldberg. *Genetic Algorithm in Search, Optimization, And Machine Learning.* Addison-Wesley Publishing Company, Inc., 1989.

[8] Qiang Guo, Robert M. Hierons, Mark Harman, and Karnig Derderian. Computing unique input/output sequences using genetic algorithms. *3rd International Workshop on Formal Approaches to Testing of Software (FATES2003)*, 2931:164–177, 2004.

[9] Edward Kit. *Software Testing in the Real World: improving the process.* Addison-Wesley Publishing Company, Inc., 1995.

[10] Yong Lei and James H. Andrews. Minimization of randomized unit test cases. *Proceedings of the 16th International Symposium on Software Reliability Engineering(ISSRE 2005)*, pages 267–276, Nov 2005.

[11] Christoph C. Michael, Gary McGraw, and Michael A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27:1085–1109, Dec 2001.

[12] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[13] Glenford J. Myers. *The Art of Software Testing.* John Wiley and Sons, Feb 1979.

[14] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, Feb 2002.