Monash University

Faculty of Information Technology

Investigating the Effect of Fitness Functions on the Performance of Automated Testing Techniques

---

This thesis is presented in partial fulfilment of the requirements for the degree of Master of Information Technology (Honours) at Monash University

By:
Ohood Dakheelallah Alrohili
23741880

Supervisor:
Dr. Aldeida Aleti

2015

DECLARATION

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the work of others has been acknowledged.

Signed by 

Name: Ohood Dakheelallah Alrohili

Date: 28/11/2015

ACKNOWLEDGEMENTS

I am grateful to my supervisor, Dr. Aldeida Aleti, for her guidance and support throughout this thesis. I am profoundly indebted to her for her useful discussions and invaluable suggestions for the thesis.

I also wish to express my gratitude to my rock in life: my husband, Mr. Mohammed Alahmadi, for his love, support and understanding. I am also thankful to my kids, Lana and Dhiaa, who always tell me to “have fun in exams” – you are my source of encouragement.

Moreover, my deepest thanks go to my family back home for their ongoing support and for always being there for me, with a special mention for my sweethearts, Lamar and Farah. I therefore lovingly dedicate this thesis to my parents, Dad Mr. Dakheelallah Alrohili and Mom Mrs. Abeda Alrohili. I honestly believe that nothing I have achieved would have been possible without your prayers and support; I hope one day my own kids will love me and be as proud of me as I am of you.

Finally, my heartfelt praise goes to God, for His countless blessings.

ABSTRACT

Testing is technically and economically crucial for ensuring software quality. One of the most challenging testing tasks is to create test suites that will reveal potential defects in software. However, as the size and complexity of software systems increase, the task becomes more labour-intensive and manual test data generation becomes infeasible. To address this issue, researchers have proposed different approaches to automate the process of generating test data using optimisation techniques. Due to the non-linear nature of fitness functions and the large search space, one direction of research has focussed on using metaheuristics, which are approximate methods of finding good solutions in a reasonable amount of time.

For many years, researchers have been proposing different definitions of fitness functions for use in the automatic generation of test data, utilising different measurements to evaluate the quality of the test data produced. Investigating the conditions under which a fitness function will either succeed in producing, or fail to produce high software coverage is critical for understanding the suitability and limitations of different fitness function definitions, as well as for automated test data generation. This study examines the performance of a variety of fitness functions across diverse types of software, providing resources from which these conditions may be derived. Experiments show the performance of fitness functions is problem-dependent, such that the coverage obtained by a given function is impacted by the software features being tested. To a large extent, software features which can be captured by software metrics may be seen as an indicator of the possibility of a given fitness function achieving high coverage.

***Keywords**: Fitness function, test data generation, metaheuristics, software metrics*

Table of Contents

List of Figures

List of Tables

Abbreviations and Acronyms

BDF	Distance-oriented fitness function
CBO	Coupling between object classes
CFF	Control-oriented fitness function
CFG	Control flow graph
CK	Chidamber and Kamerer's metrics
CLF	Coverage-oriented fitness function
COM1	Combined fitness function
COM2	Combined fitness function
CYC	McCabe cyclomatic complexity
DIT	Depth of inheritance tree
GA	Genetic algorithm
LCOM	Lack of cohesion of methods
NCL	Number of classes
NOA	Number of attributes
NOC	Number of children
NOM	Number of methods
NSA	Number of static attributes
NSM	Number of static methods
NTG	Number of test goals
OO	Object-oriented
RFC	Response for a class
rs	Spearman’s correlation coefficient
TLOC	Total lines of code
WMC	Weighted methods per class

Chapter 1: INTRODUCTION

1.1 Background

Organisations and individuals worldwide have exponentially increased their reliance on software to do their jobs effectively and efficiently. In turn, the quality of this software will significantly affect the quality of their work. Software quality refers to the degree to which software complies with its functional and non-functional requirements (Alander & Mantere, 2005; Aleti & Grunske, 2014). The functional requirements of an item of software will ensure that it does what it was designed to do and does not do anything unintended (Alander & Mantere, 2005). On the other hand, software's non-functional requirements refer to its usability, reliability, efficiency, portability, maintainability and compatibility (Aleti & Grunske, 2014). It is therefore crucial to ensure software quality, as failure occurring when it is run could lead to serious consequences (Gaikwad & Lodha, 2014). For instance, the reliability of safety-critical systems, such as those used in cars, is vital. Moreover, the cost of software failure was estimated at 0.6 % of the GDP in the US in 2002 (Copeland, 2004).

Software testing is a process of verification and validation performed during the software development process, in order to ensure software quality. Among the most expensive tasks in this process is the generation of test data which will fulfil a testing criterion (Ghani & Clark, 2009), accounting for approximately 40% of the total software development budget (Xiao, El-Attar, Reformat & Miller, 2007). Automating test data generation can significantly reduce the cost of testing, thus decreasing the overall cost of the entire software development process.

A variety of automated test data generation techniques have been developed in the past few decades. Random test data generators (Bird & Munoz, 1983; Thevenod-Fosse & Waeselynck, 1993; Voas, Morell & Miller, 1991) are some of the earliest techniques used. These automatically create random inputs until an acceptable one is found. Since test data is devised at random with no knowledge of the software structure or information on the test requirements being incorporated into the generation process, random test data generators may fail to find test data to satisfy such needs (Pargas, Harrold & Peck, 1999).

Another research direction for the automation of test data generation, which also constitutes the topic of this thesis, has focussed on using metaheuristic search techniques. Metaheuristics

are a family of optimisation algorithms that utilise heuristics in order to find solutions to combinatorial problems at a reasonable computational cost (Dreo & Siarry, 2007; El-Attar, Miller, Reformat & Xiao, 2007). This approach to the testing of automation is called search-based software testing (SBST) (El-Attar et al., 2007). It has been successfully applied to many testing problems, including functional testing (Korel, 1990; Mansour & Salame, 2004), non-functional testing (Nossal & Puschner, 1998), temporal testing (Eyres, Jones, Sthamer & Wegener, 1997) and mutation testing (Harman & Jia, 2008).

Applications of metaheuristics for test data generation have drawn great interest from both the research community and industrial organisations, such as Daimler, Microsoft, Nokia, Ericsson, Motorola, and IBM (Orso & Rothermel, 2014). This interest was motivated by the advantages offered by metaheuristics. Firstly, these are generic methods which are ready for adaptation to any testing problem for which a test criterion can be measured, numerically assessed and transformed into a fitness function (Anand et al., 2013; Lakhotia, McMinn & Harman, 2010). Secondly, metaheuristic search algorithms are able to cope with noise, partial data and inaccurate fitness (Harman, 2010). Furthermore, the search space of test data generation, which is the space of possible inputs into the software, is very large, with the absence of known optimal solutions (Harman & Clark, 2004). Thus, it is applicable to use metaheuristics to seek good solutions and any testing problem for which a fitness function can be defined may be tackled in this way.

1.2 Motivation and Objectives

In an automated test data process, metaheuristics are used to generate test cases that will result in high coverage. Coverage refers to the percentage of code tested (Copeland, 2004). The ultimate aim is to achieve maximum coverage (Gross et al., 2009), as this will lead to a higher probability of finding bugs in the software (Arcuri & Fraser, 2013). Investigations have shown that many faults can be detected when code coverage reaches 100% (Hutchins, Foster, Goradia & Ostrand, 1994). The fitness function utilised will guide the search; in effect, rewarding test data that are close to achieving high coverage. Thus, the effectiveness of metaheuristics in actually covering the code will largely depend on the definition of the fitness function.

For the last two decades, researchers have been proposing different definitions of fitness functions for use in the automation of test data generation, where different measurements are

used to evaluate the quality of the test data produced. These measurements include structural coverage (Roper, 1997); approach level (Pargas, Harrold & Peck, 1999); a distance calculation (Fraser, Arcuri & McMinn, 2013; Shamshiri, Rojas, Fraser & McMinn, 2015), or the combination of more than one measure (Tracey, 2000; Wegener, Baresel & Sthamer, 2001). For each new definition of fitness introduced, an empirical study is conducted to determine which metaheuristic techniques perform best when using the new function, usually based on a limited set of test objects. However, the results from these experimental studies are generally not insightful (Hooker, 1995), as they are restricted to considering just one fitness function and are limited to the type or size of benchmark test objects used. If a study shows the superiority of one definition of the fitness function, it may be claimed there are other definitions which have not been included in the study, but which are able to provide outperformed coverage; or else there are untested objects where we would expect the new fitness function to be outperformed by other functions. Most of the existing comparisons do not describe the conditions under which a fitness function may produce high or low coverage (Smith-Miles & Lopes, 2012).

Two key challenges need to be addressed. The first involves determing which fitness definition is likely to offer the best coverage for a test object benchmark. Useful for this are studies where diverse fitness functions are compared across a reasonable quantity of test objects, with the type of test objects meeting the interests of the study (e.g. embedded or object-oriented (OO) systems, or container classes). A good experimental study will reveal the relationship between the performance of a fitness function and the characteristics of the software being tested (Smith-Miles & Lopes, 2012). The outcome can be an automated fitness function selection model, predicting the coverage obtained from the given function which is likely to be best for a given test object.

The second key challenge to address is determining which types of test object software testers can expect a given fitness function to produce high coverage for and why. Currently, there is a lack of understanding of how the relative performance of different functions depends on the software being tested (Stützle & Fernandes, 2004). This highlights the need to measure features of software and explore their relationship with the coverage achieved through fitness functions. Addressing this question is a valuable means of understanding the strengths and weaknesses of different functions, with implications for an improved definition of these functions.

The abovementioned gaps in existing research have provided the motivation for this thesis, the research objectives are listed below:

- To investigate the influence of different definitions of fitness functions on the performance of the automated testing techniques.

- To gather insights into the relationship between features of software systems and the coverage achieved by fitness functions, for the purposes of understanding the performance of these functions.

Through a comprehensive evaluation, we have shown that the effectiveness of a fitness function is problem-dependent. Features of a test object that can be captured by code metrics may largely be seen as an indicator, in terms of a function's capacity to achieve high coverage.

1.3 Structure of the Thesis

The thesis is organised into the following chapters:

Chapter 2 reviews and analyses work in the field of search-based automatic test data generation.

The research methodology is presented in Chapter 3, which also explains the design of the experiments.

Chapter 4 presents an analysis of the study findings. Illustrated here is how software aspects influence the performance of automated testing techniques.

Chapter 5 summarises and concludes the thesis, with an outline of future research directions.

Chapter 2: BACKGROUND AND RELATED WORK

2.1 Introduction

A software test consists of two components: a test case consisting of a sequence of input values, to be passed to the program upon execution, in order to observe the program's behaviour (Tonella, 2004), together with a definition of the expected outcomes. A test suite is a set of test cases combined for the purposes of test execution. Identifying a test suite that satisfies a selected testing criterion is known as test data generation (Korel, 1990).

This chapter reviews work in the field of search-based automatic test data generation for structural testing, wherein test cases are defined on the basis of their internal program structures. Formulating test cases by using knowledge of the internal program code will increase confidence that software errors can be detected by the tests (Gross, Kruse, Wegener & Vos, 2009). This chapter also identifies current gaps in existing studies.

The rest of the chapter is structured as follows. Section 2.2 briefly explains the basic concepts, while Section 2.3 reviews examples of metaheuristic search techniques. The next section surveys previous studies on the generation of test data using metaheuristics, as well as the results achieved. A discussion on metaheuristic techniques in the topic under study is presented in Section 2.5. Finally, software metrics are investigated in Section 2.6, with Section 2.7 concluding the chapter.

2.2 Basic Concepts

Many structural testing approaches refer to the control flow graph (CFG), first presented by Allen (1970). A CFG is a directed graph for a program under study. For a program, 'F', a CFG is denoted as G = (N; E; s; e), where 'N' is a set of nodes, 'E' is a set of edges, and 's' and 'e' are the unique entry and exit nodes to the graph. Each node in this graph corresponds to a statement in the code and each edge represents a possible transfer of control between two statements. Nodes corresponding to decision statements (such as 'if' or 'while' statements) are referred to as 'branching nodes' and their outgoing edges as 'branches'. Each branch is labelled with a Boolean expression, referred to as a 'predicate', describing the conditions

under which a branch is traversed. For instance, Node 2 in Figure 2.1 is a branching node and its predicate is (i<j).

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

Figure 2.1 a program snip

An *input variable* ‘' of a program ‘P’ is either the variable that appears as an input parameter of a procedure, or in an input statement. This variable could be of a different data type, such as Boolean, integer, real, etc. Given that a vector of program input variables is I= (i_1,i_2,…..,i_n), the domain D_{i1} of the input variable i_1 is a set of all values that i_1 can have. The domain ‘D’ of program ‘P’ is the cross product, $D = D_{i1} \times D_{i2} \times \ldots D_{xn,}$. A *program input* ‘x’ refers to a single point ‘x’ in the *n*-dimensional input space, $x \in D$. Furthermore, a *path* through a CFG is a sequence of ordered nodes. An example of a path is <entry, 1, 2(F), 6, exit> (Figure 2.2). A path is feasible if there is a program input for which the path is traversed during program execution; otherwise, the path is not feasible.

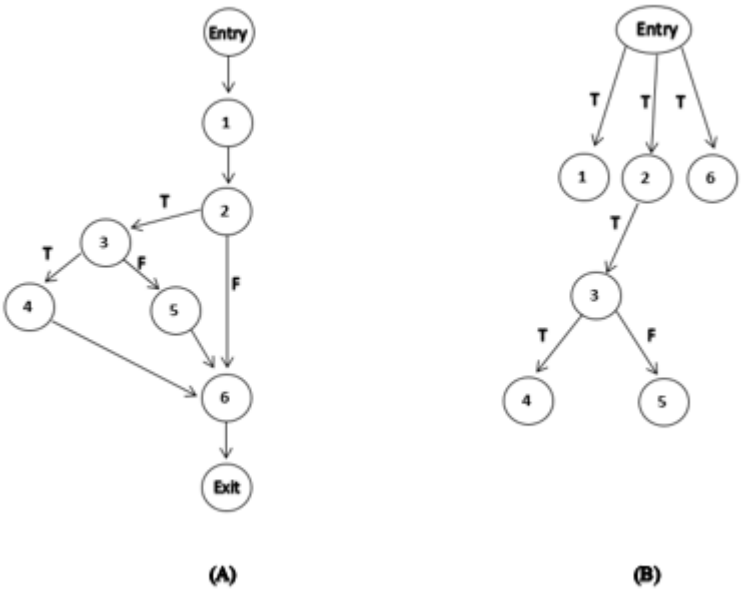


Figure 2.2 the control-flow graph (CFG) of the example shown in Figure 2.1; nodes are labelled with the statement number of the corresponding statement, and (B) its control-dependence graph (CDG) (adapted from Pargas, Harrold & Peck, 1999, p. 267)

A control dependency graph (CDG) is a directed graph used to demonstrate *control dependency* concepts. *Control dependency* describes the reliance of an executed node on the outcome of its previous branching nodes (i.e. an edge (X, Y) means that node 'Y' is control - dependent on node 'X') (Ferrante, Ottenstein & Warren, 1987). For example, in Figure 2.2, Node 3 is control dependent on 2(T). Node 2 itself has no control dependencies, except the entry node. This information can be captured by a CDG. Figure 2.2 shows the CDG for the example shown in Figure 2.1.

In structural testing, after formulating the program as a directed graph, metaheuristics are employed to produce test data which will cover the desired graph elements (nodes, branches or paths).

2.3 Metaheuristic Search Techniques

Metaheuristic search techniques rely upon the concept of a move from one solution to another. These moves depend on the evaluation of candidate solutions, performed using a fitness function (McMinn, 2004). Fitness functions serve to compare and contrast search solutions with regard to the test criteria (Lakhotia et al., 2010). The fitness function is tailored to direct the search into an optimal solution by estimating how close a candidate solution is to satisfying a test criterion (McMinn, 2004). This section briefly reviews three metaheuristic algorithms that have been most widely applied in software testing: hill climbing (HC), simulated annealing (SA) and genetic algorithms (GA). These have been selected because they represent a comprehensive range of different metaheuristic algorithms.

2.3.1 Hill Climbing

Hill climbing is a local search algorithm. It attempts to enhance one solution at a time, with a randomly selected solution as a starting point. The neighbourhood of this candidate point is examined in order to find a better neighbour. If a better candidate is discovered, then it replaces the current solution. This process is repeated until no better neighbours can be found for the current solution.

The hill climbing technique is likened to climbing a hill within a 'landscape' of a function to maximise fitness. Peaks in the landscape signify local optimal fitness values, while troughs reflect solutions with the poorest fitness values. Two different climbing strategies can in fact

be found. In a ‘steepest ascent’ climbing technique, all neighbours are assessed, with the neighbour offering the best improvement being selected to replace the present solution. In the ‘first ascent’, neighbours are randomly investigated, with the first neighbour to offer an improvement being selected.

Hill climbing is easy to implement and can provide quick results. Figure 2.3 gives a description of the algorithm. Nevertheless, this technique can easily become stuck with the local optimal. For instance, the hill climbing technique can lead to a solution at the peak of a hill that is locally, but not necessarily globally, optimal. In this case, the search is trapped at the peak of the hill and will be unable to investigate different areas of the landscape.

Solution space S
Neighbourhood structure N
Fitness function to be maximized *fit*
Select a starting solution $s \in S$
Repeat
    Select $s' \in N(s)$ such that $fit(s') > fit(s)$
    $s \leftarrow s'$
Until $fit(s) \geq fit(s'), \forall s' \in N(s)$

Figure 2.3 High-level description of a hill climbing algorithm (adapted from McMinn, 2004, p. 107)

Moreover, results gained from the hill climbing strategy depend on the initial solution. A typical solution for minimising this issue is to extend the algorithm with a series of ‘restarts’, including diverse initial solutions, in order to sample more of the search space (McMinn, 2004).

2.3.2 Simulated Annealing

Similar to the hill climbing technique, simulated annealing is based on the neighbourhood search idea. However, it is less dependent on the starting solution. The original algorithm proposed by Metropolis, Rosenbluth, Rosenbluth, Teller and Teller (1953) simulated the technique with the chemical process of annealing. Thirty years later, Kirkpatrick (1984) proposed a form of this algorithm as the basis of the search mechanism.

Simulated annealing works by always accepting better solutions and probabilistically accepting worse solutions. In accepting poorer solutions, the search aims to escape from local

optima in the hope of finding better candidates. The probability (P) of accepting the inferior candidate solution under consideration is measured as follows:

$$P = e\text{^}(\ \delta/t) \tag{2.1}$$

where δ is the difference in fitness values between the current solution and the neighbour, and t is a control parameter known as the temperature (of the analogy with the physical annealing procedure).

At the start of a search, the temperature is high to allow for less restricted movement around the search space. It is therefore less dependent on the starting solution. The temperature is gradually decreased as the search progresses. The basic algorithm can be seen in Figure 2.4.

```
Solution space S
Neighbourhood structure N
Number of solutions to consider at each temperature level num solns
Fitness function to be minimised fit
Select a starting solution s ∈ S
Select an initial temperature t > 0
Repeat
        it ← 0
        Repeat
                Select s' ∈ N(s) at random
                If fit (s') −fit (s) < 0
                        s ← s'
                Else
                        Generate a random number r, 0 ≤ r < 1
                        If r < e^(δ/t) Then s ← s'
                End If
                it ←it + 1
        Until it = num solns
        Decrease t according to cooling schedule
until stopping condition is reached
```

Figure 2.4 High-level description of a simulated annealing algorithm (adapted from McMinn, 2004, p. 108)

2.3.3 Genetic Algorithms (GAs)

Genetic algorithms (GAs) use Darwin's evolution principle. Solutions to a problem are known as individuals and the components of the solution are referred to as genes. GAs work

on a population of solutions instead of just one solution at a time, making this a global optimisation technique (Sthamer, 1995). Thus, the search can simultaneously explore the landscape in multiple directions, resulting in better sampling of the search space, when compared to a local search.

The initial population is randomly produced or seeded with pre-selected individuals. Next, the population is iteratively evolved through two primary operators: crossover and mutation, which are used to evolve new offspring. In a one-point crossover, a single crossover point is randomly chosen in two parent solutions. Crossover operations recombine the genes of Parent 1 up to the crossover point and the genes subsequent to the crossover point of Parent 2, in order to form Child 1, and vice versa for Child 2 (as can be seen in Figure 2.5). The mutation operations randomly change the value of a randomly selected gene in the offspring.

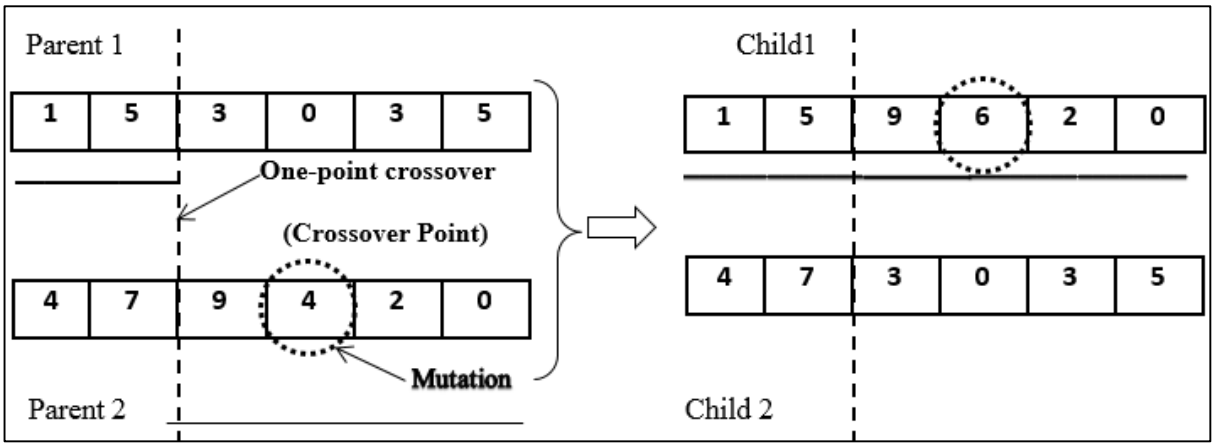


Figure 2.5 Crossover and mutation operations are applied to Parent 1 and Parent 2 to produce new offspring (adapted from Mantere & Alander, 2005, p. 316)

In spite of the randomised nature of the genetic operators, GAs are not a random search; they exploit old knowledge held in a parent population to produce new offspring with enhanced performance. In this manner, the population improves in every generation because it is reproduced by selecting individuals with the best fitness values. Moreover, different selection mechanisms can be utilised to choose individuals which will produce offspring for the next generation. For instance, in the fitness-proportionate selection technique (Holland, 1975), the probability of an individual being chosen for reproduction is proportional to its fitness. In the linear ranking mechanism (Whitley, 1989), individuals are sorted according to their fitness value. As a result, the probability of an individual being selected is then proportional to rank, rather than directly in relation to the respective fitness value.

GAs iteratively perform evaluation, selection and recombination procedures until the search finds an optimal solution or some stopping criterion has been met, such as the number of generations or time. A genetic algorithm is shown in Figure 2.6.

```
Randomly generate or seed initial population P
Repeat
        Evaluate fitness of each individual in P
        Select parents from P according to selection mechanism
        Recombine parents to form new offspring
        Construct new population P' from parents and offspring
        Mutate P'
        P ←P'
until stopping condition is reached
```

Figure 2.6 High-level description of a genetic algorithm (adapted from McMinn, 2004, p. 111)

2.4 Test Data Generation Using Metaheuristic Algorithms

To successfully adapt a metaheuristic technique for generating test data, it is essential to reformulate the latter as a search problem. Following this, one should define the necessary parameters related to the chosen metaheuristic approach, such as fitness functions, representation, and move or genetic operators (Clarke et al., 2003; McMinn, 2004).

For the generation of test data using metaheuristic techniques, one solution is a test case consisting of a sequence of input values, passed to the program upon execution to observe its behaviour (Tonella, 2004). Thus, a set of test cases will form a search space and the representation of a candidate is usually a floating point number and binary code derived from the underlying data types of the programming language used (Harman & Jones, 2001; Harman, Mansouri & Zhang, 2009).

A neighbourhood structure in a candidate solution for local searches constitutes a collection of solutions that are in some respects close to the current candidate solution (Tracey, Clark & Mander, 1998). For numerical variables, such as real and integer variables, the neighbourhood will be within a range of values that surround each individual value. The neighbours of Boolean variables are ‘TRUE’ or ‘FALSE’ values, while the neighbours of enumerated variables will consist of any value from the enumeration (Tracey, Clark & Mander, 1998).

A schematic overview of the most common approaches in the literature is presented in Figure 2.7. In these approaches, test inputs are generated with regard to a test adequacy goal, which is a human input into the process (Harman et al., 2009).

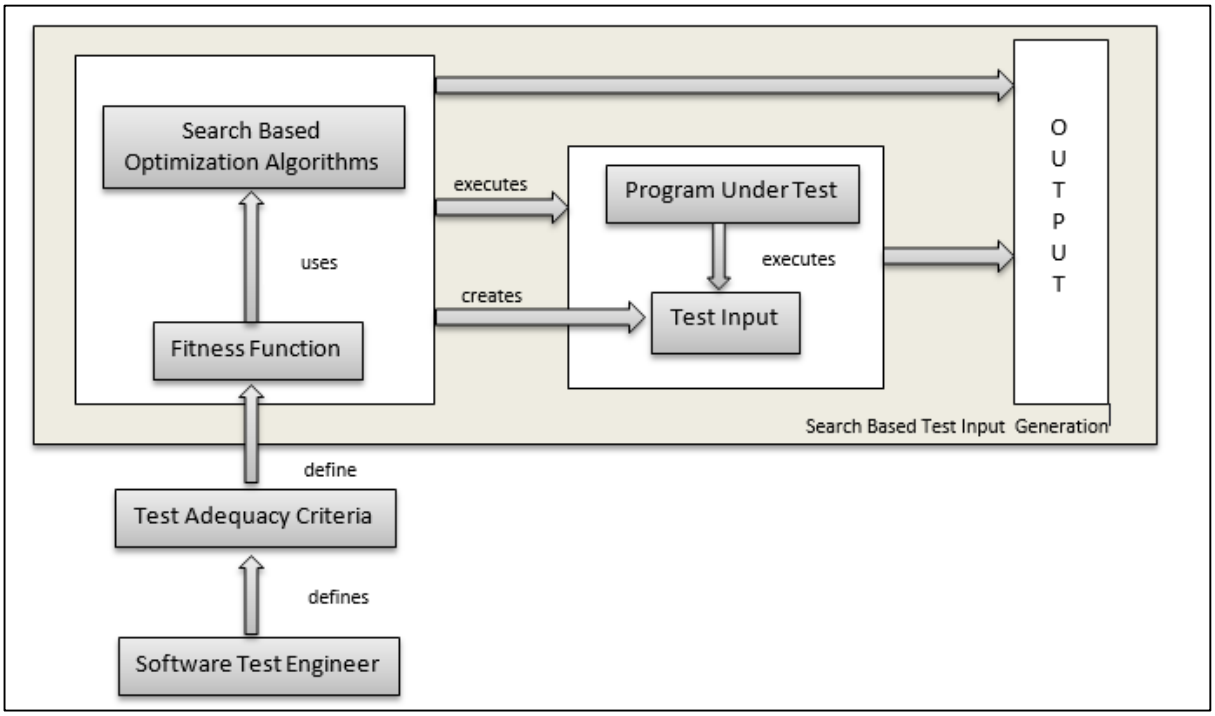


Figure 2.7 Schematic overview of automated test data generation using metaheuristic search techniques (adapted from Harman, Mansouri & Zhang, 2009, p. 10)

A test adequacy criterion for structure testing is a testing aim that can be numerically measured and assessed, e.g. covered branches or statements. The test criterion is coded as a fitness function (Harman, 2007). Once the fitness function is defined for a test criterion, then the generation of test inputs can be automated using search-based software testing tools (Harman et al., 2009).

The fitness function is used to evaluate the performance of candidate test inputs, according to the current optimum. Usually, there is a spread of solutions, ranging in fitness from very poor to good (Watkins & Hufnagel, 2006). In order to assess fitness, the program is executed for the inputs generated, as shown in Figure 2.7. The execution is then monitored to examine the fitness of the inputs and whether they satisfy the test criterion (Harman et al., 2009).

The fitness function plays a vital role in searches conducted using metaheuristic algorithms to test data generation problems. The degree of search success could critically depend on the definition of the fitness of individuals (Baresel, Sthamer & Schmidt, 2002; Harman, 2007; Watkins & Hufnagel, 2006). A well-defined fitness function increases the likelihood of

finding a solution and reaching higher overall code coverage. Better guidance of the search process can lead to optimisation with less iteration, thus consuming fewer system resources in the process (Baresel et al., 2002; Watkins & Hufnagel, 2006).

Plotting the fitness of all possible solutions to a problem will result in a *fitness landscape* that a metaheuristic algorithm will explore to find a good solution (Collet & Rennard, 2007). A fitness landscape is used to visualise a testing problem and its candidate solutions with respect to their perceived fitness. The highest peak in the landscape represents the solution with the highest fitness value, this then being the best solution to the problem in hand.

Previous research into the automation of test data generation using metaheuristic algorithms can be categorised based on the definition of the fitness function optimised in four groups: the goal-oriented approach, the chaining approach, the coverage-oriented approach and the structure-oriented approach. The approaches are summarised in Table 2.1 and discussed in detail in the following sections.

Table 2.1 A summary of test data generation studies using metaheuristics; hill climbing (HC), simulating annealing (SA), and genetic algorithms (GAs)

Article	Search Algorithm			Fitness Function	Test Objects
	HC	SA	GA		
Miller & Spooner (1976)	+			Maximising coverage	From the literature (a matrix factorisation subroutine and sorting method)
Korel (1990)	(AVM)			Branch distance	A simple Pascal program that has array and pointer structure and does not contain procedure calls
Korel (1992)	+			Minimising branch distance (goal-oriented)	Simple Pascal program that has array and pointer structure and does not contain procedure calls
Xanthakis et al. (1992)			+	Minimising branch distance	Pascal programs
Watkins (1995)			+	Path coverage	The popular triangle classification problem and two experimental programs containing no loops
Sthamer (1995)			+	Maximising branch coverage	Four ADA problems (triangle classification, linear search, remainder calculation, and direct sort)

Ferguson & Korel (1996)	+			Improving hard-to-cover predicate branch coverage (a chaining approach)	Pascal problem (only supports the generation of integer test data)
Jones et al. (1996)			+	Branch coverage	Six ADA problems (quadratic equation, triangle classification, linear search, remainder calculation, direct sort, and generic quicksort)
Gallagher & Narasimhan (1997)	+			Maximising coverage (a chaining approach)	Large ADA programs (60,000 lines of code)
Roper (1997)			+	Maximising branch coverage	-
Tracey et al. (1998)		+		Minimising branch distance	Small ADA programs (20 to 200 lines of code)
Tracey, Clark & Mander (1998)		+		Minimising branch distance	A number of small ADA examples
Pargas et al. (1999)			+	Maximising statement and branch coverage (goal-oriented)	Six C-test programs
Tracey (2000)	+	+	+	Controlling dependency information with branch distance	Two industrial systems
Michael et al. (2001)			+	Conditions coverage (a chaining approach)	C++ problem (2,000 lines of code)
Wegener et al. (2001)			+	Approaching level with the branch distance	Seven programs (5 to 154 lines of code)

2.4.1 Early Attempts

Miller and Spooner (1976) were the first to suggest using search algorithms to automate test data generation. The tester selects a path through the program and then produces a straight-line version from the program that is equivalent to that path. The branch predicates in the path are extracted and rearranged into Boolean assignments as a ‘path constraint’ form. A

real-valued function (*f*) is built for the whole path, to estimate how close all the branch predicates on the path are to being satisfied. It is counted as positive when all the branch predicates are true, and negative if the opposite is the case.

Test data are derived by choosing an initial set of data and then applying a numerical optimisation algorithm, attempting to maximise the value of *f*. This is terminated when the value of *f* becomes positive. The function, *f* is assigned by repeatedly executing the straight-line version and automatically collecting the path constraint values.

It was not until 1990 that Miller and Spooner's research directions were continued by Korel (1990). In brief, to execute a particular path, Korel's (1990) approach initially executes the program with some arbitrary inputs. If an undesired branch is taken, a local search algorithm known as the alternating variable method (AVM) (Cooper & Glass, 1965; Gill & Murray, 1974) is used to guide the program execution along the desired path, using a specific fitness function derived from the predicate of the desired branch. This fitness function has a real value, referred to as *branch distance*, which measures how close the predicate is to being executed.

The main weakness of Korel's (1990) method is its limited ability to detect the infeasibility of the path (Ince, 1987; Muchnick & Jones, 1981; DeMillo et al., 1987). If an infeasible path is selected and the infeasibility is not detected, then a significant computational effort could be spent before the search process terminates. The problem of a path's infeasibility means that Korel's (1990) approach is best suited for software featuring a relatively small number of paths to reach the selected node (Korel, 1992).

2.4.2 The Goal-oriented Approach

Based on his previous work, Korel (1992) developed a new method, known as the goal-oriented approach. This alleviates the problem of a path's infeasibility. The idea of this approach is to concentrate purely on branches that influence the execution of the goal node, ignoring branches with no influence. This was achieved through the classification of a program structure based exclusively on flow graph information, with regard to a goal node classed as critical, semi-critical, or non-essential. This classification is determined prior to program execution. Critical statements lead the execution flow away from the target node, while semi-critical statements can lead to the latter through the back of a loop. Non-essential statements do not have any influence on execution flow.

The search process will determine whether the program's execution should continue through the current branch, or via an alternative branch (i.e. the branch currently being reached does not lead to the execution of the goal node). If an undesirable execution flow at the current branch is monitored, then a real-valued function, called a 'distance function', is associated with the branch. Metaheuristic algorithms are used to automatically find new inputs that will change flow execution at the current branch. If the search fails to find new inputs, it is terminated in the case of the critical branch, or will be continued through the current branch if it is classed as semi-critical. The reason for this is that the target node may still be reached in the next iteration of the loop, even if the control flow diverges from the goal node at the semi-critical branch.

The goal-oriented method has some limitations. For instance, because this approach is based solely on a flow graph of the program, this makes some nodes difficult to reach for some programs (Ferguson & Korel, 1995), because the execution of a certain goal node could require prior execution of other nodes in the program.

2.4.3 The Chaining Approach

The chaining approach (Ferguson & Korel, 1995, 1996; Korel, 1996) extends the goal-oriented approach (Korel, 1992) to handle its limitations. This approach uses program dependency concepts, combined with a program flow graph, to find solutions to branch predicates by identifying a chain of nodes that affect the execution of the target node. This chain is built up iteratively during execution and it must be visited prior to the execution of the target node, in order to increase the chances of reaching the selected node.

The approach begins by executing a program for random inputs. During execution, a search process will decide whether execution should continue through the current branch, or whether an alternative branch should be taken because it does not lead to the target node. When the latter case occurs, program execution is suspended and new inputs are generated, in order to change the flow of execution at this branch. Data dependence analysis is used as a guide to determine which input variables may affect a given branch predicate. If the search process fails to find the inputs, the chaining approach will attempt to alter the flow and identify the chain of the 'essential' nodes by using data dependence concepts and enabling the chain to be executed first.

The work of Gallagher and Narasimhan (1997) adopts Ferguson and Korel's (1996) chaining approach, but unlike Ferguson and Korel's (1996) method, their approach supports the generation of real numeric types, strings and enumerated data structures. In addition, the execution is not initially forced along the entire path, but is rather advanced progressively, satisfying constraints step by step. This method of one decision at a time will help in the early detection of path infeasibility. However, this method may become stuck in a local optimum.

Michael, McGraw and Schatz (2001) use a GA to cover all branches in a program for condition coverage by delaying attempts to satisfy a particular condition until test data that matches that condition is found. If a condition has been visited, but only one branch has been exercised, GA shall be employed to cover the other branch. This method, however, may not be suitable for generating test data for a single branch.

The main contribution of the chaining approach is the use of data dependence, which improves the efficiency of optimisation searches (Gallagher & Narasimhan, 1997). Although the chain approach can be effective for a larger class of programs, the use of the 'find-any-path' concept could present some drawbacks. Firstly, it is hard to predict the coverage to be provided because different paths exercise different branches, resulting in different levels of coverage (Edvardsson, 1999). Secondly, the search time will significantly increase if there is a high number of paths which need to be considered when processing a chain (Harman et al., 2009).

2.4.4 Coverage-oriented Approaches

In coverage-oriented approaches, the fitness of individuals is rewarded on the basis of covered program structures. Coverage refers to the percentage of lines of a code that has been tested (Copeland, 2004). The ultimate aim is to achieve maximum coverage (Gross et al., 2009), as this will lead to a higher probability of faults or bugs being found in the software (Arcuri & Fraser, 2013). Various forms of coverage measures are used, as follows (Beizer, 2002):

- Statement coverage - estimates the percentage of program statements covered during testing.

- Branch coverage - measures the extent to which branch statements in the code are covered during the test. Examples of branch statements are ‘switch’, ‘if-else’ and ‘do-while’ statements.
- Path coverage - measures the number of feasible paths through the graph produced during the test.

Roper (1997) experiments with a genetic algorithm-based program to attain branch coverage. Test data which cover more program branches are rewarded with higher fitness values. This method lacks guidance for structures with a strong chance of being visited, such as a deeply nested structure, or a branch predicate that needs a particular value from a large domain in order to be true.

In his work, Watkins (1995) concentrates on full path coverage for programs. Test data that follow unexecuted paths are assigned higher fitness values than those that pass via paths which have already been covered, by penalising the fitness values of individuals that follow paths already used during the search. However, this penalisation of executed paths does not exploit the information in the branch predicates (Tracey, 2000). As a result, the quality of guidance for the search technique in discovering new and hitherto undiscovered paths is reduced.

Generally, coverage-oriented approaches do not achieve full coverage for large software systems, because they suffer from a lack of the kind of guidance provided for those unexplored structures, which are then only explored using a small sample of the overall input domain (McMinn, 2004).

2.4.5. Structure-oriented Approaches

The fitness of individual rewards on the basis of either branch distance, control structures, or both.

2.4.5.1 Branch Distance-oriented Approaches

Branch distance-oriented approaches exploit information from branch predicates, which evaluate how far a predicate is from obtaining its opposite value. This is similar to earlier work by Miller and Spooner (1976).

The work of Xanthakis et al. (1992) was the first to apply GA in the generation of structural test data. This method follows similar lines to earlier work by Miller and Spooner and it therefore suffers from problems which resemble those discussed in Section 2.4.1, such as the limited ability to detect path infeasibility. A tester chooses a path and then the branch predicates are extracted from it. A GA is employed to find test data which will satisfy all branch predicates in the path at once. The fitness function sums up the values of all branch distances.

Studies by Tracey et al. (Tracey, Clark, Mander & McDermid, 1998; Tracey, Clark & Mander, 1998) employ simulated annealing to the generation of structural test data. The approach aims to search for test data which will cover the program's statements in turn. Thus, the fitness function indicates whether or not the target statement has been exercised. The fitness function is the branch distance, which indicates how close the current execution is to adopting the desired branch according to the decision made. This means the fitness function will return zero if the current execution leads to the target condition (branch or statement); otherwise, it returns positive values.

The search proceeds while looking for test data to cover each statement in turn. When the search process stagnates at one node and no further progress can be made, the approach will attempt to generate test data for the next target node. Unlike Korel's approach (1990), the newly generated test data do not need to conform to an already successful sub-path. However, this leads to the search losing information about its progress (McMinn, 2004). The reason for this is because a solution that deviates from the desired path at an early stage of a search will be assigned similar fitness values to those which deviate at an advanced stage of a search.

The work of Sthamer (1995) and Jones, Sthamer and Eyres (1996) has attempted to exercise all branches in the software. Hence, it is not necessary to choose a path. A program is instrumented to dynamically calculate fitness values during the execution of a program for generating test data. The fitness function will exploit information contained in the branch predicate, in order to compute how far it is from exercising the program branches. The fitness function is formulated to return zero if test data that exercise the target branch are found. For instance, if a branch predicate a==b has to be evaluated as 'True', the fitness value will then be set as |a-b|, or according to the Hamming distance. For a branching condition a>= b, the fitness function is formed by |b - a|. In every evaluation, if the test data obtain the fitness

value 0, it will indicate that the search has found test data which will fulfil the branching condition.

The main criticism of branch distance-oriented techniques is that control information about the target is not included in the fitness function. For instance, suppose the target branch is 6 in the graph shown in Figure 2.8. The test case 't1' that follows the path <1(F),7 > and the test case 't2' that goes through the path <1(T),2(F),4 > will be rewarded with the same low fitness value under Jones et al.'s (1996) schema, although 't2' is closer to target node 6. This happens because no control dependence information has been incorporated into the fitness evaluation. This may cause the search to get stuck in local optima, thereby making it difficult to obtain full coverage (McMinn, 2004; Wegener, Baresel & Sthamer, 2001). The control-oriented approaches discussed in the next section will address this problem.

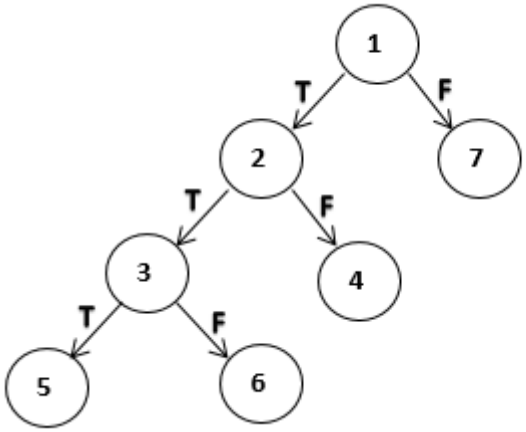


Figure 2.8 An example of a control flow graph (CFG) (adapted from Pargas et al., 1999, p. 279)

2.4.5.2 Control-oriented Approaches

Control-oriented approaches use control information for the fitness function. This is achieved by using a control dependency graph to determine predicate paths for the intended node.

Pargas et al. (1999) apply a GA for statement and branch coverage, guided by the control dependences in the program. For a goal node, a sequence of control-dependent nodes is specified, which should be exercised for the execution of the goal node. The fitness function is equivalent to the number of successful control-dependent node executions. Under this schema, the test case 't1' that follows the path <1(T), 2(F), 4 > has a higher fitness value than the test case which goes through the path <1(F), 7 >.

If *dn* is the number of control-dependent nodes for the current target branch and *en* is the number of successfully executed control-dependent nodes, Pargas et al.'s (1999) fitness function can be expressed as follows:

$$Fit = dn - en \tag{2.2}$$

However, it is worth noting that only using control structures in fitness functions will form plateaux on the fitness landscape (McMinn, 2004). As there is no distance information that can be exploited, this will result in insufficient guidance towards unexplored structures. For instance, Pargas et al.'s (1999) fitness landscape shown in Figure 2.10 has three plateaux. If the solutions fail to fulfil one of the branch predicates, no branch distance information will be given on how to descend the landscape for the search process, as guidance for those individuals who are closer to exercising the desired node.

```
void landscape_example (int i, int j)
{
        if (i >= 10 && i <= 20)
        {
                if (j >= 0 && j <= 10)
                {
                        // statement
                        // ...
                }
        }
}
```

Figure 2.9 Example for comparing fitness functions

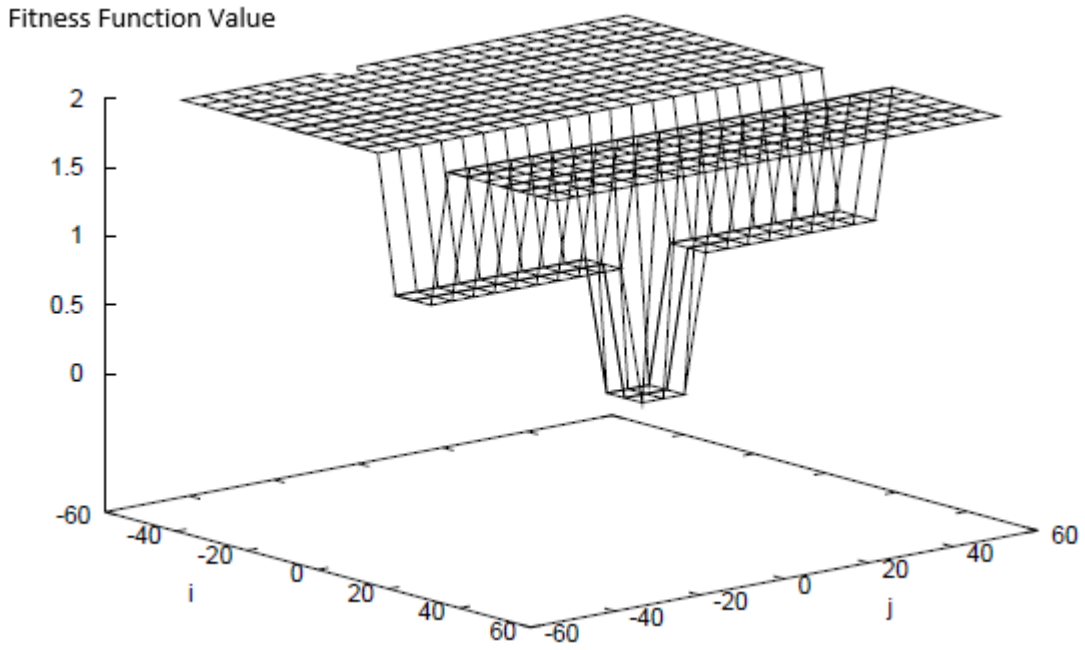


Figure 2.10 Pargas et al.'s (1999) fitness landscape, for the example illustrated in Figure 2.9 (adapted from McMinn, 2004, p. 130)

2.4.5.3 Combined Approaches

Tracey (2000) uses simulated annealing to combine both branch distance and control information for testing critical safety systems. The branch distance is calculated using Sthamer's (1995) approach, described in Section 2.4.5.1. The control information in this method refers to the number of control-dependent nodes which are successfully reached during an execution. This control information is used to scale branch distance values.

If *dn* is the number of control-dependent nodes for the current target branch and *en* is the number of successfully executed control-dependent nodes, while *bd* is the branch distance, the fitness function, *fit* will be computed using the following formulae (McMinn, 2004):

$$fit = \frac{en}{dn} \times bd \tag{2.3}$$

This scheme, however, results in unnecessary local optima in the fitness landscape (McMinn, 2004).

Wegener et al.'s (2001) approach follows the work of Tracey (2000) in computing branch distance. The fitness function is zero if the target is reached; otherwise, it is computed by normalising the branch distance and adding it to the approach level, as follows (Harman & McMinn, 2010):

$$fit = (dn - en - 1) + norm\ (bd) \tag{2.4}$$

The $(dn - en - 1)$ calculation is referred to as the approach level, *al*. Hence, the computed fitness function in Equation 2.4 can be expressed as follows:

$$fit = al + norm\ (bd) \tag{2.5}$$

The fitness function in Equation 2.5 has been used in many studies, such as the work of Harman and McMinn (2010). Wegener et al.'s (2001) fitness landscape is shown in Figure 2.11.

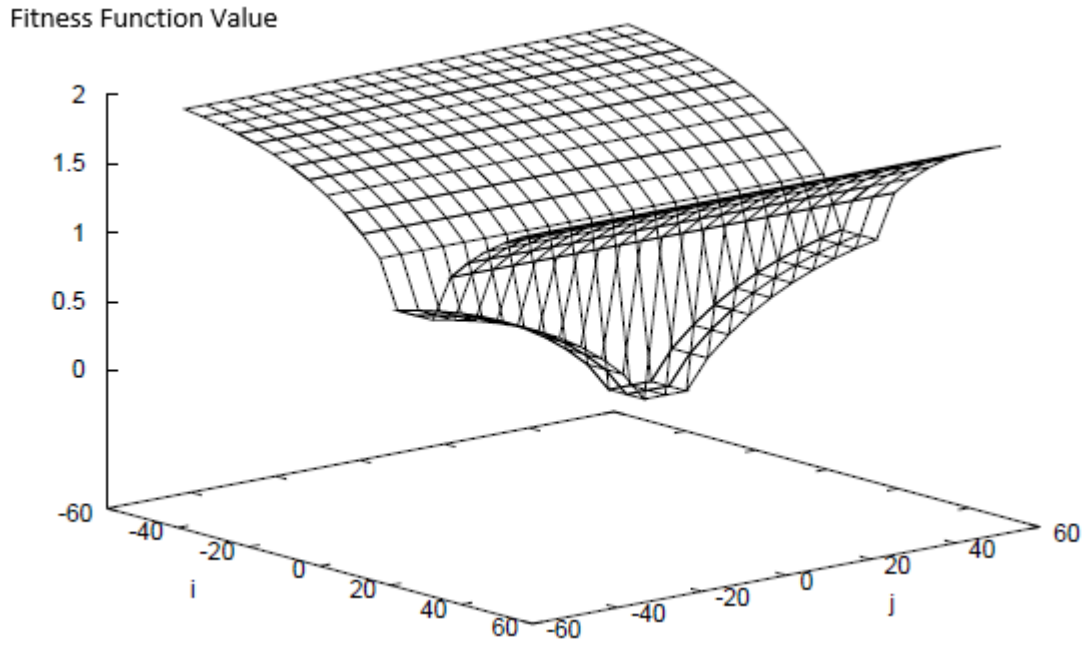


Figure 2.11 Wegener et al.'s (2001) fitness landscape for the example illustrated in Figure 2.9 (adapted from McMinn, 2004, p. 132)

Although Wegener et al.'s (2001) fitness function imposes a fitness landscape which resembles that of Pargas et al. (1999), the branch distance information employed by Wegener et al. (2001) prevents plateaux from forming on the fitness landscape. As depicted in Figure 2.11, a sweeping landscape results from each level to the next. However, the landscape presented by Wegener et al. (2001) is for the simple program shown in Figure 2.9 and a similar landscape is not guaranteed for all test objects. Using the same fitness function with other programs that have different structures will impose different landscapes. Therefore, it is important to investigate how the internal program structure could affect the performance of a fitness function. Hence, this study will examine the performance of different fitness functions in problems with different structures.

2.5 Discussion on Metaheuristics for the Generation of Test Data

Different fitness functions have been presented and used to automate the test data generation process, aiming for more cost-effective testing. To assess the performance of a certain fitness function in terms of how effective and costly it is in attaining the test goals, previous research has compared the results produced by metaheuristic algorithms, both with each other and with alternatives, such as random generation. A summary of studies that assess the performance of fitness functions is recorded in Table 2.2.

Table 2.2 Summary of Empirical Studies Assessing the Performance of Fitness Functions Designed to Test the Generation of Data. Hill climbing (HC); simulating annealing (SA); genetic algorithms (GAs); memetic algorithms (MA), and random testing (RT).

Article	Search Method					Evaluation of Performance Factors	Test Objects	Outcomes of the Experiment
	HC	SA	GA	MA	RT			
Pargas et al. (1999)			+		+	Statement and branch coverage	Six small test programs (21-82 lines of code)	GAs outperformed the random method for three programs
Tracey (2000)	+	+	+		+	Branch distance	Two industrial systems (nuclear primary protection system and a civil aircraft engine controller)	GAs are the most effective and efficient of the techniques
Tracey et al. (2000)			+			Branch coverage	95 ADA programs between 10 and 200 lines of code	Approach proposed for the use of optimisation techniques to generate test data in the testing of exceptions
Michael, McGraw & Schatz (2001)			+		+	Condition coverage	C/C++ programs (triangle classification and an autopilot control program for a Boeing 737)	GAs obtain a much higher coverage than the random algorithms
Wegener et al. (2001)			+		+	Branch coverage	Seven programs (5 to 154 lines of code)	GAs achieve the highest levels of coverage, with greater

								efficiency
Mansour & Salame (2004)		+	+			Path coverage	Eight functions of fewer than 86 lines of code	More paths were covered when using SA; GAs were faster than SA
Wang & Jeng (2006)	+		+	+		Branch coverage	Six programs	Memetic Algorithms outperform HC and GA
Miller, Reformat & Zhang (2006)			+		+	Branch coverage	6 programs (3. conversion of a hexadecimal number to a decimal number; bonus calculation; quadratic formula; triangle classification and two sort programs)	A little difference for simple programs; GAs can achieve 100% branch coverage in a much smaller number of generations for large programs
Watkins & Hufnagel (2006)			+			Path coverage	Triangle classification and an industrial program (CAPBDG)	The branch predicate and inverse path probability approaches were the best performers
Xiao et al. (2007)		+	+			Condition-decision coverage	Five C/C++ programs	GAs are consistently the best performers
Harman & McMinn (2007)			+			Branch coverage	Six real world programs	GAs can often be outperformed by HC and RT
Arcuri & Yao (2008)	+	+	+	+	+	Branch distance	Nine Java container classes	MA has the best performance

								in all the containers except Vector.
Harman & McMinn (2010)	+		+			Maximised coverage	Nine programs	HC outperformed GAs
Bhattacharya et al. (2011)	+	+	+			Branch coverage	A small synthetic example	GAs outperformed HC and SA
Fraser, Arcuri & McMinn (2013)			+	+		Branch distance	16 open source projects	The MA achieved up to a 32% higher branch coverage than the standard GA
Fraser & Arcuri (2014)			+		+	Branch distance	100 open source projects	A significant improvement in GAs over RT when employing the Randoop tool (Pacheco & Ernst, 2007), while similar results were produced when utilising the EvoSuite tool (Fraser & Arcuri, 2011b)
Shamshiri, Rojas, Fraser & McMinn (2015)			+		+	Branch distance	1,000 randomly selected classes	Little difference between the coverage achieved by GAs, compared to RT

The studies shown in Table 2.2 exhibit broad variation in terms of types and sizes of test objects, the methods of evaluation used, the number of experimental runs and data analysis.

2.5.1 Choice of Test Objects

In software engineering, the selection of test objects in which metaheuristic methods are employed is critical, since the test objects have a paramount effect on the results obtained (Fraser & Arcuri, 2013b). Thus, for any empirical investigation in software engineering, it is preferable to consider several sizes, types and structures of software (Arcuri & Briand, 2014), in order to gain sufficient confidence that the results obtained are not just applicable to the test objects used (Ali, Briand, Hemmati & Panesar-Walawege, 2010), unless the proposed method is to exclusively target a certain type of software.

Table 2.2 demonstrates that the test objects used in the studies were limited in their size, type and functionality. Using a specific type or functionality of test object is only reasonable if the proposed fitness function is aimed at that type. For instance, Arcuri and Yao (2008) aimed purely at container classes, while Tracey (2000) targeted embedded systems. Table 2.2 also illustrates that the test objects used are limited in their size. However, selecting test objects with a small number of lines is reasonable if researchers want to manually investigate the test object code. For instance, the triangle classification program is a benchmark used in many studies, as seen in Table 2.2. The triangle classification program is designed to classify a triangle, whether isosceles, scalene, equilateral, or invalid. This program is widely used for path coverage testing because it has a manageable number of paths (i.e. 14 paths in all). Subsequently, researchers tend to investigate each path individually, which is challenging when the test object has a high number of paths. Although using a small test object meant the investigation could be extensive, it highlights doubts as to whether or how results might be generalised to real world programs, where they are usually of a substantially larger size and have a much more complex structure.

2.5.2 Baselines for Comparison

Generally, when the performance of algorithms is investigated, it is required to include a method as a baseline for comparison (Johnson, 2002), and metaheuristic algorithms are no exception. According to Ali et al. (2010), in their systematic review of an empirical investigation into search-based, test-case generation, the metaheuristic technique “can only be assessed if it is compared with a carefully selected, meaningful baseline” (Ali et al., 2010, p.

747). The reason for this is that in the absence of known optimal solutions (Harman & Clark, 2004), using a baseline for comparison will justify and verify the results obtained using a metaheuristic technique. In other words, the baseline method will demonstrate whether or not the process of generating test data for a given software can be handled effectively by employing metaheuristic algorithms.

The baseline for comparison could be a simple technique, such as a random search, or another metaheuristic algorithm, such as hill climbing. Table 2.3 shows the baseline comparison used in the studies discussed above.

2.5.3 Number of Experimental Runs

Since metaheuristic techniques are stochastic by nature (i.e. executing the same method twice will yield different results) (Dreo & Siarry, 2007), it is crucial to account for the random variation in metaheuristic methods. Explicitly, this is usually accomplished by running the algorithms multiple times in order to capture the variation of the results obtained (Ali et al., 2010).

In software engineering, it is not recommended to run the algorithms for fewer than 30 runs, as this could result in unacceptably low levels of statistical power (i.e. it becomes very difficult to use statistical tests to detect and prove large differences) (Arcuri & Briand, 2014). However, there are exceptions, such as applying these algorithms in embedded systems, where the process of generating each test suite could be extremely expensive in terms of time and resources (Arcuri, Iqbal & Briand, 2010). Table 2.3 shows how many runs were performed by the studies discussed above.

2.5.4 Data Analysis

As mentioned earlier, running a sufficient number of algorithms is essential to allow statistical analysis to be carried out on the data. As metaheuristic techniques are attributed with random variation, the use of statistical tests is of paramount importance in determining whether the difference between comparisons is due to chance, or if there is a real difference between them (Wohlin et al., 2012). Empirical studies that do not back their claims through statistical evidence may cast doubts on their actual effectiveness (Arcuri & Briand, 2014; Wohlin et al., 2012).

Table 2.3 shows that some of the above studies have included empirical analyses, supported by some kind of statistical testing. In particular, these are t–tests (Efron, 1969) and Wilcoxon Mann–Whitney tests (Mann & Whitney, 1947), where algorithms are compared in a pairwise fashion. On the other hand, ANOVA was used in Watkins and Hufnagel's (2006) work for multiple comparisons. Conversely, Fraser et al. (2013) and Bhattacharya et al. (2011) used effect size measures to quantify the relative effectiveness of algorithms.

Table 2.3 Baselines for comparison, repetitions and statistical tests, used in the empirical studies in Table 2.2: Hill climbing (HC) and random testing (RT)

Article	Baselines for comparison	Repetitions	Statistical Tests	Reported Metrics
Pargas et al. (1999)	-	32 runs	-	CYC LOC
Tracey (2000)	RT	-	-	-
Tracey et al. (2000)	-	-	-	-
Michael, McGraw & Schatz (2001)	-	5 runs	-	-
Wegener et al. (2001)	-	4-6 runs	-	CYC LOC
Mansour & Salame (2004)	HC and Korel's algorithm (1990)	20 runs	-	CYC LOC
Wang & Jeng (2006)	-	-	-	-
Miller, Reformat & Zhang (2006)	RT	10 runs	-	-
Watkins, & Hufnagel (2006)	RT	9-100 runs	ANOVA	-
Xiao et al. (2007)	RT	-	-	-
Harman, & McMinn (2007)	HC and RT	30 runs	t−test with level 0.01	-

Arcuri A, Yao X. (2008)	RT	100 runs	Mann Whitney U-tests	-
Harman & McMinn (2010)	RT	60 times	One-tailed Wilcoxon rank sum test	-
Bhattacharya, Sakti, Antoniol, Guéhéneuc & Pesant (2011)	-	20 times	t-test and Cohen d-effect size	-
Fraser, Arcuri & McMinn (2013)	-	30 runs	Two-tailed Mann-Whitney and effect sizes (Aˆ12) U-test	-
Fraser & Arcuri (2014)	-	10 runs	Two-tailed Mann-Whitney and effect sizes (Aˆ12) U-test	-
Shamshiri, Rojas, Fraser & McMinn (2015)	-	50 runs	Mann-Whitney U-test at a level of = 0:05	-

2.5.5 Summary

Despite the above limitations with regard to the type and size of test objects; the method of evaluation used, and the number of experimental runs and data analysis, these experiments are encouraging as they have been effective in proving the usefulness of metaheuristic search techniques. However, the conclusions of these studies are limited, since each study has used only one fitness function to measure the quality of the solutions generated. These studies did not culminate in any sort of recommendations as to which definitions of fitness functions offer better performance. Most useful are studies where diverse fitness functions are compared across a reasonable quantity of test objects. The outcome can be an automated fitness function selection model predicting the coverage obtained from the given function

which is likely to be best for a given test object. Therefore, this study investigates the usefulness of different fitness functions with regard to their ability to achieve high coverage.

Moreover, most of the above studies fail to explain the conditions under which the tested functions are expected to offer high or low coverage. It could be claimed that there are untested objects, where one would expect the fitness function being tested to be outperformed by other functions. For instance, a test object with multiple flag variables may not be successfully tested with one particular fitness function, but it could achieve high coverage using others (Baresel & Sthamer, 2003). Currently, there is a lack of understanding of how the relative performance of different functions depends on the software under study (Stützle & Fernandes, 2004). Addressing this issue will therefore increase our understanding of which test objects are difficult to cover for any given function.

The present study will look into the possible relationship between the characteristics of the test objects and the coverage obtained from a fitness function. The aim is to understand the extent to which achieving high coverage is related to the characteristics of the software being tested. This work will provide software architects and testers with a means of supporting their decisions, as they will know that software with specific characteristics will enable a specific fitness function to achieve higher coverage.

To be able to assess the difficulty of a test object to be covered by a given fitness function, software measurements which describe software features can be used. The following section discusses some of these measurements.

2.6 Software Metrics

A software metric is a measurement directly collected from a program's source code (Subramanyam & Krishnan, 2003). Measurement refers to the process by which numerical values are assigned to software attributes in a way which will describe them according to clearly defined rules (Fenton & Bieman, 2014). As such, software metrics are a valuable means of measuring and evaluating certain characteristics of software. These metrics are therefore used as an indicator of the performance of specific software

Software metrics can be categorised into three groups: code metrics, process metrics and resource metrics (Bhatti, 2011). Process metrics are more relevant to the measurement of software process attributes, such as the effort required, estimated duration and process

quality, while resource metrics are needed for the estimation of resources required for a software project, such as physical resources and man-power (developers).

Code metrics refer to those metrics which are directly countable from the source code. They provide software developers with a valuable means of gaining insight into the code they are developing. Since the focus of this study is on the generation of test data for structure testing, wherein test cases are defined on the basis of the program code, it will exclusively discuss and utilise code metrics. The following section explains the latter.

2.6.1 Software Code Metrics

Code metrics can be used to understand which parts of a code should be redeveloped or more thoroughly tested. These metrics are utilised to identify potential risks and discern the current state of a project (Lee, 2007). Code metrics can be further grouped into sub-categories, based on the different measures they perform: quantitative metrics, metrics of program flow control complexity and object-oriented metrics.

2.6.1.1 Quantitative Metrics

Quantitative metrics are concerned with the quantitative characteristics of a code, such as the total number of methods of a program, the number of operations or operands in a program and the number of comments and average lines per method.

LOC could well be the simplest and most elementary measure. They count the number of source code lines and the original purpose of this metric was to estimate the man-power (developers) needed for a project. LOC could also be used to observe what changes could occur in different versions of a software item, in terms of LOC. However, LOC cannot be used to compare programs in different languages, as programming languages can vary greatly, depending on the language syntax and style (Bhatti, 2011).

Halstead metrics (Halstead, 1977) are a well-known example of quantitative metrics. Halstead's Length (HALL) is based on the number of operators and operands used in a program. In contrast, Halstead's Vocabulary (HALV) counts the number of different operators and operands. Halstead derives other metrics, as shown in Figure 2.12.

Primitive Metrics:
Number of unique operators: μ_1
Number of unique operands: μ_2
Total occurrences of operators: N_1
Total occurrences of operands: N_2
Potential operator count: μ_1^*
Potential operands count: μ_2^*

Derived Halstead:
Program length: $N = N_1 + N_2$
Program's vocabulary size: $\mu = \mu_1 + \mu_2$
Program volume (program content): $V = N * \log_2 \mu$
Program's potential volume (the minimum size of a program): $V^* = (2 + \mu_2^*) \log_2 (2 + \mu_2^*)$
Program level: $L = V^*/V$
Program difficulty: $D = 1/L$
Error estimate for a program: $\hat{L} = \frac{2}{\mu_1} * \frac{\mu_2}{N_2}$
The intelligence content of a program: $I = \hat{L} * V$
Effort required to generate a program: $E = \frac{V}{L} = \frac{\mu_1 N_2 N \log_2 \mu}{2\mu_2}$
Programming of each program by time (in seconds): $T = E/18$

Figure 2.12 Derived Halstead metrics

Although Halstead metrics enable developers to perform useful measures on code, there are criticisms of Halstead's methodology for deriving certain mathematical relationships between the derived metrics (Marco, 1997). Marco (1997) shows that the computation of Halstead metrics for simple programs, such as 'Bubble sort' suggests that the program is complex. Subsequently, it is recommended to also include other metrics. The reason for this is that by combining several software measurements, the weaknesses of individual metrics may be compensated (Lammermann et al., 2008).

2.6.1.2 Metrics of Program Flow Complexity

This group of metrics is dependent on an analysis of the program's CFG. An analysis of the CFG will give the complexity of the source code. Complexity refers to the difficulty of understanding the program code, explaining it to others, detecting and correcting its defects, then maintaining it (Harrison, Magel, Kluczny & DeKock, 1982). The metrics of a program flow's complexity are built based on the assumption that the more complex the source code, the more likely it is to contain errors.

The most commonly used metric in this category is McCabe's cyclomatic complexity (CYC) (McCabe, 1976). This is solely based on the program's CFG. CYC is the maximum number

of linearly independent paths in a method. A path is linear if there is no branching statement in the execution flow of the corresponding code. The higher the CYC value, the more paths there are in the program, thus leading to more difficulties for a developer in understanding the different paths. Figure 2.13 illustrates how CYC is computed.

Cyclomatic complexity (CYC) is calculated as**:** $M = E - N + P$
Where
M McCabe metric
E Number of edges of the graph of a program
N Number of nodes of the graph
P Number of connected components

Figure 2.13 Cyclomatic complexity metric

The complexity level of a program is decided on the basis of the computed CYC values, as shown in Table 2.4. CYC metrics have received criticism regarding their weighting scheme. For instance, CYC metrics evaluate simple and nested condition structures in the same manner. Therefore, they receive the same weight. However, it is known that nested conditional structures are harder and more complex to understand than simple non-nested structures (Bhatti, 2011). Furthermore, CYCs consider a program on the basis of the number of branching nodes present, irrespective of program size, or the size of the code under each branching node.

Table 2.4 McCabe's cyclomatic complexity ranges (adapted from Bhatti, 2011)

CYC Value	Code Complexity
1 – 10	A simple program, without much risk
11 – 20	More complex program with moderate risk
21 – 50	Complex program with high risk
50+	Untestable program with high risk

2.6.1.3 Object-oriented (OO) Metrics

The above metrics were proposed before the concept of object-oriented (OO) programming was established. However, the coding style of OO programming endorses the need for a new

kind of metrics which can represent OO concepts, such as inheritance, coupling and cohesion (Basili, Briand & Melo, 1996).

The most frequently used metrics in the above group are Chidamber and Kamerer's (CK) metrics (Chidamber & Kemerer, 1994). CK metrics measure the cohesion, complexity, coupling and inheritance of OO systems, namely: weighted methods per class (WMC); depth of inheritance (DIT); number of children (NOC); coupling between objects (CBO); response for class (RFC), and lack of cohesion in methods (LCOM). These metrics can be defined as follows:

- WMC measure the sum of the complexity of all methods defined in a class. The complexity refers to CYC values. While CYC is measured at class level, WMC gives an overall view of the class complexity.

- DIT defines the maximum length from the node to the root of the hierarchy tree of a class. In a longer path in a class, it is more likely to be difficult to estimate its behaviour. This may indicate an inappropriate abstraction in the design (Tang, Kao & Chen, 1999). An empirical study of DIT validates that the deeper the inheritance, the greater the chance of fault detection (Briand, Morasca & Basili, 1996). DIT could be seen as an indicator of complexity; the higher the DIT value, the more complex the software is supposed to be (Li & Henry, 1993b).

- NOC counts the number of immediate subclasses subordinated to a child class in the class hierarchy. A class with a large number of children tends to be more complex. Therefore, NOC can be seen as a measure of testability; the more children in a class, the more changes need to be tested.

- CBO defines the connection and interdependency of an object in a class with other objects (Bansiya & Davis, 2002). A class is considered to be coupled with another class if it calls its method or accesses its instance variables (Tang et al., 1999). CBO is a count of the number of other classes to which it is coupled.

- RFC is the cardinality of the set of all methods that can be invoked, in response to a message to an object or the method of the class. This includes local methods and methods in other classes. The larger the number of methods that respond to a message, the greater the complexity of that class.

- LCOM measures the disparate nature of methods in the class. LCOM is the number of disjoint pairs of methods in a class which do not share any member variable. A higher value of LCOM implies a lack of cohesion. However, LCOM only considers methods and instance variables implemented in the class, whereas what is inherited is excluded.

Besides the CK metrics, Abreu and Carapuça (1994) propose class-level OO oriented metrics, based on the quantity of the source code. Class-level OO metrics simply count the number of attributes of a class, such as public attributes, private attributes and other object-oriented attributes, as shown in Table 2.5.

Table 2.5 Class-level OO metrics (adapted from D'Ambros, Lanza & Robbes, 2012)

Name	Description
Fan-in	The number of other classes that reference the measured class
Fan-out	The number of classes referenced by the measured class
NOA	The number of attributes
NOPA	The number of public attributes
NOPRA	The number of private attributes
NOAI	The number of attributes inherited
LOC	The number of lines of code in a class
NOM	The number of methods
NOPM	The number of public methods
NOPRM	The number of private methods
NOMI	The number of methods inherited

2.6.2 Code Metrics for Software Testing

Software metrics give software developers the means to analyse the code, in order to be able to improve it. A wide range of software metrics is used for different purposes in various software development phases. For instance, software metrics have been used for maintainability (Li & Henry, 1993b; Nogueira, Ribeiro, Carlos & Zenha-Rela, 2014; Riaz, Mendes & Tempero, 2009); defect analysis (Subramanyam & Krishnan, 2003), and software quality (Lincke, Gutzmann & Löwe, 2010; Rosenberg & Hyatt, 1997).

Few existing studies focus on the testability of software systems. Testability refers to the degree to which a software enables another software to be validated (Standard, 2005). Harrison and Samaraweera (1996) examined whether there is an association between the number of test cases and design metrics in terms of code quality and the time required to produce it for functional or OO paradigms. They found that the number of test cases could be used as an indicator of the effort needed for functional and OO programs.

Bruntink and van Deursen (2006) investigated whether the values of CK metrics in a class are correlated to the size of the corresponding test suite and the required testing effort for that class. The results of their study suggest there is a significant correlation between CK metrics and test-level metrics. Shrivastava and Jain (2010) used CK metrics to propose design metrics, namely an automated test case for unit testing (ATCUT), in order to predict the effort needed for class testing.

However, the above studies focus on the effort required for testing. There are very few studies which have endeavoured to look into the possible correlation between code metrics and the coverage percentage obtained from testing with metaheuristics. Lammermann et al. (2008) investigated the suitability of code metrics for the assessment of whether search-based testing can be successfully performed for a given OO program. The result of the study suggests that there is a mediocre correlation between search-based testability and the code metrics used in the study, i.e. LOC, Halstead's length, Halstead's vocabulary and CYC.

Daniel and Boshernitsan (2008) proposed a technique that uses code metrics for predicting the test coverage achieved. They used code metrics and coverage to train decision tree classifiers. The code metrics applied in the study were CYC, number of static field, number of private class and number of public class. They found that the classifiers can accurately predict a coverage degree of a given test object.

The use of code metrics for developing a better understanding of automatic testing using metaheuristics is still an area with plenty of potential for research. This work investigates whether code metrics used as an indicator of high coverage would be achievable through metaheuristic tests. If this is the case, then it should be possible to decide, before the test, which fitness function should be chosen for a given test object. Subsequently, when the measurements of a given test object show that it would be difficult to achieve high coverage using a certain fitness function, program transformations could be initiated, such as the flag removal. This would lead to improvements in metaheuristic performance. This knowledge will lead to a better understanding of how the software structure can adversely affect performance or the accuracy of the automated testing techniques using metaheuristics.

2.7 Conclusion

To sum up, this chapter has surveyed, analysed, reviewed and critiqued previous work undertaken using metaheuristic algorithms to generate test data for structural testing. Previous research can be categorised based on the definition of the fitness function optimised in four groups: the goal-oriented approach; the chaining approach; the coverage-oriented approach, and the structure-oriented approach.

An understanding of how different fitness functions may affect the progress of a search can improve the application of metaheuristics to software testing. Therefore, this study will examine a number of different fitness functions in problems of varying levels of difficulty, as a means of assessing their relative performance.

Following the above, this study will investigate the reason for obtaining specific coverage from each fitness function, by analysing the potential relationships between the attributes of test objects and the coverage obtained. The investigation of the effects of test object properties on the suitability of fitness functions is important, since this could increase our understanding of what makes test cases hard to generate. It is achieved using software measurements as source information for evaluating whether or not a fitness function can successfully cover a given test object.

The study of the possible relationship between the characteristics of a software item and the desired coverage is important, since it is expected to contribute to the improvement of the design for generating test data. This work will provide software architects and testers with a

means of supporting and validating their decisions, as they will be aware that a specific fitness function can achieve higher coverage, through the application of software with certain characteristics. The next chapter will discuss the research methodology for this study.

Chapter 3: RESEARCH METHODOLOGY AND EXPERIMENT DESIGN

3.1 Introduction

The selection of the most appropriate research methodology will rely on the study's objectives. This chapter discusses the research design used and illustrates how the experiment is set up for this study. The rest of the chapter is structured as follows: the research questions are provided in Section 3.2 and the research methodology is presented in Section 3.3. The design for the experiment is discussed in Section 3.4. Finally, Section 3.5 summarises the chapter.

3.2 Research Questions

The objectives of this study are identified as follows:

- Investigating the influence of different definitions of fitness functions on the performance of the automated testing techniques.

- Gathering insights into the relationship between features of software systems and the coverage achieved by fitness functions, in order to understand the performance of functions.

In order to fulfil these objectives, there are two main research questions posed:

Research Question 1 (RQ1): **Does the choice of a particular fitness function affect the performance of automated testing techniques?**

Hypothesis 0 (H0): the mean values of the coverage obtained from all the fitness functions are the same.

Hypothesis 1 (H1): the mean values of the coverage obtained from all the fitness functions are different.

Research Question 2 (RQ2): **Is there a correlation between software metric values and the coverage obtained by a given fitness function?**

Hypothesis 0 (H0): There is no correlation between software metric values and the coverage obtained by a given fitness function.

Hypothesis 1 (H1): There is a correlation between software metric values and the coverage obtained by a given fitness function.

3.3 Research Methodology

This section discusses the major components and outlines the different phases of this research. Figure 3.1 illustrates the conceptual framework of the research design. The selection of testing criteria; search technique; fitness functions; baseline for comparison; search-based testing tool; benchmark test objects, and software metrics are described below.

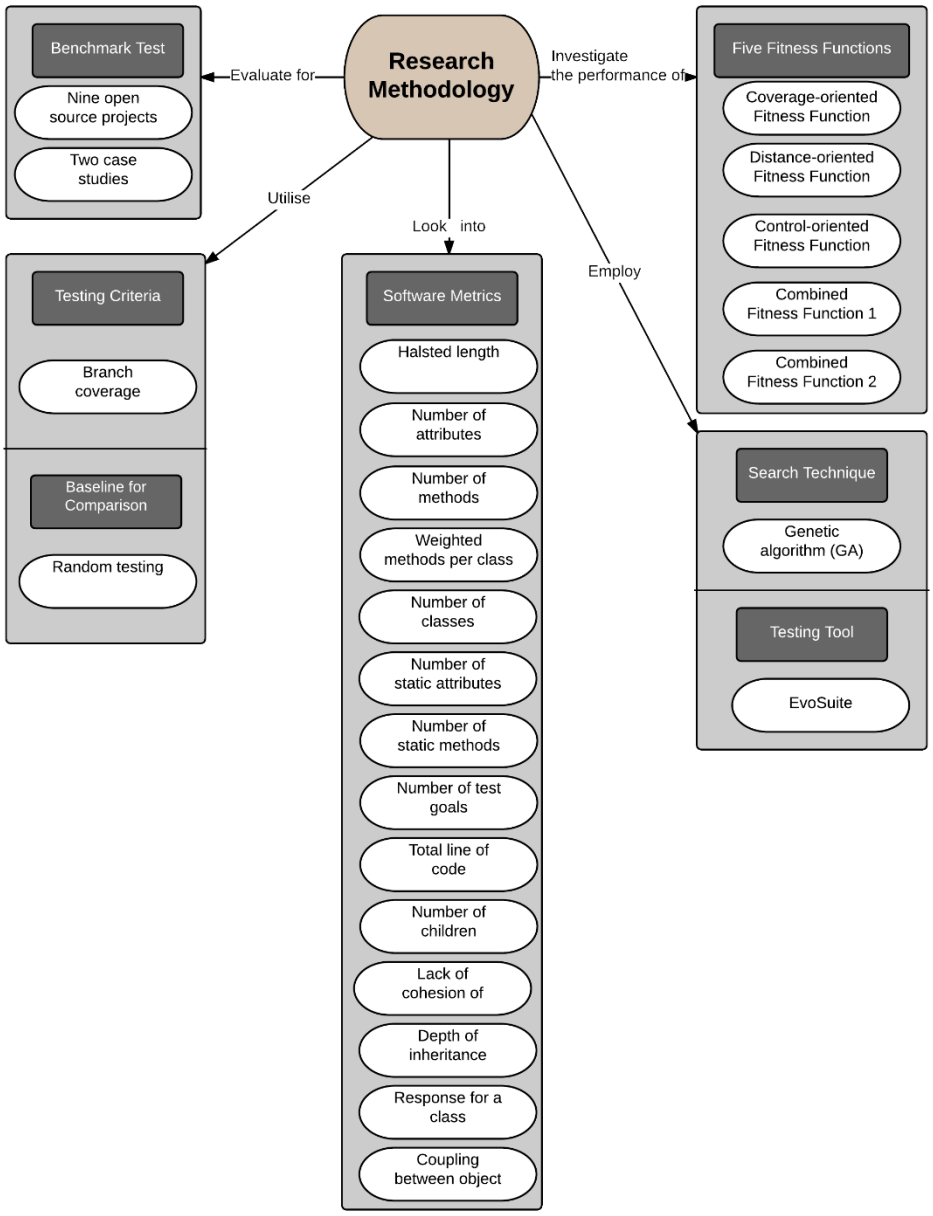


Figure 3.1 Conceptual framework of the research design

3.3.1 Testing Criteria

To evaluate the quality of an individual, this study focuses on branch coverage as a test goal, aimed at covering all the branch statements in the code, such as 'switch', 'while' and 'if-else'. This is in consideration of the fact that branch coverage is motivated by the significant cost of manually generating branch coverage test inputs (Harman & McMinn, 2010). An experimental study shows that branch coverage criteria perform consistently with adequate efficiency and effectiveness, compared with other coverage criteria (Gupta & Jalote, 2008).

Every branch in the object being tested was treated as a goal. The aim was to exercise each goal twice to cover the true and false conditions of the branch. The reason for this was to avoid directing the search towards covering the true value of a predicate, ignoring the case where a false value is exercised. Otherwise, the search would oscillate between the two contrasting values of the branches (true or false), resulting in partial coverage of the code (Fraser & Arcuri, 2013b).

3.3.2 The Metaheuristic Search Technique

The genetic algorithm (GA) was the technique of choice to automate the generation of test data. The outcomes of studies in Table 2.2 show that GAs outperform local search methods for test data generation. These studies show that GAs have a greater ability to increase the quality of tests and save the computational cost of testing. As discussed in Chapter 2, the way local search techniques work is by maintaining a solution and exploring the search space through steps within its neighbourhood; going from the current solution to a more appropriate close one. However, local search techniques are unable to exploit the local correlation in the structures of the search space in this way (Harman & Jones, 2001; Kirsopp, Shepperd & Hart, 2002). On the other hand, GAs iteratively perform evaluation, selection and recombination procedures, which enable them to explore the search space in multiple directions. Hence, GAs are much more effective for generating test data (Lefticaru & Ipate, 2008).

3.3.3 Fitness Functions

The measures for the fitness of test suites varied, as discussed in Chapter 2. Some studies have used branch coverage (Roper, 1997), approach level (Pargas et al., 1999), a distance calculation (Fraser et al., 2013; Shamshiri et al., 2015), or other measures (Tracey, 2000; Wegener et al., 2001). The research question motivating this empirical study is concerned

with investigating the usefulness of these measures in identifying the fitness of individuals for branch coverage. This is achieved through a sequence of experiments designed to evaluate the relative performance of the five fitness functions used: coverage level function (CLF); branch distance function (BDF); control fitness function (CFF); combined fitness function 1 (COM1), and combined fitness function 2 (COM2). These were selected to represent a sufficiently extensive range of fitness function categories.

These different functions were then evaluated in terms of the branch coverage achieved. The aim was to divide the code for the object being tested into a series of partial goals, each designed to exercise a specific branch value (true or false condition). To satisfy each partial goal, the fitness function guided the search; in effect, rewarding test suites that came close to satisfying the partial goal. If a test suite covered the partial goal, it was rewarded with a fitness value of '0'; otherwise, the respective test suite received a value greater than '0', depending on the definition of the fitness function utilised. As the functions measure fitness differently, the fitness value of the test suites was normalised in the range [0, 1] to allow a fair comparison between different measurements. The following normalisation function was used (Arcuri, 2013):

$$norm(x) = \frac{x}{(x+1)} \tag{3.1}$$

In the sections which follow, the definition of each of the fitness functions implemented is explained. The strengths and weaknesses of these functions have been discussed in Chapter 2 and are summarised below in Table 3.1.

Table 3.1 Specifications of the tested fitness functions

Number	Title	Citation	Name	Category of Fitness Calculation		
				Coverage-oriented	Branch - oriented	Control-oriented
1	CLF	(Roper, 1997)	Coverage level function	+		
2	BDF	(Fraser & Arcuri, 2013b)	Branch distance function		+	
3	CFF	(Pargas et al., 1999)	Control fitness function			+
4	COM1	(Tracey, 2000)	Combined fitness function 1		+	+
5	COM2	(Wegener et al., 2001)	Combined fitness function 2		+	+

3.3.3.1 Coverage Level Fitness Function (CLF)

The CLF was originally proposed by Roper (1997) for branch coverage. This function is modified in this study to exercise both branch values (true and false), rather than arriving at just one value, as in the original definition of the function. In this function, an individual is rewarded on the basis of the number of branches executed. The individual covering the highest number of branches in the code will gain the lowest fitness values. This function is primarily concerned with ensuring that the highest possible level of coverage is achieved.

Let *T* be a test suite and *B*, the set of branches of the object being tested; the coverage level *cl (b,T)* for each branch $b \in B$ on test suite *T* is defined as follows:

$$cl(b,T) = \begin{cases} 0 & \text{If both values of the branch have been covered,} \\ 0.5 & \text{If the predicate has been executed once (either true or false condition)} \\ 1 & \text{Otherwise.} \end{cases} \quad (3.2)$$

Hence, the overall fitness function of a test suite is calculated using the following equation:

$$fit(T) = |M| - |MT| + \sum_{b \in B} cl(b,T) \quad (3.3)$$

where *M* is the set of methods in the object and *MT* is the set of methods executed during the test. The difference $|M| - |MT|$ is used to reward coverage of methods in the test objects which have no branch statements.

3.3.3.2 Distance-oriented Fitness Function (BDF)

The BDF (Fraser & Arcuri, 2013b) uses a branch distance measurement which reflects how close a branch's predicate is to being reached. Suppose, for example, the aim is to execute the true value of the branch (L == 7) in Figure 3.2. If 'L' has the value of 5, then the branch distance is computed using the formula |L-7|, which is 2 in this case. The closer the input data is to 7, the closer the branch will be to being considered as exercised (i.e. being true). In the event that the branch statement is encountered in the body of a loop, the smallest branch distance is used.

```
while (true) {
        if (L==7)  // branching statement
        {
                //other statements
        }
```

Figure 3.2 Branch distance example

Let *B* be the set of branches of the test object and the minimal branch distance, $d_{min}(b,T)$ for each branch $b \in B$; the branch distance $d(b,T)$ for each branch $b \in B$ in test suite *T* shall be defined as follows:

$$d(b,T) = \begin{cases} 0 & \text{If the branch has been covered} \\ \text{norm}(\text{dmin}(b,T)) & \text{If the predicate has been executed at least twice} \\ 1 & \text{Otherwise} \end{cases} \quad (3.4)$$

where $d_{min}(b, T)$ is '0' if at least one of the branch's values (true or false) has been covered, and > 0 otherwise. Thus, the fitness function of a test suite *T* is:

$$fit(T) = |M| - |MT| + \sum_{b \in B} d(b,T) \quad (3.5)$$

where *M* is the set of methods and *MT* is the set of methods executed during the test. Again, the difference $|M| - |MT|$ is used to reward coverage of methods in the test objects which have no branch statements.

3.3.3.3 Control-oriented Fitness Function (CFF)

CFF utilises Pargas et al.'s (1999) definition. In this function, control information is extracted from the control dependency graph, which is equivalent to the test object's code, in order to compute an individual's fitness value. The fitness value is equivalent to the number of successful control dependent node executions towards the intended branch (refer to Chapter 2 for a complete evaluation of the function).

Let *dn* be the number of control dependent nodes for the current target branch, and *en* be the number of successfully executed control-dependent nodes; the fitness function $f(b,T)$ for each branch $b \in B$ in test suite *T* is defined as follows:

$$f(b,T) = norm\ (dn - en) \tag{3.6}$$

where *T* is the test suite, *B* is the set of branches of the test object and *norm* is a normalisation function in the range [0, 1]. Thus, the fitness function of a test suite *T* is:

$$fit\ (T) = |M| - |MT| + \sum_{b \in B} f(b,T) \tag{3.7}$$

where *M* is the set of methods in *T* and *MT* is the set of methods executed during the test. The difference $|M| - |MT|$ is used to reward the coverage of methods in the test objects with no branch statements.

3.3.3.4 Combined Fitness Function 1 (COM1)

COM1 is built on Tracey's (2000) definition that combines both branch distance and control information. The branch distance is calculated using Equation (3.4), mentioned above in Section 3.3.3.2. The control information here refers to the number of control-dependent nodes successfully attained during execution. This control information is used to scale branch distance values (Refer to Chapter 2 for a description of this function).

Let *T* be the test suite and *B* be the set of branches of the test object; the fitness function *f (b,T)* for each branch $b \in B$ on test suite *T* is defined as follows:

$$f\ (b,T) = norm\ (\frac{en}{dn} \times d(b,T)) \tag{3.8}$$

The *f (b,T)* value is normalised: $norm$ in the range [0, 1], *dn* is the number of control-dependent nodes for the current target branch, *en* is the number of successfully executed control-dependent nodes, and $d\ (b,T)$ is the branch distance.

The fitness function of a given test suite, *T* is:

$$fit\ (T) = \ |M| - \ |MT\ | + \sum_{b \in B} f\ (b,T) \tag{3.9}$$

where *M* is the set of methods and *MT* is the set of methods executed during the test. The difference $|M| - \ |MT\ |$ is used to reward the coverage of methods in the test objects which have no branch statements.

3.3.3.5 Combined Fitness Function 2 (COM2)

COM2 is a version of the fitness function used in two prior studies by Harman & McMinn (2010) and Wegener et al. (2001). The function combines both branch distance and an approach level in its measurement. 'Approach level' (*al*) indicates how close the executed path is, as compared to the required partial aim. It is proportionate to the numbers of control-dependent nodes which are not encountered in the path executed by the test data and is computed as follows:

$$al\ = (dn\ - en - 1) \tag{3.10}$$

where *dn* is the number of control-dependent nodes for the current target branch and *en* is the number of successfully executed control-dependent nodes. Hence, the fitness function *f (b,T)* for each branch $b \in B$ in test suite *T* is defined as follows:

$$f(b,T) = al + norm\ (\ d(b,T)) \tag{3.11}$$

The branch distance $d(b,T)$ is computed using Equation (3.4). The overall fitness function of a test suite *T* with respect to all branches is computed as:

$$fit\ (T) = \ |M| - \ |MT\ | + \sum_{b \in B} f(b,T) \tag{3.12}$$

where *M* is the set of methods and *MT* is the set of methods executed during the test. The difference $|M| - \ |MT|$ is used to reward coverage of methods in the test objects which have no branch statements.

3.3.4 Baseline for Comparison

When the performance of metaheuristics is investigated, it is essential to include a method as a baseline for comparison (Johnson, 2002), in order to justify and verify the results obtained using the metaheuristic technique (Ali et al., 2010). For software testing, it is recommended to use RT as a comparison baseline to assess the performance of search-based testing techniques (Harman, Mansouri & Zhang, 2009). Several prior studies have utilised RT as a method of evaluation (Arcuri & Yao, 2008; Harman & McMinn, 2007, 2010).

Although RT is the easiest of the automated testing techniques to apply, different studies show that it still gives good coverage. For instance, different researchers (Bird & Munoz, 1983; Thevenod-Fosse & Waeselynck, 1993; Voas, Morell & Miller, 1991) have proved the usefulness of the random test generator, compared with other test data generation techniques. One investigation showed that RT results in similar coverage of the GA (Shamshiri et al., 2015).

This study employs RT as a baseline for comparison for a dual purpose: It helps identify whether the object being tested is simple enough to be covered by a simple testing technique; otherwise, it justifies why it is necessary to use a complex metaheuristic technique.

Since RT is used here as a natural baseline in order to understand the effectiveness of the fitness functions used, we are studying it with basic implementation. This means that the search process randomly generates sequences of statements to the test objects, coupled with randomly generated inputs. When a test case exercises a new branch statement that has not

been covered, it is inserted into a test suite, or else deleted. However, this strategy could increase the size of the test suite, resulting in a high execution cost (Shamshiri et al., 2015).

One way of circumventing this problem is to determine the length (L) of test cases that will be generated during the search (Arcuri & Yao, 2008). Although this implies there will always be a maximum limit (L) to the length, the test cases can still have a random length.

3.3.5 Search-based Testing Tool

This study employs EvoSuite (Fraser & Arcuri, 2013b) in order to perform experiments in the domain of test generation for object-oriented software. EvoSuite is an open source search-based test data generation tool, which automatically generates JUnit test suites for a given Java problem. EvoSuite uses a sandbox to take care of any potentially unsafe operations that may harm the host machine, such as the deletion of files. This feature is vital because the test objects used in this study are real-world programs and they will most likely have unsafe operations.

It is appropriate to apply EvoSuite in this study for two different reasons. Firstly, EvoSuite aims to evolve the entire test suite generation, considering the simultaneous testing of all the tested branches in the programs. As such, the individuals of the search are test suites (sets of test cases), contrary to the more common strategy of using an individual as a test case. An investigation into the benefits of the entire test suite showed that it leads to better results, because the feasibility or difficulty of a single test case does not affect the overall performance of the entire test suite (Fraser & Arcuri, 2013a).

The second reason for using EvoSuite is to allow for a fair comparison between GA and RT generation, by implementing both techniques on the same platform. According to Johnson (2002), when different techniques are being assessed in relation to each other, it is vital to ensure reasonably fair implementations for all these techniques in order for their effectiveness to be comparable.

EvoSuite evolves candidate test suites aimed at covering all test goals, while at the same time minimising the total size of the suite (i.e. reducing the number of test cases and their length). As such, when there is a tie between test suites with respect to their fitness values, EvoSuite chooses the test suite which is composed of a cumulatively lower number of statements. This is done with the aim of improving search performance, since the longer the test suite, the

more memory and execution time required (Fraser & Arcuri, 2013b). Hence, the optimal solution, T_0, in a search problem refers to the test suites which exercise all branch statements in the code. This will have the minimum number of statements, compared with other test suites.

3.3.6 Structure-based Software Metrics

To answer RQ2, we investigated measurements of test objects' source code, in order to figure out whether software properties influence the performance of fitness functions. In order to select suitable metrics for the experiment, we drew upon previous reviews of metrics (Bruntink & van Deursen, 2006; Simons, Singer & White, 2015; Sjøberg, Anda & Mockus, 2012). Different and widely-used tools were employed to collect measures of the test objects' source code, namely *JHawk*[1] (Lincke, Lundberg & Löwe, 2008), *VizzAnalyzer*[2] (Löwe, Ericsson, Lundberg, Panas & Pettersson, 2003), *Eclipse Metrics Plugin 1.3.6*[3] (Narasimhan, Parthasarathy & Das, 2009), and *EvoSuite*[4] (Fraser & Arcuri, 2013b). The tools and metrics are shown in Table 3.2. The plus sign '+' indicates that a metric is calculated using the corresponding tool.

[1] http://www.virtualmachinery.com/jhawkprod.htm
[2] www.arisa.se
[3] http://metrics.sourceforge.net/
[4] http://www.evosuite.org/

Table 3.2 The selected metrics

Type	Level	Name	Description	Tools			
				JHawk	VizzAnalyzer	Eclipse Metrics Plugin 1.3.6	EvoSuite
Size	Method	HALL	Halstead Length	+			
	Class	NOA	Number of attributes			+	
		NOM	Number of methods			+	
		WMC	Weighted methods per class		+		
	System	NCL	Number of classes			+	
		NSA	Number of static attributes			+	
		NSM	Number of static methods			+	
		NTG	Number of test gaols				+
		TLOC	Total line of code		+		
Complexity	Class	DIT	Depth of inheritance		+		
		NOC	Number of children		+		
		RFC	Response for a class		+		
		LCOM	Lack of cohesion of methods		+		
	System	CBO	Coupling between object classes		+		

In selecting the metrics, we considered those which measure two different aspects of a test object: size and complexity. These aspects have been measured on three levels: method, class and system, in order to gain a broader view of the features of the test objects.

Size-related metrics are quantitative measurements which express the length of a test object in terms of the numbers of attributes, methods, classes and branches in it. Although the size of the code would not be expected to be the basis for a direct assessment of the code's testability, it is plausible that larger test objects would impede the attainment of the test's goals. Hence, size-related measures could lead to conclusions about the possibility of achieving high coverage. This group also includes measures evaluating the number of possible paths through the object's code, based on a control flow graph (CFG) of the object. This would help determine whether the flow control properties of a given object can influence the ability of the fitness function to guide the search progress towards those test suites offering higher coverage.

An empirical investigation has shown that the more complex the software, the more complex the corresponding test cases (Nogueira, 2012). We investigated whether this was true for the coverage obtained using a given function; in other words, whether software complexity also has an effect on the coverage achieved using a fitness function. Complexity-related metrics

quantify the structural properties of a test object, such as measures of coupling, dependency, cohesion, and the nature of the object's hierarchy. These aspects result in a complex control flow, which might complicate a search for better solutions, resulting in the fitness functions delivering lower performance.

3.3.6.1 Description of Metrics

The following is a brief description of the metrics presented, based on how the tools implemented them:

- **Halstead Length (HALL)** counts the total number of operators and operands in a method. Operators refer to method names, arithmetical operators and language keywords, whereas operands express numeric and string constants.

- **Number of methods (NOM)** refers to the methods in a class. NOM includes private, public and overridden methods.

- **Weighted method count (WMC)** represents the sum of weights for the methods of a class. The method's weight count was calculated according to suggestions made by Li and Henry (1993a), where a method's weight is equal to the number of possible alternative paths through the code.

- **Number of test goals** (**NTG**) counts numbers of branches twice, taking into account the two values (true and false) of a branch, plus the number of branchless methods. Branchless methods are those methods which do not contain branching statements and which could be covered by simply calling it.

- **Lines of code (LOC)** count the non-blank and non-comment lines of code of a program.

- **Depth of inheritance tree (DIT)** expresses the maximum inheritance path from the class to the root class. DIT values are calculated for each class. They indicate the longest distance to the root of the hierarchy. This value is on an absolute scale ranging from '0' to the longest path towards the root class.

- **Number of children (NOC)** indicates the number of direct subclasses of a class. NOC is calculated for each class. The values are integer values ranging from '0' for

no children, to the maximum number of children a class will have on an absolute scale.

- **Coupling between object classes (CBO)** expresses the number of classes to which a class is coupled. A class is considered as coupled to another class if it calls its method or accesses its instance variables (Tang et al., 1999).

- **Response for a class (RFC)** represents the set of methods which can potentially be invoked in response to a message received by an object of the class. This includes local methods and methods in other classes.

- **Lack of cohesion of methods (LCOM)** counts the number of disjointed pairs of methods in a class which do not share any member variables. LCOM were first implemented according to suggestions made by Chidamber and Kemerer (1994). LCOM only considers methods and instance variables implemented in the class, whereas inherited ones are excluded.

3.3.6.2 Tools for Measuring Software Metrics

Three different and widely-used tools are employed to collect measures of the test objects' source code, namely *VizzAnalyzer, JHawk* and *Eclipse Metrics Plugin 1.3.6*. *VizzAnalyzer* (Löwe et al., 2003) is a maintenance analyser plugin for the Eclipse IDE that was developed at Växjö University (Sweden). It reads software code and performs a number of software metrics. Here, *VizzAnalyzer* data produced by the metrics was imported into Excel, where it was further analysed. *VizzAnalyzer* has in fact been used in several different studies (Barkmann, Lincke & Lowe, 2009; Panas, Lincke, Lundberg & Lowe, 2005; Strein, Lincke, Lundberg & Lowe, 2007; Wingkvist, Ericsson, Lincke & Lowe, 2010).

On the other hand, *JHawk* (Lincke et al., 2008) is a metrics calculation plugin for the Eclipse IDE that allows snapshots of code metrics to be recorded. *Eclipse Metrics Plugin 1.3.6* (Narasimhan et al., 2009) is an open source tool for collecting the code metrics from the source code.

3.3.7 Benchmark Test Objects

To analyse the performance of the fitness functions, we selected programs which the test generation system could be trialled on. In this research, we refer to such programs as *test*

objects. We randomly but uniformly selected nine different test objects from SF110's corpus of open source projects (Fraser & Arcuri, 2014). SF100 is a statistically sound representative of a collection of 100 Java projects selected from the SourceForge website, which is one of the largest repositories of open source projects on the Web. To avoid bias caused by only considering the open source code, we also selected two case studies from the literature: string case study subjects from Maragathavalli (2011) and a numerical case study previously employed by Arcuri and Briand (2011). Both case studies were written in the Java programming language.

The test objects were selected according to their testability. Each of these objects should be run independently and in a way where they can properly de-allocate the resources used, such as the memory. We balanced different types of classes: container classes, classes which make greatly use of string process and numerical functions. Table 3.3 summarises the properties of these test objects.

Table 3.3 The set of test objects

Number	Name	Description	Branches
1	a4j	API handles all Web service requests to and from Amazon.com	390
2	water-simulator	Agent-based urban water demand simulator	64
3	dsachat	A program for the role-play game, DSA	3224
4	greencow	Printing a single statement	1
5	Petsoar	An open source answer to Sun's J2EE PetStore project	26
6	Follow	Monitor ('follow') text file system	560
7	Lilith	A logging and access event viewer for Logback, log4j & java.util.logging	2258
8	Heal	Health Education Assets Library	4906
9	Jgaap	A Graphical Authorship Attribution Framework	98
10	NumericalCaseStudy	A numerical case study previously employed by Arcuri and Briand (2011)	209
11	StringCaseStudy	String case study subjects from (Maragathavalli, 2011)	607
∑			12343

Since this study focuses on branch coverage as a test criterion, particular attention was paid to the number of branches each test object contained. As can be seen from Table 3.3, the selected test objects vary greatly in the number of branch statements. The final number of branch statements totalled 12,343, ranging from a small project with just a single branch statement to a larger project with 2,258. An investigation revealed that a large number of branching statements in a test object could result in complex, discontinuous and non-linear search spaces (Lammermann et al., 2008). Such search spaces could reduce the performance of the automated search techniques to that of a random search. Therefore, the number of branch statements in a test object will directly influence the effectiveness of the fitness functions.

When selecting the test objects, care was taken to ensure these objects had a wide value spectrum for each selected metric. Table 3.4 shows the measures collected for each test object. HALL varied between 19 and 8,012; NOM ranged between 1 and 2,301, NOA varied from 0 to 980 and NCL, from 1 to 532. TLOC ranged between 13 and 52,553, and NOC, between 0 and 101. WMC ranged from 1 to 5077, while CBO varied between 0 and 1,174, NSA between 0 and 553 and NSM, between 1 and 114. NTG showed differences between 11 and 85,742, with DIT attaining values between 0 and 152, and RFC 1 and 3,473. Lastly, LCOM varied between 1 and 33,191.

Table 3.4 Collected metric values for each test object

Test Objects	HALL	NOA	NOM	WMC	NCL	NSA	NSM	NTG	TLOC	DIT	NOC	CBO	RFC	LCOM
a4j	1463	174	478	696	45	1	1	85742	5151	0	0	65	553	13026
water-simulator	1364	551	352	539	73	134	2	1035	12294	16	16	233	488	677
dsachat	6401	140	228	532	31	163	6	3250	5107	6	6	93	337	1060
greencow	19	0	1	1	1	0	1	11	13	0	0	0	1	1
petsoar	307	192	461	591	96	22	12	9258	4813	66	65	311	851	1809
follow	6605	142	396	524	81	74	18	886	7948	46	40	267	551	5111
lilith	8012	980	2301	5077	532	553	91	77146	52553	152	87	1174	2807	31581
heal	5338	465	1521	3516	205	173	114	51842	30054	103	101	873	3473	33191
jgaap	1060	87	83	136	22	0	2	1331	1527	13	8	59	105	75
NumericalCaseStudy	1937	6	14	94	11	7	2	3946	615	0	0	0	16	15
StringCaseStudy	780	5	31	220	12	1	3	4505	1002	0	0	0	33	17
∑	33286	2742	5866	11926	1109	1128	252	238952	121077	402	323	3075	9215	86563

3.4 Experimental Design

This section explains the design of the experiments, including the search technique settings, search budget, procedure for the experiments and pilot testing.

3.4.1 Search Technique Settings

Experiments were conducted in the domain of test generation for object-oriented software using GAs. The aim was to produce test suites (sets of test cases) for a given test object, such

that the test suite maximises branch coverage, while at the same time minimising the number and length of the test cases.

3.4.1.1 Representation

In these experiments, an individual is an entire test suite of variable size. The search space is composed of all possible test suites and in its entirety, it can range in size from 1 to a predefined maximum '*N'* of test suites. A test case consists of a sequence of method calls to construct objects in the test objects and call methods on them. The test case will range in size from 1 to l_{max} possible statements. A statement can be an object constructor, a method call, a variable assignment, a field, or a primitive.

Since test case length and test suite size may vary, individual representation is also of variable size. However, although the research space produced is large, it is finite, since its parameters (i.e. N and l_{max}) are finite.

3.4.1.2 Search Operators

The initial population is randomly generated. Subsequently, the population is iteratively evolved through two primary operators: crossover and mutation, which are used to evolve new offspring (Figure 3.3). The new generation is evolved in iterations until a stopping criterion is met.

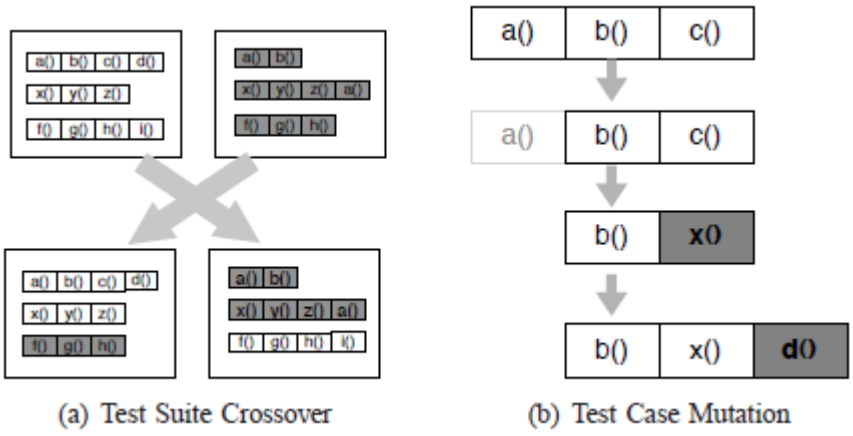


Figure 3.3 Search Operators: crossover is applied at test suite level; mutation is applied to test cases and test suites (adapted from Fraser & Arcuri, 2013b, p. 280)

The crossover between two test suites, 'T1' and 'T2' produced two offspring: Child 1 and Child 2. A single crossover point 'r' was randomly chosen from [0, 1] in two parent test

suites. Crossover operations recombined the genes of T1 up to the crossover point and the genes after the crossover point of T2, to form Child 1 and vice versa for Child 2.

The mutation operator is implemented after the crossover operator. The mutation operator is applied at two levels: test suite and test case level. The mutation operations will randomly change a randomly chosen test suite. When test suite *T* is selected to be mutated, its test cases are mutated with a probability of 1/|T |, where *T* is the number of test cases in the test suite. When a test case is mutated, three different operations are sequentially applied: delete, change and add. In a test case of length (L), each of its statements are deleted or changed with the probability, 1/L, where 'L' is the number of statements in a test case. When deleting statements leads to invalidated dependencies within the test case, dependent statements are also removed. The changing test case means modifying its statements, taking into consideration the validity of the entire test case; for instance, changing a method call while retaining its return type. Lastly, the insertion process refers to adding a new statement to the test case with a probability of $\sigma = 1/L$ in a random position.

3.4.1.3 Setting up Parameters

As the parameter configurations of the GAs will greatly affect the coverage level achieved (Lammermann et al., 2008), the GA parameters were kept constant during all the experiments. This is necessary to ensure a fair comparison between results. The first parameter is the *population size*, which specifies the number of test suites generated for the initial population. *Population size* remained the same during the optimisation process (i.e. the next generation has the same size as the initial generation). The *population size* for the GA was set to 80 and the maximum test suite size (maximum number of test cases in a test suite) was set to N = 100, with the length of the test cases set to L = 80. These settings were experienced as being suitable, at a point where the test case covered did not require a longer time.

The second parameter is the *crossover rate*: when two test suites are chosen to proceed to the next generation, the *crossover rate* will determine the probability with which these two test suites are crossed over. The crossover rate here is set at .75, which is considered to be 'best practice' (Fraser & Arcuri, 2013b).

The third parameter is the *mutation rate*, which specifies the probability with which a test suite will be altered. As explained in Section 3.4.1.2, the mutation rate depends on test suite

size and test case length. The initial probability of adding a new statement in a test case was set to $\sigma = 0.5$, whereas the initial probability for inserting a test case into a test suite was set to $\sigma = 0.1$.

The fourth parameter is the *elitism rate,* which determines the percentage or number of best suites in a population which will automatically survive to the next generation. The *elitism rate* was set at 1, which means one test suite. The best of the current population (its elite) is automatically copied to the next generation.

The fifth parameter is the *selection mechanism*, which is the algorithm used to choose parent test suites for reproduction from the current population. Rank selection was applied, where each individual was chosen according to a probability that is proportionate to its rank and individuals were ranked with respect to their fitness values (Srinivas & Patnaik, 1994). The benefit of rank selection is that a fitter individual will not dominate the others, resulting in premature convergence (Arcuri & Fraser, 2013). When there are two test suites with the same fitness, shorter solutions are awarded higher ranks. Therefore, a test suite which obtains better branch coverage and which is of shorter length will have a higher chance of being selected for reproduction.

3.4.2 Search Budget

A search budget is a vital factor in search-based software engineering experiments (Arcuri & Fraser, 2013). Search budget refers to the conditions under which the search should be stopped, as finding an optimal solution is not always guaranteed (Harman & Clark, 2004). The search budget can be expressed in many different formats, such as maximum execution time, number of fitness evaluations and maximum number of statements executed. The common format of the search budget in the literature is the number of fitness evaluations (Safe, Carballido, Ponzoni & Brignole, 2004). However, as the test objects in this experiment vary greatly in length, comparisons based on fitness evaluations can be meaningless, since one test suite can be either extremely short or long compared with the others. To allow better and less biased comparisons, the stopping condition was selected to be the maximum number of statements executed, set at 1,000,000. That is, the search stops when a test suite with 100% branch coverage is found, or 1,000,000 statements have been executed.

3.4.3 Experimental Procedure

Experiments were performed on a computer running a Linux operating system, with 2.5 GHz computing cores and four gigabytes of memory. Since this study is a large scale experiment, EvoSuite was used as a command-line tool, rather than an Eclipse plugin. For each experiment, a shell-script was predefined to call EvoSuite with parameters set as command line inputs to EvoSuite. The outputs were re-directed to local files. When all experiments were finished, the outputs were collected and analysed.

Each test object was taken in turn, with the aim of recording the level of coverage that could be achieved by each fitness function. Each trial was repeated 30 times to take the random nature of the search technique into consideration. Therefore, in total, we had 11×6×30=1,980 experiments. The computational time required to finish a single trial may vary for each test object. For instance, each trial of the smallest test object (greencow) took approximately five minutes, whereas the largest test object (lilith) required around three to four hours to finish a single run.

3.4.4 Pilot Testing

A *pilot experiment* was carried out prior to conducting the empirical study. The reason was to assess the study's feasibility and to estimate the time, cost and search budget needed to accomplish it in full. Carrying out such a pilot experiment will permit an estimation of appropriate settings for the experiments and enhance the research design, prior to the performance of a large scale empirical study.

The pilot experiment was conducted with two fitness functions: coverage level function (CLF) and branch distance function (BDF) on only three test objects: a4j, greencow and jgaap. All experiments in the pilot study were repeated 10 times. Based on the results obtained from this, modifications were made to the set-up for the experiments. For instance, two stop conditions were tested: the number of fitness evaluations and the number of the statements executed. Preliminary trials showed that the number of fitness evaluations would favour test objects which are short in length. Moreover, the computational time taken to run the pilot experiment would help us estimate the time required to run the empirical study and subsequently, to determine the numbers of test objects that could be tested during the given timeframe to complete this study.

3.5 Summary

This chapter presents the research questions and hypotheses that support it. It also discusses the research approach employed in this study. Five different functions have been chosen and implemented using Java programming language. To analyse the performance of the fitness functions, we selected nine open-source projects and two case studies from the literature. A total of 14 metrics were selected to evaluate different aspects of the test objects, such as size and complexity.

The chapter also explains the design for the experiments in terms of the search technique settings, search budget, and procedure for the experiments. The next chapter will provide the study's findings and will analyse them in the light of the research questions.

Chapter 4: EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Introduction

This chapter presents the results of the data collected from 1980 experiments conducted to examine the coverage achieved by five fitness functions. Throughout this examination, we investigated the performance of different fitness functions and the extent to which aspects of the test objects influence their effectiveness. The rest of this chapter is organised into the following sections: Section 4.2 presents a summary of the findings. An analysis of the findings' significance relative to the research questions is then presented in Section 4.3. A discussion on the threats to the validity of this study follows in Section 4.4. Section 4.5 then summarises the chapter.

4.2 Summary of the Findings

The means and standard deviations over the 30 runs of the fitness functions and test objects are shown in Table 4.1. A higher mean indicates better performance, whereas a lower standard deviation result means more consistent performance. The columns in the Table below represent the five fitness functions tested, along with RT; the rows representing the test objects. Bold text is used to identify the best performing methods in each trial.

Table 4.1 The mean and standard deviation of the 30 trials of the six fitness function: coverage level function (CLF), branch distance function (BDF), control fitness function (CFF), combined fitness function 1 COM1, combined fitness function 2 (COM2)

	Mean						Std					
Test Object	CLF	BDF	CFF	COM1	COM2	RT	CLF	BDF	CFF	COM1	COM2	RT
a4j	**0.780**	0.755	**0.780**	0.640	0.655	0.525	**0.013**	0.024	**0.013**	0.263	0.235	0.287
water-simulator	0.598	0.577	0.575	0.581	0.580	**0.757**	0.069	0.036	0.027	0.031	**0.024**	0.183
dsachat	0.738	0.700	0.740	0.740	**0.741**	0.736	0.021	0.096	**0.012**	**0.012**	0.013	0.018
greencow	1.000	1.000	1.000	1.000	1.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000
petsoar	0.603	0.615	0.616	0.573	**0.662**	0.150	0.084	0.064	0.068	0.164	**0.009**	0.058
follow	0.820	0.810	0.816	0.822	0.807	**0.827**	0.025	0.027	0.048	**0.022**	0.050	0.028
Lilith	0.578	0.601	0.231	0.649	0.640	**0.660**	0.247	**0.076**	0.259	0.167	0.171	0.123
heal	0.495	**0.618**	0.614	0.396	0.319	0.498	0.240	0.021	**0.011**	0.285	0.295	0.230
jgaap	0.752	0.734	**0.769**	0.754	0.688	0.768	0.022	**0.021**	0.029	**0.021**	0.158	0.066
NumericalCaseStudy	0.845	**0.851**	0.835	0.844	0.838	0.836	0.026	0.025	0.028	**0.022**	0.023	0.024
StringCaseStudy	0.679	0.688	0.687	**0.696**	0.687	0.680	0.021	**0.011**	0.016	0.027	0.017	0.014

The results show that there is no one individual fitness function that outperforms all the others in every test object. The performance of each function is different for different test objects. For instance, while CLF has the best performance in 'a4j' (with 78% coverage), it was the worst performing function in 'heal'. Similarly, COM1 produced the best coverage for StringCaseStudy, but provided the worst results for 'heal', with only 39% of the branching statements being attained.

In the case of 'greencow', which is the smallest test object, all fitness functions found test suites with 100% branch coverage, although it must be borne in mind that 'greencow' is a very trivial test object, with only 1 method, 1 branch and 13 lines. In the larger test objects, the six functions do not achieve 100% branch coverage: the larger the test object, the lower the branch coverage. Examples of this are 'heal' (1,521 methods, and 4,906 branches) and 'lilith' (2,301 methods, and 2,258 branches), which have the lowest branch coverage of any of the test objects, with the coverage percentage not exceeding 70% of any of these large test objects. For instance, CFF achieved only 23% of the branching statements in 'lilith' and COM1 and COM2 performed very poorly, with only 39% and 31% of the branches exercised, respectively. However, there are cases where good-quality test cases were generated for large problems. For example, in the case of 'dsachat' (228 methods, and 3,224 branches), despite being a test object with a large number of branches, branch coverage was high for all functions, exceeding 70% in every case.

The fairly poor performance of the six functions over the objects: 'heal' and 'lilith' can be explained by the fact that they are large objects with a high number of methods and branches which will result in a large and complex search space of input parameters. These two objects have similar characteristics. For instance, 'heal' and 'lilith' are the largest with regard to the number of static methods with 114 and 91, respectively. Static methods are considered as a strongly disturbing influence on evolutionary testability (Lammermann, Baresel & Wegener, 2008), because they often need to be invoked many times to reach a condition, such as in a case where local static variables are used as a timer and the only possible way to cover a particular branch is by repeating method calls multiple times.

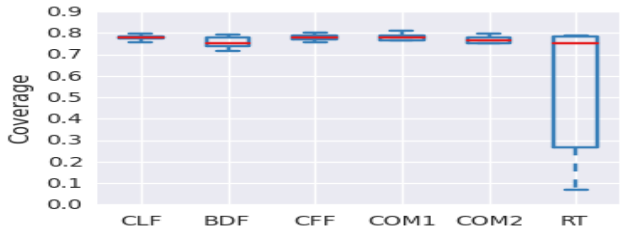
For medium-sized test objects, like 'water-simulator' (352 methods and 64 branches), NumericalCaseStudy (14 methods and 209 branches), and StringCaseStudy (31 methods and 607 branches), all the functions exhibit similar performance, producing test suites with very

similar coverage. The biggest difference in the coverage percentage is observed for the object ‘petsoar’ (461 methods and 26 branches), while the five GA implementations produced test suites with close degrees of coverage (around 60%), random testing (RT) struggles to produce test suites with similar results, finishing with just 15%.

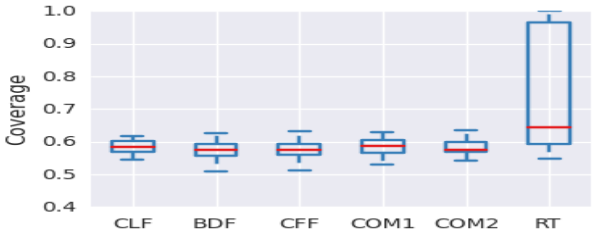
However, RT did not always perform worse than the GA implementations at branch coverage. In fact, it performed better than other techniques on two objects: ‘follow’ and ‘lilith’. The reason behind this might be that sophisticated methods, such as GAs might incur extra computational overheads, due to their complex behaviour, compared with naive techniques such as RT. The effect of these overheads can be extremely high and were hard to define before running the experiments.

We observe that COM1 and COM2, have almost the same performance. This is more evident for the following objects: ‘water-simulator’, ‘dsachat’ and ‘lilith’, where performance is almost identical. Both COM1 and COM2 involve both branch distance and approach level in their calculation of fitness values. On the other hand, the CLF function does not outperform the other functions for any of the test objects, except for object ‘A4j’, where CLF produced the same degree of CFF coverage, namely 78%. A possible explanation of this case might be that CLF provides little guidance on structures that are unlikely to be covered by chance, such as deeply nested structures. This is because CLF tends to reward test suites that cover the longest paths through the test object. Thus, the exclusive use of coverage as fitness criteria means that the search progress is mainly guided to cover a long and easily accessible structure. Hence, CLF’s schema was unable to attain superior performance for most of the test objects.

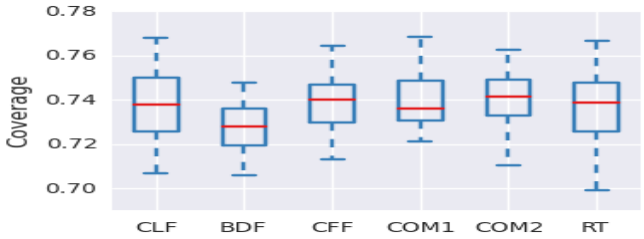
For a better understanding of the distribution of fitness values for the 30 runs, results are visualised as boxplots in Figure 4.1. The results presented in the boxplots are collected within the evaluation of different fitness functions, expressed on the x-axis. The y-axis denotes the coverage values. The middle red lines represent the median value for each case.



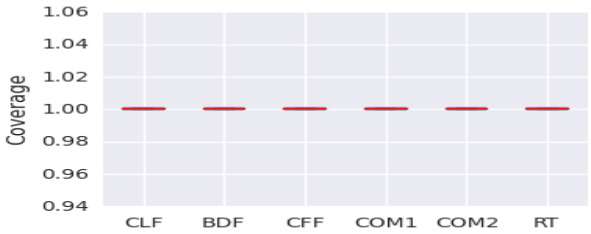
(a) a4j



(b) water-simulator

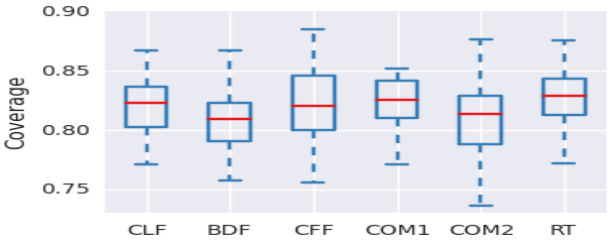


(c) dsachat

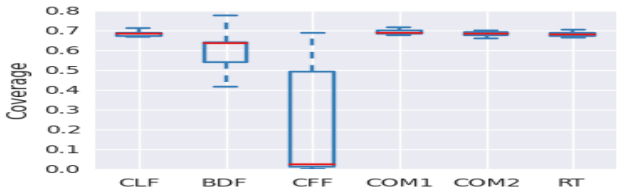


(d) greencow

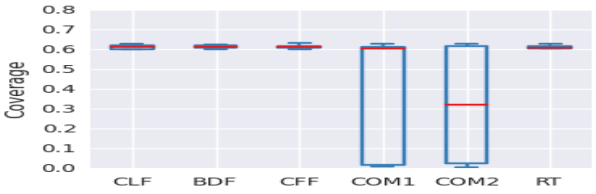
(e) petsoar



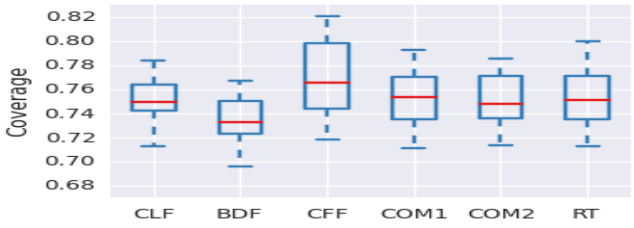
(f) follow



(g) lilith

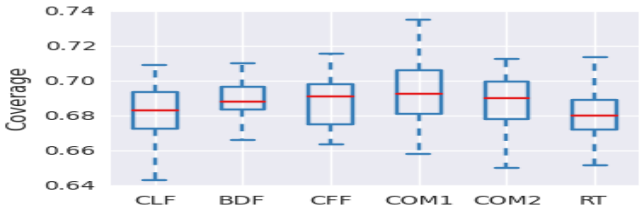


(h) heal



(i) jgaap

(j) NumericalCaseStudy



(k) StringCaseStudy

Figure 4.1 The distribution of coverage in the 30 runs for coverage level function (CLF), branch distance function (BDF), control fitness function (CFF), combined fitness function 1 (COM1), combined fitness function 2 (COM2) and random testing (RT).

The above results show that for each test object, there is a fitness function that produced better coverage for it. However, there is no best function to use on all test objects. A function could be the best for one test object, but the worst for the others. This can be seen in Figure 4.1, where the coverage obtained by CFF for 'lilith' was very low compared to other functions, although CFF produced high coverage for the 'heal' object. The opposite was noted for COM1 and COM2, while there was a wide range of variation in the fitness values obtained for the 'heal object, which was not true of the coverage produced for the 'lilith' object.

These results show that the performance of different functions depends on the software object being tested. That is, there are likely to be features of a test object that make it hard for certain functions to produce similar coverage in it to what is produced in other objects. This highlights the need to measure software features and explore their relationship with the coverage achieved using fitness functions. To this end, the second aim of this study is to figure out what features in a test object will make it hard for a particular search schema to guide the search towards higher coverage.

4.3 Hypothesis Testing

Statistical analysis was performed using the *scipy* and *numpy* Python frameworks and the R statistical tool (Ihaka & Gentleman, 1996). In order to determine which statistical test type (parametric or non-parametric) should be used for testing the hypotheses, the underlying data distribution was examined to find out whether it corresponded to the assumptions of the parametric tests, in terms of the probability distribution of the data. As the data obtained does not correspond to the assumptions of the parametric tests regarding the normality and equal variance of the data, non-parametric tests were chosen for a sound analysis of the data.

4.3.1 Answer to the First Research Question

> ***(RQ1):*** **Does the choice of a particular fitness function affect the performance of automated testing techniques?**

As seen in the discussion in Section 4.2, the fitness function produces diverse degrees of coverage across the different test objects. In this section, the hypothesis testing is described, carried out to determine whether there is a significant difference in the coverage obtained by

fitness functions for each test object. Statistical significance provides an indication of how likely it is that the differences observed between the techniques being compared is due to chance.

Kruskal–Wallis' non-parametric test (Kruskal & Wallis, 1952) was used to check for a significant difference in the performance of the fitness functions. The Kruskal–Wallis test makes no assumptions regarding the distribution of the results in terms of their normality or variance. However, the distribution of coverage attained by the functions must be similar in shape and scale, in order to use the median as a measure of comparison. The distribution of the functions' performance was examined and the results show they were not identical (i.e. they have different shapes and variability). It can be seen in the boxplots in Figure 4.1, that the scale of distribution for RT (for example) is much larger than with the other functions. If the distribution of performance for each function is similar, the length of the box and whiskers plot should be approximately the same. Therefore, means (rather than medians) are used as the main measure for the Kruskal-Wallis test. The 30 repeated trials for each of the test objects were submitted for Kruskal-Wallis analysis. The functions were compared with each other, with a null hypothesis of no significant difference in the performance achieved by the fitness functions, as follows:

Hypothesis 0 (H0): the mean values of the coverage obtained from all the fitness functions are the same.

Hypothesis 1 (H1): the mean values of the coverage obtained from all the fitness functions are different.

Table 4.2 illustrates the Kruskal-Wallis test results from the mean of the coverage obtained by the functions. Tests which were deemed to be statistically significant at a 0.05 level of significance are shown in bold text. For the test objects, 'greencow', 'follow' and 'NumericalCaseStudy', the test results show the functions performing to the same degree in these objects. This is not surprising for 'greencow', as this is the smallest test object, where all the fitness functions found the optimal solution. On the other hand, eight Kruskal-Wallis tests were found to be statistically significant. Therefore, there is evidence to suggest a difference in the coverage achieved by fitness functions for those eight test objects, whereupon the null hypothesis of the first question is rejected for the majority of the test objects.

Table 4.2 The Kruskal-Wallis tests on 30 runs of branch coverage in the test objects. Statistically significant differences at a level of significance of 0.05 are shown in bold type.

Test Object	Kruskal Wallis Test
a4j	**P < .05**
water-simulator	**P < .05**
Dsachat	**P < .05**
greencow	1
Petsoar	**P < .05**
Follow	0.12
Lilith	**P < .05**
Heal	**P < .05**
jgaap	**P < .05**
NumericalCaseStudy	0.08
StringCaseStudy	**P < .05**

However, Kruskal-Wallis test is an omnibus test statistic (Kruskal & Wallis, 1952). It does not indicate which particular functions are statistically significantly different from the others; it only indicates that at least one fitness function was different. To understand where exactly these differences lie (i.e. between which fitness functions), we must use a post-hoc test. In those test objects where the Kruskal-Wallis test was significant, pairwise comparisons were made using the Wilcoxon–Mann–Whitney (Mann & Whitney, 1947) test, to determine where significant differences in the coverage obtained were detected. Wilcoxon–Mann–Whitney is a non-parametric test for comparing two independent groups, based on ranking.

However, performing several Wilcoxon–Mann–Whitney tests has a tendency to inflate the Type I error. To compensate for this error inflation, a Bonferroni adjustment (Bonferroni,

1936) was made to adjust the level of significance at which the Wilcoxon–Mann–Whitney test would be run. The adjusted level of significance, α is computed as follows:

$$\alpha = \frac{p}{\mathrm{z}} \tag{4.1}$$

where *z* is the number of pairwise tests and *p* is the level of significance.

Let *F* be the number of search schemas to be tested, whereby the number of pairwise comparisons, *z* is calculated as follows:

$$z = \frac{F(F-1)}{2} \tag{4.2}$$

Hence, the Wilcoxon–Mann–Whitney tests were run with the adjusted *p* level, α .003. Table 4.3 demonstrates the Wilcoxon–Mann–Whitney test calculations.

Table 4.3 Pairwise comparison of medians with respect to coverage using a Wilcoxon–Mann–Whitney test. Statistically significant differences at a level of 0.003 are shown in bold type.

	a4j				
	BDF	**CFF**	**COM1**	**COM2**	**RT**
CLF	**<0.003**	0.47	0.21	**<0.003**	**<0.003**
BDF		**<0.003**	0.04	0.25	0.07
CFF			0.23	**<0.003**	**<0.003**
COM1				0.07	0.01
COM2					0.12

	water-simulator				
	BDF	**CFF**	**COM1**	**COM2**	**RT**
CLF	0.10	0.10	0.33	0.21	**<0.003**
BDF		0.49	0.20	0.24	**<0.003**
CFF			0.19	0.24	**<0.003**
COM1				0.29	**<0.003**
COM2					**<0.003**

	Dsachat				
	BDF	**CFF**	**COM1**	**COM2**	**RT**
CLF	0.01	0.28	0.36	0.20	0.42
BDF		**<0.003**	**<0.003**	**<0.003**	0.00
CFF			0.35	0.32	0.28
COM1				0.21	0.39
COM2					0.19

	petsoar				
	BDF	**CFF**	**COM1**	**COM2**	**RT**
CLF	0.46	0.36	0.36	0.00	**<0.003**
BDF		0.42	0.26	**<0.003**	**<0.003**
CFF			0.40	0.01	**<0.003**
COM1				0.03	**<0.003**
COM2					**<0.003**

	lilith				
	BDF	**CFF**	**COM1**	**COM2**	**RT**
CLF	**<0.003**	**<0.003**	0.05	0.48	0.19
BDF		**<0.003**	**<0.003**	**<0.003**	**<0.003**
CFF			**<0.003**	**<0.003**	**<0.003**
COM1				0.01	**<0.003**
COM2					0.18

	heal				
	BDF	**CFF**	**COM1**	**COM2**	**RT**
CLF	0.19	0.38	0.07	0.03	0.27
BDF		0.20	**<0.003**	**<0.003**	0.02
CFF			0.01	**<0.003**	0.09
COM1				0.33	0.11
COM2					0.06

	jgaap				
	BDF	**CFF**	**COM1**	**COM2**	**RT**
CLF	**<0.003**	0.02	0.32	0.38	0.42
BDF		**<0.003**	**<0.003**	0.03	**<0.003**
CFF			0.02	0.01	0.05
COM1				0.26	0.47
COM2					0.29

	StringCaseStudy				
	BDF	**CFF**	**COM1**	**COM2**	**RT**
CLF	**<0.003**	0.06	0.01	0.04	0.28
BDF		0.49	0.17	0.36	0.00
CFF			0.16	0.42	0.02
COM1				0.17	0.00
COM2					0.02

Significant differences are more evident in some test objects than others. For example, in the 'lilith' object, the Mann–Whitney analysis indicates a statistical significance in the difference between the performances of almost all the search schemes, while for the 'StringCaseStudy' object, only one pair comparison out of fifteen showed significant differences. For the object, 'water-simulator', there is an insignificant difference between the performance of the functions. Hence, the coverage produced from a certain function is not the same across all the test objects. For example, while these functions offer significantly different coverage of some objects, they may perform the same in others. This poses the question whether the internal structure of objects is responsible for this inconsistent performance of the functions across those objects.

4.3.2 Discussion on the Analysis of the First Question

The ultimate aim in automated test case generation is to achieve ideal coverage (i.e. 100%), as this will lead to a higher probability of finding faults in the software. Branch coverage of less than 100% indicates that some branches are difficult to reach. Uncovered branching statements could be infeasible for different reasons, such as private methods, dead code, and abstract methods, as also observed by Goldberg, Wang and Zimmerman (1994). For instance, private methods cannot be directly accessed in the generated test suites; it needs to be invoked in the class's public methods. Dead code may produce infeasible branches or unreachable methods, such as abstract class methods, or abstract methods overridden in all concrete subclasses.

We observed that the performance of the fitness functions varied according to the test object used. This means that the performance of the fitness functions was problem-dependent and suggests that the coverage obtained by a particular fitness function is likely to be influenced by the features of the test object at hand. This led us to investigate how strongly the performance of different functions depends on the objects being tested. To this end, we measured features of the test objects and explored their relationship with the coverage achieved using the different fitness functions.

An important observation which can be made is that RT produced similar results to the more sophisticated GA implementations. These findings are in line with Shamshiri et al. (2015), but in contrast with the results found by Fraser and Arcuri (2014), who report the superior

performance of GA over RT. These conflicting results suggest that the performance of the automated testing technique is problem-dependent.

4.3.3 Answer to the Second Research Question

RQ2: **Is there a correlation between software metric values and the coverage obtained by a given fitness function?**

The second research question examines the relationship between the performance of fitness functions and the code metric values of the test objects. A correlation analysis was performed using Spearman's rank-order correlation coefficient (Sheskin, 2007) for each pair of <the coverage given by a fitness function, code metric values>. Hypotheses were formulated, such that the null hypothesis makes no assumption of a correlation, as follows:

Hypothesis 0 (H0): There is no correlation between software metric values and the coverage obtained by a given fitness function.

Hypothesis 1 (H1): There is a correlation between software metric values and the coverage obtained by a given fitness function.

Spearman's rank-order correlation coefficient is a non-parametric technique for measuring the degree and strength of a relationship between two variables. The values of these variables are ranked according to both variables and then the rankings are correlated, thus minimising the impact of any nonlinear relationships between the two variables. We decided to use Spearman's correlation coefficient (and not the more common Pearson correlation), since this correlation measurement makes no assumptions about the underlying data distribution and is independent of the nature of the relationship between the two variables (Gravetter & Wallnau, 2013).

The numerical coefficient 'rs' ranges between [-1:+1], in which: rs > 0 indicates a positive correlation; rs < 0 indicates negative correlation (or correlation in the reverse direction), and rs = 0 implies no correlation. The higher the 'rs' value, the stronger the relationship between the two variables. The strength of the relationship is interpreted according to the following categories (Christmann & Badgett, 2009): very weak in the range [0.000 to 0.200], weak in the range [0.201 to 0.400], moderate in the range [0.401 to 0.600], strong in the range [0.601 to 0.800] and very strong in the range [0.801 to 1.000]. Besides the value of coefficient 'rs',

Spearman's rank-order correlation test gives the significance level *p* in each correlation, indicating the probability that the correlation is due to chance.

Before applying Spearman's test, all the metric values are normalised in the range [0:1], thus allowing a fair comparison of variables coming from different ranges. The values are normalised using the following equation (Everitt & Skrondal, 2002):

$$Zi = \frac{x_i - \min(X)}{\max(X) - \min(X)} \tag{4.3}$$

where $X=(x_1, x_2,...,x_n)$: the value corresponding to metric *i*, min *(X)* is the smallest value measured for metric *I*; max *(X)* is the maximum measured value for metric *i*: x_i is a metric value for a test object, and Z_i is the the normalised value. Table 4.4 shows the normalised code metric values.

Table 4.4 Code metric values after being normalised in the range [0:1]

Test Objects	HALL	NOA	NOM	WMC	NCL	NSA	NSM	NTG	TLOC	DIT	NOC	CBO	FRC	LCOM
a4j	0.181	0.178	0.207	0.137	0.083	0.002	0.000	1.000	0.098	0.000	0.000	0.055	0.159	0.392
water-simulator	0.168	0.562	0.153	0.106	0.136	0.242	0.009	0.012	0.234	0.105	0.158	0.198	0.140	0.020
dsachat	0.798	0.143	0.099	0.105	0.056	0.295	0.044	0.038	0.097	0.039	0.059	0.079	0.097	0.032
greencow	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
petsoar	0.036	0.196	0.200	0.116	0.179	0.040	0.097	0.108	0.091	0.434	0.644	0.265	0.245	0.054
follow	0.824	0.145	0.172	0.103	0.151	0.134	0.150	0.010	0.151	0.303	0.396	0.227	0.158	0.154
lilith	1.000	1.000	1.000	1.000	1.000	1.000	0.796	0.900	1.000	1.000	0.861	1.000	0.808	0.951
heal	0.665	0.474	0.661	0.692	0.384	0.313	1.000	0.605	0.572	0.678	1.000	0.744	1.000	1.000
jgaap	0.130	0.089	0.036	0.027	0.040	0.000	0.009	0.015	0.029	0.086	0.079	0.050	0.030	0.002
NumericalCaseStudy	0.240	0.006	0.006	0.018	0.019	0.013	0.009	0.046	0.011	0.000	0.000	0.000	0.004	0.000
StringCaseStudy	0.095	0.005	0.013	0.043	0.021	0.002	0.018	0.052	0.019	0.000	0.000	0.000	0.009	0.000

Table 4.5 below depicts the values of Spearman's coefficient correlation 'rs' for each pair of <the coverage given by a fitness function, code metric values>. The significant correlations at 95% confidence level are identified in bold. For the significant correlations, we can reject the H0 – there is some correlation between the performance of the fitness function and the code metric value; 41 pairs of metrics, among a total of 84, have a significant correlation between them and thus, have the null hypothesis rejected. The strength of these correlations is discussed next for each class of metrics.

Table 4.5 Spearman's rank-order correlation coefficient (rs) in the top Table, and significance (p) in the bottom Table. Statistically significant correlations at a level of 0.05 are shown in bold type.

	HALL	NOA	NOM	WMC	NCL	NSA	NSM	NTG	TLOC	DIT	NOC	CBO	FRC	LCOM
CLF	-0.0091	-0.6545	-0.5000	-0.6273	-0.6000	-0.5388	-0.4828	-0.3455	-0.5636	-0.5629	-0.5909	-0.5597	-0.5727	-0.4273
BDF	-0.1636	-0.7636	-0.6000	-0.7182	-0.6909	-0.6530	-0.5380	-0.3909	-0.6909	-0.6606	-0.6699	-0.6606	-0.6455	-0.5182
CFF	-0.2455	-0.7909	-0.6545	-0.7636	-0.7364	-0.6895	-0.5839	-0.4636	-0.7182	-0.7072	-0.6886	-0.7065	-0.6727	-0.5545
COM 1	-0.0818	-0.7364	-0.5727	-0.6909	-0.6727	-0.6073	-0.5196	-0.3727	-0.6364	-0.6420	-0.6606	-0.6422	-0.6364	-0.4909
COM 2	-0.1091	-0.7273	-0.6091	-0.7182	-0.7000	-0.6256	-0.5839	-0.4273	-0.6455	-0.6699	-0.6979	-0.6698	-0.6818	-0.5455
RT	0.0455	-0.6182	-0.5364	-0.6455	-0.6364	-0.5023	-0.5564	-0.4364	-0.4909	-0.6002	-0.6281	-0.5964	-0.6273	-0.4636

Correlation coefficient (rs)

	HALL	NOA	NOM	WMC	NCL	NSA	NSM	NTG	TLOC	DIT	NOC	CBO	FRC	LCOM
CLF	0.9788	**0.0289**	0.1173	**0.0388**	0.0510	0.0872	0.1325	0.2981	0.0710	0.0714	0.0556	0.0734	0.0655	0.1899
BDF	0.6307	**0.0062**	0.0510	**0.0128**	**0.0186**	**0.0294**	0.0878	0.2345	**0.0186**	**0.0269**	**0.0241**	**0.0269**	**0.0320**	0.1025
CFF	0.4669	**0.0037**	**0.0289**	**0.0062**	**0.0098**	**0.0189**	0.0593	0.1509	**0.0128**	**0.0149**	**0.0191**	**0.0151**	**0.0233**	0.0767
COM 1	0.8110	**0.0098**	0.0655	**0.0186**	**0.0233**	**0.0475**	0.1014	0.2589	**0.0353**	**0.0332**	**0.0269**	**0.0331**	**0.0353**	0.1252
COM 2	0.7495	**0.0112**	**0.0467**	**0.0128**	**0.0165**	**0.0395**	0.0593	0.1899	**0.0320**	**0.0241**	**0.0169**	**0.0242**	**0.0208**	0.0827
RT	0.8944	**0.0426**	0.0890	**0.0320**	**0.0353**	0.1154	0.0755	0.1797	0.1252	0.0509	**0.0385**	0.0528	**0.0388**	0.1509

Significance (p) value

4.3.3.1 Size-related Measures

Size-related measures include Halsted length (HALL); number of attributes (NOA); number of classes (NCL); number of static methods (NSM); number of static attributes (NSA); number of methods in a class (NOM); weight of method (WMC); number of test goals (NTG), and total line of code (TLOC). Most of these metrics are negatively correlated with the coverage obtained from the fitness function, suggesting that large test objects are harder to cover.

Halsted length (HALL) counts the total number of arithmetical operators, numerics, constants and language keywords, such as 'return' and 'continue'. There is a lack of evidence to suggest a correlation between HALL and any of the fitness functions. Unlike the results for HALL, the values of coefficient 'rs' for number of attributes (NOA) suggest a very strong negative correlation between NOA and the coverage achieved by all functions. A possible explanation is given by the definition of NOA. The test object variables require initialisation before testing can be performed. A higher number of variables will mean a longer time required to construct a test case, resulting in high execution costs.

A class is considered as the heart of OO languages. There is evidence of a statistically significant correlation between number of classes (NCL) metric values and between the performances of all fitness functions, except for CLF. Here, Spearman's correlation coefficient is in the range [-0.73, -0.63], which suggests a strong negative correlation. The coverage produced by a test case is likely to be influenced by the number of classes.

For number of static methods (NSM), there is a lack of evidence to suggest a correlation between NSM and the performance of the functions used. In contrast, number of static attributes (NSA) metric values correlate with the coverage obtained by all the functions, except for CLF; the more static the attributes, the lower the capacity of the functions to direct the progress of the research towards full coverage. An example which will explain such correlation is when static attributes are used as a timer in the code. In this case, to cover a branching statement that has a timer, it is required to generate an input sequence of that static attribute to cover the branch. For instance, 'branching statement 2' in Figure 4.2 requires the execution of the function *static_Example* at least seven times for the true branch of 'branching statement 2' to become feasible, since the value of the static attribute, called *counter*, will be retained at the end of the method call until the next time it is executed. In this

case, plateaux will form on the fitness landscape, because the search goal is not satisfied until the method executes at least seven times for the predicate of ‘branching statement 2’ to be true.

```
const int maximum   = 7;
static int counter = 0;
bool static_Example (int a, int b, int c)
{
        if (counter < maximum) // branching statement 1
                counter ++;
        if (counter >= maximum ) // branching statement 2
                return True;
        return False;
}
```

Figure 4.2 An example of static attributes

The number of methods in a class (NOM) metric only correlates strongly with the performance of CFF and COM2 functions, whereas the weight of method (WMC) metric values strongly correlate with coverage obtained by all fitness functions. The weight of the method is proportionate to the number of possible alternative paths through the code. In contrast to WMC, there is a lack of evidence to suggest a correlation between number of test goals (NTG) and any of the fitness functions. NTG is in fact proportioned to the number of branches in a test object. The results obtained for WMC and NTG suggest that it is not the number but rather the positioning of the branches in a test object which will primarily influence performance; for example, if two test objects have the same number of branches, but differ in the positioning of the branches, such that the first object has branches in linear order, while the other has branches in a nested structure. In this case, the latter object is the harder to cover. The results obtained from Spearman's rank correlation for these two metrics confirm this explanation.

The last metric to be discussed in this group is total line of code (TLOC). The results show a statistically significant correlation between TLOC and the coverage achieved by the fitness functions, except for CLF. This correlation may explain the 100% absent coverage achieved by the functions for the test objects, except for ‘greencow’: a trivial test object consisting of only of 13 lines of code. While the ‘greencow’ object was easy to cover, the larger test objects were more difficult in terms of reaching a high level of coverage.

To sum up, the larger a test object in terms of the number of attributes, number of classes, number of possible alternative paths (higher WMC) and number of lines of code, the more difficult it is to obtain high coverage using the functions presented here.

4.3.3.2 Complexity-Related Measures

The second group of measures evaluates the complexity of the test objects. This group includes depth of inheritance (DIT); number of children (NOC); coupling between object classes (CBO); response for a class (RFC), and lack of cohesion of methods (LCOM).

Two metrics of the complexity-related measures deal with inheritance: DIT expresses the maximum inheritance path from a class to the root class, whereas NOC indicates the number of direct subclasses of a class within the class. Both metrics are strongly correlated to all the functions. A possible explanation for this might be that a test object with a longer inheritance path tends to contain a higher number of inherited attributes, methods and children classes. In this case, it might become harder for the functions presented to guide the search progress in exercising such complex structures.

CBO and RFC metrics measure dependencies on external classes and methods. CBO expresses the number of classes to which a class is coupled. A class is considered coupled to another class if it calls its method or accesses its instance variables. RFC represents the set of methods which can potentially be invoked in response to a message received by an object from the class. These two metrics are significantly correlated with all fitness functions. When testing a class with dependencies on other classes or methods, the attributes of these classes be initialised before they are used. Thus, the test will not only include the class being tested, but may also include the coupled classes and dependent methods from other classes. Therefore, the amount of initialisation required before testing will influence the search budget.

The last metric to be discussed in this group is lack of cohesion of methods (LCOM). While CBO measures how classes interact with each other, LCOM focuses on how a single class is designed. LCOM measures how closely related the methods of a class are to each other. The results show that LCOM does not correlate with the coverage obtained by any of the functions presented. LCOM only considers methods and instance variables implemented in the class, whereas inherited ones are excluded. Thus, the results show that cohesion does not

act as a critical influence on the performance of the functions presented, while coupling and inherence do.

While the above complexity-related metrics correlate with all the functions presented, there is a lack of evidence to confirm such a correlation with CLF. CLF rewards test suites on the basis of the execution of branching statements, as discussed in Chapter 3. Therefore, it does not exploit information from the code regarding its inheritance, cohesion or coupling. As this function mainly counts the number of covered branches, it does not differentiate whether they are present in a nested structure, such as multiple inheritance, or a simple class. Therefore, the performance of CLF does not correlate with these metrics.

In summary, complexity-related metrics: DIT, NOC, CBO, FRC can be seen as an indicator of the possible difficulty for the functions presented to guide the search process in covering a given object.

4.3.4 Discussion on the Analysis of the Second Question

Based on the experiments, the relationship between code measures and function performance will help us understand whether high coverage is possible for certain software systems. The results show that each fitness function presented is correlated with some of the measures and the strength of this correlation varied. This knowledge provides a better understanding of how the fitness functions utilised could be influenced by the internal structure of a test object. For instance, if a test object contains a large number of methods, it is hard to expect that CFF or COM2 functions will guide the search to obtain higher coverage, as the performance of these functions is negatively correlated with the number of methods. In this case, other functions are recommended, which are not influenced by the application of a high number of methods, such as COM1.

Spearman's rank correlation test is used to investigate the correlation between the performance of the fitness functions and the code measurement. However, it is important to note that a correlation relationship does not mean a cause and effect relationship. In other words, the above results do not suggest that any of the code measures give rise to the performance obtained, or vice versa. The correlations permit us to conclude that is possible to observe certain testability indicators (based on the degree of coverage obtained) through the measurements computed from a test object's internal structure. A cause and effect relationship is subject to further future study.

The correlation between NOM and the coverage obtained by the fitness functions needs further examination, in order to give a better understanding of the impact of the type of method: private, public, or overridden. The analysis of the impact of private and overridden methods on the achievement of coverage is important, since both types of method could increase the probability of low coverage, as discussed above in Section 4.3.2.

4.4 Threats to Validity

There are threats to validity in any empirical study, the current one included. This section presents a brief overview of threats to validity and how they have been addressed. One potential source of threat to the *internal validity* of this research stems from the experimental design. When comparing different functions, it is essential to ensure that the comparison is as reliable as possible. One potential source of bias consists of the settings used for setting up the experiments. Therefore, care was taken to ensure that all parameters which do not reflect the experimental procedure were kept constant during all the experiments.

Another threat to *internal validity* might come from using metaheuristics, since they are stochastic by nature. To mitigate this threat, all experiments were repeated 30 times and rigorous statistical procedures were followed to evaluate the results. To examine the performance of one function compared to the other, a test for statistically significant differences in the mean of the coverage was performed. Care was taken to first check the distribution of the data, in order to ensure the most suitable statistical test was selected for data analysis.

A source of bias may include the choice of test objects used in the empirical study, which could potentially affect its *external validity*, i.e. the extent to which it is possible to generalise from the results obtained. The wide range of software types makes it impossible to cover all possible kinds of software. However, where possible, a variety of software type and sources were used. The study draws upon codes from open source projects and case studies from the literature. This has resulted in a total of 1,109 classes, 5,866 methods, 85,562 lines of code and 12,343 branch statements, producing a large pool of results from which to make observations. Nevertheless, we acknowledge that the results may not be generalised to other automated testing techniques, programming languages, or paradigms.

Threats to *construct validity* may be related to the way the performance of fitness functions is assessed. Branch coverage was used as a test criterion to evaluate the quality of the test suites generated. However, the results may be different for other test suite properties, such as the size of the test suite or the length of the test cases. Whether these properties are negatively related with branch coverage is a matter for further study.

4.5 Summary

This chapter has introduced and evaluated results collected from 1,980 experiments. The results obtained from the six search schemas suggest that the effectiveness of a fitness function is problem-dependent. We were able to demonstrate a significant correlation between code measures (most notably NOA, WMC, DIT, NOC, CBO, and RFC) and the performance of fitness functions. In the next chapter, conclusions to this thesis and recommendations for further research are presented.

Chapter 5: CONCLUSION

5.1 Research Summary and Contributions

The success of applying automated test data generation largely depends on how the definition of the fitness function accurately represents the test aim. For many years, researchers have been proposing different definitions of fitness functions for automatically generating test data. In this study, the performance of five fitness functions are investigated: coverage level function (CLF), branch distance function (BDF), control fitness function (CFF), combined fitness function 1 (COM1), and combined fitness function 2 (COM2). These functions utilise different measurements to evaluate the quality of the test data produced. CLF uses structural coverage measurements orignally proposed by Roper (1997), whereas BDF employs branch distance measures. In CFF, The fitness value is equivalent to the number of successful statement executions through the code towards the target branch. Both COM1 and COM2 functions combine more than one measure to evaluate the fitness value.

The empirical study was performed on a total of 85,562 lines of code and 12,343 branch statements. The results of the study show there was no fitness function that outperformed all the other functions in all test objects; each function in fact provided diverse coverage over the test objects, i.e. while providing best performance on some test objects, functions were outperformed in others. Depending on the test objects, the worst performing function also varied. This indicates that the performance of a fitness function is problem-dependent, leading to investigation into whether the coverage obtained is impacted by features of software systems.

Identifying those features of software systems which make the generation of test cases difficult will help us understand the suitability and limitations of definitions of fitness functions. Thus, one of the main tasks of this study was to measure software features and explore their relationship with the coverage achieved by the fitness functions. This research has aimed to provide a starting point for using software metrics to pinpoint the relationship between the performance of a fitness function and the characteristics of the software being tested. We looked at the following software metrics: Halsted length (HALL); number of attributes (NOA); number of classes (NCL); number of static methods (NSM); number of static attributes (NSA); number of methods in a class (NOM); weight of method (WMC);

number of test goals (NTG); total line of code (TLOC); depth of inheritance (DIT); number of children (NOC); coupling between object classes (CBO); response for a class (RFC), and lack of cohesion of methods (LCOM).

The study showed that software metrics are useful as an indicator of the possibility of reaching high coverage using a given fitness function. The results show that the larger a test object in terms of NOA, NCL, number of possible alternative paths (higher WMC) and TLOC, the more difficult it is to obtain high coverage using any of the functions presented. Testing those test objects which proved difficult to cover using a given fitness function should be enhanced by considering other techniques; for instance, by increasing the search budget for the test.

The results also show that it is not the number of branches, but rather the positioning of the branches in a test object which will primarily influence the function's performance. Branches present in nested structures will increase the number of possible paths through the code, leading to low coverage of the code. Testing objects with a high number of possible alternative paths should be enhanced by using adequate code transformation; for instance, transforming the code to a simpler version with fewer alternative paths, yet maintaining the functionality of the software. Thus, these challenging characteristics (in this case higher alternative paths) will no longer have a dramatic effect on the performance of the fitness functions.

Investigating influences resulting from the characteristics of test objects plays a decisive role in determining the coverage level provided by a given fitness function. When the performance of a given function is hindered by the presence of certain software features, it is hard to expect that function to produce test suites with high coverage. This facilitates a deeper understanding of the strengths and weakness of the fitness functions, with implications for improved definitions of such functions. The fitness function of the structural test needs to be further improved in the light of the characteristics of the test objects.

Ideally, for a new software, different fitness functions should be compared, in order to be able to select the function which is more likely to provide high coverage. Ultimately, fitness function selection should be automated, so that the most appropriate fitness function is utilised for a given test object. It is expected that the extra computational overheads will pay off and more research is hoped for in this direction.

Nevertheless, the findings of this study should be regarded as exploratory rather than conclusive, as they show how it is possible to find a significant relationship between software characteristics (through code metrics commonly used to evaluate size and complexity) and the coverage obtained by the fitness function. While these results are insightful, they are still not generalisable. This study contributes to the research direction which focuses on answering a question which has intrigued researchers for some time; about how we can determine when a test object type is manageable for specific automated testing techniques.

5.2 Future Work

The investigation could be extended to other search algorithms, such as hill climbing (HC) and simulated annealing (SA). A broader understanding would be gained of how performance may be influenced across different search techniques when utilising different definitions of fitness functions.

This study has focused on structural testing, wherein test cases are defined on the basis of internal program structure. A study similar to this one should be carried out on functional testing techniques, whereby test cases are produced, depending on the software requirements and specifications and without any knowledge of the internal structure of the software.

Another possible area of future research would be to build a machine-learning model that uses the software metric values applied in this study to predict the test coverage that could be achieved by different fitness functions. The coverage obtained by the functions and the software metric values measured from the test objects investigated in this study would be used as input for specific machine-learning techniques. Examples of these techniques include Bayesian networks and decision trees. Ultimately, automated fitness function selection is of potential benefit to software testers and developers.

References

Ali, S., Briand, L.C., Hemmati, H. & Panesar-Walawege, R.K. (2010). A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering, 36*(6), 742-762.

Arcuri, A. (2013). It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability, 23*(2), 119-147.

Arcuri, A. & Briand, L. (2011). *Adaptive random testing: An illusion of effectiveness?* Paper presented at the Proceedings of the 2011 International Symposium on Software Testing and Analysis.

Arcuri, A. & Briand, L. (2012). Formal analysis of the probability of interaction fault detection using random testing. *IEEE Transactions on Software Engineering, 38*(5), 1088-1099.

Arcuri, A. & Briand, L. (2014). A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability, 24*(3), 219-250.

Arcuri, A. & Fraser, G. (2013). Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering, 18*(3), 594-623.

Arcuri, A., Iqbal, M.Z. & Briand, L. (2010). Black-box system testing of real-time embedded systems using random and search-based testing. *Testing Software and Systems* (pp. 95-110): Springer.

Arcuri, A. & Yao, X. (2008). Search based software testing of object-oriented containers. *Information Sciences, 178*(15), 3075-3095.

Bansiya, J. & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering, 28*(1), 4-17.

Baresel, A. & Sthamer, H. (2003). *Evolutionary testing of flag conditions.* Paper presented at Genetic and Evolutionary Computation Conference (GECCO 2003)..

Barkmann, H., Lincke, R. & Lowe, W. (2009). *Quantitative evaluation of software quality metrics in open-source projects.* Paper presented at the International Conference on Advanced Information Networking and Applications Workshops (WAINA'09).

Basili, V.R., Briand, L.C. & Melo, W.L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering, 22*(10), 751-761.

Bhatti, H.R. (2011). *Automatic Measurement of Source Code Complexity.* Master's thesis, Lulea University of Technology, Lulea, Sweden, 2011 [12] Милютин А.,«Метрики кода программного обеспечения» http://www. viva64. com/ru/a/0045.

Binder, R. (2000). *Testing object-oriented systems: Models, patterns, and tools*: Addison-Wesley Professional.

Bland, J.M. & Altman, D.G. (1995). Multiple significance tests: the Bonferroni method. *BMJ, 310*(6973), 170.

Briand, L.C., Morasca, S. & Basili, V.R. (1996). Property-based software engineering measurement. *IEEE Transactions on Software Engineering, 22*(1), 68-86.

Bruntink, M. & van Deursen, A. (2006). An empirical study into class testability. *Journal of Systems and Software, 79*(9), 1219-1232.

Chidamber, S.R. & Kemerer, C.F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476-493.

Christmann, E.P. & Badgett, J.L. (2009). *Interpreting assessment data: Statistical techniques you can use*: NSTA Press.

Collet, P. & Rennard, J.-P. (2007). Stochastic optimization algorithms. *arXiv preprint arXiv:0704.3780*.

Corder, G.W. & Foreman, D.I. (2009). *Nonparametric statistics for non-statisticians: A step-by-step approach*: John Wiley & Sons.

D'Ambros, M., Lanza, M. & Robbes, R. (2012). Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering, 17*(4-5), 531-577.

Daniel, B. & Boshernitsan, M. (2008). *Predicting effectiveness of automatic testing tools.* Paper presented at 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008. (ASE 2008).

Dreo, J. & Siarry, P. (2007). Stochastic metaheuristics as sampling techniques using swarm intelligence. *I-Tech Education and Publishing*: December.

Abreu, F.B. & Carapuça, R. (1994). Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software, 26*(1), 87-96.

Efron, B. (1969). Student's t-test under symmetry conditions. *Journal of the American Statistical Association, 64*(328), 1278-1302.

Everitt, B.S. & Skrondal, A. (2002). *The Cambridge Dictionary of Statistics*. Cambridge University Press: Cambridge.

Fenton, N. & Bieman, J. (2014). *Software metrics: A rigorous and practical approach*: CRC Press.

Fraser, G. & Arcuri, A. (2011). *EvoSuite: Automatic test suite generation for object-oriented software.* Paper presented at the Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering.

Fraser, G. & Arcuri, A. (2013a). 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 1-29.

Fraser, G. & Arcuri, A. (2013b). Whole test suite generation. *IEEE Transactions on Software Engineering, 39*(2), 276-291.

Fraser, G. & Arcuri, A. (2014). A large-scale evaluation of automated unit test generation using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM), 24*(2), 8.

Fraser, G. Arcuri, A. & McMinn, P. (2013). *Test suite generation with memetic algorithms.* Paper presented at the Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation.

Goldberg, A., Wang, T.-C. & Zimmerman, D. (1994). *Applications of feasible path analysis to program testing.* Paper presented at the Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis.

Gravetter, F. & Wallnau, L. (2013). *Essentials of statistics for the behavioral sciences*: Cengage Learning.

Gross, H., Kruse, P.M., Wegener, J. & Vos, T. (2009). *Evolutionary white-box software test with the evotest framework: A progress report.* Paper presented at the International Conference on.Software Testing, Verification and Validation Workshops, 2009 (ICSTW'09).

Gupta, A. & Jalote, P. (2008). An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. *International Journal on Software Tools for Technology Transfer, 10*(2), 145-160.

Halstead, M.H. (1977). *Elements of Software Science (Operating and Programming Systems Series)*: Elsevier Science Inc.

Harman, M., Mansouri, S.A. & Zhang, Y. (2009). Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Department of Computer Science, King's College London, *Tech. Rep. TR-09-03*.

Harman, M. & McMinn, P. (2007). *A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation.* Paper presented at the Proceedings of the 2007 International Symposium on Software Testing and Analysis.

Harman, M. & McMinn, P. (2010). A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering, 36*(2), 226-247.

Harrison, R. & Samaraweera, L. (1996). Using test case metrics to predict code quality and effort. *ACM SIGSOFT Software Engineering Notes, 21*(5), 78-88.

Harrison, W., Magel, K., Kluczny, R. & DeKock, A. (1982). Applying software complexity metrics to program maintenance. *Computer, 9*(15), 65-79.

Hooker, J.N. (1995). Testing heuristics: We have it all wrong. *Journal of Heuristics, 1*(1), 33-42.

Hutchins, M., Foster, H., Goradia, T. & Ostrand, T. (1994). *Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria.* Paper presented at the Proceedings of the 16th International Conference on Software Engineering.

Ihaka, R. & Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics, 5*(3), 299-314.

Johnson, D.S. (2002). A theoretician's guide to the experimental analysis of algorithms. *Data Structures, near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges, 59*, 215-250.

Kruskal, W.H. & Wallis, W.A. (1952). Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association, 47*(260), 583-621.

Lakhotia, K. (2009). *Search–Based Testing.* King's College London.

Lammermann, F., Baresel, A. & Wegener, J. (2008). Evaluating evolutionary testability for structure-oriented testing with software measurements. *Applied Soft Computing, 8*(2), 1018-1028.

Lee, Y. (2007). *Automated source code measurement environment for software quality*: ProQuest.

Lefticaru, R. & Ipate, F. (2008). *A comparative landscape analysis of fitness functions for search-based testing.* Paper presented at the 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2008 (SYNASC'08). .

Li, W. & Henry, S. (1993a). *Maintenance metrics for the object oriented paradigm.* Paper presented at the First International Software Metrics Symposium, 1993. .

Li, W. & Henry, S. (1993b). Object-oriented metrics that predict maintainability. *Journal of Systems and Software, 23*(2), 111-122.

Lincke, R., Gutzmann, T. & Löwe, W. (2010). *Software quality prediction models compared.* Paper presented at the 10th International Conference on Quality Software (QSIC), 2010.

Lincke, R., Lundberg, J. & Löwe, W. (2008). *Comparing software metrics tools.* Paper presented at the Proceedings of the 2008 International Symposium on Software Testing and Analysis.

Löwe, W., Ericsson, M., Lundberg, J., Panas, T. & Pettersson, N. (2003). *Vizzanalyzer-a software comprehension framework.* Paper presented at the Third Conference on Software Engineering Research and Practise in Sweden, Lund University, Sweden.

Mann, H.B. & Whitney, D.R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 50-60.

Maragathavalli, P. (2011). Search-based software test data generation using evolutionary computation. *arXiv preprint arXiv:1103.0125*.

Marco, L. (1997). Measuring software complexity. *Enterprise Systems Journal (April).*

McCabe, T.J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, (4), 308-320.

Narasimhan, V.L., Parthasarathy, P. & Das, M. (2009). Evaluation of a suite of metrics for component based software engineering (CBSE). *Issues in Informing Science and Information Technology, 6*(5/6), 731-740.

Nogueira, A.F. (2012). *Predicting software complexity by means of evolutionary testing.* Paper presented at the Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering.

Nogueira, A.F., Ribeiro, B., Carlos, J. & Zenha-Rela, M. (2014). *On the Evaluation of Software Maintainability Using Automatic Test Case Generation.* Paper presented at the 9th International Conference on the Quality of Information and Communications Technology (QUATIC), 2014..

Pacheco, C. & Ernst, M.D. (2007). *Randoop: Feedback-directed random testing for Java.* Paper presented at the Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion.

Panas, T., Lincke, R., Lundberg, J. & Lowe, W. (2005). *A qualitative evaluation of a software development and re-engineering project.* Paper presented at the 29th Annual IEEE/NASA Software Engineering Workshop, 2005..

Pargas, R.P., Harrold, M.J. & Peck, R.R. (1999). Test-data generation using genetic algorithms. *Software Testing Verification and Reliability, 9*(4), 263-282.

Riaz, M., Mendes, E. & Tempero, E. (2009). *A systematic review of software maintainability prediction and metrics.* Paper presented at the Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement.

Roper, M. (1997). Computer aided software testing using genetic algorithms. *10th International Quality Week*, San Francisco.

Rosenberg, L.H. & Hyatt, L.E. (1997). Software quality metrics for object-oriented environments. *Crosstalk Journal, 10*(4).

Safe, M., Carballido, J., Ponzoni, I. & Brignole, N. (2004). On stopping criteria for genetic algorithms *Advances in Artificial Intelligence–SBIA 2004* (pp. 405-413): Springer.

Shamshiri, S., Rojas, J.M., Fraser, G. & McMinn, P. (2015). *Random or Genetic Algorithm Search for Object-Oriented Test Suite Generation?* Paper presented at the Proceedings of the 2015 Conference on Genetic and Evolutionary Computation.

Sharma, R., Gligoric, M., Arcuri, A., Fraser, G. & Marinov, D. (2011). Testing container classes: Random or systematic? *Fundamental Approaches to Software Engineering* (pp. 262-277): Springer.

Sheskin, D. (2007). Spearman's rank-order correlation coefficient. *Handbook of Parametric and Nonparametric Statistical Procedures*, 1353-1370.

Shrivastava, D.P. & Jain, R. (2010). Metrics for Test Case Design in Test Driven Development. *International Journal of Computer Theory and Engineering, 2*(6), 952-956.

Simons, C., Singer, J. & White, D.R. (2015). Search-based Refactoring: Metrics are not Enough *Search-Based Software Engineering* (pp. 47-61): Springer.

Sjøberg, D.I., Anda, B. & Mockus, A. (2012). *Questioning software maintenance metrics: a comparative case study*. Paper presented at the Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement.

Smith-Miles, K. & Lopes, L. (2012). Measuring instance difficulty for combinatorial optimization problems. *Computers & Operations Research, 39*(5), 875-889.

Srinivas, M. & Patnaik, L.M. (1994). Genetic algorithms: A survey. *Computer, 27*(6), 17-26.

Standard, I. (2005). Software engineering–software product quality requirements and evaluation (square)–guide to square. *ISO Standard, 25000*, 2005.

Strein, D., Lincke, R., Lundberg, J.& Lowe, W. (2007). An extensible meta-model for program analysis. *IEEE Transactions on Software Engineering, 33*(9), 592-607.

Stützle, T. & Fernandes, S. (2004). New benchmark instances for the QAP and the experimental analysis of algorithms *Evolutionary Computation in Combinatorial Optimization* (pp. 199-209): Springer.

Subramanyam, R. & Krishnan, M.S. (2003). Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering, 29*(4), 297-310.

Tang, M.-H., Kao, M.-H. & Chen, M.-H. (1999). *An empirical study on object-oriented metrics.* Paper presented at the Sixth International Software Metrics Symposium, 1999.

Tonella, P. (2004). Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes, 29*(4), 119-128.

Tracey, N.J. (2000). *A search-based automated test-data generation framework for safety-critical software*: Citeseer.

Wang, H.-C., Jeng, B. & Chen, C.-M. (2006). *Structural testing using memetic algorithm.* Paper presented at the Proceedings of the Second Taiwan Conference on Software Engineering.

Wegener, J., Baresel, A. & Sthamer, H. (2001). Evolutionary test environment for automatic structural testing. *Information and Software Technology, 43*(14), 841-854.

Wingkvist, A., Ericsson, M., Lincke, R. & Lowe, W. (2010). *A metrics-based approach to technical documentation quality.* Paper presented at the Seventh International Conference on the Quality of Information and Communications Technology (QUATIC), 2010..

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B. & Wesslén, A. (2012). *Experimentation in Software Engineering*: Springer Science & Business Media.