GENETIC ALGORITHM BASED

TEST DATA GENERATOR

by

Irman Hermadi

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

In Partial Fulfillment of the Requirements

for the degree of

MASTER OF SCIENCE

IN

INFORMATION & COMPUTER SCIENCE DEPARTMENT

KING FAHD UNIVERSITY

OF PETROLEUM & MINERALS

Dhahran, Saudi Arabia

May 2004

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by Irman Hermadi under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE.

Thesis Committee

_____
Dr. Moataz A. Ahmed (Advisor)

_____
Dr. Muhammad Alsuwaiyel (Member)

_____
Dr. Jarallah Alghamdi (Member)

_____
Dr. Kanaan Faisal
(Department Chairman)

_____
Dr. Mohammad Al-Ohali
(Dean of Graduate Studies)

_____
Date

DEDICATION

This thesis is lovingly dedicated to my parents:

Mrs. Hj. Hatianah Rustandi, for all I am started in her arms.

&

H. Didi Rustandi, for his fervent love for my education.

ACKNOWLEDGEMENTS

All thanks are due to Allah first and foremost for His countless blessing. Acknowledgment is due to King Fahd University of Petroleum & Minerals for supporting this research.

My unrestrained appreciation goes to my advisor, Dr. Moataz A. Ahmed, for all the help and support he has given me throughout the course of this work and on several other occasions. I simply cannot imagine how things would have proceeded without his help, support, and patience. I also wish to thank my thesis committee members, Dr. Muhammad Alsuwaiyel and Dr. Jarallah Alghamdi, for their help, support, and contributions.

I also acknowledge my many colleagues and friends as I had a pleasant, enjoyable and fruitful company with them.

Finally, I wish to express my gratitude to my family members for being patient with me and offering words of encouragements to spur my spirit at moments of depression.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

THESIS ABSTRACT

NAME: Irman Hermadi

TITLE: GENETIC ALGORITHM BASED TEST DATA GENERATOR

MAJOR FIELD: COMPUTER SCIENCE

DATE OF DEGREE: MAY 2004

Software testing is meant to increase confidence in the correctness of software. It is a laborious and time-consuming work; and spends almost a half of development resources. Generally, the testing goal is to reveal as many faults as possible, with a limitation on the number of test data to be used. The challenge, in this case, is in being able to minimize the number of test data while maximizing coverage. Obviously, automating the test data generation process is expected to significantly reduce the overall development cost. There are evidences that Genetic Algorithm (GA) has been successfully used in developing test data generators. However, there is no common ground for assessing and comparing these GA based test data generators. In this thesis, based on our critical survey, we present and use a set of attributes for assessing and comparing these generators. Our critical survey has revealed that existing GA-based test data generators suffer from some problems. This thesis presents our attempt to overcome one of these problems; that is the ability to deal with multiple target paths at one time. We have designed a GA based test data generator that is able to overcome this problem. Moreover, we have implemented a set of variations of the generator. Experimental results show that our test data generator is more powerful than others.

ملخص الرسالة

الاســـــــم : عرمان هيرمادي

العنـــــوان : مولد معطيات الاختبار المبني على الخوارزمية الجينية

التخـــصص : علوم الحاسب الآلي

تاريخ التخـــرج : مايو 2005

يقصد من اختبار البرامج زيادة الثقة بصحة تلك البرامج. فهي عملية شاقة ومضيعة للوقت؛ قد تستنفذ تقريباً نصف موارد عملية التطوير. بصورة عامة، إن هدف من الاختبار هو الكشف على أكبر قدر ممكن من الأخطاء مع استخدام عدد محدود من المعطيات. والتحدي في هذه الحالة هو المقدرة على انقاص عدد المعطيات مع زيادة توسيع تغطية الاختبار. فمن الواضح أن نتوقع عند ميكنة توليد معطيات الاختبار انخفاض التكلفة الإجمالية لتطوير البرمجيات بصورة ملحوظة. فهناك مؤشرات تبين نجاح استخدام الخوارزميات الجينية في توليد معطيات الاختبار.

في هذا البحث وبناءاً على دراسات نقدية، استعرضنا واستخدمنا مجموعة من الصفات لتقييم ومقارنة مولدات معطيات الاختبار. لقد أوضحت دراستنا النقدية أن المولدات الموجودة حالياً والمبنية على الخوارزمية الجينية تعاني من بعض المشكلات. هذا البحث يستعرض محاولاتنا للتغلب على إحدى تلك المعضلات وهي القدرة على التعامل مع عدة مسارات الهدف في آن واحد. لقد صممنا مولد معطيات الاختبار المبني على الخوارزمية الجينية له القدرة على التغلب على تلك المعضلة. وعلاوةً على ذلك، قمنا باجراء عدة تطبيقات مختلفة لهذا المولد. فلقد أظهرت النتائج المخبرية أن مولدنا أقوى من باقي المولدات الموجودة.

CHAPTER 1

INTRODUCTION

1.1. Software Testing

Software needs to be tested properly and thoroughly, such that any misbehavior during the runtime can be detected and fixed in advance, before its delivery. However, well-tested software is not guaranteed to be error-free or bug-free. Most of the problems reported by users are identified as execution of untested code. This is because either the order in which statements were executed in actual use differed from that during testing, a combination of untested input values are given, extreme inputs, or the user's environment was never tested [45].

Software testing is laborious and time-consuming work; it spends almost 50% of software system development resources [3][37][45]. Generally, the goal of software testing is to design a set of minimal number of test cases such that it reveals as many faults as possible. Testing, itself, is defined as the process of executing a program with the intent of finding errors. Hence, a pair of input and its expected output, which is called a *test case*, is said to be successful if it succeeds to uncover errors, and not vice versa. In other words, a good test case is one that has a high probability of detecting an as-yet undiscovered error

[45]. An input datum for a tested program, which is subset of a test case, is called *test datum*.

1.2. The Research Problem

As mentioned earlier, software testing is a lengthy and time-consuming work [38]. Absolutely, an automated software testing can significantly reduce the cost of developing software. Other benefits include: the test preparation can be done in advance, the test runs would be considerably fast, and the testing execution can be performed during night-shift and remotely [45]. The last but not the least is that the confidence of the testing result can be increased [1].

However, software-testing automation is not a straightforward process. For years, many researchers have proposed different methods to generate test data automatically, i.e. different methods for developing test data/case generators [5].

Commonly, searching for an input datum in a pool (domain/set) of possible input data is dealt with as an optimization problem [19]. In the early age of automation of software testing, most of the test data generators were using gradient descent algorithms. However, these algorithms were inefficient, and time-consuming, and could not escape from local optima in the search space of the domain of possible input data [25]. These issues necessitate the need to investigate the suitability of meta-heuristic search algorithms, e.g. simulated annealing, genetic algorithms, and ant colony optimization as a better

alternative for developing test data generators. However, up to our knowledge, so far, researchers have been only interested in using genetic algorithm to generate test data [43].

Wegener et al. have showed the suitability of using evolutionary algorithms in software testing [43]. Using evolutionary computations, researchers have done some work in developing genetic algorithms (GA)-based test data generators [6][43]. However, in trying to identify strengths and weaknesses of the various techniques available for developing test-data generators available, we found that there is no well-defined set of quality attributes that can be used to compare such various techniques.

Moreover, as discussed in Chapter 3, one of our observations over existing GA based test data generators is that they can generate only one test datum at a time. Accordingly, in trying to generate a set of test data (i.e., more than one test datum) to satisfy particular criteria (called test adequacy criteria, e.g. branch coverage) under consideration, the test-data generator should be used more than one time (one run for each required test datum). This practice, however, does not take advantage of the fact that some of the required test data can be readily available as by-products when trying to find other test data. This, hence, makes those existing test-data generators inefficient in trying to generate multiple test data. A detailed critical survey and review of existing approaches is demonstrated in Chapter 3.

Our motivation to investigate these problems derives from the benefits aforementioned.

1.3. Main Contributions

The main contributions of this thesis work are the following:

- Proposing a set of attributes for assessing and comparing GA-based test data generators;
- Comparing the available GA-based test data generators in light of the proposed set of attributes;
- Proposing a GA-based test-data generator that is capable of to generating multiple test data to cover multiple target paths[1] at one run;
- Implementing and comparing a number of variations of the proposed generator[2];
- Conducting experiments to demonstrate the strength of the proposed approach using Matlab.

---

[1] The *path coverage criterion* is concerned with the execution of (selected) paths in the program. We adopt the path coverage criteria since it achieves the utmost coverage [7][22][33][39]. Chapter 2 discusses the different test adequacy criteria in details, and provides justifications for adopting path coverage as an effective criterion.

[2] Each variation has a different form of the fitness function we propose. Chapter 2 gives all the necessary background on Genetic Algorithms and the role of the fitness function.

1.4. Organization of Thesis

The rest of the thesis is organized as follows. Chapter 2 gives a succinct background of software testing and genetic algorithms. Chapter 3 presents our scheme, composed of a set of attributes, for comparing and evaluating GA-based test data generators. It also presents an extensive critical literature survey of related works with a comparison based on our evaluation scheme. Chapter 4 describes the design details of the proposed fitness function along with its different variations. Chapter 5 discusses the experiments setup, results, and analysis. Chapter 6 concludes the thesis work and discusses further work.

CHAPTER 2

SOFTWARE TESTING

2.1. Introduction

This chapter discusses software testing techniques and different test adequacy criteria. The chapter also gives the necessary background on how Genetic Algorithms work.

2.2. Software Testing Techniques

Generally, software-testing techniques are classified into two categories: *static analysis* and *dynamic testing* [18][33][38]. In static analysis, a code reviewer reads the program source code, statement by statement, and visually follows the logical program flow by feeding an input. This type of testing is highly dependent on the reviewer's experience. Static analysis uses the program requirements and design documents for visual review. In contrast, dynamic testing techniques execute the program under test on test input data and observe its output. Usually, the term *testing* refers to just dynamic testing.

The following subsections give a brief background on these two testing categories.

2.2.1. Static Analysis

For years, the majority of the programmers assumed that the programs are written solely for machine execution and are not intended to be read by human, and that the only way to test a program is by executing it on a machine. This manner began to change in the early 1970s, because of the Weinberg's work on "The Psychology of Computer Programming" [46]. Weinberg provided a convinced argument for why programs should be read by people and indicated that this could be an effective error-detection process.

Experience has shown that static analysis, a.k.a. non-computer-based or human testing, methods are quite effective in finding errors [33]. Static analysis methods are meant to be applied during the period that is between the code completion and the beginning of the execution-based testing.

Typical static analysis methods are code inspections, code walkthroughs, desk checking, and code reviews [33]. Code inspections and walkthroughs are the two primary static analysis methods and they have a lot in common. Inspections and walkthroughs involve the reading or visual inspection of a program by a team of people. Both methods involve some preparatory works by the participants. The climax is a meeting of the minds, i.e. brainstorming, in a conference-like gathering held by the participants. The objective of the meeting is to find errors, but not to find solutions to the errors, i.e. to test but not to debug.

2.2.1.1. Code Inspections

Code inspection is a set of procedures and error-detection techniques for group code reading. Most discussions of code inspections focus on the procedures, forms to be filled out, and so on.

During the inspection session, two activities are conducted: code narration and code examination. Code is read statement by statement and analyzed with respect to a checklist of historically common programming errors (e.g. data-reference, data-declaration, computation, comparison, control-flow, input/output, interface).

2.2.1.2. Code Walkthroughs

The initial procedure is identical to that of the inspection process. The difference, however, is in that rather than simply reading the program or using error checklists, one of the participants designated as a tester comes to the meeting with a small set of paper test cases that represent sets of input and expected output for the tested program or module. During the meeting, each test case is mentally executed, i.e. the test data are walked through the logic of the program. The state of the program, i.e. the values of the variables, is monitored on paper or a blackboard.

Definitely, the test cases must be simple in nature and few in number, because people execute programs at a much slower rate than a machine. Thus, the test cases themselves do not play a critical role; rather, they serve as a vehicle for getting started and for questioning the programmer about his or her logic and assumptions. In most

walkthroughs, more errors are found during the process of questioning the programmer than are found directly by the test cases themselves.

2.2.1.3. Desk Checking

Desk checking can be seen as a one-person inspection or walkthrough; a person reads a program, checks it with respect to an error list, and/or walks test data through it.

There are three main reasons that desk checking, for most people, is relatively unproductive: completely undisciplined process, the principle that people are generally ineffective in testing their programs, and no competition like in the teamwork.

2.2.1.4. Code Reviews (Peer Ratings)

Code review is a technique for evaluating anonymous programs in terms of their overall quality, maintainability, extensibility, usability, and clarity. The purpose of the review is to provide programmer assessment. A group of programmers is given some selected programs to rate based on a certain scale written in the review forms.

2.2.2. Dynamic Testing

Dynamic testing techniques execute the program under test on test input data and observe its output. Usually, the term *testing* refers to dynamic testing. There are two types of dynamic testing: black-box and white-box. White-box testing is concerned with the degree to which test cases exercise or cover the logical flow of the program [33]. Black-

box testing, on the other hand, tests the functionalities of software regardless of its internal structure, a.k.a. functional or specification–based testing.

The following subsections give a brief background on these two types of dynamic testing.

2.2.2.1. White Box Testing

White-box testing is more widely applied [33]. It is also called logic-coverage testing or structural testing, because it sees the structure of the program [47]. The objective of white box testing is to exercise of the different logic structures and flows in the program [33].

Adequacy of logic-coverage testing can be judged using different criteria [47]: statement, decision (a.k.a. branch), condition, decision/condition, multiple-condition, and path-coverage; ordered from the weakest to the strongest [15] [22] [24] [33].

Statement coverage criterion requires every statement in the program to be executed at least once. Unfortunately, this is a weak criterion because while it exercises every statement at least once, it does not guarantee exercising the same statement in different flows, if any. For example, in a program segment consists of a statement S1, followed by a selection statement IF (A>1) THEN S2 followed by another statement S3, one is not required to generate input test datum that exercises the FALSE branch in order to satisfy statement coverage criterion. In this case, the test case checks for the correctness of the sequence S1-S2-S3, but not for the correctness of the sequence S1-S3; which is possible to have a problem.

A stronger logic-coverage criterion is known as *decision coverage* or *branch coverage* [13]. This criterion states that one must write enough test cases such that each decision, i.e. IF statement, has a **TRUE** and **FALSE** outcome at least once. In other words, each branch direction must be traversed at least once. Decision coverage can be shown, usually, to satisfy statement coverage. Since every statement is on some sub-path emanating either from a branch statement or from the entry point of the program, every statement would have been executed if every branch is executed. The following example shows why branch coverage criterion is stronger than the statement one, as in an selection statement IF (A>1) THEN X=S2, in order to fulfil branch coverage criterion one must generate, at least, two test input data that satisfy both TRUE and FALSE branches regardless any statements that follow both branches, while in statement coverage criterion the tester needs only to generate input test data that leads to TRUE branch only, since no statement follows the FALSE one.

The problem with branch coverage is that it does not check for all the different sequences. For example, in a two serial selection statements: IF C1 THEN S1 ELSE S2, followed by IF C2 THEN S3 ELSE S4, the branch coverage will just test S1-S3 and S2-S4 sequences OR S1-S4 and S2-S3 sequences. In fact, all those sequences must be checked in order to reveal any potentially infeasible combinations of sequences.

A criterion that has larger coverage than decision coverage is *condition coverage*. In this case, one writes enough test cases such that each condition in a decision takes on all possible outcomes at least once.

Although the condition coverage criterion appears, at first glance, to satisfy the decision coverage criterion, it does not always do so. If the decision **IF (A AND B)** is being tested, the condition coverage criterion would require one to write two test cases – **A** is **TRUE**, **B** is **FALSE**, and **A** is **FALSE**, **B** is **TRUE** – but this would not cause the **THEN** clause of the **IF** statement to execute.

As with decision coverage, condition coverage does not always lead to the execution of each sequence. A criterion that combines these two criteria is decision/condition coverage. It requires sufficient test cases such that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once.

A weakness with decision/condition coverage is that although it may appear to check the effect of all outcomes of all conditions, it frequently does not because certain conditions mask other conditions, e.g. in **IF (A AND B)**, the outcome of statement will be **FALSE** if **A** is **FALSE** without considering **B**'s value at all. Nevertheless, errors in logical expressions are not necessarily made visible by the condition coverage and decision/condition coverage criteria, since these criteria do not test all possible combinations of condition outcomes in each decision. A criterion that covers this problem is multiple condition coverage. This criterion requires one to write sufficient test cases such that all possible combinations of condition outcomes in each decision, and all points of entry, are invoked at least once.

The utmost coverage is achieved by path coverage, since it covers all the previous-mentioned testing coverage criteria [24][47]. Path coverage criterion is concerned with the execution of (selected) paths (i.e., sequences) in the program. Since in a program with loops the execution of every path is usually infeasible, complete path testing is not considered in such cases as a feasible testing goal.

2.2.2.2. Black Box Testing

Black-box testing, a.k.a. functional or specification-based testing, tests the functionalities of software against its specification, regardless of its structure. There are four types of black-box testing: equivalence partitioning, boundary-value analysis, cause-effect graphing, and error guessing [33].

Equivalence partitioning partitions the input domain of a program into a finite number of equivalence classes such that one can reasonably assume (but, of course, not be sure absolutely) that a test of a representative value of each class is equivalent to a test of any other value within the corresponding class. That is, if one test case in an equivalence class detects an error, all other test cases in the equivalence class would be expected to find the same error. Conversely, if a test case did not detect an error, we would expect that no other test cases in the equivalence class would find an error. This is with the exception that a subset of the equivalence class falls within another equivalence class, since equivalence classes may overlap one another. The equivalence partitioning concept maybe applied to white-box testing as well.

Boundary conditions are those situations directly on, above, and beneath the edges of input equivalence classes and output equivalence classes. Experience shows that test cases that explore boundary conditions have a higher payoff than test cases that do not [33].

One weakness of boundary-value analysis and equivalence partitioning is that they do not explore combinations of input data as decision/condition coverage does in white-box testing. A cause-effect graphing came to tackle this problem [33]. It is a formal language into which a natural-language specification is translated, which also points out incompleteness and ambiguities in the specification. The graph is actually a digital-logic circuit (a combinatorial logic network), but rather than using standard electronics notation, a somewhat simpler notation is used. The idea is pretty similar with decision/condition coverage in the white box testing, while boundary-value analysis and equivalence partitioning are similar with condition coverage criterion in white box testing. Thus, using similar analogy in white box testing, cause-effect graphing outperforms boundary-value analysis and equivalence partitioning.

Error guessing is largely an intuitive and ad-hoc process, whose procedure is difficult to formalize. This technique needs an expertise that is able to smell out errors. The basic idea is to enumerate a list of possible errors or error-prone situations and then write test cases based on the list.

2.3. Test Data Generation

In this thesis, we focus on white-box testing as it is more widely applied [43]. The first step in applying a white-box testing is to select a test adequacy criterion, e.g. statement or branch coverage. The next step, then, is to find a set of test data that satisfies the selected adequacy criterion, which is called adequate test data. In testing a program, adequate test data generation is the process of identifying a set of test data, which satisfies given testing criterion [20][34][41]. Generating adequate test data manually is labour intensive and time-consuming process. This problem has motivated researchers to create test data generators that can examine a program's structure and generate adequate test data automatically [14]. How to generate test data automatically? How to evaluate them? These are the major questions that researches in the area of automated software testing are trying to find answers for [20][33][41][42][45].

It is not a trivial task to judge whether a finite set of input test data is adequate or not. The goal is to uncover as many faults as possible with a potent set of a constrained number of tests. Obviously, a test series that has the potential to uncover many faults is better than one that can only uncover few.

A number of automatic test data generation techniques have been developed [45]. Pargas [35] classifies these techniques into random test data generator, structural or path-oriented test data generator, goal-oriented test data generator, and intelligent test data generator. The first three types are also common with the classifications of test data generators done by Edvardsson [10] and Korel [20]. Test data generator is a system (program) that

generates the input data for a target program such that these input data satisfy a particular testing objective (i.e., adequacy criterion).

Random test data generators select random inputs for the test data from some distribution [8][20]. Structural test data generators typically use the program's control flow graph, select a particular path, and use a technique such as symbolic evaluation to generate test data for that path [29][37][36][20]. Goal-oriented test data generators select inputs to execute the selected goal, such as statement, irrespective of the path taken [20]. Intelligent test data generators often rely on sophisticated analysis of the code, to guide the search for new test data [26][35][37][36].

In general, the process of automatic structural test data generation, for path coverage, consists of three major steps: (1) construction of control logic graph (*see* Appendix B), e.g. control flow graph (CFG) or control dependence graph (CDG); (2) path selection; and (3) test data generation that involves dynamic execution of the target program.

The target program must be instrumented in order to monitor the assessment of testing objective when the program is executed with given input data. In most test data generator, the instrumentation is considered to be pre-process stage before the generator can actually be used [10]. This instrumentation process is the process of inserting probes (tags) at the beginning of every block of code of interest, i.e. at the beginning/ending of each function and after the true and false outcomes of each condition. For example in path coverage, these tags are used to monitor and provide the test data generator with a feedback on the traversed path within the program while it is executed with trial test data.

As discussed in the next section, search techniques play important role in generating proper test data using the feedback the test data generator gets from the target program.

2.4. Automated Software Testing as a Search Problem

Searching for an input datum in a pool (domain/set) of possible input data that conforms to the test adequacy criteria, e.g. forcing traversing a specific path, is a search problem.

In the early age of automation of software testing, most of the test data generators were using gradient descent algorithms [20]. The essence of this type of methods is a kind of hill-climbing, so they are quite inefficient, time-consuming, and could not escape from local optima in the search space of the domain of possible input data.

Accordingly, meta-heuristic search algorithms proposed a potential better alternative for developing test data generators [10][43]. Efficient existing meta-heuristic search algorithms include Simulated Annealing (SA), Taboo Search (TS), Genetic Algorithm (GA), Ant Colony Optimization (ACO), and Particle Swarm Optimization (PSO). Each of these search algorithms has its own advantages and disadvantages over the others. They are strongly problem domain dependent, because they use domain-dependent "knowledge" or heuristics related to domain of the problem under consideration.

Among these algorithms, Wegener et al. have shown the suitability of using evolutionary algorithms (e.g., Genetic Algorithms) in software testing [44].

2.5. Genetic Algorithms Based Test Data Generator

Genetic Algorithms (GAs) were invented by John Holland in the 1960s and were developed by Holland and his students and colleagues at the University of Michigan in the 1960s and the 1970s [31]. In contrast with evolution strategies and evolutionary programming, Holland's original goal was not to design algorithms to solve specific problems, but rather to formally study the phenomenon of adaptation as it occurs in nature and to develop ways in which the mechanisms of natural adaptation might be imported into computer systems [17].

GAs have been very interesting area of study in many disciplines since it was published for the first time. Researches are growing rapidly regarding either the behaviour or the application of GA for a particular purpose since then. Some applications of GA are optimization, automatic programming, machine learning, economics, immune systems, ecology, population genetics, evolution and learning, and social systems [31].

Among the features owned by GA, that other normal optimization and search procedures do not have, direct manipulation of solution representation to a problem, search from a population (not a single point), search via sampling (a blind search), and search using stochastic operators (non-deterministic rules) [9].

Figure 1 shows the steps for basic GA.

Step 0: Define a genetic representation of the problem.
Step 1: Create an initial population $P(0) = x_1, \ldots, x_n$. Set $t = 0$.
Step 2: Compute the average fitness $f'(t)$. Assign each individual the normalized fitness value.
Step 3: Assign each x_i a survival probability $p(x_i, t)$ proportional to its normalized fitness. Using this distribution, select N vectors or parents from $P(i)$. This gives the set of the selected parents.
Step 4: Pair all parents at random using their survival probability forming $N/2$ pairs. Apply crossover with a certain probability to each pair and other genetic operators such as mutation, forming a new population $P(t+1)$.
Step 5: Set $t = t + 1$, return to Step 2.

Figure 1: Basic GA Steps

In order to use GA for solving an optimization problem, we need to know how to *represent* the problem as well as its solution in a *chromosome expression*, i.e. sort of a sequence of binary digits that resembles the chromosome-like sequence, which GA can understand and manipulate [12][40]. GA works on this encoded problem and delivers the interpreted result as the problem solution; hence, the user should provide the semantic of the encoded problem. The most widely used representation is binary string. However, recently, in a more advanced GA, representation can be extended into higher numbering system, up to more complicated data structure [24][31]. Investigation for more advanced representations is still going, e.g. character, integer, float, grouped, messy, record, etc [31].

A fitness value of an individual is the measure of its strength to survive in the next generation [16][17][40]. It reflects the chance an individual has to be present directly in the next generation or to be selected for mating with other individuals in the current

generation to produce children for next generation. Fitness value is calculated based on the syntax and semantic of individual representation, mostly it is a normalized value of its objective value such that it can be minimized or maximized accordingly.

A complete iteration (one run) from Step 2 to Step 4 is called a generation. The stopping criteria comprise a desired number of generations, and a measure of convergence or saturation.

Actually, there are two approaches for implementing GA as a problem solver [25]. First, classical Genetic Algorithms, which operate on binary strings, which require a modification of the original problem into an appropriate form (suitable for GA). This would include a mapping between potential solutions and binary representation, taking care of decoders or repair algorithms, etc. Second, GA would leave the problem unchanged, modifying an individual representation of a potential solution (using "natural" data structures), and applying appropriate "genetic" operators.

A good operator is the one that could guide the search faster, reduce the search time, and reduce the search space significantly. Many advanced GA's operators have been explored, while some researchers are trying to create parameter-less GA, where user does not need to select/adjust the operators [43].

Two major operators are used in almost every implementation of GA: Crossover and Mutation operators. Simple crossover operator means single point or uniform crossover, while simple mutation means native mutation as specified in [12]. For example, given two binary-string individuals x_1 = {1 0 1 0 1} and x_2 = {0 1 0 1 0} with single-point crossover

and mutation rates are 0.9 and 0.1, respectively. In the crossover stage, GA generates a random number between 0 and 1, if the number happens to be 0.5, which is actually less than 0.9, then these two individual will be crossed to each other at a specified point, which is randomly selected in between them, assume the selected point is 3. Hence, the new individuals are x_1 = {1 0 1 1 0} and x_2 = {0 1 0 0 1}, while in uniform crossover, both bit-sequences are shuffled between these two individuals, i.e. x_1 = {1 1 1 1 1} and x_2 = {0 0 0 0 0}. In case of mutation after crossing-over, GA randomly generates a random number between 0 and 1 for each individual, if it happens to be less than 0.1 then any bits, which its position is again selected randomly, within an individual will be flipped, i.e. from 0 to 1 or vice versa. Based on experiences, typically, mutation rate is set between 0 and 0.1, while crossover rate is between 0.6 and 1 [9][12][19][22][23][25]. Actually, these two operator rates control the population in terms of exploration and exploitation of the search space. In order to choose the most suitable rate, trial and error approach is still the most widely used method among researchers.

During the selection stage, GA will most probably select individuals that have performances above the current population average to appear in the upcoming intermediate generation, since the upcoming final generation would result form the crossover and mutation stages. Due to this fact, whoever has less strength will vanish as GA evolves from generation to generation. The upcoming (final) population definitely may contain copies of previous individuals (i.e., parents), as well as some new individuals that are totally different from their ancestors. The variation level of new individuals introduced into this new population depends on the crossover and mutation rates. Higher

crossover rate will completely mix the characteristics of both parents into its offspring, while higher mutation rate will certainly produce offspring that has different traits with its parents, i.e. the offspring introduces new traits that do not exist in its parents at all.

Why GA works? This is a very interesting question to anyone who either knows or does not know GA before. GA works based on the number of schemata (sometimes called higher–order structures, hyper planes, or similarity templates) being processed from generation to generation. A short, low-order, and above-average schema is called a building block, since it is going to be reproduced more and more exponentially in subsequent generations [12].

Interested readers in theoretical background and/or application of GA are encouraged to consult distinguished references [12][9][25][31].

Many research papers showed that GA has a promising future in developing test data generators [36][37][39][40]; moreover, some papers have showed that GA outperforms both simulated annealing and taboo search [43].

CHAPTER 3

CRITICAL SURVEY OF GENETIC ALGORITHM BASED TEST DATA GENERATORS

3.1. Introduction

In this chapter, we present a set of attributes we propose for comparing the strengths and drawbacks of the different GA-based test data generators. Then, we discuss related work in light of this set of attributes. We conclude the chapter with a summary of the outcome of our critical survey.

3.2. GA Based Test Data Generator Attributes

Judging a test data generator should not only be based on its effectiveness and efficiency, but also on the underlying characteristics that affect its effectiveness and efficiency. Hence, throughout surveying existing related work on GA-based test data generators, we identified some attributes that can be used for comparing and assessing the strengths and weaknesses of GA-based test data generators. We expect this set of attributes to help in enhancing existing test data generators as well as guiding researchers trying to develop new test data generators using GA. Our proposed attributes are discussed in the sequel.

1) Testing Objective

In all related work, the objectives of testing come before anything else described; in order to be able to evaluate testing achievement.

The testing objective highly affects the way the testing process is conducted and the kind of measures to be used to assess the process [33]. For example, in structural software testing, if we choose statement coverage then we need to know how to monitor whether the program execution with a particular test data has reached certain statement within the program or not. Therefore choosing this testing objective, i.e. statement coverage, has enforced the tester to monitor executed statements within the program.

2) Fitness Function

In most of the meta-heuristic search techniques, especially GA, the testing objective is converted into an objective function, which is furthermore converted into a fitness function that is to be optimized to find a solution for the problem under consideration. The way in which heuristics of the test data generation problem is incorporated into the fitness function contributes significantly to the performance of the test data generator [2][4][32].

Based on our survey, we observed several (sub) attributes of the fitness function: building blocks, normalization, balancing/weighting, adjustment, traversal method, and neighborhood influence. The followings are the descriptions for these (sub) attributes.

1) Building Blocks

A building block is a constituent of the fitness function. The constituents of a fitness function affect its effectiveness/efficiency in directing the search toward the desired goal. For example, in statement coverage, the number of covered statements by a test data can give information about the closeness of this test data to a target test data. The more building blocks the fitness function considers, the more effective/efficient it is in finding the target test data.

2) Normalization

This attribute is meant to tell whether the values of building blocks of the fitness function are normalized across the individuals of the population. Normalization would allow more meaningful comparisons between the different individuals' fitness functions.

3) Balancing/Weighting

Balancing/Weighting is used to differentiate between the contributions of the different building blocks to the overall fitness value. For example, a fitness function might consist of two building blocks: A and B. Heuristics might suggest that A should have double the weight of B.

4) Adjustment

Adjustment applies to any building block of a fitness function and/or to the overall fitness function, according to the selected criterion defined by the test generator designer.

The adjustment operations can be addition, multiplication, or whatever necessary actions (e.g. multiplication of any building blocks with a chosen number) the designer considers required to refine the fitness function.

5) Traversal Method

This attribute is only meaningful for test data generators that adopt the path-coverage criteria for their testing objectives. The traversal method is the way of measuring the "closeness" between the path exercised by an individual, i.e. a generated input data, and a target path. Our survey reveals that two ways for calculating such "closeness" exist: path-wise and branch-wise (a.k.a. statement-wise). In the path-wise traversal method, the fitness function does not consider matched (sub) paths after the first deviation. While, in the branch-wise one, the fitness function considers subsequent matched (sub) paths after any number of deviations. For example, in Figure 2, assume that the darker line (left side) represents the target path while the lighter one (right side) is the path that traversed by an individual trying to satisfy the target path. Using the path-wise traversal method, the common flow (or nodes) between the two paths will be just the first branch. The rest, starting immediately afterward, would be considered as mismatch. On the other hand, using the branch-wise method, the common flow (or nodes) will also include those branches starting with the one that is being pointed to by the lower arrow.

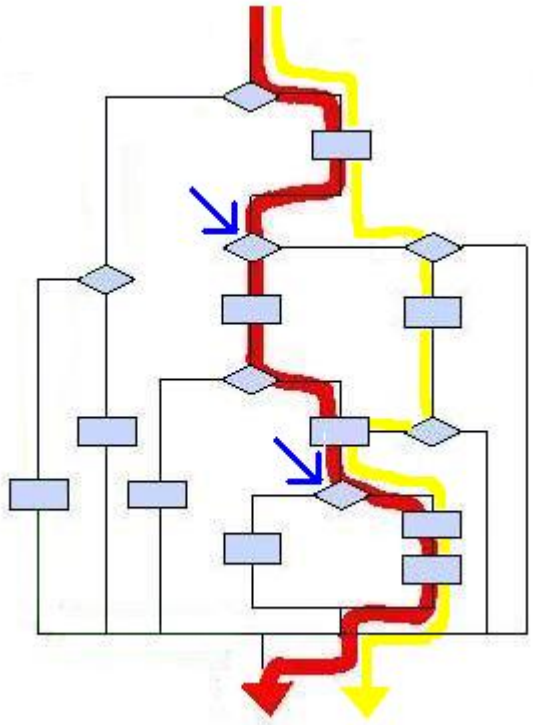

Figure 2: Path-wise vs. Predicate-wise traversal method

The path-wise approach considers all nodes from the first unmatched node found to the rest of the path as violations in comparing the target path with the individual's traversed path. Thus, the number of violations V equals to the number of nodes from the first unmatched node up to the end of the target path. For example, consider a traversed path TR_1 = {1 -5 2 2 3 -2 4 0 5 6} and a target path TG_1 = {1 0 2 1 3 1 4 0 5 1}; where a negative value means TBD, a positive value means TBD, and a zero value means TBD…. If we rewrite the path in term of traversed-target pair then it will be TR_1-TG_1 = {(-5, 0), (2, 1), (-2, 1), (0, 0), (6, 1)}, hence V = 3, since in staring from the third pair (-2, 1) the traversed path is not in the same direction (different signs) anymore with the target one.

(-2, 1), (0, 0), and (6, 1) are considered to be violations in this case (i.e., V=3) with regard to the path-wise approach.

Predicate-wise approach is a more relaxed approach in considering the number of violations *V* within the fitness value, since it allows shuffling as many as unmatched sub-paths within the matched sub-paths. The calculation of the number of violations *V* considers all matched predicates even after encountering unmatched sub-paths. Hence, the distance *D* equals to the summation of all *PVs* of matched nodes. For example, considering the same TR_1-TG_1 = {(-5, 0), (2, 1), (-2, 1), (0, 0), (6, 1)} from the previous path-wise traversing example, *V* = 1, since it only has one unmatched node, i.e. the third pair (-2, 1).

6) Neighborhood Influence

This attribute is meaningful for GA-based test generators that are trying to generate multiple test data at a time. It reflects whether a fitness function considers the pressure of competition among individuals to satisfy targets. There are two types of fitness value of an individual: its own fitness value and its fitness value that is influenced by the targets and/or other individuals competing to cover similar targets.

Furthermore, there are three types of influence that affect the influenced fitness: targets influence, other individuals influence, and both targets and individuals influence at a time.

3) GA Type

This attribute reflects the type of GA being used. Possible types include, but not limited to: simple GAs, modified GAs, hybrid GAs (GA combined with other heuristic search techniques), and parallelized GAs. For more details on these GA types, the reader is advised to refer to [9][12][25][31].

4) GA's Operators

Selecting the GA operators (i.e. crossover and mutation operators) and their probabilities highly affects the GA performance [31]. This attribute examines the operators employed in the different test generators surveyed.

5) Individual Representation

This attribute examines the different types of representation being employed by the test generators surveyed. Possibilities include: binary string, character string, integer, float, grouped, messy, and record [23][24][25][31].

6) Input Parameters Domain

Input parameters domain highly affects the individual's representation in GA. Hence, it influences the GA operators' performance in exploring and exploiting the search space.

7) Stopping Criteria

The stopping criteria indicate how the generator decides to stop its search effort. That could be based on the number of generations, number of satisfied targets, etc.

8) Program Size

This attribute reflects the category of programs, as far as the size is concerned, which have been used to validate the test generator under discussion. In general, rough categorization can be used: small, medium, or large. However, in discussing the different generators surveyed, we mention the actual programs that were used.

9) Program Development Paradigm

This attribute indicates as whether a test generator is meant to test procedural-oriented programs, or object-oriented programs. There is a distinction; for example, in testing object-oriented program, the test data generated might need to be able to create an object.

10) Implementation Issues

This attributes is related to the implementation details of the test generator. It reflects the implementation programming language, platform, its usability, reusability, and portability, etc.

11) Multiple Targets

A test data generator can either try to satisfy one target or multiple targets at a time. As we pointed out in Chapter 1, a test data generator that considers finding multiple targets at once is expected to be more efficient.

12) Handling Infeasible Paths

In path coverage criteria, the very existence of infeasible target paths, logically, is suspected to affect or, more precisely, hinder the search for correct test data, unless there is evidence that proves the opposite [7].

Actually, smarter test data generator can identify and isolate those targets with high possibility of being infeasible targets such that they do not hamper the search. However, a potential infeasible target is to not an infeasible one until the analytical examination confirms [7][22].

13) Validity/Soundness

This attribute is to show whether the test data generator does generate the test data that exercises the program in the way the generator claims it would. For example, considering the path-coverage criteria, the data must traverse the given target path.

14) Efficiency

This attribute reflects the amount of resources (e.g. time, memory) that the test data generator consumes in generating the right test data.

15) Program Type

Mainly, GA-based test data generators have been applied to two types of target program: real time and non-real time. The target program type highly affects the objective of testing. For example, in real time systems, testing usually focuses on *time constraint*s; while in non-real time systems, the focus is usually *path coverage*.

3.3. GA Based Approaches to Test Data Generator

In this section, we present a summary discussion of some existing works based on our set of identified attributes.

Due to lack of information and/or applicability, there are five attributes that we have not been able to discuss with regard to the existing generators: multiple targets, handling infeasible target, validity/soundness, efficiency, and program type. Bueno's generator is the only generator that is able to handle the infeasible targets [7]. Alander's generator is the only one that handles real-time system [1]. None has worked on satisfying multiple targets at once.

The Work by Pei *et al*. [36] in 1994

Pei *et al*. observed that most of the test data generators, which developed in their era, were using symbolic evaluation. Using the symbolic evaluation method, the generator establishes predicate equations under static condition and solves them to derive test data. The symbolic evaluation method is hard to be put into practical use, since the complexity of solving the set of predicate equations is exponential. Recent methods, then, were using actual program execution and minimization search methods, which lack efficacy and efficiency (see Section 2.4). These drawbacks had inspired *Pei et al.* to develop a path-coverage test data generator that employs genetic algorithm.

In their work, Pei *et al*. proposed two different fitness functions, shown in Equation 1, which is simple but less sensitive, and Equation 2, which is complicated but more sensitive, as follow:

$$f = C - \left\{ 10 \times n + 5 \times \frac{n(n-1)}{2} \right\} \qquad \textbf{Equation 1}$$

$$F = F_1 + F_2 + \ldots + F_n \qquad \textbf{Equation 2}$$

In Equation 1, C is a big adjustment number; n is the number of matching branches (or nodes) between traversed sub paths and target sub paths. The third term is a scaling factor that depends on the magnitude of n.

The second fitness function (Equation 2) is the sum of the branch function (F_i) on the path. Suppose all the branch predicates are of the form *E1* **op** *E2*, where *E1* and *E2* are arithmetic expressions, and **op** is one of the logical operations {<, ≤, >, ≥, =, ≠}. Each

branch predicate can be transformed to the equivalent function of the form shown in Table 1.

Table 1: Equivalent branch function

Branch predicate	**Branch function**	**Condition**
$E1 > E2$	$F = E1 - E2$	$E1 - E2 > 0$
$E1 \geq E2$	$F = 0$	$E1 - E2 < 0$
$E1 < E2$	$F = E2 - E1$	$E2 - E1 > 0$
$E1 \leq E2$	$F = 0$	$E2 - E1 < 0$
$E1 = E2$	$F = \text{ABS}(E1 - E2)$	$\text{ABS}(E1 - E2) > 0$
$E1 \neq E2$	$F = 0$	$\text{ABS}(E1 - E2) < 0$

Pei *et al.* construct CFG for the program under test, generate a finite number of selected target paths that are susceptible of error prone, and feed these target paths one by one into their GA-based test data generator manually. The generator runs as many as the number of target paths, since it can only accept one target path at a time.

The test data generation process is divided into three steps: (reduced) CFG construction, target paths generation and selection, and test program execution and data generation.

Pei *et al.* tested their generator using a minimum-maximum program, whose output is the minimum and maximum numbers in an array of integer numbers. They found out that 8 out of 21 selected target paths are infeasible and showed that their approach could find all feasible target paths. Discussion of the approach based on our attributes is given in Table 2.

Table 2: Attributes of Pei's approach

No	Attributes	Approach
1	Testing Objective	Path coverage
2	Fitness Function	Two different fitness functions. First fitness function *f* has the form shown in Equation 1. Building block: Number of matching nodes between target path and traversed path Normalization: No Balancing/Weighting: No Adjustment: Using a certain large number and a scaling factor to adjust the overall fitness value Traversal Method: Branch-wise Neighborhood Influence: Not Applicable (N/A) Second fitness function *F* is the summation of its branch functions along its target path has the form as shown in Equation 2. Building block: Branch predicate value Normalization: No Balancing/Weighting: No Adjustment: No Traversal Method: Branch-wise Neighborhood Influence: N/A
3	GA Type	Simple GA
4	GA's Operators	Simple GA operators, crossover rate is between 0.6 and 0.7, and mutation rate is 0.001
5	Individual Representation	Binary strings
6	Input Parameters Domain	Positive integer numbers
7	Stopping Criteria	Number of generations
8	Program Size	Minimum-maximum program
9	Program Development Paradigm	Has been validated against a functional oriented program
10	Implementation Issues	Implemented using C

The Work by Roper *et al*. [37] in 1995

Roper *et al*. developed a test data generator, using C++, that has an aim to traverse all the branches within a target program, which developed in C. Their generator takes a program and instruments it automatically with probes to provide feedback on the branch coverage achieved.

Roper *et al.* translated the concepts of GA to the problem of test data generation. Firstly, population is considered as a set of test data that can be used in executing a program. An initial population is randomly generated according to the format and type of data used by the program. Then GA takes this initial population and evolves it towards a solution. The evolution stops when it reaches the required branch coverage.

Secondly, each individual in the population is an element in the test data set, and the fitness of an individual corresponds to its coverage. For example, in a program with two sets of branches (say an **IF-THEN-ELSE** statement inside a **WHILE** loop), a group of data item which covers all four branches would have a fitness level of 1.0, whereas one which covers only two would have a fitness level of 0.5 and so on. The population is evaluated by running the program with each individual and assessing their fitness values.

We discuss Roper *et al*. approach based on our proposed attributes in Table 3.

Table 3: Attributes of Roper's approach

No	Attributes	Approach
1	Testing Objective	Branch coverage
2	Fitness Function	Building block: Number of matched branches Normalization: No Balancing/Weighting: N/A Adjustment: No Traversal Method: Branch-wise Neighborhood Influence: N/A
3	GA Type	Simple GA
4	GA's Operators	Simple GA operators, mutation rates are 0.3 at the chromosome level and 0.05 at the gene level, and crossover rate is 0.4
5	Individual Representation	String of characters
6	Input Parameters Domain	ASCII characters
7	Stopping Criteria	The required branch coverage level or number of generations, whichever comes first
8	Program Size	Two small programs: a three-nested-selections program and a four-characters-matching program having four sequential selections
9	Program Development Paradigm	Has been tested against two functional oriented programs
10	Implementation Issues	Implemented using C++ with automatic instrumentation

The Work by Jones *et al*. [19] in 1996

Jones *et al*. developed a GA-based test data generator to achieve branch coverage. The individual representation is a sequence of binary strings. This sequence of binary strings is converted to a decimal number prior to the program execution.

Jones *et al*. use an unrolled CFG to represent one, two, or more iterations for each loop, thus the CFG is acyclic. The unrolled CFG is called control flow tree. A program is instrumented so that as it executes with a test case, it records the branches it reaches and the fitness of that test case. As each branch is executed, the test data generator automatically moves to the next branch in a breadth-first search of the control flow tree. A summary evaluation of the approach is given in Table 4.

Table 4: Attributes of Jones's approach

No	Attributes	Approach
1	Testing Objective	Branch coverage
2	Fitness Function	They propose two fitness functions as follow. First fitness function applies weighted hamming distance of branch predicate value. Building block: Branch predicate expression Normalization: No Balancing/Weighting: More significant bit has more weight. Adjustment: No Traversal Method: Branch-wise Neighborhood Influence: N/A The second fitness function is using reciprocal of branch predicate value. Building block: Branch predicate expression Normalization: No Balancing/Weighting: N/A Adjustment: Overall fitness value is reciprocal of its branch predicate value. Traversal Method: Branch-wise Neighborhood Influence: N/A In case of conditional loops, the number of iterations is considered. The reciprocal approach is more efficient on occasions when the two operands that are connected by an operator are numeric variables. In the case of compound predicates, the fitness for each predicate may be determined separately, and an overall fitness is calculated by multiplication for conjoined predicates and addition for disjoined predicates.
3	GA Type	Simple GA
4	GA's Operators	The employed GA utilizes three types of crossover (i.e. single, double, and uniform) and simple mutation.
5	Individual Representation	Binary strings
6	Input Parameters Domain	Numbers
7	Stopping Criteria	No information
8	Program Size	Six small programs evaluate the approach, i.e. quadratic equation solver, triangle classifier, remainder

		calculation, linear search, binary search, and generic quicksort.
9	Program Development Paradigm	Has been validated against six functional oriented programs
10	Implementation Issues	They wrote both the test data generator and all the tested programs in Ada83.

The Work by Alander *et al.* [1] in 1997

The approach proposed by Alander *et al*. comes under the heading of automated dynamic stress testing. The idea is to produce test cases in order to find problematic situations like processing time extremes. In addition to stress testing, they also try the possibilities to test real time software using GA by identifying the situation where the software has the slowest reaction time.

Alander *et al*. faced a problem in using GA for stress testing: the selection of fitness function in non-functional requirements testing, e.g. response time is somewhat non-deterministic, since the same inputs do not always result in the same response time; even though the testing is conducted in the real environment where the program is expected to work on.

Alander *et al*. attempted to identify peak load conditions under which the system fails by applying stress testing. The system is subjected to peak loads for key operational parameters: transaction volume, user load, file activity, error rates, or their combinations.

There is no detail description of the experiment in their research paper, e.g. individual representation, calculation of fitness function. Thus, we do not discuss the approach in a tabular form.

The Work by Pargas *et al*. [35] in 1999

The work done by Pargas *et al.* is an improvement to Jones *et al.*'s work [35]. The approach they presented also uses branch information to evaluate the fitness function, except it uses control dependence graph for the fitness evaluation, which they claimed that it can give more precise fitness evaluation than Jones *et al.*'s and Michael *et al.*'s [35] approaches. Too see this, consider the control dependence graph shown in Figure 3.

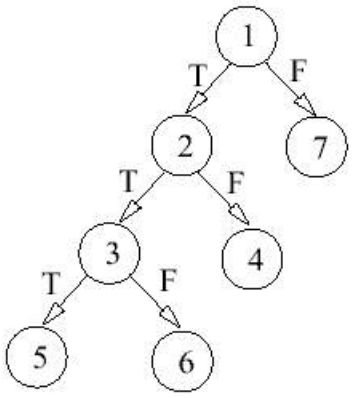

Figure 3: Example of control dependence graph

Suppose that there were two test cases, t_1 and t_2, such that the path through the control-dependence graph for *t1* is 1, 2, 4, and for t_2 is 1, 7. Furthermore, suppose that node 5 is the target of the search. Under the approach presented by Pargas *et al.*, t_1 would be given a higher fitness than t_2 because it has predicate 1T in common with the predicate path of target node 5. But, under the Jones *et al.*'s approach, t_1 and t_2 would be given the same

low fitness because neither test case executes the target or one of its siblings in the control-dependence graph; the fact that t_1 is closer to the target than t_2 is not incorporated into the fitness calculation.

Pargas et al. presented the analysis of their system in term of time and space complexity and compared their approach with random test data generator. They parallelized the work to make it faster. They also, claim that the approach can provide path coverage with minor modifications. Discussion of the approach based on our attributes is presented in Table 5.

Table 5: Attributes of Pargas's approach

No	Attributes	Approach
1	Testing Objective	The approach complies with both statement and branch coverage. Moreover, they argue that the approach can accommodate path and definition-use coverage with minor modifications.
2	Fitness Function	Building block: Number of common branch predicates in the CDG of a program Normalization: No Balancing/Weighting: N/A Adjustment: No Traversal Method: Branch-wise Neighborhood Influence: N/A
3	GA Type	Simple GA
4	GA's Operators	Employed GA in the approach utilizes single point crossover with rate 0.9 & simple mutation with rate 0.1.
5	Individual Representation	String of characters
6	Input Parameters Domain	Numbers
7	Stopping Criteria	Number of generations or coverage level
8	Program Size	It was tested with six small programs tests the developed generator, i.e. bubble sort, bisection method, triangle classifiers, four balls bouncing, array elements classification, and middle value. However, they claim

		in their report that the approach can handle larger programs with multiple procedures, i.e. higher scalability
9	Program Development Paradigm	Functional oriented programs
10	Implementation Issues	The proposed approach makes use of the available tool named Aristotle to generate a program map, i.e. a CDG, and an instrumented version of the program. The approach parallelizes execution of the instrumented program on a single test data among as many available as processors that improves overall execution time almost linearly. In this case, the proposed approach also employs automatic load balancing to prevent processors from being locked in time-consuming loops. The generator (named TGen), which runs on UNIX, stops when it has exceeded given time limit or number of maximum attempts.

The Work by Michael *et al.* [25][26][27][28][29][30] in 1997, 1998, 2001

Michael *et al.* implemented Korel's function minimization [26] approach in their GA-based test data generator. They have built a test data generator called GADGET (Genetic Algorithm Data Generation Tool), which has the capability to instrument a program automatically with no limitation in the programming language, but it has restriction that it can only accept scalar input. GADGET has the condition-decision coverage as its adequacy criteria.

Korel's fitness function is shown in Table 6. It is a summation of the branch functions.

Table 6: Korel's fitness function

Decision Type	Example	Fitness Function
Inequality	If ($c >= d$) …	$f(x) = \begin{cases} d - c & \text{If } d \geq c \\ 0 & \text{Otherwise} \end{cases}$
Equality	If ($c == d$) …	$f(x) = \lvert d - c \rvert$
True/False Value	If (c) …	$f(x) = \begin{cases} 1000 & \text{If } c = \text{FALSE} \\ 0 & \text{Otherwise} \end{cases}$

GADGET uses simple GA as well as differential GA. The difference between Differential GA and the Simple GA is in the recombination process [26]. In the Differential GA, each input parameter I_i in the child I is calculated by

$$I_i = A_i + \alpha(B_i - C_i)$$

Equation 3

Where α is a parameter to adjust the search movement in the space, i.e. conservative/exploitative ($0 < \alpha \leq 1$) or extreme/explorative ($\alpha > 1$). For example, see the following figure with $\alpha = 0.4$

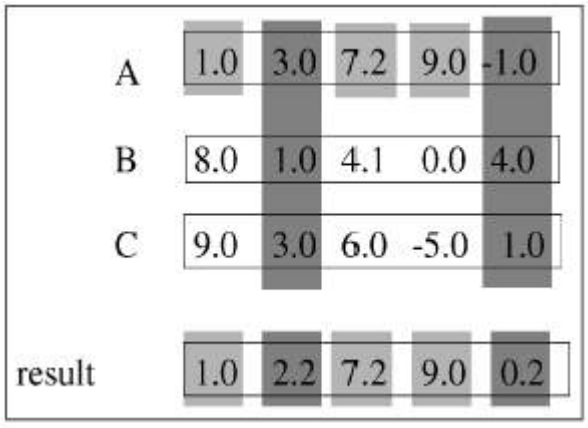

Figure 4: Illustration of recombination process in differential GA

Let *I* be a new offspring. Then $I_2 = 3.0 + 0.4 * (1.0 – 3.0) = 2.2$.

Michael *et al*'s result shows that, in general, the simple GA outperforms the differential one. For the first time in the history of automated test data generation, GADGET is tested with a big program named b737, which is part of an autopilot system (real-world control software). They reported that the performance of random test generation deteriorates for larger programs. An evaluation of their approach is given in Table 7.

Table 7: Attributes of Michael's work

No	Attributes	Approach
1	Testing Objective	Branch coverage
2	Fitness Function	Building block: Branch predicate expression Normalization: No Balancing/Weighting: N/A Adjustment: No Traversal Method: Branch-wise Neighborhood Influence: N/A
3	GA Type	Simple GA and differential GA
4	GA's Operators	Simple GA operators and differential crossover
5	Individual Representation	Binary strings
6	Input Parameters Domain	Numbers
7	Stopping Criteria	Number of generations or branch coverage level
8	Program Size	They have tested the approach with 9 small programs as well as one big program b737, which is a C program that is part of an autopilot system.
9	Program Development Paradigm	Has been validated against small and big programs to show its scalability
10	Implementation Issues	The tool name is GADGET and developed in C

The Work by Bueno and Jino [7] in 2000

Bueno *et al*. proposed an approach that utilizes control and data flow dynamic information. The proposed approach is meant to fulfill path coverage testing. In addition, it also tackles the identification of potentially infeasible program paths by monitoring the progress of the search for required test data.

Bueno *et al.* proposed the fitness function, *Ft*, to evaluate each individual.

$$Ft = NC - \left(\frac{EP}{MEP} \right)$$ **Equation 4**

NC is the value of the path similarity computed considering the number of coincident branches between the executed path and the target one, from the entry node up to the node where the executed path is different from the intended one. This value can vary from 1 to the number of branches in the target path.

EP is the absolute value of the predicate function (see Table 8) associated to the branch where there is a deviation from the target path. The value reflects the error that causes the executed path to deviate from the intended one.

MEP is the predicate function maximum value among the candidate solutions that executed the same branch of the intended path.

The fitness value echoes the fact that the larger number of correctly executed branches, the closer is the individual to the desired path. From several individuals with the same number of correct branches, the most fitting are those with smaller *EP*. This value

indicates the error that causes the deviation and measures how distant is the candidate solution from executing the correct branch.

Observe that the value of (*EP*/*MEP*) is a measure of the candidate solution error with respect to all the solutions that executed the right path up to the same deviation predicate. The value is used as a solution penalty. Thus, the search dynamics is characterized by the coexistence of two objectives: maximizing the number of correctly executed branches and minimizing the predicate function of the covered predicates.

The predicate function *EP* is attained by dynamic data flow analysis [20][21], which is based on Korel's function. Each simple predicate *E1 op E2* is transformed into the form EP *rel* 0, where *rel* is one of the followings: $<$, $\leq$, $=$, and $\neq$. For example, a predicate $a > c$ is transformed to $c - a < 0$. *EP* is actually a function (directly or indirectly) of program input variables. Thus, changes on these variables have the potential of influencing the function's value. Moreover, it is possible to manipulate input variables to minimize a given value of predicate *EP*.

Table 8 summarizes *EP* calculation: the first column depicts possible predicate types considering the various relational operators, second column contains predicate functions, and *rel* is the appropriate operator for *EP rel* 0. *LG*(*E1*) refers to positive Boolean predicates and *LG*(!*E1*) to negative Boolean predicates. k2 is given as a penalty to violations of Boolean predicates, which they set to 100, while k1 is meant to avoid division by zero when *E1* = *E2*, which they set to 0.3.

Table 8: Predicate functions

Predicate	Predicate Function	rel
$E1 > E2$	$EP = E2 - E1$	$<$
$E1 \geq E2$	$EP = E2 - E1$	$\leq$
$E1 < E2$	$EP = E1 - E2$	$<$
$E1 \leq E2$	$EP = E1 - E2$	$\leq$
$E1 = E2$	$EP = \lvert E1 - E2 \rvert$	$=$
$E1 \neq E2$	$EP = \frac{1}{\lvert E1 - E2 \rvert + k1}$	$\neq \frac{1}{k1}$
$LG(E1)$	$EP = k2$ if $E1 = 0$ $EP = 0$ if $E1 \neq 0$	
$LG(!E1)$	$EP = 0$ if $E1 = 0$ $EP = k2$ if $E1 \neq 0$	

In the case of character comparisons, they calculate *EP* using the ASCII values associated to the characters. While in string comparisons, they sum up all the absolute values of the differences between the ASCII values associated to the characters in each position of the string.

Compound predicates that involve logical AND operator are treated as the summation of its *EP* functions and the lowest value of them for compounded conditions with logical OR operator.

In addition to the fitness function, they also consider the identification of potentially infeasible paths by monitoring the progress of the best fitness found. The approach considers a continual population's best fitness improvement as an indication of feasible path is covered. On the other hand, attempts to generate test data for infeasible paths result, invariably, in a persistent lack of progress because of infeasible predicate.

The experimental results show the approach validity and its benefit. Discussion of the approach based on our attributes is depicted in Table 9.

Table 9: Attributes of Bueno's work

No	Attributes	Approach
1	Testing Objective	Path coverage
2	Fitness Function	Building block: Number of matched branches and branch predicate value Normalization: Of the two building blocks, only the predicate value is normalized to the maximum predicate value among candidate solutions that executed the same branches of the target path Balancing/Weighting: No Adjustment: No Traversal Method: Path-wise Neighborhood Influence: N/A
3	GA Type	The approach utilizes proportional selection scheme and elitism inside the GA
4	GA's Operators	Simple GA operators
5	Individual Representation	Binary strings
6	Input Parameters Domain	Numbers
7	Stopping Criteria	Number of generations and path coverage level
8	Program Size	Six small programs exercise the proposed test data generator: floatcomp, quotient, strcomp, find, tritype, and expint
9	Program Development Paradigm	Functional oriented programs
10	Implementation Issues	They run each test program 10 times to reduce random variations. Moreover, they apply two execution modes: one with initialized population and the other with initial random population.

The Work by Lin and Yeh [22] in 2000

Lin *et al*.'s work is an extension to the work done by Jones *et al*. In their work, the level of coverage had been increased from branch testing to path testing and the ordinary (weighted) hamming distance has been extended such that it can handle different ordering of the target paths that have the same branch nodes.

They extend the hamming distance from the first order to the n^{th} order (n>1) to measure the distance between two paths, which named Extended Hamming Distance (EHD). The rationale here is that, in path testing, two different paths may contain the same branches but in different sequences, where the simple hamming distance is no longer suitable. They name the fitness function SIMILARITY, since it calculates the similar items with respect to their ordering within the two different paths, e.g. branches, between the current executed path and the target path. The greater SIMILARITY leads to the better fitness. The higher order SIMILARITY is more significant than its lower order counterpart is. The highest-ordered SIMILARITY between two paths is therefore the most significant one.

The test data generator consists of four basic steps: CFG construction, target path selection, test data generation and execution, and test result evaluation. The first generation of test data is generated at random. Then the generated test data are fed to the program for execution. One test data will be exercised in one and only one selected path. The survivors of test cases to the next generation are chosen according to the fitness function. After all test data in the present generation are fed, the new generation of test

data is generated by the operators of reproduction, crossover, and mutation. The system will automatically generate the next generation of test data until one of the test data covers the target path or it has exceeded the specified maximum number of generations. The objective of each run of the test data generator is to satisfy only a single target path, so the generator must run at least as many as number of target paths.

Lin *et al*. have used triangle classifier as their tested program. They reported that the quality of generated test data is higher than the ones that produced by random testing, because the test data generator can direct the generation of test data to the desirable range fast. Table 10 presents a summary of our evaluation based on our attributes.

Table 10: Attributes of Lin's work

No	Attributes	Approach
1	Testing Objective	Path coverage (extension of Jones's work [19])
2	Fitness Function	Building block: Branch predicate expression Normalization: Normalized EHD Balancing/Weighting: Proposed approach is assigning different weights for each level of the hamming distances that compose normalized EHD. Adjustment: No Traversal Method: Path-wise Neighborhood Influence: N/A
3	GA Type	Simple GA
4	GA's Operators	The simple GA used in the approach puts into practice two-point crossover with probability 0.9 and a simple mutation with rate set to the reciprocal of the length of the individual bit string
5	Individual Representation	48-bit length string
6	Input Parameters Domain	Integer number
7	Stopping Criteria	Number of generations and path coverage level
8	Program Size	A simple triangle classifier program
9	Program Development Paradigm	Has been validated against small functional oriented
10	Implementation Issues	Implemented using C

The Work by Wegener *et al.* [44] in 2002

Wegener *et al*. developed fully automatic GA-based test data generator for structural testing, specifically statement and branch coverage, of real-world embedded software systems.

The proposed fitness function consists of two major building blocks: approximation level, and normalized predicate local distance. Overall fitness value is the summation of the approximation level value and the local distance value.

The approximation level indicates the number of continuously matched branching nodes between the traversed path by an individual and a target path (or they call it as partial aim).

The local distance is calculated for the individual by means of the branching conditions in the branching node in which the target node is missed. The local distance of a branching node that contains multiple conditions is a combination of the local distances of each condition. The report does not describe the normalization of local distance.

For a node of the type **A OR B**, the local distance is the minimum value between single predicate **A** and **B**. In the case of **A AND B**, the local distance is the sum of each single predicate **A** and **B**.

An individual with a fitness value 0 means that it satisfies the partial aim. Although their tool works on only one partial aim after the other, it takes into consideration the execution of a test datum that usually leads to passing several partial aims. Thus, the test soon focuses on those partial aims that are difficult to reach. The stopping criteria used are full statement/branch coverage and number of generations, depends on which one is satisfied first.

They have developed a tool environment to automate test case design for different structural testing methods, which consists of the following six components.

1. Parser to analyze the program.
2. Graphical User Interface to enter the specification of the input domain of the program.
3. Instrumenter to capture the program structures executed by the generated test data.
4. Test driver generator to generate a test bed running the program with the generated test data.
5. Test controller that includes the identification and administration of the partial aims for the test and which guarantees an efficient test by defining a processing order and storage of initial values for the partial aims.
6. Toolbox of evolutionary algorithms to generate the test data.

Their report does not discuss as whether multiple targets can be covered at one time. However, the approach, more precisely, the test control, evaluates all individuals generated with respect to all unachieved targets. Thus, other targets found by chance are identified, and individuals with good fitness values for one or more targets are noted and stored for seeding the next subsequent testing of uncovered targets.

The approach instruments the test program automatically and assumes the targets are given.

They reported that full coverage of some programs is achieved, but not for all. According to their report, they are investigating whether infeasible statements/branches or the number of generations are some of the reasons for not being able to achieve full coverage in some programs. They compared the results with the test data that are generated using random testing, which apparently result in much lower coverage. We discuss their approach based on our attributes in Table 11.

Table 11: Attributes of Wegener's work

No	Attributes	Approach
1	Testing Objective	Statement and branch coverage
2	Fitness Function	Building block: Approximation level and normalized predicate local distance (see **Error! Reference source not found.**) Normalization: Only the local distance value is mapped into the value between 0 and 1. Balancing/Weighting: No Adjustment: No Traversal Method: Path-wise Neighborhood Influence: N/A
3	GA Type	The approach exploits initialized population and multi-population GA with maximal number of generation 200. Migration between sub-populations is every 20 generations in a complete net structure (5% migration rate). Competition between sub-populations is every 5 generations (division pressure of 3)
4	GA's Operators	The GA used in the approach applies discrete recombination with rate of 1 and multiple strategies, i.e. different mutation range for each sub-population, which leads to different search strategies: from a globally oriented search. This allows more exploration, when employing a large mutation range to a very fine search; and more exploitation, when employing a small mutation range
5	Individual Representation	Integer numbers
6	Input Parameters Domain	Numbers and characters
7	Stopping Criteria	Number of generations and path coverage level
8	Program Size	Five small programs: asin, atof, classiftria, powi, and incbet
9	Program Development Paradigm	Functional oriented programs
10	Implementation Issues	Implemented using Matlab. The proposed approach makes use of available tool *Genetic Algorithm Toolbox for Use with Matlab* (*GEATbx*) that can support real, integer, and binary coding representation of individuals

The Work by Ghazi and Ahmed [11] in 2003

The proposed approach is meant to test component-based software in order to achieve pair-wise coverage, that is to generate minimum number of test data that have maximum coverage of pair-wise configurations. Pair-wise testing, which is of type black box testing, is a specification based testing criterion. It requires that for each pair of components, each pair of instances of these components are covered by at least one test configuration. A test configuration is a combination of instances of different components. Pair-wise testing is also applicable to testing single component software. In this case, a test configuration maybe looked at as a combination of values of the component's input parameters.

The fitness function is calculated as the number of distinct pair-wise configurations covered by an individual divided by the total number of possible pair-wise configurations.

A summary discussion of the approach is presented in Table 12.

Table 12: Attributes of Ghazi's work

No	Attributes	Approach
1	Testing Objective	Pair-wise coverage
2	Fitness Function	Building block: Number of test data configurations Normalization: A fitness value is relative to the number of entire test data configurations required. Balancing/Weighting: N/A Adjustment: N/A Traversal Method: N/A Neighborhood Influence: N/A
3	GA Type	Not clear
4	GA's Operators	Not clear
5	Individual Representation	Not clear
6	Input Parameters Domain	Not clear
7	Stopping Criteria	Not clear
8	Program Size	Since the tester only needs the interface of the test programs hence the size of the test program is not important, i.e. not required in black box testing type. However, the number of components (or input parameters, in case of a single component testing) can give some information about the scalability of the approach. In their case, they tested their generator with four components, each of which has 3 possible values
9	Program Development Paradigm	Not clear
10	Implementation Issues	They reported that the approach have been tested on "Placing A Telephone Call" problem that achieves 90% coverage with 9 configurations per individual and 100% coverage with 11 configurations per individual

3.4. Conclusion on Existing GA Based Test Data Generators

Our critical survey of the state-of-the-art in GA based test data generators, using our set of attributes, discloses some drawbacks related to the existing approaches. The following are some issues we were able to identify:

1. **Testing objective:** Most of the test generators, we surveyed, were developed to satisfy statement and/or branch coverage. Only small number of them was working on path coverage and stress testing. To the best of our knowledge, path coverage criterion has the largest coverage among other type of structural coverage testing criteria [33]. Related works in path coverage testing, e.g. Lin's work [22], have the limitation of handling single target path at a time.
2. **Instrumentation and target generation:** Manual instrumentation and target generation reduce the scalability of the test data generator. Only some existing works instrument the test programs automatically, e.g. works done by Roper *et al.*, Pargas *et al.*, and Michael *et al*.
3. **Program size:** The scalability of the test program is still low for all the works. However, only one of the approaches exercises its test data generator using real world embedded system [30].
4. **Program development:** Although trends in the recent and coming years are moving toward object-oriented development, to the best of our knowledge, none of the approaches works on testing object-oriented programs.
5. **Multiple targets:** There were no attempts to satisfy multiple targets at a time.

6. **Handling infeasible target:** Of all the works, only Bueno's approach [7] is capable of identifying potentially infeasible targets. Unless the tester analyze the target paths intensively, Bueno's heuristic still could not decide that the target is infeasible
7. **Validity/Soundness:** None of the works has shown rigorous analysis (either experimentally or analytically) of the validity or soundness of their approach, to provide confidence that it really does what it claims.
8. **Comparisons:** Most of the related works compare their approaches to random testing as opposed to other approaches. Therefore, one would not be able to tell which generator is better as far as effectiveness and efficiency are concerned.
9. **Program type:** Most of the test programs are non real-time software. Only Alander's approach works on testing real-time system [1].

An appropriate handling of these issues would significantly enhance the performance of GA-based test data generators. We present our attempt at addressing some of these issues in the next chapter. We mainly focus on trying to satisfy multiple target paths at a time.

CHAPTER 4

PROPOSED APPROACH

4.1. Introduction

This chapter describes details of our proposed approach, to test data generation using GA; more precisely, it describes our fitness function. Our approach follows from our effort to deal with one of the problems resulting from our critical survey in Chapter 3; that is the multi-target paths satisfaction at one time.

4.2. The Problem

As has been demonstrated in Chapter 3, many GA-based test data generators adopted statement or branch coverage as their objectives. However, by nature, path coverage criterion covers statement and branch coverage criteria, which makes it the utmost coverage [33]. Thus, an effective software structural testing should have path coverage as its objective.

As Chapter 3 concluded, none of the works on satisfying path coverage consider satisfying multiple target paths at a time, i.e. achieving a set of required target paths in a single run of GA. Clearly, satisfying multiple paths at a time would require incorporating these paths within the fitness calculation. The rationale behind considering multiple paths

at a time is based on the observations that in trying to satisfy a single path, other paths might be satisfied as a by-product. Based on this observation, trying to satisfy multiple paths at a time is expected to greatly increase the efficacy and efficiency of the test data generator, i.e. attaining more coverage with less resources than a single-path test generator would need to cover the same number of paths (over multiple runs).

An essential characteristic of GA based structural testing is that the fitness function is constructed based on the test program. A well-constructed fitness function may significantly boost the possibility of finding a solution and reaching higher coverage [2]. Based on our previous critical survey of existing approaches, we have identified several crucial attributes of a fitness function that may be used in guiding the design of a good fitness function in term of search effectiveness and efficiency: building blocks, normalization, balancing/weighting, adjustment or rewarding, path traversal method, and neighborhood influence.

Guided by our fitness function, we have developed a GA based test data generator for multi-path coverage at one run. The outcome of our wok is the following:

- Developing a test data generator that satisfies the path coverage testing criterion.
- Generating test data that satisfies multiple target paths at a time.
- Designing and implementing several fitness function candidates that consider good building blocks design, normalization, weighting, rewarding, path traversal technique, and neighborhood influence.
- Performing performance comparison between our work and others.

4.3. Research Approach

This section presents our approach to develop our fitness function. The section starts with presenting some terminologies, followed by demonstrating the different fitness function candidates we developed.

4.3.1. Terminology

The following terms used in demonstrating our fitness function candidates:

1) A chromosome C_i represents the i^{th} individual within a population C (i.e., a set of chromosomes), for the program or software under test (hereafter SUT). Each individual represents input datum.
2) A predicate value, PV, is the distance value of a predicate (i.e., condition) according to Korel's distance function (*see* Table 13 below) [20]. It is either greater than zero (> 0) meaning a FALSE branch is traversed, or less than or equal to zero (≤ 0) meaning a TRUE branch is taken. In the case of compound predicates, the distance PV is simply the summation of its primitive distances for conjoined predicates and the minimum of its primitive distances for disjoined predicates (as shown in Table 13, equations no. 7 and 8).

Table 13: Korel's distance function

No	Predicate	Distance if path taken is different
1	A = B	ABS(A – B)
2	A ≠ B	K
3	A < B	(A – B) + k
4	A ≤ B	(A – B)
5	A > B	(B – A) + k
6	A ≥ B	(B – A)
7	X OR Y	MIN(Distance(X), Distance(Y))
8	X AND Y	Distance(X) + Distance(Y)

Where, k is the smallest step for the input data of the program, i.e. the resolution of the number that a programming language can represent or manipulate, in spite of the machine representation. For example, in most programming languages the "integer type" has k = 1.

3) A path P_i is the path traversed by C_i. Each path, *P,* is represented by the sequence of branching nodes/predicates (hence called nodes), i.e. a statement where the program is heading to different branches logically, traversed by *P*, along with their associated *PV* pairs. For example, see Figure 5, a path P_1 that would go through the nodes B1, B2, and B3, with *PVs* = -5, 1, and 0, respectively is represented as P_1 = {1 -5 2 1 3 0}, which means {1 T 2 F 3 T} or {1 TRUE 2 FALSE 3 TRUE}. For simplicity, target paths are treated little differently, since there is concept of *distance*. The *PV* of a condition, with respect to a target path, is represented either by zero (1) or by one (0), to represent FALSE and TRUE, respectively. For example, we represent a target path T_1 = {1 T 2 F 3 T} as T_1 = {1 0 2 1 3 0}.

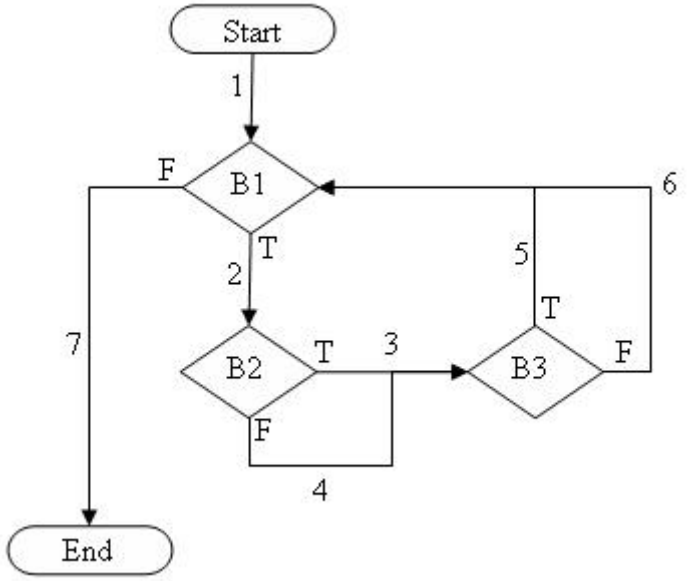

Figure 5: CFG for a minimaxi SUT

4) The length of a path, $lP_i = \frac{|P_i|}{2}$, represents the number of nodes traversed. The division by two is taking place due to the fact that each node is represented by a pair of values: the node number, and the corresponding *PV*.

5) For a path P_i, the x^{th} pair of node index and its *PV* can be accessed through $P_{i.((2*x)-1)}$ and $P_{i.(2*x)}$ respectively; where $1 \leq x \leq lP_i$.

6) A target path TG_i is the i^{th} path in the set of target paths *TG*. For each target path, the tester should suggest (find) an appropriate test datum that is expected exercise that target path[3].

7) A traversed path TR_i is the path traversed by executing chromosome C_i. The *TR* set is the set of traversed paths by the population.

[3] Appendix A gives an example of target sets for some test programs.

8) A sub-path SP_i is a sub-sequence of a path P_i.

9) A matched sub-path between a TR_i and a TG_j is a common sub-sequence between the two paths.

10) An unmatched sub-path between a TR_i and a TG_j is an uncommon sub-sequence between the two paths.

11) Matched node between two paths is a node whose number and value in both paths are the same, i.e. $P_{i.((2*x)-1)} = P_{j.((2*x)-1)}$; where $\{(1 \le x \le lP_i)$ AND $(1 \le x \le lP_j)\}$[4].

12) Unmatched node-branch is a matched node where its branch value has different signs in the corresponding paths (*see* Equation 5).

$$pD_{ij.k} = \begin{cases} abs(TR_{j.2k}); \text{if} \left[\begin{array}{l} (Sign_{ij.2k} < 0) \text{OR} \\ \{(Sign_{ij.2k} = 0) \text{AND} (TG_{i.2k} \oplus TRPV_{j.2k})\} \end{array} \right] \\ 0 \qquad\qquad ; \text{otherwise} \end{cases}$$

Equation 5

Equation 5 presents the distance D calculation in the node level. $TG_{i.2k}$ and $TRPV_{j.2k}$ are the distances, in position k, of target path i and traversed path j, respectively. Function $Sign_{ij.2k}$ is meant to check whether the $TG_{i.2k}$ and $TR_{j.2k}$ predicate values have the same sign; it is formally described below.

$$Sign_{ij.2k} = TG_{i.2k} * TR_{j.2k}$$

Equation 6

[4] For the sake of calculation simplicity, the node positions in both target and traversed path must match. This imposes a limitation in the sense that matched subpaths that have unmatched node positions can not be captured as a contribution to the fitness value. Future work will address this issue.

$TRPV_{j.2k}$ in Equation 7 is a function to convert $TR_{j.2k}$ predicate value into 0 or 1.

$$TRPV_{j.2k} = \begin{cases} 0 \ ; \text{if } TR_{j.2k} < 0 \\ 1 \ ; \text{otherwise} \end{cases} \quad \textbf{Equation 7}$$

13) An unmatched node is a node where whose number and value in the corresponding paths are not the same.

4.3.2. Fitness Function Design

This section discusses the decisions made with regard to the fitness function design.

4.3.2.1. Building Blocks

The basic building blocks of our proposed fitness function candidates are based on comparing traversed paths to target paths in terms of distance *D* and violation *V*. *D* tells how far the traversed path from the target path in term of predicate values for the "unmatched node-branches". *V* tells how many unmatched nodes exist between the target path and the traversed one.

The objective is to minimize the distance *D* and violation *V*. Equation 8 shows the building blocks of the proposed fitness function.

$$IF_{ij} = D_{ij} + V_{ij} \quad \textbf{Equation 8}$$

where,

i = index of target path

j = index of chromosome's traversed path

IF = Intermediate Fitness that look at the fitness of a chromosome with respect to one target path. It is considered as a building block for the overall chromosome fitness where the fitness with respect to all target paths is considered.

D = Distance (or *PV*). In case of the predicate-wise traversal method, *D* is equal to the sum of all the *PVs* of unmatched node-branches.

V = Violation (number of unmatched nodes). Path violation *V* with respect to predicate-wise traversal technique is defined as the number of unmatched nodes between the actual traversed path and the target path along both paths from the beginning to the end.

4.3.2.2. Distance and Violation Calculation

We calculate the predicate value using Korel's distance function [20] as shown in Table 13. The distance equals to zero if the node-branch of both the target path and the traversed path are matching.

For example, consider Figure 5, assuming: B1 is "index ≤ length", B2 is "max < number(index)", and B3 is "min > number(index)". Given, situation for (current) chromosome number 2 as follow: index = 1, length = 2, max = 10, min = 5, number(1) = 2, and a target path TG_1 = {1 1 2 0 3 1} or TG_1 = {1 T 2 F 3 T}. Which path is taken/traversed by this chromosome number 2, TR_2? Based on Equation 8, the distance between TG_1 and TR_2, i.e. D_{12}, is equal to the summation of all distances, i.e. distance B1 (D_{B1}), distance B2 (D_{B2}), and distance B3 (D_{B3}). According to Korel's distance function (*see* Table 13) and assuming that k is equal to 1; therefore, D_{B1} = index – length, D_{B2} = (max – number(1)) + 1, and D_{B3} = (number(1) – min) + 1. These result in the following:

D_{B1} = 1–2 = -1, D_{B2} = (10–2)+1 = 9, and D_{B3} = (2–5)+1 = -2; where if we convert to distance they will be D_{B1} = 0, D_{B2} = 0, and D_{B3} = 0, since TR_2 took the same path with TG_1 {T F T} . In case we have another target path (TG2) that has path {T F F}, thus, D_{22} = 0 + 0 + ABS(-2), and V_{22} = 1, since B3 had been violated.

In the next subsections, we present the details of our approach for calculating the distance D and violation V in a more formal way. We first start by introducing the absolute measures distance and number of violations; then we present our approach for normalizing such measure.

4.3.2.3. Plain Distance and Violation

Plain (i.e., absolute) distance measure is equal to the summation of all predicate values of unmatched node-branches. On the other hand, plain violations equals to the number of unmatched nodes that is determined by the path traversal approaches in the previous chapter sub section 5), i.e. either path-wise or predicate-wise. Obviously, both the plain distance and violation values are not bounded.

The followings are the plain distance function pD_{ij} (Equation 9) between a target path TG_i and a traversed path TR_j along with its supporting functions under predicate-wise path traversal method.

$$pD_{ij} = \sum_{k=1}^{lC_{ij}} mC_{ij.k} \times pD_{ij.k} \qquad \textbf{Equation 9}$$

Equation 9 describes the calculation of distance *D* in the path level, i.e. the summation of all absolute predicate values between target path TG_i and traversed path TR_j.

$$mC_{ij.k} = \begin{cases} 1; if\left(TG_{i.(2k-1)} = TR_{j.(2k-1)}\right) \\ 0; otherwise \end{cases} \quad \textbf{Equation 10}$$

$mC_{i.j.k}$ has the role to match the node sequence between two paths.

$$lC_{ij} = \begin{cases} lTG_i \text{ ; if } lTG_i \le lTR_j \\ lTR_j \text{ ; otherwise} \end{cases} \quad \textbf{Equation 11}$$

The length counter lC_{ij} in Equation 11 is to count the minimum number of conditions encountered in target path TG_i and traversed path TR_i.

Equation 12 and Equation 13, below, calculate the plain violation pV_{ij} considering all target paths; and pcV_{ij} that both considers all paths as well as is influenced by other chromosomes in the population.

$$pV_{ij} = \begin{cases} \sum_{k=1}^{lC_{ij}} \left(pV_{ij.k} * mD_j\right); & \text{if } lX_{ij} = 0 \\ \left(lX_{ij} + \sum_{k=1}^{lC_{ij}} pV_{ij.k}\right) * mD_j \text{ ; otherwise} \end{cases} \quad \textbf{Equation 12}$$

$$pcV_{ij} = \begin{cases} \sum_{k=1}^{lC_{ij}} \left(pV_{ij.k} * mcD \right); & \text{if } lX_{ij} = 0 \\ \left(lX_{ij} + \sum_{k=1}^{lC_{ij}} pV_{ij.k} \right) * mcD \text{ ; otherwise} \end{cases}$$

Equation 13

Equation 14 describes the calculation of plain violation pV_{ij} in the node level, indicated by position index k, which is needed by Equation 12.

$$pV_{ij.k} = \begin{cases} 1 \text{ ; if } \left[\begin{array}{l} \left(TG_{i.(2k-1)} \neq TR_{j.(2k-1)} \right) \text{OR} \\ \left\{ \begin{array}{l} \left(Sign_{ij.2k} < 0 \right) \text{OR} \\ \left(\left(Sign_{ij.2k} = 0 \right) \text{AND} \left(TG_{i.2k} \oplus TRPV_{j.2k} \right) \right) \end{array} \right\} \end{array} \right] \\ 0 \text{ ; otherwise} \end{cases}$$

Equation 14

Equation 15 computes the number of comparisons required between a target path i and a traversed path j.

$$lX_{ij} = \begin{cases} 0 & \text{; if } lTG_i \leq lTR_j \\ lTG_i - lTR_j & \text{ ; otherwise} \end{cases}$$

Equation 15

While Equation 16 is meant to find the maximum distance between a traversed path j and all existing target paths, Equation 17 calculates the maximum distance between all existing traversed paths and all existing target paths.

$$mDj = \text{MAX Over } i\left(pD_{ij} \right)$$

Equation 16

$$mcD = \text{MAX Over } ij\left(pD_{ij}\right)$$ **Equation 17**

Using the path-wise path traversal approach, plain distance pD_{ij} and violation pV_{ij} are calculated in a similar manner to that of the predicate-wise approach until the first unmatched node-branch is reached. The following are the calculation for pD_{ij} (Equation 18), pV_{ij} (Equation 19) and pcV_{ij} (Equation 20).

$$pD_{ij} = \begin{cases} \sum_{k=1}^{x} pD_{ij.k}; \text{where } 1 \le x \le lC_{ij}; if\left({}_{first}\, match_{ij.x} = 1\right) \\ 0; otherwise \end{cases}$$ **Equation 18**

$$pV_{ij} = \begin{cases} \begin{cases} \sum_{k=1}^{x}\left(pV_{ij.k} * mD_{j}\right) & ; \text{if } lX_{ij} = 0 \\ & \text{where } 1 \le x \le lC_{ij} \quad ; if\left({}_{first}\, match_{ij.x} = 1\right) \\ \left(lX_{ij} + \sum_{k=1}^{x} pV_{ij.k}\right) * mD_{j} & ; \text{otherwise} \end{cases} \\ 0; otherwise \end{cases}$$ **Equation 19**

$$pcV_{ij} = \begin{cases} \begin{cases} \sum_{k=1}^{x}\left(pV_{ij.k} * mcD\right) & ; \text{if } lX_{ij} = 0 \\ & \text{where } 1 \le x \le lC_{ij} \quad ; if\left({}_{first}\, match_{ij.x} = 1\right) \\ \left(lX_{ij} + \sum_{k=1}^{x} pV_{ij.k}\right) * mcD & ; \text{otherwise} \end{cases} \\ 0; otherwise \end{cases}$$ **Equation 20**

$$_{first}match_{ij.x} = \begin{cases} 1; if \begin{cases} \left(TG_{i.(2x-1)} = TR_{j.(2x-1)}\right) AND \\ \left[\left(Sign_{ij.x} < 0\right) OR \left(\left(Sign_{ij.x} = 0\right) AND \left(TG_{i.x} \oplus TRPV_{j.x}\right)\right)\right] \end{cases} \\ 0; otherwise \end{cases}$$ **Equation 21**

4.3.2.4. Normalized Distance and Violation

Distance D is normalized by either mD_j as shown in Equation 22, which considers all target paths, or by mcD as shown in Equation 23, which considers all target paths as well as other chromosomes.

$$nD_{ij} = \frac{pD_{ij}}{mD_j}$$ **Equation 22**

$$ncD_{ij} = \frac{pD_{ij}}{mcD}$$ **Equation 23**

On the other hand, violation pV_{ij} normalized by its length, i.e. TG_i, as shown in Equation 24.

$$nV_{ij} = \frac{pV_{ij}}{\left|TG_i\right|}$$ **Equation 24**

4.3.2.5. Neighborhood Influence

Since we try to satisfy multiple targets at the same time, the fitness of a chromosome should consider all target paths. Accordingly, we allow two distance normalization

schemes: based on the target paths (we refer to it as Op) only; and based on both target paths and other chromosomes (we refer to it as Oc).

4.3.2.6. Weighting

We use weights to allow differentiation between the contribution of the different building blocks of the fitness function, that is the distance *D* and the violation *V* to the overall fitness (*see* Equation 8).

$$IF_{ij} = \left(W_{ij} * D_{ij}\right) + \left((1 - W_{ij}) * V_{ij}\right)$$

Equation 25

where,

W is a weight that reflects how much the distance *D* should contribute to the overall fitness value; it also, indirectly, represents the contribution of the violation *V* to the fitness value. It ranges from zero to one; 0 means no contribution and 1 means full contribution.

Setting the weight to 0.5 would mean that both distance and violation are having the same level of contribution to the fitness value.

We allow two ways for selecting the weights: Static Weighting and Dynamic Weighting. In static weighting, the weight is determined by the user; in the range from zero to one.

Dynamic weighting allows the calculation of the weights during the runtime; where each generation may assign different weights to the distance-violation pair. Dynamic weighting is a weight assignment using a normalized value that changes from generation to

generation unattended. Dynamic weighting has two schemas: distance-based and violation-based weight assignments.

Distance-based weight dW_{ij} is a normalized distance value, which is the chromosome's distance divided by the maximum distance of all chromosomes in one generation. Actually, dW_{ij} is meant to normalize total building block values relative to the actual normalized distance in the current generation.

$$dW_{ij} = \frac{pD_{ij}}{MAXOver_j(pD_{ij})}$$ **Equation 26**

Assigning normalized violation value, i.e. $pVij$, divided by its length, i.e. lTG_i, to the weight is the basic idea of dynamic violation-based weight assignment. The pV_{ij} normalizes fitness value relative to the actual normalized violation in the current generation.

$$vW_{ij} = \frac{pV_{ij}}{lTG_i}$$ **Equation 27**

4.3.2.7. Rewarding

For each target path, the chromosome that has the least distance in trying to satisfy this target path among any other chromosomes in the population is a winner with regard to that target path, i.e. the one that will get a reward; that is its fitness will be positively affected. The rationale, here, is to give such a chromosome a better chance of survival to the next generation since it is the closest to some target paths.

Reward of a target path TG_i is given by deducting certain value, R_i, from the intermediate fitness value IF_{ix} of x^{th} chromosome that has the smallest distance with TG_i. (see Equation 29). Reward is given as a *deduction* from the intermediate fitness function since the objective is to *minimize* the overall fitness value of a chromosome.

$$IF_{ix} = IF_{ix} - R_i \text{ ; where } IF_{ix} = \text{MIN Over } i(IF_{ij}) \quad \textbf{Equation 28}$$

Reward R_i given with respect to the target path TG_i. R_i is normalized to the summation of all *IF*s of traversed paths that are trying to satisfy TG_i.

$$R_i = 1 - \frac{MINOver_j(IF_{ij})}{\sum_{j=1}^{|TR|} IF_{ij}} \quad \textbf{Equation 29}$$

4.3.2.8. Final Fitness Calculation

In order to get the overall fitness value we have to sum up and normalize the whole intermediate fitness values for each chromosome.

The following is the final fitness, F_j, calculation for chromosome j considering all target paths.

$$F_j = \frac{pF_j}{|TG|} \quad \textbf{Equation 30}$$

$$pF_j = \sum_{i=1}^{|TG|} IF_{ij} \quad \textbf{Equation 31}$$

The following is the final fitness, F_j, calculation for chromosome j considering all target paths and all chromosomes.

$$F_j = \frac{cF_j}{|TG|} \qquad \textbf{Equation 32}$$

$$cF_j = \sum_{i=1}^{|TG|} NIF_{ij} \qquad \textbf{Equation 33}$$

The following normalized version is used if all the *IF*s of chromosome j are calculated using normalized distance and violation.

$$NIF_{ij} = \frac{IF_{ij}}{\sum_{k=1}^{|TR|} IF_{ik}} \qquad \textbf{Equation 34}$$

4.4. Proposed Fitness Function Candidates

Based on the above discussion on the different decisions that can be made with regard to the fitness function design, Table 14, below, lists the possible variation points along with their corresponding values. For the sake of easier reference, abbreviations of the possible values are shown under the "Code" column in the table.

Table 14: Possible fitness function combinations

No	Attributes	Values	Code
1	Path traversal approach	Path-wise Predicate-wise	Ph Pr
2	Neighborhood influence	Other paths Other paths & chromosomes	Op Oc
3	Distance & violation normalization	Plain (not normalized) Normalized	P N
4	Weighting scheme	No weight Static Distance-based (dynamic) Violation-based (dynamic)	Wn Ws Wd Wv
5	Rewarding	No reward Reward	Rn Rw
6	Final fitness normalization	Plain (not normalized) Normalized	P N

Therefore, there will be 128 (=2*2*2*4*2*2) possible fitness function candidates for all the combination of these attributes. For example, one candidate might take the following attributes: predicate-wise, relative to other paths, normalized distance-and-violation, distance-based weighting, no rewarding and normalized final fitness. Hence, the selected final fitness function will be the following (nD_{ij} and nV_{ij} are calculated as prescribed in the Equation 22 and Equation 24).

$$F_j = \frac{\sum_{i=1}^{|TG|}\left\{\sum_{i=1}^{lC_{ij}}\left\{\left(dW_{ij} * nD_{ij}\right)+\left(\left(1-dW_{ij}\right)* nV_{ij}\right)\right\}\right\}}{|TG|}$$ **Equation 35**

4.4.1. Fitness Function Candidates Roadmap

The fitness functions roadmap is meant to give a comprehensive picture of the proposed fitness function candidates and their relationships. The roadmap contains tables for distance and violation computation, weighting, rewarding, and final fitness computation.

4.4.1.1. Distance and Violation Computation

The following equations apply to both distance *D* and violation *V* calculation at the path level.

Table 15: Distance and violation computations roadmap

No	Measure	Path Traversal (PT)	Normalization	Neighborhood Influence (NI)	Equation To Use
1	Distance (D)	Ph	P		Equation 18
2			N	Op	Equation 22
3				Oc	Equation 23
4		Pr	P		Equation 9
5			N	Op	Equation 22
6				Oc	Equation 23
7	Violation (V)	Ph	P	Op	Equation 19
8				Oc	Equation 20
9			N		Equation 24
10		Pr	P	Op	Equation 12
11				Oc	Equation 13
12			N		Equation 24

4.4.1.2. Weighting

In case of weighting scheme is applied, we use Equation 25 instead of Equation 8 for calculating intermediate fitness value *IF*.

4.4.1.3. Rewarding

If we apply rewarding then we deduct the reward *R* from an intermediate fitness value *IF* of a selected individual as shown in Equation 28.

4.4.1.4. Final Fitness Computation

The final fitness is calculated using one of the following equations that depend on neighborhood influence and normalization attributes.

Table 16: Final fitness computation roadmap

No	Neighborhood	Normalization	Equation To Use
1	Op	P	Equation 31
2		N	Equation 30
3	Oc	P	Equation 33
4		N	Equation 32

4.4.2. Reduced Possible Fitness Function Candidates

Based on our intuition, we expect that normalized values would be more rational than the plain ones. Accordingly, to reduce the number of possible fitness function candidates to be investigated, we limit the scope to those properties in which that the possible values are normalized. In this case, we exclude the plain possible values in attribute 3 and 6 in Table 14. This way, we end up having 32 (=2*2*4*2) possible fitness function candidates.

CHAPTER 5

EXPERIMENTS AND RESULTS

5.1. Introduction

In this chapter, we present and assess the performance (i.e., strengths and weaknesses) of all proposed fitness functions (i.e., candidates) using several tests. The chapter also discusses the implementation of our GA-based test data generator, including its design, setup, and implementation issues. Finally, we present experimental results and analysis.

5.2. Experiments Design

We have conducted 7 different experiments using Matlab; each experiment considers different software under test (SUT). Each experiment is comprised of sets of runs; one set for each fitness function, where average performance over each set is reported. The 32 candidate fitness functions are distinguished from each other by their attributes settings (i.e., the values set for the variation points that were discussed in the previous chapter) as shown in the following Table 17.

Table 17: Fitness function candidates

No	PT	NI	R	W	Candidate Codes
1	Ph	Op	Rn	Wn	Ph-Op-Rn-Wn
2				Ws	Ph-Op-Rn-Ws
3				Wd	Ph-Op-Rn-Wd
4				Wv	Ph-Op-Rn-Wv
5			Rw	Wn	Ph-Op-Rw-Wn
6				Ws	Ph-Op-Rw-Ws
7				Wd	Ph-Op-Rw-Wd
8				Wv	Ph-Op-Rw-Wv
9		Oc	Rn	Wn	Ph-Oc-Rn-Wn
10				Ws	Ph-Oc-Rn-Ws
11				Wd	Ph-Oc-Rn-Wd
12				Wv	Ph-Oc-Rn-Wv
13			Rw	Wn	Ph-Oc-Rw-Wn
14				Ws	Ph-Oc-Rw-Ws
15				Wd	Ph-Oc-Rw-Wd
16				Wv	Ph-Oc-Rw-Wv
17	Pr	Op	Rn	Wn	Pr-Op-Rn-Wn
18				Ws	Pr-Op-Rn-Ws
19				Wd	Pr-Op-Rn-Wd
20				Wv	Pr-Op-Rn-Wv
21			Rw	Wn	Pr-Op-Rw-Wn
22				Ws	Pr-Op-Rw-Ws
23				Wd	Pr-Op-Rw-Wd
24				Wv	Pr-Op-Rw-Wv
25		Oc	Rn	Wn	Pr-Oc-Rn-Wn
26				Ws	Pr-Oc-Rn-Ws
27				Wd	Pr-Oc-Rn-Wd
28				Wv	Pr-Oc-Rn-Wv
29			Rw	Wn	Pr-Oc-Rw-Wn
30				Ws	Pr-Oc-Rw-Ws
31				Wd	Pr-Oc-Rw-Wd
32				Wv	Pr-Oc-Rw-Wv

The explanation for each column is described in Table 14 (Chapter 4). Each of the 32 candidates is exercised at least ten times (i.e., runs) for each SUT.

As discussed below, we mainly assess the performance via three measures: generation-to-generation (G2G) achievement, the best fitness, and cluster convergence (*phi*).

G2G achievement graph is used to analyze the effectiveness and efficiency of each fitness function candidate, while cluster convergence graph is used to analyze the exploration and exploitation behavior of each fitness function candidate. The best fitness graph is to meant analyze the best candidate solution behavior over generations[5]. Having these graphs will help us in comparing the different candidates. More details about these types of graph can be found in [40].

5.2.1. SUTs Preparation

We have selected six test programs as SUTs for experimentations. Each SUT poses special characteristics which we would like to investigate the performance of our candidate fitness functions against. We briefly discuss these SUTs below. For each SUT, we instrument the original program without changing its semantic. Then, we construct the corresponding CFG and generate a list of selected target paths from it. We developed our test data generator using Matlab. We also developed an instrumented version of the considered SUTs in Matlab. The instrumented source code for each SUT, along with the corresponding CFG and selected target paths are presented in the Appendix A.

[5] It is worth noting here that the best fitness function graph can go below zero due to the application of the rewarding scheme in some candidates.

- Minimum-maximum (*mm*): Given an array of numbers, *mm* is a program to find the minimum and maximum numbers within the array. The program has two sequential selection statements inside a loop in which all the conditions (predicates) are simple/primitive. During our experiment, we allowed the length of the array to be variable, and restricted the content of the array to integer numbers.
- Triangle classifier (*tc*): Given three numbers, *tc*, is a program to classify whether these numbers form a triangle or not. If they are, then the program determines whether the triangle is scalene, isosceles, or equilateral. Triangle classifier has three nested selection statements in which all the decisions are compound predicates.
- Bubble sort (*bs*): Given an array of numbers, *bs*, is a program to sort these numbers in an increasing order. The program has two loops that are nested and one selection that is nested inside the inner loop. The outer loop contains compound predicate.
- Insertion sort (*is*): Given an array of numbers, *is*, is a program to sort these numbers in an increasing order. The program has two loops that are nested and one selection that is nested inside the inner loop. The inner loop contains compound predicates.
- Binary search (*ns*): Given an array of numbers and a key, *ns*, is a program to find a key among these numbers. The program has a single loop that contains a single selection.
- Minimum-maximum and triangle classifier (*mt*): Given three numbers, this combined program outputs both the minimum-maximum and the triangle classification as well.

This program is formed from *mm* and *tc* to allow much more complexity when investigating the performance of our candidate fitness functions.

5.2.2. GA Parameters Setup

In using GA, the values of GA parameters must be set up before hand. The followings are the values for all GA parameters that we use (*see* Table 18). Selection of these values was subject to trial-and-error practice. Initially, GA parameters are set to the values that are mostly used and considered promising in the previous related works. Gradually, based on the feedback from one experiment, parameters are refined in subsequent experiments.

Table 18: GA parameters setup

No	Parameter	Value
1	Population Size	30
2	No Of Generations	100
3	Generation Gap	0.8
4	Selection Method	Roulette Wheel
5	Crossover Method	Single Point
6	Crossover Probability	0.5 or 0.9
7	Mutation Probability	0.1 or 0.3
8	Chromosome Type	SUT-based
9	Chromosome Size	Variable
10	Allele Base	10
11	Allele Range	±1000

These parameters, however, slightly vary from one experiment to another based on the corresponding SUT. In other words, the same treatment was not (could not be) applied across all SUTs. The gain from these variations is that they give an idea on how to setup

the parameters when dealing with other test programs that have similar characteristics with the SUTs we have used for experimentations.

It is worth noting here that the parameters setting is not only dependent on the characteristics of the SUT, but also on the fitness function candidate adopted as well as the number of target paths being considered. For example, a more complicated test program with a larger number of target paths is expected to require a larger population size and a larger number of generations to find effective test data.

5.2.3. Things to Record

The following pieces of information are very important to record per generation in order to assess the experiment outcomes. During our experimentations, we write this information for each fitness function candidate to a text file.

Fitness function candidate index: To identify the fitness function candidate that is being used.

Initial and remaining target paths in each generation: To measure the path coverage achieved.

The best chromosomes along with their fitness values in each generation: To analyze the behavior of the fitness function employed.

The successful chromosomes along with their covered target paths in each generation: These are the generated test data that cover some target paths, which have not been covered yet.

Generation-to-generation coverage: This measure is meant to assess the strength and efficiency of the fitness function, and the difficulty of the target paths, as well. This consists of a pair of generation number and its number of satisfied target paths in a run of a specific fitness function candidate.

Cluster convergence coefficient (a.k.a. phi) in each generation: To measure the speed of convergence of the population generated from generation to generation. The value of this metric is calculated as the best fitness divided by the average fitness of the current generation in case of minimization. On the other hand, it is calculated as the average fitness divided by the best fitness in case of maximization. *Phi* is approaching one as the population converges to a single value.

Experiment duration: To note the duration of each experiment conducted. The more complex the problem in term of the program complexity and the number of selected target paths to satisfy, the longer time duration it takes; assuming the same population size and number of generations. However, we do not record this data, since they are only used to demonstrate the SUT complexity, which is a characteristic of the SUT as opposed to the fitness function candidate; moreover, this data vary from one machine to another and from one environment to another.

5.3. Design and Implementation Issues

During the implementation of and experimentation with our approach, there were several issues to take care of in order to get the expected results. These issues are reported as follow.

5.3.1. Generation and Selection of Target Paths

One issue with regard to satisfying multiple targets at a time is that the target paths may have different lengths. Moreover, in the case of looping, it is desirable to cover, at least, no iterations, one iteration, and two iterations; which in turn causes variable chromosome length.

5.3.2. Instrumentation of SUTs

A tag to monitor the traversed path (in response to executing a particular test datum) and to record the distance, i.e. the predicate value, is inserted right before and after any condition-decision. This tagging process is done manually for each test program. Actually, the tag is like a function call that returns the branch number and its distance when it is invoked.

5.4. Graphs and Measurements

For each SUT, at least the G2G achievement graph is presented and analyzed in this chapter. In case of SUTs that have infeasible paths, two more graphs are added: the best

fitness and *phi*. The best fitness and *phi* graphs would not be meaningful if the number of generations is less than 10. Therefore, in our experiments, we do not plot these graphs for SUTs that contain only feasible paths, since most of those paths were found within the first 5 generations.

5.4.1. GA and Fitness Function Parameters Setup

Based on initial trial-and-error results, we selected the rates to be: either 0.1 or 0.3 for static weight, either 0.5 or 0.9 for crossover, and either 0.1 or 0.3 for mutation. The logic behind the selection of the static weight is that the predicate distance contributes much less than the violation. As for the crossover and mutation rate, we are trying to maintain a balance between the exploration and exploitation of the search space. The following table shows all these parameter values that we have tested.

Table 19: GA's and fitness function's parameters possible values

No	Static Weight	Crossover Rate	Mutation Rate	Condition
1	0.1	0.5	0.1	0.1-0.5-0.1
2			0.3	0.1-0.5-0.3
3		0.9	0.1	0.1-0.9-0.1
4			0.3	0.1-0.9-0.3
5	0.3	0.5	0.1	0.3-0.5-0.1
6			0.3	0.3-0.5-0.3
7		0.9	0.1	0.3-0.9-0.1
8			0.3	0.3-0.9-0.3

The last column (i.e. *Condition*) summarizes the sequence of fitness functions' and GA's rates, i.e. static weight followed by crossover and mutation rates. For example, 0.1-0.5-0.1

means the values for static weight, crossover rate, and mutation rate are 0.1, 0.5, and 0.1, respectively.

As a pre-experiment to find the best combination to use, we applied all the parameter values combinations shown in Table 19 to all 32 fitness function candidates using the minimum-maximum (*mm-i*) as a test program with infeasible paths included in the set of target paths. We used the number of successes, i.e. the number of fitness function candidates that found all the required feasible target paths, to assess the effectiveness of the each. Table 20 presents the result corresponding to each parameter-value combination.

Table 20: Effectiveness of parameter-value combinations

No	Combination	No of successes
1	0.1-0.5-0.1	15
2	0.1-0.5-0.3	19
3	0.1-0.9-0.1	15
4	0.1-0.9-0.3	19
5	0.3-0.5-0.1	16
6	0.3-0.5-0.3	17
7	0.3-0.9-0.1	16
8	0.3-0.9-0.3	17

Two of these combinations have the same number of successes: 0.1-0.5-0.3 and 0.1-0.9-0.3. Accordingly, we just arbitrarily selected one of them, that is 0.1-0.9-0.3. The rationale behind this selection is the higher exploration and exploitation abilities that are due to the higher crossover and mutation rates, respectively.

Table 21 below lists all the conducted experiments, where each experiment is composed of 10 runs per fitness function candidate; except for no 4, 5, and 7; which are composed of 20 runs each to allow more confidence.

Table 21: Experiment treatments

No	SUT	No of Target	No of Infeasible Paths
1	Binary Search (*ns*)	7	0
2	Insertion Sort (*is*)	4	0
3	Triangle (*tr*)	4	0
4	Minimaxi-f (*mm-f*)	13	0
5	Minimaxi-i (*mm-i*)	21	8
6	Bubble Sort (*bs*)	14	11
7	Minmax-Tri (*mt*)	84	20

Experiments 4 and 5 are meant to observe the effect of infeasible paths on the behavior of the different fitness function candidates.

5.4.2. Experiments

In this section, we discuss all the experimental results. We organize the discussion as per experiment, i.e. per SUT.

5.4.3. Binary Search (*ns*)

Almost all (6.96 out of 7 on the average) the feasible target paths were found by all the candidates during this experiment. All target paths were found during the first 2 (or 1.25 on the average) generations. These results were based on the average of 10 runs. Moreover, in some runs, all the feasible target paths were found in the first (i.e., initial)

generation. Clearly, this behavior was due to the exploration achieved by the random population developed in initial generations. Accordingly, we could not show the fitness and *phi* behaviors for fitness functions corresponding to those runs, i.e. those graphs would not be meaningful.

The following figure (*see* Figure 6) shows the effectiveness and efficiency all candidates over 10 runs. Effectiveness is indicated by the number of paths found (PF) for a SUT that has only feasible target paths (PFF), on the average; that is PFF-Avg. Efficiency, on the other hand, is indicated by the last generation (LG) for a SUT that has only all feasible target paths (LGF), on the average, where all paths were found; that is LGF-Avg.

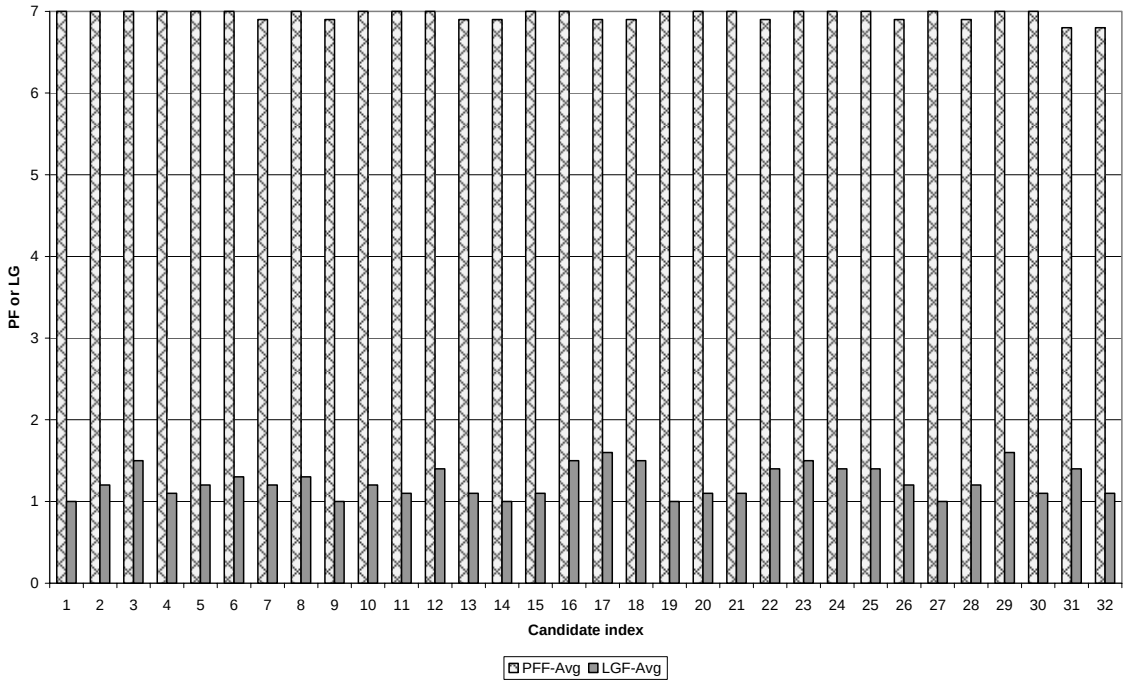

Figure 6: G2G achievement of binary search on the average of 10 runs

As shown in Figure 6, the difference between the candidates performance was insignificant. No candidate can be claimed as an absolute best or as an absolute worst.

5.4.4. Insertion Sort (*is*)

During this experiment, as an average of 10 runs, almost all (3.7 out of 4 on the average) feasible target paths were found within 2.5 (or 1.475 on the average) generations. In some runs, all the feasible target paths were found in the first (initial) generation. The same explanation with binary search is applicable.

Figure 7 shows the effectiveness (indicated by PFF-Avg) and efficiency (indicated by LGF-Avg) of all candidates over 10 runs.

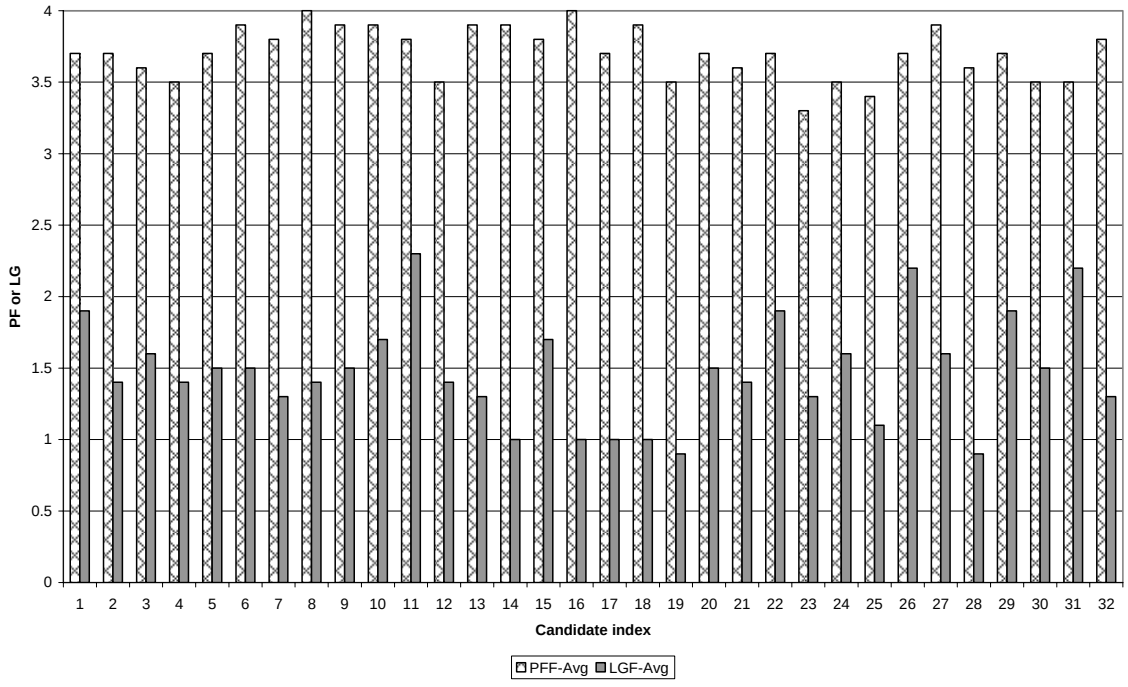

Figure 7: G2G achievement of insertion sort on the average of 10 runs

As shown in Figure 7, the difference between the candidates performance was insignificant. No candidate can be claimed as an absolute best or as an absolute worst.

5.4.5. Triangle (*tr*)

In the triangle classification SUT, all candidates found all (4 out of 4 on the average) feasible target paths were found within not more than 10 (or 7.6 on the average) generations; according to a 10 run-experiment.

Figure 8 summarizes the effectiveness (indicated by PFF-Avg) and efficiency (indicated by LGF-Avg) of all candidates over 10 runs.

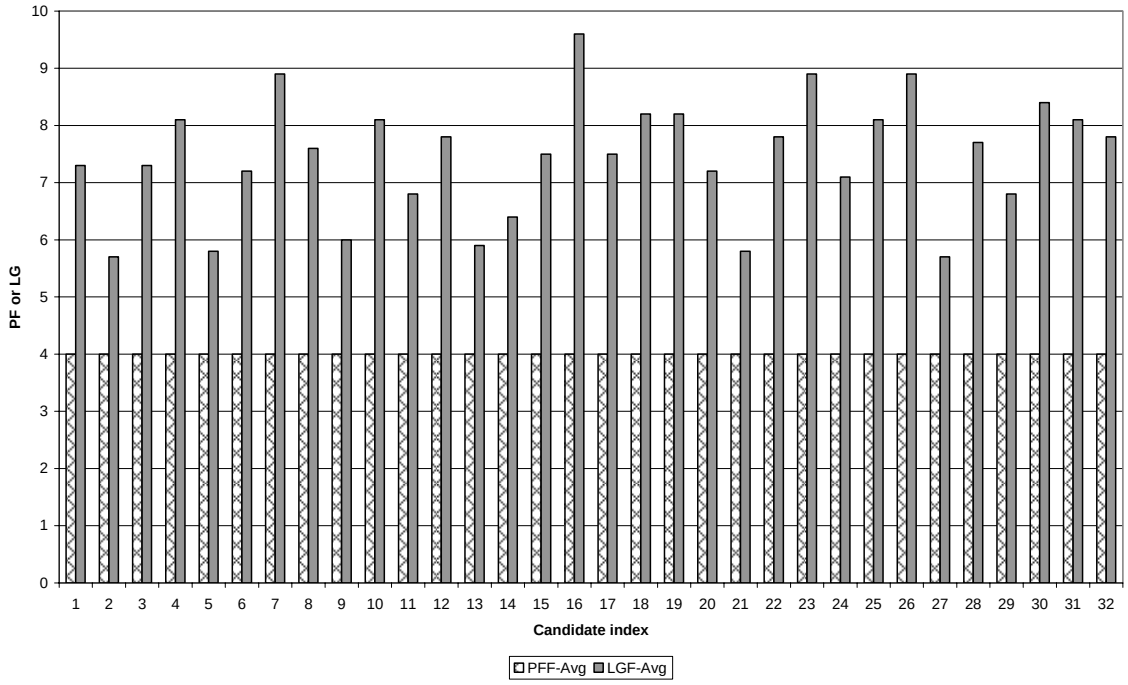

Figure 8: G2G achievement of triangle classifier on the average of 10 runs

As shown in Figure 8, the two most efficient candidates are candidates with indices 2 and 27, which represent Ph-Op-Rn-Ws, and Pr-Oc-Rn-Wd, respectively; that is the candidate that is applying path-wise traversal technique, no reward, and static weight, and the other candidate that is having predicate-wise traversal technique, no reward, and distance-based weight, respectively.

5.4.6. Minimaxi-f (*mm-f*)

With regard to the minimaxi-f SUT, almost all (12.6 out of 13 on the average) feasible target paths were found within not more than 14 (or 9.7 on the average) generations over 20 run-experiment.

Figure 9 summarizes the effectiveness (indicated by PFF-Avg) and efficiency (indicated by LGF-Avg) of all candidates over 20 runs.

As shown in Figure 9, the difference between the candidates performance was insignificant. No candidate can be claimed as an absolute best or as an absolute worst.

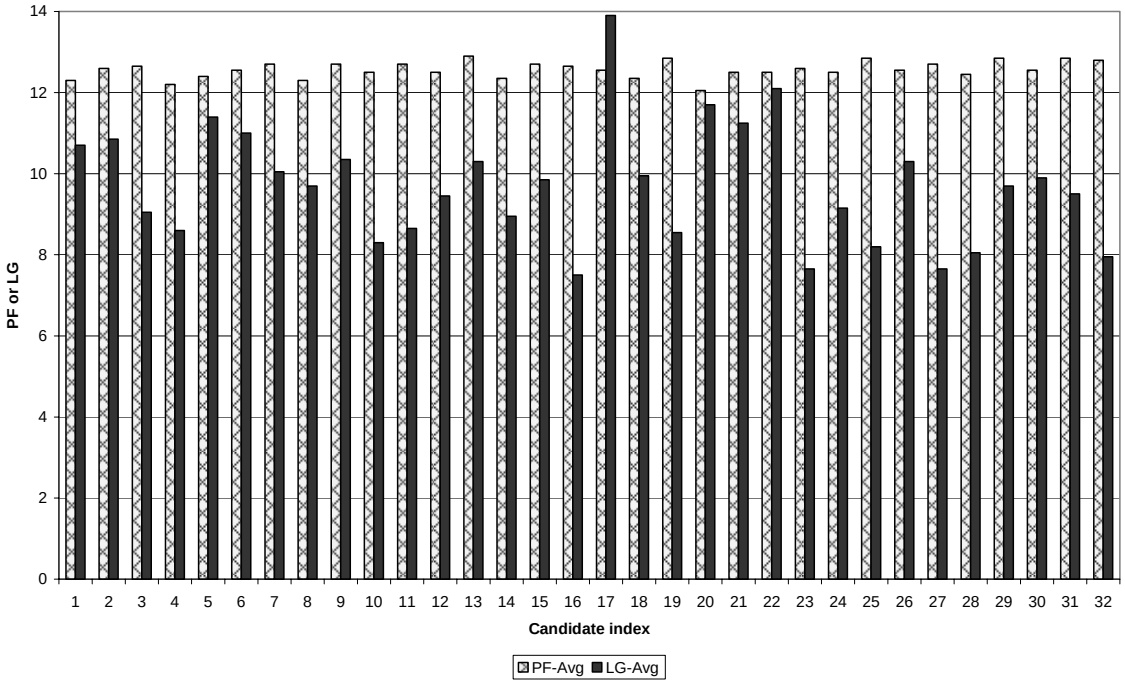

Figure 9: G2G achievement of minimaxi-f on the average over 20 runs

5.4.7. Minimaxi-i (*mm-i*)

In this experiment, all candidates were able to find almost all (12.52 out of 13 on the average) feasible target paths were found within not more than 36 (or 15.56 on the average) generations; in a 20-run experiment. However, if we observe the *phi* graph (*see* Figure 11 of these candidates for the 17th run (arbitrarily chosen), we will be able to see that some candidates are really doing more exploitation (stable line) of the search space while others are doing more exploration (fluctuated line). Moreover, the best fitness graph (*see* Figure 12) shows that, for the 17th run some of the best individuals of some candidates are indeed affected by other individuals in the population (fluctuated line), i.e. not only affected by the target paths. Fluctuated lines in this case show that the fitness of

the “best” individual may drop from one generation to another due to: the competition with other individuals, and/or the removal of the covered-already target paths from the set of targets.

Figure 10, below, summarizes the effectiveness (indicated by PFF-Avg) and efficiency (indicated by LGF-Avg) of all candidates over 20 runs.

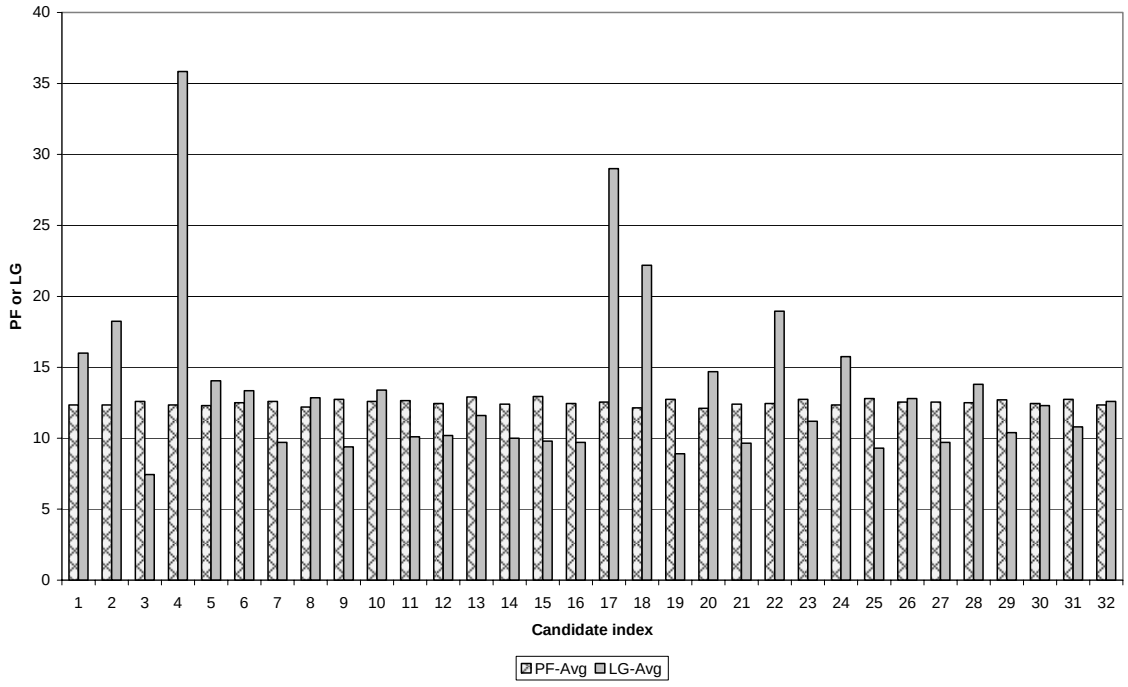

Figure 10: G2G achievement of minimaxi-i on the average over 20 runs

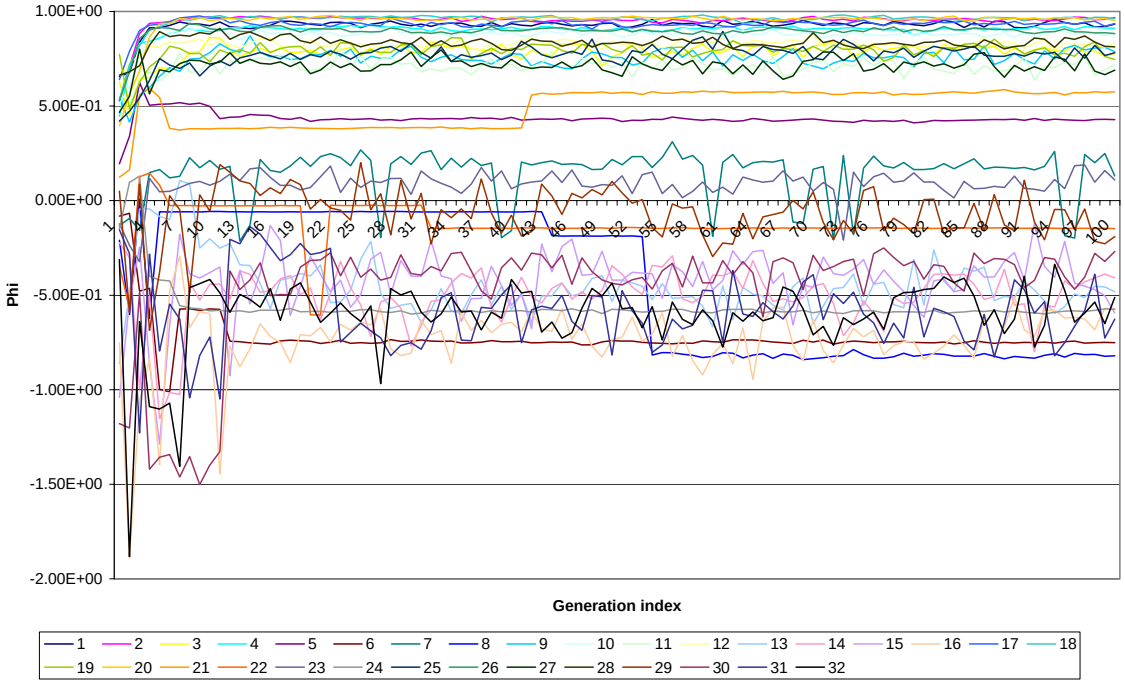

Figure 11: *Phi* graph of *mm-i* for a particular run

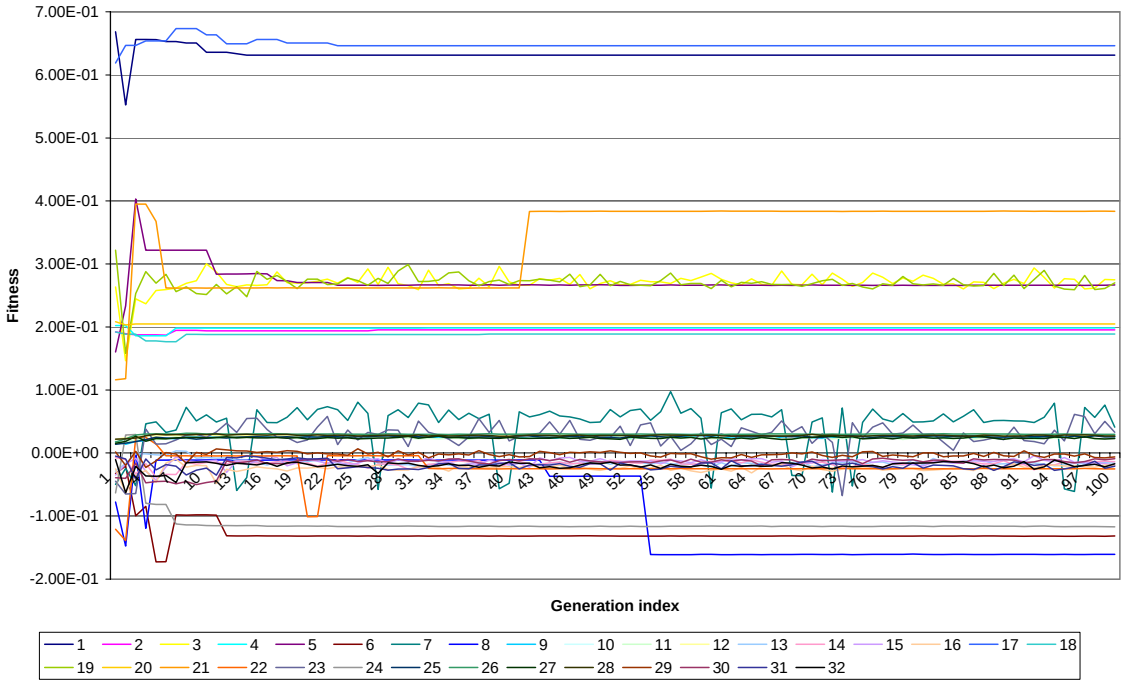

Figure 12: Best fitness graph of *mm-i* for a particular run

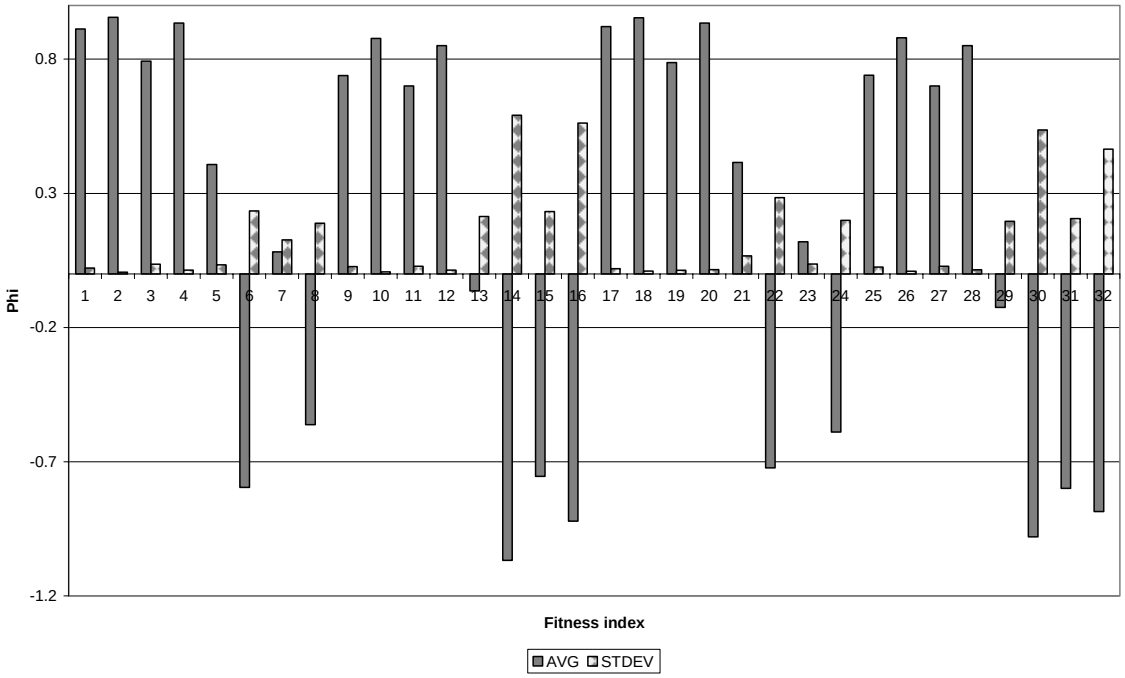

Figure 13: *Phi* average (over 20 runs) graph of *mm-i*

Figure 13 describes the *phi* average and standard deviation over 20 runs for each candidate. On the average (over 20 runs, where each run has 100 generations), candidates that apply rewarding (indicated by negative *phi*; *see* Figure 13) scheme seem to allow more exploration within generations of a run (*see* Figure 14; more fluctuation more exploration).

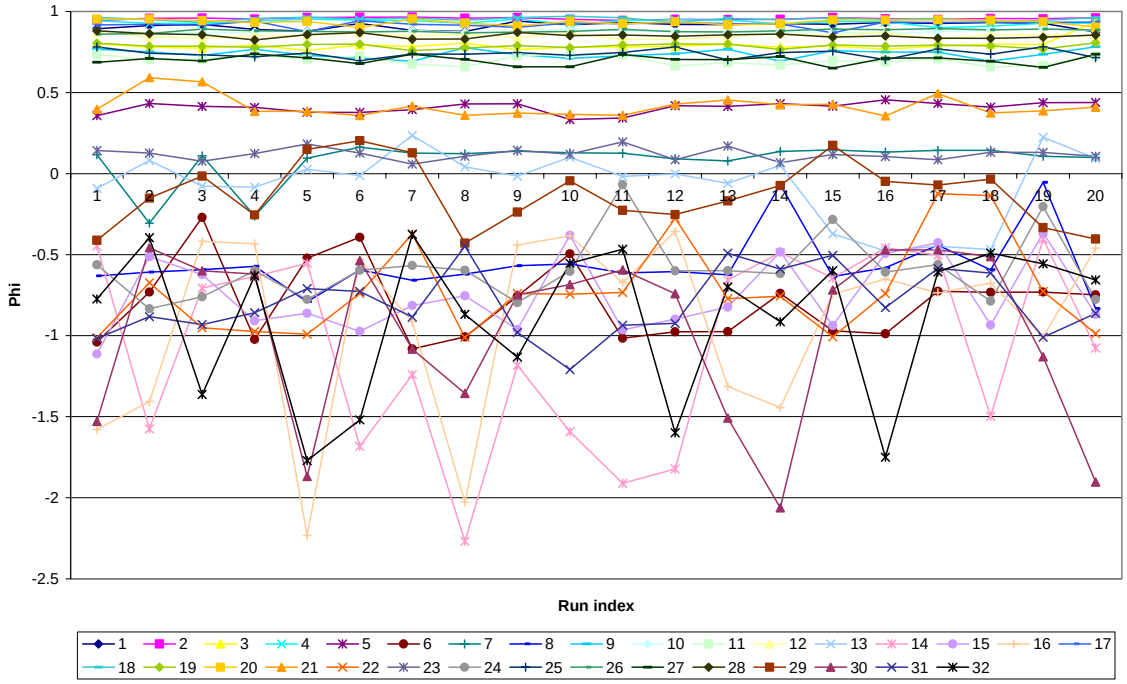

Figure 14: *Phi* average (over 20 runs, each has 100 generations) graph for each candidate of *mm-i*

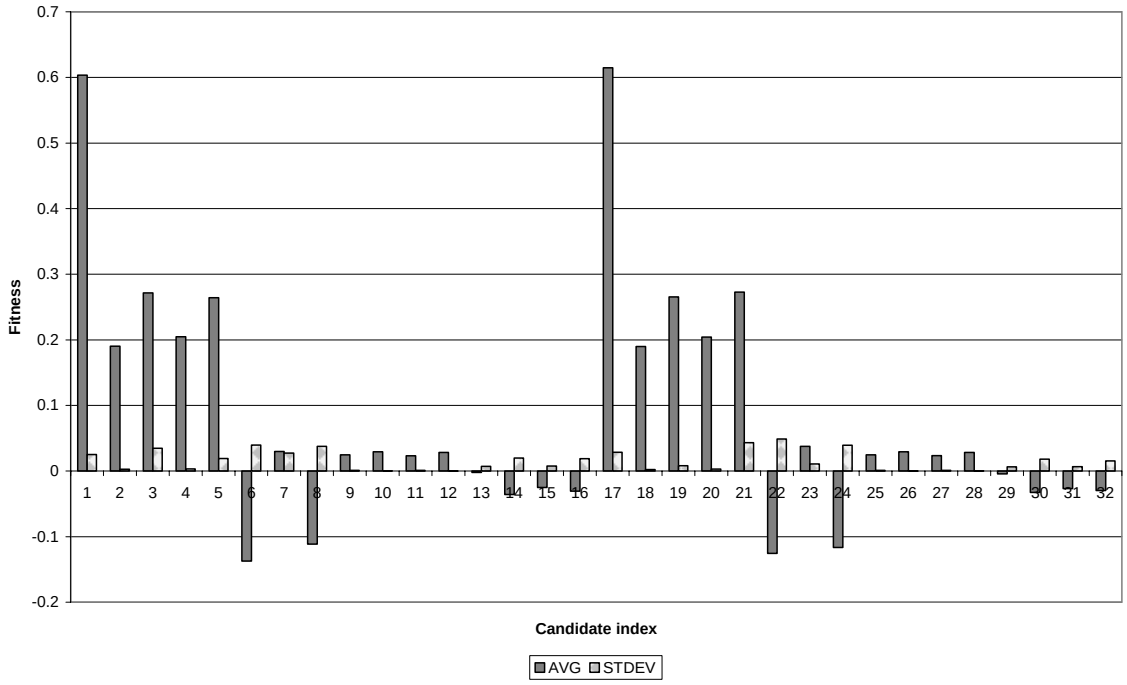

Figure 15: Best fitness average (over 20 runs) graph of *mm-i*

Figure 15 depicts the best fitness average over 20 runs for each candidate. Candidates that employ rewarding scheme (*see* Figure 14; indicated by negative value) explore more (*see* Figure 16; indicated by more number of negative fluctuated lines) search space than the ones without rewarding.

Negative fitness values are due to rewarding scheme only, since covered target paths are excluded immediately from the current targets and the generator recalculates all the fitness values afterward.

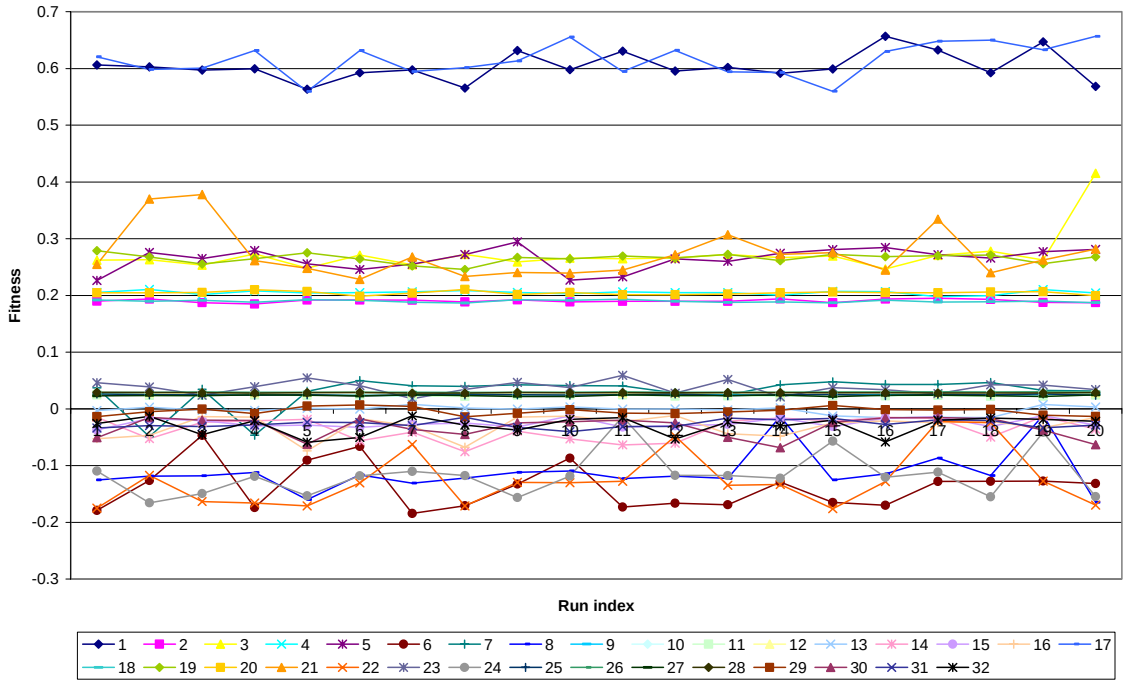

Figure 16: Best fitness average (over 100 generations) graph for each candidate of *mm-i*

5.4.8. Bubble Sort (*bs*)

In bubble sort, almost all (2.98 out of 3 on the average) feasible target paths were found within not more than 2 (or 1.06 on the average) generations by all candidates in 10-

run experiments. In this case, we could not really see the contribution of each candidate towards finding the target paths, since most of the target paths were found by chance in the first two generations. Obviously, this behavior was due to the exploration achieved by the random population developed in initial generations. Accordingly, we could not show the fitness and *phi* behaviors of the fitness functions corresponding to those experiments.

The following figure summarizes the effectiveness (indicated by PFF-Avg) and efficiency (indicated by LGF-Avg) of all candidates over 10 runs.

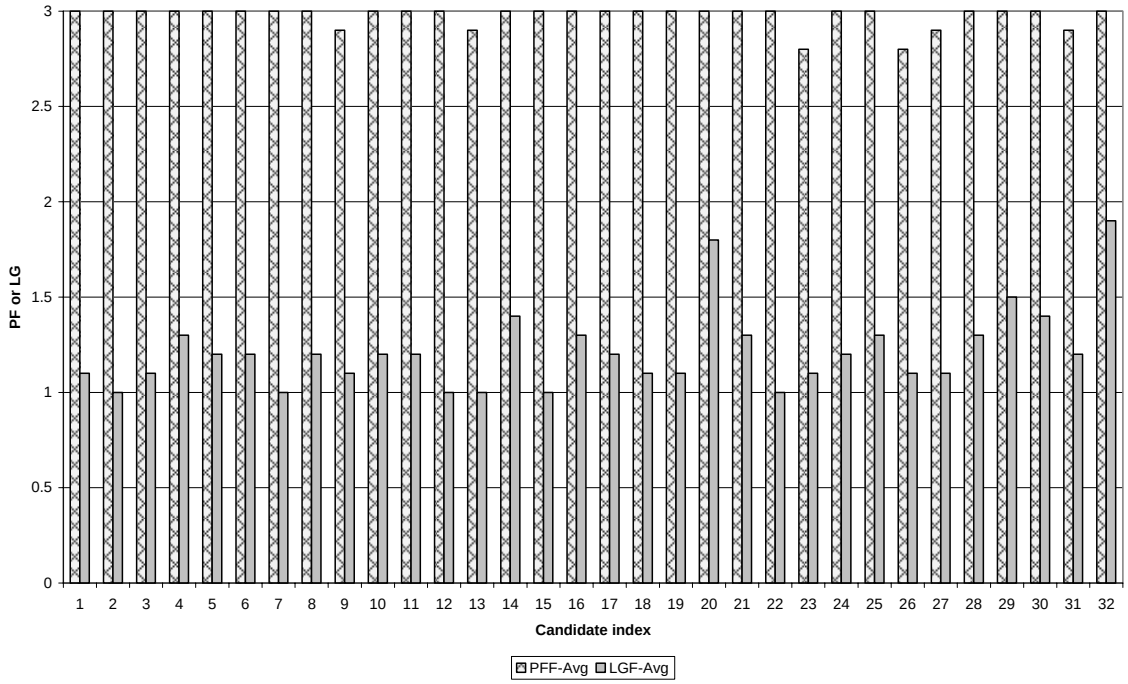

Figure 17: G2G achievement of bubble sort on the average of 10 runs

5.4.9. Minimaxi-Tri (*mm-t*)

Minimaxi-tri is the most challenging SUT among the set we used in our experiments; this is because it is a combination of *mm* and *tr*, and it has a large number of infeasible target paths. In these experiments, almost all (19.3 out of 20 on the average) feasible target paths were found within 30 (or 19.4 on the average) generations in 20-run experiments by most candidates. However, if we observe *phi* graphs (*see* Figure 19) of these candidates for a particular run (9^{th} run; arbitrarily chosen), we will be able to see that some candidates exploit the search space much more than others do.

The following figure summarizes the effectiveness (indicated by PFF-Avg) and efficiency (indicated by LGF-Avg) of all candidates over 20 runs.

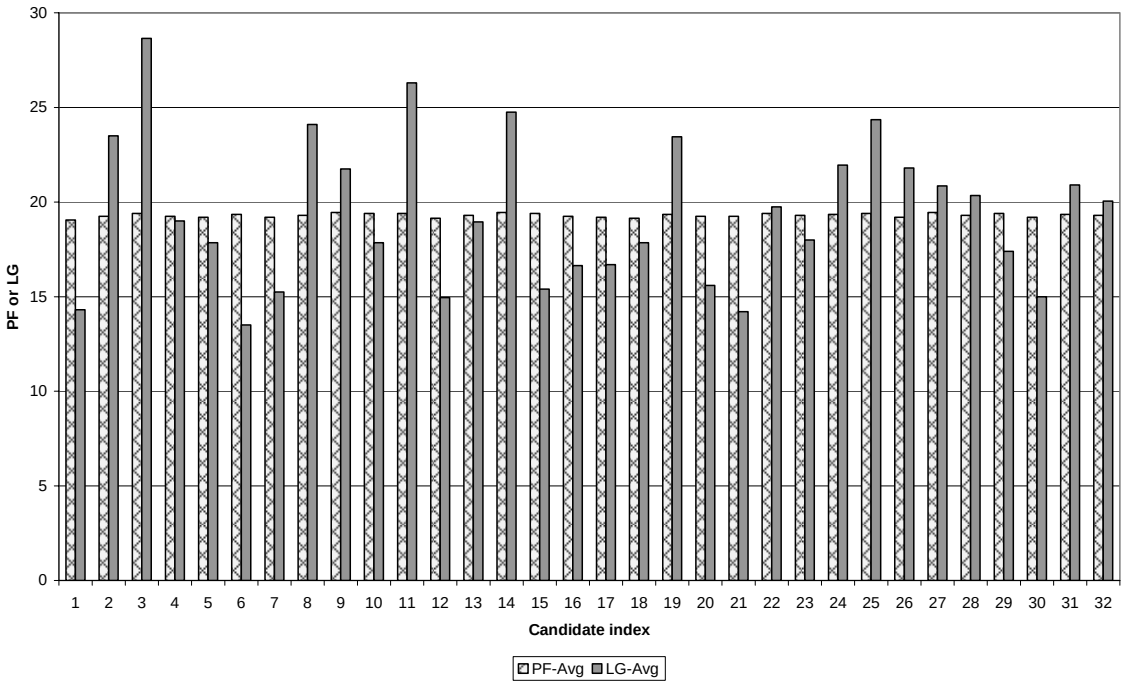

Figure 18: G2G achievement of *mt* on the average over 20 runs

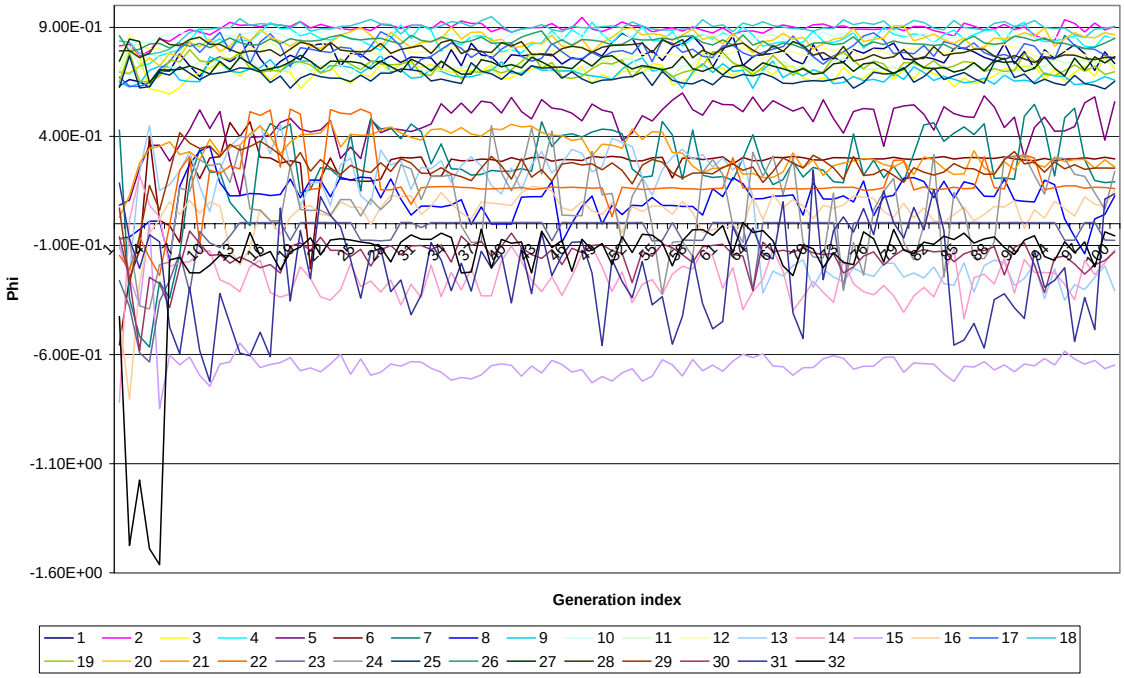

Figure 19: *Phi* graph of *mt* for a particular run

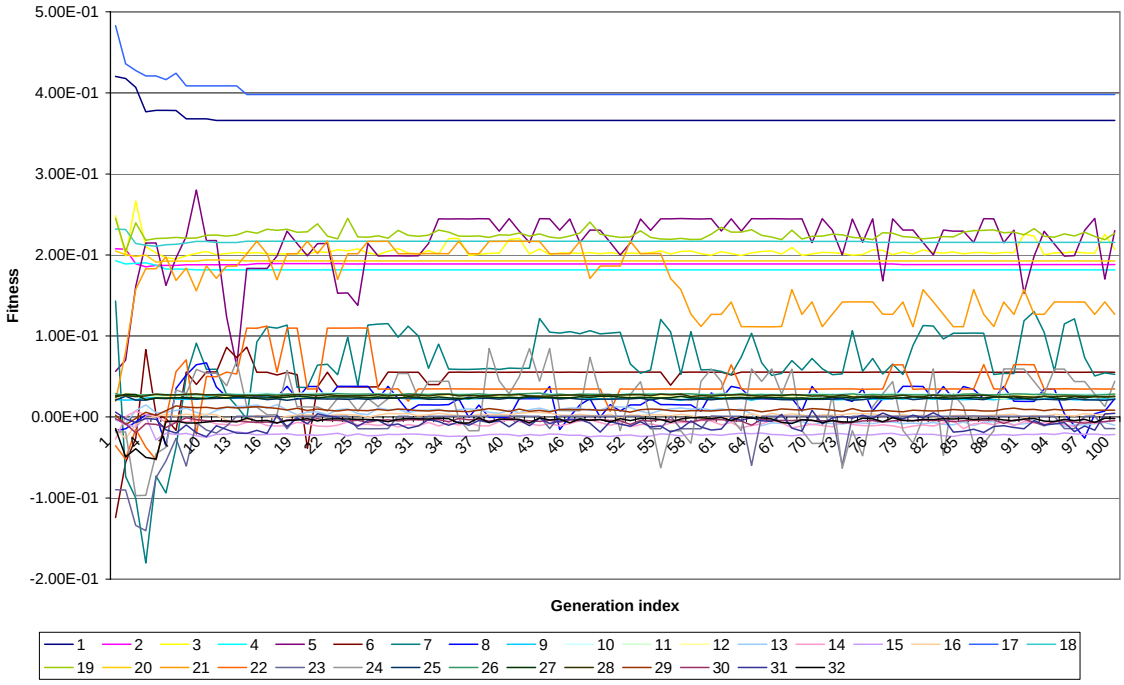

Figure 20: Best fitness graph of *mt* for a particular run

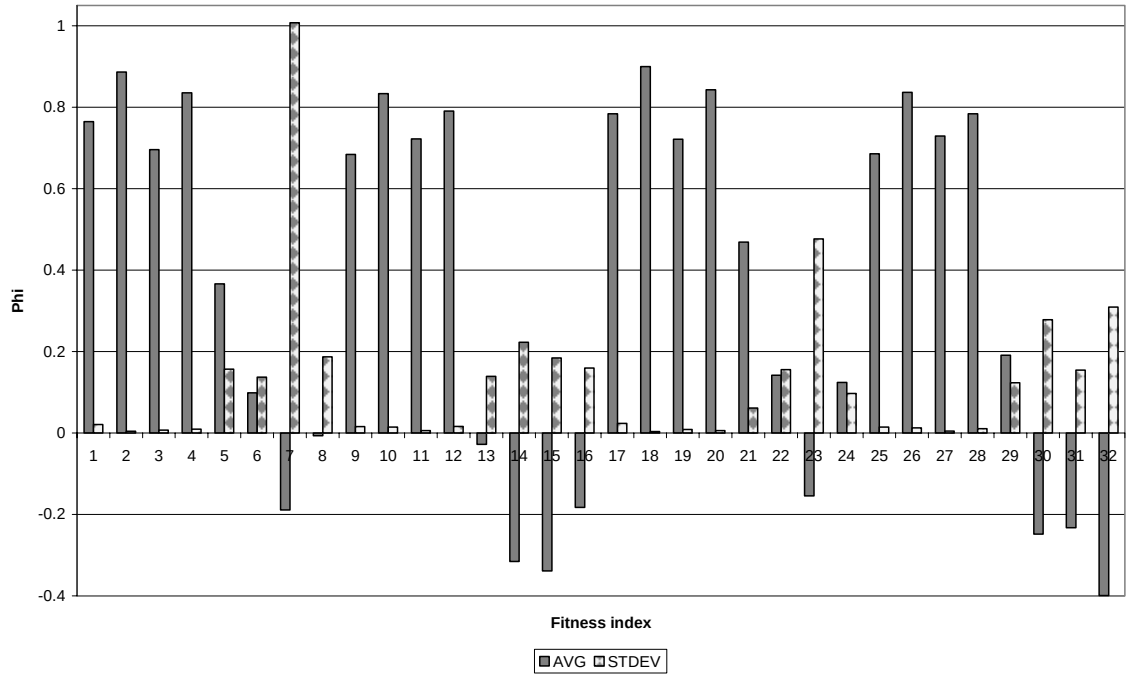

Figure 21: *Phi* average (over 20 runs) graph of *mt*

Most of the candidates (based on 20 runs, with 100 generations each) that employ rewarding scheme (indicated by negative *phi*) have higher level of exploration, since they have higher standard deviation (*see* Figure 21) and fluctuated *phi* average as can be seen on Figure 22.

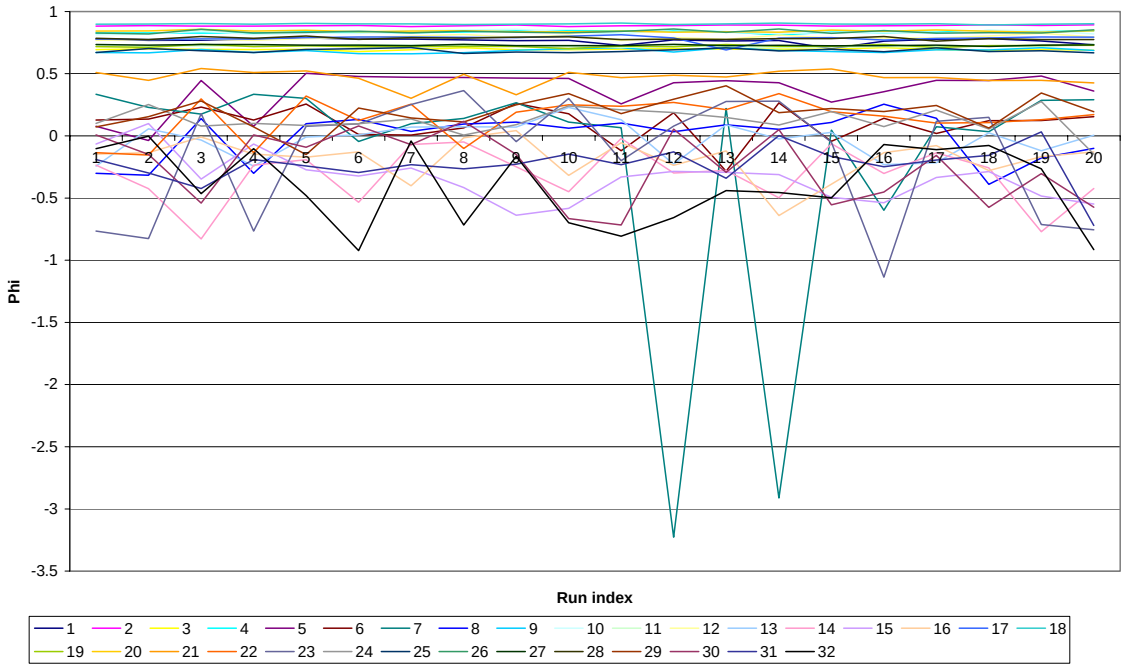

Figure 22: *Phi* average (over 100 generations) graph for each candidate of *mt*

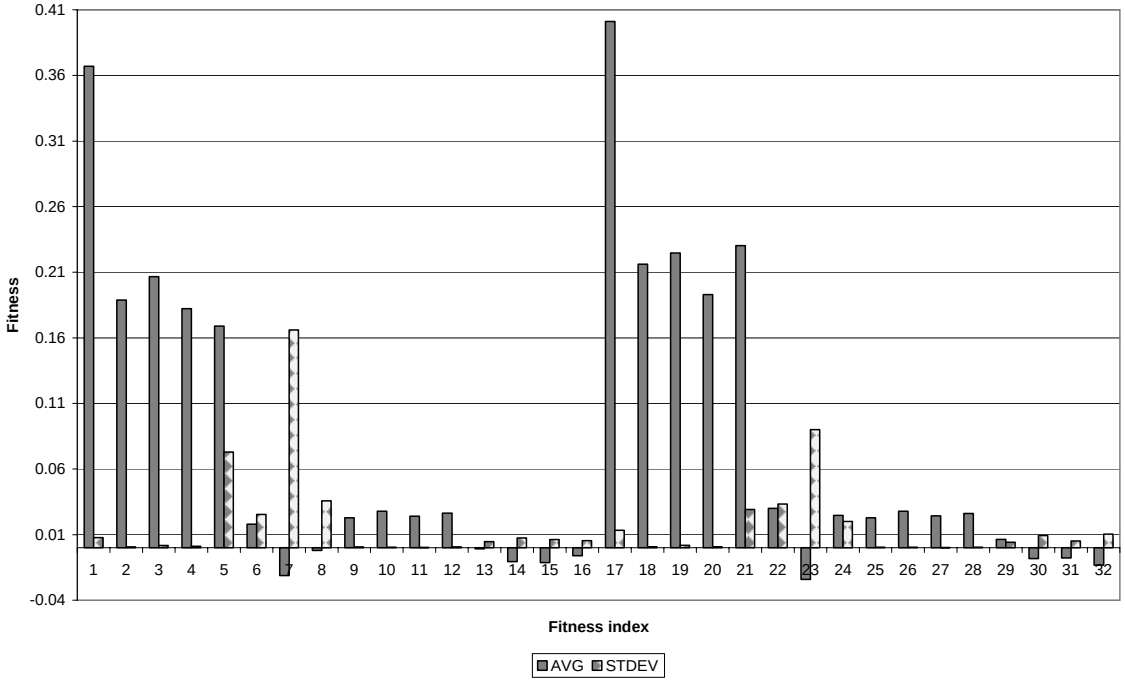

Figure 23: Best fitness average (over 20 runs) graph of *mt*

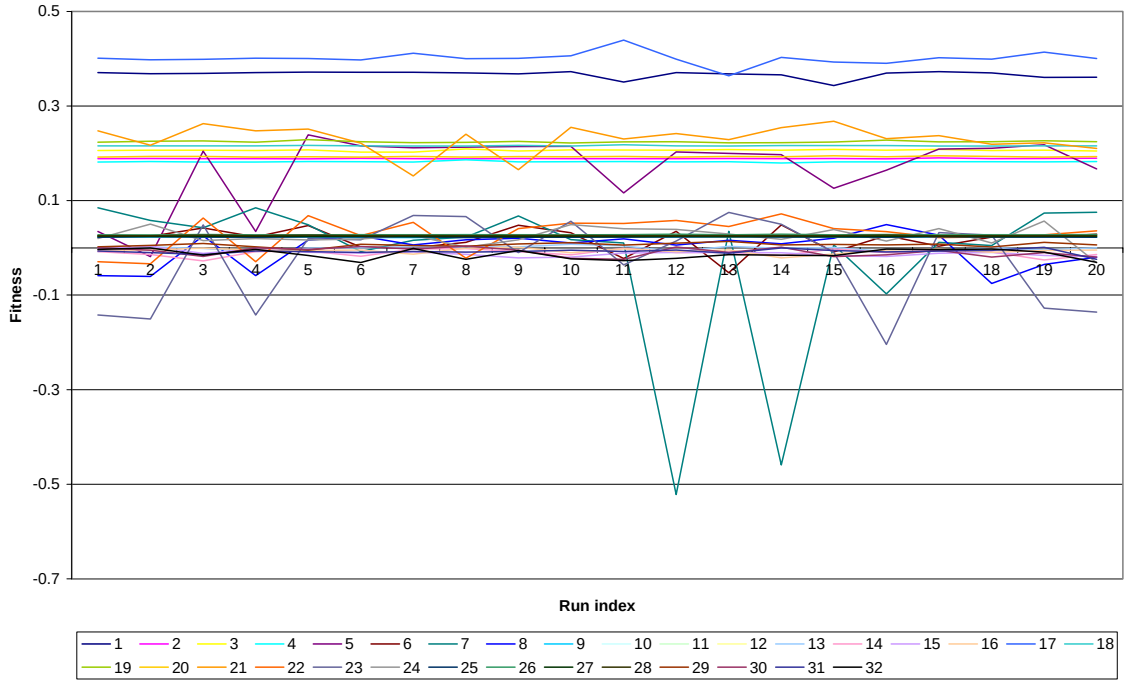

Figure 24: Best fitness average (over 100 generations) graph for each candidate of *mt*

Figure 23 plots the best fitness average over 20 runs for each candidate. The candidates that utilize rewarding scheme (*see* Figure 23; indicated by negative value) do exploration more (*see* Figure 24) than the ones without rewarding.

5.5. Analysis of Results

In this section we discuss the results in light of the effect of the existence of infeasible paths, path traversal techniques, neighborhood influence, rewarding, and group/cluster attributes view.

5.5.1. The Existence of Infeasible Paths

SUTs no 4 and 5, that is minimaxi (*mm*) program with feasible path only (*mm-f*), and with both feasible and infeasible path (*mm-i*), from Table 21 are used to measure the effect of infeasible path existence. The results are depicted in Figure 25 and Figure 26 that describe the effectiveness (PF) and efficiency (LG), respectively.

In the absence of infeasible paths (i.e., *mm-f*), on the average (over the 32 candidates, with 20 runs each), the number of PF, (Figure 25) is 12.57 out of 13; with a standard deviation 0.48. Average LG is 9.7, with a standard deviation of 4.05. On the other hand, with the presence of infeasible paths (i.e., *mm-i*; Figure 26), an average of 12.52 out of 13 feasible paths were found with a standard deviation of 0.5. Average LG is 13.56 with a standard deviation of 9.9. Therefore, the existence of infeasible paths are not hindering GA-based test data generator in finding the entire given feasible target paths; this assertion is concluded due to the observation that average PF of feasible vs. infeasible paths is 12.57 vs. 12.52. The major difference is that the maximum LG is higher if infeasible paths are present, that is 9.7 vs. 13.56 with a higher standard deviation.

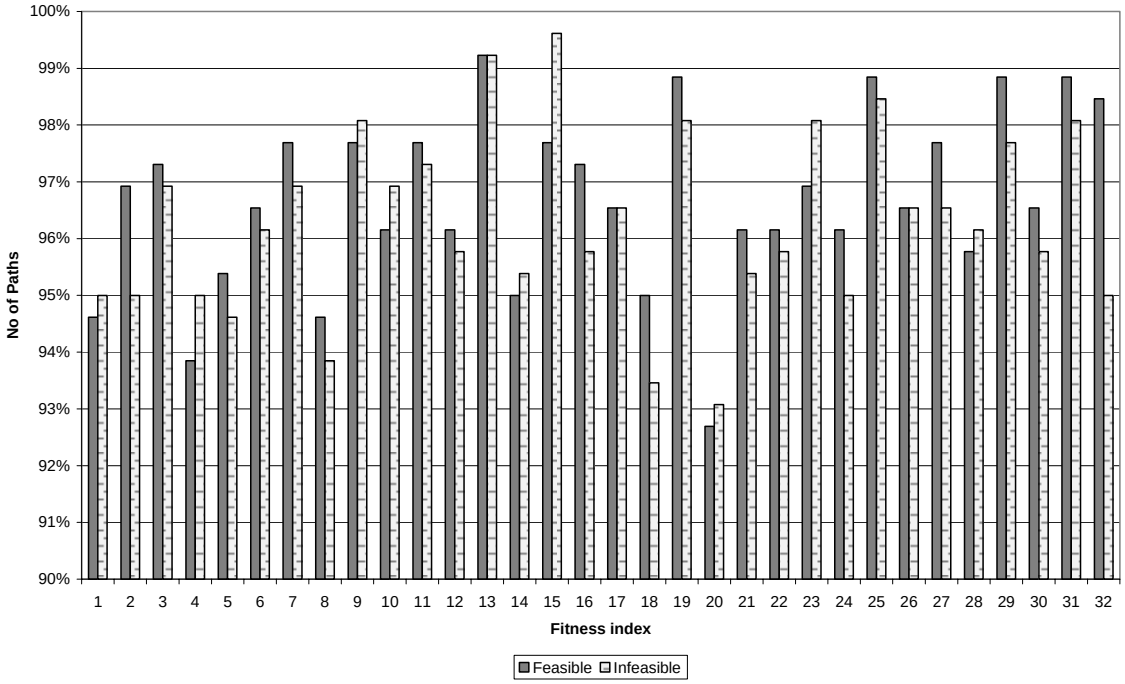

Figure 25: The effect of infeasible path to effectiveness

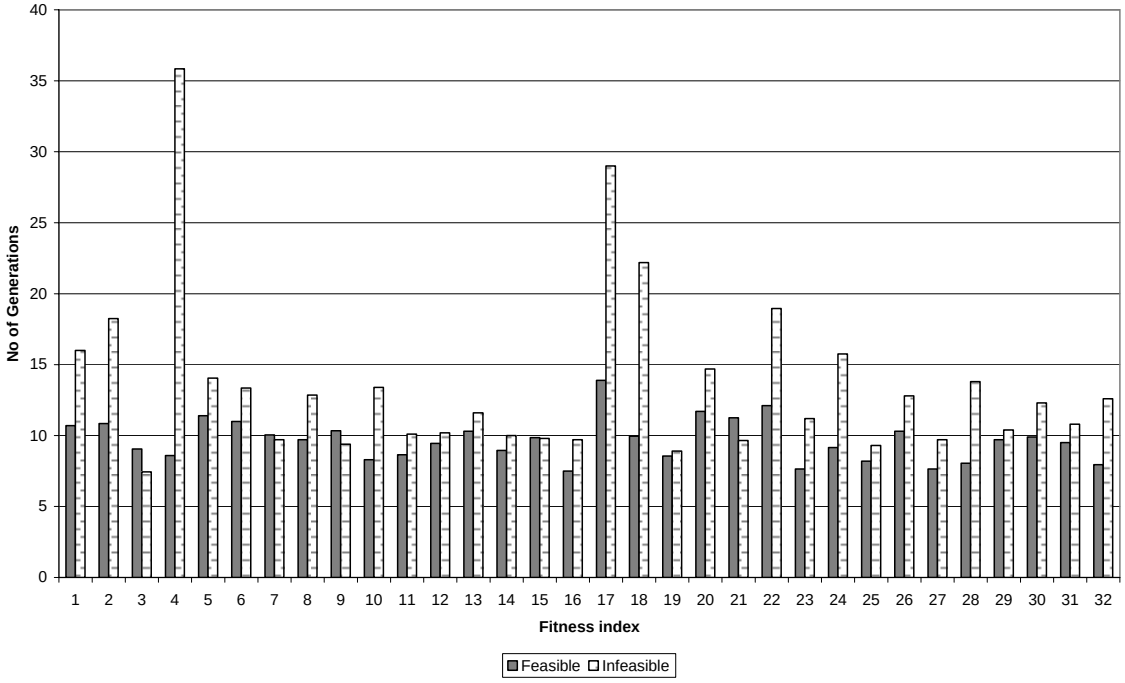

Figure 26: The effect of infeasible path to efficiency

5.5.2. Neighborhood Influence

Considering the *mm-f*, *mm-i*, and *mt* SUTs, we have summarized that the effect of neighborhood influence to the effectiveness and efficiency of the test data generator for all candidates are summarized in Table 22. On the average, the presence of neighborhood influence has a positive impact on performance of the corresponding candidates.

Note that -Avg2 suffix indicates average of average. For example, PF-Avg2 means that the average of PF of candidates that are categorized as having (or not having) neighborhood influence, which are also the average of PF over several runs, i.e. 10 runs for *mm-f*, 20 runs for both *mm-i* and *mt*.

Table 22: The effect of neighborhood influence to PF and LG

No	SUT	Neighborhood Influence			
		Absent		Present	
		PF-Avg2	LG-Avg2	PF-Avg2	LG-Avg2
1	*mm-f*	12.48	10.35	12.66	9.04
2	*mm-i*	12.42	16.12	12.61	10.99
3	*mt*	19.27	18.98	19.34	19.83

The average of PF and LG from Table 22 shows that candidates utilizing neighborhood influence are more effective (indicated by higher PF) than otherwise, but not more efficient (indicated by comparing the LGs). The following figures depict more comparisons between the competing candidates (*see* Figure 27 to Figure 32 for test programs: *mm-f*, *mm-i*, and *mt*). Figure 27 up to Figure 32 compare the performance of the candidates that share all variation points' settings, and they only differ on whether or not they allow neighborhood influence. Figure 27 and Figure 28 show the effect of

neighborhood influence to effectiveness and efficiency of *mm-f*. Figure 29 and Figure 30 show the effect of neighborhood influence to effectiveness and efficiency of *mm-i*. Figure 31 and Figure 32 show the effect of neighborhood influence to effectiveness and efficiency of *mm-t*.

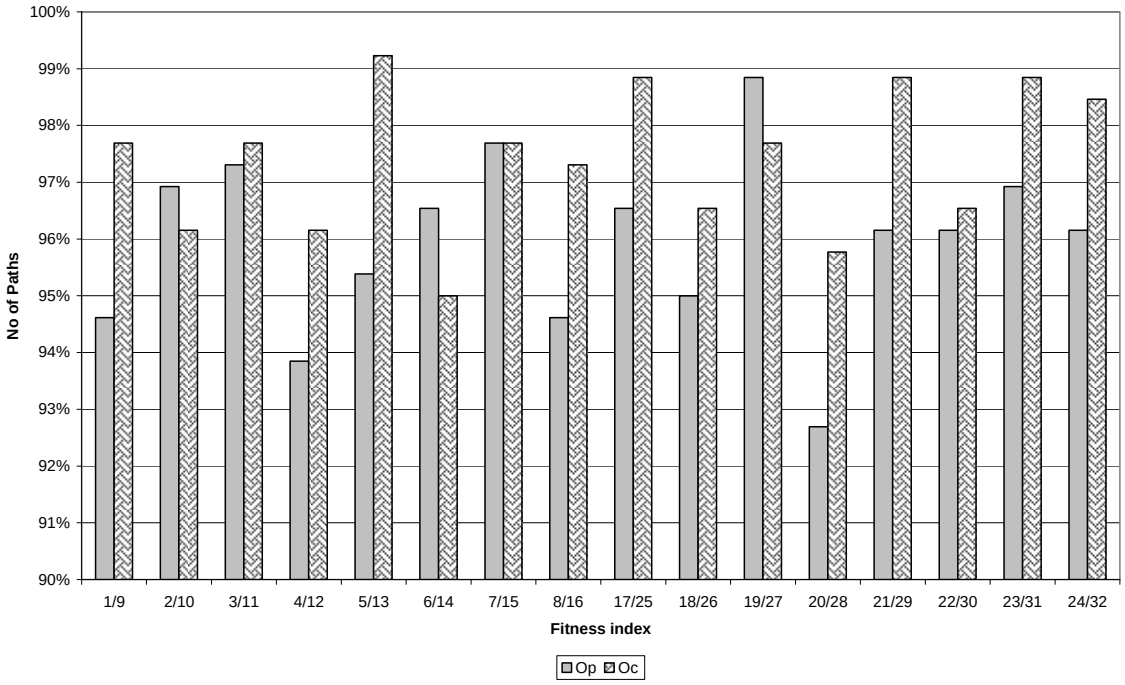

Figure 27: The effect of neighborhood influence to effectiveness for *mm-f*

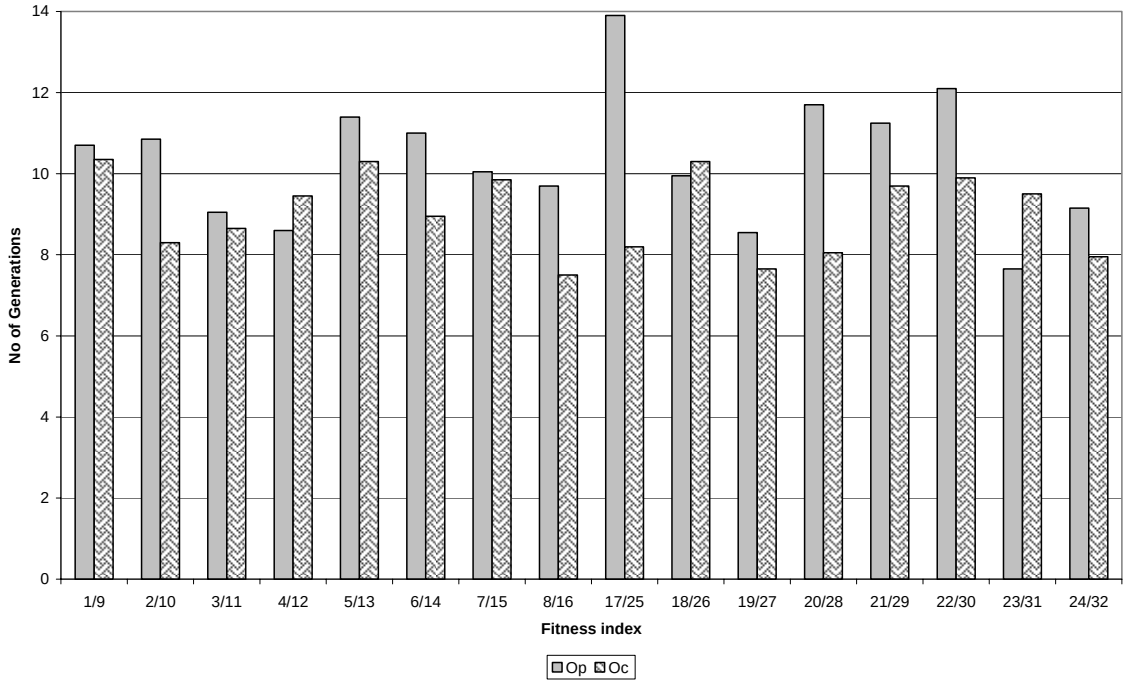

Figure 28: The effect of neighborhood influence to efficiency for *mm-f*

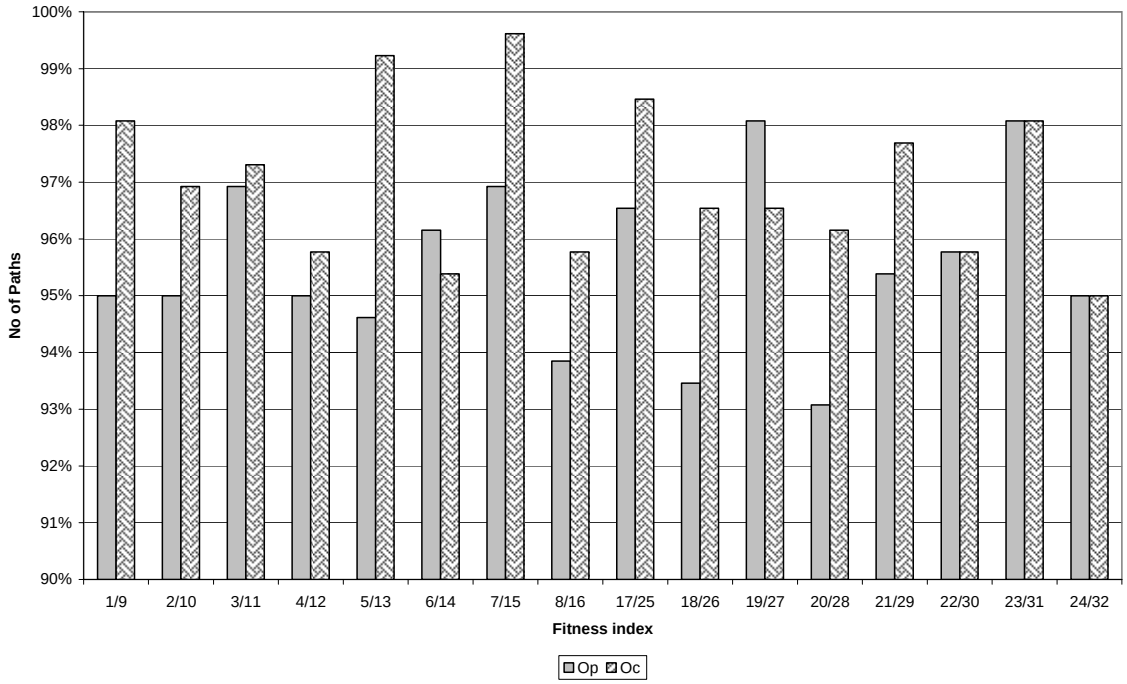

Figure 29: The effect of neighborhood influence to effectiveness for *mm-i*

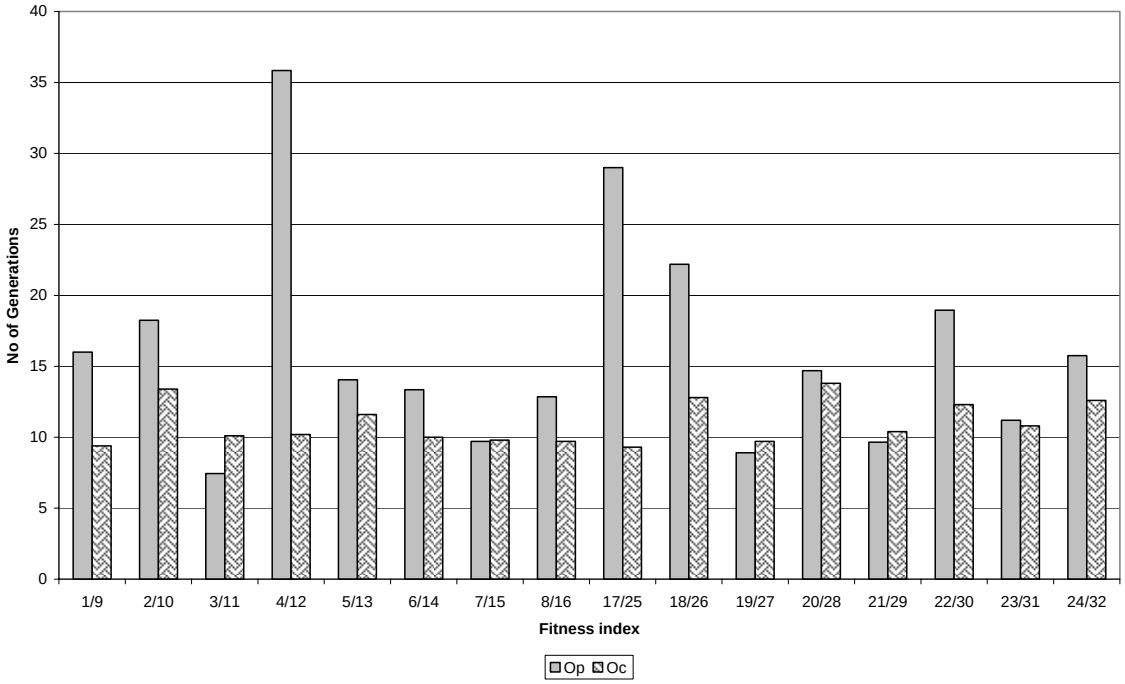

Figure 30: The effect of neighborhood influence to efficiency for *mm-i*

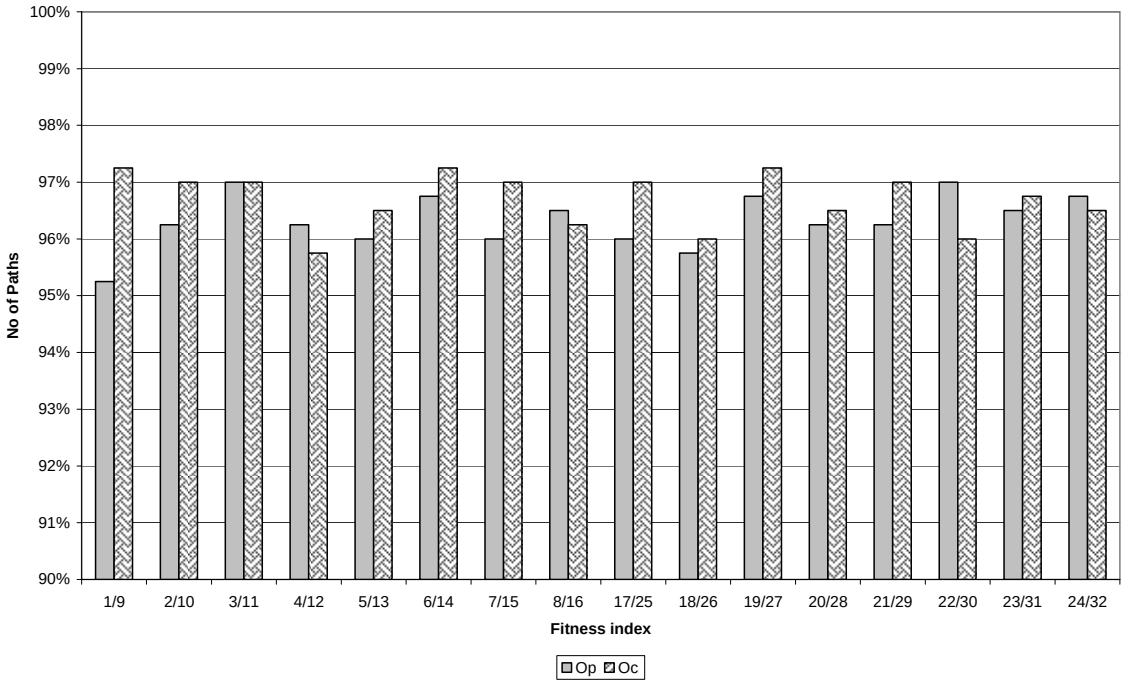

Figure 31: The effect of neighborhood influence to effectiveness for *mt*

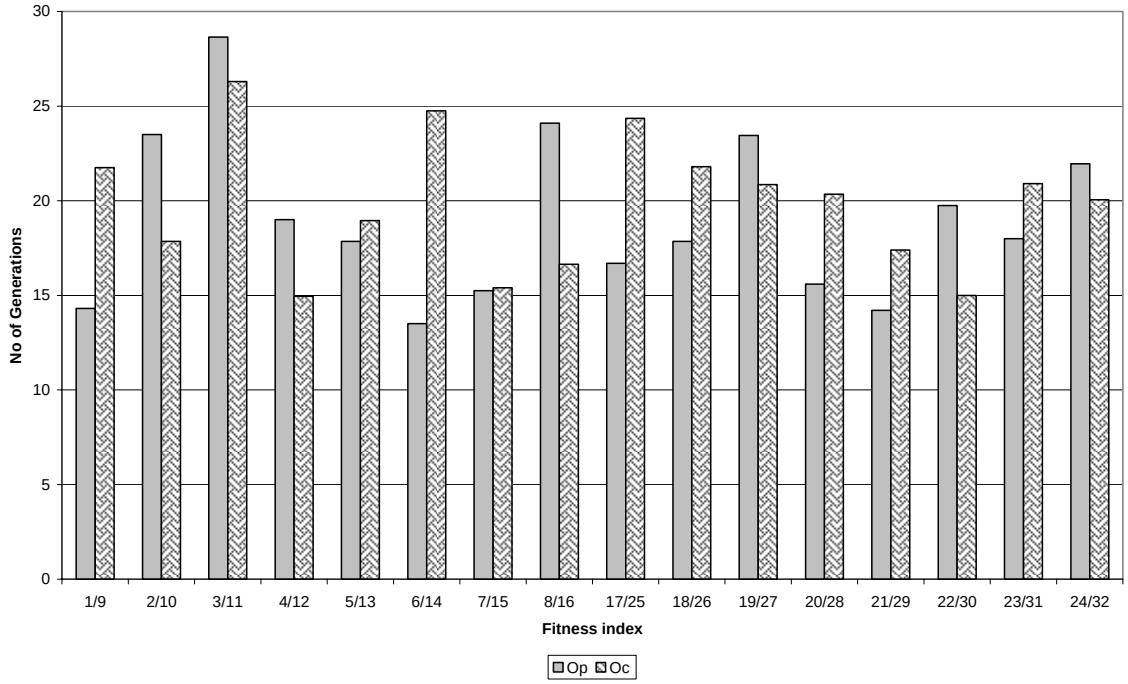

Figure 32: The effect of neighborhood influence to efficiency for *mt*

In general, neighborhood influence has increased the effectiveness of the fitness function candidates.

5.5.3. Path Traversal Technique

As we can see fromTable 23, on the average, there is no significant difference in performance, in term of effectiveness and efficiency, between the candidates applying the path-wise vs. the predicate-wise traversal technique.

However, the following figures depict more comparisons between the competing candidates (*see* Figure 33 to Figure 38 for test programs: *mm-f*, *mm-i*, and *mt*) both in term of effectiveness and efficiency. Figure 33 up to Figure 38 describe the fitness function

candidates that are applying path-wise (or predicate-wise) traversal techniques. Figure 33 and Figure 34 show the effect of path traversal technique to effectiveness and efficiency of *mm-f*. Figure 35 and Figure 36 show the effect of path traversal technique to effectiveness and efficiency of *mm-i*. Figure 37 and Figure 38 show the effect of path traversal technique to effectiveness and efficiency of *mm-t*.

Table 23: The effect of path traversal technique to PF and LG

No	SUT	Path Traversal Technique			
		Path-wise		Predicate-wise	
		PF-Avg2	LG-Avg2	PF-Avg2	LG-Avg2
1	*mm-f*	12.54	9.67	12.59	9.72
2	*mm-i*	12.53	13.23	12.51	13.88
3	*Mt*	19.30	19.55	19.30	19.26

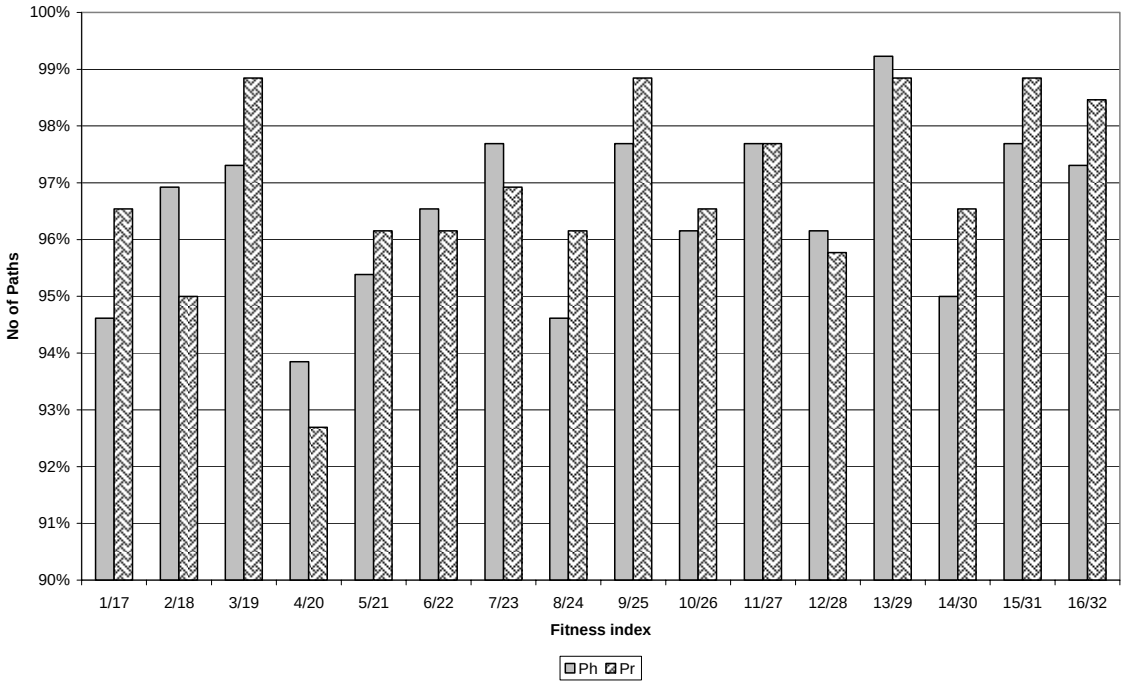

Figure 33: The effect of path traversal method to effectiveness for *mm-f*

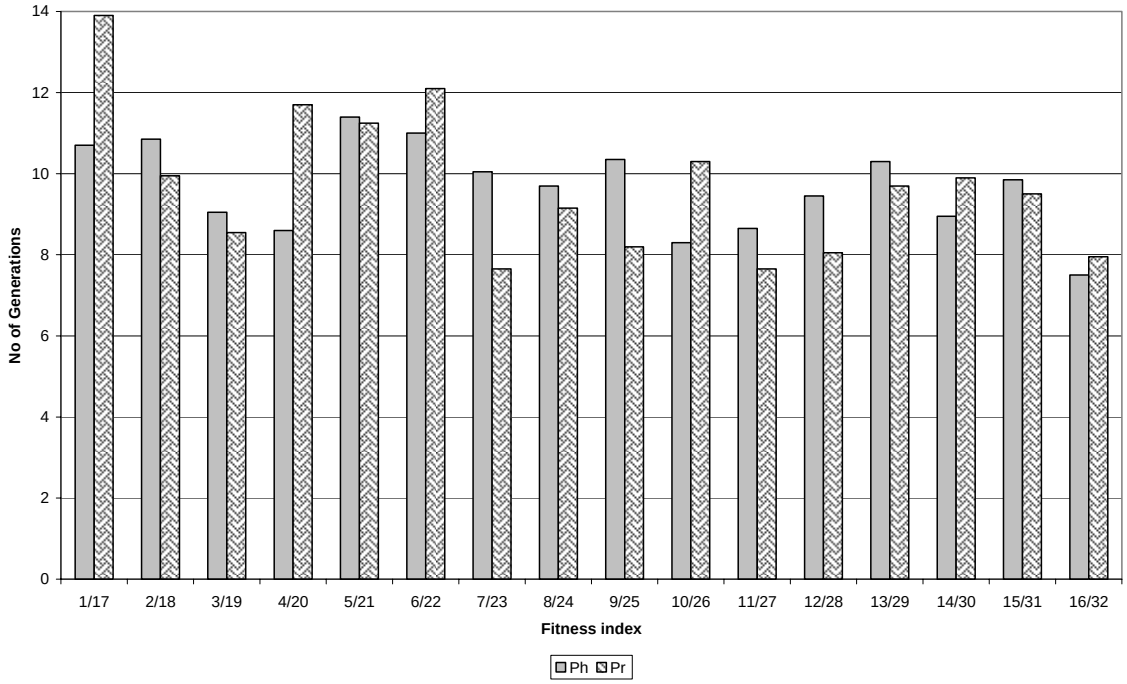

Figure 34: The effect of path traversal method to efficiency for *mm-f*

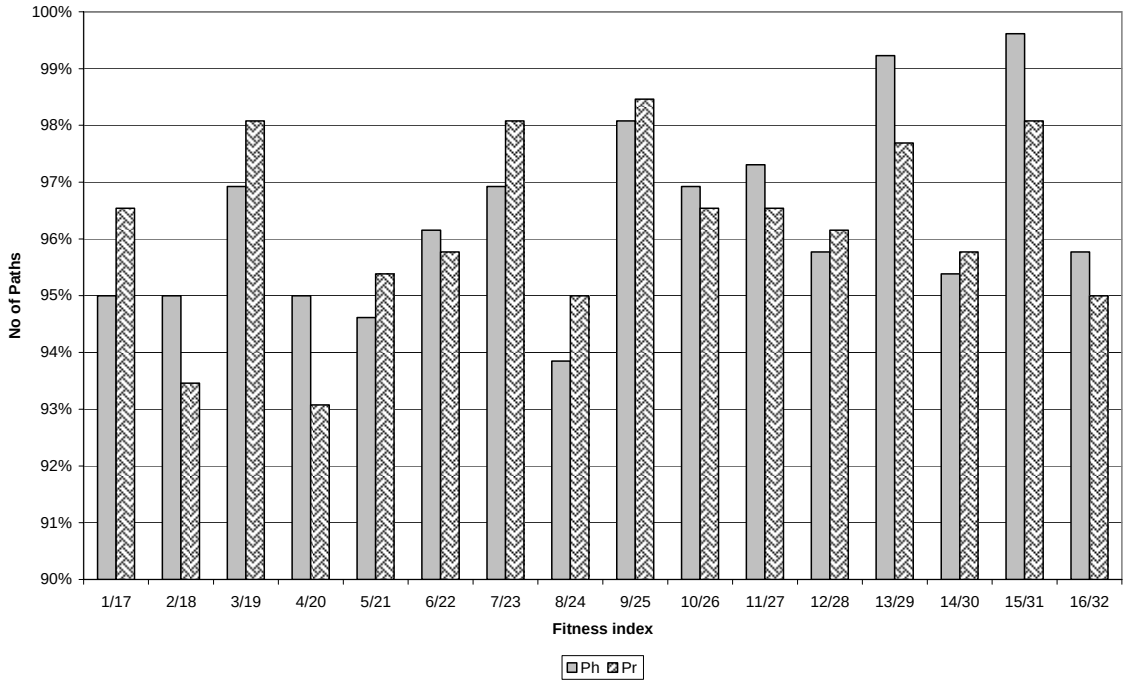

Figure 35: The effect of path traversal method to effectiveness for *mm-i*

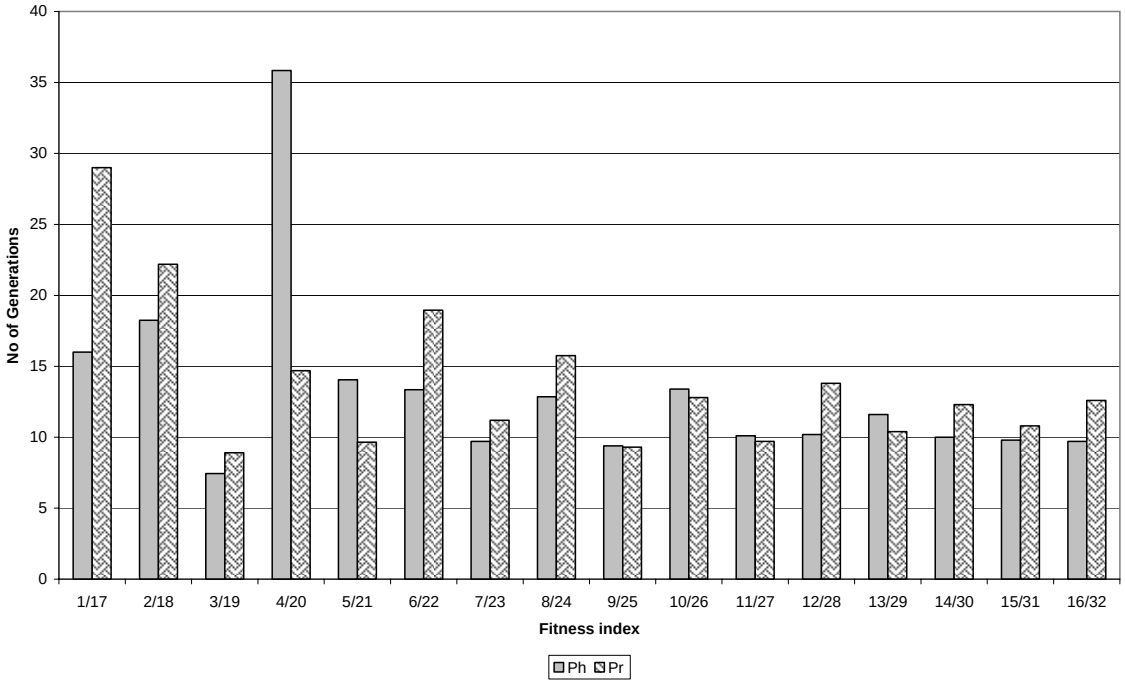

Figure 36: The effect of path traversal method to efficiency for *mm-i*

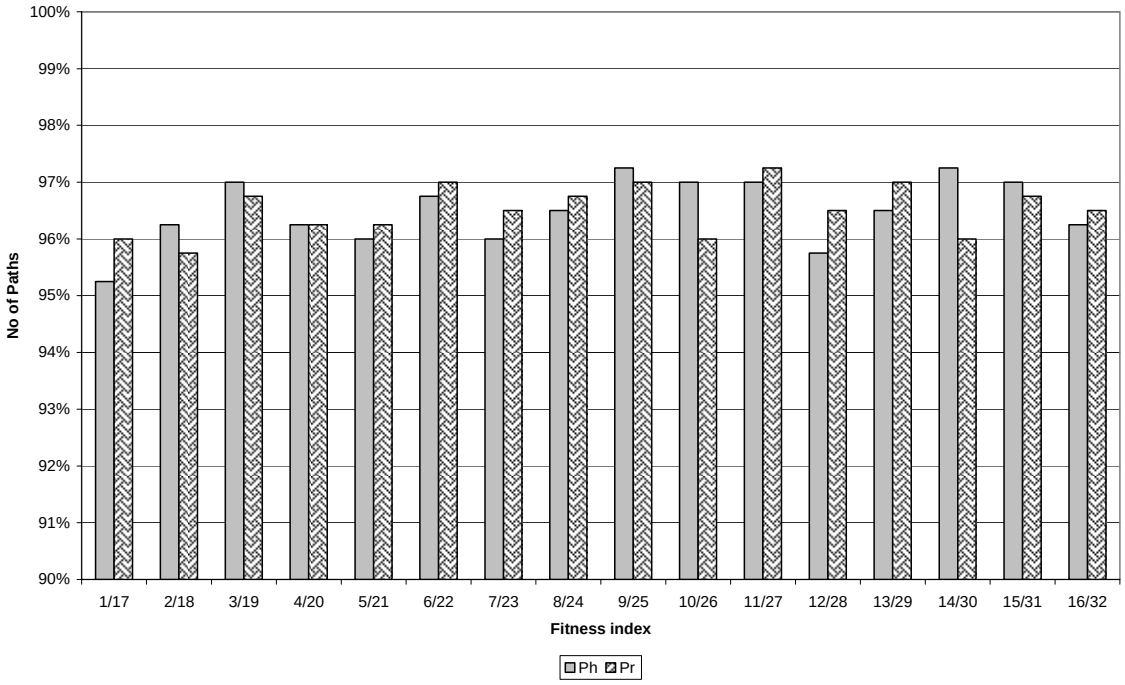

Figure 37: The effect of path traversal method to effectiveness for *mt*

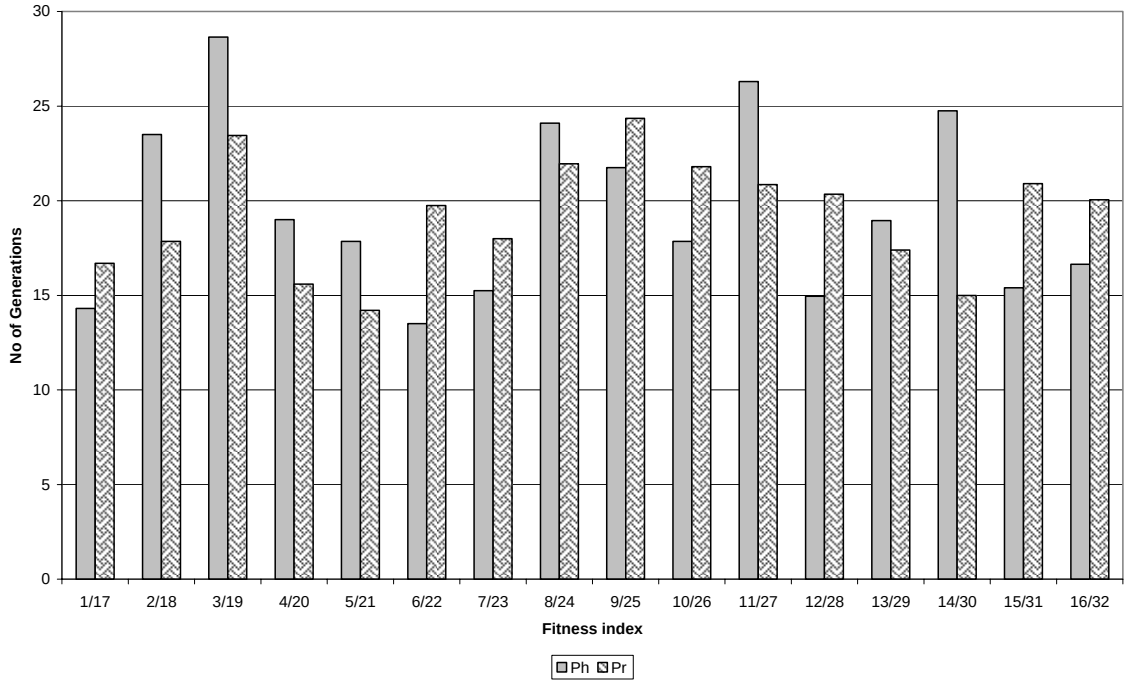

Figure 38: The effect of path traversal method to efficiency for *mt*

Neither path-wise nor predicate-wise traversal technique performs better than each other.

5.5.4. Weighting

On the average, distance-based weighting (Wd) outperforms other weighting schemas both in term of PF and LG (*see* Table 24). Moreover, no weighting (Wn) is the second best after Wd.

Not all the time that Wd performs better than the others, therefore we elaborate more on the behavior in Figure 43 and Figure 44 (test program *mt*). We, also, elaborate on the behavior with respect to *mm-f* and *mm-i* in Figure 39, Figure 40, Figure 41, and Figure 42.

Table 24: The effect of weighting to PF and LG

No	SUT	Weighting Scheme							
		Wn		Ws		Wd		Wv	
		PF-Avg2	LG-Avg2	PF-Avg2	LG-Avg2	PF-Avg2	LG-Avg2	PF-Avg2	LG-Avg2
1	*mm-f*	12.63	10.73	12.49	10.17	12.72	8.87	12.43	9.01
2	*mm-i*	12.59	13.68	12.43	15.16	12.70	9.71	12.34	15.68
3	*mt*	19.28	18.19	19.30	19.25	19.36	21.10	19.27	19.08

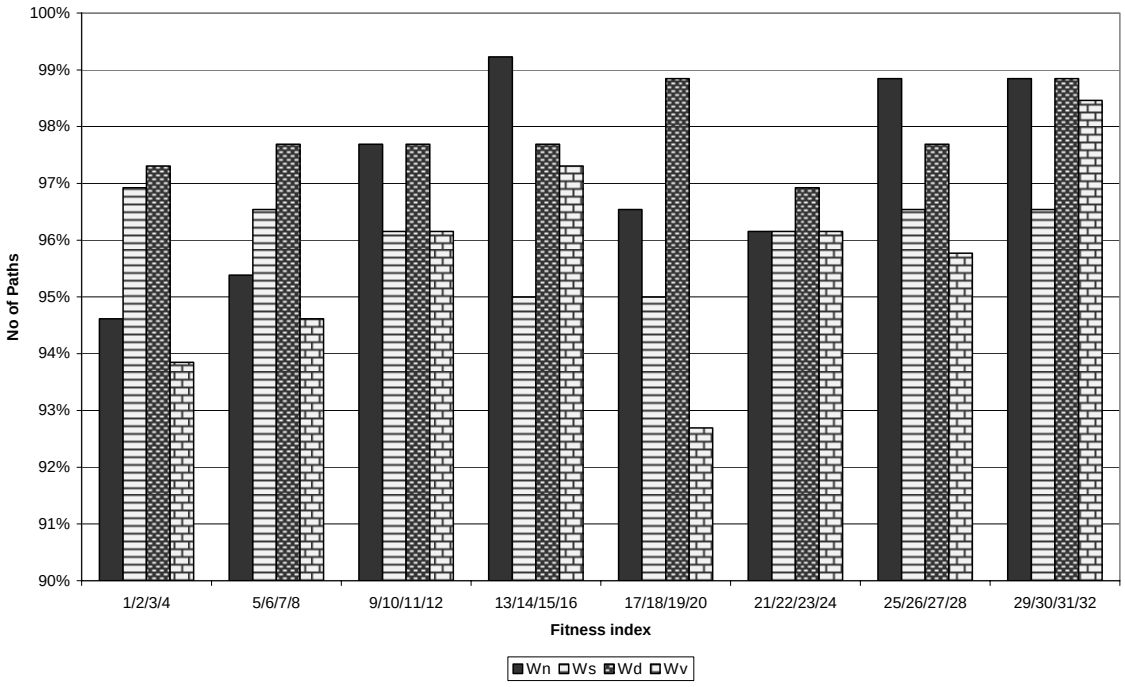

Figure 39: The effect of weighting to effectiveness for *mm-f*

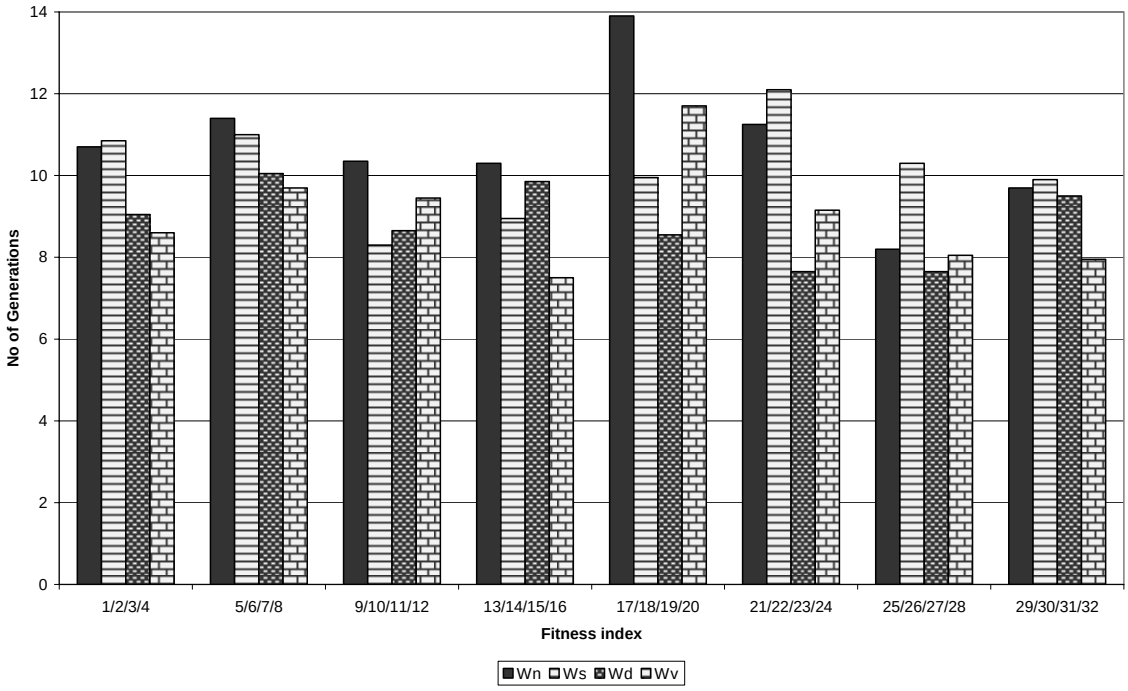

Figure 40: The effect of weighting to efficiency for *mm-f*

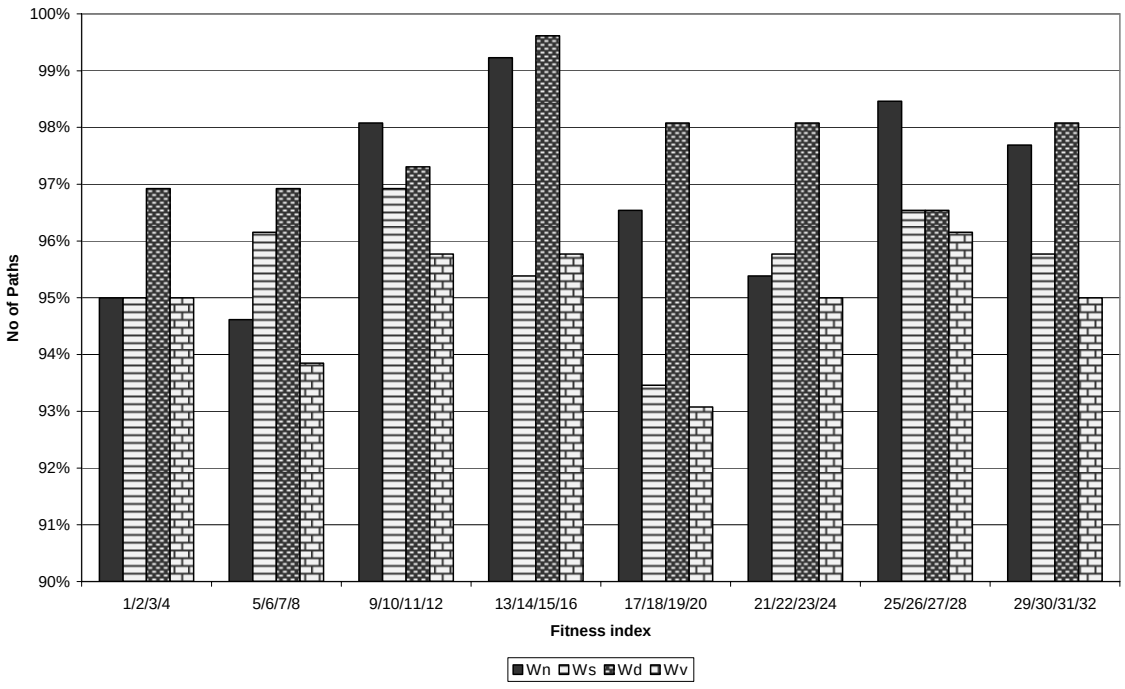

Figure 41: The effect of weighting to effectiveness for *mm-i*

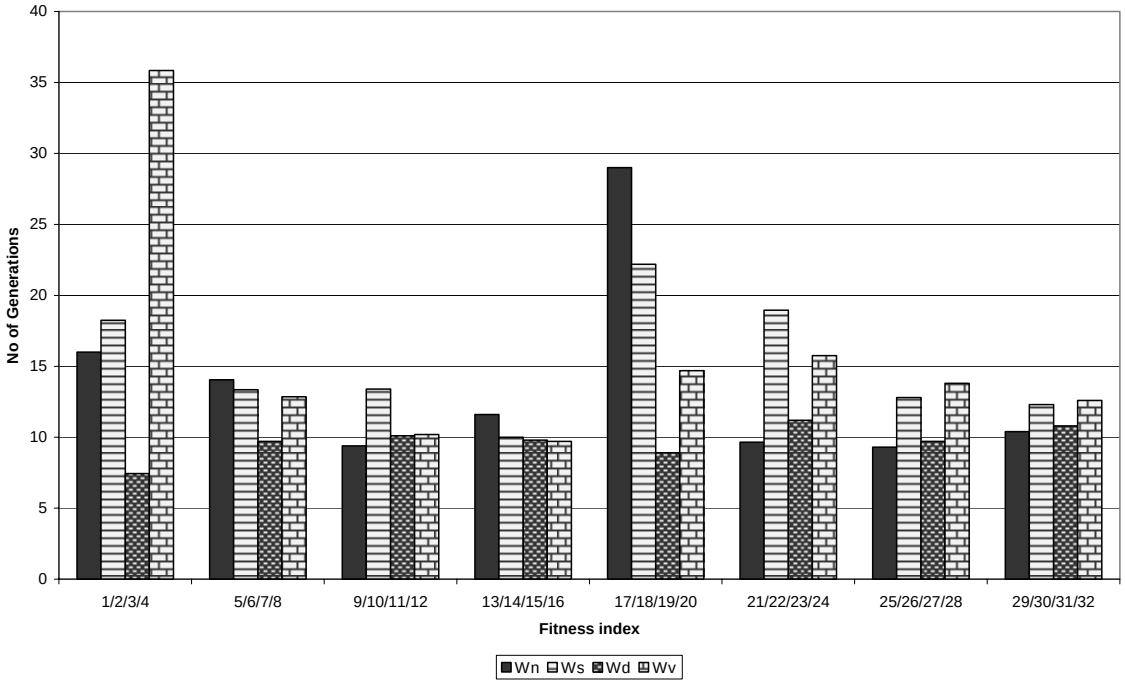

Figure 42: The effect of weighting to efficiency for *mm-i*

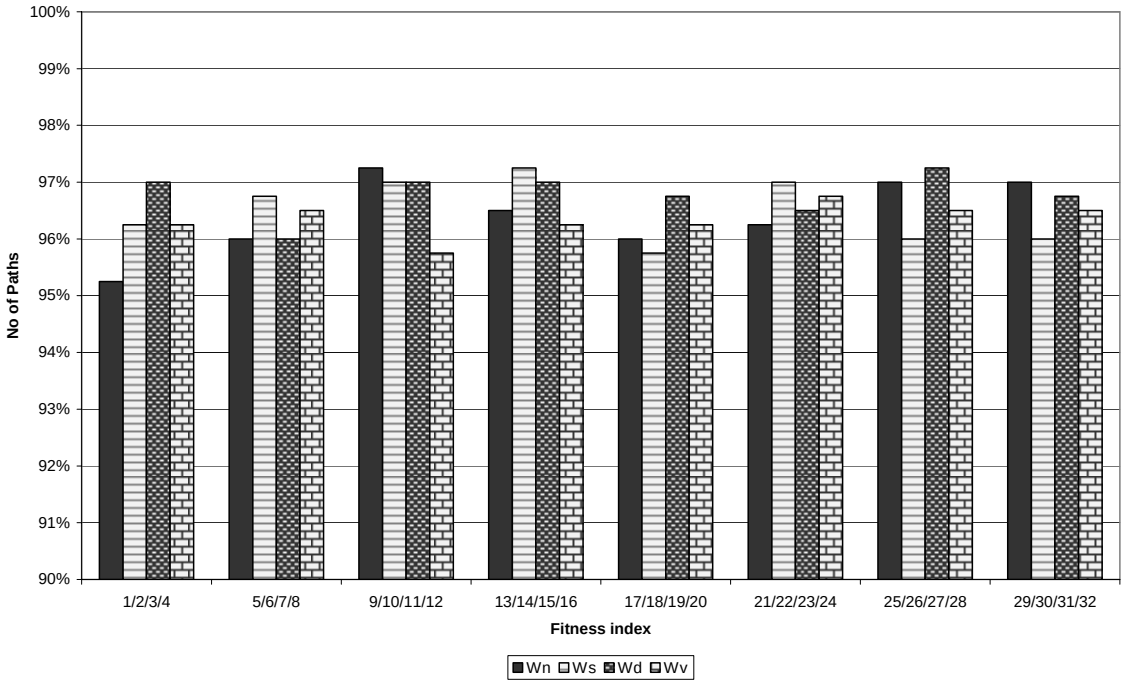

Figure 43: The effect of weighting to effectiveness for *mt*

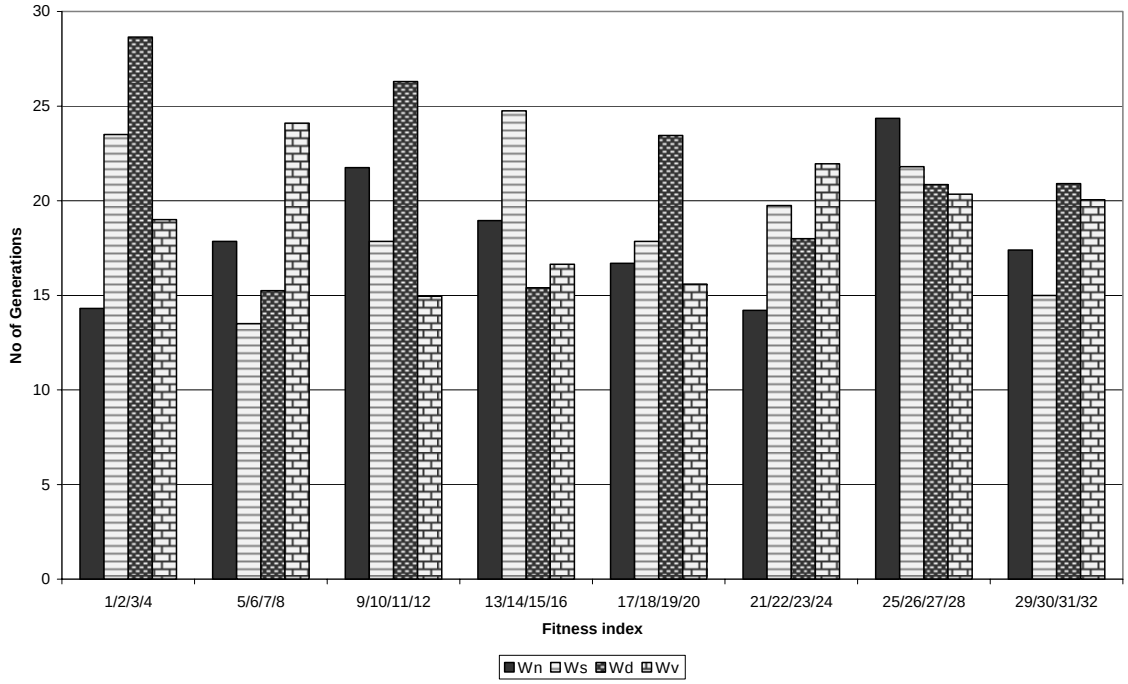

Figure 44: The effect of weighting to efficiency for *mt*

5.5.5. Rewarding

As shown in Table 25, rewarding does not really give significant enhancement in term of effectiveness. However, in term of efficiency, on the average, giving reward is better than its counterpart. More representative visualizations of the effect of rewarding are presented from Figure 45 up to Figure 50.

Table 25: The effect of rewarding to PF and LG

No	SUT	Reward			
		Absent		Present	
		PF-Avg2	LG-Avg2	PF-Avg2	LG-Avg2
1	*mm-f*	12.53	9.64	12.61	9.75
2	*mm-i*	12.50	15.07	12.53	12.04
3	*mt*	19.29	20.45	19.31	18.36

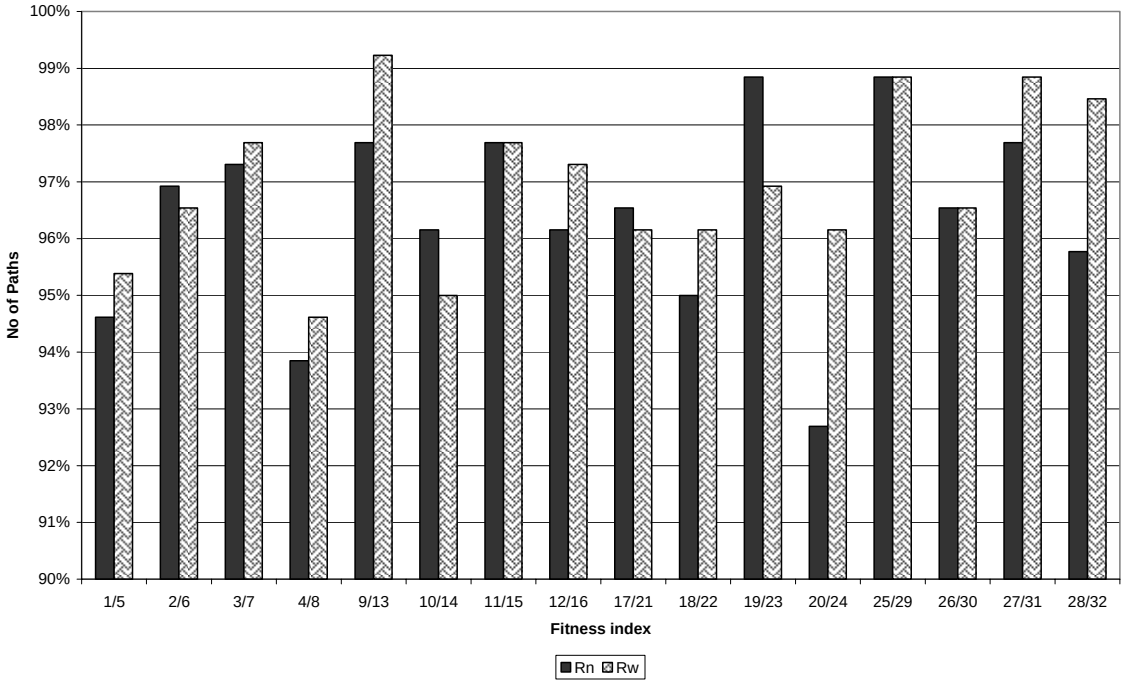

Figure 45: The effect of rewarding to effectiveness for *mm-f*

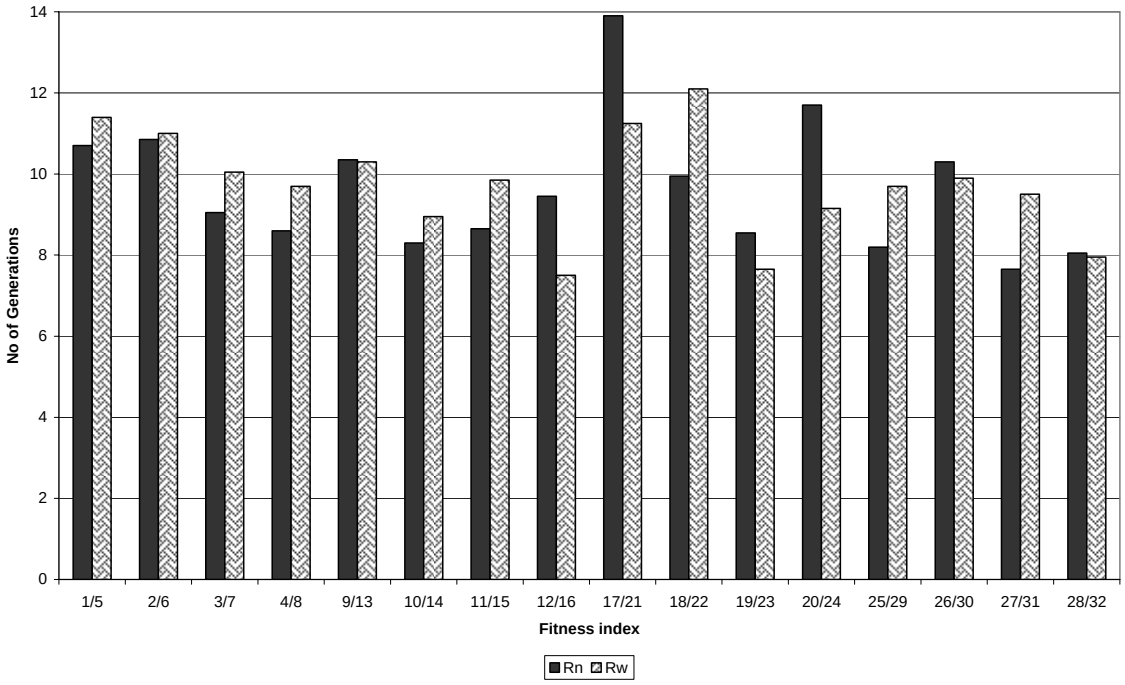

Figure 46: The effect of rewarding to efficiency for *mm-f*

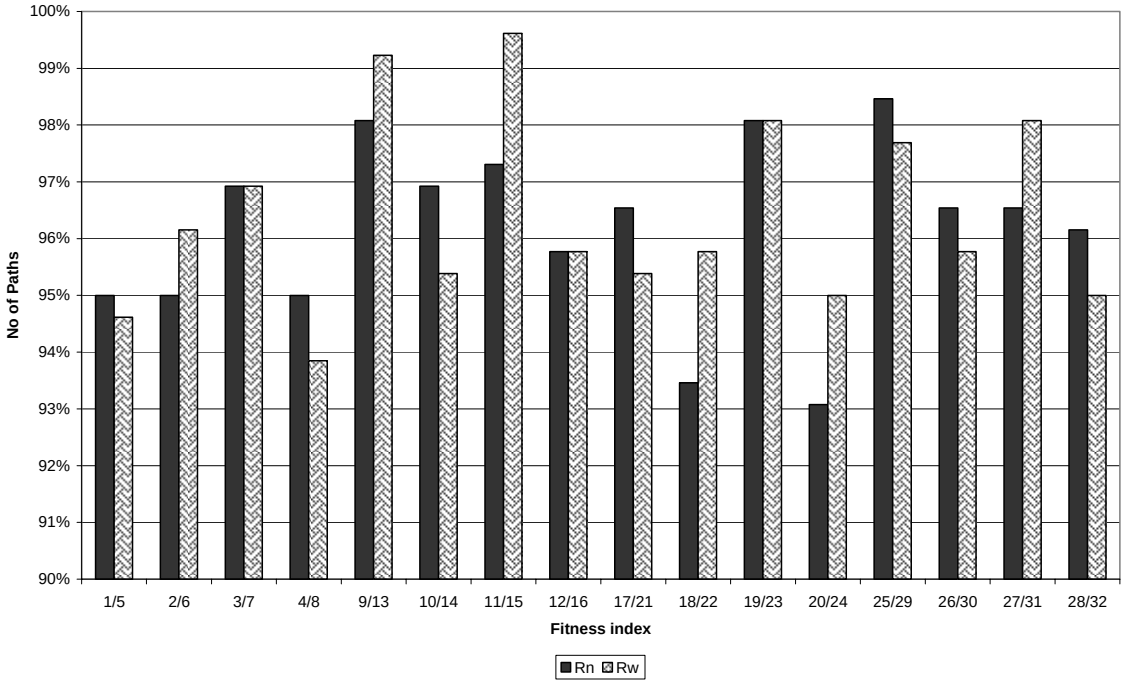

Figure 47: The effect of rewarding to effectiveness for *mm-i*

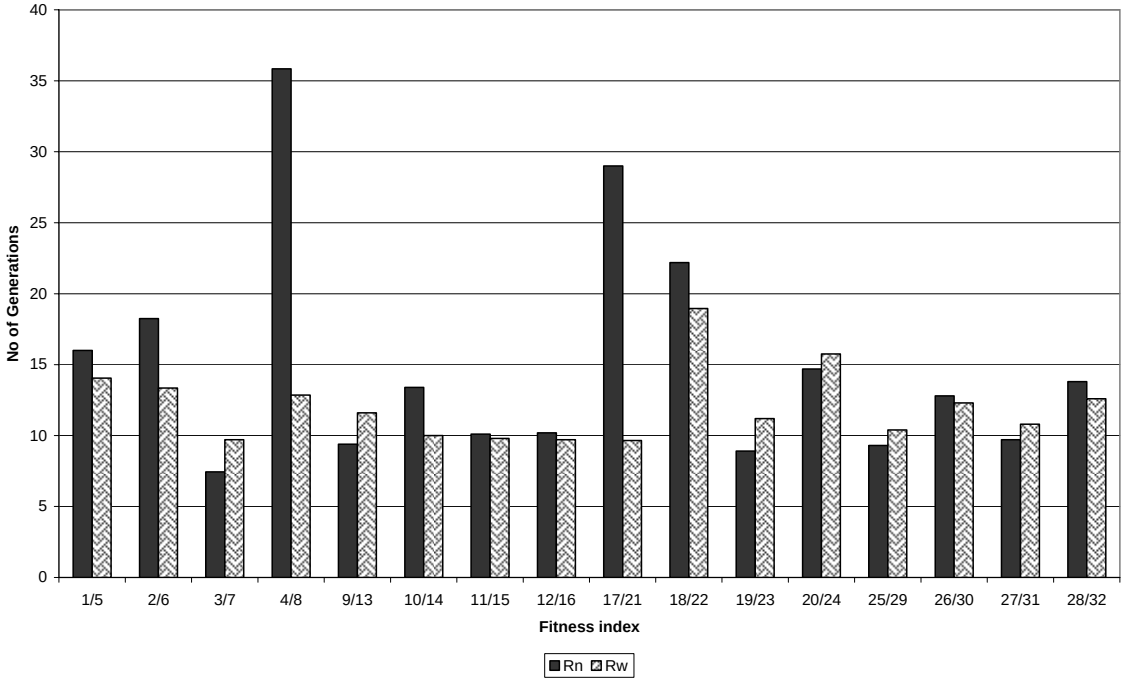

Figure 48: The effect of rewarding to efficiency for *mm-i*

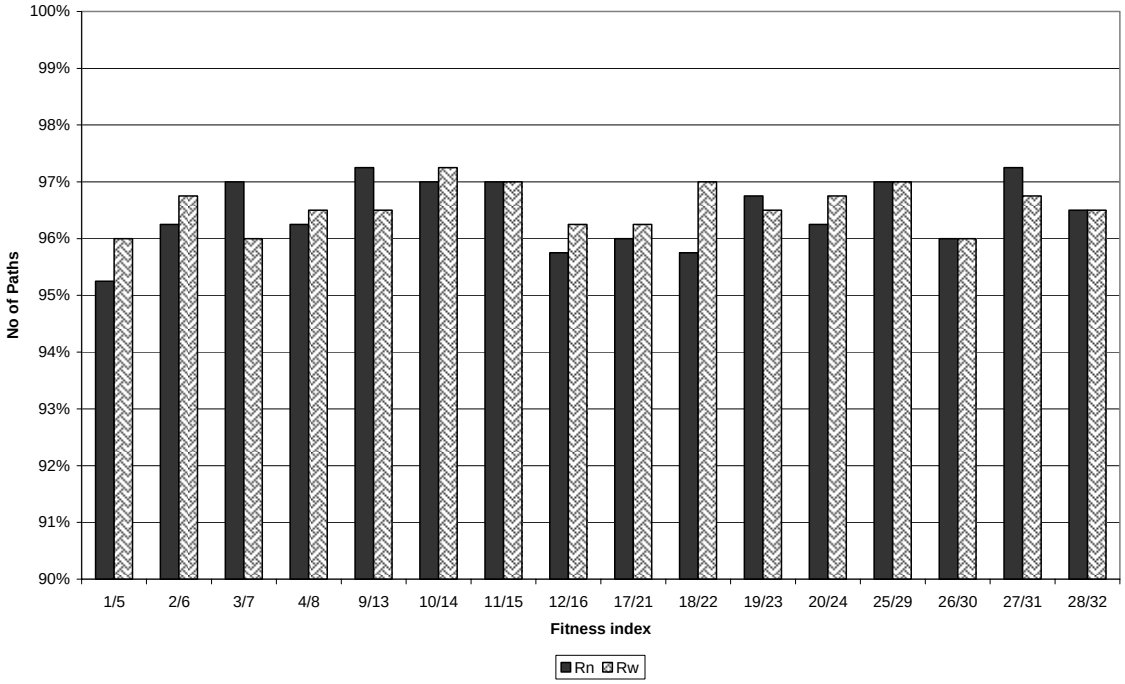

Figure 49: The effect of rewarding to effectiveness for *mt*

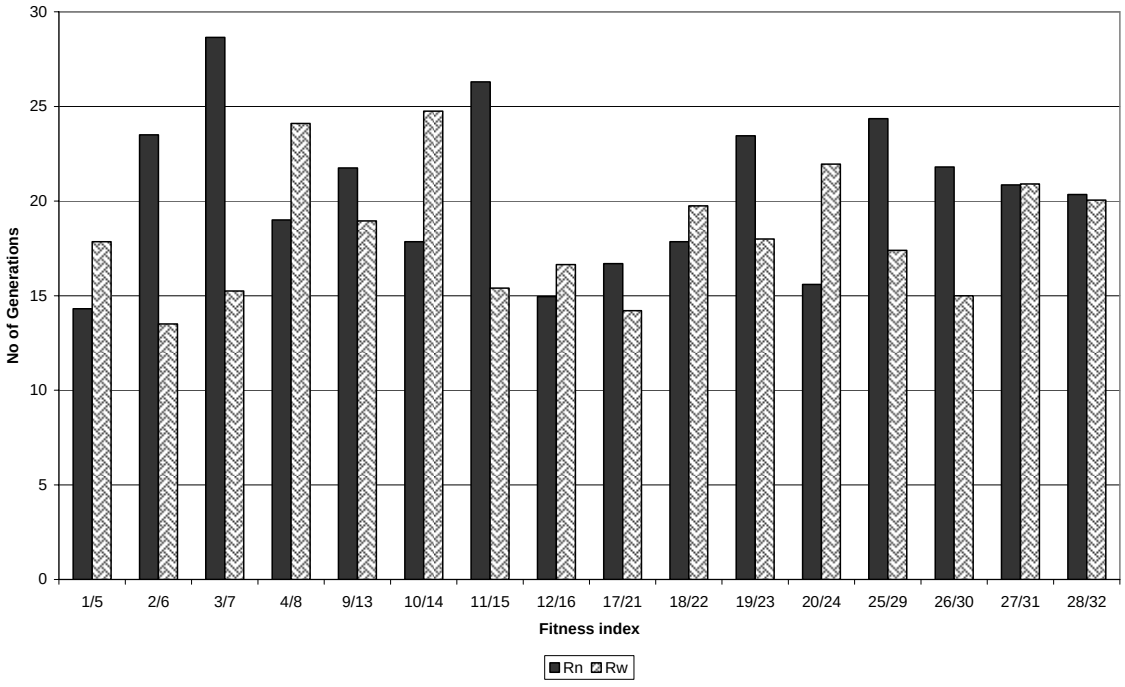

Figure 50: The effect of rewarding to efficiency for *mt*

5.5.6. Predicate Type

The compound predicate in a selection statement, e.g. (((A == B) & (B ~= C)) | ((B == C) & (C ~= A)) | ((C == A) & (A ~= B))), is expected to have an influence on the search progress. Our experiments confirmed this expectation. In our experiments, SUTs that contain compound predicates are *tr* and *mt*. In *tr*, the LG (on the average of 10 runs) is 7.6 generations in order to find 4 feasible target paths, while others require less LG with the same or higher number of target paths, e.g., *ns* (LG = 1.25 with 7 feasible target paths) and *is* (LG = 1.475 with 4 number of target paths). In *mt*, almost all (19.3 out of 20) feasible target paths were found within 20 generations (19.4 on the average). Please consult Appendix A for more detail about the predicates used in each test programs.

5.5.7. Path Length

Based on our experiments, the shorter target paths were covered in the earlier generations. However, a few short paths were covered in the middle generations.

5.5.8. Composite Analysis

As it might be clear from the previous sections, we could not really tell which attributes contribute significantly in efficiently finding all the required feasible target paths. Therefore, we needed a composite analysis, i.e. analyzing the result by fixing two or more attributes (variation points) at a time as opposed to only single one; hoping we can find a distinguishing pair of attributes.

The following shows our observations on the candidates' performance with regard to test programs which have infeasible target paths and are considered to be more complex than the others:

- *mm-f*: The best candidates are fitness function no 13 and 16, in term of PF and LG, respectively (*refer to* Figure 9).
- *mm-i*: The best candidates are fitness function no 15 and 3, in term of PF and LG, respectively (*refer to* Figure 10).
- *mt*: The best candidates are fitness function no 9 and 14, in term of PF, and 6 in term of LG (*refer to* Figure 18).

In term of PF, the common attributes are path-wise traversal technique (Ph) and neighborhood influence (Oc), although rewarding also gives a significant contribution in some experiments. And, in term of LG, the common attributes are path-wise traversal technique and without neighborhood influence (Op).

Thus, based on these two combined attributes, i.e. path traversal method tight together with neighborhood influence, we summarize the PF & LG in Table 26, and plot the bar charts for those three test programs in Figure 51 to Figure 55.

Based on the average of composite analysis (Table 26), the PhOc pair combination performs better than the others. However, PhOc pair combination does not outperform the others all the time, thus, for a closer look at the behavior of these attributes, we present three graphs for the three SUTs (Figure 51 to Figure 56): *mm-f*, *mm-i*, and *mt*. In Figure 51 and Figure 52, PrOc pair combination outperforms the others. In Figure 53 and Figure

54, PhOc pair combination outperforms the others, on the average, in terms of both PF and LG. However, in Figure 55 and Figure 56, PhOc pair combination outperforms the others in term of PF, and PrOp pair combination outperforms the others in term of LG.

Table 26: Path traversal and influence pair for composite analysis

No	SUT	Combination of path traversal and neighborhood influence							
		PhOp		PhOc		PrOp		PrOc	
		PF-Avg2	LG-Avg2	PF-Avg2	LG-Avg2	PF-Avg2	LG-Avg2	PF-Avg2	LG-Avg2
1	*mm-f*	12.46	10.17	12.63	9.17	12.49	10.53	12.70	8.91
2	*mm-i*	12.41	15.94	12.64	10.53	12.44	16.29	12.58	11.46
3	*mt*	19.25	19.52	19.35	19.58	19.28	18.44	19.33	20.09

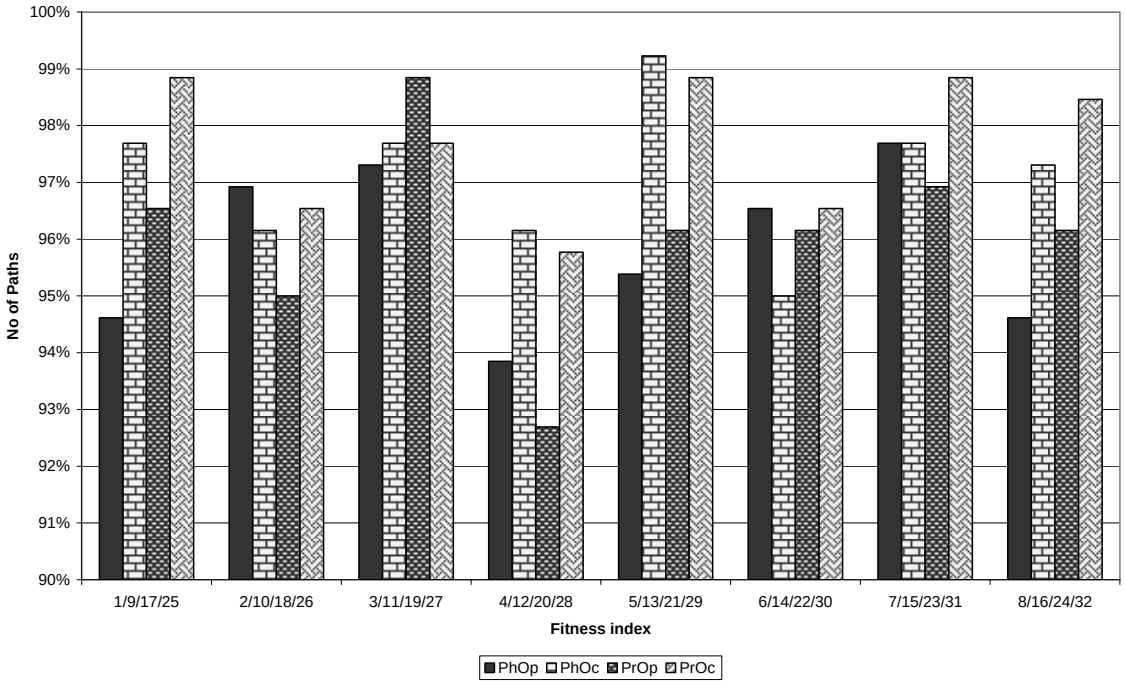

Figure 51: Composite analysis of effectiveness for *mm-f*

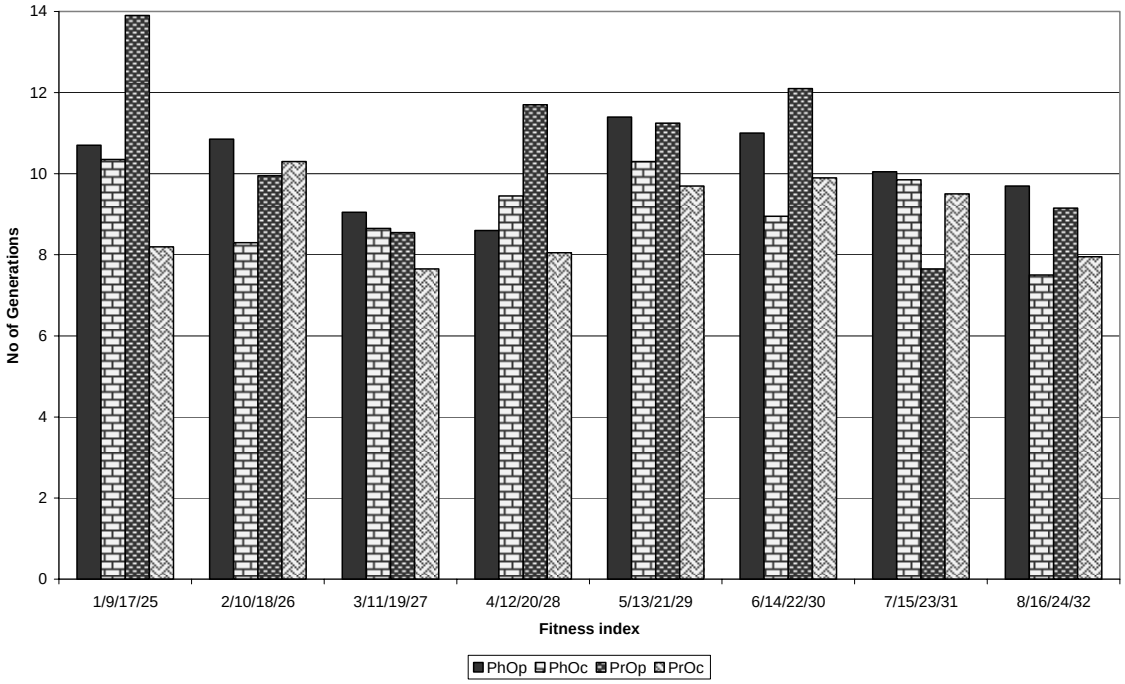

Figure 52: Composite analysis of efficiency for *mm-f*

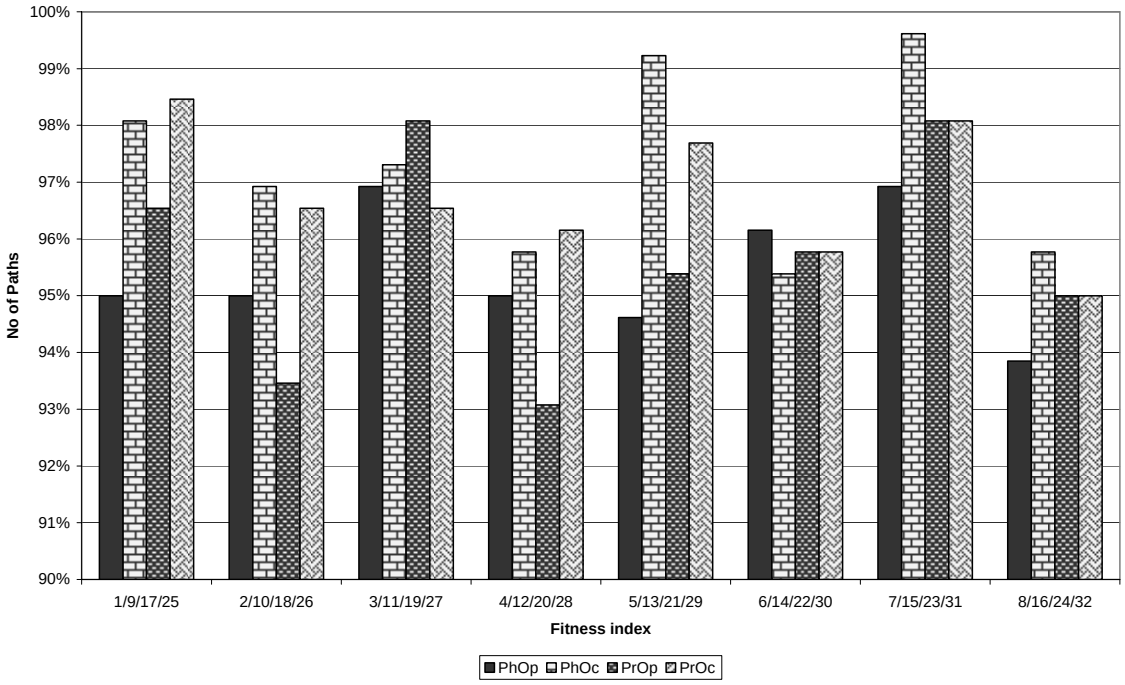

Figure 53: Composite analysis of effectiveness for *mm-i*

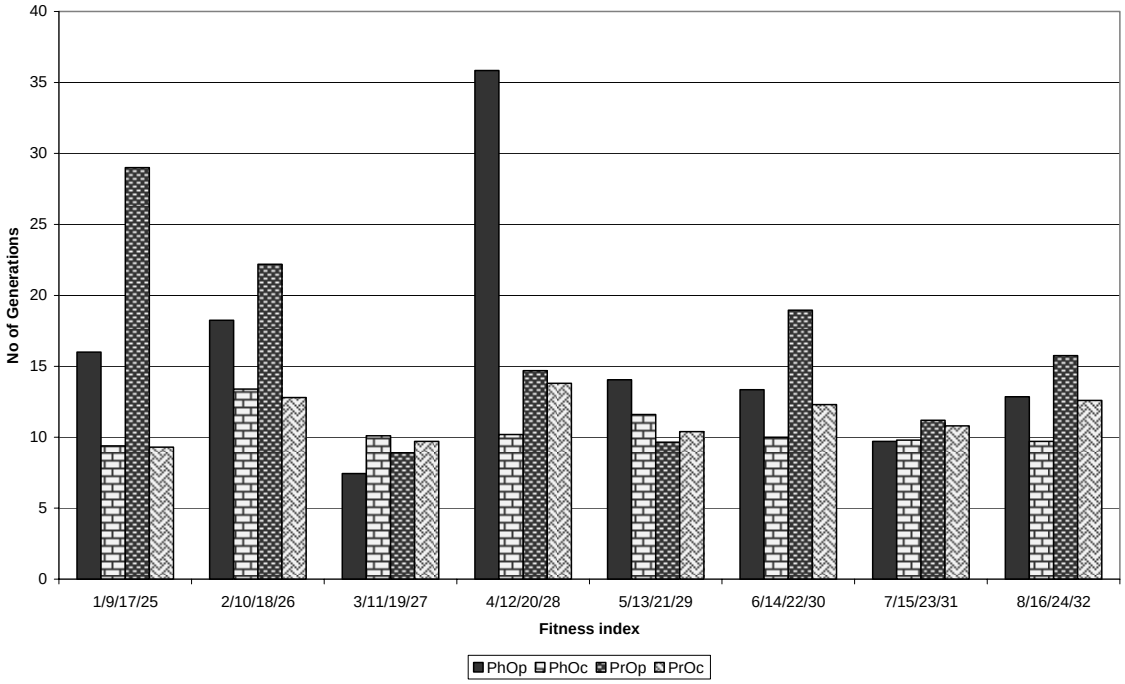

Figure 54: Composite analysis of efficiency for *mm-i*

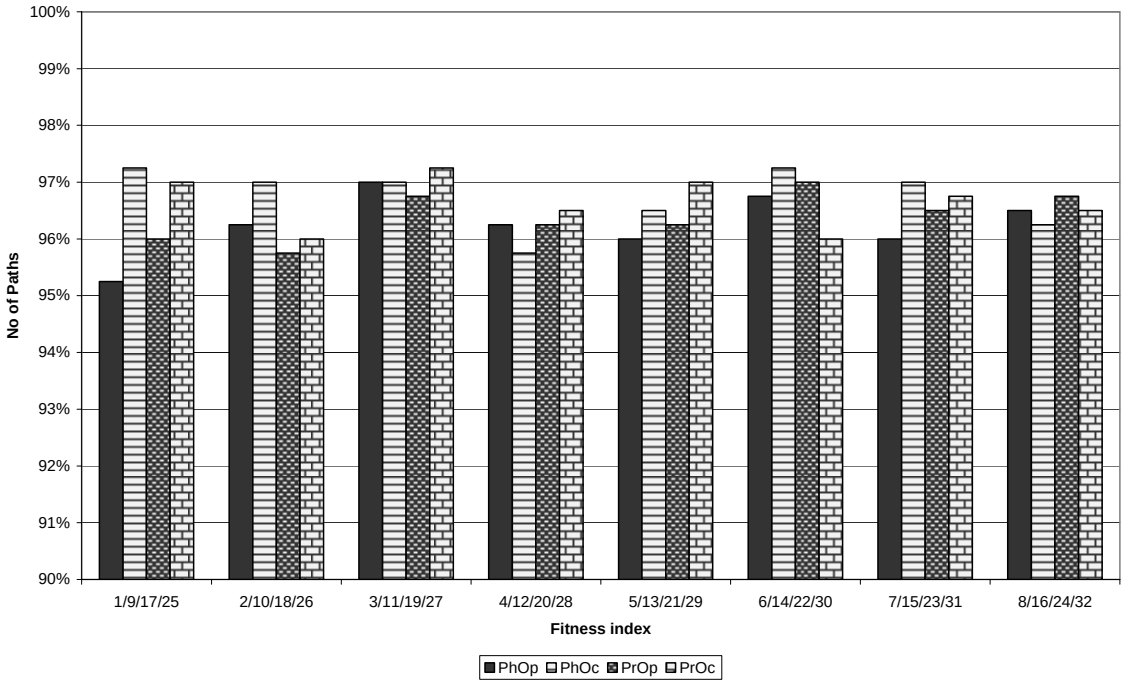

Figure 55: Composite analysis of effectiveness for *mt*

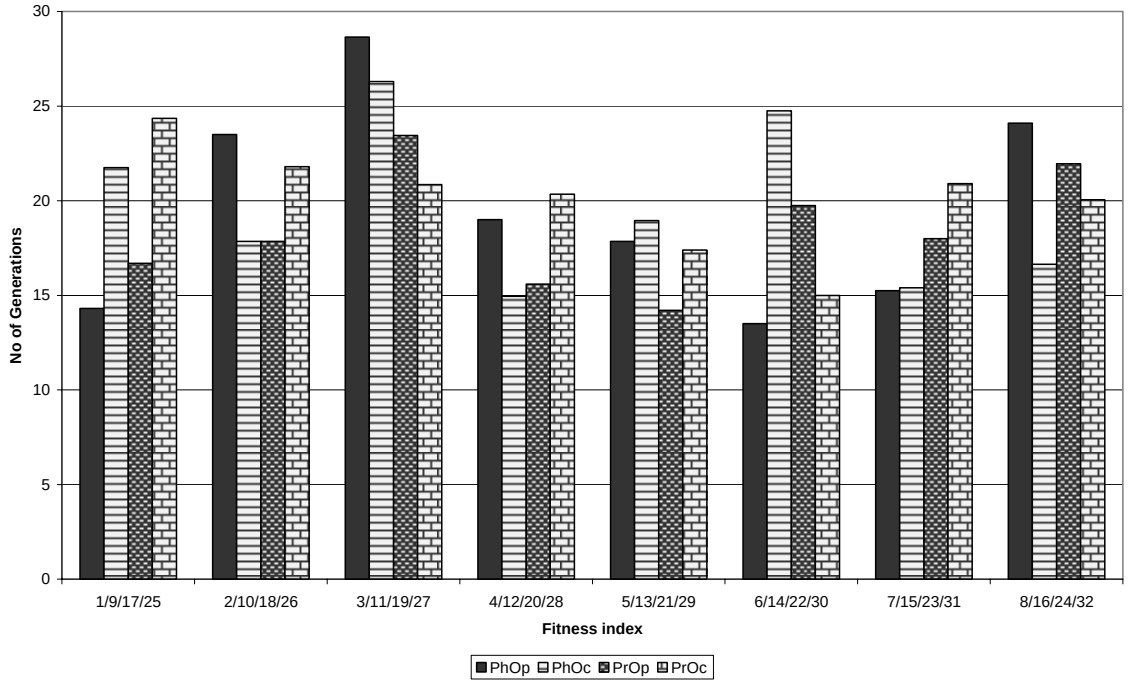

Figure 56: Composite analysis of efficiency for *mt*

5.5.9. Comparison with Other Works

In *tr* SUT (i.e., "the name of the program"), the target path that leads to equilateral triangle is the most difficult path to cover by random testing [22], since the path is covered if and only if the three input parameters are positive and equal. The probability of randomly covering this path is 2^{-30} (that is $(2^{15}*1*1)/(2^{15}*2^{15}*2^{15})$ where each positive integer is 15 bits). Thus, based on the theory of probability, it would take random testing 2^{30} test cases to reach the target. Using our test generator[6], it takes only 180 test cases (6

[6] In the set of comparisons presented in this section, we use fitness function number 30; that is considered to be the most optimal and representative candidate, i.e. in terms of effectiveness and efficiency, for comparator to these selected previous works, i.e. Pei's and Lin's works.

generations with 30 individuals, i.e. test cases, each) on the average to find the required target path, as well as other target paths (see Table 27).

Table 27: Comparison between Lin's work and ours

		Our approach		Lin's work	
No	Target Path	Found in gen (avg of 3 runs)	No of test data	Found in gen	No of test data
1	1-2-4-6-8	4	120	10	10100
2	All paths in Lin's work	4	120	10	10100

In Pei's work [36], there are 21 target paths, where 8 of them are infeasible, in testing a minimum-maximum (*mm*) program. Among the feasible target paths, the last three paths are the most difficult ones to cover as reported in his work. Thus, we use these target paths for comparison (Table 28)

Table 28: Comparison between Pei's work and ours

No	Target Path	Our approach		Pei94	
		Population	Results out of gen#	Population	Results out of gen#
1	0-1-3-5-6-1-2-5-6-7	30	3	100	15-gen (1429 runs)
				50	68-gen (3042 runs)
2	0-1-3-5-6-1-3-4-6-7	30	2	500	2-gen (1385 runs)
				50	79-gen (3512 runs)
3	0-1-3-5-6-1-3-5-6-7	30	8	1000	14-gen (14986 runs)
4	All three paths	30	7		

The results on the Table 28 show that Pei's test generator obtains the three target paths within 15, 2, and 14 generations on the average, respectively. While with our test generator, the average of three runs shows that the respective target paths were found within 3, 2, and 8 generations with smaller population sizes. Please note that since Pei's approach works on a single target path at a time. For more fairness, we conducted an experiment having one target path at a time. We have also conducted another experiment having all three target paths at one time. The results are shown in Table 28. Row no 4 of the table shows our generator was able to find all of them within 7 generations on the average.

Table 28 shows that the number of generations needed to cover a certain path depends on its level of difficulty; since Pei reported that 0-1-3-5-6-1-3-5-6-7 path is the most difficult one [36].

Table 29 depicts the detailed results of the different runs of our test generator, over 20 runs; trying to cover the three most difficult target paths in Pei's work either by a single path or all paths at a time. This table is meant to show the consistency of the results.

Table 29: The results of our work using candidate index 30 over 20 runs for minimum-maximum

Run	The last three and most difficult paths in Pei's work											
	0-1-3-5-6-1-2-5-6-7 path			0-1-3-5-6-1-3-4-6-7 path			0-1-3-5-6-1-3-5-6-7 path			All paths		
	Pop 30	Pop 50	Pop 100	Pop 30	Pop 50	Pop 100	Pop 30	Pop 50	Pop 100	Pop 30	Pop 50	Pop 100
Avg	2.85	2.05	1.35	2.25	2.20	1.45	7.55	5.25	3.60	7.30	6.25	3.55
Std	1.95	0.83	0.49	1.25	1.20	0.60	10.11	2.02	1.54	3.45	3.63	1.93
Min	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2.00	1.00	1.00	1.00	1.00
Max	8.00	3.00	2.00	5.00	5.00	3.00	47.00	9.00	7.00	17.00	14.00	8.00

On the average, the larger population sizes the smaller number of generations required to find the target path(s).

In Lin's work, the "equilateral" target path of a triangle classifier program was selected to show the ability of searching for test cases for a specific path by using genetic algorithms compared to random testing. Hence, we use the same target path to compare Lin's work with ours. Table 27 shows the comparison.

Lin's test data generator was able to cover the target path after 10 generations, with a 1000 individuals each; that is a total of10100[7] test data on average. Our generator, however, was able to cover the target path using only 120 test cases (that is four generations, with 30 individuals each), on an average of four runs. Moreover, since Lin's approach works on a single target path at a time, we conducted an experiment having one target path at a time.

Table 30, below, shows the results of our generator, based on 20 runs, in trying to cover only the "equilateral" target path. The table also shows the results when trying to satisfy all target paths at a time.

[7] Lin's generator found the required target path in the 11^{th} generation in a 100^{th} individual.

Table 30: The results of our work after 20 runs for triangle classifier

Run	Target paths					
	1-2-4-6-8 path			All paths		
	Pop 30	Pop 50	Pop 100	Pop 30	Pop 50	Pop 100
Avg	6.35	5.35	3.45	5.90	5.50	3.80
Std	4.09	1.84	1.67	2.99	2.70	1.91
Min	2.00	2.00	1.00	3.00	2.00	1.00
Max	21.00	9.00	7.00	13.00	13.00	9.00

On the average, the larger the population sizes the smaller number of generations required to find the target path(s), which also supported by smaller standard deviation.

5.5.10. Conclusion on Observations

In general, our candidate fitness functions showed to be effective and efficient in handling the required feasible target paths, regardless of the existence of infeasible paths, the path length, and the compound predicates complexity.

The existence of infeasible paths, if any, is not hindering the test data generator to find all given feasible target paths rather it is helping in exploring the search space. In this case, the candidates that employ rewarding scheme seemed to be more effective in exploring the search space.

In general, predicate-wise candidates are slightly more effective than the path-wise ones, while the path-wise candidates are more efficient than the predicate-wise ones.

On the average, the fitness functions that are utilizing neighborhood influence are more effective than otherwise, but not more efficient due to more computation time.

Generally, candidates applying rewarding scheme are better than their counterparts. Violation-based weighting is the second best after the static one.

Usually, many target paths are satisfied by individuals in the first generation. This is due to the initial set of target paths that is relatively large, combined with the exploration attained by the randomized selection of the initial population. Later on, the set of target paths becomes smaller as previously satisfied target paths are removed form the set. For example in bubble sort, almost all (2.98 out of 3) feasible target paths were found within the first 2 generations, which means that these paths are easy to find randomly.

Increasing the number of target paths, especially the infeasible ones, increases the computation time, since the complexity of the calculation of a candidate is proportional to the number of target paths (*for instance, refer to mt*).

The type of the predicate influences the search progress: composed predicates (*for instance, refer to tr* and *mt*) with the logical operator AND and predicates involving equality relational operator are harder to solve and tend to generate a higher lack of progress in the search.

Deeper predicates through the path are harder to satisfy. Longer paths have more constraints to satisfy (*for instance, refer to* the target paths for *mt*).

CHAPTER 6

CONCLUSION

6.1. Introduction

This chapter presents a summary of our major contributions in this thesis work to the software testing community. It also provides a few suggestions for future research directions.

6.2. Summary of Contributions

The thesis research has resulted in the following contributions to knowledge and tools:

1) Proposed a set of attributes for assessing and comparing GA-based test data generators.

2) Presented an extensive critical survey and evaluation (in light of the proposed attributes) of the state-of-the-art GA based test data generators.

3) Presented a GA-based test data generator that is capable of to generating multiple test data to cover multiple target paths at one run.

4) Demonstrated the capabilities of the proposed approach through empirical validation and compared a number of variations of the proposed generator. The variations of the

generator provide flexibilities in applying: traversal technique, weighting scheme, and rewarding scheme.

5) Reported promising experimental results that show that our test data generator is more effective and more efficient than existing generators; due to that fact that it allows covering multiple target paths with less number of test data generated.

6.3. Limitations and Further Works

The following are the limitations of the work:

- Manual CFG construction takes more time to do and reduces the generator scalability.
- Manual target paths generation requires tester creativity (since it is a tricky job to trap any potentially errors) and limits the scalability of generator.
- Manual program instrumentation. This process is a programming language dependent work, which must be done carefully such that it is not changing the semantic of the program. Hence, manual attempt needs extra work and time. Furthermore, it decreases the generator scalability.
- With regard to predicate-wise traversal, our fitness function does not consider the matched subpaths that have unmatched positions for a positive contribution to the fitness value. It only considers those subpaths that have the same positions.

Future Works will try to address the above limitations. Moreover, we will also try to investigate capabilities to allow automatic identification of potential infeasible program paths. Testing object oriented software will be another objective of our future research.

Considering matching subpaths that do not have matched positions will be given a high priority in our future work.

REFERENCES

[1] Alander, J.T., Mantere, T., and Turunen, P. Genetic Algorithm Based Software Testing. http://citeseer.ist.psu.edu/40769.html. 1997.

[2] Baresel, A., Sthamer, H., and Schmidt, M. Fitness Function Design to improve Evolutionary Structural Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO-2002, New York, USA, 9-13th July 2002.

[3] Beizer, B. Software Testing Techniques. Van Nostrand Reinhold, New York. 1982.

[4] Belanche, L.A. A Study in Function Optimization with the Breeder Genetic Algorithm. LSI Research Report LSI-99-36-R. Universitat Polit`ecnica de Catalunya, 1999.

[5] Berndt, D.J., Fisher, J., Johnson, L., Pinglikar, J., and Watkins, A. Breeding Software Test Cases with Genetic Algorithms. In *Proceedings of the Thirty-Sixth Hawai`i International Conference on System Sciences (HICSS-36),* Hawaii, January 2003.

[6] Berndt, D.J. and Watkins A. Investigating the Performance of Genetic Algorithm-Based Software Test Case Generation. In *Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE'04)*, pp. 261-262, University of South Florida, March 25-26, 2004.

[7] Bueno, P.M.S. and Jino, M. Identification of Potentially Infeasible Program Paths by Monitoring the Search for Test Data. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE '00)*, pg 209-218, Grenoble, France, 11-15 September 2000.

[8] Chu, H.D. An Evaluation Scheme of Software Testing Techniques. http://citeseer.ist.psu.edu/68763.html. 1996.

[9] Davis, L. Handbook of Genetic Algorithms. NY: ITP. 1991.

[10] Edvardsson, J. A Survey on Automatic Test Data Generation. In *Proceedings of the Second Conference on Computer Science and Engineering* in Linkoping, pp. 21-28. ESCEL, October 1999.

[11] Ghazi, S.A. and Ahmed, M.A. Pair-wise Test Coverage Using Genetic Algorithms. In *Proceedings of the Congress on Evolutionary Computation 2003*, Canberra, Australia, December 8-12, 2003.

[12] Goldberg, D.E. Genetic Algorithms: in Search, Optimization & Machine Learning. Addison Wesley, MA. 1989.

[13] Gupta, N., Mathur, A.P., and Soffa, M.L. Generating Test Data For Branch Coverage. In *Proceedings of the fifteenth IEEE International Conference on Automated Software*, Grenoble, France, September 11-15, 2000.

[14] Gupta, N., Mathur, A.P., and Soffa, M.L. Automated Test Data Generation Using An Iterative Relaxation Method. In *Foundations of Software Engineering*, pp. 231-244. 1998.

[15] Hamlet, D. Foundations of Software Testing: Dependability Theory. Portland State University. 1994.

[16] Harman, M., Hu, L., Hierons, R., Baresel, A., and Sthamer, H. Improving Evolutionary Testing by Flag Removal. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO-2002, New York, USA, 9-13th July 2002.

[17] Holland, J. Adaptation in Natural and Artificial Systems. MIT Press, MA. 1975.

[18] Huang, J.C. An Approach to Program Testing. *ACM Computing Surveys*, Volume 7, No. 3, September 1975.

[19] Jones, B., Sthamer, H., and Eyres, D. Automatic Structural Testing Using Genetic Algorithms. *Software Engineering Journal* 11(5), September 1996, pp. 299-306.

[20] Korel, B. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, Volume 16, No. 8, pp. 870-879, August, 1990.

[21] Korel, B. Automated Test Data Generation for Programs with Procedures. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis*. ACM Press, pp. 209-215, 1996.

[22] Lin, J.C. and Yeh, P.L. Using Genetic Algorithms for Test Case Generation in Path Testing. In *Proceedings of the 9th Asian Test Symposium* (ATS'00). Taipei, Taiwan, December 4-6, 2000.

[23] McGraw, G., Michael, C., and Schatz, M. Generating Software Test Data by Evolution. Technical report, Reliable Software Technologies, Sterling, VA. February 9, 1998.

[24] McMinn, P. Improving Evolutionary Testing in the Presence of State Behaviour. PhD Transfer Report, University of Sheffield. October 2002.

[25] Michalewicz, Z. Genetic Algorithms + Data Structures = Evolution Programs, 2nd Extended Edition. NY: Springer-Verlag. 1994.

[26] Michael, C.C., McGraw, G.E., and Schatz, M.A. Generating Software Test Data by Evolution. *IEEE Transactions on Software Engineering*, Volume 27, Number 12, pp. 1085-1110, December, 2001.

[27] Michael, C.C. and McGraw, G.G. Opportunism and Diversity in Automated Software Test Data Generation. Technical report, Reliable Software Technologies, Sterling, VA. December 8, 1997.

[28] Michael, C.C., McGraw, G.E., Schatz, M.A., and Walton, C.C. Genetic Algorithms for Dynamic Test Data Generation. Technical report, Reliable Software Technologies, Sterling, VA. May 23, 1997.

[29] Michael, C.C., McGraw, G.E., Schatz, M.A., and Walton, C.C. Genetic algorithms for dynamic test data generation. In *Proceedings of the 12th IEEE International Automated Software Engineering Conference* (ASE 97), pp. 307-308, Tahoe, NV, 1997.

[30] Michael, C.C. and McGraw, G.E. Automated Software Test Data Generation for Complex Programs. Technical report, Reliable Software Technologies, Sterling, VA. 1998.

[31] Mitchell, M. An Introduction to Genetic Algorithms. Reading, MA: MIT. 1999.

[32] Munteanu, C., Lazarescu, V., and Radoi, C. A New Strategy in Optimization using Genetic Algorithms. In *Proceedings of IEEE* Melecon '98, vol.1, pp. 415-419, 1998, ISBN 0-7803-3879-0.

[33] Myers, G.J. The Art of Software Testing. John Wiley & Sons, New York. 1979.

[34] Offutt, A.J., Jin, Z., and Pan, J. The dynamic domain reduction procedure for test data generation. *Software Practice and Experience*, Volume 29, No. 2, pp. 167-193. 1999.

[35] Pargas, R.P., Harrold, M.J., and Peck, R.R. Test-Data Generation Using Genetic Algorithms. *Journal of Software Testing, Verification and Reliability*. 1999.

[36] Pei, M., Goodman, E.D., Gao, Z., and Zhong, K. Automated Software Test Data Generation Using A Genetic Algorithm. Technical Report GARAGe of Michigan State University, June 1994.

[37] Roper, M., Maclean, I., Brooks, A., Miller, J., and Wood, M. Genetic Algorithms and the Automatic Generation of Test Data. http://citeseer.ist.psu.edu/135258.html. 1995.

[38] Sommerville, I. Software Engineering. 6th Ed. Addison-Wesley, USA. 2001.

[39] Sthamer, H.H., Wegener, J., and Baresel, A. Using Evolutionary Testing to improve Efficiency and Quality in Software Testing. In *Proceedings of the second Asia-Pacific Conference on Software Testing Analysis & Review*, Melbourne, Australia, 22-24th July 2002.

[40] Sthamer, H.H. The Automatic Generation of Software Test Data Using Genetic Algorithms. PhD Dissertation, University of Glamorgan. November, 1995.

[41] Tracey, N.J., Clark, J., Mander, K., and McDermid, J. An Automated Framework for Structural Test-Data Generation. In *Proceedings 13th IEEE Conference in Automated Software Engineering*, Hawaii, October 1998.

[42] Tracey, N.J., Clark, J., and Mander, K. The Way Forward for Unifying Dynamic Test Case Generation: The Optimisation-Based Approach. In *IFIP International Workshop on Dependable Computing and its Applications (DCIA 98)*, Johannesburg, January 1998.

[43] Wegener, J., Baresel, A., and Sthamer, H. Suitability of Evolutionary Algorithms for Evolutionary Testing. In *Proceedings of the 26th Annual International Computer Software and Applications Conference*, Oxford, England, August 26-29, 2002.

[44] Wegener, J., Buhr, K., and Pohlheim, H. Automatic Test Data Generation for Structural Testing of Embedded Software Systems by Evolutionary Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO-2002, New York, USA, 9-13th July 2002.

[45] Whittaker, J.A. What Is Software Testing? And Why Is It So Hard? *IEEE Software*. February, 2000.

[46] Weinberg, G.M. The Psychology of Computer Programming: Silver Anniversary Edition. Dorset House Publishing Co., New York, USA. 1998.

[47] Zhu, H., Hall, P., and May, J. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29 (4): 366 – 427, December 1997.

APPENDIX A

SOFTWARE UNDER TESTS (SUTs)

A.1. Minimum-maximum (minimaxi.m)

```
function [traversedPath, miniMaxi] = minimaxi(num)

numLength = length(num);
mini = num(1);
maxi = num(1);
idx = 2;
traversedPath = []; % traversedPath contains branch# and its corresponding branchVal.

traversedPath = [traversedPath 1 fitnessMiniMaxi(1, [idx numLength])]; % instrument
while (idx <= numLength) % Branching #1

   traversedPath = [traversedPath 2 fitnessMiniMaxi(2, [maxi num(idx)])]; % instrument
   if maxi < num(idx) % Branching #2
      maxi = num(idx);
   end

   traversedPath = [traversedPath 3 fitnessMiniMaxi(3, [mini num(idx)])]; % instrument
   if mini > num(idx) % Branching #3
      mini = num(idx);
   end

   idx = idx+1;
   traversedPath = [traversedPath 1 fitnessMiniMaxi(1, [idx numLength])]; % instrument
end % while end

miniMaxi = [mini maxi];
```

Figure 57: Source code of minimum-maximum

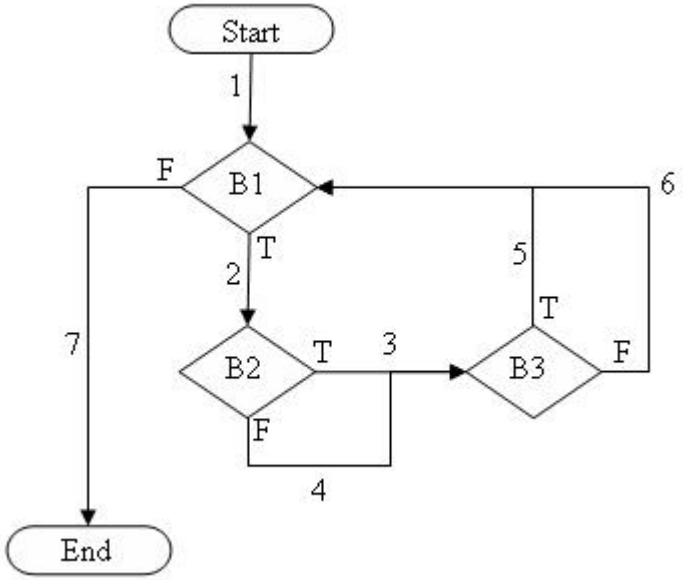

Figure 58: CFG of minimum-maximum

```
PS = {
   % 1-7
   [1 1];
   % 1-2-3-5-7
   [1 0 2 0 3 0 1 1]; % Infeasible path
   % 1-2-3-6-7
   [1 0 2 0 3 1 1 1];
   % 1-2-4-5-7
   [1 0 2 1 3 0 1 1];
   % 1-2-4-6-7
   [1 0 2 1 3 1 1 1];
   % 1-2-3-5-2-3-5-7
   [1 0 2 0 3 0 1 0 2 0 3 0 1 1]; % Infeasible path
   % 1-2-3-5-2-3-6-7
   [1 0 2 0 3 0 1 0 2 0 3 1 1 1]; % Infeasible path
   % 1-2-3-5-2-4-5-7
   [1 0 2 0 3 0 1 0 2 1 3 0 1 1]; % Infeasible path
   % 1-2-3-5-2-4-6-7
   [1 0 2 0 3 0 1 0 2 1 3 1 1 1]; % Infeasible path
   % 1-2-3-6-2-3-5-7
   [1 0 2 0 3 1 1 0 2 0 3 0 1 1]; % Infeasible path
   % 1-2-3-6-2-3-6-7
   [1 0 2 0 3 1 1 0 2 0 3 1 1 1];
   % 1-2-3-6-2-4-5-7
   [1 0 2 0 3 1 1 0 2 1 3 0 1 1];
   % 1-2-3-6-2-4-6-7
   [1 0 2 0 3 1 1 0 2 1 3 1 1 1];
   % 1-2-4-5-2-3-5-7
   [1 0 2 1 3 0 1 0 2 0 3 0 1 1]; % Infeasible path
   % 0-1-3-4-6-1-2-5-6-7
   [1 0 2 1 3 0 1 0 2 0 3 1 1 1];
   % 0-1-3-4-6-1-3-4-6-7
   [1 0 2 1 3 0 1 0 2 1 3 0 1 1];
   % 0-1-3-4-6-1-3-5-6-7
   [1 0 2 1 3 0 1 0 2 1 3 1 1 1];
   % 0-1-3-5-6-1-2-4-6-7
   [1 0 2 1 3 1 1 0 2 0 3 0 1 1]; % Infeasible path
   % 0-1-3-5-6-1-2-5-6-7
   [1 0 2 1 3 1 1 0 2 0 3 1 1 1];
   % 0-1-3-5-6-1-3-4-6-7
   [1 0 2 1 3 1 1 0 2 1 3 0 1 1];
   % 0-1-3-5-6-1-3-5-6-7
   [1 0 2 1 3 1 1 0 2 1 3 1 1 1]
};
```

Figure 59: Selected target paths of minimum-maximum

A.2. Triangle Classifier (triangle.m)

```
function [traversedPath, type] = triangle(sideLengths)

traversedPath = [];
A = sideLengths(1); % First side
B = sideLengths(2); % Second side
C = sideLengths(3); % Third side

traversedPath = [traversedPath 1 fitnessTriangle(1, A, B, C)]; % instrument Branch # 1
if ((A+B > C) & (B+C > A) & (C+A > B)) % Branch # 1
   traversedPath = [traversedPath 2 fitnessTriangle(2, A, B, C)]; % instrument Branch # 2
   if ((A ~= B) & (B ~= C) & (C ~= A)) % Branch # 2
      type = 'Scalene';
   else
      traversedPath = [traversedPath 3 fitnessTriangle(3, A, B, C)]; % instrument Branch # 3
      if (((A == B) & (B ~= C)) | ((B == C) & (C ~= A)) | ((C == A) & (A ~= B))) % Branch #
3
         type = 'Isosceles';
      else
         type = 'Equilateral';
      end
   end
else
   type = 'Not a triangle';
end
```

Figure 60: Source code of triangle classifier

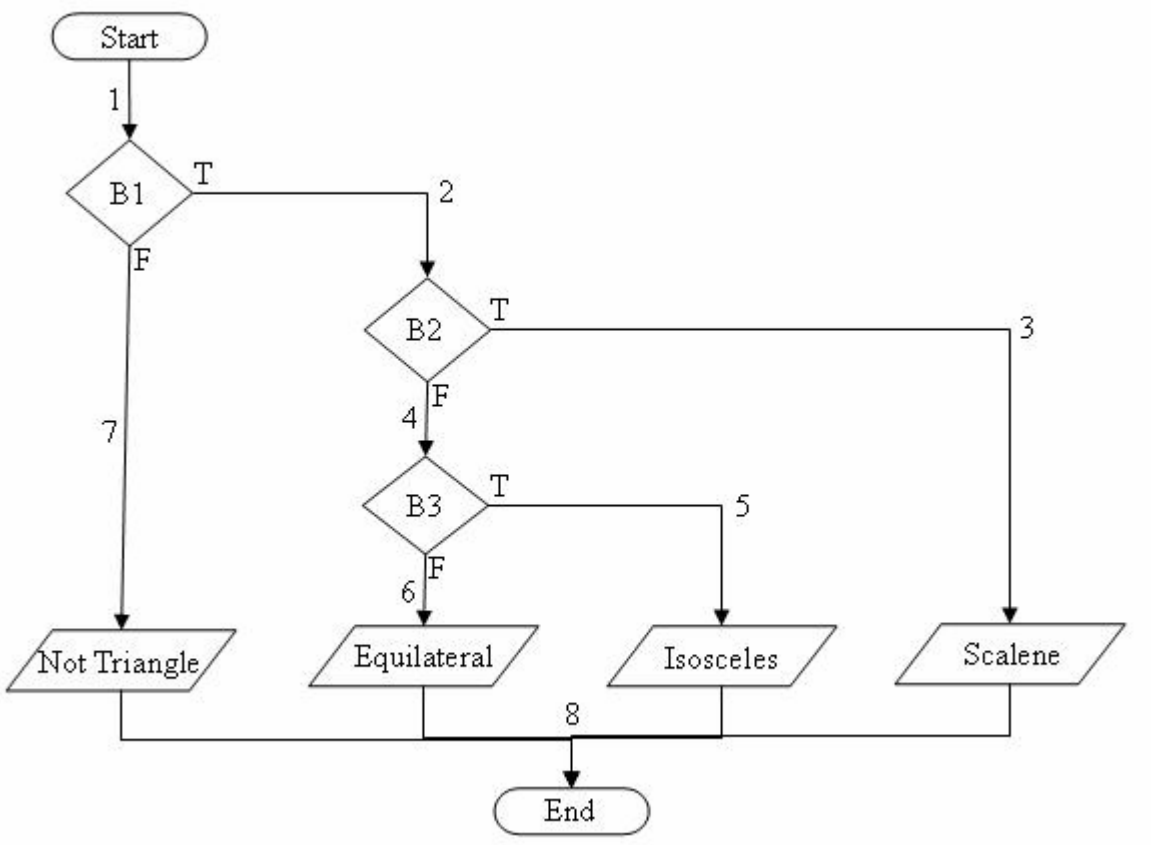

Figure 61: CFG of triangle classifier

```
PS = {
    % 1-7-8
    [1 1];
    % 1-2-3-8
    [1 0 2 0];
    % 1-2-4-6-8
    [1 0 2 1 3 1];
    % 1-2-4-5-8
    [1 0 2 1 3 0]
 };
```

Figure 62: Selected target paths of triangle classifier

A.3. Bubble Sort (bubble.m)

```
function [traversedPath, sortedArray] = bubble(anyArray)
%function sortedArray = bubble(anyArray)

sorted = 0; % 0 means false
i = 1; n = length(anyArray);
traversedPath = [];

traversedPath = [traversedPath 1 fitnessBubble(1, [i (n-1) ~sorted])]; % instrument Branch # 1
while ((i <= (n-1)) & ~sorted), % Branch # 1
   sorted = 1;

   j = n;
   traversedPath = [traversedPath 2 fitnessBubble(2, [j (i+1)])]; % instrument Branch # 2
   for j=n:-1:i+1 % Branch # 2

      traversedPath = [traversedPath 3 fitnessBubble(3, [anyArray(j) anyArray(j-1)])]; % instrument Branch # 3
      if (anyArray(j) < anyArray(j-1)) % Branch # 3
         %exchange(anyArray(j), anyArray(j-1));
         temp = anyArray(j);
         anyArray(j) = anyArray(j-1);
         anyArray(j-1) = temp;
         sorted = 0;
      end

      traversedPath = [traversedPath 2 fitnessBubble(2, [(j-1) (i+1)])]; % instrument Branch # 2
   end

   i = i + 1;
   traversedPath = [traversedPath 1 fitnessBubble(1, [i (n-1) ~sorted])]; % instrument Branch # 1
end
sortedArray = anyArray;
```

Figure 63: Source code of bubble sort

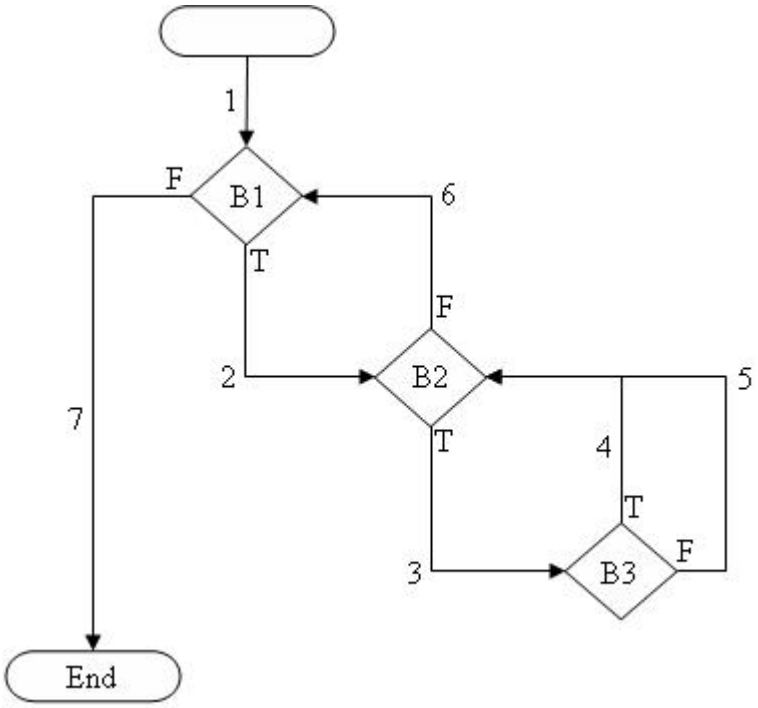

Figure 64: CFG of bubble sort

```
PS = { ...
% 1-7
  [1 1];
% 1-2-6-7
  [1 0 2 1 1 1];

% Target paths to repeat their sub-paths
% 1-2-3-4-6-7
  [1 0 2 0 3 0 2 1 1 1];
% 1-2-3-5-6-7
  [1 0 2 0 3 1 2 1 1 1];
% 1-2-3-4-3-5-6-7
  [1 0 2 0 3 0 2 0 3 1 2 1 1 1];
% 1-2-3-5-3-4-6-7
  [1 0 2 0 3 1 2 0 3 0 2 1 1 1];
% 1-2-3-4-3-4-6-7
  [1 0 2 0 3 0 2 0 3 0 2 1 1 1];
% 1-2-3-5-3-5-6-7
  [1 0 2 0 3 1 2 0 3 1 2 1 1 1];
% 1-2-3-4-3-5-3-5-6-7
  [1 0 2 0 3 0 2 0 3 1 2 0 3 1 2 1 1 1];
% 1-2-3-5-3-5-3-4-6-7
  [1 0 2 0 3 1 2 0 3 1 2 0 3 0 2 1 1 1];
% 1-2-3-4-3-4-3-5-6-7
  [1 0 2 0 3 0 2 0 3 0 2 0 3 1 2 1 1 1];
% 1-2-3-5-3-5-3-4-6-7
  [1 0 2 0 3 1 2 0 3 1 2 0 3 0 2 1 1 1];
% 1-2-3-4-3-5-3-4-6-7
  [1 0 2 0 3 0 2 0 3 1 2 0 3 0 2 1 1 1];
% 1-2-3-5-3-4-3-5-6-7
  [1 0 2 0 3 1 2 0 3 0 2 0 3 1 2 1 1 1];
};
```

Figure 65: Selected target paths of bubble sort

A.4. Insertion Sort (insertion.m)

```
function [traversedPath, sortedArray] = insertion(anyArray)
%function sortedArray = insertion(anyArray)

k = 1; % The smallest integer increment
traversedPath = [];
n = length(anyArray);

i = 2;
traversedPath = [traversedPath 1 fitnessInsertion(1, [i n])]; % instrument Branch # 1
for i=2:n % Branch # 1
   x = anyArray(i);
   j = i - 1;

   traversedPath = [traversedPath 2 fitnessInsertion(2, [j anyArray(j) x])]; % instrument Branch
# 2
   while ((j > 0) & (anyArray(j) > x)), % Branch # 2
      anyArray(j+1) = anyArray(j);
      j = j - 1;

      if (j > 0), % Added for instrumentation purpose only
         traversedPath = [traversedPath 2 fitnessInsertion(2, [j anyArray(j) x])]; % instrument
Branch # 2
      else
         traversedPath = [traversedPath 2 k]; % anyArray(j) is undefined, because j=0.
      end
   end
   anyArray(j+1) = x;

   traversedPath = [traversedPath 1 fitnessInsertion(1, [(i+1) n])]; % instrument Branch # 1
end
sortedArray = anyArray;
```

Figure 66: Source code of insertion sort

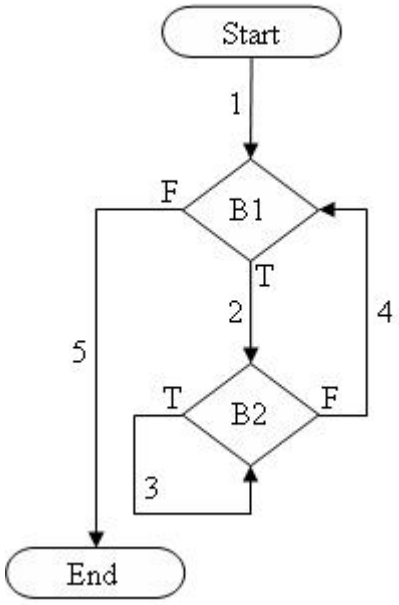

Figure 67: CFG of insertion sort

```
PS = {
   % 1 5
     [1 1];
   % 1 2 4 5
     [1 0 2 1 1 1];
   % 1 2 3 4 5
     [1 0 2 0 2 1 1 1];
   % 1 2 3 3 4 5
     [1 0 2 0 2 0 2 1 1 1];
   % 1 2 4 2 3 4 5
     [1 0 2 1 1 0 2 0 2 1 1 1];
   % 1 2 3 4 2 4 5
     [1 0 2 0 2 1 1 0 2 1 1 1];
  };
```

Figure 68: Selected target paths of insertion sort

A.5. Binary Search (binary.m)

```
function [traversedPath, itemIndex] = binary(itemNumbers)
%function itemIndex = binary(item, numbers)

item = itemNumbers(1);
numbers = itemNumbers(1,2:end);

lowerIdx = 1;
upperIdx = length(numbers);
traversedPath = [];

traversedPath = [traversedPath 1 fitnessBinary(1, [lowerIdx upperIdx])]; % instrument Branch
# 1
while (lowerIdx ~= upperIdx), % Branch # 1

   temp = lowerIdx + upperIdx; % additional statement
   if (mod(temp, 2) ~= 0), temp = temp - 1; end % additional statement
   idx = temp / 2;

   traversedPath = [traversedPath 2 fitnessBinary(2, [numbers(idx) item])]; % instrument
Branch # 2
   if (numbers(idx) < item), % Branch # 2
      lowerIdx = idx + 1;
   else
      upperIdx = idx;
   end

   traversedPath = [traversedPath 1 fitnessBinary(1, [lowerIdx upperIdx])]; % instrument
Branch # 1
end

if (item == numbers(lowerIdx)), % Additional code that returns -1 if the item is not found
   itemIndex = lowerIdx;
else
   itemIndex = -1;
end
```

Figure 69: Source code of binary search

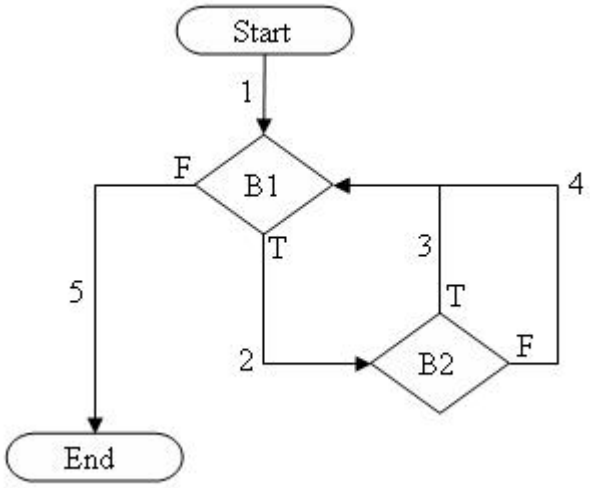

Figure 70: CFG of binary search

```
PS = {
   % 1 5
   [1 1];
   % 1 2 3 5
   [1 0 2 0 1 1];
   % 1 2 4 5
   [1 0 2 1 1 1];
   % 1 2 3 2 3 5
   [1 0 2 0 1 0 2 0 1 1];
   % 1 2 4 2 4 5
   [1 0 2 1 1 0 2 1 1 1];
   % 1 2 3 2 4 5
   [1 0 2 0 1 0 2 1 1 1];
   % 1 2 4 2 3 5
   [1 0 2 1 1 0 2 0 1 1];
  };
```

Figure 71: Selected target paths of binary search

A.6. Minimum-Maximum and Triangle Classifier (mmTriangle.m)

```
function [traversedPath, minimaxi, type] = program6(num)

numLength = length(num);
mini = num(1);
maxi = num(1);
idx = 2;
traversedPath = []; % traversedPath contains branch# and its corresponding branchVal.

traversedPath = [traversedPath 1 fitnessMiniMaxi(1, [idx numLength])]; % instrument
while (idx <= numLength) % Branching #1

   traversedPath = [traversedPath 2 fitnessMiniMaxi(2, [maxi num(idx)])]; % instrument
   if maxi < num(idx) % Branching #2
      maxi = num(idx);
   end

   traversedPath = [traversedPath 3 fitnessMiniMaxi(3, [mini num(idx)])]; % instrument
   if mini > num(idx) % Branching #3
      mini = num(idx);
   end

   idx = idx+1;
   traversedPath = [traversedPath 1 fitnessMiniMaxi(1, [idx numLength])]; % instrument
end % while end

minimaxi = [mini maxi];
A = num(1); % First side
B = num(2); % Second side
C = num(3); % Third side

traversedPath = [traversedPath 4 fitnessTriangle(1, A, B, C)]; % instrument Branch # 4
if ((A+B > C) & (B+C > A) & (C+A > B)) % Branch # 4

   traversedPath = [traversedPath 5 fitnessTriangle(2, A, B, C)]; % instrument Branch # 5
   if ((A ~= B) & (B ~= C) & (C ~= A)) % Branch # 5
      type = 'Scalene';
   else

      traversedPath = [traversedPath 6 fitnessTriangle(3, A, B, C)]; % instrument Branch # 6
      if (((A == B) & (B ~= C)) | ((B == C) & (C ~= A)) | ((C == A) & (A ~= B))) % Branch #
6
         type = 'Isosceles';
      else
         type = 'Equilateral';
```

```
        end
    end
else
    type = 'Not a triangle';
end
```

Figure 72: Source code of mmTriangle

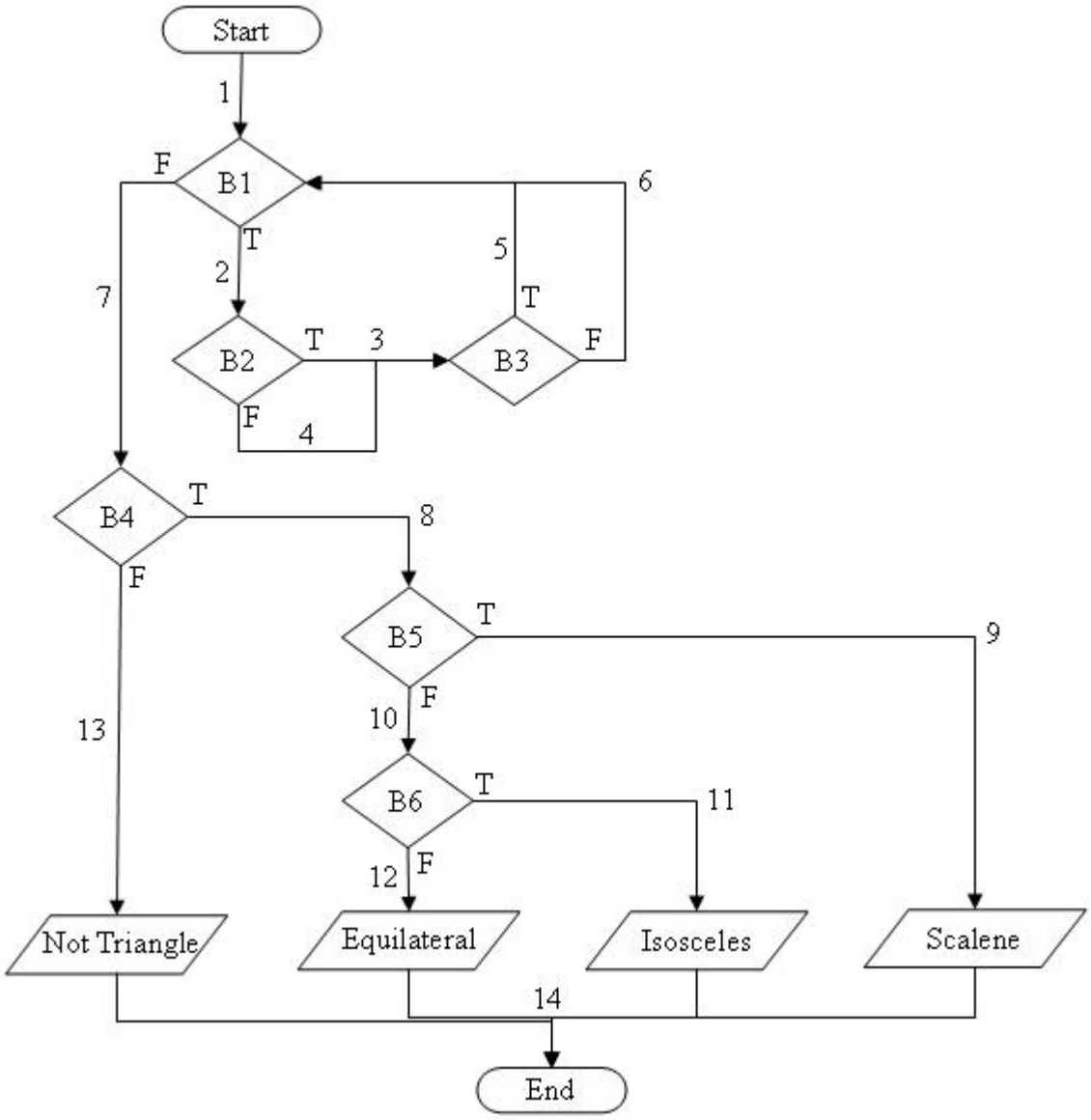

Figure 73: CFG of mmTriangle

```
PS = {
   % First combination: Tail => Equilateral
   % 0-7
   [1 1 4 0 5 1 6 1];
   % 0-1-2-4-6-7
   [1 0 2 0 3 0 1 1 4 0 5 1 6 1]; % Infeasible path
   % 0-1-2-5-6-7
   [1 0 2 0 3 1 1 1 4 0 5 1 6 1];
   % 0-1-3-4-6-7
   [1 0 2 1 3 0 1 1 4 0 5 1 6 1];
   % 0-1-3-5-6-7
   [1 0 2 1 3 1 1 1 4 0 5 1 6 1];
   % 0-1-2-4-6-1-2-4-6-7
   [1 0 2 0 3 0 1 0 2 0 3 0 1 1 4 0 5 1 6 1]; % Infeasible path
   % 0-1-2-4-6-1-2-5-6-7
   [1 0 2 0 3 0 1 0 2 0 3 1 1 1 4 0 5 1 6 1]; % Infeasible path
   % 0-1-2-4-6-1-3-4-6-7
   [1 0 2 0 3 0 1 0 2 1 3 0 1 1 4 0 5 1 6 1]; % Infeasible path
   % 0-1-2-4-6-1-3-5-6-7
   [1 0 2 0 3 0 1 0 2 1 3 1 1 1 4 0 5 1 6 1]; % Infeasible path
   % 0-1-2-5-6-1-2-4-6-7
   [1 0 2 0 3 1 1 0 2 0 3 0 1 1 4 0 5 1 6 1]; % Infeasible path
   % 0-1-2-5-6-1-2-5-6-7
   [1 0 2 0 3 1 1 0 2 0 3 1 1 1 4 0 5 1 6 1];
   % 0-1-2-5-6-1-3-4-6-7
   [1 0 2 0 3 1 1 0 2 1 3 0 1 1 4 0 5 1 6 1];
   % 0-1-2-5-6-1-3-5-6-7
   [1 0 2 0 3 1 1 0 2 1 3 1 1 1 4 0 5 1 6 1];
   % 0-1-3-4-6-1-2-4-6-7
   [1 0 2 1 3 0 1 0 2 0 3 0 1 1 4 0 5 1 6 1]; % Infeasible path
   % 0-1-3-4-6-1-2-5-6-7
   [1 0 2 1 3 0 1 0 2 0 3 1 1 1 4 0 5 1 6 1];
   % 0-1-3-4-6-1-3-4-6-7
   [1 0 2 1 3 0 1 0 2 1 3 0 1 1 4 0 5 1 6 1];
   % 0-1-3-4-6-1-3-5-6-7
   [1 0 2 1 3 0 1 0 2 1 3 1 1 1 4 0 5 1 6 1];
   % 0-1-3-5-6-1-2-4-6-7
   [1 0 2 1 3 1 1 0 2 0 3 0 1 1 4 0 5 1 6 1]; % Infeasible path
   % 0-1-3-5-6-1-2-5-6-7
   [1 0 2 1 3 1 1 0 2 0 3 1 1 1 4 0 5 1 6 1];
   % 0-1-3-5-6-1-3-4-6-7
   [1 0 2 1 3 1 1 0 2 1 3 0 1 1 4 0 5 1 6 1];
   % 0-1-3-5-6-1-3-5-6-7
   [1 0 2 1 3 1 1 0 2 1 3 1 1 1 4 0 5 1 6 1];

   % Second combination: Tail => Scalene
   % 0-7
```

```
[1 1 4 0 5 0];
% 0-1-2-4-6-7
[1 0 2 0 3 0 1 1 4 0 5 0]; % Infeasible path
% 0-1-2-5-6-7
[1 0 2 0 3 1 1 1 4 0 5 0];
% 0-1-3-4-6-7
[1 0 2 1 3 0 1 1 4 0 5 0];
% 0-1-3-5-6-7
[1 0 2 1 3 1 1 1 4 0 5 0];
% 0-1-2-4-6-1-2-4-6-7
[1 0 2 0 3 0 1 0 2 0 3 0 1 1 4 0 5 0]; % Infeasible path
% 0-1-2-4-6-1-2-5-6-7
[1 0 2 0 3 0 1 0 2 0 3 1 1 1 4 0 5 0]; % Infeasible path
% 0-1-2-4-6-1-3-4-6-7
[1 0 2 0 3 0 1 0 2 1 3 0 1 1 4 0 5 0]; % Infeasible path
% 0-1-2-4-6-1-3-5-6-7
[1 0 2 0 3 0 1 0 2 1 3 1 1 1 4 0 5 0]; % Infeasible path
% 0-1-2-5-6-1-2-4-6-7
[1 0 2 0 3 1 1 0 2 0 3 0 1 1 4 0 5 0]; % Infeasible path
% 0-1-2-5-6-1-2-5-6-7
[1 0 2 0 3 1 1 0 2 0 3 1 1 1 4 0 5 0];
% 0-1-2-5-6-1-3-4-6-7
[1 0 2 0 3 1 1 0 2 1 3 0 1 1 4 0 5 0];
% 0-1-2-5-6-1-3-5-6-7
[1 0 2 0 3 1 1 0 2 1 3 1 1 1 4 0 5 0];
% 0-1-3-4-6-1-2-4-6-7
[1 0 2 1 3 0 1 0 2 0 3 0 1 1 4 0 5 0]; % Infeasible path
% 0-1-3-4-6-1-2-5-6-7
[1 0 2 1 3 0 1 0 2 0 3 1 1 1 4 0 5 0];
% 0-1-3-4-6-1-3-4-6-7
[1 0 2 1 3 0 1 0 2 1 3 0 1 1 4 0 5 0];
% 0-1-3-4-6-1-3-5-6-7
[1 0 2 1 3 0 1 0 2 1 3 1 1 1 4 0 5 0];
% 0-1-3-5-6-1-2-4-6-7
[1 0 2 1 3 1 1 0 2 0 3 0 1 1 4 0 5 0]; % Infeasible path
% 0-1-3-5-6-1-2-5-6-7
[1 0 2 1 3 1 1 0 2 0 3 1 1 1 4 0 5 0];
% 0-1-3-5-6-1-3-4-6-7
[1 0 2 1 3 1 1 0 2 1 3 0 1 1 4 0 5 0];
% 0-1-3-5-6-1-3-5-6-7
[1 0 2 1 3 1 1 0 2 1 3 1 1 1 4 0 5 0];

% Third combination: Tail => Not Triangle
% 0-7
[1 1 4 1];
% 0-1-2-4-6-7
[1 0 2 0 3 0 1 1 4 1]; % Infeasible path
```

```
% 0-1-2-5-6-7
[1 0 2 0 3 1 1 1 4 1];
% 0-1-3-4-6-7
[1 0 2 1 3 0 1 1 4 1];
% 0-1-3-5-6-7
[1 0 2 1 3 1 1 1 4 1];
% 0-1-2-4-6-1-2-4-6-7
[1 0 2 0 3 0 1 0 2 0 3 0 1 1 4 1]; % Infeasible path
% 0-1-2-4-6-1-2-5-6-7
[1 0 2 0 3 0 1 0 2 0 3 1 1 1 4 1]; % Infeasible path
% 0-1-2-4-6-1-3-4-6-7
[1 0 2 0 3 0 1 0 2 1 3 0 1 1 4 1]; % Infeasible path
% 0-1-2-4-6-1-3-5-6-7
[1 0 2 0 3 0 1 0 2 1 3 1 1 1 4 1]; % Infeasible path
% 0-1-2-5-6-1-2-4-6-7
[1 0 2 0 3 1 1 0 2 0 3 0 1 1 4 1]; % Infeasible path
% 0-1-2-5-6-1-2-5-6-7
[1 0 2 0 3 1 1 0 2 0 3 1 1 1 4 1];
% 0-1-2-5-6-1-3-4-6-7
[1 0 2 0 3 1 1 0 2 1 3 0 1 1 4 1];
% 0-1-2-5-6-1-3-5-6-7
[1 0 2 0 3 1 1 0 2 1 3 1 1 1 4 1];
% 0-1-3-4-6-1-2-4-6-7
[1 0 2 1 3 0 1 0 2 0 3 0 1 1 4 1]; % Infeasible path
% 0-1-3-4-6-1-2-5-6-7
[1 0 2 1 3 0 1 0 2 0 3 1 1 1 4 1];
% 0-1-3-4-6-1-3-4-6-7
[1 0 2 1 3 0 1 0 2 1 3 0 1 1 4 1];
% 0-1-3-4-6-1-3-5-6-7
[1 0 2 1 3 0 1 0 2 1 3 1 1 1 4 1];
% 0-1-3-5-6-1-2-4-6-7
[1 0 2 1 3 1 1 0 2 0 3 0 1 1 4 1]; % Infeasible path
% 0-1-3-5-6-1-2-5-6-7
[1 0 2 1 3 1 1 0 2 0 3 1 1 1 4 1];
% 0-1-3-5-6-1-3-4-6-7
[1 0 2 1 3 1 1 0 2 1 3 0 1 1 4 1];
% 0-1-3-5-6-1-3-5-6-7
[1 0 2 1 3 1 1 0 2 1 3 1 1 1 4 1];

% Forth combination: Tail => Isosceles
% 0-7
[1 1 4 0 5 1 6 0];
% 0-1-2-4-6-7
[1 0 2 0 3 0 1 1 4 0 5 1 6 0]; % Infeasible path
% 0-1-2-5-6-7
[1 0 2 0 3 1 1 1 4 0 5 1 6 0];
% 0-1-3-4-6-7
```

```
    [1 0 2 1 3 0 1 1 4 0 5 1 6 0];
    % 0-1-3-5-6-7
    [1 0 2 1 3 1 1 1 4 0 5 1 6 0];
    % 0-1-2-4-6-1-2-4-6-7
    [1 0 2 0 3 0 1 0 2 0 3 0 1 1 4 0 5 1 6 0]; % Infeasible path
    % 0-1-2-4-6-1-2-5-6-7
    [1 0 2 0 3 0 1 0 2 0 3 1 1 1 4 0 5 1 6 0]; % Infeasible path
    % 0-1-2-4-6-1-3-4-6-7
    [1 0 2 0 3 0 1 0 2 1 3 0 1 1 4 0 5 1 6 0]; % Infeasible path
    % 0-1-2-4-6-1-3-5-6-7
    [1 0 2 0 3 0 1 0 2 1 3 1 1 1 4 0 5 1 6 0]; % Infeasible path
    % 0-1-2-5-6-1-2-4-6-7
    [1 0 2 0 3 1 1 0 2 0 3 0 1 1 4 0 5 1 6 0]; % Infeasible path
    % 0-1-2-5-6-1-2-5-6-7
    [1 0 2 0 3 1 1 0 2 0 3 1 1 1 4 0 5 1 6 0];
    % 0-1-2-5-6-1-3-4-6-7
    [1 0 2 0 3 1 1 0 2 1 3 0 1 1 4 0 5 1 6 0];
    % 0-1-2-5-6-1-3-5-6-7
    [1 0 2 0 3 1 1 0 2 1 3 1 1 1 4 0 5 1 6 0];
    % 0-1-3-4-6-1-2-4-6-7
    [1 0 2 1 3 0 1 0 2 0 3 0 1 1 4 0 5 1 6 0]; % Infeasible path
    % 0-1-3-4-6-1-2-5-6-7
    [1 0 2 1 3 0 1 0 2 0 3 1 1 1 4 0 5 1 6 0];
    % 0-1-3-4-6-1-3-4-6-7
    [1 0 2 1 3 0 1 0 2 1 3 0 1 1 4 0 5 1 6 0];
    % 0-1-3-4-6-1-3-5-6-7
    [1 0 2 1 3 0 1 0 2 1 3 1 1 1 4 0 5 1 6 0];
    % 0-1-3-5-6-1-2-4-6-7
    [1 0 2 1 3 1 1 0 2 0 3 0 1 1 4 0 5 1 6 0]; % Infeasible path
    % 0-1-3-5-6-1-2-5-6-7
    [1 0 2 1 3 1 1 0 2 0 3 1 1 1 4 0 5 1 6 0];
    % 0-1-3-5-6-1-3-4-6-7
    [1 0 2 1 3 1 1 0 2 1 3 0 1 1 4 0 5 1 6 0];
    % 0-1-3-5-6-1-3-5-6-7
    [1 0 2 1 3 1 1 0 2 1 3 1 1 1 4 0 5 1 6 0];
};
```

Figure 74: Selected target paths of mmTriangle

APPENDIX B

CONTROL LOGIC GRAPHS (CLGs)

B.1. Control Flow Graph (CFG)

A control flow graph of program *P* is a directed graph *G* = (*N, A, s, e*) where: *N* is set of nodes, *A* is binary relation on *N* (a subset of *N* x *N* that referred to as a set of edges), *s* and *e* are, respectively, unique entry and unique exit node ($s, e \in N$). A node in *N* corresponds to the smallest single entry, single-exist executable part of a statement in *P* that can not be further decomposed; such a part is referred to as an instruction. A single instruction corresponds to an assignment statement, an input or output statement, or the <expression> part of a selection statement, e.g. **IF-THEN-ELSE**, or a looping statement, e.g. **WHILE**, in which case it is called a test instruction. An edge $(n_i, n_j) \in A$ corresponds to a possible transfer of control from instruction n_i to n_j. An edge (n_i, n_j) is called a branch if n_i is a test instruction, e.g. selection. Each branch in the CFG can be labeled by a predicate, referred to as a branch predicate, describing a condition under which the branch will be traversed.

In a reduced CFG (hereafter CFG) of program, the edges of sequencing nodes are merged as a short sub-path and different branches, which include in selection or looping statement is taken as an independent sub-path separately. Each sub-path in the CFG can be labeled by certain number. In fact, a path in a CFG is a sequence of this kind of sub-path and the path is identified by the sequence of these labeled numbers.

An input variable of a program *P* is a variable that appears in an input statement or it is in an input parameter of a function or procedure. Input variable may be of different types, e.g. integer, real, or Boolean. Let $I = (x_1, x_2, ..., x_n)$ be a vector of input variables of

program P. The domain D_{ri} of input variable x_i is a set of all values which x_i can hold. The domain D of a program means a cross product, i.e. $D = D_{r1} \times D_{r2} \times ... \times D_{rn}$. A single point x in the n-dimensional input space D, $x \in D$, is referred to as a program input.

A path P_k in a CFG is a sequence $P_k = [n_{k0}, n_{k1}, ..., n_{kq}]$ of instructions, such that $n_{k0} = s$, $n_{kq} = e$, and for all i, $0 \leq i < q$, $(nki, nki+1) \in A$. Suppose P_i is a path through a program P. Then the path domain $D_i = D(P_i)$ for P_i is the subset of the input domain which causes P_i to be executed. The path computation $C_i = C(P_i)$ for Pi is the function which is computed by the sequence of computations in P_i. A path is feasible if there exists a program input x for which the path is traversed during the program execution, otherwise the path is infeasible.

B.2. Control Dependence Graph (CDG)

Control dependence for a program is defined in terms of the program's CFG and the post-dominance relation that exists among the nodes in the CFG. Given such a CFG, and nodes W and V in that graph, W is post-dominated by V if every directed path from W to the exit (not including W or exit) contains V. For statements (nodes) X and Y in a CFG, Y is control dependent on X if and only if (1) there exists a directed path P from X to Y with all Z in P (excluding X and Y) post-dominated by Y and (2) X is not post-dominated by Y. In a CDG, nodes represent statements, and edges represent the control dependencies between statements – an edge (X, Y) in a CDG means that Y is control dependent on X.

An acyclic path in the CDG from the root of the graph to a node in the graph contains a set of predicates that must be satisfied by an input that causes the statement associated with the node to be executed; such a path is called a control-dependence predicate path. Unstructured transfers of control, e.g. **GOTO**, **CONTINUE**, or **BREAK**, can cause the occurrence of more than one control-dependence predicate path for statements following the transfers. However, the number of control-dependence predicate paths is generally small.

VITA

Irman Hermadi, who was born on 11 March 1975 in Bogor, Indonesia, obtained Bachelor of Science (BS) degree with honors in Computer Science with Jurusan Ilmu Komputer (Department of Computer Science) from Institut Pertanian Bogor (Bogor University of Agriculture), Bogor, Indonesia in April 1999. Prior to attending King Fahd University of Petroleum & Minerals (KFUPM), he worked as a full time lecturer from May 1999 to January 2001 in the Jurusan Teknik Informatika (Department of Informatics Engineering) in Sekolah Tinggi Teknologi Telkom (Telkom School of Engineering), Bandung, Indonesia. In February 2001, Hermadi joined KFUPM as a Research Assistant to pursue the master's degree. The successful defense of this thesis in May 2004 marks his acquisition of the Master of Science (MS) degree in Computer Science. Hermadi's research interests include soft computing, evolutionary computation, software testing, and evolutionary structural software testing. He can be reached at irman_hermadi@yahoo.com or irman_hermadi@hotmail.com.