Path Testing using Genetic Algorithm

Irman Hermadi

A thesis submitted in fulfilment

of the requirements of the degree of

Doctor of Philosophy

School of Engineering and Information Technology

University College

University of New South Wales

Australian Defence Force Academy

December 1, 2015

Originality Statement

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

Signed

Date

Acknowledgements

All thanks are due to Allah first and foremost for His countless blessing.

Acknowledgment is due to the Indonesia General Directorate of Higher Edu-cation (IGDHE), Ministry of National Education, Republic of Indonesia for supporting this PhD research, and the University of New South Wales at the Australian Defence Force Academy (UNSW@ADFA) for providing the completion scholarships.

My unrestrained appreciation goes to my advisor, Dr. Chris Lokan, for all the help, guidance, support, and patience he has given me throughout the course of this work and on several other occasions. I also wish to thank my co-supervisor, A/Prof. Dr. Ruhul Sarker, for his help, support, and contributions.

I also acknowledge my many colleagues and friends as I had a pleasant and fruitful company with them, especially, Amr S. Ghoneim, Nurhadi Siswanto, Theam Foo Ng, Shir Li Wang, Helen Gilroy, Dominik Beck, Miriam Brandl, Gabriele Lo Tenero, Federica Salvetti, Dimas Okky Nugroho, Hendra Gunawan Harno, Sheila Tobing, Medria Hardhienata, Ida Nurhayati, Saber El-Sayed, Ahmed Fathi, Heba El-Fiqi, Mohammad Esmaeil Zadeh, Eman Hasan, Farhana Naznin, George Leu, and Bing Wang.

I am also indebted to many members of the school, who provided such a wonderful atmosphere, especially, Mrs Elizabeth Carey, Ms Christa Cordes, Ms Rong Wang, Ms Elvira Berra, and Ty Everett. I also would like to thank Denise Russel for having the thesis proofread.

The special loving support from my dear wife Anne, daughter Aliya, and son Imran that gave me strength and confidence when I thought none existed. I sincerely appreciate you.

Finally, I wish to express my gratitude to my *dad*, *mom*, *brothers*, *parents in law*, and *family members* for being patient with me and for their love for my success.

Abstract

Software testing is a critical stage in the life cycle of software development. The testing process costs hundreds of billions of dollars per year worldwide. Therefore, even small improvements in it could lead to very large savings.

In software, testing mainly refers to dynamic testing. It involves designing and generating proper test cases for the software, executing the software with these test cases, and observing the results. The aim is to find maximum number of errors with minimum number of test cases.

Black-box testing involves seeing whether the program behaves as expected. White-box testing involves knowledge of the code, to see which parts of the software were exercised during program execution. This thesis is concerned with white-box testing.

The strongest form of white-box testing is path testing, in which test data is sought that leads to all different execution paths through a program being exercised. Searching for an input datum in the program's input space that meets the path coverage criterion is a search problem. Evolutionary algorithms have proven to be suitable for searching for test data.

The aim of this thesis is to evaluate and improve the use of genetic algorithms (GA) for path testing. First, the capability and limitations of GA-based path testing are identified. Next we investigate possible ways to improve GA for test data generation. The obtained results are a collection of test programs; a classification scheme for test programs; understanding key GA parameters and identifying an effective default GA parameter setup; understanding the effects of infeasible paths; a model that stops GA searching when it seems ineffective to keep going, saving time while still finding almost all feasible paths; and identifying an effective hybrid of GA and local search techniques.

The results have the following implications:

- The collection of test programs can serve as a testing benchmark for proposed software testing approaches.

- The classification of test programs provides ways to group test programs with similar characteristics.

- The key parameters are shown in order to be population size, allele range, and number of generations. Suitable default values for these parameters are identified, which can represent a common setup to be used with an unknown new test program.
- The existence of infeasible paths is shown not to hinder the process of test data generation, rather it improves the performance by encouraging diversity in the population. This means that the costly task of analyzing target paths to identify and remove infeasible ones is not necessary.
- The proposed stopping criteria can stop searching whenever it is not worth continuing because the likelihood of finding further paths is too low. It is shown to be effective in stopping searching quite early, while missing almost no feasible paths. One less arbitrary system parameter (maximum number of generations to search) can be removed as a result.
- The hybrid GAs improve the performance both in terms of path coverage and number of generations required for the search.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Software testing is an important phase in the life cycle of software development. Errors in software can have very bad consequences. Finding errors in software is critical, because if it does not work as expected, it can have significant impacts including financial loss, time loss, the creation of bad reputation, and injury or even loss of life.

Some examples of incidents related to software failure are [1]: in February 2003 the U.S. Treasury Department sent 50,000 social security checks by mail with the beneficiaries' names missing; in October 1999 NASA lost its Mars Climate Orbiter (an interplanetary weather satellite), which cost $125 million, due to data conversion error (the system expected metric units in meters but received them in the English units of yards); and in June 1996 the European Space Agency's launch of the Ariane 5 rocket's first flight failed, resulting in an uninsured loss of $500 million, because of the lack of exception

handling for a floating-point conversion (loss of precision).

Thus, detecting errors before the software goes into operation is very important, and this is the task of software testing. Software testing is a process of making sure that the software being developed will function as specified and/or executing it with the intention of finding errors [2].

The testing process can consume almost half the total cost of software development [3]. Since this amounts to hundreds of billions of dollars per year worldwide [1], even small improvements could lead to very large savings.

Dynamic software testing involves designing and generating suitable test cases for the software. It feeds the test cases into the system and observes the results. The aim is to produce the minimum number of test cases that can find the maximum number of errors. Since test data generation is expensive [3], there is a great deal of interest in automating it.

Dynamic testing is divided into black box and white box testing. In the former, a program is given a set of test cases the input of each of which is fed into the program, and the actual output is compared with its expected output. In white box testing (a.k.a. structural testing), the internal structure (e.g. source code or control flow graph CFG) of the program is visible. CFG is a logical flow that shows the transfer of controls from one statement to another inside a program. White box testing involves executing a program and seeing which parts of it are executed; if there are any unreachable parts of the program after thorough testing, it is likely that the program contains logical error(s).

White box testing can be classified into (from the weakest to the strongest)

[2, 4, 5, 6, 7]: statement, decision/branch, condition, decision-condition, and path coverage. Statement coverage aims for every statement of code in the program to be executed at least once. Decision coverage aims for each branch of each decision to be exercised at least once. Condition and decision-condition coverages are stronger versions of branch coverage. Path testing is the strongest form of structural testing, but the hardest to achieve. Its aim is to execute every logical path through the program. This is difficult in the presence of loops, because executing a loop once before loop termination is considered to be a different path from executing it twice, and so on. Therefore, in a program that has loops, complete path coverage testing is impossible. However, a limited version of path coverage, i.e. limiting the number of loop iterations required, may be achieved.

Given the scale of testing, even small gains add up to a lot [3, 1]. This has driven a desire to automate it. The next challenge is to answer the following question. Is black/white box testing amenable to automation?

As one of the black box testing techniques, equivalence partitioning is to find minimum test cases that cover maximum equivalence partitions [2]. It has two main steps [2]: (1) identifying the equivalence classes and (2) defining the test cases. The former involves examining each input condition, which is a sentence or phrase in the specification and partitioning it into two or more groups. The latter involves selecting specific values from each valid and invalid equivalence class. Although this testing is a mainly heuristic process, it is still possible to be automated.

In white-box testing, one must see the internal structure of the program in order to assess the coverage of its logical flow [2]. Since the structure/flow

of a test program is visible, it is also called (logic) coverage or structural testing. Coverage testing has the following stages: selection of coverage criterion, construction of coverage (assessment) function, instrumentation of test program, and test data generation. Criteria of the coverage are selected based on the test program requirements. The coverage function can be constructed from the structure of a test program with respect to the chosen coverage criteria. The instrumentation is a process of parsing and inserting probes into the test program at certain spots concerning the coverage function needs. All the involved stages are amenable to automation.

This research is concerned with white box testing, in particular, path testing. Path testing is worth studying because it is the strongest form of white box testing. It has the greatest chance of detecting potential errors.

1.2 GA-based Path Testing

Searching for an input datum in an input space of a program that satisfies a testing coverage criterion, e.g., directing to navigate to a particular path, is a search problem. This branch of knowledge is part of what is now called search-based software engineering (SBSE).

Most test data generators used a gradient descent algorithm to automate software testing during the early stages of its automation [8]. The core of the algorithm is a kind of hill-climbing. It was quite time-consuming and could not escape from local optima in the input space.

Later on, meta-heuristic search algorithms became a promising alternative for developing test data generators [9, 10]. These algorithms include

Simulated Annealing (SA), Taboo Search (TS), Genetic Algorithm (GA), Ant Colony Optimization (ACO), and Particle Swarm Optimization (PSO). Each has its advantages and limitations. The algorithms are highly problem domain dependent because of the usage of domain-dependent knowledge or heuristics related to the domain of the test program. Of these algorithms, Wegener et al. have proven the suitability of using evolutionary algorithms in test data generation [11].

Recently, many successful applications of genetic algorithms for the generation of test data for structural testing have been evidenced [12]. Most of these research has concentrated on statement or branch coverage, with little on path coverage [13, 5, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25], and has treated high coverage as the only objective. In reality, most of the time, test data generation is undertaken with not only one objective but others, such as maximizing coverage while minimizing memory consumption or minimizing the number of test cases. Some research exploring a multiobjective GA (MOGA) for test data generation is beginning to appear, but with weaker criteria than those of path coverage [26].

1.3 Improving GA-based Path Testing

The aim of this thesis is to improve the use of GA for path coverage test data generation.

First, the capability and limitations of GA for this purpose will be identified. This step involves understanding its behavior with different parameter values and different fitness functions, and the types of programs or paths for

which it is difficult/easy for GA to generate test data.

Then we investigate ways to improve GA for test data generation. We investigate the practical setup of GA parameters; how to stop a GA run as soon as the search is not worth continuing further; and hybridizing GA with other search techniques.

Setting up the parameters using generic values is most practical in GA-based path testing. To search for best parameter values for each test program is a time-consuming trial and error process. It would be an invaluable contribution to know a common parameter setup that works well for most test programs without missing significant numbers of target paths.

Having infeasible paths in a program is almost inevitable, especially in programs that have many selections, loops, and/or compound selection statements. Therefore, in general it is unlikely that searching will be able to stop with all target paths found. Determining when to stop searching for test cases to cover further uncovered paths is very crucial. Premature stopping could be costly, by missing some feasible paths, while continuing to search for too long wastes time, and can never be successful if some target paths are infeasible. So, a mechanism is needed that could stop the test data generator when it is not worth continuing while achieving a high level of coverage.

Hybridization is one way that may improve the test data generation. A GA can be hybridized with a local search algorithm such as hill climbing. The purpose is to locally improve a certain number of best candidates within the GA population. Thus, it is expected to improve GA performance in general.

1.4 Contributions to Knowledge

The thesis's main contributions are a new test program classification system, new stopping criteria, and hybridization of GA-based path testing. The following are the details:

1. Classification of test programs based on their structure and the complexity of their expressions. The classification is utilizable for exploring the characteristics of test programs with respect to the approaches being used.

2. Collection of some test programs that are used in the GA-based coverage testing criteria studies and mapping them to the classification. This enables a set of benchmark test programs to be identified, supporting comparisons between different studies.

3. Providing a generic parameter setup for practical GA-based test data generation.

4. Studying the effect of the existence of infeasible paths. We show that infeasible paths do not degrade the performance of the generator. Making all target paths feasible, by going to the effort of removing the infeasible ones, causes the test data generation to work longer to find the same number of feasible paths.

5. Proposing a new dynamic stopping criterion, avoiding the need for a parameter to represent the maximum number of search generations. This is helpful and practical in the presence of infeasible target paths.

6. Hybridizing GA with local search LS, showing that this can improve path testing, and identifying a way to combine GA with LS that performs best.

1.5 Structure of The Thesis

The remainder of this thesis is organized into the following chapters. This organization reflects the methodology adopted for this research.

Chapter 2 presents a theoretical background of the main issues studied in the thesis.

Chapter 3 reviews the relevant literature and state-of-the-art of evolutionary path testing. In detail, the chapter covers search-based software engineering (SBSE), GA-based test data generation, software reliability growth models, infeasible path detection, and test programs.

Chapter 4 describes a classification scheme for test programs, and the choice of test programs used in this research. It also describes the test program preparation steps, and the preparation and execution of the experiments conducted in this research.

Chapter 5 explores the challenges and key parameters in GA-based path testing. It also provides a baseline and basic experimental setup for the following chapters. Best and common parameters setups are extracted from the results.

Chapter 6 presents and evaluates a model for deciding when to stop searching for test cases to cover paths that are not covered yet. The chapter

presents the theory behind the model and evaluates its performance experimentally with different decision rules.

Chapter 7 describes hybridization between GA and local search. Several hybrid variants are elaborated in detail. Training and testing phases yield experimental results that validate the analysis empirically. The analysis includes test program classification, effects of infeasible paths, and effects of the dynamic stopping condition.

Chapter 8 summarizes the results and analysis and concludes the thesis.

Chapter 2

Theoretical Background

This chapter briefly explains primary topics in search-based software testing, in order to build up fundamental knowledge to understand the thesis work. In detail, it covers topics on software testing, test data generation, search-based software testing, GA-based test data generation, infeasible path detection, and software reliability.

2.1 Software Testing

The purpose of software testing is to detect software defects, so they can be fixed and the software then functions as specified when it is delivered to its users [3, 27].

Although it is not possible to remove every error in a large software package, the goal of testing is to remove as many as possible during the development cycle. However, as it can only reveal the presence of errors in software, not their absence, testing cannot prove that a program is bug-free

[3, 27]. In other words, if the testing found any errors then the software is certainly faulty. On the other hand, if no errors were found it does not necessarily mean that the software is flawless.

In general, software testing approaches are classified into two groups [28, 7, 3]: static and dynamic methods. Static testing requires no actual execution of software; rather it analyzes the internal software representation. It includes symbolic execution, which assigns expressions to variables as a path in the code, and domain reduction, which reduces input domain until no further reduction is feasible and generates random input from it. In static analysis, a code reviewer reads the program source code, statement by statement, and mentally follows the logical program flow by tracing the processing of 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. This type of testing is not completely manual work because some automated supports are available.

In contrast, dynamic testing techniques execute the program and observe its output. Most logical errors are hard to find, unless the actual program is physically executed with input combinations that expose incorrect behaviour. Usually, the term "testing" in relation to software refers to dynamic testing.

In software testing, test cases are fed into the program, and the behaviour or actual output is observed. The main task in dynamic software testing is test case generation, i.e. producing a set of input data (test data) or a pair of input data and its expected output, that meets certain testing criteria. Because generating test cases is expensive, the objective of test case generation is to generate as few test cases as possible that can reveal as many

errors as possible.

2.1.1 Static Analysis

For years, the majority of programmers assumed that the only way to test a program is by executing it on a machine. This attitude began to change in the early 1970s, because of Weinberg's work, "The Psychology of Computer Programming" [29]. Weinberg provided a convincing argument for programs being 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 [7]. They are intended to be applied during the period between code completion and the beginning of execution-based testing.

Typical static analysis methods are code inspections, code walkthroughs, desk checking, and code reviews [7]. Code inspections and walkthroughs are the two primary static analysis methods. They have a lot in common, as they both involve the reading or visual inspection of a program by a group of people. Both methods involve some preliminary work by the participants. The climax is a brainstorming meeting in a conference-like gathering, the objective of which is to find errors but not solutions to them, i.e., to test but not debug.

Code Inspection This is a set of procedures and error-detection techniques for code reading by a group [7]. Its discussions focus on the procedures, forms to be filled out, etc.

Two main activities are conducted: code narration and code examination. The code is read, line by line, and analyzed with respect to a checklist of common programming errors, e.g., data-reference, data-declaration, computation, comparison, control-flow, input/output, and interface errors [7].

Code Walkthroughs Its task is similar to that of the inspection process [7]. The difference is that, rather than simply reading the program or using error checklists, one participant acts as a tester with a set of test cases that consists of sets of input and expected output. During the assembly, each test data is walked through the logic of the program and the programmer explains the logic of their code.

Desk Checking This can be considered as a one-person inspection/walkthrough [7]. A person reads code, checks it, and/or walks test data through it.

Code Review (Peer Rating) This is a method to review anonymous programs in terms of quality, maintainability, extensibility, usability, and clarity [7]. The purpose of the code review is to provide an assessment of the programmer. A group of programmers rates some selected unidentified programs based on a scale written on a review form.

2.1.2 Dynamic Analysis

Dynamic testing techniques execute the program and observe its output. There are two types of dynamic testing [7, 27]: black-box and white-box. The latter is concerned with the degree to which test cases exercise or cover the logical flow of the program [7]. On the other hand, black-box testing

tests the functionality of the software regardless of its internal structure, a.k.a. functional or specification-based testing.

1. White-box testing. It is also called logic-coverage testing or structural testing because it looks at the structure of a program [30]. Its objective is to exercise the different logical structures and flows in the program. The adequacy of logic-coverage testing can be judged using different criteria [30]: statement, decision (branch), condition, decision/condition, multiple-conditions, and path-coverage (ordered from the weakest to the strongest [6, 5, 4, 7]).

   Statement coverage This criterion requires every statement in the program to be executed at least once. This is a weak criterion because, while it exercises every statement at least once, it does not guarantee that the same statement is exercised in different flows. For example, given the following program segment:

   ```
   S1
   IF (A>1) THEN S2
   S3
   ```

   One is not required to generate an input test datum that exercises the FALSE branch in order to satisfy the statement coverage criterion. In this case, test cases may check for the correctness of the sequence S1-S2-S3, but not necessarily for the correctness of the sequence S1-S3, which may have a problem.

   Decision (branch) coverage This has a stronger logic-coverage criterion [31], which states that one must write enough test cases for each decision, i.e., an IF statement, to have at least one TRUE and one FALSE outcome. The following example shows why the branch coverage criterion is stronger than statement coverage.

```
IF (A>1) THEN X = S2
```

In order to fulfil the branch coverage criterion, one must generate at least two test input data that satisfy both TRUE and FALSE branches regardless of any statements that follow either, whereas to fulfil the statement coverage criterion, the tester needs to generate only input test data that leads to the TRUE branch.

The problem with branch coverage is that it does not check for all different sequences; for example, in the following two serial selection statements.

```
IF C1 THEN S1
    ELSE S2
IF C2 THEN S3
    ELSE S4
```

Branch coverage might test only the S1-S3 and S2-S4 sequences OR the S1-S4 and S2-S3 sequences. In fact, all sequences of S1 to S4 should be checked in order to reveal any potentially infeasible combinations of sequences.

Condition coverage This criterion has stronger coverage than that of decision coverage, as sufficient test cases must be written for each condition in a decision to handle 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. If the decision IF (A AND B) is being tested, the condition coverage criterion will require one to write two test cases, i.e. A is TRUE and B is FALSE, and A is FALSE and B is TRUE, neither of which

would cause the THEN clause of the IF statement to execute.

The two cases already covered all possible conditions for A and B such as A is TRUE or FALSE, and B is TRUE or FALSE. The way of combining both conditions, that causes the TRUE side will not be executed. As with decision coverage, condition coverage does not always lead to the execution of each sequence.

Decision/condition coverage This criterion combines both the decision and condition coverage criteria.

It requires sufficient test cases for each condition in a decision to handle all possible outcomes at least once, each decision to handle all possible outcomes at least once, and each point of entry to be invoked at least once. A weakness of 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 the statement will be FALSE if A is FALSE without considering B's value at all.

Also, errors in logical expressions are not necessarily made visible by the condition coverage and decision/condition coverage criteria, since they do not test all possible combinations of condition outcomes in each decision.

Multiple-conditions coverage This criterion covers the problem faced by decision/condition coverage.

It requires one to write sufficient test cases for all possible combinations of condition outcomes in each decision, and all points of

entry, to be invoked at least once.

Path coverage This achieves the utmost logical coverage since it covers all the previously-mentioned testing coverage criteria [4, 30, 7]. The path coverage criterion is concerned with the execution of all logically different paths in a program. For a program with loops, since the execution of every path is usually infeasible, complete path testing is not considered a feasible testing goal.

2. Black-box testing, a.k.a. functional or specification-based testing, tests the functionalities of a software against its specification, regardless of its structure. There are four types [7]: equivalence partitioning, boundary-value analysis, cause-effect graphing, and error guessing.

   Equivalence partitioning partitions the input space of a program into a set of equivalence classes. An input from a particular class is equivalent to any other input within the same class. If this input exposed an error then any other inputs from this class could expose the same error too. On the contrary, if this input did not expose an error then no other inputs in the equivalent class would expose an error.

   Boundary-value analysis analyzes the boundary conditions which are directly on, above and/or beneath the edges of the input and output equivalence classes.

   Empirical evidence shows that test cases that make use of values on and adjacent to boundaries between equivalence classes are more effective than those that do not [7]. Boundary-value analysis and equivalence partitioning do not explore combinations of input

data as decision/condition coverage does. However, cause-effect graphing has been developed to tackle this problem [7].

Cause-effect graphing is a formal language that translates natural-language specification. It takes out incompleteness and ambiguities in the specification. It is similar to that of the decision/condition coverage criterion. This analogy means cause-effect graphing outperforms boundary-value analysis and equivalence partitioning.

Error guessing is an ad-hoc and intuitive process. Its procedure is difficult to formalize. It needs expertise to point out errors. It works by enumerating a list of error-prone situations and generating test cases based on the list.

2.2 Test Data Generation

In this thesis, we focus on white-box testing.

The first step in applying white-box testing is to select an adequacy criterion, e.g., statement coverage, branch coverage, path coverage. The next is to generate a set of test data that satisfies the selected adequacy criterion, which is called adequate test data [8, 32, 33]. As generating adequate test data manually is a labour intensive and time-consuming process, researchers have been motivated to create test data generators that can examine a program's structure and generate adequate test data automatically [34]. However, how to generate test data automatically and evaluate them are major questions to which researchers in the area of automated software testing are

trying to find the answers [8, 7, 33, 35, 36].

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

A number of automatic test data generation techniques have been developed [36]. A test data generator is a system (or a set of programs) that generates the input data for a target program such that these input data satisfy a particular testing adequacy criterion. Pargas [37] classifies them into random, structural or path-oriented, goal-oriented, and intelligent. The first three are in accord with the classifications of test data generators espoused by Edvardsson [9] and Korel [8].

Random test data generators select random inputs for the test data from a distribution [8, 38]. 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 to cause that path to be executed [8, 13, 39, 40]. Goal-oriented test data generators select inputs to execute the selected goal, such as a statement, irrespective of the path taken [8]. Intelligent test data generators often rely on a sophisticated analysis of the code, to guide the search for new test data [13, 39, 37, 41].

This thesis is concerned with automatic structural test data generation, for path coverage. In general, the process consists of three major steps:

1. Construction of a control logic graph, e.g. control flow graph (CFG), control dependence graph (CDG),

2. Identification of different execution paths,

3. Test data generation, which involves dynamic execution of the target program.

The target program must be instrumented in order to monitor assessments of its testing objective when it is executed with given input data. In most test data generators, the instrumentation is considered to be in the pre-process stage before the generator can be used [9]. This instrumentation process involves inserting probes (tags) at the beginning/ending of every block of the code of interest, i.e., at the beginning/ending of each function and after the true and false outcomes from each condition. For example, in path coverage, these tags are used to monitor and provide the test data generator with feedback on the path traversed within the program while it is being executed with trial test data.

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

2.3 Search-based Software Testing

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

In the early stages of automated software testing, most test data generators used gradient descent algorithms [8]. As the essence of this type of

method is a kind of hill-climbing, it was 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 potentially better alternative for developing test data generators [9, 10]. Efficient existing meta-heuristic search algorithms include Simulated Annealing (SA), Taboo Search (TS), GA, Ant Colony Optimization (ACO), and Particle Swarm Optimization (PSO). Each has its advantages and disadvantages. They are strongly problem domain dependent because they use domain-dependent knowledge or heuristics related to the domain of the problem under consideration.

Of these algorithms, Wegener et al. have shown the suitability of using EA in software testing [11].

2.4 Path Testing

White-box testing is widely adopted [7]. It is also called logic-coverage or structural testing because it requires the structure of the program to be visible [30]. The main objective of the testing is to exercise different logic structures and flows in the program [7].

Different logic-coverage criteria are available, as described in Section 2.1.2. The utmost coverage is achieved by path coverage.

Path coverage is concerned with the execution of all different logical paths through the program. In a program with loops, the execution of every path

is usually not feasible. Thus, complete path testing is not considered in such cases. One way to handle these cases is to limit the number of iterations of loops. For example, every loop may be considered only up to a certain number of iterations: e.g. 0, 1, and 2 iterations. The logic is to cover all possible combinations: no iterations (not entering the loop), execute the loop once, and execute the loop multiple times, which is represented by two iterations.

2.5 GA-based Test Data Generation

GAs were invented by John Holland in the 1960s and developed by him, his students and colleagues at the University of Michigan during the 1960s and 1970s [42]. 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 [43].

Since that time, GAs have been a very interesting area of study in many disciplines and the number of research studies into either their behaviours or applications for particular purposes has increased rapidly. Some applications of GAs are optimization, automatic programming, machine learning, economics, immune systems, ecology, population genetics, evolution and learning, and social systems [42].

Some features of GAs, which other normal optimization and search procedures do not have, are their capabilities to directly manipulate a representation of a solution to a problem, as well as search from a population (not

a single point) via sampling (a blind search) and using stochastic operators (non-deterministic rules) [44].

The basic steps in GA are:

1. Define a genetic representation of the problem
2. Create an initial population $P(0) = x_1, \ldots, x_n$, and set t = 0
3. Compute the average fitness $f'(t)$, and assign each individual the normalized fitness value
4. 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 $P(i)$, which gives the set of selected parents
5. Pair all parents at random using their survival probabilities to form $\frac{N}{2}$ pairs; then apply a crossover with a certain probability to each pair, and other genetic operators such as mutation, to form a new population $P(t+1)$
6. Set $t = t + 1$ and return to Step 3

In order to use a GA to solve an optimization problem, we need to know how to represent the problem as well as its solution in a chromosome-like expression, i.e., a sequence of binary digits that a GA can understand and manipulate [45, 46]. As the GA then works on this encoded problem and delivers the interpreted result as the problem solution, the user should provide the semantics of the encoded problem. The most widely used representation is a binary string. However, in recent developments, representation can

be extended into higher numbering systems, up to more complicated data structures [4, 42, 47]. Investigations into more advanced representations, e.g., character, integer, float, grouped, messy, and record, are continuing [42].

The fitness value of an individual is the measure of its strength to survive in the next generation [43, 46, 48]. It reflects the chance that the individual has to be directly present in the next generation or to be selected for mating with other individuals in the current generation to produce children for the next generation. This fitness value is calculated based on the syntax and semantics of the individual representation and is, mainly, a normalized value of its objective value such that it can be minimized or maximized accordingly.

A complete iteration from Step 3 to Step 5 above is called a generation. The stopping criteria comprise a desired number of generations or a measure of convergence or saturation.

There are two approaches for implementing a GA as a problem solver [49]: (1) a classical GA, which operates on a binary string, requires modification of the original problem into an appropriate form (suitable for a GA). This includes a mapping between potential solutions and binary representations, taking care of the decoders or repairing algorithms, etc; and (2) a GA can leave a problem unchanged but modify an individual representation of a potential solution (using "natural" data structures) and apply appropriate "genetic" operators.

A good operator is one that can guide a search faster, thereby reducing the time it takes and significantly reducing its search space. Many advanced GA operators have been explored, and some researchers are trying to create

a parameter-less GA for which the user does not need to select or adjust operators [10].

Two major operators are used in almost every implementation of a GA: crossover and mutation. A simple crossover operator is a single-point or uniform crossover while simple mutation means native mutation as specified in [45]. For example, consider two binary-string individuals $x_1 = 10101$ and $x_2 = 01010$ with single-point crossover and mutation rates of 0.9 and 0.1, respectively. In the crossover stage, the GA generates a random number between 0 and 1. Suppose it happens to be 0.5, which is less than 0.9: hence the two individuals cross towards each other at a randomly selected point between them; suppose that point is 3. Hence, the new individuals are $x_1 = 10110$ and $x_2 = 01001$. In a uniform crossover, both bit-sequences are shuffled between these two individuals, i.e. $x_1 = 11111$ and $x_2 = 00000$. In the case of a mutation after a cross-over, the GA generates a random number between 0 and 1 for each and, if it happens to be less than 0.1, any bits within that individual, the positions of which are again selected randomly, will be flipped, i.e., from 0 to 1 or vice versa. Based on experience, typically, the mutation rate is set to between 0 and 0.1 and the crossover rate to between 0.6 and 1 [45, 44, 49, 50, 51, 5]. These two operator rates control the population in terms of the exploration and exploitation of the search space. In order to choose the most suitable rate, a trial and error approach is still the most widely used method among researchers.

During the selection stage, a GA will most probably select individuals that have performances above that of the current population average, to persist in the upcoming intermediate generation resulting from the crossover

and mutation stages. Individuals with lower strength will vanish as the GA evolves from generation to generation. The next population is highly likely to contain copies of previous individuals (i.e., parents), as well as some new individuals that are totally different from their ancestors. The degree of variation among the new individuals introduced into this new population depends on the crossover and mutation rates. A high crossover rate will completely mix the characteristics of both parents into its offspring, while a high mutation rate will produce an offspring that has different traits from its parents, i.e., the offspring introduces new traits that do not exist in its parents at all.

Why does a GA work? This is a very interesting question for anyone regardless of whether he/she knows about GAs. A 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 will be reproduced more and more in subsequent generations [45].

Initially, metaheuristic techniques gained greater popularity than traditional optimisation methods due to their capabilities to solve complex optimisation problems more effectively and efficiently [52]. As time passed, it was realised that metaheuristic and metaheuristic/non-metaheuristic techniques do not compete against, but complement, each other, which led to the development of hybrid methods [52]. In other words, a hybrid combines the strengths of all participating methods in order to compensate or reduce, if not eliminate, one or all of their weaknesses [53]; for example, although a GA is one of the most successful metaheuristic techniques, it may still suffer

from premature convergence. However, as SA is often slow to converge to an optimal or near-optimal solution, hybridising these two methods is one way of overcoming their disadvantages [54].

Readers that are interested in the theoretical background to, and/or applications of, GAs are encouraged to consult distinguished references [45, 44, 49, 42, 53, 52].

Many research papers have shown that GA has a promising future in the development of test data generators [13, 39, 46, 55] and some indicate that it outperforms both Simulated Annealing and Taboo Search [10]. However, in 2004, Mansour and Salame [18] reported that Simulated Annealing slightly outperforms Genetic Algorithm for path testing.

In testing coverage criteria, especially branch or path coverage, most test data generators make use of approximation level and branch distance (or predicate value) as components in their fitness functions [56]. The approximation level measures the number of overlapping branches between a traversed path and a target path: more overlapped branches means closer to the target path. The branch distance is the value required to change the traversed path changes to the target path on the last overlapped branch: to make the branch distance 0. The distance is calculated using Korel's distance function as listed in Table 2.1 [8].

2.6 Infeasible Path Detection

Infeasible path detection approaches can be classified into three classes: static, dynamic, and hybrid. A static approach uses an analytical process

Table 2.1: Korel's Distance Function

No	Branch	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)

to generate test data and/or detect feasible (or infeasible) paths while a dynamic one makes use of actual program execution using real test data. A hybrid approach merges both analysis and real execution approach. A static approach always gives a correct result, but it requires meticulous work and more time, and it is hard to automate. On the other hand, a dynamic approach produces results with varying degree of correctness, but it is easy to automate. Hybrid approaches aim to combine the benefits of always getting correct results and easy automation.

Two common major drawbacks in existing work are lack of test problem scalability and generalization.

2.7 Software Reliability

A new stopping criterion for GA-based path testing is investigated in this thesis. It is inspired by software reliability growth models. The following presents the background for this idea.

Reliability is one characteristic of software quality [57, 27, 3]. According

to ISO 9126, it can be defined as "a set of attributes that bear on the capability of software to maintain its performance level under stated conditions for a stated period of time" [57].

Reliability is a user-oriented quality metric related to the operation of a software system. A fault-free software is considered to be highly reliable, and even one that has an acceptable level of frequency of failure may also be considered reliable. In discussing reliability, the followings are three of its key concepts [57]:

Failure is an observable behaviour of a program execution that is different from the expected one.

Fault is something that can cause a failure, which is most likely invisible and very hard to detect in advance.

Time being short between two successive failures indicates less reliable software. Two forms of time are execution time τ and calendar time t.

Two ways of measuring reliability are counting the number of failures over periodic intervals and assessing a failure's intensity [57]. The former is the total number of failures until time τ and denotes a cumulative failure count $\mu(\tau)$. The latter is the number of failures observed per unit time after time τ and denotes failure intensity $\lambda(\tau)$. Hence, the relationship between them can be formulated as $\lambda(\tau) = \frac{d\mu(\tau)}{d\tau}$.

From a user's perception, software reliability has always been influenced by the number of faults and the user's operational profile [57], which is related to the ways in which the user operates the software. The number of faults is mainly affected by: (1) the size and complexity of the code; (2)

the development process's characteristics; (3) the education, experience and training of the development personnel; and (4) the operational environment.

If one knows how to measure the reliability of software, one can apply this in several ways. The one that is relevant here is that it can be used to decide whether more tests need to be conducted [57].

A way of formalizing the software reliability concept is to develop mathematical models for $\mu(\tau)$ and $\lambda(\tau)$. The following are five basic assumptions for developing a reliability model [57]:

1. Faults in the programs are independent.

2. The execution time between failures is larger than the instruction's execution time.

3. The potential test space covers the used space.

4. The set of inputs per test run is randomly chosen.

5. The fault causing a failure is immediately fixed or else its re-occurrence is not counted again.

Intuitively, by observing new system failures and fixing previous faults, there will be fewer faults remaining and a smaller failure intensity. In other words, as $\mu(\tau)$ increases $\lambda(\tau)$ decreases. As depicted in Fig. 2.1, two types of decrement processes for failure intensity can be defined as follows [57]:

Basic model The failure intensity decreases **constantly** as failures are manifested, and their corresponding faults fixed.

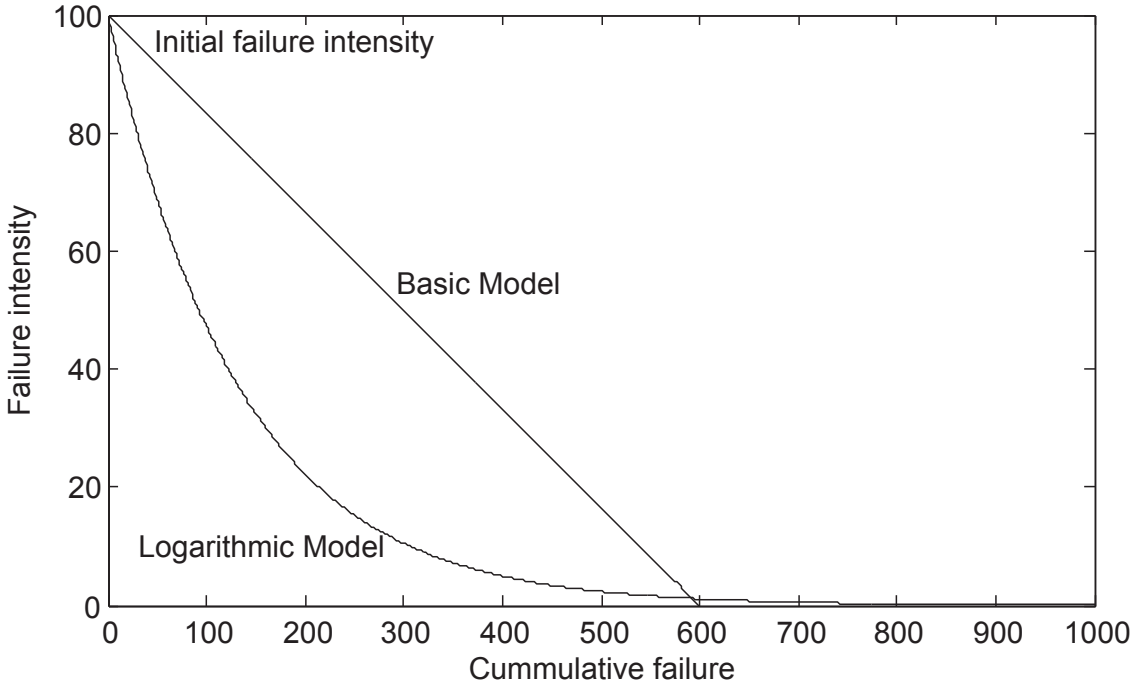


Figure 2.1: Plot of $\lambda(\mu)$

Logarithmic model Each failure intensity level is **smaller** than the previous one as failures are manifested, and their corresponding faults fixed.

Having established the basic assumptions and intuitive idea, the next step is to define the parameters of the models and then build them. Three identified parameters are the initial failure intensity λ_0, the expected number of failures over infinite time v_0, and the non-linear drop failure intensity θ, which give the following models [57]:

Basic model Assumption: $\lambda(\mu) = \lambda_0(1 - \frac{\mu}{v_0})$

$\frac{d\mu(\tau)}{d\tau} = \lambda_0(1 - \frac{\mu(\tau)}{v_0})$

$\mu(\tau) = \lambda_0(1 - \frac{\mu}{v_0})$

$\lambda(\tau) = \lambda_0 e^{-\frac{\lambda_0 \tau}{v_0}}$

Logarithmic model Assumption: $\lambda(\mu) = \lambda_0 e^{-\theta\mu}$

$$\frac{d\mu(\tau)}{d\tau} = \lambda_0 e^{-\theta\mu(\tau)}$$

$$\mu(\tau) = \frac{ln(\lambda_0\theta\tau+1)}{\theta}$$

$$\lambda(\tau) = \frac{\lambda_0}{\lambda_0\theta\tau+1}$$

For example, given $\lambda_0 = 10$, $v_0 = 500$, and $\theta = 0.0075$ for Logarithmic Model. Fig. 2.2 describes the function $\mu(\tau)$, the cumulative failure μ as a function of the execution time τ. In Fig. 2.3, the failure intensity λ as a function of the execution time τ is depicted.

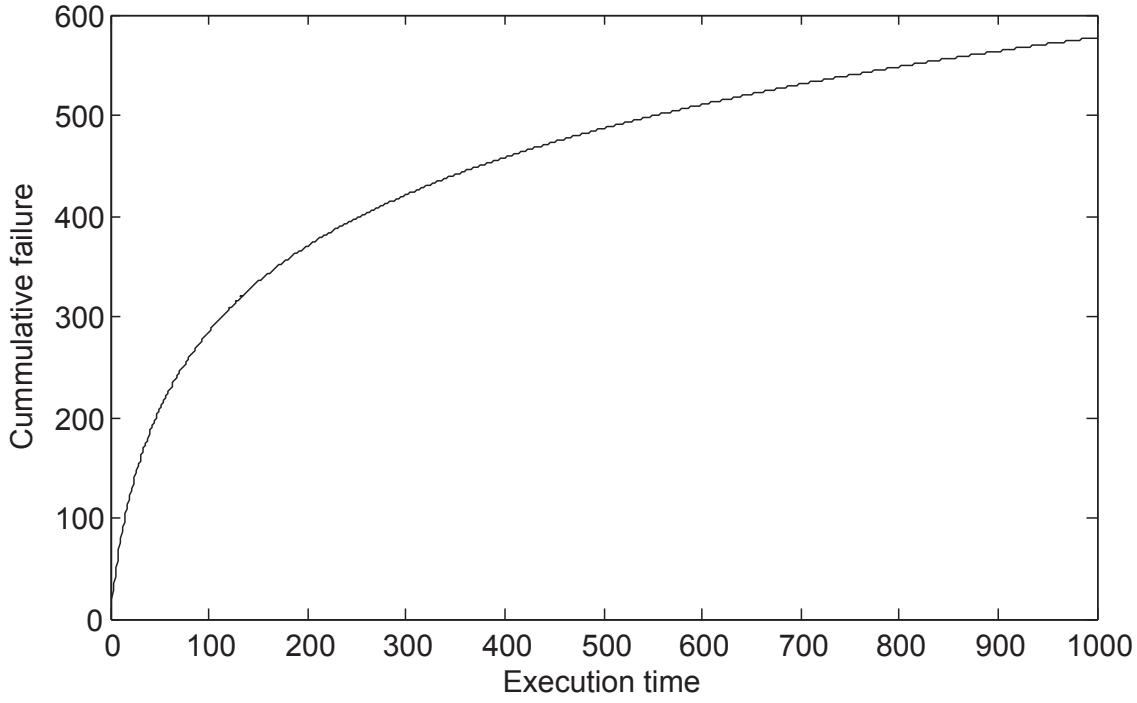


Figure 2.2: Plot of $\mu(\tau)$

The following are the details of the model. The model can be defined in terms of a random process $\{M(\tau), \tau \geq 0\}$ which represents the number of failures occurred during execution time τ. The distribution of $M(\tau)$ has the

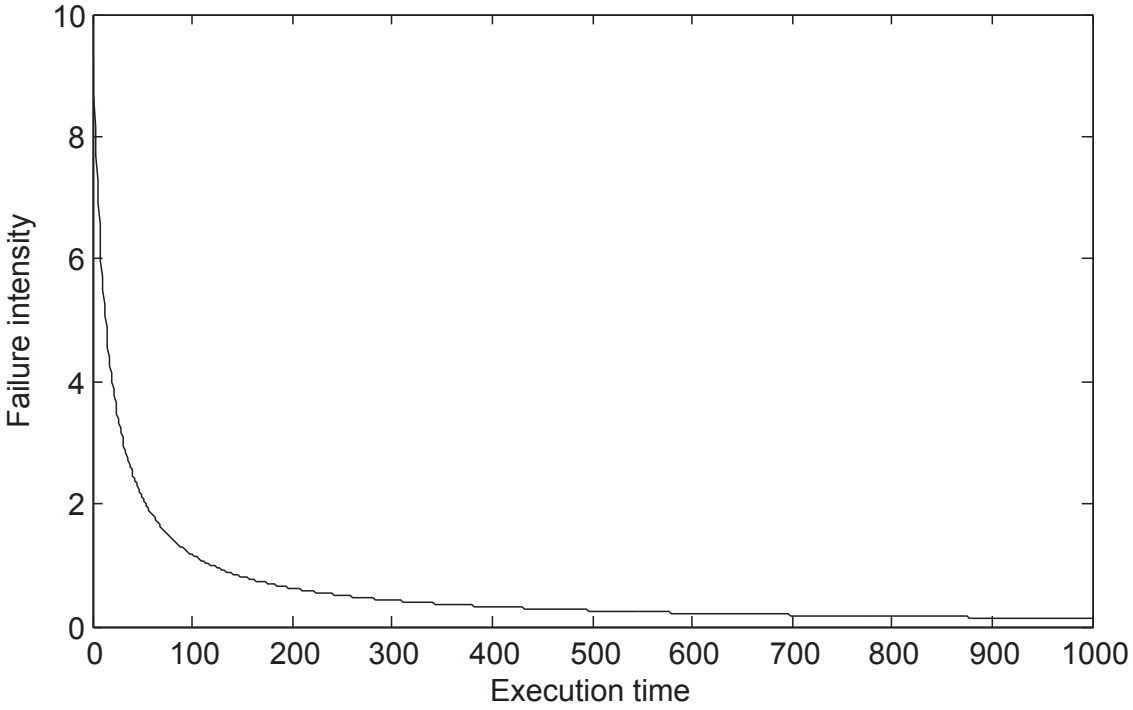


Figure 2.3: Plot of $\lambda(\tau)$

mean value function

$$\mu(\tau) = E[M(\tau)] \tag{2.1}$$

and the failure intensity function

$$\lambda(\tau) = \frac{d\mu(\tau)}{d\tau} \tag{2.2}$$

The following model assumptions for the execution time component are [58]:

1. There is no failure observed at time $\tau = 0$, i.e. $M(0) = 0$ with probability one.

2. The failure intensity will decrease exponentially with the expected number of failures experienced. In other words, $\lambda(\tau) = \lambda_0 e^{-\theta\mu(\tau)}$, where λ_0

and θ are the initial failure intensity and the rate of reduction in the normalized failure intensity per failure, respectively.

3. For a small time interval $\Delta\tau$ the probabilities of one and more than one failure during $(\tau, \tau + \Delta\tau]$ are $\lambda(\tau)\Delta\tau + o(\Delta\tau)$ and $o(\Delta\tau)$, respectively, where $\frac{o(\Delta\tau)}{\Delta\tau} \to 0$ as $\Delta\tau \to 0$. This is to prove that the model is of type Poisson.

Having these assumptions in mind, both $\mu(\tau)$ and $\lambda(\tau)$ can be derived as [58]:

$$\mu(\tau) = \frac{1}{\theta} ln(\lambda_0 \theta \tau + 1) \tag{2.3}$$

$$\lambda(\tau) = \frac{\lambda_0}{\lambda_0 \theta \tau + 1} \tag{2.4}$$

$\mu(\tau)$ represents the number of failures that occurred during execution time τ. $\lambda(\tau)$ is the failure intensity, the expected number of failures at a certain point in time. λ_0 is the initial failure intensity. θ is the rate of reduction in the normalized failure intensity per failure.

In [58], the two unknown parameters λ_0 and θ are estimated using maximum likelihood estimation method to guess the product of $\Phi = \lambda_0 \theta$ by using conditional joint density function. Two types of failure data were used for the estimations, i.e. failure intervals and number of failures per interval.

1. Failure Intervals Estimation based. Estimation is performed at a specified time τ_e where m failures had occurred during $(0, \tau_e]$ period. So,

$$\hat{\theta} = \frac{1}{m} ln(\hat{\Phi} \tau_e + 1) \tag{2.5}$$

and

$$\hat{\lambda_0} = \frac{\hat{\Phi}}{\hat{\theta}}. \tag{2.6}$$

2. Number of Failures per Interval Estimation based. Assume that an observation interval $(0, x_p]$ is partitioned into a set of p disjoint subintervals $(0, x_1], (x_1, x_2], \ldots, (x_{p-1}, x_p]$ and the number of failures in each subinterval is recorded. Let $y_l (l = 1, 2, \ldots, p)$ be the number of failures in $(0, x_l]$. So,

$$\hat{\theta} = \frac{1}{y_p} ln(\hat{\Phi} x_p + 1) \tag{2.7}$$

and

$$\hat{\lambda_0} = \frac{\hat{\Phi}}{\hat{\theta}}. \tag{2.8}$$

Deciding when to stop testing for unfound errors is analogous to deciding when to stop searching for uncovered target paths.

Equation 2.4 calculates the expected number of paths found at τ generations; it provides information on whether to stop searching. The likelihood of finding new target paths θ in Equation 2.4 is updated using $\hat{\theta}$ in Equation 2.7. Software reliability models base their decisions on the history of how many new errors are found after a given amount of testing; the analogy here is that the decision of when to stop searching is based on the history of how many new target paths are covered after a given amount of searching. This number of errors is computed using Equation 2.3.

2.8 Summary

This chapter has presented essential materials on GA-based path testing.

Sections 2.1 to 2.3 set the scene. Section 2.1 described approaches to testing, in particular defining different coverage criteria for white-box testing and highlighting the value of path coverage. Section 2.2 outlined some issues relating to test data generation, and Section 2.3 illustrated search-based approaches to test data generation. Section 2.4 addressed path testing in particular.

Sections 2.5 to 2.7 underpin the contributions of the thesis. Section 2.5 described search-based test generation using GA, and hybrid methods including GA (these are the main topics of Chapters 5 and 7). Section 2.6 discussed infeasible paths in path coverage, which both motivates the need for a criterion to decide when to stop searching (Chapter 6), and is studied in its own right for its effect on convergence (Chapter 7). Section 2.7 introduced the theory behind the stopping criteria studied in Chapter 6.

The next chapter reviews the development in the literature of these fundamental topics.

Chapter 3

Literature Review

This chapter presents a literature review on topics related to search-based software testing. The fundamental theories on the topics are described in the previous Chapter 2. In detail, the review covers advancements in the following topics: search-based software engineering, genetic algorithm (GA) based test data generation, software reliability growth models, infeasible path detection, and test programs.

3.1 Search-based Software Engineering

During the past decade, there has been a tremendous amount of work undertaken in search-based software engineering (SBSE), i.e., the application of search techniques to solve optimisation problems in the context of software engineering. SBSE is very interesting in terms of its flexibility, that is, it can be adjusted to an automatic or semi-automatic approach, and has the capability to handle problems with typically huge search spaces and competing

and conflicting objectives.

SBSE can be applied throughout the whole software engineering life cycle, ranging from project planning and requirements gathering to maintenance as well as re-engineering [59]. So far, though, SBSE is mostly used in the testing and debugging phase in the software development life cycle [12, 60]. For example, the most challenging question in software testing is to find the smallest set of test cases that cover all branches or whatever testing criteria are in a program. This is essentially an optimization question.

In 2009, Harman *et al.* [12] conducted a comprehensive survey on SBSE, covering over 500 publications. The survey reported that 70% of the publications relate to software testing. It also reported that 435 publications are on evolutionary algorithms, which includes genetic algorithms (beyond 315 publications), genetic programming (beyond 65 publications), and evolutionary strategies (beyond 55 publications). Thus, it is obvious that the most embraced technique in SBSE is evolutionary algorithms.

On one hand, GA has the following advantages: stochastic process, direct representation of the problem solution domain, handling constraint and objective functions, handling multiple solutions, handling large search space, not dependent on error surface (so it can solve multi-dimensional, non-differential, non-continuous, and non-parametric problems), handling modelling, and easier to run in parallel. On the other hand, it has the following drawbacks: no guarantee that optimal solution will be obtained and, like other optimization techniques, there is no assurance of constant optimization running (or response) times, making it hard to implement in real time situations.

In this research, GA is used for path testing because it can solve types of problem that the software testing has, i.e. test data generation in particular. In order to generate test data, a vast area of test data search space must be explored to find test data that meet the testing criteria. GA also has objective (or heuristic) function with the following surface characteristics: non-differential, non-continuous, and multi-modal. So, these are the driving factors that considered in choosing GA for path testing. Further, it also has been widely used for test data generation and empirically proven. In reality, finding near-optimal solutions with roughly constant running time is still acceptable for most of the optimization problems. This does not make GA unfavorable for test data generation, because all non-deterministic techniques share the same characteristics.

There are other meta-heuristic techniques that are population and stochastic-based, such as Ant Colony Optimization and Particle Swarm Optimization. However, GA has the advantages of direct encoding of problem domain, simpler operations, and fewer parameters, whose values can be set up using generic values, while producing optimal solutions with fewer resources for most optimization problems. GA's performance can be improved significantly by creating and/or selecting appropriate fitness functions and fine-tuning its parameters to fit the semantics of the problem domain and the representation of the solution. Having non-GA based optimization techniques would possibly require alternative fitness functions, and definitely require other parameter setups. Knowing these techniques' characteristics, there is no guarantee that they will work more effectively and efficiently on average. Thus, we consider that it is better to concentrate on improving GA's performance for path testing than to explore other meta-heuristic techniques.

3.2 GA-based Test Data Generation

In reviewing the literature, a set of assessment attributes is required to compare different approaches [56]: objective(s), fitness function, benchmark programs (including complexity measure {e.g. cyclomatic complexity, number of branches, and nesting level for selection and loop}, input {domain and representation}, and constraint{s}) and genetic operators.

Selecting what kind of information to use to guide the search has always been the main issue in applying GA for testing, because it will be the essence of its fitness function. Finding the best combination of parameter values can also be interesting and challenging, because the parameter values can make a significant difference to the overall performance of the search technique.

The fitness function typically consists of branch or predicate distance (BD) [8] and approximation level (AL) [61, 62]. Predicate distance is a quantifiable difference in the branch statement between a target path and the actual path taken. The approximation level shows the number of matched branches between a target path and the actual path traversed until they deviate from each other. Ahmed and Hermadi [20] employed GA to satisfy path testing. Approximation level and branch distance were used as components for the fitness function. Ahmed and Hermadi conducted experiments with several fitness functions. The best fitness function was found to be the following: normalized BD and AL, path-wise traversal for AL computation, no weighted BD and AL, and relative fitness values to the population. A separate contribution of their work was to aim to minimize the number of fitness function evaluations by generating a set of test data that can cover multiple target paths in each generation.

In 2009, Blanco *et al.* [63] initiated scatter search (SS), another evolutionary algorithm, to generate test data automatically. They reasoned for SS that it solves the same class of combinatorial optimization problems and also has the same control flow graph representation as software test data generation. Their objective was to efficiently construct a small set of test data for branch coverage. Two versions of the test data generator were developed, in which one is hybridized with local search. Thirteen benchmark programs were selected to validate and compare both generators in terms of number of test data generated and time consumed. They reported that SS does not always outperform other work reviewed in the paper; however it does on average. Additionally, hybridizing it with local search accelerates its efficiency.

In 2008, Sagarna and Yao [64] proposed an approach for generating test data using multiobjective evolution, which considers branch coverage testing and penalty constraints at the same time. It made use of branch distance and approximation level as its fitness function, as described in [33, 61, 62].

Alba and Chicano [26] analyzed the application of parallel and sequential evolutionary algorithms for condition coverage test data generation. A study of the influence of several parameters in their proposed approach is an additional aspect of their work. Decentralized evolutionary strategy (ES) and decentralized GA were proposed. Twelve test programs were used to test the approaches. The results showed that there was no significant difference between the decentralized versions and the panmictic versions, in terms of coverage or effort.

In 2007, Harman *et al.* [65] developed weighted and Pareto GAs for multi-

objective test data generation. Their objectives were to satisfy branch coverage and to minimize dynamic memory allocation. Both versions of GA's fitness functions were composed from branch distance and approximation level. They used the IGUANA (Input Generation Using Automated Novel Algorithms) framework, developed by McMinn [66] using NSGA II [67]. Weighted GA uses stochastic universal sampling as its selection method, while Pareto GA uses elitist selection and reinsertion strategy. The results revealed that neither algorithm outperformed the other on all test data. However, the work has shown a promising result for applying a multi-objective EA approach to generating test data.

In 2007, Yoo and Harman [68] applied multi-objective EA for selecting test cases in regression testing, aiming at minimizing or avoiding retest-all method, i.e. re-running all test cases whenever new changes are introduced. Their objectives were to meet the following criteria: code coverage, past fault-detection history, and execution cost. The work was implemented using NSGA-II [67] and its variant vNSGA-II.

The next work in this review was on testing coverage using hybrid EA. In 2004, Ferreira and Vergilio [69] worked on code/decision coverage using random, GA, and hybrid GA. They reported that hybrid GA is more effective, with a lower average running time.

In the same year, McMinn and Holcombe [70] merged a chaining approach (CA) into GA's fitness function, such that inherent data dependencies are taken into consideration. This was meant to guide the search directly into a potential unexplored input domain. Later, McMinn and Holcombe extended the previous work [71] of Baresel *et al.*, that is able to generate input sequence

[72], and their own work that addresses difficulties with internal variables, using hybrid GA with CA [73, 74]. The CA was initially introduced in 1996 by Ferguson and Korel [75, 76, 77]. The basic idea is to set the current state of a test program to a certain condition that meets an individual target structure, e.g. statement or branch coverage. This is done by selecting and executing a sequence of statements that assign boolean flags, enumerations, and counters to specific values.

In 2006, Wappler and Wegener [78] developed hybrid genetic programming (GP) for branch unit testing of object-oriented software. The results show hybrid GP is feasible and can outperform random branch testing. They also reported that further investigation of node distance function being used in its fitness function is required.

In 2007, Sofokleous and Andreou [79] proposed a dynamic software testing framework using hybrid GA, which was designed to satisfy selected testing coverage criteria.

In 2008, Arcuri and Yao [80] developed a memetic algorithm (MA) for generating test data for object-oriented software. Branch distance was used as the guiding function. Hill climbing (HC), GA, and MA were compared, with MA found to outperform the other two algorithms.

Table 3.1 lists all the work on testing coverage test data generation, in chronological order.

Table 3.1: Testing Coverage based on EA

Year	**Reference**	**Coverage**	**Technique(s)**
1994	Pei [13]	Path	GA
1995	Jones [81]	Branch	GA
	Jones [82]	Branch	GA
	Roper [39]	Code (Statement)	GA
	Sthamer [46]	Branch	GA
1996	Jones [50]	Branch	GA
1997	Roper [83]	Branch	GA
1998	Jones [84]	Branch	GA
1999	Pargas [37]	Branch	GA
2000	Bueno [14]	Path	GA
	Lin [5]	Path	GA
2001	Bueno [15]	Path	GA
	Lin [85]	Branch	GA
	Wegener [61]	Branch	EA
2002	Baresel [62]	Statement	EA
	Bueno [16]	Path	GA
2003	Diaz [86]	Branch	TS
	Hermadi [17]	Path	GA
2004	Ferreira [69]	Decision	GA, Hybrid GA
	Hermadi [56]	Path	GA
	Mansour [18]	Path	GA, SA
	McMinn [70]	Branch	Hybrid EA with CA (Chaining Approach)
2005	Girgis [19]	Path	GA
	Hierons [87]	Branch	EA
	Korel [88]	Branch	HC/AVM (Alternating Variable Method)
	Liu [89]	Code	ACO, GA
	McMinn [90]	Code	EA
	McMinn [71]	Branch	Hybrid EA with CA

Continued on next page

Table 3.1 – continued from previous page

Year	Reference	Coverage	Technique(s)
	Sagarna [91]	Branch	EDA
	Wappler [92]	Statement, Branch, Condition	EA
	Xie [93]	Code	GA, SA
	Xie [94]	Statement, Branch	GA
2006	Alshraideh [95]	Branch	GA
	McMinn [96]	Branch	EA
	Sagarna [97]	Branch	SS, EDA
	Seesing [98]	Branch	GP
	Seesing [99]	Branch	GP
	Wang [100]	Branch	GA
	Wappler [101]	Branch	GP
	Wappler [78]	Branch	GA, Hybrid GP
	Levin [102]	Branch	GA
2007	Blanco [103]	Branch	Scatter Search
	Harman [65]	Branch & dynamic memory allocation	NSGA-II
	Liaskos [104]	Data flow (d-u)	GA
	Liaskos [105]	Data flow (d-u)	GA, AIS
	Sagarna [106]	Branch	EDA
	Sagarna [107]	Branch	EDA, SS
	Sofokleous [79]	Selected coverage	Hybrid GA
2008	Arcuri [80]	Unit & minimize the length of the test sequences	RS, HC, SA, GA, MA (Memetic Algorithm)
	Chen [21]	Path	GA
	Gupta [108]	Program	GA
	Wang [109]	Branch	EA
	Makai [110]	Branch	GA
	Lefticaru [22]	Path	SA, GA, PSO
	Harman [111]	Branch	EA
2009	McMinn [112]	Branch	GA, HC

Continued on next page

Table 3.1 – continued from previous page

Year	Reference	Coverage	Technique(s)
2010	Hermadi [23]	Path	GA
	Li [24]	Path	GPSMA
2011	Alshraideh [113]	Branch	Multi-pop GA

Table 3.1 permits some observations on trends in software coverage testing, hybrid methods, multi-objective approaches, and non-GA methods. EA based coverage testing started in 1994, and its numbers increased since 2004. The most popular coverage testing is branch coverage, followed by path coverage. One of the approaches uses multiobjective GA, such as NSGA-II. Some hybrid methods were proposed since 2004, such as hybrid GA, hybrid EA with chaining approach (CA), memetic algorithm (MA), hybrid genetic programming (GP), and Genetic-Particle Swarm Mixed Algorithm (GPSMA). Some research work also compared GA to non-GA methods for the same testing coverage criteria. They are Tabu search (TS), simulated annealing (SA), hill climbing (HC), alternating variable method (AVM), ant colony optimization (ACO), estimation of distribution algorithm (EDA), scatter search (SS), artificial immune system (AIS), random search (RS), particle swarm optimization (PSO), and BLAST.

The path coverage literature started with Pei *et. al.* in 1994 [13]. They proposed a dynamic path coverage, which had been mostly developed using symbolic execution in their era. They considered symbolic execution to be impractical, as the complexity of a set of predicate equations is exponential. Pei *et. al.* [13] developed a test data generator for path testing, using a genetic

algorithm with two variants of fitness functions. The first fitness function was based on the number of matching branches, while the second was based on the branch predicate values, which is more sensitive. Only one test program was used to validate the approach. It takes an array of integer numbers and returns minimum and maximum values of the array. The 21 target paths included 13 feasible paths and 8 infeasible paths. The approach was able to cover all 13 feasible paths.

Jones *et. al.* [50] presented a GA-based branch coverage test data generator. Their fitness function made use of weighted Hamming distance to branch predicate values. They used unrolled control flow graph of a test program such that it is acyclic. Six small programs were used to test the approach.

In 2000, Lin and Yeh [5] extended Jones *et al.*'s work [50] from branch coverage to path coverage. The ordinary (weighted) Hamming distance was extended to handle different ordering of target paths that have the same branches. The fitness function is called SIMILARITY, which computes similar items with respect to their ordering within two different paths between actual executed path and the target path. Only one program was used to test the approach, i.e. simple triangle classifier. They reported that the approach outperformed random search.

Bueno *et al.* [14, 15, 16] proposed an approach that utilizes control and data flow dynamic information to achieve path coverage testing using GA. In addition, the work also tackled the detection of infeasible paths by monitoring the progress of evolutionary search. The fitness function was formulated by number of coincidence branches and the normalized branch predicate value

at which the actual executed path starts to deviate from the target path. Six small test programs were used to validate the approach, with 10 repetitions each to minimize random variations. Two execution modes were used, i.e. one with initialized population and the other with a random initial population. The experiment results were promising.

In 2003, Hermadi and Ahmed [17] presented evolutionary test data generation for path testing using multiple paths. Prior to this work, almost all of the evolutionary test data generators only sought to cover a single target path at a time. The fitness function used the number of matching branches and branch predicate values using Korel's fitness function [8]. It also considered path traversal techniques, neighbourhood influence, weighting, and normalization. Three small programs were used to validate the approach: minimum-maximum finder, triangle classifier, and a combination of both of them. Results were more effective and efficient by tackling multiple paths at a time.

Mansour and Salame [18] compared simulated annealing (SA), genetic algorithm (GA), and Korel's algorithm (KA) for path testing using weighted Hamming distance as the objective function. Eight programs were selected to test the approach. The empirical results showed that SA and GA were able to cover more paths than KA, and SA was slightly better than GA. In terms of time complexity, KA was the fastest and GA was faster than SA.

In 2008, Ahmed and Hermadi [20] extended Hermadi and Ahmed's work of 2003 [17]. The extensions were adding a rewarding scheme and using a more efficient test data generator. A total of 32 fitness function variations were tested empirically and analysed to determine which was the best. There

were 7 test programs used in the experiments. The results demonstrated that the approach was better compared to other existing work.

In the same year, Chen and Zhong [21] developed a multi-population genetic algorithm (MPGA) for path testing. This work has been improving GA-based path testing as described in Section 1.4. The work reported that the proposed approach outperformed a simple genetic algorithm based approach, using the triangle classifier as the test program.

To summarize: several EAs have been studied for test data generation; some observations have been made here about how GA compares with other methods on different types of problems, and hybridization of GA with local search has been suggested to improve GA's performance; most work has been on weaker testing criteria than path testing; and research with multi-objective GA is in its early stages.

3.3 Stopping Criteria for Evolutionary Computation

In the real world, most problems are computationally expensive and constrained. Thus an evolutionary algorithm should be stopped as soon as the pseudo-optimal solution has been detected, because the actual optimum is unknown or the evolutionary process is not worth continuing because no further improvement is likely.

Up to now, the most used stopping criteria are objective value, fitness value, number of generations, time elapsed, number of stall generations, and

stall time. Most of these involve arbitrary decisions on when to stop searching. Thus, adaptive and real-time stopping criteria are urgently required that make computation efficient while being effective by delivering (near) optimal solutions.

In general, there are three classes of adaptive stopping criteria for evolutionary algorithms [114]: improvement-based criteria, movement-based criteria, and distribution-based criteria. Improvement-based criteria monitor improvement of the objective function value: if the improvement decreases to a small value over several generations then it can be considered that the search has converged. Movement-based criteria consider convergence in the search space: at early generations, individuals were scattered in the search space, and towards the end of evolutionary process these individuals normally converge to one spot. Distribution-based criteria are similar to movement-based criteria, and use distance to measure the distribution of individuals. For example, in 2002, Bergh [115] used no-further-improvement stopping criteria and distribution-criterion to halt his global particle swarm optimizer.

In 1996, Aytug and Koehler [116] studied convergence behaviour or stopping criteria for finite length genetic algorithms. The criteria will enable genetic algorithm to set the maximum number of generations (or number of fitness function executions) given a confidence that it has seen all the possible individual strings irrespective of its starting state. Analytically proven, the result was promising.

Greenhalgh and Marshall [117] described convergence properties for genetic algorithms using Markov Chain Model in 2000. Their approach specified the upper bound on the number of generations, and they proved analyt-

ically that the upper bound is the lowest so far.

In 2008, Zielinski and Laur [114] experimented with some improvement-, movement-, and distribution-based criteria to stop differential evolution algorithms. They used 16 test functions, and found that distribution-based criteria were best in terms of convergence rate and computation time.

In 2009, Trautmann *et al.* [118] proposed two approaches, i.e. offline and online modes, using statistical methods to detect convergence of multi-objective evolutionary algorithms. Offline mode analyses performance indicators of some repeating runs with increasing number of generations using statistical tools, and decides what the optimal number of generations is. The online mode monitors variance of the performance indicators. If they fall below given thresholds, or the overall trend indicates stagnation, the evolutionary process stops. The experimental results showed that both modes are effective and efficient by avoiding maximum loss of computation time.

In 2010, Studniarski [119] found the maximum number of generations at which to stop evolutionary algorithms, using a Markov Chain approach. The proposed approach uses only some properties of mutation. Studniarski analytically proved that the approach works.

Up to now, no stopping criteria have used the analogy of software reliability modelling. Software reliabilty modelling uses the numbers of errors recorded over some period of software running time to calculate the probability of error occurrence, and uses this probability to predict the future occurrence of errors. In software testing context, one of the main issues is to gain confidence that the software will function correctly at any running time; this is what software reliability is all about. So, in this research, the analogy

of a software reliability growth model is adopted to propose stopping criteria that stop GA as soon as the tester has reached a desired level of confidence that further searching will not cover more target paths.

3.4 Software Reliability Growth Models

Many software reliability growth models are available nowadays. Some criteria for selecting a good model are: it is able to predict future failure behaviour; it is able to produce meaningful results; it is simple, widely applicable, and based on sound assumptions.

In general, software reliability models are classified into two schemes [58]. First, the finite failure category model, which consists of exponential, weibull, and pareto; this model fits Poisson, Binomial, and other type distributions. Second, the infinite failure category model, which consists of geometric, inverse linear, inverse polynomial, and power models; this fits type T1, T2, T3, and Poisson distributions as defined in [58].

The Logarithmic-Poisson model described in [58] is the most appropriate for this thesis work. This is because it can predict future paths finding, it produces a prediction number that is interpretable as number of paths covered, and its assumptions match the type of data available in this research.

A variety of statistical techniques exist that aim to do the same thing (predict logarithmically decaying phenomena) and are similar in data behaviors with the proposed approach in the thesis, i.e. finite failures category models. These include Musa Execution Time [120], Goel-Okumoto NHPP (non homogeneous Poisson process) [121, 122], Moranda Geometric Poisson

[123], Shneidewind [124], and Crow [125]. Most of these models agree that failure intensity is equally reduced as each failure is exercised, and correction is made.

Musa's Execution Time Model [120] was introduced in 1974. It makes use of exponential Poisson distribution to predict software reliability growth. Musa's 1984 execution time version [58] proposed basic and logarithmic execution time models. This version is categorized in the finite failures category which takes into account failure intensity over time. The basic model describes that failure intensity reduction is constant over time, while the logarithmic one is considered as a logarithmic Poisson process.

In 1979, Goel and Okumoto [121] proposed a stochastic model to describe software failure manifestation using NHPP. The model is tested using data collected from Naval Tactical Data System (NTDS) and showed good fitting in describing the failure phenomenon.

In 1985, Goel [122] surveyed the proposed analytical models to approach software reliability measurement for the previous 15 years. For each model, the survey provided an overview, critical analysis of the underlying assumptions, limitations, and applicability. Four classes of analytical models were presented, based on the failure history of software and the nature of the failure process: times between failures models, failure count models, fault seeding models, and input domain based models.

Moranda [123] proposed a Geometric Poisson model to measure reliability growth of software in 1975. The model is classified into finite failures category models that use exponential Poisson approach by Musa [58] in 1984. The model estimate risk rate of the failure interval decreases in a geometric

fashion.

Shneidewind [124] presented an estimation method to supply values of parameters of a non-homogeneous Poisson process to model failure occurrences in software. The estimation process used a combination of maximum likelihood and weighted least squares methods. The software error data were taken from Naval Tactical Data System. The test produced promising results. This model falls into the category of finite failures, which employ exponential Poisson approach according to Musa's classification [58] in 1984.

3.5 Infeasible Path Detection

In 1994, Jasper *et. al* [126] proposed new presentations combined with automated theorem proving to overcome the problems of exponential number of paths explosion and feasible path determination. Offutt *et. al* [127] developed mathematical constraints to automatically detect infeasible paths. Gustafsson *et. al* [128] proposed 3 algorithms that are inclusive to one another for detecting infeasibility of nodes, pairs of nodes, and paths, i.e. infeasible nodes are covered by the infeasible pairs of nodes, and the infeasible paths cover the infeasible pairs of nodes. Souter and Pollock [129] demonstrated an approach to identify type infeasibility of call chains in object-oriented programs. Most recently, Balakrishnan [130] proposed static analysis of an abstract interpretation of a program to infer infeasible paths, and a syntactic language refinement technique that is able to automatically exclude semantically infeasible paths from a program during static analysis.

The following are dynamic approaches. In 2000, Bueno and Jino [14]

monitored the improvement of best fitness values of a target path over generations, using this as an indication of whether the path is likely infeasible or not. Zhuang *et. al* [131] used correlation between branches to identify infeasible paths for memory anomaly detection. Suhendra *et. al* [132] detected path infeasibility in a directed acyclic graph but only tracked conflicting pairs of branch-assignment or branch-branch. Ngo *et. al* [133] determined infeasibility by empirical correlation evidence between some conditional statements along the path. Yan *et. al* [134] presented a method to generate feasible paths for basis path testing, based on path linear independence and the minimal subset of feasible paths that satisfies test coverage. Ju *et. al* [135] shared similar ideas with Ngo *et. al* [133] and Suhendra *et. al* [132]; they employed 4 infeasible path patterns by observing conflicting pairs, i.e. pairs of assignment-branch or branch-branch. In 2010 Delahaye *et. al* [136] proposed a method to generalize an infeasible path automaton.

3.6 Test Programs

This section summarizes the test programs that have appeared in the literature on test coverage using evolutionary algorithms.

Table 3.2 presents the test programs, their source, and the problem they address.

Table 3.2: Collection of Test Programs

No	Test Program	Ref.(s)	Description
1	triangle ahmed	[137, 50, 37, 5, 14, 41, 15, 16, 10, 11, 17, 26, 20, 21, 64]	determination of triangle types: equilateral, isosceles, scalene, not triangle
2	minimaxi ahmed	[13, 17, 20]	determination of minimum and maximum numbers from an array of numbers
3	insertion ahmed	[26, 20]	sorting an array of numbers using insertion sort
4	bisection ahmed	[138, 37, 20, 63]	calculation of a number square root using bisection method
5	binary ahmed	[50, 20]	searching a key (number) from an array of numbers by returning the index if it is found and nothing if is not found
6	bubble ahmed	[37, 20, 25]	sorting an array of numbers using bubble sort
7	gcd ahmed	[26, 20]	calculation of greatest common divisor between two integers
8	remainder ahmed	[50, 20, 64, 63]	calculation of a division remainder
9	mmTriangle	[17, 20]	artificial program by merging minimaxi (TP 10) with triangle (TP 1) in serial
10	triangle mansour	[18]	classification of triangle types: scalene, isosceles, right, iso-right, equilateral
11	expint rapps	[139]	accepts an integer and a float variable for exponentiation

Continued on next page

Table 3.2 – continued from previous page

No	Test Program	Ref.(s)	Description
12	quotient gallagher	[140]	calculation of the quotient and the remainder of the division of two positive integers
13	tritype bueno	[16]	triangle classifier
14	expint bueno	[34, 14, 15, 16]	accepts an integer and a float variable for exponentiation
15	quotient bueno	[14, 15, 16]	calculation of the quotient and the remainder of the division of two positive integers
16	strcomp bueno	[14, 15, 16]	comparing 3 chars with a 5-char string
17	floatcomp bueno	[14, 15, 16]	performing simple computation on three floating point variables
18	find bueno	[14, 15, 16]	partially sort an array
19	bubble gong	[25]	sorting an array of numbers using bubble sort
20	flex gong	[25, 141]	a unix utility taken from GNU site
21	space gong	[25, 141]	to read a file that contains several ADL statements and check the contents of the file for adherence to the ADL grammar and specific consistency rules
22	linear search	[50]	using linear search on arrays of integers and characters
23	quicksort	[50, 26]	quicksort application on arrays of integers and characters
24	shell	[26]	sorting by shell method
25	triangle wegener	[61, 63]	to classify a triangle either equilateral, isosceles, orthogonal, obtuse angle, or not-triangle
26	triangle myers	[7, 63]	to determine the types of triangle: equilateral, isosceles, scalene, or not-triangle

Continued on next page

Table 3.2 – continued from previous page

No	Test Program	Ref.(s)	Description
27	triangle michael	[41, 63]	triangle classification
28	triangle sthamer	[46, 63]	more complete triangle classification that also checks the right angles of the triangle
29	remainder sthamer	[46, 63]	to calculate the remainder of a division
30	grading	[18]	computing the letter grade from a numeric score
31	roll dice-1	[18]	is to read sum1 of die value and determine a game status
32	roll dice-2	[18]	is to read sum1 and sum2 of die values and determine a game status
33	interview	[18]	is to read subject, college, and age, then decide whether to interview a candidate
34	answer	[18]	is to read a value and evaluate an expression, then classify the result as correct or too high
35	guess	[18]	is to guess a number then check whether it is the magic number
36	evaluate	[18]	is to read two numbers and evaluate an expression, and then classify the result as too high or too low
38	elements classification	[142, 37, 14, 15, 16]	placing all elements of an integers array less than given key element (by supplying its index) on the left and greater than or equal on the right of it
39	quadratic	[138, 50, 63]	solution of quadratic equation
40	four balls	[37]	determines the weight of the balls relative to each other out of four given integers representing the weight of balls

Continued on next page

Table 3.2 – continued from previous page

No	Test Program	Ref.(s)	Description
41	middle value	[37]	determines the middle value out of three given integers
42	asin	[11]	typical C library function to calculate the arcsin or arccos of a double type number
43	atof	[61, 11, 64, 63]	typical C library function to convert a string to its corresponding float value
44	powi	[11]	a function that raise a float to integer, i.e. $float^{int}$
45	incbet	[11]	a larger test function that calculates the incomplete beta-integral out of three given floats argument
46	line coverage	[143, 63]	to determine the position of a line with respect to a rectangle
47	sed	[25, 141]	Linux patches
48	netflow	[61, 26]	network optimization
49	calday	[26, 63]	to calculate the day of the week
50	crc	[26]	cyclic redundant code
51	heapsort	[26]	sorting by heapsort
52	select	[26]	to select the k^{th} element of unordered list
53	bessel	[26]	Bessel J_n and Y_n functions
54	simulated annealing	[26]	simulated annealing method
55	complex branch	[63]	artificial program that contains several difficult branches
56	number of days	[63]	to calculate the number of days between two dates
57	quadratic sthamer	[46, 63]	to determine the type of quadratic equation roots: real and unequal, real and equal, or complex

The average number of test programs (NoTP) used in the most relevant literature in testing coverage using EAs is 6. Table 3.3 describes how many test programs are used in each main reference.

Table 3.3: Number of Test Programs

Year	**Reference**	**Coverage**	**NoTP**
1994	Pei [13]	Path	1
1996	Jones [50]	Branch	6
1999	Pargas [37]	Branch	6
2000	Lin [5]	Path	1
	Bueno [14]	Path	6
2001	Bueno [15]	Path	6
2002	Bueno [16]	Path	6
	Wegener [10]	Branch	5
2003	Hermadi [17]	Path	3
2004	Mansour [18]	Path	8
2007	Alba [26]	Condition	12
2008	Ahmed [20]	Path	9
	Chen [21]	Path	1
	Sagarna [64]	Branch	3
2009	Blanco [63]	Branch	13
2010	Hermadi [23]	Path	12
2011	Gong [25]	Path	4

Table 3.4 shows the most used test programs in the literature. The two most used test programs have no loops, and all their target paths are feasible. The others are more difficult for path coverage, by involving one or more of loops, variable-length input, complicated branch statements, and infeasible target paths.

Table 3.4: The Most Used Test Programs

Test Program(s)	**Used** (times)
triangle	15
triangle variants	11
elements classification	5
bisection, quotient, expint, atof	4
minimaxi, bubble, remainder, floatcomp, quadratic	3

Table 3.5 lists which test programs have been used in research with which testing coverage criteria. The acronyms P, B, and C stand for path, branch, and code coverage respectively. Two test programs used in almost all coverage research are triangle and remainder. Triangle has no loops and no infeasible paths, while Remainder has one loop and at least one infeasible path if a limit is placed on the number of iterations.

Test programs that have been used in research on two or more coverage criteria are bubble, insertion, binary, bisection, gcd, elements classification, quicksort, and calday.

Table 3.5: Test Program based on the Testing Coverage

No	**Test Program**	**Coverage**
1	triangle ahmed	P, B, C
2	minimaxi ahmed	P
3	insertion ahmed	P, C
4	bisection ahmed	P, B
5	binary ahmed	P, B
6	bubble ahmed	P, B
7	gcd ahmed	B, C
8	remainder ahmed	P, B, C

Continued on next page

Table 3.5 – continued from previous page

No	Test Program	Coverage
9	mmTriangle	P
10	triangle mansour	P
11	expint rapps	P
12	quotient gallagher	P
13	tritype bueno	P
14	expint bueno	P
15	quotient bueno	P
16	strcomp bueno	P
17	floatcomp bueno	P
18	find bueno	P
19	bubble gong	P
20	flex gong	P
21	space gong	P
22	linear search	B
23	quicksort	B, C
24	shell	C
25	triangle wegener	B, C
26	triangle myers	B, C
27	triangle michael	B, C
28	triangle sthamer	B, C
29	remainder sthamer	B, C
30	grading	P
31	roll dice-1	P
32	roll dice-2	P
33	interview	P
34	answer	P
35	guess	P
36	evaluate	P
38	elements classification	P, B
39	quadratic	B
40	four balls	B

Continued on next page

Table 3.5 – continued from previous page

No	Test Program	Coverage
41	middle value	B
42	asin	B
43	atof	B
44	powi	B
45	incbet	B
46	line coverage	B
47	sed	P
48	netflow	C
49	calday	B, C
50	crc	C
51	heapsort	C
52	select	C
53	bessel	C
54	simulated annealing	C
55	complex branch	B
56	number of days	B
57	quadratic sthamer	B

Test programs do not add information if their relevant characteristics are the same as other test programs. The 57 test programs identified above include some that are redundant, in that their expression structure or logic structure is equivalent to other programs. Section 4.1 presents a classification scheme, based on expression structure and logic structure, which is used in this thesis to select a non-redundant subset of test programs.

The average number of test programs used in the literature for EA based coverage testing is 6. As shown in Table 3.3, more test programs were used in later years. In research, more test programs means more general results can

be achieved. So, most of the industrial research involves more test programs and more challenges.

The two most used test programs are Triangle and Remainder. They are popular due to their representativeness, in terms of selection statements, having loops or not, and path infeasibility. They are used in this thesis, along with others that include more loops, variable-length input, complicated branch statements, presence of infeasible paths, and real-time processing.

Chapter 4

Experimental Design

This chapter describes a classification scheme for test programs, selection of test programs, preparation for using a test program, and operation of the test data generation system.

4.1 Selection of Test Programs

4.1.1 Classification Scheme for Test Programs

Numerous test programs are used in the literature. It is unclear how they were chosen for use in testing the proposed approaches. It is also not clear that test programs differ from each other in relevant ways, for the purpose of validating testing approaches. In other words, how do we know that a given new test program will provide new information, and is not essentially equivalent to another test program that has already been studied?

For this purpose, we propose a classification scheme for test programs.

The classification can be used to select test programs that cover a range of relevant characteristics; also to recognize if a new test program fills in a gap or just replicates existing test programs. The classification scheme is used when selecting test programs for study in this thesis.

Two aspects to the classification are control structures (structure classification) and expression structures (expression classification) in conditional expressions. The control structures are directly related to the CFG while the expression structures are related to the various types of decision coverage.

Two types of control structures are selection (**S**) and iteration (**IP**). A selection could be IF, IF-ELSE, or SWITCH statement. An iteration could be FOR, WHILE, or DO-WHILE loop. These cover the control structures typically found in most programming languages.

The classification considers sequence vs. nesting because these lead to different paths in a program. Further detail on the paths identification is presented in Section 4.2.2.3.

In principle, there is no limit to number of selections or loops in a program. In order to bound the set of possibilities, the numbers for consideration are limited to 4 of each. Possible values for **S** are 0, 1, 2, and 3 (representing 3 or more) selections, and for **IP** are 0, 1, 2, and 3 (representing 3 or more) loops, respectively. The trivial case (0 loops and 0 selections) is not included in the classification because it means only one path, and there is no need to search for test data. In total, the structure classification has 101 classes (the detail is in Appendix A.1).

For example, the control structure of tA2008 contains 3 increasingly

nested IF-THEN-ELSE selections (source code in Section 4.2.2.1). It is classified as class **S03NIII** in the structure classification (in Appendix A.1). The **S03NIII** code means **S** for structure classification, **03** for 0 loops and 3 selections, **NIII** for 3 increasingly nested selections.

In the expression classification, a simple expression can be an arithmetic (**A**), relational (**R**) expression, or **A** and **R**, i.e. an operator with its corresponding operand(s). Simple expressions can be combined using logical connectors AND (**N**) or OR (**O**). More complex expressions (compound expression) can be constructed by using one or more logical operators connecting two or more simple expressions. The details of the operators are shown in Table A.4.

The complexity measure of each program is cyclomatic complexity (CC) [144]. Two versions of CC are proposed for the purpose of these classifications: CCL (the lower bound of CC) and CCU (the upper bound of CC). CCL measures the number of paths in the control flow graph, and is simply equal to CC, while CCU also takes into account the expression structures (or compound statement with logical connectors AND and OR) in the selections or iterations, which is relevant to the classification of expression structures. For example, consider a test program with an IF-THEN statement and a compound statement inside the IF (or condition) part with an AND logical connector: The CC for this program is 2, while its CCL is 3 because the AND operator will break down the logical flow into two possibilities. Thus, this measure is very sensitive in detecting implicit logical flow in path testing.

Theoretically, the number of logical connectors in an expression has no limit. For the classification purpose, we limit the set of possible logical

connectors to 2, which means a maximum of 3 simple expressions can be present. A test program with more than 2 logical connectors in an expression would be classified the same as a program with 2 logical connectors.

The order of the connector matters, because it leads to exercising different paths (breakdown of decisions). For example, **N-O** and **O-N** are considered two different classes. Connector **N** does not consider the second operand value if the first one is already FALSE. Similarly, connector **O** does not consider the second operand if the first one is already TRUE. In total, the number of classes is 24 (see Appendix A.2 for the detail).

A program can fall into several classes in the expression classification because it can have many expressions with different types and/or number of logical connectors. For example, tA2008 is included in the following classes **E1R**, **E3NNRRR**, and **E3NNIII**. The **E3NNRRR** code means **E** for the expression classification, **3** for 3 simple expressions, **NN** for **N-N** logical operators, and **RRR** for 3 simple relational expressions.

Details of test programs and their classification are described in Table A.2 and Table A.5 for the structure classification and the expression classification, respectively. The first 28 test programs in Table 3.2 are available in the classification tables. The remainder could not be classified because we do not have access to their source code.

4.1.2 Choice of Test Programs

The test programs used in this thesis are selected based on the structure classification and the expression classification presented in the previous Sec-

tion 4.1.1. They are selected from different classes from both classification schemes and variations of the target paths.

In order to be able to compare the proposed approach properly with other path testing approaches, every test program we have found in the path testing literature was classified according to both classification schemes. Unfortunately, the number of these test programs and their variations are not enough to fill in all the classes in either of the classification tables at the time the experiments were conducted. In addition, some of the test programs fall into the same classes as others in one or both classification schemes, and this condition makes classification tables sparser. In time, we hope that these empty classes could be filled in with new test programs.

The purpose of the classification schemes is to classify test programs based on their logical characteristics, which drive the logical flow inside the programs while they are being executed. In choosing test programs for this research, all possible logical flow drivers, such as looping, selection, and expression, have been considered to some extent, while the classification schemes consider more, such as their numbers (or occurrences) and/or repetition structures (serial or nested), and complexity of the branch expression/statement along a logical path.

The test programs studied in this research are generally one page of code or less. When modern software may contain millions of lines of code, it may be questioned whether such small test programs are relevant. The point to remember is that path testing applies at the level of single code modules, or methods. Thus the question is whether these test programs are representative of methods in typical software.

In 2013, Fraser and Arcuri [145, 146, 147, 148] selected 100 open source software projects (called SF100 corpus) randomly from SourceForge (sourceforge.net) for the purpose of object oriented testing. The SF100 corpus contains 8,784 classes and 291,639 bytecode branches [145]. On average, it has 87.84 classes per project, and 33.20 branches per class [145]. What we need to know is not branches per class, but branches per method. This information is not published for the SF100 corpus, but other evidence suggests that the number of methods (NOM) in a typical class is around six [149, 150]. Lanza *et al.* [149] stated that the average NOM per class is 6.5, which is the average between the lower and the upper NOM per class. Haug *et al.* [150] described that the average NOM per class is between 5.67 and 5.97 for two projects. Averages of six methods per class and 33 branches per class suggests that the average number of branches in a method (or test program) is 5.5. Our test programs average 7 branches, which is 27% more. Further, in 2014 Fraser and Arcuri [151] and Fraser *et al.* [152] conducted an experiment using an extended SF110 corpus, which consists of 110 projects, with 23,886 classes, more than 800,000 bytecode level branches, and 6.6 millions of lines of code. Methods in this corpus have an average CC value of 2.63, while ours is 5.4 (recall Table 4.3), which is more than twice that of SF110.

In summary, firstly, path testing applies at the level of single code modules or methods. Even a large program would be broken into modules, and the modules used for testing here are typical in size compared to advice when writing code modules now (no more than a page). Thus, the test programs are not unrepresentative. Secondly, although the number of test programs is limited, i.e. limited instance space, the nature of all typical combination of programs' characteristics (or combination of path control flows) have been

well addressed. As long as the fitness function represents relevant building blocks, i.e. considering Struct and Expr, the search algorithm will perform well (proportional to the search space) regardless of the increase of the test program's complexity [153, 154]. So, the limited instance space does not affect significantly the conclusion drawn.

Table 4.1 lists all 21 test programs which are selected from the collection mentioned in Chapter 3.6. It shows **Test Program** name, (abbreviated) **Name**, **Total** number of target paths, and number of feasible paths (**Feas.**) included in **Total**.

These test programs are selected from each class in both the structure and the expression classifications. If some of them belong the same class, they have different number of target paths with respect to feasible and infeasible paths. The other 7 classifiable programs were not used as test programs, due to their redundancy.

The 21 test programs cover only 11.88% (or 12) of the structure classes and 37.5% (or 9) of the expression classes. In the structure classification, 7 classes have 1 test program, 3 classes have 2 test programs, 1 class has 3 test programs, and 1 class has 5 test programs. In the expression classification, all the test programs (100%) have at least one simple relational statement, 7 test programs (33.33%) have one logical connector, and 3 test programs (14.29%) have two logical connectors.

For identification purpose, a test program is named after its (real) name followed by its author's (last) name and year in which it is published, respectively. For example, *triangleAhmed2008* means its real program name is *triangl*e, which appears in a publication whose author's name is *Ahmed* and

was published in *2008*. The abbreviated version **Name** will be used for the rest of the thesis.

Table 4.1: List of Test Programs

No	Test Program	Name	No. of Paths	
			Total	Feas.
1	triangleAhmed2008	tA2008	4	4
2	minimaxiAhmed2008	mmA2008	13	13
3	insertionAhmed2008	iA2008	6	5
4	bisectionAhmed2008	bisA2008	9	6
5	binaryAhmed2008	binA2008	7	7
6	bubbleAhmed2008	bubA2008	15	4
7	gcdAhmed2008	gA2008	8	5
8	remainderAhmed2008	rA2008	5	4
9	mmTriangleAhmed2008	mtA2008	52	20
10	triangleMansour2004	tM2004	8	7
11	expintRapps1985	eiR1985	12	3
12	quotientGallagher1997	qG1997	21	4
13	tritypeBueno2002	ttB2002	8	8
14	expintBueno2002	eiB2002	31	8
15	quotientBueno2002	qB2002	27	10
16	strcompBueno2002	scB2002	15	4
17	floatcompBueno2002	fcB2002	5	5
18	findBueno2002	fB2002	32	8
19	bubbleGong2011	bG2011	20	11
20	flexGong2011	fG2011	30	30
21	spaceGong2011	sG2011	32	32

These names are listed in Table 4.1. Comprehensive details of two of the test programs are presented in this chapter as examples, i.e. tA2008 and mmA2008. The rest are provided in Appendix B.

Input description for each test program is shown in Table 4.2. The

description covers information on input **Type**, **Size**, **Range**, and **Space**. **Type** can be integer, real, float, character, or string (a set of characters). For example, *int+* means positive integer input. **Size** is the input size or number of input components. For variable length input, **Low** and **Upp** indicates its lower and upper limits, respectively. Each input component (or allele) has its range of values, which is stated under column **Range**. The size of the input space is written in the column **Space**. For example, if a program requires an int+ input with fix length 3, i.e. 3 positive integers all the time, that ranges between 0 and 200 each then its input space size is equal to 201^3 (= 8,120,601).

Table 4.3 summarises the logical structure of the test programs. The structure includes number of loops **No.**, loops configuration **Config.**, and cyclomatic complexity number **CC**. Loops configuration applies only to test programs with 2 loops or more because it describes the order of the loops. As for CC, **Low** is the basic CC number that equals to the number of decision points plus one while **Upp** is the details version of **Low**, such that each selection statement contains no logical operators anymore. In other words, **Upp** means each and every selection statement has been broken down into simple statements (as opposed to compound one). A simple statement is an expression that has arithmetic operators and/or relational operators only.

Based on the number of loops in Table 4.3, the test programs are classified into four classes, i.e. no loops, single loops, double loops, and multiple loops. This classification is presented in Table 4.4. Full details of the classification, which decomposes the configuration of loops, are described in Appendix A.1.

Based on the logical expression complexity inside the selection statement

Table 4.2: Input of Test Programs

Name	**Input**					
	Type	**Size**		**Range**		**Space (size)**
		Low	**Upp**	**Low**	**Upp**	
tA2008	int+	3	3	0	200	8,120,601
mmA2008	int	1	3	-100	100	8,161,203
iA2008	int	1	6	-100	100	66,273,881,404,206
bisA2008	real	2	2	1.72	1.75	900,000,000
binA2008	int	2	6	-100	100	66,273,881,404,005
bubA2008	int	1	6	-100	100	66,273,881,404,206
gA2008	int+	2	2	0	200	40,401
rA2008	int+	2	2	0	200	40,401
mtA2008	int+	3	3	0	200	8,120,601
tM2004	int+	3	3	0	200	8,120,601
eiR1985	int+	2	2	0	200	40,401
qG1997	int+	2	2	1	200	40,000
ttB2002	int+	3	3	0	200	8,120,601
eiB2002	int+	2	2	-100	100	40,401
qB2002	int+	2	2	0	200	40,401
scB2002	int+	8	8	1	128	72,057,594,037,927,936
fcB2002	int+	3	3	-100	100	8,120,601
fB2002	int	2	6	1	200	1,608,040,201,000
bG2011	int	1	8	-100	100	2,664,210,362,169,924,600
fG2011	int	7	7	-100	100	13,254,776,280,841,400
sG2011	int	5	5	-100	100	328,080,401,001

(looping criteria), the test programs are classified into three classes, i.e. 1 (simple or no logical operator), 2, and 3 or more, as seen in Table 4.5. Detailed breakdown of this classification can be found in Appendix A.2.

Table 4.3: Structure of Test Programs

Name	Structure			
	Loops		CC	
	No.	config.	Low	Upp
tA2008	0		4	13
mmA2008	1		4	4
iA2008	2	nested	4	5
bisA2008	1		5	7
binA2008	1		3	3
bubA2008	2	nested	4	5
gA2008	1		4	4
rA2008	1		4	4
mtA2008	1		7	16
tM2004	0		5	5
eiR1985	1		4	4
qG1997	2	serial	4	4
ttB2002	0		8	11
eiB2002	3	serial	11	15
qB2002	2	serial	6	7
scB2002	1		5	5
fcB2002	0		5	8
fB2002	3	serial	9	10
bG2011	2	nested	4	4
fG2011	0		7	7
sG2011	0		6	6

4.2 Experimental Setup

4.2.1 Overview

One must provide the following prior to conducting path testing, with respect to the program under test: source code, CFG, target paths, instrumented

Table 4.4: Structure Classification of Test Programs

Loops	**Test Program**
0	tA2008, tM2004, ttB2002, fcB2002, fG2011, sG2011
1	mmA2008, bisA2008, binA2008, gA2008, rA2008, mtA2008, eiR1985, scB2002
2	iA2008, bubA2008, qG1997, qB2002, bG2011
≥ 3	eiB2002, fB2002

Table 4.5: Expression Classification of Test Programs

Exp.	**Test Program (TP)**
1	mmA2008, iA2008, bisA2008, binA2008, bubA2008, gA2008, rA2008, mtA2008, tM2004, eR1985, qG1997, ttB2002, eiR1985, eiB2002, qB2002, scB2002, fcB2002, fB2002, bG2011, fG2011, sG2011, linear search, quicksort, shellsort, triangle myers, triangle michael, triangle sthamer, triangle wegener, triangle sthamer, triangle wegener
2	iA2008, ttB2002, qG1997, qB2002, shellsort, triangle wegener, bubA2008, fcB2002, fB2002
≥ 3	tA2008, mtA2008, bisA2008, triangle myers, triangle michael

code, and fitness function. The process is illustrated in this section for two test programs. Details for all test programs are given in Appendix B.

All test programs are rewritten in Matlab (C-like language) source code.

4.2.2 Example: tA2008

The following steps are necessary for testing tA2008 for path coverage.

4.2.2.1 Source Code

The following is the source code of tA2008.

```
function [type] = triangle(sideLengths)
A = sideLengths(1); % First side
B = sideLengths(2); % Second side
C = sideLengths(3); % Third side
if ((A+B > C) && (B+C > A) && (C+A > B)) % Branch # 1
    if ((A ~= B) && (B ~= C) && (C ~= A)) % Branch # 2
        type = 'Scalene';
    else
        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
```

4.2.2.2 Control Flow Graph

The following Figure 4.1 is CFG of tA2008. tA2008 has four (logical) paths and no loops.

4.2.2.3 Target Path Identification

The requirement for path testing that all paths inside the test program must be exercised at least once. However, due to the presence of loops the number of paths could be a huge number. Therefore, we limit the number of iterations to 3 situations, i.e. 0 (means the loop is not executed at all), 1, and 2. The reasoning is that a loop is logically tested if it is exercised not at all, once, and multiple times.

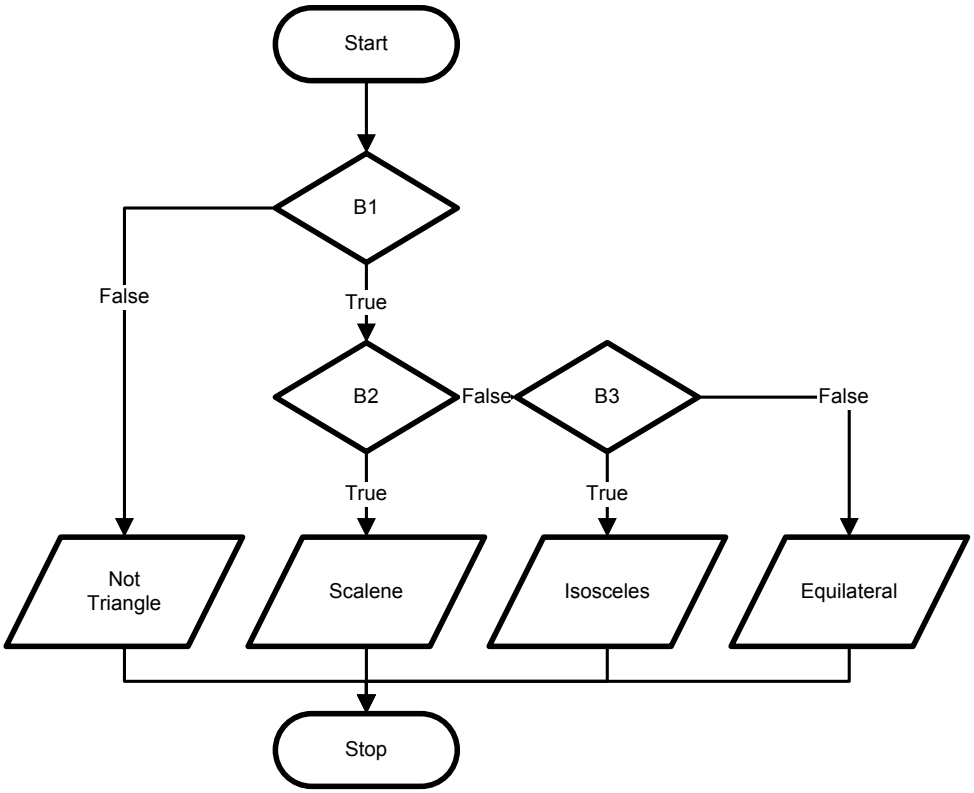


Figure 4.1: CFG of tA2008

For a test program without any loops, the number of (required) target paths for path testing is equal to its CC number.

In general, if k is the number of selections then the number of target paths p is as follows

$$p = \begin{cases} 2^k & \text{serial} \\ k+1 & \text{nested} \end{cases}$$

If k is the number of loops then p is formulated as

$$p = \begin{cases} 3^k & \text{serial} \\ n_k & \text{nested} \end{cases}$$

As for nested loops, n_k is derived from the following

$$n_k = \begin{cases} 1 & k = 0 \\ \sum_{i=0}^{2} \{n_{(k-1)}\}^i & k > 0 \end{cases}$$

For loops in series, the number of times each loop is executed is independent of the number of times the other loops are executed. The number of target paths is three raised to the power of the number of loops. For example, if there are 3 loops in series, the number of paths is equal to $3 \times 3 \times 3$ ($=3^3 = 3^k$).

The number of paths in nested loops is a bit more complicated to calculate. It all starts with a single path with no loops, i.e. $k = 0$ (the outer loop is not executed). Nested loops are only executed if the outer loop is exercised, i.e. the single and double or more iterations paths. So, the inner loop will be executed once in the single iteration that yields another 3^1 paths, and twice in the sequential order that produces 3^2 paths. In total, for two nested loops, i.e. one loop nested inside the other, will have 1 path that avoids the inner loop, 3 paths for the single inner loop iteration, and 9 paths for the twice inner loop iterations. The number of loops can be formulated as follows.

$$p = \begin{cases} 1 & = 3^0 & & n_0 \\ 1+1+1 & = \sum_{i=0}^{2} (3^0)^i & = \sum_{i=0}^{2} (n_0)^i & n_1 \\ 1+3+3^2 & = \sum_{i=0}^{2} \left\{ \sum_{j=0}^{2} (3^0)^j \right\}^i & = \sum_{i=0}^{2} (n_1)^i & n_2 \\ & & \vdots & \vdots \\ & & = \sum_{i=0}^{2} \{n_{(k-1)}\}^i & n_k \end{cases}$$

In reality, a program most likely will not contain only selections or only loops. In case of selections and loops are interleaved, the inner most branches will be calculated first as if it were in a recursive pattern.

In the case of tA2008, there are no loops. We can see from the CFG (Figure 4.1) that there are four paths, representing Not Triangle, Scalene, Isosceles, and Equilateral.

4.2.2.4 Target Path Representation

In this thesis, we represent a path by using a sequence of selection number - decision pairs coding. For example, the path that leads to *Not Triangle* can be written as [**1 F**], which means it traverses selection (or **B**ranch) number **1** and takes **F**(alse) decision. For calculation purpose, (**T**)rue is represented by 0 (originally derived from 0 distance for desired branch/decision) and **F** by 1 in the rest of the thesis. With this notation, the four target paths in tA2008 are [**1 1**] for *Not Triangle*, [**1 0 2 0**] for *Scalene*, [**1 0 2 1 3 0**] for *Isosceles*, and [**1 0 2 1 3 1**] for *Equilateral*.

4.2.2.5 Instrumented Test Program

In the instrumented version, a probe that has variable *traversedPath* in it is inserted before any selection statements for tracking traversed path. Upon entering a selection, the probe merges previously traversed branch number - decision pair with the currently executed pair in *traversedPath*. The current executed branch is assigned a decision value by fitness function *fitnessTriangle()*. After the execution, *traversedPath* will contain a path traversed by

particular input.

```
function [traversedPath, type] = triangle(sideLengths)
traversedPath = [];
A = sideLengths(1); % First side
B = sideLengths(2); % Second side
C = sideLengths(3); % Third side
% instrument Branch # 1
traversedPath = [traversedPath 1 fitnessTriangle(1, A, B, C)];
if ((A+B > C) && (B+C > A) && (C+A > B)) % Branch # 1
    % instrument Branch # 2
    traversedPath = [traversedPath 2 fitnessTriangle(2, A, B, C)];
    if ((A ~= B) && (B ~= C) && (C ~= A)) % Branch # 2
        type = 'Scalene';
    else
        % instrument Branch # 3
        traversedPath = [traversedPath 3 fitnessTriangle(3, A, B, C)];
        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
```

4.2.2.6 Fitness Function

The following is source code for *fitnessTriangle()* function. Korel's fitness function is embedded in the function. Each case inside the switch statement represents each selection function in the tA2008. So, each case is already embedded with Korel's fitness function (see *branchVal*) on how to evaluate the decision based on the parameter values supplied to it.

```
function branchVal = fitnessTriangle(branchNo, A, B, C)
    k = 1; % the smallest step for integer
    switch (branchNo)
      case 1,
          % branch #1: (A+B > C) & (B+C > A) & (C+A > B)
          term(1) = C - (A+B);
          term(2) = A - (B+C);
```

```
            term(3) = B - (C+A);
            for i=1:3
                if (term(i) < 0),
                    term(i) = term(i) - k;
                else
                    term(i) = term(i) + k;
                end
            end
            branchVal = sum(term);
            if (~((A+B > C) & (B+C > A) & (C+A > B)) & (branchVal < 0)), % Branch # 1
                branchVal = -branchVal;
            end
        case 2,
            % branch #2: (A ~= B) & (B ~= C) & (C ~= A)
            if (A ~= B), term(1) = 0; else term(1) = k; end
            if (B ~= C), term(2) = 0; else term(2) = k; end
            if (C ~= A), term(3) = 0; else term(3) = k; end
            branchVal = sum(term);
        case 3,
            % branch #3: ((A == B) & (B ~= C)) | ((B == C) & (C ~= A)) | ((C == A) & (A ~= B))
            if (A == B), subTerm(1) = 0; else subTerm(1) = abs(A-B); end
            if (B ~= C), subTerm(2) = 0; else subTerm(2) = k; end
            if (B == C), subTerm(3) = 0; else subTerm(3) = abs(B-C); end
            if (C ~= A), subTerm(4) = 0; else subTerm(4) = k; end
            if (C == A), subTerm(5) = 0; else subTerm(5) = abs(C-A); end
            if (A ~= B), subTerm(6) = 0; else subTerm(6) = k; end
            term(1) = subTerm(1) + subTerm(2);
            term(2) = subTerm(3) + subTerm(4);
            term(3) = subTerm(5) + subTerm(6);
            branchVal = min(term);
    end
end
```

4.2.3 Example: mmA2008

The second test program is mmA2008 with the following source code. It has a single loop, with two sequential selection **IFs** inside the loop. Figure 4.2 is CFG of mmA2008.

```
function [miniMaxi] = minimaxi(num)
numLength = length(num);
mini = num(1);
maxi = num(1);
idx = 2;
while (idx <= numLength) % Branching #1
    if maxi < num(idx) % Branching #2
```

```
        maxi = num(idx);
    end
    if mini > num(idx) % Branching #3
        mini = num(idx);
    end
    idx = idx+1;
end % while end
miniMaxi = [mini maxi];
end
```

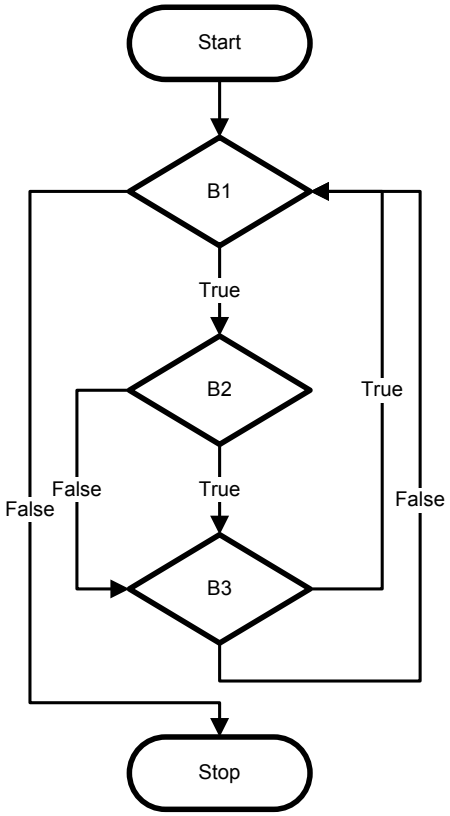


Figure 4.2: CFG of mmA2008

The existence of the loop at **B1** increases the number of paths. The following are the 13 target paths for mmA2008.

- loop executed 0 times:
  [1 1]

- loop executed one time:
  [1 0 2 0 3 1 1 1]
  [1 0 2 1 3 0 1 1]
  [1 0 2 1 3 1 1 1]

- loop executed two times:
  [1 0 2 0 3 1 1 0 2 0 3 1 1 1]
  [1 0 2 0 3 1 1 0 2 1 3 0 1 1]
  [1 0 2 0 3 1 1 0 2 1 3 1 1 1]
  [1 0 2 1 3 0 1 0 2 0 3 1 1 1]
  [1 0 2 1 3 0 1 0 2 1 3 0 1 1]
  [1 0 2 1 3 0 1 0 2 1 3 1 1 1]
  [1 0 2 1 3 1 1 0 2 0 3 1 1 1]
  [1 0 2 1 3 1 1 0 2 1 3 0 1 1]
  [1 0 2 1 3 1 1 0 2 1 3 1 1 1]

The instrumented version of mmA2008 source code is as follows.

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

The following is the code for fitness function *fitnessMiniMaxi()*. Korel's fitness function is already formulated in every *branchVal* evaluation.

```
function branchVal = fitnessMiniMaxi(branchNo, predicate)
    k = 1; % the smallest step for integer
    switch (branchNo)
      case 1,
          % branch #1: (idx <= numLength)
          branchVal = predicate(1) - predicate(2);
      case 2,
          % branch #2: (maxi < num(idx))
          branchVal = predicate(1) - predicate(2);
      case 3,
          % branch #3: (mini > num(idx))
          branchVal = predicate(2) - predicate(1);
    end
    if ((branchNo == 2) || (branchNo == 3)),
        if (branchVal < 0)
            branchVal = branchVal - k;
        else
            branchVal = branchVal + k;
        end
    end
end
```

4.2.4 Test Program Preparation

Appendix B presents full details (source code, CFG, target paths, instrumented source code, fitness function) for all test programs. All of the details have been prepared manually, in this thesis. Automation of these tasks is an important research area, but beyond the scope of this thesis.

4.3 System Operation

In order to apply the proposed approach appropriately, a testing manual for using the test generation system (TGS) has been developed. The manual

consists of two major parts: testing guide and operating instruction. The detail of it is presented in Appendix C.1.

4.3.1 Preparation

Test program preparation has been described in Section 4.2.

4.3.2 Test Data Representation

For each test problem, a test case is one set of input values. Each input (chromosome) has its domain, which is further decomposed into sub-domains, i.e. allele range. For tA2008, the input is three integers, e.g. (3, 4, 5), and each integer has its domain, e.g. 0 to 10000.

4.3.3 Execution

In general, the execution of GA-based path testing is as follows:

1. Initial population is generated
2. For each chromosome, i.e. set of input values
   (a) Execute instrumented program with this input
   (b) Record which execution path is followed
   If this is a path not covered before,
   - record this test data,
   - note this path is now covered

(c) Evaluate fitness

3. If stopping condition not yet reached, go to Step 2

4. Record statistics

Ideally, if there are N target paths, the search stops with N different chromosomes recorded, each of which causes execution of a different target path. In practice, one path could be covered by many test data, but it is not the other way around. So, only the first test datum that covered a path is recorded. When a path is covered, it is removed from the target list in the next generation, if the search is still going on.

Information regarding test data that cover the same path is a potential heuristic for supporting identification of equivalence partitioning. It is also useful for estimating the path complexity with respect to the other target paths. This is an interesting research area.

4.3.4 Experiment

Each variation of a test program is run 30 times. The same 30 seeds are used for all variations on experiments. The test programs are those identified in Section 4.1.2.

Chapter 5

Challenges and Key Parameters

5.1 Introduction

Hundreds of papers on testing and debugging in search based software engineering (SBSE) have appeared in the literature from as early as 1976 [12]. Most of them have concentrated on either branch or statement coverage and relatively few have considered path testing. Those that have, used GAs to evolve test cases.

Path testing can find deeper logical errors that may not be found using branch or statement coverage, because errors associated with different numbers of iterations through loops can be exposed. Each different number of iterations in loops is considered a different path. For example, a loop can be executed in several ways, such as no iterations, once, twice, and more. All these executions represent different paths.

Path testing requires coverage of the feasible target paths as well as recog-

nition of infeasible paths. It is unlikely that complete path coverage can be found because a loop may be repeated infinite times. One way to overcome this problem is to limit the executions for a loop to a reasonable number. Even though this is not complete path coverage, the expectation is that it is enough to expose the most likely errors.

This work aims to analyze some challenges of path testing, identify the control parameters that have most effect on the performance of GA-based path testing, and find appropriate values for those control parameters. The challenges include test program instrumentation, target paths generation, control flow generation, fitness function generation, and infeasible paths detection. Relying on analytical and manual work to handle these challenges is time consuming and error prone. In particular, this is likely to happen as test programs are getting bigger and more complicated. Although automating these processes could have more benefit by having bigger test programs and more number of test programs, the effort to do it does not relate to the contribution of the thesis. In this work, all manual works have been carefully taken by verifying and validating several times.

The experiments were conducted using 21 test programs from the literature, varying each parameter through a set of plausible values. In a two-step process the best parameter settings as well as the most influential parameters were determined.

The chapter is organized as follows. Section 5.2 describes proposed approaches, test programs preparation, and experiments design. In Section 5.3, the experimental results are presented. Section 5.4 explains the analysis on parameters setup, test program characteristics, path testing adequacy,

and search technique. Some threats to the experimental results validity are showed in Section 5.5. Section 5.6 concludes the chapter.

5.2 Approach and Test Programs

The experiments described in this chapter aim to shed some light on the challenges and limitations of path testing. These experiments build on the previous work of Ahmed and Hermadi in 2008 [20]. The previous experiments concentrated on the effect of using different fitness functions. These experiments seek to reveal behaviors of path testing as parameter settings are changed, e.g. path coverage achievement as the population size and other parameters vary. Test programs and their target paths are selected from the literature. The following sections describe the details of the experiments.

5.2.1 Test Programs

In this research, 21 test programs are selected for experimentation as listed in Table 4.1. The details of the test programs are presented in Section 4.1.

The test programs cover a range of program types and so are a valid set of test data. The details are described in the classification of test programs in Section 4.1.1

5.2.2 Search Setup

GA is used as the search technique in these experiments, in line with other recent research in this area [12].

The following settings are used: random seeded initial population generation, Roulette Wheel selection (RWS), single point crossover, and generation gap. Generation gap is set to 90%, i.e. 90% of the population experiences GA's operators. 30 runs were made with each combination of parameter settings, always using the same 30 random number seeds.

The fitness function used in the experiments combines approximation level and branch distance [20]. Approximation level (AL) measures similarity/dissimilarity between the path taken by an input data and a target path; it is counted as the number of matched/unmatched branches. The count continues as far as the first unmatched branch encountered. Branch distance (BD) is calculated as Korel's distance function [8], if the path taken differs from the target path of interest. AL and BD can be combined in different ways, each combination representing a different fitness function. The best fitness function out of 32 different fitness functions, studied in [56], is chosen in this research. It is one that considers all current (uncovered) target paths, path-wise traversal, normalized branch distance, normalized violation (negation of AL), normalized fitness value, and no weighting between AL and BD.

In test programs that have variable-sized chromosomes, the range of chromosome lengths is determined by two extreme values related to the number of loop iterations in the test programs, i.e. exercising no iterations and many iterations. For example, suppose a test program may require at least two positive integers to either avoid a loop or enter a loop once, but it needs three positive integers to execute a loop twice: thus its input length must be varied between two and three positive integers. For example, a test program

that accepts an input between 2 and 3 integers at a time may have input either [-1 30] or [10 -3 400].

5.2.3 Control Parameters

Four GA parameters are considered in the experiments. They are population size, number of generations, allele range, and mutation rate. Others are fixed, as mentioned in Section 5.2.2. The ranges of values for those parameters were chosen based on past experiences [56, 20].

To begin with, 9 different values (10, 30, 50, 70, 100, 150, 200, 250, 300) were investigated for population size, 5 values (50, 100, 250, 500, 1000) for the number of generations, 4 ranges ([0 10000], [0 1000], [0 100], [0 10]) for allele range, and 2 values (0.15, 0.3) for mutation rate.

As the experiments progressed, the ranges of the parameter values were narrowed down as it became clear that some values were beyond the range of being useful. This approach of setting GA parameters is similar to that of the racing algorithm [155].

5.3 Experimental Results (Key Parameters)

5.3.1 Best Parameters

For each test program, a two-phase analysis was done. Firstly, best parameter settings were identified by cumulative plotting of the results over all runs as parameters vary. Secondly, deciding which parameter matters most was then

done by seeing the effect of varying each parameter by itself while each other parameter is fixed at its best value.

5.3.1.1 Detailed Example

To illustrate, the process of finding the best parameter settings for population size, number of generations, allele range, and mutation rate is presented in detail for tA2008.

In searching for best population size, Figure 5.1 depicts that at least 85% of 1200 runs per population size that covered all 4 target paths is achieved at a population size of 150. Having bigger population size such as 200, 250, and 300 does not significantly increase the number of runs that cover the same number of paths. Thus, the population size 150 is the optimal choice.

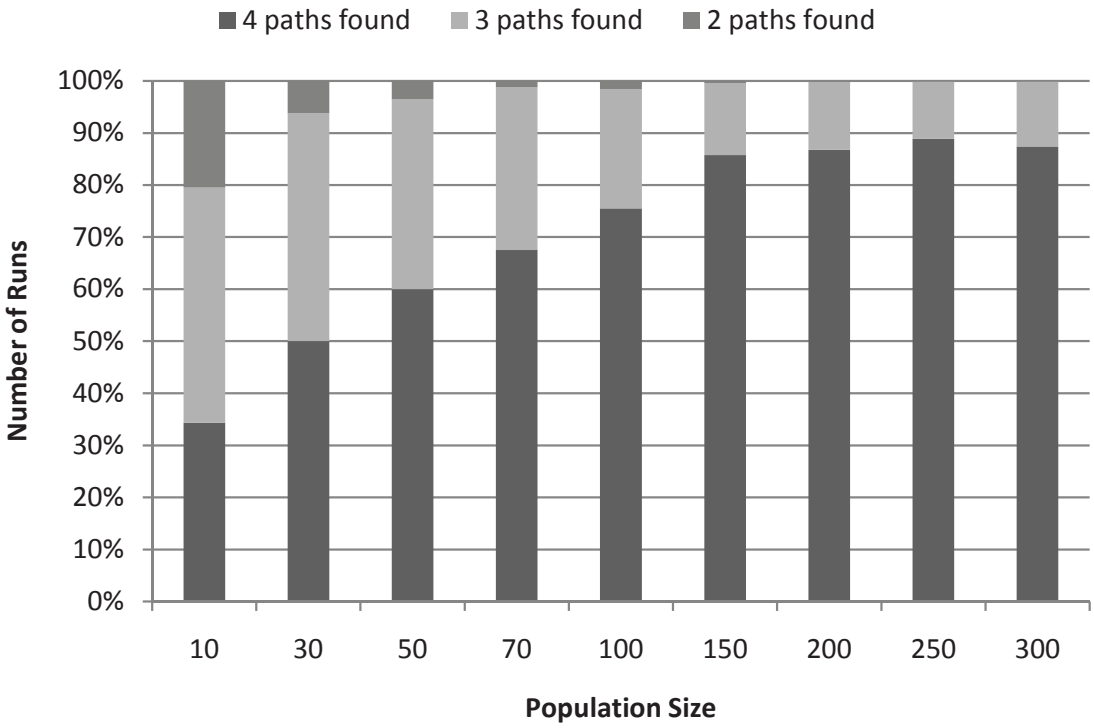


Figure 5.1: Effect of population size on tA2008

Similarly, Figure 5.2 shows cumulative number of runs as number of generations varies. At 250 generations, over 73% of 2160 runs found all the paths. Neither doubling (500) nor quadrupling (1000) the number of generations significantly increases the number of runs that found all the paths. This also means that the efforts exerted are doubled or quadrupled, but the gain is not commensurate. So, 250 generations is taken as the optimal parameter value.

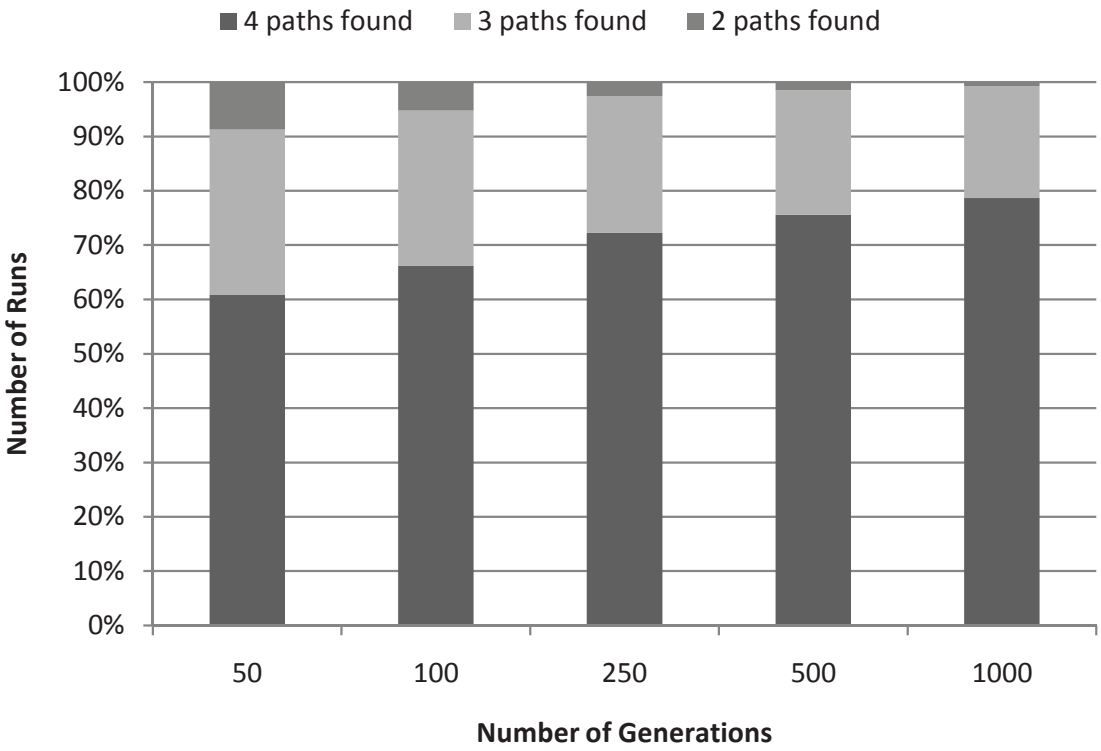


Figure 5.2: Effect of number of generations on tA2008

As for allele range, the best range is [0 10], with which over 98% of 2700 runs covered all the paths as presented in Figure 5.3. This means that even though the range is very narrow, it suffices to cover almost all the paths.

In Figure 5.4, mutation rate 0.3 has the most number of runs that found all the paths: 72.8% of 5400 runs found all the paths. However, this is very

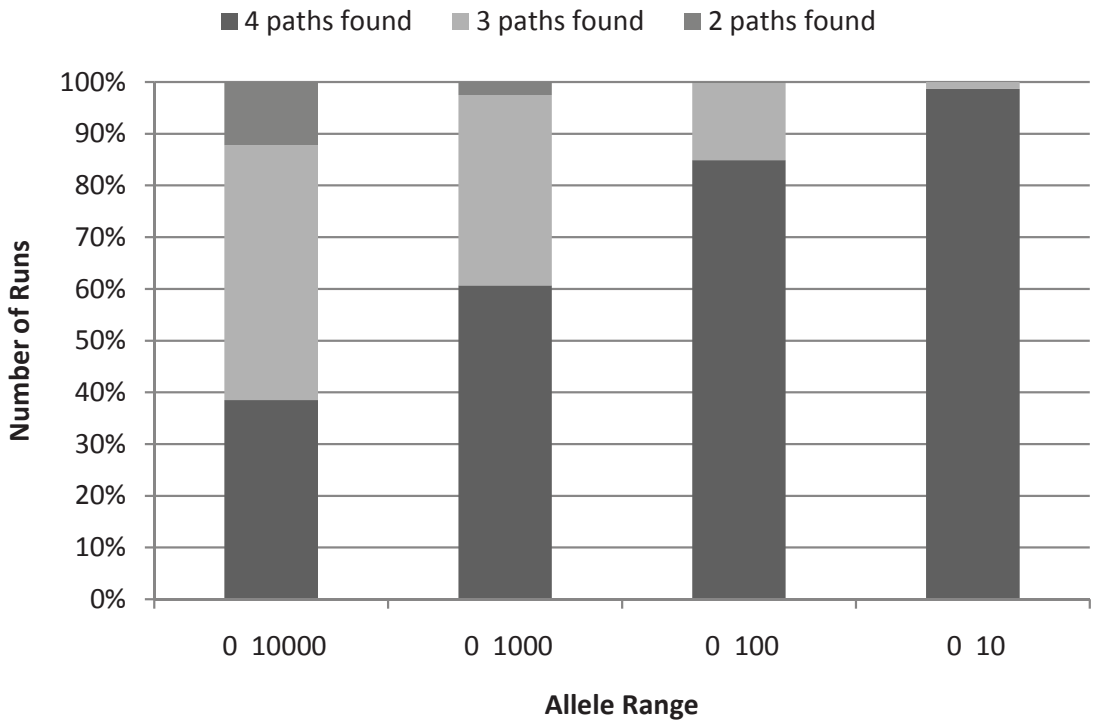


Figure 5.3: Effect of allele range on tA2008

close to 0.15 mutation rate which achieved 68.5%.

5.3.1.2 All Test Programs

The tA2008 program was investigated first. There are 360 different combinations of parameter values, with 30 runs per combination, requiring 10800 runs.

The experimental results suggested that the number of generations and the allele range could be narrowed down. For the next 5 programs, 216 combinations of parameter values were used (dropping 1000 for the number of generations, and the longest allele range). 30 runs per combination were still executed, requiring 6480 runs.

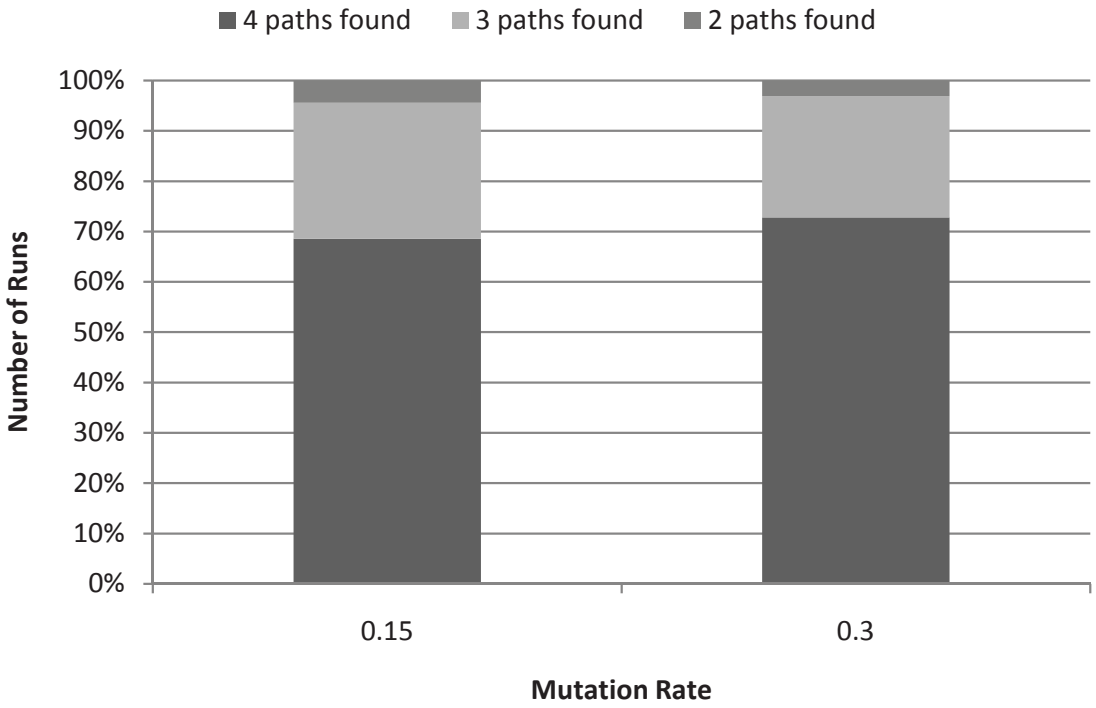


Figure 5.4: Effect of mutation rate on tA2008

Based on the analysis of results from the first 6 test programs, the parameter ranges for the next 3 programs were further reduced to (30, 100, 250) for population size, (50, 500) for number of generations, 2 allele ranges that are program dependent, and 0.15 for mutation rate. Each program required 360 runs (12 combinations $\times$ 30 runs).

The remaining 12 test programs were run using the same parameter settings as had been used for the previous test program(s) with the most similar program characteristics (classified similarly in the proposed classification of test programs). For example, eR1985 and qG1997 have similar characteristics to some other test programs in terms of input type and length e.g. binA2008, bubA2008, gA2008, iA2008, mmA2008, and rA2008. For both of these, a low population size (50) was best.

Having regard to the cumulative plots for all parameters, Table 5.1 presents the best parameter settings for all of the test programs. It has best combinations of population size **Pop**, number of generations **Gens**, allele range **Allele**, and mutation rate **Mut**.

Table 5.1: Best Parameter Settings

No	Program	Pop	Gens	Allele	Mut
1	tA2008	150	250	0 10	0.30
2	mmA2008	200	100	-10 10	0.15
3	iA2008	70	50	-10 10	0.15
4	bisA2008	100	100	1.72 1.75	0.30
5	binA2008	30	50	-10 10	0.15
6	bubA2008	50	50	-10 10	0.30
7	gA2008	250	50	0 20	0.10
8	rA2008	250	500	0 20	0.10
9	mtA2008	250	50	0 20	0.10
10	tM2004	150	250	0 10	0.30
11	eiR1985	50	50	0 10	0.30
12	qG1997	50	50	1 10	0.30
13	ttB2002	250	500	0 20	0.15
14	eiB2002	250	500	-10 10	0.15
15	qB2002	250	500	0 20	0.15
16	scB2002	250	500	1 128	0.15
17	fcB2002	250	500	-10 10	0.15
18	fB2002	100	100	1 200	0.15
19	bG2011	100	50	-10 10	0.15
20	fG2011	250	500	-10 10	0.15
21	sG2011	250	50	-10 10	0.15

Table 5.2 presents path coverage when the best parameter setting was used for each test program. The table has number of target paths **Paths**, number of feasible paths **Feas**, maximum of **Feas** found **PFM**, average over

30 runs of **Feas** found **PF**, number of **Feas** missing out of **PFM MPFM**, number of missing **Feas** out of **PF MPF**, **CCL**, and **CCU**. Twenty out of 21 test programs achieve 100% coverage of feasible paths in at least one run. bisA2008 program has 3 paths remaining not found, this is due to the requirement that the root of the equation being solved need to be exactly 0 (zero), so any approximations are not accepted. In other words, these 3 paths are feasible but they are less likely to find unless the root is exactly 0.0. The least coverage is seen with fG2011, which has the highest **MPF**.

Table 5.2: Path Coverage

Program	**Paths**	**Feas**	**PFM**	**PF**	**MPFM**	**MPF**	**CCL**	**CCU**
tA2008	4	4	4	4.00	0	0	4	13
mmA2008	13	13	13	13.00	0	0	4	4
iA2008	6	5	5	5.00	0	0	4	5
bisA2008	9	9	6	6.00	3	3	5	7
binA2008	7	7	7	7.00	0	0	3	3
bubA2008	15	4	4	4.00	0	0	4	5
gA2008	8	5	5	5.00	0	0	4	4
rA2008	5	4	4	4.00	0	0	4	4
mtA2008	52	20	20	19.00	0	1	7	16
tM2004	8	7	7	6.23	0	0.77	5	5
eiR1985	12	3	3	2.97	0	0.03	4	4
qG1997	21	4	4	4.00	0	0	4	4
ttB2002	8	8	8	8.00	0	0	8	11
eiB2002	31	5	5	5.00	0	0	11	15
qB2002	27	10	10	10.00	0	0	6	7
scB2002	15	4	4	4.00	0	0	5	5
fcB2002	5	5	5	5.00	0	0	5	8
fB2002	32	8	8	7.87	0	0.13	9	10
bG2011	20	11	11	11.00	0	0	4	4
fG2011	30	30	30	25.13	0	4.87	7	7
sG2011	32	32	32	32.00	0	0	6	6

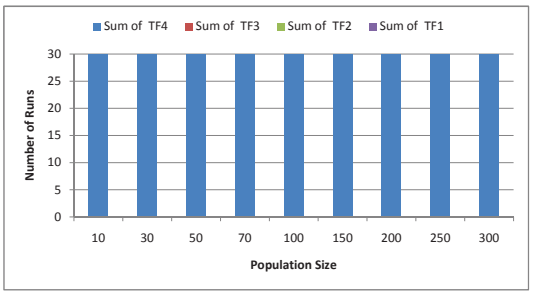
Considering Table 5.1, a reasonable set of common parameter values is as follows: **Pop** 100, **Gens** 100, **Allele** about 200 range, crossover rate 0.9, and **Mut** 0.15. Path coverage using this common set of parameter values is also evaluated throughout this thesis, and compared with path coverage using the best parameter values.

The reason for considering a common set of parameter values is to understand the performance that can be expected by a tester starting on a new test program. They will not know what the best parameters are for the program, and considerable effort (perhaps more than is saved by automating the search for test data) might be needed to find them. The best parameters setup is expected to cover more paths than that of the common one, but it is valuable to know what path coverage is possible without spending time or adding testing complexity to find the optimal setup.

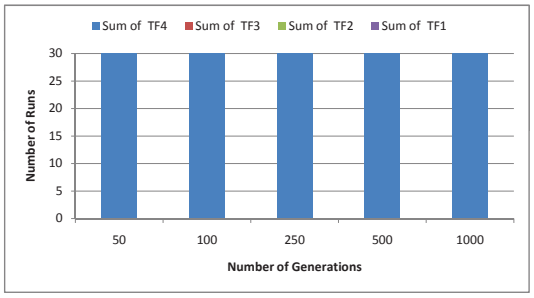
5.3.2 Influence of Parameters

Varying one parameter and fixing the rest is a way to identify parameter influence as mentioned earlier. For example, Figure 5.5 shows which parameter most matters for program tA2008. Varying population, generation, or mutation rate does not change the path coverage as shown in Figure 5.5(a), Figure 5.5(b), and Figure 5.5(d). On the contrary, Figure 5.5(c) shows that varying the allele range between [0 100] and [0 10] changes the path coverage.

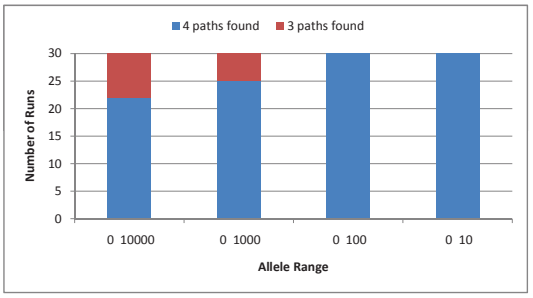
Another example is mmA2008: varying 3 parameters can change the path coverage. Figure 5.6 presents the parameters variation. Population size and allele range are the parameters that matter as shown in Figure 5.6(a) and



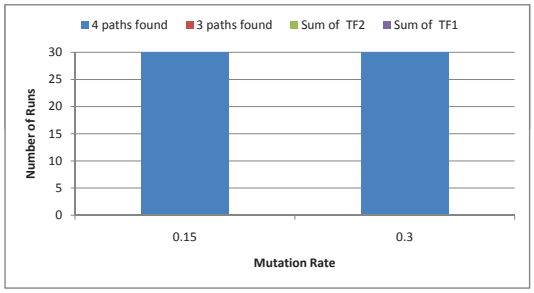
(a) Varying population



(b) Varying generation

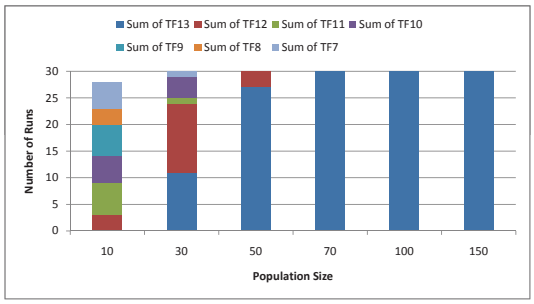


(c) Varying allele range

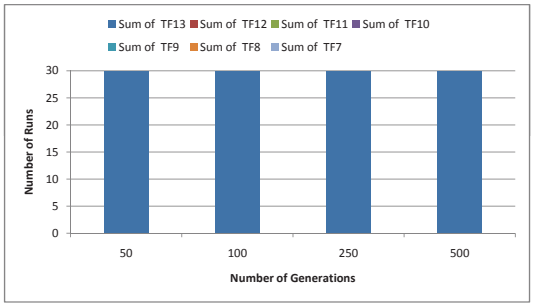


(d) Varying mutation rate

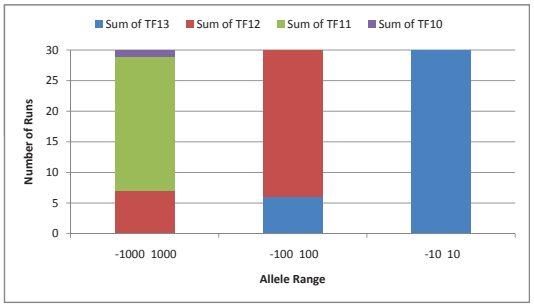
Figure 5.5: Parameters influence for tA2008



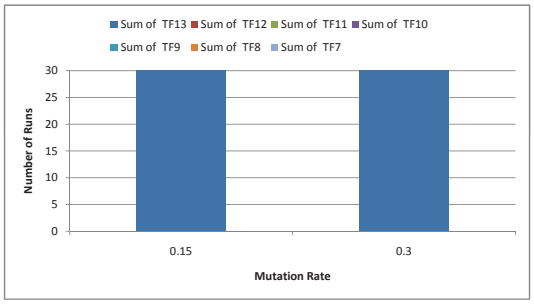
(a) Varying population



(b) Varying generation



(c) Varying allele range



(d) Varying mutation rate

Figure 5.6: Parameters influence to mmA2008

Figure 5.6(c), respectively.

In general, more target paths are found, regardless of the test program, with larger population sizes, more generations, narrower allele ranges, and higher mutation rate. If these four parameters are ordered in decreasing order of impact, they are: population size, allele range, number of generations, and mutation rate. However, the magnitude of increment varies from one test program to the next.

Further observations of the experimental results are:

1. The hardest target paths for GA to cover involve generating multiple input numbers with exactly the same values, regardless of the test pro-

gram. For example, a path that requires 3-integer input to be exactly the same numbers (such as in tA2008, mtA2008, tM2004, and ttB2002) is always covered last.

2. One of the hardest paths is one that must be covered by producing an exact real input number from an approximation. For example, a path in bisA2008 is always found the latest on average, because its input data must be exactly zero input value that is computed from the previous statements.

3. The longer the path the longer time is consumed in terms of the program execution and the number of generations. A longer path means more branches to traverse in the path representation. In most of the test programs, the longer paths are the later ones to be covered, although not always. For example, some longer paths are found earlier than the shorter one such as in mmA2008 and mtA2008.

4. The more complex a branch the longer time to cover it. For example, some shorter paths in mmA2008 and mtA2008 were covered later than the longer. The path is shorter but the complexity that builds up from each traversed branch adds up, making it harder to cover.

5. In general, the number of paths found (almost) logarithmically decreases as number of generations increases as shown in Figure 5.7.

6. The presence of loop(s) greatly increases the computation complexity. For example, mtA2008 has more paths and longer paths (20 feasible out of 52 paths; 1 loop) than its components tA2008 (4 feasible out of 4 paths) and mmA2008 (13 feasible out of 13 paths; 1 loop).

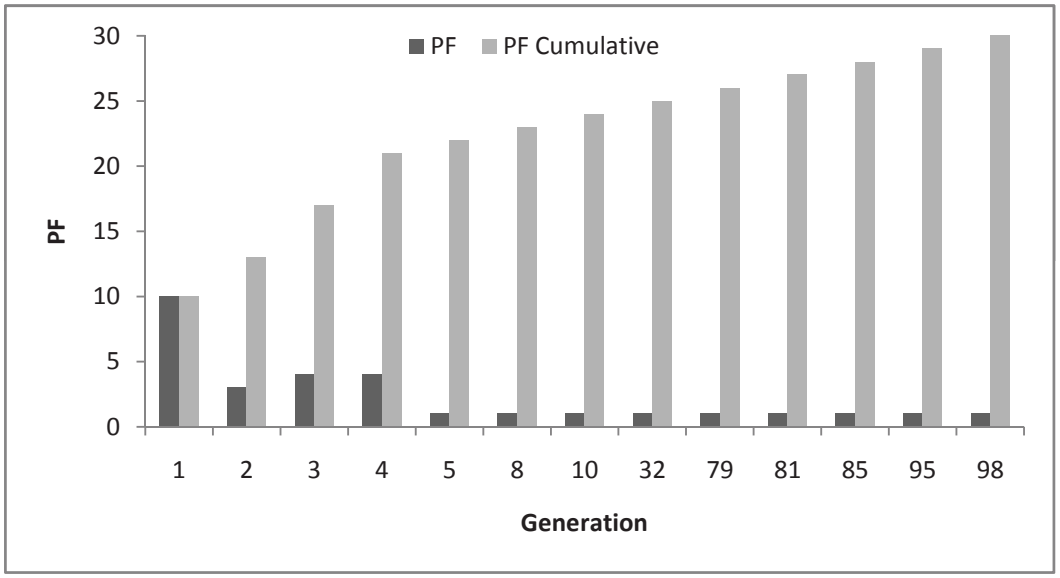


Figure 5.7: PF of the 2^{nd} run of fG2011 using best parameters setup

7. Higher CC number means more difficult to cover the paths.

8. Variable length input means more time complexity due to larger search space. Test programs that have such inputs are mmA2008, iA2008, binA2008, bubA2008, fB2002, and bG2011.

5.3.3 Path Complexity of Classes of Test Programs

Intuitively, complexity of program (target) paths (hereafter paths complexity) is a function of many aspects of the program: input size, control flow structure, complexity of selection statements, and infeasible paths. Path complexity can be approached using the one or more of the following: the percentage of average number of path founds (over several runs or repetitions), the average number of test data generated (over several runs or repetitions), the size of the input space, the percentage of infeasible (target) paths, and the ratio between infeasible and feasible (target) paths. Except

for the first, for all these measures a higher value means a more complex test program.

The measures in percentage, i.e. the percentage of average number of paths found and the percentage of infeasible paths, are comparable, which means it makes sense to compare the value of a test program with the value of other test programs. The percentage of average number of paths found pf is calculated as the average number of paths covered over some repetitions divided by the number of feasible paths of the test program. It is formulated in Equation 5.1.

$$pf = \frac{\frac{1}{n} \times \sum_{i=1}^{n} |pf_i|}{|fp|} \tag{5.1}$$

where n is the number of repetitions, $|pf_i|$ is the number of paths found for the i^{th} repetition, and $|fp|$ is the number of feasible paths. This measure is to find the efficacy of the search and the maximum value, i.e. 100%, means that all the (target) paths are feasible and covered.

The average number of test data generated *notd* is calculated as in Equation 5.2.

$$notd = \frac{\sum_{i=1}^{n} |td_i|}{n} \tag{5.2}$$

where n is the number of repetitions and $|tdi|$ is the number of test data generated until the last feasible paths found for the i^{th} repetition. It also means that if a test program has 5 feasible paths regardless of any number of infeasible ones, and the maximum number of generations has been reached and only 4 feasible paths were found before the maximum number of generations then the number of test data generated is up to where the 4th feasible one was found. *notd* is a measure of the search efficiency. In order to make

it a comparable measure, it must be normalized or scaled to certain range.

The percentage of infeasible paths if is formulated as in Equation 5.3.

$$if = 1 - \frac{|fp|}{|tp|} \tag{5.3}$$

where $|fp|$ is the number of feasible paths and $|tp|$ is the number of all target paths. The if is to measure the composition of infeasible target paths.

The size of input space is is formulated as in Equation 5.4.

$$is = \prod_{i=1}^{n} |D_i| \tag{5.4}$$

where n is the number of input components and $|D_i|$ is the size of the i^{th} input component.

The ratio between feasible and infeasible paths rif is computed as in Equation 5.5.

$$rif = \frac{|ip|}{|fp|} \tag{5.5}$$

where $|ip|$ is the number of infeasible paths and $|fp|$ is the number of feasible paths.

The rif is to measure the strength or influence of the infeasible paths over the feasible ones. is, if, and rif are inherent for a test program, while pf and *notd* apply to the execution of the test program with certain parameters setup.

In this report, all the measures are considered for the analysis as listed in the following Table 5.3. Column Best Pars means that the search is using the best parameter setup for each test program, Column Comm Pars means that

the search is using the common parameter setup for every program, while Column C-Only Pars means the search is using common parameters and all the infeasible paths are excluded from the target paths for all test programs.

Table 5.3: Path Complexity Measures

Program	is	Paths		if	rif	Best Pars		Comm Pars		C-Only Pars	
		All	Feas			pf	notd	pf	notd	pf	notd
tA2008	8,120,601	4	4			100.0%	195.0	92.5%	3,570.0	92.5%	3,570.0
mmA2008	8,161,203	13	13			100.0%	820.0	90.8%	3,513.3	90.8%	3,513.3
iA2008	66,273,881,404,206	6	5	16.7%	0.2	100.0%	102.7	100.0%	81.9	100.0%	74.7
bisA2008	900,000,000	9	9			66.7%	183.0	66.7%	183.0	66.3%	200.0
binA2008	66,273,881,404,005	7	7			100.0%	33.9	100.0%	30.0	100.0%	30.0
bubA2008	66,273,881,404,206	15	4	73.3%	2.8	100.0%	50.0	100.0%	55.0	100.0%	56.7
gA2008	40,401	8	5	37.5%	0.6	100.0%	257.5	71.4%	5,242.5	88.0%	7,308.3
rA2008	40,401	5	4	20.0%	0.3	100.0%	250.0	99.3%	4,192.5	94.2%	4,000.0
mtA2008	8,120,601	52	20	61.5%	1.6	95.0%	1,700.0	87.7%	2,900.0	89.7%	10,350.0
tM2004	8,120,601	8	7	12.5%	0.1	89.0%	319.5	58.1%	2,675.0	57.1%	3,135.0
eiR1985	40,401	12	3	75.0%	3.0	99.0%	53.5	100.0%	496.5	86.7%	1,151.5
qG1997	40,000	21	4	81.0%	4.3	100.0%	50.0	100.0%	50.0	100.0%	50.0
ttB2002	8,120,601	8	8			100.0%	5,907.5	87.9%	767.5	87.9%	766.7
eiB2002	40,401	31	5	83.9%	5.2	100.0%	250.0	98.0%	3,132.5	98.6%	4,132.5
qB2002	40,401	27	10	63.0%	1.7	100.0%	257.5	86.3%	750.0	89.0%	3,900.0
scB2002	72,057,594,037,927,936	15	4	73.3%	2.8	100.0%	767.5	95.8%	2,825.0	96.7%	4,191.7
fcB2002	8,120,601	5	5			100.0%	292.5	98.0%	4,007.5	98.0%	4,008.3
fB2002	1,608,040,201,000	32	8	75.0%	3.0	98.4%	487.0	98.4%	487.0	95.8%	633.3
bG2011	2,664,210,362,169,924,600	20	11	45.0%	0.8	100.0%	153.0	100.0%	147.0	100.0%	146.7
fG2011	13,254,776,280,841,400	30	30			83.8%	64,792.5	31.0%	14,682.5	31.0%	14,682.5
sG2011	328,080,401,001	32	32			100.0%	3,032.5	55.5%	16,432.5	55.5%	16,432.5
Average				55.2%	2.0	96.8%	3,807.4	86.5%	3,153.4	86.6%	3,920.7
Min				12.5%	0.1	66.7%	33.9	31.0%	30.0	31.0%	30.0
Max				83.9%	5.2	100.0%	64,792.5	100.0%	16,432.5	100.0%	16,432.5
Median				63.0%	1.7	100.0%	257.5	95.8%	2,675.0	92.5%	3,513.3
StDev				25.8%	1.6	8.1%	14,040.5	19.1%	4,461.2	18.5%	4,704.1

The following Table 5.4 lists the test programs based on its structure (**Struct** column), expression (**Expr** column) classification, and combination of both (**SC** column), which means the second and the third digits in Appendix A.1 that represent complexity of a program without considering its structure. Recall, in the **Struct**, class 0 means that the program has no loop, at most, and in the **Expr**, class 1 means that the program has one simple expression, at most, i.e. no logical connectors in it.

In this subsection, the aim of the analysis is to find out what program characteristics affect the search performance in term of path coverage and number of test data generated. In other words, it is to analyse the efficacy and the efficiency of the search method.

Table 5.4: Test Program Class

No	Program	Classification		
		Struct	**Expr**	**SC**
1	tA2008	0	3	03
2	mmA2008	1	1	11
3	iA2008	2	2	22
4	bisA2008	1	3	13
5	binA2008	1	1	11
6	bubA2008	2	2	22
7	gA2008	1	1	11
8	rA2008	1	1	11
9	mtA2008	1	3	13
10	tM2004	0	1	01
11	eiR1985	1	1	11
12	qG1997	2	2	22
13	ttB2002	0	2	02
14	eiB2002	3	1	31
15	qB2002	2	2	22
16	scB2002	1	1	11
17	fcB2002	0	2	02
18	fB2002	3	2	32
19	bG2011	2	1	21
20	fG2011	0	1	01
21	sG2011	0	1	01

The parameters setup being used for this classification analysis is the common one.

The best parameters setup is expected to cover more paths than that of the common one. To check this, we conducted statistical t-test, comparing the efficacy measure pf with significance level α=0.05. The null hypothesis H_0 is that there is no significant difference in pf between the best and the common parameters setup. The t-test results that H0 is rejected with P(T$\leq$t)

two-tail = 0.01, which is less than α. This means that best parameters setup has better efficacy than the common one. This result matches with expectation.

Another thing to consider is that having infeasible paths might hinder the search. So, we conducted t-test for the test programs having all (feasible and infeasible) paths and only feasible paths by removing the infeasible ones, if any. The test result shows that there is no significant different in path coverage between having all paths and only feasible ones; with P(T≤t) two-tail = 0.98, which is greater than α. That is, the presence of infeasible paths in the list of target paths does not hinder the search for the feasible paths.

In order to achieve the aim of this sub section, i.e. investigating the relationship between the classification schemes and the programs characteristics, the following pf, if, and $notd$ measures of the paths complexity is plotted against the classification scheme using common parameters setup and a curve is fitted using non-linear regression model. The model is evaluated using the goodness of fit R^2; if it is 1 then exactly fit, and the closer to 1 the fitter the curve.

Figure 5.8 describes the SC classification that merges structure classification (the first digit) and expression classification (the second digit). So, SC 01 means that no loops and one simple expression. In the figure, there are 3 SC classes that have no test programs, i.e. Class 12, 23, and 33. pf-C means the pf (number of paths found) when the common parameters setup is applied.

Figure 5.9 describes the structure classification. In the figure, the number of paths found (pf-C) increases polynomially (Poly. (pf-C); $y = 0.0045x^3 -$

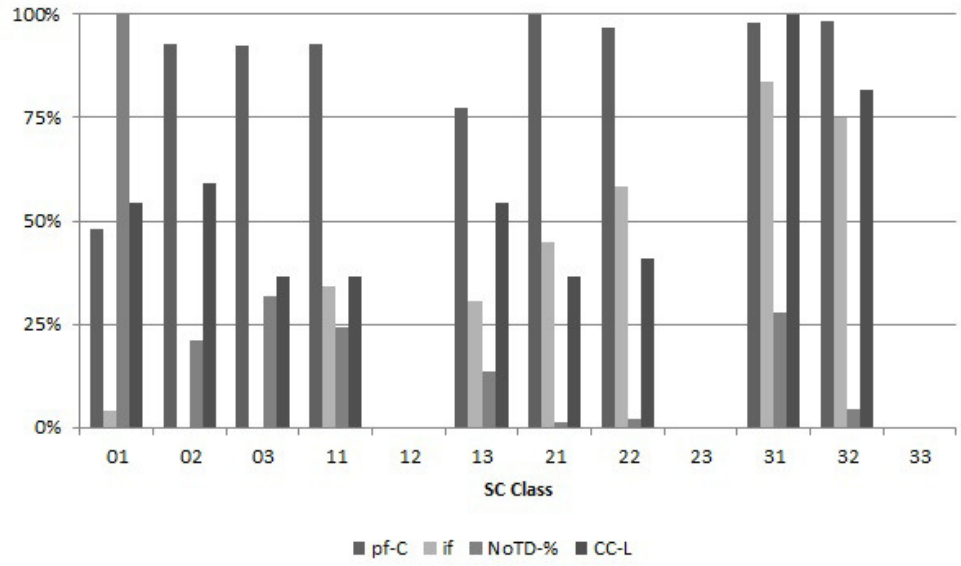


Figure 5.8: SC Classification

$0.0777x^2 + 0.3858x + 0.3924$; $R^2 = 1$) as the number of loops increases, while the number of infeasible paths (if) increases exponentially (Expon. (if); $y = 0.0022e^{1.5153x}$; $R^2 = 0.9583$). The number of test data ($notd$) increases polynomially (Poly. (notd); $y = 0.0046x^3 + 0.0562x^2 - 0.4081x + 0.6021$; $R^2 = 1$) as the numbers of infeasible paths and loops increase. Further, if any of the paths is infeasible then the search will necessarily continue for the maximum number of generations, so *notd* reaches the maximum number of test data; while in the figure the number of test data is only considered until the last path was found irrespective of having infeasible paths or not.

A two stage analysis has been conducted to evaluate the experimental results.

The first stage is to see how **Struct** and **Expr** affect the difficulty of the search problem, as represented by the numbers of feasible paths (fp) and infeasible paths (if). Table 5.5 and Table 5.6 present the regression

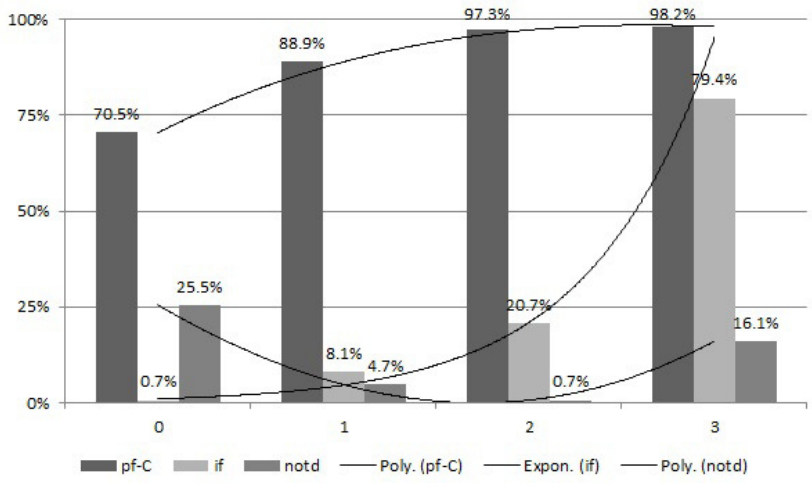


Figure 5.9: Structure Classification

analysis of **Struct** and **Expr** against fp and of **Struct** and **Expr** against if, respectively.

Table 5.5: The regression of **Struct** and **Expr** against fp

Element	**Coefficient**	**p-value**
Intercept	14.556	0.007
Struct	-2.914	0.137
Expr	-1.110	0.654

In Table 5.5, only the intercept has p-value below 0.05, representing statistical significance: neither **Struct** nor **Expr** significantly affects fp. Further, the regression adjusted R^2 value 0.0313 is very low, which means that **Struct** and **Expr** explain almost none of the variation in fp.

Table 5.6: The regression of **Struct** and **Expr** against if

Element	**Coefficient**	**p-value**
Intercept	-5.148	0.261
Struct	7.023	0.001
Expr	2.987	0.202

In Table 5.6, the p-value of **Struct** is significant, which suggests that

Struct is relevant to (or does affect) if. However, **Expr** does not affect if. In addition, the adjusted R^2 value is 0.45, so the model explains nearly half of variation in if.

Analysis of Variance (ANOVA) showed no significant relationship between fp and any of **Struct** (p-value = 0.137), **Expr** (p-value = 0.654), and their interaction (p-value = 0.116). For if, there was again no significant relationship with **Expr** (p-value=0.202) or the interaction term (p-value=0.9999), but there was a significant relationship with **Struct** (p-value=0.0007). This confirms the regression analysis.

Suppose we consider the percent of total paths that are infeasible (if percent), instead of just the raw number of infeasible paths. The results are the same: the p-value of **Struct** is significant (below 0.05), while the p-values of **Expr** and the interaction term are not.

To summarise, the regression analysis and ANOVA have confirmed the following findings: **Struct** is significantly related to if but not to fp, **Expr** is not significantly related to either of fp and if, and there is no significant interaction between **Struct** and **Expr**.

The second analysis is to see if and how **Struct** and **Expr** affect the performance of the searching system. This is done by investigating whether there is a relationship between **Struct** and **Expr** and the pf and **notd**.

The rationale for this is that **Struct** and **Expr** may affect fp and if (we have seen that **Struct** does, but **Expr** does not. In turn, fp and if have an indirect effect on pf and **notd**: fp puts an upper limit on the value of pf, and the presence of infeasible paths means that a stopping condition is

needed and this affects **notd**.

Tables 5.7 to 5.9 present the regression analysis of **Struct** and **Expr** against *pf* and **notd** using **best**, common (**comm**), and common only (**commo**) parameter values.

Table 5.7: The regression of **Struct** and **Expr** against **notd-best**

Element	**Coefficient**	**p-value**
Intercept	14489.83	0.095
Struct	-4489.94	0.178
Expr	-3428.61	0.422

In Table 5.7, none of the p-values is below 0.05, so neither **Struct** nor **Expr** significantly affect **notd**. Furthermore, the adjusted R^2 value is very low, and the p-value of ANOVA for the interaction term is 0.317 (well above 0.05), which means **Struct** and **Expr** can be considered as independent variables.

Table 5.8: The regression of **Struct** and **Expr** against **notd-comm**

Element	**Coefficient**	**p-value**
Intercept	8927.09	0.0009
Struct	-2315.19	0.0165
Expr	-1931.86	0.1075

In Table 5.8, the p-values of the intercept and **Struct** are well below 0.05, so **Struct** affects **notd**. However, the adjusted R^2 value is 0.28 (quite low). The p-value of ANOVA for the interaction term is 0.388, suggesting again that there is no significant interaction between **Struct** and **Expr**.

The results in Table 5.9, are the same as in Table 5.8: the p-values of the intercept and **Struct** are significant, **Expr** is not significant, the adjusted

Table 5.9: The regression of **Struct** and **Expr** against **notd-commo**

Element	**Coefficient**	**p-value**
Intercept	8319.64	0.0047
Struct	-2176.93	0.0442
Expr	-1180.36	0.3797

R^2 value is low (0.15), and the p-value of ANOVA for the interaction term is not significant (0.39).

Since **Expr** is not seen to be significantly related to any of the dependent variables of interest, i.e. fp, if, pf, and **notd**, the rest of the analysis of the performance of the system concentrates on **Struct** only.

The regression equation relating **Struct** to pf, using best parameter values, has a very low adjusted R^2 value (0.058). The p-value of ANOVA for **Struct** is 0.152 (not significant) and for the intercept it is 0.0002 (significant).

The regression equation relating **Struct** to pf, using common parameter values, has an even lower adjusted R^2 value (-0.038). The p-value of ANOVA for **Struct** is 0.608 (not significant) and for intercept is 0.00006 (significant). These results are identical when using common parameter values but excluding any test program that has infeasible paths.

In summary, **Expr** is not significantly related to the difficulty of the search (fp and if), or to the performance of the searching system (pf or **notd**, for all parameters variations). **Struct** is significantly related to if and if percent: as **Struct** increases, the number of infeasible paths increases. **Struct** is also weakly related to **notd**, with all parameters variations. There is no significant interaction between **Struct** and **Expr**, so they may be considered as two independent variables.

5.4 Analysis

The analysis presented in this section is based on best parameter settings, test program characteristics, path testing adequacy, and search technique used.

Recall, as seen in Table 5.1, population size can be set to 100 for low or medium CC programs, and 250 for medium or high CC programs. The number of generations can be set to 50 for low or medium CC programs, or 500 for medium or high CC programs. Having allele range as narrow as possible is always better, as long as the region of feasible solutions is not removed. Higher mutation rate helps for some test programs, but broadly it can just be set to one reasonable value.

Recall that the common parameter values are as follows: **Pop** 100, **Gens** 100, **Allele** about 200 range, crossover rate 0.9, and **Mut** 0.15. Table 5.10 shows the path coverage when all test programs are processed using this parameter setup.

Comparing **PF** with best parameters and common parameters, paired t-test results in P(T≤t) two-tail = 0.07, which is more than 0.05. This suggests that there is no statistically significant different in **PF** between using the best and common parameters setup. However, the best parameters setup is usually better on average.

Using common parameters reduces computational time, by avoiding the need of fine tuning to find the best parameters. Moreover, knowing that there is no significant difference between using best and common parameters gives confidence that using common parameters setup will not significantly

Table 5.10: Path Coverage using Common Parameter Values

Program	**Feas**	**Allele**	**PFM**	**PF**	**MPFM**	**MPF**	**CCL**	**CCU**
tA2008	4	0 100	4	3.70	0	0.30	4	13
mmA2008	13	-100 100	13	11.80	0	1.20	4	4
iA2008	5	-100 100	5	5.00	0	0.00	4	5
bisA2008	9	1.72 1.75	6	6.00	3	3.00	5	7
binA2008	7	-100 100	7	7.00	0	0.00	3	3
bubA2008	4	-100 100	4	4.00	0	0.00	4	5
gA2008	5	0 200	5	3.57	0	1.43	4	4
rA2008	4	0 200	4	3.97	0	0.03	4	4
mtA2008	20	0 200	18	17.53	2	2.47	7	16
tM2004	7	0 200	6	4.07	1	2.93	5	5
eiR1985	3	0 200	3	3.00	0	0.00	4	4
qG1997	4	1 200	4	4.00	0	0.00	4	4
ttB2002	8	0 200	8	7.03	0	0.97	8	11
eiB2002	5	-100 100	5	4.90	0	0.10	11	15
qB2002	10	0 200	10	8.63	0	1.37	6	7
scB2002	4	1 128	4	3.83	0	0.17	5	5
fcB2002	5	-100 100	5	4.90	0	0.10	5	8
fB2002	8	1 200	8	7.87	0	0.13	9	10
bG2011	11	-100 100	11	11.00	0	0.00	4	4
fG2011	30	-100 100	13	9.30	17	20.70	7	7
sG2011	32	-100 100	32	17.77	0	14.23	6	6

reduce effectiveness.

Test program characteristics can be used to describe computational complexity of generating test data for required paths. Looking at Table 5.2, CC alone (either **CCL** or **CCU**) could not be used to estimate test program complexity with respect to path testing. There is no connection, because they are not dependent on one another, between **CCL** and/or **CCU** and **MPFM** and/or **MPF**. However, CC, number of loops, and loop configura-

tion together are good candidates to measure the complexity.

Increasing the population size increases effectiveness more than increasing the number of generations. According to the schemata theorem of genetic algorithms, the number of good/promising chromosomes is proportional to population size. For example, Figure 5.6(a) shows that population variation matters more to path coverage than generation variation as can be seen in Figure 5.6(b). A formula for deriving suitable population size was presented by Goldberg et. al. [156]. The population sizes we studied align with that formula, i.e. population size corresponds to promising input data.

Narrowing the allele range significantly improves the effectiveness of the test data generator. Narrowing the allele range means reducing the search space itself. For example, in mmA2008, narrowing the allele range from [−100 100] to [−10 10] increases the number of runs that cover 13 paths from 6 runs to 30 runs. However, it is more risky to do this, because optimal solution(s) can be missed unless the domain of the search space, and possibly optimal solution location(s), are well known.

Infeasible paths do not always mean a flawed algorithm, but if one of them appears then it is worth investigating to ensure that the algorithm is not incorrect.

Loops contribute more to path infeasibility than selections. Most of the time, infeasible paths caused by loop(s) do not mean faults, it is merely that their input is outside the feasible space.

On the contrary, selection-based infeasible paths have higher chance to identify faulty logic, although this does not always apply. In reality, if a

program contains both loops and selections then most likely it will have infeasible paths with intertwined paths between loops and selections. For example, eR1985 has two paths, i.e., [1 1 2 1 3 0] and [1 1 2 1 3 1], that need to traverse to FALSE decisions of both the first and the second branches (see its target paths in the Appendix B.9). These paths are infeasible because both branches hold some expressions that do not allow both of them to go to FALSE decisions at the same time. Therefore, there is nothing wrong with them because they are infeasible by design.

GA has following characteristics [45]: incompleteness, probabilistic, and randomness. This means that (1) whenever a path is not found it does not mean that the path is infeasible, it just may not have been found yet; and (2) running GA with the same parameters but different seeds can give different outputs. As an example for (1), 3 feasible paths in bisA2008 are always covered last if at all, because all of them require precise zero input values that are calculated from the previous statements. The example for (2) is that each experiment treatment in this research is always run 30 times with different random seed each time and all of them produced different outputs.

5.5 Threats to Validity

There are at least two threats to the validity of the approach: program scaling, and determining the range of values for variable-length chromosomes.

As the test program gets larger, sizing population, determination of allele range, and generating target paths may be slightly different. There should be a more rigorous or formulated approach for deciding what must be the

range of chromosome length for test programs that require variable-length chromosomes.

The approach used in the experiments to decide the range is done manually, based on the number of loops required by target paths. In sorting algorithms such as iA2008 and bubA2008, the input length varies from 1 to 6 to satisfy the different loops in the target paths. For example, one path can require 1 input, such as [2] or [30], while another one requires 3 inputs, such as [1 3 90] or [30 -4 28]. Such a manual approach is not practical for larger programs.

So, both population sizing and formulated variable length chromosome must be related to program size or at least to its input size. For example, algorithms that take various number of inputs must use variable-length chromosomes, such as mmA2008, iA2008, binA2008, bubA2008, fB2008, and bG2011. Further, as the range of variable-length chromosome increases the population size must increase too.

5.6 Conclusions

The experimental results have shown that varying the population size and the allele range generally has significant impact on path testing performance, while varying the number of generations and mutation rate has less impact in term of number of paths found.

Population size and number of generations can be set to 100 or 250 and 50 or 500, respectively, for low-medium or medium-high complexity programs. The narrowest possible allele range really helps the search, as long as feasible

solution regions are not omitted as a result. Mutation rate can be set to a single reasonable value.

CC alone is not enough to estimate complexity of test program with respect to path testing. It has to be combined with some other measures that give more information on path coverage.

These conclusions will assist us in future work on GA-based path testing. Further investigation is needed to understand test program complexity, such as combination of CC, number of loops, and loops configuration. Further work can also involve how to decide whether a path is infeasible whenever no test data has been found for it; and whether other types of search techniques, for example combining GA with local search, are suitable for path testing. These are the subjects of later chapters.

Chapter 6

Automating the Decision of When to Stop Searching

6.1 Introduction

Any evolutionary algorithm requires one or more stopping criteria to halt the evolution [45].

Most real-world problems are computationally expensive and constrained. An evolutionary algorithm should be stopped as soon as a pseudo-optimal solution seems to have been detected, because the actual optimum is unknown and there may be no assurance that further improvement is likely, or even possible.

The objective of this chapter is to develop criteria to halt the evolutionary test data generation process as soon as it seems not worth continuing, without compromising testing confidence level. This can be achieved if there

is a mechanism that can predict the likelihood of producing test data that can cover target paths that are not yet covered, in the coming evolutionary processes.

The idea that is proposed in this chapter was inspired by reliability models used in software testing [58]. These models estimate the future reliability of software, based on the past history of what errors were found, and when. Originally, such models were proposed to forecast logarithmically decaying phenomena that are common in nature.

In software testing, reliability means the probability of failure-free operation of software at a particular time in a certain environment: in other words, the probability that no new faults will be encountered as the software continues to execute beyond a given time. The analogy to test data generation is to determine the probability that no new paths will be covered as the search continues for further generations. When this probability falls below a certain threshold, the search can cease. The assumption is that any paths that are not yet covered at the time when searching stops are infeasible.

In path testing, it is most likely that some target paths will be infeasible. The existence of an infeasible path means that it can never succeed to search until all target paths are covered: searching beyond covering the last feasible path is worthless. To keep searching for infeasible paths is useless and wastes resources for no possible gain. Any criteria that can stop the process earlier, without compromising testing reliability, i.e. by not missing any feasible paths, will be able to save much time and cost. In particular, stopping criteria that are based on the information obtained during the test process itself (i.e. number of failures found, and when), are desirable, rather

than using arbitrary values for stopping parameters. Moreover, having these stopping criteria at hand can identify potentially infeasible paths, though further analytical investigation is required to be sure.

6.2 Approach

6.2.1 Model fitting

Estimating the number of failures expected in a testing interval is analogous to estimating the number of paths to be covered in a test data generation. To complete the analogy, we must identify what things in reliability growth models map to corresponding things in evolutionary path coverage.

In Section 3.4, the reliability growth model defines that $\lambda(\tau)$ represents the expected number of failures λ at a certain point in time τ. In evolutionary path coverage, $\lambda(\tau)$ is to be the expected number of previously-uncovered target paths λ that are covered at a certain generation τ. λ_0 is the number of paths that are covered in the first generation. θ is the rate of reduction in the normalized path coverage rate. Both values of λ_0 and θ can be obtained from the data.

In our experiments the value of θ is initially set to 0.3, because preliminary investigation shows that the average number of paths covered from one finding generation to the next is always about a third across all the test programs. This value is refined after each new generation.

Every new generation adds a tuple to the series of (generation number, newly covered paths) pairs ("generations-paths pairs"). For example, if 13

paths were covered in the first generation, no new paths were covered in generations 2 to 4, and 2 new paths were covered in generation 5, the first five tuples in the series would be {(1,13), (2,0), (3,0), (4,0), (5,2)}.

A series of curves can be constructed by fitting the series of generation-paths pairs. For example, after two generations a curve can be fitted based on the points {(1,13), (2,0)}. After 3 generations, a curve can be fitted based on three points {(1,13), (2,0), (3,0)}, and so on.

From this equation, a new refinement or estimate of θ can be calculated. The curve can be used either to predict the number of new paths found in the next generation, using Equation 2.4, or the number of generations required to cover a certain number of paths, using Equation 6.1.

In order to visualize the approach, let us plot a few curves for the following set of data using Equation 2.4. Suppose the (x, y) pairs of generation number x and non-zero number of paths found y are {(1, 13), (5, 2), (16, 1), (22, 1)}; for other x values up to 25 the y value is zero. Figure 6.1 shows the set of curves fitted as the number of data points increases from 2 to 25. The topmost curve has the least number of pairs of data, and the bottom one has the most pairs of data present. As the number of pairs of data increases, the curves tend to converge. In this example, the curves converge in fewer than 10 generations.

6.2.2 Decision Rules

One of the main research questions is how to make use of the model to stop the evolutionary path testing as soon as it is identified that it is not useful to

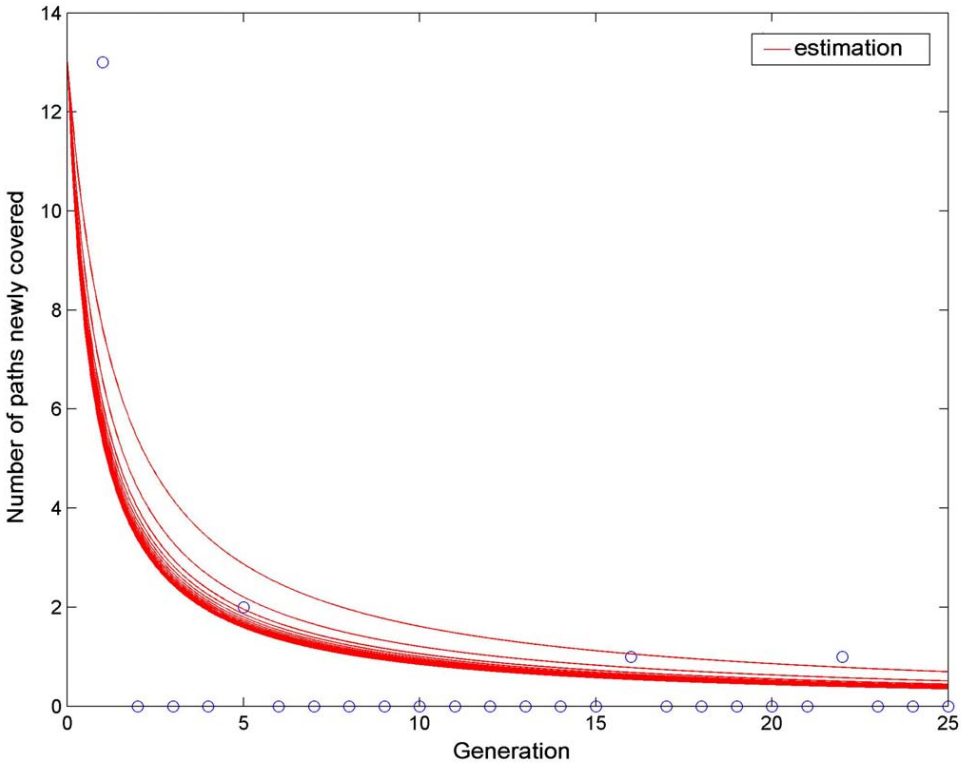


Figure 6.1: Plot of $\lambda(\tau)$

continue the search any more. This means a decision rule must be defined.

We investigate two proposed decision rules, separately and then in combination.

6.2.2.1 Reliability Rule:

One decision rule is to stop the search when the probability of finding new paths $\lambda(\tau)$ in generation τ, i.e. Equation 2.4, is small enough, as indicated by reaching a certain threshold. This is called (coverage) **reliability rule**, because it also can be interpreted as at what generation $\tau(\lambda)$ will a given reliability λ be achieved. This is stated in Equation 6.1, which is derived

from Equation 2.4.

$$\tau(\lambda) = \frac{\frac{\lambda_0}{\lambda} - 1}{\lambda_0 \theta} \tag{6.1}$$

6.2.2.2 Stability Rule:

Another rule is to stop as soon as the fitted curves from consecutive generations have converged sufficiently i.e. when the change in θ from one generation to the next falls below a threshold.

6.2.2.3 Reliability Stability Rule:

This rule combines the above two rules, to require that both the Reliability Rule and the Stability Rule be satisfied. The Stability Rule means that extrapolations are stable, and the Reliability Rule means that the chance of finding test data to cover newly uncovered paths is low.

6.3 Experiments

The purpose of the experiments is to study the proposed method as an alternative stopping criterion in evolutionary path testing. What sort of information is most helpful that can be extracted from the model? What confidence level can be achieved about feasible and infeasible paths should the approach be applied? How does the performance vary across different runs for the same test program, or across different test programs?

6.3.1 Test Programs

The same 21 test programs studied in Chapter 5 were used to answer these questions. Seven of them have no infeasible paths: tA2008, mmA2008, binA2008, ttB2002, fcB2002, fG2011, and sG2011. The rest have one or more infeasible paths.

Test programs are divided into two groups, for parameter tuning and evaluation. The parameter tuning group is used to identify appropriate thresholds for the decision rules (what threshold value of λ is best for deciding that further searching is not worthwhile, and what threshold change in θ is best for judging that a model is stable). The evaluation group is used to evaluate the resulting rules.

The parameter tuning group consists of mmA2008, iA2008, bisA2008, and mtA2008. These four programs cover the set of different program characteristics, making them a reasonable training set: some include infeasible paths, and others do not; they have a range of CC, and all include loops. The rest form the evaluation group.

6.3.2 Setup

The aim of the experiments is to find useful decision rules, as described in Section 6.2.2. The main aim is to find optimal thresholds for the rules and to investigate their respective strengths and weaknesses.

In the parameter tuning phase, 30 runs were made with each of the four programs selected for tuning. The best threshold settings were identified in this phase. 30 runs were then made with these settings, for each evaluation

program.

The population size for each tuning program varied from 50 to 250, depending on the program, using the best values identified in Table 5.1 and as suggested in [23]. The tuning programs were set up using the best parameter values while the evaluation programs were investigated using both the best and common parameters values.

For the reliability rule, three different threshold values were investigated for λ: 0.5 (meaning 0.5 probability that a new path is found in the next generation of searching), 0.25, and 0.1. There is a trade-off: it takes more generations to reach a lower threshold, but there is less chance of missing feasible paths by stopping early.

For the stability rule, θ starts at 0.3 as mentioned in Section 6.2.1. The value of θ is refined each time the model is updated. We observe that each time the model is refitted, the curve becomes more stable. A stable curve means that the expected number of paths to be covered will not change much in many more generations to come. This is one of the signs to stop looking for new paths. This stability can be detected by small changes in θ values across generations or so called $\Delta\theta$. As the required $\Delta\theta$ must be small enough, without prior information on how small it should be, experiments are needed to identify a good threshold for $\Delta\theta$. So, six thresholds were investigated for $\Delta\theta$: $\frac{1}{1000}$ (=0.001), $\frac{1}{2000}$ (=0.0005), $\frac{1}{3000}$ (=0.00033), $\frac{1}{4000}$ (=0.00025), $\frac{1}{5000}$ (=0.0002), and $\frac{1}{10000}$ (=0.0001).

6.4 Experimental Results (Stopping Criteria)

6.4.1 Performance measures

In this section, various statistics are reported.

Some represent parameter settings:

- **Gens**: the number of generations at which searching will stop. If all target paths are covered sooner, searching stops straight away; otherwise it continues until this number of generations. **Gens** is set to 100 throughout, based on the best values identified in Table 5.1 and experience from [23];
- the threshold applied to the value of λ;
- the threshold applied to the value of $\Delta\theta$.

The rest report performance:

- **PFM** (paths found maximum): the maximum number of target paths found (across 30 runs);
- **PF** (average paths found): the mean number of target paths found (averaged across 30 runs);
- **PFMR** (paths found maximum using the rule): the maximum number of target paths found if the rule is applied (across 30 runs);
- **PFR** (average paths found using the rule): the mean number of target paths found if the rule is applied (averaged across 30 runs);

- **GR** (number of generations using the rule): the number of generations at which the decision rule suggests that searching should stop (averaged across 30 runs). Note that it is possible for **GR** to be greater than **Gens**, if the history suggests that searching is still worthwhile when **Gens** generations are reached;

- **MP** (missed feasible paths): the number of feasible target paths that are missed by stopping early instead of going through to **Gens** generations (averaged across 30 runs);

- **MTP** (missed target paths): the number of test programs that miss out some of their target paths;

- **MPP** (missed feasible paths in percentage): **MP** in percentage;

$$\textbf{MPP} = \frac{\textbf{MP}}{\textbf{PF}} \times 100 \tag{6.2}$$

- **Eff** (efficiency), efficiency achieved: the fraction of total execution time that is saved by stopping early instead of going through to **Gens** generations (averaged across 30 runs);

$$\textbf{Eff} = \frac{\textbf{Gens - GR}}{\textbf{Gens}} \tag{6.3}$$

- **Ex-Eff** (efficiency that excludes inefficiency): exclusive **Eff** that excludes negative efficiency;

- **ITP** (inefficient test programs): number of test programs that have negative efficiency (i.e. searching continues beyond **Gens** generations) when stopping rule is applied.

6.4.2 Execution time

Another important aspect of implementing a rule is its computational time. In other words, rule execution should not cause detrimental computation overhead, which would mean that the benefit of saving time from stopping the search earlier is not significant.

Experiments show that the execution time cost involved in fitting curves and making early-stopping decisions is very small. In our research environment, the average execution time over 30 runs per GA generation of mtA2008 using best parameters setup was about 9.72 seconds, whereas the average execution time for curve fitting was about 0.05 seconds. Thus, any cost incurred in applying the decision rules considered here is insignificant.

6.4.3 Training programs

6.4.3.1 Reliability Rule (RR)

Table 6.1 summarizes the training result for RR. At $\lambda = 0.1$, there is very little chance that a path is missed, but that level of confidence is reached by searching for longer so the efficiency is lower, i.e. 40.2% (the average of **Eff** at $\lambda = 0.1$ across 4 training programs, shown in Table 6.1) compared to $\lambda = 0.5$ and $\lambda = 0.25$ which achieve 88.3% and 76.2%, respectively.

$\lambda \leq 0.25$ is a reasonable trade-off between efficiency (above 60% for all four programs) and getting an effective result (3% chance or less of missing paths, in all four programs). Based on this, $\lambda \leq 0.25$ and $\lambda \leq 0.10$ are selected for thorough testing.

Table 6.1: Training Programs, Reliability Rule

Program	**Gens**	λ	**GR**	**PFR**	**MPP** (%)	**Eff** (%)
mmA2008	100	0.50	16.00	12.93	0.54	84.00
		0.25	32.23	13.00	0.00	67.77
		0.10	80.94	13.00	0.00	19.06
iA2008	100	0.50	6.29	5.00	0.00	93.71
		0.25	13.20	5.00	0.00	86.80
		0.10	33.10	5.00	0.00	66.90
bisA2008	100	0.50	6.51	5.43	5.24	93.49
		0.25	13.39	5.57	2.97	86.61
		0.10	34.04	5.67	1.22	65.96
mtA2008	100	0.50	18.02	16.62	3.37	81.98
		0.25	36.31	17.13	0.41	63.69
		0.10	91.09	17.20	0.00	8.91

6.4.3.2 Stability Rule (SR)

Table 6.2 summarizes the result of the SR rule with the four training programs. In the first three training programs, $\Delta\theta \leq 0.00025$ (column $\Delta\theta$) is the point at which no missing path is encountered (column **MPP(%)**) and yet it still has high efficiency on average. On the other hand, $\Delta\theta \leq 0.0001$ has the fewest number of missed paths. So, $\Delta\theta \leq 0.00025$ and $\Delta\theta \leq 0.0001$ are selected for thorough testing.

6.4.3.3 Reliability Stability Rule (RSR)

Table 6.3 summarizes the RSR rule with the training programs. The thresholds of λ and $\Delta\theta$ for RSR are as identified immediately above. With these thresholds, there are no missing paths in the first three training programs, and efficiency above 50% is achieved for all four programs.

Table 6.2: Training Programs, Stability Rule

Program	**Gens**	$\Delta\theta$	**GR**	**PFR**	**MPP** (%)	**Eff** (%)
mmA2008	100	0.001	13.74	12.93	0.54	86.26
		0.0005	21.07	12.93	0.54	78.93
		0.00033	26.03	12.93	0.54	73.97
		0.00025	30.03	13.00	0.00	69.97
		0.0002	33.33	13.00	0.00	66.67
		0.0001	43.17	13.00	0.00	56.83
iA2008	100	0.001	22.41	5.00	0.00	77.59
		0.0005	31.55	5.00	0.00	68.45
		0.00033	39.41	5.00	0.00	60.59
		0.00025	44.69	5.00	0.00	55.31
		0.0002	49.69	5.00	0.00	50.31
		0.0001	70.82	5.00	0.00	29.18
bisA2008	100	0.001	23.47	5.70	0.52	76.53
		0.0005	34.93	5.70	0.52	65.07
		0.00033	42.07	5.82	0.00	57.93
		0.00025	47.10	5.79	0.00	52.90
		0.0002	53.63	5.80	0.00	46.37
		0.0001	72.50	5.73	0.00	27.50
mtA2008	100	0.001	12.17	15.67	8.90	87.83
		0.0005	20.58	16.70	2.91	79.42
		0.00033	24.04	16.88	2.03	75.96
		0.00025	27.32	16.82	2.09	72.68
		0.0002	31.76	17.04	0.47	68.24
		0.0001	64.15	17.22	0.06	35.85

Table 6.4 presents the distribution of **GR** across all runs with $\lambda \leq 0.25$ and $\Delta\theta \leq 0.00025$, i.e. minimum **Min**, maximum **Max**, average **Mean**, and standard deviation **Stdev**. It shows that **GR** has little variation, which means the prediction is quite stable, with **StDev** about 1 generation across all runs for all four programs.

Table 6.3: Training Programs, RSR Rule

Program	**Gens**	**GR**	**PFR**	**MPP** (%)	**Eff** (%)
mmA2008	100	30.53	13.00	0.00	69.47
iA2008	100	22.41	5.00	0.00	77.59
bisA2008	100	33.00	6.00	0.00	67.00
mtA2008	100	32.33	16.67	1.92	67.67

Table 6.4: Training Programs, GR Distribution

Program	**Min**	**Max**	**Mean**	**Stdev**
mmA2008	30	31	30.53	0.51
iA2008	22	25	22.41	1.05
bisA2008	33	33	33.00	0.00
mtA2008	32	33	32.33	0.58

As just noted, two **RSR**s are investigated. Based on the training results, the focus will be the combination of **RSR**s and there are two λ values and two $\Delta\theta$ values that construct two **RSR** combinations, i.e. $\lambda \leq 0.25$ and $\Delta\theta \leq 0.00025$ (**RSR1**), and $\lambda \leq 0.1$ and $\Delta\theta \leq 0.0001$ (**RSR2**).

6.4.4 Testing programs

Each of these **RSR**s will be executed using the best and common parameters setups. So, total number of testing combinations is four, i.e. **RSR1** with best parameters (**RSR1-B**), **RSR1** with common parameters (**RSR1-C**), **RSR2** with best parameters (**RSR2-B**), and **RSR2** with common parameters (**RSR2-C**).

Table 6.5 presents the results for **RSR1-B** for all 21 test programs. Mostly, efficiency is not relevant for programs with no infeasible paths because the search will stop as soon as all target paths are covered; this is

usually before **GR** would suggest stopping. However, a few runs of programs with no infeasible paths still can go beyond **Gens** generations: in six test programs, i.e. rA2008, mtA2008, tM2004, ttB2002, fB2002, and fG2011, some feasible paths have been missed in a very few runs, that would have been found if the search had not stopped early according to **RSR1-B**. On average, the number of missed paths is 0.48. Two test programs have negative efficiencies, i.e. mtA2008 and sG2011. A negative efficiency means that the suggested number of generations by the rule is higher than the default 100 generations. mtA2008 has more infeasible target paths (above 60% of the total target paths) than feasible ones; this could be one of the factors that made the rule suggest stopping at more than 100 generations. As for sG2011, even though it has no infeasible paths, it is still not efficient. This could be due to the high number of target paths, i.e. 32 paths. On average, the efficiency is 59% and **Ex-Eff** is 72.85%.

Table 6.6 shows the results for **RSR1-C**. As expected, in comparison with **RSR1** with best parameters, the number of missing paths is not as good: it increases to 0.84 on average. The number of test programs that miss some feasible paths has doubled; from six to ten test programs. They are gA2008, rA2008, mmA2008, tM2004, eiR1985, qB2002, scB2002, fB2002, fG2011, and sG2011. However, only 3 out of 8 are the same as the test programs that miss the feasible paths in **RSR1-B**. On average, the efficiency is 74%, with no inefficiency at all.

Table 6.7 displays the results for **RSR2-B**. This has the most stringent thresholds, and the best parameter settings for each test program, so it would be expected to miss the fewest feasible paths. The number of missing paths is

Table 6.5: Testing Programs RSR1-B

No	**Program**	**PFM**	**PF**	**PFMR**	**PFR**	**GR**	**MP**	**Eff** %
1	tA2008	4	4.00	4	4.00	25.83	0.00	89.67
2	mmA2008	13	13.00	13	13.00	30.00	0.00	70.00
3	iA2008	5	5.00	5	5.00	23.00	0.00	54.00
4	bisA2008	6	6.00	6	6.00	20.93	0.00	79.07
5	binA2008	7	7.00	7	7.00	19.10	0.00	61.80
6	bubA2008	4	4.00	4	4.00	29.00	0.00	42.00
7	gA2008	5	5.00	5	5.00	22.10	0.00	55.80
8	rA2008	4	4.00	4	3.97	26.27	0.03	94.75
9	mtA2008	20	19.00	20	18.88	78.79	0.12	-57.58
10	tM2004	7	6.23	7	6.21	21.34	0.02	57.32
11	eiR1985	3	2.97	3	2.97	29.37	0.00	41.26
12	qG1997	4	4.00	4	4.00	25.00	0.00	50.00
13	ttB2002	8	8.00	8	7.53	19.11	0.47	96.18
14	eiB2002	5	5.00	5	5.00	22.00	0.00	95.60
15	qB2002	10	10.00	10	10.00	27.07	0.00	94.59
16	scB2002	4	4.00	4	4.00	27.67	0.00	94.47
17	fcB2002	5	5.00	5	5.00	22.40	0.00	95.52
18	fB2002	8	7.87	8	7.67	20.61	0.20	79.39
19	bG2011	11	11.00	11	11.00	27.33	0.00	45.34
20	fG2011	30	25.13	18	15.83	31.33	9.30	87.47
21	sG2011	32	32.00	32	32.00	95.40	0.00	-90.80

0.18 on average, and there are only four test programs that miss any feasible paths, i.e. ttB2002, mtA2008, fB2002, and fG2011. In term of efficiency, more test programs were less efficient, i.e. 9 test programs. This is due to more restricted rule in setting up number of generations as more feasible paths were found as the search progresses. On average, total efficiency is 8.74% and **Ex-Eff** 66.86% on average.

Table 6.8 exhibits the results for **RSR2-C**. This has the same stringent

Table 6.6: Testing Programs RSR1-C

No	Program	PFM	PF	PFMR	PFR	GR	MP	Eff %
1	tA2008	4	3.83	4	3.83	29.53	0.00	70.47
2	mmA2008	13	11.63	11	11.41	32.5	0.22	67.50
3	iA2008	5	5.00	5	5.00	22.50	0.00	77.50
4	bisA2008	6	6.00	6	6.00	20.90	0.00	79.10
5	binA2008	7	7.00	7	7.00	19.00	0.00	81.00
6	bubA2008	4	4.00	4	4.00	25.23	0.00	74.77
7	gA2008	5	3.57	1	1.00	6.57	2.57	93.43
8	rA2008	4	3.97	4	3.80	56.80	0.17	43.20
9	mtA2008	18	17.53	18	17.53	67.16	0.00	32.84
10	tM2004	6	4.20	6	3.90	37.83	0.30	62.17
11	eiR1985	3	2.97	1	1.00	6.19	1.97	93.81
12	qG1997	4	4.00	4	4.00	25.00	0.00	75.00
13	ttB2002	8	7.03	8	7.03	19.13	0.00	80.87
14	eiB2002	5	4.90	5	4.90	23.83	0.00	76.17
15	qB2002	10	8.63	8	8.00	20.00	0.63	80.00
16	scB2002	4	3.83	1	1.00	7.00	2.83	93.00
17	fcB2002	5	4.90	5	4.90	25.16	0.00	74.84
18	fB2002	8	7.87	8	7.67	20.61	0.20	79.39
19	bG2011	11	11.00	11	11.00	27.62	0.00	72.38
20	fG2011	13	9.30	12	7.69	26.31	1.61	73.69
21	sG2011	32	17.77	18	10.69	27.30	7.08	72.70

thresholds as **RSR2-B**, but uses the common parameter setting. It has the fewest missing paths on average, i.e. 0.1 paths. It has the same number (8) of test programs that miss some feasible paths as **RSR1-C**, and more programs that miss some feasible paths than **RSR2-B**. Only 3 of them match the test programs that miss feasible paths with **RSR1-C**. On average, total efficiency is 32% and **Ex-Eff** is 34.59% on average.

Table 6.9 shows which test programs have missed one or more feasible

Table 6.7: Testing Programs RSR2-B

No	**Program**	**PFM**	**PF**	**PFMR**	**PFR**	**GR**	**MP**	**Eff** %
1	tA2008	4	4.00	4	4.00	51.37	0.00	79.45
2	mmA2008	13	13.00	13	13.00	76.27	0.00	23.73
3	iA2008	5	5.00	5	5.00	72.13	0.00	-44.26
4	bisA2008	6	6.00	6	6.00	64.90	0.00	35.10
5	binA2008	7	7.00	7	7.00	58.10	0.00	-16.20
6	bubA2008	4	4.00	4	4.00	88.00	0.00	-76.00
7	gA2008	5	5.00	5	5.00	70.20	0.00	-40.40
8	rA2008	4	4.00	4	4.00	55.17	0.00	88.97
9	mtA2008	20	19.00	20	18.87	112.70	0.13	-125.40
10	tM2004	7	6.23	7	6.23	66.57	0.00	-33.14
11	eiR1985	3	2.97	3	2.97	89.53	0.00	-79.06
12	qG1997	4	4.00	4	4.00	32.67	0.00	34.66
13	ttB2002	8	8.00	8	7.87	58.60	0.13	88.28
14	eiB2002	5	5.00	5	5.00	70.00	0.00	86.00
15	qB2002	10	10.00	10	10.00	61.93	0.00	87.61
16	scB2002	4	4.00	4	4.00	48.52	0.00	90.30
17	fcB2002	5	5.00	5	5.00	70.83	0.00	85.83
18	fB2002	8	7.87	8	7.86	61.38	0.01	38.62
19	bG2011	11	11.00	11	11.00	66.80	0.00	-33.60
20	fG2011	30	25.13	30	21.61	90.60	3.46	63.76
21	sG2011	32	32.00	32	32.00	135.37	0.00	-170.74

paths with which **RSR**. Only one test program misses feasible paths with every **RSR**.

Table 6.10 shows statistics of rules in term of **MP**. Although both **RSR1-C** and **RSR2-C** have the same size of **ITP**, **RSR2-C** has the fewest missing paths on average, i.e. 0.1 paths. Further, it also has the lowest standard deviation on average, i.e. 0.17 paths.

Table 6.11 presents statistics of rules in term of **Eff**. **RSR1-C** is the

Table 6.8: Testing Programs RSR2-C

No	Program	PFM	PF	PFMR	PFR	GR	MP	Eff %
1	tA2008	4	3.83	4	3.83	29.53	0.00	70.47
2	mmA2008	13	11.63	13	11.57	63.50	0.07	36.50
3	iA2008	5	5.00	5	5.00	71.00	0.00	29.00
4	bisA2008	6	6.00	6	6.00	65.03	0.00	34.97
5	binA2008	7	7.00	7	7.00	59.00	0.00	41.00
6	bubA2008	4	4.00	4	4.00	41.37	0.00	58.63
7	gA2008	5	3.57	5	3.18	106.00	0.32	-6.00
8	rA2008	4	3.97	4	3.97	72.30	0.00	27.70
9	mtA2008	18	17.53	18	17.53	91.97	0.00	8.03
10	tM2004	6	4.20	6	4.10	112.83	0.10	-12.83
11	eiR1985	3	2.97	3	2.40	91.60	0.57	8.40
12	qG1997	4	4.00	4	4.00	32.67	0.00	67.33
13	ttB2002	8	7.03	8	7.03	59.53	0.00	40.47
14	eiB2002	5	4.90	5	4.90	62.57	0.00	37.43
15	qB2002	10	8.63	10	8.63	56.60	0.00	43.40
16	scB2002	4	3.83	4	3.82	60.82	0.00	39.18
17	fcB2002	5	4.90	5	4.90	68.63	0.00	31.37
18	fB2002	8	7.87	8	7.86	61.38	0.00	38.62
19	bG2011	11	11.00	11	11.00	66.73	0.00	33.27
20	fG2011	13	9.30	13	8.76	89.31	0.45	10.69
21	sG2011	32	17.77	32	15.96	99.23	0.30	0.77

most efficient rule with no test program showing any inefficiencies.

Table 6.12 summarizes **PF** and **PFR** for all test programs across all applicable rules. Using paired t-Test for means, **PFR**s for **RSR1-C** and **RSR2-C** are significantly different with $P(T \leq t) = 0.03$; that is; the more stringent rule covers significantly more paths.

Table 6.9: Test Programs with Missing Paths

No	Program	RSR1-B	RSR1-C	RSR2-B	RSR2-C
1	tA2008				
2	mmA2008		x		x
3	iA2008				
4	bisA2008				
5	binA2008				
6	bubA2008				
7	gA2008		x		x
8	rA2008	x	x		
9	mtA2008				
10	tM2004		x		x
11	eiR1985		x		x
12	qG1997				
13	ttB2002	x		x	
14	eiB2002				
15	qB2002		x		
16	scB2002		x		x
17	fcB2002				
18	fB2002	x	x		x
19	bG2011				
20	fG2011	x	x	x	x
21	sG2011		x		x

Table 6.10: Statistics of Rules on **MP**

Rule	MTP	MP	STD	MIN	MAX
RSR1-B	2	0.48	2.02	0.00	9.30
RSR1-C	0	0.84	1.69	0.00	7.08
RSR2-B	9	0.18	0.77	0.00	3.52
RSR2-C	2	0.17	0.42	0.00	1.81

6.5 Analysis

6.5.1 Test Program Classification

Analytically, a test problem can fall into the following classes. Knowing how many are in each class can help to understand the value of the proposed

Table 6.11: Statistics of Rules on Efficiency

Rule	**ITP**	**Eff**(%)	**STD**	**MIN**	**MAX**
RSR1-B	2	58.85	48.73	-90.80	96.18
RSR1-C	0	73.99	14.46	32.84	93.81
RSR2-B	9	8.74	78.28	-170.74	90.30
RSR2-C	2	31.96	23.03	-12.83	70.47

Table 6.12: Summary of PF and PFR

Program	Paths		PF		PFR			
	All	Feas	Best	Comm	RSR1-B	RSR1-C	RSR2-B	RSR2-C
tA2008	4	4	4.00	3.83	4.00	3.83	4.00	3.83
mmA2008	13	13	13.00	11.63	13.00	10.50	13.00	11.57
iA2008	6	5	5.00	5.00	5.00	5.00	5.00	5.00
bisA2008	9	9	6.00	6.00	6.00	6.00	6.00	6.00
binA2008	7	7	7.00	7.00	7.00	7.00	7.00	7.00
bubA2008	15	4	4.00	4.00	4.00	4.00	4.00	4.00
gA2008	8	5	5.00	3.57	5.00	1.00	5.00	3.18
rA2008	5	4	4.00	3.97	3.97	3.73	4.00	3.97
mtA2008	52	20	19.00	17.53	18.88	17.54	18.87	17.53
tM2004	8	7	6.23	4.20	6.21	3.90	6.23	4.10
eiR1985	12	3	2.97	2.97	2.97	1.00	2.97	2.40
qG1997	21	4	4.00	4.00	4.00	4.00	4.00	4.00
ttB2002	8	8	8.00	7.03	7.53	7.03	7.87	7.03
eiB2002	31	5	5.00	4.90	5.00	4.90	5.00	4.90
qB2002	27	10	10.00	8.63	10.00	8.00	10.00	8.63
scB2002	15	4	4.00	3.83	4.00	1.00	4.00	3.82
fcB2002	5	5	5.00	4.90	5.00	4.90	5.00	4.90
fB2002	32	8	7.87	7.87	7.67	7.67	7.86	7.86
bG2011	20	11	11.00	11.00	11.00	11.00	11.00	11.00
fG2011	30	30	25.13	9.30	15.83	7.69	21.61	8.76
sG2011	32	32	32.00	17.77	32.00	10.69	32.00	15.96

approach.

Class F1 All paths are feasible, and all are found before **GR** says to stop. For these programs, searching stops early anyway, so the proposed approach is just overhead. However, as noted in section 6.4.2, timing test shows the overhead is trivial, so this is not a problem.

Class F2 All paths are feasible, but **GR** would suggest stopping before the last feasible path(s) are actually found. For these, the **GR** approach means there is a loss of performance, i.e. it is wrong to assume that all paths not found yet when **GR** says to stop are infeasible. Execution time is shorter, but something is lost, so this is a trade-off.

Class I1 Some paths are infeasible, but all that are feasible are found by the time **GR** says to stop. The assumption that missed paths are infeasible is correct. **GR** saves time, and there is no loss of accuracy, so the proposed approach is beneficial.

Class I2 Some paths are infeasible, and some that are actually feasible get missed if searching stops at the time suggested by **GR**. The assumption that all remaining uncovered paths are infeasible is wrong. Like Class **F2**, it is a trade-off of run time for coverage.

Table 6.13 presents the classification of test programs. One test program is completely in **F1** and **F2** across all **RSR** combinations, i.e. tA2008 and fG2011, respectively. Across all rules, completely in **F1** is fine, but completely in **F2** means it needs further elaboration. As for fG2011, although all its paths are feasible, they are many, i.e. 30 paths, and its space is the third largest among the test programs (see Table 4.2).

Seven test programs are completely in **I1**, i.e. iA2008, bisA2008, bubA2008, mtA2008, qG1997, qB2002, and bG2011. The rest are combinations between I1/F1 and I2/F2.

In term of rule restriction, the expected changes are from more to less restricted: from I1/F1 to I2/F2, but not the opposite. Variations between the best and common parameters setup under rule influence are not comparable. In other words, the comparison is only applicable between more and less restricted rules, e.g. RSR1-B and RSR2-B. For example, in Table 6.13, gA2008 changes class from I1 to I2 when the rule changes from best to common parameters.

Detailed inspection of the results from each run showed that the missing paths are always the same ones, regardless of the parameter setup, the rule restriction, randomness, or repetitions. So, the difficulty level of a path does not change regardless of the treatments.

6.5.2 Decision Rules

The combined rule RSR merges the strengths of both RR and SR. RR ensures that the likelihood of covering further paths is low. As $\lambda \rightarrow 0$ it is still possible that further paths might be found but of course with very low probability.

On the other hand, SR considers the stability of the number of generations prediction. As long as $\Delta\theta$ is high, the predicted λ values over generations may be low but they are not stable. In other words, SR affects the rate of change of λ over generations.

As for the thresholds, smaller values mean less chance of missing feasi-

Table 6.13: Test Programs Classification

Program	**RSR1-B**	**RSR1-C**	**RSR2-B**	**RSR2-C**
tA2008	F1	F1	F1	F1
mmA2008	F1	F2	F1	F2
iA2008	I1	I1	I1	I1
bisA2008	I1	I1	I1	I1
binA2008	F1	F1	F1	F1
bubA2008	I1	I1	I1	I1
gA2008	I1	I2	I1	I2
rA2008	I2	I2	I1	I1
mtA2008	I1	I1	I1	I1
tM2004	I1	I2	I1	I2
eiR1985	I1	I2	I1	I2
qG1997	I1	I1	I1	I1
ttB2002	F2	F1	F2	F1
eiB2002	I1	I1	I1	I1
qB2002	I1	I2	I1	I1
scB2002	I1	I2	I1	I2
fcB2002	I1	I1	I1	I2
fB2002	F2	F2	F1	F2
bG2011	I1	I1	I1	I1
fG2011	F2	F2	F2	F2
sG2011	F1	F2	F1	F2

ble paths, but they also mean searching continues for longer, costing more computation time. On average over all test programs, the lower the value of λ the longer time required. Selecting the threshold should be based on the user's preference. So, it depends on how much the user wants to spend resources and is willing to tolerate the chance of missing some feasible paths.

6.5.3 Efficacy and Efficiency

In all of the testing programs, very small 0.1 paths **MP** shows up for **RSR2-C** (see Table 6.10), and **GR** is more efficient. The proposed approach seems to be effective and beneficial.

This approach might suggest that it is not yet time to stop searching when the normal maximum number of generations is reached. In this case efficiency would appear to be negative. However, it would imply that by keeping searching there is a real possibility to cover more paths. Thus an arbitrary upper limit on the number of generations is replaced by an upper limit determined by the history of finding test cases that cover paths: in other words, an arbitrary parameter can be replaced by parameters related to the searching performance itself. The tester can define their preferred limits on stability and the probability of covering new paths, and let these determine the time spent searching.

6.5.4 Threats To Validity

The following are considered to be challenges to make the approach applicable. Firstly, fine tuning the rule thresholds could be better with more training programs, because the results could be more generic and representative. However, the training programs used here represent a range of relevant program characteristics. So, we believe the thresholds used here are reasonable.

Secondly, testing with further programs that have different characteristics is needed. This is in order to gain more confidence that the approach will

be successful for a range of programs. Again, the test programs used here display a range of characteristics; however, broader testing is a matter for further work.

6.6 Conclusions

The proposed approach, inspired by software reliability growth models, is a novel and promising approach to be used as a stopping criterion in evolutionary path testing. The justification and the experimental results have shown its feasibility.

In deciding the threshold values for rules, the user's preferences and the resources available should be considered. The over-riding objective is path coverage. As long as this is achieved, efficiency can be a consideration. In other words, path coverage cannot be compromised to achieve higher efficiency. The reliability model parameter values identified for in this work, i.e. $\lambda \leq 0.1$ and $\Delta\theta \leq 0.0001$ (**RSR1-C**), are effective, with 0.1 feasible paths missed and 32% more efficient on average over 21 test programs.

To conclude, this dynamic stopping criterion approach means one less arbitrary parameter to worry about, as it is replaced by parameters oriented to the tester's priorities.

Chapter 7

Hybridization with Local Search

7.1 Introduction

Genetic Algorithm (GA) has been successfully employed for path testing [12]. It has been empirically proven to be an effective and efficient approach for generating test data. Apart from its successes, one of the main issues in GA based path testing is its computational time. As program complexity increases the number of target paths dramatically increases too, especially when the program includes loops.

Generating test data that can cover a huge number of target paths can be much harder. In addition, higher program complexity also means longer execution time per data input. This applies just as much to methods other than GA, because the nature of path testing as a dynamic testing approach requires a test program to be executed using real data. Test program exe-

cution adds a significant contribution to the overall computation of fitness evaluation of a data input. Thus, an aim is to reduce the number of fitness evaluations to lessen its computational time, without compromising its effectiveness. One potential way to achieve this efficiency is to hybridize GA with local search (LS).

In this chapter, GA hybridization is investigated as an approach to creating more effective and more efficient GA-based path testing. Construction of hybrid variants is based on the position/order of LS, size of LS, and variable sizing of LS. The experimental results are assessed in term of number of paths found and number of generations taken. Analysis section sheds some light on what variants work for which test programs and why they work. Some validity threats and conclusions end the chapter.

7.2 Local Search

Hill climbing (HC) is used as the LS algorithm. Basically, HC will exploit the surrounding areas of selected input data for better path coverage. HC will replace a selected input datum with a more fit neighbor, if any is available.

The best few members of the population will be selected to participate in HC. The number of (local) neighbors for each selected member is decided based on the size of its program input. For performance measures, both the number of paths found and the number of generations are used as efficacy and efficiency measures, respectively. The number of generations corresponds to the number of fitness evaluations.

The number of chromosomes (sets of input data) to be selected for LS is

termed LS size. A set of neighbors will be generated for each selected input data. A selected input datum will be replaced by its best neighbor. At the end of LS call, all newly generated and successfully selected (for being the best) input data will be inserted to the population. The following is the pseudo-code.

1. Select the N fittest members of population
2. Set the number of stall iterations SI to STALL_ITERATIONS (3 is the default value)
3. Initiate SI counter stall-it to 0
4. For each selected input data do
   (a) Generate n neighbors; n is the size of input datum
   (b) Evaluate the neighbors
   (c) If the best neighbor is better, then replace the input datum and set stall-it to 0, else increment stall-it by 1
   (d) If stall-it $\leq$ SI and not all paths are covered yet then go to (4a), else go to (5)
5. Insert the survived and newly created input data into the population

The number of neighbors equals the length of input datum. For example, the number of neighbors for a 3-integer input datum is 3. Each input has two possible movements: it could be varied up or down. A neighbor is generated by randomly selecting an element of the input and a direction to vary. The amount of subtraction or addition is based on the smallest increment or

decrement of input data type. For example, an integer would be -1 or +1 and a real would be -0.1 or +0.1. Further, this amount also can be multiplied or divided several times, in order to make larger step (or leap) in the input data (or search) space in order to avoid local optimum. The following pseudo code describes the generation of neighbors.

1. Set number of neighbors n for an input datum **INPUT**
2. If input datum is integer then set step to INT_STEP, else set step to REAL_STEP
3. For n neighbors do
   (a) Set IS to the size of input datum
   (b) Generate a set of IS random integers between -1 and 1 named loci
   (c) Multiply loci with step
   (d) Create neighbor by adding loci to **INPUT**
   (e) Check and adjust the newly created neighbor to be valid input
4. Return a set of n newly created neighbors

7.3 Hybrid Variants

LS can be called in several different ways based on its calling position and number of iterations per call. The calling position determines its role to prepare the population while the number of iterations contributes to how much local optimization is wanted.

LS can be called in several different positions with respect to generations in GA: (1) after the initial population is created but before commencing evolution, (2) after every generation, (3) after every certain number of generations, and (4) after every certain number of stall generations. In case (1), the initial population will be optimized locally with the hope that GA will start with a good population. In case (2), every newly generated population is locally optimized before entering the next generation of the evolutionary process. Case (3) ensures that the population is optimized locally after every certain number of generations. This will refresh the input population with exploitive members of the population. Case (4) introduces some locally optimized population in the hope that it can make its way out of a stagnant situation.

LS can be run once, or for a certain number of iterations, or repeatedly until there is no further improvement in best fitness value and/or path coverage. The more iterations are done, the more exploitive the population. Fewer iterations mean less time, but could miss potential improvement.

In this experiment, LS is called at the end of each certain number of generations, and it will run until there is no further improvement in either best fitness value or path coverage after 5 iterations, on each selected neighbor.

Three LS variations are constructed: LS-GA (LS is called once, before any generation of GA), GA-LS (LS is called after every certain number of generations of GA), and LS-GA-LS (combination of LS-GA and GA-LS). LS-GA aims to optimize locally the initial population, to boost GA performance at the beginning in the hope that good seeds will find more paths more quickly at later stages. GA-LS means that LS is called periodically, aiming

to improve the population in the hope that it will make it easier to cover new paths in every generation.

LS-GA-LS aims to combine the benefits of better chromosomes in the beginning and improving the population at intervals later.

In principle, we would expect LS-GA to be a bit better than GA; GA-LS to be better than LS-GA, because LS is executed multiple times instead of just once; and LS-GA-LS to be better than all the others.

In Memetic Algorithm (MA), LS is executed in every generation and applied on each population member. In contrast, we investigate less frequent use of LS, and LS using some selected population members only.

Analytically, the proposed approach will outperform native MA in terms of the average number of generations. It means that the number of generated test data by the proposed approach is less than that those of MA generated for the same path coverage. It also means fewer fitness evaluations, avoiding an expensive part of test data generation.

7.4 Experiments

Comparison between GA and hybrid GA (hGA) with LS is performed against 18 and 21 test programs for training and testing, respectively. Each test program is run 30 times with different random seeds. The same set of random seeds is used across all test programs.

GA is set up with elitism and common parameters across all test problems. However, the length of input datum (chromosome) and the allele range are

set based on each test program's input requirements.

As for LS size, the objectives are exploitation (not exploration) and short execution. For these reasons, the size must be at least 1 but not too large. Five numbers were chosen as candidates for LS size: 1, 3, 5, 10, and 15. Experiments show that these sizes do not add significant execution time.

Other than fixed LS size, variable LS size is also introduced. Variable LS size (LSv) changes its size in each call until it reaches a specified limit, i.e. can be increasing or decreasing. Logically, the search is getting harder as generations progress so that more local searching may help. So, increasing LS size is selected, from 1 initially to 10 finally with an increment of 1 each time.

Having LSv in mind, two more variations are added to the collection of hybrids: GA-LSv and LS-GA-LSv. LSv-GA is not applicable because in this variation, LS is only called once, thus has no chance to change its size.

Experiments are designed to have both training and testing processes. The training process is to find the optimal LS size in each applicable variant, and ultimately find the optimal hybrid among them all. In testing, the optimal LS size and the optimal hybrid from training are tested with different feasibility of target paths, i.e. tests that may include infeasible paths, and feasible paths only.

Eighteen of the 21 test programs are used for training. The other 3 programs are used for testing only, i.e. bG2011, fG2011, and sG2011. These programs are selected for testing because they are more challenging, and are intended to assess the generalization ability of the proposed approach. In the

training process, the total number of experiments is 17 (3 fixed LS size times 5, plus 2 LSv). For naming purpose, LSx means LS with size x (LS=x), and LSvx means LSv with range size x (LSv=x). For example, GA-LS5 means hybrid GA-LS with size 5, and GA-LSv110 means hybrid GA-LSv with size between 1 and 10.

In the testing process, the optimal hybrid in each LS size (3 hybrids) and 2 hybrid LSv are used to test all 21 test programs. Statistical paired t-Test is also conducted to judge the significance of differences between two compared hybrids.

7.5 Experimental Results (Hybrid Variants)

7.5.1 Training

The following sub sections present different size of LS. Each sub section reveals which hybrid variant performs best in term of number of paths found **PF** and number of generations **Gen**. Number of paths (**Paths {All}**) and number of feasible paths (**Paths {Feas}**) for each test program are also shown in each comparison table.

7.5.1.1 Hybrids with LS size 1

Table 7.1 shows the comparison between GA and its hybrids with LS=1 (that is, one chromosome is chosen for local search). GA-LS outperforms GA, LS-GA, and LS-GA-LS both in terms of **PF** and **Gen**. GA-LS has the highest PF and the least Gen with values 6.12 and 8.76, respectively. So, GA-LS1 is

selected as the best in its size, i.e. LS=1.

Table 7.1: Hybrid GA with LS size 1 using All Paths

No	Program	Paths		GA		GA-LS		LS-GA		LS-GA-LS	
		All	Feas	PF	Gen	PF	Gen	PF	Gen	PF	Gen
1	tA2008	4	4	3.70	23.80	3.77	17.17	3.87	19.27	3.77	19.37
2	mmA2008	13	13	11.80	17.57	11.93	18.07	11.70	15.57	11.87	21.90
3	iA2008	6	5	5.00	1.07	5.00	1.07	5.00	1.07	5.00	1.07
4	bisA2008	9	9	6.00	1.77	6.00	1.77	6.00	1.70	6.00	1.70
5	binA2008	7	7	7.00	1.00	7.00	1.00	7.00	1.00	7.00	1.00
6	bubA2008	15	4	4.00	1.10	4.00	1.10	4.00	1.07	4.00	1.07
7	gA2008	8	5	3.57	20.97	3.40	15.20	3.30	25.63	3.60	22.30
8	rA2008	5	4	3.97	16.77	3.97	15.17	3.83	14.60	3.87	12.03
9	mtA2008	52	20	17.53	11.60	17.53	10.47	17.40	12.43	17.47	14.93
10	tM2004	8	7	4.07	17.83	4.13	17.97	3.63	23.40	4.07	21.23
11	eiR1985	12	3	3.00	9.93	3.00	7.67	3.00	9.03	3.00	8.17
12	qG1997	21	4	4.00	1.00	4.00	1.00	4.00	1.00	4.00	1.00
13	ttB2002	8	8	7.03	3.07	7.00	1.13	7.03	1.33	7.07	1.33
14	eiB2002	31	5	4.90	12.53	4.97	14.37	4.90	24.07	4.83	14.07
15	qB2002	27	10	8.63	3.00	8.60	1.77	8.43	1.60	8.57	8.87
16	scB2002	15	4	2.87	12.30	2.97	14.87	2.87	19.43	2.90	16.87
17	fcB2002	5	5	4.90	16.03	4.93	13.23	4.77	12.83	4.83	11.43
18	fB2002	32	8	7.87	4.87	7.93	4.70	7.90	6.80	7.90	7.10
	Average	15.44	6.94	6.10	9.79	6.12	8.76	6.04	10.66	6.10	10.30

Figure 7.1 presents relative PF of GA and each hybrid variant. The test programs are represented as a number in the x-axis. For example, number 1 is for tA2008 in the first row in Table 7.1. A positive value means more **PF** than GA, which is desirable. In most of the test programs, GA-LS1 graph is on or above the x-axis. LS1-GA performed the least in test program 10, achieving the least **PF**, i.e. 0.4 paths less than GA. GA-LS has the fewest test programs with negative **PF** (**-PF**), i.e. 3, while LS-GA-LS and LS-GA have 5 and 7, respectively.

Figure 7.2 shows efficiency of each hybrid variant in term of **Gen** compared to GA. A positive value means more **Gens**, which is less efficient and

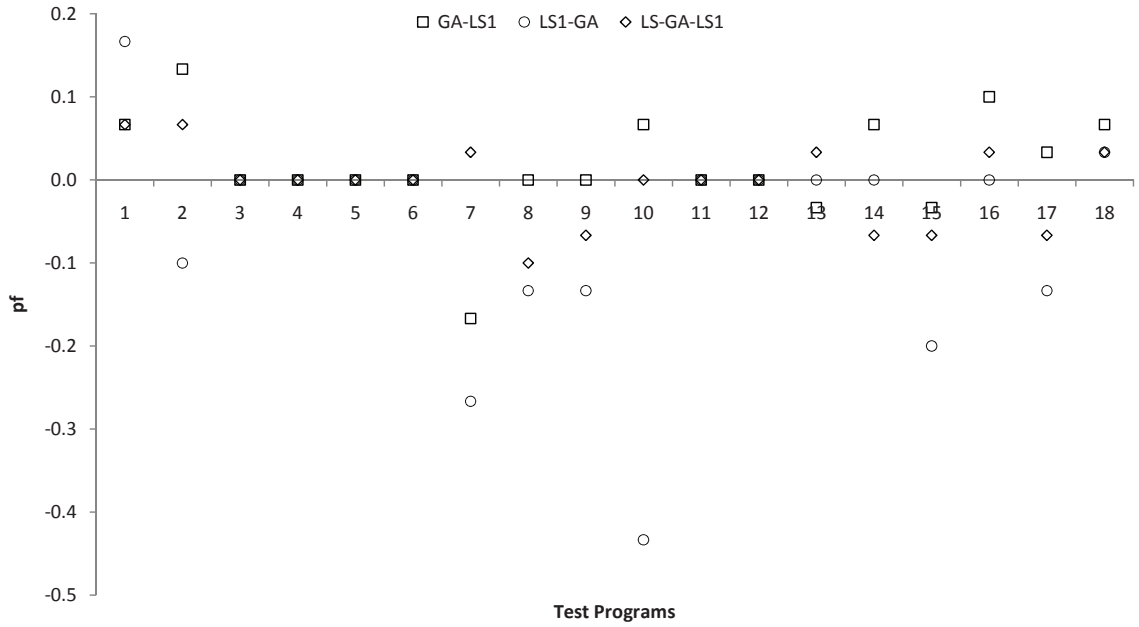


Figure 7.1: Comparison of PF with LS=1

undesirable. In other words, lower values are better. Most of the time, GA-LS1 is closest to the x-axis and some points are below the axis. LS1-GA has the most positive points in the figure, which indicates more generations are required in some test programs compared to GA. GA-LS1 is more efficient than LS1-GA and LS1-GA-LS1, with more test programs with the least generations. GA-LS1 has 9 test programs with negative **Gens** (**-Gens**), while LS1-GA and LS1-GA-LS1 have 7 and 5 test programs, respectively.

The aim of LS-GA-LS was to improve at the start and also during execution, so LS-GA-LS should be better than both LS-GA and GA-LS. However, it is not. This happens because only one chromosome is varied in LS before GA starts, so it has only a minor effect.

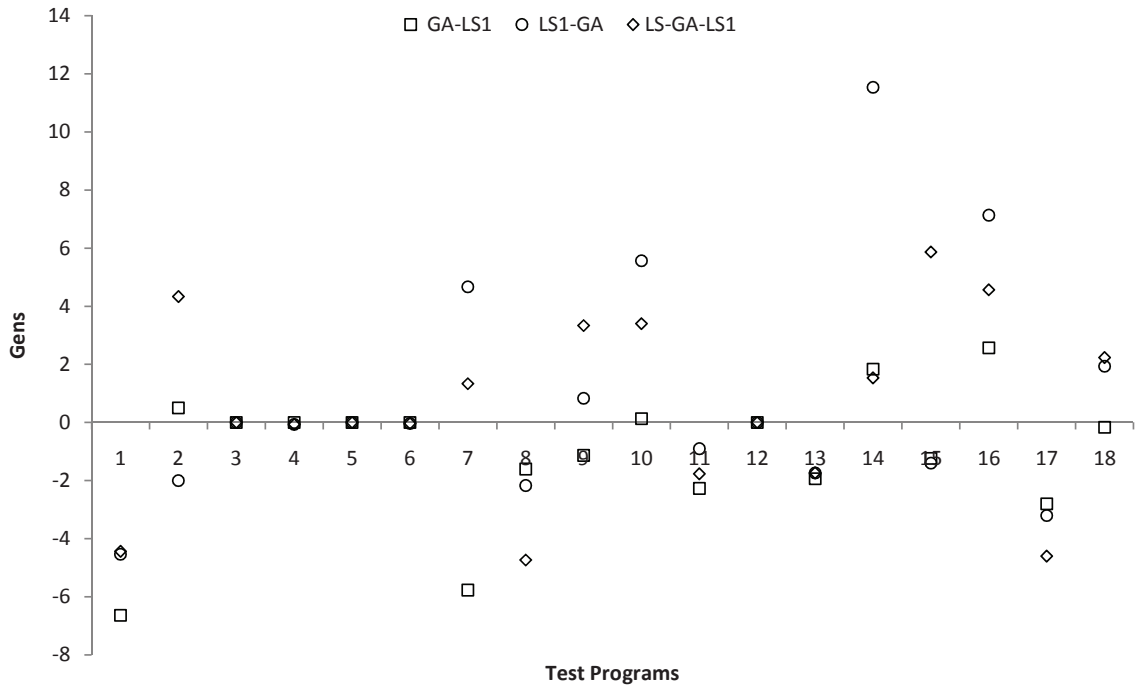


Figure 7.2: Comparison of Gens with LS=1

7.5.1.2 Hybrids with LS size 3

Table 7.2 presents the comparison between GA and its hybrids with LS=3. Although GA-LS3 is not the most efficient, it is the best performed because it has the highest **PF** of 6.15, which is the most desirable feature.

Figure 7.3 presents **PF** comparison between GA and its hybrids with LS=3. GA-LS3 has the highest points among LS3-GA and LS-GA-LS3. It also has the least **-PF** with 3 test programs, while LS3-GA and LS-GA-LS3 have 11 and 5 test programs respectively.

Figure 7.4 compares **Gen** between GA and its hybrids. LS3-GA is the most efficient with 8 **-Gen** test programs, while GA-LS3 and GA-LS3 have 6 and 7 test programs, respectively. The figure also reveals that GA-LS3 graph is about the x-axis.

Table 7.2: Hybrid GA with LS size 3 using All Paths

No	Program	Paths		GA		GA-LS		LS-GA		LS-GA-LS	
		All	Feas	PF	Gen	PF	Gen	PF	Gen	PF	Gen
1	tA2008	4	4	3.70	23.80	3.87	16.87	3.87	19.53	3.83	12.53
2	mmA2008	13	13	11.80	17.57	12.03	19.13	11.63	16.10	12.00	19.33
3	iA2008	6	5	5.00	1.07	5.00	1.07	5.00	1.07	5.00	1.07
4	bisA2008	9	9	6.00	1.77	6.00	1.77	6.00	1.67	6.00	1.67
5	binA2008	7	7	7.00	1.00	7.00	1.00	7.00	1.00	7.00	1.00
6	bubA2008	15	4	4.00	1.10	4.00	1.10	3.97	1.03	3.97	1.03
7	gA2008	8	5	3.57	20.97	3.73	19.47	3.50	19.13	3.70	20.77
8	rA2008	5	4	3.97	16.77	3.97	13.07	3.83	10.00	3.80	9.47
9	mtA2008	52	20	17.53	11.60	17.50	9.20	17.40	16.20	17.43	11.13
10	tM2004	8	7	4.07	17.83	4.17	18.53	3.93	21.97	4.20	17.57
11	eiR1985	12	3	3.00	9.93	2.93	6.83	3.00	10.10	3.00	6.60
12	qG1997	21	4	4.00	1.00	4.00	1.00	4.00	1.00	4.00	1.00
13	ttB2002	8	8	7.03	3.07	7.07	5.33	7.00	1.13	7.00	1.13
14	eiB2002	31	5	4.90	12.53	4.97	16.50	4.87	10.97	4.97	13.10
15	qB2002	27	10	8.63	3.00	8.70	9.50	8.43	1.43	8.63	13.67
16	scB2002	15	4	2.87	12.30	2.90	13.43	2.83	16.90	2.93	16.53
17	fcB2002	5	5	4.90	16.03	4.87	13.60	4.77	13.23	4.87	18.33
18	fB2002	32	8	7.87	4.87	7.93	5.60	7.93	3.83	7.97	5.30
	Average	15.44	6.94	6.10	9.79	6.15	9.61	6.05	9.24	6.13	9.51

Again, the actual ranking does not seem as expected of GA, LS-GA, GA-LS, LS-GA-LS. This is very likely because there are still too few chromosomes modified by the initial LS for it to have noticeable effect.

7.5.1.3 Hybrids with LS size 5

Table 7.3 shows the comparison between GA and its hybrids with LS=5. Both GA-LS5 and LS-GA-LS5 have equal **PF**s, i.e. 6.13 paths, but LS-GA-LS5 outperforms GA-LS5 in term of **Gens**, i.e. 8.43 vs. 8.93 generations. So, LS-GA-LS5 is more preferable than GA-LS5.

Figure 7.5 presents PF comparison of hybrids with LS=5. GA-LS5 and

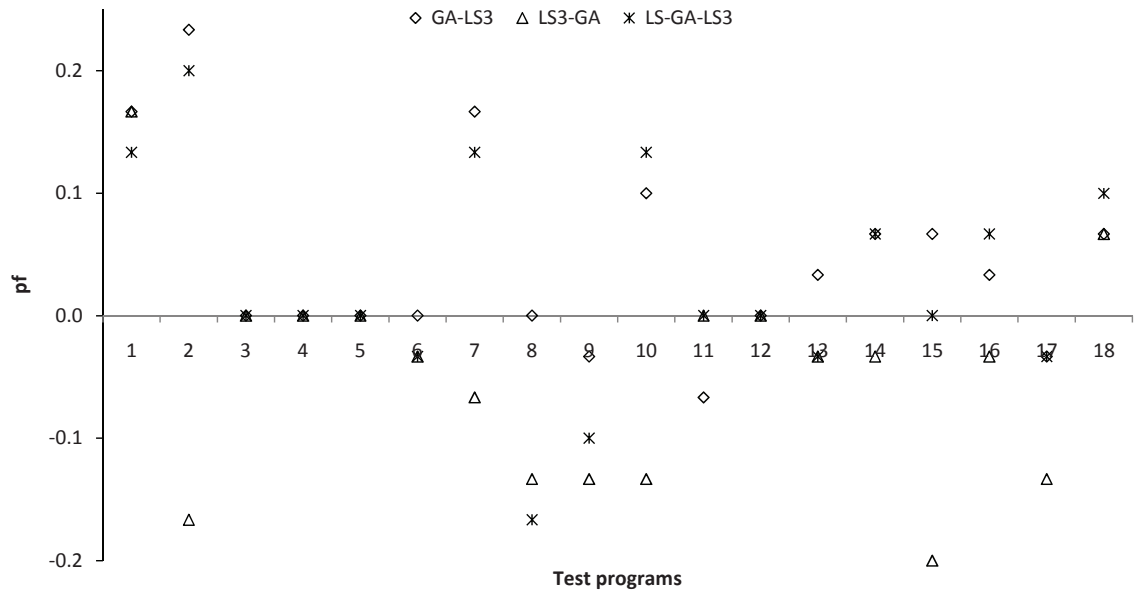


Figure 7.3: Comparison of PF with LS=3

LS-GA-LS5 cover the same number of paths on average, but they have different number of **-PF** test programs, i.e. 4 and 3 test programs, respectively. LS5-GA-LS5 has the most **-PF**, i.e. 6 test programs.

Figure 7.6 shows that both GA-LS5 and LS-GA-LS5 have some points that are close to each other. However, LS-GA-LS5 is more efficient because it has the most **-Gens**, i.e. 12 test programs. The rest have 8 and 10 test programs for GA-LS5 and LS5-GA, respectively.

By now, the observed LS-GA-LS ranking is best, as expected. This indicates that LS needs to work on enough chromosomes before it is very effective.

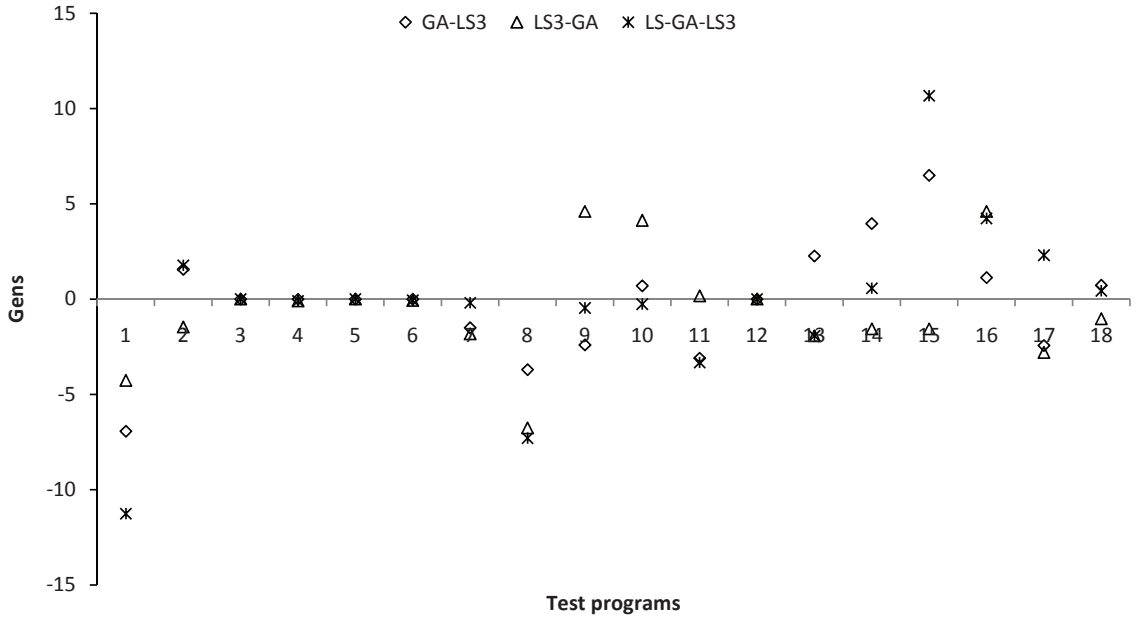


Figure 7.4: Comparison of Gens with LS=3

7.5.1.4 Hybrids with LS size 10

Table 7.4 presents the comparison between GA and its hybrids with LS=10. LS-GA-LS is the optimal variant in this LS size because it has the most **PF** 6.18 paths regardless of its **Gens** value. Its **Gens** with 8.64 generations is only outperformed by GA-LS10 with **Gens** 8.12 generations.

Figure 7.7 shows that some **PF**s between GA-LS10 and LS-GA-LS10 are close to each other. They are very similar in **PF**, but LS-GA-LS10 has some points that are higher than GA-LS10. This has made LS-GA-LS10 the best with 2 **-PF** test programs. GA-LS10 and LS10-GA have 4 and 5 **-PF** test programs, respectively.

In Figure 7.8, both GA-LS10 and LS-GA-LS10 have similar **Gens** patterns. Although GA-LS10 has the most **-Gens** with 10 test programs, on average the most efficient is LS-GA-LS10 with 9 test programs. LS10-GA is

Table 7.3: Hybrid GA with LS size 5 using All Paths

No	Program	Paths		GA		GA-LS		LS-GA		LS-GA-LS	
		All	Feas	PF	Gen	PF	Gen	PF	Gen	PF	Gen
1	tA2008	4	4	3.70	23.80	3.73	15.17	3.83	18.10	3.87	19.77
2	mmA2008	13	13	11.80	17.57	12.17	16.50	11.77	18.73	11.97	12.93
3	iA2008	6	5	5.00	1.07	5.00	1.07	5.00	1.07	5.00	1.07
4	bisA2008	9	9	6.00	1.77	6.00	1.77	6.00	1.60	6.00	1.60
5	binA2008	7	7	7.00	1.00	7.00	1.00	7.00	1.00	7.00	1.00
6	bubA2008	15	4	4.00	1.10	4.00	1.10	4.00	1.07	4.00	1.07
7	gA2008	8	5	3.57	20.97	3.53	20.03	3.67	19.27	3.63	18.63
8	rA2008	5	4	3.97	16.77	3.97	17.97	3.87	11.40	3.87	11.07
9	mtA2008	52	20	17.53	11.60	17.53	8.70	17.33	11.17	17.43	10.57
10	tM2004	8	7	4.07	17.83	4.13	15.07	3.83	29.33	4.10	17.17
11	eiR1985	12	3	3.00	9.93	2.93	7.80	3.00	8.97	3.00	7.73
12	qG1997	21	4	4.00	1.00	4.00	1.00	4.00	1.00	4.00	1.00
13	ttB2002	8	8	7.03	3.07	7.00	1.13	7.00	1.13	7.00	1.13
14	eiB2002	31	5	4.90	12.53	4.87	10.07	4.90	15.50	4.97	11.50
15	qB2002	27	10	8.63	3.00	8.63	4.87	8.60	4.83	8.67	6.70
16	scB2002	15	4	2.87	12.30	2.90	13.37	2.97	11.37	2.97	11.93
17	fcB2002	5	5	4.90	16.03	4.97	19.07	4.97	14.83	4.93	12.63
18	fB2002	32	8	7.87	4.87	7.93	5.03	7.97	4.07	7.97	4.20
	Average	15.44	6.94	6.10	9.79	6.13	8.93	6.09	9.69	6.13	8.43

the least **-Gens** with 6 test programs.

LS-GA-LS is best in both **PF** and **Gens** terms. This shows that the observed ranking is as expected, and this size of LS is effective for improving the generator.

7.5.1.5 Hybrids with LS size 15

Table 7.5 shows the comparison between GA and its hybrids with LS=15. GA-LS has the most **PF**, with 6.17 paths found on average, but LS-GA-LS15 is very close with only 0.02 paths difference.

Figure 7.9 presents **PF** for all LS=15 hybrids relative to GA. Five test

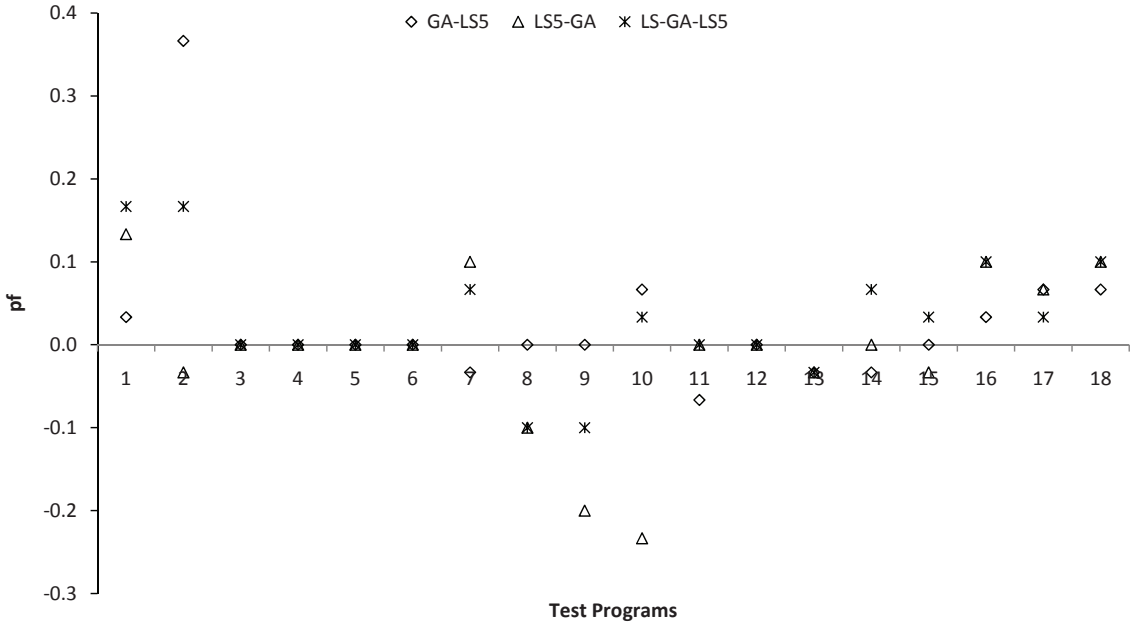


Figure 7.5: Comparison of PF with LS=5

programs have peaks in both opposite directions, i.e. test program 2, 7, 9, 10, and 15. All these programs have **-PF** if either LS15-GA or LS-GA-LS15 is used. It is possible that the first generation is locally over optimized, which may lead to the traps of local optima, before any evolutionary process is started. Similarly, although LS-GA-LS15 first generation is locally over optimized, its coming generations are locally re-optimized repeatedly, which means LS-GA-LS15 outperforms LS15-GA. This also means that GA-LS15 outperforms both of them because it never calls LS before any evolutionary process, which means it never has the initial generation that is locally over optimized. Moreover, this also shows that LS=15 is the limit of effective LS. Only GA-LS15 has no **-PF** test programs. LS15-GA and LS-GA-LS15 have 9 and 6 **-PF** test programs, respectively.

Figure 7.10 shows **Gens** for all LS=15 hybrids relative to GA. Program 15 is the least efficient for both GA-LS15 and LS-GA-LS15. While both GA-

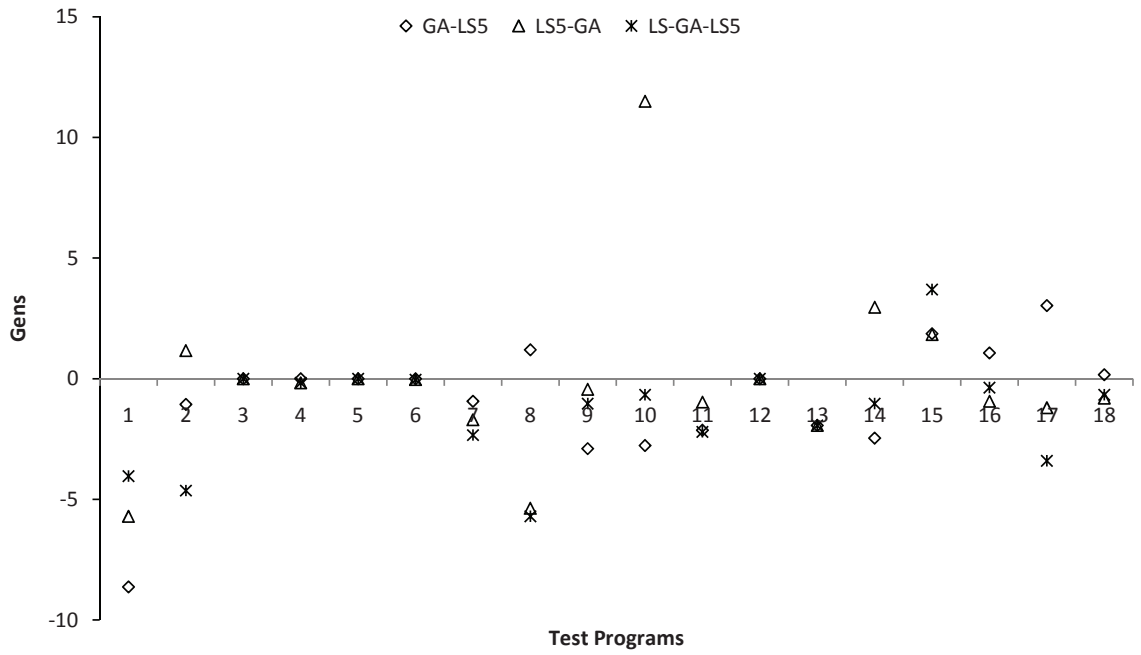


Figure 7.6: Comparison of Gens with LS=5

LS15 and LS15-GA have the same 9 **-Gens** test programs each, LS-GA-LS15 has 11 test programs.

7.5.1.6 Hybrids with variable LS size

Table 7.6 presents the comparison between GA and its variable LS size variants. LS-GA-LSv110 is the most desirable hybrid with the most **PF** 6.23 paths and the least **Gen** 6.67 generations. Both variable size variants are very competitive to each other with only 0.03 paths and 0.09 generations different.

Figure 7.11 shows the efficacy graphs of GA-LSv110 and LS-GA-LSv110 compared to GA. Both graphs have similar patterns.

Figure 7.12 describes the efficiency graph of GA-LSv110 and LS-GA-LSv110 compared to GA. Both GA-LSv110 and LS-GA-LSv110 have similar

Table 7.4: Hybrid GA with LS size 10 using All Paths

No	Program	Paths		GA		GA-LS		LS-GA		LS-GA-LS	
		All	Feas	PF	Gen	PF	Gen	PF	Gen	PF	Gen
1	tA2008	4	4	3.70	23.80	3.83	17.53	3.90	20.30	3.90	9.53
2	mmA2008	13	13	11.80	17.57	12.13	10.37	11.83	18.50	12.17	11.00
3	iA2008	6	5	5.00	1.07	5.00	1.07	5.00	1.07	5.00	1.07
4	bisA2008	9	9	6.00	1.77	6.00	1.77	6.00	1.83	6.00	1.83
5	binA2008	7	7	7.00	1.00	7.00	1.00	7.00	1.00	7.00	1.00
6	bubA2008	15	4	4.00	1.10	4.00	1.10	4.00	1.13	4.00	1.13
7	gA2008	8	5	3.57	20.97	3.70	14.43	3.63	22.90	3.67	19.60
8	rA2008	5	4	3.97	16.77	3.90	12.07	3.90	11.40	3.87	11.30
9	mtA2008	52	20	17.53	11.60	17.50	8.37	17.37	10.90	17.50	10.90
10	tM2004	8	7	4.07	17.83	4.33	23.60	3.70	19.93	4.37	18.87
11	eiR1985	12	3	3.00	9.93	2.93	7.90	3.00	6.77	3.00	5.23
12	qG1997	21	4	4.00	1.00	4.00	1.00	4.00	1.00	4.00	1.00
13	ttB2002	8	8	7.03	3.07	7.03	1.13	7.00	1.10	7.07	3.07
14	eiB2002	31	5	4.90	12.53	4.93	10.73	4.93	14.53	4.93	16.50
15	qB2002	27	10	8.63	3.00	8.77	10.93	8.57	1.73	8.87	18.93
16	scB2002	15	4	2.87	12.30	2.90	7.57	2.93	15.27	2.97	8.90
17	fcB2002	5	5	4.90	16.03	4.87	10.63	4.97	21.93	5.00	11.50
18	fB2002	32	8	7.87	4.87	7.93	4.90	8.00	5.67	8.00	4.17
	Average	15.44	6.94	6.10	9.79	6.15	8.12	6.10	9.83	6.18	8.64

performance. GA-LSv110 has 10 **-Gens** test programs and LS-GA-LSv110 has 2 more test programs than GA-LSv110. This makes LS-GA-LSv110 more efficient than GA-LSv110.

Table 7.7 compares hybrids with fixed LS size 10 and variable LSv110.Only GA-LS10 and LS-GA-LS10 are comparable with GA-LSv110 and LS-GA-LSv110 because they have the most treatment similarities, i.e. variant type and LS size. On the average, LS-GA-LSv110 outperforms all of them in terms of both **PF** and **Gen**.

Figure 7.13 shows comparison between hybrids with fixed LS=10 and LSv110. LS-GA-LSv110 is the most dominant in all test programs. Most of

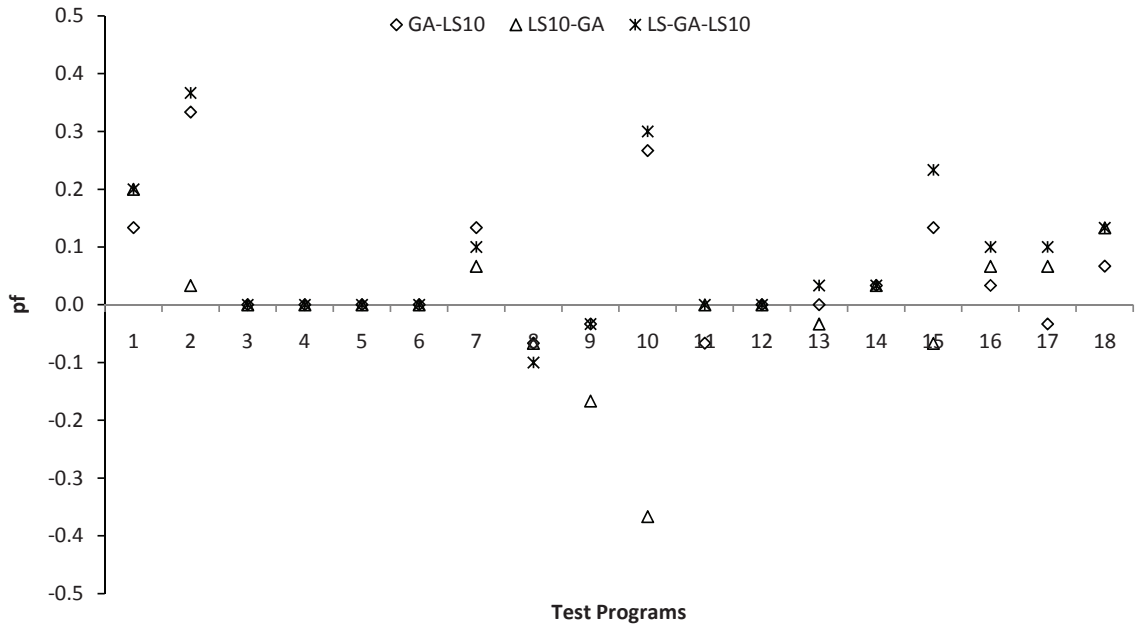


Figure 7.7: Comparison of PF with LS=10

the graphs follow similar patterns. LS10-GA has test program 10 with the most **-PF**.

Figure 7.14 presents **Gens** comparison between hybrids with fixed LS=10 and LSv110. Most of the graphs have similar patterns, except LS10-GA that slightly deviates from the rest. The number of **-Gens** test programs for all hybrids are LS10-GA 6, GA-LS10 10, LS-GA-LS10 9, GA-LSv110 9, and LS-GA-LSv110 11.

7.5.1.7 Best Hybrids based on LS Size

Figure 7.15 and Figure 7.16 show the best hybrids in its LS Size class in terms of **PF** and **Gens**, respectively. In Figure 7.15, almost all PF graphs follow similar pattern, except for GA-LS1 graphs that have some points with larger deviations in test program 7 and 15. In general, the graphs suggest

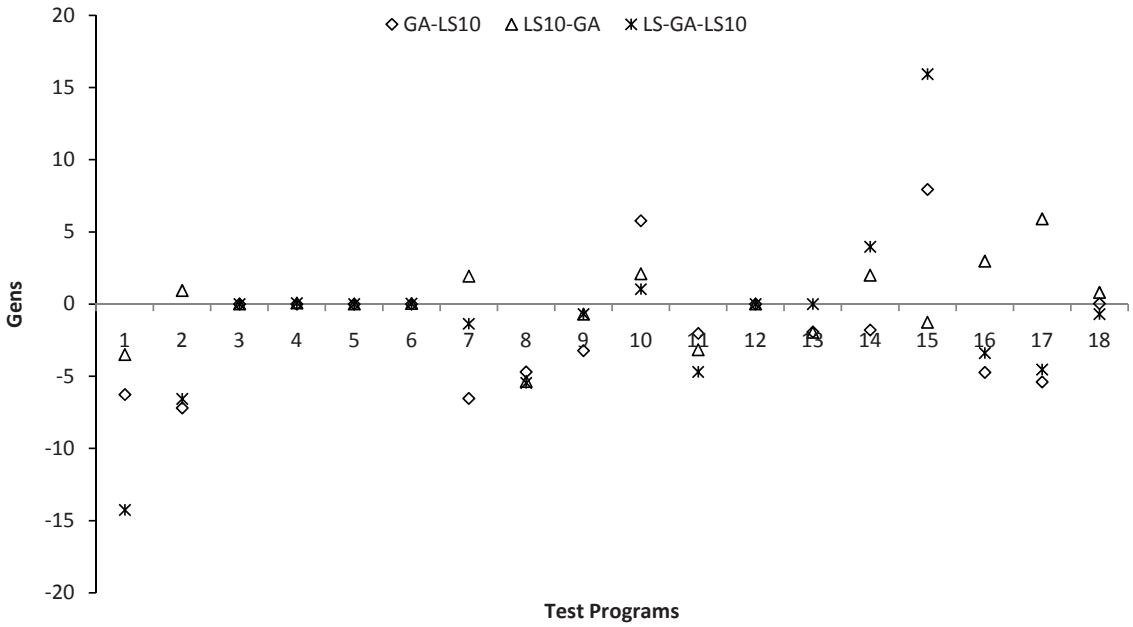


Figure 7.8: Comparison of Gens with LS=10

that hybrids perform better than GA.

In Figure 7.16, all graphs have similar shapes and are close to each other. The graphs show that most of the time all hybrids perform more efficiently than GA, although some test programs require more computational time, such as test program 13, 14, 15, and 16. Test program 15 is the least efficient for almost all hybrids, except for GA-LS1, which has the least LS size. It has a high number of feasible paths and a high number of infeasible paths. If this was the explanation, test program 9 should also have poor efficiency, but it does not. It turns out that test program 15 has 2 loops, while test program 9 has 1 loop only. In term of path complexity, adding another loop into a program means greatly multiplying its existing number of paths. So, its computational complexity is much harder. This means that its paths are harder to cover than that of test program 9. The number of **-Gens** test programs for all hybrids are GA-LS1 9, GA-LS3 6, LS-GA-LS5 3, LS-GA-

Table 7.5: Hybrid GA with LS size 15 using All Paths

No	Program	Paths		GA		GA-LS		LS-GA		LS-GA-LS	
		All	Feas	PF	Gen	PF	Gen	PF	Gen	PF	Gen
1	tA2008	4	4	3.70	23.80	3.77	14.87	3.80	19.67	3.83	9.23
2	mmA2008	13	13	11.80	17.57	12.13	9.83	11.63	9.93	12.10	8.77
3	iA2008	6	5	5.00	1.07	5.00	1.07	5.00	1.07	5.00	1.07
4	bisA2008	9	9	6.00	1.77	6.00	1.77	6.00	1.83	6.00	1.83
5	binA2008	7	7	7.00	1.00	7.00	1.00	7.00	1.00	7.00	1.00
6	bubA2008	15	4	4.00	1.10	4.00	1.10	4.00	1.07	4.00	1.07
7	gA2008	8	5	3.57	20.97	3.63	15.00	3.53	21.10	3.47	17.80
8	rA2008	5	4	3.97	16.77	3.93	15.73	3.90	14.67	3.90	15.10
9	mtA2008	52	20	17.53	11.60	17.53	8.20	17.47	13.97	17.60	13.63
10	tM2004	8	7	4.07	17.83	4.37	19.40	3.90	14.07	4.00	7.60
11	eiR1985	12	3	3.00	9.93	3.00	8.10	3.00	8.47	3.00	5.77
12	qG1997	21	4	4.00	1.00	4.00	1.00	4.00	1.00	4.00	1.00
13	ttB2002	8	8	7.03	3.07	7.07	5.37	7.00	1.10	7.03	1.10
14	eiB2002	31	5	4.90	12.53	4.93	10.47	4.93	10.80	4.87	11.80
15	qB2002	27	10	8.63	3.00	8.87	19.20	8.47	1.70	8.80	18.37
16	scB2002	15	4	2.87	12.30	2.90	3.57	2.97	14.03	2.97	7.03
17	fcB2002	5	5	4.90	16.03	4.93	12.47	4.77	13.63	4.83	9.67
18	fB2002	32	8	7.87	4.87	7.93	4.87	7.83	4.70	7.83	3.90
	Average	15.44	6.94	6.10	9.79	6.17	8.50	6.07	8.54	6.12	7.54

LS10 10, GA-LS15 10, GA-LSv110 11, and LS-GA-LSv110 12.

7.5.2 Testing

7.5.2.1 Hybrids with All Paths

Table 7.8 shows the testing results for hybrid GA with fixed LS size and involving all paths both feasible and infeasible ones. The hybrids are all the optimal variant within each class of LS size.

Why might the inclusion of infeasible paths matter? An infeasible path means that the search for test data could go forever unless some other stop-

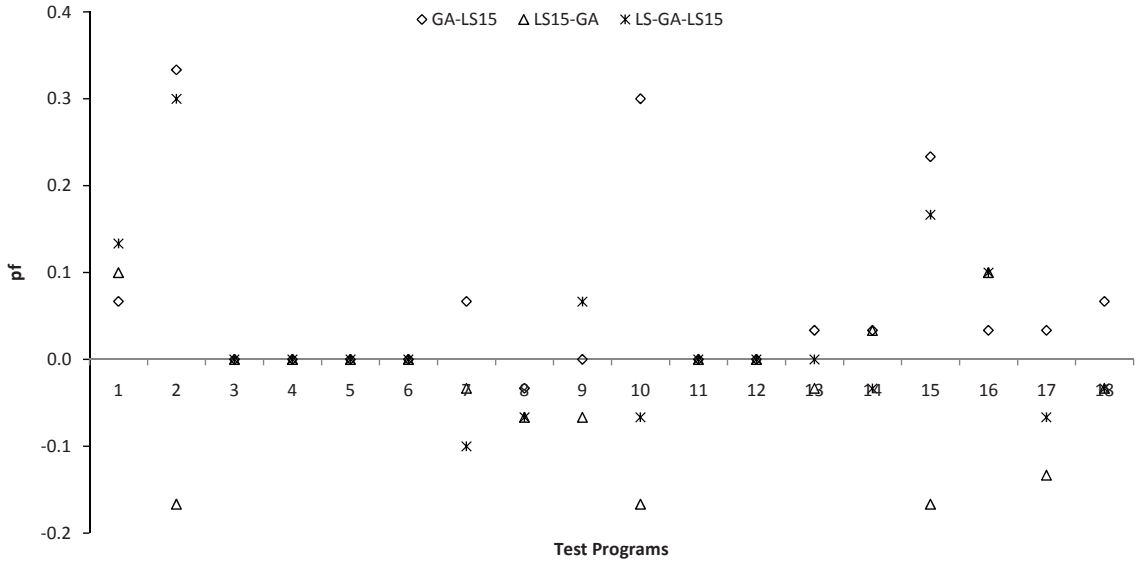


Figure 7.9: Comparison of PF with LS=15

ping conditions are being used. On the other hand, it also draws the search in other directions, adding diversity.

Both LS-GA-LS5 and LS-GA-LS10 have equal **PF**s, but the former is slightly more efficient (0.21 generations) than the latter. All the hybrids are very competitive because all of their **PF**s lie in the range of 0.16 paths only (max **PF** - min **PF**). Similarly, the **Gen** for all hybrids are within the range of 1.1 generations only.

Table 7.9 presents the testing results for hybrid variants with variable LS size and considering all paths. LS-GA-LSv110 slightly outperforms GA-LSv110 with 0.04 paths more and 0.01 generations less in terms of **PF** and **Gen**, respectively. Across all hybrid variants, LS-GA-LSv110 is the most effective and efficient one with the highest **PF** 7.86 paths and the least **Gen** 10.96 generations. Overall, LSv hybrids perform better than the fixed LS size ones both in terms of **PF** and **Gen**.

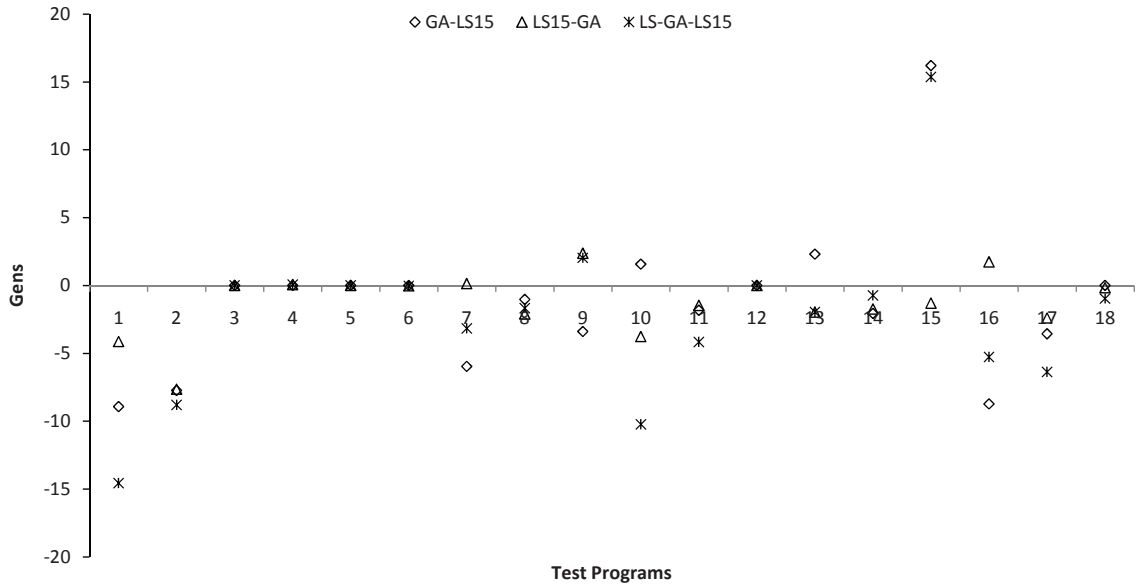


Figure 7.10: Comparison of Gens with LS=15

Figure 7.17 presents comparison of **PF** across all hybrids. Almost all hybrid graphs are on or above the x-axis and only a few are slightly below. This means that the hybrids outperform GA in general. The efficacy between hybrids is very competitive as shown by the closeness among the graphs. The improvement made by hybrids are within 0.5 paths in most of the test programs, except for the last two of them that improved significantly, i.e. test program 20 and 21.

Figure 7.18 shows comparison of **Gen** for all hybrids. In most test programs, hybrids are more efficient than GA, except for four test programs: 14, 15, 16, and 21. Test program 15 has the least efficiency. The graphs generally follow similar patterns. The number of **-Gens** test programs for all hybrids are GA-LS1 10, GA-LS3 8, LS-GA-LS5 16, LS-GA-LS10 11, GA-LS15 11, GA-LSv110 13, and LS-GA-LSv110 15.

Table 7.6: Hybrid GA with LSv size 1 to 10 using All Paths

No	Program	Paths		GA		GA-LSv110		LS-GA-LSv110	
		All	Feas	PF	Gen	PF	Gen	PF	Gen
1	tA2008	4	4	3.70	23.80	4.00	4.50	3.93	4.73
2	mmA2008	13	13	11.80	17.57	12.23	9.47	12.23	6.87
3	iA2008	6	5	5.00	1.07	5.00	1.07	5.00	1.07
4	bisA2008	9	9	6.00	1.77	6.00	1.77	6.00	1.83
5	binA2008	7	7	7.00	1.00	7.00	1.00	7.00	1.00
6	bubA2008	15	4	4.00	1.10	4.00	1.10	4.00	1.13
7	gA2008	8	5	3.57	20.97	3.53	10.40	3.67	11.97
8	rA2008	5	4	3.97	16.77	3.90	13.50	3.93	12.83
9	mtA2008	52	20	17.53	11.60	17.57	8.53	17.47	9.13
10	tM2004	8	7	4.07	17.83	4.63	16.03	4.93	17.60
11	eiR1985	12	3	3.00	9.93	3.00	6.63	3.00	5.57
12	qG1997	21	4	4.00	1.00	4.00	1.00	4.00	1.00
13	ttB2002	8	8	7.03	3.07	7.00	1.13	7.10	1.67
14	eiB2002	31	5	4.90	12.53	4.90	13.97	4.97	11.90
15	qB2002	27	10	8.63	3.00	8.90	20.47	8.90	18.47
16	scB2002	15	4	2.87	12.30	2.97	2.17	2.97	3.90
17	fcB2002	5	5	4.90	16.03	4.97	4.63	5.00	5.13
18	fB2002	32	8	7.87	4.87	7.93	4.40	7.97	4.33
	Average	15.44	6.94	6.10	9.79	6.20	6.76	6.23	6.67

7.5.2.2 Hybrids with Feasible Paths

Infeasible paths make the search infinite, if searching continues until all target paths are found. This raises an interesting question, what might be the impact of only having feasible target paths? The expectation is that the search would be quicker to cover all the feasible paths.

Table 7.10 shows the two most similar fixed LS size variants with the variable LS size ones. LS-GA-LSv110 slightly found more paths than its

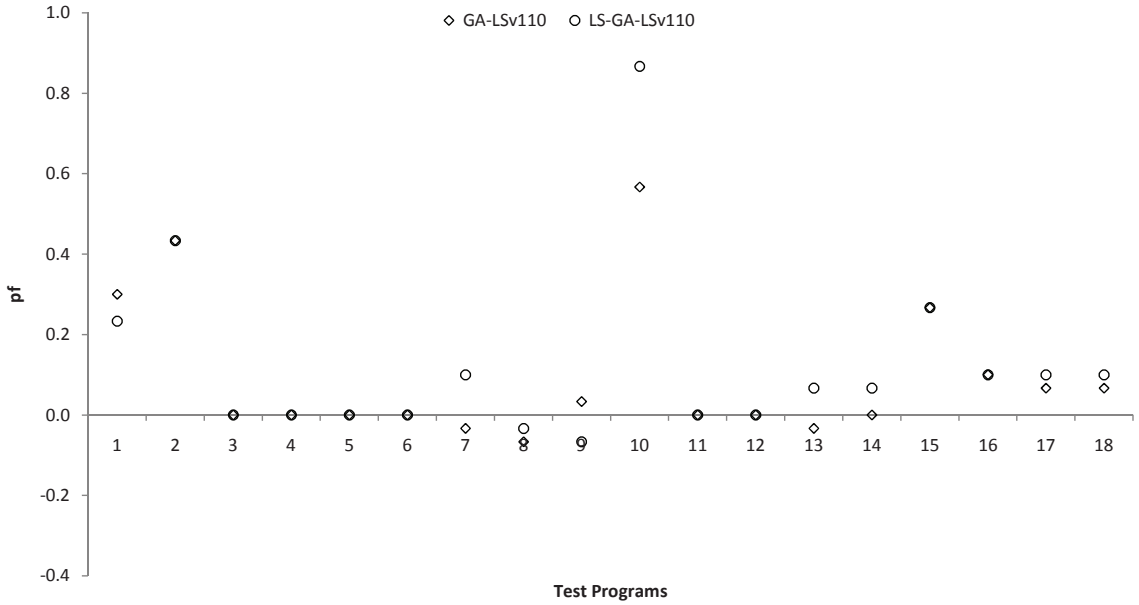


Figure 7.11: Comparison of PF for LSv hybrids

Table 7.7: Hybrid Comparison between Fixed and Variable LS Sizes

No	Program	Paths		GA		GA-LS10		LS-GA-LS10		GA-LSv110		LS-GA-LSv110	
		All	Feas	PF	Gen	PF	Gen	PF	Gen	PF	Gen	PF	Gen
1	tA2008	4	4	3.70	23.80	3.83	17.53	3.90	9.53	4.00	4.50	3.93	4.73
2	mmA2008	13	13	11.80	17.57	12.13	10.37	12.17	11.00	12.23	9.47	12.23	6.87
3	iA2008	6	5	5.00	1.07	5.00	1.07	5.00	1.07	5.00	1.07	5.00	1.07
4	bisA2008	9	9	6.00	1.77	6.00	1.77	6.00	1.83	6.00	1.77	6.00	1.83
5	binA2008	7	7	7.00	1.00	7.00	1.00	7.00	1.00	7.00	1.00	7.00	1.00
6	bubA2008	15	4	4.00	1.10	4.00	1.10	4.00	1.13	4.00	1.10	4.00	1.13
7	gA2008	8	5	3.57	20.97	3.70	14.43	3.67	19.60	3.53	10.40	3.67	11.97
8	rA2008	5	4	3.97	16.77	3.90	12.07	3.87	11.30	3.90	13.50	3.93	12.83
9	mtA2008	52	20	17.53	11.60	17.50	8.37	17.50	10.90	17.57	8.53	17.47	9.13
10	tM2004	8	7	4.07	17.83	4.33	23.60	4.37	18.87	4.63	16.03	4.93	17.60
11	eiR1985	12	3	3.00	9.93	2.93	7.90	3.00	5.23	3.00	6.63	3.00	5.57
12	qG1997	21	4	4.00	1.00	4.00	1.00	4.00	1.00	4.00	1.00	4.00	1.00
13	ttB2002	8	8	7.03	3.07	7.03	1.13	7.07	3.07	7.00	1.13	7.10	1.67
14	eiB2002	31	5	4.90	12.53	4.93	10.73	4.93	16.50	4.90	13.97	4.97	11.90
15	qB2002	27	10	8.63	3.00	8.77	10.93	8.87	18.93	8.90	20.47	8.90	18.47
16	scB2002	15	4	2.87	12.30	2.90	7.57	2.97	8.90	2.97	2.17	2.97	3.90
17	fcB2002	5	5	4.90	16.03	4.87	10.63	5.00	11.50	4.97	4.63	5.00	5.13
18	fB2002	32	8	7.87	4.87	7.93	4.90	8.00	4.17	7.93	4.40	7.97	4.33
	Average	15.44	6.94	6.10	9.79	6.15	8.12	6.18	8.64	6.20	6.76	6.23	6.67

sibling GA-LSv110 by 0.01 paths only. Overall, both LSv variants outperform the LS variants in terms of efficacy and efficiency. LSv hybrids have found

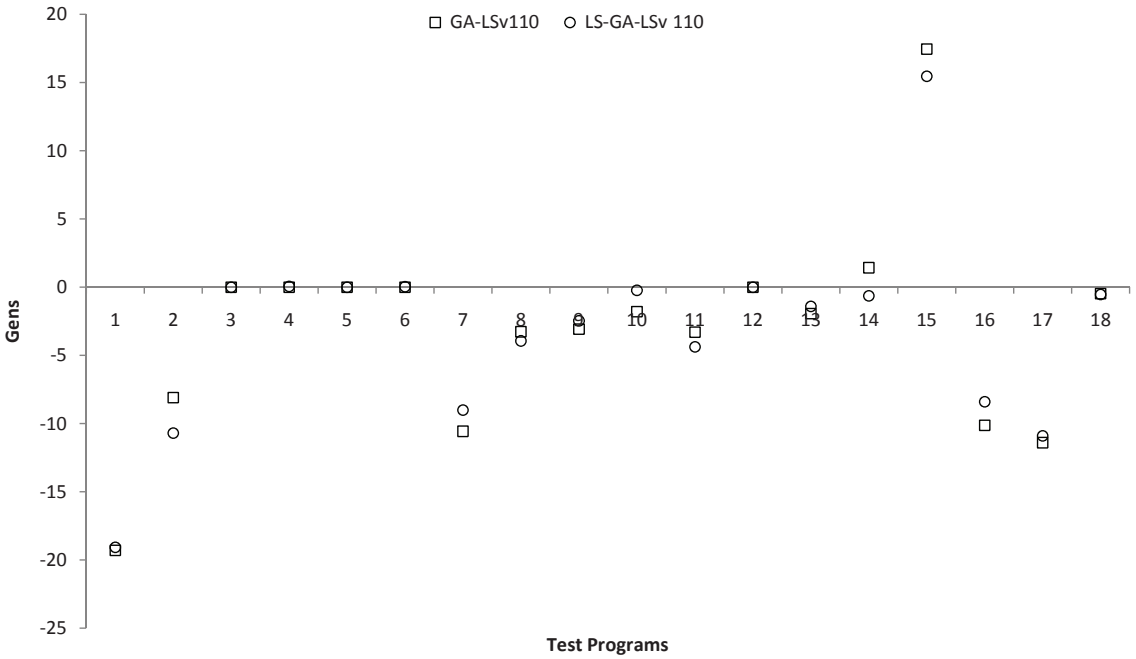


Figure 7.12: Comparison of Gens for LSv hybrids

Table 7.8: Testing, Hybrid GA with fixed LS size and All Paths

No	Program	GA		GA-LS1		GA-LS3		LS-GA-LS5		LS-GA-LS10		GA-LS15	
		PF	Gen	PF	Gen	PF	Gen	PF	Gen	PF	Gen	PF	Gen
1	tA2008	3.70	23.80	3.77	17.2	3.9	16.9	3.87	19.77	3.90	9.53	3.77	14.87
2	mmA2008	11.80	17.57	11.93	18.1	12.0	19.1	11.97	12.93	12.17	11.00	12.13	9.83
3	iA2008	5.00	1.17	5.00	1.2	5.0	1.2	5.00	1.17	5.00	1.17	5.00	1.17
4	bisA2008	6.00	1.83	6.00	1.8	6.0	1.8	6.00	1.66	6.00	1.90	6.00	1.83
5	binA2008	7.00	1.00	7.00	1.0	7.0	1.0	7.00	1.00	7.00	1.00	7.00	1.00
6	bubA2008	4.00	1.10	4.00	1.1	4.0	1.1	4.00	1.07	4.00	1.13	4.00	1.10
7	gA2008	3.57	20.97	3.40	15.2	3.7	19.5	3.63	18.63	3.67	19.60	3.63	15.00
8	rA2008	3.97	16.77	3.97	15.2	4.0	13.1	3.87	11.07	3.87	11.30	3.93	15.73
9	mtA2008	17.53	11.60	17.53	10.5	17.5	9.2	17.43	10.57	17.50	10.90	17.53	8.20
10	tM2004	4.07	17.83	4.13	18.0	4.2	18.5	4.10	17.17	4.37	18.87	4.37	19.40
11	eiR1985	3.00	9.93	3.00	7.7	2.9	6.8	3.00	7.73	3.00	5.23	3.00	8.10
12	qG1997	4.00	1.00	4.00	1.0	4.0	1.0	4.00	1.00	4.00	1.00	4.00	1.00
13	ttB2002	7.03	3.07	7.00	1.1	7.1	5.3	7.00	1.13	7.07	3.07	7.07	5.37
14	eiB2002	4.90	12.53	4.97	14.4	5.0	16.5	4.97	11.50	4.93	16.50	4.93	10.47
15	qB2002	8.63	3.00	8.60	1.8	8.7	9.5	8.67	6.70	8.87	18.93	8.87	19.20
16	scB2002	3.83	11.30	3.93	13.9	3.9	12.4	3.93	10.93	3.93	7.90	3.86	2.57
17	fcB2002	4.90	16.03	4.93	13.2	4.9	13.6	4.93	12.63	5.00	11.50	4.93	12.47
18	fB2002	7.87	4.87	7.93	4.7	7.9	5.6	7.97	4.20	8.00	4.17	7.93	4.87
19	bG2011	11.00	1.47	11.00	1.5	11.0	1.5	11.00	1.40	11.00	1.33	11.00	1.47
20	fG2011	9.30	58.73	9.70	56.7	9.8	49.5	9.83	53.53	9.53	48.23	9.60	45.33
21	sG2011	17.77	65.73	18.77	70.0	20.0	63.1	21.80	67.83	21.07	73.83	20.43	64.20
	Average	7.09	14.35	7.17	13.57	7.26	13.63	7.33	13.03	7.33	13.24	7.29	12.53

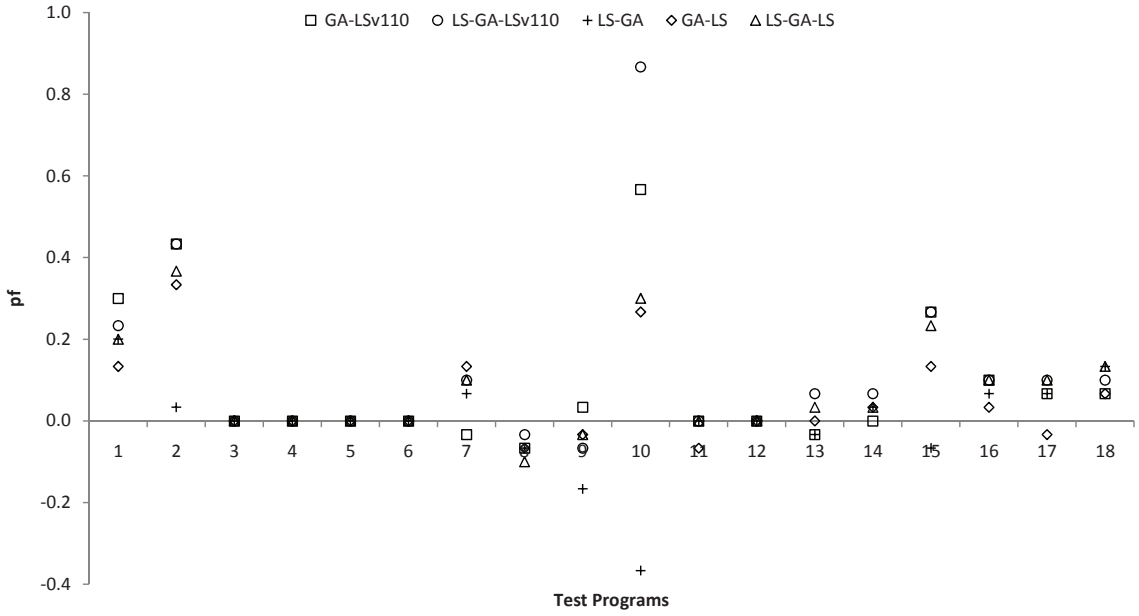


Figure 7.13: Comparison of PF for hybrids with LS=10 and LSv≤10

at least 0.5 paths more than the fixed size ones.

Figure 7.19 presents a comparison of **PF** between LS10 and LSv hybrids, with all infeasible paths excluded. On average, all hybrids found more paths than GA, although some hybrids perform worse than GA in 3 test programs: 7, 14, and 18. For all hybrids, a significant improvement is achieved in the last test program 21.

Figure 7.20 shows comparison of **Gens** between LS10 and LSv hybrids with infeasible paths excluded from the list of target paths. The number of **-Gens** test programs for all hybrids are GA-LS10 12, LS-GA-LS10 13, GA-LSv110 12, and LS-GA-LSv110 14.

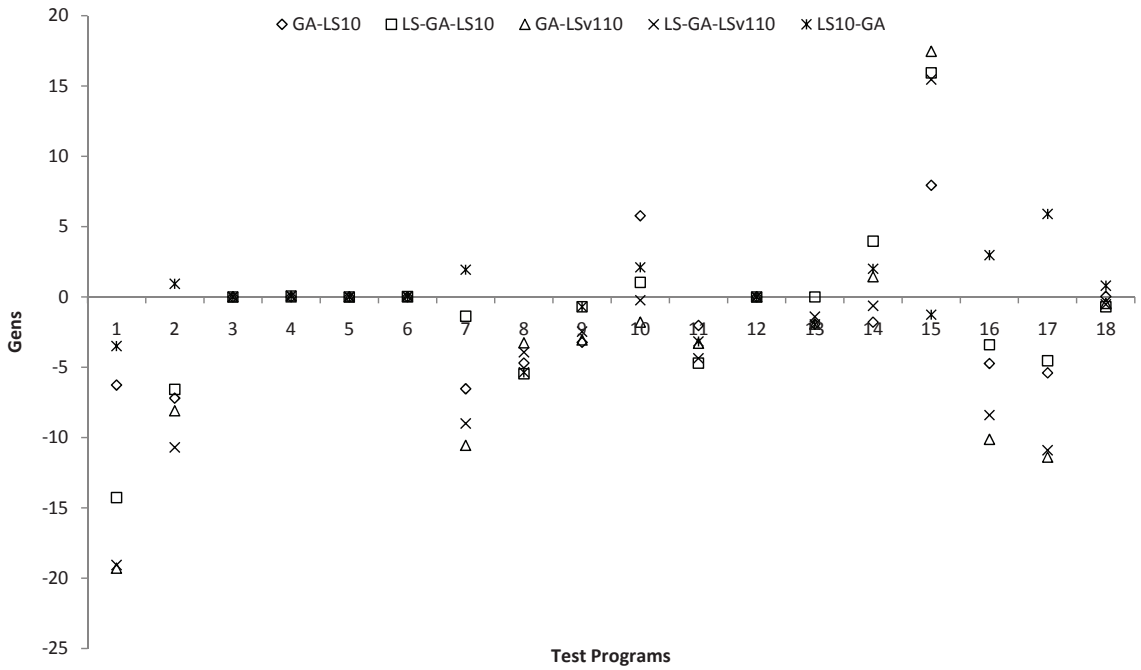


Figure 7.14: Comparison of Gens for hybrids with LS=10 and LSv≤10

7.5.2.3 Hybrids with SRMSC

When there are infeasible target paths, searching can never stop with all target paths found. It can only stop when an arbitrary number of generations is reached, or some other stopping condition is used.

Table 7.11 presents the performance of LS-GA-LSv110 variant using all target paths, that makes use of rules **RSR1-C** and **RSR2-C**. **RSR1-C** and **RSR2-C** allow the search to stop when the likelihood of finding new target paths drops below a threshold. The idea is to understand the trade-offs involved in number of target paths found versus execution time, if these stopping rules are used; **RSR2-C** is more conservative. **RSR1-C** can achieve more efficiency up to 78% by sacrificing 1.87 paths compared to LS-GA-LSv110 without any rules, while **RSR2-C** is about 29% more efficient by losing 1.14 paths.

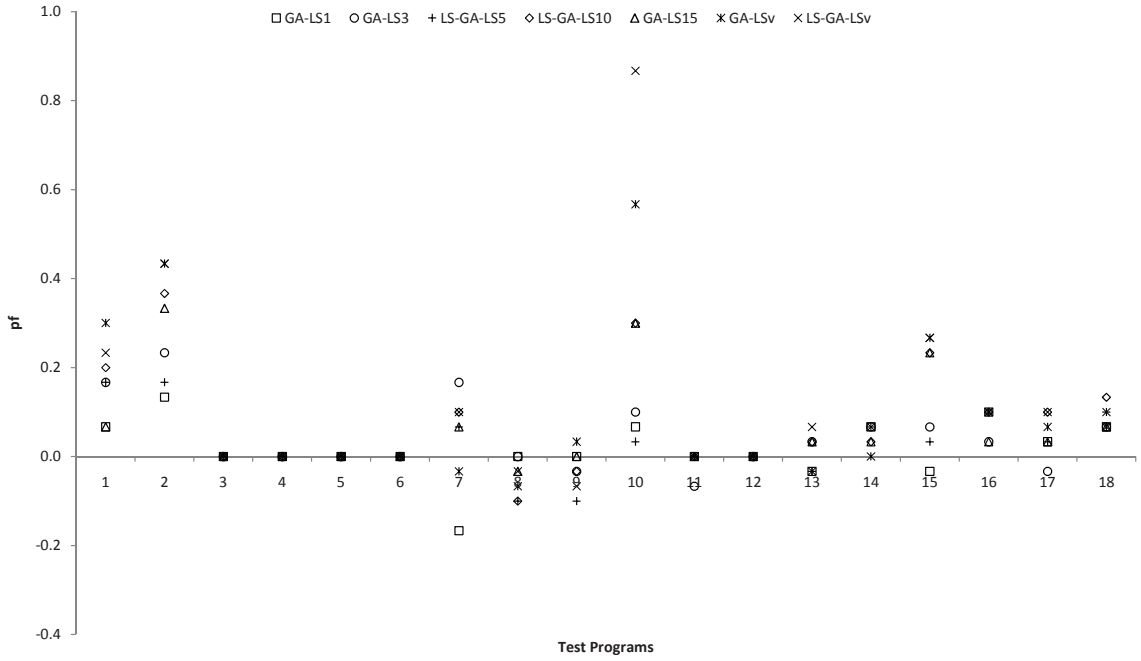


Figure 7.15: Comparison of PF for the best hybrids in LS size group

Figure 7.21 shows **PF** with **RSR1-C** and **RSR2-C** on LS-GA-LSv110 hybrid. The application of rules has suggested the hybrid to stop earlier in some test programs, resulting in missing some paths that should be found if the search continued. The biggest loss is suffered by **RSR1-C** with 1.1 paths missing while **RSR2-C** missed 0.37 paths. The test program that missed the most paths is test program 21.

Figure 7.22 presents the efficiency achieved by applying **RSR1-C** and **RSR2-C** on LS-GA-LSv110 hybrid. **RSR1-C** and **RSR2-C** could achieve efficiency about 80% and 30%, respectively. The range of variation in **RSR2-C** is twice that in **RSR1-C**.

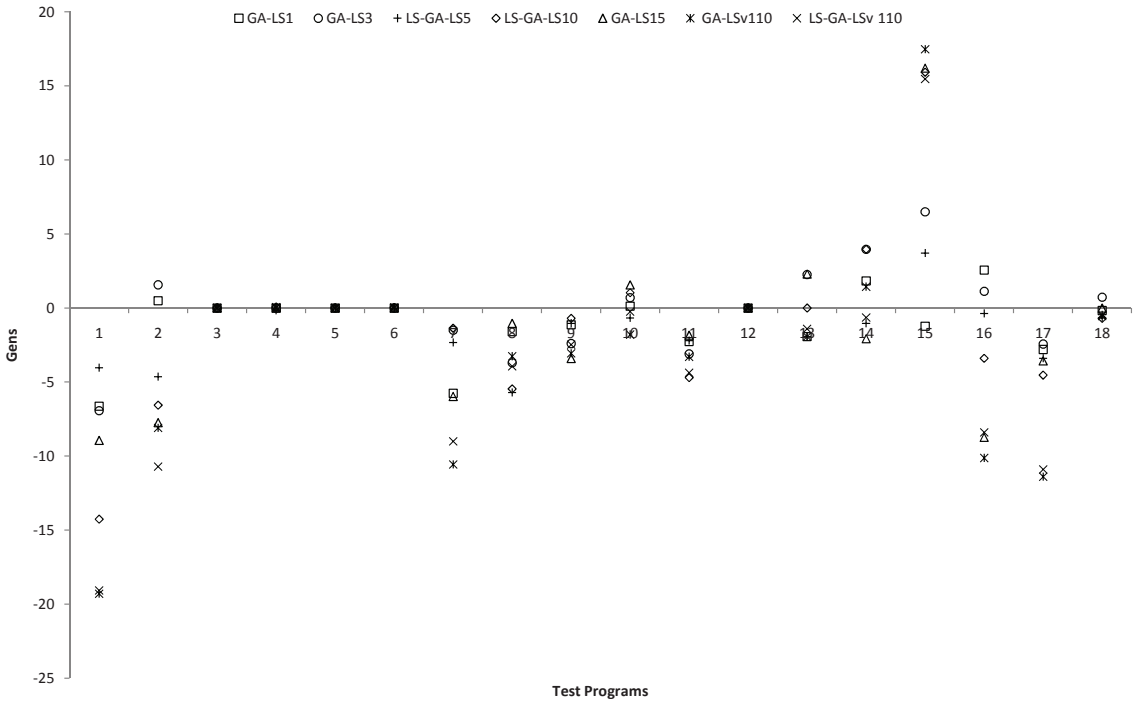


Figure 7.16: Comparison of Gens for the best hybrids in LS size group

7.6 Analysis (Hybrid & Local Search)

7.6.1 Class of Test Programs

The experimental results of hybrid GA application can be classified into the following groups:

1. **CO**. Nothing changes in both path coverage and efficiency. Hybridization is of no benefit to GA.

2. **CD**. Both path coverage and efficiency experience degradation. Implementation of LS is a disadvantage to GA in both respects.

3. **CDP**. Less path coverage. There is no change in efficiency, but less path coverage has failed the whole idea of hybridization.

Table 7.9: Testing, Hybrid GA with variable LS size and All Paths

No	Program	GA		LS-GA-LS10		GA-LSv110		LS-GA-LSv110	
		PF	Gen	PF	Gen	PF	Gen	PF	Gen
1	tA2008	3.70	23.80	3.90	9.53	4.00	4.50	3.93	4.73
2	mmA2008	11.80	17.57	12.17	11.00	12.23	9.47	12.23	6.87
3	iA2008	5.00	1.17	5.00	1.17	5.00	1.17	5.00	1.17
4	bisA2008	6.00	1.83	6.00	1.90	6.00	1.83	6.00	1.90
5	binA2008	7.00	1.00	7.00	1.00	7.00	1.00	7.00	1.00
6	bubA2008	4.00	1.10	4.00	1.13	4.00	1.10	4.00	1.13
7	gA2008	3.57	20.97	3.67	19.60	3.53	10.40	3.67	11.97
8	rA2008	3.97	16.77	3.87	11.30	3.90	13.50	3.93	12.83
9	mtA2008	17.53	11.60	17.50	10.90	17.57	8.53	17.47	9.13
10	tM2004	4.07	17.83	4.37	18.87	4.63	16.03	4.93	17.60
11	eiR1985	3.00	9.93	3.00	5.23	3.00	6.63	3.00	5.57
12	qG1997	4.00	1.00	4.00	1.00	4.00	1.00	4.00	1.00
13	ttB2002	7.03	3.07	7.07	3.07	7.00	1.13	7.10	1.67
14	eiB2002	4.90	12.53	4.93	16.50	4.90	13.97	4.97	11.90
15	qB2002	8.63	3.00	8.87	18.93	8.90	20.47	8.90	18.47
16	scB2002	3.83	11.30	3.93	7.90	3.93	1.17	3.93	2.90
17	fcB2002	4.90	16.03	5.00	11.50	4.97	4.63	5.00	5.13
18	fB2002	7.87	4.87	8.00	4.17	7.93	4.40	7.97	4.33
19	bG2011	11.00	1.47	11.00	1.33	11.00	1.47	11.00	1.33
20	fG2011	9.30	58.73	9.53	48.23	10.77	44.33	11.03	49.73
21	sG2011	17.77	65.73	21.07	73.83	29.97	63.63	29.97	59.80
	Average	7.09	14.35	7.33	13.24	7.82	10.97	7.86	10.96

4. **CDE**. Less efficient. Hybridization has added extra cost without any improvement in path coverage.

5. **CI**. Both path coverage and efficiency have improved. The hybridization shows its fruitful result.

6. **CIP**. More path coverage but no change in efficiency. The hybrid pri-

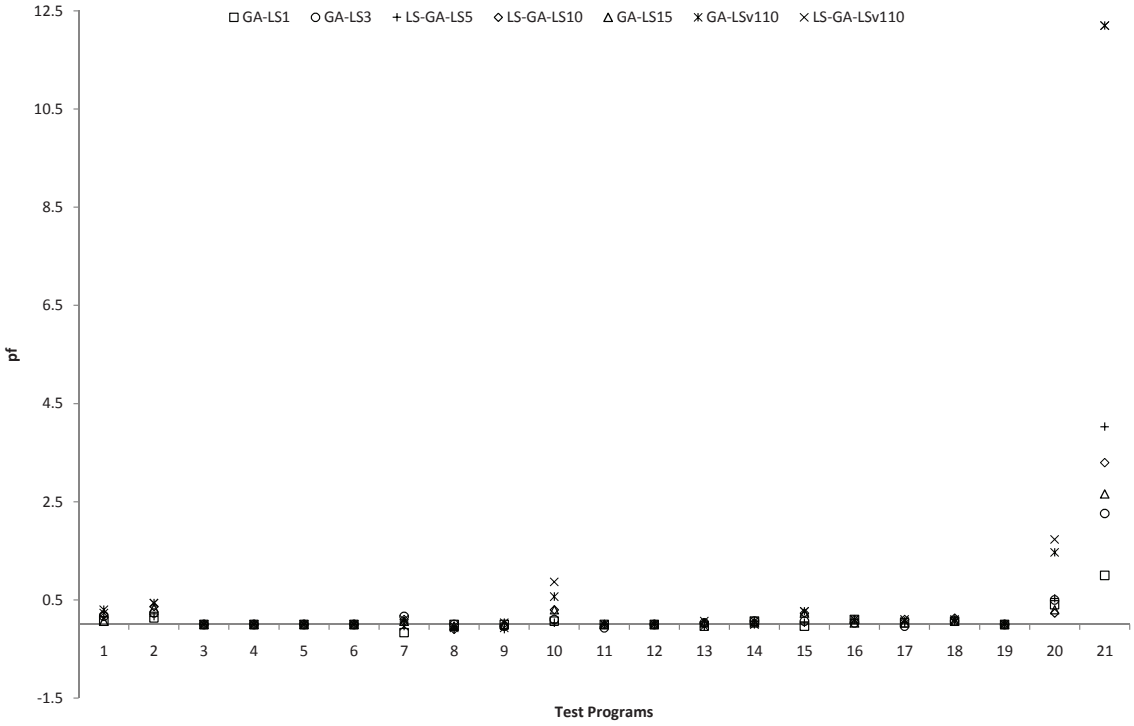


Figure 7.17: Testing, Comparison of PF for all hybrids with all paths

oritized effectiveness over efficiency and yet still maintained efficiency.

7. **CIE**. More efficient but no change in path coverage. Hybrid GA has better efficiency and is advisable to use.

8. **CIDP**. More path coverage but less efficient. Assuming that effectiveness should be prioritized over efficiency, hybrid has benefit.

9. **CIDE**. More efficient but less path coverage. Hybrid GA has failed to preserve the most important objective, i.e. path coverage. So, it is suggested not to be implemented.

The two most desirable classes are CI and CIP. The two least desirable ones are CD and CDP. The Following Table 7.12 shows test programs classification based on the best hybrid LS-GA-LSv (see Table 7.9 and Figure

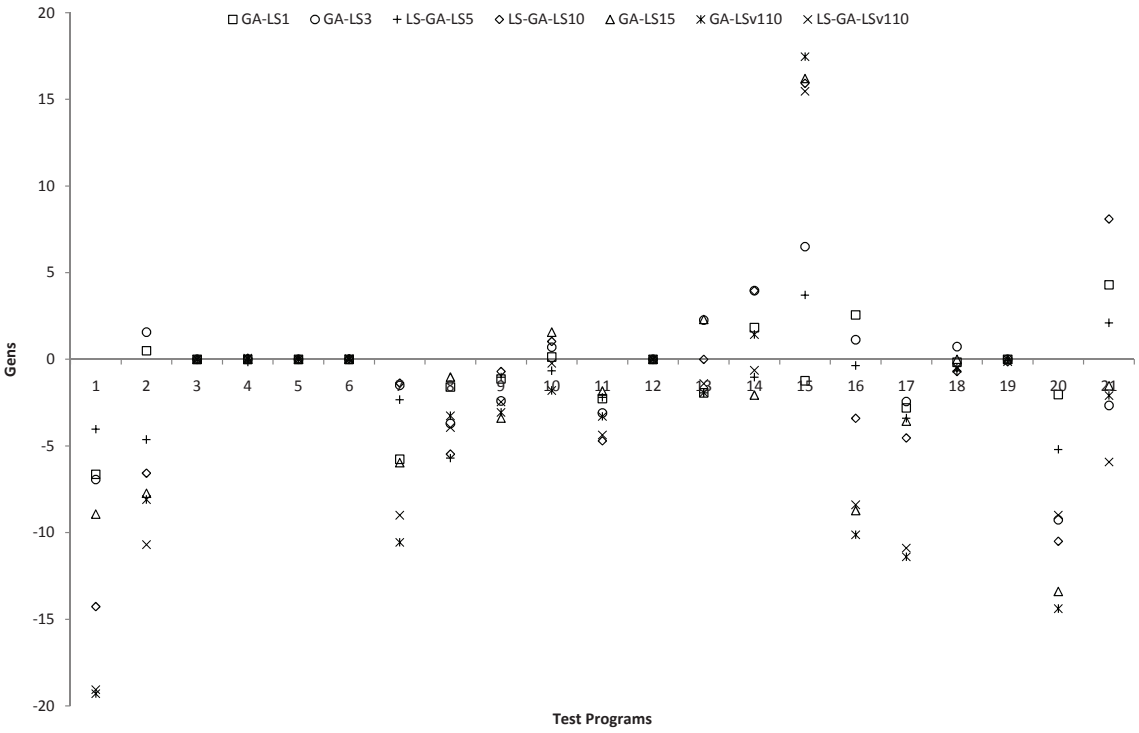


Figure 7.18: Testing, Comparison of Gens for all hybrids with all paths

7.17). The rows have been organized in such a way that from top to bottom is from the most desirable class to the least desirable one.

In Table 7.12, most of the test programs are in the most desirable class CI, i.e. 11 test programs. Another 3 test programs are still in the desirable classes CIDP and CIE. CO is the mid-class, which has 3 test programs with no improvement in any ways. Four test programs are in the less desirable classes CDE and CIDE. None of the test programs is in the least desirable classes CDP and CD.

The distribution of test programs in Table 7.12 suggests that the hybrid is better than GA. In other words, more test program performances can be improved by the hybrid.

Table 7.10: Testing, Hybrid GA with Feasible Paths

No	Program	Paths		GA		GA-LS10		LS-GA-LS10		GA-LSv110		LS-GA-LSv110	
		All	Feas	PF	Gen	PF	Gen	PF	Gen	PF	Gen	PF	Gen
1	tA2008	4	4	3.70	23.80	3.83	17.53	3.90	9.53	4.00	4.50	3.93	4.73
2	mmA2008	13	13	11.80	17.57	12.13	10.37	12.17	11.00	12.23	9.47	12.23	6.87
3	iA2008	6	5	5.00	1.07	5.00	1.07	5.00	1.07	5.00	1.07	5.00	1.07
4	bisA2008	9	9	5.97	2.00	5.97	2.13	6.00	1.83	5.97	2.07	6.00	1.83
5	binA2008	7	7	7.00	1.00	7.00	1.00	7.00	1.00	7.00	1.00	7.00	1.00
6	bubA2008	15	4	4.00	1.13	4.00	1.13	4.00	1.07	4.00	1.13	4.00	1.07
7	gA2008	8	5	4.40	29.23	4.30	21.43	4.13	24.83	4.33	10.93	4.27	21.00
8	rA2008	5	4	3.77	16.00	3.73	9.40	3.80	18.10	3.73	18.20	3.77	14.37
9	mtA2008	52	20	17.93	41.40	18.37	51.90	18.23	31.77	18.67	43.30	18.77	45.43
10	tM2004	8	7	4.00	20.90	4.17	22.67	4.17	20.57	4.37	19.33	4.37	24.80
11	eiR1985	12	3	2.60	23.03	2.63	15.20	2.77	22.90	2.70	13.00	2.63	15.00
12	qG1997	21	4	4.00	1.00	4.00	1.00	4.00	1.00	4.00	1.00	4.00	1.00
13	ttB2002	8	8	7.03	3.07	7.03	1.13	7.07	3.07	7.00	1.13	7.10	1.67
14	eiB2002	31	5	4.93	16.53	4.97	18.47	4.97	18.30	4.47	2.33	4.43	1.27
15	qB2002	27	10	8.90	15.60	8.87	14.03	8.70	15.27	8.83	17.13	8.83	24.33
16	scB2002	15	4	3.87	16.77	3.97	8.03	4.00	7.87	4.00	1.97	4.00	1.93
17	fcB2002	5	5	4.90	16.03	4.87	10.63	5.00	11.50	4.97	4.63	5.00	5.13
18	fB2002	32	8	7.67	6.33	7.67	2.07	7.53	1.87	7.67	2.07	7.57	4.70
19	bG2011	20	11	11.00	1.47	11.00	1.47	11.00	1.47	11.00	1.47	11.00	1.47
20	fG2011	30	30	9.30	58.73	9.70	46.97	9.53	48.23	10.77	44.33	11.03	49.73
21	sG2011	32	32	17.77	65.73	20.00	62.60	21.07	73.83	29.97	63.63	29.97	59.80
	Average	17.14	9.43	7.12	18.02	7.29	15.25	7.33	15.53	7.84	12.56	7.85	13.72

All test programs in class CI by using GA has **PF** 7.16 paths and **Gen** 22.95 generations on the average. The hybrid is able to find 8.61 paths, which is 1.45 paths more than GA. In term of path finding, 1.45 paths is very meaningful. Further, the hybrid has the chance to make use of LS during the search and is able to achieve efficiency – about 30% generations less than GA, or 16.06 generations, using the hybrid.

In class CO, all test programs have found all their feasible target paths in very few generations. So, it is most likely that the LS in the hybrid version has not really been engaged for finding the paths. This explains why it is not getting more paths and/or fewer generations in searching for the target paths than GA.

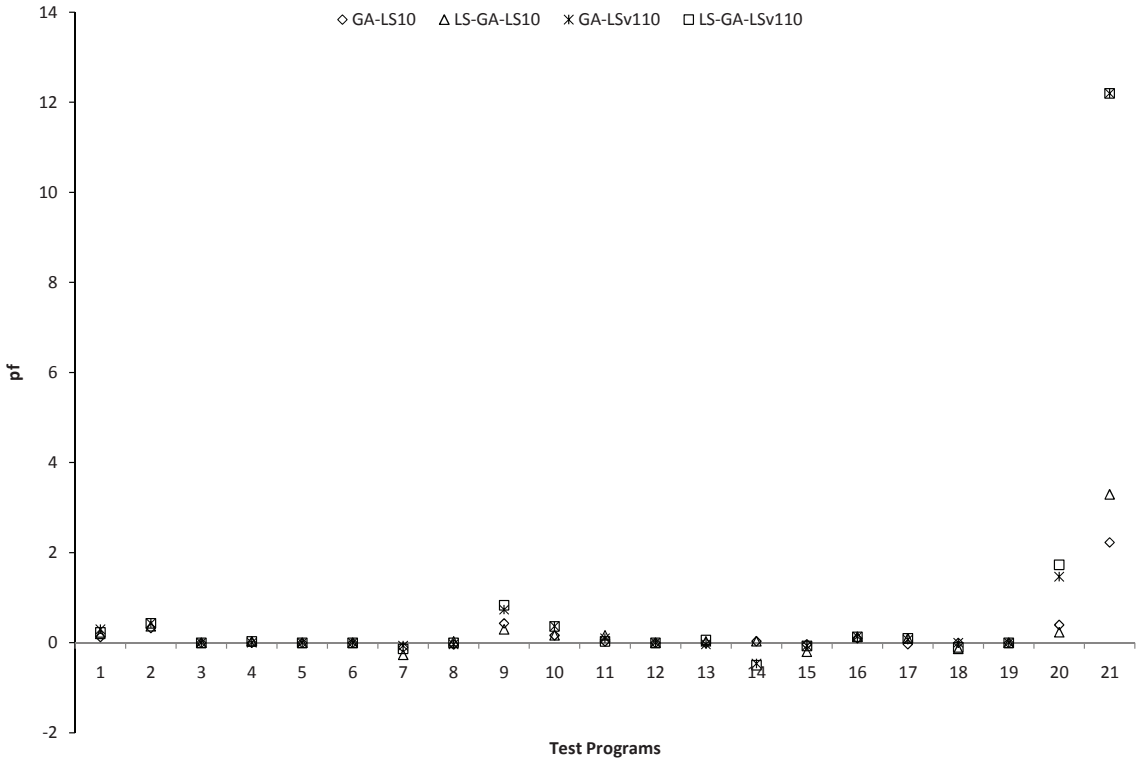


Figure 7.19: Testing, Comparison of PF for LS=10 and LSv hybrids with feasible paths

Test programs in class CDE, i.e. test program 4 (bisA2008) and 6 (bubA2008), have the average of **Gen** less than 2 generations each, i.e. 1.9 and 1.13, respectively. This indicates that additional process in the hybrid already adds more computational time than helping in finding more paths.

As for test programs in class CIDE, the improvement in efficiency has been traded off with a bit fewer paths. However, the loss on paths found is very small, less than 0.1 paths in any test program, i.e. 0.03 paths for test program 8 (rA2008) and 0.07 paths for test program 9 (mtA2008). In contrast, the improvement achieved in term of Gen is above 20% in any test programs, i.e. 3.93 out of 16.77 generations (23.4%) for test program 8 and 2.47 out of 11.6 generations (21.3%) for test program 9. So, the trade-off for

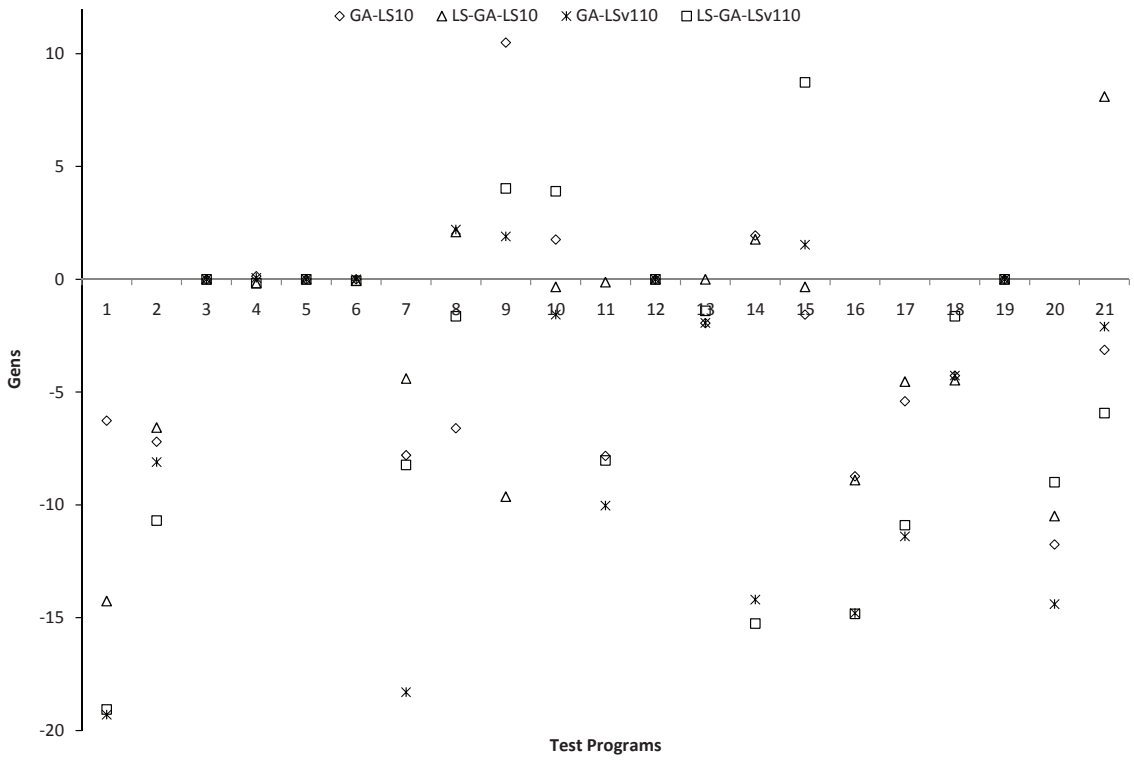


Figure 7.20: Testing, Comparison of Gen for LS=10 and LSv hybrids with feasible paths

the test programs in class CIDE is still acceptable because the loss in number of paths is small compared to the gain in efficiency.

At test program 21 (sG2011), the hybrid found almost 70% paths more than GA. Test program sG2011 has the most feasible paths among the test programs, i.e. 32 paths, and no loops. Test program 20 (fG2011) has the second largest number of feasible paths, i.e. 30 paths, but its input space size is at least 40,400 times that of sG2011 (see Table 4.2). This makes it harder for the hybrid to find more paths in fG2011 than in sG2011. All the test programs that gained 0.5 paths more have no loops at all (see Table 4.2). In detail, the improvements on GA are 0.87 paths for tM2004 (test program 10), 1.73 paths for fG2011, and 12.2 paths for sG2011 (see Table

Table 7.11: Testing, Hybrid GA with SRMSC

No	Program	Paths		GA		RSR1-C			RSR2-C		
		All	Feas	PF	Gen	PF	Gen	Eff	PF	Gen	Eff
1	tA2008	4	4	3.83	8.40	3.27	30.70	69.30	3.27	94.83	5.17
2	mmA2008	13	13	11.63	16.50	10.29	24.00	76.00	10.80	61.33	38.67
3	iA2008	6	5	5.00	1.17	5.00	22.20	77.80	5.00	70.40	29.60
4	bisA2008	9	9	6.00	1.83	6.00	20.90	79.10	6.00	65.87	34.13
5	binA2008	7	7	7.00	1.00	7.00	19.00	81.00	7.00	59.00	41.00
6	bubA2008	15	4	4.00	1.10	4.00	25.27	74.73	4.00	41.40	58.60
7	gA2008	8	5	3.57	20.97	1.04	7.00	93.00	2.77	100.03	-0.03
8	rA2008	5	4	3.97	16.77	3.77	27.67	72.33	3.93	72.27	27.73
9	mtA2008	52	20	17.53	11.60	13.67	32.00	68.00	15.50	87.20	12.80
10	tM2004	8	7	4.20	26.63	1.07	7.00	93.00	3.23	102.70	-2.70
11	eiR1985	12	3	3.00	9.93	1.05	6.15	93.85	2.00	66.69	33.31
12	qG1997	21	4	4.00	1.00	4.00	25.00	75.00	4.00	38.00	62.00
13	ttB2002	8	8	7.03	3.07	7.00	19.13	80.87	7.00	59.53	40.47
14	eiB2002	31	5	4.90	12.53	5.00	23.83	76.17	4.87	64.73	35.27
15	qB2002	27	10	8.63	3.00	7.50	19.50	80.50	8.73	56.70	43.30
16	scB2002	15	4	3.83	11.30	1.00	7.00	93.00	2.15	91.30	8.70
17	fcB2002	5	5	4.90	16.03	5.00	24.67	75.33	4.30	59.57	40.43
18	fB2002	32	8	7.87	4.87	7.69	20.75	79.25	7.83	61.43	38.57
19	bG2011	20	11	11.00	1.47	9.50	24.50	75.50	10.80	65.90	34.10
20	fG2011	30	30	9.30	58.73	6.70	25.90	74.10	6.90	87.53	12.47
21	sG2011	32	32	17.77	65.73	16.29	34.94	65.06	20.95	89.63	10.37
	Average	17.14	9.43	7.09	13.98	5.99	21.29	78.71	6.72	71.24	28.76

7.9 in columns GA and LS-GA-LSv110).

There is not enough evidence to say in advance whether using the hybrid application will be beneficial, from test program input size, cyclomatic complexity number, and structure of loops. In other words, so far there is no direct commonality in these terms above of the test programs that fall into desirable classes. However, it is suspected that there is such complex relation among the terms and number of feasible paths. This is a topic for future research.

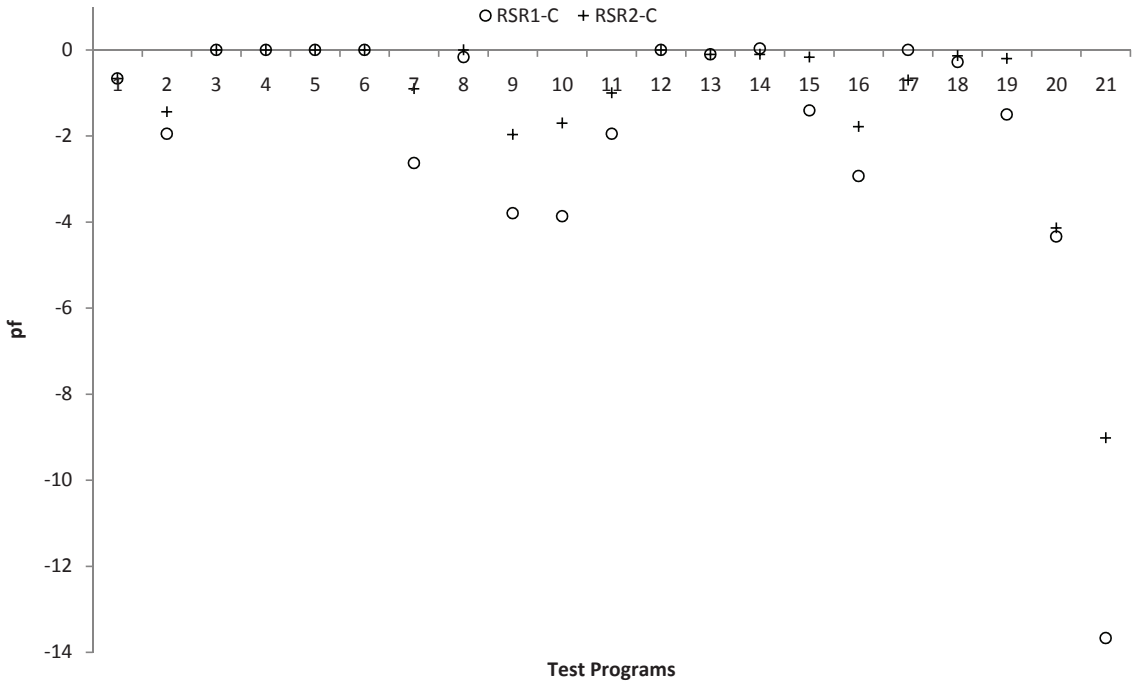


Figure 7.21: Comparison of PF between RSR1-C and RSR2-C for LS-GA-LSv110

Table 7.12: Classification of Test Programs

Class	Test Programs
CI	1, 2, 7, 10, 13, 14, 16, 17, 18, 20, 21
CIP	
CIDP	15
CIE	11, 19
CO	3, 5, 12
CDE	4, 6
CIDE	8, 9
CDP	
CD	

In general, the use of hybrid is encouraged as it performs better than GA. In order for hybrids to be more useful, there are two things to be considered. Firstly, whatever hybridization is being used, it should have a chance to be exercised, i.e. making sure that LS is called at least once during the evolution.

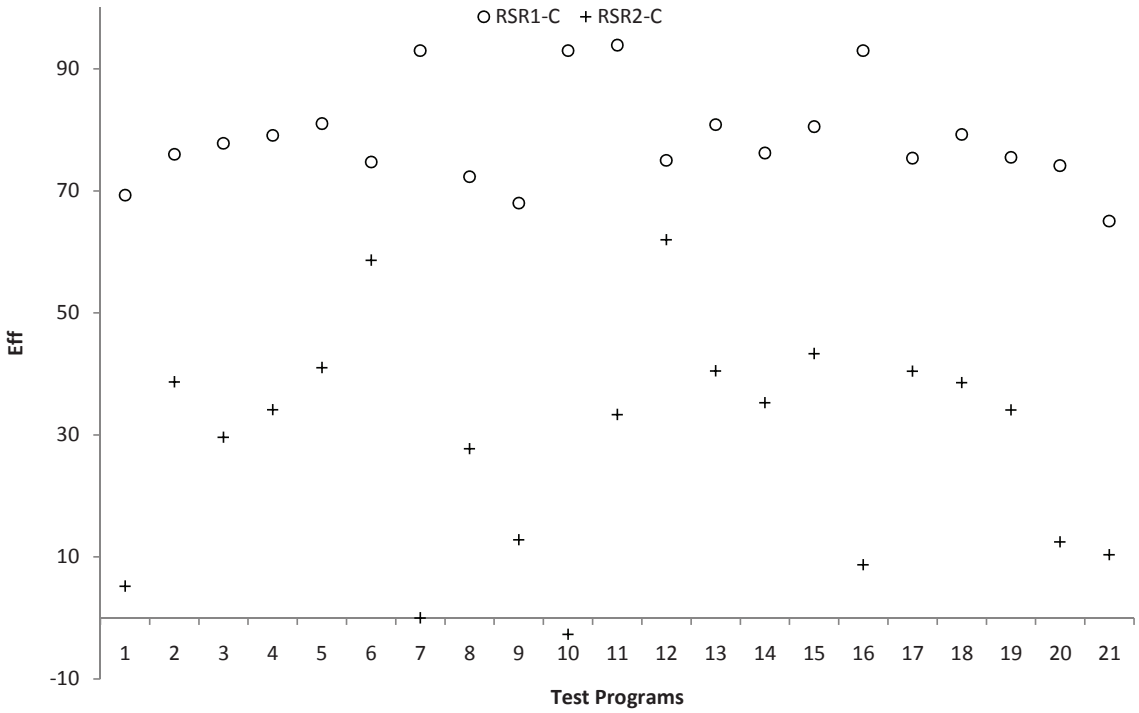


Figure 7.22: Comparison of Eff between RSR1-C and RSR2-C relative to LS-GA-LSv110

Essentially, the more it is called the better. Secondly, a test program with at least 5 or more feasible target paths is likely to get the benefit out of hybrid. The more paths to cover the more reason to use the hybrid.

7.6.2 Effects of Infeasible Paths

Making all target paths feasible by removing all the infeasible ones does not cause GA and its hybrids to perform better. On the contrary, removing infeasible paths has made GA to find 0.03 more paths, but also made GA less efficient by 25.6% (compare **PF**(GA) columns in Table 7.9 and Table 7.10).

Statistical paired t-Test of **PF** has shown that there is no significant

difference between having infeasible paths (see **PF**(GA) column in Table 7.9) and not having infeasible paths (by means of deliberate removal of infeasible paths; see **PF**(GA) column in Table 7.10). The null hypothesis of the test says that both samples, i.e. before and after feasibility treatments of target paths, are taken from distributions with equal population means. The test gave a two-tail P($T <= t$) value of 0.55, which is not enough evidence to reject the null hypothesis at 5% significance level.

Similar statistical test for **Gen** has indicated that there is a significant different between allowing and avoiding infeasible target paths. The two-tail P($T <= t$) yields 0.03 value, which means the test has enough evidence to reject the null hypothesis at 5% significance level. **Gen** is significantly lower when infeasible paths are not removed.

Intuitively, having infeasible paths will add diversity in the evolutionary process. The population will be evolving in different directions as it attempts to cover the infeasible paths, leading to faster coverage of the feasible paths. So, the number of paths found will be about the same amount on average, but it will be covered faster. This is important because it means the difficult task of analyzing all paths to determine if they are feasible is not necessary; it is even detrimental.

This finding is general as long as the fitness constructs are functions of paths that consists of branch distance and approximation level. As for the genetic operators, they are not parts of the fitness function that guides the search. Rather, their roles are to explore and exploit the input search space according to the direction that is provided by the fitness function.

7.6.3 SRMSC for hybrid

Statistical t-Test paired two sample for means is conducted to find out whether the treatment of applying SRMSC rules will affect LS-GA-LSv110 hybrid performance or not. The rules used are the ones that use common parameters setup (recall Section 6.4.4), i.e. **RSR1-C** and **RSR2-C**. The test are NR (no rule) vs. **RSR1-C**, NR vs. **RSR2-C**, and **RSR1-C** vs. **RSR2-C**. Each test is applicable to both terms **PF** and **Gen**, so there are six tests in total.

In term of PF, NR vs. **RSR1-C** test results t 2.78, P(T<=t) two-tail 0.01, and t Critical two-tail 2.09. This means that there is a significant difference for having **RSR1-C** with 1.8 paths less than without the rule. For NR vs. **RSR2-C**, the results are t 2.53, P(T<=t) two-tail 0.02, and t critical two-tail 2.09. This indicates that there is a significant difference for having **RSR2-C** with 1.14 paths less. As for **RSR1-C** vs. **RSR2-C**, the test yields t -2.81, P(T<=t) two-tail 0.01, and t critical one-tail 1.72. This points out that **RSR1-C** is significantly producing 0.72 paths less than **RSR2-C**. This is not surprising since **RSR2-C** involves more stringent thresholds and is expected to keep searching for longer.

In the case of **Gen**, NR vs. **RSR1-C** test results t -3.08, P(T<=t) two-tail 0.01, and t critical one-tail 1.72. In addition, **RSR1-C** suggest 10.33 generations more than NR (10.96 generations) on average (compare **Gen** of LS-GA-LSv110 between in Table 7.9 and in Table 7.11). However, **RSR1-C** has an almost half standard deviation of **Gen** than NR, which tells that it is more stable. NR vs. **RSR2-C** test produces t -15.32, P(T<=t) two-tail 0.01, and t critical one-tail 1.72. This indicates very significant different

Gen between NR and **RSR2-C**. On average, **RSR2-C** recommends 60.28 generations more than NR (10.96 generations). This is an indication of very significant different **Gen** between NR and **RSR2-C**. The last test results for **RSR1-C** vs. **RSR2-C** returns t -10.69, P(T<=t) two-tail 0.00, and t critical one-tail 1.72. This reveals that **RSR2-C** advocates more generations, by 49.95 generations on average.

Logically, recommended stopping generations proposed by the rule fall into the following cases. Firstly, if all target paths are feasible and all of them were found before the recommended number of generations, the evolution will stop immediately. In this case, the rule has no effect. Secondly, infeasible paths will force the evolutionary process to keep on going forever should they exist in the target paths. The stopping rule avoids this problem, by ordering to stop at a certain number of generations and yet still have confidence that most if not all of the feasible ones are already found.

7.6.4 Statistical Test for Hybrids

7.6.4.1 All Paths

Table 7.13 summarizes all the t-Tests paired two sample for means conducted between GA and each of its hybrids across all 21 test programs. The legend is t for Statistic t, P1 for P(T<=t) one-tail, t1 for t critical one-tail, P2 for P(T<=t) two-tail, t2 for t critical two-tail, d-mean for **PF** mean different between the hybrid and GA, and H0 for decision on accepting or rejecting H0.

In general, the hybrids are not significantly different with GA. The test

Table 7.13: PF t-Test Paired Two Sample for Means

No	Paired two sample	t	P1	t1	P2	t2	d-mean	H0
1	GA vs. GA-LS1	-1.58	0.06	1.72	0.13	2.09	0.08	accept
2	GA vs. GA-LS3	-1.57	0.07	1.72	0.13	2.09	0.17	accept
3	GA vs. LS-GA-LS5	-1.27	0.11	1.72	0.22	2.09	0.24	accept
4	GA vs. LS-GA-LS10	-1.53	0.07	1.72	0.14	2.09	0.24	accept
5	GA vs. GA-LS15	-1.56	0.07	1.72	0.13	2.09	0.20	accept
6	GA vs. GA-LSv10	-1.27	0.11	1.72	0.22	2.09	0.73	accept
7	GA vs. LS-GA-LSv10	-1.33	0.10	1.72	0.20	2.09	0.77	accept

results reveal that there is not enough evidence to reject the null hypothesis (H0), which is the two samples are taken from the same population distribution. The acceptance is shown by P2 in Table 7.13: all values above 0.1, which is higher than the 5% significance level.

In Table 7.13, d-mean values for the first 5 fixed LS-size hybrids are much less than 0.5 paths. This means that the hybrids have less possibility even to find another one path. However, the variable LS-size hybrids (recall LSv hybrids) are likely able to find another path as its values are above 0.7 paths.

Table 7.14 presents all the test results for **Gen**. P2 values in the table mostly are greater than 0.05. This means that having additional LS in hybrids do not really add computation on average. There are 3 paired two samples that reject the null hypothesis: GA vs. LS-GA-LS5, GA vs. GA-LSv110, and GA vs. LS-GA-LSv110. For these 3, the differences are significant. For example, **Gen** is reduced from 14.35 on average to 10.96 with LS-GA-LSv110, a relative improvement of 24%. Table 7.9 shows that **Gen** is better, not worse, with the hybrid methods than with GA.

Table 7.14: Gen t-Test paired two sample for means

No	Paired two sample	t	P1	t1	P2	t2	d-mean	H0
1	GA vs. GA-LS1	1.45	0.08	1.72	0.16	2.09	0.77	accept
2	GA vs. GA-LS3	0.96	0.17	1.72	0.35	2.09	0.72	accept
3	GA vs. LS-GA-LS5	2.60	0.01	1.72	0.02	2.09	1.32	reject
4	GA vs. LS-GA-LS10	0.82	0.21	1.72	0.42	2.09	1.10	accept
5	GA vs. GA-LS15	1.44	0.08	1.72	0.17	2.09	1.82	accept
6	GA vs. GA-LSv10	2.09	0.02	1.72	0.05	2.09	3.38	reject
7	GA vs. LS-GA-LSv10	2.31	0.02	1.72	0.03	2.09	3.39	reject

7.6.4.2 Feasible Paths

In order to be comparable, all hybrids use the same LS size 10. Target paths of each program are only the feasible ones.

Table 7.15 shows t-Test results for **PF** for all pairs. P2 values for all pairs are at least 0.12, which is greater than the 5% significance level. This means that no hybrid has a significantly different population distribution from any other. The hybrids with variable LS size has more chance to find a path more than GA, i.e. at least 0.7 paths for each of GA-LSv110 and LS-GA-LSv110 (see d-mean column in Table 7.15).

Table 7.15: PF t-Test paired two sample for means of feasible paths

No	Paired two sample	t	P1	t1	P2	t2	d-mean	H0
1	GA vs GA-LS10	-1.62	0.06	1.72	0.12	2.09	0.17	accept
2	GA vs LS10-GA	-1.12	0.14	1.72	0.28	2.09	0.12	accept
3	GA vs LS10-GA-LS10	-1.36	0.09	1.72	0.19	2.09	0.21	accept
4	GA vs GA-LSv110	-1.24	0.11	1.72	0.23	2.09	0.72	accept
5	GA vs LS10-GA-LSv110	-1.26	0.11	1.72	0.22	2.09	0.73	accept

In Table 7.16, only one pair is unable to reject H0 (see column P2). This indicates that most of the hybrids significantly require fewer generations

compared to GA, except for LS10-GA hybrid (see column Gen for GA and all hybrids in Table 7.10). Although LS10-GA has no significant difference with GA, it is still bit more efficiency than GA, because it only calls LS once at the beginning compared to the other hybrids that could call LS multiple times during the evolution. On average, the hybrids needs 3 fewer generations than GA.

Table 7.16: Gen t-Test paired two sample for means of feasible paths

No	Paired two sample	t	P1	t1	P2	t2	d-mean	H0
1	GA vs GA-LS10	2.56	0.01	1.72	0.02	2.09	2.77	reject
2	GA vs LS10-GA	2.01	0.03	1.72	0.06	2.09	2.23	accept
3	GA vs LS10-GA-LS10	2.21	0.02	1.72	0.04	2.09	2.49	reject
4	GA vs GA-LSv110	3.46	0.00	1.72	0.00	2.09	5.46	reject
5	GA vs LS10-GA-LSv110	2.75	0.01	1.72	0.01	2.09	4.29	reject

This means that statistically, there is no significant difference in paths found, but there is a significant difference in the number of generations. In general, when confronted with a new test program some infeasible paths are likely. So it is likely that hybrid is best.

7.6.5 Repetition Issue

In practice, the number of repetitions (runs) required to get certain percentage of coverage is very important for path testing. It will add more confidence level to the tester that the test is sufficient. Actually, this is the case for almost any empirical software engineering and not only path testing.

In Figure 7.23, in the first run, GA can achieve path coverage of 91% using common parameters setup and 97% using the best parameters setup

on average out of 198 feasible paths. GA covers maximum 96% target paths using the common parameters setup from run eight up to run thirty, while best parameters setup reaches 100% coverage at run twenty-four.

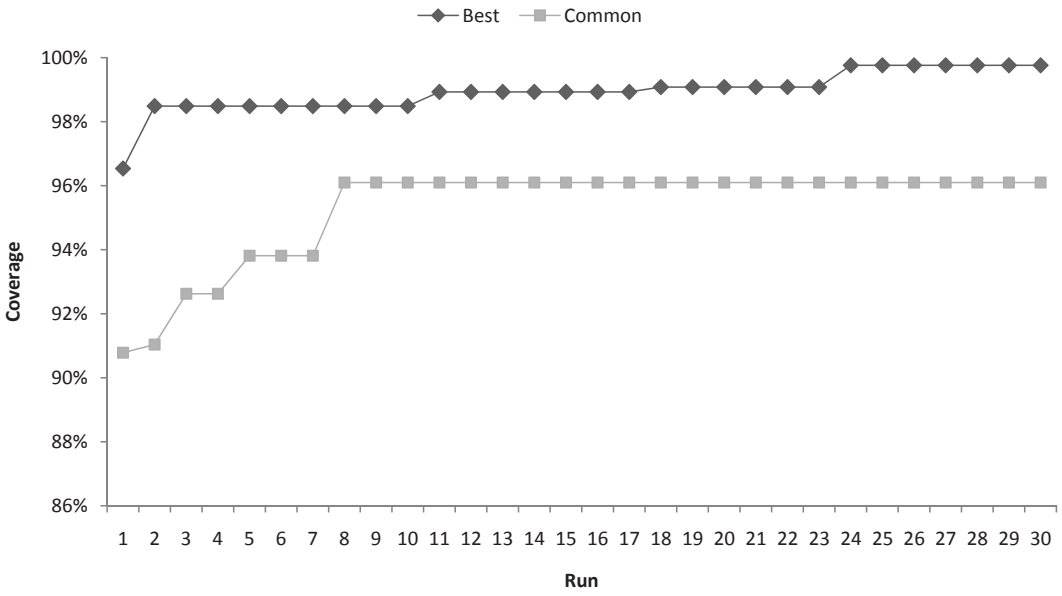


Figure 7.23: Cumulative Path Coverage of Best vs. Common Parameter Setups for GA

In that light, it is valuable to know that 96% of target paths are found in only 8 runs, with common parameters. We have used 30 runs throughout the thesis to provide data for statistical analysis, but in real operation we see that far fewer runs would be needed.

In reality, given a new test program, one has no idea of its best parameter settings, so common settings are the ones to consider.

7.7 Threats to Validity

Two things considered as crucial threats in hybridizing GA with LS for path testing are external and internal threats.

An external threat is an outside factor that could affect in taking conclusions of the experiments. For example, number of test programs, perception, decision maker influence, and tester/programmer interests. The number of test programs might have affected the generalization of the empirical conclusions. To handle this threat, 21 test programs were involved in the experiments with varying characteristics in terms of CC, path complexity (or number of feasible/infeasible target paths), and input space. This is way above the average number of test programs in research on coverage testing, i.e. 6 test programs (see Table 3.3).

Internal threats would be contributing to faulty approach, hybrid, or experiment. The internal threat is to make sure that the components of hybrid are integrated and exercised properly. LS must be called several times in order to fully use its advantage to exploit the surrounding search space of potential input data. The more it is called, the more chance it can contribute to overall performance. Therefore, to overcome this potential threat, GA was hybridized with different sizes and ordering position of LS, and meeting statistical requirements such as each treatment is repeated 30 times with different seeds each time, and conducting statistical test to support the conclusions.

7.8 Conclusions

Some hybridization techniques between GA and LS for path testing have been proposed. Various techniques are based on the position of LS in the evolutionary process, the number of LS calls, and LS size. Based on these

criteria, 17 hybrids are defined, which consist of 15 hybrids with fixed LS size (LSx; x for size) and 2 hybrids with variable LS size (LSvx; x for size). Five LS sizes are proposed: 1, 3, 5, 10, and 15. Each class can form combinations of LS-GA, GA-LS, and LS-GA-LS. For example, class LS1 consist of LS1-GA, GA-LS1, and LS1-GA-LS1 (or LS-GA-LS1). So, LSx hybrids are 15 in total. LSvx hybrids consist of GA-LSv110 and LS-GA-LSv110, where x is 110, which means the size ranges from 1 to 10 with 1 increment each time it is called.

Eighteen test programs were used for training, to find out which LSx hybrids outperform others in the same LS size class. The same test programs were also used to train LSvx hybrids and compare their performance with LSx hybrids. Another 3 new test programs were added for testing the best LSx hybrids in its class and LSvx hybrids. So, there are 21 test programs for testing in total.

Training results yield five best LSx hybrids: GA-LS1, GA-LS3, LS-GA-LS5, LS-GA-LS10, and GA-LS15. The best and the second best among other hybrids are LS-GA-LSv110 and its sibling GA-LSv110, respectively. Testing has shown that LS-GA-LSv110 outperforms other hybrids. Hybrid LS-GA-LSv110 found on average 0.77 paths more and required 3.39 generations less than GA. The proposed hybrids have empirically been proven to be more effective and more efficient on average than GA alone.

The existence of infeasible paths does not hinder the evolutionary process in covering the feasible ones. On the contrary, it increases the selection pressure and perturbation evolution, which leads to finding feasible paths more quickly. In other words, removing infeasible paths from target paths,

should the program have them, will make the evolution slower in covering the feasible ones. This is important because it means the difficult task of analyzing all paths to determine if they are feasible is not necessary, and is even detrimental.

For test programs with no a priori knowledge about its target paths feasibility, and stopping number of generations, the following can be used as guidelines. Firstly, it can be left to SRMSC to decide how many generations are required to reach a given confidence level that all feasible paths have been found. The average number of generations in our test programs is about 70, using the most conservative stopping rule, and almost no feasible paths are missed that would be found if searching continued up to 100 generations. Secondly, it only needs about 10 runs or less to find nearly all of the feasible target paths.

Chapter 8

Conclusions and Future Research

The chapter consists of three sections: contributions, findings, and future research. The contributions section discusses the significance of new approaches and explains their experimental limitations. The findings section lists all empirical observations from the experimental results. Research directions are presented in the Future Research section.

As mentioned in Chapter 1, the aim of the thesis is to improve path testing using GA. This is achieved in the thesis by investigating three research areas:

- firstly, some fundamental research questions on path testing using GA, such as how GA-based single objective path testing works, how it behaves as GA parameters vary, what are the limits of the test data generator as test programs vary, what kind of fitness functions are good, how to compare test data generators, and what test programs to use

and how to choose them. This is the subject of Chapter 5.

- specific research questions on how to improve GA-based path testing by stopping earlier whenever it is not worth continuing. This is the subject of Chapter 6.

- Some potential improvements can be done by hybridizing with local search (LS), or by using variants of evolutionary algorithm (EA). Chapter 7 investigated the first of these.

Other avenues for improving evolutionary path coverage include parallelizing to speed up; multiobjective optimization and automation of the whole GA-based path testing. These are promising research questions, but beyond the scope of this thesis.

8.1 Contributions

The thesis has proposed new test program classification systems, new stopping criteria, and hybridization of GA-based test data generators. The following are the major contributions:

1. Collection of test programs. The collection draws together some test programs that are used in path and/or branch testing research. Reconstructable test programs in the collection can be used as benchmarks for white-box testing.

2. Test program classification. Test programs are classified based on their structures, including loops, decisions, and selection expression complexity. The classification helps to understand how GA-based test data

generation works on certain test program characteristics, and can help to determine whether new test programs are useful because programs with similar characteristics are not already represented in the collection.

3. Parameters setup. Identifying a standard parameter setup for GA-based test data generation makes the generator easy to use and more handy to run, when a new test program is encountered. We identified settings for different complexity problems, and which test program attributes determine complexity for this purpose.

4. Evidence is presented that the existence of infeasible paths does not harm test data generation. Making all target paths feasible by removing the infeasible ones causes the test data generator to take more generations to cover the same number of paths than if the infeasible ones are present. In practice, this means that one does not need to worry about infeasible paths in new test programs, so long as one is able to stop searching at some sensible point.

5. Dynamic stopping criteria. A method is presented to stop searching for test data when it seems not worth continuing anymore, based on the history of path coverage. This is particularly important should any target paths be infeasible. This stopping condition is adaptive and data-based, not arbitrary. This also means that there is one less system parameter (the maximum number of generations) to worry about.

6. Hybrid GA-based test data generator. Hybridizing GA with local search LS has been shown able to find more paths more quickly on

average than just using GA. Experiments have compared several hybridization methods and identified the best overall.

8.2 Findings

8.2.1 Classification of Test Programs

Hitherto, selection criteria for a test program to be used in path testing research are not clear. Mostly, they are selected due to having some basic logical control flows, and being used in previous (related) work.

Two classifications of test programs are proposed, based on logical control flows and control expression complexity. They are called structure classification and expression classification, respectively.

In structure classification, a test program is classified based on the number of logical control nodes. A control node can be either an iteration (e.g. FOR, WHILE, and DO -like statements) or selection (e.g. IF, IF-ELSE, and SWITCH -like statements). The numbers of nodes considered are 0, 1, 2, and 3 (or more). The order of two or more nodes can be sequential or nested into one another. Based on the number of nodes and their order, there are 101 classes in total.

In expression classification, a test program is categorized with its number of sub (or simple) control expressions. Two or more control expressions are combined using logical connector(s) AND or OR. Each control expression is either a relational or arithmetic-relational expression. The control expression with the most sub control expressions in a test program is considered for

classification. In total, there are 24 classes in the expression classification.

Lower-bound Cyclomatic Complexity (CCL) number is used to measure class complexity both in structure and expression classifications. The classification has finer granularity level than CCL. Thus, several classes may have equal CCLs.

28 test programs have been identified from the literature. Between them they cover 13 out of 101 classes in the proposed structure classification. There are 5 of 1 test program (1TP) classes (4.95%), 4 2TP classes (3.96%), 3 3TP classes (2.97%), and 1 7TP classes (1%). This leaves a sparse classification, because 88% of classes are not yet represented by a test program.

As for expression classification, all 28 test programs have at least one simple relational expression. Nine test programs have two connected simple expressions that are connected using logical connector AND or OR. Five test programs have three connected simple expressions. Nine classes are filled in with the following details 3 1TP classes, 1 3TP classes, 2 4TP classes, 1 6TP classes, 1 8TP classes, and 1 28TP classes. Thus, 62.5% of classes have no test programs.

Knowing both classifications enables further analysis on test data generator performance and test program characteristics. Results showed that a test program that is likely to be able to achieve full path coverage has the following common characteristics: no or single loop, more feasible target paths (above 20), and simple expressions or expressions with at most one logical connector.

This classification scheme was used to select the 21 test programs used

in this thesis.

8.2.2 Path Complexity

Path complexity is primarily affected by the number of loops or selections. The number of paths, both feasible and infeasible, increases exponentially as the number of loops increases. This also means longer paths due to more branches to be traversed. Infeasible paths do not hinder the search for finding the feasible ones, but they would make the search continue infinitely unless some stopping condition is used other than all target paths being covered.

Having the right stopping criteria is highly recommended, because in reality and practice, no one knows whether the (target) paths of a program are infeasible without complicated analysis. This analysis is tedious and laborious, and gets more difficult as the number of paths and the path lengths increases.

The expression of loop or selection statement also contributes additional complexity to a path. In this case, a more complex expression means more complex function of input for a path that traverses the expression. This is because the expression complexity directly decides the size of the input space. Thus, it could change the input space. Changing the input space could change the probability of covering a path. Less input space means less probability to generate that particular input.

8.2.3 Path Infeasibility

Unless an analytical examination is conducted, saying a path is infeasible is very hard. In reality, dynamic infeasible path detection can only be conducted by conducting exhaustive input search. The search is unlikely to happen due to excessive amount of work and resources required.

More plausible dynamic approaches in detecting infeasible paths are by employing empirical observations and some statistical methods. They are monitoring best fitness improvement over generations, observing correlation between selection branches, and tracking conflict pairs of branch-assignment or branch-branch. However, all these methods are only to detect (potential) infeasible paths and not capable of stating that the paths are infeasible.

Our experimental results show that the presence of infeasible paths among the target paths does not hinder the test data generator in covering the feasible ones, rather they are helpful in term of keeping the selection pressure high so as to maintain competition. The infeasible paths create a competitive environment among generated input data in the population, and more feasible paths are found more quickly. In other words, making target paths all feasible by analytically removing all the infeasible ones means more generations are required to cover the same number of paths as when the infeasible ones are included.

8.2.4 Key Parameters and Parameters Setup

The experimental results have shown that allele range and population size are the two most influential parameters in term of path coverage. The less

influential parameters are number of generations and mutation rates.

Two parameter setups were extracted from the empirical results: best and common parameters setups, respectively. Best parameters were identified for each program while common ones are selected across all test programs.

In reality, best parameters setup is impractical because for each new test program it needs several runs with different combinations of parameter values to figure out. These runs are expensive in term of time and resources. So, the common one is more useful in practice.

For the common parameter setup, population size can be set to 100 and 250 for low-medium and medium-high complexity programs, respectively. The number of generations can be assigned to 50 and 500 for the same complexity levels, respectively. Allele range can be made as narrow as feasible without leaving out input space that covers feasible paths. As a guide, an integer can be set to between -100 and +100, a positive integer between 0 and +200, real about 200 intervals that depends on required accuracy (e.g. 0.01), letter ASCII code, and word based on the letter. Cross over and mutation rates matter less, and can be assigned common values 0.9 and 0.1, respectively.

In general, all parameters are encouraged to be set up in the most restricted manner possible. For example, an integer input can initially be set up between -100 and 100, but if some target paths are still uncovered after certain number of generations then it can be enlarged between -1000 and 1000 or whatever range is reasonable without adding too much computational time. In addition, knowledge about program input and what the program does can be helpful in determining its allele range.

Another important finding is that CC number is not really useful in estimating the complexity of a test program with respect to path testing. For example, there is no pattern allowing us to estimate the number of generations or population size required based on the CC number.

8.2.5 Dynamic Model for Stopping Criteria

A reliability growth model can be used to estimate the number of future failures expected in black-box software testing, based on the history of defects found during testing so far. In white-box testing, the analogy to the number of defects detected so far, and when, is the number of target paths covered so far, and when. This has been the inspiration for a model, based on software reliability growth models, to predict the number of paths that may be expected to be covered in further searching. This can be used as condition to stop generating test data when it is not worth continuing anymore. As the estimated likelihood of finding further paths approaches a certain threshold it is time to stop.

The adopted path coverage model has two parameters, namely the expected number of paths found λ at a certain generation and the rate of reduction in the normalized path coverage rate θ. For parameter θ, the difference between current and previous θ values $\Delta\theta$ is used as a stopping criterion. Changes in the values of these two parameters are monitored as searching proceeds, so that the generator stops when certain thresholds are reached.

Empirical validation has shown that both parameters λ and $\Delta\theta$ are useful as stopping criteria. The best values identified for λ and $\Delta\theta$ are 0.1 and

0.0001, respectively. On average, the application of these criteria missed only 0.1 feasible paths while achieving about 30% more efficiency over 21 test programs. The number of missing feasible paths is small. So, the model is empirically proven to be an effective stopping criterion.

In general, the model can be used as stopping criteria for a test program with no information of its target paths feasibility. The parameters for the model can be assigned initial values as aforementioned. As the number of paths found from the first generation is tracked, gradually the parameter values can be tightened up to increase the likelihood that all feasible paths are covered, and all paths that are not covered are infeasible.

8.2.6 Hybridization

Seventeen hybrid variants of GA-based test data generator with local search LS were proposed. They are designed based on the following criteria, i.e. relative position of LS with GA, fixed LS size, and variable LS size. Two of them incorporate variable LS size.

The experimental results demonstrated that the two variable LS size hybrids are the best, over 21 test programs. The two show very similar performance to one another in terms of number of paths found and number of generations. In the end, hybrid LS-GA-LSv110 outperforms other hybrid variants. It covered 0.77 paths more and required 3.39 generations less than GA. In general, a hybrid is recommended for a test program with at least five feasible paths with sparse distribution of number of paths found over generations.

The presence of infeasible paths does not hamper the test data generator in finding the feasible paths. On the contrary, it helps feasible paths to be found earlier, i.e. in fewer generations, by creating higher selection pressure in the generation of a new population. It means that taking out infeasible paths from target paths, if there are any, will require more generations to cover the same number of paths. It is difficult to analyze paths to decide which are infeasible; so it is good to note that there is no advantage — in fact, a disadvantage — to do so.

8.3 Future Research

There are many areas for improvement in the area of path testing and GA based path testing specifically. The following are major future research directions.

Filling up the sparse structure and expression classifications tables is one future work. The more complete the table, the more test programs can be used for path testing benchmarks. Moreover, it will allow more comprehensive understanding on how path testing works and what kind of test programs are difficult to cover.

It is concluded that having infeasible paths in the target paths can be helpful in finding more target paths in the earlier generations. Plausibly, more feasible paths are covered by chance while GA is searching for test data for the infeasible ones. Having this logic in mind, might the insertion of infeasible paths assist to cover more paths in fewer generations?

Manual instrumentation and fitness function generation for a test pro-

gram are relatively not difficult. However, these two tasks become more and more complicated and error prone as the program gets bigger in terms of number of selections, number of loops, and selection statements complexities. Automating the tasks will make path testing quicker, less difficult, and less error prone.

This research has approached the analysis of multivariate data sets by considering only one value of parameter or measure/dependent variable at a time. This could miss interaction effects that involve multiple values/factors. Thus, performing extra statistical analysis involving multiple parameters is further work.

This research has investigated only one of many statistical approaches for predicting logarithmically decaying phenomena, which could be used to identify stopping criteria for searching. Investigating other methods can be a topic for further work.

Appendix A

Test Programs Classification

A.1 Structure Classification

Table A.1 presents program structure classification. The acronyms used in the table are **IP** for Iteration Point (the number of loops), **S** for Selection (the number of simple selection statements), **Conf.** for configuration **IP** and **S**, i.e. serial or parallel or nested, **PS** for Point of Selection (the order of occurrences of something representing selections {could be IF or UNLESS}, e.g. **PS1** for IF statement in the first order), **CCL** for Cyclomatic Complexity Number, and **Class** for the Class Name, **I** for IF statement, and **P** for loop statement, e.g. WHILE, DO, and FOR.

Table A.1: Program Structure Classification

No	IP	S	Conf.	Point-Selections (PS)				CCL	Class
				PS1	PS2	PS3	PS4		
1	0	1		I (IF)				2	S01I

Continued on next page

Table A.1 – continued from previous page

No	IPs	Ss	Conf.	Point-Selections (PS)				CCL	Class
				PS1	PS2	PS3	PS4		
2		2	Serial (S)	I	I			3	S02SII
3			Nested (N)	I	I			3	S02NII
4		3	S-S	I	I	I		4	S03SSIII
5			S-N	I	I	I		4	S03SNIII
6			N-S	I	I	I		4	S03NSIII
7			N	I	I	I		4	S03NIII
8		≥ 4						5	S04
9	1	0		P(oint)				2	S10P
10		1	S	P	I			3	S11SPI
11			N	P	I			3	S11NPI
12				I	P			3	S11NIP
13		2	S-S	P	I	I		4	S12SSPII
14				I	P	I		4	S12SSIPI
15				I	I	P		4	S12SSIIP
16			S-N	P	I	I		4	S12SNPII
17				I	P	I		4	S12SNIPI
18				I	I	P		4	S12SNIIP
19			N-S	P	I	I		4	S12NSPII
20				I	P	I		4	S12NSIPI
21				I	I	P		4	S12NSIIP
22			N	P	I	I		4	S12NPII
23				I	P	I		4	S12NIPI
24				I	I	P		4	S12NIIP
25		≥ 3						5	S13
26	2	0	S	P	P			3	S20SPP
27			N	P	P			3	S20NPP
28		1	S-S	P	P	I		4	S21SSPPI
29				P	I	P		4	S21SSPIP
30				I	P	P		4	S21SSIPP
31			S-N	P	P	I		4	S21SNPPI

Continued on next page

Table A.1 – continued from previous page

No	IPs	Ss	Conf.	Point-Selections (PS)				CCL	Class
				PS1	PS2	PS3	PS4		
32				P	I	P		4	S21SNPIP
33				I	P	P		4	S21SNIPP
34			N-S	P	P	I		4	S21NSPPI
35				P	I	P		4	S21NSPIP
36				I	P	P		4	S21NSIPP
37			N	P	P	I		4	S21NPPI
38				P	I	P		4	S21NPIP
39				I	P	P		4	S21NIPP
40		2	S-S-S	P	P	I	I	5	SPPII
41				P	I	P	I	5	SPIPI
42				P	I	I	P	5	SPIIP
43				I	P	I	P	5	SIPIP
44				I	I	P	P	5	SIIPP
45				I	P	P	I	5	SIPPI
46			S-S-N	P	P	I	I	5	SPPII
47				P	I	P	I	5	SPIPI
48				P	I	I	P	5	SPIIP
49				I	P	I	P	5	SIPIP
50				I	I	P	P	5	SIIPP
51				I	P	P	I	5	SIPPI
52			S-N-S	P	P	I	I	5	SPPII
53				P	I	P	I	5	SPIPI
54				P	I	I	P	5	SPIIP
55				I	P	I	P	5	SIPIP
56				I	I	P	P	5	SIIPP
57				I	P	P	I	5	SIPPI
58			N-S-S	P	P	I	I	5	SPPII
59				P	I	P	I	5	SPIPI
60				P	I	I	P	5	SPIIP
61				I	P	I	P	5	SIPIP

Continued on next page

Table A.1 – continued from previous page

No	IPs	Ss	Conf.	Point-Selections (PS)				CCL	Class
				PS1	PS2	PS3	PS4		
62				I	I	P	P	5	SIIPP
63				I	P	P	I	5	SIPPI
64			S-N	P	P	I	I	5	SPPII
65				P	I	P	I	5	SPIPI
66				P	I	I	P	5	SPIIP
67				I	P	I	P	5	SIPIP
68				I	I	P	P	5	SIIPP
69				I	P	P	I	5	SIPPI
70			N-S	P	P	I	I	5	SPPII
71				P	I	P	I	5	SPIPI
72				P	I	I	P	5	SPIIP
73				I	P	I	P	5	SIPIP
74				I	I	P	P	5	SIIPP
75				I	P	P	I	5	SIPPI
76			N(S-S)	P	P	I	I	5	SPPII
77				P	I	P	I	5	SPIPI
78				P	I	I	P	5	SPIIP
79				I	P	I	P	5	SIPIP
80				I	I	P	P	5	SIIPP
81				I	P	P	I	5	SIPPI
82			N(S-N)	P	P	I	I	5	SPPII
83				P	I	P	I	5	SPIPI
84				P	I	I	P	5	SPIIP
85				I	P	I	P	5	SIPIP
86				I	I	P	P	5	SIIPP
87				I	P	P	I	5	SIPPI
88			N(N-S)	P	P	I	I	5	SPPII
89				P	I	P	I	5	SPIPI
90				P	I	I	P	5	SPIIP
91				I	P	I	P	5	SIPIP

Continued on next page

Table A.1 – continued from previous page

No	IPs	Ss	Conf.	Point-Selections (PS)				CCL	Class
				PS1	PS2	PS3	PS4		
92				I	I	P	P	5	SIIPP
93				I	P	P	I	5	SIPPI
94			N(N)	P	P	I	I	5	SPPII
95				P	I	P	I	5	SPIPI
96				P	I	I	P	5	SPIIP
97				I	P	I	P	5	SIPIP
98				I	I	P	P	5	SIIPP
99				I	P	P	I	5	SIPPI
100		≥ 3						6	S23
101	≥ 3							4	S3

Table A.2 classifies test programs based on their structure.

Table A.2: Structure Classification of Test Programs

Class	Test Program (TP)
S01I	quicksort
S02SII	
S02NII	
S03SSIII	
S03SNIII	
S03NSIII	
S03NIII	tA2008, triangle michael, triangle myers
S04	tM2004, ttB2002, fcB2002, fG2011, sG2011, triangle wegener, triangle sthamer

Continued on next page

Table A.2 – continued from previous page

Class	Test Program (TP)
S10P	
S11SPI	
S11NPI	binA2008, linear search
S11NIP	
S12SSPII	
S12SSIPI	eiR1985
S12SSIIP	
S12SNPII	
S12SNIPI	
S12SNIIP	
S12NSPII	mmA2008, bisA2008
S12NSIPI	
S12NSIIP	
S12NPII	
S12NIPI	gA2008
S12NIIP	rA2008, remainder sthamer
S13	mtA2008, scB2002
S20SPP	
S20NPP	
S21SSPPI	
S21SSPIP	
S21SSIPP	
S21SNPPI	qG1997
S21SNPIP	
S21SNIPP	
S21NSPPI	
S21NSPIP	
S21NSIPP	
S21NPPI	iA2008, bubA2008, bG2011
S21NPIP	
S21NIPP	

Continued on next page

Table A.2 – continued from previous page

Class	Test Program (TP)
SPPII	
SPIPI	
SPIIP	
SIPIP	
SIIPP	
SIPPI	
SPPII	
SPIPI	
SPIIP	
SIPIP	
SIIPP	
SIPPI	
SPPII	
SPIPI	
SPIIP	
SIPIP	
SIIPP	
SIPPI	
SPPII	
SPIPI	
SPIIP	
SIPIP	
SIIPP	
SIPPI	
SPPII	
SPIPI	
SPIIP	
SIPIP	
SIIPP	
SIPPI	
SPPII	

Continued on next page

Table A.2 – continued from previous page

Class	Test Program (TP)
SPIPI	
SPIIP	
SIPIP	
SIIPP	
SIPPI	
SPPII	
SPIPI	
SPIIP	
SIPIP	
SIIPP	
SIPPI	
SPPII	
SPIPI	
SPIIP	
SIPIP	
SIIPP	
SIPPI	
SPPII	
SPIPI	
SPIIP	
SIPIP	
SIIPP	
SIPPI	
SPPII	
SPIPI	
SPIIP	
SIPIP	
SIIPP	
SIPPI	
S23	qB2002
S3	eiB2002, fB2002, shellsort

A.2 Expression Classification

Table A.3 classifies test programs based on their expression. The acronyms used in the table are **NoE** for Number of Simple Expressions, **Exp1** for expression number 1, and **CCL** for Cyclomatic Complexity Number.

Table A.4 shows all the operators that could involve in a statement expression.

Table A.5 is the expression classification of test programs.

Table A.3: Program Expression Classification

No	**NoE**	**Connector**	**Expressions**			**CCL**	**Class**
			Exp1	**Exp2**	**Exp3**		
1	1		Relational			1	E1R
2			Arith-Rel			1	E1I
3	2	AND (N)	Relational	Relational		2	E2NRR
4				Arith-Rel (I)		2	E2NRI
5			Arith-Rel	Arith-Rel		2	E2NII
6		OR (O)	Relational	Relational		2	E2ORR
7			Relational	Arith-Rel		2	E2ORI
8			Arith-Rel	Arith-Rel		2	E2OII
9	3	AND-AND	Relational	Relational	Relational	3	E3NNRRR
10				Relational	Arith-Rel	3	E3NNRRI
11				Arith-Rel	Arith-Rel	3	E3NNRII
12			Arith-Rel	Arith-Rel	Arith-Rel	3	E3NNIII
13		OR-OR	Relational	Relational	Relational	3	E3OORRR
14			Relational	Relational	Arith-Rel	3	E3OORRI
15			Relational	Arith-Rel	Arith-Rel	3	E3OORII
16			Arith-Rel	Arith-Rel	Arith-Rel	3	E3OOIII
17		AND-OR	Relational	Relational	Relational	3	E3AORRR
18				Relational	Arith-Rel	3	E3AORRI
19				Arith-Rel	Relational	3	E3AORIR
20				Arith-Rel	Arith-Rel	3	E3AORII
21			Arith-Rel	Relational	Relational	3	E3AOIRR
22					Arith-Rel	3	E3AOIRI
23				Arith-Rel	Relational	3	E3AOIIR
24					Arith-Rel	3	E3AOIII

Table A.4: Types of Operators

Type	**Operators**	**Sign**
Arithmetic	Addition	+
	Substraction	-
	Multiplication	*
	Division	/
Relational	Equal To	==
	Not Equal To	!=
	Less Than	<
	Greater Than	>
	Less Than or Equal To	<=
	Greater Than or Equal To	>=
Logical	And	&&
	Or	||
	Not	~

Table A.5: Expression Classification of Test Programs

Class	**Test Program (TP)**
E1R	mmA2008, iA2008, bisA2008, binA2008, bubA2008, gcd, rA2008, mtA2008, tM2004, eR1985, qG1997, ttB2002, eiR1985, eiB2002, qB2002, scB2002, fcB2002, fB2002, bG2011, fG2011, sG2011, linear search, quicksort, shellsort, triangle myers, triangle michael, triangle sthamer, triangle wegener
E1I	bisA2008, tM2004, ttB2002, eiR1985, eiB2002, scB2002, triangle sthamer, triangle wegener
E2NRR	iA2008, ttB2002, qG1997, qB2002, shellsort, triangle wegener
E2NRI	bubA2008
E2NII	fcB2002
E2ORR	ttB2002, fB2002, triangle wegener
E2ORI	
E2OII	
E3NNRRR	tA2008, mtA2008, triangle myers, triangle michael
E3NNRRI	
E3NNRII	
E3NNIII	tA2008, mtA2008, triangle myers, triangle michael
E3OORRR	
E3OORRI	
E3OORII	
E3OOIII	
E3AORRR	
E3AORRI	
E3AORIR	
E3AORII	
E3AOIRR	bisA2008
E3AOIRI	
E3AOIIR	
E3AOIII	

Appendix B

Test Programs

Description of each test problem consists of instrumented source code, CFG, target paths, and fitness function, respectively. The original source code is not included, because it can easily be extracted from the instrumented version by removing all the probes.

B.1 Test Program iA2008

The source code of iA2008

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

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

The CFG of iA2008

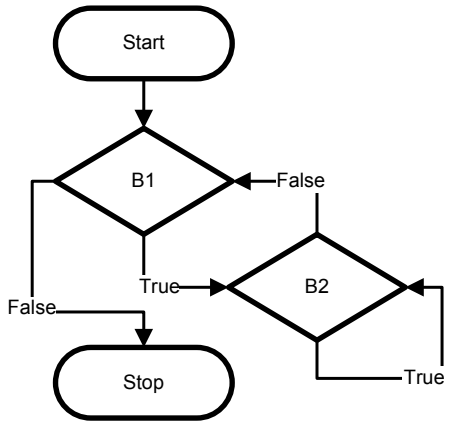


Figure B.1: CFG of iA2008

The target paths of iA2008

```
[1 1];
[1 0 2 1 1 1];
[1 0 2 0 2 1 1 1];
[1 0 2 0 2 0 2 1 1 1]; % infeasible
[1 0 2 1 1 0 2 0 2 1 1 1];
[1 0 2 0 2 1 1 0 2 1 1 1]
```

The fitness function of iA2008

```
function branchVal = fitnessInsertion(branchNo, predicate)
k = 1; % the smallest step for integer
```

```
switch (branchNo)
  case 1,
      % Branch #1: (for i=2:n) ==> (i <= n)
      branchVal = predicate(1) - predicate(2);
  case 2,
      % Branch #2: ((j > 0) & (anyArray(j) > x))
      term(1) = 0 - predicate(1); % predicate(1) = j
      % predicate(2) = anyArray(j); predicate(3) = x
      term(2) = predicate(3) - predicate(2);
      for i=1:2
          if (term(i) < 0)
              term(i) = term(i) - k;
          else
              term(i) = term(i) + k;
          end
      end
      branchVal = term(1) + term(2);
    % Branch # 2
 if (~((predicate(1) > 0) & (predicate(2) > predicate(3)) ) & (branchVal < 0)),
          branchVal = -branchVal;
    end
end
```

B.2 Test Program bisA2008

The source code of bisA2008

```
function [traversedPath, roots] = bisection(input)
EPS_ABS = 1e-2; % constant
EPS_STEP = 1e-2; % constant
traversedPath = []; % traversedPath contains branch# and its corresponding branchVal
a = input(1);
b = input(2);
c = NaN;
if (f(a) * f(b)) >= 0,
    return;
end
% B1 instrument
traversedPath = [traversedPath 1 fitnessBisection(1, a, f(a), b, f(b), EPS_ABS, EPS_STEP)];
while (b-a >= EPS_STEP || (abs(f(a)) >= EPS_ABS && abs(f(b)) >= EPS_ABS))
    c = (a + b)/2;
    traversedPath = [traversedPath 2 fitnessBisection(2, f(c))]; % B2 instrument
    if (f(c) == 0)
        roots = c;
        return;
    else
        traversedPath = [traversedPath 3 fitnessBisection(3, f(a), f(c))]; % B3 instrument
```

```
        if (f(a)*f(c) < 0)
            b = c;
        else
            a = c;
        end
    end
    % B1 instrument
    traversedPath = [traversedPath 1 fitnessBisection(1, a, f(a), b, f(b), EPS_ABS, EPS_STEP)];
end
roots = c;
end

% find the root of the following function y = 3*x^2 + 10*x - 3
function y = f(x)
    % y = 3*x^2 + 10*x - 3;
    y = x^2 - 3; % root = 1.7344
end
```

The CFG of bisA2008

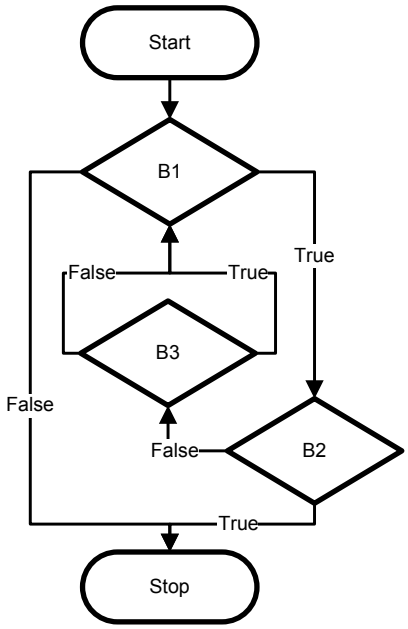


Figure B.2: CFG of bisA2008

The target paths of bisA2008

```
[1 1]; % Reach B1 then end (No loop @ B1)
[1 0 2 0]; % Reach B2 then end (No loop @ B2)
[1 0 2 1 3 0 1 1]; % One loop @ B1 or B2 with True @ B3
```

```
[1 0 2 1 3 1 1 1]; % One loop @ B1 or B2 with False @ B3
[1 0 2 1 3 0 1 0 2 0]; % Two loops @ B1 & one loop @ B2 with True @ B3
[1 0 2 1 3 1 1 0 2 0]; % Two loops @ B1 & one loop @ B2 with False @ B3
[1 0 2 1 3 0 1 0 2 1 3 0 1 1]; % Two loops @ B1 & B2 with 2 Trues @ B3
[1 0 2 1 3 0 1 0 2 1 3 1 1 1]; % Two loops @ B1 & B2 with True & False @ B3
[1 0 2 1 3 1 1 0 2 1 3 1 1 1]; % Two loops @ B1 & B2 with 2 Falses @ B3
```

The fitness function of bisA2008

```
function branchVal = fitnessBisection(branchNo, a, fa, b, fb, EPS_ABS, EPS_STEP)
k = 1; % the smallest step for integer
switch (branchNo)
  case 1,
      % branch #1: (b-a >= EPS_STEP || (abs(f(a)) >= EPS_ABS && abs(f(b)) >= EPS_ABS))
      term(1) = b-a >= EPS_STEP;
      term(2) = abs(fa) >= EPS_ABS;
      term(3) = abs(fb) >= EPS_ABS;
      if term(1),
          val(1) = 0;
      else
          val(1) = EPS_STEP - (b-a);
      end
      if term(2),
          val(2) = 0;
      else
          val(2) = EPS_ABS - abs(fa);
      end
      if term(3),
          val(3) = 0;
      else
          val(3) = EPS_ABS - abs(fb);
      end
      value = min(val(1), (val(2)+val(3)));
  case 2,
      % branch #2: f(c) == 0; note: f(c) = a
      if (a == 0),
          value = 0;
      else
          value = abs(a);
      end
  case 3,
      % branch #3: f(a)*f(c) < 0; note: f(a) = a; f(c) = fa
      if a * fa < 0,
          value = 0;
      else
          value = (a * fa) + k;
      end
end
branchVal = value;
```

B.3 Test Program binA2008

The source code of binA2008

```
function [traversedPath, itemIndex] = binary(itemNumbers)
item = itemNumbers(1);
numbers = itemNumbers(1,2:end);
lowerIdx = 1;
upperIdx = length(numbers);
traversedPath = [];
% instrument Branch # 1
traversedPath = [traversedPath 1 fitnessBinary(1, [lowerIdx upperIdx])];
while (lowerIdx ~= upperIdx), % Branch # 1
    temp = lowerIdx + upperIdx; % additional statement
    if (mod(temp, 2) ~= 0), temp = temp - 1; end % additional statement
    idx = temp / 2;
    % instrument Branch # 2
    traversedPath = [traversedPath 2 fitnessBinary(2, [numbers(idx) item])];
    if (numbers(idx) < item), % Branch # 2
        lowerIdx = idx + 1;
    else
        upperIdx = idx;
    end
    % instrument Branch # 1
    traversedPath = [traversedPath 1 fitnessBinary(1, [lowerIdx upperIdx])];
end
% Additional code that returns -1 if the item is not found
if (item == numbers(lowerIdx)),
    itemIndex = lowerIdx;
else
    itemIndex = -1;
end
end
```

The CFG of binA2008

The target paths of binA2008

```
[1 1];
[1 0 2 0 1 1];
[1 0 2 1 1 1];
[1 0 2 0 1 0 2 0 1 1];
[1 0 2 1 1 0 2 1 1 1];
[1 0 2 0 1 0 2 1 1 1];
[1 0 2 1 1 0 2 0 1 1];
```

The fitness function of binA2008

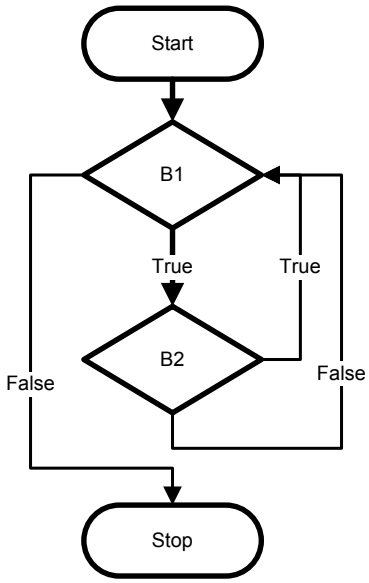


Figure B.3: CFG of binA2008

```
function branchVal = fitnessBinary(branchNo, predicate)
k = 1; % the smallest step for integer
switch (branchNo)
  case 1,
      % Branch #1: (lower ~= upper)
      if (predicate(1) ~= predicate(2)), branchVal = 0; else branchVal = k; end
  case 2,
      % Branch #2: (numbers(idx) < item)
      branchVal = predicate(1) - predicate(2);
      if (branchVal < 0),
          branchVal = branchVal - k;
      else
          branchVal = branchVal + k;
      end
end
```

B.4 Test Program bubA2008

The source code of bubA2008

```
function [traversedPath, sortedArray] = bubble(anyArray)
sorted = 0; % 0 means false
i = 1; n = length(anyArray);
traversedPath = [];
```

```
% instrument Branch # 1
traversedPath = [traversedPath 1 fitnessBubble(1, [i (n-1) ~sorted])];
while ((i <= (n-1)) && ~sorted), % Branch # 1
    sorted = 1;
    j = n;
    % instrument Branch # 2
    traversedPath = [traversedPath 2 fitnessBubble(2, [j (i+1)])];
    for j=n:-1:i+1 % Branch # 2
        % instrument Branch # 3
        traversedPath = [traversedPath 3 fitnessBubble(3, [anyArray(j) anyArray(j-1)])];
        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
        % instrument Branch # 2
        traversedPath = [traversedPath 2 fitnessBubble(2, [(j-1) (i+1)])];
    end
    i = i + 1;
    % instrument Branch # 1
    traversedPath = [traversedPath 1 fitnessBubble(1, [i (n-1) ~sorted])];
end
sortedArray = anyArray;
end
```

The CFG of bubA2008

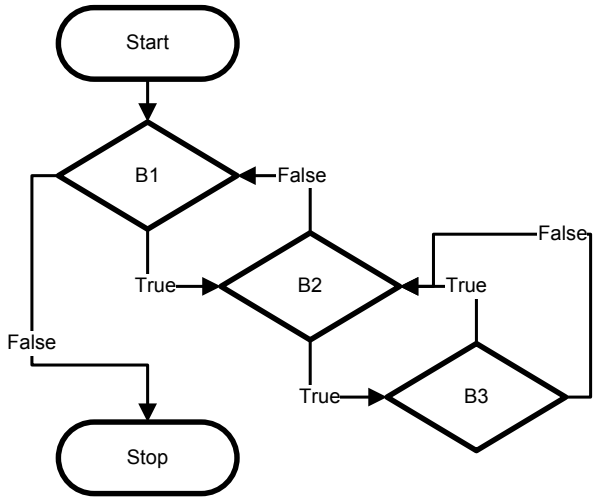


Figure B.4: CFG of bubA2008

The target paths of bubA2008

```
[1 1];
[1 0 2 1 1 1];
[1 0 2 1 1 0 2 1 1 1];
[1 0 2 0 3 0 2 1 1 1];
[1 0 2 0 3 1 2 1 1 1];
[1 0 2 0 3 0 2 0 3 0 2 1 1 1];
[1 0 2 0 3 1 2 0 3 1 2 1 1 1];
[1 0 2 0 3 0 2 0 3 1 2 1 1 1];
[1 0 2 0 3 1 2 0 3 0 2 1 1 1];
[1 0 2 0 3 0 2 0 3 1 2 0 3 1 2 1 1 1];
[1 0 2 0 3 1 2 0 3 1 2 0 3 0 2 1 1 1];
[1 0 2 0 3 0 2 0 3 0 2 0 3 1 2 1 1 1];
[1 0 2 0 3 1 2 0 3 1 2 0 3 0 2 1 1 1];
[1 0 2 0 3 0 2 0 3 1 2 0 3 0 2 1 1 1];
[1 0 2 0 3 1 2 0 3 0 2 0 3 1 2 1 1 1];
```

The fitness function of bubA2008

```
function branchVal = fitnessBubble(branchNo, predicate)
    k = 1; % the smallest step for integer
    switch (branchNo)
      case 1,
          % Branch #1: ((i <= (n-1)) && ~sorted)
          term(1) = predicate(1) - predicate(2);
          if predicate(3) > 0, term(2) = 0; else term(2) = k; end
          if (term(2) <= 0) && (term(1) <= 0)
              branchVal = 0;
          else
              branchVal = term(1) + term(2);
          end
      case 2,
          % Branch #2: j=n:-1:i+1 ==> (j >= i+1)
          branchVal = predicate(2) - predicate(1);
      case 3,
          % branch #3: (anyArray(j) < anyArray(j-1))
          branchVal = predicate(1) - predicate(2);
          if (branchVal < 0)
              branchVal = branchVal - k;
          else
              branchVal = branchVal + k;
          end
    end
end
```

B.5 Test Program gA2008

The source code of gA2008

```
function [traversedPath y] = gcd(number)
    a = number(1);
    b = number(2);
    % traversedPath contains branch# and its corresponding branchVal
    traversedPath = [];
    traversedPath = [traversedPath 1 fitnessGCD(1, a)]; % B1 Instrument
    if (a == 0),
        y = b;
    else
        traversedPath = [traversedPath 2 fitnessGCD(2, b)]; % B2 Instrument
        while b ~= 0
            traversedPath = [traversedPath 3 fitnessGCD(3, a, b)]; % B3 Instrument
            if a > b
                a = a - b;
            else
                b = b - a;
            end
            traversedPath = [traversedPath 2 fitnessGCD(2, b)]; % B2 Instrument
        end
        y = a;
    end
end
```

The CFG of gA2008

The target paths of gA2008

```
[1 0] % Reach B1 then end
[1 1 2 1]; % No loop @ B2
[1 1 2 0 3 0 2 1]; % IF: One loop @ B2 with True @ B3
[1 1 2 0 3 1 2 1]; % One loop @ B2 with False @ B3
[1 1 2 0 3 0 2 0 3 0 2 1]; % IF: Two loops @ B2 with T & T @ B3
[1 1 2 0 3 0 2 0 3 1 2 1]; % Two loops @ B2 with T & F @ B3
[1 1 2 0 3 1 2 0 3 0 2 1]; % IF: Two loops @ B2 with F & T @ B3
[1 1 2 0 3 1 2 0 3 1 2 1]; % Two loops @ B2 with F & F @ B3
```

The fitness function of gA2008

```
function branchVal = fitnessGCD(branchNo, a, b)
    k = 1; % the smallest step for integer
    switch (branchNo)
      case 1,
```

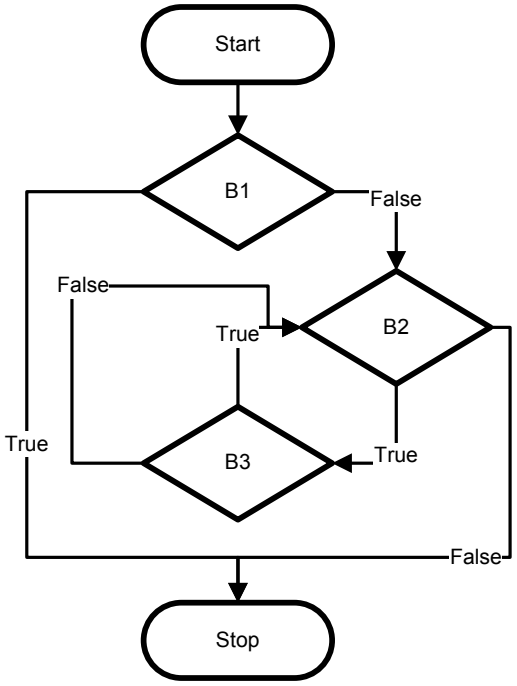


Figure B.5: CFG of gA2008

```
        % branch #1: (a == 0) => F: abs(a)
        if (a == 0),
            value = 0;
        else
            value = abs(a);
        end
    case 2,
        % branch #2: b ~= 0
        if (a ~= 0),
            value = 0;
        else
            value = k;
        end
    case 3,
        % branch #3: a > b
        if a > b,
            value = 0;
        else
            value = (b - a) + k;
        end
  end
  branchVal = value;
end
```

B.6 Test Program rA2008

The source code of rA2008

```
function [traversedPath, y] = remainder(input)
    traversedPath = [];
    a = input(1);
    d = input(2);
    traversedPath = [traversedPath 1 fitnessRemainder(1, d)]; % B1 instrument
    if d == 0 % divisor can not be zero
        y = NaN;
    else
        traversedPath = [traversedPath 2 fitnessRemainder(2, a, d)]; % B2 instrument
        if a < d
            y = a;
        else
            traversedPath = [traversedPath 3 fitnessRemainder(3, a, d)]; % B3 instrument
            while a >= d
                a = a - d;
                traversedPath = [traversedPath 3 fitnessRemainder(3, a, d)]; % B3 instrument
            end
            y = a;
        end
    end
end
```

The CFG of rA2008

The target paths of rA2008

```
[1 0] % Reach B1 then end
[1 1 2 0]; % Reach B2 then end
[1 1 2 1 3 1]; % IF: No loop @ B3
[1 1 2 1 3 0 3 1]; % One loop @ B3
[1 1 2 1 3 0 3 0 3 1]; % Two loops @ B3
```

The fitness function of rA2008

```
function branchVal = fitnessRemainder(branchNo, a, d)
    k = 1; % the smallest step for integer
    switch (branchNo)
      case 1,
          % branch #1: d == 0 => F: abs(d)
          if (a == 0),
              value = 0;
          else
```

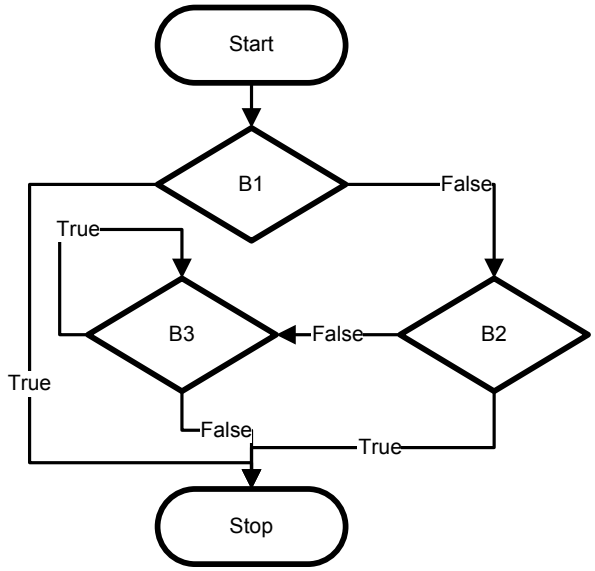


Figure B.6: CFG of rA2008

```
            value = abs(a);
        end
    case 2,
        % branch #2: a < d
        if (a < d),
            value = 0;
        else
            value = (a - d) + k;
        end
    case 3,
        % branch #3: a >= d
        if a >= d,
            value = 0;
        else
            value = (d - a);
        end
  end
  branchVal = value;
end
```

B.7 Test Program mtA2008

The source code of mtA2008

```
function [traversedPath, minimaxi, type] = mmTriangle(num)
    numLength = length(num);
    mini = num(1);
    maxi = num(1);
    idx = 2;
    % traversedPath contains branch# and its corresponding branchVal
    traversedPath = [];
    traversedPath = [traversedPath 1 fitnessMiniMaxi(1, [idx numLength])];
    while (idx <= numLength) % Branching #1
        traversedPath = [traversedPath 2 fitnessMiniMaxi(2, [maxi num(idx)])];
        if maxi < num(idx) % Branching #2
            maxi = num(idx);
        end
        traversedPath = [traversedPath 3 fitnessMiniMaxi(3, [mini num(idx)])];
        if mini > num(idx) % Branching #3
            mini = num(idx);
        end
        idx = idx+1;
        traversedPath = [traversedPath 1 fitnessMiniMaxi(1, [idx numLength])];
    end
    minimaxi = [mini maxi];
    A = num(1); % First side
    B = num(2); % Second side
    C = num(3); % Third side
    % instrument Branch # 4
    traversedPath = [traversedPath 4 fitnessTriangle(1, A, B, C)];
    if ((A+B > C) & (B+C > A) & (C+A > B)) % Branch # 4
        % instrument Branch # 5
        traversedPath = [traversedPath 5 fitnessTriangle(2, A, B, C)];
        if ((A ~= B) & (B ~= C) & (C ~= A)) % Branch # 5
            type = 'Scalene';
        else
            % instrument Branch # 6
            traversedPath = [traversedPath 6 fitnessTriangle(3, A, B, C)];
            % Branch # 6
            if (((A == B) & (B ~= C)) | ((B == C) & (C ~= A)) | ((C == A) & (A ~= B)))
                type = 'Isosceles';
            else
                type = 'Equilateral';
            end
        end
    else
        type = 'Not a triangle';
    end
end
```

The CFG of mtA2008

The target paths of mtA2008

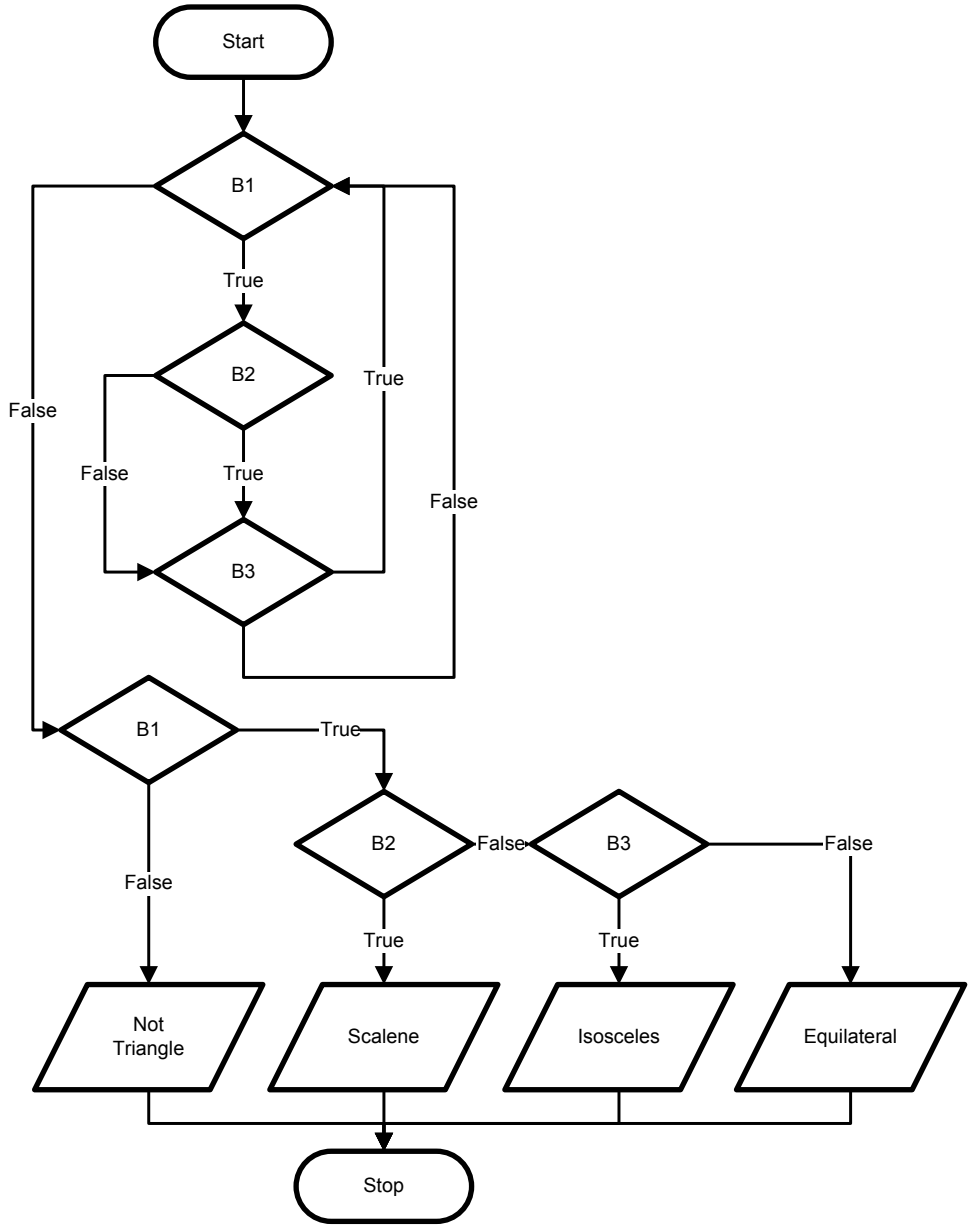


Figure B.7: CFG of mtA2008

```
% First combination: Tail => Equilateral
[1 1 4 0 5 1 6 1];
[1 0 2 0 3 1 1 1 4 0 5 1 6 1];
[1 0 2 1 3 0 1 1 4 0 5 1 6 1];
[1 0 2 1 3 1 1 1 4 0 5 1 6 1];
[1 0 2 0 3 1 1 0 2 0 3 1 1 1 4 0 5 1 6 1];
[1 0 2 0 3 1 1 0 2 1 3 0 1 1 4 0 5 1 6 1];
[1 0 2 0 3 1 1 0 2 1 3 1 1 1 4 0 5 1 6 1];
[1 0 2 1 3 0 1 0 2 0 3 1 1 1 4 0 5 1 6 1];
```

```
[1 0 2 1 3 0 1 0 2 1 3 0 1 1 4 0 5 1 6 1];
[1 0 2 1 3 0 1 0 2 1 3 1 1 1 4 0 5 1 6 1];
[1 0 2 1 3 1 1 0 2 0 3 1 1 1 4 0 5 1 6 1];
[1 0 2 1 3 1 1 0 2 1 3 0 1 1 4 0 5 1 6 1];
[1 0 2 1 3 1 1 0 2 1 3 1 1 1 4 0 5 1 6 1];

% Second combination: Tail => Scalene
[1 1 4 0 5 0];
[1 0 2 0 3 1 1 1 4 0 5 0];
[1 0 2 1 3 0 1 1 4 0 5 0];
[1 0 2 1 3 1 1 1 4 0 5 0];
[1 0 2 0 3 1 1 0 2 0 3 1 1 1 4 0 5 0];
[1 0 2 0 3 1 1 0 2 1 3 0 1 1 4 0 5 0];
[1 0 2 0 3 1 1 0 2 1 3 1 1 1 4 0 5 0];
[1 0 2 1 3 0 1 0 2 0 3 1 1 1 4 0 5 0];
[1 0 2 1 3 0 1 0 2 1 3 0 1 1 4 0 5 0];
[1 0 2 1 3 0 1 0 2 1 3 1 1 1 4 0 5 0];
[1 0 2 1 3 1 1 0 2 0 3 1 1 1 4 0 5 0];
[1 0 2 1 3 1 1 0 2 1 3 0 1 1 4 0 5 0];
[1 0 2 1 3 1 1 0 2 1 3 1 1 1 4 0 5 0];

% Third combination: Tail => Not Triangle
[1 1 4 1];
[1 0 2 0 3 1 1 1 4 1];
[1 0 2 1 3 0 1 1 4 1];
[1 0 2 1 3 1 1 1 4 1];
[1 0 2 0 3 1 1 0 2 0 3 1 1 1 4 1];
[1 0 2 0 3 1 1 0 2 1 3 0 1 1 4 1];
[1 0 2 0 3 1 1 0 2 1 3 1 1 1 4 1];
[1 0 2 1 3 0 1 0 2 0 3 1 1 1 4 1];
[1 0 2 1 3 0 1 0 2 1 3 0 1 1 4 1];
[1 0 2 1 3 0 1 0 2 1 3 1 1 1 4 1];
[1 0 2 1 3 1 1 0 2 0 3 1 1 1 4 1];
[1 0 2 1 3 1 1 0 2 1 3 0 1 1 4 1];
[1 0 2 1 3 1 1 0 2 1 3 1 1 1 4 1];

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

The fitness function of mtA2008 consists of tA2008 and mmA2008 fitness functions.

B.8 Test Program tM2004

The source code of tM2004

```
function [traversedPath type] = triangleMansour2004(sideLengths)
    traversedPath = [];
    A = sideLengths(1); % First side
    B = sideLengths(2); % Second side
    C = sideLengths(3); % Third side
    type = 'Scalene';
    % instrument Branch # 1
    traversedPath = [traversedPath 1 fitnessTriangleMansour2004(1, A, B)];
    if (A == B)
        % instrument Branch # 2
        traversedPath = [traversedPath 2 fitnessTriangleMansour2004(2, B, C)];
        if (B == C)
            type = 'Equilateral';
        else
            type = 'Isosceles';
        end
    else
        % instrument Branch # 3
        traversedPath = [traversedPath 3 fitnessTriangleMansour2004(3, B, C)];
        if (B == C)
            type = 'Isosceles';
        end
    end
    % instrument Branch # 4
    traversedPath = [traversedPath 4 fitnessTriangleMansour2004(4, A, B, C)];
    if (A^2 == (B^2 + C^2))
        type = 'Right';
    end
end
```

The CFG of tM2004

The target paths of tM2004

```
[1 0 2 1 4 1]; % SP1-p1 @mansour2004; isosceles with A=B
[1 1 3 0 4 1]; % SP1-p2 @mansour2004; isosceles with B=C
[1 0 2 0 4 1]; % SP1-p3 @mansour2004; equilateral
```

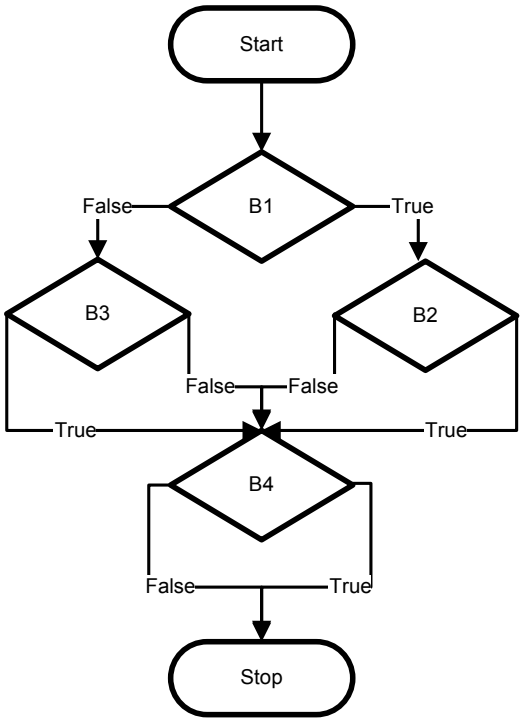


Figure B.8: CFG of tM2004

```
[1 1 3 1 4 0]; % SP1-p4 @mansour2004; right-angle
[1 1 3 0 4 0]; % IF: SP1-p5 @mansour2004; isosceles + right-angle
[1 1 3 1 4 1]; % SP1-p6 @mansour2004; scalene
[1 0 2 0 4 0]; % added by the author: equilateral + right-angle: inf
[1 0 2 1 4 0]  % added by the author: isosceles + right-angle
```

The fitness function of tM2004

```
function branchVal = fitnessTriangleMansour2004(branchNo, A, B, C)
    switch (branchNo)
      case 1,
          % branch #1: (A == B)
          branchVal = abs(A-B);

      case {2, 3},
          % branch #2: (B == C)
          branchVal = abs(A-B);

      case 4,
          % branch #4: (A^2 == (B^2 + C^2))
          branchVal = abs(A^2 - (B^2 + C^2));
    end
end
```

B.9 Test Program eR1985

The source code of eR1985

```
function [traversedPath Z] = expintRapps1985(integers)
    traversedPath = [];
    X = integers(1);
    Y = integers(2);
    % instrument Branch #1
    traversedPath = [traversedPath 1 fitnessExpintRapps1985(1, Y)];
    if (Y >= 0)
        power = Y;
    else
        power = -Y;
    end
    Z = 1;
    % instrument Branch #2
    traversedPath = [traversedPath 2 fitnessExpintRapps1985(2, power)];
    while (power ~= 0)
        Z = Z * X;
        power = power - 1;
        traversedPath = [traversedPath 2 fitnessExpintRapps1985(2, power)];
    end
    % instrument Branch #3
    traversedPath = [traversedPath 3 fitnessExpintRapps1985(3, Y)];
    if (Y < 0)
        Z = 1 / Z; % this is the original one; by removing if TRUE
    end
    Z = Z + 1;
end
```

The CFG of eR1985

The target paths of eR1985

```
[1 0 2 1 3 0];
[1 0 2 1 3 1];
[1 1 2 1 3 0];
[1 1 2 1 3 1];

[1 0 2 0 2 1 3 0];
[1 0 2 0 2 1 3 1];
[1 1 2 0 2 1 3 0];
[1 1 2 0 2 1 3 1];

[1 0 2 0 2 0 2 1 3 0];
[1 0 2 0 2 0 2 1 3 1];
[1 1 2 0 2 0 2 1 3 0];
```

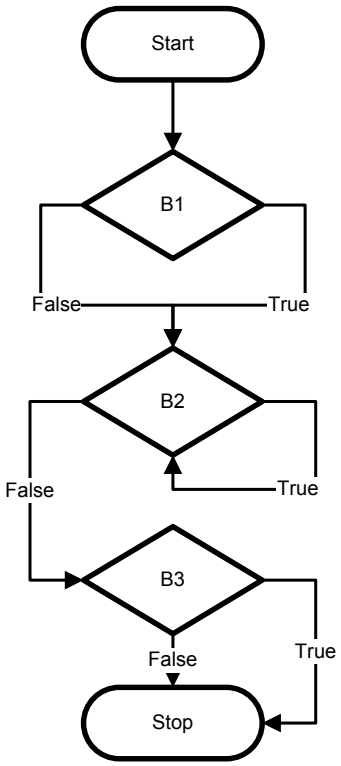


Figure B.9: CFG of eR1985

```
[1 1 2 0 2 0 2 1 3 1];
```

The fitness function of eR1985

```
function branchVal = fitnessExpintRapps1985(branchNo, A)
    k = 1;
    switch (branchNo)
      case 1,
          % branch #1: (A >= 0)
          branchVal = -A;
      case 2,
          % branch #2: (A ~= 0)
          if (A ~= 0)
              branchVal = 0;
          else
              branchVal = k;
          end
      case 3,
          % branch #3: (A < 0)
          branchVal = A + 1;
    end
end
```

B.10 Test Program qG1997

The source code of qG1997

```
function [traversedPath q r] = quotientGallagher1997(integers)
    traversedPath = [];
    q = 0; % q: quotient
    r = integers(1); % r: remainder; integers(1): nominator
    t = integers(2); % integers(2): denominator
    % instrument Branch # 1
    traversedPath = [traversedPath 1 fitnessQuotientGallagher1997(1, r, t)];
    while (r >= t)
        t = t * 2;
        traversedPath = [traversedPath 1 fitnessQuotientGallagher1997(1, r, t)];
    end
    % instrument Branch # 2
    traversedPath = [traversedPath 2 fitnessQuotientGallagher1997(2, t, integers(2))];
    while (t ~= integers(2))
        q = q * 2;
        t = t / 2;
        % instrument Branch # 3
        traversedPath = [traversedPath 3 fitnessQuotientGallagher1997(3, t, r)];
        if (t <= r)
            r = r - t;
            q = q + 1;
        end
        traversedPath = [traversedPath 2 fitnessQuotientGallagher1997(2, t, integers(2))];
    end
end
```

The CFG of qG1997

The target paths of qG1997

```
% no loop @ B#1 & B#2
[1 1 2 1];

% no loop @ B#1 & one loop @ B#2 with TRUE @ B#3
[1 1 2 0 3 0 2 1];
% no loop @ B#1 & one loop @ B#2 with FALSE @ B#3
[1 1 2 0 3 1 2 1];
% no loop @ B#1 & two loops @ B#2 with two TRUEs @ B#3
[1 1 2 0 3 0 2 0 3 0 2 1];
```

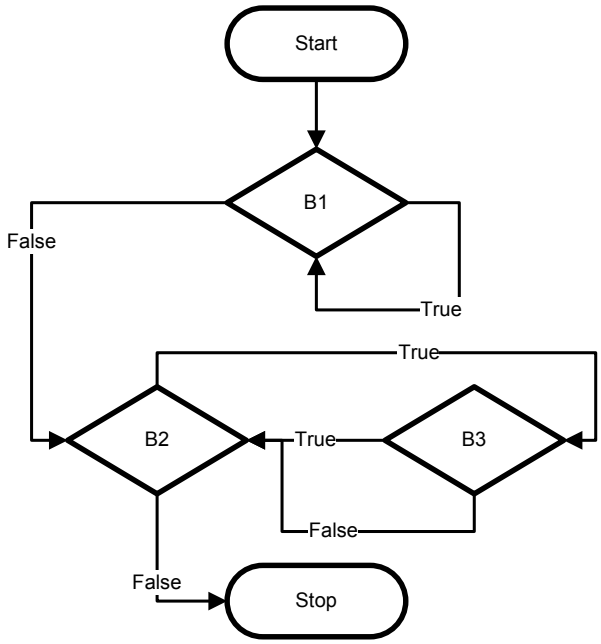


Figure B.10: CFG of qG1997

```
% no loop @ B#1 & two loops @ B#2 with TRUE then FALSE @ B#3
[1 1 2 0 3 0 2 0 3 1 2 1];
% no loop @ B#1 & two loops @ B#2 with two FALSEs @ B#3
[1 1 2 0 3 1 2 0 3 1 2 1];
% no loop @ B#1 & two loops @ B#2 with FALSE then TRUE @ B#3
[1 1 2 0 3 1 2 0 3 0 2 1];

% one loop @ B#1 and no loop @ B#2
[1 0 1 1 2 1];
% one loop @ B#1 & one loop @ B#2 with TRUE @ B#3
[1 0 1 1 2 0 3 0 2 1];
% one loop @ B#1 & one loop @ B#2 with FALSE @ B#3
[1 0 1 1 2 0 3 1 2 1];
% one loop @ B#1 & two loops @ B#2 with two TRUEs @ B#3
[1 0 1 1 2 0 3 0 2 0 3 0 2 1];
% one loop @ B#1 & two loops @ B#2 with TRUE then FALSE @ B#3
[1 0 1 1 2 0 3 0 2 0 3 1 2 1];
% one loop @ B#1 & two loops @ B#2 with two FALSEs @ B#3
[1 0 1 1 2 0 3 1 2 0 3 1 2 1];
% one loop @ B#1 & two loops @ B#2 with FALSE then TRUE@ B#3
[1 0 1 1 2 0 3 1 2 0 3 0 2 1];

% two loops @ B#1 and no loop @ B#2
[1 0 1 0 1 1 2 1];
% two loops @ B#1 & one loop @ B#2 with TRUE @ B#3
[1 0 1 0 1 1 2 0 3 0 2 1];
% two loops @ B#1 & one loop @ B#2 with FALSE @ B#3
```

```
[1 0 1 0 1 1 2 0 3 1 2 1];
% two loops @ B#1 & two loops @ B#2 with two TRUEs @ B#3
[1 0 1 0 1 1 2 0 3 0 2 0 3 0 2 1];
% two loops @ B#1 & two loops @ B#2 with TRUE then FALSE @ B#3
[1 0 1 0 1 1 2 0 3 0 2 0 3 1 2 1];
% two loops @ B#1 & two loops @ B#2 with two FALSEs @ B#3
[1 0 1 0 1 1 2 0 3 1 2 0 3 1 2 1];
% two loops @ B#1 & two loops @ B#2 with FALSE then TRUE @ B#3
[1 0 1 0 1 1 2 0 3 1 2 02 1 3 0 2 1];
```

The fitness function of qG1997

```
function branchVal = fitnessQuotientGallagher1997(branchNo, A, B)
    k = 1; % the smallest step for integer
    switch (branchNo)
      case 1,
          % branch #1: (r >= t); r=A; t=B
          branchVal = B - A;
      case 2,
          % branch #2: (t ~= integers(2)); t=A; integers(2)=B
          if (abs(A - B) == 0)
              branchVal = k;
          else
              branchVal = 0;
          end
      case 3,
          % branch #3: (t <= r)
          branchVal = A - B;
    end
end
```

B.11 Test Program tB2002

The source code of tB2002

```
function [path, type, area] = tritypeBueno2002(side)
    a = side(1);
    b = side(2);
    c = side(3);
    path = [];
    % Instrument Branch #1
    path = [path 1 fitnessTritype(1, a, b, c)];
    if ((a < b) || (b < c))
       type = 'Invalid input. Input must be ordered a >= b >= c';
```

```
            area =  0;
        % Instrument Branch #2
        path = [path 2 fitnessTritype(2, a, b, c)];
        elseif ( a >= (b + c) )
            type = 'Not a triangle';
            area = 0;
        % Instrument Branch #3
        path = [path 3 fitnessTritype(3, a, b, c)];
        elseif ( (a ~= b) && (b ~= c) ) % /* escaleno */
            as = a*a;
            bs = b*b;
            cs = c*c;
            % Instrument Branch #4
            path = [path 4 fitnessTritype(4, as, bs, cs)];
            if (as == bs + cs) % /* retangulo */
                type = 'Rectangle';
                area = b * c / 2.0;
            else
                s = (a+b+c) / 2.0;
                area = sqrt(s*(s-a)*(s-b)*(s-c));
                % Instrument Branch #5
                path = [path 5 fitnessTritype(5, as, bs, cs)];
                if ( as < bs + cs )
                    type = 'Agudo'; % /* agudo */
                else
                    type ='Obtuso'; % /* obtuso */
                end
            end
        % Instrument Branch #6
        path = [path 6 fitnessTritype(6, a, b, c)];
        elseif ( (a == b) && (b == c) )
            type = 'Equilateral'; % /* equilatero */
            area = a*a*sqrt(3.0)/4.0;
        else
            type = 'Isosceles'; % /* isoceles */
            % Instrument Branch #7
            path = [path 7 fitnessTritype(7, a, b)];
            if ( a == b )
              area = c*sqrt(4*a*b-c*c)/4;
            else
              area = a*sqrt(4*b*c-a*c)/4;
            end
        end
    end
```

The CFG of tB2002

The target paths of tB2002

```
[1 1]; % equilateral
```

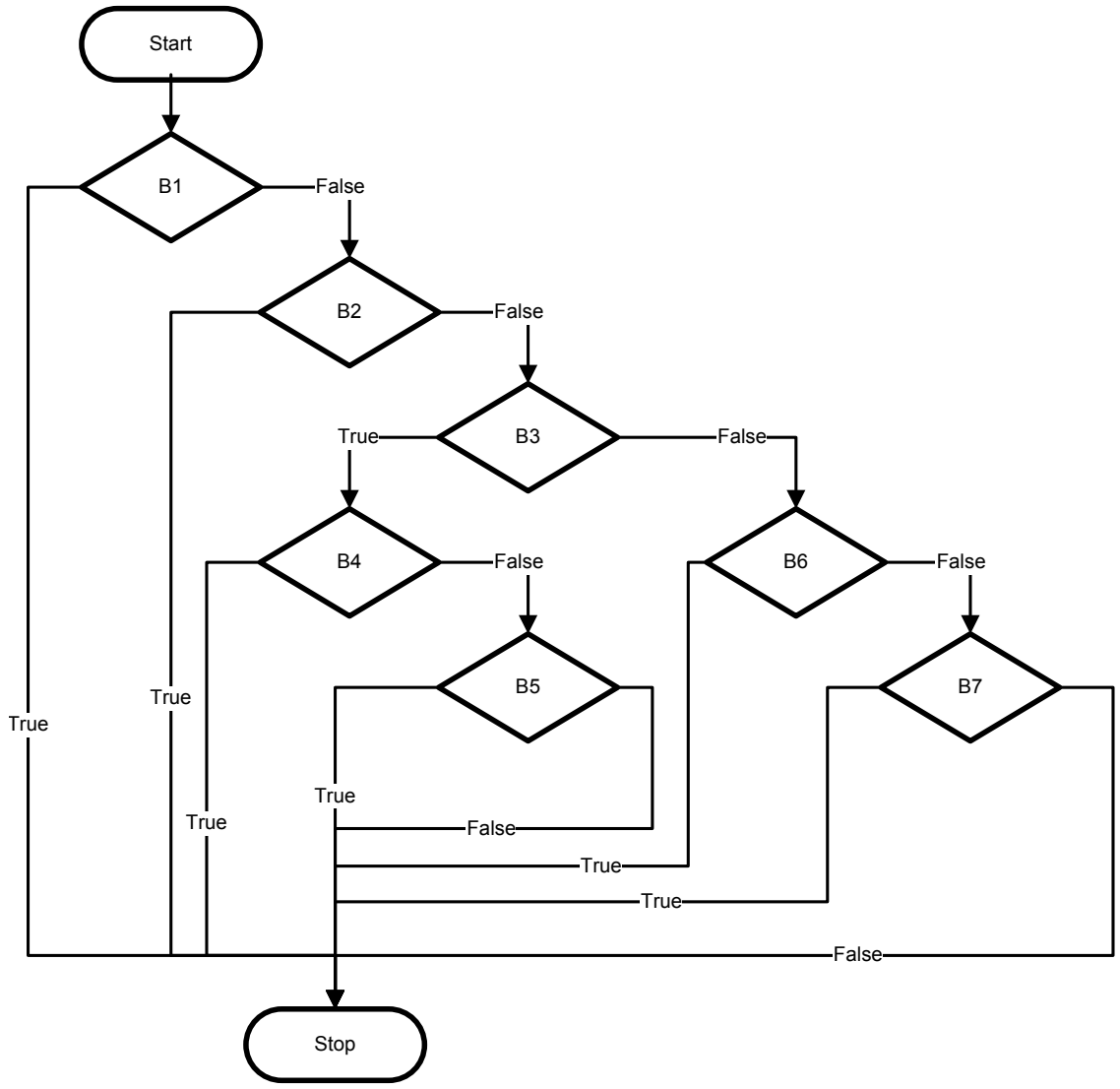


Figure B.11: CFG of tB2002

```
[1 0 2 1]; % invalid input
[1 1 3 0]; % not a triangle
[1 1 4 0 6 1]; % rectangle
[1 1 7 1]; % isosceles
[1 1 4 1 5 0]; % agudo
[1 1 4 1 5 1]; % obtuso
[1 1 3 0]; % escalano
```

The fitness function of tB2002

```
function branchVal = fitnessTritype(branchNo, a, b, c)
```

```
    k = 1; % the smallest step for integer
    switch (branchNo)
       case 1,
          % branch #1: ((a < b) || (b < c))
          term1 = fitnessKorel('<', a, b);
          term2 = fitnessKorel('<', b, c);
          branchVal = fitnessKorel('||', term1, term2);
       case 2,
          % branch #2: a >= (b + c)
          branchVal = fitnessKorel('>=', a, (b+c));
       case 3,
          % branch #3: (a ~= b) && (b ~= c)
          term1 = fitnessKorel('~=', a, b);
          term2 = fitnessKorel('~=', b, c);
          branchVal = fitnessKorel('&&', term1, term2);
       case 4,
          % branch #4: as == bs + cs
          branchVal = fitnessKorel('==', a, (b+c));
       case 5,
          % branch #1: as < bs + cs
          branchVal = fitnessKorel('<', a, (b+c));
       case 6,
          % branch #2: (a == b) && (b == c)
          term1 = fitnessKorel('==', a, b);
          term2 = fitnessKorel('==', b, c);
          branchVal = fitnessKorel('&&', term1, term2);
       case 7,
          % branch #3: a == b
          branchVal = fitnessKorel('==', a, b);
    end
end
```

The Korel's fitness function

```
function distance = fitnessKorel(operator, operand1, operand2)
    k = 1;
    % distance <= 0 means TRUE branch is exercised
    switch operator
        case '=='
            distance = abs(operand1 - operand2);
        case '~='
            if (abs(operand1 - operand2) > 0)
                distance = 0;
            else
                distance = k;
            end
        case '<'
            distance = operand1 - operand2 + k;
        case '<='
            distance = operand1 - operand2;
```

```
            case '>'
                distance = operand2 - operand1 + k;
            case '>='
                distance = operand2 - operand1;
            case '||'
                distance = min(operand1, operand2);
            case '&&'
                distance = max(operand1, operand2);
        end
    end
```

B.12 Test Program eB2002

The source code of eB2002

```
function [path, result] = expintBueno2002(numbersIn)
    n = numbersIn(1); % integer
    x = numbersIn(2); % float
    path = [];
    MAXIT = 100;
    EULER = 0.5772156649;
    FPMIN = 1.0e-30;
    EPS = 1.0e-7;
    nm1 = n - 1;
    % instrument B1
    path = [path 1 fitnessExpintBueno2002(1, n, x)];
    if (n < 0 || x < 0.0 || (x == 0.0 && (n == 0.0 || n==1)))
        result = 0;
    %     disp('bad arguments in expintBueno2002');
    % instrument B2
    path = [path 2 fitnessExpintBueno2002(2, n, 0)];
    elseif (n == 0)
        result = exp(-x)/x;
    % instrument B3
    path = [path 3 fitnessExpintBueno2002(3, x, 0.0)];
    elseif (x == 0.0)
        result = 1.0/nm1; % strangy: what is nm1?
    % instrument B4
    path = [path 4 fitnessExpintBueno2002(4, x, 1.0)];
    elseif (x > 1.0)
        b = x + n;
        c = 1.0 / FPMIN;
        d = 1.0 / b;
        h = d;
        % instrument B5
        i = 1; % for instrumentation purpose only
        path = [path 5 fitnessExpintBueno2002(5, i, MAXIT)];
```

```
    for i=1 : MAXIT
        a = -i * (nm1 + i);
        b = b + 2.0;
        d = 1.0 / (a*d+b);
        c = b + a / c;
        del = c * d;
        h = h * del;
        % instrument B6
        path = [path 6 fitnessExpintBueno2002(6, abs(del-1.0), EPS)];
        if (abs(del-1.0) < EPS) % abs is fabs in C
            result = h * exp(-x);
            return;
        end
    path = [path 5 fitnessExpintBueno2002(5, i, MAXIT)];
    end
    disp('continuated fraction failed in expint');
else
    % ans = (nm1!=0 ? 1.0/nm1 : -log(x)-EULER);
    % is interpreted as follows
    % instrument B7
    path = [path 7 fitnessExpintBueno2002(7, nm1, 0)];
    if (nm1 ~= 0)
        result = 1.0 / nm1;
    else
        result = -log(x)-EULER;
    end
    fact = 1.0;
    % instrument B8
    i = 1; % for instrumentation purpose only
    path = [path 8 fitnessExpintBueno2002(8, i, MAXIT)];
    for i = 1 : MAXIT
        fact = fact * (-x / i);
        % instrument B9
        path = [path 9 fitnessExpintBueno2002(9, i, nm1)];
        if (i ~= nm1)
            del = -fact / (i - nm1);
        else
            psi = -EULER;
            % instrument B10
            ii = 1; % for instrumentation purpose only
            path = [path 10 fitnessExpintBueno2002(10, ii, nm1)];
            for ii = 1 : nm1
                psi = psi + (1/ii);
            path = [path 10 fitnessExpintBueno2002(10, ii, nm1)];
            end
            del = fact * (-log(x) + psi);
        end
        result = result + del;
        % instrument B11
        path = [path 11 fitnessExpintBueno2002(11, abs(del), (abs(result)*EPS))];
        if (abs(del) < abs(result) * EPS) % abs is fabs in C
            return;
```

```
            end
        path = [path 8 fitnessExpintBueno2002(8, i, MAXIT)];
        end
        disp('series failed in expint');
    end
end
```

The CFG of eB2002

The target paths of eB2002

```
% No loop at B1
[1 1];

[1 0 2 0]; % input: [0 0]
[1 1 3 1]; % input: [0 1]
[1 0 2 1]; % input: [1 0]
[1 1 4 1]; % input: [2 0]

% One loop at B1 via B2-TF and B5-TF
[1 0 2 0 3 0 4 0 5 0];
[1 0 2 0 3 0 4 0 5 1];
[1 0 2 1 3 0 4 0 5 0];
[1 0 2 1 3 0 4 0 5 1];

% One loop at B1 via B2-TF and B4-F
[1 0 2 0 3 0 4 1];
[1 0 2 1 3 0 4 1];

% One loop at B1 via B2-TF and no loop at B6 & B7 via B8-TF
[1 0 2 0 3 1 6 1 7 1 8 0];
[1 0 2 0 3 1 6 1 7 1 8 1];
[1 0 2 1 3 1 6 1 7 1 8 0];
[1 0 2 1 3 1 6 1 7 1 8 1];

% One loop at B1 via B2-TF and one loop at B6 & B7 via B8-TF
[1 0 2 0 3 1 6 0 6 1 7 0 7 1 8 0];
[1 0 2 0 3 1 6 0 6 1 7 0 7 1 8 1];
[1 0 2 1 3 1 6 0 6 1 7 0 7 1 8 0];
[1 0 2 1 3 1 6 0 6 1 7 0 7 1 8 1];

% One loop at B1 via B2-TF and two loops at B6 & B7 via B8-TF
[1 0 2 0 3 1 6 0 6 1 6 0 6 1 7 0 7 1 7 0 7 1 8 0];
[1 0 2 0 3 1 6 0 6 1 6 0 6 1 7 0 7 1 7 0 7 1 8 1];
[1 0 2 1 3 1 6 0 6 1 6 0 6 1 7 0 7 1 7 0 7 1 8 0];
[1 0 2 1 3 1 6 0 6 1 6 0 6 1 7 0 7 1 7 0 7 1 8 1];

% One loop @B1 via B2-TF and one loop @B6 & two loops @B7 via B8-TF
[1 0 2 0 3 1 6 0 6 1 7 0 7 1 7 0 7 1 8 0];
[1 0 2 0 3 1 6 0 6 1 7 0 7 1 7 0 7 1 8 1];
```

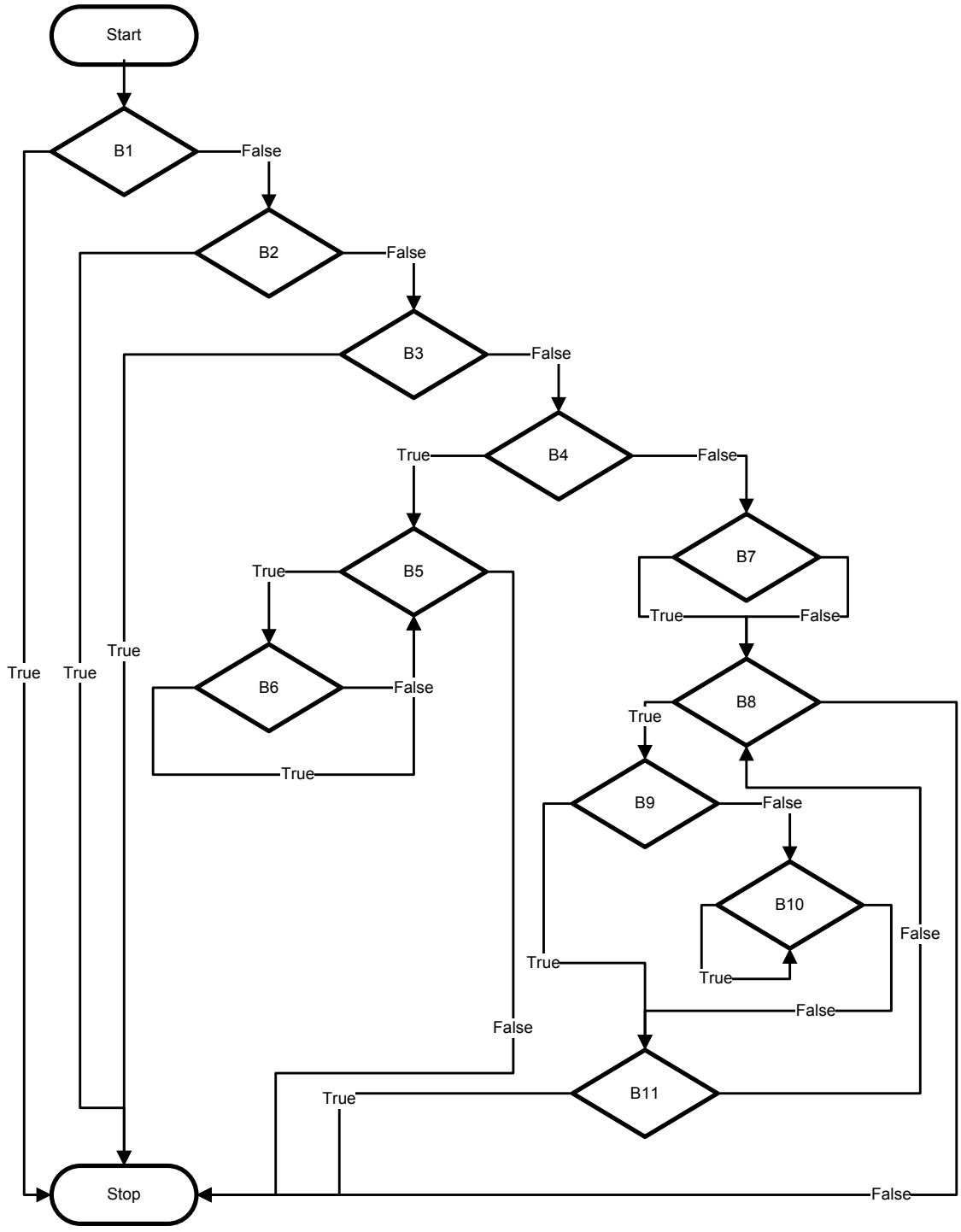


Figure B.12: CFG of eB2002

```
[1 0 2 1 3 1 6 0 6 1 7 0 7 1 7 0 7 1 8 0];
[1 0 2 1 3 1 6 0 6 1 7 0 7 1 7 0 7 1 8 1];
```

```
% One loop @B1 via B2-TF and one loop @B7 & two loops @B6 via B8-TF
[1 0 2 0 3 1 6 0 6 1 6 0 6 1 7 0 7 1 8 0];
[1 0 2 0 3 1 6 0 6 1 6 0 6 1 7 0 7 1 8 1];
[1 0 2 1 3 1 6 0 6 1 6 0 6 1 7 0 7 1 8 0];
[1 0 2 1 3 1 6 0 6 1 6 0 6 1 7 0 7 1 8 1];
```

The fitness function of eB2002

```
function branchVal = fitnessExpintBueno2002(branchNo, A, B)
    k = 1; % the smallest step for integer
    switch (branchNo)
      case 1,
          % branch #1: (n < 0 || x < 0.0 || (x == 0.0 && (n == 0.0 || n==1)))
          % n = A; x = B
          t1 = fitnessKorel('<', A, 0);
          t2 = fitnessKorel('<', B, 0.0);
          t3 = fitnessKorel('==', B, 0.0);
          t4 = fitnessKorel('==', A, 0.0);
          t5 = fitnessKorel('==', A, 1);
          t45 = fitnessKorel('||', t4, t5);
          t345 = fitnessKorel('&&', t3, t45);
          t12 = fitnessKorel('||', t1, t2);
          t12345 = fitnessKorel('||', t12, t345);
          branchVal = t12345;
      case 2,
          % branch #2: (n == 0)
          branchVal = fitnessKorel('==', A, B);
      case 3,
          % branch #3: (x == 0.0)
          branchVal = fitnessKorel('==', A, B);
      case 4,
          % branch #4: (x > 1.0)
          branchVal = fitnessKorel('>', A, B);
      case 5,
          % branch #5: (i <= MAXIT)
          % i = A; MAXIT = B
          branchVal = fitnessKorel('<=', A, B);
      case 6,
          % branch #6: (abs(del-1.0) < EPS)
          branchVal = fitnessKorel('<', A, B);
      case 7,
          % branch #7: (nm1 ~= 0)
          branchVal = fitnessKorel('~=', A, B);
      case 8,
          % branch #8: (i <= MAXIT)
          % i = A; MAXIT = B
          branchVal = fitnessKorel('<=', A, B);
      case 9,
          % branch #9: (i ~= nm1)
```

```
            branchVal = fitnessKorel('~=', A, B);
        case 10,
            % branch #10: ii <= nm1
            branchVal = fitnessKorel('<=', A, B);
        case 11,
            % branch #11: (abs(del) < abs(result) * EPS)
            branchVal = fitnessKorel('<', A, B);
    end
end
```

B.13 Test Program qB2002

The source code of qB2002

```
function [path, q, r] = quotientBueno2002(operands)
    path = [];
    n = operands(1); % First number
    d = operands(2); % Second number
    q = 0;
    % instrument B1
    path = [path 1 fitnessQuotientBueno2002(1, d)];
    if (d ~= 0)
       % instrument B2
       path = [path 2 fitnessQuotientBueno2002(2, d, n)];
       if ( (d > 0) && (n > 0) )
          q = 0;
          r = n;
          t = d;
          % instrument B3
          path = [path 3 fitnessQuotientBueno2002(3, r, t)];
          while (r >= t)
             t = t * 2;
          % instrument B3
          path = [path 3 fitnessQuotientBueno2002(3, r, t)];
          end
          % instrument B4
          path = [path 4 fitnessQuotientBueno2002(4, t, d)];
          while (t ~= d)
             q = q * 2;
             t = t / 2;
          % instrument B5
          path = [path 5 fitnessQuotientBueno2002(5, t, r)];
             if (t <= r)
                r = r - t;
                q = q + 1;
             end
          % instrument B4
```

```
            path = [path 4 fitnessQuotientBueno2002(4, t, d)];
            end
        end
    end
end
```

The CFG of qB2002

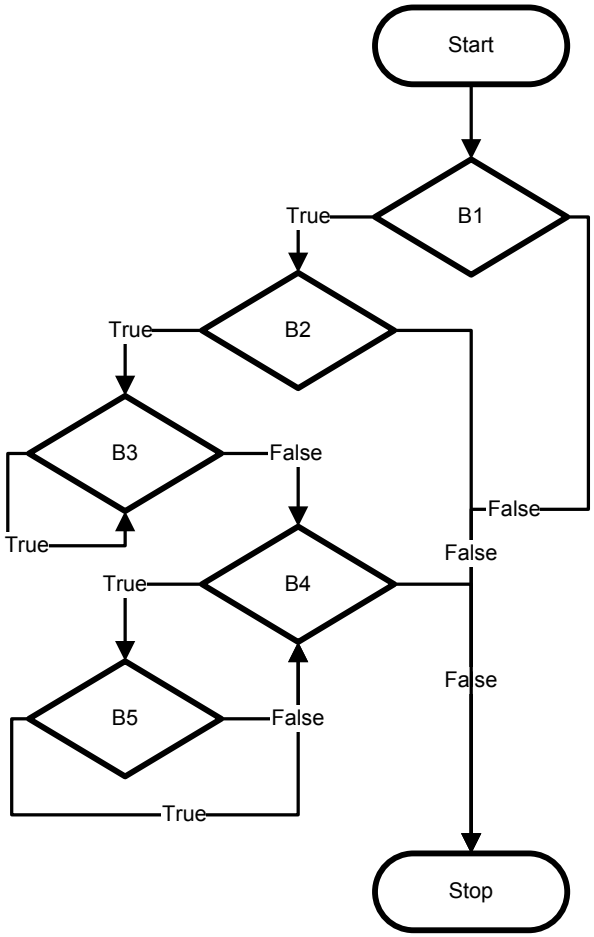


Figure B.13: CFG of qB2002

The target paths of qB2002

```
[1 1]; % f
[1 0 2 1]; % f

% No loop at B3 & no loop at B4
[1 0 2 0 3 1 4 1]; % f
```

```
% No loop at B3 & One loop at B4 with TRUE at B5
[1 0 2 0 3 1 4 0 5 0 4 1];
% No loop at B3 & One loop at B4 with FALSE at B5
[1 0 2 0 3 1 4 0 5 1 4 1];

% No loop at B3 & two loops at B4 with TRUE-TRUE at B5
[1 0 2 0 3 1 4 0 5 0 4 0 5 0 4 1];
% No loop at B3 & two loops at B4 with FALSE-FALSE at B5
[1 0 2 0 3 1 4 0 5 1 4 0 5 1 4 1];
% No loop at B3 & two loops at B4 with TRUE-FALSE at B5
[1 0 2 0 3 1 4 0 5 0 4 0 5 1 4 1];
% No loop at B3 & two loops at B4 with FALSE-TRUE at B5
[1 0 2 0 3 1 4 0 5 1 4 0 5 0 4 1];

% One loop at B3 & no loop at B4
[1 0 2 0 3 0 3 1 4 1];

% One loop at B3 & one loop at B4 with TRUE at B5
[1 0 2 0 3 0 3 1 4 0 5 0 4 1]; % f
% One loop at B3 & one loop at B4 with FALSE at B5
[1 0 2 0 3 0 3 1 4 0 5 1 4 1];

% One loop at B3 & two loops at B4 with TRUE-TRUE at B5
[1 0 2 0 3 0 3 1 4 0 5 0 4 0 5 0 4 1];
% One loop at B3 & two loops at B4 with FALSE-FALSE at B5
[1 0 2 0 3 0 3 1 4 0 5 1 4 0 5 1 4 1];
% One loop at B3 & two loops at B4 with TRUE-FALSE at B5
[1 0 2 0 3 0 3 1 4 0 5 0 4 0 5 1 4 1];
% One loop at B3 & two loops at B4 with FALSE-TRUE at B5
[1 0 2 0 3 0 3 1 4 0 5 1 4 0 5 0 4 1];

% Two loops at B3 & no loop at B4
[1 0 2 0 3 0 3 0 3 1 4 1];

% Two loops at B3 & one loop at B4 with TRUE at B5
[1 0 2 0 3 0 3 0 3 1 4 0 5 0 4 1];
% Two loops at B3 & one loop at B4 with FALSE at B5
[1 0 2 0 3 0 3 0 3 1 4 0 5 1 4 1];

% Two loops at B3 & two loops at B4 with TRUE-TRUE at B5
[1 0 2 0 3 0 3 0 3 1 4 0 5 0 4 0 5 0 4 1]; % f
% Two loops at B3 & two loops at B4 with FALSE-FALSE at B5
[1 0 2 0 3 0 3 0 3 1 4 0 5 1 4 0 5 1 4 1];
% Two loops at B3 & two loops at B4 with TRUE-FALSE at B5
[1 0 2 0 3 0 3 0 3 1 4 0 5 0 4 0 5 1 4 1]; % f
% Two loops at B3 & two loops at B4 with FALSE-TRUE at B5
[1 0 2 0 3 0 3 0 3 1 4 0 5 1 4 0 5 0 4 1];

% Feasible
[1 0 2 0 3 0 3 0 3 0 3 1 4 0 5 0 4 0 5 1 4 0 5 1 4 1];
[1 0 2 0 3 0 3 0 3 0 3 1 4 0 5 0 4 0 5 1 4 0 5 0 4 1];
```

```
[1 0 2 0 3 0 3 0 3 0 3 1 4 0 5 0 4 0 5 0 4 0 5 1 4 1];
```

The fitness function of qB2002

```
function branchVal = fitnessQuotientBueno2002(branchNo, a, b)
    k = 1; % the smallest step for integer
    switch (branchNo)
       case 1,
          % branch #1: d ~= 0
          branchVal = fitnessKorel('~=', a, 0);
       case 2,
          % branch #2: (d > 0) && (n > 0)
          term1 = fitnessKorel('>', a, 0);
          term2 = fitnessKorel('>', b, 0);
          branchVal = fitnessKorel('&&', term1, term2);
       case 3,
          % branch #3: r >= t
          branchVal = fitnessKorel('>=', a, b);
       case 4,
          % branch #4: t ~= d
          branchVal = fitnessKorel('~=', a, b);
       case 5,
          % branch #1: t <= r
          branchVal = fitnessKorel('<=', a, b);
    end
end
```

B.14 Test Program scB2002

The source code of scB2002

```
function [path, result] = strcompBueno2002(strin)
    path = [];
    result = ' ';
    i = 1;
    % strin is an array of integers (double) with length 8.
    str = char(strin);
    % instrument B1
    path = [path 1 fitnessStrcompBueno2002(1, i, str)];
    while ((str(i) ~= ' ') && (i <= 5))
       i = i + 1;
    path = [path 1 fitnessStrcompBueno2002(1, i, str)];
    end
    % instrument B2
```

```
    path = [path 2 fitnessStrcompBueno2002(2, i, str)];
    if (~strcmp(str(1:5),'test1'))
       % instrument B3
       path = [path 3 fitnessStrcompBueno2002(3, i, str)];
       if (str(6) == 'a')
          % instrument B4
          path = [path 4 fitnessStrcompBueno2002(4, i, str)];
          if (str(7) == 'b')
             % instrument B5
             path = [path 5 fitnessStrcompBueno2002(5, i, str)];
             if (str(8) < 'c')
                result = 'Gotcha';
             end
          end
       end
    end
end
```

The CFG of scB2002

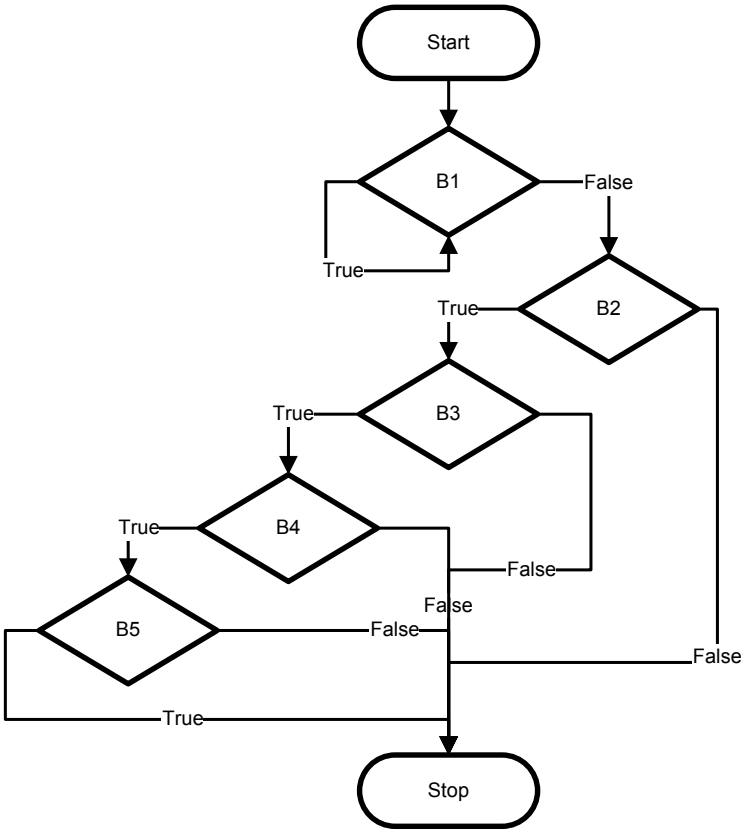


Figure B.14: CFG of scB2002

The target paths of scB2002

```
[1 1 2 1];
[1 1 2 0 3 1];
[1 1 2 0 3 0 4 1];
[1 1 2 0 3 0 4 0 5 1];
[1 1 2 0 3 0 4 0 5 0];

% One loop at B1
[1 0 1 1 2 1];
[1 0 1 1 2 0 3 1];
[1 0 1 1 2 0 3 0 4 1];
[1 0 1 1 2 0 3 0 4 0 5 1];
[1 0 1 1 2 0 3 0 4 0 5 0];

% Two loops at B1
[1 0 1 0 1 1 2 1];
[1 0 1 0 1 1 2 0 3 1];
[1 0 1 0 1 1 2 0 3 0 4 1];
[1 0 1 0 1 1 2 0 3 0 4 0 5 1];
[1 0 1 0 1 1 2 0 3 0 4 0 5 0];

% Five loops at B1 & exit at B2
[1 0 1 0 1 0 1 0 1 0 2 1];
```

The fitness function of scB2002

```
function branchVal = fitnessStrcompBueno2002(branchNo, i, str)
    k = 1; % the smallest step for integer
    switch (branchNo)
      case 1,
          % branch #1: ((str(i) ~= ' ') && (i <= 5))
          bv1 = fitnessKorel('~=', str(i), ' ');
          bv2 = fitnessKorel('<=', i, 5);
          branchVal = fitnessKorel('&&', bv1, bv2);
      case 2,
          % branch #2: ~strcmp(str(1:5),'test1')
          branchVal = ~strcmp(str(1:5),'test1');
      case 3,
          % branch #3: (str(6) == 'a')
          branchVal = fitnessKorel('==', str(6), 'a');
      case 4,
          % branch #4: (str(7) == 'b')
          branchVal = fitnessKorel('==', str(7), 'b');
      case 5,
          % branch #5: (str(8) < 'c')
          branchVal = fitnessKorel('<', str(8), 'c');
    end
end
```

B.15 Test Program fcB2002

The source code of fcB2002

```
function [traversedPath, result] = floatcompBueno2002(floats)
    traversedPath = [];
    f1 = floats(1); % First number
    f2 = floats(2); % Second number
    f3 = floats(3); % Third number
    result = ' ';
    % instrument B1
    traversedPath = [traversedPath 1 fitnessFloatcomp(1, f3, f2)];
    if (f3 > f2) % B1
        % instrument B2
        traversedPath = [traversedPath 2 fitnessFloatcomp(2, f2, f1)];
        if (f2 > f1) % B2
            result = 'f3 > f2 > f1';
            t = f1 + f2;
            % instrument B3
            traversedPath = [traversedPath 3 fitnessFloatcomp(3, t, f3)];
            if (t < f3)
               result = 'f3 > f1 + f2';
               t2 = f1 * f2;
               % instrument B4
               traversedPath = [traversedPath 4 fitnessFloatcomp(4, t2, f3)];
               if (((t2 - f3) <= 5) && ((t2 - f3) >= 0))
                  result = '(((f1 * f2) - f3) <= 5) && (((f1 * f2) - f3) >= 0))';
               end
            else
               result = 'f3 <= f1 + f2';
            end
        end
    end
end
```

The CFG of fcB2002

The target paths of fcB2002

```
[1 1];
[1 0 2 1];
[1 0 2 0 3 1];
[1 0 2 0 3 0 4 1];
[1 0 2 0 3 0 4 0];
```

The fitness function of fcB2002

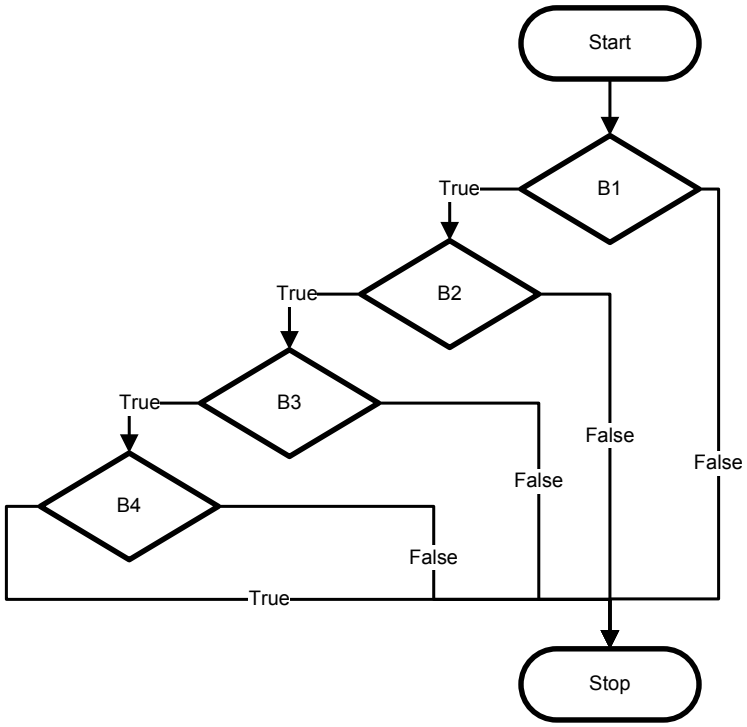


Figure B.15: CFG of fcB2002

```
function branchVal = fitnessFloatcomp(branchNo, A, B)
    k = 1; % the smallest step for integer
    switch (branchNo)
      case 1,
          f3 = A; f2 = B;
          branchVal = fitnessKorel('>', f3, f2);
      case 2,
          f2 = A; f1 = B;
          branchVal = fitnessKorel('>', f2, f1);
      case 3,
          t = A; f3 = B;
          branchVal = fitnessKorel('<', t, f3);
      case 4,
          % branch #4: (((t2 - f3) <= 5) && ((t2 - f3) >= 0))
          t2 = A; f3 = B;
          bv1 = fitnessKorel('<=', (t2-f3), 5);
          bv2 = fitnessKorel('>=', (t2-f3), 0);
          branchVal = fitnessKorel('&&', bv1, bv2);
    end
end
```

B.16 Test Program fB2002

The source code of fB2002

```
function [path, a] = findBueno2002(numbersIn)
    path = [];
    f = numbersIn(1); % key or index
    a = numbersIn(2:end); % an array of integers to be re-arranged
    % n = length(numbers);
    b = 0;
    m = 1;
    ns = length(a);
    % Probe added on 02.09.2010
    if f > ns
       f = mod(ns,f);
    end
    i = 1;
    % instrument B1
    path = [path 1 fitnessFindBueno2002(1, m, ns, b)];
    while ((m < ns) || b)
        % instrument B2
        path = [path 2 fitnessFindBueno2002(2, ~b)];
        if (~b)
            i = m;
            j = ns;
        else
            b = 0;
        end
    % /*-------------------------------*/
        % instrument B3
        path = [path 3 fitnessFindBueno2002(3, i, j)];
        if (i > j)
            % instrument B4
            path = [path 4 fitnessFindBueno2002(4, f, j)];
            if (f > j)
                % instrument B5
                path = [path 5 fitnessFindBueno2002(5, i, f)];
                if (i > f)
                    m = ns;
                else
                    m = i;
                end
            else
                ns = j;
            end
    % /*-------------------------------*/
        else
            % instrument B6
            path = [path 6 fitnessFindBueno2002(6, a(i), a(f))];
            while (a(i) < a(f))
                i = i + 1 ;
```

```
            path = [path 6 fitnessFindBueno2002(6, a(i), a(f))];
            end
            % instrument B7
            path = [path 7 fitnessFindBueno2002(7, a(f), a(j))];
            while (a(f) < a(j))
                j = j - 1 ;
            path = [path 7 fitnessFindBueno2002(7, a(f), a(j))];
            end
            % instrument B8
            path = [path 8 fitnessFindBueno2002(8, i, j)];
            if (i <= j)
                w = a(i);
                a(i) = a(j);
                a(j) = w;
                i = i + 1;
                j = j - 1;
            end
            b = 1;
        end
        path = [path 1 fitnessFindBueno2002(1, m, ns, b)];
    end
end
```

The CFG of fB2002

The target paths of fB2002

```
% No loop at B1
[1 0]; % feasible

% All feasible paths:
[1 0 2 1 3 1 6 1 7 1 8 0 1 0 2 0 3 0 4 1 1 0];
[1 0 2 1 3 1 6 1 7 0 7 1 8 0 1 0 2 0 3 0 4 0 5 0 1 0];
[1 0 2 1 3 1 6 1 7 1 8 0 1 0 2 0 3 1 6 1 7 1 8 0 1 0 2 0 3 0 4 1 1 0];
[1 0 2 1 3 1 6 1 7 1 8 0 1 0 2 0 3 1 6 1 7 0 7 1 8 1 1 0 2 0 3 0 4 1 1 0];
[1 0 2 1 3 1 6 1 7 0 7 1 8 0 1 0 2 0 3 0 4 1 1 0];

% One loop at B1 via B2-TF and B5-TF
[1 0 2 0 3 0 4 0 5 0];
[1 0 2 0 3 0 4 0 5 1];
[1 0 2 1 3 0 4 0 5 0];
[1 0 2 1 3 0 4 0 5 1];

% One loop at B1 via B2-TF and B4-F
[1 0 2 0 3 0 4 1];
[1 0 2 1 3 0 4 1];

% One loop at B1 via B2-TF and no loop at B6 & B7 via B8-TF
[1 0 2 0 3 1 6 1 7 1 8 0];
[1 0 2 0 3 1 6 1 7 1 8 1];
```

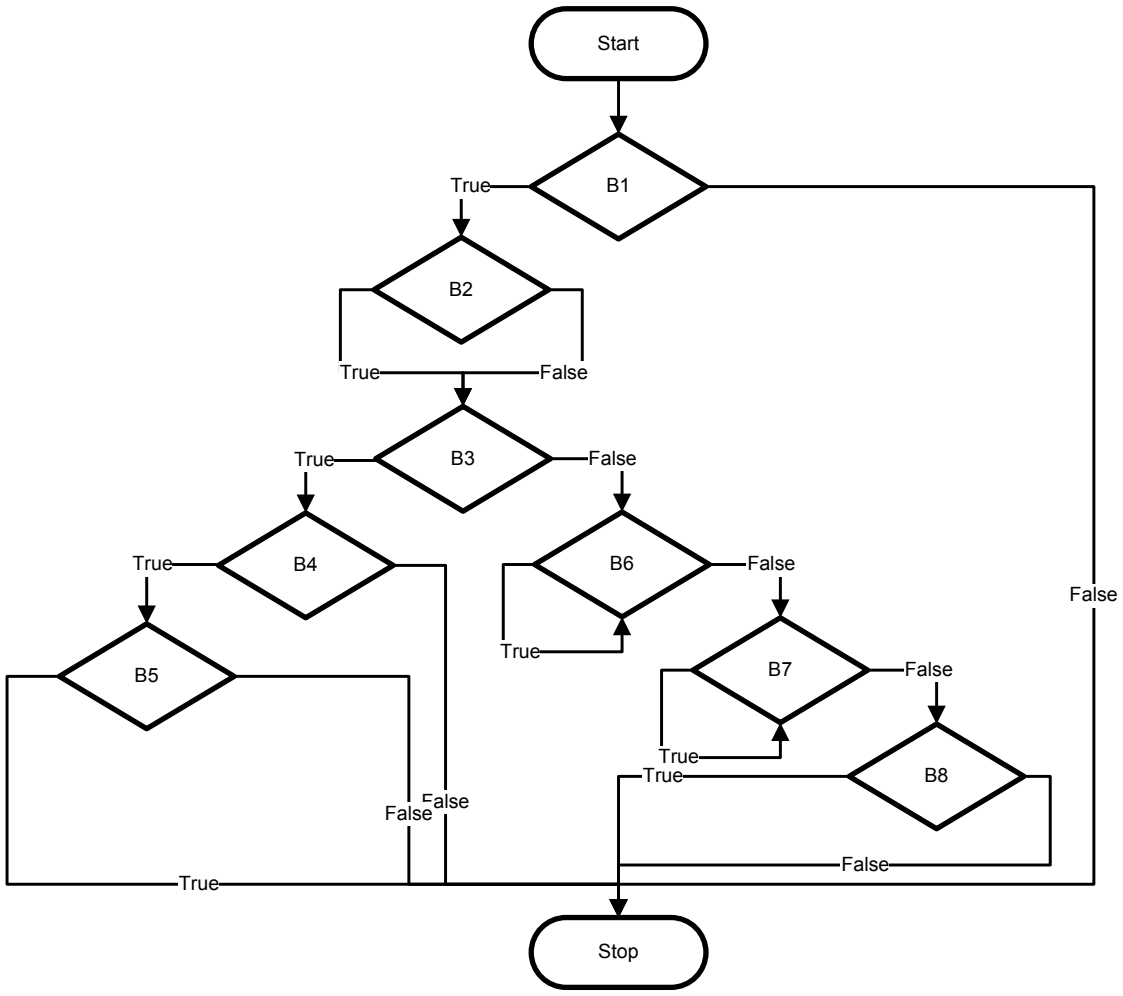


Figure B.16: CFG of fB2002

```
[1 0 2 1 3 1 6 1 7 1 8 0];
[1 0 2 1 3 1 6 1 7 1 8 1];

% One loop at B1 via B2-TF and one loop at B6 & B7 via B8-TF
[1 0 2 0 3 1 6 0 6 1 7 0 7 1 8 0];
[1 0 2 0 3 1 6 0 6 1 7 0 7 1 8 1];
[1 0 2 1 3 1 6 0 6 1 7 0 7 1 8 0];
[1 0 2 1 3 1 6 0 6 1 7 0 7 1 8 1];

% One loop at B1 via B2-TF and two loops at B6 & B7 via B8-TF
[1 0 2 0 3 1 6 0 6 1 6 0 6 1 7 0 7 1 7 0 7 1 8 0];
[1 0 2 0 3 1 6 0 6 1 6 0 6 1 7 0 7 1 7 0 7 1 8 1];
[1 0 2 1 3 1 6 0 6 1 6 0 6 1 7 0 7 1 7 0 7 1 8 0];
[1 0 2 1 3 1 6 0 6 1 6 0 6 1 7 0 7 1 7 0 7 1 8 1];

% One loop @B1 via B2-TF and one loop @B6 & two loops @B7 via B8-TF
```

```
[1 0 2 0 3 1 6 0 6 1 7 0 7 1 7 0 7 1 8 0];
[1 0 2 0 3 1 6 0 6 1 7 0 7 1 7 0 7 1 8 1];
[1 0 2 1 3 1 6 0 6 1 7 0 7 1 7 0 7 1 8 0];
[1 0 2 1 3 1 6 0 6 1 7 0 7 1 7 0 7 1 8 1];

% One loop @B1 via B2-TF and one loop @B7 & two loops @B6 via B8-TF
[1 0 2 0 3 1 6 0 6 1 6 0 6 1 7 0 7 1 8 0];
[1 0 2 0 3 1 6 0 6 1 6 0 6 1 7 0 7 1 8 1];
[1 0 2 1 3 1 6 0 6 1 6 0 6 1 7 0 7 1 8 0];
[1 0 2 1 3 1 6 0 6 1 6 0 6 1 7 0 7 1 8 1];
```

The fitness function of fB2002

```
function branchVal = fitnessFindBueno2002(branchNo, A, B, C)
    k = 1; % the smallest step for integer
    switch (branchNo)
      case 1,
          % branch #1: ((m < ns) || b)
          term1 = fitnessKorel('<', A, B);
          branchVal = fitnessKorel('||', term1, C);
      case 2,
          % branch #2: (~b); means A = ~b
          branchVal = A;
      case 3,
          % branch #3: (i > j)
          branchVal = fitnessKorel('>', A, B);
      case 4,
          % branch #4: (f > j)
          branchVal = fitnessKorel('>', A, B);
      case 5,
          % branch #5: (i > f)
          branchVal = fitnessKorel('>', A, B);
      case 6,
          % branch #6: (a(i) < a(f))
          branchVal = fitnessKorel('<', A, B);
      case 7,
          % branch #7: (a(f) < a(j))
          branchVal = fitnessKorel('<', A, B);
      case 8,
          % branch #8: (i <= j)
          branchVal = fitnessKorel('<=', A, B);
    end
end
```

B.17 Test Program bG2011

The source code of bG2011

```
function [traversedPath, pop] = bubbleGong2011(depop)
    [px,py]=size(depop);
    traversedPath = [];
    i = 1; % added for instrumentation purpose
    traversedPath = [traversedPath 1 fitnessBubbleGong2011(1, [i px])];
    for i=1:px % Branch # 1
        q=1;
        p=depop(i,:);
        j = 1; % added for instrumentation purpose
        traversedPath = [traversedPath 2 fitnessBubbleGong2011(2, [j (py-1)])];
        for j=1:py-1 % Branch # 2
            k = j+1; % added for instrumentation purpose
            traversedPath = [traversedPath 3 fitnessBubbleGong2011(3, [k py])];
            for k=j+1:py % Branch # 3
                d(q,1)=p(k)-p(j)+0.1;
                d(q,1)=1-1.001^(-d(q,1));
                d(q,2)=p(j)-p(k);
                d(q,2)=1-1.001^(-d(q,2));
                traversedPath = [traversedPath 4 fitnessBubbleGong2011(4, [p(j) p(k)])];
                if p(j)>p(k) % Branch # 4
                    temp=p(j);
                    p(j)=p(k);
                    p(k)=temp;
                    d(q,1)=0;
                else
                    d(q,2)=0;
                end
                q=q+1;
            traversedPath = [traversedPath 3 fitnessBubbleGong2011(3, [k py])];
            end
        traversedPath = [traversedPath 2 fitnessBubbleGong2011(2, [j (py-1)])];
        end
        pop(i,:) = p;
    traversedPath = [traversedPath 1 fitnessBubbleGong2011(1, [i px])];
    end
end
```

The CFG of bG2011

The target paths of bG2011

```
% B1's loop
  %  => not required as if at least one individual is processed
%   [1 1]; % no loop
```

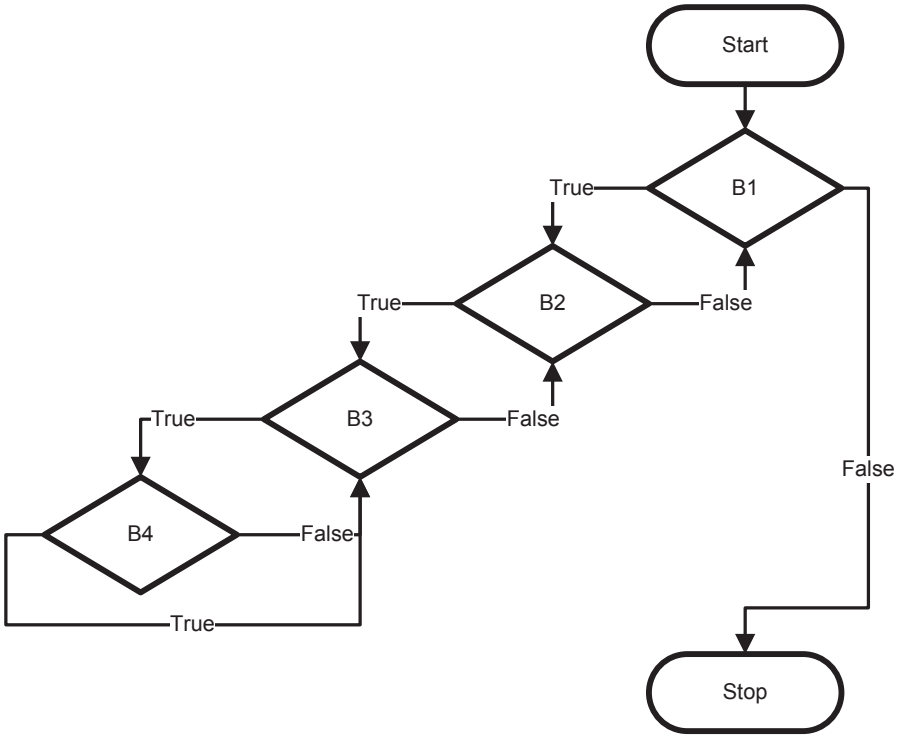


Figure B.17: CFG of bG2011

```
  [1 0 2 1 1 1]; % B2: no loop

  %  => not required as if at most one individual is processed
%   [1 0 2 1 1 0 2 1 1 1]; % 2 loops

% B2's loop
  [1 0 2 0 3 1 2 1 1 1]; % 1 loop & no loop @ B3
  [1 0 2 0 3 1 2 0 3 1 2 1 1 1]; % 2 loops

% B3's loop
  [1 0 2 0 3 0 4 1 3 1 2 1 1 1]; % 1 loop
  [1 0 2 0 3 0 4 0 3 1 2 1 1 1]; % 1 loop
  [1 0 2 0 3 0 4 0 3 0 4 0 3 1 2 1 1 1]; % 2 loops
  [1 0 2 0 3 0 4 0 3 0 4 1 3 1 2 1 1 1]; % 2 loops
  [1 0 2 0 3 0 4 1 3 0 4 1 3 1 2 1 1 1]; % 2 loops
  [1 0 2 0 3 0 4 1 3 0 4 0 3 1 2 1 1 1]; % 2 loops

  % Input-based target path (feasible)
  % 1-number input
  [1 0 2 1 1 0]; % Input ([x]), e.g. [1]

  % 2-number input
  [1 0 2 0 3 0 4 1 3 0 2 0 1 0]; % Input [x y]; x <= y
  [1 0 2 0 3 0 4 0 3 0 2 0 1 0]; % Input [x y]; x > y
```

```
% 3-number input;
% [0 0 0], [0 0 1], [0 1 2]
[1 0 2 0 3 0 4 1 3 0 4 1 3 0 2 0 3 0 4 1 3 0 2 0 1 0];

% [0 1 0]
[1 0 2 0 3 0 4 1 3 0 4 1 3 0 2 0 3 0 4 0 3 0 2 0 1 0];

% [1 0 0]
[1 0 2 0 3 0 4 0 3 0 4 1 3 0 2 0 3 0 4 0 3 0 2 0 1 0];

% [1 2 0]
[1 0 2 0 3 0 4 1 3 0 4 0 3 0 2 0 3 0 4 0 3 0 2 0 1 0];

% [0 2 1]
[1 0 2 0 3 0 4 1 3 0 4 1 3 0 2 0 3 0 4 0 3 0 2 0 1 0];

% [2 1 0]
[1 0 2 0 3 0 4 0 3 0 4 0 3 0 2 0 3 0 4 0 3 0 2 0 1 0];

% [1 0 2]
[1 0 2 0 3 0 4 0 3 0 4 1 3 0 2 0 3 0 4 1 3 0 2 0 1 0];

% [2 0 1]
[1 0 2 0 3 0 4 0 3 0 4 1 3 0 2 0 3 0 4 0 3 0 2 0 1 0];
```

The fitness function of bG2011

```
function branchVal = fitnessBubbleGong2011(branchNo, predicate)
    k = 1; % the smallest step for integer
    switch (branchNo)
      case {1, 2, 3}
          % Branch #1: for i=1:px ==> i <= px
          % Branch #2: for j=1:py-1 ==> j <= py-1
          % branch #3: for k=j+1:py ==> k <= py
          branchVal = predicate(1) - predicate(2);
      case 4,
          % Branch #4: if p(j)>p(k)
          branchVal = predicate(2) - predicate(1);
          if branchVal < 0
             branchVal = branchVal - k;
          else
             branchVal = branchVal + k;
          end
    end
end
```

B.18 Test Program fG2011

The source code of fG2011

```
% The function accept a population of 7-number inputs
function [traversedPath, pop] = flexGong2011(depop)
    [px,py]=size(depop);
    traversedPath = [];

    i = 1; % added for instrumentation purpose
    traversedPath = [traversedPath 1 fitnessFlexGong2011(1, [i px])];
    for i=1:px % Branch # 1
        q=1;
        p=depop(i,:);
        lex_compat=p(1);
        C_plus_plus=p(2);
        fulltbl=p(3);
        csize =p(4);
        unspecified=p(5);
        fullspd=p(6);
        C_plus=p(7);

        d(1,1)=2;
        d(1,1)=1-1.001^(-d(1,1));
        d(1,2)=lex_compat;
        d(1,2)=1-1.001^(-d(1,2));
        d(1,3)=0;
        d(2,1)=2;
        d(2,1)=1-1.001^(-d(2,1));
        d(2,2)=C_plus_plus;
        d(2,2)=1-1.001^(-d(2,2));
        d(2,3)=0;
        d(3,1)=2;
        d(3,1)=1-1.001^(-d(3,1));
        d(3,2)=fulltbl;
        d(3,2)=1-1.001^(-d(3,2));
        d(3,3)=0;
        d(4,1)=abs( csize-unspecified)+2;
        d(4,1)=1-1.001^(-d(4,1));
        d(4,2)=2;
        d(4,2)=1-1.001^(-d(4,2));
        d(4,3)=0;
        d(5,1)=2;
        d(5,1)=1-1.001^(-d(5,1));
        d(5,2)=fullspd;
        d(5,2)=1-1.001^(-d(5,2));
        d(5,3)=0;
        d(6,1)=2;
        d(6,1)=1-1.001^(-d(5,1));
        d(6,2)=C_plus;
        d(6,2)=1-1.001^(-d(5,2));
```

```
    d(5,3)=0;
    u=3*ones(1,6);

  traversedPath = [traversedPath 2 fitnessFlexGong2011(2, [lex_compat 0])];
 if (lex_compat ~= 0) % Branch # 2
        d(1,1)=0;
        u(1)=1;

    traversedPath = [traversedPath 3 fitnessFlexGong2011(3, [C_plus_plus 0])];
    if (C_plus_plus ~= 0) % Branch # 3
 flexerror = 'Can not use -+ with -l option';
      d(2,1)=0;
    else
      d(2,2)=0;
    end

    traversedPath = [traversedPath 4 fitnessFlexGong2011(4, [fulltbl 0])];
    if (fulltbl ~= 0) % Branch # 4
 flexerror='Can not use -f or -F with -l option';
      d(3,1)=0;
    else
      d(3,2)=0;
    end
  else
    d(1,2)=0;
  end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  traversedPath = [traversedPath 5 fitnessFlexGong2011(5, [csize unspecified])];
  if (csize == unspecified) % Branch # 5
        d(4,1)=0;

    traversedPath = [traversedPath 6 fitnessFlexGong2011(6, [fullspd 0])];
    if (fullspd ~= 0) % Branch # 6
 csize = 'DEFAULT_CSIZE';
      d(5,1)=0;
 else
      d(5,2)=0;
 csize = csize;
    end
  else
    d(4,2)=0;
  end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  traversedPath = [traversedPath 7 fitnessFlexGong2011(7, [C_plus 0])];
  if (C_plus ~= 0) % Branch # 7
    suffix='cc';
    d(6,1)=0;
  else
    d(6,2)=0;
    suffix='c';
    outfilename = 'outfile_path';
  end
```

```
        pop(i,:) = p;
      traversedPath = [traversedPath 1 fitnessFlexGong2011(1, [i px])];
end
```

The CFG of fG2011

The target paths of fG2011

```
[1 0 2 1 5 1 7 1];
[1 0 2 1 5 1 7 0];

[1 0 2 1 5 0 6 0 7 0];
[1 0 2 1 5 0 6 0 7 1];
[1 0 2 1 5 0 6 1 7 0];
[1 0 2 1 5 0 6 1 7 1];

[1 0 2 0 3 1 4 1 5 1 7 1];
[1 0 2 0 3 1 4 1 5 1 7 0];
[1 0 2 0 3 0 4 1 5 1 7 1];
[1 0 2 0 3 0 4 1 5 1 7 0];
[1 0 2 0 3 1 4 0 5 1 7 1];
[1 0 2 0 3 1 4 0 5 1 7 0];
[1 0 2 0 3 0 4 0 5 1 7 1];
[1 0 2 0 3 0 4 0 5 1 7 0];

[1 0 2 0 3 1 4 1 5 0 6 1 7 1];
[1 0 2 0 3 1 4 1 5 0 6 0 7 1];
[1 0 2 0 3 1 4 1 5 0 6 1 7 0];
[1 0 2 0 3 1 4 1 5 0 6 0 7 0];
[1 0 2 0 3 1 4 0 5 0 6 1 7 1];
[1 0 2 0 3 1 4 0 5 0 6 0 7 1];
[1 0 2 0 3 1 4 0 5 0 6 1 7 0];
[1 0 2 0 3 1 4 0 5 0 6 0 7 0];
[1 0 2 0 3 0 4 1 5 0 6 1 7 1];
[1 0 2 0 3 0 4 1 5 0 6 0 7 1];
[1 0 2 0 3 0 4 1 5 0 6 1 7 0];
[1 0 2 0 3 0 4 1 5 0 6 0 7 0];
[1 0 2 0 3 0 4 0 5 0 6 1 7 1];
[1 0 2 0 3 0 4 0 5 0 6 0 7 1];
[1 0 2 0 3 0 4 0 5 0 6 1 7 0];
[1 0 2 0 3 0 4 0 5 0 6 0 7 0];
```

The fitness function of fG2011

```
function branchVal = fitnessFlexGong2011(branchNo, predicate)
    k = 1; % the smallest step for integer
```

```
    switch (branchNo)
      case 1,
          % Branch #1: for i=1:px ==> i <= px
          branchVal = predicate(1) - predicate(2);
      case {2, 3, 4, 6, 7},
          % Branch #2: if (lex_compat ~= 0)
          % Branch #3: if (C_plus_plus ~= 0)
          % Branch #4: if (fulltbl ~= 0)
          % Branch #6: if (fullspd ~= 0)
          % Branch #7: if (C_plus ~= 0)
          if predicate(1) ~= 0
            branchVal = 0;
          else
            branchVal = k;
          end
      case 5,
          % Branch #5: if (csize == unspecified)
          if predicate(1) == predicate(2)
            branchVal = 0;
          else
            branchVal = k;
          end
    end
end
```

B.19 Test Program sG2011

The source code of sG2011

```
% The function accepts a population of 5-number inputs
function [traversedPath, pop] = spaceGong2011(depop)
    [px,py]=size(depop);
    traversedPath = [];

    i = 1; % added for instrumentation purpose
    traversedPath = [traversedPath 1 fitnessSpaceGong2011(1, [i px])];
    for i=1:px % Branch # 1
        q=1;
        p=depop(i,:);
        unit1=p(1);
        unit2=p(2);
        unit3=p(3);
        error1=p(4);
        error2=p(5);

        d(1,1)=abs(unit1-1);
        d(1,2)=2;
```

```
  d(1,1)=1-1.001^(-d(1,1));
  d(1,2)=1-1.001^(-d(1,2));
  d(2,1)=abs(unit2-2);
  d(2,1)=1-1.001^(-d(2,1));
  d(2,2)=2;
  d(2,2)=1-1.001^(-d(2,2));
  d(3,1)=abs(unit3-3);
  d(3,1)=1-1.001^(-d(3,1));
  d(3,2)=2;
  d(3,2)=1-1.001^(-d(3,2));
  d(4,1)=abs(error1-0);
  d(4,1)=1-1.001^(-d(4,1));
  d(4,2)=2;
  d(4,2)=1-1.001^(-d(4,2));
  d(5,1)=abs(error2-0);
  d(5,1)=1-1.001^(-d(5,1));
  d(5,2)=2;
  d(5,2)=1-1.001^(-d(5,2));

  u=zeros(1,5);

traversedPath = [traversedPath 2 fitnessSpaceGong2011(2, [unit1 1])];
if unit1 == 1  % Branch # 2
  x_ptr=10;
  d(1,1)=0;
else
  d(1,2)=0;
end

traversedPath = [traversedPath 3 fitnessSpaceGong2011(3, [unit2 2])];
if unit2 == 2 % Branch # 3
  x_ptr=100;
  d(2,1)=0;
else
  d(2,2)=0;
end

traversedPath = [traversedPath 4 fitnessSpaceGong2011(4, [unit3 3])];
if unit3 == 3 % Branch # 4
  x_ptr=1000;
  d(3,1)=0;
else
  d(3,2)=0;
end

traversedPath = [traversedPath 5 fitnessSpaceGong2011(5, [error1 0])];
if error1 == 0 % Branch # 5
  d(4,1)=0;
 % return 1
else
  d(4,2)=0;
end
```

```
      traversedPath = [traversedPath 6 fitnessSpaceGong2011(6, [error2 0])];
      if error2 == 0 % Branch # 6
         d(5,1)=0;
      else
        d(5,2)=0;
      end

      pop(i,:) = p;
    traversedPath = [traversedPath 1 fitnessSpaceGong2011(1, [i px])];
    end
end
```

The CFG of sG2011

The target paths of sG2011

```
  %  => not required as if at least one individual is processed
%   [1 1];

  % One loop @ B1 with full combinations @ B2, B3, B4, B5, B6
  [ 1 0 2 0 3 0 4 0 5 0 6 0 1 0 ];
  [ 1 0 2 0 3 0 4 0 5 0 6 1 1 0 ];
  [ 1 0 2 0 3 0 4 0 5 1 6 0 1 0 ];
  [ 1 0 2 0 3 0 4 0 5 1 6 1 1 0 ];
  [ 1 0 2 0 3 0 4 1 5 0 6 0 1 0 ];
  [ 1 0 2 0 3 0 4 1 5 0 6 1 1 0 ];
  [ 1 0 2 0 3 0 4 1 5 1 6 0 1 0 ];
  [ 1 0 2 0 3 0 4 1 5 1 6 1 1 0 ];
  [ 1 0 2 0 3 1 4 0 5 0 6 0 1 0 ];
  [ 1 0 2 0 3 1 4 0 5 0 6 1 1 0 ];
  [ 1 0 2 0 3 1 4 0 5 1 6 0 1 0 ];
  [ 1 0 2 0 3 1 4 0 5 1 6 1 1 0 ];
  [ 1 0 2 0 3 1 4 1 5 0 6 0 1 0 ];
  [ 1 0 2 0 3 1 4 1 5 0 6 1 1 0 ];
  [ 1 0 2 0 3 1 4 1 5 1 6 0 1 0 ];
  [ 1 0 2 0 3 1 4 1 5 1 6 1 1 0 ];
  [ 1 0 2 1 3 0 4 0 5 0 6 0 1 0 ];
  [ 1 0 2 1 3 0 4 0 5 0 6 1 1 0 ];
  [ 1 0 2 1 3 0 4 0 5 1 6 0 1 0 ];
  [ 1 0 2 1 3 0 4 0 5 1 6 1 1 0 ];
  [ 1 0 2 1 3 0 4 1 5 0 6 0 1 0 ];
  [ 1 0 2 1 3 0 4 1 5 0 6 1 1 0 ];
  [ 1 0 2 1 3 0 4 1 5 1 6 0 1 0 ];
  [ 1 0 2 1 3 0 4 1 5 1 6 1 1 0 ];
  [ 1 0 2 1 3 1 4 0 5 0 6 0 1 0 ];
  [ 1 0 2 1 3 1 4 0 5 0 6 1 1 0 ];
  [ 1 0 2 1 3 1 4 0 5 1 6 0 1 0 ];
  [ 1 0 2 1 3 1 4 0 5 1 6 1 1 0 ];
  [ 1 0 2 1 3 1 4 1 5 0 6 0 1 0 ];
```

```
  [ 1 0 2 1 3 1 4 1 5 0 6 1 1 0 ];
  [ 1 0 2 1 3 1 4 1 5 1 6 0 1 0 ];
  [ 1 0 2 1 3 1 4 1 5 1 6 1 1 0 ];

  %  => not required as if at most one individual is processed
  % Two loops @ B1 with some combinations @ B2, B3, B4, B5, B6
%   [ 1 0 2 0 3 0 4 0 5 0 6 0 1 1 2 0 3 0 4 0 5 0  6 0 1  1  ];
%   [ 1 0 2 0 3 0 4 0 5 0 6 1 1 1 2 0 3 0 4 0 5 0  6 1 1  1  ];
%   [ 1 0 2 0 3 0 4 0 5 1 6 0 1 1 2 0 3 0 4 0 5 1  6 0 1  1  ];
%   [ 1 0 2 0 3 0 4 0 5 1 6 1 1 1 2 0 3 0 4 0 5 1  6 1 1  1  ];
%   [ 1 0 2 0 3 0 4 1 5 0 6 0 1 1 2 0 3 0 4 1 5 0  6 0 1  1  ];
%   [ 1 0 2 0 3 0 4 1 5 0 6 1 1 1 2 0 3 0 4 1 5 0  6 1 1  1  ];
%   [ 1 0 2 0 3 0 4 1 5 1 6 0 1 1 2 0 3 0 4 1 5 1  6 0 1  1  ];
%   [ 1 0 2 0 3 0 4 1 5 1 6 1 1 1 2 0 3 0 4 1 5 1  6 1 1  1  ];
%   [ 1 0 2 0 3 1 4 0 5 0 6 0 1 1 2 0 3 1 4 0 5 0  6 0 1  1  ];
%   [ 1 0 2 0 3 1 4 0 5 0 6 1 1 1 2 0 3 1 4 0 5 0  6 1 1  1  ];
%   [ 1 0 2 0 3 1 4 0 5 1 6 0 1 1 2 0 3 1 4 0 5 1  6 0 1  1  ];
%   [ 1 0 2 0 3 1 4 0 5 1 6 1 1 1 2 0 3 1 4 0 5 1  6 1 1  1  ];
%   [ 1 0 2 0 3 1 4 1 5 0 6 0 1 1 2 0 3 1 4 1 5 0  6 0 1  1  ];
%   [ 1 0 2 0 3 1 4 1 5 0 6 1 1 1 2 0 3 1 4 1 5 0  6 1 1  1  ];
%   [ 1 0 2 0 3 1 4 1 5 1 6 0 1 1 2 0 3 1 4 1 5 1  6 0 1  1  ];
%   [ 1 0 2 0 3 1 4 1 5 1 6 1 1 1 2 0 3 1 4 1 5 1  6 1 1  1  ];
%   [ 1 0 2 1 3 0 4 0 5 0 6 0 1 1 2 1 3 0 4 0 5 0  6 0 1  1  ];
%   [ 1 0 2 1 3 0 4 0 5 0 6 1 1 1 2 1 3 0 4 0 5 0  6 1 1  1  ];
%   [ 1 0 2 1 3 0 4 0 5 1 6 0 1 1 2 1 3 0 4 0 5 1  6 0 1  1  ];
%   [ 1 0 2 1 3 0 4 0 5 1 6 1 1 1 2 1 3 0 4 0 5 1  6 1 1  1  ];
%   [ 1 0 2 1 3 0 4 1 5 0 6 0 1 1 2 1 3 0 4 1 5 0  6 0 1  1  ];
%   [ 1 0 2 1 3 0 4 1 5 0 6 1 1 1 2 1 3 0 4 1 5 0  6 1 1  1  ];
%   [ 1 0 2 1 3 0 4 1 5 1 6 0 1 1 2 1 3 0 4 1 5 1  6 0 1  1  ];
%   [ 1 0 2 1 3 0 4 1 5 1 6 1 1 1 2 1 3 0 4 1 5 1  6 1 1  1  ];
%   [ 1 0 2 1 3 1 4 0 5 0 6 0 1 1 2 1 3 1 4 0 5 0  6 0 1  1  ];
%   [ 1 0 2 1 3 1 4 0 5 0 6 1 1 1 2 1 3 1 4 0 5 0  6 1 1  1  ];
%   [ 1 0 2 1 3 1 4 0 5 1 6 0 1 1 2 1 3 1 4 0 5 1  6 0 1  1  ];
%   [ 1 0 2 1 3 1 4 0 5 1 6 1 1 1 2 1 3 1 4 0 5 1  6 1 1  1  ];
%   [ 1 0 2 1 3 1 4 1 5 0 6 0 1 1 2 1 3 1 4 1 5 0  6 0 1  1  ];
%   [ 1 0 2 1 3 1 4 1 5 0 6 1 1 1 2 1 3 1 4 1 5 0  6 1 1  1  ];
%   [ 1 0 2 1 3 1 4 1 5 1 6 0 1 1 2 1 3 1 4 1 5 1  6 0 1  1  ];
%   [ 1 0 2 1 3 1 4 1 5 1 6 1 1 1 2 1 3 1 4 1 5 1  6 1 1  1  ];

% % Input-based target paths: feasible paths
% % 5-number input: [0 0 0 0 0]
%   [1 0 2 1 3 1 4 1 5 0 6 0 1 0];
%
% % [0 0 0 0 1]
%   [1 0 2 1 3 1 4 1 5 0 6 1 1 0];
%
% % [0 0 0 1 0]
%   [1 0 2 1 3 1 4 1 5 1 6 0 1 0];
%
% % [0 0 1 0 0], [0 1 0 0 0]
%   [1 0 2 1 3 1 4 1 5 0 6 0 1 0];
%
```

```
% % [1 0 0 0 0]
%   [1 0 2 0 3 1 4 1 5 0 6 0 1 0];
%
% % [0 0 0 1 1]
%   [1 0 2 1 3 1 4 1 5 1 6 1 1 0];
%
% % [0 0 1 1 0]
%   [1 0 2 1 3 1 4 1 5 1 6 0 1 0];
%
% % [0 1 1 0 0]
%   [1 0 2 1 3 1 4 1 5 0 6 0 1 0];
%
% % [1 1 0 0 0]
%   [1 0 2 0 3 1 4 1 5 0 6 0 1 0];
%
% % [0 0 1 1 1]
%   [1 0 2 1 3 1 4 1 5 1 6 1 1 0];
%
% % [0 1 1 1 0]
%   [1 0 2 1 3 1 4 1 5 1 6 0 1 0];
%
% % [1 1 1 0 0]
%   [1 0 2 0 3 1 4 1 5 0 6 0 1 0];
%
% % [0 1 1 1 1]
%
% % [1 1 1 1 0]
%
% % [1 1 1 1 1]
```

The fitness function of sG2011

```
function branchVal = fitnessSpaceGong2011(branchNo, predicate)
    k = 1; % the smallest step for integer
    switch (branchNo)
      case 1,
          % Branch #1: for i=1:px ==> i <= px
          branchVal = predicate(1) - predicate(2);
      case {2, 3, 4},
          % Branch #2: if unit1 == 1
          % Branch #3: if unit2 == 2
          % Branch #4: if unit3 == 3
          if predicate(1) == predicate(2)
            branchVal = 0;
          else
            branchVal = k;
          end
      case {5, 6},
          % Branch #5: if error1 == 0
          % Branch #6: if error2 == 0
```

```
            if predicate(1) == 0
              branchVal = 0;
            else
              branchVal = k;
            end
        end
    end
```

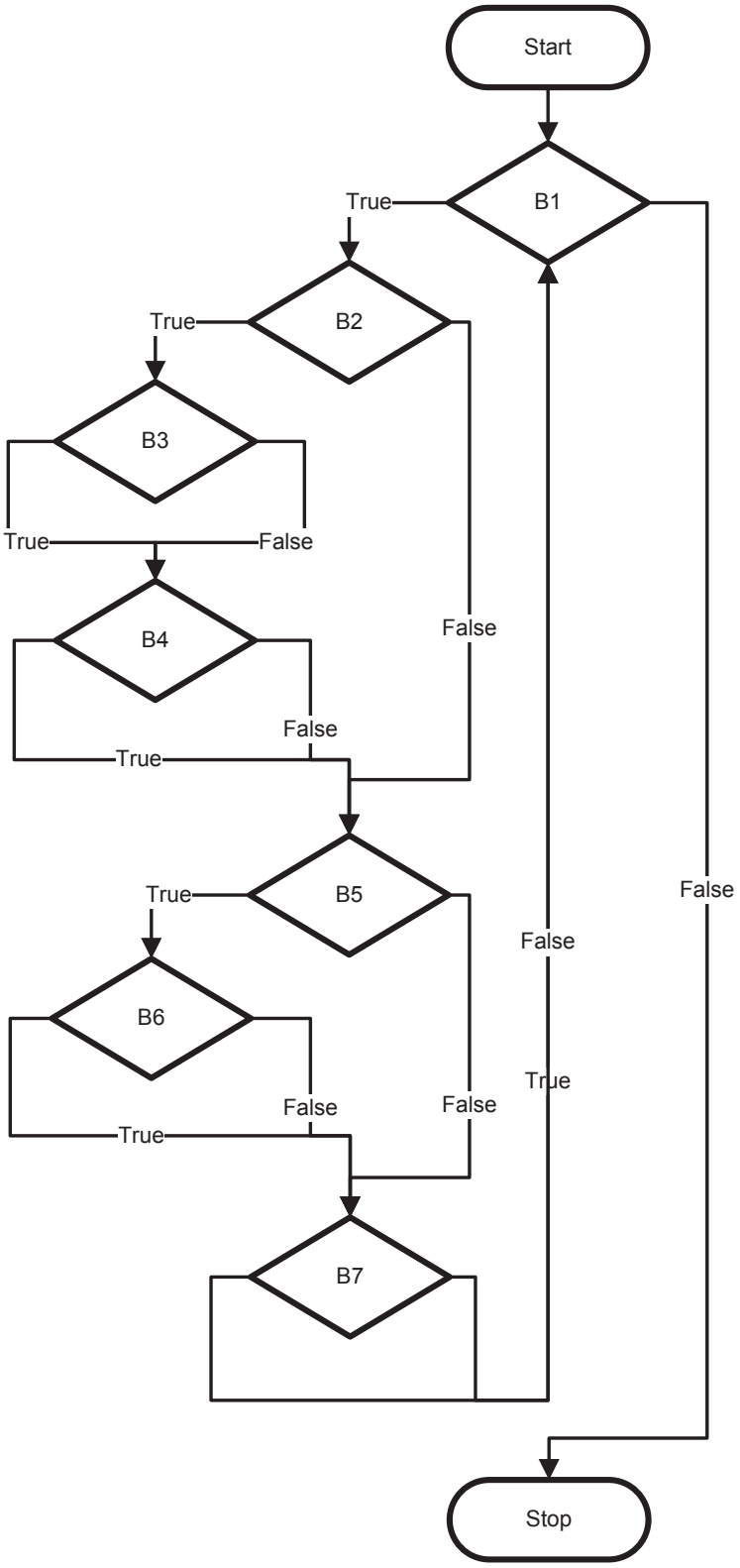


Figure B.18: CFG of fG2011

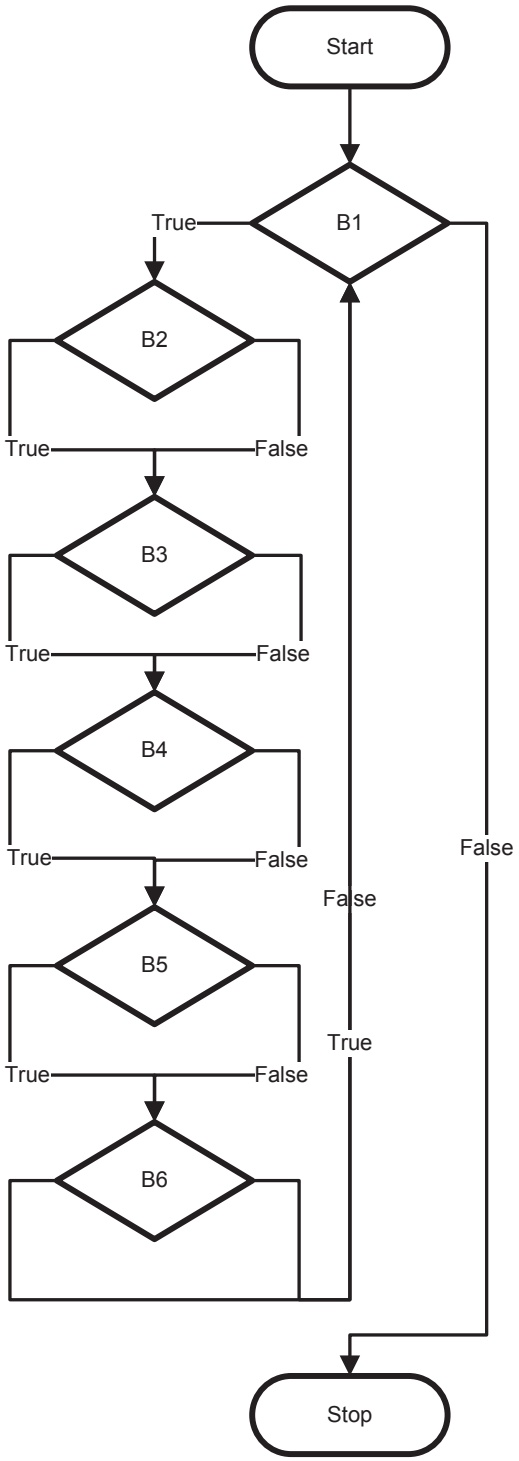


Figure B.19: CFG of sG2011

Appendix C

Test Generation System

Figure C.1 is the legend for the flow chart symbols used in the rest of the thesis.

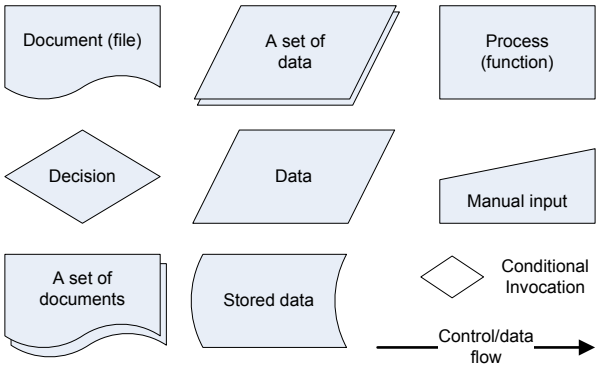


Figure C.1: Flow Chart Symbols

C.1 Testing Manual

In order to use the proposed approach properly, a testing manual has been developed. It consists of two major parts, i.e. testing guide and operating instruction.

C.1.1 Testing Guide

Figure C.2 shows step by step guide for conducting path testing. Mainly, there are two things to do prior calling TDGOI 1.0, i.e. setting up parameter values and determining stopping condition of the test data generator (TDG).

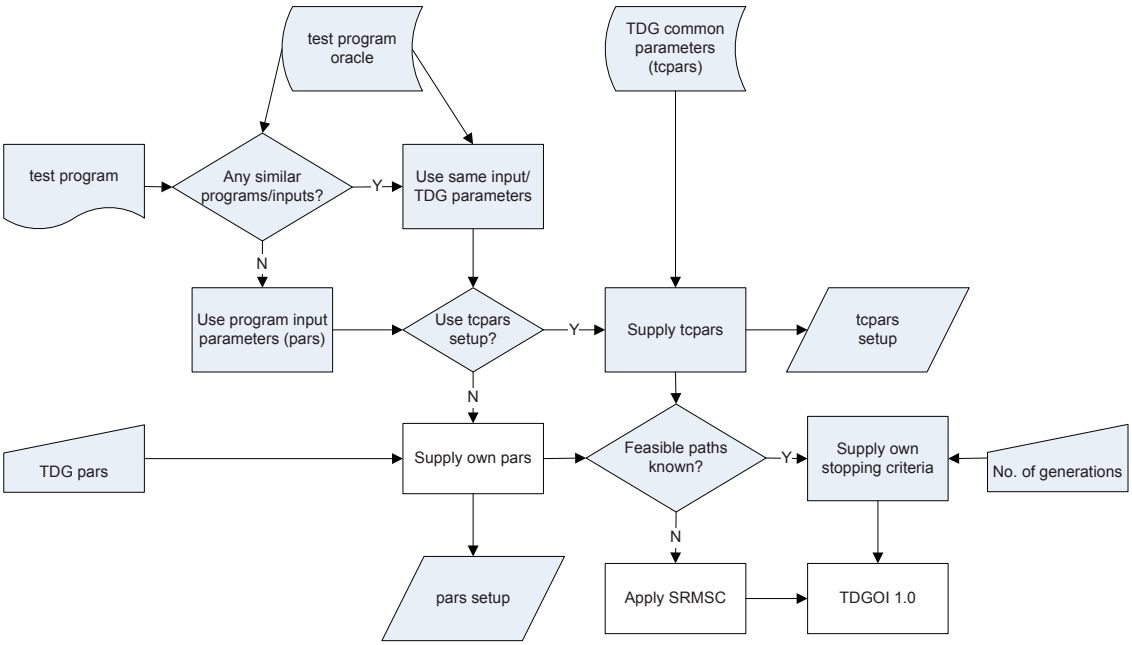


Figure C.2: Testing Guide 1.0

Basically, there are two main steps in Testing Guide 1.0. First is to find similar program in the test program oracle with the current test program, in terms of logical structure and/or input type/criteria taken. If more than one oracle test programs are similar then choose the most similar one and use the same parameters setup and/or input type/criteria out of if for that particular program under test. The parameters setup and input type/criteria used by the oracle test programs have been proven to be the optimal one over certain number of feasible parameters values and/or input type/criteria.

Second is to encourage dynamic stopping criteria if feasible paths are

not known yet, otherwise number of generations will be supplied by the user. This is crucial, because mistakenly choosing inappropriate criteria could lead to missing (some) feasible paths and/or incorrectly identifying feasible paths as infeasible ones. The dynamic stopping criteria employed makes use of software reliability model (SRM) growth analogy. SRM is to monitor the performance of TDG while it is in progress. It is called SRMSC, which stands for SRM for Stopping Criteria.

C.1.2 Operating Instruction

Having decided the parameters setup and stopping criteria in the previous Sub Chapter C.1.1, the next step is TDG operating instruction v1.0 (TDGOI 1.0), which is depicted in Figure C.3.

TDGOI 1.0 has two main roles, i.e. to control the operations of TDG 2.0 and SR 2.0. TDG 2.0 is a representation of the main part of path testing, i.e. generating test data. SR 2.0 is to summarize one of the fundamental output files produced by TDG 2.0 for further analysis.

The operating instruction shows that there are three pre-processes before calling test data generator TDG 2.0. The processes are instrumentation, target paths generation, and fitness function construction. Instrumentation is to probe a test program such that it is able to monitor logical path exercised by certain test data. Its output is instrumented test program. Target path generation is to enumerate all the required and existed logical paths in a test program. It gathers all the target paths in a target list. Fitness function construction is to formulate a function to evaluate a test data for a particular

Figure C.3: TDG Operating Instruction 1.0

test program. It evaluates the closeness of a path taken by test data with a target path in guiding the search conducted by TDG.

TDG 2.0 produces five active text based output files:

1. Summary run-wise, which contains basic information and grouped by run-wise. The information are date, time, folder, file name, GA parameters setup, TDG parameters setup, initial target paths, target paths along its satisfying test (input) data as covered in progress, population

(set of test data) every generation, population fitness (set of test data fitness) every generation, and the best fitness of each generation.
File name format example, 1_triangle_1_2_1_-1_100_100_0.9_0.9_0.1.txt

2. bestFitness run-wise, which contains number of paths found on every finding generation, the best fitness and input data every generation, and grouped by run-wise.
File name format example, 1_triangle_1_2_1_-1_100_100_0.9_0.9_0.1_BestFitness.xls

3. traceFile run-wise, which records the evolution of population after every evolutionary operation, e.g. just after selection, crossover, mutation, and insertion, and grouped by run-wise.
File name format example, 1_triangle_1_2_1_-1_100_100_0.9_0.9_0.1_trace.txt

4. Summary of a set of run-wise, which writes generation-to-generation progress, i.e. number of paths found on every finding generation.
File name format example, Summary_G2G_triangle_30_20111102T112957.xls

5. Summary of a set of benchmark-wise, which summarizes Point 4 over several benchmarks, e.g. number of paths found in every run, descriptive statistic over all runs.
File name format example, Summary_G2G_All__20111102T112956.txt

The fifth output file (Point 5) is subject to further analysis and an input to SR 2.0. The outputs of SR 2.0 are five text based files in comma separated values (CSV) format:

1. _comp.xls, which compares performance between GA and its variants -based TDG

2. _comp_exec.xls, which compares performance between GA and its variants -based TDG, and provides comparison measures between GA and its variants -based TDG, i.e. normalized number of paths found over total paths for each test program (N-pf), number of paths found (pf), and elapsed time (et)

3. .xls, which converts the fifth output file of TDG 2.0 (Point 5 of TDG 2.0 output file) into excel format, i.e .xls

4. _comp_exec_gen.xls, which is similar output with Point 2, but it is grouped by generation-wise

5. _comp_exec_lsv.xls, which is similar output with Point 2 with additional comparison with variable local search

C.2 Path Testing Architecture

The architecture of path testing comprises of four major components: TDG, SRMSC, benchmark generator (BG), and summary reader (SR).

C.2.1 Test Data Generator

Figure C.4 depicts the inside of TDG, which is currently on version 2 (TDG 2.0). It has four sub components: dagerFetcher (dF), six versions of TDG, (abstract) GA, and local, which is the local search.

dF acts as a feeder to all the versions of TDG. It takes input dF_feeder, which contains a set of benchmarks. Each benchmark consists of a test

program, its target paths, and its parameters setup.

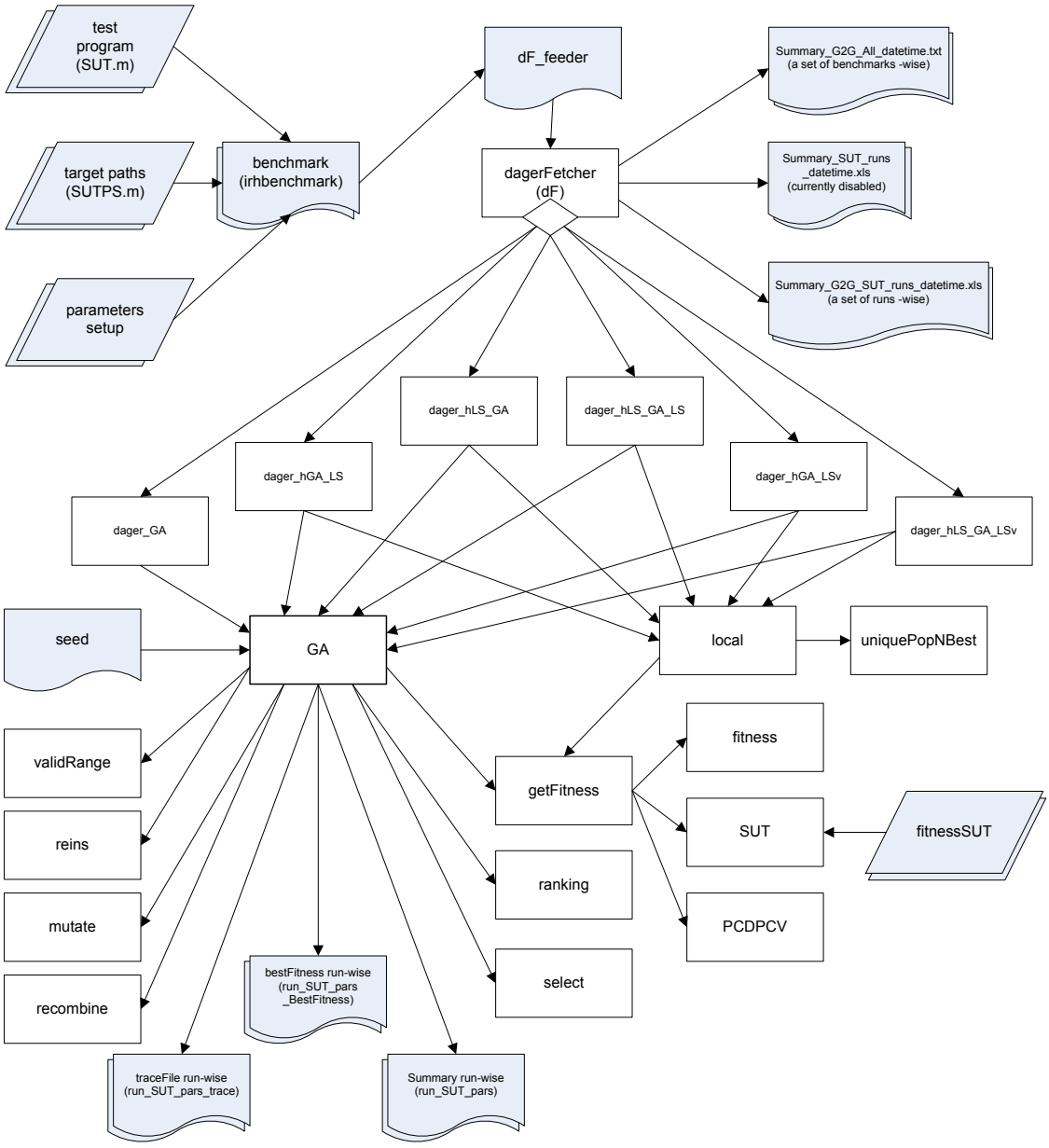


Figure C.4: Test Data Generator 2.0

Six versions of TDG are

1. dager_GA, GA based TDG

2. dager_hGA_LS, hybrid TDG using GA and local search (LS)

3. dager_hLS_GA, hybrid TDG using LS and GA
4. dager_hLS_GA_LS, hybrid TDG using LS, GA, and LS
5. dager_hGA_LSv, hybrid TDG using GA and variable-size LS (LSv)
6. dager_hLS_GA_LSv, hybrid TDG using LS, GA, and LSv

GA has the following seven sub components:

1. getFitness, a function to evaluate fitness function, which comprises of another three sub components: fitness, SUT, and PCDCV.
   fitness is a set of Korel's fitness functions. SUT is to call a given (instrumented) test program and to evaluate a fed test data according to Korel's fitness function. It also requires fitnessSUT as its input to match which fitness function need to be used at which branch (or predicate). PCDCV is a function to compute branch distance and approximation level.
2. ranking, a function to rank all the chromosomes in population based on their fitness values.
3. select, a selection function that selects which chromosomes are chosen to participate and/or survive in the next generation.
4. recombine, a function that mates two selected parent chromosomes.
5. mutate, a function that makes slight perturbation to one or more sub components of a chromosome.
6. reins, a function to insert all newly generated chromosomes into the next population.

7. validRange, a function to check whether a newly generated chromosome is in valid form of a test data or not. If it is invalid then it maps the invalid one to a valid one.

C.2.2 SRM for Stopping Criteria

SRMSC is a function to simulate the application of SRM growth as one of the stopping criteria for TDG 2.0. Figure C.5 shows the architecture of SRMSC.

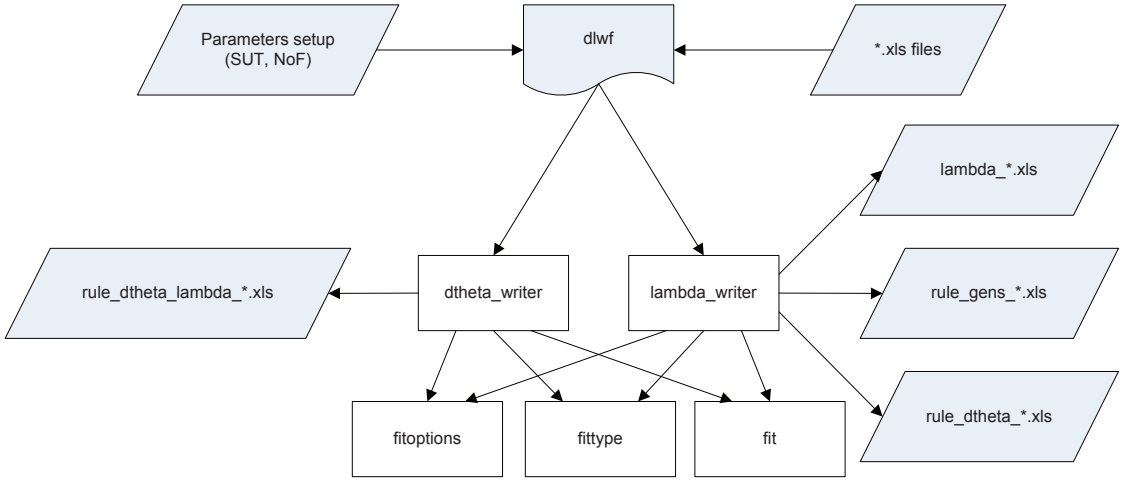


Figure C.5: SRMSC

SRMC has the following main functions: dtheta_writer, lambda_writer, fitoptions, fittype, and fit. While the first two functions are defined in details in Sub Chapter 6.2.2, the rest are parts of Curve Fitting Matlab Toolbox.

Function dtheta_writer is to simulate Stability Rule by computing a set of θ values across several generations. Then the set is scanned to monitor whether any changes in between two adjacent values ($\Delta\theta$) is less than certain threshold. If it does then the search will stop and report its performance in terms of number of paths found and time elapsed at the stopping point.

Function lambda_writer is a simulation function for Reliability Rule that generates a set of λ values along some generations. It also monitors the set to find any changes between two respective values ($\Delta\lambda$) that is less than pre-defined limit. It will stop the search as soon as the limit is reached and report its performance in the forms of number of paths found and time elapsed at that time.

SRMC takes an input and yields four output files. Its input consists of parameters setup and excel files. These files are the fifth output file of TDG 2.0, which is a summary of a set of benchmark-wise (See Chapter C.1.2).

The outputs are

1. lambda_*.xls, which contains three set of λ values and their generation to generation achievements, i.e. number of paths found. This is to monitor their performances in order to find out which λ value achieves the most reasonable number of paths found over certain number of generations.

2. rule_gens_*.xls, which lists combinations of some $\Delta\theta$ values and some $\Delta\lambda$ values after every certain number of generations. This is to seek the most reasonable combination that balances number of paths found over number of generations.

3. rule_dtheta_*.xls, which contains list of $\Delta\theta$ over several generations.

4. rule_dtheta_lambda_*.xls, which has similar contents with Point 2, but it only lists certain combinations of $\Delta\theta$ values, i.e. 0.001 ($\frac{1}{1000}$), 0.0005 ($\frac{1}{2000}$), 0.00033 ($\frac{1}{3000}$), 0.00025 ($\frac{1}{4000}$), 0.0002 ($\frac{1}{5000}$), and 0.0001 ($\frac{1}{10000}$), and $\Delta\lambda$ values, i.e. 0.1 and 0.25.

C.2.3 Benchmark Generator

Benchmark Generator 1.5, which is shown in C.6 has ability to generate setup (preparation) file for running several test programs. It makes parameters setup as its input and produces a Matlab (text) file with .m extension.

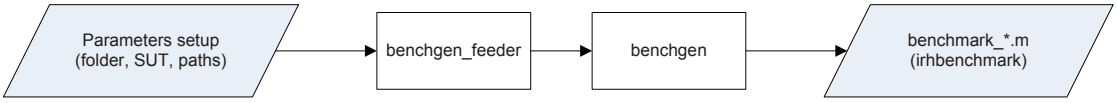


Figure C.6: Benchmark Generator 1.5

The parameters setup consists of folder name to store the output file, test program, target paths, set of number of generations, set of population sizes, input (or chromosome) length, values range for each allele, mutation rate, and type of GA. The output file will enumerate each combination of parameter values whose values range are defined in the input file.

C.2.4 Summary Reader

The following Figure C.7 shows five functions to further summarize and analyze the output of TDG 2.0. Each of these functions serves different analysis.

The purpose for each function is

1. summary_reader, a function to convert descriptive (textual) summary into tabular form in CSV format.

2. summary_reader_comp, a similar function with summary_reader with additional feature that it can compare different approaches, which are applied to the same test program.

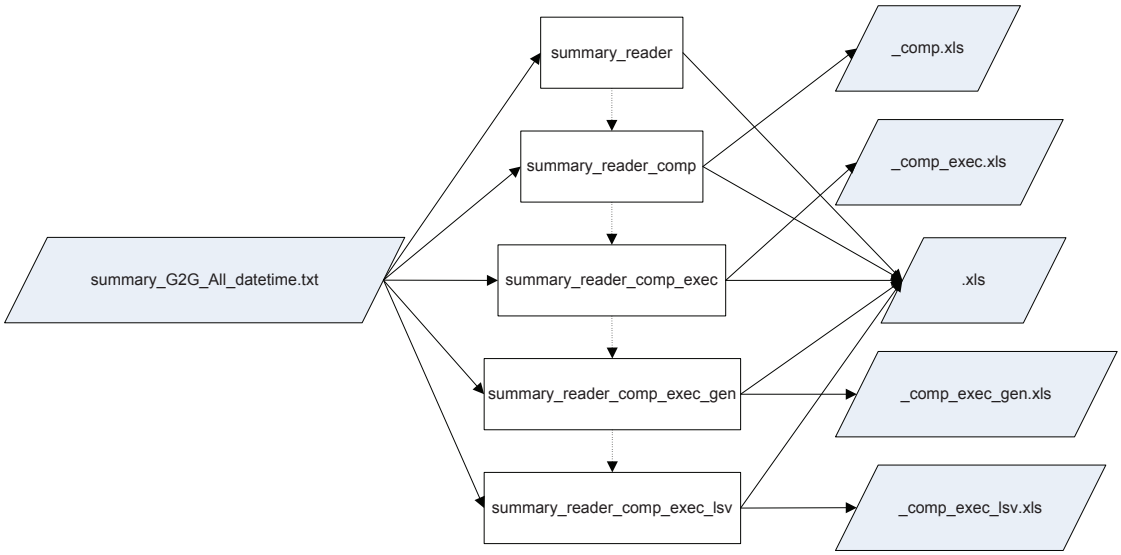


Figure C.7: Summary Reader 2.0

3. summary_reader_comp_exec, a similar function with summary_reader_comp with additional comparative executive summary across certain number different approaches over the same test program in terms of number of paths found and elapsed time.

4. summary_reader_comp_exec_gen, a similar function with summary_reader_comp_exec that can accept any number of different approaches under generic column names.

5. summary_reader_comp_exec_lsv, a similar function with summary_reader_comp_exec that is able to compare with hybrid GA with variable size local search (LSv).

References

[1] John E. Bentley. Software Testing Fundamental – Concepts, Roles, and Terminology. In *Proceedings of the SAS Users Group International 30*, pages 141–30, April 2005.

[2] Glenford J. Myers. *The Art of Software Testing.* World Association Inc., 2004.

[3] Ian Sommerville. *Software Engineering.* Addison-Wesley, 9^{th} edition, 2011.

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

[5] Jin-Cherng Lin and Pu-Lin Yeh. Using genetic algorithms for test case generation in path testing. In *Proceedings of the* 9^{th} *Asian Test Symposium 2000 (ATS '00)*, pages 241–246, December 2000.

[6] Dick Hamlet. Foundations of software testing: Dependability theory. *SIGSOFT Software Engineering Notes*, 19:128–139, December 1994.

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

[8] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[9] Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the* 2^{nd} *Conference on Computer Science and Engineering in Linköping*, pages 21–28. ECSEL, October 1999.

[10] Joachim Wegener, André Baresel, and Harmen Sthamer. Suitability of evolutionary algorithms for evolutionary testing. In *Proceedings of the* 26^{th} *Annual International Computer Software and Applications Conference 2002 (COMPSAC 2002)*, pages 287–289, 2002.

[11] Joachim Wegener, Kerstin Buhr, and Hartmut Pohlheim. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 1233–1240, San Francisco, CA, USA, jul 2002. Morgan Kaufmann Publishers Inc.

[12] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, April 2009.

[13] Min Pei, Erik D. Goodman, Zongyi Gao, and Kaixiang Zhong. Automated software test data generation using a genetic algorithm. Technical report, Michigan State University, June 1994.

[14] Paulo Marcos Siqueira Bueno and Mario Jino. Identification of potentially infeasible program paths by monitoring the search for test data. In *Proceedings of the* 15^{th} *IEEE International Conference on Automated Software Engineering 2000 (ASE 2000)*, pages 209–218, Grenoble, France, 2000. IEEE Computer Society.

[15] Paulo Marcos Siqueira Bueno and Mario Jino. Automatic test data generation for program paths using genetic algorithms. In *Proceedings of the* 13^{th} *International Conference on Software Engineering & Knowledge Engineering (SEKE '01)*, pages 2–9, Buenos Aires, Argentina, 2001.

[16] Paulo Marcos Siqueira Bueno and Mario Jino. Automatic test data generation for program paths using genetic algorithms. *International Journal of Software Engineering & Knowledge Engineering (IJSEKE)*, 12(6):691–709, 2002.

[17] Irman Hermadi and Moataz A. Ahmed. Genetic Algorithm based test data generator. In *Proceedings of the 2003 Congress on Evolutionary Computation (CEC)*, volume 1, pages 85–91, December 2003.

[18] Nashat Mansour and Miran Salame. Data generation for path testing. *Software Quality Control*, 12(2):121–136, 2004.

[19] Moheb R. Girgis. Automatic test data generation for data flow testing using a genetic algorithm. *Journal of Universal Computer Science*, 11(6):898–915, 2005.

[20] Moataz A. Ahmed and Irman Hermadi. GA-based Multiple Paths Test Data Generator. *Computers & Operations Research*, 35:3107–3124, October 2008.

[21] Yong Chen and Yong Zhong. Automatic path-oriented test data generation using a multi-population genetic algorithm. In *Proceedings of the* 4^{th} *International Conference on Natural Computation, 2008 (ICNC '08)*, volume 1, pages 566–570, October 2008.

[22] Raluca Lefticaru and Florentin Ipate. Functional search-based testing from state machines. In *Proceedings of the* 1^{st} *International Conference on Software Testing, Verification, and Validation 2008*, pages 525–528, April 2008.

[23] Irman Hermadi, Chris Lokan, and Ruhul Sarker. Genetic algorithm based path testing: Challenges and key parameters. *World Congress on Software Engineering*, 2:241–244, 2010.

[24] Kewen Li, Zilu Zhang, and Jisong Kou. Breeding software test data with genetic-particle swarm mixed algorithm. *Journal of Computers*, pages 258–265, February 2010.

[25] Dunwei Gong, Wanqiu Zhang, and Xiangjuan Yao. Evolutionary generation of test data for many paths coverage based on grouping. *System Software*, 84(12):2222–2233, December 2011.

[26] Enrique Alba and Francisco Chicano. Observations in using parallel and sequential evolutionary algorithms for automatic software testing. *Computers & Operations Research*, 35:3161–3183, February 2007.

[27] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2010.

[28] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7:113–128, September 1975.

[29] G.M. Weinberg. *The Psychology of Computer Programming*. Dorset House Pub., 1998.

[30] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29:366–427, December 1997.

[31] Mary Lou Soffa, Aditya P. Mathur, and Neelam Gupta. Generating test data for branch coverage. In *Proceedings of the* 15^{th} *IEEE international conference on Automated software engineering*, ASE '00, pages 219–, Washington, DC, USA, 2000. IEEE Computer Society.

[32] A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction procedure for test data generation. *Software Practice and Experience*, 29:167–193, February 1999.

[33] Nigel Tracey, John Clark, Keith Mander, and John Mcdermid. An automated framework for structural test-data generation. In *Proceedings of the* 13^{th} *IEEE Conference on Automated Software Engineering*, pages 285–288, 1998.

[34] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. *SIGSOFT Software Engineering Notes*, 23(6):231–244, 1998.

[35] Nigel J. Tracey, John A Clark, and Keith C Mander. 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.

[36] James A. Whittaker. What is software testing? and why is it so hard? *IEEE Software*, 17:70–79, 2000.

[37] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification And Reliability*, 9(4):263–282, 1999.

[38] Huey-Der Chu. An evaluation scheme of software testing techniques. In *IFIP TC5 WG5.4* 3^{rd} *Internatinal Conference on Reliability, Quality and Safety of software-intensive systems*, pages 259–262, London, UK, UK, 1997. Chapman & Hall, Ltd.

[39] Marc Roper, Iain Maclean, Andrew Brooks, James Miller, and Murray Wood. Genetic algorithms and the automatic generation of test data. Technical report, Semin. Arthr. Rheum, 1995.

[40] Christoph C. Michael, Gary E. McGraw, Michael A. Schatz, and Curtis C. Walton. Genetic algorithms for dynamic test data generation. In *Proceedings of the* 12^{th} *international conference on Automated software engineering (formerly: KBSE)*, ASE '97, pages 307–308, Washington, DC, USA, 1997. IEEE Computer Society.

[41] Christoph C. Michael, Gary E. McGraw, and Michael A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[42] Melanie Mitchell. *An Introduction to Genetic Algorithms.* The MIT Press, February 1998.

[43] John H. Holland. *Adaptation in Natural and Artificial Systems.* The MIT Press, Cambridge, MA, USA, 1975.

[44] Lawrence Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[45] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1^{st} edition, 1989.

[46] Harmen-Hinrich Sthamer. *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD Dissertation, University of Glamorgan, November 1995.

[47] R. Feldt and S. Poulding. Finding test data with specific properties via metaheuristic search. In *Software Reliability Engineering (ISSRE), 2013 IEEE* 24^{th} *International Symposium on*, pages 350–359, Nov 2013.

[48] Mark Harman, Lin Hu, Robert M. Hierons, André Baresel, and Harmen Sthamer. Improving evolutionary testing by flag removal. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 1359–1366, San Francisco, CA, USA, July 2002. Morgan Kaufmann Publishers Inc.

[49] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, NY, 2^{nd} edition, 1994.

[50] Bryan F. Jones, Harmen-Hinrich Sthamer, and D.E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering*, 11(5):299–306, September 1996.

[51] Gary McGraw, Christoph Michael, and Michael Schatz. Generating software test data by evolution. Technical report, Reliable Software Technologies, Sterling, VA, February 1998.

[52] Johann Dréo, Alain Pétrowski, Patrick Siarry, and Eric Taillard. *Metaheuristics for Hard Optimization: Methods and Case Studies*. Springer, Germany, 2006.

[53] Sadiq M. Sait and Habib Youssef. *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1^{st} edition, 1999.

[54] Hyunchul Kim, Y. Hayashi, and K. Nara. The performance of hybridized algorithm of ga, sa, and ts for thermal unit maintenance scheduling. In *IEEE 1995 International Conference on Evolutionary Computation*, volume 1, page 114, nov-1 dec 1995.

[55] Harmen Hinrich Sthamer, Joachim Wegener, and André Baresel. Using evolutionary testing to improve efficiency and quality in software testing. In *In Proceedings of the* 2^{nd} *Asia-Pacific Conference on Software Testing Analysis and Review (AsiaSTAR)*, Melbourne, AU, July 2002.

[56] Irman Hermadi. Genetic Algorithm based Test Data Generator. Master's thesis, King Fahd University of Petroleum & Minerals (KFUPM), June 2004.

[57] Sagar Naik and Piyu Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. Wiley, August 2008. hardcover; 648 pages.

[58] John D. Musa and K. Okumoto. A logarithmic poisson execution time model for software reliability measurement. In *Proceedings of the* 7^{th} *International Conference on Software Engineering*, ICSE '84, pages 230–238, Piscataway, NJ, USA, 1984. IEEE Press.

[59] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833 – 839, 2001.

[60] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226 –247, March 2010.

[61] Joachim Wegener, Andre Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841 – 854, 2001. Special Issue on Software Engineering using Metaheuristic Innovative Algorithms.

[62] André Baresel, Harmen Sthamer, and Michael Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO '02: Proceedings of the 2002 Genetic and Evolutionary Computation Conference*, pages 1329–1336, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[63] Raquel Blanco, Javier Tuya, and Belarmino Adenso-Diaz. Automated test data generation using a scatter search approach. *Information and Software Technology*, 51(4):708–720, April 2009.

[64] Ramón Sagarna and Xin Yao. Handling constraints for search based software test data generation. In *ICSTW '08: Proceedings of the IEEE International Conference on Software Testing Verification and Validation 2008*, pages 232–240, April 2008.

[65] Mark Harman, Kiran Lakhotia, and Phil McMinn. A multi-objective approach to search-based test data generation. In *GECCO '07: Proceedings of the 2007 Genetic and Evolutionary Computation Conference*, pages 1098–1105, New York, NY, USA, July 2007. ACM.

[66] Phil McMinn. IGUANA: Input Generation Using Automated Novel Algorithms. A plug and play research tool. Technical report, University of Sheffield, 2007.

[67] Kalyanmoy Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[68] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *ISSTA '07: Proceedings of the 2007 international symposium on software testing and analysis*, pages 140–150, New York, NY, USA, 2007. ACM.

[69] Luciano Petinati Ferreira and Silvia Regina Vergilio. Tdsgen: An environment based on hybrid genetic algorithms for generation of test data. In *Genetic and Evolutionary Computation GECCO 2004*, volume 3103 of *Lecture Notes in Computer Science*, pages 1431–1432. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-24855-2-165.

[70] Phil McMinn and Mike Holcombe. Hybridizing evolutionary testing with the chaining approach. In *GECCO 2004: Proceedings of the Genetic and Evolutionary Computation Conference 2004, Lecture Notes in Computer Science*, pages 1363–1374. Springer Verlag, 2004.

[71] Phil McMinn and Mike Holcombe. Evolutionary testing of state-based programs. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1013–1020, New York, NY, USA, 2005. ACM.

[72] André Baresel, Hartmut Pohlheim, and Sadegh Sadeghipour. Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. In *Proceedings of the 2003 international conference on Genetic and evolutionary computation: PartII*, GECCO'03, pages 2428–2441, Berlin, Heidelberg, 2003. Springer-Verlag.

[73] Phil McMinn and Mike Holcombe. The state problem for evolutionary testing. In *Proceedings of the 2003 international conference on Genetic and evolutionary computation: PartII*, GECCO'03, pages 2488–2498, Berlin, Heidelberg, 2003. Springer-Verlag.

[74] Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[75] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transaction on Software Engineering Methodology*, 5(1):63–86, 1996.

[76] Roger Ferguson and Bogdan Korel. Generating test data for distributed software using the chaining approach. *Information and Software Technology*, 38(5):343–353, May 1996.

[77] Bogdan Korel. Automated test data generation for programs with procedures. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 209–215, New York, NY, USA, 1996. ACM.

[78] Stefan Wappler and Joachim Wegener. Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 06)*, pages 851–858, Vancouver, BC, Canada, 2006. IEEE.

[79] Anastasis A. Sofokleous and Andreas S. Andreou. Batch-optimistic test-cases generation using genetic algorithms. In *Proceedings of the* 19^{th} *IEEE International Conference on Tools with Artificial Intelligence, 2007 (ICTAI 2007)*, volume 1, pages 157–164. IEEE, 2007.

[80] Andrea Arcuri and Xin Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008.

[81] Bryan F. Jones, Harmen-Hinrich Sthamer, and D.E. Eyres. Generating test data for ada procedures using genetic algorithms. In *Proceedings of the* 1^{st} *International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA 95)*, pages 65–70, London, UK., 1995. IEEE.

[82] Bryan F. Jones, Harmen-Hinrich Sthamer, X. Yang, and D.E. Eyres. The automatic generation of software test data sets using adaptive search techniques. *Transactions on Information and Communications Technologies: Software Quality Management*, 11:435–444, 1995.

[83] Marc Roper. Computer aided software testing using genetic algorithms. In *Proceedings of the* 10^{th} *International Software Quality Week*, page 9T1117, San Francisco, California, USA, 1997. Software Research Institute.

[84] Bryan F. Jones, D. E. Eyres, and Harmen-Hinrich Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing. *Computer Journal*, 41(2):98–107, 1998.

[85] Jin-Cherng Lin and Pu-Lin Yeh. Automatic test data generation for path testing using gas. *Information Sciences*, 131(1-4):47 – 64, 2001.

[86] Eugenia Díaz, Javier Tuya, and Raquel Blanco. Automated software testing using a metaheuristic technique based on tabu search. In *Proceedings of the* 18^{th} *IEEE International Conference on Automated Software Engineering, 2003*, pages 310–313, Montreal, Canada, October 2003. IEEE.

[87] Robert M. Hierons, Mark Harman, and Chris Fox. Branch-coverage testability transformation for unstructured programs. *Computer Journal*, 48(4):421–436, 2005.

[88] Bogdan Korel, Mark Harman, S. Chung, P. Apirukvorapinit, R. Gupta, and Q. Zhang. Data dependence based testability transformation in

automated test generation. In *Proceedings of the* 16^{th} *IEEE International Symposium on Software Reliability Engineering (ISSRE 05)*, pages 245–254, Chicago, Illinios, USA, 2005. IEEE Computer Society.

[89] Xiyang Liu, Bin Wang, and Hehui Liu. Evolutionary search in the context of object-oriented programs. In *Proceedings of the* 6^{th} *Metaheuristics International Conference (MIC 05)*, Vienna, Austria, 2005.

[90] Phil McMinn, David Binkley, and Mark Harman. Testability transformation for efficient automated test data search in the presence of nesting. In *Proceedings of the* 3^{rd} *UK Software Testing Workshop*, pages 165–182. ACM, September 2005.

[91] Ramón Sagarna and J. A. Lozano. On the performance of estimation of distribution algorithms applied to software testing. *Applied Artificial Intelligence*, 19(5):457–489, 2005.

[92] Stafen Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO 05)*, pages 1053–1060, Washington, D.C., USA, 2005. ACM.

[93] Xiaoyuan Xie, Baowen Xu, Changhai Nie, Liang Shi, and Lei Xu. Configuration strategies for evolutionary testing. In *Proceedings of the* 29^{th} *Annual International Computer Software and Applications Conference (COMPSAC 05)*, pages 13–14, Edinburgh, UK, 2005. IEEE Computer Society.

[94] Xiaoyuan Xie, Baowen Xu, Liang Shi, Changhai Nie, and Y. He. A dynamic optimization strategy for evolutionary testing. In *Proceedings of the* 12^{th} *Asia-Pacific Software Engineering Conference (APSEC 05)*, pages 568–575, Taipei, Taiwan, 2005. IEEE Computer Society.

[95] Mohammad Alshraideh and Leonardo Bottaci. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability*, 16(3):175203, 2006.

[96] Phil McMinn, Mark Harman, David Binkley, and Paolo Tonella. The species per path approach to search-based software test data generation. In *International Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 13–24. ACM, 17–20 July 2006.

[97] Ramón Sagarna and Jose A. Lozano. Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms. *European Journal of Operational Research*, 169(2):392–412, March 2006.

[98] Arjan Seesing. Evotest: Test case generation using genetic programming and software analysis. Master thesis, Delft University of Technology, Delft, The Netherlands, 2006.

[99] Arjan Seesing and Hans-Gerhard Gro. A genetic programming approach to automated test generation for object-oriented software. *International Transactions on System Science and Applications*, 1(2):127–134, 2006.

[100] Hsiu-Chi Wang. A hybrid genetic algorithm for automatic test data generation. Master thesis, National Sun Yat-sen University, Department of Information Management, Taiwan, China, 2006.

[101] Stefan Wappler and Joachim Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Proceedings of the* 8^{th} *annual Conference on Genetic and Evolutionary Computation (GECCO 06)*, pages 1925–1932, Seattle, Washington, USA, 2006. ACM.

[102] Stella Levin and Amiram Yehudai. Evolutionary testing: A case study. *Lecture Notes in Computer Science including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics*, 4383 LNCS:1–24, 2006.

[103] Raquel Blanco, Javier Tuya, Eugenia Daz, and Belarmino Adenso-Daz. A scatter search approach for automated branch coverage in software testing. *International Journal of Engineering Intelligent Systems (EIS)*, 15(3):135–142, September 2007.

[104] Konstantinos Liaskos, Marc Roper, and Murray Wood. Investigating data-flow coverage of classes using evolutionary algorithms. In *Proceedings of the* 9^{th} *annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1140–1140, New York, NY, USA, 2007. ACM.

[105] Konstantinos Liaskos and Marc Roper. Automatic test-data generation: An immunological approach. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 77–81, September 2007.

[106] Ramón Sagarna, Andreas Arcuri, and Xin Yao;. Estimation of distribution algorithms for testing object oriented software. In *IEEE Congress on Evolutionary Computation 2007*, pages 438 – 444, Singapore, September 2007.

[107] Ramón Sagarna. *An Optimization Approach for Software Test Data Generation*. PhD Thesis, University of the Basque Country, San Sebastian, Spain, 2007.

[108] Nirmal Kumar Gupta and Mukesh Kumar Rohil. Using genetic algorithm for unit testing of object oriented software. In *Proceedings of the 2008 First International Conference on Emerging Trends in Engineering and Technology*, ICETET '08, pages 308–313, Washington, DC, USA, 2008. IEEE Computer Society.

[109] Yan Wang, Zhiwen Bai, Miao Zhang, Wen Du, Ying Qin, and Xiyang Liu. Fitness calculation approach for the switch-case construct in evolutionary testing. In *Proceedings of the* 10^{th} *annual conference on Genetic and evolutionary computation*, GECCO '08, pages 1767–1774, New York, NY, USA, 2008. ACM.

[110] Matthew Charles Makai. Incorporating design knowledge into genetic algorithm-based white-box software test case generators. Master thesis, Virginia Technology, April 2008.

[111] Mark Harman. Testability transformation for search-based testing. In *Keynote of the* 1^{st} *International Workshop on Search-based Software*

Testing (SBST) in conjunction with ICST 2008, Lillehammer, Norway, 2008.

[112] Phil McMinn, David Binkley, and Mark Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering and Methodology*, 18(3):11:1–11:27, June 2009.

[113] Mohammad Alshraideh, Basel A. Mahafzah, and Saleh Al-Sharaeh. A multiple-population genetic algorithm for branch coverage test data generation. *Software Quality Control*, 19(3):489–513, September 2011.

[114] Karin Zielinski and Rainer Laur. Stopping criteria for differential evolution in constrained single-objective optimization. In UdayK. Chakraborty, editor, *Advances in Differential Evolution*, volume 143 of *Studies in Computational Intelligence*, pages 111–138. Springer Berlin Heidelberg, 2008.

[115] Frans Van Den Bergh. *An Analysis of Particle Swarm Optimizers*. PhD thesis, Pretoria, South Africa, South Africa, 2002. AAI0804353.

[116] Haldun Aytug and Gary J. Koehler. Stopping criteria for finite length genetic algorithms. *INFORMS Journal on Computing*, 8(2):183–191, 1996.

[117] David Greenhalgh and Stephen Marshall. Convergence criteria for genetic algorithms. *SIAM J. Comput.*, 30(1):269–282, April 2000.

[118] H. Trautmann, T. Wagner, B. Naujoks, M. Preuss, and J. Mehnen. Statistical methods for convergence detection of multi-objective evolutionary algorithms. *Evol. Comput.*, 17(4):493–509, December 2009.

[119] Marcin Studniarski. Stopping criteria for genetic algorithms with application to multiobjective optimization. In *Proceedings of the* 11^{th} *International Conference on Parallel Problem Solving from Nature: Part I*, PPSN'10, pages 697–706, Berlin, Heidelberg, 2010. Springer-Verlag.

[120] John D. Musa. A theory of software reliability and its application. *IEEE Transactions on Software Engineering*, 1(3):312–327, 1975.

[121] Amrit L. Goel and Kazu Okumoto. Time-dependent error-detection rate model for software reliability and other performance measures. *IEEE Transactions on Reliability*, R-28(3):206 –211, August 1979.

[122] Amrit L. Goel. Software reliability models: Assumptions, limitations, and applicability. *IEEE Trans. Softw. Eng.*, 11:1411–1423, December 1985.

[123] P. Moranda. Predictions of software reliability during debugging. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 327–332, Washington D.C., January 1975.

[124] Norman F. Schneidewind. Analysis of error processes in computer software. *SIGPLAN Notices*, 10(6):337–346, April 1975.

[125] L.H. Crow. Reliability analysis for complex, repairable systems. *Reliability and Biometry*, pages 379–410, 1974. Edited by F. Proshan and R. J. Serfling, SIAM.

[126] Robert Jasper, Mike Brennan, Keith Williamson, Bill Currier, and David Zimmerman. Test data generation and feasible path analysis. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '94, pages 95–107, New York, NY, USA, 1994. ACM.

[127] A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7:165–192, September 1997.

[128] Jan Gustafsson, Andreas Ermedahl, and Bjrn Lisper. Algorithms for infeasible path calculation. In *Proceedings of The* 6^{th} *International Workshop on Worst Case Execution Time Analysis (WCET '06)*, volume 4, 2006.

[129] Amie L. Souter and Lori L. Pollock. Characterization and automatic identification of type infeasible call chains. *Information and Software Technology*, 44(13):721–732, 2002.

[130] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivancic, Ou Wei, and Aarti Gupta. Slr: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In *Static Analysis Symposium/Workshop*, pages 238–254, 2008.

[131] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Using branch correlation to identify infeasible paths for anomaly detection. In *Proceedings of the* 39^{th} *Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 113–122, Washington, DC, USA, 2006. IEEE Computer Society.

[132] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Proceedings of the* 43^{rd} *annual Design Automation Conference*, DAC '06, pages 358–363, New York, NY, USA, 2006. ACM.

[133] Minh Ngoc Ngo and Hee Beng Kuan Tan. Heuristics-based infeasible path detection for dynamic test data generation. *Information and Software Technology*, 50(7-8):641–655, 2008.

[134] Jun Yan and Jian Zhang. An efficient method to generate feasible paths for basis path testing. *Information Processing Letters*, 107(3-4):87–92, 2008.

[135] Lei Ju, Bach Khoa, Huynh Abhik, and Roychoudhury Samarjit Chakraborty. A systematic classification and detection of infeasible paths for accurate wcet analysis of esterel programs. In *Proceedings of the Singaporean-French IPAL Symposium 2009*, February 2009.

[136] M. Delahaye, B. Botella, and A. Gotlieb. Explanation-based generalization of infeasible path. In *Proceedings of the* 3^{rd} *International Conference on Software Testing, Verification and Validation (ICST) 2010*, pages 215–224, April 2010.

[137] C.V. Ramamoorthy, Siu-Bun F. Ho, and W.T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2(4):293–300, December 1976.

[138] A. Jefferson Offutt, Jie Pan, Tong Zhang, and Kanupriya Tewary. Experiments with data flow and mutation testing. Technical Report ISSE-TR-94-105, ISSE, 1994.

[139] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11:367–375, 1985.

[140] Matthew J. Gallagher and V.Lakshmi Narasimhan. Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering*, 23:473–484, 1997.

[141] Software artifact Infrastructure Repository (SIR). A repository of software-related artifacts meant to support rigorous controlled experimentation. online, 2011. http://sir.unl.edu/portal/index.html.

[142] Phyllis G. Frankl. *The Use of Data Flow Information for The Selection and Evaluation of Software Test Data*. PhD thesis, New York University, New York, NY, USA, 1987.

[143] Eugenia Díaz, Javier Tuya, Raquel Blanco, and José Javier Dolado. A tabu search algorithm for structural software testing. *Computer and Operation Research*, 35(10):3052–3072, October 2008.

[144] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[145] Gordon Fraser and Andrea Arcuri. Sound empirical evidence in software testing. In Martin Glinz Gail Murphy Mauro Pezzè, editor, *ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM/IEEE, 2012.

[146] Gordon Fraser and Andrea Arcuri. Evosuite at the sbst 2013 tool competition. In *SBST workshop*, 2013.

[147] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, (online), 2013.

[148] Gordon Fraser and Andrea Arcuri. Evosuite: On the challenges of test case generation in the real world. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 362–369, March 2013.

[149] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2010.

[150] Michael Haug, Eric W. Olsen, and Lars Bergman. *Software Process Improvement: Metrics, Measurement, and Process Modelling*. Springer, 2013.

[151] Gordon Fraser and Andrea Arcuri. A large scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology*, 2014. To appear.

[152] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology*, 2014. To appear.

[153] B. Carter and Kihong Park. Scalability problems of genetic search. In *Systems, Man, and Cybernetics, 1994. Humans, Information and Technology., 1994 IEEE International Conference on*, volume 2, pages 1591–1596 vol.2, Oct 1994.

[154] Dirk Thierens. Scalability problems of simple genetic algorithms. *Evol. Comput.*, 7(4):331–352, December 1999.

[155] Mauro Birattari, Thomas Stutzle, Luis Paquete, and Klaus Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18. Morgan Kaufmann, 2002.

[156] David E. Goldberg, Kalyanmoy Deb, and James H. Clark. Genetic algorithms, noise, and the sizing of populations. *Complex Systems*, 6:333–362, 1991.