AN EXPERIMENTAL STUDY OF NEIGHBOURHOOD BASED META-HEURISTICALGORITHMS FOR TEST CASE GENERATION SATISFYING THE MODIFIED CONDITION / DECISION COVERAGE CRITERION

ARIFUL HAQUE

MASTERS OF SCIENCE
(SOFTWARE ENGINEERING)

UNIVERSITI MALAYSIA PAHANG

UNIVERSITI MALAYSIA PAHANG

DECLARATION OF THESIS AND COPYRIGHT

Author's Full Name : ______________________________

Date of Birth : ______________________________

Title : ______________________________

____

____

Academic Session : ______________________________

I declare that this thesis is classified as:

☐ CONFIDENTIAL (Contains confidential information under the Official Secret Act 1997)*

☐ RESTRICTED (Contains restricted information as specified by the organization where research was done)*

☑ OPEN ACCESS I agree that my thesis to be published as online open access (Full Text)

I acknowledge that Universiti Malaysia Pahang reserves the following rights:

1. The Thesis is the Property of Universiti Malaysia Pahang
2. The Library of Universiti Malaysia Pahang has the right to make copies of the thesis for the purpose of research only.
3. The Library has the right to make copies of the thesis for academic exchange.

Certified by:

_____________________ (Student's Signature)

_____________________ (Supervisor's Signature)

____
New IC/Passport Number
Date:

____
Name of Supervisor
Date:

NOTE : * If the thesis is CONFIDENTIAL or RESTRICTED, please attach a thesis declarationletter.

Universiti
Malaysia
PAHANG
Engineering • Technology • Creativity

SUPERVISOR'S DECLARATION

I hereby declare that I have checked this thesis and in my opinion, this thesis is adequate in terms of scope and quality for the award of the degree of Masters of Science (Software Engineering).

____

(Supervisor's Signature)

Full Name : Prof. Dr. Kamal Zuhairi Bin Zamli

Position : Professor

Date : May 2018

STUDENT'S DECLARATION

I hereby declare that the work in this thesis is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at Universiti Malaysia Pahang or any other institutions.

____

(Student's Signature)

Full Name :Ariful Haque

ID Number : MCS13002

Date : May 2018

AN EXPERIMENTAL STUDY OF NEIGHBOURHOOD BASED META-HEURISTIC ALGORITHMS FOR TEST CASE GENERATOR SATISFYING THE MODIFIED CONDITION / DECISION COVERAGE CRITERION

ARIFUL HAQUE

Thesis submitted in fulfillment of the requirements
for the award of the degree of
Master of Science (Software Engineering)

Faculty of Computer Systems and Software Engineering

UNIVERSITI MALAYSIA PAHANG

May 2018

ACKNOWLEDGEMENTS

First of all, I would like to show my sincere gratitude and thank to my supervisor Prof. Dr. Kamal Z. Zamli. His endless enthusiasm, constantly advice, support and patient have encouraged me in completing this research work of master degree.

I would also like to thank my wife Salma Sultana Tajia to keep inspiring and encouraging me to continue the research and complete the thesis. Thanks to my parents who not only supported me, also funded my study in initial level. Without this support, it would not be easy for me to start the Masters course.

Next, my appreciations are expressed towards the administrative staffs in *the Faculty of Computer Systems and Software Engineering* (FSKKP), Institute of Postgraduate Studies (IPS) and *International Office* (IO) for their passionate to handle all forms of official tasks during my master study. The appreciations also go to all members of Bangladeshi community, local and foreign friends in UMP.

Finally, this research work of Master study is funded and supported by UMP Postgraduate Research Grant Scheme (PRGS) - "*Development of AI Based Test Data Generation Strategy Satisfying MC/DC Criterion*", Graduate Research Scheme (GRS) - "*Development of AI Based Test Data Generation Strategy Satisfying MC/DC Criterion*" from IPS, UMP. I would to thank UMP for funding the research and helping me to concentrate on the study.

ABSTRAK

Pengujian perisian adalah satu bahagian penting dan tidak boleh di kompromi dalam pembangunan perisian bagi menentusahkan fungsi yang betul dan mengurangkan risiko kegagalan. Apabila perisian digunapakai dalam aplikasi misi tahap kritikal, kegagalan boleh menyebabkan kehilangan nyawa dan harta benda. Oleh yang demikian, adalah menjadi kewajipan untuk menguji semua kemungkinan laluan fungsi perisian secara menyeluruh. Pengujian menyeluruh adalah mustahil kerana sumber yang terhad dan masalah kekangan masa. Ramai penyelidik mencadangkan kriteria Ubahan Keadaan / Liputan Keputusan (MC / DC) sebagai penyelesaian kepada masalah ini masalah terutamanya apabila input melibatkan pembolehubah Boolean. Selalunya, MC / DC boleh mengurangkan bilangan kes-kes ujian secara mendadak dan memastikan semua laluan kritikal diuji. Bagi menjana kes-kes ujian yang memuaskan kriteria MC / DC, ramai penyelidik menggunakan algoritma meta-heuristik kejiranan (termasuk Penyepuhlindapan Simulasi (SA) dan Pendakian Bukit (HC)). Walaupun berguna, penggunaan algoritma kejiranan lain (termasuk Algorithma Bah Besar (GD) dan Penerimaan Tertunda Pendakian Bukit(LAHC)) belum cukup diterokai. Bagi mengenal pasti kekuatan dan kelemahan algoritma ini untuk MC / DC, kajian ini mencadangkan satu kajian eksperimen yang melibatkan empat algoritma meta-heuristik kejiranan. Selain daripada Penerimaan Tertunda Pendakian Bukit (LAHC) dan Great Algoritma (GDA) yang dilaksanakan sendiri, penyelidikan ini juga mengimplementasi semula algoritma Simulasi Penyepuhlindapan (SA) dan Pendakian Bukit (HC) untuk analisa perbandingan. Algoritma telah digunakan untuk menjana kes-kes ujian selama sembilan ungkapan Boolean yang berbeza kerumitan. Prestasi setiap algoritma dibandingkan dari segi bilangan kes ujian yang dihasilkan serta masa yang diperlukan. Pengalaman kami menunjukkan bahawa semua algoritma menjana kes ujian yang hampir sama, tetapi dari segi prestasi, mereka berbeza antara satu sama lain. Hasil terperinci kajian ini akan membantu jurutera ujian untuk memilih algoritma yang mereka perlukan untuk menjana kes ujian dengan cekap dan optimum.

ABSTRACT

Software testing is an important part of software development as it ensures the proper functionality of software and reduces the risk of failure. In the case when software is being adopted in a mission critical application, failure can lead to loss of life and fortunes. Therefore, it is mandatory to test all possible functional paths of the software exhaustively. Exhaustive testing is costly and time consuming and with the higher number of inputs, the number of test cases increases exponentially. Many researchers suggested the adoption of Modified Condition / Decision Coverage (MC/DC) criterion as a solution to the problem particularly when the inputs involve Boolean variables. Often, MC/DC can reduce the number of test cases dramatically and ensure critical paths are tested. To generate test cases that satisfy MC/DC criterion, many researchers adopt neighborhood based meta-heuristics algorithms (including that of Simulated Annealing and Hill Climbing) as the problem itself is neighborhood based. Although useful, the existing algorithms does not provide any comparative data to select an algorithm based on the problem size and difficulty and the use of other neighborhood algorithms (including Great Deluge and Late Acceptance Hill Climbing) has not been sufficiently explored as well. In order to identify the strength and weakness of these algorithms for MC/DC compliant test cases, this research proposes an experimental study involving four neighborhoods based meta-heuristic algorithms. We have chosen four neighborhood based algorithms which are commonly used in optimization problems and divided them in newly implemented and re-implemented category. Late Acceptance Hill Climbing (LAHC) and the Great Deluge Algorithm (GDA) which are our new implementation, Simulated Annealing (SA) and Hill Climbing (HC) are re-implemented to generate test cases satisfying MC/DC criterion for comparative analysis. The algorithms are used to generate test cases for nine different Boolean expressions of different size and complexities. Performance of each algorithm is compared in terms of number of test cases generated as well as the run time required. Our experience indicates that all the algorithms generate nearly similar number of test cases, but in terms of performance, they differ from one another. The elaborated result of the study will help test engineers to choose the algorithm they need to generate test cases efficiently and optimally.

TABLE OF CONTENT

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACO	Ant Colony Optimization
AMP	Adaptive Memory Programming
CC	Condition Coverage
CFG	Control Flow Graph
DC	Decision Coverage
ET	Evolutionary Testing
FAA	Federal Aviation Administration
GDA	Great Deluge Algorithm
GPS	Global Positioning System
GUI	Graphical user Interface
HC	Hill Climbing
IDE	Integrated Development Environment
LAHC	Late Acceptance Hill Climbing
LP	Linear Programming
MC/DC	Modified Condition Decision Coverage
MCC	Multiple Condition Coverage
SA	Simulated Annealing

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

In the twenty first century, human life has become more comfortable, fast and productive than at any time in history. The main reason for this change is that, now machines and electronic devices can very efficiently do the work that people used to do previously, and they can do it many times more efficiently than humans. Though machines are physically doing the main work, software is controlling machines to perform their task. Without software, almost all the machines and devices will be reduced to some pieces of junk hardware.

From this perspective, application of software can be found almost everywhere. From digital wristwatches to robots going to Mars or under Atlantic sea, are controlled by software. From Global Positioning System (GPS) suggesting roads to choose to avoid traffic jam, to autopilot taking control of airplanes in the sky, satellite orbiting Earth to perform seamless communication around the Earth in seconds are run by sophisticated software.

The comfort in human life achieved using software does have errors as well. By the nature of software development, software may have faults and they can malfunction in certain condition or time. Software needs to be very carefully designed and rigorously tested before it is marketed, to avoid or at least reduce chances of malfunctioning.

Software testing is the process of finding defects (i.e. sometimes involve executing the software of interest) and of validating the software/system against its specification. Often, software testing adopts a set of test cases (also termed test suite) to perform the actual tests. A basic test case contains number of certain input parameters and expected output. When these test cases are executed, behaviour of the software system is monitored to determine its correctness.

Often, test cases are required to cover an accepted percentage of code lines and functions of the system being tested. A variety of coverage criteria have been proposed to assess the effectiveness of the sampled set of test cases. As far as structural testing is concerned, criteria exercising aspects of control flow, such as statement, branch and path coverage, have been the most common criteria (Zamli, Al-Sewari, & Hassin, 2013).

Owing to resources and timing constraints, development of the right test cases covering the aforementioned criteria is very challenging as there can be thousands to millions of test cases depending on the size of particular software of interest. In the worst scenario, if the software needs redesign and recode, all the test cases might need to be rewritten also. For this reason, there is a need for an automated test case generation to alleviate such problem.

Continuing from these aforementioned discussions, the next few sections highlight overviews on software testing, problem statement discussion, thesis aim and objectives, and research contributions. Finally, the thesis outline will be elaborated in final section.

1.1.1 Importance of Software Testing

The effect of the software malfunction can vary with the criticality of the system in which it is used. Malfunction of mobile phone software or a website providing daily news or movie information can have no or very little effect on the necessities of human life, but failure of banking and financial software may affect in financial loss. In June 2006, due to a software bug, a top telecommunication company send incorrect bill to its 11,000 customers and they were over-billed up to several thousand dollars each. They fixed the bug immediately, but correcting the billing errors took much longer (Hower, 2010). After two months, in August 2006, 21,000 student load borrowers private information was converted to public information by a software defect in a US Government student loan service on its website (Kumar, Raghu, & Kumar, 2013). In January 2009, a large health insurance denied the coverage for needed medicines and cancelled some benefits. Later, it was found those insurance policies were mistakenly sold owing to the problem in their computer system. The company was banned by regulators from selling certain types of insurance policies because of the bug in their system. The regulatory agency stated that the problems were posing "a serious threat to the health and safety" of beneficiaries (Hower, 2010).

A glitch in the safety of critical software like nuclear plan management system, air traffic collision avoidance system or aircraft flight control system may have serious financial loss as well injury or loss of human life. In August 2008, more than 600 U.S. airline flights were significantly delayed because of a database mismatch resulting in a software glitch in the U.S. FAA air traffic control system (Hower, 2010). Another example is the failure of Ariane 5, the rocket launched in 1996 by the European Space Agency that exploded just forty seconds after its lift-off from Kourou, French Guiana. The rocket was on its first voyage, and its development cost $7 billion. The cause of the failure was an overflow in a conversion from a 64-bit floating point to a 16-bit integer. The overflow caused the rocket computer to shut down for few seconds and lose all contact with the ground station.

Incidents like this clearly show the importance of Quality Assurance in Software Development life cycle. Software testing is one of the tools of Software Quality Assurance (SQA), which validate the software against its actual requirements. Depending on the genre of the software, different level of testing is required before release to the market. Obviously, the level of ‘enough’ testing for a word processing application is *not enough* for an air traffic collision avoidance system. Hence different testing standards have emerged to give some guidelines for the right software testing standards. As just one example, for the aerospace domain software should be compliant with RTCA/DO-178B standard “*Software Considerations in Airborne Systems and Equipment Certification*”. The document provides a set of verification and validation activities to properly test a safety critical system. Failure to comply with the standard will prevent the software to get aerospace market release approval from Federal Aviation Administration (FAA).

While various criteria call for detailed attention and intensive research, the focus of this work is on one of the criteria included in DO-178B, namely “Modified Condition/ Decision Coverage” (MC/DC). To be specific, MC/DC criterion dictates each condition within a predicate (i.e. a set of Boolean expressions can independently influence the outcome of the decision - while the outcome of all other conditions remains constant. In this manner, MC/DC criterion subsumes statements, decisions, and path coverage.

1.2 Problem Statement

As highlighted earlier, test cases are required to cover an accepted percentage of code lines and functions of the system being tested. At a glance, statement, decision and path coverage appear sufficiently effective exercising the various parts of the software implementation. Nonetheless, a closer look reveals otherwise. Statement, branch, and path coverages are often susceptible to the problem of masking.

For small inputs, the problem of masking can be straightforwardly addressed by considering all exhaustive input combinations. Yet, for large inputs involving complex predicates, the number of exhaustive combinations can be prohibitively large. Additionally, as the software is modified and new test cases are often added to the test suite, the cost of regression testing keeps increasing. This scenario causes significant challenge for test engineers, that is, in terms of getting the required sample test cases. To address the test-suite size problem within the context of structural testing, many researchers have started to advocate the usage of Modified Condition/Decision Coverage (MC/DC) criterion as a strategy to systematically minimize the number of test cases for testing, while keeping the effectiveness of testing at its highest.

As the problem of test case generation fulfilling MC/DC criterion is NP complete, no single approach can generate optimal test set for every predicate consideration in polynomial time, especially involving large and complex expressions. Furthermore, the process of finding a set of test cases to achieve MC/DC criteria is typically a labour intensive activity requiring much automation support.

Many researchers have already provided different solutions to the problem of Test Case Generation that satisfy MC/DC criterion. Awedikian’s uses Hill Climbing (HC) algorithm to generate MC/DC compliant test suite (Awedikian, Ayari, & Antoniol, 2009). Ghani and Clark adopts Simulated Annealing(SA) (Ghani & Clark, 2009) while Ghada El-Sayed’s adopts Genetic Algorithm(GA) for the same purpose (El-Sayed, Salama, & Wahba, 2015).

While the aforementioned works have been useful to address the problem of MC/DC test suite generation, they are not without limitations. HC has been criticized for the tendency to fall into local optima solution. As such, generating MC/DC for large scale predicate of Boolean expression can be problematic. SA based strategy is often overly sensitive to the annealing schedule and initial starting points. Thus, poor annealing schedule and too distant starting point can also lead to local optima solution. GA based strategy is computationally heavy owing to the need to frequently interact with the search environment to perform mutation and crossover. Furthermore, the fact that GA adopts population based approach may lead to slow convergence as it does not sufficiently utilize the current best found solution (i.e. lack of exploitation of existing knowledge). For this reason, given that MC/DC is inherently neighbourhood biased (i.e. changing one Boolean value at-a-time and holding other Boolean constant), population based meta-heuristic like GA is deemed unsuitable.

Despite its potential, the adoptions of Great Delude Algorithm (GDA) and Late Acceptance Hill Climbing Algorithm (LAHC) (E. K. Burke & Bykov, 2012) as the neighbourhood based solution for generating MC/DC compliant test suite have not been sufficiently investigated in the literature. Praises for its simplicity, GDA adopts a simple flood analogy: the water rises continuously and the proposed solution must lie above the surface to survive. Late Acceptance Hill Climbing Algorithm (LAHC) improves the standard HC Algorithm by delaying the acceptance of good solution within a memory such that local optima problem can be overcame.

Given the aforementioned prospect, thesis investigates the adoption of GDA and LAHC based strategy for MC/DC compliant test suite generation. To ensure fair comparison, this work also re-implemented the SA and HC algorithms to allow objective analysis between these four implementations.

1.2.1 Problem with Existing Works and Motivation

Many researchers has already provided different solutions to the problem of Test Case Generation that satisfy MC/DC criterion. Some mentionable work are, Ghani & Clark's use of Simulated Annealing (SA)(Ghani & Clark, 2009)to find MC/DC satisfied test cases, Awedikian's use of Hill Climbing algorithm to solve this very problem. Ghada El-Sayed's use of Genetic Algorithms etc. Though there are other researchers also who used these same Neighbourhood based algorithms, in this study, after search in several indexing sites, no research paper was found any implementation of Great Delude Algorithm (GDA), Late Acceptance Hill Climbing Algorithm (LAHC) to solve this problem. Also no comparative analysis of existing solutions (in terms of generated test case number and required time) was found.

Considering the fact of no use of GDA and LAHC to generate MC/DC satisfied Test Cases and no proper information about, among these neighbourhood based algorithms, which algorithm perform better to generate test cases satisfying MC/DC criterion, I become motivated to implement GDA and LAHC to development of a new test data generation strategy satisfying MC/DC criterion and also re-implement SA and HC to perform a comparative analysis between this four implementations.

1.3 Research Aim and Objective

The aim of this research is to design and implement neighbourhood based test case generation strategies comprising of Simulated Annealing (SA), Hill Climbing (HC), Great Deluge Algorithm (GDA) and Late Acceptance Hill Climbing (LAHC) for generating MC/DC compliance test suite. The key objectives of this research are:

I. Study the use of neighbourhood based metaheuristic algorithms for test case generation satisfying the MC/DC Criterion

II. Develop two re-implementations of neighbourhood based algorithms based on Simulated Annealing (SA), Hill Climbing (HC) and two new implementation of neighbourhood based algorithm Great Deluge (GDA), and Late Acceptance Hill Climbing (LAHC) that satisfy MC/DC.

III. Compare and analyse the performance of the developed strategies in terms of test case size and execution time.

1.4 Scope of the Research

The scope of this research is to develop four strategies for test case development using neighbourhood based meta-heuristic algorithms. Two of the strategies are new strategy development and two is re-implementation of the algorithm. All the test cases must satisfy MC/DC Criterion.

In this thesis, four test case development strategies are proposed which are developed using Simulated Annealing (SA), Hill Climbing (HC), Late Acceptance Hill Climbing (LAHC) and Great Deluge Algorithm (GDA) meta-heuristic algorithms. An automated test case generation tool will be developed using the new strategy which will provide output as test cases based on given input as Boolean expression e.g. `(a&&b)!c`. The tool will provide facility to choose any of the four strategies which will be used to generate the output.

A comparative analysis of performance among four new developed test data generation strategy is performed in this thesis. In this research, the developed tool will provide additional meta-data about performance of the strategy which will be used to perform this comparative analysis of performance based on the result size and execution time.

1.5 Organization of the Thesis

This rest of the thesis is organized as follows:

Chapter 2 gives overviews on MC/DC criterion and use of Meta-heuristics algorithm in test case generation. Moreover, existing work on Test Case generation to satisfy MC/DC criterion is highlighted.

Chapter 3 illustrates the methodology and strategy implementation of Test case generation satisfying MC/DC criterion. The chapter provides new approach of four algorithms and explains them.

Chapter 4 involves result and discussion. Functions / Boolean expressions under experiments, configuration of algorithms are explained.

Chapter 5 draws the conclusion of this research work along with the scope for future work. Finally, the chapter ends with a closing remark.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

Much work has been published in automated test data generation for software engineering, and particularly software testing. In this chapter, a selective summary of research work on some of the systematic approaches adopted in MC/DC criterion based automated test case generation strategy are being presented. The discussion is organized into a few sections, involving discussion about MC/DC; Structural Testing; Search Based Software Testing; Meta-heuristic Algorithms in test data generation and the four different algorithms adopted in this research to experimentally generate test data on some sample problems. The chapter ends with a brief summary.

2.2 MC/DC Criterion

MC/DC was developed to ease the testing of complex Boolean expressions in safety-critical applications such as traffic collision avoidance systems in aircrafts, and is now rigorously applied to several critical domains including patient monitoring systems in hospitals and nuclear power control systems(Paul & Lau, 2014).

2.2.1 Definitions

For sake of completeness we report in the following paragraphs the basic definitions which are preliminary to the concept of MC/DC test data generation.

A **condition** is a Boolean expression containing no Boolean operators. As an example, `a <b` is a condition, while `a ||b` is not, since it contains the Boolean operator `OR`, denoted by the symbol `||`.

A **decision** is a Boolean expression composed of conditions connected by Boolean operators. A decision without a Boolean operator is a condition. If a condition appears more than once in a decision, each occurrence should be treated as a distinct condition(Hayhurst, Veerhusen, Chilenski, & Rierson, 2001).

For example, a < b && a < c is a decision composed of two conditions. Here && is the Boolean operator AND. Each if statement in a code is a decision statement and it contains one or more conditions.

Other terms that need to be defined are a predicate, major clause and minor clause. A **predicate** is a synonym used for condition. A **major clause** is the condition the test aims to prove that it affects correctly the outcome of the decision, while **minor clause**s are all other conditions in the decision.

2.2.2 Structural Coverage and MC/DC

Structural coverage analysis provides a means to confirm that the requirements-based test procedures have been exercised the code structure (Authority, 1992; Hayhurst et al., 2001). The purpose of structural coverage analysis is to complement requirements-based testing by:

1. Providing evidence that the code structure was verified to the degree required for the applicable software level;
2. Providing a means to support demonstration of absence of unintended functions; and
3. establishing the thoroughness of requirements-based testing.

Structural coverage criteria may be categorized into data flow and control flow criteria. While data flow criteria measure the flow of data between variable assignments and references to the variables, control flow criteria measure the flow of control between statements and sequences of statements. In the latter criteria, forming most of the DO-

178B standard, the degree of structural coverage achieved is measured in terms of statement invocations, Boolean expressions evaluated, and control constructs exercised. The different control flow criteria can be summarized as follows:

The **Statement Coverage** (SC) criteria insists that *every executable statement* in the program is invoked at least once during software testing. It is considered aweak criterion as itis has been seen to be insensitive to some control structures.

Decision Coverage (DC) requires that *every decision* in the program has taken all possible outcomes at least once. DC calls for just two test cases for a decision: one which has a true(T)outcome, and the other which has a false(F) outcome for that decision. For simple decisions with just one condition and no Boolean operators, decision coverage ensures complete testing of control constructs. However, for decisions that contain one or more Boolean operators, e.g. A OR B, two suitably chosen test cases (e.g. TF and FF) will affect the outcome of the decision to become both T and F, but leave the effect of B untested. In other words, the DC criteria for a complex decision with more than one condition cannot distinguish between the decision A OR B, and the decision with just one condition A.

Condition coverage (CC) requires that *every condition* in each of the decisions of a program has taken all possible outcomes at least once. Thus, CC overcomes the problem posed by DC, namely the under-testing of multiple conditions in a decision. However, CC does not require that the decision take on all possible outcome sat least once. This provides a loophole for the bugs to remain hidden, and does not guarantee the code to be error-free. For example, for the decision A OR B, the two test cases TF and FT is sufficient to meet the CC criteria, but do not cause the decision to take on all possible outcomes.

Condition/ Decision Coverage (C/DC) combines the requirements for decision coverage with those of condition coverage. That is, there must be sufficient test cases to get both the decision outcomes T and F, and span through both the condition values of T and F for either of the conditions A and B. It may be noted that a minimum of two test cases are necessary for each decision to satisfy the C/DC criteria. Using the same example A OR B, just two test cases TT and FT would be sufficient to meet the C/DC requirement.

However, these two tests do not distinguish the correct expression `A OR B` from the expression `A`, or the expression `B`, or even from the expression `A AND B`.

These limitations of the above three criteria, *viz.* DC, CC and C/DC express the necessity of a certain coverage condition to be framed so that any bug in the code can be easily located. This is effectively fulfilled by the following criteria, accepted as standard for Level A software in DO-178B.

The **Modified Condition/Decision Coverage** (MC/DC) criterion enhances the condition/decision coverage or C/DC criterion by requiring that *each condition* should be shown to *independently affect the outcome of the decision*. The independence requirement ensures that the effect of each condition is tested relative to the other conditions. However, achieving MC/DC requires more thoughtful selection of the test cases, and in general, a minimum of (n + 1) test cases for a decision with n inputs. For the above example concerning the decision `a OR b`, a test suite consisting of three test cases `TT`, `FT` and `FF` satisfy the MC/DC criteria completely. For decisions with a large number of inputs, MC/DC requires considerably more test cases than any of the coverage measures discussed above.

Finally, there is another criterion called the **multiple condition coverage**, which requires that test cases ensure each possible combination of inputs to a decision is executed at least once; that is, multiple condition coverage requires exhaustive testing of the input combinations to a decision. Hypothetically, multiple condition coverage is the strongest and the most desirable structural coverage measure. However, it is computationally intensive, and turns out to be infeasible for testing of most practical codes. For a decision with n inputs, multiple condition coverage requires 2^n tests.

On the other hand, MC/DC is intended to assure, with a high degree of confidence, that requirements-based testing has demonstrated that each condition in each decision in the source code has the proper effect on the outcome of the decision (Hayhurst et al., 2001). The goal of MC/DC criterion is thus to prove that:

- every decision in the program under test has taken all possible outcomes at least once,

- every condition in a decision of the program has taken all possible outcomes at least once,

- more importantly, that each condition in a decision affects independently and correctly the outcome of that decision.

An additional advantage of enforcing the MD/DC criteria is that the tester can locate where exactly the error is in a decision, if any.

2.2.3 MC/DC Test Cases

There are two MC/DC variants. In the first one, also referred to as the unique cause MC/DC, minor clauses must hold the same Boolean value for the two values of the major clause. The second interpretation of MC/DC, a weaker criterion known as the masking MC/DC, allows minor clauses to be different (Ammann, Offutt, & Huang, 2003). In this work, we will consider the strongest interpretation, the unique cause MC/DC, as it is the criterion required by the standard DO-178B.

In order to generate the test suite to cover the MC/DC criterion for one decision, the major clause value should vary while the minor clauses outcomes are fixed, to show the effects of the major clause on the entire decision. Boolean conditions such as `a < b`are denoted by capital letters representing the condition outcome (A, B, C, etc.) and the Boolean outcomes are denoted true (`T`) or false (`F`).

To help understanding a decision such as (A and B), logical operators (or, and, etc.) are presented schematically by logical gates, and a truth table is built for the entire logical circuit. This truth table represents the truth table for the decision under test.

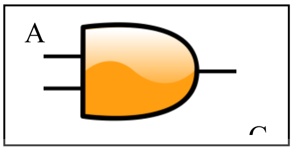
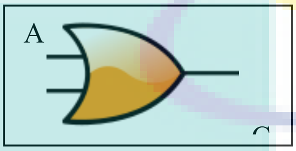
<table>
<tr><th>Schematic Representation</th><th>Coding format</th><th>Truth Table</th></tr>
<tr><td> </td><td>C := A <b><i>and</i></b> B</td><td><table><tr><th>A</th><th>B</th><th>C</th></tr><tr><td>T</td><td>T</td><td>T</td></tr><tr><td>T</td><td>F</td><td>F</td></tr><tr><td>F</td><td>T</td><td>F</td></tr><tr><td>F</td><td>F</td><td>F</td></tr></table></td></tr>
<tr><td> </td><td>C := A <b><i>or</i></b> B</td><td><table><tr><th>A</th><th>B</th><th>C</th></tr><tr><td>T</td><td>T</td><td>T</td></tr><tr><td>T</td><td>F</td><td>T</td></tr><tr><td>F</td><td>T</td><td>T</td></tr><tr><td>F</td><td>F</td><td>F</td></tr></table></td></tr>
</table>

Figure 2.1 Representations of Elementary Logical Gates

Source: Hayhurst et al. (2001)

Figure 2.1 illustrates the logical ***AND*** and ***OR*** Boolean operators represented by logical gates. It also provides the truth table for these gates. Each row in the truth table presents a possible test case, thus in the truth table of the ***and*** gate for example, we have four possible test cases for the decision A ***and*** B. However, when developing test sets, we want also to minimize the number of test cases required to cover the MC/DC criterion. Thus, for each major clause, we search for a pair of rows where the condition outcome varies, the minor clauses outcomes are fixed and the outcome of the entire decision varies. For the A ***and*** B truth table, the pairs of rows for each major clause are:

- Major clause as A: (TT→ T), (FT→ F), where B is held fixed at T.
- Major clause as B: (TT→ T), (TF→ F) , where A is held fixed at T.

Thus, the final test set is [(TT), (FT), (TF)]. As a general rule, a set of test cases with one more than the number of inputs is needed to provide the minimal coverage for a multiple-input decision.

```
1: public Integer calculate(Integer x, Integer y, Integer z)
2: {
3:    Integer result = -1;
4:    Boolean fail = false;
5:    if (x < 0 || y < 0 || z < 0)
6:    {
7:           fail = true;
8:    }
9:    else
10:   {
11:          fail = false;
12:   }
13:   if (result == 0);
14:   {
15:          if (z == 0);
16:          {
17:                 result = x + y;
18:          }
19:          else
20:          {
21:                 if (z > x && z > y || z > x + y)
22:                 {
23:                        result = z;
24:                 }
25:                 else
26:                 {
27:                        result = x + y
28:                 }
29:          }
30:   }
31:   return result;
32: }
```

Figure 2.2 Calculate Method

Consider the Java code in Figure 2.2; suppose we are interested in generating input data to satisfy the MC/DC for the decision at line 20:

`(z > x && z > y || z > x + y).`

We denote the conditions `(z > x)` by `A`, `(z > y)` by `B` and `(z > x + y)` by `C`, thus the decision is denoted as :

`(A && B || C).`

We build the truth table of the correspondent logical circuit.

Table 2.1 Truth table of decision at line 21 of the Calculate method

#	A	B	C	A && B || C
0	T	T	T	T
1	T	T	F	T
2	T	F	T	T
3	T	F	F	F
4	F	T	T	T
5	F	T	F	F
6	F	F	T	T
7	F	F	F	F

Searching the truth table for the pairs of rows for each major clause, we obtain:

- A: (5,1)
- B: (3,1)
- C: (7,6), (5,4), (3,2)

To save space test cases were represented by a decimal coding thus for example the number 4 stands for `FTTT` in the truth table. We have two minimal sets to cover the MC/DC criterion: [1,3,5,4] and [5,1,3,2]. We can choose any one of the two sets.

2.3 Overview of Software Testing

Though software testing may identify significant amount of bugs in the system, it never means zero defect (Schulmeyer, 1990). There is no proper guideline for when to stop testing. It’s a trade-off between budget, time and quality. Depending on the testing techniques, software testing is usually divided into two main categories, viz. Black box testing and White box testing. Grey box testing is an intermediate category combining features of both these two categories. The three categories are briefly described as follows:

2.3.1 Black Box Testing

Black-Box Testing is also known as few other names, e.g. Functional Testing, Requirement Based Testing, Data Driven Testing, Input / Output driven (Hetzel & Hetzel, 1988). In Black-box testing, test cases are originate from functional requirement of the system and program structure is not a concern here (Perry, 1992). This is the reason, this is known as functional testing. In Black-box testing, the testing process emphasize on executing the functions and examining their input and output data. Here the test engineer consider the system under test as a black-box and only input, output and specification is visible. In practice, multiple numbers of inputs are used and outputs are compared against its specification to validate the correctness.

There has been less activity in the area of search-based functional testing compared to structural testing (McMinn, 2004). Functional tests can be derived from different forms of specification. For tests derived in this way, a common barrier in the automation of test data generation is that a mapping needs to be provided from the abstract model of the specification to the concrete form of the implementation. For system tests, there is also a potential problem in the size of the search space.

There is another category of testing known as **Grey-box testing**, which combines both structural and functional information for the purposes of testing. Grey box testing can be done for different constructs such as Assertions or Exceptions (N. J. Tracey, 2000). Assertions specify constraints that apply to some state of a computation. When an assertion evaluates to false, an error has been found in the program. The ability to embed arbitrary assertions within programs and be able to search for test data in order to check their violation is a very powerful concept. Some languages such as C++, Java and Ada provide explicit constructs to handle errors or ‘exceptions’ (as errors are called in these languages) so that exception-related code can be separated from the main logic of the program. One particular grey-box testing method consists of generating test data for raising such exception, and then for the structural coverage of the exception handler.

2.3.2 White Box Testing

White-box testing is also known as *structural testing*. In this technique, software is considered as white box, which means anything going on inside the box is visible from outside and is under observation. Testing planes are compiled according to the details of software implementation, such as programming language, logic etc. Test cases are derived from the program structure. Rather than focusing on output, the tester concentrates on how the statements are executing, how the data is passing through the branches and paths of code.

There are several white box coverage criteria (Ammann et al., 2003; Pandita, Xie, Tillmann, & de Halleux, 2010) implementing control flow testing, of which statement, branch and path coverage are the most common (Awedikian et al., 2009). At a glance, statement, branch and path coverage is sufficiently effective exercising the various parts of the software implementation. But in deep beneath the statement, branch and path coverage, they all are vulnerable to the masking problem (Chilenski & Miller, 1994; Zamli et al., 2013) Here the use of AND and OR operations to form compound predicates as the control flow for statement, branch and path coverage can potentially be problematic. Masking problem is described elaborately in Problem Statement section.

2.4 Search Based Software Engineering

Search Based Software Engineering (SBSE) applies search based optimization algorithms to solve optimization problems drawn from software engineering. The optimization algorithms are used to find most favorable solutions for a specific problem. SBSE techniques can be applied when potential solution space is large and complex and still need to address multiple objectives and/or constraints (Harman, 2007). SBSE is getting popular in because such situations are common in software engineering.

As early as 2001, Harman and Jones (Harman & Jones, 2001) claim that “a new field of software engineering research and practice is emerging: search-based software engineering.” The paper argues that software engineering is an ideal backdrop for the application of metaheuristic search techniques such as tabu search, simulated annealing and genetic algorithm. Such search-based techniques could provide solutions to the difficult problems of balancing competing (and sometimes inconsistent) constraints and may suggest ways of finding acceptable solutions in situations where perfect solutions are either theoretically impossible or practically infeasible. To develop the field of search-based software engineering, the need for a reformulation of classic software engineering problems as search problems is proposed. The paper briefly sets out key ingredients for successful reformulation and evaluation criteria for SBSE.

The search based optimization algorithms used in SBSE are meta-heuristic algorithms such as Linear Programming, Hill climbing, Tabu Search, Simulated Annealing, Ant Colonies, Random Search, Particle Swarm Optimization, LP, Genetic Algorithm, and Genetic programming. The search mechanism of these algorithms in large search spaces is often guided by a fitness function that captures properties of the acceptable software artefacts we seek.

In the past five years, SBSE techniques were used for several applications such as model checking (Alba, Chicano, Ferreira, & Gomez-Pulido, 2008), Regression testing (Li, Harman, & Hierons, 2007), maintenance, test case generation (McMinn & Holcombe, 2006), etc. We are mostly interested in the application of SBSE to software testing.

2.5 Search Based Software Testing

We use the term Search Based Software Testing (SBST) throughout our research to point the application of SBSE to the software testing problems. An important issue in software testing is generating the test inputs automatically to apply them in the system under test.

Multiple test cases or a set of test cases are required to cover a test criterion. For each test caes, we need to generate test input data. This process is known as test data generation. A successful test data generation is when the generated input parameters of the system under test is generated in a specific way that satisfy the test case. A simple process to generate the data is to explore the entire parameter space and find the perfect combination of values. However, for systems with multiple parameters, exploring the entire parameter space will increase exponentially, and a full-scale also known as exhaustive search becomes time consuming and cost in affective. For instance, aerospace software systems can take up to 15 or more parameters; in this cases, if the parameters are integers, the parameter's space is $2^{(15*32)}$. This is an optimization problem and solution to this problem is to use an approximation algorithm to search for the data, which is known as heuristic technique. Such a technique risk to either not find a solution, when there exists one, or to find a non-optimal solution. However, it allows searching for the data in a reduced search space and in a reduced search time.

A number of approaches based on heuristic search methods have been developed. In general, SBST uses search based optimization techniques to formulate the test data generation problem as a search optimization problem (Lakhotia, Harman, & McMinn, 2008). This problem is then addressed using heuristic methods; it can also be formulated as a constraint optimization problem or a constraint satisfaction problem (Sagarna & Yao, 2008).

Jia, Cohen, Harman and Petke (Jia, Cohen, Harman, & Petke, 2015) have developed a hyper-heuristic algorithm for combinatorial interaction testing (CIT), which learns search strategies across a broad range of problem instances, providing a single generalist approach. The hyper-heuristic is an ensemble approach that chooses the best strategy for a particular problem from a bewildering choice of CIT techniques known to software engineers, each specialized for a particular task. The authors report experiments to show that the proposed algorithm competes with best known solutions across constrained and unconstrained problems. For all 26 real-world subjects, the hyper-heuristic algorithm equals or outperforms the best result previously reported in literature. An evidence is also presented that the algorithm's strong generic performance results from its unsupervised learning.

The meta-heuristic search techniques used in Search Based Software Testing (SBST) are high-level search frameworks that use heuristic methods to find solutions to combinatorial problems at a rational computational cost. To guide the search, meta-heuristic algorithms need a fitness function to represent the combinatorial problem. For this, the testing criterion is transformed into a fitness function. The search space is the space of possible inputs to the system under test.

SBST has proved to be effective to some extent because it has a wealth of optimization techniques upon which to draw and because the common nature of the approach allows it to be adapted by a wide range of test data generation problems; in principle, all that is required to adapt a search based technique to a different test adequacy criterion is a new fitness function (Lakhotia et al., 2008).

Works on heuristic based approaches to software engineering testing problems date back to as early as 1976, when Miller and Spooner used optimisation techniques for test data generation (Miller & Spooner, 1976). In 1992, Xanthakis et al. were the first to apply meta-heuristic optimization technique for test data generation (Xanthakis et al., 1992). In recent years, several approaches that use meta-heuristic search techniques to automatically generate the test inputs for a given test criterion have been proposed. As the MC/DC is a structural testing criterion, we are mainly interested in meta-heuristics based works done on structural testing.

Here, we present two main researches done in this field. The first one is by Miller and Spooner, as it was the first work to use search based optimization techniques for test data generation (Miller & Spooner, 1976). The second work done by Korel is the first to use data analysis to help the heuristic search (Korel, 1990b). In the following section, we will present more recent work that uses different search approaches for SBST problems.

The software tester is given the responsibility to select the input data for which she evaluates the its quality(i.e. meeting its functional specifications). If done manually, such a practice is extremely costly, difficult and laborious. Moreover, due to mundane limitations, the manual testing may often fail to reveal serious bugs in the software. Search based software engineering (SBSE) aims at solving optimization problems by applying heuristics techniques to explore large input spaces of a software, and find out the minimal set of inputs required for testing consideration. The generation of input data can be modelled as a search problem in a large search space that we aim to optimize. The input search space is the set of all possible variable values that the software can take as input, while each of these values (solutions) are graded according to a fitness function defined by the user to quantitatively express the level to which the software fulfils the goal of the particular test. Thus, we apply the SBSE techniques to our research at hand.

Meta-heuristic search techniques used in SBST are high-level frameworks which utilize different search techniques to find solutions to combinatorial problems at a reasonable computational cost. The problem may be NP-complete or NP-hard, or a polynomial time algorithm is known to exist but still beyond the reach of available computational resources. Metaheuristics are not standalone algorithms in themselves, but rather strategies that can be readily adapted to specific problems such as software testing.

For each software testing problem, there are usually two main decisions to be implemented: the first one being the encoding of the solution, for example, how many variables a solution has, their types, etc.; and the second being the transformation of the test criteria into a fitness function, i.e. the degree to which the final goal of the test is achieved. Once these two are ascertained for a particular problem at hand, it can very well fit in to a metaheuristic solution technique. The generic nature of the solution methodology makes SBST very attractive for metaheuristic searches. Simply by changing the input space and fitness function, the approach can be adapted to different test data generation problems.

Metaheuristic search techniques have been applied to automate test data generation in different areas such as: structural (or white-box) testing, functional (orblack-box) testing, sometimes a testing that combines both structural and functional information (or gray-box testing), and non-functional testing. Structural testing is one of the most prospective areas where metaheuristic search techniques have received the greatest share of attention from researchers. MC/DC falls in this category of testing.

2.5.1 Solution

In SBST, a solution is a test input data generated for which the software is executed and its output tested for coherence with the expected outcome. The input data is the set of values of the variables which the software requires for its input, and on which the output possibly depends. The input space within which test data is sought is typically large, but well defined, which makes different metaheuristic search algorithms very well suited and effective for software testing problems. One MC/DC test case is executed at a time using one solution for each decision in the code. If we need to generate test data for a Triangle function, for example, the function’s parameters are three integers *a*, *b* and *c*. Thus, an input data is a triplet of integers, forming a solution drawn from a 3-dimensional search space.

The function of a metaheuristic is to choose the best set of solutions or input parameter values for which the software will be most rigorously tested. Thus, it is an optimization problem of maximizing the efficiency of tests with minimal test data. A good metaheuristic can be expected to approach the efficiency of exhaustive software tests, using only a small fraction of the exhaustive test data.

2.5.2 Fitness Function

The fitness function of a metaheuristic algorithm tries to quantify the extent to which its objective has been achieved by a particular solution. It is useful in guiding the heuristic effectively into a promising neighbourhood of the search space, seeking the perfect solution. For the testing problem, the test criterion objective is translated into a fitness function. The form of the fitness function is generated offline by the user, typically before the actual search for the data is started. When the meta-heuristic algorithm generates solutions one by one, the fitness is calculated for each solution generated and its value is used to compare and contrast the solutions with respect to the overall search goal. In most of the paradigms, better solutions marked by higher fitness values are either used directly, or to influence later solutions.

In SBST, the test objectives need to be defined numerically and transformed into a fitness function. Objective function is the quantified form of the optimizing goal that is either to be maximized or minimized. In case of maximization, as when finding out the number of correct outputs, the fitness function is usually taken to be the objective function itself

$$F(\mathbf{x}) = f(\mathbf{x}) \quad \text{for maximization}$$

where $F(\mathbf{x})$ is the fitness function, and $f(\mathbf{x})$ is some objective function that is to be maximized. On the other hand, if it is a minimization problem, like minimization of number of errors or mismatches, then fitness is taken to be the reciprocal of the objective function

$$F(\mathbf{x}) = \frac{1}{1+f(\mathbf{x})} \quad \text{for minimization}$$

where the addition of 1 to the objective function prevents division by zero.

The search space is the system under test input domain and the fitness is computed by monitoring program execution results. With feedback from the objective function, the metaheuristic searches for ‘better’ solutions based on knowledge and experience of previous solutions. A proper selection of the objective function is therefore critical to the success of the search heuristic (McMinn, 2004).

2.6 Structural Testing

Automation of structural testing and structural coverage criteria has been the most widely investigated subjects. In the beginning of structural testing automation, Miller and Spooner used local search to develop the automation strategy (Miller & Spooner, 1976). Their goal was to automate the generation of input data that cover particular paths in a program. They prepared the problem as a numerical maximization problem and used heuristic approach to solve it. Miller & Spooner was focused to generate floating point data to cover test cases for branch testing criterion. The used approach fixes every integer parameters in the program, and tries to produce values for the remaining floating point parameters for all possible paths in the program. Since every program execution takes the form of a straight-line program, it is possible to collect any path constraints for a given execution. The approach uses the collected constraints to form the fitness function. The approach applies numerical techniques for constrain maximization and the process starts from a random initial point. A loop iterates until the fitness function become positive which is the final goal here to complete the process (Miller & Spooner, 1976).

This discussed approach has two disadvantages. First it only targets floating points parameters; second, often inappropriate paths are selected; as a result, significant computational effort is wasted analysing these paths and trying to find data covering them.

Korel extended the work later in 1990 (Korel, 1990b). Like Miller, goal of this research is to generate test cases that cover branch testing criterion. While the earlier work depends on the static constraints in the program execution to form the fitness function guiding the search, Korel uses a dynamic method that relies on the actual execution of the program with input data.

Initially random inputs are given to execute the program. During each execution for a selected branch, a search procedure determines whether the execution should continue through the current branch or an alternative branch should be taken. This choice is made based on the control flow graph of the program, determined prior to the execution of the program. Branches are organized into categories as critical branches, required branches, semi-required branches, and non-required branches. These categories represent the control dependencies between branches. Thus, if the execution flow is diverging from the targeted branch, a real valued function is associated with this branch; the fitness value. A minimization search algorithm is then used to automatically generate data to change the flow of execution at this branch. To speed up the search, Korel uses a data flow analysis technique to determine input variables that are responsible for the undesirable program behaviour. The technique is used for software programs with some high numbers of parameter such as big size arrays, and it aims at detecting which of the input influences more the targeted branch. Therefore, if $T = <n_{kl}, n_{k2}, ..., n>$ is a path navigated on a program input x, where x can be an array of 30 elements, the technique controls the influence of the elements of x on the nodes in the path (in terms of used variables), and the influence of each node n_k on the node following it until the targeted branch is touched. This way, for a certain target branch, the author only considers the influential input variables (elements of the array in our example) in the search technique (Korel, 1990b).

Though this approach improves the previous work (Miller & Spooner, 1976), it does not present an implementation of the data flow analysis. On the other hand, the data flow analysis is used only to choose the input variables influencing the branch to test and thus starting the search of required data for these variables. However, the author does not take advantage of dependencies to guide the search using the data flow analysis information. This is what we call data dependencies between the nodes of the program, and it can actually help the search converge faster to the required solution.

The local search used to find the test data can lead to local optimum solutions in the search space when trying to minimize the fitness function. In order to, overcome this problem, researchers investigated more sophisticated meta-heuristics such as the Simulated Annealing, Hill Climbing, Great Deluge Algorithm and many other search algorithms. We will discuss the work done on these algorithms in the following section.

2.7 Use of Metaheuristics Algorithms to Automate Test Case Generation

2.7.1 Metaheuristics Algorithms

A metaheuristic is a general algorithmic framework which can be applied to different optimization problems with relatively few modifications to make them adapted to a specific problem. Many optimization problems of practical as well as theoretical importance consist of the search for the "best" configuration of a set of variables to achieve some goals that depend on those variables. Metaheuristics are typically high-level strategies which guide an underlying, more problem specific heuristic, to increase their performance. The ultimate objective is to avoid the disadvantages of iterative improvement by allowing the local search to escape from local optima to which a local search typically gets stuck. This is achieved by either allowing worsening moves or generating new starting solutions for the local search in a more intelligent way than just providing random initial solutions(Stützle, 1998).

Many metaheuristic algorithms, called nature-inspired algorithms, are based on different natural phenomena, such as ants searching for food, evolution of species, or highlands escaping from flood. As an interpretation of nature, they are typically not perfect in finding the best or optimum goal, but are able to find what are called sub-optimal or near-optimal solutions. However, in doing so they trade-off quality with cost, thus achieving solutions which are very near to the best, but at much lower and affordable time and computation.

Meta-heurisitic algorithms have been applied to many software engineering activities (i.e. termed Search based Software Engineering) ranging from Requirement Engineering to Software Quality Assurance(Ghani & Clark, 2009). SBSE is playing a major role in Software Testing by introducing Search Based Test Data Generation (SBTDG) approach(Varshney & Mehrotra, 2013). SBSE applies metaheuristic algorithms like Simulated Annealing, Hill Climbing, Genetic Algorithm, Great Deluge, or Tabu Search in the relevant search space to find near-optimal solutions. In the search space, metaheuristic algorithms use a fitness function that compare current and neighbour solution and choose the better option based on the search goal. For test data generation, the objective function of the algorithm is configured based on the problem scope (in our case MC/DC criterion) to generate test data.

New metaheuristic algorithm like Late Acceptance Hill Climbing (E. K. Burke & Bykov, 2012; Bykov, 2011) are becoming very promising to researchers to solve searching problems. Known for quite some time, and perform very efficiently, viz. Great Deluge Algorithm (Dueck, 1993) was never implemented for automated test data generation satisfying MC/DC criterion.

In today's literature, several works use meta-heuristic algorithms as search tools to automate the test cases or test suite generation. Due to the computational complexity of the search problem, exact techniques like linear programming are mostly impractical for large scale software engineering problems and manual search is mostly impossible. Thus, Search Based Software Engineering (SBSE) is an approach to apply meta-heuristic search techniques like Genetic Algorithms, Simulated Annealing, and Tabu Search to seek solutions for combinatorial problems at a reasonable computational cost. In SBSE, we apply search techniques to search large search spaces, guided by a fitness function that compare solutions with respect to the search goal and determine which is the better solution and thus to direct the automated search into potentially promising areas of the search space (McMinn, 2004).

For test data generation, this involves the transformation of test criteria to objective functions. For each test criterion, a family of different objective functions is needed. The algorithm iterates then to generate the appropriate data to make the fitness function as close as possible to zero, meaning the algorithm found the data for the test case.

For each problem solved using meta-heuristic techniques, there are usually two main decisions of implementation. The first one being the encoding of the solution, i.e., the structure (e.g., array, tree), how many variables it has, their types, etc, and the second main decision is the transformation of the test criteria into a fitness function. The fitness function models the closeness of the input data to cover the criterion tested. It is usually calculated at the end of each iteration and it compares and contrasts the solutions with respect to the overall search goal to guide the search into a promising neighbourhood of the search space.

There are several types of meta-heuristic algorithms that were used in literature to automate the data generation. We will describe here works based on tabu search, simulated annealing, the Great Deluge algorithm and evolutionary techniques such as the genetic algorithm.

2.7.2 Evolutionary Testing

Evolutionary approaches are search algorithms tailored to automate and support testing activities, i.e., to generate test input data. They are often referred to as evolutionary based software testing or simply Evolutionary Testing (ET). Genetic algorithm (GA) is an ET algorithm.

2.7.2.1 Genetic Algorithm (GA)

In 1997, GA was used by Wegener et al. to test real-time systems for functional correctness. A common definition of a real-time system is that it must deliver the result within a specified time interval and this adds an extra constraint to the validation of such systems, namely that their temporal correctness must be checked (Wegener, Sthamer, Jones, & Eyres, 1997). The standard technique for real-time testing is the classification tree method; it was used to generate the test cases forming the objective of the search. The genetic algorithm aimed to find the longest execution time, and then the shortest of the real system response time. Wegener et al. concluded from their work that genetic algorithms are able to check large programs and they show considerable promise in establishing the validity of the temporal behaviour of real-time software(Wegener et al., 1997).

Tonella (Tonella, 2004) in a 2004 ACM Conference reported the use of GA to produce test cases automatically for unit testing of classes in a generic application condition. Test cases were formulated as chromosomes, which include information regarding the objects to create, the methods to invoke and the values to be used as input. The proposed methodology evolves test cases with the aim of maximizing a suitable measure quantifying coverage. The article describes implementation of the algorithm and its application to classes from the Java standard library. Usage of a genetic algorithm for the unit testing of classes was reported to be extremely powerful. Optimal coverage of the public cmethod branches was achieved within a reasonable computation time, and the resulting test suites obtained were quite compact. The study on the fault revealing capability of the generated test cases described in this paper, however, were admitted to be quite preliminary. The article also expresses the need to carry out additional experiments in future with more number of faults seeded into the class methods.

Nguyen et al. (Nguyen et al., 2012) argues that autonomy in software agents defies the complete control of the user, thus requiring a more thorough and accurate testing procedure testing of autonomous agents. This necessarily involves a wide range of test case contexts that can search for the most demanding test cases— even those that are not apparent to the developers or testers. The article address this problem by introducing an approach to testing autonomous agents that uses evolutionary optimisation to generate such demanding test cases. The authors propose a methodology to derive objective or fitness functions that regulate evolutionary algorithms. The approach is evaluated with two simulated autonomous agents. The reported results illustrate that the approach is effective in finding good test cases automatically.

In a very recent paper by Soltani, Panichella and van Deursen (Soltani, Panichella, & van Deursen, 2016), a new solution for automatic crash reproduction is proposed based on evolutionary unit test generation techniques. As a first step towards software debugging, reproduction of software crashes is a labour-intensive and time-consuming task. In this scenario, the paper tries to overcome the limitations adversely affecting the capabilities of existing state-of-the-art solutions in generating helpful tests that trigger specific execution paths. The proposed solution uses crash data from collected stack traces to guide search-based algorithms toward the generation of unit test cases that can reproduce the original crashes in case of a wide range of crashes. The methodology implemented is an extension of Evo Suite, which uses a suitable novel fitness function defined for crash reproduction. Results from a preliminary study on real crashes from Apache Commons libraries show that our solution can successfully reproduce crashes which are not reproducible by two other state-of-art techniques.

A recent article by Rathore et al.(Rathore, Bohara, Prashil, Prashanth, & Srivastava, 2011)presents a technique for automatic test-data generation in software testing. The proposed approach is based on a hybridization of genetic algorithm and the tabu search method. It combines the strength of both metaheuristic techniques and produces efficient results. The conventional approach for test-data generation using genetic algorithm is modified by applying a tabu search heuristic in the mutation step. It also incorporates backtracking process to move the search away from local optima. The experimental results show that the proposed hybrid algorithm is effective in providing test data and its performance is better than simple genetic algorithm.

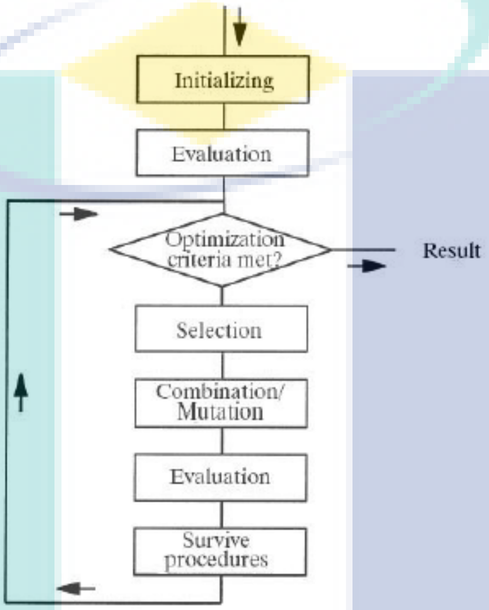


Figure 2.3 Block diagram of the genetic algorithm

The used GA's block diagram is illustrated in the algorithm iterates with a population of candidate solutions. It initializes with a randomly generated population then it evolves by combining and mutating the current generation in order to generate possible solutions. An evaluation is performed on the newly generated solutions and a selection technique is then used to transmit only the fittest individuals into the next generation. The algorithm iterates until a solution is found to satisfy the optimization criteria.

2.7.3 Local Search Techniques

Local search is a family of metaheuristic algorithms based on the concept of neighbourhood of the current configuration (solution). There are mainly two types of local search, the gradient descent techniques and more advanced techniques such as simulated annealing and Tabu search. The main idea of local search is to start with one initial solution and modify it iteratively following certain criteria. At each iteration, the current solution is modified according to some deterministic or probabilistic rule to produce a new candidate solution. The candidate can be accepted or rejected according to some given acceptance criteria. If accepted it becomes the valid solution for the next iteration. If the candidate is rejected, then the next iteration is carried out with the same existing solution. Usually the search continues till no further improvement is possible. In such a situation convergence is said to have been achieved, and the optimal solution is printed.

Local search algorithms rely on the neighbourhood of the current solution. To solve an optimization problem using local search algorithms, we need first to define the solution space $\boldsymbol{S}$, i.e., the search space of possible solutions, and the objective function $f(s)$ that evaluates a real value in $\mathbb{R}$ for each solution $s \in \boldsymbol{S}$ such that:

$$f : \boldsymbol{S} \to \mathbb{R}$$

The definition of a neighbourhood is very crucial for any local search algorithm. A function $\mathcal{N}$ is usually defined as $\mathcal{N}(s) \subseteq \mathcal{P}(\boldsymbol{S})$ that associates s with a subset of all possible solutions $\mathcal{P}(\boldsymbol{S})$ as 'neighbours' of the current solution s such that:

$$: \boldsymbol{S} \to \mathcal{P}(\boldsymbol{S})$$

In each iteration, the algorithm defines the set of neighbours of the current solution, selects one of the neighbours and makes it the new current solution.

We call a solution s in the solution neighbourhood $\mathcal{N}(s)$ a local minimum if

$$\forall \ s' \epsilon \, \mathcal{N}(s), \qquad f(s) \leq f(s')$$

Alternatively, we call s in the solution neighbourhood $\mathcal{N}(s)$ a local maximum if,

$$\forall \ s' \epsilon \mathcal{N}(s), \qquad f(s) \geq f(s')$$

A typical schema of a local search algorithm is shown in Figure 2.4.

```
1: Build initial configuration s
2: Best_S ⟵ s
3: Iterate
4:     Select s' from $\mathcal{N}$(s)
5:     s ⟵ s'          // not obligatory
6:     If (s > Best_S)
7:          Best_S ⟵ s
8: Return Best_S
```

Figure 2.4 Local Search Schema

There are different strategies for the choice of a neighbour solution s' from the possible neighbours $\mathcal{N}(s)$ of s at line 4 in Figure 2.4. Each local search metaheuristic algorithm applies a different neighbour selection strategy.

The affectation of the current solution by the neighbour solution at line 5 in Figure 2.4 is not mandatory. In fact in some local search algorithms such as the descent algorithm or the Hill-Climbing (HC) methodology (Appleby, Blake, & Newman, 1961), the neighbour solution becomes the current solution only if its fitness is better than the current solution. This kind of approach is called the ‘greedy’ approach (E. K. Burke & Bykov, 2012), where the algorithm looks for short-term benefit, and stays at the risk of losing sight of better solutions which could have been achievable through smaller immediate sacrifices or gambits. The HC methodology is reported to be very fast, but the disadvantage is that the algorithm can converge quickly to a local optimum and the search is stopped prematurely before reaching the global optima when the search space is similar to Figure 2.5. Such search spaces have more than one local optima, and are referred to as multimodal functions. There are several established methods to deal with such complex multimodal optimization problems, some of which are discussed below.

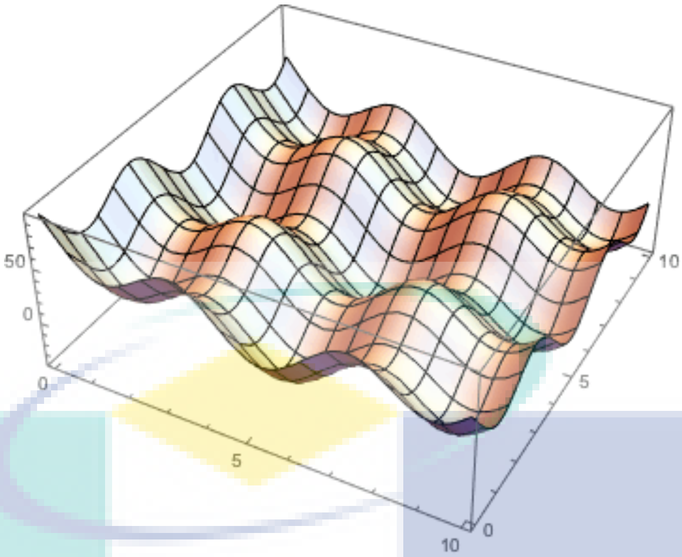


Figure 2.5 Local optima problem in local search

One technique to deal with the situation when the search gets stuck in some local optima consists of performing a random restart when an optimum is intermediately reached. The algorithm restarts the search several times, each time with a new randomly generated initial solution. This approach allows exploring different regions of the search space, however its inconvenience lies in the fact that the algorithm does not benefit from the knowledge acquired during the previous search. A second approach is to add a bit of randomness in the local search, thus, the algorithm can accept some of the degrading solutions on the short term, which might lead to a better optimum on the long run. An example of such an algorithm is simulated annealing. A third approach is to memorize the solutions already visited and ban them and their neighbourhoods in future searches, so that the algorithm is forced to try unexplored neighbours. This is the case of Tabu search.

2.7.3.1 Tabu Search

A technique of interest in software testing is Tabu Search (TS) proposed by Glover(F Glover, 1985). This method evaluates the complete set of possible modifications of the current solution and the candidate with the best cost is accepted. To avoid cycling, it was proposed to also reject solutions, which were already accepted at relatively recent previous iterations. For this purpose, TS maintains a list of previous solutions (or moves) known as the “tabu list”, where all elements are compared with the complete set of candidates at each iteration. The method has two interesting properties.

Firstly, it does not employ any version of a problem-dependent cooling schedule like simulated annealing (discussed in the following subsection). Secondly, as noted by Laguna and Glover (Fred Glover, 1997), the employing of a tabu list follows the idea of the “intelligent” use of information collected during the search. Later, this idea was expanded and called Adaptive Memory Programming (AMP). In addition to Tabu Search, the authors also considered the GA and ACO as examples of AMP(Taillard, Gambardella, Gendreau, & Potvin, 2001).

Tabu search-based approach has been proposed by Hwang, Yin and Yeh (Hwang, Yin, & Yeh, 2006) for more efficient composition of near-optimal test sheets from very large item banks, while meeting multiple assessment criteria. Based on the proposed approach, a computer-assisted testing system has been developed, and a series of experiments have been conducted to compare the efficiency and efficacy of this approach with other approaches. The experimental results show that the new approach is desirable for the composition of near-optimal test sheets from large item banks.

Díaz, Tuya, Blanco and Dolado (Díaz, Tuya, Blanco, & Dolado, 2008) present a tabu search metaheuristic algorithm for the automatic generation of structural software tests. It is a novel work since tabu search is applied to the automation of the test generation task, whereas previous works have used other techniques such as genetic algorithms. The developed test generator has a cost function for intensifying the search and another for diversifying the search that is used when the intensification is not successful. It also combines the use of memory with a backtracking process to avoid getting stuck in local minima. Evaluation of the generator was performed using complex programs under test and large ranges for input variables. Results show that the developed generator is both effective and efficient.

2.7.3.2 Simulated Annealing Algorithm

Ghani and Clark (Ghani & Clark, 2009) introduce an automatic framework to extend the capabilities of search based testing technique for MC/DC. The advantage of this framework is that it can be used to test stronger coverage criteria such as Multiple Condition Coverage and MC/DC. Simulated annealing optimization technique is used into this framework. The framework is compared with other tools for software testing;

Triangle, CalDate, Quadratic, Complex and Expint. The result shows that the search by this framework manages to get 100 % coverage for all tools except for *Expint*.

To overcome the limitations associated with local search optimum, Simulated Annealing (SA) was used as another type of meta-heuristic search algorithms. Tracey et al. proposed in 1998, an optimization-based framework to be applied to a number of structural testing problems (N. Tracey, Clark, Mander, & McDermid, 1998). Tracey's work focuses on branch coverage. Their goal is to search for program input which forces execution of the desired part of the software under test. For the search to succeed, a fitness function is needed to guide the search, relating a program input to a measure of how "good" the input is to achieve a certain test target. The fitness function returns good values for test-data that nearly executes the desired statement and bad values for test-data that is a long way from executing the desired statement. In general, the input domain of most programs is likely to be very large, and given the complexities of systems it is extremely unlikely that the fitness surface would be linear or continuous. The size and complexity of the search space therefore limits the effectiveness of simple gradient-descent or neighbourhood searches as they are likely to get stuck in locally optimal solutions and hence fail to find the desired test-data (N. Tracey et al., 1998). Thus, a more sophisticated approach is needed such as the SA. SA allows movements which worsen the value of the fitness function based on a control parameter known as the temperature. At the early stage of the search iterations, inferior solutions are accepted with relative freedom, but as the search progresses, accepting inferior solutions becomes more and more restricted. The aim of accepting these inferior solutions is to accept a short-term penalty in the hope of longer term rewards.

The fitness function designed by Tracey et al. evaluates to zero if the branch predicate evaluates to the desired condition and positive otherwise (N. Tracey et al., 1998). It is designed based on the structure of the system under test; for each predicate controlling the target node, if the target node is only reachable if the branch predicate is true then the fitness of the branch predicate is added to the overall fitness for the current test-data otherwise the fitness of (branch predicate) is used. For loop predicates, the desired number of iterations determines whether the fitness of the loop predicate or (loop predicate) is used. The simulated annealing search uses this to guide its generation of test-data until either it has successfully found test-data or until the search freezes and no further progress can be made (N. Tracey et al., 1998).

The automation framework was tested on small Ada 95 programs to cover the branch coverage criterion. The programs ranged from 20 to 200 lines of codes. The reported coverage percentage is 100% for all but one case; the failing case achieved 100% branch coverage in 40 out of the 50 trials. The search time of SA is 2 to 35 seconds. Unfortunately, the programs tested are not available and thus we were unable to verify their structural complexity. Moreover, no comparison with other search techniques performance is presented. Still, this work provides an automated platform for structural testing. We aim in our work to build a similar platform, however achieving the MC/DC coverage and not the branch coverage.

One of the most well-known metaheuristics which accepts slightly worse solutions in the beginning of the search is Simulated Annealing (SA) proposed by Kirkpatrik, Gellat and Vecci (Kirkpatrick, Gelatt, & Vecchi, 1983). The method models the physical process of heating a material followed by controlled cooling, which increases the grain size and consequently helps to remove certain defects like internal stresses, effectually minimizing the thermodynamic free energy of the system.

It is a stochastic algorithm, which accepts a worse candidate with the probability

$$P = \begin{cases} 1, & \text{if } f(s') \le f(s) \\ \exp\left[\dfrac{f(s') - f(s)}{T}\right], & \text{otherwise} \end{cases}$$

where $s$and $s'$are the current and candidate solutions respectively, and T is a control parameter called ‘temperature’ which varies as the search progresses characterizes the cooling schedule.

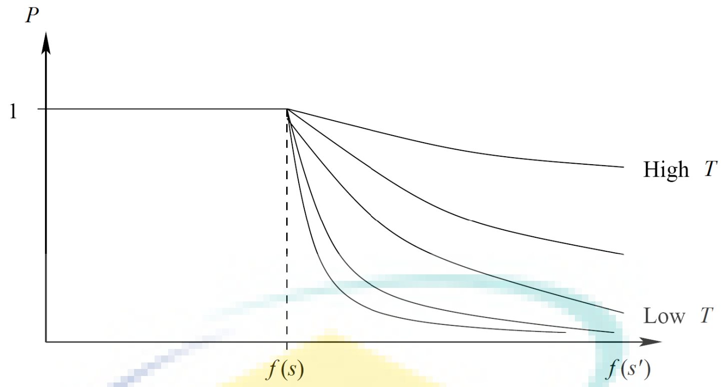


Figure 2.6 Simulated Annealing Algorithm. Initial Temperature to cooling down process.

Simulated annealing (SA) algorithm deals with an initial temperature, cooling rate, current solution and a neighbour solution. At the beginning, SA generates an initial current solution and start keeps searching for result in its search space. On every iterations, SA generates a neighbour solution *s'* based on its current solution *s*. Then, the fitness function process the current and neighbour solution to return results *f*(*s'*) and *f*(*s*) respectively. If the neighbour solution is better than the current solution, determined by the condition *f*(*s'*)>*f*(*s*), then SA accepts the neighbour solution and stores it as the current best solution. If current solution is better than the neighbour solution, then SA generates a probability *P*(*T*, *s'*, *s*) and a random number between 0 and 1. If probability is greater than the random number, only then the current solution will be stored as best solution. At the end of the loop, temperature will be reduced based on cooling rate. Once the temperature reaches sub-zero, i.e. $T \leq 0$, the algorithm stops searching. Steps of this algorithm are presented in Figure 2.7.

```
s ←GenerateInitialSolution()
T ← T0  // initial temperature
while termination condition not met do
    Select s' from 𝒩(s)
    if f(s') > f(s)
        s ←  s'
    else
        Accept s' as new solution with probability P(T, s', s)
    end if
Update T  // based on cooling rate
end while
```

Figure 2.7 Schema for Simulated Annealing

Different researchers have proposed different values of initial temperatures so that a certain percentage of worsening moves are accepted in the beginning. For example Johnson et al. proposed (D. S. Johnson, Aragon, McGeoch, & Schevon, 1989) to have T_0 between 40% and 90%, while Thomson and Dowsl and proposed 75% (Thompson & Dowsland, 1995), and Cohn and Fielding 95%(Cohn & Fielding, 1999). The final value of the temperature should be zero or at least close to zero in order to guarantee convergence. A popular cooling schedules called ‘geometric cooling’ is represented by the expression

$$T_i = T_{i\text{-}1} * \beta$$

i.e. the temperature at the i^{th} iteration is equal to the previous temperature $T_{i\text{-}1}$ multiplied by a user-defined cooling factor β ($0 < \beta < 1$). However, some authors have suggested the use of alternative functions, such as the ‘quadratic cooling schedule’ (Andersen, Vidal, & Iversen, 1993) or even temporary increases in the temperature, for example, adaptive cooling(Thompson & Dowsland, 1995) or reheating (Osman, 1993).

2.7.3.3 Great Deluge Algorithm

One further variant of hill climbing and simulated annealing is the great flood or great deluge algorithm first introduced by G. Dueck (Dueck, 1993). It is also termed threshold accepting. This follows a strategy similar to simulated annealing but often displays more rapid convergence. Instead of using probability to decide on a move when the cost is higher, a worse feasible solution is chosen if the cost is less than the current threshold. This threshold value is sometimes referred to as the water level which, in a profit maximizing problem, would be rising rather than falling (as is happening in this case). As the algorithm progresses, the threshold is reduced, moving it closer to the optimal cost. This algorithm has not been used for finding covering arrays and so it may provide some interesting new results(Cohen, 2004).

Bryce and Colbourn (Cohen, 2004) compare the results of a simple greedy algorithm called density algorithm with the results from four heuristic search algorithms, namely simple hill climbing, SA, TS, and the great deluge algorithm to construct a methodology which dispenses one test at a time. The proposed algorithm attempts to maximize the number of*t*-tuples covered by the earliest tests so that if a tester only runs a partial test suite, the test should include as many *t*-tuples as possible. Heuristic search is shown to provide effective methods for achieving such coverage.

The great deluge algorithm (GDA) is originally proposed by Dueck (Dueck, 1993). GDA is developed while further experimenting with the threshold accepting (TA) algorithm, which produced better results than the SA algorithm. It is a mathematical allegory of the great deluge that was described in the Book of Genesis of the Bible, and believed to have occurred sometime around 3000 BC. The algorithm walks around in a country in search of dry land, where it begins to rain without end. However, it never steps beyond the ever-increasing water level, and at last reaches the highest points in the country signifying the optimum or near-optimum solution. The basic idea is very similar to simulated annealing (Aarts & Korst, 1991). However, the difference lies in their acceptance rules for worse intermediate solutions. The GDA is a one-parameter algorithm, where it is necessary to choose only one parameter for the best possible performance of the metaheuristic, whereas the cooling schedule of SA is dependent on a sequence of parameters (Dueck, 1993). The algorithm was first applied to the 442 cities travelling salesman (Grötschel’s) problem, but later saw applications in a host of other practical domains.

The control parameter in GDA is called ‘water level’ which plays the part of the cooling schedule in simulated annealing. Its value is set to a value higher than the expected penalty of the best solution at the start of the search. Then the water level is decreased during the search until it reaches a value of zero. In classical GDA, it was recommended that the initial value of water level be equal to the initial cost function and that it should be lowered linearly during the search. However, other variations were also proposed, such as: initialization with a higher level(E. Burke, Elliman, Ford, & Weare, 1995), non-linear level lowering(Obit, Landa-Silva, Ouelhadj, & Sevaux, 2009) or reheating mechanisms(Mcmullan, 2007) . During the search the algorithm explores solutions in the neighbourhood of the best solution. A new solution with a lower penalty is accepted straight away replacing the best solution. A new solution with a higher penalty is accepted only if this worse penalty is not higher than the current water level.

```
Choose an initial solution s
Choose WL     // initial water level
Choose Up     // initial rain speed
for i = 0 to n  // no. of iterations
    Generate a small stochastic perturbation s' of the solution s
    if f(s') > WL
s ⟵ s'
    end if
WL = WL + Up
end for
```

Figure 2.8 Schema for Great Deluge Algorithm

The value of the rain speed *Up* is the single parameter which trades off quality of the solution with computation time. With a high rain speed the algorithm converges very fast, but the results are poor. A slower rain speed gives the algorithm more time to search for high land, and eventually, running for longer computation time, returns much better results. The optimum speed for a particular problem depends on its complexity and the available resources.

2.7.3.4 Late Acceptance Hill Climbing

Late Acceptance Hill Climbing (LAHC) algorithm was first introduced by Burke and Bykov in 2008 (E. K. Burke & Bykov, 2008). This algorithm is developed based on the general Hill Climbing (HC) algorithm. While HC compares the candidate solution with the current solution for acceptance, LAHC delays this comparison till the candidate is compared with a solution which was 'current' several steps before. In LAHC, each current solution still takes on the role of an acceptance benchmark, but only to be used in some later step. The net effect of this 'late acceptance' criteria is that LAHC also allows some worsening moves, which can help to avoid local minima/maxima. Secondly the method being a derivative of HC, one of the simplest iterative search algorithm in optimization literature till today removes its tendency of getting stuck locally, but still maintains the original simplicity through using a single parameter denoting the length of the list to be remembered. The LCHC was first applied to the exam timetabling problem, and tested on 13 different benchmark problems taken from the University of Toronto collection with different values of the list length *L*. The experiment with LCHC was carried out alongside experiments on the same problems with simple HC. It was reported that an increase in *L* increases the computational cost and simultaneously helps to achieve better solutions.

The algorithm was later applied to the liner shipping fleet repositioning problem by Tierney (Tierney, 2013) in 2013. The class of problem belongs to the shipping industry, which involves the movement of vessels between routes in a liner shipping network subject to complex costs and timing restrictions. The author carries out experiments with 88 different instances, using both with LAHC and SA, taking care to maintain the same neighbourhoods and tuning processes at every instance. The paper concludes with the observation that LAHC fails to overcome the local optima obstacle as effectively as SA. The reason of failure of LAHC in solving a difficult and complicated real-world problem like the shipping fleet problem was attributed to be the possible simplicity of the algorithm in comparison to SA. Possibility of future work was indicated towards hybridizations of LAHC and SA, as well as extensions to LAHC proposed by Burke and Bykov (E. K. Burke & Bykov, 2012). In a paper by Yuan, Zhang and Shao (Yuan, Zhang, & Shao, 2015) published in 2015, an integer programming model is constructed and solved for the two-sided assembly line balancing problem containing the three additional constraints of zoning, positional, and synchronism, over and above the fundamental constraints of the conventional line balancing problem which make the problem quite complex. The integer programming model is reported to be suitable for solving small-sized problems, but does not fare well in solving problems of large size. compared with results obtained using the late acceptance hill-climbing algorithm. The proposed algorithm is tested on four small-sized problems and three large-sized problems. The experiment validates the effectiveness of the LAHC algorithm.

Cohen, Gibbons, Mugridge and Colbourn (Cohen, Gibbons, Mugridge, & Colbourn, 2003) study the mixed level covering array and propose a new object, called the variable strength covering array, which provides a more robust environment for software interaction testing. Initial results are presented suggesting that heuristic search techniques such as tabu search, simulated annealing are more effective than some of the known greedy methods like simple hill climbing, for finding smaller sized test suites. The authors present a discussion of an integrated approach for finding covering arrays, discussing how application of these techniques can be used to construct variable strength arrays. The principal idea was to use pair-wise or *t*-way testing to provide a guarantee that all pairs or *t*-way combinations are tested together.

The Late Acceptance Hill-Climbing algorithm (LAHC) is a recently invented general-purpose metaheuristic. It was first presented at the PATAT 2008 conference by Burke and Bykov (Appleby et al., 1961), where the method was applied to examination timetabling problems. Since then LAHC has become highly popular and seen application in different problems like the travelling salesman problem, scheduling problems in the manufacturing (Yuan et al., 2015) or shipping industry (Tierney, 2013), and has even been extended to solve multi-objective optimization problems (Vancroonenburg & Wauters, 2013).

The LAHC start from a single stochastic initial solution and at each iteration it evaluates a new candidate in order to accept or reject it. To employ its acceptance rule, LAHC maintains a fixed length list of previous values of the current cost function. The candidate cost is compared with the last element of the list and if not worse, then accepted. After the acceptance procedure, the cost of the new current solution is inserted into the beginning of the list and the last element is removed from the end of the list. Note that the inserted current cost is equal to the candidate's cost in the case of accepting only, but in the case of rejecting it is equal to the previous value.

```
Produce an initial solution s
Calculate initial cost function C(s)
for all k ∈ (0, 1, 2, …, L-1) do
    Ĉk ← C(s)
end for
I ← 0                // assign the initial number of iteration
do until a chosen stopping condition
    Construct a candidate solution s'
    Calculate its cost function C(s')
    v ← I mod L
    if C(s') ≤ Ĉv then
        s ← s'           // accept candidate
        Ĉv ← C(s)        // insert cost value into the list
    I ← I + 1            // increment the number of iteration
end do
```

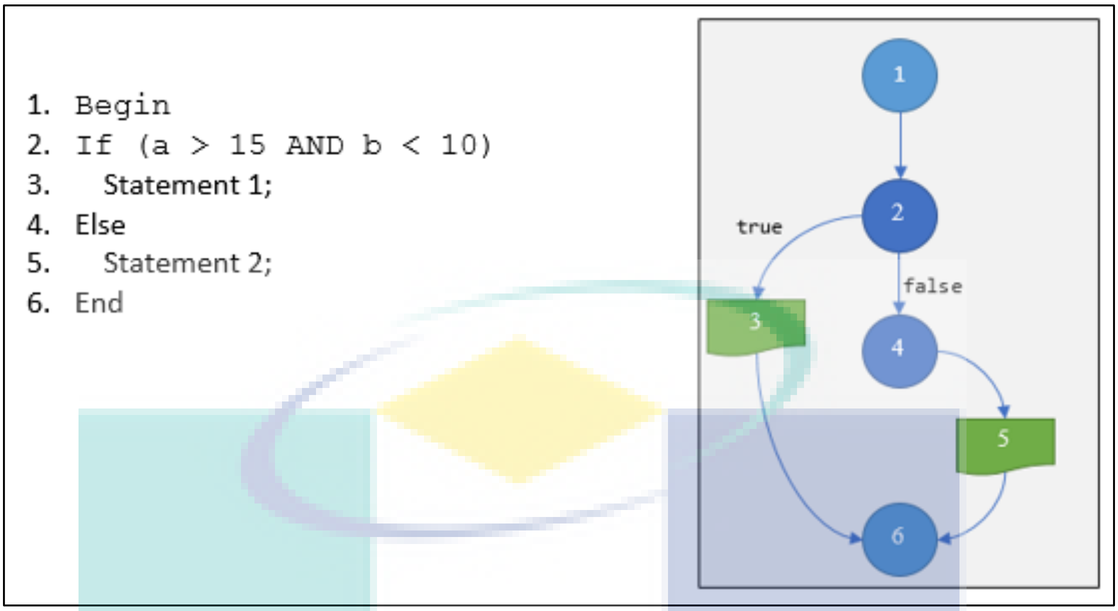
Figure 2.9 Schema for the LAHC algorithm

Source: E. K. Burke & Bykov (2008)

The novelty of the LAHC is a new acceptance condition. Its basic idea is to accept the candidates with cost function better (or equal) than the cost of the solution, which was the current several iterations before. Thus, the LAHC maintains the fitness array (memory size) $\hat{C}_k$, ($k = 1, 2, ..., L$) of a particular length L, which contains previous current costs. At each iteration, the candidate cost $C(s')$ is compared with the last element of the fitness array $\hat{C}_L$. After the acceptance decision, the current cost is inserted into the beginning of the fitness array for later comparison, and the last element is removed from the fitness array. Steps of LAHC algorithm are presented in Figure 2.9.

2.8 Masking Problem and Modified Condition / Decision Coverage (MC/DC) Criterion

To control a flow of software functionality, control flow operator is used such as AND, OR and NOT operators. For example, consider two predicates `(a AND b)` and `(a OR b)`. Result of `(a AND b)` is always false when a is false, regardless the value of b and vice versa. For result of `(a OR b)`, the result is always true, only if a is true, regardless the value of b and vice versa. In this situation, a and b is masking each other. The situation is visualized in Figure 2.10.



```
1. Begin
2. If (a > 15 AND b < 10)
3.    Statement 1;
4. Else
5.    Statement 2;
6. End
```

Figure 2.10 Simple function with AND condition

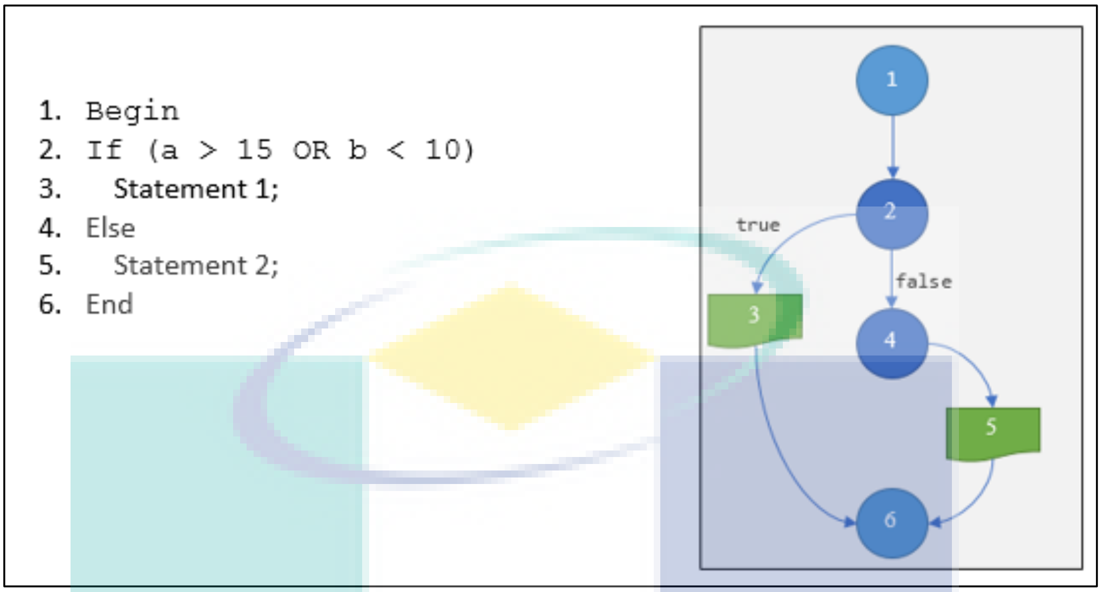
Considering a = 20and b = 8We can get a control

- Path i: 1, 2, 3, 6

Again, considering a = 12and b = 8 We can get another control

- Path ii: 1, 2, 4, 5, 6

All paths are covered without changing the value of b.



```
1. Begin
2. If (a > 15 OR b < 10)
3.    Statement 1;
4. Else
5.    Statement 2;
6. End
```

Figure 2.11 Simple function with OR condition and control flow graph

To perform the same inspection for OR operator, considering: a = 20 andb = 8, we can get control flow path

- Path i: 1, 2, 3, 6

Yet again Considering a = 12 and b = 8, the control flow path will be

- Path 1: 1, 2, 4, 5, 6

All paths are covered without changing the value of b.

In Figure 2.10 and Figure 2.11, it is clearly visible that for both AND and OR operator, decision coverage is 100% even without need to change the value of b. Here a is always masking b and giving a misleading coverage. To illustrate further, consider the nested if statements for both AND and OR operator as shown in Figure 2.11.

```
1. Begin
2. If (a > 15)
3.     If (b < 10)
4.          Statement 1;
5.     Else
6.          Statement 2;
7. Else
8.     Statement 2;
9. End
```

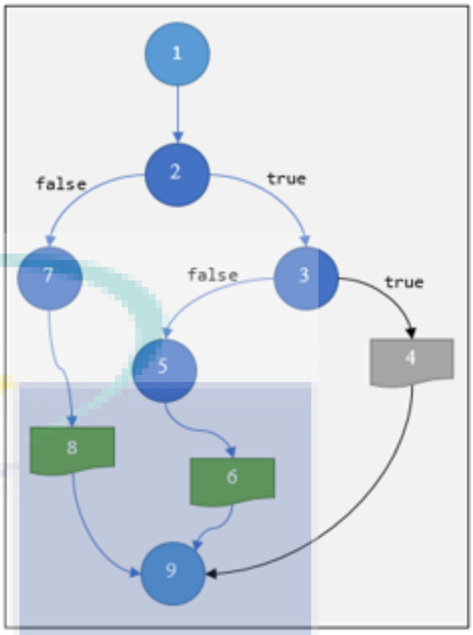
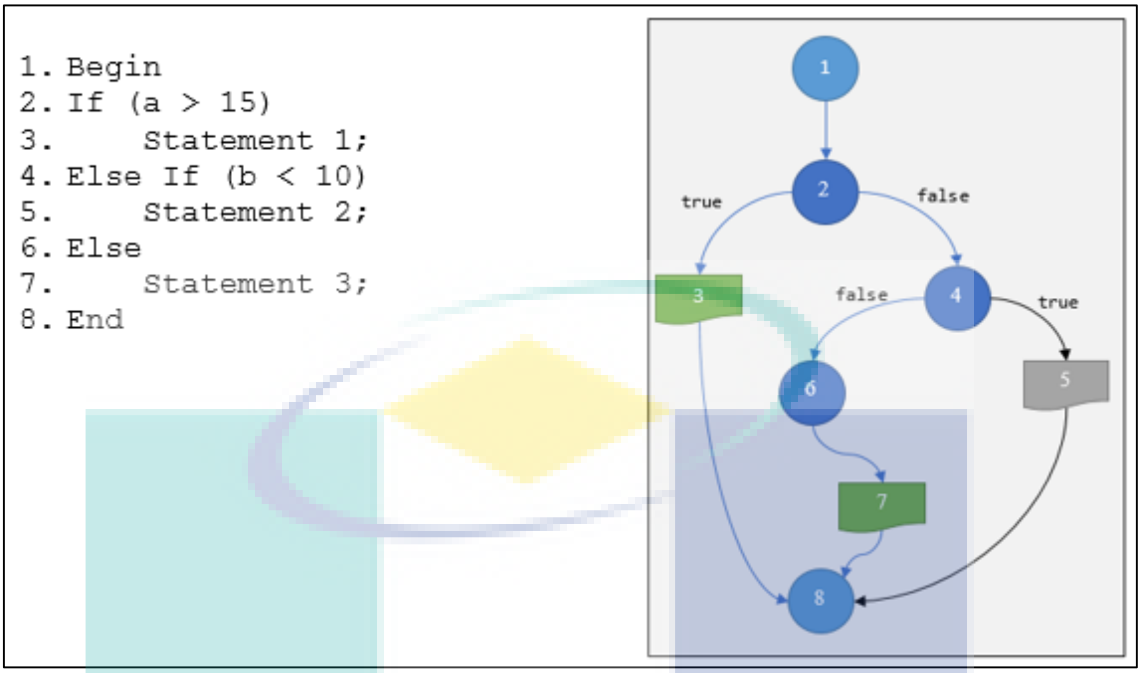


Figure 2.12 Equivalent If statement for AND operator of Figure 2.10

We have successfully identified new control flow paths. New paths of Figure 2.12 are:

- Path i: 1, 2, 7, 8, 9
- Path ii: 1, 2, 3, 4, 9
- Path iii: 1, 2, 3, 5, 6, 9



```
1. Begin
2. If (a > 15)
3.     Statement 1;
4. Else If (b < 10)
5.     Statement 2;
6. Else
7.     Statement 3;
8. End
```

Figure 2.13 Equivalent If statement for OR operator of Figure 2.11

Newly identified Control flow paths of Figure 2.13 are:

- Path i: 1, 2, 3, 8
- Path ii: 1, 2, 4, 6, 7, 9
- Path iii: 1, 2, 4, 5, 9

In Figure 2.12, When same inputs are given for a AND b (a=20, b=8 and a=12, b=8), *Path ii: 1, 2, 3, 4, 9* remain uncovered.

Using same inputs in Figure 2.13 for OR based control flow, *Path iii: 1, 2, 4, 5, 9* remains uncovered.

The main concern here is how to cover the uncovered paths and at the same time eliminate the masking problem of both AND and OR operations.

Table 2.2 A Domestic Case Studies Probability to Overcome Software Failures

Coverage Criteria	Statement Coverage	Decision Coverage	Condition Coverage	Condition / Decision Coverage	MC/DC	Multiple Condition Coverage
Every point of entry and exit in the program has been invoked at least once		•	•	•	•	•
Every statement in the program has been invoked at least once	•					•
Every decision in the program has taken all possible outcomes at least once		•		•	•	•
Every condition in a decision in the program has taken all possible outcomes at least once			•	•	•	•
Every condition in a decision has been shown to independently affect that decision’s outcome					•	•
Every combination of condition outcomes within a decision has been invoked at least once						•

Source: Hayhurst et al. (2001)

Referring to Table 2.2, Multiple Condition Coverage (aka Exhaustive testing) is most desirable solution to solve masking problem but this is only appropriate for small inputs. However, considering MCC is practically infeasible specially when the combinations are large. Here, the number of conditions grew with 2^n where n is the number of Boolean variables.

Condition coverage (CC) and Condition/Decision coverage(C/DC) are also possible. CC dictates that every condition in a decision has taken all possible outcomes at least once. C/DC requires CC and dictates the true and false decision outcome at least once. Despite being useful, CC and C/DC does not consider independence as the criteria for selecting test cases. Statement coverage and Decision coverage covers only small part of coverage criteria.

It is clear that MC/DC is the most viable alternative but with significantly reduced test size as compared to MCC. Here, MC/DC dictates that each condition within a predicate can independently influence the outcome of the decision. MC/DC is a stricter form of decision coverage. For decision coverage, each decision statement must evaluate to true on some execution of the program and must evaluate to false on some execution of the program. MC/DC, however, requires execution coverage at the condition level. Along with Hayhurst et al (Hayhurst et al., 2001), other researchers (Awedikian et al., 2009; Chang & Huang, 2007; Ghani & Clark, 2009; Jones & Harrold, 2003) also suggested to use MC/DC to solve masking problem.

The aim of the MC/DC criterion is to confirm that each condition in a decision affect properly the outcome of the decision. Once the following statements bellow is confirmed, the testing will achieve MC/DC criterion:

1. Each entry and exit point is invoked

2. Each decision tries every possible outcome

3. Each condition in a decision takes on every possible outcome

4. Each condition in a decision is shown to independently affect the outcome of the decision.

Considering the `if` statement in Figure 2.10, containing the decision `if(a>15 AND b<10)`, MC/DC criterion requires to prove that each part of the decision `(a>15)` and `(b<10)` will affect properly the outcome of this decision. To test this program, we need (i) `a=true && b=true`, (ii) `a=true && b=false`, and (iii) `a=false && b=true` or three test cases as per the MC/DC criterion. Our goal here is to generate these test cases automatically from a given equivalent Boolean expression e.g. `(a&&b)`.

The MC/DC criterion is suggested for use in structural testing to solve Masking problem (also mentioned in previous section). More importantly this is a required testing criterion as mentioned in DO-178B safety assessment document. The document divides the software into multiple levels in a decreasing order of safety criticality, from level A to D. Level A is most safety critical which is defined as "*Where a software/hardware failure would cause and/or contribute to a catastrophic failure of the aircraft flight control systems*" by the FAA (Hayhurst et al., 2001).

For a better understanding of MC/DC criterion, Hayhurst et al. one may refers to the explanation of MC/DC in "*A Practical Tutorial on Modified Condition/ Decision Coverage*" (Hayhurst et al., 2001). However, implementation of MC/DC is left as an open choice to researchers.

Chilenski and Miller(Chilenski & Miller, 1994) developed MC/DC criteria to achieve a degree of confidence in the software with fewer test cases but same effective as exhaustive testing (Multiple Condition Coverage). MC/DC becomes an important requirement for RTCA/DO-178B document, Software Considerations in Airborne Systems and Equipment Certification (L. A. Johnson, 1998), that is the primary processes used by aviation software developers to achieve FAA approval of airborne computer software(Authority, 1992; Chilenski & Miller, 1994). DO-178B describes software life cycle activities, design considerations and enumerates sets of objectives for the software development life cycle processes. The objectives are set based on the software level determined by a system safety assessment. For level A software, tests should achieve Modified Condition/Decision Coverage (MC/DC).

Achieving MC/DC criterion is a complicated task. Hayhurst et al. published a tutorial (Hayhurst et al., 2001) and a practical approach (Hayhurst & Veerhusen, 2001) on MC/DC in 2001 for better understanding of it. Here Hayhurst provided detailed definition of MC/DC and defined scope of it as in a Boolean expression `(a AND b)`or`(a AND c)`, `a`, `b` and `c` Boolean variables are 3 inputs only that contains 4 conditions (first `a`, `b`, `c` and second `a`) because each occurrence of `a` is considered as unique condition. They also discussed how to test AND, OR and NOT logical operators and how to evaluate the result. With the help of their tutorial, MC/DC pairs can be identified although for large input involving complex conditions, it will be hard for human to identify. Thus, the need of automation is required to simplify the process.

Jones and Harrold (Jones & Harrold, 2003) have proposed an algorithm for MC/DC with prioritization. There are four steps involved to reduce the number of test cases. The steps are: removing uncovered pairs, identifying test cases, assigning test cases contributions and removing weakest test cases. Here, prioritization involves two steps which are selecting the highest entity coverage and choosing the highest contribution values test cases.

In another work Jun-Ru and Chin-Yu (Chang & Huang, 2007) usefully exploit n-cube graph in order to generate appropriate MC/DC compliant test data. In this case, the vertex of the cube represents the resultant Boolean enumeration for predicates under evaluation. Each vertex is traversed and arranged and evaluated using Gray Code sequence ordering until all the required sequences are covered. A tool called TASTE (Tool for Automatic Software Regression Testing) has been developed as a result.

Kandl and Kirner exploit MC/DC to automotive domain. The goal of their study (Kandl & Kirner, 2011) is to inspect the error detection rate of a set of test that attains maximum possible MC/DC coverage. The first stage was done by generating the test cases. Here, test cases are generated using a model checker followed by transforming the program into three different errors circumstances. The results proved that fewer errors were detected when a system was tested with a set of test that attains maximum possible MC/DC on the code.

2.9 Summary

Search Based Software Engineering (SBSE) has emerged to be a widening field of study and active research from the start of the millennium. As a part of SBSE Search Based Software Testing (SBST) claims a major share of the efforts of researchers. The different categories of SBST include structural testing (also called white box testing), functional (black box) testing and non-functional (grey box) testing. Structural or white box testing reveals the most of the bugs as the code is visible to the tester, making it possible to indicate which part of it contain bugs to be removed.

Different metaheuristics have been seen to perform well in generating a test suite for different testing conditions including modified condition/ decision coverage (MC/DC). Of them hill climbing (HC) being the most naïve, but is at the same time the simplest. Different worsening moves were made possible by tabu search (TS), simulated annealing (SA), and the great deluge algorithm (GDA), which help the solution to move to other less-explored areas of the search space when it gets stuck in some local optima. Though the introduction of complexity in these algorithms has produced better results, the simplicity of the greedy hill climbing algorithm seems to have been lost in such sophistication. Another advancement in this line is the application of evolution-based algorithms like Genetic Algorithm (GA) or the Bat-Inspired algorithm to software testing problem.

However, late trends in SBST have seen a refuge to the old Hill Climbing criteria, but with a delay in comparison of the candidate solution with the current solution a few iterations back. This algorithm calls itself the Late Acceptance Hill Climbing (LAHC) Algorithm. Several algorithms like GDA and LAHC has seen little application to the software testing problem, particularly in testing the MC/DC criteria.

CHAPTER 3

METHODOLOGY

3.1 Introduction

In the previous chapter, the review of existing work has been presented. This chapter will elaborate the methods of neighbourhood-based algorithms and approach steps of generating MC/DC pairs by modifying four selected neighbourhood-based algorithms.

3.1.1 Justification of Neighbourhood-based metaheuristics algorithm

Solutions of MC/DC always found in pairs. By changing one value of a predicate, next MC/DC pair can be found. Neighbourhood-based meta-heuristics algorithms works exactly in same pattern. These algorithms start the heuristic from a random position and move to next position by changing the value of current position. This similarity of finding solution makes all neighbourhood-based algorithms a better candidate to apply to find MC/DC solutions.

3.2 Algorithm Implementation Design

Results of a 1999 survey of the aviation software industry showed that more than 75% of the survey respondents stated that meeting the MC/DC requirement in DO-178B was difficult, and 74% of the respondents said the cost was either substantial or nearly prohibitive (Hayhurst & Veerhusen, 2001; Hayhurst et al., 2001). In fact, the main challenge when trying to achieve the MC/DC coverage is to overcome the complexity of the code under test; it is not sufficient to generate test data for a program's decisions isolated, rather test data should be appropriately chosen to reach the targeted decisions and to achieve the relative test cases.

The most common approach to analyse the structure of the code under test is to extract its control flow graph (CFG). In fact, most of the structural testing approaches rely on the CFG to measure coverage and guide the search for the test input data. We can cite (Korel, 1990a) as examples the work of Korel in 1990 on branch coverage, the work of Baresel (Harman, Hu, Hierons, Baresel, & Sthamer, 2002) in 2002 on structural testing using evolutionary testing relying on the CFG to build the fitness function and the work of McMinn (McMinn, 2004) in 2004 also using the CFG to build the fitness function for branch testing MC/DC Test case generation Algorithms.

3.2.1 Control Flow Graph

A CFG is a graph representing the program structure. The CFG nodes represent computations. While in some CFG forms, a node represents one statement of code, in other CFG forms, a node can represent a code segment depending on the convention used; a segment being one or more lexically contiguous statements with no conditionally executed statements in it (Binder, 2000). Nodes can be named or numbered by any useful convention.

The edges (also called branches) represent the flow of control, which is usually a conditional transfer of control between a node and another one. An edge connects two nodes, representing the entry into and the exit from the statement. The entry point of a program is represented by an entry node with no incoming edges. The exit point of a program on the other hand is represented by the exit node with no outbound edges (Binder, 2000).

The CFG is an essential for the MC/DC testing because the flow of control is directed by the conditional nodes in the CFG; these nodes being predicate expressions such as an “`if`”, “`while`”, “`do until`”, etc. The CFG of a program is the fundamental structure required to guide the input data into the correct path to reach a targeted decision.

```
public Integer calculate(Integer x, Integer y, Integer z)
{
    Integer result = -1;
    Boolean fail = false;
    if (x < 0 || y < 0 || z < 0)
    {
        fail = true;
    }
    else
    {
        fail = false;
    }
    if ( ! fail )
    {
        x = y;
        result = 0;
    }
    if (result == -1);
    {
        if (z == 0);
        {
            result = x + y;
        }
        else
        {
            if (z > x && z > y || z > x + y)
            {
                result = z;
            }
            else
            {
                result = x + y
            }
        }
    }
    return result;
}
```

Figure 3.1 Calculation function

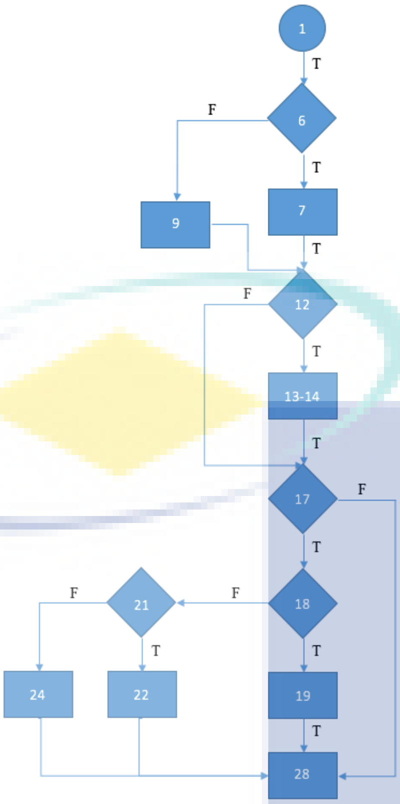


Figure 3.2 CFG of calculation method

The CFG of the `calculate()` method is shown in Figure 3.1. To reach the node 19 of Figure 3.2 for example, nodes 1 to 3 are traversed, then either the true edge or the false edge of both decisions nodes 6 and 12 can be traversed. However, the true edges of nodes 17 and 18 must be traversed to reach 14.

3.2.2 Decision coverage and MC/DC coverage

Decision coverage, also known as branch coverage, is achieved when each edge in a CFG is covered, and thus every edge from a decision node is traversed at least once. Ensuring that all decisions in the CFG are tested at least once implies the necessity to reach the decisions first. Let us assume that we want to test the decision at line 16 in the Calculate method in Figure 3.1. At a first look at the CFG, we can deduce that the input data generated must traverse the true branch of the decision at line 17 and the else branch of the decision at line 18 to reach the target decision. We can deduce that our target decision at line 21 is dependent on the flow of control through the decisions 17 and 18. We call such dependencies control dependencies. Moreover, a test data diverging away from the target at line 18 would be closer to the target from a test data diverging at line

17. In general, to automate the search for test data reaching a target statement, we need a cost function that determines which test data is closer to reach the target node. The cost function verifies for each test data how many controlling nodes were traversed in the required manner. The more traversed controlling nodes the better the cost function. This cost function is the Control Dependencies fitness function.

The problem with the search relying solely on the control dependencies between nodes is that it ignores prior statements that need to be executed first to make the path feasible to reach the target. Going back to our targeted decision at line 21, even though this decision does not appear to depend on decisions at lines 6 and 12, following the CFG, the outcome of these decisions play a decisive role in reaching our target. In fact, the variable "`result`" used at line 17 depends on the true branch of the decision at line 12. In turn, the decision at line 12 depends on the variable "`fail`", which is modified in the else branch of the decision at line 6.

As proposed by (McMinn & Holcombe, 2006) for branch coverage, we will integrate both cost functions for MC/DC coverage. The integrated fitness functions form the Approach level.

Assuming now that we reach the target decision with a test data xi, then we need to verify if xi achieves one of the MC/DC test goals. Moreover, if two test data xi and yi reach the target decision, but none of them achieve an MC/DC condition, a new cost function is needed to measure which of two test data is closer to achieve the test goal t the condition. In this case, the evaluation function relies on the structure of the target decision and the test case at hand. Such an evaluation function is called Branch fitness function.

3.3 MC/DC Pair Generation using Neighbourhood-based Algorithm

The first strategy used to automate the test data generation for structural testing was local search used by Miller and Spooner in 1976 (Miller & Spooner, 1976). A solution in a testing problem is an input test data. The objective function for a specific target in the program under test is the fitness function generated for each test case. Thus, applying local search to the MC/DC data generation problem, the algorithm evolves one test input data as a possible solution, improving it in iterations, aiming to reach a test data that would achieve the MC/DC test goal selected.

3.3.1 Stopping Criteria

The goal of the local search being minimization of the fitness function, the first stopping criterion is the fitness function of an individual found equal to zero; in this case the test data achieved the test goal. The algorithm selects another test goal and restarts the search, until all MC/DC test goals for the target are reached. If after a maximum allowed number of iterations, the algorithm fails to find a test data with a zero fitness function, the search is forced to stop so that it does not loop forever. In this case, it is either that the test goal is impossible to achieve or that the algorithm just failed in its search. Either way, the test goal is reported as failed.

3.3.2 Hill Climbing

Hill Climbing (Awedikian, 2009) algorithm deals with current and neighbour solution. At the beginning of the search, initial current solution is generated. An index is used to control the loop. The loop continues to search in the search space until the index reaches the stopping condition that is iterator in this case. In every iteration of the loop, a neighbour solution is generated based on current solution. The fitness function compares the current solution with neighbour solution and return the result. If fitness function is true which mean the neighbour solution is better than the current solution, then current solution will be replaced by neighbour solution. If the fitness function is false, the value of the current solution remains same and in next iteration it generates another neighbour solution from the same current solution.

The pseudo code of Hill Climbing (Awedikian, 2009) algorithm is given at Figure 3.3. The algorithm start from line 1 (indicated as 1:) by initiating `result` array and variable initiation continues until line 3. In line 2 and 3, the function initiate `mcdc_pairs` and `iterator` variable. The algorithm starts with an outer loop at line 4. This outer loop continues until all the predicates in the expression is executed. Before starting the search algorithm, a routine randomly chooses an initial current solution **s** in line 5. An `index` variable is initiated at line 6 to control the algorithm loop. The search loop at line 7 continue to iterate until the `index` is less then `iterator`. In each iteration of the algorithm, at line 8, a neighbour solution η is generated based on current solution **s**. At line 10, a fitness function compares the current **s** and neighbour solution η and return result as `T` or `F`. If condition is `T` there, that means the neighbour solution η is better than current solution s. Then value of neighbour solution η is set to current solution **s**, both current **s** and neighbour solution η is appended as `mcdc_pair` which is our test case (line 12:15) and the `index` is increased at line 17. If the condition is `F`, that means neighbour solution η is worst then current solution **s** and then index is increased and in next iteration a new neighbour is generated from the previous current solution. At the end of the function, the `mcdc_pairs` variable will have our expected test suite.

```
 1: Begin
 2:    Array result
 3:    Array mcdc_pairs
 4:    Integer iterator
 5:    While result size < expression size
 6:            Generate current solution s
 7:            Integer index
 8:            While index < iterator
 9:                    Generate move neighbor ∈η(s)
10:                    Where η is neighbor solution.
11:                    If ( ƒ(s) – ƒ(η) ) < 0
12:                            Set result = index(expression)
13:                            Set mcdc_pairs = mcdc_pairs(s)
14:                            Set mcdc_pairs = mcdc_pairs(η)
15:                            Set s = η
16:
17:                    End if
18:                    Set index++
19:            End While
20:    End While
21: End
```

Figure 3.3 Pseudo Code of Hill Climbing Algorithm

3.3.3 Great Deluge Algorithm

In our implementation of Great Deluge algorithm, the step starts from step 0 and the loop is completed while the `step size` becomes ≥ the current `expression size`. Variable `expression` contains the Boolean expression. While our outer loop start execution, we generate an initial current solution, set a `current_water_level` (line 10) and flood_level = initial_water_level (line 11) which was set at the beginning. At line 12, the inner while loop starts and continue until (current_water_level<rain_drop_speed) AND (flood_level ≤delta_level). Inside the inner loop at line 13, we generate a neighbour solution based on current solution. Then in line 15 the fitness function decides if the neighbour solution is better than the current solution. If neighbour solution is better than current solution, then assign neighbour solution to current solution, add current and neighbour solution to mcdc_pairs collection and add the current expression position to step (Line17-20). After the fitness function check, increase the current_water_level and decrease the flood_level as per water_level. While the outer loop completes its first iteration, it moves to next expression variable to search and start with generating a new current solution. By the time the outer loop complete the heuristic in Boolean expression, we have our list of mcdc_pairs as test suite and each pair is considered as test case.

```
1: Begin
 2:    Array mcdc_pairs
 3:    Float initial_water_level iwl
 4:    Float rain_drop_speed rdp
 5:    Integer step = 0
 6:    Float initial_water_level il
 7:    Float delta_level dl
 8:    Float water_level = (iwl - dl) / rdp
 9:    While step size ≤ expression size
10:    Generate current solution s
11:           Integer current_water_level cwl
12:           Float flood_level fl = iwl
13:           While ((cwl < rds) AND (fl ≤ dl))
14:                  Generate move neighbor ∈η(s)
15:                  Where η is neighbor solution.
16:                  If ( ƒ(s) - ƒ(η)) < 0
17:                         If step ∉ index(expression)
18:                         Set step = index(expression)
19:                         Set mcdc_pairs = mcdc_pairs(s)
20:                         Set mcdc_pairs = mcdc_pairs(η)
21:                                Set s = η
22:                         End If
23:                  End If
24:                  Set index++
25:                  Set fl = fl - water_level
26:           End while
27:    End while
28: End
```

Figure 3.4 Great Deluge Algorithm Pseudo Code

3.3.4 Simulated Annealing

The algorithm takes `initial_temperature` and `cooling_rate` as parameters. The '`expression`' in line 4 contains the Boolean expression to test. This starts with a local search from line 4 to ensure all expression variables are loop through. Before starting the final search, a process randomly chose an initial solution current solution s (line 5). The search loop continues until the temperature **τ** cool down to 0 or less (line 7). Inside the loop, in each iteration, the temperature **τ** is dropped while multiplying it with `cooling_rate` ∝ (line 22). Inside the loop, a neighbour solution η is generated based on current solution **s** (line 8). The fitness function determines if the current solution s will continue in the next iteration or current solution will be replaced by neighbour solution η (line 10). In case of false result from fitness function, a probability function *p* determines the fate of current solution (line 16-20).

```
 1: Begin
 2:   Array result
 3:   Array mcdc_pairs
 4:   Select a cooling_rate ∝
 5:   While result< expression size
 6:          Generate current solution s
 7:          Set initial temperature τ
 8:          While τ> 0
 9:                 Generate move neighbor ∈η(s)
10:                 Where η is neighbor solution.
11:                 If ( f(s) - f(η)) < 0
12:                        Set result = index(expression)
13:                        Set mcdc_pairs = mcdc_pairs(s)
14:                        Set mcdc_pairs = mcdc_pairs(η)
15:
16:                 Else
17:                        Set Δ = s energy - η(energy)
18:                        Set probability p = $e^{-\Delta/\tau}$
19:                        If p> between 0 and 1
20:                               Set s = η
21:                        End If
22:                 End If
23:                 Set τ = τ * ∝
24:          End While
25:    End While
26: End
```

Figure 3.5 Pseudo code of Simulated Annealing Algorithm

3.3.5 Late Acceptance Hill Climbing

The algorithm starts with outer loop at line 5, before which initialization of the variables is done. The while loop continues until `result < expression variable size`. On every iteration, the `history_memory` makes a reset (line 6) and starts to fill the memory with current solutions (line 10) until the `history_memory = memory_size` is reached. Now `memory_size` is a pre-defined variable here. An `index` variable, starting from 0 (line 13) is used to count the iteration of second loop (line 31) and used to get the memory from `history_memory` (line 18). In the second loop at line 8, the algorithm generates a neighbour solution based on current solution. The fitness function (line 19) checks if the candidate will take the role of current solution at the next iteration. If fitness function return `T` and result not contain expression variable position (line 20), add the solution in `mcdc_pair` array (line 22) and set current solution = neighbour solution (line 24). When then `index` become ≥ then `iterator` the second loop will stop (line 27-29).

```
Begin
Array result, mcdc_pairs, history_memory
Integer iterator
Integer memory_size
Integer loop
While result size < expression variable size
   Reset history_memory
   Reset loop = 0
   While loop<memory_size
          Generate current solution s
          Push history_memory(s)
          Set loop++
   End While
Integer index = 0
Boolean is_complete = false
While is_complete = false
          Generate move neighbor $\in \eta(s)$
          Where $\eta$ is neighbor solution.
          Integer value   = index mod memory_size
          Set memory= history_memory
          If $( f(s) - f(\eta)) < 0$
                 If $step \notin$ index(expression)
                        Set step = index(expression)
                        Set mcdc_pairs = mcdc_pairs(s)
                        Set mcdc_pairs = mcdc_pairs($\eta$)
                        Set $s = \eta$
                 End if
          End if
          If index>= then iterator
                 is_complete = true
                 break;
          End if
          Set index++
   End While
End While
End
```

Figure 3.6 Late Acceptance Hill Climbing Pseudo Code

3.4 Algorithm Calibration

We present in the following sections the algorithmic settings for each of the HC, SA, GD and the LAHC algorithms.

We calibrated the algorithms with the help of a simple experiment. We set a counter in each algorithm and administrate the search loop to run 500 times strictly. We run SA first and record the execution time and output. Then we move to GD and did the same. Until the GD records match with SA, we keep tuning the algorithm variables. When GD is settled, we did same for LAHC. After the calibration, algorithm configuration values are given in Table 3.1

Table 3.1 Algorithm Configuration Values

Algorithm	Parameter	Value
Hill Climbing (HC)	Iterator	100
Simulated Annealing (SA)	Cooling Rate	0.01
	Initial Temperature	10000.00
Great Deluge Algorithm (GDA)	Iterator	10
	Final level	0.05
Late Acceptance Hill Climbing (LAHC)	Iterator	10
	History Memory Size	100

After finalizing the algorithm parameters in Table 3.1, we used selected parameters from Table 3.2 and execute each algorithm for 30 times. The result of outcome is presented in next chapter.

3.4.1 Comparative Studies

For each iteration, we have collected number of test cases, execution time and cumulative data of minimum and maximum test cases; minimum, maximum and average execution time for all algorithms. Our experiment is performed in Intel® Core i5 (1.4GHz, 3MB L3, 256KB L2 cache) with 4GB of RAM on Mac OS X Yosemite Operating System.

3.5 Subject Expressions

For our experiment, we applied all four algorithm implementations in all nine Boolean expressions of Table 3.2 and recorded the output. These Boolean expressions are expressions converted from the functions in Function Name column of same table. Algorithms are set to run 30 times for each Boolean expressions. Before executing final experiment, we calibrate our algorithms with their different variables, so that each algorithm takes close enough average time.

Table 3.2 List of Boolean expressions for experiment

Expression No.	Symbol	Expression	No. of conditions	Group
1	***E1***	$(a+b+c)$	3	A Simple
2	***E2***	$((a+b)+c)$	3	
3	***E3***	$(a+(b+c))$	3	
4	***E4***	$(a(bc)+\bar{d}))$	4	B Medium Complex
5	***E5***	$((a\bar{b})+(bd+c))$	4	
6	***E6***	$((a+\bar{c})(b+cd))$	4	
7	***E7***	$(abc\bar{d}(ef)(g+h))$	8	C Complex
8	***E8***	$(abcd)+(ef+\bar{g}h))$	8	
9	***E9***	$((a\bar{b})+(cd)+(\bar{d}f)+(\bar{g}h))$	8	

In this experiment, Expressions of Group A in Table 3.2 are considered as simple complexity expression as they have lowest number of conditions and decision in our experiment. Group B is considered as medium complexity expressions as they have more decisions then group A and less then Group B. Group *C* is considered as complex expressions as they have heights number of conditions and decisions in our list of expressions to test.

3.6 Approach Steps

The automation of the testing approach is divided mainly into five steps illustrated in Figure 3.7.

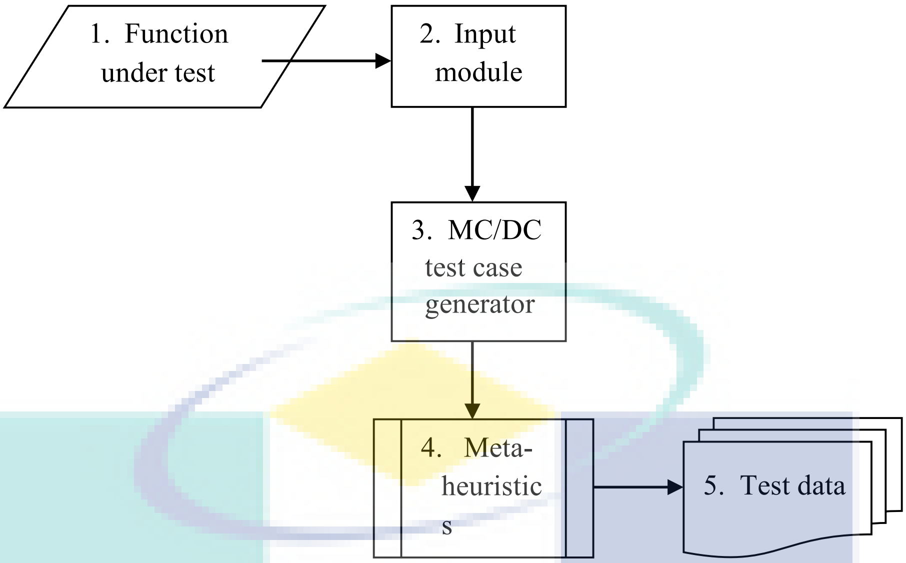


Figure 3.7 Approach Steps in the Automation of Testing

The function under test is first fed into the Input module. The Input module validate the input, identify size of expression and list of logical operators and output to the MC/DC test suite generator. The MC/DC test case generator control the meta-heuristics algorithm configuration and fed the expression and expression size. The fourth step is the meta-heuristic algorithm that process the expression and output the result as test data.

3.7 Summary

Different criteria for software testing that assures the program to be bug-free has been discussed. Starting with the essential definitions, the Modified Condition/ Coverage Criteria is shown an enhancement of the simple combination of the Condition criterion and the Coverage criterion. Though the multiple condition coverage criteria assures an exhaustive testing, it is not always feasible due to its exponential time complexity of the 2^n test cases it require for a decision with n inputs which calls for the MC/DC to provide the minimal test cases which require each condition to independently affect the outcome of a decision. This gives a reasonable guarantee to be error-free, and makes fault finding convenient, whilst maintaining a linear time complexity with $n + 1$ test cases.

The method for automated test data generation is discussed in the next section of this chapter. This comprises of the employment of four metaheuristics, namely hill climbing, great deluge algorithm, simulated annealing and the late acceptance hill climbing in an effort to find the optimal test cases required to satisfy the MC/DC criteria for a sample condition is generated with the help of each of these metaheuristics. The pseudo code is presented for each.

It remains to be seen which of these metaheuristics perform better with respect to generation of automated test cases for the MC/DC criteria. This forms the focus of the next chapter.

CHAPTER 4

RESULTS AND DISCUSSION

4.1 Introduction

In this chapter, we report results from a preliminary experimental study carried out to evaluate the performance of our approach for MC/DC automatic test input data generation using Hill Climbing (HC), Simulated Annealing (SA), Great Deluge (GD) and Late Acceptance Hill Climbing (LAHC). Each algorithm is compared with other algorithms. In the next subsections, we briefly describe selected Boolean expressions to experiment and the main experimental steps, details about the algorithmic settings, and finally, we present results and their interpretation.

4.2 Result Presentation

In Table 4.1 and Table 4.2, the Expression Number column refers to the Expression Number column of Table 3.2, meaning data in row 1 of Table 4.1 and Table 4.2 is data of Boolean expression `(a+b+c).` Condition column indicates number of conditions (literals) in the Boolean expression. From third column, every two columns in Table 4.1 and every three columns in Table 4.2 are grouped for one algorithm. *Min* and *Max* columns in Table 4.1 contain the maximum and minimum number of test cases for current algorithm, while Min, Max and Avg in Table 4.2 represents average, maximum and minimum run times for the algorithms. From now on, Expression Id. 1 to 9 will be symbolized as $E1$, $E2$, $E3$, …, $E9$., while the number of conditions are symbolized by n_c. The expressions $E1$, $E2$ and $E3$ of Table 4.2 are Boolean expression with $n_c = 3$. These can be considered as simple expressions. We call this as *Group A*. Boolean expressions in $E4$, $E5$, $E6$ are with $n_c = 4$. We will call it *Group B*. *Group C* contains the three Boolean expressions $E7$, $E8$, $E9$ with $n_c = 8$. *Group C* is considered as complex expressions.

Table 4.1 Result of minimum, maximum number of test cases generated

Expression Id. (*E)**	**No. of conditions (n_c)**	**Simulated Annealing**		**Hill Climbing**		**Great Deluge Algorithm**		**Late Acceptance Hill Climbing**	
		Min	Max	Min	Max	Min	Max	Min	Max
1	3	4	5	4	5	4	5	4	5
2	3	4	4	4	4	4	4	4	4
3	3	4	4	4	4	4	4	4	4
4	4	5	7	5	7	5	7	5	7
5	4	5	7	5	7	5	7	5	8
6	4	5	8	5	8	5	8	6	7
7	8	9	10	9	10	9	10	9	10
8	8	12	16	11	15	11	16	14	16
9	8	12	16	11	15	11	16	13	16

Table 4.2 Result of minimum, maximum and average run time from experiments

*E**	n_c	Simulated Annealing			Hill Climbing			Great Deluge Algorithm			Late Acceptance Hill Climbing		
		Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
1	3	0.08	0.22	0.15	**0.03**	0.61	**0.14**	0.07	0.52	0.21	0.12	0.59	0.30
2	3	**0.02**	1.42	**0.28**	0.08	1.16	0.36	0.06	1.77	0.41	0.12	0.72	0.37
3	3	**0.02**	1.24	0.22	0.07	0.42	**0.18**	**0.02**	0.65	0.19	0.05	1.00	0.25
4	4	0.05	0.43	**0.19**	**0.03**	1.49	0.36	0.10	1.18	0.51	0.11	1.23	0.64
5	4	0.08	0.75	**0.24**	0.06	1.08	0.25	**0.07**	1.25	0.27	0.11	1.02	0.51
6	4	**0.06**	1.24	**0.40**	0.07	1.02	**0.40**	0.08	1.82	0.42	0.11	2.31	0.81
7	8	**0.72**	29.34	**7.72**	0.92	23.99	10.19	2.49	28.65	10.65	72.14	451.75	205.18
8	8	**0.26**	1.68	**0.93**	0.31	2.36	1.21	0.28	2.72	1.06	6.00	71.27	26.02
9	8	0.34	3.51	1.47	**0.12**	3.80	1.28	0.28	2.75	**0.99**	6.42	71.59	24.92

4.3 Discussion

In the following subsections, we present the comparison of the efficiencies of the four algorithms in producing the exact number of test cases and the time required to produce them from the minimum and maximum test cases produced in our experiments, and the minimum, maximum and average run times recorded in the respective algorithms.

4.3.1 Group A: Expressions with 3 conditions

In *Group A*, SA, HC, GD and LAHC perform exactly same in number of test case generation. All the algorithms have 4 test cases in minimum and maximum category. But in terms of performance, they actually fluctuate.

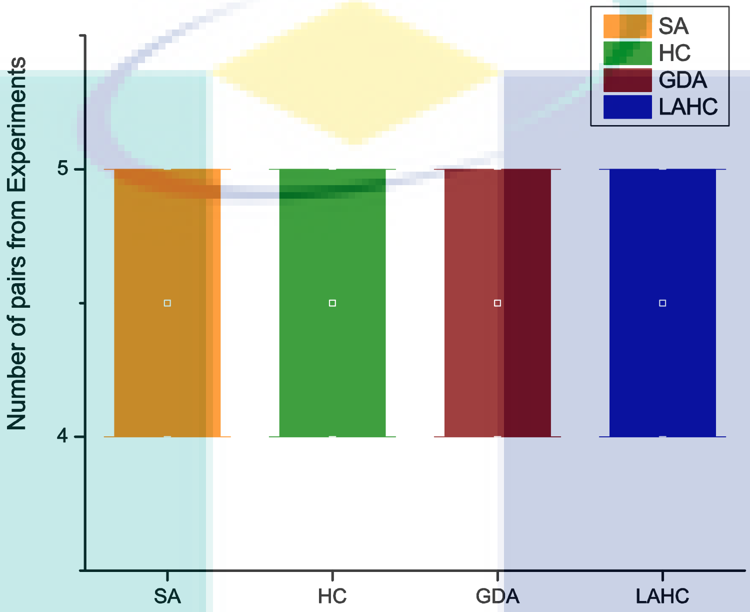


Figure 4.1 Number of test cases generated for the Expression 1 (Group A)

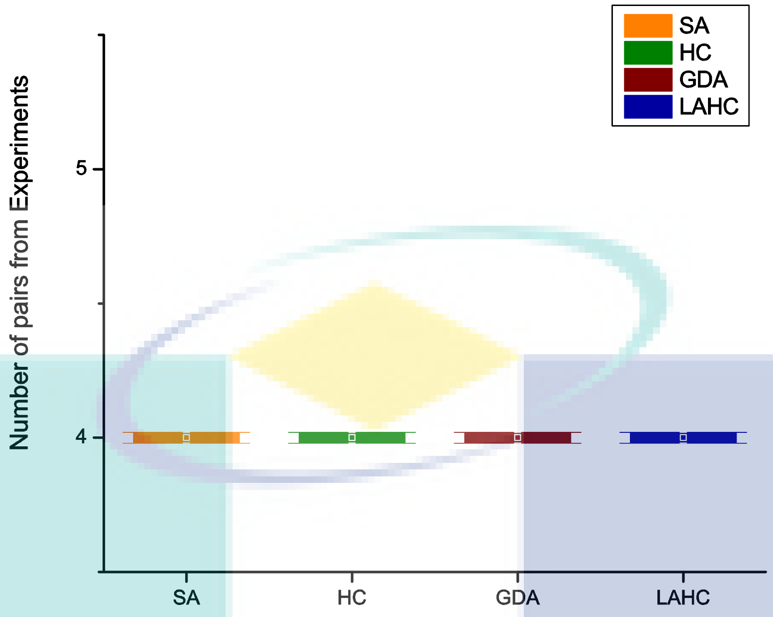


Figure 4.2 Number of test cases generated for the Expression 2 (Group A)

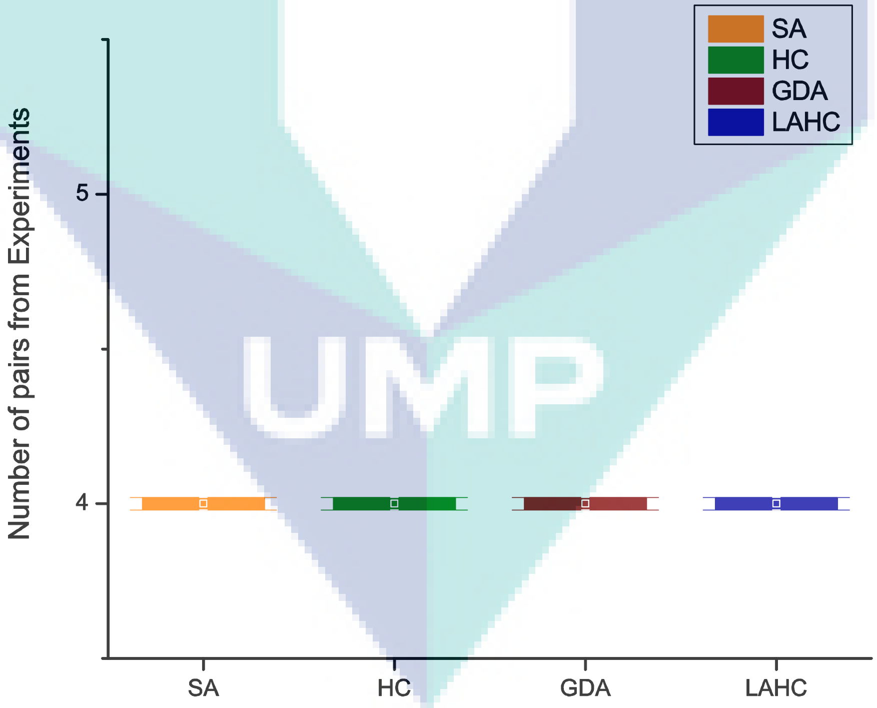


Figure 4.3 Number of test cases generated for the Expression 3, Group A.

In Figure 4.1, Figure 4.2, Figure 4.3, figures represent the number of test cases generated for Expressions of Group A. Each bar represents the number of test cases generated for different algorithms, with the bottom denoting the minimum and the top denoting the maximum number of test cases generated.

In Group A, For E2 and E3, all algorithms generated 4 test cases and for E1, 5 test cases. We can easily say we observed a pure reflection of No Free Lunch Theorem (Wolpert & Macready, 1997) here, as even the algorithms have different number of parameters to configure and tune, which increase or decrease complexity of algorithms, but here the complexity does not effects the output of the algorithms.

In the next section, we will look for the performance of the algorithms and will try to see if the complexity effects the time performance.

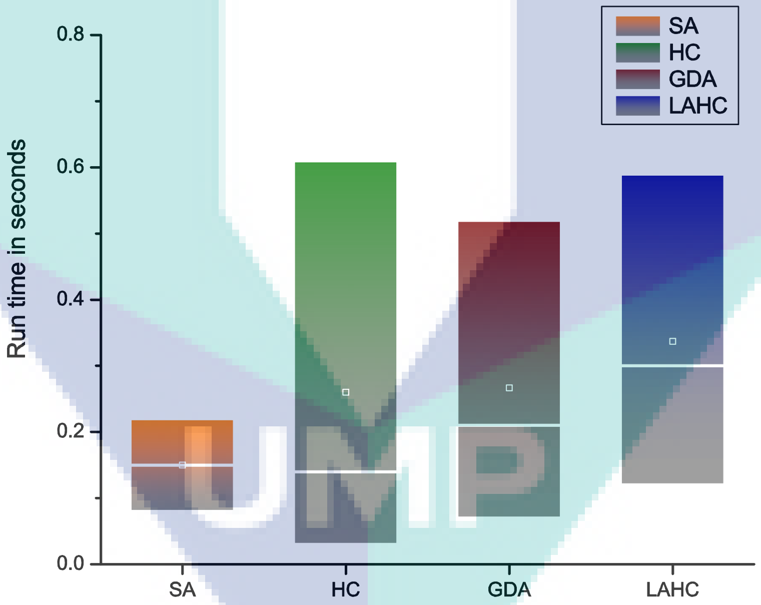


Figure 4.4	Time Performance of different algorithms for Expression 1(Group A)

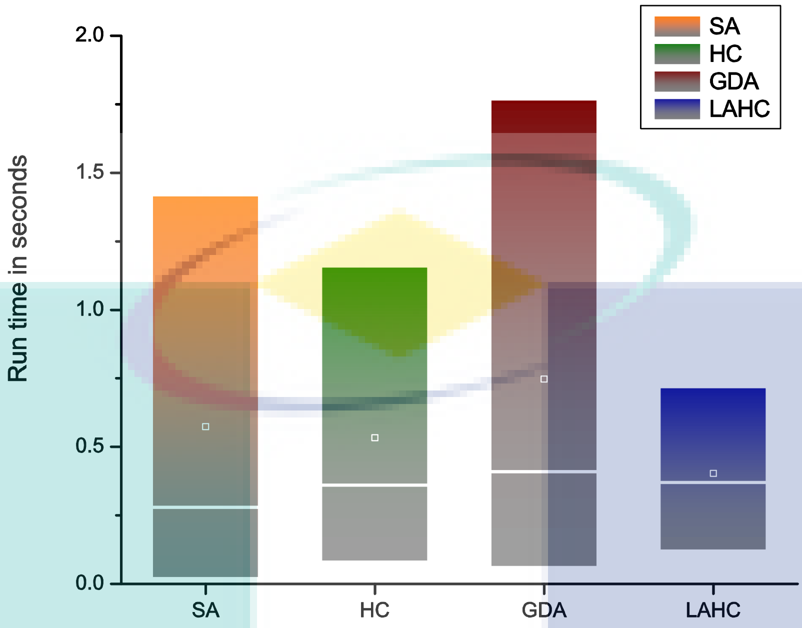


Figure 4.5 Time Performance of different algorithms for Expression 2 (Group A)

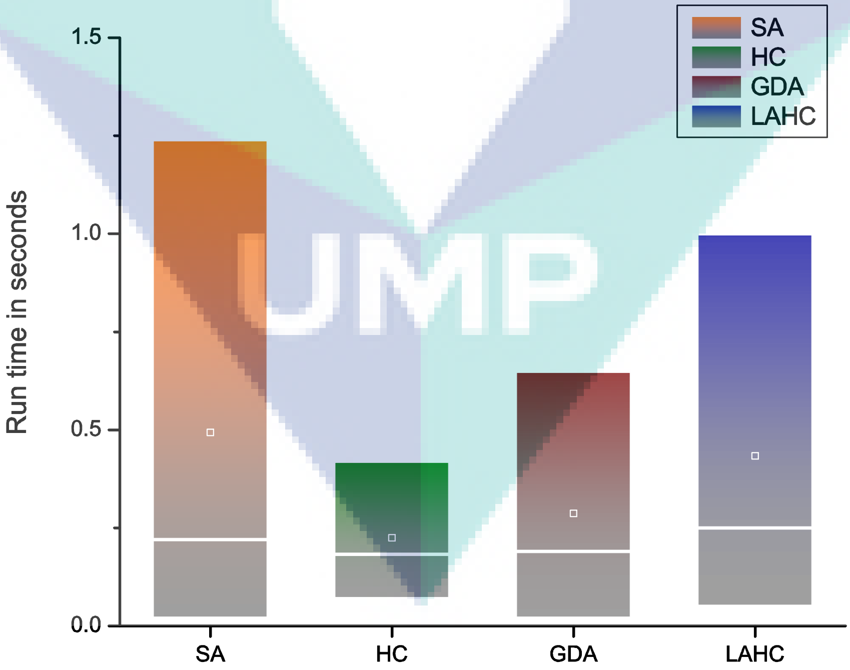


Figure 4.6 Time Performance different algorithms for Expression 3 (Group A)

In Figure 4.4, Figure 4.5, Figure 4.6, each bar represents the run time in seconds for different algorithms, with the bottom denoting the minimum, the top denoting the maximum, and the white horizontal line in between denoting the average time taken in different runs of the same expression.

From Figure 4.5 and Figure 4.6, it is clearly visible that the average test case generation time of HC is the lowest for *E1* and *E3*. SA is wining here for *E2*. Both GD and LAHC require a higher time to execute the same expression while producing the same result. In case of maximum time required, GD is performing very poor as for a case it took 1.77 seconds to execute *E2*. SA is 2^{nd} in low performing in our experiment as *E2* and *E3* took more than 1.20 seconds for each. LAHC looks pretty stable and no big jump among same type of expressions for maximum time required.

4.3.2 Group B: Expressions with 4 conditions

Maximum and minimum number of test cases generated for *Group B* is presented in Figure 4.4. For *minimum* test cases, SA, GD, HC and LAHC performed exactly same (5 test cases) except only for *E6* (LAHC produce 6 test cases). For *maximum* number of test cases, SA, GD, HC produce same result for all expressions (*E4* = 7, *E5* = 7, *E6* = 8) except LAHC produce 8 test cases for *E5* and 7 test cases for *E4* and E6.

We can come to a conclusion that

- In *Group B;* while SA, GD, HC has an exact output, LAHC trend to produce more test cases.

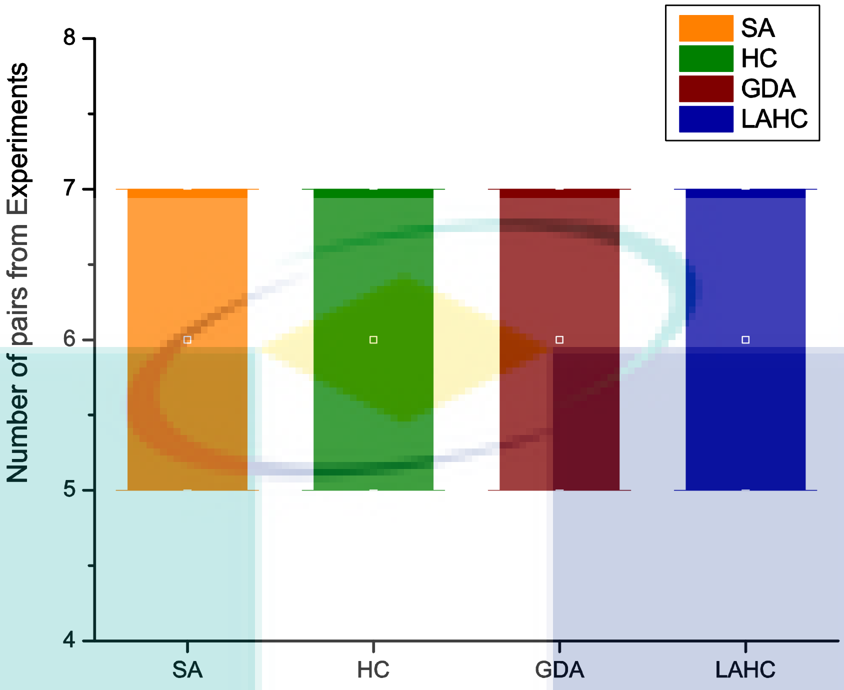


Figure 4.7 Number of test cases generated for the Expression 4 (Group B)

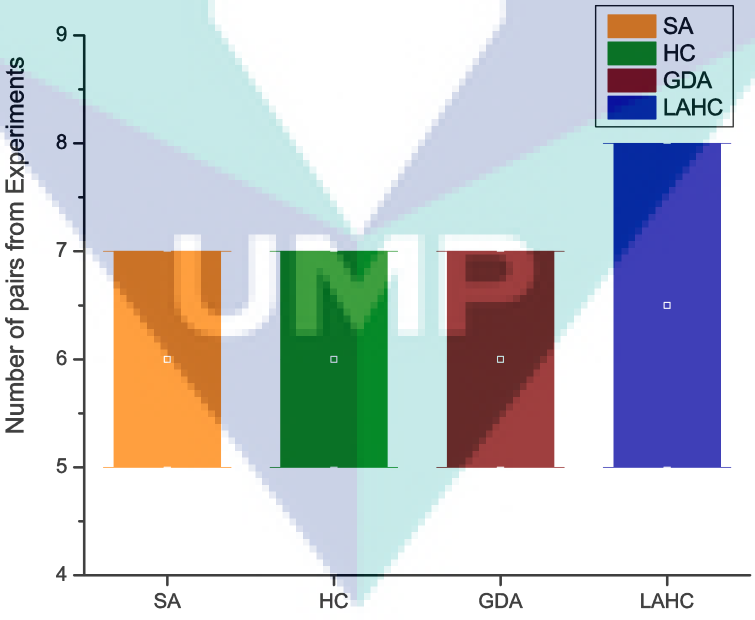


Figure 4.8 Number of test cases generated for the Expression 5 (Group B)

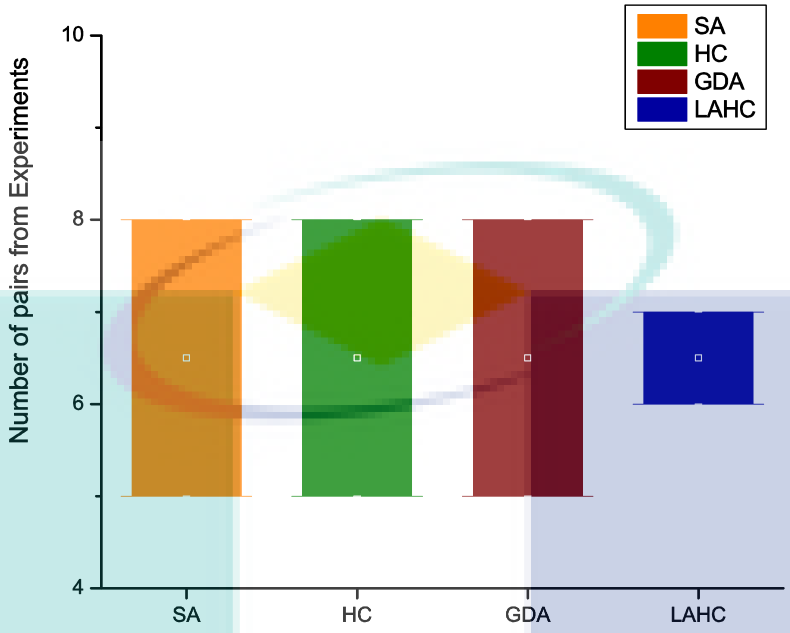


Figure 4.9 Number of test cases generated for the Expression 6 (Group B)

Results of execution time for this group is visualized in Figure 4.10, Figure 4.11, Figure 4.12. For Average execution time, SA wins for all expressions with lowest average followed by HC, GD and LAHC after it. Though average of *E5* and *E6* are pretty consistence in all algorithms; *E4*, which has $a + \bar{a}$ logic, in average takes more time to produce test cases, except GD has best average for this expression. When SA, HC, GD has average between *0.24* to *0.27* seconds, LAHC has *0.51* seconds to produce result. LAHC has very consistent *minimum* execution time while *maximum* execution time is very inconsistent compared to other algorithms. We can conclude here LAHC has worst *average* while SA has the best average execution time. Although minimum execution time is relatively near between all algorithms, maximum execution time differs in all algorithms. SA, GD, LAHC took more time for *E6* while HC managed it very well, though HC fail to handle *E5* in a short period of.

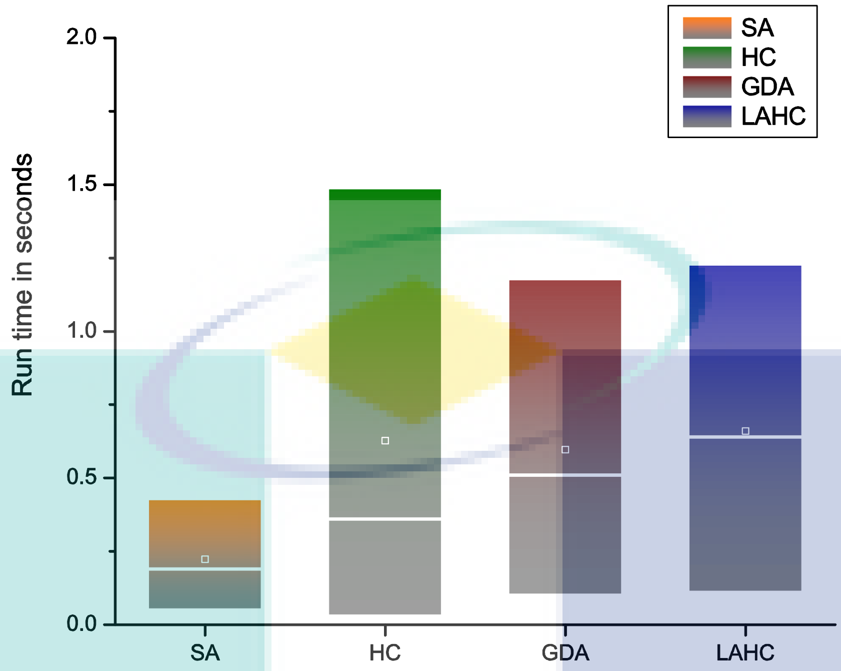


Figure 4.10 Result of Execution time of Expression 4 (Group B)

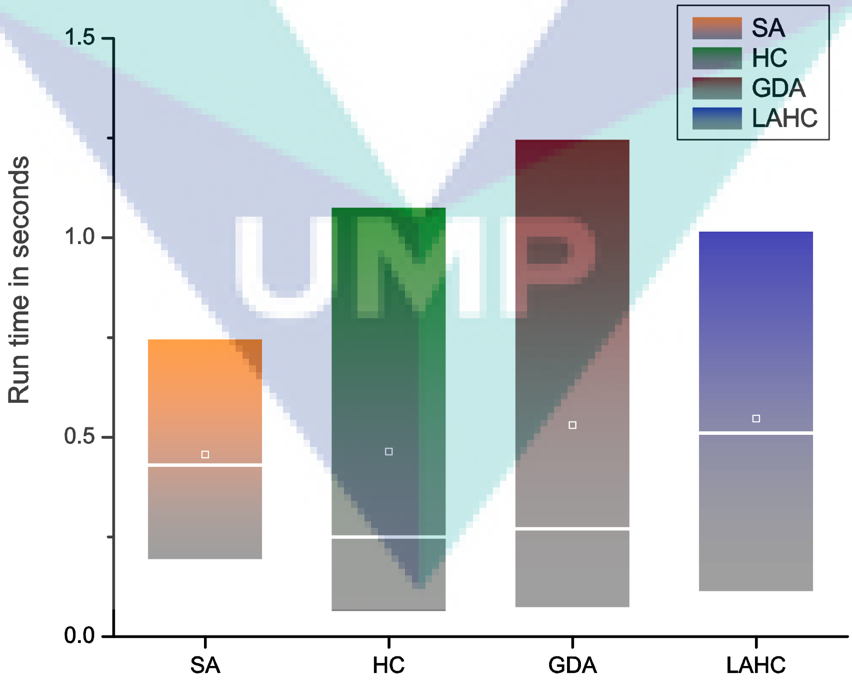


Figure 4.11 Result of execution time for Expression 5 (Group B)

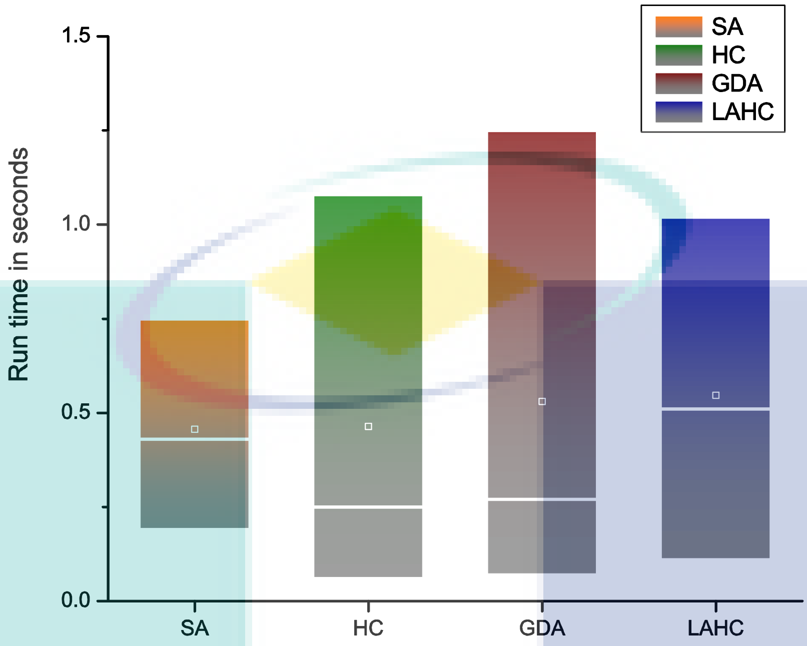


Figure 4.12 Time Performance of Algorithms forExpression 6 in Group B

4.3.3 Group C: Expressions with 8 conditions

In *Group C*, though all expressions have 8 conditions, *E7* is one large expression and logics are grouped inside the parentheses. *E8* is constructed with 2 groups of logics, each containing 4 conditions and *E9* has 4 groups of logics with 2 conditions in each group. In the Figure 5.6, maximum and minimum number of test cases are presented in the form of a bar chart. For *E7*, all the algorithms generated 9 *minimum* test cases and 10 *maximum* test cases while LAHC have generated 10 *minimum* cases as well. For *E8* and *E9*), algorithms trend to generate more test cases for these expressions. For these two expressions, HC & GD generated *lowest minimum* test cases (11 test cases). SA is in 2nd position with 12 test cases and LAHC produce poorest result with 14 and 13 test cases. For maximum test cases of *E8* and *E9*, HC produces 15 test cases only where SA, GD and LAHC product 16 test cases. After observing this relational behavior of algorithms and grouped Boolean expressions, If one or more decision exists in a Boolean expression/predicate, that dramatically affects the output of algorithms ignoring the number of total conditions.

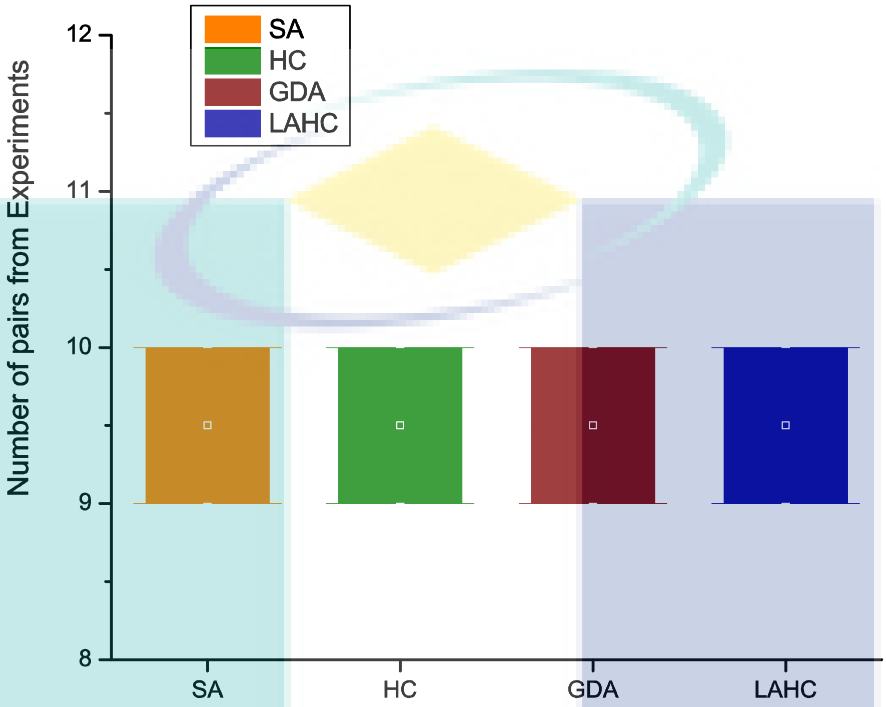


Figure 4.13 Number of test cases generated for the expression 7 in Group C.

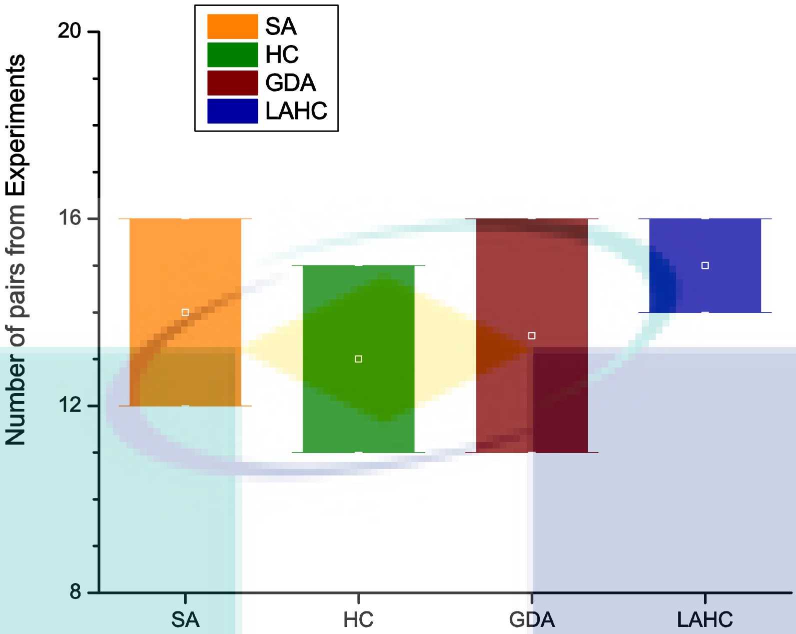


Figure 4.14 Number of test cases generated for the expression 8 in Group C.

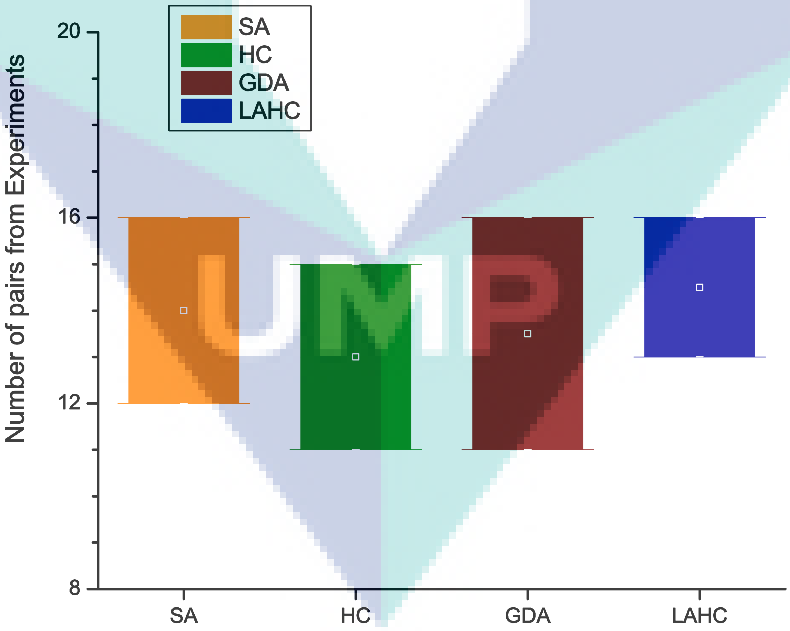


Figure 4.15 Number of test cases generated for the expression 9 in Group C.

Though the algorithms generated more test cases for *E8* and *E9*, with *E7* having less test cases, all algorithms performed almost similarly in execution time except LAHC, which took excessively long time to produce test cases for *E7*. For *E7*, while we recorded *average* time for SA *7.72* seconds, HC *10.19 seconds* and GD *10.65 seconds*, LAHC took average time of *205.18 seconds* for same 8 condition expressions. In terms of maximum time required, LAHC took *451.75 seconds* (7.54 minutes) while 2nd maximum time from other algorithms is *28.65 seconds* by GD. Lowest *maximum* time is taken by HC, *23.99 seconds*. The execution times are presented in Figure 4.16,Figure 4.17 and Figure 4.18 To accommodate the huge variation of times recorded by LAHC (nearly 10 times) in comparison with SA, HC and GD, the time is plotted on a logarithmic scale to enable comparison on the same axis.

Algorithms showed a completely different behaviour while generating output for other 2 expressions. SA, HC, GD took average of less than *1.5 seconds* for *E8* and *E9*. Even LAHC took *26.02* seconds (*E8*) and *24.92 seconds* (*E9*) for generating test cases for them.

From this observation, we can say,

- though the algorithms generated *lower number of test cases* for one big group of conditions, there is a price in terms of *execution time*.

- algorithms took significantly *less time* for same number of conditions with multiple decisions in an expression. Here, algorithms compensate time with number of test cases.

- Comparing number of test cases and execution time, LAHC is poor for such use, especially for expressions with more conditions.

- GD is also low performing compared to SA and HC, yet not totally outcast like LAHC.

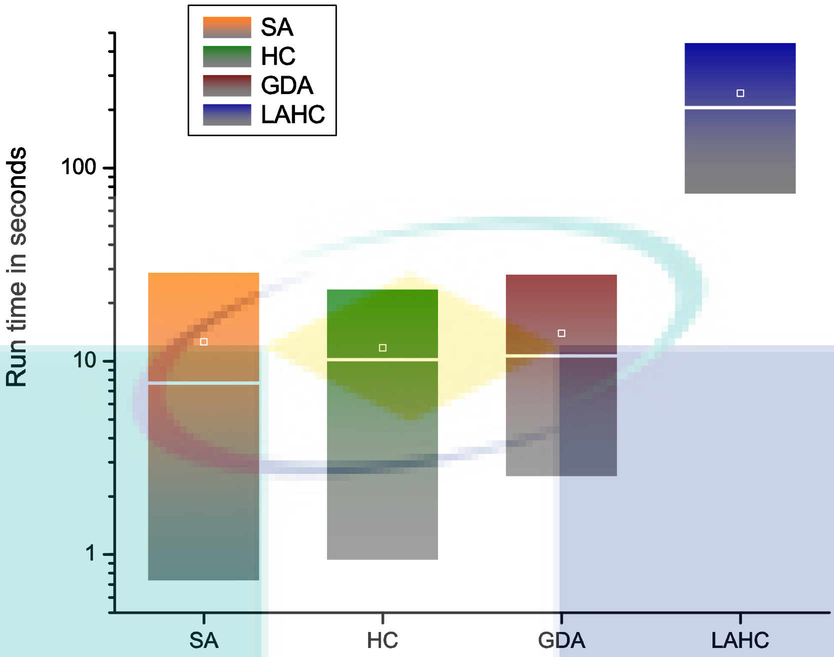


Figure 4.16 Time Performance of Algorithms for the expression 7 in Group C

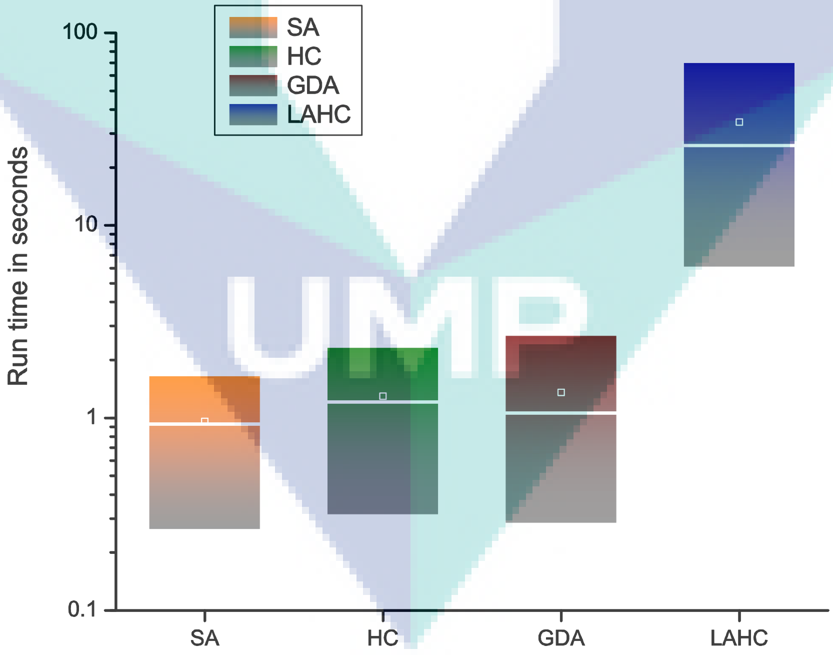


Figure 4.17 Time Performance of Algorithms for the expression 7 in Group C

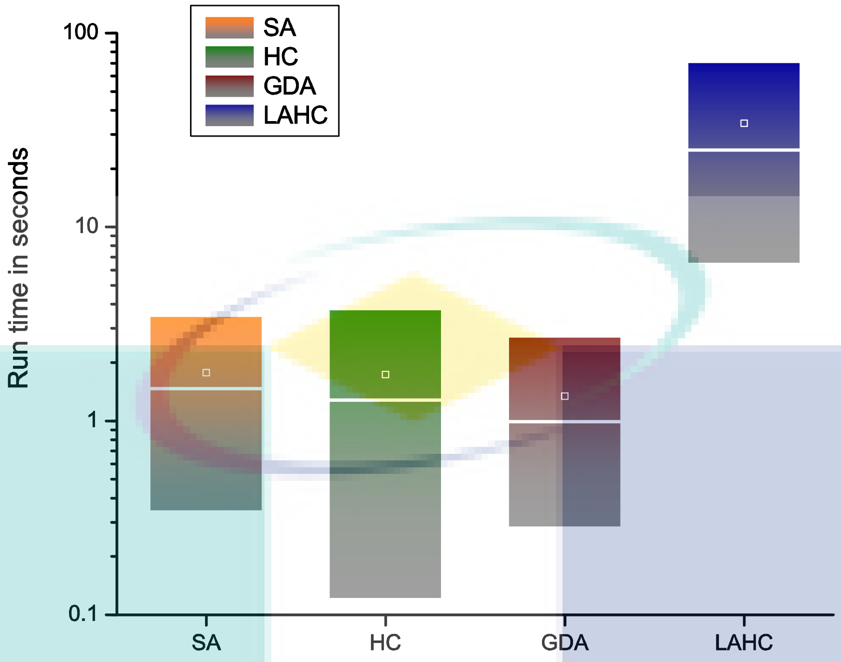


Figure 4.18 Time Performance of Algorithms for the expression 9 (Group C)

The run-time is plotted on a logarithmic scale to accommodate the huge variation of times recorded by the algorithms

4.4 Observation

From the results, we have observed Time performance and output of algorithms also depends on various other features. Here is an explanation about the observed features are given:

4.4.1 Presence of No Free Lunch Theorem:

In all our figures in this chapter, we can see different algorithms are producing different results and performing differently without any specific order. This does not mean an algorithm is better than another algorithm. As per No Free Lunch Theorem (Wolpert & Macready, 1997), one algorithm producing better result for a specific problem might not show the same performance in another algorithm. In Chapter 2, we discussed LAHC is performing excellent in few problems (Verstichel & Berghe, 2009) but here in most expressions, especially in Group C, the algorithm does not produce satisfactory result compared to other algorithms. Average time requires to generate the test cases for SA and HC are lower from GD and LAHC.

4.4.2 Exploration and Exploitation

Hill Climbing (Renders & Bersini, 1994) Algorithm is better at Exploitation as the algorithm always look for the next better solution based on the current solution in local optimum. Late Acceptance Hill Climbing derived the Exploitation property from Hill Climbing algorithm and use stored memory exploit few steps older solutions to achieve the best result. Simulated Annealing try to find a solution by exploiting previous solution. If the current solution is not better than the previous solution, SA use exploration technique to find next solution. Great Deluge Algorithm can also be configured to have a quicker exploration rate (Mcmullan, 2007) by tuning the rain fall parameter.

4.4.3 Easy to use

From this observation, we can summarize,

- Though the algorithms generated *lower number of test cases* for one big group of conditions, the *cost is execution time* for that.

- Algorithms took significantly *less time* for same number of conditions with multiple decisions in an expression. Here algorithms compensate time with number of test cases.

- Comparing number of test cases and execution time, LAHC is poor for such use, specially for expressions with more conditions.

- GD is also low performing compared to SA and HC, yet not totally outcast like LAHC.

4.5 Summary

In a nutshell, the results for the four different algorithms reveal quite a few important observations regarding accuracy and efficiency in generating test cases for software testing. The first observation is, the four algorithms do not differ much from the accuracy point of view. This is shown by the fact that the number of test cases generated are roughly equal for all the four algorithms. However, so far as the run times of the test case generation code is concerned, thus indicating efficiency of the algorithms, the most consistent performer seems to be the simplest algorithm, viz. Hill Climbing. And, interestingly, the algorithm that required the most time is Late Acceptance Hill Climbing. Great Deluge performed better and ranks second in requiring less time to get the required number of test cases, followed by Simulated Annealing which ranks third.

CHAPTER 5

CONCLUSION

5.1 Introduction

The previous chapter has subjected our implemented algorithms with several experiments in order to discover its comparative analysis. Within the context of this research, this chapter discusses the impact of the results achieved and implication for future work.

5.2 Concluding Remarks

This research was aimed to study and investigate neighbourhood based mate-heuristic algorithms for test suite generation satisfying MC/DC criterion. Than develop two new neighbourhood based metaheuristics algorithm implementation to generate Test suite that satisfy MC/DC criterion and perform a comparative analysis with two re-implemented meta-heuristic algorithms. The research objectives for this study were as follows:

1. Study and investigate the use of neighbourhood metaheuristic algorithms for test case generation satisfying the MC/DC Criterion.

2. Develop four new implementations of neighbourhood based strategies based on Simulated Annealing, Hill Climbing, Great Deluge, and Late Acceptance Hill Climbing that satisfy MC/DC.

3. Compare and analyse the performance of the developed strategies in terms of test case size and execution time.

Addressing the first objective, we elaborate on the need of Modified Condition/ Decision Criteria (MC/DC) in software structural testing and demonstrate the need of neighbourhood based metaheuristics search algorithms in finding a suitable test suite within a reasonable time and cost. Different neighbourhood search algorithms are illustrated, and their relative advantages and disadvantages are discussed, keeping in the background their application to different domains by previous researchers.

Concerning the second objective, we implemented two new neighbourhood based metaheuristic strategies, namely Late Acceptance Hill Climbing (LAHC) and the Great Deluge (GD) Algorithm in finding an optimal test suite for 9 exemplary expressions of different complexity. The performance of these two algorithms in terms of number of test cases generated, and run time (minimum, maximum and average) is compared with two other re-implemented neighbourhood based meta-heuristics algorithms Simulated Annealing (SA) and Hill Climbing (HC).

As for the final objective, we successfully identified the strength and weaknesses of each of these algorithms and their behavioural activity toward different sizes of Boolean expressions. In particular, it was observed that while GD performed well for most expressions, LAHC could outperform the other search strategies only for the simple expressions.

5.3 Contributions

Main contributions of this research are:

1. Two (LAHC and GDA) new implementation of neighbourhood based meta-heuristics algorithm to generate test suite satisfying MC/DC criterion.

2. Comparative performance analysis of Neighbourhood based meta-heuristics algorithms between SA, HC, LAHC and GDA.

5.4 Scope of Future Work

As future extension of this study, an in-depth analysis about the performance of LAHC and GD by tuning their configuration parameters (where they need not stay sync with other algorithms) can be implemented. Implementation of other search algorithms like the Tabu Search can also be conceived, with due comparison with the metaheuristic search strategies already carried out in this thesis. Population based algorithms and Global optimization techniques can be applied to generate MC/DC criterion compliant test cases. Bat algorithm, Cuckoo algorithm, Firefly algorithm are new optimization approaches and can be applied to test case generation. I also suggest to Memetic algorithm to generate test case satisfying MC/DC Criterion as this algorithm use both Local and Global optimization techniques. Such a study with five or six search strategies and different parametric settings can really help the software tester to get an optimal test suite with high reliability and lowered cost.

REFERENCES

Aarts, E., & Korst, J. (1991). Simulated annealing and boltzmann machines: a stochastic approach to combinatorial optimization and neural computing: Wiley.

Alba, E., Chicano, F., Ferreira, M., & Gomez-Pulido, J. (2008). Finding deadlocks in large concurrent java programs using genetic algorithms. Paper presented at the Proceedings of the 10th Annual conference on Genetic and Evolutionary computation.

Ammann, P., Offutt, J., & Huang, H. (2003). Coverage criteria for logical expressions. Paper presented at the 14th International Symposium on Software Reliability Engineering, 2003.

Andersen, K., Vidal, R. V. V., & Iversen, V. B. (1993). Design of a teleprocessing communication network using simulated annealing Applied Simulated Annealing (pp. 201-215): Springer.

Appleby, J., Blake, D., & Newman, E. (1961). Techniques for producing school timetables on a computer and their application to other scheduling problems. The Computer Journal, 3(4), 237-245.

Authority, F. A. (1992). Software considerations in airborne systems and equipment certification Document No. RTCA/DO-178B.

Awedikian, Z. (2009). Automated test data generation for mc/dc test criterion using metaheuristic algorithms. (Masters), Université De Montréal, Canada.

Awedikian, Z., Ayari, K., & Antoniol, G. (2009). Mc/dc automatic test input data generation. Paper presented at the Proceedings of the 11th Annual Conference On Genetic And Evolutionary Computation, 2009.

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

Burke, E., Elliman, D., Ford, P., & Weare, R. (1995). Examination timetabling in British universities: A survey. Paper presented at the International Conference on the Practice and Theory of Automated Timetabling.

Burke, E. K., & Bykov, Y. (2008). A late acceptance strategy in hill-climbing for exam timetabling problems. Paper presented at the Practice and Theory of Automated Timetabling (PATAT) Conference, 2008, Montreal, Canada.

Burke, E. K., & Bykov, Y. (2012). The Late Acceptance Hill-Climbing Heuristic (CSM-192). Retrieved from UK:

Bykov, Y. (2011). The Late Acceptance Hill-Climbing Algorithm For The Magic Square Problem. Article. Nottingham, United Kingdom.

Chang, J.-R., & Huang, C.-Y. (2007). A study of enhanced mc/dc coverage criterion for software testing. Paper presented at the Proceedings of 31st Annual International Computer Software and Applications Conference, 2007.

Chilenski, J. J., & Miller, S. P. (1994). Applicability of modified condition/decision coverage to software testing. Software Engineering Journal, 9(5), 193-200.

Cohen, M. B. (2004). Designing Test Suites for Software Interactions Testing. (Doctor of Philosophy), The University of Auckland, Auckland, New Zealand.

Cohen, M. B., Gibbons, P. B., Mugridge, W. B., & Colbourn, C. J. (2003). Constructing test suites for interaction testing. Paper presented at the Software Engineering, 2003. Proceedings. 25th International Conference on.

Cohn, H., & Fielding, M. (1999). Simulated annealing: searching for an optimal temperature schedule. SIAM Journal on Optimization, 9(3), 779-802.

Díaz, E., Tuya, J., Blanco, R., & Dolado, J. J. (2008). A tabu search algorithm for structural software testing. Computers & Operations Research, 35(10), 3052-3072.

Dueck, G. (1993). New optimization heuristics: the great deluge algorithm and the record-to-record travel. Journal of Computational Physics, 104(1), 86-92.

El-Sayed, G., Salama, C., & Wahba, A. (2015). Optimization of Generated Test Data for MC/DC Intelligent Software Methodologies, Tools and Techniques (Vol. 532, pp. 161-172): Springer.

Ghani, K., & Clark, J. A. (2009). Automatic test data generation for multiple condition and MCDC coverage. Paper presented at the Proceedings of 4th International Conference on Software Engineering Advances, 2009.

Glover, F. (1985). Future Paths for Integer Programming and Links to Artificial Intelligence', CAAI Report 85-8. Center for Applied Artificial Intelligence, University of Colorado, October.

Glover, F. (1997). Tabu search and adaptive memory programming—advances, applications and challenges Interfaces in computer science and operations research (pp. 1-75): Springer.

Harman, M. (2007). Search based software engineering for program comprehension. Paper presented at the 15th IEEE International Conference on Program Comprehension, 2007. .

Harman, M., Hu, L., Hierons, R. M., Baresel, A., & Sthamer, H. (2002). Improving Evolutionary Testing By Flag Removal. Paper presented at the The Genetic and Evolutionary Computation Conference (GECCO) 2002, New York City, United States.

Harman, M., & Jones, B. F. (2001). Search-based software engineering. Information and Software Technology, 43(14), 833-839.

Hayhurst, K. J., & Veerhusen, D. S. (2001). A practical approach to modified condition/decision coverage. Paper presented at the Digital Avionics Systems, 2001. DASC. 20th Conference.

Hayhurst, K. J., Veerhusen, D. S., Chilenski, J. J., & Rierson, L. K. (2001). A practical tutorial on modified condition/decision coverage: National Aeronautics and Space Administration, Langley Research Center.

Hetzel, W. C., & Hetzel, B. (1988). The complete guide to software testing: QED Information Sciences Wellesley, MA.

Hower, R. (2010). Software QA and testing frequently-asked-questions. Software QA Test. Retrieved from http://www.softwareqatest.com/qatfaq1.html

Hwang, G.-J., Yin, P.-Y., & Yeh, S.-H. (2006). A tabu search approach to generating test sheets for multiple assessment criteria. IEEE Transactions on Education, 49(1), 88-97.

Jia, Y., Cohen, M. B., Harman, M., & Petke, J. (2015). Learning combinatorial interaction test generation strategies using hyperheuristic search. Paper presented at the Proceedings of the 37th International Conference on Software Engineering-Volume 1.

Johnson, D. S., Aragon, C. R., McGeoch, L. A., & Schevon, C. (1989). Optimization by simulated annealing: an experimental evaluation; part I, graph partitioning. Operations research, 37(6), 865-892.

Johnson, L. A. (1998). DO-178B, Software considerations in airborne systems and equipment certification. Crosstalk, October.

Jones, J. A., & Harrold, M. J. (2003). Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Transactions on Software Engineering, 29(3), 195-209.

Kandl, S., & Kirner, R. (2011). Error detection rate of mc/dc for a case study from the automotive domain Software Technologies for Embedded and Ubiquitous Systems (pp. 131-142): Springer.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simmulated annealing. science, 220(4598), 671-680.

Korel, B. (1990a). Automated software test data generation. IEEE Transactions on Software Engineering, 16(8), 870-879.

Korel, B. (1990b). A dynamic approach of test data generation. Paper presented at the Software Maintenance, 1990, Proceedings., Conference on.

Kumar, C. S., Raghu, D., & Kumar, P. R. (2013). A Domestic Case Studies Probability to Overcome Software Failures. Journal of Telematics and Informatics, 1(1), 20-25.

Lakhotia, K., Harman, M., & McMinn, P. (2008). Handling dynamic data structures in search based testing. Paper presented at the Proceedings of the 10th annual conference on Genetic and evolutionary computation.

Li, Z., Harman, M., & Hierons, R. M. (2007). Search algorithms for regression test case prioritization. IEEE Transactions on Software Engineering, 33(4), 225-237.

McMinn, P. (2004). Search-based software test data generation: a survey. Software testing, Verification and reliability, 14(2), 105-156.

McMinn, P., & Holcombe, M. (2006). Evolutionary testing using an extended chaining approach. Evolutionary Computation, 14(1), 41-64.

Mcmullan, P. (2007). An extended implementation of the great deluge algorithm for course timetabling. Paper presented at the International Conference on Computational Science.

Miller, W., & Spooner, D. L. (1976). Automatic generation of floating-point test data. IEEE Transactions on Software Engineering, 2(3), 223-226.

Nguyen, C. D., Miles, S., Perini, A., Tonella, P., Harman, M., & Luck, M. (2012). Evolutionary testing of autonomous software agents. Autonomous Agents and Multi-Agent Systems, 25(2), 260-283.

Obit, J., Landa-Silva, D., Ouelhadj, D., & Sevaux, M. (2009). Non-linear great deluge with learning mechanism for solving the course timetabling problem. Paper presented at the 8th Metaheuristics International Conference (MIC 2009).

Osman, I. H. (1993). Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. Annals of Operations Research, 41(4), 421-451.

Pandita, R., Xie, T., Tillmann, N., & de Halleux, J. (2010). Guided test generation for coverage criteria. Paper presented at the Software Maintenance (ICSM), 2010 IEEE International Conference on.

Paul, T. K., & Lau, M. F. (2014). A systematic literature review on modified condition and decision coverage. Paper presented at the Proceedings of the 29th Annual ACM Symposium on Applied Computing.

Perry, W. E. (1992). A standard for testing application software: Auerbach Publishers.

Rathore, A., Bohara, A., Prashil, R. G., Prashanth, T., & Srivastava, P. R. (2011). Application of genetic algorithm and tabu search in software testing. Paper presented at the Proceedings of the Fourth Annual ACM Bangalore Conference.

Renders, J.-M., & Bersini, H. (1994). Hybridizing genetic algorithms with hill-climbing methods for global optimization: two possible ways. Paper presented at the Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on.

Sagarna, R., & Yao, X. (2008). Handling constraints for search based software test data generation. Paper presented at the Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on.

Schulmeyer, G. G. (1990). Zero defect software: McGraw-Hill, Inc.

Soltani, M., Panichella, A., & van Deursen, A. (2016). Evolutionary testing for crash reproduction. Paper presented at the Proceedings of the 9th International Workshop on Search-Based Software Testing.

Stützle, T. (1998). Local search algorithms for combinatorial problems. Darmstadt University of Technology PhD Thesis, 20.

Taillard, É. D., Gambardella, L. M., Gendreau, M., & Potvin, J.-Y. (2001). Adaptive memory programming: A unified view of metaheuristics. European Journal of Operational Research, 135(1), 1-16.

Thompson, J., & Dowsland, K. A. (1995). General cooling schedules for a simulated annealing based timetabling system. Paper presented at the International Conference on the Practice and Theory of Automated Timetabling.

Tierney, K. (2013). Late acceptance hill climbing for the liner shipping fleet repositioning problem. Paper presented at the 14th Workshop of the EURO Working Group "EU/ME: the metaheuristics community", 2013, Hamburg, Germany.

Tonella, P. (2004). Evolutionary testing of classes (1581138202). Retrieved from

Tracey, N., Clark, J., Mander, K., & McDermid, J. (1998). An automated framework for structural test-data generation. Paper presented at the Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on.

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

Vancroonenburg, W., & Wauters, T. (2013). Extending the late acceptance metaheuristic for multi-objective optimization. Paper presented at the Proceedings of the 6th Multidisciplinary International Scheduling conference: Theory & Applications (MISTA2013).

Varshney, S., & Mehrotra, M. (2013). Search based software test data generation for structural testing: a perspective. SIGSOFT Software Engineering Notes, 38(4), 1-6. doi:10.1145/2492248.2492277

Verstichel, J., & Berghe, G. V. (2009). A late acceptance algorithm for the lock scheduling problem Logistik Management (pp. 457-478): Springer.

Wegener, J., Sthamer, H., Jones, B. F., & Eyres, D. E. (1997). Testing real-time systems using genetic algorithms. Software Quality Journal, 6(2), 127-135.

Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. IEEE transactions on evolutionary computation, 1(1), 67-82.

Xanthakis, S., Ellis, C., Skourlas, C., Le Gall, A., Katsikas, S., & Karapoulios, K. (1992). Application of genetic algorithms to software testing. Paper presented at the Proceedings of the 5th International Conference on Software Engineering and its Applications.

Yuan, B., Zhang, C., & Shao, X. (2015). A late acceptance hill-climbing algorithm for balancing two-sided assembly lines with multiple constraints. Journal of Intelligent Manufacturing, 26(1), 159-168.

Zamli, K. Z., Al-Sewari, A. A., & Hassin, M. H. M. (2013). On test case generation satisfying the mc/dc criterion. International Journal of Advances in Soft Computing and Its Applications, 5(3).

APPENDIX A

Publications:

1. *An Automated Tool for MC/DC Test Data Generation,* Presented in IEEE Symposium on Computers & Informatics (ISCI) 2014, 130-135

2 *Comparative Performance Analysis of Simulated Annealingand Late Acceptance Hill Climbing Algorithm for Generating the MC/DC Compliant Test Suite,* Published in “8th SOFTEC Asia Conference 2015”.

Awards:

1. **Silver Medal in CITREx 2014** by UMP For project “A Web Based Automated Tool for Generating MCDC Compliant Test Suite”

2. **Silver Medal** in **Malaysia Technology Expo (MTE) 2015** for project MCDC + Pairwise Test Case Generation Tool "MC/DC Pro”.