Testing and Fault Localization of Phylogenetic Inference Programs Using Metamorphic Technique

by

Md. Shaik Sadi

A Thesis Submitted for the Degree of
Master of Science
at Faculty of Information and Communication Technologies

Swinburne University of Technology
John Street, Hawthorn- 3122
Australia
2013
Abstract

Many phylogenetic inference programs are available to infer evolutionary relationships among taxa using aligned sequences of characters, typically DNA or amino acid. These programs are often used to infer the evolutionary history of species. However, it is in most cases impossible to systematically verify the correctness of the tree returned by these programs, as the correct evolutionary history is generally unknown and unknowable. Neither is it possible to verify whether any non-trivial tree is correct in accordance with the specification of the often complicated search and scoring algorithms used during computation. Since there is either no mechanism or infeasible (called test oracle) to get a mechanism that we can use to verify the correctness of the returned tree, testing the correctness of any phylogenetic inference program suffers from the oracle problem (a well known problem in software testing). Due to the oracle problem, testing is ineffective in determining inputs that cause the program to fail, and thus the testing team is unable to pass failure-causing inputs to the debugging team for locating the fault(s). Though there are many testing and fault localization techniques, they cannot be applied when programs
suffer from the oracle problem. Here, we demonstrate how to apply a simple software testing technique, called *Metamorphic Testing (MT)*, to alleviate the oracle problem in testing and fault localization in phylogenetic inference programs. Metamorphic testing checks whether certain necessary properties (called metamorphic relations, MRs) of a program are satisfied based on multiple inputs and outputs of the programs. In case a MR is being violated, the program has a failure. We found that metamorphic testing can detect failures in faulty phylogenetic inference programs. Furthermore, we document our experiences in using MT in statistical fault localization.
To my loving parents....
Acknowledgements

I would like to express sincere gratitude to my principal coordinating supervisor Dr. Edmonds MF Lau for the excellent support and guidance given to me during the completion of my study. His valuable review helped me to complete the thesis. Without him, this dissertation would not have been possible.

I would like to extend thanks to Professor Tsong Yueh Chen, Dr. Michael Charleston and Dr. Joshua W.K. Ho for their valuable suggestions and support on my research work. I would also like to mention the ITS department and want to give my heartiest thank for their hardware support and technical suggestions for my experiment setup.

Finally, I would like to thank my beloved wife. She supported me every while, encouraged me a lot, discussed with me on the ideas of my research and gave me lots of valuable feedback. I wish to thank my colleagues and friends who encouraged me to complete this thesis. My sincere thanks also go to my parents, my elder brother and sister for their love, understanding, and support.
Declaration

I herewith declare that I have produced this thesis with my own work without the prohibited assistance of third parties. This study has not previously been presented as a thesis.

Md. Shaik Sadi,

Dated
The Author’s Publications

M.S. Sadi, F.-C. Kuo, J.W.K. Ho, M.A. Charleston, and T.Y. Chen,
Terminologies

Some of the terminologies will need to be elaborated before we go to the details of this thesis. These terms are grouped in the domain of Bioinformatics and Software Engineering:

**Bioinformatics**

**Nucleotide:** Basic building block that makes up nucleic acid like DNA and RNA [1].

**DNA (Deoxyribonucleic acid):** DNA is a nucleic acid containing the genetic information. It consists of nucleotides.

**DNA Sequence:** DNA sequence is the collection of atoms that make up the nucleic acid. DNA represents the biological information of a living thing.

**Taxon, plural Taxa:** Any classified group in biology which is related to organisms is known as taxon [2].

**Software Engineering**

**Failure:** The variation between the delivered output and the correct output is called a failure.
**Failure-causing input:** An input of a software is said to be failure-causing if it reveals the failure of the software. Otherwise it is a non-failure-causing input.

**Fault:** A fault might be an incorrect logic, step, data definition and usage in the software.
Contents

Abstract ii

Dedication iv

Acknowledgement v

Declaration vi

Publications vii

Terminologies viii

Contents x

List of Tables xiv

List of Figures xv

1 Introduction 1
## CONTENTS

1.1 Software Testing and Debugging .................................. 2
1.2 Bioinformatics Programs ........................................... 3
1.3 Phylogenetic Inference programs .................................... 4
1.4 Aim of This Thesis .................................................. 6
1.5 Structure of the Thesis .............................................. 7
1.6 Contributions ....................................................... 7

2 Background ............................................................ 9

2.1 Bioinformatics ....................................................... 9
2.2 Computational Phylogenetics ........................................ 10
2.3 Software Testing ..................................................... 11
   2.3.1 Limitation of Software Testing ............................... 12
2.4 Fault Localization .................................................. 14
   2.4.1 Slice-based Fault Localization ............................... 16
   2.4.2 State-based Fault Localization ............................... 16
   2.4.3 Spectra-based Fault Localization ............................. 17
   2.4.4 Statistical Fault Localization ................................. 18
2.5 Metamorphic Testing ............................................... 19
2.6 Related Work ....................................................... 20
   2.6.1 Applications of Metamorphic Testing ....................... 20
   2.6.2 Methodologies and Framework using MT .................... 21
   2.6.3 Metamorphic Testing of Bioinformatics Programs ........... 22
   2.6.4 Metamorphic Testing for Fault Localization in Non-Bioinformatics programs ........................ 23
3 Subject Selection and MR Generation

3.1 Importance of Subject Program ............................................... 24
3.2 Program Selection ................................................................. 25
3.3 DNAPARS Program ............................................................... 26
  3.3.1 Input ................................................................. 26
  3.3.2 How DNAPARS Works ..................................................... 27
  3.3.3 Output ................................................................. 28
  3.3.4 An example total length calculation .................................... 29
  3.3.5 Metamorphic Relation .................................................... 30
3.4 DNAPENNY Program ............................................................. 35
  3.4.1 Input ................................................................. 35
  3.4.2 How DNAPENNY Works .................................................. 35
  3.4.3 Output ................................................................. 36
  3.4.4 Metamorphic Relation .................................................... 37
3.5 DNAML Program ................................................................. 37
  3.5.1 Input ................................................................. 37
  3.5.2 How DNAML Works ......................................................... 38
  3.5.3 Output ................................................................. 38
  3.5.4 Metamorphic Relation .................................................... 38

4 Metamorphic Testing on Phylogenetic Inference Programs

4.1 Experimental Setup ............................................................. 41
  4.1.1 Input Selection ............................................................ 42
  4.1.2 Output Comparison ....................................................... 43
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.3</td>
<td>Mutants Generation</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Results and Analysis</td>
<td>46</td>
</tr>
<tr>
<td>4.2.1</td>
<td>DNAPARS Result</td>
<td>47</td>
</tr>
<tr>
<td>4.2.2</td>
<td>DNAPENNY Result</td>
<td>49</td>
</tr>
<tr>
<td>4.2.3</td>
<td>DNAML Result</td>
<td>50</td>
</tr>
<tr>
<td>4.3</td>
<td>Discussion</td>
<td>50</td>
</tr>
<tr>
<td>4.4</td>
<td>Threats to Validity</td>
<td>52</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Internal Validity</td>
<td>52</td>
</tr>
<tr>
<td>4.4.2</td>
<td>External Validity</td>
<td>52</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Construct Validity</td>
<td>53</td>
</tr>
<tr>
<td>5</td>
<td>Metamorphic Cooperative Bug Isolation</td>
<td>54</td>
</tr>
<tr>
<td>5.1</td>
<td>Cooperative Bug Isolation (CBI)</td>
<td>55</td>
</tr>
<tr>
<td>5.2</td>
<td>Application of MT in CBI</td>
<td>58</td>
</tr>
<tr>
<td>5.3</td>
<td>Experimental setup</td>
<td>60</td>
</tr>
<tr>
<td>5.4</td>
<td>Results and Analysis</td>
<td>61</td>
</tr>
<tr>
<td>5.5</td>
<td>Discussion</td>
<td>64</td>
</tr>
<tr>
<td>5.6</td>
<td>Threats to Validity</td>
<td>64</td>
</tr>
<tr>
<td>5.6.1</td>
<td>External Validity</td>
<td>64</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Internal Validity</td>
<td>65</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions and Future Work</td>
<td>66</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>69</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Mutants for *DNAPARS* .............................................. 45
4.2 New mutants for *DNAPENNY* ........................................ 45
4.3 Mutants for *DNAML* .................................................. 46
4.4 Metamorphic testing effectiveness in terms of killing mutants for
   *DNAPARS* program .................................................... 48
4.5 Metamorphic testing effectiveness in terms of killing mutants for
   *DNAPENNY* program ................................................ 49
4.6 Metamorphic testing effectiveness in terms of killing mutants for
   *DNAML* program ..................................................... 50
5.1 Fault Localization in *DNAPARS* program by *Importance* .... 62
5.2 Fault Localization in *DNAPENNY* program by *Importance* .... 63
List of Figures

3.1 Input file (infile) consisting DNA sequences . . . . . . . . . . . . . 26
3.2 Output tree in outfile generated by \textit{DNAPARS} program . . . . . 28
3.3 Output tree in newick format in outtree file for \textit{DNAPARS} and \textit{DNAML} . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.4 Total length calculation of \textit{DNAPARS} program . . . . . . . . . 29
3.5 Matrix format of DNA sequences . . . . . . . . . . . . . . . . . . . . 30
3.6 Output tree in outfile generated by \textit{DNAPENNY} program . . . . 36
3.7 Output tree in newick format in outtree file generated by \textit{DNAPENNY} without branch lengths . . . . . . . . . . . . . . . . . . . . . . . . . . 37
3.8 Output tree in outfile by \textit{DNAML} program . . . . . . . . . . . . 39
Chapter 1

Introduction

Testing and fault localization are difficult tasks in software development. They are essential means to ensure the quality of the software under development. Conventional software testing and fault localization techniques cannot be applied in some software application domain like bioinformatics, simulation, optimization and scientific computing [3] because either there is no mechanism (called “test oracle”) or it is infeasible to get a mechanism to verify the correctness of outputs of such software. This issue is known as the oracle problem in software testing.

Metamorphic testing (MT) [4] was proposed by Chen et al. that can alleviate the oracle problem in software testing. This innovative testing technique has been applied successfully in many application domains [5; 6; 7; 8] including bioinformatics [9]. Phylogenetic inference program, a subclass of bioinformatics programs, has its own special characteristics that have not been addressed by the metamorphic testing researchers.

In this thesis, we investigate the issues and present our results of applying
metamorphic testing in testing and localizing faults in phylogenetic inference programs.

1.1 Software Testing and Debugging

Software testing is to select some test inputs for the software under test, executing the software using the test inputs and verifying the testing result [10]. It refers to a process of executing the program with an intention to find failures [11]. Software testing can be either functional or non functional. Functional testing focuses on the softwares complains with the specifications. On the other hand non functional testing takes quality attributes such as reliability, maintainability, usability and portability in to account.

Many testing techniques have been built to test software and assume that there is a test oracle to verify the output(s) of the software. If the test oracle does not exist, it is not easy to find failure in the software. Unfortunately, many real life programs in the domain of, say, numerical analysis, bioinformatics, graph theory and simulation, suffer from the existence of test oracle [4; 5; 9; 12]. Furthermore, without the test oracle, it is then even more difficult to debug the software under test as developers may not be able to precisely determine when the software fails.

If the execution results of some test inputs on a program do not satisfy its specification/requirements, then we call it a faulty program. The process of locating and removing faults from a faulty program is known as software debugging. Although it has been an active area of research for the past decades, the fault
identification rate is still not so promising. There are two major steps in software debugging: fault localization and fault correction.

The process of identifying and localizing failures in the source code is known as fault localization and the action of correcting these faults is called fault correction. Fault localization is the most time-consuming and difficult task in software debugging [13].

Executed test input is either failure-causing or non-failure-causing. If an executed test input reveals failure of the software, the test input is defined as failure-causing input; otherwise it is a non-failure-causing input. Traditional fault localization process needs failure-causing inputs to re-execute the program and to check the program state on particular break points for finding the fault location in the source code. This traditional technique is time-consuming, error-prone. It requires (1) developers’ experience and knowledge on the source code to guess the faulty code block and (2) a test oracle to determine whether a certain test input is failure-causing. As mentioned previously, without a test oracle, it is difficult for developers to debug the software.

1.2 Bioinformatics Programs

Bioinformatics is an interdisciplinary research area that uses computational techniques to analyze biological data. Bioinformatics programs usually manage and analyze large and complex biological dataset to obtain biological information. Such information helps in the field of agriculture, human health, environments and biotechnology.
Many bioinformatics programs invoke complex processing procedures to search useful information within large and complex biological dataset. They pose a great challenge in developing a good testing strategy to ensure the reliability of the software being implemented because these programs often focus on building system-level biological models to analyze biological information. Building such models usually involves using computationally intensive methods. Hence, most of these programs adopt heuristic approaches. As a result, it is very difficult to verify the correctness of these programs due to the lack of a test oracle. As such, conventional testing approaches may not be applicable in these programs.

Since metamorphic testing can alleviate the oracle problem, and has already been successfully applied in many different fields to detect failures [5; 14; 15], bioinformatics researcher has also tried applying metamorphic testing in some bioinformatics programs [9]. These studies encourage us to apply MT on phylogenetics inference program, a special class of bioinformatics programs.

1.3 Phylogenetic Inference programs

A fundamental concept in biology is that different taxa (a collection of organisms i.e. different species) evolve from a common ancestor. The description of the evolutionary history of a group of taxa is called a phylogeny and is typically inferred from DNA sequences of different taxa. It is represented as an evolutionary tree, called a phylogenetic tree. Phylogenetic inference programs are used to infer the evolutionary history of a group of taxa and to generate a phylogenetic tree, and broadly applied in biological research. Besides, phylogenetic inference
programs are used in modern pharmaceuticals research for discovery of drug, designing of genetically enhanced organisms, and understanding of rapidly mutating viruses [16].

Different statistical and computational methods are available to infer phylogenetic trees. Development of such methods has been a major research focus of computational phylogenetics for more than 30 years [16; 17; 18]. Unfortunately, much less attention has been paid to the practice on how these methods are implemented in software. It is obvious that incorrect method implementation of the software can lead to an incorrect estimation of phylogenetic trees, which may misguide the design of follow-up experiments and analysis and then result in misleading biological conclusions.

Most of the phylogenetic inference programs are heuristic in nature [19], computationally expensive [20; 21] and use vast search space for calculating the phylogenetic trees. Therefore, determining a test oracle is difficult and hence it is difficult to test these programs and to localize faults in case developers need to debug the programs.

In numerical computation, two basic types of numeric errors are rounding error and truncation error. Rounding error is a miscomputation that results from rounding off numbers to a convenient number of decimals. For example, if 5.946782 is rounded to two decimal places (5.95) then the rounding error is \((5.95 - 5.946782) = 0.003218\). Truncation error is caused by truncating an infinite sum and approximating it by a finite sum. For example, the infinite series \(1/2 + 1/4 + 1/8 + 1/16 + 1/32\ldots\) adds up to exactly 1. However, if we add up first three terms and ignore the rest, we get \(1/2 + 1/4 + 1/8 = 7/8\), producing a truncation
error of $1 - 7/8$, or $1/8$.

Most of the phylogenetic inference programs use some score calculation to generate phylogenetic tree and, hence, suffer from truncation errors and rounding errors during the calculation process. Truncation errors are often neglected by most of the phylogenetic inference programs [22]. Although phylogenetic inference programs suffer from rounding error problem, some programs assume that the data are error free [23]. Due to these errors output might be changed in these programs. As there are no means to verify the output of these programs, it is difficult to test and localize the fault of these programs.

1.4 Aim of This Thesis

The main aim of this thesis is to ensure the correctness of phylogenetic inference programs. As we know testing and debugging are crucial tasks to ensure the program correctness of software, we focus on the testing and debugging of these programs. We found that these programs suffer from the oracle problem that makes it difficult and sometimes impossible to test and debug.

Furthermore, due to the oracle problem in phylogenetic inference programs, it is difficult to get the failure-causing and non-failure-causing inputs because these inputs are used by fault localization techniques to localize the fault. Hence debugging becomes difficult for phylogenetic inference programs.

This lack of testing and fault localization capability in phylogenetic inference programs encouraged us to apply and alleviate the oracle problem in phylogenetic inference programs using metamorphic testing. In particular, our aim is to (1)
address the oracle problem using metamorphic testing and (2) localize the faults using the metamorphic testing result for phylogenetic inference programs.

1.5 Structure of the Thesis

Chapter 2 describes literature review and background of this thesis. Chapter 3 discusses about the subject program selection and MR generation. Chapter 4 presents the metamorphic testing results on phylogenetics inference programs. Chapter 5 discusses the application of MT for localizing the faults in phylogenetics inference programs. Chapter 6 concludes the thesis with possible future work.

1.6 Contributions

In this study, we address the oracle problem with metamorphic testing in phylogenetic inference program. This study will also demonstrate to the bioinformatics community that to ensure the correctness of phylogenetic inference program, metamorphic testing is applicable and is effective for testing.

In phylogenetics, it is generally not possible to verify whether output, that is the estimated phylogenetic tree is correct, because it is not possible to go back in time and observe the evolutionary pattern. Here we will restrict ourselves to verifying whether the estimated tree is consistent with the intention of the methods that were used to construct these trees.

In literature, we found that fault localization takes lots of effort from total development task [24]. Though there are many fault localization techniques, most
of them cannot be applied when program suffers from the oracle problem. The result of violation and non-violation of MRs can help the debugging team to localize the fault(s) in software. In this study, we propose to apply MT to help localizing fault for phylogenetic inference programs. To sum up, our contributions to the testing and fault localization of phylogenetic inference programs are as follows.

- Most of the phylogenetic inference programs suffer from the oracle problem. This problem has been addressed and alleviated by metamorphic testing.

- We apply metamorphic testing result to help localizing faults in phylogenetic inference programs.
Chapter 2

Background

This chapter is divided in six main sections. The first section describes the bioinformatics, the second section describes computational phylogenetics, the third section describes software testing, the fourth and fifth sections describe fault localization and metamorphic testing, respectively and the last section describes related work.

2.1 Bioinformatics

The application of computer science and information technology to the field of medicine and biology is known as bioinformatics. It applies all the common technology used in software engineering. Bioinformatics programs are being used to manage and analyze large and complex biological dataset to obtain the biological information. Primary goal of bioinformatics is to increase the understanding of biological processes. To best understand the biological processes, bioinformatics
programs need to apply different computationally complex methods in the field of study. One of the major research areas of bioinformatics is computational phylogenetics where the evolutionary history of species is studied.

2.2 Computational Phylogenetics

As we know, the description of the evolutionary history of a group of taxa is called a phylogeny and the estimation of evolutionary relationships through DNA sequence comparison is called phylogenetics analysis. Phylogenetic analysis is a crucial task to discover the evolutionary history of species.

The application of computational methods to phylogenetic analysis is known as computational phylogenetics. Over the years, with the growth of computer capacity and refinement of computational logic techniques, computational phylogenetics has become an active research area. Many heuristic, stochastic and probabilistic models have been built to define the evolutionary relationship among taxa.

Two categories of computational method used in phylogenetic inference programs to evaluate DNA sequences and infer the evolutionary history are summarized as follows.

Maximum parsimony method infers the DNA sequences of different taxa and applies a set of algorithms to search for the phylogenetic trees containing the smallest total number of evolutionary changes. The evolutionary changes are the number of nucleotide changes between DNA sequences of taxa. More than one phylogenetic tree with the same number of evolutionary changes can be found
using this method. Searching the optimal phylogenetic tree using Maximum Parsimony method is a NP-hard problem [25]. Hence, many heuristics have been applied to find the optimal phylogenetic tree or close to the optimal phylogenetic tree. This Maximum Parsimony method can be slow when processing a large set of DNA sequences, and their computation suffers from rounding errors.

Another popular method for inference of phylogenetic tree is Maximum Likelihood. This method was first used in phylogenetic inference by Cavalli-Sforza [26]. To find the likelihood tree, a probabilistic model is used that maximizes the likelihood of a given set of DNA sequences. It uses probability to evaluate the evolutionary history. It is computationally expensive when applied to large set of DNA sequences as it requires to search all possible combinations of the tree topology.

2.3 Software Testing

The purpose of software testing is to find failures and ensure a certain level of quality before the release of the software. According to Myers [11], software testing is the process of executing a program with an intention to find failures. Parrington and Roper [27] argued that software testing cannot prove the absence of failures in software.

Gelperin and Hetzel [28] have classified five different stages of testing evolution according to the changing focus of testing during software life cycle. Until 1956, testing was mostly debugging focused. Baker differentiated testing from debugging [29]. From 1957 - 1978, testing was more about demonstration that
the program satisfies the specification. Later two periods of testing (1979-1982 and 1983-1987), called as destruction and evaluation oriented periods respectively, were more focused on finding the failures. The last and ongoing period of testing started from 1988 is referred to as prevention oriented whose main goal is to prevent failures. Testing remains the popular means of verifying program correctness.

For detecting failures it is advised to follow “the sooner the better policy”. In the software development process, repair cost goes higher if an issue is discovered at a later phase [30]. Adequate and effective testing is essential for ensuring the quality of software. One study has been done by NIST in 2002 which shows software defects cost US$ 22.2 to $59.5 billion annually from the U.S. economy [31].

Testing can be difficult due to the nature of the programs. For example, as explained earlier, most of the phylogenetic inference programs use heuristics to deal with a list of search spaces. The results depend on which computationally expensive algorithms have been used in the programs [20; 21]. Verifying correctness of these programs is difficult and hence it becomes difficult to test and debug.

### 2.3.1 Limitation of Software Testing

There are two major challenges in software testing. One of the challenges and the most general problem is incompleteness of software testing. It is often infeasible to test a software (S) with all possible test inputs as the input domain (D) can be infinitely large. Hence, most testers select a suitable test input set (T) for testing.
It is too difficult to select the size of $T$ such that the test results derived from $T$ can represent the results derived from $D$. This problem is known as reliable test set problem in software testing [32]. Reliable test set problem is an active research area, but this problem is not directly related to this study.

Another limitation of software testing is the oracle problem [32]. According to Cem Kaner [33], test oracle is the combination of an “originator”, a “comparator” and an “evaluator”. An originator stores expected results, a comparator compares the actual and expected results, and an evaluator determines whether the test results pass or fail. Defining a test oracle for software under test is not always easy due to the complex nature of the software. It is known as oracle problem in software testing when the test oracle is unavailable or is difficult to apply. Oracle problem, thus, can occur in the following two scenarios:

- Scenario 1: There is no available test oracle for the tester to verify the correctness of the output.

- Scenario 2: There is an oracle. However, it is infeasible or impractical for the tester to apply the oracle to verify the correctness of the output.

Scenario 2 can be illustrated with an example. Suppose we have a large spreadsheet with stock details of a supermarket. Each row in the spreadsheet corresponds to a stock item and stores unit cost, stock level and the total cost calculated as the unit cost multiplied with the stock level of a stock item ($= \text{unitcost} \times \text{currentstocklevel}$) is stored in the spreadsheet. Suppose there are 5000 stock items in the supermarket, which implies there will be 5000 rows in the
spread sheet. The grand total is calculated as the sum of total costs for all stock items. Now if we want to verify the grand total calculated by the spreadsheet, we can calculate the grand total manually with a calculator. In this case the oracle is \( \sum_{i=1}^{5000} S_i C_i \), where \( S_i C_i \) denote the current stock level and the unit cost of stock item \( i \), respectively. However, it is not feasible to type the cost and stock level of 5000 items and sum those up. This is an example of scenario 2 where there is an oracle however, it is infeasible to apply.

To address the oracle problem, Weyuker suggests to check the software outputs by some identity relations [34]. Identity relations are used by Cody and Waite [35] to test numerical programs. Comparing the outputs between previous versions of the software and the current version of software can also help in verifying output correctness [36].

2.4 Fault Localization

A necessary phase in software development is software debugging. This is a process of locating and correcting fault in software. Testing indicates the presence of failures in software and gives a list of failure-causing inputs. Given this list of inputs, debugging team can start finding the root cause of the failure in the program.

Fault localization is one of the two major steps in software debugging process [37]. The process of identifying faulty statement(s) in the source code is called fault localization. There are also two major phases in fault localization [38]. The first phase is to identify suspicious code that may contain failure and the
second phase is to examine the suspicious code to find whether it has the failure or not. According to Wong et al. [38] most of the fault localization techniques focus on the first phase of fault localization.

Fault localization is difficult [39], tedious and time-consuming [13]. To localize a fault, programmer’s intuition about the fault location is explored first. If it fails, then a particular fault localization technique will be applied to help the programmer to identify the faults by narrowing down the search domain of the potential failure causing statements. However, a specific fault localization technique is not necessarily applicable for every program [38] due to its complexity and nature.

Fault localization techniques use failure-causing and non-failure-causing test inputs to locate the fault in the source code. Failure-causing and non-failure-causing test inputs can be determined in the testing process using an existing test oracle. It is very difficult to determine the failure-causing and non-failure-causing inputs by the testing process when software suffers from the oracle problem. As a result, fault localization becomes difficult. We have found in the literature that to alleviate this problem in fault localization, a slicing technique mslice has been established by Xie et al. [40] and it was applied to localize the fault of those software that suffer from the oracle problem. Further discussion on mslice can be found in Section 2.6.

There are many fault localization techniques. Among them, slice-based [41], state-based [42], spectra-based [43] and statistical fault localization [44] are very well known. Most of these use the failure-causing and non-failure-causing inputs with their execution trace (the sequence of code executed during the execution
of a computer program with an input) to prioritize the suspicious code based on its likelihood of fault. Suspicious code with higher priority is checked before the code with lower priority.

### 2.4.1 Slice-based Fault Localization

Program Slicing [41] narrows down the program code and finds the most suspicious slice (a set of program statements) that could contain faults. At the very beginning, slicing is static and only uses source code for analysis. Later on, slicing uses the execution trace of a program [41] to analyze the potential fault. Slice-based fault localization does not show the location of a fault in a single statement, but it shows a suspicious slice. It is difficult to make a slice for a large program when one statement has many dependencies on other statements.

### 2.4.2 State-based Fault Localization

According to Wong [38], “a program state consists of variables and their values at a particular point during execution.” Program states are used in fault localization to identify the failure. In this approach variables are changed to determine which one is the cause of program failure.

Successful run is the execution of a non-failure-causing test input. On the other hand, failed run is the execution of a failure-causing test input. A state-based fault localization technique, called delta debugging, is proposed by Zeller et al. [42] It considers an execution of a program as a set of program states. This technique finds out the differences in states in successful and failed test runs.
with the help of their memory graphs [45]. It replaces the value of a variable at a state in a successful test run by the value of that variable at that state in a failed test run. The execution is repeated with the replaced variable value and if the same failure is not observed the variable is not considered relevant to the failure. This technique focuses on the variables and their values that are relevant to the failure. Although Delta debugging has been shown to be effective, the comparison of memory graphs makes it difficult in practice due to the increased size of the search space.

State based fault localization is an effective means of fault localization, but this technique assumes that there is a test oracle to verify the correctness of the program output.

### 2.4.3 Spectra-based Fault Localization

Program spectra are the execution profiles that show the execution path of a program. Spectrum-based fault localization is an approach based on a list of program spectra that works with successful and failed runs to evaluate which spectra have the higher chance of containing a fault.

There are several types of program spectra defined by Harrold [46], such as program spectra, path spectra, and branch spectra. Several tools have been built with this technique. Tarantula [43] is one of them and is well known. It focuses on coverage and execution result of test cases and ranks the statements based on suspiciousness. The suspiciousness of a particular statement $s$ is defined by

$$\frac{\% failed(s)}{\% passed(s) + \% failed(s)}$$
where \( \%\text{passed}(s) \) is the ratio of the number of successful test inputs that execute the statement \( s \) to the total number of successful test input in the test set and \( \%\text{failed}(s) \) is the ratio of the number of failed test inputs that execute the statement \( s \) to the total number of failed test input in the test set. The statement having the highest suspiciousness value has the most likelihood of program failure. Spectra-based fault localization technique is easy to use and can be integrated easily with any testing procedure. However this technique also assumes the existence of test oracle.

### 2.4.4 Statistical Fault Localization

Statistical fault localization technique works with successful and failed runs. The technique analyzes the execution traces and measures the behaviors of the program’s predicate to determine the fault \cite{47}. Most of the statistical fault localization technique instruments the program at some points where there are some branch conditions and functions return values \cite{48}. In this technique, the results of the predicates or the return values of a function are correlated to the failure of the program.

Statistical fault localization have some advantages and disadvantages. This technique ignores most of the predicates which are not related to the failure and, hence, reduces the complexity. This technique does not only measure the spectra difference but creates a model to measure the behavior of predicates. This technique also assumes the existence of test oracle to measure the behavior of predicates which is not always easy to get.
2.5 Metamorphic Testing

Metamorphic testing is an approach to alleviate the oracle problem. This approach does not rely on the test oracle to verify the output, but rather, checks the expected relations among inputs and outputs of the program under test. These relations are called Metamorphic Relations (denoted as MRs henceforth). They are derived based on the properties of the algorithm/specification being implemented. In this testing method, some initial test inputs called original test inputs are generated, using some existing test input generation methods. According to the derived MRs, new test inputs called follow-up test inputs are generated based on the original test inputs. The program is executed with both the original and follow-up test inputs, and their outputs are compared according to the MRs. If the comparison of any original and follow-up test input pair does not satisfy the corresponding MR then it implies that the program has a fault.

Metamorphic testing can best be understood with an example. Let us consider a program \( P \) that implements cosine function for testing. We know that \( \cos(0^\circ) = 1 \), \( \cos(60^\circ) = 0.5 \) and \( \cos(90^\circ) = 0 \). Suppose that \( \cos(59^\circ) \) returns 0.512. We do not know whether \( \cos(59^\circ) \) is computed correctly or not. We say that there is no test oracle to test this program. However, we know the property that \( \cos(2x) = 2\cos^2(x) - 1 \), which can be used as a metamorphic relation to test the program. We can take \( 59^\circ \) as the original test input. Then, \( 29.5^\circ (= 59^\circ/2) \) can be regarded as the follow-up test input, and we run the cosine program using \( 29.5^\circ \) as the input. We then verify whether \( P(59^\circ) \) is equal to \( 2(P(29.5^\circ))^2 - 1 \) or not. If \( P(59^\circ) \neq (2(P(29.5^\circ))^2 - 1) \), the MR is said to be violated and we can
conclude that the cosine program is incorrect. Thus, we are able to detect faults in the program \( P \) even we are not able to verify the correctness of the computed output of \( P(59^\circ) \) or \( P(29.5^\circ) \).

Testing programs using identity relation has been used long before. Identity relations are mostly used by numerical programs [35], however it is worth to note that MRs are not restricted to identity relations only. they can take any form of relations. Since metamorphic testing employs some relations between the input and output of a program for testing, this testing method does not need to know the correctness of individual outputs and therefore does not require a test oracle.

There are many programs that suffer from the oracle problem. In other words, the program has no test oracle or it is difficult to determine the oracle. The programs that implement heuristic algorithms are prominent among them. Heuristic algorithms are commonly used to solve problems that involve calculation of local optima [7]. The calculation is based on approximations that are guided by the available knowledge, and deliver an output that can be the global optima or local optima or close to these two optima. Determining the correctness of the output of such heuristic programs is difficult as no test oracle is available.

### 2.6 Related Work

#### 2.6.1 Applications of Metamorphic Testing

Metamorphic testing has been applied in many different software applications and has been found to be effective. It was applied in several numerical programs
and was also applied in integration, function minimization and linear equation [49]. Zhou et al. first applied MT in non-numerical problems like the shortest path algorithm, computer graphics and compiler design [5]. Empirical studies have been conducted to measure the effectiveness of MT by using different implementations of the matrix determinant computation program [50]. MT has been successfully applied in decision making algorithms and it found real fault in the program [6]. Apart from this, MT was successfully applied in some software application area like optimization [7], machine learning [8; 14], stochastic methods [51]. Chen et al. applied MT on a set of bioinformatics programs where there is no test oracle and found that MT is effective to find fault [9]. MT was found to be useful and effective for end user programmers [52]. MT was also applied in network simulation [12; 53] and web service [54].

2.6.2 Methodologies and Framework using MT

In literature we have found there are many attempts to integrate MT with other testing techniques. Chen et al. integrates MT with fault-based testing [32] and with global symbolic execution. Tse et al. applied MT in unit testing [55] and integration testing [56] in order to test context sensitive middleware applications. Empirical studies were done by Mayers and Guderlei [50] for selection of good metamorphic relations and found that a combination of MRs is more effective than a single one. For improving the effectiveness and efficiency, iterative metamorphic testing was proposed by Dong [57]. In that study, follow-up test inputs were used as the original test input and the testing was conducted using execution
path analysis. MT had also been applied to genetic algorithm where MRs were used to design the fitness function [58]. An automated testing framework using MT was first introduced by Gotleib and Botella [59]. This testing framework automated the testing process. In this study, the authors tried to automate a manual process which worked only for those programs that had been implemented with programming language C and they did not address any performance issues of this automated framework. Another Metamorphic testing framework was proposed by Murphy et al. [60]. In their work they generalized the framework for all programming languages and the tester did not need to access the source code.

2.6.3 Metamorphic Testing of Bioinformatics Programs

Software testing and fault localization are highly crucial tasks in software development. Quality of software highly depends on these. Chen et al. first introduced the use of MT on some bioinformatics programs and found this automatic testing technique is applicable on testing those bioinformatics programs that suffer from the oracle problem [9]. They applied MT on two application domain in bioinformatics: network simulation and high throughput data processing. They discussed the application procedure of metamorphic testing on those programs like MR generation from the domain knowledge and test input generation from the MRs. They generated nine faulty versions of GNLab [61] program and three faulty versions for SeqMap [62] program and then applied MT on those faulty versions. They found that different faulty versions of the program violate different MRs. They also discussed the applicability of MT on different domain of
bioinformatics.

2.6.4 Metamorphic Testing for Fault Localization in Non-Bioinformatics programs

Besides the application of MT in different domains, MT is also applied in fault localization. Based on MT, Xie et al. [40] introduced a new slicing technique called metamorphic slice (mslice) and applied it in spectrum based fault localization (SBFL). They found SBFL is infeasible to apply in many application domains that suffer from the oracle problem. When SBFL uses traditional slice it needs the testing result of a single test input whereas SBFL needs the metamorphic testing result of a metamorphic test pair (original test input and its corresponding follow-up test input) that violates and does not violate an MR when it uses the mslice. In many programs, it is difficult to obtain the testing result of a single test input due to the oracle problem. In that case, application of SBFL using traditional slice becomes difficult. To alleviate this problem, mslice is used in SBFL to localize the fault.

To generate the mslice they use the union of the execution traces of both original and follow-up test inputs. Mslice was used in spectra based fault localization and found no significant difference between the empirical results of mslice with other slicing techniques. So, use of mslice in SBFL is an effective means to localize the fault in those programs that suffer from test oracle problem.
Chapter 3

Subject Selection and MR Generation

To do testing and fault localization on phylogenetic inference programs by using Metamorphic testing, we first selected subject programs and identified MRs. The following sections will describe the selection of subject programs and description of identified MRs.

3.1 Importance of Subject Program

This research is designed to investigate the applicability of MT to address the oracle problem in phylogenetic inference programs. There are a number of phylogenetic inference programs. The phylip package is the oldest widely-distributed, one of the most widely used, the sixth most frequently cited phylogeny pack-
age by the bioinformatics community [63]. In this package, there are many such
programs. Among which, we select dnapars, dnapenny and dnaml for our inves-
tigation because they do not have non-deterministic behaviour. Since the use of
MT requires certainty in MRs to generate follow-up test cases, MT does not work
well for non-deterministic programs 1.

3.2 Program Selection

Phylogenetic inference programs are extensively used in bioinformatics research.
A number of phylogenetic software packages namely PHYLIP [63], PAUP [64],
MEGA [65], MRBAYES [66], RAxML [67] etc. are available for generating phy-
logenetic trees. We have chosen DNAPARS, DNAPENNY and DNAML pro-
grams from PHYLIP version 3.68 for this study. DNAPARS, DNAPENNY and
DNAML have 8600, 7781 and 9527 lines of code excluding the comment lines,
respectively. All the programs are written in C programming language. These
programs present the user with a command-line interface to execute different
software functionalities. Different execution options may require different input
files. However, all programs require DNA sequences of different taxa. Some pro-
grams also ask a phylogenetic tree to be inputted. Inputs, outputs, algorithms
and related MRs of these three programs are detailed in the following sections.

1Although we cannot guarantee that the testing result on these selected subject programs
represent the whole picture of testing phylogenetic inference programs, however this can give
us an idea for further investigations
3.3 DNAPARS Program

3.3.1 Input

One essential input file called “infile” to the DNAPARS programs is the one consisting of multiple taxa, each of which is represented by a DNA sequence. In each DNA sequence of a taxon, each character is called a “nucleotide”. Nucleotides are the basic building block of nucleic acids. \( n \) taxa and \( m \) nucleotide are presented by a \( n \times m \) matrix (\( n \) rows and \( m \) columns). Each line of the input file contains the name of the taxa with its DNA sequence. The first 10 characters of a line is the name of taxa (must be filled by blank spaces if the species name has fewer than ten characters). The column of nucleotides is called “site”. A sample input file containing 20 DNA sequences of 50 sites is shown in Figure 3.1.

```
20 50
spe0 ATGAGGCTCTGAGGGTCCTGACCAGCTACAGTGCTGTGGCCGCGAGGCTTTGGCGGCCG
spe1 AAGGTTTACACCTGCTTGTGCGGTAGTCTCAGGGCGATCAGGTAGGGCG
spe2 TATATAGGCGGAGCTTTTTCTACT CCTCCTCTTACTGTTGCTCTG
spe3 GGGGCTGGGGGCTGGGAGCTTTACTACTGGTGGAGGGCGATCAGGTAGGGCG
spe4 ATTGACAGATGGGCGGCAAAAAATATAGGGAGGGCTTTCAAGGGTCTCAC
spe5 TAAATTTACACTGGTCTCTGACATCTCCCGGAGTCTTTTCTCTCTCTTT
spe6 CGGTCTACGCGGCAAAGCTTGCTTACGGCAACTATCTCTTCTCTGCTACGGG
spe7 CATGGACGGGATGGGATGGGATGGGATGGGATGGGATGGGATGGGATGGG
spe8 CCCGACTGAGGGCAGATCTACATTCTACCTGCGAGCTTTTACCAGGGCA
spe9 CGATTCCGAGCTGATCTACATTCTACCTGCGAGCTTTTACCAGGGCA
spe10 TACCGGTCTCAGGTTAAGGTAGGAGTACCTTTTTGAGGAGACACCAAA
spe11 CCGTACGGTCTGAGGGGAGATTTTTTAGGAGTTATGTGAATCTCTCTGAGG
spe12 TAGACGAGGTGATGTTCTGAGGGCTTTTTCTCTATTCTCTATTCTCTATTCTCT
spe13 TACCTGTACGGGGAACCCGCTAACATGTGGAAGCTGCAACCCGAGCCATCCACCC
spe14 GGAGCGGGGAGGCGGCGAACGCGATTTTTAGGAGTTATGTGAATCTCTCTGAGG
spe15 ATGGTACGGTCTGAGGGGAACCCGCTAACATGTGGAAGCTGCAACCCGAGCCATCCACCC
spe16 GAAATCGCGAGGAAGGTGTTGCTAACCAGGACGAGCGCACTTTTTGAGGAGTTATGTGAATCTCTCTGAGG
spe17 TAGAGCGGGGAGGCGGCGAACGCGATTTTTAGGAGTTATGTGAATCTCTCTGAGG
spe18 ATGGTACGGTCTGAGGGGAACCCGCTAACATGTGGAAGCTGCAACCCGAGCCATCCACCC
spe19 ACAATACTGTTGTGTGGGAAACAAAGATCATACTACAAGGACAGCGGCTAT
```

Figure 3.1: Input file (infile) consisting DNA sequences
3.3.2 How *DNAPARS* Works

*DNAPARS* implements the maximum parsimony method as discussed in Section 2.2 to construct phylogenetic trees. Based on the given DNA sequences, *DNAPARS* calculate nucleotide changes (or called evolutionary steps) among sites, to generate phylogenetic trees. Evolutionary steps calculated for the entire tree is called the total length, while those calculated for the branch is called the branch length. To get a better idea of calculating the nucleotide changes or the evolutionary steps, please see subsection 3.3.4.

The maximum parsimony method aims to minimize the number of evolutionary steps to construct the maximum parsimony tree. Maximum parsimony tree is a phylogenetic tree which has the smallest total length. To get this maximum parsimony tree, an initial phylogenetic tree is prepared with the first \( n \) taxa of the infile (\( n = 3 \) for *DNAPARS*). The *DNAPARS* program then expands this tree by appending one new taxon at a time, and search for the maximum parsimony tree.

*DNAPARS* uses a heuristic algorithm to search for a locally optimal tree. The main objective of this heuristic algorithm is to minimize the total number of changes needed to describe the evolution of given DNA sequences. The heuristics proceeds as follows: adding one taxon to the tree, each pair of adjacent branches may be swapped to get the local maximum parsimony tree. Once all the taxa are added to the tree, subtree rearrangements are attempted to find the global maximum parsimony tree.
3.3.3 Output

*DNAPARS* aims to construct trees with the shortest “total length”. The outputs of *DNAPARS* is stored in files. One is called “outfile” another one is called “outtree”. Figure 3.2 shows the one output tree in “outfile” and the total length in the figure is 452.00. “outtree” file presents the output tree(s) in Newick format [68]. Figure 3.3 is an example of Newick format of the tree for *DNAPARS* program where 0.35697, 0.22394, 0.29697 etc. are the branch lengths.

```
+-----spe18
 |     |-----spe17
 |     |       |-----spe6
 +-----8 +-----spe14         +-----spe10
 |     |     |-----spe12        |-----spe2
 |     |     +-----18        |     |-----spe16
 |     |                +-----12        +-----spe13
 |     |                       |     +-----spe16
 |                         |     |-----spe5
 |                       +-----spe10
 |                     +-----spe19
 |                     +-----spe11
 |                     +-----spe4
 |                     +-----spe5
 +-----15

Figure 3.2: Output tree in outfile generated by *DNAPARS* program
```
3.3.4 An example total length calculation

To depict the calculation of total length, we present the following input and output. We consider an input file containing 5 species and 11 taxa shown in figure 3.4(a). The corresponding nucleotide calculation is also shown in figure 3.4(b) and output tree is shown in figure 3.4(c). If we scan the first site (column in input) in the input file, we encounter ‘A’, ‘A’, ‘C’, ‘G’ and ‘G’. The first nucleotide is considered as the base. If the same nucleotide is found again, a ‘.’ (dot) is placed to indicate no change. So for the first site, we put ‘A’, ‘.’, ‘C’, ‘G’ and our calculations shows that there are 2 changes (‘A’->‘C’, ‘C’->‘G’).

In this way the number of changes are calculated for all the sites and the sum of all these changes is referred to as the total length. In this example, the calculated total length is 14.

![Figure 3.4: Total length calculation of DNAPARS program](image)
3.3.5 Metamorphic Relation

In this study, we have analyzed the properties of the chosen DNAPARS program and defined some relevant metamorphic relations. Seven metamorphic relations were developed for this program. We will represent the DNA sequences as a matrix $X$ to facilitate the discussion of the MRs. For $n$ taxa and $m$ sites, matrix $X = \{x_{ij} | 1 \leq i \leq n, 1 \leq j \leq m\}$, where $x_{ij}$ is a nucleotide. $A$, $T$, $C$, $G$ are the most common nucleotide encountered in real DNA sequences, hence in the study all $x_{ij} \in \{A, T, C, G\}$. An example input $X$ is given in Figure 3.5.

We use $X$ and $X'$ to denote the original and the follow-up inputs, respectively when describing MRs. We also use $T$ and $T'$ to represent a set of original and follow-up output trees for DNAPARS, and $t$ and $t'$ to denote the corresponding total lengths.

$$X = \begin{pmatrix}
  x_{11} & x_{12} & \cdots \\
  x_{21} & x_{22} & \cdots \\
  \vdots & \vdots & \ddots
\end{pmatrix} = \begin{bmatrix}
  ATCGAAGCAA \\
  AGCGATGTTG \\
  AGCGATATTT \\
  ATTGATGCAC
\end{bmatrix}$$

Figure 3.5: Matrix format of DNA sequences

The followings show the terminologies used by the phylogenetics community. We have used these terminologies for explaining our MRs.

- **Parsimony-uninformative site (also called conserved site):** These are sites that contain the same nucleotide in all sequences (e.g., sites 1, 4 and 5 in Figure 3.5)

- **Hypervariable site:** If all the nucleotides of a site are different then the site is called Hypervariable site (e.g., site 10 in Figure 3.5).
• **Singleton site:** The sites that are mostly conserved except for a change in one sequence (i.e. sites that have two types of nucleotides, one occurs \(n-1\) times and the other one occurs only once in the sequence) are called singleton sites [65] (e.g., sites 3, 6 and 7 in Figure 3.5).

• **Parsimony-informative site:** All sites other than the above three (Parsimony-uninformative, Hypervariable and Singleton) sites provide some useful information for constructing a phylogenetic tree, and therefore are called parsimony-informative sites (e.g., sites 2, 8 and 9 in Figure 3.5).

The MRs for **DNAPARS** are discussed below.

**MR1:** If we generate a follow-up input \(X'\) by swapping two sites (the columns) in the original input \(X\), then the set of original and follow-up output trees \(T\) and \(T'\) are identical and their corresponding total lengths \(t\) and \(t'\) are equal. Thus MR1 corresponds to the rule that the output of the programs should be independent of the order of the sites.

**Example:** The follow-up input \(X'\), by interchanging column 3 and column 8 of the original input \(X\) in Figure 3.5 looks like:

\[
X' = \begin{bmatrix}
A & G & T & G & A & T & G & C & T & G \\
A & G & T & G & A & T & A & C & T & T \\
A & T & C & G & A & T & G & T & A & C
\end{bmatrix}
\]

**Expected Output Relation:** \(T = T'\) and \(t = t'\).

**MR2:** If we insert \(k \ (k>0)\) number of parsimony-uninformative sites into the original input \(X\) to generate a follow-up input \(X'\), then the set of original and follow-up output trees \(T\) and \(T'\) are identical and their corresponding total lengths
$t$ and $t'$ are equal. Insertions of parsimony-uninformative sites are order independent and hence can be placed after any site of the original input $X$.

**Example:** We add five ($k=5$) parsimony-uninformative sites (consisting of nucleotide character A) into the original input $X$ in Figure 3.5 to generate a follow-up input $X'$:

$$X' = \begin{bmatrix}
\end{bmatrix}$$

**Expected Output Relation:** $T = T'$ and $t = t'$.

**MR3:** If we remove some parsimony-uninformative sites from the original input $X$ to generate a follow-up input $X'$, then the set of original and follow-up output trees $T$ and $T'$ are identical and their corresponding total lengths $t$ and $t'$ are equal. This MR, like the previous one, corresponds to the rule that the output of the parsimony-based programs should be completely independent of what we have classed as parsimony-uninformative sites.

**Example:** We can see that there are three parsimony-uninformative sites (sites 1, 4 and 5) in the original input $X$ in Figure 3.5. If we remove the two parsimony-uninformative sites (sites 1 and 5) from the original input $X$ in Figure 3.5 and generate $X'$, $X'$ looks like:

$$X' = \begin{bmatrix}
T & C & G & A & G & C & A & A \\
G & C & G & T & G & T & T & G \\
G & C & G & T & A & T & T & T \\
T & T & G & T & G & C & A & C
\end{bmatrix}$$

**Expected Output Relation:** $T = T'$ and $t = t'$.
MR4: If we extend the DNA sequences in the original input $X$ by the concatenation of each DNA sequence with itself to generate a follow-up input $X'$, then the set of original and follow-up trees $T$ and $T'$ are identical and the follow-up total length $t'$ is twice the original total length $t$. This corresponds to our belief that the (local) optimality of any tree will not be affected by duplicating all the data.

Example: The follow-up input $X'$ by concatenating the DNA sequence with itself at the end in the original input $X$ in Figure 3.5 is:


Expected Output Relation: $T = T'$ and $2t = t'$

MR5: If we add some hypervariable sites into the original input $X$ to generate a follow-up input $X'$, then the set of original and follow-up output trees $T$ and $T'$ are identical. Hypervariable site(s) can be placed after any site of the original input $X$. This MR is only true for the input file of $n = 4$ sequences. This MR specify that parsimony-based programs should be completely independent of sites that do not provide additional information about the tree structure.

Example: We add two hypersensitive sites to the original input $X$ in Figure 3.5 to generate follow-up input $X'$:


Expected Output Relation: $T = T'$

33
MR6: If we apply the same transformation to permute all the characters in every DNA sequence, for example (A→T, T→G, G→C, C→A), in the original input X to generate a follow-up input X’, then the set of original and follow-up trees T and T’ are identical and their corresponding total lengths t and t’ are equal. Thus MR6 corresponds to the rule that the output is independent of the label we ascribe to each character.

Example: We create a follow-up input X’ from the original input X in Figure 3.5 by changing (A→G, T→C, G→A, C→T):

\[
X' = \begin{bmatrix}
G & C & T & A & G & G & A & T & G & G \\
G & A & T & A & G & C & G & C & C & C \\
G & C & C & A & G & C & A & T & G & T \\
\end{bmatrix}
\]

Expected Output Relation: T = T’ and t = t’.

MR7: If we add a duplicate DNA sequence of any taxon in the original input X to create a follow-up input X’, then, (1) the trees of the original and follow-up output sets, T and T’ respectively, should differ only for the duplicate taxa such that in the follow-up output tree, the duplicates are grouped together in a subtree, and (2) the total lengths of the original and follow-up trees t and t’ should be the same and the output is independent on where the duplicate DNA sequence is placed. This is equivalent to saying that identical sequences must be joined in a subtree of zero length, which is an assumption of any phylogenetic method.

Example: By adding the duplicate DNA sequence of the first taxon (first
row) before the third row of original input $X$ in Figure 3.5, a follow-up input $X'$ will be created as follows:

$$X' = \begin{bmatrix}
A & G & C & G & A & T & G & T & T & G \\
A & G & C & G & A & T & A & T & T & T \\
A & T & T & G & A & T & G & C & A & C
\end{bmatrix}$$

**Expected Output Relation:** $T = T'$ (except the subtree of duplicate taxon will be grouped with the taxon being duplicated) and $t = t'$.

### 3.4 DNAPENNY Program

#### 3.4.1 Input

*DNAPENNY* uses the same input file “infile” as *DNAPARS* for generating phylogenetic trees. For details of “infile”, see Section 3.3.1.

#### 3.4.2 How DNAPENNY Works

*DNAPENNY* also implements the maximum parsimony method (discussed in Section 2.2) to construct trees. Although *DNAPARS* and *DNAPENNY* programs have a common goal, they use different algorithms to generate the phylogenetic tree. To get the maximum parsimony tree, an initial phylogenetic tree is prepared with the first $n$ taxa of the infile ($n = 2$ for *DNAPENNY*). The *DNAPENNY* program then expands this tree in the same way as *DNAPARS* by appending one new taxon at a time, and search for the maximum parsimony tree.
DNAPENNY uses the “branch and bound” algorithm to identify a global maximum parsimony tree [69]. At each step of tree construction, if the length of a branch exceeds the predefined bound, that branch will not be extended further and other branches will be tried. However, this branch and bound algorithm is more computationally expensive than the heuristics used in DNAPARS.

3.4.3 Output

DNAPENNY also generates “outfile” and “outtree” files. Figures 3.6 and 3.7 show an example tree generated by DNAPENNY in “outfile” and “outtree” files, respectively. We found 465.000 total length in Figures 3.6. DNAPENNY does not output branch lengths to the “outtree” file.

Figure 3.6: Output tree in outfile generated by DNAPENNY program
3.4.4 Metamorphic Relation

Since DNAPENNY and DNAPARS both implement Maximum Parsimony method, we use same seven MRs as discussed in Section 3.3.5 for both the programs.

3.5 DNAML Program

3.5.1 Input

DNAML program also use the same “infile” as DNAPARS and DNAPENNY programs for generating the phylogenetic tree. However, executing the DNAML program with menu options (“U” together with “L”) requires another input file called “intree”. The “intree” file contains a phylogenetic tree in newick format.

Since one output of the DNAML program is “outtree” containing the newick format of phylogenetic trees, this “outtree” file can be renamed as “intree” and used as an input file for DNAML. The “outtree” of DNAML is the same as that of DNAPARS. Figure 3.3 shows an example of “intree” file of DNAML.
3.5.2 How *DNAML* Works

*DNAML* uses the maximum likelihood method mentioned in Section 2.2 to maximize the likelihood of a given set of DNA sequences. In this method, the evolution of taxa is considered as a stochastic process, in which “evolutionary changes among sites” depend on some set of probabilities generated by the Markov model [70]. Based on each set of probabilities generated, the nucleotides are changed to get the likelihood tree.

3.5.3 Output

*DNAML* aims to generate trees with the highest “likelihood” against an evolutionary model. *DNAML* also generates “outfile” and “outtree” file. Figure 3.8 shows an example tree generated by *DNAML* in “outfile”. It does not output total lengths to the “outfile” file rather it outputs the highest likelihood. “outtree” file is same as the “outtree” file of *DNAPARS* program.

3.5.4 Metamorphic Relation

For *DNAML*, we generated two different metamorphic relations and used those MRs in testing *DNAML*. Details of these MRs will be given later. For running the original and follow-up inputs on *DNAML*, we have used two input files, called “infile” (DNA sequence file) and “intree” (containing phylogenetic trees of the DNA sequences). The “intree” file is actually the “outtree” file which is generated by the *DNAML* program while executing the program with the default option. The default option only uses the “infile” as input and constructs “outtree” and
“outfile” as output. We rename the “outtree” file as “intree” file and use it as input to execute the program for testing. For the first MR, “infile” is the same for original and follow-up inputs but intrees are different. For the second MR, “intree” is the same in both original and follow-up test inputs and infiles are different.

For ease of discussion we will use $X$ in Figure 3.5 as the original input. The “intree” file of $X$ is denoted as original “intree” $S$ and is given below:

$$(seq4:0.06250(seq3:0.16250,seq2:0.06250):0.25000,seq1:0.26250);$$

We will also use $S'$ to denote a follow-up “intree”. For DNAML, $l$ and $l'$ represent original and follow-up likelihood, respectively.
MR A: If we generate a tree by swapping two taxa of any subtree in the bottom layer of the original intree file \( S \) where the two taxa share one immediate ancestor and if we append the newly generated tree to the original intree file to generate a follow-up intree file \( S' \), then the set of original and follow-up output trees \( T \) and \( T' \) are identical and their corresponding likelihoods \( l \) and \( l' \) are equal.

Example: We swap seq2 and seq3 in the original intree \( S \) to generate a modified tree and append the modified tree to \( S \) to generate a follow-up intree \( S' \):

\[
(seq4:0.06250,(seq3:0.16250,seq2:0.06250):0.25000,seq1:0.26250);
(seq4:0.06250,(seq2:0.06250,seq3:0.16250):0.25000,seq1:0.26250);
\]

Expected Output Relation: \( T = T' \) and \( l = l' \).

Note that in the application of this MR, the infiles of the original and follow-up inputs are the same, but their intrees are different.

MR B: This MR is similar to MR6 defined to test DNAPARS and DNAPEENNY in the previous subsection. If we generate the follow-up input as mentioned in MR6 and also use intree, the sets of follow-up and original output tree are identical and their corresponding likelihoods are equal.

Example: Same as the example used to explain MR6.

Expected Output Relation: \( T = T' \) and \( l = l' \).

Note that in the application of this MR, the intrees of the original and follow-up inputs are the same, but their infiles are different.
Chapter 4

Metamorphic Testing on

Phylogenetic Inference Programs

This chapter describes the experiment setup and metamorphic testing results for

\textit{DNAPARS, DNAPENNY} and \textit{DNAML} programs.

4.1 Experimental Setup

In the experiment, the original and follow-up inputs generation as well as original and follow-up outputs verification were conducted automatically. The automated process was implemented using the programming language C# because I had experience in coding with C#.
4.1.1 Input Selection

As mentioned in Section 2.5, the original inputs of each MR can be generated using some existing test input selection methods. For testing phylogenetic inference programs, we have considered two types of inputs, namely real and random. Real and random inputs are generated from real and random DNA sequences respectively. A set of 5000 real inputs which consists of real DNA sequences of different taxa (hereafter, referred to as $Original_{\text{real}}$) are obtained from TreeFam [71]. TreeFam (Tree families database) is a repository of phylogenetic trees of animal genes and have a good collection of real DNA sequences. It also gives reliable information of evolutionary history of species. As mentioned in Chapter 3, each DNA sequence of taxon is a string of character set. Each random DNA sequence is a string of characters, and each of the character is randomly generated from the characterset $\{A, T, G, C\}$. On the other hand, another set of 5000 random inputs which consists of random DNA sequences (hereafter, referred to as $Original_{\text{random}}$) that were randomly generated. Hence, a total of 10000 inputs are generated. These inputs are used in Section 4.1.3 to check whether (1) the faulty versions of the program are syntactically equivalent to the original program or not and (2) to check the obvious fault (like crash during execution, falling in infinite loops, containing printing mistakes etc.) in the faulty version of the program.
4.1.2 Output Comparison

The three programs chosen for this experiment (DNAPARS, DNAPENNY and DNAML) may generate one or more phylogenetic tree as output for a single input. Each tree of the original output is matched against all the trees of the follow-up output. We have focused on the tree structure and the total length for testing.

4.1.3 Mutants Generation

Mutation analysis [72] was conducted to measure the effectiveness of metamorphic testing for DNAPARS, DNAPENNY and DNAML. Mutants are faulty program versions generated by seeding faults in the original version of a program. If any input pair shows the violation of a particular MR in a mutant, we can say the mutant has been killed by the MR. Some very simple mutation operators are used to generate the mutants for our experiment. Mutants were generated randomly by using an automated Perl script. The script can generate mutants by randomly mutating one statement at a time.

The PHYLIP package includes many source code files to create the executable files. As DNAPARS, DNAPENNY and DNAML all use seq.c and phylip.c files, we decided to generate the mutants by modifying these two files. However, most parts of the code in phylip.c are related to exception handling. So we left that file unchanged and only mutated seq.c file for generating mutants. Initially, 30 mutants were generated by mutating seq.c.

We have randomly selected one among DNAPARS, DNAPENNY and DNAML
programs to start our experiment with. The first selected program was DNA-
PARS and we ran the program with the mutated seq.c files. We have excluded those mutants having the same outputs as the original version after running 10000 inputs (input set (Original_random) and input set (Original_real)) because these mutants could be semantically equivalent to the original program. We have also excluded those mutants that have obvious faults, such as crash during execution, not generating any output tree, falling in infinite loops, containing printing mistakes in tree structure of the output, printing negative lengths because detecting these faults do not require any test oracle.

From a review of the literature we failed to find any indication as to what can be a sufficient number of mutants to be used for an experiment. Also a review of the relevant research indicates 3 and 9 mutants were used [9] respectively in two different programs. As such, we limited the number of mutants to be used for this initial experiment to ten. And finally, ten mutants (M1 - M10) listed in Table 4.1 were selected for our study.

DNAPENNY has been selected as our second program for the experiment. It will be good if DNAPENNY can also use these 10 mutants, M1 - M10, created for DNAPARS. However, after running the DNAPENNY executable file with 10000 inputs (input set Original_random and input set Original_real), faults in M2, M4, M6, M8, M9 and M10 in Table 4.2 were found to be irrelevant to DNAPENNY because the corresponding faulty statements in seq.c were never executed by DNAPENNY. These six mutants were then excluded from the study of DNAPENNY. We then generated another six new mutants for DNAPENNY by mutating seq.c file. These six new mutants are given in Table 4.2. These six
Table 4.1: Mutants for DNAPARS

<table>
<thead>
<tr>
<th>Mutant</th>
<th>File</th>
<th>Line#</th>
<th>Original Statement</th>
<th>Faulty Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>seq.c</td>
<td>738</td>
<td>ns = 1 &lt;&lt;G;</td>
<td>ns = 1 &lt;&lt;C;</td>
</tr>
<tr>
<td>M2</td>
<td>seq.c</td>
<td>2807</td>
<td>if (i == j)</td>
<td>if (i != j)</td>
</tr>
<tr>
<td>M3</td>
<td>seq.c</td>
<td>565</td>
<td>if (ally[alias[i - 1] - 1] != alias[i - 1])</td>
<td>if (ally[alias[i - 1] - 1] &gt;= alias[i - 1])</td>
</tr>
<tr>
<td>M4</td>
<td>seq.c</td>
<td>1115</td>
<td>for (i = a; i &lt;b; i++)</td>
<td>for (i = a; i &lt;= b; i++)</td>
</tr>
<tr>
<td>M5</td>
<td>seq.c</td>
<td>567</td>
<td>j = i + 1;</td>
<td>j = i - 1;</td>
</tr>
<tr>
<td>M6</td>
<td>seq.c</td>
<td>992</td>
<td>for (i = (long)A; i &lt;= (long)O; i++)</td>
<td>for (i = (long)A; i &gt; (long)O; i++)</td>
</tr>
<tr>
<td>M7</td>
<td>seq.c</td>
<td>575</td>
<td>itemp = alias[i - 1];</td>
<td>itemp = alias[i + 1];</td>
</tr>
<tr>
<td>M8</td>
<td>seq.c</td>
<td>1137</td>
<td>for (j = (long)A; j &lt;= (long)O; j++)</td>
<td>for (j = (long)A; j &gt; (long)O; j++)</td>
</tr>
<tr>
<td>M9</td>
<td>seq.c</td>
<td>1077</td>
<td>else p-&gt;numsteps[i] += weight[i];</td>
<td>else p-&gt;numsteps[i] -= weight[i];</td>
</tr>
<tr>
<td>M10</td>
<td>seq.c</td>
<td>1182</td>
<td>for (j = (long)A; j &lt;= (long)O; j++)</td>
<td>for (j = (long)A; j &gt; (long)O; j++)</td>
</tr>
</tbody>
</table>

Mutants were used for DNAPENNY together with M1, M3, M5 and M7. As a result, we have 10 mutants for DNAPENNY.

Table 4.2: New mutants for DNAPENNY

<table>
<thead>
<tr>
<th>Mutant</th>
<th>File</th>
<th>Line#</th>
<th>Original Statement</th>
<th>Faulty Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>M11</td>
<td>seq.c</td>
<td>959</td>
<td>if (p-&gt;base[i] == 0) {}</td>
<td>if (p-&gt;base[i] != 0) {}</td>
</tr>
<tr>
<td>M12</td>
<td>seq.c</td>
<td>566</td>
<td>if (j &lt; i)</td>
<td>if (j &gt; i)</td>
</tr>
<tr>
<td>M13</td>
<td>seq.c</td>
<td>1277</td>
<td>if (p-&gt;back == p1)</td>
<td>if (p-&gt;back != p1)</td>
</tr>
<tr>
<td>M14</td>
<td>seq.c</td>
<td>726</td>
<td>for (i = 0; i &lt; spp; i++)</td>
<td>for (i = 0; i &gt;= spp; i++)</td>
</tr>
<tr>
<td>M15</td>
<td>seq.c</td>
<td>1278</td>
<td>else if (p-&gt;back == p2)</td>
<td>else if (p-&gt;back != p2)</td>
</tr>
<tr>
<td>M16</td>
<td>seq.c</td>
<td>1479</td>
<td>if (other == *root)</td>
<td>if (other != *root)</td>
</tr>
</tbody>
</table>

Related to the mutants of DNAML, it is unfortunate that all the mutants for DNAPARS and DNAPENNY are related to the parsimony method. Since DNAML use the maximum likelihood method to generate phylogenetic tree, MR1-MR7 are not applicable for DNAML program. That is why another two MRs (MR
A and MR B) are defined for DNAML. Since, these two newly defined MRs are related to the permutations of the taxa in a subtree as well as permutation of characters (represented as characters i.e. A, C, G, T) in the input DNA sequences, we have to mutate the code that work on the characters of DNA sequences. As a result, a new set of five mutants were generated for DNAML. Table 4.3 lists these five mutants.

<table>
<thead>
<tr>
<th>Mutant</th>
<th>File</th>
<th>Line#</th>
<th>Original Statement</th>
<th>Faulty Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>M17</td>
<td>seq.c</td>
<td>464</td>
<td>sumg += w * (*freqg) * treenode[i] - &gt;x[j][0][(long)G - (long)A] / sum;</td>
<td>sumg -= w * (*freqg) * treenode[i] - &gt;x[j][0][(long)G - (long)A] / sum;</td>
</tr>
<tr>
<td>M18</td>
<td>seq.c</td>
<td>460</td>
<td>sum += (*freqg) * treenode[i] - &gt;x[j][0][(long)G - (long)A];</td>
<td>sum -= (*freqg) * treenode[i] - &gt;x[j][0][(long)G - (long)A];</td>
</tr>
<tr>
<td>M19</td>
<td>seq.c</td>
<td>465</td>
<td>sumt += w * (*freqt) * treenode[i] - &gt;x[j][0][(long)T - (long)A] / sum;</td>
<td>sumt -= w * (*freqt) * treenode[i] - &gt;x[j][0][(long)T - (long)A] / sum;</td>
</tr>
<tr>
<td>M20</td>
<td>seq.c</td>
<td>461</td>
<td>sum += (*freqt) * treenode[i] - &gt;x[j][0][(long)T - (long)A];</td>
<td>sum -= (*freqt) * treenode[i] - &gt;x[j][0][(long)T - (long)A];</td>
</tr>
<tr>
<td>M21</td>
<td>seq.c</td>
<td>468</td>
<td>sum = suma + sumc + sumg + sumt;</td>
<td>sum = suma - sumc + sumg + sumt;</td>
</tr>
</tbody>
</table>

4.2 Results and Analysis

It should be noted that Original_real and Original_random consist of 5000 real and 5000 random inputs respectively, as described in section 4.1.1. We used all these inputs as a criterion to select the mutants for our experiments as mentioned previ-
ously. These 10000 inputs will be treated as a test case pool for our experiments. Taking the execution time of the programs with test inputs into consideration, we decided 1000 test cases for each program would be sufficient for this preliminary experiment. Moreover, similar experimental studies applying MT on other problem domains also use 1000 test inputs. In our experiments, we have used 1000 original inputs (500 real and 500 random) for testing each of chosen three programs. These 500 real and 500 random inputs are selected randomly from the Original\_real and Original\_random, respectively. For each MR and each original input, we generated one follow-up test input. As a result, we have 7000 follow-up inputs for DNAPARS and DNAPENNY, and 2000 follow-up inputs for DNAML. In order to test all mutants of each program, a total of \((k \times m \times 1000)\) pairs of original and follow-up inputs were executed where the program under test has \(m\) MRs and \(k\) mutants. For DNAPARS and DNAPENNY, \(m = 7, k = 10\) and for DNAML \(m = 2, k = 5\).

### 4.2.1 DNAPARS Result:

The results of applying MT to the ten mutants of DNAPARS is summarized in Table 4.4. From the results we have found that most of the mutants are killed by MR7. We have also found that MR1 could only reveal failures in M3. Among the 70000 Input pairs, 18232 pairs (26.05%) violated the MRs and hence revealed failures. As we have used both real DNA sequences and randomly generated DNA sequences, we evaluated the effectiveness for different types of inputs separately. Among 18232 pairs, 8753 pairs (25.01%) were generated from real DNA sequences...
inputs and 9479 (27.08%) pairs were generated from randomly generated DNA sequences inputs. In summary, on average, one out of four original and follow-up input pairs can reveal a failure. This is an encouraging result on the effectiveness of metamorphic testing.

A close inspection of the results of applying MT on DNAPARS reveals two interesting observations- while M7 is killed by MR2 with random input only, on the other hand, M9 was killed by MR5 with real input only. Thus, it seems that random inputs and real inputs are somehow complementary to each other.

Table 4.4: Metamorphic testing effectiveness in terms of killing mutants for DNAPARS program

<table>
<thead>
<tr>
<th>MRs</th>
<th>Types</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>M7</th>
<th>M8</th>
<th>M9</th>
<th>M10</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1</td>
<td>Real</td>
<td>0</td>
<td>0</td>
<td>127</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>0</td>
<td>0</td>
<td>172</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MR2</td>
<td>Real</td>
<td>0</td>
<td>0</td>
<td>455</td>
<td>6</td>
<td>17</td>
<td>500</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>0</td>
<td>0</td>
<td>490</td>
<td>39</td>
<td>158</td>
<td>500</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>500</td>
</tr>
<tr>
<td>MR3</td>
<td>Real</td>
<td>0</td>
<td>475</td>
<td>478</td>
<td>43</td>
<td>263</td>
<td>496</td>
<td>122</td>
<td>0</td>
<td>0</td>
<td>496</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>0</td>
<td>500</td>
<td>481</td>
<td>65</td>
<td>225</td>
<td>500</td>
<td>131</td>
<td>0</td>
<td>0</td>
<td>500</td>
</tr>
<tr>
<td>MR4</td>
<td>Real</td>
<td>0</td>
<td>0</td>
<td>465</td>
<td>2</td>
<td>374</td>
<td>0</td>
<td>66</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>0</td>
<td>0</td>
<td>493</td>
<td>3</td>
<td>500</td>
<td>76</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MR5</td>
<td>Real</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>243</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>0</td>
<td>0</td>
<td>66</td>
<td>102</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MR6</td>
<td>Real</td>
<td>489</td>
<td>0</td>
<td>492</td>
<td>0</td>
<td>410</td>
<td>0</td>
<td>66</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>497</td>
<td>0</td>
<td>499</td>
<td>0</td>
<td>492</td>
<td>0</td>
<td>113</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MR7</td>
<td>Real</td>
<td>0</td>
<td>500</td>
<td>227</td>
<td>232</td>
<td>168</td>
<td>267</td>
<td>39</td>
<td>192</td>
<td>205</td>
<td>279</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>0</td>
<td>500</td>
<td>236</td>
<td>336</td>
<td>213</td>
<td>252</td>
<td>41</td>
<td>301</td>
<td>189</td>
<td>264</td>
</tr>
</tbody>
</table>

Total number of input pairs = 70000; Number of violations = 18232
Total number of random input pairs = 35000; Number of violations = 9479
Total number of real input pairs = 35000; Number of violations = 8753
4.2.2 **DNAPENNY** Result:

The testing results of **DNAPENNY** using MT on ten mutants are given in Table 4.5. From the results we have found that MR1 could only reveal failure in M3. Among the 70000 Input pairs, 17603 pairs (25.15%) revealed failures, which included 8273 (23.64%) pairs from real inputs and 9330 (26.66%) pairs from random inputs. Similar to the observations in applying MT on **DNAPARS**, on average, one out of four original and follow-up input pairs can reveal a failure.

Analyzing the results of testing **DNAPENNY**, we have found that all MRs perform better in revealing failure with random inputs. This observation is also similar to that observed in testing **DNAPARS**.

Table 4.5: Metamorphic testing effectiveness in terms of killing mutants for **DNAPENNY** program

<table>
<thead>
<tr>
<th>MRs</th>
<th>Types</th>
<th>M1</th>
<th>M3</th>
<th>M5</th>
<th>M7</th>
<th>M11</th>
<th>M12</th>
<th>M13</th>
<th>M14</th>
<th>M15</th>
<th>M16</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1</td>
<td>Real</td>
<td>0</td>
<td>89</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>0</td>
<td>178</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MR2</td>
<td>Real</td>
<td>0</td>
<td>450</td>
<td>17</td>
<td>500</td>
<td>0</td>
<td>0</td>
<td>500</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>0</td>
<td>469</td>
<td>131</td>
<td>500</td>
<td>32</td>
<td>0</td>
<td>500</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MR3</td>
<td>Real</td>
<td>0</td>
<td>466</td>
<td>245</td>
<td>121</td>
<td>496</td>
<td>21</td>
<td>0</td>
<td>496</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>0</td>
<td>468</td>
<td>223</td>
<td>128</td>
<td>500</td>
<td>87</td>
<td>0</td>
<td>500</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MR4</td>
<td>Real</td>
<td>0</td>
<td>465</td>
<td>347</td>
<td>66</td>
<td>0</td>
<td>79</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>0</td>
<td>493</td>
<td>500</td>
<td>76</td>
<td>0</td>
<td>74</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MR5</td>
<td>Real</td>
<td>12</td>
<td>84</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>6</td>
<td>184</td>
<td>42</td>
<td>2</td>
<td>0</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MR6</td>
<td>Real</td>
<td>490</td>
<td>477</td>
<td>388</td>
<td>64</td>
<td>0</td>
<td>67</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>488</td>
<td>488</td>
<td>479</td>
<td>108</td>
<td>0</td>
<td>87</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MR7</td>
<td>Real</td>
<td>0</td>
<td>210</td>
<td>161</td>
<td>26</td>
<td>500</td>
<td>30</td>
<td>465</td>
<td>500</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>0</td>
<td>236</td>
<td>234</td>
<td>39</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>25</td>
</tr>
</tbody>
</table>

Total number of input pairs = 70000; Number of violations = 17603
Total number of random input pairs = 35000; Number of violations = 9330
Total number of real input pairs = 35000; Number of violations = 8273
4.2.3 **DNAML Result:**

Table 4.6 shows the testing results of *DNAML* using MT on five mutants. From the results we have found that MR B kills all mutants. However, no violation was detected by MR A. This implies that though the outputs might have been incorrectly computed, their interrelationship implied by MR A still held. Among the 10000 Input pairs, 4996 pairs (49.96%) revealed failures, 2500 (50%) pairs were from real inputs and 2496 (49.92%) pairs were random inputs. In summary, on average, one out of two original and follow-up input pairs can reveal a failure.

Table 4.6: Metamorphic testing effectiveness in terms of killing mutants for *DNAML* program

<table>
<thead>
<tr>
<th>MRs</th>
<th>Types</th>
<th>M17</th>
<th>M18</th>
<th>M19</th>
<th>M20</th>
<th>M21</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR A</td>
<td>Real</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MR B</td>
<td>Real</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>496</td>
<td>500</td>
</tr>
</tbody>
</table>

Total number of input pairs = 10000; Number of violations = 5996
Total number of random input pairs = 5000; Number of violations = 2496
Total number of real input pairs = 5000; Number of violations = 2500

4.3 **Discussion**

As mentioned earlier, phylogenetic inference programs such as *DNAPARS*, *DNAPENNY* and *DNAML*, have the oracle problem. Metamorphic testing result on mutant versions of three phylogenetic inference programs has shown promising fault detection rate.

From the results of testing all the three programs with MT, we found that
different MRs detect faults in different mutants. This phenomenon suggests that, defining more MRs is helpful to detect different types of faults. While executing the programs with one MR, the faulty statement may not be executed and hence the fault may remain undetected. Employing a variety of MRs to test a program will thus be beneficial to detect different faults.

In our experiment, we have used both real DNA sequences and randomly generated DNA sequences as inputs. Furthermore, some mutants were killed by MRs with one type of input only. Based on the analysis, we recommend the developers and scientists to test phylogenetic inference programs with both types of inputs.

Our study illustrates that metamorphic testing method can be helpful in detecting faults and hence can alleviate the oracle problem in testing phylogenetic inference programs. Thus MT can enable the systematic and automated output verification of these programs.

Defining the MRs requires some background knowledge on the algorithm of the program. As scientists often possess this type of knowledge, designing MRs should be relatively straightforward to them [73]. As such it is most likely that MT will be readily applicable by the bioinformatics community.
4.4 Threats to Validity

4.4.1 Internal Validity

Threats to the internal validity include the correctness of identified MRs for the phylogenetic inference programs, correctness implementation of MRs in terms of the generation of original and follow-up inputs, and comparison among original and follow-up outputs.

To verify the correctness of the identified MRs, we invited bioinformatics researchers to review the identified MRs. To verify the correctness of the implementation of the MRs, we compared our results with the results obtained by another research student (not actively involved in this research) who independently implemented the identified MRs of this research.

Another threat to the internal validity involves the process of generation of mutants. We have used a Perl script to randomly and automatically seed a fault to the source code. After generating the mutants, we manually verified each mutant is differed from the original source by just one change.

4.4.2 External Validity

The programs used in our experiment are taken from real life applications. For example, the PHYLIP package was developed in 1983 and is widely used in the bioinformatics research field. These programs are comparatively large in terms of lines of code. We have also used three different programs that used different methods to implement phylogenetic trees which generalize our experiment.
4.4.3 Construct Validity

The main threat in construct validity is the matrices we have used for measuring the effectiveness of MT on phylogenetic inference programs. Here we have used the percentage of input pairs that violate MRs for each mutant to evaluate the effectiveness which has been used in other research papers of MT [7]. So, the threat is mitigated.
Fault localization is an important step in software debugging. To find the faulty statements, fault localization techniques require the information of executed inputs along with their execution traces. The execution trace of an input is the sequence of code executed when the program is executed with an input. Executed input can be either failure-causing or non-failure-causing. For ease of discussion, we use $eTrace(t)$ to denote the execution trace of input $t$, $FCI$ to denote the set of all failure-causing inputs and $NFCI$ to denote the set of all non-failure-causing inputs.

It is difficult to obtain the failure-causing and non-failure-causing inputs when the software under test suffers from the oracle problem. Metamorphic testing technique alleviates the oracle problem. It returns the information about the
violation and non-violation of MRs. Such information can be utilized in fault localization technique. In this chapter we demonstrate how the information of violation and non-violation of MRs can be used in a statistical fault localization technique called “Cooperative Bug Isolation” [44; 74] for phylogenetic inference programs.

5.1 Cooperative Bug Isolation (CBI)

Cooperative bug isolation (CBI) is a type of statistical fault localization technique. CBI proposes collecting the execution trace of branches, function returns and scalar assignments [44]. The program under test is instrumented to collect the execution traces.

However, there is one issue in using CBI. Collecting the execution trace for a large scale program is time-consuming and requires lot of space. Liblit et al. [44] proposed a sampling technique that can decide at run time whether to collect or not to collect the execution information of a predicate (“observed” or “not observed”). A coin flip is used to decide whether the execution information of a predicate is collected or not. The sampling is adjusted by sampling rate. If more predicates are chosen not to be observed, then the more accurate finding of failure-related predicate will be compensated. Liblit’s chose to use sampling rate of 1/100 in their study [44].

CBI consists of the following two steps. First, the source code of a program is instrumented so as to observe the predicates of a program. The program will then be executed with test cases, the execution traces will be collected. The execution
results may be failure or success. A predicate is observed if it has been sampled during execution. The information of predicate condition (whether it is true or false) is also stored in the execution trace. A predicate is observed to be true if the predicate is sampled and is evaluated to true during the execution. If a predicate is sampled and is evaluated to false, it is observed to be false. Second, analysis of the execution traces to rank those predicates that might be the cause of program failures, according to the proposed metrics.

Suppose, a test input set \( T = \{t_1, t_2, ..., t_n\} \) is used in the program. Some inputs will be failure-causing (so those belong to \( \text{FCI} \)) and some will be non-failure-causing inputs (so those belong to \( \text{NFCI} \)). Program execution trace will allow CBI to calculate the probability of predicate \( p \) observed to be true implies failure by the following formula:

\[
\text{Failure}(p) = \frac{\sum_{i=1}^{n} F_i(p)}{\sum_{i=1}^{n} S_i(p) + \sum_{i=1}^{n} F_i(p)}
\]

, where

\[
F_i(p) = \begin{cases} 
1 & \text{if } p \text{ was observed to be true at least once in } eTrace(t_i) \text{ and } t_i \in \text{FCI} \\
0 & \text{if } p \text{ was always observed to be false in } eTrace(t_i) \text{ and } t_i \in \text{FCI}
\end{cases}
\]

and

\[
S_i(p) = \begin{cases} 
1 & \text{if } p \text{ was observed to be true at least once in } eTrace(t_i) \text{ and } t_i \in \text{NFCI} \\
0 & \text{if } p \text{ was always observed to be false in } eTrace(t_i) \text{ and } t_i \in \text{NFCI}
\end{cases}
\]

A number of predicates might be observed to be true in the execution trace of failure-causing inputs but have no influence on the failure. So, another metric called “Context” is also used in CBI. \( \text{Context}(p) \) is the probability that the execution of the predicate \( p \) implies failure and is calculated by the following formula:

\[
\text{Context}(p) = \frac{\sum_{i=1}^{n} F_i(p \text{ observed})}{\sum_{i=1}^{n} S_i(p \text{ observed}) + \sum_{i=1}^{n} F_i(p \text{ observed})}
\]
where

\[ F_i(p_{\text{observed}}) = \begin{cases} 
1 & \text{if } p \text{ observed at least once in } eTrace(t_i) \text{ and } t_i \in \text{FCI} \\
0 & \text{if } p \text{ has never been observed in } eTrace(t_i) \text{ and } t_i \in \text{FCI}
\end{cases} \]

and

\[ S_i(p_{\text{observed}}) = \begin{cases} 
1 & \text{if } p \text{ observed at least once in } eTrace(t_i) \text{ and } t_i \in \text{NFCI} \\
0 & \text{if } p \text{ has never been observed in } eTrace(t_i) \text{ and } t_i \in \text{NFCI}
\end{cases} \]

These two above metrics help finding the most important metric, called “Increase” in CBI. \( \text{Increase}(p) \) is the probability that the predicate \( p \) observed to be true increases the probability of causing failure and is calculated by:

\[ \text{Increase}(p) = \text{Failure}(p) - \text{Context}(p) \]

Predicates that have \( \text{Increase}(p) \leq 0 \) are discarded from consideration because they have no predictive power according to CBI [44]. The remaining predicates are then prioritized using another metric “importance”. It gives an indication of the relationship between predicates and the program fault. The formula of \( \text{Importance}(p) \) where \( p \) is a predicate is given by:

\[ \text{Importance}(p) = \frac{2}{\text{Increase}(p) + \frac{\log(F(p))}{\log(\text{NumF})}} \]

where

\[ F(p) = \sum_{i=1}^{n} F_i(p) = \text{The number of failure-causing input in which } p \text{ is observed to be true} \]

and

\[ \text{NumF} = \text{Number of failure-causing test input} \]

Predicates are ranked according to the importance. Predicates with a higher importance score need to be examined first to help the developer find the fault.
5.2 Application of MT in CBI

As testing of phylogenetic inference program suffers from the oracle problem, determining whether inputs are failure-causing or non-failure-causing is difficult. Hence to apply CBI in phylogenetic inference programs, violation and non-violation information from input pairs in Chapter 4 are taken into account.

An input pair is called a violated input pair if an MR is violated by the pair. Otherwise the pair is called a non-violated input pair. For ease of discussion, we denote \( \text{VP} \) as the set of all violated input pairs and \( \text{NVP} \) as the set of all non-violated input pairs. Statistical fault localization uses traditional testing results to compute the probability that a predicate being true implies failure.

However, in metamorphic testing, we need to use the violation and non-violation results of an MR to do the computation. For each of these, we need the execution result of two different test inputs, namely original and follow-up test input to determine whether an MR is violated or not. The complication is that these test input pairs may cause the predicate \( p \) to evaluate differently (for example, \( p \) may be evaluated to true for the original test case and false for the follow-up test case). As a result, violation of MR may be related to \( p \) being true in either the original or the follow-up test input, or both. Hence, for each input pair (the original and the follow-up test cases), we have two execution traces (one for the original test case and the other for the follow-up test case). In order to record whether such traces lead to a failure (that is, a violation of a particular MR), we use the logical OR operator to determine the final truth value of the predicate \( p \) based on the results in individual traces because those code related
to \( p \) being true have been executed. As a result, a union execution trace, denoted as \( \text{ueTrace}(tp) \), of the predicate \( p \) is formed by applying the logical OR operator to the results of the individual execution traces.

In the following, we discuss how we can compute the failure, context, increase and importance from the union execution trace of all input pairs.

Suppose, a set of input pair \( TP = \{tp_1, tp_2, ..., tp_n\} \). Some input pairs are violated (so those belong to \( \text{VP} \)) some are non-violated (so those belong to \( \text{NVP} \)). Using the violated and non-violated input pairs along with their union execution traces, we can calculate the failure, context and increase of predicate \( p \). They are given below:

Failure\((p) = \frac{\sum_{j=1}^{n} V_j(p)}{\sum_{j=1}^{n} NV_j(p) + \sum_{j=1}^{n} V_j(p)} \), where

\[ V_j(p) = \begin{cases} 1 & p \text{ was observed to be true at least once in } \text{ueTrace}(tp_i) \text{ and } tp_i \in \text{VP} \\ 0 & p \text{ was always observed to be false in } \text{ueTrace}(tp_i) \text{ and } tp_i \in \text{VP} \end{cases} \]

and

\[ NV_j(p) = \begin{cases} 1 & p \text{ was observed to be true at least once in } \text{ueTrace}(tp_i) \text{ and } tp_i \in \text{NVP} \\ 0 & p \text{ was always observed to be false in } \text{eTrace}(tp_i) \text{ and } tp_i \in \text{NVP} \end{cases} \]

Context\((p) = \frac{\sum_{j=1}^{n} V_j(p \text{ observed})}{\sum_{j=1}^{n} NV_j(p \text{ observed}) + \sum_{j=1}^{n} V_j(p \text{ observed})} \), where

\[ V_j(p \text{ observed}) = \begin{cases} 1 & p \text{ was observed at least once in } \text{ueTrace}(tp_i) \text{ and } t_i \in \text{VP} \\ 0 & p \text{ has never been observed in } \text{ueTrace}(tp_i) \text{ and } t_i \in \text{VP} \end{cases} \]

and

\[ NV_j(p \text{ observed}) = \begin{cases} 1 & p \text{ was observed at least once in } \text{ueTrace}(tp_i) \text{ and } tp_i \in \text{NVP} \\ 0 & p \text{ has never been observed in } \text{ueTrace}(tp_i) \text{ and } tp_i \in \text{NVP} \end{cases} \]

And Increase\((p) = \text{Failure}(p) - \text{Context}(p) \)
We also discard the predicates that have $Increase(p) \leq 0$ and calculate the importance for the remaining predicates. The formula for importance is given by:

$$Importance(p) = \frac{1}{Increase(p)} + \frac{1}{\log(V(p))} \frac{1}{\log(NumV)}$$

where

$V(p) = \sum_{j=1}^{n} V_j(p) =$ Number of violated test input pair in which $p$ is observed to be true and

$NumV =$ Number of violated test input pair

### 5.3 Experimental setup

We conducted an experiment to investigate the applicability of MT in CBI. This experiment used $DNAPARS$, $DNAPENNY$ and $DNAML$ and their mutants as well as the input pairs in metamorphic testing described in Chapter 4. For the collection of the execution traces with the condition value of predicates, this experiment used 1000 test results for each MR for each mutant used in section 4.2. That is, the results of 1000 input pairs applied to each mutant for each MR are used here. We have used $(m \times 1000)$ test results (where, $m$ is the number of metamorphic relations) for each mutant to compute failure, context, increase and importance. For $DNAPARS$ and $DNAPENNY$, we defined 7 MRs ($m = 7$), so in total 7000 test input pairs are used for each mutant. For $DNAML$, we defined 2 MRs ($m = 2$), so in total 2000 test input pairs are used for each mutant. If number of MR increases, the number of input pairs will increase. We instrumented the source code to get the execution trace to monitor the predicates. We collected
the execution traces of the instrumented mutants, executed with the test input pairs. We focus on the branching of the conditions (for example the true and false results of an if-conditional are treated as two different branches) [44; 48] for this experiment.

Since DNAPARS and DNAPENNY are large scale programs, in this study we set the sampling rate to 1/50. On average, we expected to have 140 (= 7000/50) observations for each predicate in each mutant. As we have only 2000 test input pairs for DNAML, we need to use a higher sampling rate than 1/50. We have used the sampling rate to 1/15 for DNAML, hoping that we have a similar number of observations for comparison. On average, we expected to have approx. 133.3 (= 2000/15) observations for each predicate in each mutant. Execution traces of inputs are stored in database to calculate failure, context, increase and importance.

5.4 Results and Analysis

To measure how likely a predicate is associated with failure, we sort the predicates in descending order by their importance value. Based on the order, we make a ranking list of predicates for each mutant. The ranking is based on all the predicates observed in all ueTrace for the mutant and have \( \text{Increase}(p) > 0 \).

As mentioned earlier in Section 5.1, CBI [44], those predicates whose increase value is zero or less have no predictive power. In other words, they are not related to failure and can safely be discarded. In our experiment, we also discard those predicates whose increase value is zero or less.
Table 5.1: Fault Localization in *DNAPARS* program by *Importance*

<table>
<thead>
<tr>
<th>Mutant</th>
<th>Faulty Predicate</th>
<th>Failure</th>
<th>Context</th>
<th>Increase</th>
<th>Importance</th>
<th>Rank</th>
<th>Total number of ranked site</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3</td>
<td>if (ally[alias[i - 1] - 1] &gt;= alias[i - 1])</td>
<td>0.6928</td>
<td>0.68942</td>
<td>0.00338</td>
<td>0.00672</td>
<td>70</td>
<td>153</td>
</tr>
<tr>
<td>M4</td>
<td>for (i = a; i &lt;= b; i++)</td>
<td>0.154</td>
<td>0.15388</td>
<td>0.00012</td>
<td>0.00018</td>
<td>128</td>
<td>138</td>
</tr>
</tbody>
</table>

*Note:* Other mutants are discarded as the *increase* values of the faulty predicate of those mutants are zero or less.

Table 5.1 shows the results of those mutants of *DNAPARS* programs whose faulty predicates are in the ranking list. The first column is the mutant number. The second column is the faulty predicate. The third, fourth, fifth and sixth columns represent the *failure*, *context*, *increase* and *importance* values of the predicate, respectively. The last two columns are the rank number of the faulty predicate in ranking list for that mutant and the number of predicates used in ranking list. This ranking gives us an idea how likely the predicate associates with failure. The higher the rank, the lower the chance of that predicate being faulty. So, predicate with rank 2 has a higher chance of being faulty than those with rank 100.

From the Table 5.1, we found that faulty predicates were in the ranking list.
for M3 and M4, however the others are not in the ranking list because they were discarded for “their faulty predicates” having increase value zero or less. In the results in Table 5.1, we see the ranking of the faulty predicates in the mutants are high. Based on the results, we can say, the application of MT in statistical fault localization for DNAPARS is not very promising.

Table 5.2: Fault Localization in DNAPENNY program by Importance

<table>
<thead>
<tr>
<th>Mutant</th>
<th>Faulty Predicate</th>
<th>Failure</th>
<th>Context</th>
<th>Increase</th>
<th>Importance</th>
<th>Rank</th>
<th>Total number of ranked site</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3</td>
<td>if (ally[alias[i - 1] - 1] (&gt;=) alias[i - 1]) &amp; (p-base[i] != 0)</td>
<td>0.6848</td>
<td>0.68212</td>
<td>0.00268</td>
<td>0.00539</td>
<td>21</td>
<td>34</td>
</tr>
<tr>
<td>M11</td>
<td>if (p-base[i] != 0)</td>
<td>0.4278</td>
<td>0.42771</td>
<td>0.00009</td>
<td>0.00012</td>
<td>57</td>
<td>57</td>
</tr>
<tr>
<td>M12</td>
<td>if (j &gt; i)</td>
<td>0.3376</td>
<td>0.07293</td>
<td>0.26467</td>
<td>0.42254</td>
<td>2</td>
<td>61</td>
</tr>
<tr>
<td>M16</td>
<td>if (other != *root)</td>
<td>0.0097</td>
<td>0.00918</td>
<td>0.00052</td>
<td>0.00099</td>
<td>43</td>
<td>63</td>
</tr>
</tbody>
</table>

Note: Other mutants are discarded as the increase values of the faulty predicate of those mutants are zero or less.

Since lower ranks of predicates imply higher chance of being faulty, the rankings presented in Table 5.2 indicate that the chance of the predicates being faulty is low. For example the ranking of faulty predicate of M12 is 2 among 61. We consider this rank low and it implies the chance of the predicate being faulty is high. On the other hand, the ranking of faulty predicate of M16 is 43 among 63. We consider this rank high, which implies the chance of the predicate being
faulty is low. Overall, we can say that the result of applying MT in statistical fault localization for \textit{DNAPENNY} is fairly good.

No faulty predicate has been found in their ranking for any of the mutants we have used in the experiment for \textit{DNAML} because all the faulty predicates are discarded as their \textit{increase} value is always zero or less. As a result, applying MT in statistical fault localization for DNAML is poor.

5.5 Discussion

We have integrated MT with a statistical fault localization technique to find the faulty predicate in three phylogenetic inference programs. The experiment did not give promising result. However, in the experiment of \textit{DNAPENNY}, we find the ranking of faulty predicate for M12 is 2 which is clearly a good ranking. Besides this, we have successfully shown how the information of violation and non-violation of MRs can be used in statistical fault localization for phylogenetic inference programs. This is our initial study; in future we will conduct the experiment with more test input pairs and mutants.

5.6 Threats to Validity

5.6.1 External Validity

The main threat to the external validity of this study is the number of subject programs. The experiment was carried out on three programs. We cannot claim
that the results carry over all other phylogenetic inference programs. This result may be seen as an starting point for further investigation. We still have a plan to do the experiment with more phylogenetic inference programs to empirically validate the applicability of MT on statistical fault localization for phylogenetic inference programs.

5.6.2 Internal Validity

The main threat to the internal validity is the correctness of the programs used in the experiment which include source code instrumentation, collecting execution traces, computation of failure, context and increase for each predicate. We have manually verified these programs so that they produce the desired outputs for our experiments. For example, we checked whether the programs that collect execution traces could dynamically invoke the subject programs used for this experiment and trace the execution correctly.
Chapter 6
Conclusions and Future Work

Software testing and fault localization are the most essential parts in the software development life cycle to ensure the quality. In practice, it is difficult to test and localize the fault in the software that suffers from the oracle problem. We have identified the oracle problem in phylogenetic inference programs and have applied MT to alleviate the oracle problem. We also integrate MT with statistical fault localization to investigate the applicability of MT on statistical fault localization for phylogenetic inference programs.

Fault localization use testing results to find the faulty statements in the source code. Although MT reveals the failure using a number of MRs, it does not need test oracle for testing. We integrate MT with statistical fault localization. We have found that the information of violation and non-violation of MRs can be used in statistical fault localization.

Our contributions to the testing and fault localization of phylogenetic inference programs are as follow:

- We have investigated the testing issues of phylogenetic inference programs and presented the testing result of some phylogenetic inference programs
using Metamorphic Testing.

- We found that the information of violation and non-violation of MRs can be used in a statistical fault localization technique.

A number of phylogenetic inference programs have been implemented since the last three decades. To our knowledge, unfortunately no testing tool has been developed to systematically test these software, despite their significant impact in various bioscience and medicine areas. As part of our future work, we plan to develop an automated general MT tool for testing phylogenetic inference programs.

For testing phylogenetic programs with MT we have selected three subject programs namely *DNAPARS*, *DNAPENNY* and *DNAML* and identified MRs for those programs. We have a future plan to identify some general MRs that are applicable for all phylogenetic inference programs.

There are many different statistical fault localization techniques. This study only considered CBI to evaluate the applicability of MT in statistical fault localization. In future, we will also apply MT on other statistical fault localization techniques to evaluate their applicability. Although the mutants we have used in fault localization experiment are generated randomly however each mutant only contains exactly one fault (one faulty statement) in the source code. We also have a future plan to evaluate the applicability of MT on statistical fault localization using multiple faults in the source code for phylogenetic inference programs.

In our study we have integrated MT with statistical fault localization for phylogenetic inference programs. In our future work we have a plan to apply
this integrated approach in other application domains that suffer from the oracle problem to measure the effectiveness.
References


[54] W. Chan, S. Cheung, and K. Leung, “Towards a metamorphic testing methodology for service-oriented software applications,” in *Proceedings of
the Fifth International Conference on Quality Software (QSIC’05), September 2005, pp. 470 – 476.


[63] J. Felsenstein, “Phylip,”


[72] M. R. Woodward and K. Halewood, “From weak to strong, dead or alive? an analysis of some mutation testing issues,” in *Proceedings of the 2nd Workshop*
REFERENCES

on Software Testing, Verification, and Analysis (TVA’88), July 1988, pp. 152–158.
