Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

February 2016

Dynamically Testing Graphical User Interfaces

Santo Carino
The University of Western Ontario

Supervisor
James Andrews
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

© Santo Carino 2016

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Part of the Software Engineering Commons

Recommended Citation
Carino, Santo, "Dynamically Testing Graphical User Interfaces" (2016). *Electronic Thesis and Dissertation Repository*. 3476.
https://ir.lib.uwo.ca/etd/3476

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact tadam@uwo.ca.

Abstract

Software test generation for GUIs is a hard problem. The goal of this thesis is to investigate different methods for dynamically generating tests for GUIs. We introduce the concept of an event-pair graph, which is used to represent and measure test suites, and show how it can be used to generate tests and measure GUI coverage. Before we can begin generating tests, we first want to determine which is better: a small test suite with a few long tests or a large test suite with many short tests. Therefore, we designed and conducted a study to determine which is more effective. We found that moderate to long tests perform better than short tests. We then move on to discuss seven test generation algorithms. Two are based on random selection, two are based on greedy selection, one is based on Q-Learning, and the last two are based on ant colony optimization. We conducted a study in order to compare the performance of each algorithm. We measured code coverage, GUI coverage, time to run, and faults found. The results show that the greedy algorithms performed the best. Finally, we conducted a study in order to determine if any of the GUI coverage metrics can be used to predict code coverage, and we conducted a study to determine if any of the coverage metrics can be used to predict the faults found. The results show that event pairs are good at predicting code coverage, and that predicting faults is difficult.

Keywords: dynamic GUI testing, event-pair graphs, test length effectiveness

Acknowledgements

I would like to thank my advisor, Dr. Jamie Andrews, for his mentorship over the course of my career in academia. As an advisor, Jamie has always let me explore the research areas that interest me the most. Along the way, he has challenged my ideas and my methods to ensure that they were correct and to keep me on track to succeed. I am a better researcher for his efforts and I am grateful for his guidance.

I would also like to thank my family and friends for their constant support in my efforts to earn a doctorate. They laid the foundation on which I could build my career and they have supported me through all my successes and failures. The success they have achieved in their own lives drives me to work harder and to improve myself. I am lucky to have them as support and I am grateful for everything they have given me.

Contents

Abstract **i**

Acknowlegements **ii**

List of Figures **vi**

List of Tables **viii**

1 Introduction **1**
- 1.1 Testing . . . 2
- 1.2 GUI Testing . . . 3
- 1.3 GUI Testing Architectures . . . 4
  - 1.3.1 Static Model Architecture . . . 4
  - 1.3.2 Dynamic Model Architecture . . . 5
  - 1.3.3 Static vs. Dynamic . . . 6
- 1.4 Contributions of Thesis . . . 7
  - 1.4.1 Event-pair Graphs . . . 7
  - 1.4.2 GUI Test Length Effectiveness . . . 8
  - 1.4.3 GUI Test Generation Algorithms and Analysis . . . 8
  - 1.4.4 GUI Test Suite Adequacy Criteria . . . 9
- 1.5 Thesis Organization . . . 9

2 Background and Related Work **10**
- 2.1 Testing Techniques . . . 10
  - 2.1.1 Random Testing . . . 10
  - 2.1.2 Search-based Testing . . . 13
- 2.2 Q-Learning . . . 14
- 2.3 Ant Colony Optimization . . . 15
  - 2.3.1 Ant System . . . 16
  - 2.3.2 Ant-Q . . . 17
- 2.4 GUI Test Case Generation . . . 18
  - 2.4.1 Static Testing . . . 19
  - 2.4.2 Dynamic Testing . . . 21
- 2.5 Test Case Length Effectiveness . . . 24
  - 2.5.1 Test Length for Non-GUI Testing . . . 24
  - 2.5.2 Test Length for GUI Testing . . . 25

2.6 Test Adequacy Criteria . . . 26
2.7 Summary . . . 28

3 Graphical User Interfaces **29**
3.1 GUI Structure . . . 29
3.2 GUI State . . . 33
3.2.1 Modal Windows . . . 33
3.3 Event-flow Graphs . . . 34
3.4 Event-pair Graphs . . . 36
3.4.1 Constructing XEPGs . . . 38
3.5 Summary . . . 41

4 Test Length Effectiveness for GUIs **42**
4.1 Motivation . . . 42
4.2 Test Case Generation . . . 43
4.3 Test Length Selection . . . 43
4.4 Experimentation and Analysis . . . 44
4.4.1 Code Coverage . . . 44
4.4.2 Uncaught Exceptions . . . 47
4.4.3 Running Time . . . 47
4.5 Summary . . . 51

5 Dynamic GUI Test Generators **52**
5.1 System Architecture . . . 52
5.2 Random Test Generator . . . 58
5.2.1 Random Selection . . . 58
5.2.2 Random Minimum Selection . . . 59
5.3 Greedy Test Generator . . . 59
5.3.1 Event-pair Greedy Selection . . . 61
5.3.2 Distance Reduction Greedy Selection . . . 63
5.4 Q-Learning Test Generator . . . 68
5.4.1 Calculating State Change . . . 68
5.4.2 Parameter Tuning . . . 71
5.5 Ant Colony Optimization Test Generators . . . 73
5.5.1 Event Selection . . . 74
5.5.2 Pheromone Update . . . 75
5.5.3 Parameter Tuning . . . 76
5.6 Summary . . . 82

6 Comparing Test Generators **84**
6.1 Experiment Setup . . . 84
6.2 Code Coverage . . . 86
6.2.1 Statement Coverage . . . 86
6.2.2 Branch Coverage . . . 87
6.3 Event Metrics . . . 91

6.3.1 Event Coverage . . . 91
6.3.2 Event-pair Coverage . . . 93
6.3.3 Event-1-pair Coverage . . . 93
6.4 Error Metrics . . . 95
6.5 Cost . . . 97
6.6 Fitness Function Evaluation . . . 100
6.7 Lessons Learned . . . 102
6.8 Summary . . . 103

7 Test Adequacy Criteria **104**
7.1 Results Analysis . . . 104
7.1.1 Coverage Analysis . . . 104
7.1.2 Exception Analysis . . . 106
7.2 Summary . . . 108

8 Conclusion **110**
8.1 Summary of Contributions . . . 110
8.2 Future Work . . . 112

Bibliography **113**

Curriculum Vitae **118**

List of Figures

1.1 Address Book Application . . . 3
1.2 Static Architecture. . . . 5
1.3 Dynamic Architecture. . . . 6

3.1 Address Book Hierarchy . . . 32
3.2 Modal Window Example . . . 34
3.3 Non-Modal Window Example . . . 35
3.4 Address Book EFG . . . 36
3.5 Address Book XEPG . . . 37
3.6 Updated Address Book XEPG . . . 41

4.1 Statement Coverage . . . 46
4.2 Branch Coverage . . . 46
4.3 Average Errors Found . . . 48
4.4 Run Time . . . 50
4.5 Cost Accumulation . . . 50

5.1 System Architecture . . . 53
5.2 Random Minimum Example with Two Tests . . . 61
5.3 Initial State with Uncovered Edges . . . 63
5.4 Resulting EPG After Test 1: E1, E2, E5 . . . 64
5.5 Test Two's Path . . . 64
5.6 Distance Example: Current EPG . . . 67
5.7 Distance Example: Test Case with Two Options . . . 67

6.1 Statement Coverage . . . 87
6.2 ArgoUML Cov. Increase . . . 88
6.3 Buddi Cov. Increase . . . 88
6.4 Gantt Project Cov. Increase . . . 88
6.5 TerpS. Cov. Increase . . . 88
6.6 TerpWord Cov. Increase . . . 88
6.7 TimeSlot Cov. Increase . . . 88
6.8 Branch Coverage . . . 90
6.9 Event Coverage . . . 92
6.10 Event-pair Coverage . . . 94
6.11 Event-1-pair Coverage . . . 96
6.12 Unique Uncaught Exceptions . . . 98

6.13 Cost 99
6.14 ArgoUML Scatter 101
6.15 Buddi Scatter 101
6.16 Gantt Project Scatter 101
6.17 TerpSpreadsheet Scatter 101
6.18 TerpWord Scatter 101
6.19 TimeSlot Scatter 101

7.1 Statement vs. Events 107
7.2 Branch vs. Events 107
7.3 Statement vs. EPs 107
7.4 Branch vs. EPs 107
7.5 Statement vs. E1Ps 107
7.6 Branch vs. E1Ps 107
7.7 Exceptions vs. Statements 109
7.8 Exceptions vs. Branches 109
7.9 Exceptions vs. Events 109
7.10 Exceptions vs. EPs 109
7.11 Exceptions vs. E1Ps 109

List of Tables

3.1 Java Swing Containers 31
3.2 Java Swing Widgets 31
3.3 Test Case Event Pairs 38

4.1 Test Suite Size and Length Selection 43
4.2 Application Summary 44
4.3 Coverage Analysis 45
4.4 Unique Uncaught Exceptions - Total(Average) 47
4.5 Unique Uncaught Exceptions 48
4.6 Time to Run (Hours) 49

5.1 Component Filters 55
5.2 Distance Reduction Example 66
5.3 Q-Learning Parameter Tuning - Largest mean in bold 73
5.4 Ant System Parameter Tuning - Largest mean in bold 81
5.5 AntQ Parameter Tuning - Largest mean in bold 82

6.1 Application Summary 85
6.2 Test Suite Details 86
6.3 Statement Coverage 89
6.4 Branch Coverage 89
6.5 Event Coverage 91
6.6 Event-pair Coverage 93
6.7 Event-1-pair Coverage 95
6.8 Unique Uncaught Exceptions - Average (Total) 97
6.9 Cost (Hours) 98
6.10 Fitness Function Evaluation 100

Chapter 1

Introduction

Software testing is an integral part of the software development life-cycle. It encompasses many different methodologies and is applicable at multiple points in the life-cycle. Testing can even be used as a development methodology, which is known as test-driven development [11]. Software testing can be broken down into its many types, such as unit testing, integration testing, system testing, and regression testing. Each type of testing contains its own methods and best practices as each type of testing has different goals. For example, unit testing's goal is to ensure that a unit of code works as intended, can handle proper and improper inputs, and meets the specified requirements, whereas system testing looks at how the system works as a whole, with all units working together.

Though each type of testing differs in terms of their scope, the overall goal is the same: to find errors in the system and ensure proper functionality. Myers defines testing as follows: *"Testing is the process of executing a program with the intent of finding errors"* [51]. He goes on to state: *"An unsuccessful test case is one that causes a program to produce the correct results without finding any errors."*

The importance of proper testing cannot be overstated. A 2002 report determined that the cost of inadequate software testing infastructure cost developers and users $59.5 billion [30]. In extreme and rare circumstances, the failure of proper testing procedures can lead to loss of

life [40]. Though such examples are uncommon, they support the notion that testing is a vital process.

As testing has expanded, the idea of automated test generation arose. Tools such as Java PathFinder [63], Pex [61], and DART [31] were created to automatically test applications. Furthermore, as graphical user interfaces (GUI) have become commonplace, the need for automated GUI test generation became apparent. Typical GUI testing involves manually writing test cases using libraries such as Selenium [57], which allow the test to interact with an interface. Another method of testing is a capture/replay system [37], where a tester records their interactions with a GUI and plays the recording back at a later time. Both manually written tests and capture/replay systems are still in use today. However, both methods of testing are slow, expensive, and brittle to changes in the interface. This lead to automated GUI test generation using systems such as GUITAR [52] that allow for automatic GUI test generation.

In this thesis we investigate dynamic GUI test generation algorithms. Dynamic GUI testing systems allow for real-time testing of applications through their user interface. We are interested in creating efficient algorithms that are able to thoroughly test a system in terms of achieving high levels of code coverage, as well as finding possible bugs.

1.1 Testing

Algorithm 1 Add two values and return the result

function ADD(a, b)
    $result \leftarrow a + b$
    return *result*
end function

Here we describe a typical testing method, called unit testing, in order to better understand the concept. Algorithm 1 shows an addition method that returns the sum of two values provided as input parameters. We can test this method by providing two input values and asserting the return value: `assertEquals(add(1,2), 3)`. We can add additional tests to test for differ-

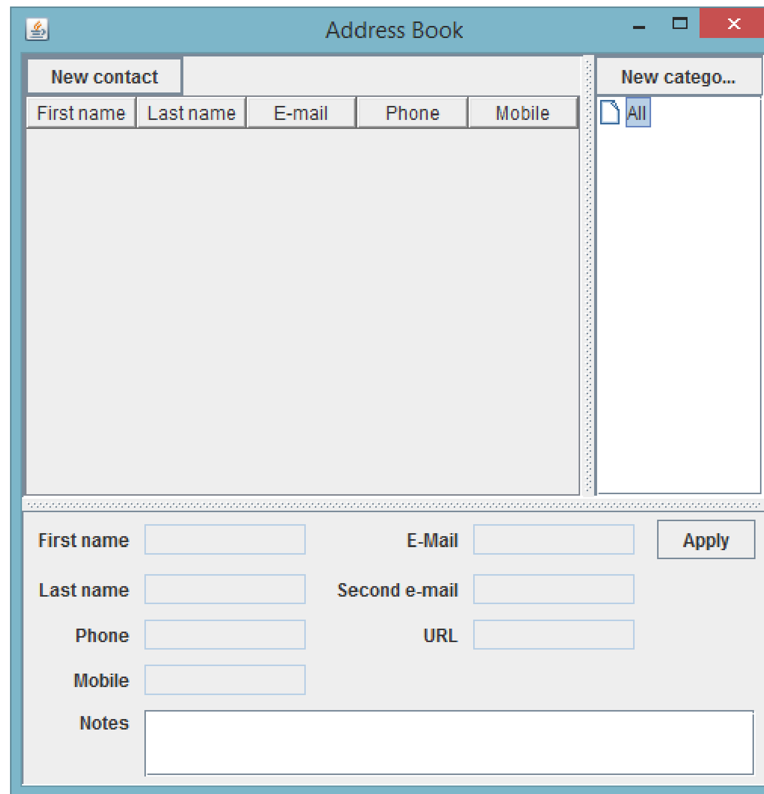

Figure 1.1: Address Book Application

ent classes of values, such as negative numbers, positive and negative numbers combined, or extreme values, such as the maximum integer value for 32 bit machines. Any failed assertion is either an indication of a bug in the application under test (AUT), or within the test. The developer or test engineer would have to investigate the cause of the failure.

1.2 GUI Testing

GUI testing differs from traditional testing in that we do not invoke methods directly; instead we interact with the application through its interface. Figure 1.1 shows a simple address book application. In order to test the application, we would execute a series of events. For example, a test case may be as follows: {New Contact, First Name, Last Name}. The test case would result in the New Contact button being clicked, and then the First Name and Last Name text fields would be filled with data. An important part of the test case is the oracle, which is how

we determine if the test passes or fails. In the case of GUI testing, there are generally two types of oracles. The first type of oracle involves checking the correctness of the GUI state [43], and the second type of oracle involves checking for crashes, hangs, and thrown exceptions. The issue with the former oracle is that it requires a *golden state* in which to compare to. Creating the golden state is a time-consuming process as it has to be verified manually. The latter oracle is simpler, and any failures can indicate a fault in the code; however, it tells us nothing about whether or not the state is correct.

1.3 GUI Testing Architectures

There are two distinct types of GUI testing architectures: static and dynamic. A static testing system attempts to discover the structure of the GUI before generating tests offline, whereas a dynamic testing system will make decisions in real time as the GUI is being explored. Both architectures have their merits, and we discuss them in this section.

1.3.1 Static Model Architecture

Figure 1.2 shows a basic architecture for a static GUI testing system. Depending on the system, the specific architecture would differ, but many systems have a similar structure. The architecture contains three main components, the *GUI Ripper, Test Generator,* and *Test Runner*. The AUT is passed to the GUI Ripper, which parses the GUI hierarchy and extracts its structure. The structure of the GUI is stored in a file in a specific format, typically XML. The GUI model is then fed to the Test Generator, which generates sequences of events based on the GUI model and stores them in files. Each test case is stored in its own file. The Test Generator uses a custom algorithm when selecting which events to place in each test case. For example, it could attempt to cover all GUI events or all paths of length two. The last phase of testing is when the tests are run against the AUT. The test cases are passed to the Test Runner, which attempts to locate the event specified in the test file and execute it on the AUT.

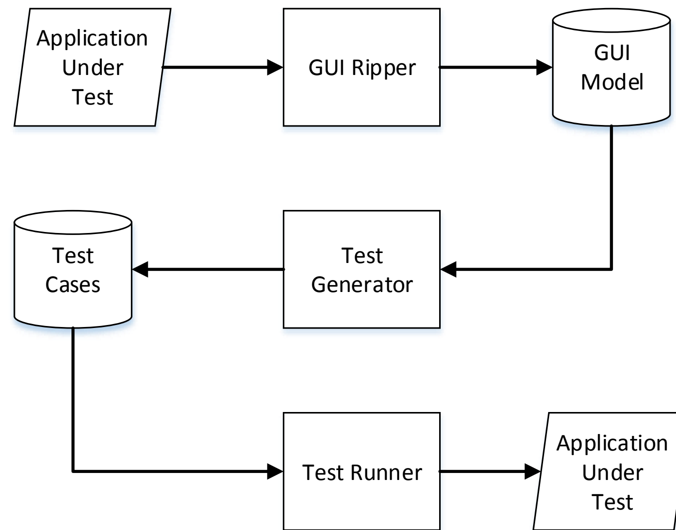

Figure 1.2: Static Architecture.

1.3.2 Dynamic Model Architecture

A typical dynamic model architecture is shown in Figure 1.3. Like the static model architecture, the dynamic architecture contains three main components: *GUI Ripper, Event Selector*, and *Event Executor*. The GUI Ripper, though it shares the same name as the static version, works differently. In the static model, the GUI Ripper parses the entire hierarchy, whereas in the dynamic model, the Ripper only parses the available widgets. We define an available widget as one that is accessible to the user in the AUT's current state. The dynamic system executes events in real time, so only those events which can be immediately executed are of interest. The GUI Ripper extracts the available events and passes them to the Event Selector. The Event Selector is the core of the system as it contains a specific event selection algorithm, such as random selection or greedy selection. Once an event has been selected for execution, it is passed to the Event Executor. The Event Executor determines the type of event being run, such as a button click or text input, and runs the event on the AUT. The AUT accepts the event, reacts, and the new GUI state is ripped by the GUI Ripper. This continues until some criterion is met, such as test length.

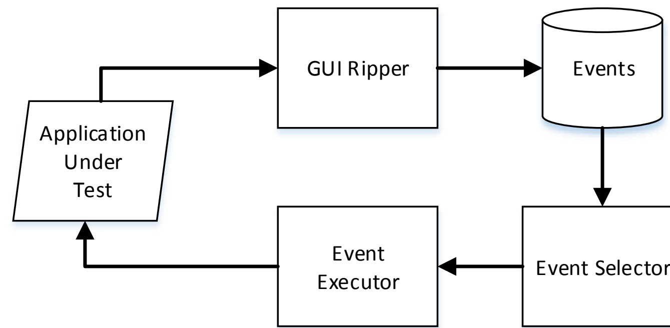

Figure 1.3: Dynamic Architecture.

1.3.3 Static vs. Dynamic

Both static and dynamic architectures have their merits. Static systems allow for the GUI to be ripped and explored offline. It is possible that static analysis can be performed on the structure, or it can even be modified by hand to ensure correctness. Furthermore, test generation can aim to cover specific targets, such as full event coverage or n-way testing, where n is some path length. The main fault with the static architecture is the existence of infeasible test cases [68]. That is, a test case can contain a sequence of events that is not possible to run on the AUT, as some widget to be interacted with is not accessible. For example, we can imagine an application with a save mechanism such that the user must click `File` then `Save`; however, the application only allows the `Save` button to be clicked when a change has been made. The GUI model would show a connection, $File \rightarrow Save$, and since the requirement that a change must be made cannot be represented as the model encompasses all possible interactions, it is possible that a generated test will contain the transition from `File` to `Save` without meeting the condition. This will cause the test to fail prematurely. In Memon's dissertation in which he describes GUITAR [44], he reports that 19.3% of generated tests are infeasible. Huang et al. [35] created a method of repairing broken tests using a genetic algorithm; however, the process can take days to run. Another issue with static testing is that the GUI ripper may not fully discover all widgets, such as those windows that require a password.

The benefit of dynamic testing is that all tests are feasible, as only those widgets which are available are selected for execution. Furthermore, as the GUI model is being discovered, its representation is updated and stored as it would be in the static architecture. Of course, the model may not be complete as there is no guarantee that the test algorithm will find all available windows and tabs. Since dynamic testing is shown to be powerful at testing GUIs, and since it does not suffer from infeasible test cases, we chose to research and implement a dynamic testing system.

1.4 Contributions of Thesis

The goal of this thesis is to investigate dynamic GUI testing algorithms and their effectiveness. In this section we discuss the specific contributions made.

1.4.1 Event-pair Graphs

We introduce and describe event-pair graphs (EPG), which are influenced by event-flow graphs (EFGs) [45]. EFGs represent an AUT's GUI event model and are generated by GUITAR. They are an example of the GUI Model of Figure 1.2. EPGs are used to represent event relationships for events that have been executed within the same test case. They can be used as both an input to a test generator and a method of evaluation. A test generator may use an EPG to determine which pairs of events have not been executed together and then test those events. Once the testing is complete, the EPG can be used to determine how many pairs of events were actually tested. We investigate whether the use of EPGs is an effective form of test generation and whether or not it is a good measure of test suite quality. Our research questions are as follows.

- Are EPGs an effective form of test generation when compared to random selection and other forms of testing?
- Are event pairs an effective measure of test suite quality compared to traditional measures

such as code coverage?

1.4.2 GUI Test Length Effectiveness

Before we can begin running our algorithms to generate GUI test cases, we first need to know how long to make our test cases, as the length of a test may impact its effectiveness. We investigate the effect test length has on GUI test suite performance and show that moderate to long test cases perform better than short test cases in both their ability to cover code and find faults. Since GUI testing is an expensive process, determining a good test length is vital to the process as it can save hundreds of hours of work. Furthermore, a specific test length may have the ability to find more faults. Our research question is as follows.

- Is it better for a test suite to consist of many short tests or a few long tests?

1.4.3 GUI Test Generation Algorithms and Analysis

The heart of this thesis is the investigation of various types of GUI test generation algorithms. We design and analyze multiple algorithms including random selection, greedy selection, Q-Learning, and ant colony optimization. These algorithms were chosen as they represent a wide array of algorithm types. Random and greedy selection are easy to implement and are effective at testing GUIs; Q-Learning is a behavorial reinforcement learning techinque that has been shown to be effective when applied to GUIs; and ant colony optimization is a search-based technique that we applied to the GUI domain as it has been shown to be good at finding paths through a graph.

We first discuss each algorithm in detail and then we compare their effectiveness. We look at how much code coverage is achieved, the number of faults found, and the number of covered events. Though multiple GUI test generation systems have been created, few have been tested against one another. It is often the case that a dynamic GUI test generator is compared to a static test generator or is compared to a non-GUI test generator. We want to know if one type

of algorithm is superior to the others. Our research questions are as follows.

- Are we able to generate tests that perform better than random selection?
- Are any of the techniques described better than the rest?

1.4.4 GUI Test Suite Adequacy Criteria

We investigate what makes a GUI test suite good. In non-GUI testing, a tester may want to exercise all conditions, exercise all definition/use pairs, or simply cover all statements. With GUI testing, we want to know which coverage goals lead to a high number of faults found. For instance, we may want to execute each GUI event at least once, or we may want to cover all event pairs. We study the effects that each type of coverage metric has in relation to faults found and code covered. Our research questions are as follows.

- Is there at least one factor that can be used to predict the quality of a test suite in terms of the number of faults found, such as event coverage or statement coverage?
- Is there a relationship between GUI coverage and statement and branch coverage?

1.5 Thesis Organization

Chapter 2 looks at the background and related work in regards to both testing and GUI testing. Chapter 3 looks at the structure of GUIs, and event-flow graphs and event-pair graphs. In Chapter 4 we present our study on effectiveness of test length on GUI test suite performance. Chapter 5 describes and analyses multiple GUI test generators. We look at generators in the form of random selection, greedy selection, Q-Learning, and ant colony optimization. In Chapter 6 we compare and contrast the results of the different test generators in order to determine if one is superior to the rest. In Chapter 7 we investigate test adequacy criteria. Finally, we conclude in Chapter 8.

Chapter 2

Background and Related Work

In this chapter we will look at a variety of test generation techniques for both GUI and non-GUI systems. We will also look at Q-Learning, which is a behavioral reinforcement learning technique, as well as ant colony optimization. Finally, we look at what makes a test suite adequate.

2.1 Testing Techniques

In this section we look at two main testing techniques that have inspired research in GUI testing: random testing and search-based testing. There exist a variety of testing techniques, such as symbolic execution [38] [67] [20] and concolic testing [58] [2]; however, we focus on the methods studied in this thesis.

2.1.1 Random Testing

A variety of testing systems that employ random testing have been created. Claessen and Hughes [19] created QuickCheck to test properties of Haskell programs using random inputs. Godefroid et al. [31] created DART, which combines random testing and dynamic analysis to explore program paths. Ciupa et al. [18] designed and created ARTOO, which uses adaptive

random testing, and applied it to object oriented software.

Hamlet [33] argues for the importance of random testing when specific requirements are met. He states that random testing is often dismissed and that random testing of an application describes testing that is done in haste, which is the opposite of systematic testing. Systematic methods often attempt to meet some minimum standard, such as 80% path coverage, which he argues does not have a scientific basis, but rather it is based on the intuition that 80% coverage of something is better than nothing. He believes that if random testing was the standard in which systematic approaches were tested against, it could help practitioners write better standards or improve the systematic methods. He ends by stating that the application of random testing requires three criteria: the test is at the system level, there exists an operational profile (a quantative representation of how a system will be used) and the ability to generate representative pseudorandom values, and there exists an effective oracle, which is a method of determining whether a test passes or fails.

Hamlet's three requirements can be applied to random testing for GUIs. GUI testing is a form of system testing so it meets the first requirement. The second requirement states that an operational profile is required. For GUI testing, an operational profile would be the measure of how often each event is executed, and the distribution of any input values. This information could be obtained by creating usage profiles for multiple users. The third requirement can also apply to GUI testing. A common oracle is to check for hangs, crashes and thrown exceptions. Of course the main issue with this type of oracle is that it ignores the semantics of the system. However, an oracle dealing with expected outputs could be manually written or described if necessary. Automatically generating such an oracle is not yet possible.

Duran and Ntafos [27] evaluated random testing in order to determine the method's effectiveness. For their first set of experiments, they ran random tests on various programs such as SIN, SORT and BINARY SEARCH, and found that random testing performed well in finding bugs. They ran more experiments on five applications, comparing the results to branch testing and required pairs testing, and measured the code coverage. They found that in the cases of

the triangle classification programs (determining, based on the three length values, whether the values form a triangle), random testing performed worse. This is due to the fact that the random algorithm failed to generate triangles with two or three equal sides. In the four other programs, random testing performed better than branch testing and better than the required pairs testing for one program. The authors concluded by stating that random testing has been shown to be a cost effective method of testing and that it is able to both find errors and achieve good levels of code coverage. However, they state that random testing should be paired with tests for special cases.

Pacheco et al. [53] developed Randoop, which is a feedback-directed random test generator. Randoop is often used as a baseline for comparison by other researchers. It works by incrementally building test cases based on previously generated tests. When a test is generated, it is executed and compared against a set of filters and contracts. These determine if the test is illegal, redundant, contract-violating, or useful for future tests. The system uses contracts to determine if a test passes. The contracts used detect whatever exceptions are thrown, and compare an object to itself using the *equals* method, the *hashcode* method, and the *toString* method. The authors compared Randoop to systematic and undirected random testing on four non-trivial applications and found that Randoop outperformed both techniques in terms of coverage and error detection.

Arcuri et al. [6] examined the current state of random testing research, and proved theorems in regards to such things as the number of random tests needed to be run in order to achieve the desired target results. The theorems put forth by the authors, and supported by empirical evaluations, determine the minimum number of tests required to meet target goals, determine how random testing can scale better than some types of partition testing, and finally, determine predictability of random testing in relation to the distribution of the testing targets. The authors state that in the case of black-box testing, when an automated oracle is available, random testing is a viable option. They conclude by stating that the results in the current literature are inconsistent and that the “jury is still out” on whether random testing is a good option.

The evidence suggets that random testing is effective in terms of finding bugs and achieving high levels of code coverage. This gives us confidence that, when applied to GUI testing, random testing may be an effective form of testing.

2.1.2 Search-based Testing

Search-based software testing is a subset of search-based software engineering. The idea is to use a search algorithm, such as a genetic algorithm [48] or hill climbing [56], in order to solve an engineering problem. Harman and Jones [34] argued for the effectiveness of these techniques and discussed the ingredients for reformulation and evaluation criteria for search-based software engineering. Reformulating software engineering as search-based software engineering requires three things: a representation of the problem which is open to symbolic manipulation, a fitness function, and a set of manipulation operators. The authors discussed how best to evaluate search-based methods. They suggested creating a base line comparison by using random search and they state, “to achieve a measure of base line acceptability, a metaheuristic technique must out-perform a purely random search.” Furthermore, they state that any algorithm worthy of consideration needs to outperform random search.

An example of such a system is EvoSuite [28], which is a test generation technique that takes an evolutionary approach to generating whole test suites. The system generates whole test suites as candidate solutions. The test suites are evolved through crossover and mutation operators. Crossover occurs by exchanging test cases between test suites. Mutation can occur at the test suite level or at the test case level. Mutation at the test suite level involves adding new tests, whereas mutation at the test case level involves adding, deleting, or changing statements. The fitness function used to evaluate a test suite is the overall branch distance. The authors used mutation testing to evaluate their generated test suites. A mutant that goes undiscovered indicates a deficiency with the solution.

Arcuri and Yao [7] studied the application of search-based techniques on object-oriented containers. They developed testing techniques that use random search, hill climbing, simulated

annealing [39], a genetic algorithm, and a memetic algorithm [50]. The fitness function of their algorithms is defined as branch distance. They compared their algorithms on seven container classes, such as Stack and Vector. They found that the memetic algorithm performed best in all situations except for the vector class.

Search-based software testing has shown to be a viable method for testing; however, it still requires a lot more research. In terms of GUI testing, we can apply search-based methods as the GUI, represented as a graph, can be explored using various search techniques.

2.2 Q-Learning

Q-Learning [64] [1] is a form of reinforcement learning for model-free systems. Agents within a Q-Learning environment are able to find optimal paths in a Markov decision process [55] (a framework for modeling decision making with partly stochastic outcomes) by determining the reward for actions taken in a system, which does not require a model of the system beforehand. The process works by having agents execute each action in each state and evaluate those actions based on the resulting reward or penalty. By attempting each action in each state, the agent learns which actions are best. The result of taking an action a in state s is calculated as follows:

$$Q(s,a) = Q(s,a) + \alpha[reward(s,a) + \gamma \max_{a'} Q(\delta(s,a),a') - Q(s,a)] \tag{2.1}$$

where α is the learning rate, γ is the discount factor, $reward(s,a)$ is the reward of taking action a in state s, $\delta(s,a)$ is the state reached by executing the action a from state s, and $Q(\delta(s,a),a')$ indicates the Q-value associated with the action a' when executed from the state $\delta(s,a)$. The learning rate determines how much influence the newly acquired information will have on the Q-value. The rate is set in the range [0,1]. A value of one will only consider the new information, whereas a value closer to zero will favour the current information. The discount factor determines the importance of the current reward versus the new reward. A value close to one will favour new rewards, whereas as a value close to zero favours the current reward. The

reward value is the fitness of taking an action a is state s. The reward values are determined by the engineer beforehand.

Q-Learning is used by Mariani et al. [41] in their system, AutoBlackTest. The reward function is the amount of change that occurs in the GUI state. The more change that occurs, the higher the reward. Q-Learning is applicable to dynamic GUI testing, as the model of the GUI is unknown until it is explored. Furthermore, the actions are generally deterministic and can be represented as a Markovian decision process. We implement our own version of this system as one of our dynamic testing algorithms.

2.3 Ant Colony Optimization

Ant colony optimization (ACO) [26] is a technique for finding good paths in a graph. Most notably, it has been applied to the travelling salesman problem [25]. Since GUIs are represented as graphs, and since we want to find good paths through the GUI in order to test it, it follows that ACO might be a good technique for GUI testing.

ACO is based on the observation of real ants [21]. Ants, searching for a food source, explore randomly while leaving a trail of pheromones behind. Over time, as more ants discover and follow the same path, more pheromone is deposited on the path causing the colony to converge towards a single path to the food source. Furthermore, the pheromones evaporate over time and so the less travelled paths become less desirable. This idea was adapted to solving existing search problems, such as the travelling salesman problem, as it was recognized that the ants will often choose the more efficient paths.

The basic algorithm works by releasing ants at different nodes in the graph. Each ant traverses the graph, visiting each node once, and returning to its initial node. Once the ants have completed their tours, the edges of the graph are updated based on the quality of each ant's tour. We look at and implement two ant systems in our dynamic testing environment, the original Ant System and Ant-Q.

2.3.1 Ant System

The Ant System discussed here is the original proposed system and all equations are taken from [22]. Ant systems work by releasing a generation of ants to explore the graph. After each ant in a generation has completed its tour, the pheromone value of each edge is updated. The value τ_{ij} represents the pheromone on the edge going from vertex i to vertex j, and is updated as follows:

$$\tau_{ij} = (1 - \rho) \cdot \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{2.2}$$

where ρ is the evaporation rate, m is the number of ants, and $\Delta\tau_{ij}^{k}$ is the quantity of pheromone laid on edge (i,j) by ant k:

$$\Delta\tau_{ij}^{k} = \begin{cases} Q/L_k & \text{if any ant } k \text{ used edge (i,j) in its tour} \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

where Q is a constant, and L_k is the length of the tour constructed by ant k.

As ants are exploring the graph, they decide the next vertex to visit through a stochastic mechanism. When an ant k is in vertex i, the probability of going to vertex j is calculated as follows:

$$p_{ij}^{k} = \begin{cases} \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{c_{il} \in N(s^p)} \tau_{il}^{\alpha} \cdot \eta_{il}^{\beta}} & \text{if } c_{ij} \in N(s^p) \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

where s^p is the ant's current solution (initially empty and then extended every time a new component is selected), $N(s^p)$ is the set of feasible components; that is, the nodes reachable from i and still unvisited (not in s^p). The parameters α and β control the relative importance of the pheromone values versus the heuristic information η_{ij}, which is given by:

$$\eta_{ij} = \frac{1}{d_{ij}} \tag{2.5}$$

where d_{ij} is the distance betwen vertex i and j. The heuristic value is used to give an advantage to shorter edges.

Further improvements were made to ACO, such as with the MAX-MIN ant system [60], which restricts the pheromone values of an edge to a predefined range. Furthermore, future ant systems introduced the concept of only reducing the edges that were traversed by an ant, rather than reducing all edges.

2.3.2 Ant-Q

One variant of ACO is called Ant-Q [23] [24], which combines ACO with Q-Learning. As each ant traverses the graph it updates each edge it touches using Q-Learning reinforcement. After each ant in a generation has completed its tour, the edges traversed by the ants are updated using ACO. The edges are updated using the following equation:

$$AQ(r,s) = (1-\alpha)\cdot AQ(r,s) + \alpha\cdot\left(\Delta AQ(r,s) + \gamma\cdot \underset{z\in N(s^p)}{\text{Max}} AQ(s,z)\right) \tag{2.6}$$

where $AQ(r,s)$ is the Ant-Q-value. The α and γ are the learning step and the discount factor, respectively. The ΔAQ value, which represents the amount of pheromone to deposit, is zero except after each ant has completed its tour. ΔAQ is calculated as follows:

$$\Delta AQ(r,s) = \begin{cases} \frac{W}{L_{k_{ib}}} & \text{if } (r,s)\in \text{ tour done by agent } k_{ib} \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

where W is a constant, k_{ib} is the ant with the best tour for this iteration, and $L_{k_{ib}}$ is the length of the tour. This equation uses the best ant in the iteration; however, it is also possible to use the global best ant, which is the best ant from any iteration. The authors found that the iteration best was slightly better than the global best.

When an ant is in state r, it selects the next state s using the pseudo-random proportional selection rule:

$$s = \begin{cases} \max\limits_{u \in N(s^p)} \{AQ(r,u)^{\alpha} \cdot HE(r,u)^{\beta}\} & \text{if } q \leq q_0 \\ P & \text{otherwise} \end{cases} \tag{2.8}$$

where α and β provide weights to both the AQ values and the heuristic value (HE), q is a random value in the range [0,1], q_0 ($0 \leq q_0 \leq 1$) is a parameter that determines the probability of making a random choice, and P is the probability selection rule stated in equation 2.4.

The major difference between the Ant System and Ant-Q in regards to their update policy is that the Ant System takes into account the tour of all m ants when updating the edges, whereas Ant-Q only takes into account the iteration or global best ant. Furthermore, the Ant System updates *all* edges, whereas Ant-Q only updates edges that were traversed by an ant.

We implement both ant systems in our GUI test generators. However, we modify them to fit our needs. For example, we are not looking for a single best path through the graph, but rather many good paths; therefore when updating the edge values in our Ant-Q system, we use the result of all the ants and not only the iteration best. The specifics of each system will be discussed in Chapter 5.

2.4 GUI Test Case Generation

As mentioned in the introduction, there are two main methods of generating GUI test cases. The first method involves ripping a model of the GUI and generating tests. The second method involves dynamically ripping the GUI state and executing events. We look at both types of systems in this section.

2.4.1 Static Testing

Static testing has received a great deal of attention from the research community. The tool most commonly used in the reseach is GUITAR [52], which works similarily to the static model described in Chapter 1. GUITAR rips a GUI into a directed graph, called an event-flow graph (EFG), and generates tests by exploring different paths through the EFG. An EFG represents the flow of events through a GUI. The vertices of the graph represent GUI events, and the edges represent dependencies. For example, if there exists an edge $<e_1, e_2>$, then event e_2 can be executed after event e_1. GUITAR runs the generated test cases on the AUT.

Various techniques for generating tests have been investigated. Brooks and Memon [12] created usage profiles of users' interactions with a system and then developed a probabilistic model based on the profiles. The test generator uses the model to create test cases that represent interactions that a user is most likely to execute. The results showed that the generated tests were at most 22% of the size of of tests that were generated from the original profiles. Furthermore, the tests were able to find more faults per test than the tests based on the profiles alone.

Memon et al. [46] used a goal-driven approach to generate test cases for GUIs. The authors used *planning*, which is a method borrowed from artificial intelligence. The idea is to give the planner a set of operators, an initial state, and a goal state. The planner can then produce a sequence of operators that will transform the initial state to the goal state. The system requires manual intervention by the tester. The tester has to set the preconditions and effects of the hierarchical operators. Furthermore, the tester has to create the initial states and goal states. With this information the planner can generate sequences of actions to change the initial states into the goal states. The authors showed the systems viability by generating tests on Microsoft's WordPad.

Yuan et al. [68] explored the idea of incorporating event context into GUI test generation. By context they mean the combination of events, the sequence length, and the position of events within the sequence. As well, they developed new coverage criteria based on combinatorial

testing. The authors created a new event graph, called an event-interaction graph (EIG), which is based on EFGs. An EIG is similar to an EFG, except that EIGs exclude system events, such as those events that open menus. Instead, they combine the system event with the target event, thus reducing the size of the graph and gaining the ability to generate more meaningful tests. Tests are generated by taking a combinatorial approach using covering arrays to generate combinations of events. The authors also defined new coverage criteria, such as *t-cover*, which states that a test suite is *t-cover-adequate* if it covers all possible *event-t-tuples*. They tested their system on eight applications and were able to find new bugs.

Yuan and Memon [70] used GUITAR to generate test cases using GUI run-time state feedback. The system generates a seed test suite, which is generated based on a ripped EIG. When the test suite is run, they capture the event-semantic interaction (ESI) relationships, which is the effect a GUI event has on all other events. A semantic relationship occurs when the execution of an event changes the properties of some other widget. Based on these relationships, the authors proposed a new graph called the event-semantic interaction graph. Tests can then be generated by walking the graph and outputting the event sequences. They conducted two studies on eight applications and found that the test suites based on ESI relationships found more faults than their code-, event-, and event-interaction-graph counterparts.

Memon and Soffa [47] addressed the problem of regression testing for GUI testing. When a system changes its interface, many of the GUI tests will break. The authors researched the number of broken tests that occur, as well as how to repair said tests. They represent the GUI using a GUI control-flow graph (G-CFG). A G-CFG is created for both the original and updated application. The two graphs are compared in order to determine which test cases are broken. The G-CFG is then used to repair the broken test cases by replacing infeasible events with feasible ones. The authors evaluated their system by manually creating tests using a capture/replay system for both Adobe Reader and MicroSoft WordPad. The results showed that they are able to repair a high number of broken tests.

Arlt et al. [8] used static analysis to analyze an application's bytecode and generate an event

dependancy graph (EDG). The authors extracted the event handlers for each event. They used a static analyzer on the extracted bytecode and determined a write/read relationship between the events. For example, event e_1 may write to a variable called *text* and event e_2 may read and use that variable. The EDG is used to represent such relationships. Furthermore, the EDG is used in conjunction with the EFG to construct test cases. The edges in the EDG only represent a write/read relationship, and so the EFG is used to connect the events within a single test. The authors generated sequences of tests for four applications and were able to find previously unknown bugs.

We can see from the literature that static GUI testing has had a lot of focus. A variety of test generation techniques have been devised, as well as test coverage criteria. Much of this work influences the research conducted in dynamic testing, such as how to best represent the GUI in a graph.

2.4.2 Dynamic Testing

A variety of dynamic testing techniques have been researched that make use of metaheuristic search techniques such as ant colony optimization, genetic algorithms, and Q-Learning.

Bauersfeld et al. [10] developed an ant colony system for testing GUIs that uses Method Call Trees (MCT) as a fitness function. A MCT is a tree structure that represents the flow of method calls during an application's runtime. The idea is that calling a method from different points in the application can lead to better code coverage and result in more bugs being found. For instance, if a method $m1()$ is called by both methods $m2()$ and $m3()$, it is preferable to have method $m1()$ called from both methods, as the context in which a method is called affects its behaviour. The authors build MCTs by instrumenting the AUT and logging the entry and exit points of each method. This data allows them to automatically create a MCT. Furthermore, the authors were able to combine multiple MCTs from different runs into a single MCT. The system tests an application by use of ants. Each ant represents a test case and a set number of ants represents a generation. After each generation of ants has completed its run, the paths

explored by the ants are updated depending on the size of the MCT generated by each ant. Paths that generated larger MCTs would receive more pheromones and therefore would be favoured in future test cases. The ant system was compared to random selection. The results showed that the method is able to generate tests that result in large MCTs compared to random selection. However, the authors do not report on the code coverage or faults found.

Gross et al. [32] developed EXSYST as a method to automatically test GUIs. The authors conducted a study to show how non-GUI test generators find false faults. False faults are created by non-GUI test generators by placing the system in an invalid state, which would normally be disallowed by the interface. They generated tests using Randoop [53] and showed that all the faults found are false. This is due to the fact that Randoop bypasses the implicit constraints of the system imposed by the GUI. EXSYST uses a genetic algorithm based on EvoSuite [28] that attempts to generate test cases that achieve high levels of code coverage. The system uses branch distance as a fitness function. Branch distance evaluates how far a predicate is from obtaining its opposite value [4]. The system generates a set number of test suites by randomly walking through the GUI. It then uses a genetic algorithm to perform crossover and mutation at both the test suite level and the test level. Crossover happens by splitting each test suite at a random point and exchanging the test cases. Care is taken to ensure that the test suite sizes remain constant so that no test suite is larger than another. Mutation occurs by either adding test cases or mutating an existing test case. The test cases can be mutated by a delete, change, or insert operation. For each action in a test case, there is a chance it is removed. As well, there is a chance that the parameters of an operation are randomly changed. Finally, there is a random probability that a random action is inserted at any point in the test case. Mutating a test case can cause the test to break, as the sequence of actions is not guaranteed to be correct. The system creates a model of the GUI in memory and updates it as the tests are executed. The model can also be used to repair broken test cases; however, it is not guaranteed to work in all cases. If a test is executed and it is determined that the test is infeasible, the test is halted and the model is updated. The system was compared to EvoSuite, GUITAR, and Randoop. In four

out of five cases EXSYST was able to achieve higher levels of code coverage.

Mariani et al. [41] developed AutoBlackTest on top of IBM Functional Tester to test GUIs. AutoBlackTest uses Q-Learning to find good paths through a GUI. The system uses a custom fitness function that calculates the ratio of change between two GUI states and uses the result to determine good event sequences. The function works by capturing the GUI state before an event is executed, executing the event, capturing the resulting state and calculating the difference between the two states. Events that result in a large state change will be favoured in future selections. The system uses Q-Learning to assign weights, called Q-values, to edges based on the fitness function. Q-values are calculated based on two values: the reward of an action (fitness) and the best Q-value of a successor edge. Every time an event is executed, its edge is updated using both values. AutoBlackTest uses an ϵ-greedy selection process, which states that a random action is selected with probability ϵ and the action with the highest Q-value is selected with probability 1-ϵ. The authors conducted experiments to determine the best ϵ value and found that a value of 0.8 resulted in the highest code coverage–that is, random actions occur 80% of the time. The system was compared to GUITAR and was shown to cover more code and find more bugs.

One problem with many of the dynamic testing systems discussed in this section is that they are compared to non-GUI testing systems or to model-based systems. Bauersfeld et al. [10] compared their ant colony system to dynamic random selection; however, they only reported on the size of the method call trees and not on the faults found or code covered. Gross et al. [32] compared EXSYST against GUITAR, Randoop, and EvoSuite. GUITAR is a model-based system, and both Randoop and EvoSuite are non-GUI testing systems. Mariani et al. [41] compared AutoBlackTest against GUITAR as well. It would be beneficial to compare dynamic testing algorithms against one another, as well as compare them against pure random selection. Testing against random selection is important as random testing represents a base line for comparison. One goal of this thesis is to compare multiple algorithms against each other, as well as to compare them to random selection.

2.5 Test Case Length Effectiveness

Test length can have a significant impact on the effectiveness of a test suite. Research has been conducted on the effect of test length in regards to code coverage and faults found in both traditional testing and GUI testing. We conducted our own study on the effect of test length on GUI test suites, which is discussed in Chapter 4.

2.5.1 Test Length for Non-GUI Testing

Some research has been conducted on the effect of test length on test length performance for traditional software testing. Fraser and Gargantini [29] ran experiments to study the effects of length on specification-based test case generation. The authors wanted to know two things: the influence test length has on fault-finding capabilites and the influence of test length on computational cost. They ran experiments on two subject applications, the Safety Injection System and the Cruise Control model. They used the NuSMV [17] model checker to generate tests. The techniques used to generate tests are mutation testing, transition pair coverage, modified condition decision coverage, pairwise testing, and random testing. The authors ran thousands of test suites and report on some of their results. Overall they found that there are special cases that need to be considered when determining the length, but that in general having fewer longer test cases is preferable to many short test cases. Increasing the test length can reduce the overall test suite size and length and increase the fault finding capabilities.

Arcuri [5] conducted experiments on six Java containers using four test generation algorhtms: random search, hill climbing, an evolutionary algorithm, and a genetic algorithm. Each algorithm attempts to cover all possible branches in a class. His goal was to show that difficult software testing benchmarks can be made trivial by using longer test sequences and that choice of length is important for software testing. The author ran 52 000 experiments and found that in general longer tests perform better and that in some instances some of the branches are infeasible for the shorter tests. The author goes on to prove two general theorems. The first describes

conditions in which longer test sequences are generally better and the second gives conditions in which they are not; however, it does not mean they are any worse.

Andrews et al. [3] addressed the issue of test length for random testing. Their goal was to show how test length can significantly impact the effectiveness of random testing in terms of the number of failures discovered as well as the length of the failing trace. The authors performed an empirical study on five applications, as well as a case study of a JPL flight software system. They found that test length has a major influence on the effectiveness of random testing for interactive programs. Furthermore, changing the test length can have an effect on the number of failures found by an order of magnitude.

These studies indicate that test length is an important variable in software testing and that long test cases generally perform better than short test cases. This work influenced our own, which we discuss in Chapter 4.

2.5.2 Test Length for GUI Testing

Little research has been conducted for GUI test length effectiveness. Xie and Memon [65] studied the effect of increasing both test suite size and test length on fault finding effectiveness. The authors looked for faults in the GUI state by comparing the test results to a golden state that was previously created. They conducted experiments using GUITAR and generated test suites of increasing size where each test suite consisted of test cases of similar size and with the same event composition. The authors conducted an ANOVA analysis and found that there was a statistically significant difference in faults found when test suite size is increased. The authors attributed this difference to the fact that events are executed multiple times with different preceding events. The authors also investigated whether increasing the length of individual test cases has a significant effect on finding faults. They conducted an experiment using test lengths ranging from 2 to 20, where each test suite contained the same number of tests. The results show that increasing the test length did not result in a statistically significant difference in faults found. However, they did find that longer test cases can find faults that shorter test

cases cannot find.

One issue with this study is that GUITAR employs the concept of "reaching steps" as it generates test cases. A reaching step is simply an event that is required to be executed so that the AUT reaches a specific state so that the test can begin. GUITAR generates tests by selecting any path of a specified length in the EFG and then works backwards, adding the required events, until it reaches an initial event. Therefore, a test of length n can consist of m reaching steps, which results in a true length of $n + m$. So although they are comparing length 2 to length 4, etc., it is likely that the length 2 tests contain many reaching steps and thus have a longer length.

Bae et al. [9] conducted a study to compare model-based and dynamic event extraction-based GUI testing techniques. The authors created a system called GUITester that is able to test systems using both a model-based approach and a dynamic approach. GUITester can rip and create EFGs similarily to GUITAR, as well as make decisions in real time. The authors compared the two approaches on five subject applications using a variety of test lengths and test suite sizes, totaling 104 test suites. Their results showed that the dynamic system outperformed the model-based system in terms of code coverage on all AUTs. The code coverage increased when the test length increased; however, the returns were diminishing. Furthermore, the results showed that the number of nonexecutable test cases for the model-based approach grew rapidly as the test lengths were increased; in one case 77.2% of all length 100 tests were nonexecutable. This high failure rate may explain why the previous paper discussed by Xie and Memon found no improvement with longer test cases.

2.6 Test Adequacy Criteria

In this section we look at what makes a test suite adequate. It is important to test practioners to know what to optimize. Should test suites aim to cover a lot of code, cover definition/use (DU) pairs (the definitions of variables and their uses in the application), or cover all paths? We

will look at the current research conducted in this area of testing in regards to both traditional testing and GUI testing. This complements our own work as we study test adequacy criteria in this thesis.

Test adequacy criteria were studied by Hutchins et al. [36]. They conducted an empirical study on the effectiveness of both the all-edges (all paths in the application's control flow graph) and all-DUs coverage criteria. They ran experiments on 130 faulty programs based on seven moderately sized programs. They generated seven thousand tests for each program and studied the relationship between coverage and fault detection. They found that test suites that achieved 90% code coverage achieved significantly better results than other test suites of the same size. Furthermore, they saw more improvement as coverage increased from 90% to 100%. However, they found that having 100% coverage itself is not always indicative of a test suite's effectiveness.

In regards to GUI testing, Strecker and Memon [59] conducted experiments on two subject applications in order to determine which factors results in high fault coverage. The authors looked at six variables: the length of a test case, the number of events (size), length two event coverage, length three event coverage, mutant type, branch points, and statement coverage. They seeded faults into both applications using a mutant generator and generated test cases using GUITAR; they limited the length of the tests to 20 events. For the first application, called FreeMind, it was found that statement-coverage-adequate test suites were the dominant factor in detecting faults. For the second application, called CrosswordSage, three factors were found to contribute to the faults found: the mutant type, statement-coverage-adequate test suites, and length three event coverage, as well as the interaction between the mutation type and statement coverage. The results validated the use of statement coverage as measure of test adequacy.

This work inspires our own study into test adequacy. Our study differs from these as we used a dynamic test generator rather than a model-based test generator. We are interested in studying both white- and black-box criteria. For white-box criteria, we look at statement and branch coverage, and for black-box criteria, we look at event-pair coverage, event coverage,

and event-1-pair coverage.

2.7 Summary

In this chapter we looked at the current literature for both GUI and non-GUI testing practices. We looked at the effectiveness of both random and search-based software testing for non-GUI testing and discussed how it could be applied to GUI testing. Following that, we looked at the state of model-based testing, most notably the use of GUITAR, and how it has been used to test GUIs. We then looked at various dynamic testing systems where each one applies a different test generation technique ranging from evolutionary algorithms to Q-Learning. We discussed studies on test length for non-GUI testing and showed how the research indicates that longer tests are generally more effective than shorter tests. The research conducted on test length for model-based GUI testing found no notable increase in faults found as the length increased. The study conducted comparing the model-based system to the dynamic system found that increasing the length results in more code coverage. We also discussed two techniques for exploring a graph: Q-Learning and ant colony optimization, which we use as test generators later in this thesis. Finally, we discussed the current work in test adequacy criteria that showed the types of coverage criteria that have a significant impact on the number of faults found in an application.

Chapter 3

Graphical User Interfaces

In this chapter we look at the structure of graphical user interfaces in order to better understand how they can be tested. First we look at the definition of a GUI and its core components. We describe how a GUI is constructed in a hierarchy and the type of components a GUI can consist of. Following that, we look at the concept of a GUI state and the concept of modality. We then look at two ways GUIs can be represented as graphs. The first method is to represent GUIs in an event-flow graph and the second method is to represent GUIs in an event-pair graph. We discuss both graphs in detail and discuss their uses and differences.

3.1 GUI Structure

A GUI is created in a hierarchical structure. The structure consists of containers and widgets. A container is a bounding structure that surrounds widgets. Table 3.1 shows some of the important Java Swing containers. A widget is any object with which a user can interact. Table 3.2 displays a list of some of the widgets for Java Swing and their respective actions. We formally define GUIs as follows.

Definition A *graphical user interface* (GUI) consists of a set of containers and widgets, arranged in a hierarchy, which users can interact with by executing events. GUIs generate and

consume events based on a user's actions.

Definition An *event* is a combination of a user action, such as a mouse click, and a widget. Events are generated by users acting on GUI widgets.

GUIs are event-based applications. That is, they respond to actions from a user or from another system, and generate events. Events tell the system that an action has taken place. There are different types of actions, such as mouse clicks, keyboard input, and touch input. Each widget within a GUI responds to specific actions. A button can respond to a mouse click, as well as keyboard input, such as when it is in focus and the user presses *Enter*. Some widgets, such as tables, have more complex inputs. A table can have multiple rows selected or multiple columns selected; the entire table can be selected; as well, values can be input into individual cells. Events are consumed by the system by means of an event handler. An event handler is a special interface built into GUI applications that allows for events to be captured and consumed. In Java, event handlers are added to a widget and the widget can start listening for events. The following code snippet is an example of adding an event handler (called Action Listener) to a Java button.

```
myButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        doSomethingInteresting();
    }
});
```

When the button is clicked, the *actionPerformed()* method is called. Each widget in the GUI has to implement its own event handler. Furthermore, widgets can implement multiple event handlers, such as how to handle both a key press and key release.

As an example of a GUI hierarchy, Figure 3.1 shows the hierarchy for the Address Book application as depicted in Figure 1.1. At the root of all Java Swing applications is the main Frame.

Table 3.1: Java Swing Containers

Containers
Frame
Window
Panel
Tab Pane
Menubar
Popup Menu

Table 3.2: Java Swing Widgets

Widget	**Events**
Button	Click
MenuItem	Click
CheckBox	Click
RadioButton	Click
ComboBox	SelectIndex
TextComponent	SetText
Slider	SetValue
Spinner	Click
Label	Click
List	SelectRow, SelectRows, ExpandRow
Tree	SelectRow, SelectRows, ExpandRow
Table	SetCellValue, SelectCell

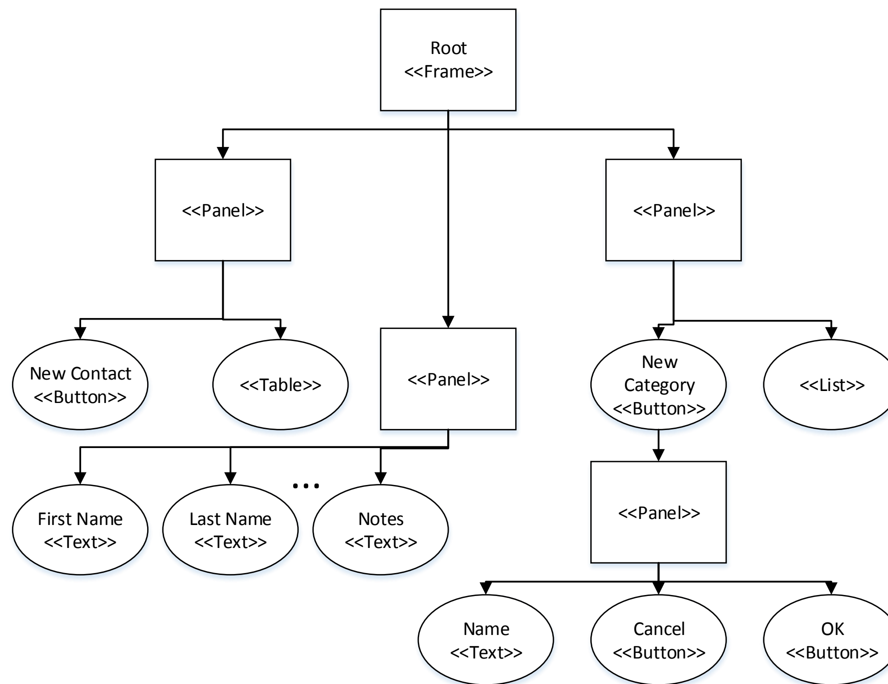

Figure 3.1: Address Book Hierarchy

Panels can be added to the frame, which themselves can contain more panels. Furthermore, widgets are stored in panels and arranged in some order. When ripping a GUI, either using a static or dynamic system, the hierarchy is explored in a depth-first or breadth-first fashion until all branches have been traversed.

In order to test a GUI system, we have to execute events on the application under test (AUT). We can execute events programmatically or we can use a robot class to emulate mouse and keyboard inputs. Executing an event programmatically involves calling provided methods, such as the *doClick*() method provided by Java Swing to click a button. Using a robot class, we can tell the machine to move the cursor to specific coordinates and press a mouse button, or we can focus on a specific text box and send keyboard signals. One drawback to using the robot class is that the tests can be fragile since they rely on specific coordinates. As well, the tests are slower as the system has to wait for the cursor to move before executing an action.

3.2 GUI State

A GUI state S is composed of a set of widgets $\{w_1, w_2, \ldots, w_n\}$, where each widget W is composed of a set of properties $\{p_1, p_2, \ldots, p_n\}$. Each property P is assigned a value V. Therefore, we define a GUI state as follows.

Definition The *state* of a GUI is the composition of all widgets W, where each widget has a set of properties P, and where each property has a fixed value V. A state S_1 is equal to a state S_2 if they both contain the same number of widgets, and the properties of each widget are equal for both states.

The concept of a GUI state is important for GUI oracles. When testing GUIs, there are two types of oracles: implicit oracles and GUI state oracles [43]. Implicit oracles only take into account hangs, crashes, and thrown exceptions, whereas state oracles look at the state of the GUI in order to determine if a bug has been found. In order for a state oracle to be implemented, a perfect (golden) state must be created first. The golden state represents the correct state of the GUI after a set number of events have been executed and it is to be used for comparison when testing. Creating the golden state is labour intensive as it has to be verified manually. When testing the system, the new state is compared to the golden state. Any differences between the two states is considered a bug and warrants an investigation.

3.2.1 Modal Windows

An important characteristic of GUIs is the concept of modal windows. A modal window is one that captures the focus of the system and does not allow users to interact with GUI widgets outside of itself. An example of a modal window is the *Save* dialog shown in Figure 3.2. A non-modal window is shown in Figure 3.3, which is a *Find/Replace* window. The modality of a window can be determined during the ripping process by checking the window's properties. When we generate tests, we have to take into account whether a window is modal. Since we often execute events programmatically, it is possible to execute events not contained within a

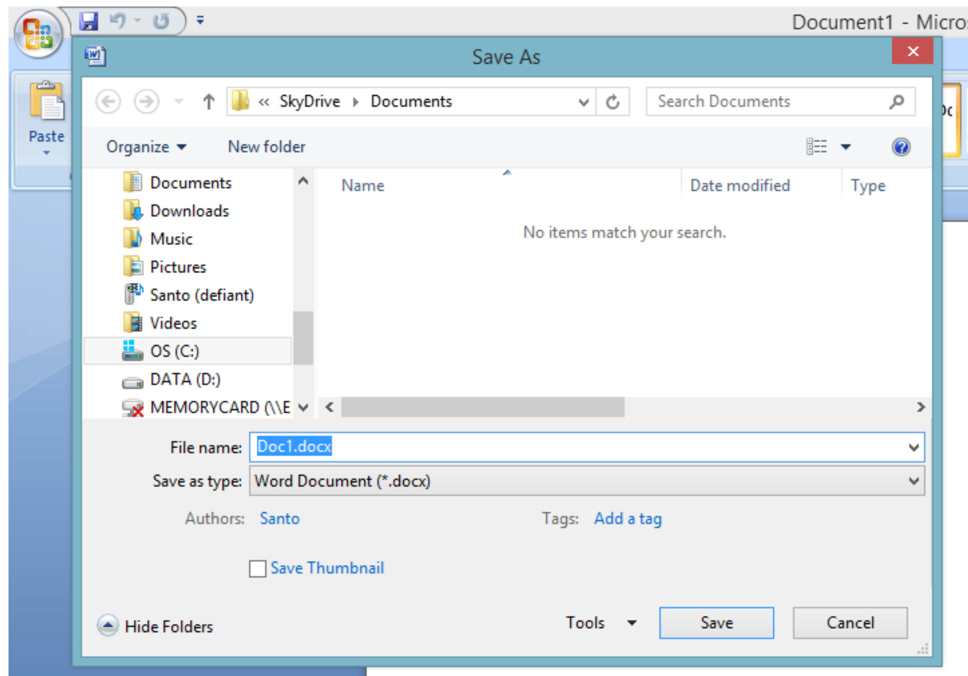

Figure 3.2: Modal Window Example

current modal window, which would break the implicit constraints enforced by the interface. Therefore, when a modal window is present, we have to ensure only its events are executed until the window is closed. Furthermore, if two or more modal windows are shown, we have to ensure the top-level window is the one being tested. An important part of GUI testing is not bypassing the implicit constraints of the system. Bypassing the constraints may lead to failures of the system; however, the failures would be spurious, since no user could execute them.

3.3 Event-flow Graphs

We define an event-flow graph and a path in an event-flow graph as follows.

Definition An *Event Flow Graph (EFG)* is a graph in which the nodes are events, and there is an edge from e to f if a user can execute f immediately after executing e.

Definition A *path* in an EFG is a sequence $(e_1, e_2, \ldots, e_n)$ of events such that event pair $\langle e_k, e_{k+1} \rangle$ corresponds to an edge in the EFG. We would say that the path has length n. For

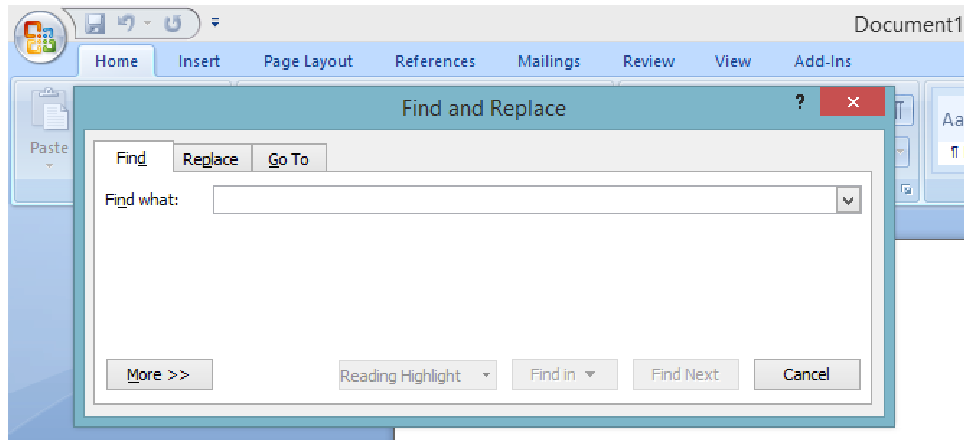

Figure 3.3: Non-Modal Window Example

consistency, n can be 0 (in the case of the empty path).

EFGs represent the flow of events in a given GUI. They are generated by GUITAR [52] and can be used to generate test cases. Figure 3.4 shows the EFG for the Address Book application. The graph does not include all events as the complete graph would be unclear. The *New Contact* and *New Category* events are blue as they represent initial events, which are events that can be executed immediately after the application has been loaded. A test case would start from one of the initial events, and then it would follow the available edges until some criterion is met, such as length. We do not use EFGs directly; instead they inspired us to create event-pair graphs.

Looking at the EFG we can see that there is no way to represent a precondition, for example, requiring that some event be executed before any future event. To illustrate how this can lead to infeasible test cases, we will give an example using the Address Book EFG in Figure 3.4. Using the EFG, we can generate a test that looks as follows: {New Category, OK, First Name}. That is, the test will click the New Category button which opens a new window. The test then clicks the OK button and finally it types a value into the First Name field. The issue with this test case is that it is not feasible. In order for data to be entered into the First Name text box, the New Contact button must be clicked. Since this precondition cannot be represented in the graph, there is no way to stop this test from being generated.

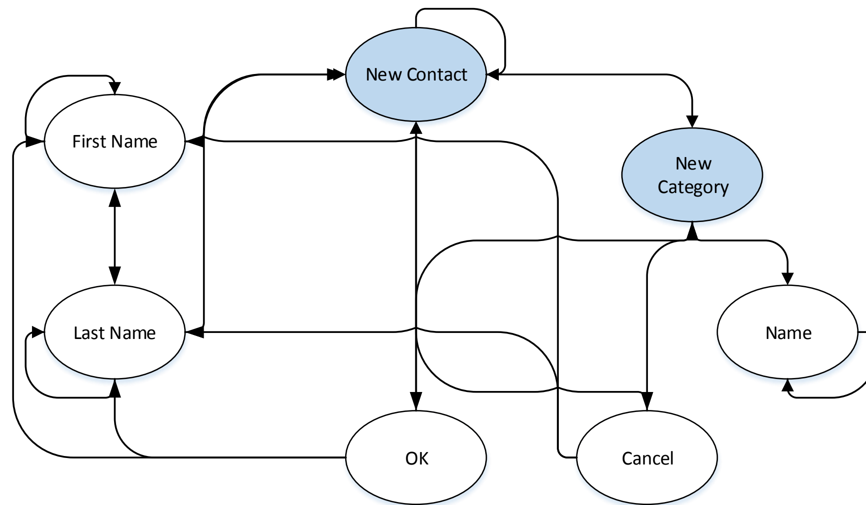

Figure 3.4: Address Book EFG

3.4 Event-pair Graphs

We now present a new representation, called event-pair graphs. We define event pairs, event-pair graphs, and extended event-pair graphs as follows.

Definition An *event pair* is a pair of events E and F such that there is a path from E to F in the EFG.

Definition An *Event-Pair Graph (EPG)* is a directed graph that contains all GUI events. There is an edge between two events e_i and e_j in the EPG if $\langle e_i, e_j \rangle$ is an event pair. EPGs are built dynamically as tests are being generated.

Definition An *Extended Event-Pair Graph (XEPG)* is an EPG with weighted edges. The weight values can be used to represent such things as distance, the number of times traversed, or any other value the algorithm requires.

We created EPGs for two purposes: to be used as inputs to test generators and as a measure of quality. An EPG can be used as an input to test generator as follows. Intitially the EPG is empty. As tests are generated, the EPG is expanded with vertices and edges. The test generator

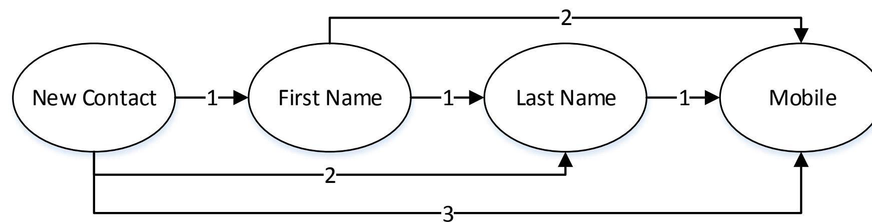

Figure 3.5: Address Book XEPG

can use this knowledge to select events that would result in more edges being added to the EPG. Our intuition is that it is better to execute a large number of unique event pairs rather than repeating the same pairs, and since the EPG represents the already covered pairs, it can be used to decide which event to select next based on the number of existing edges. Following this same intuition, we can also use EPGs as a measure of quality for a test suite. A test suite that covers more event pairs is better than a test suite that covers fewer. This is studied further in Chapter 7.

An example XEPG is shown in Figure 3.5. The figure shows the resulting XEPG for the following test case: {New Contact, First Name, Last Name, Mobile}. The XEPG shows the events executed, the order of the events, and the distance between each event. We store the distance between events because in future test cases the test generator may want to execute two events closer together, as doing so may result in a bug. For example, a test of length 30 would have events with large distance between them. It is possible that events e_1, e_2, and e_3 share common variables and methods, and that executing $\{e_1, e_3\}$ results in a bug. However, if the events are separated by some distance, and if event e_2 is executed between them, it is possible that the bug goes undiscovered. Therefore, we can attempt to find such bugs by executing events closer together.

Table 3.3 shows the event pairs covered by our example test case. The maximum number of event pairs any test case can cover is

Table 3.3: Test Case Event Pairs

Event Pairs
New Contact, First Name
New Contact, Last Name
New Contact, Mobile
First Name, Last Name
First Name, Mobile
Last Name, Mobile

$$\frac{n(n-1)}{2} \tag{3.1}$$

where n is the length of the test case. We consider all pairs as event pairs regardless of distance; however, we can use the distance values as a measure of quality of a test suite. The shorter the average distance between all nodes, the better.

The main difference between EFGs and EPGs is that EFGs represent the entire GUI and tell us how we can explore the system, whereas EPGs tell us the parts of the graph we have already explored. In an EFG, an edge between two events means that one event can be immediately executed after the other. Whereas an edge in an EPG means that the two events were executed together within the same test case. Furthermore, EFGs are generated statically before the tests are generated, whereas EPGs are generated dynamically as the tests are generated.

3.4.1 Constructing XEPGs

An XEPG is a container object that stores the list of events and the list of edges. The list of events represents the vertices of the graph and the edges represents their relationships. The class looks as follows.

```
class XEPG {

    List<Event> eventList = new ArrayList<Event>();

    List<Row> edgeList = new ArrayList<Row>();
```

```
    getEvents() {
        return eventList;
    }
    getEdges() {
        return edgeList;
    }
    addEvent(Event event) {
        eventList.add(event);
    }
    addEdge(Row row) {
        edgeList.add(row);
    }
}
```

Each event object contains the specific event's information, such as its type, position in the GUI hierarchy, and its owner. The list of rows is an adjacency matrix that represents all edges. The row object is a list of integers, where each item in the list represents an edge. If a value in the row list is 0, there is no edge between the two events; otherwise if the value is greater than 0, there is an edge and the specific value is the edge's distance. The ith row corresponds to the edges coming from the ith event.

XEPGs are stored in an XML file and read into the system before each test begins, except for the first test where the XEPG is empty. Data is added to the XEPG using an XEPG wrapper class that handles the logic of finding the events and adding or updating their edges. Algorithm 2 shows how an XEPG is constructed or updated after a test case is complete.

The *contains*, *getDistance*, *addEdge*, and *updateEdge* methods are helper methods that deal with finding the correct event and rows from the XEPG object and adding or updating the values. When updating the edge values, the current value is only overwritten if the new value is less than the current value. Since we are trying to reduce the edge values to 1, thus indicating

Algorithm 2 XEPG Construction with Distance Values

```
 1: function XEPGWrapper(XEPG epg, TestCase tc)
 2:     eventOne ← null
 3:     eventTwo ← null
 4:     for i ← 0; i < tc.length() − 1; i ← i + 1 do
 5:         eventOne ← tc.get(i)
 6:         if !epg.contains(eventOne) then
 7:             epg.addEvent(eventOne)
 8:         end if
 9:         for j ← i + 1; j < tc.length(); j ← j + 1 do
10:             eventTwo ← tc.get(j)
11:             if !epg.contains(eventTwo) then
12:                 epg.addEvent(eventTwo)
13:             end if
14:             oldDistance ← epg.getDistance(eventOne, eventTwo)
15:             newDistance ← j − i
16:             if oldDistance == 0 then                              ▷ Does not exist
17:                 epg.addEdge(eventOne, eventTwo, newDistance)
18:             else if newDistance < oldDistance then
19:                 epg.updateEdge(eventOne, eventTwo, newDistance)
20:             end if
21:         end for
22:     end for
23: end function
```

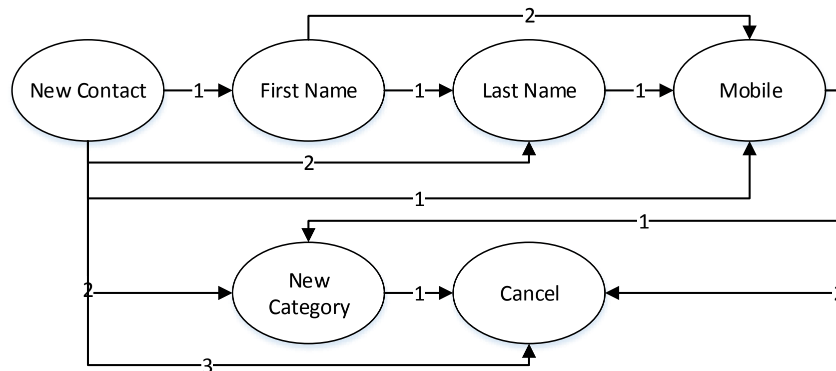

Figure 3.6: Updated Address Book XEPG

two events were executed one after the other, we discard the larger (worse) values.

Continuing with our example in Figure 3.5, if we added another test case, t_2={New Contact, Mobile, New Category, Cancel}, our XEPG is transformed into the one in Figure 3.6. The edge distance from New Contact to Mobile has been reduced to one, and two new events and their respective edges have been added to the graph.

3.5 Summary

In this chapter we defined GUIs and events. In order to better understand how GUIs are parsed and tested, we broke down the structure of GUIs and showed how they are organized in a hierarchy. We defined GUI state and discussed how it can be used as part of an oracle. As well, we looked at the concept of modal windows and discussed their effect on GUI testing. We showed how GUIs are represented in EFGs and how tests can be generated statically. Furthermore, we introduced EPGs and XEPGs and gave an example of how they can be used as both an input to a test generator as well as a method of test suite evaluation. In the future we may refer to both EPGs and XEPGs as EPGs for simplicity.

Chapter 4

Test Length Effectiveness for GUIs

4.1 Motivation

In this chapter we investigate the effect that test length has on GUI test suite performance. A GUI test case is a sequence of events; its length is the number of events in the sequence. Before we can begin testing GUIs, we need to determine whether it is favourable to have a large test suite with short test cases, or a small test suite with long test cases. It is also possible that a mix of short and long test cases is optimal, but we limit our study to a strict length for each test suite.

As mentioned in Chapter 2, Xie and Memon [65] conducted a study using GUITAR to determine if length has any effect on fault finding and found that there was no significant difference between long and short tests, but that longer tests can find some faults that shorter tests cannot. As well, Bae et al. [9] compared model-based testing to dynamic testing and used various lengths. Their work showed longer tests can perform better than shorter tests; however, greater increases in length only had a small positive effect on the code coverage. The work by Bae et al. complements our own work [16].

Table 4.1: Test Suite Size and Length Selection

Suite Size	960	480	240	120	60	30	15
Test Length	5	10	20	40	80	160	320

4.2 Test Case Generation

In order to test each application for this experiment, we used a custom dynamic test generator. The details of the system are described more in Chapter 5, and so we will briefly summarize the algorithm. The algorithm generates tests by attempting to maximize event-pair coverage. The algorithm uses a greedy selection process when choosing the next event to execute. It looks at all possible events and chooses the one that will add the most new edges to the EPG. If two or more events cover the same number of new event pairs, one is chosen at random.

4.3 Test Length Selection

Table 4.1 shows the test suite sizes and test lengths used in our experiments. The test suites ranged from having 960 tests of length 5 to having 15 tests of length 320. We chose a variety of sizes as we wanted to cover both very short and very long tests. We employed the concept of an event budget [3]. That is, each test suite contained the same number of events to run. Time is commonly used as a budget for testing; however, we wanted to investigate how well a set number of events are used depending on the test length, therefore we chose to restrict the number of events rather than the amount of time for each test suite. Each test suite contained 4800 events to ensure fairness. Looking at Table 4.1, we can see that each test suite contains 4800 events when the suite size is multiplied by the test length.

Table 4.2: Application Summary

Application	**Application Area**	**Classes**	**Lines of Code**	**Events**
FreeMind	Mind mapping	493	33443	1562
Gantt Project	Project management	689	27804	333
TerpSpreadSheet	Spreadsheet	137	5449	329
TerpWord	Word processor	208	10340	453
TimeSlot Tracker	Task organizer	487	10090	404
Total		2047	87924	3081

4.4 Experimentation and Analysis

Our experiment consisted of running tests on five subject applications. We ran each set of experiments three times and used the average for our results. Table 4.2 lists the five applications used for these experiments. All five applications appear elsewhere in the literature [14] [42] [66] [13] [41] [69] [12] and range from small to medium in size. They all represent real-world applications that are used daily, such as a word processor and a time management system.

Each test is run from the initial application state. Some GUI testing systems, such as GUITAR [52] and AutoBlackTest [41], start their tests from arbitrary states. In the case of GUITAR, it uses reaching steps to ensure the first event is available, and in the case of AutoBlackTest, it starts testing from a previously found state. Our results would not be directly applicable to such situations. The effect of test case length for systems starting from random states would require its own investigation.

We ran the experiments on a virtual machine running Linux Mint with 2 GB of RAM and an Intel Core i5, 1.7 GHz processor. All further experiments in this thesis were run using the same setup.

4.4.1 Code Coverage

The first criterion for evaluation is code coverage, which is a commonly used measure of quality for testing systems. Figures 4.1 and 4.2 show the statement and branch coverage results,

Table 4.3: Coverage Analysis

	Statement		Branch	
Lengths	**Wilcoxon**	**T-test**	**Wilcoxon**	**T-test**
5 & 10	**0.0001831**	**0.002505**	**6.104e-05**	**0.001176**
10 & 20	**0.03015**	**0.03659**	**0.01508**	**0.01264**
20 & 40	0.05536	0.1044	0.05001	0.1007
40 & 80	0.5995	0.6547	0.804	0.6402
80 & 160	0.7197	0.4046	0.7763	0.4385
160 & 320	0.4776	0.6238	0.5614	0.7366

respectively. The graphs show that there is an increase of coverage as the lengths increase. We performed a paired t-test and Wilcoxon test (p = 0.05) in order to determine if there was a statistically significant difference between any pairs of lengths. We compared each pair of lengths for all applications and found a significant difference between lengths 5 and 10, and lengths 10 and 20, but not between any other pair of lengths. Therefore we can say there is a significant increase in both statement and branch coverage up to length 20, but none afterwards. The results from the analysis can be seen in table 4.3. Each entry is a p value; entries less than 0.05 are in bold.

The coverage results for the applications are typical. The low FreeMind coverage is attributed to a set of classes that are automatically generated. Of the 33443 lines of code in FreeMind, 17200 are automatically generated. These generated classes deal with the actions associated with GUI events. Some of these actions are never covered, such as those used to add plugins to the system. Furthermore, since we do not use mouse movement inputs and since FreeMind is meant to be used by a mouse to create the mindmap, it is likely many of these actions are not utilized completely or at all. We believe that with mouse support we would see an increase in coverage.

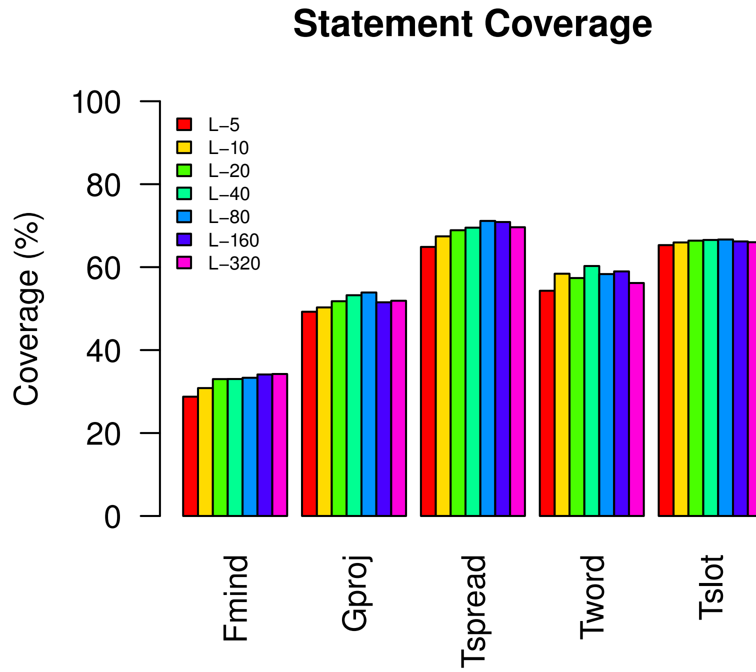

Figure 4.1: Statement Coverage

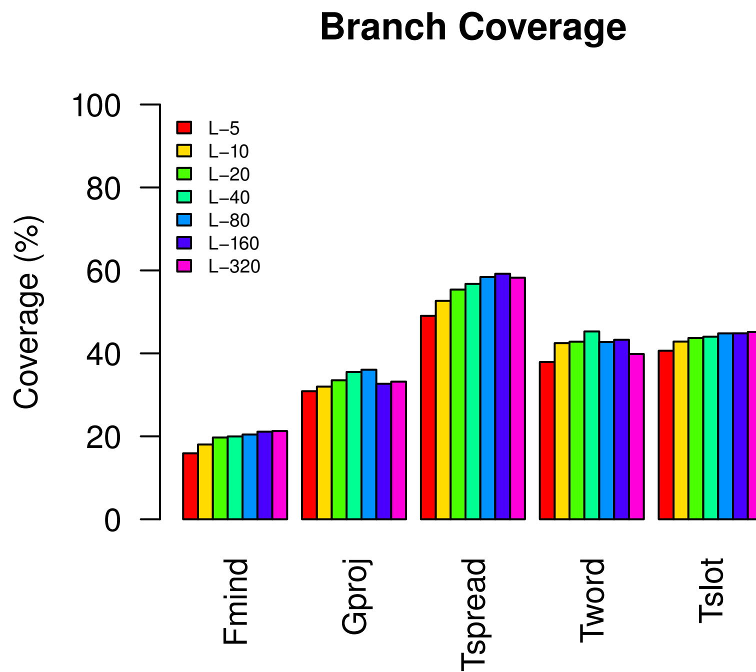

Figure 4.2: Branch Coverage

Table 4.4: Unique Uncaught Exceptions - Total(Average)

	Applications				
Length	**FMind**	**GProj**	**TSpread**	**TWord**	**TSlot**
5	1(1)	0	8(5.7)	8(5.7)	2(1.7)
10	6(3.3)	0	6(5.7)	9(7.3)	2(1.3)
20	4(3.3)	0	7(6.3)	11(7.7)	2(1.3)
40	7(4.7)	0	8(6.7)	12(9.7)	7(3)
80	10(5.3)	0	11(7.7)	10(8)	5(2.3)
160	9(6)	0	8(6)	12(9.3)	8(4.7)
320	15(9)	0	7(5)	10(8.7)	10(4.3)

4.4.2 Uncaught Exceptions

The next criterion we looked at is the number of unique uncaught exceptions found. Unique uncaught exceptions are exceptions thrown by the AUT and caught by our system; they represent a fault in the AUT. Furthermore, we only count each exception once for a test suite. An example of such an exception is the *NullPointerException*, which is thrown when an AUT attempts to access a null reference variable. Table 4.4 shows the total number of exceptions found across all three runs, as well as the average number of exceptions found. Figure 4.3 shows this information in a graph to help us better visualize the results. The graph shows that length 320 found the most errors, but it should be noted that length 320 did not find the most errors per application. Again, we ran both a paired t-test and Wilcoxon test in order to determine if there was a statistically significant difference between any length pairs. Our results show that there was a significant difference between length 20 and length 40 but not between any other pairs of lengths. The results of the analysis can be seen in Table 4.5.

4.4.3 Running Time

Our final criterion of evaluation is the cost of running each test suite. Table 4.6 and Figure 4.4 show the results. We can see that for all five applications there is large decrease in running time as the test length increases and the test suite sizes decrease. The decrease flattens out between

Table 4.5: Unique Uncaught Exceptions

Lengths	**Wilcoxon**	**T-test**
5 & 10	0.3342	0.5596
10 & 20	0.4713	0.5117
20 & 40	**0.02296**	**0.01352**
40 & 80	0.9438	0.9063
80 & 160	0.322	0.3008
160 & 320	0.6795	1

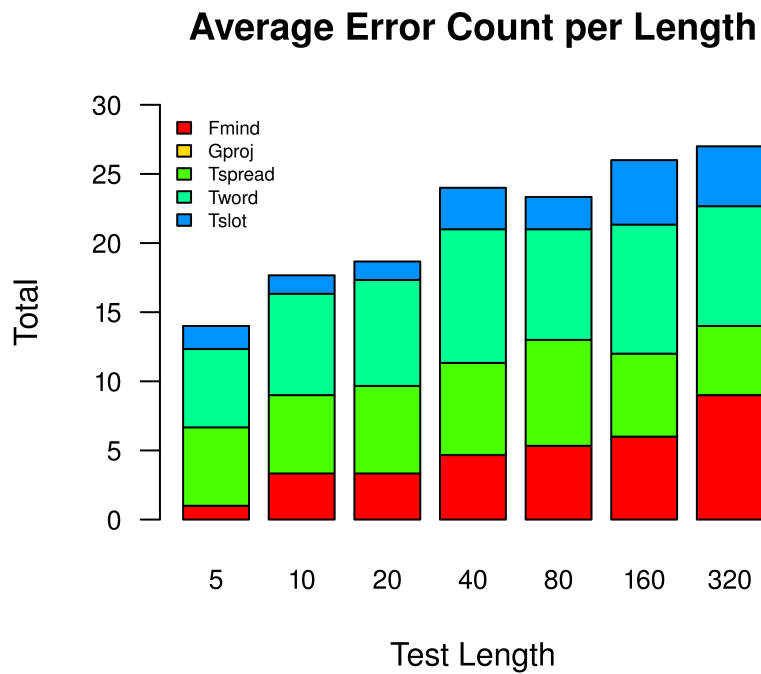

Figure 4.3: Average Errors Found

Table 4.6: Time to Run (Hours)

	Applications				
Length	**FMind**	**GProj**	**TSpread**	**TWord**	**TSlot**
5	10	9.5	7	7.1	10.1
10	7.2	6.6	5	5.3	8.1
20	5.8	5.2	4	4.4	6.6
40	5	4.6	3.4	3.9	5.9
80	4.6	4.2	3.2	3.7	5.5
160	4.4	4	3.1	3.7	5.8
320	4.5	4.2	3	3.7	5.7

length 40 and length 80.

Test case cost can be broken down into three parts: startup cost, event cost, and shutdown cost. Figure 4.5 illustrates the buildup of each of these costs. When running a GUI test, the AUT needs to be run in order for the events to be executed. Therefore, there is a startup cost as the system needs to reach its initial state before it can be tested. For larger applications, this initial waiting time can be expensive. We can represent the total cost using equation 4.1.

$$cost = n \cdot (StartCost + (EventCost \cdot Length) + ShutdownCost) \tag{4.1}$$

where n is the test suite size. The event cost is the time it takes to execute an event and wait for a reaction, such as window opening. The shutdown cost is the time taken to write the coverage information and close the program. The programs we tested had an initial wait time between 5 and 15 seconds, events were run every second, and the time to write the coverage information ranges from 1 to 3 seconds. Since some applications have a large initial waiting period, it follows that running many tests would incur a greater cost. Therefore, it is better to run a smaller number of long test cases. Furthermore, in cases where only a few events are executed, such as with length 5, the startup and shutdown costs will likely overshadow the event cost making the tests expensive to run.

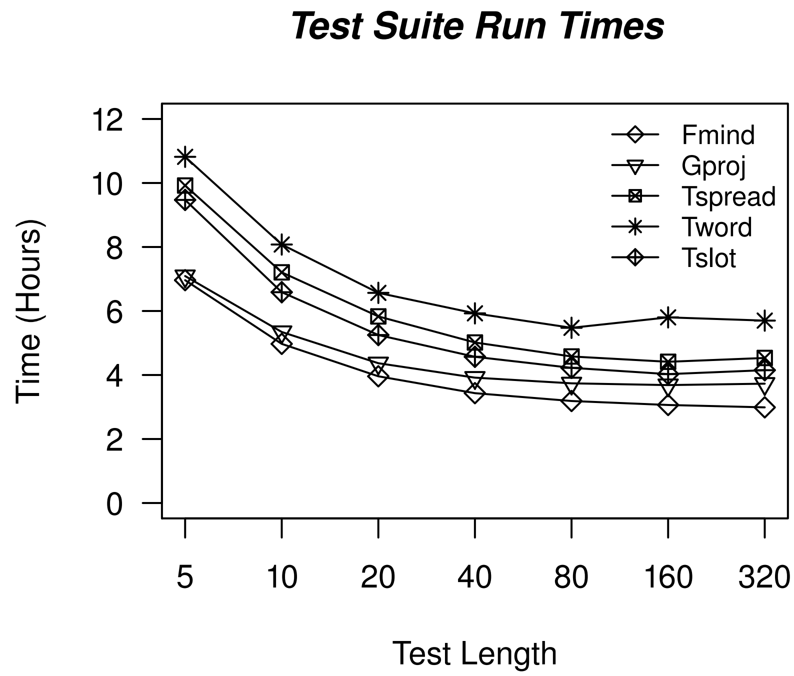

Figure 4.4: Run Time

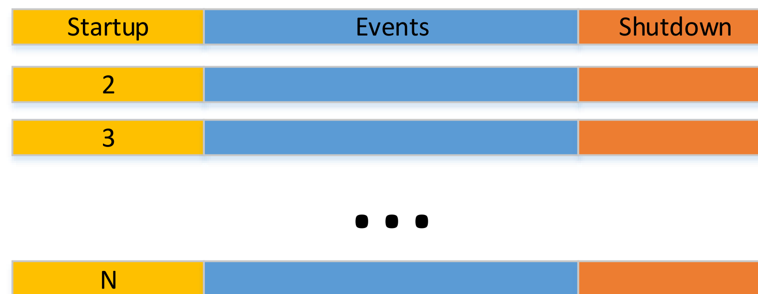

Figure 4.5: Cost Accumulation

4.5 Summary

In this chapter we discussed our experiment of determining which is better: running a large test suite of short test cases or running a small test suite or long test cases. Our results found that running tests of at least length 20 would guarantee better code coverage. Furthermore, tests of length 40 and longer outperform tests of length 20 in terms of finding faults. Overall we found that length 80 performed the best: it found maximal code coverage and fault-finding with minimal cost.

Chapter 5

Dynamic GUI Test Generators

In this chapter we will look at the system architecture and the seven dynamic test generator algorithms we implemented. We break the architecture down into the common components used by all of the algorithms and describe how they work. We then delve into the workings of each algorithm and give examples. Two of the algorithms, Q-Learning and ACO, have parameters that can be tuned that affect performance. For both ACO algorithms, we perform experiments to determine the best values for the tuning parameters.

5.1 System Architecture

In this section we will look at the system architecture and discuss the common components shared by all of the test generation algorithms. The system architecture is shown in Figure 5.1.

Application Under Test

The application under test is any application to be tested. In the case of our system, it can be any Java Swing application. The application is connected to the system by means of a window listener.

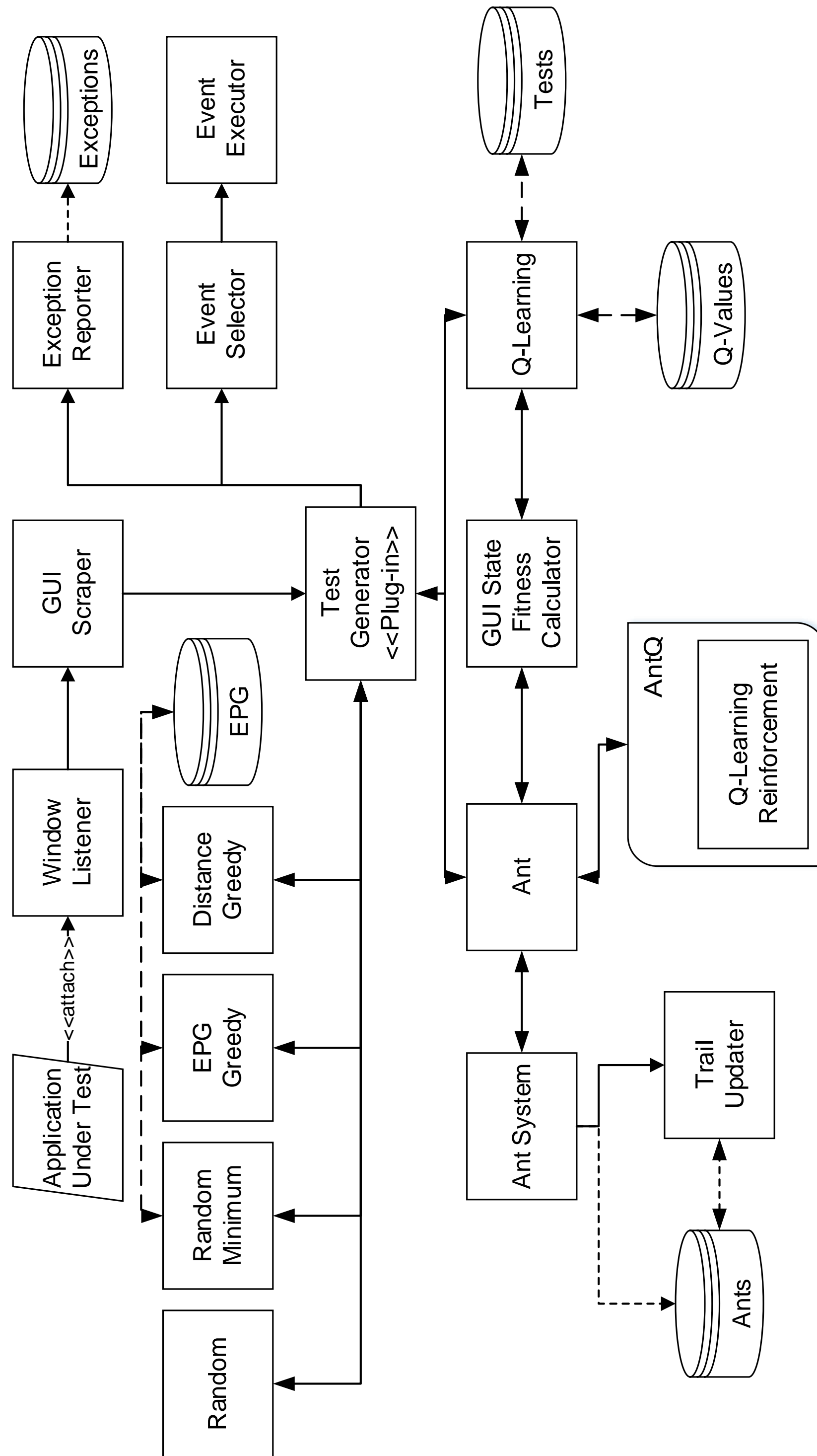

Figure 5.1: System Architecture

Window Listener

The window listener is the entry point for the application under test. The Java development kit provides a *WindowListener* interface, which we implement in order to listen for window events. The interface allows us to listen for window open, close, activated, and a variety of other window events. When a window is opened, we capture the window object and pass it to the GUI scraper. The system also determines if a modal window is open. If a modal window is open, then only that window is passed to the GUI scraper, otherwise *all* windows are passed to the scraper. This is done to ensure that the constraints imposed by the AUT are not bypassed.

GUI Scraper

The GUI scraper is how the system rips the GUI into its core components. The scraper takes in as input a GUI container (windows are a type of container) and recursively rips it until the bottom of the hierarchy is reached. Algorithm 3 shows how this procedure works.

Algorithm 3 GUI Scraper

```
 1: function GUIScraper(Container container, Boolean fullExtract)
 2:     components ← {}
 3:     children ← container.getChildren()
 4:     for child ∈ children do
 5:         if child isa Container then
 6:             if fullExtract then
 7:                 components.add(child)
 8:             end if
 9:             components.addAll(GUIScraper(child, fullExtract))
10:         else
11:             components.add(child)
12:         end if
13:     end for
14:     return components
15: end function
```

The GUI scraper works in two modes: full extraction mode or widgets only mode. The full extraction returns *every* component in the GUI, including containers. These components are used to calculate the state change ratio for both the ant colony algorithms and the Q-Learning

Table 5.1: Component Filters

Component	Filter Rules
Button	enabled, visible
TextComponent	editable, enabled, visible
Table	rows > 0, columns > 0, editable, enabled, visible
List	rows > 0, enabled, visible
Tree	rows > 0, enabled, visible
ComboBox	itemSize > 0, enabled, visible
CheckBox	enabled, visible
RadioButton	enabled, visible
Slider	enabled, visible
Spinner	enabled, visible

algorithm. The widget only mode is used to find only the components that can be executed, for example buttons and text boxes. When the GUI scraper returns the set of components for the widget only mode, the system needs to filter out the components that are not valid, such as buttons that are disabled. Table 5.1 shows the filters used for each type of component. Some of these components represent superclasses; for example, the button component can also refer to menu items. The main benefit of dynamic testing is that we are able to filter out the components based on their properties at run-time. Model-based testing methods tend to fail prematurely because they try to execute an event that is disabled. Even worse perhaps, is when a model-based system bypasses the GUI constraints and inputs data into a non-editable field such as a text field.

Test Generator

The test generator module is a superclass that is extended by each of the seven test generator algorithms. The module offers helper methods to the seven algorithms in the form of filter methods and timing methods. Each AUT can have specific filter rules specified and passed into the system via the command line. These rules can be used to ignore a specific component or window. For example, we ignore the *quit* and *exit* menu items in an application as we do not

want to shut down the AUT as we are testing it. Other examples include buttons that open a browser or pause the application for any reason. Components can be ignored based on their text or tool tip text. The timing methods are used to wait for the system to reach a stable state after an event has been executed. The amount of time to wait is set by the test engineer for each AUT as some systems may work more slowly than others.

Event Selector

The event selector applies different selection techniques depending on the algorithm being run. For random selection it will choose an event from the provided list at random using the built-in random number generator. For the random-minimum algorithm, it will choose an event with the lowest-value outgoing edge. For the Q-Learning algorithm it applies the ϵ-greedy selection process (see 2.4.2), which chooses a random event with probability ϵ and the event with the highest Q-value with probability 1-ϵ. Both the ant system and antq algorithms make use of the pseudo-random proportional selection rule, which selects the highest value event with probability q_0 and selects a random proportional event with probability 1-q_0. The selected event is passed to the event executor to be run on the AUT.

Event Executor

The event executor first determines the type of component being executed, such as a button, text box, or table. Depending on the component type, it will execute an action, or a random action if more than one exists. For components such as buttons and text boxes where only one action exists, it will programatically execute the event. In the case of a button, it will execute the *doClick*() method, and for a text box it will first place focus on the component and then send keyboard signals to type text. The text used is selected at random from a list of predefined inputs. In the case of components that have multiple event options, such as a table, it first determines the action to take using random selection. For example, a table can

have a single cell filled, a column filled to some cell y, a row filled to some cell x, or the entire table filled with data. Each of these options will have an equal probability of being selected. Once an action is decided, the cells of the table will be filled with random data chosen from the predefined list.

Exception Reporter

The exception reporter is a custom module that extends the *UncaughtExceptionHandler* provided by the JDK, which allows the system to catch and report on all uncaught exceptions thrown by a thread. When an exception is caught by the module, it generates a report containing the error message, the stack trace, and the events run up until the point the exception was thrown. The system is only able to catch uncaught exceptions, so any exceptions thrown, caught, and logged by the AUT are not reported and are not considered errors.

GUI State Fitness Calculator

The GUI state fitness calculator is used by the ant algorithms and the Q-Learning algorithm in order to generate the state information used for the fitness function, as well as to compare two states in order to calculate the ratio of change. Each of the algorithms provides a list of components that make up the current state of the AUT. The module then extracts and stores specific and relevant properties from each widget. For example, a button's text, class, position in the hierarchy, and whether or not it is enabled are all properties that are stored. The module returns an object that stores this state information to the calling algorithm module. The calculator can also take in two different states and compare them for changes. The module will return a value in the range [0,1] that represents the ratio of change. The specifics of how this is calculated is discussed in section 5.4.1.

5.2 Random Test Generator

We implemented two random test generators, and we describe their algorithms in this section. We use the random test generators as a baseline of comparison for our other five algorithms. Besides being used simply as a baseline for comparison, we also want to study the effectivness of random testing when applied to GUI testing. As mentioned in Chapter 2, few researchers actually compare against random testing, but rather choose to compare against model-based or non-GUI testing systems.

5.2.1 Random Selection

The first random algorithm we implemented is a pure random selection system. The algorithm is shown in Algorithm 4. The system starts the AUT, retrieves the current list of available components, selects one randomly, and executes it. It continues this process until the set length of the test case has been reached. Finally, it shuts down the AUT and writes the coverage information. As well, after each event is executed, it checks to see if an exception has been thrown. If so, it logs the exception to a file.

Algorithm 4 Random Selection Test Generator

```
1:  function RandomSelection(length)
2:      startAUT()
3:      for i ∈ length do
4:          components ← getAvailableComponents()
5:          event ← selectRandomEvent(components)
6:          execute(event)
7:          waitForGUI()
8:          if ExceptionManager.hasException() then
9:              ExceptionManager.reportException()
10:         end if
11:     end for
12:     shutdownAUT()
13:     writeCoverage()
14: end function
```

5.2.2 Random Minimum Selection

The random minimum selection test generator works by always executing the event with the smallest edge value. Although in Chapter 3 we described the edge weights of an XEPG as being the distance, it is possible to assign any value to the edges; it just so happens we use distance most often. In the case of the random minimum algorithm, the edge weights represent the number of times the edge has been traversed. Initially all edge values are set to zero. When an edge is traversed, the weight is increased by one. When making a selection for the next event to execute, all of the available edge weights are compared and the one with the smallest value is selected. If more than one edge meets this criterion, one is selected at random.

Algorithm 5 shows the details. Line 3 of the algorithm references a method call to *getPseudoInitialEvent()*, which introduces the concept of the pseudo initial event. An EPG can be extended to include this event, which represents the initial state of the system after startup. In order to keep track of the first *real* initial events that have already been executed in previous tests, we need to store their incoming edges in the EPG. We connect these edges to the pseudo initial event so that we can look up the edge values when deciding which real initial event we want to execute; otherwise we would have to select an initial event at random.

Figure 5.2 shows an example of how the algorithm selects edges for two test cases. The first graph in the figure shows the flow of events for the first test and it shows the edge values after the first test has been generated. The second graph shows the flow of events for the second test and the resulting edge values.

5.3 Greedy Test Generator

We created two greedy test generators that make use of EPGs. The first test generator attempts to maximize the number of event pairs covered, whereas the second test generator attempts to minimize the distance between events over time.

Algorithm 5 Random Minimum Test Generator

```
 1: function RandomMinimumSelection(length, epg)
 2:     candidates ← {}
 3:     lastEventRun ← getPseudoInitialEvent()
 4:     startAUT()
 5:     for i ∈ length do
 6:         currentMinEdge ← ∞
 7:         components ← getAvailableComponents()
 8:         for comp ∈ components do
 9:             newMinEdge ← epg.getEdgeValue(lastEventRun, comp)
10:             if newMinEdge < currentMinEdge then
11:                 currentMinEdge ← newMinEdge
12:                 candidates ← {}
13:                 candidates.add(comp)
14:             else if newMinEdge == currentMinEdge then
15:                 candidates.add(comp)
16:             end if
17:         end for
18:         event ← selectRandom(candidates)
19:         execute(event)
20:         waitForGUI()
21:         epg.update(lastEventRun, event, currentMinEdge + 1)
22:         lastEventRun ← event
23:         if ExceptionManager.hasException() then
24:             ExceptionManager.reportException()
25:         end if
26:         candidates ← {}
27:     end for
28:     shutdownAUT()
29:     writeCoverage()
30: end function
```

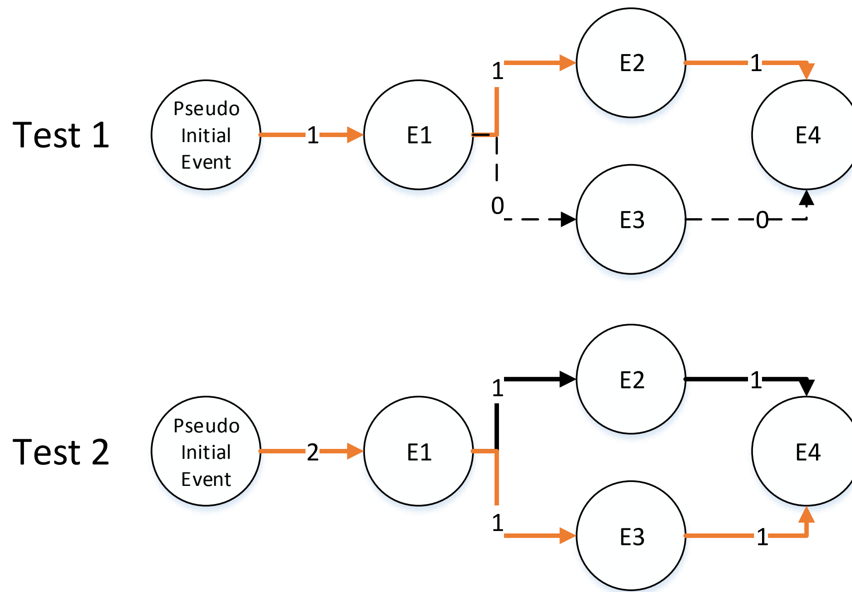

Figure 5.2: Random Minimum Example with Two Tests

5.3.1 Event-pair Greedy Selection

Algorithm 6 and 7 show the algorithm for the event-pair greedy test generator. The general idea of the algorithm is to select the event that will create the most new connections in the EPG. The random minimum test generator looks at only the most recent event run when determining the next event to select. The event-pair greedy algorithm takes into consideration *all* previously run events in the test case in order to determine the number of potential event pairs that will be covered.

We will now illustrate the algorithm with an example. Figure 5.3 shows a graph with no connected edges. The edges shown represent the potential paths a test can take. Since all of the edges in the graph are not covered, all edges have the same probability of being selected. Therefore the first test can follow any edge from the initial state to the last node. For this example the first test case will be $\{e_1, e_2, e_5\}$. The resulting XEPG is shown in Figure 5.4. Recall that the edges here only represent nodes that have been executed within the same test case and not the potential flow of execution. When generating the second test case, the algorithm uses

Algorithm 6 Event-pair Greedy Test Generator

```
1:  function EVENTPAIRGREEDY(length, epg)
2:      candidates ← {}
3:      eventsRun ← {getPseudoInitialEvent()}
4:      startAUT()
5:      for i ∈ length do
6:          maxPotentialEP ← 0
7:          components ← getAvailableComponents()
8:          for comp ∈ components do
9:              newPotentialEP ← countNewEventPairs(eventsRun, comp, epg)
10:             if newPotentialEP > maxPotentialEP then
11:                 maxPotentialEP ← newPotentialEP
12:                 candidates ← {}
13:                 candidates.add(comp)
14:             else if newPotentialEP == maxPotentialEP then
15:                 candidates.add(comp)
16:             end if
17:         end for
18:         event ← selectRandom(candidates)
19:         execute(event)
20:         epg.update(eventsRun, event)
21:         eventsRun.add(event)
22:         if ExceptionManager.hasException() then
23:             ExceptionManager.reportException()
24:         end if
25:         waitForGUI()
26:         candidates ← {}
27:     end for
28:     shutdownAUT()
29:     writeCoverage()
30: end function
```

Algorithm 7 Event-pair Counter

```
1: function COUNTNEWEVENTPAIRS(eventsRun, comp, epg)
2:     count ← 0
3:     for event ∈ eventsRun do
4:         if !epg.hasEdge(event, comp) then
5:             count ← count + 1
6:         end if
7:     end for
8:     return count
9: end function
```

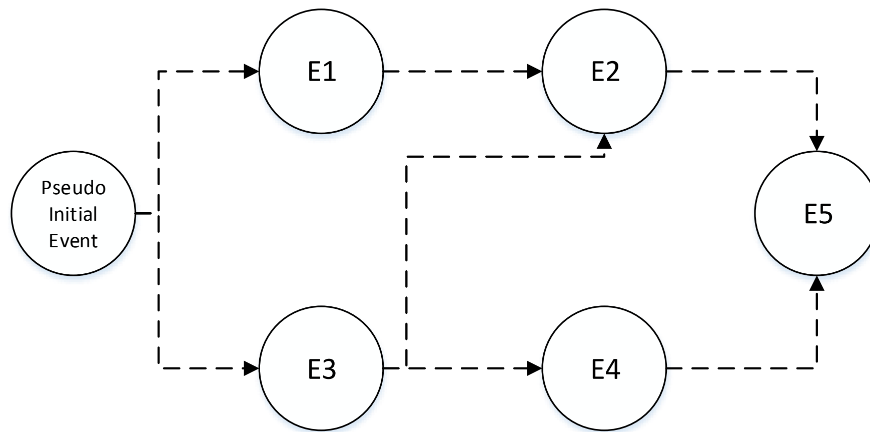

Figure 5.3: Initial State with Uncovered Edges

the EPG to determine the next event to execute. In this example it would compare events e_1 and e_3 and determine how many new pairs would be created if each one were to be selected. Since the event pair <*initialEvent*, e_1> already exists and since the event pair <*initialEvent*, e_3> does not, it would select event e_3 as it would be creating one more event pair. Following that, the next two events to compare would be e_2 and e_4. If e_2 were to be selected, it would create one more event pair with e_3, however if e_4 were to be selected, it would create event pairs with e_3 and *initialEvent*. Therefore event e_4 is selected. Finally e_5 is selected as it is the only remaining event. The result is shown in Figure 5.5. If we were to generate a third test, both e_1 and e_3 would have equal probability of being selected as neither one will add new event pairs. Furthermore, if event e_3 were to be selected, then event pair <e_3, e_2> would be guaranteed to be covered.

5.3.2 Distance Reduction Greedy Selection

The distance reduction algorithm is not very different than the event pair greedy algorithm. Instead of trying to maximize the event-pair coverage, the algorithm attempts to reduce the distance between all pairs of events. The event that has the greatest reduction in distance will be selected for execution. The idea is that by executing events closer together, we will see

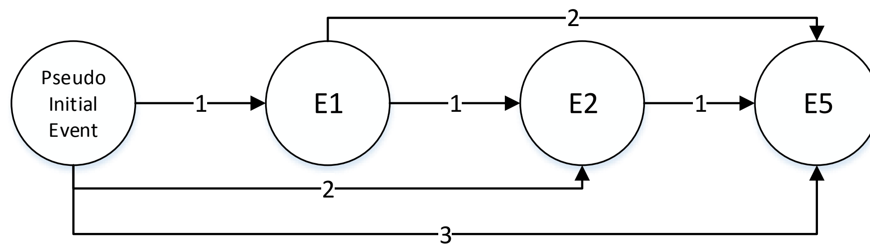

Figure 5.4: Resulting EPG After Test 1: E1, E2, E5

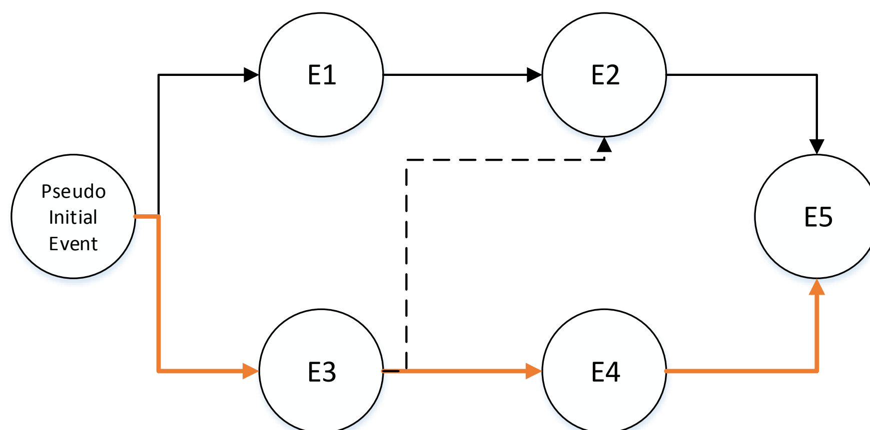

Figure 5.5: Test Two's Path

different results and possibly find bugs. Algorithms 8 and 9 show the details of the algorithm. As with the event pair greedy algorithm, the distance reduction algorithm takes into account all previously run events within a test when deciding the next event to execute.

Algorithm 8 Distance Reduction Greedy Test Generator

```
1:  function DistanceReductionGreedy(length, epg)
2:      candidates ← {}
3:      eventsRun ← {getPseudoInitialEvent()}
4:      startAUT()
5:      for i ∈ length do
6:          currentMaxReduction ← 0
7:          components ← getAvailableComponents()
8:          for comp ∈ components do
9:              newMaxReduction ← countMaxReduction(eventsRun, comp, epg)
10:             if newMaxReduction > currentMaxReduction then
11:                 currentMaxReduction ← newMaxReduction
12:                 candidates ← {}
13:                 candidates.add(comp)
14:             else if newMaxReduction == currentMaxReduction then
15:                 candidates.add(comp)
16:             end if
17:         end for
18:         event ← selectRandom(candidates)
19:         execute(event)
20:         waitForGUI()
21:         epg.update(eventsRun, event)
22:         eventsRun.add(event)
23:         if ExceptionManager.hasException() then
24:             ExceptionManager.reportException()
25:         end if
26:         candidates ← {}
27:     end for
28:     shutdownAUT()
29:     writeCoverage()
30: end function
```

To better understand how the algorithm works, we will look at an example. Figure 5.7 shows an example test case. Events e_1 and e_2 have already been selected. There are two possible events to execute following e_2, which are events e_3 and e_4. In order to determine which of these two events to select, the algorithm calculates the greatest reduction in distance

Algorithm 9 Distance Reduction Counter

```
1:  function countMaxReduction(eventsRun, comp, epg)
2:      count ← 0
3:      compPosition ← eventsRun.length() + 1
4:      eventPosition ← 1
5:      for event ∈ eventsRun do
6:          oldDistance ← epg.getDistance(event, comp)
7:          newDistance ← compPosition − eventPosition
8:          if oldDistance > 0 and newDistance < oldDistance then
9:              count ← count + (oldDistance − newDistance)
10:         else if oldDistance == 0 then                    ▷ Does not exist
11:             count ← count + newDistance
12:         end if
13:         eventPosition ← eventPosition + 1
14:     end for
15:     return count
16: end function
```

Table 5.2: Distance Reduction Example

Events	Current Distance	New Distance	Difference
e_1 & e_3	5	2	3
e_2 & e_3	1	1	0
e_1 & e_4	4	2	2
e_2 & e_4	3	1	2

for both events using the current XEPG; here we assume that the current XEPG is as in Figure 5.6. In this case the XEPG is non-empty for illustrative purposes. The event with the largest reduction is selected. Table 5.2 shows the difference reduction calculation. Event e_3 has a total reduction of three and event e_4 has a total reduction of four, therefore event e_4 will be selected. It should be noted that if the new distance is greater than the old distance, we count that difference as zero and not as a negative value. Furthermore, if there is no current distance between two events (they have never been run in the same test case), we simply add the new distance to the total difference.

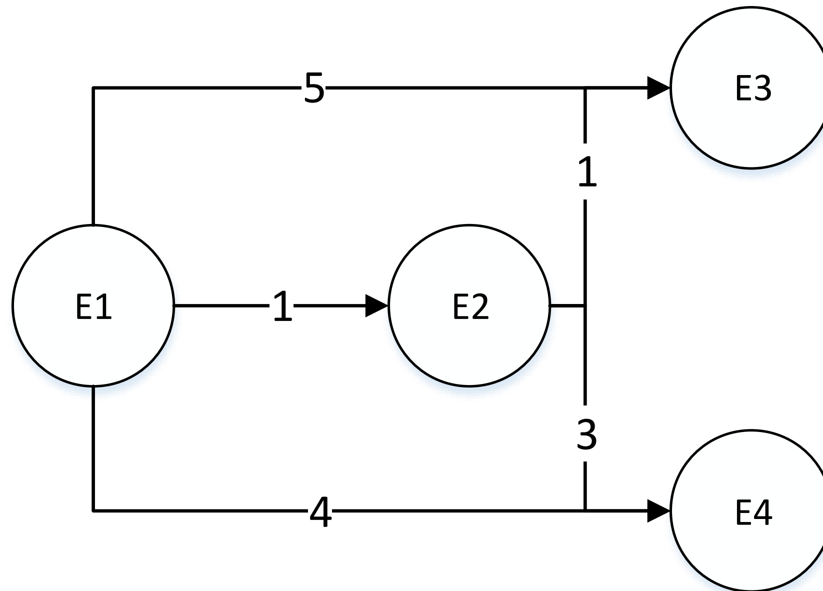

Figure 5.6: Distance Example: Current EPG

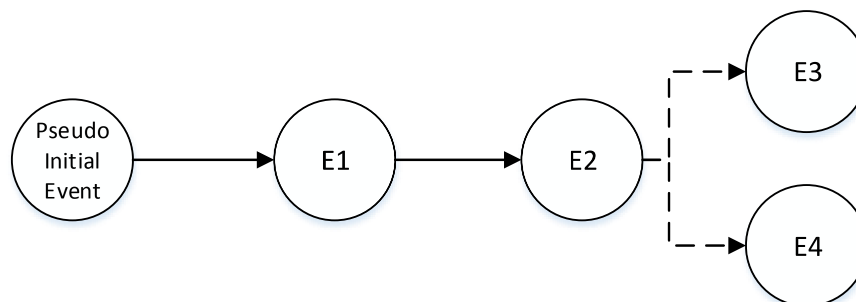

Figure 5.7: Distance Example: Test Case with Two Options

5.4 Q-Learning Test Generator

Here we describe our algorithm for generating tests using Q-Learning. We implemented the algorithm described by Mariani et al. [41]. As well, we used the same fitness function when determining the Q-values. The fitness function calculates the ratio of change between two states after an event is executed. We discuss the details of the fitness function in the next section. The algorithm for Q-Learning is shown in Algorithm 10. An interesting note about how the algorithm works is that it can start from any previously found state, including the initial state. Therefore a test will run a previously created test case in order to put the system into a specific state. However, if some event in the previous test is unable to be run, then the new test starts from the current state. The algorithm implements the ϵ-greedy selection policy, which states that a random event is selected with probability ϵ and the event with the highest Q-value is chosen with probability 1-ϵ. The q_0 value used by the algorithm represents the ϵ value and is supplied by the test engineer. We evaluate different values for this parameter in Section 5.4.2.

5.4.1 Calculating State Change

Q-Learning works by associating rewards with actions. In the context of GUI testing, the reward can be based on a number of factors, such as statement and branch coverage. We chose to base rewards on the amount of change in the GUI state. This reward was used by AutoBlackTest [41] and we borrowed it for our own work. We find the reward function interesting because if it results in good test cases, then we can say there is a link between the states of a GUI and the amount of code coverage and faults found. We also used the same fitness function for our ant algorithms. In this thesis we take the terms “reward” and “fitness” to mean the same thing.

The algorithm measures the state change ratio by capturing the state before an event is run, executing an event, and capturing the resulting state. The system takes into account two factors when calculating the state change ratio. First, it compares the properties of each widget that

Algorithm 10 Q-Learning Test Generator

```
1: function QLEARNING(length, qvalues, q_0, previousTests)
2:     lastEvent ← getPseudoInitialEvent()
3:     startAUT()
4:     testSteps ← {}
5:     previousTest ← getRandomTest(previousTests)
6:     executePrevious(previousTest)                          ▷ Execute a previous test
7:     testSteps.addAll(previousTest.getSteps())
8:     for i = 0; i < length; i + + do
9:         components ← getComponents(false)    ▷ False: extract executable widgets only
10:        currentState ← getAbstractState(getComponents(true))
11:        q ← Random.nextDouble()
12:        if q < q_0 then
13:            event ← getRandom(components)
14:        else
15:            event ← getMaxQ(lastEvent, components, qvalues)
16:        end if
17:        execute(event)
18:        waitForGUI()
19:        testSteps.add(event)
20:        if ExceptionManager.hasException() then
21:            ExceptionManager.reportException()
22:        end if
23:        newState ← getAbstractState(getComponents(true))
24:        stateDiff ← getStateChangeRatio(currentState, newState)
25:        qvalues.update(lastEvent, event, stateDiff)
26:        lastEvent ← event
27:    end for
28:    shutdownAUT()
29:    writeCoverage()
30:    writeTest(testSteps)
31: end function
```

exists in both states. Second, it looks for any new widgets that exist in the new state and not in the original state. The algorithm does not take into account any widgets that disappear–that is, widgets that exist in the original state but not the new state–since we want to encourage actions that provide access to more widgets.

The state of a GUI can be broken down into the state of the individual widgets currently within that state. Each widget within the state contains a set of properties that describe it. For example, a text box's properties would contain its current text, whether or not it is enabled and editable, and so on. Furthermore, there are special properties, which we call *traits*, that can be used to identify the widget. We need to be able to identify each widget so that we can accurately compare the properties of each widget. Formally we say a state S contains a set of widgets $<w_1, w_2, ..., w_n>$, where each widget w contains a set of properties $<p_1, p_2, ..., p_n>$. We say that a property p is a trait if it can be used to identify the widget.

When comparing two widgets, w_1 and w_2, we say $w_1 =_t w_2$ iff $| w_1 |=| w_2 | \wedge \forall t_i \in w_1, \exists t_j \in w_2$ s.t. $t_i = t_j$, where t is a trait in w. That is, two widgets are equivalent if all of their traits are equal. If one widget has a different number of traits than the other, the two widgets are considered to be different.

When comparing two states, we only compare a subset of the widgets and their properties. We call this subset the *Abstract State*. An abstract state AS contains a set of *Abstract Widgets*, where each abstract widget contains a subset of the widget's properties. We use a subset of a widget's properties as only some properties are relevant for state change comparison. The more properties we use in our calculation, the less impact the change of a single property will have.

If we have two abstract states, $AS = \{w_1, ...w_n\}$ and $AS' = \{w'_1, ...w'_n\}$, we define the restriction operator $AS \setminus_t AS' = \{w_i \mid w_i \in AS \wedge \nexists w_k \in AS'$ s.t. $w_i =_t w_k\}$; that is, the set of widgets in AS that do not appear in AS'. This restriction operator is used in our calculation of state change ratios.

Equation (5.1) is used to calculate the ratio of change between two widgets if $w_1 =_t w_2$.

The equation determines the ratio of change based on each widget's properties.

$$\text{diff}_w(w_1, w_2) = \frac{| P_1 \backslash P_2 | + | P_2 \backslash P_1 |}{| P_1 | + | P_2 |} \tag{5.1}$$

where P_i is the set of properties and $P_i \backslash P_j$ is the set of properties in P_i and not in P_j.

We use (5.2) to compare two states and determine the ratio of change. The equation looks at both the widgets that exist in the new state and not the old state, and it looks at the change in each property for each widget.

$$\text{diff}_{AS}(AS_1, AS_2) = \frac{| AS_2 \backslash_t AS_1 | + \sum_{w_1 \in AS_1, w_2 \in AS_2, w_1 =_t w_2} \text{diff}_w(w_1, w_2)}{| AS_2 |} \tag{5.2}$$

5.4.2 Parameter Tuning

To restate, calculating Q-values is done using the following equation:

$$Q(s, a) = Q(s, a) + \alpha[reward(s, a) + \gamma \max_{a'} Q(\delta(s, a), a') - Q(s, a)] \tag{5.3}$$

where α is the learning rate, γ is the discount factor, $reward(s, a)$ is the reward of taking action a in state s, $\delta(s, a)$ is the state reached by executing the action a from state s, and $Q(\delta(s, a), a')$ indicates the Q-value associated with the action a' when executed from the state $\delta(s, a)$.

For our system we use an α value of one since we want to favour the new information over the current information. Therefore, the equation simplifies to the following:

$$Q(s, a) = reward(s, a) + \gamma \max_{a'} Q(\delta(s, a), a') \tag{5.4}$$

The equation only looks at the reward of the action taken and the best Q-value of all the new potential actions.

Before we can begin testing with our Q-Learning system, we need to determine the best values for the parameters. In this case we have two parameters: the γ value and the ϵ-greedy

value (see section 2.4.2). In the case of the γ value, we choose to use 0.9 as that is the value found to be most effective by the original authors [41].

For the ϵ value, we decided to run experiments on three applications to see the effect different values have on the code coverage results. We ran experiments on three subject applications: TerpSpreadsheet, TerpWord, and ArgoUML. We selected values of 0.2, 0.4, 0.6, 0.8, and 1.0 for ϵ. The experiments consisted of test suites containing 100 tests where each test contained 20 events. We repeated each experiment five times and measured the average statement coverage. The results are shown in Table 5.3. Both TerpWord and TerpSpreadsheet achieved the highest levels of coverage with ϵ = 1.0, which is pure random selection. ArgoUML achieved the highest levels of coverage with ϵ = 0.4.

We performed ANOVA analyses followed by a Tukey tests to determine if there were significant differences between the means for each value. The results of the ANOVA analysis for TerpSpreadsheet showed that there was a significant effect of the value of ϵ on code coverage at the 0.05 significance level for the five levels [f(4, 20) = 16.13, p <0.001]. The results of the Tukey test showed that the mean score for an ϵ value of 0.2 (M = 3507, SD = 36.79) was significantly less than 0.4 (M = 3706.6, SD = 144.06), 0.6 (M = 3768.8, SD = 68.42), 0.8 (M = 3792.2, SD = 57.22), and 1.0 (M = 3896.4, SD = 46.97). Furthermore, an ϵ value of 1.0 was significantly better than 0.4.

The results of the ANOVA analysis for TerpWord showed that there was a significant effect of the random selection variable on code coverage at the 0.05 significance level for the five levels [F(4, 20) = 4.82, p = 0.007]. The results of the Tukey test showed that the mean score for an ϵ value of 0.2 (M = 5831.6, SD = 235.78) was significantly less than 1.0 (M = 6701.72, SD = 337.43). No other pairs of values were significantly different.

The results of the ANOVA analysis for ArgoUML showed that there was a significant effect of the random selection variable on code coverage at the 0.05 significance level for the five levels [f(4, 20) = 4.4, P = 0.01]. The results of the Tukey test showed that the mean score for an ϵ value of 0.2 (M = 18572, SD = 894.08) was significantly less than 0.4 (M = 19874.6,

Table 5.3: Q-Learning Parameter Tuning - Largest mean in bold

ϵ	Statements		
	TerpSpreadsheet	**TerpWord**	**ArgoUML**
0.2	3507	5832	18572
0.4	3707	6234	**19875**
0.6	3769	6327	19531
0.8	3792	6274	19301
1.0	**3896**	**6701**	19407

SD = 472.23). No other pairs of values had a significant difference.

From the results of the analyses, a random selection value of at least 0.4 for TerpWord is beneficial. For TerpSpreadsheet, a higher random value also shows to perform better with both values of 0.2 and 0.4 performing significantly worse than 1.0. Finally with ArgoUML, the results showed that a value of at least 0.4 is better than 0.2. Q-Learning requires some random selection as it allows the algorithm to run in an exploratory manner. Always selecting the best Q-value would result in the same few tests being generated. For the smaller applications we chose to use a value of 0.9. This value allows for a lot of random exploration. We decided against using a value of 1 as that would result in only random actions being taken, which defeats the purpose of Q-Learning. For the larger applications, we chose to use a value of 0.4 as it worked well for ArgoUML.

5.5 Ant Colony Optimization Test Generators

In this section we will describe our two ant algorithms. The work was originally published in [15]. Since GUIs are best represented by graphs, and since we want to traverse those graphs to find interesting paths, we decided to apply ant colony optimization algorithms to the GUI graphs. We implemented two algorithms: the traditional ant system and the antq algorithm. The traditional system implements the traditional ant colony algorithm [26], whereas the antq algorithm implements the antq algorithm as described in [24]. The main difference between

the two algorithms is how they update their paths. The ant system updates *all* paths in the graph after the generation of ants has completed its run, even those not traversed by an ant. The antq algorithm updates the paths in two phases. First, it updates the path immediately after an ant has traversed it. The edge is updated using the Q-Learning portion of the algorithm. Second, once each ant in the generation has completed its run, the paths are updated again; however, only those paths that were traversed by an ant are updated. All other paths remain untouched. The details of both algorithms can be seen in Algorithms 11 and 12.

5.5.1 Event Selection

Event selection works as follows. We assign a score to each edge (x,y) coming from the source vertex:

$$score_{xy} = \tau_{xy}^{\alpha} \cdot \eta_{xy}^{\beta} \tag{5.5}$$

where τ_{xy}^{α} is the amount of pheromone on edge (x,y) and η_{xy}^{β} is the heuristic value of edge (x,y). Using the pseudo random proportional selection rule, we select the event with the highest score as follows. We first capture the events with the highest score:

$$maxevents_x = \{y \mid score_{xy} = \max_{u \in allowed_x}(score_{xu})\} \tag{5.6}$$

where y is an event and $allowed_x$ is the set of events reachable by x. With probability q_0, the next event is chosen randomly from $maxevents_x$. The probability of every event is

$$\frac{1}{\mid maxevents_x \mid} \tag{5.7}$$

Otherwise, with probability $1 - q_0$, the next event is chosen in a weighted random fashion from $allowed_x$. The probability of every event y is

$$\frac{score_{xy}}{\sum_{u \in allowed_x}(score_{xu})} \tag{5.8}$$

An event is selected with a probability proportional to is pheromone and heuristic values. Since our algorithms do not use a heuristic, only the pheromone values are used. In order to guarantee that an event is selected, the events are given a porportional range in [0,1]. If the randomly generated value falls within the events range, the event is selected.

5.5.2 Pheromone Update

As mentioned previously, the ant algorithms differ in how they update their pheromone values. The ant system updates its pheromones using the following equation.

$$AQ(x, y) = (1 - \alpha) \cdot AQ(x, y) + \Delta AQ(x, y) \tag{5.9}$$

where α is the pheromone evaporation rate. After the generation of ants has completed its run, *all* edges are updated. If an edge receives no pheromone, then its current pheromone value will be reduced. The amount of pheromone deposited on each edge (x,y) is calculated using Equation 5.10.

$$\Delta AQ(x, y) = \begin{cases} \frac{\sum_{i=1}^{k} SC(ant_i)}{k} & \text{if (x,y) cov. by any } ant \in k_{best} \\ 0 & \text{otherwise} \end{cases} \tag{5.10}$$

where k_{best} is the set of the best k ants in the current generation and where the k value is determined by the test engineer beforehand. That is, in a generation consiting of n ants, $k \leq n$. The amount of pheromone to deposit is calculated by the $SC()$ function, which works as follows.

$$SC(ant_i) = \frac{StateChange(ant_i)}{TourLength(ant_i)} \cdot UniqueEvents(ant_i) \tag{5.11}$$

where ant_i is the i-th best ant run in the current generation, $StateChange()$ is the cumulative state change of the entire ant's run, $TourLength()$ is the length of the ant's run, and $UniqueEvents()$ is the number of unique events executed during the ant's run, where $0 \leq UniqueEvents() \leq TourLength()$. When calculating the state change using the $StateChange()$ method, the system only counts the amount of change once for each event in an ant's run, regardless if the event is executed multiple times. This is done to discourage tests repeating the same high state changing events. Furthermore, we further favour unique events by dividing by the length of the tour and multiplying by the number of unique events. This will favour tests with more unique events.

The antq algorithm works differently than the ant system algorithm. The equation for how antq updates its pheromones is as follows.

$$AQ(x,y) = (1-\alpha) \cdot AQ(x,y) + \alpha \cdot \left(\Delta AQ(x,y) + \gamma \cdot \operatorname*{Max}_{u \in allowed_y} AQ(y,u) \right) \tag{5.12}$$

where γ is the discount factor, which is a part of the Q-Learning model. When an ant makes a transition from one event to another, it immediately updates the edge using Q-Learning reinforcement. During this phase of the process the $\Delta AQ(x,y)$ value is 0. That is, we only update the edge using the current pheromone value and the best outgoing pheromone value from the new event. Once the generation of ants has completed its run, only the edges touched by an ant are updated using Equation 5.10. During this phase of the process the $\operatorname*{Max}_{u \in allowed_y} AQ(y,u)$ value is 0. That is, we only take into account the current pheromone value and the fitness of each ant's run.

5.5.3 Parameter Tuning

As with the Q-Learning algorithm, we needed to tune the parameters for both of the ant algorithms. In this section we discuss the experiments conducted for both the ant system and antq system in order to determine good parameter values.

Algorithm 11 Ant System Algorithm

```
1: function AntSystem(length, generations, numAnts, q_0, trailsGraph)
2:     for i ∈ generations do
3:         stateDiffs ← {}
4:         ants ← {}
5:         for j ∈ numAnts do
6:             cumulativeDiff ← 0
7:             antPath ← {getPseudoInitialEvent()}
8:             startAUT()
9:             for k ∈ length do
10:                components ← getComponents(false)
11:                q ← Rand.nextDouble()
12:                if q < q_0 then
13:                    event ← getMax(antPath, components, trailsGraph)
14:                else
15:                    event ← getProportional(antPath, components, trailsGraph)
16:                end if
17:                currentState ← getAbstractState(getComponents(true))
18:                execute(event)
19:                waitForGUI()
20:                newState ← getAbstractState(getComponents(true))
21:                diff ← getStateChangeRatio(currentState, newState)
22:                cumulativeDiff ← cumulativeDiff + diff
23:                currentState ← newState
24:                antPath.add(event)
25:                if ExceptionManager.hasException() then
26:                    ExceptionManager.reportException()
27:                end if
28:            end for
29:            shutdownAUT()
30:            writeCoverage()
31:            stateDiffs.add(cumulativeDiff)
32:            ants.add(antPath)
33:        end for
34:        trailsGraph.updateAll(stateDiffs, ants)
35:        //Eq. 5.9. Ant Collaboration
36:    end for
37: end function
```

Algorithm 12 AntQ Algorithm - Lines 25 & 37 show how it differs from ant system

```
1:  function AntQ(length, generations, numAnts, q_0, trailsGraph)
2:      for i ∈ generations do
3:          stateDiffs ← {}
4:          ants ← {}
5:          for j ∈ numAnts do
6:              cumalativeDiff ← 0
7:              antPath ← {getPseudoInitialEvent()}
8:              startAUT()
9:              for k ∈ length do
10:                 components ← getComponents(false)
11:                 q ← Rand.nextDouble()
12:                 if q < q_0 then
13:                     event ← getMax(antPath, components, trailsGraph)
14:                 else
15:                     event ← getProportional(antPath, components, trailsGraph)
16:                 end if
17:                 currentState ← getAbstractState(getComponents(true))
18:                 execute(event)
19:                 waitForGUI()
20:                 newState ← getAbstractState(getComponents(true))
21:                 diff ← getStateChangeRatio(currentState, newState)
22:                 cumalativeDiff ← cumalativeDiff + diff
23:                 currentState ← newState
24:                 antPath.add(event)
25:                 trailsGraph.updateTrail(antPath, getComponents(false))
26:                 //Eq. 5.12. Q-Learning Reinforcement
27:                 //ΔAQ(x, y) is 0
28:                 if ExceptionManager.hasException() then
29:                     ExceptionManager.reportException()
30:                 end if
31:             end for
32:             shutdownAUT()
33:             writeCoverage()
34:             stateDiffs.add(cumalativeDiff)
35:             ants.add(antPath)
36:         end for
37:         trailsGraph.updateTouchedTrails(stateDiffs, ants)
38:         //Eq. 5.12. Ant Collaboration
39:         //γ · Max_{u∈allowed_y} AQ(y, u) is 0
40:     end for
41: end function
```

Ant System Tuning

In order to determine good parameter values, we conducted experiments on TerpSpreadsheet, TerpWord, and ArgoUML. For both the α and q_0 values we chose to experiment with the values 0.3, 0.6, and 0.9. Since we do not know the effects of any parameter values in this context, and since it is not cost-effective to test all values in [0,1], we chose to test a spread of values. We are not necessarily looking for the optimal parameter values, but rather we are looking for guidance. As well, we want to determine if the parameter values have any significant effect whatsoever. As with the Q-Learning experiments, these experiments consisted of test suites of size 100 where each test was of length 20. We used 10 ants per generation and used the best seven ants to update the edges. We ran each experiment five times and measured the code coverage. The results can be seen in Table 5.4.

The results of the factorial ANOVA for TerpSpreadsheet showed that there was a significant effect at the 0.05 level for both the evaporaton rate and the random selection value, but not their interaction. The main effect for the evaporation rate yielded an F ratio of F(2, 36) = 9.816, p <0.001. The post-hoc Tukey test showed that both values of 0.3 (M = 3621.07, SD = 110.34) and 0.6 (M = 3618.53, SD = 87.92) were significantly better than an α value of 0.9 (M = 3505.07, SD = 85.11). The main effect for the random selection value yieled an F ratio of F(2, 36) = 8.942, p = 0.001. The Tukey test showed that a q_0 value of 0.3 (M = 3646.73, SD = 98.43) was significantly better than a value of 0.9 (M = 3520.56, SD = 107.85). The interaction between the two factors resulted in an F score of F(4, 36) = 0.695, p = 0.6.

The results of the factorial ANOVA for TerpWord showed that there was a significant effect at the 0.05 level for both the evaporation rate and the random selection value, but not their interaction. The main effect for the evaporation rate yielded an F ratio of F(2, 36) = 4.705, p = 0.015. The Tukey test showed an α value of 0.3 (M = 5976.33, SD = 365.86) was significantly better than a value of 0.9 (M = 5596.87, SD = 395.56). The main effect for the random selection value yielded an F ratio of F(2, 36) = 4.011, p = 0.027. The Tukey test revealed that a q_0 value of 0.3 (M = 6036.2, SD = 374.57) was significantly better than a value of 0.6 (M = 5693.87,

SD = 353.8). The interaction between the evaporation rate and the random selection value yielded an F ratio of F(4, 36) = 0.363, p = 0.834.

As with the two previous results, the results of the factorial ANOVA for ArgoUML showed that there was a significant effect at the 0.05 level for both the evaporation rate and the random selection value, but not their interaction. The main effect for the evaporation rate resulted in an F ratio of F(2, 36) = 7.081, p = 0.003. The Tukey test showed that an α value of 0.3 (M = 19555.13, SD = 576.38) and a value of 0.6 (M = 19762.33, SD = 676.39) were both signicantly better than a value of 0.9 (M = 19080.53, SD = 867.62). The main effect for the random selection paramater yielded an F ratio of F(2, 36) = 22.364, p <0.001. The Tukey test revealed that a q_0 value of 0.3 (M = 20027.4, SD = 379.07) was significantly better than both 0.6 (M = 19571.93, SD = 557.12) and 0.9 (M = 18798.67, SD = 723.42). Furthermore, 0.6 was significantly better than 0.9. Finally, the interaction between the two factors yielded an F ratio of F(4, 36) = 0.678, p = 0.612.

In summary, all three applications benefited from low α values and low q_0 values. Having low α values means that the pheromones will reduce at a slower rate. Since the algorithm reduces *all* pheromones regardless if the edge has been traversed, having a low α value allows for the uncovered edges to remain desirable with high pheromone values. Furthermore, low q_0 values will favour random proportional selection rather than selecting the maximum edge value, which also favours exploration.

AntQ Tuning

As with the ant system, we conducted experiments on TerpSpreadsheet, TerpWord, and ArgoUML. We ran test suites of size 100, where each test case was of length 20. We repeated each experiment five times. Since we were attempting to tune three parameters, the number of combinations (27) was too large to test in a reasonable amount of time[1]. Instead we used a two-way combinatorial system to generate all pairs of values which resulted in nine combina-

[1]Each combination took approximately 5 hours to complete.

Table 5.4: Ant System Parameter Tuning - Largest mean in bold

		Statements		
α	q_0	**TerpSpreadsheet**	**TerpWord**	**ArgoUML**
0.3	0.3	**3711**	**6286**	20011
0.3	0.6	3627	5750	19578
0.3	0.9	3526	5893	19075
0.6	0.3	3666	6107	**20264**
0.6	0.6	3598	5767	20041
0.6	0.9	3592	5823	18982
0.9	0.3	3563	5715	19807
0.9	0.6	3508	5564	19096
0.9	0.9	3444	5511	18338

tions. The results of the experiments can be seen in Table 5.5, which shows the average code coverage for each combination of parameters. Both TerpSpreadsheet and TerpWord benefited from high α, γ, and q_0 values. ArgoUML, on the other hand, had the best results with $\alpha = 0.3$, $\gamma = 0.6$, and $q_0 = 0.3$.

For each application we performed a factorial ANOVA in order to determine if any single parameter or combination of parameters resulted in significantly different means. In all three cases we found that there were no significant differences between any combination of parameter values.

Following the ANOVA tests, we also performed a Tukey test on all three applications. For both TerpSpreadsheet and TerpWord, we found that there was no significant difference between any levels for each factor. The results for ArgoUML, however, did result in significant differences. For the α values, the Tukey test showed that a value of 0.3 (M = 19941.27, SD = 502.59) was significantly better than a value of 0.6 (M = 19553.13, SD = 325.25). For the γ value, the Tukey test showed that a value of 0.9 (M = 20068.6, SD = 460.03) was significantly better than both 0.3 (M = 19491.53, SD = 291.49) and 0.6 (M = 19643.27, SD = 449.82). Finally, the q_0 results showed that a value of 0.3 (M = 19905.6, SD = 342.99) was significantly better than a value of 0.9 (M = 19573.13, SD = 468.79).

Table 5.5: AntQ Parameter Tuning - Largest mean in bold

α	γ	q_0	Statements		
			TerpSpreadsheet	**TerpWord**	**ArgoUML**
0.3	0.3	0.9	3801	6348	19395
0.3	0.6	0.3	3788	6458	20170
0.3	0.9	0.6	3791	6423	**20258**
0.6	0.3	0.6	3814	6416	19450
0.6	0.6	0.9	3792	6304	19293
0.6	0.9	0.3	3774	6412	19916
0.9	0.3	0.3	3801	6230	19629
0.9	0.6	0.6	3765	6248	19466
0.9	0.9	0.9	**3833**	**6471**	20031

In summary, we found that no individual parameter had any significant effect on the code coverage across all applications. Furthermore, for both TerpSpreadsheet and TerpWord, no level of any factor had a signficant effect. For ArgoUML, the Tukey tests did show that some levels of some factors had significantly better results. Since ArgoUML has a more complex interface consisting of many widgets, it is likely that the tests are more sensitive to different parameter values, whereas both TerpSpreadseet and TerpWord have simple interfaces that are less sensative to changes in the parameter values. As well, it could be that the combinations of values not chosen may have had a significant effect had we tested all 27 combinations.

5.6 Summary

In this section we looked at the architecture of our system and described each component in detail. We then discussed our seven algorithms. First we described the two random algorithms. The first algorithm makes selections completely randomly, whereas the second algorithm makes selections based on the minimum edge values. Next we described our two greedy algorithms. The first algorithm attempts to maximize event-pair coverage, whereas the second greedy algorithm attempts to reduce the distance between events. We then discussed our Q-Learning algorithm, which was borrowed from the literature. We described its selection pro-

cess and its fitness function, which uses state change ratios to determine good actions. Finally we discussed our two ant algorithms. The first is based on the original ACO system, and the second uses antq. We discussed the differences between the two algorithms and discussed their selection process in detail. We also ran experiments for the Q-Learning and ant algorithms in order to determine good values for their tuning paramaters.

Chapter 6

Comparing Test Generators

In this section we compare the seven test generators discussed in Chapter 5. We run experiments and analyze the code coverage, event metrics, error metrics, and cost. Furthermore, we analyze the effectiveness of the fitness function used by the Q-Learning and ant colony algorithms.

6.1 Experiment Setup

We ran experiments on six applications: ArgoUML, Buddi, Gantt Project, TerpSpreadsheet, TerpWord, and TimeSlotTracker. The details of the applications can be seen in Table 6.1. We stopped using FreeMind as it required a lot of mouse movement input, which our system does not currently support, and it was difficult to achieve high levels of coverage. We instead added two new applications, ArgoUML and Buddi, both of which have appeared in the literature [41] [62]. We ran each algorithm six times on each application. The details of the test suites can be seen in Table 6.2. We used two ϵ values for Q-Learning: 0.4 and 0.9. Based on the parameter tuning (see Section 5.4.2), we determined that the simpler applications benefited from more random selection.

The goal of the analyses is to determine if any algorithm is superior than the others, in general. Therefore, when we analyze the data, we compare the combined data from all applications

Table 6.1: Application Summary

Application	Application Area	Classes	Lines of Code	Branches
ArgoUML	UML designer	1233	54632	24074
Buddi	Budgeting software	1529	101949	43670
Gantt Project	Project management	689	27804	8926
TerpSpreadSheet	Spreadsheet	137	5449	2135
TerpWord	Word processor	208	10340	3625
TimeSlotTracker	Task organizer	487	10090	3115
Total		4283	210264	85545

rather than comparing each application separately. However, the applications all have different values for key metrics, such as number of lines of code and number of uncaught exceptions. In order to combine the data, we need to normalize the values so that no single application has a greater effect on the results than any other application, as the results are on different scales. To scale the data, we use unity-based normalization (also called feature scaling) [49] as shown in the following equation:

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{6.1}$$

where X is the measured value and X_{max} and X_{min} are the maximum and minimum values of the metric for the application, across all runs of all algorithms.

When analyzing the data, we either performed a Welch test followed by a Games-Howell post-hoc test, or we performed an ANOVA paired with a Tukey test [54]. The Welch test and ANOVA are used to analyze the differences within group means. The results generated from these tests only state whether or not there is a significant difference between the group means. The Games-Howell and Tukey post-hoc tests are used to determine which factors have significantly different means among the group. In the cases where the assumption of homogeneity of variances was violated, we used the Welch and Games-Howell tests. Otherwise we used the ANOVA and Tukey test.

Table 6.2: Test Suite Details

Algorithm	α	γ	q_0	**Ants**	**k-best**	**Length**	**Size**
Ant System	0.3	-	0.3	20	15	30	300
AntQ	0.3	0.9	0.3	20	15	30	300
Distance	-	-	-	-	-	30	300
EPG	-	-	-	-	-	30	300
Q-Learning	-	0.9	0.4/0.9	-	-	30	300
Random	-	-	-	-	-	30	300
Random Min	-	-	-	-	-	30	300

6.2 Code Coverage

In this section we analyze both the statement and branch coverage results for each application. We plot the results and run Welch tests and Games-Howell post-hoc tests to determine if any algorithm results in significantly more coverage.

6.2.1 Statement Coverage

The statement coverage results are shown in Table 6.3. In the case of ArgoUML and Buddi, the distance reduction algorithm performed the best. In the case of Gantt Project, TerpSpreadsheet, TerpWord, and TimeSlotTracker, the event pair greedy algorithm performed the best. Figure 6.1 shows a box plot for the scaled coverage results. It should be noted that the results of Buddi seem low compared to the number of statements reported in Table 6.1. However, the application includes all of its third-party library code in its single JAR file. Much of the third-party code is not used and does not contribute to the actual application.

We performed a Welch test and Games-Howell post-hoc test to determine if there were any significant differences between the seven algorithms. The Welch test showed that there was a significant difference in statement coverage at the $p < 0.05$ level. The Games-Howell test showed that both the distance reduction greedy ($M = 0.73$, $SD = 0.24$) and event-pair greedy ($M = 0.79$, $SD = 0.19$) algorithms performed significantly better than all other algorithms: ant

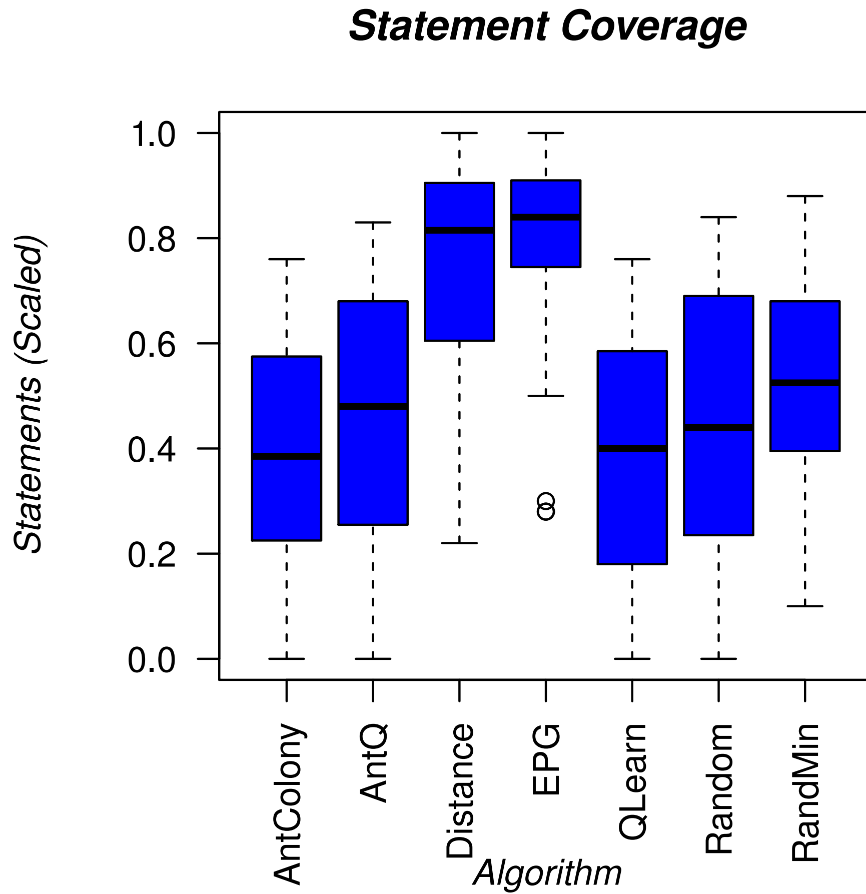

Figure 6.1: Statement Coverage

colony (M = 0.39, SD = 0.22), antq (M = 0.46, SD = 0.24), Q-Learning (M = 0.39, SD = 0.23), random (M = 0.44, SD = 0.26), and random minimum (M = 0.55, SD = 0.19). Furthermore, random minimum performed significantly better than ant colony.

Figures 6.2 to 6.7 show the increase in coverage over time for each algorithm. In all cases the event pair greedy algorithm dominates all other alogorithms.

6.2.2 Branch Coverage

The branch coverage results are shown in Table 6.4. As with the statement coverage results, both the distance reduction greedy and event-pair greedy algorithms perfomed the best in all cases. Figure 6.8 shows a box plot for the scaled coverage results.

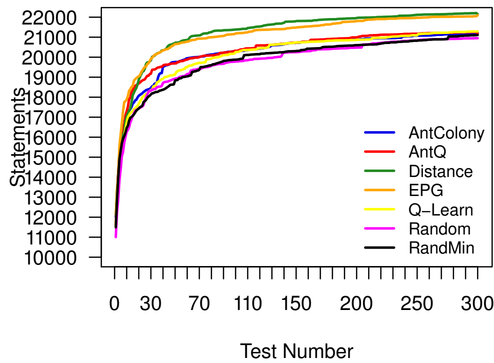

Figure 6.2: ArgoUML Cov. Increase

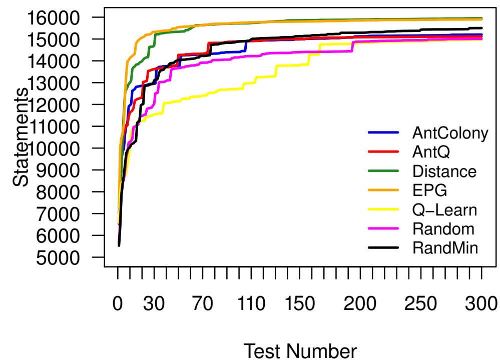

Figure 6.3: Buddi Cov. Increase

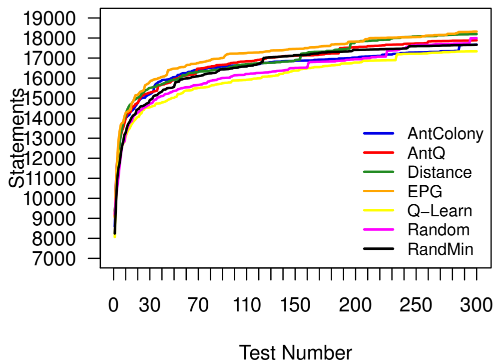

Figure 6.4: Gantt Project Cov. Increase

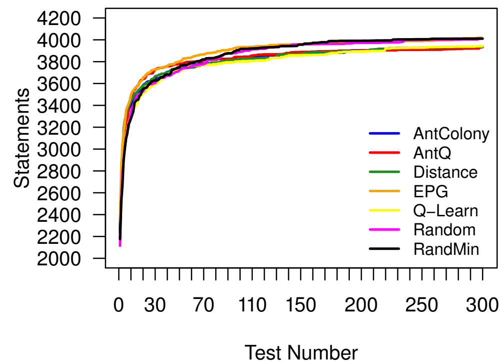

Figure 6.5: TerpS. Cov. Increase

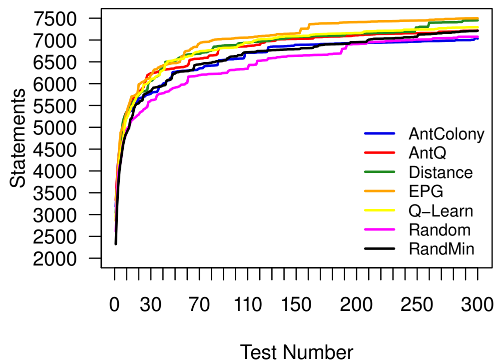

Figure 6.6: TerpWord Cov. Increase

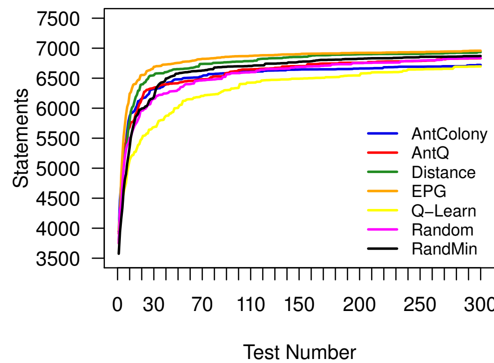

Figure 6.7: TimeSlot Cov. Increase

Table 6.3: Statement Coverage

	Algorithm						
Application	**Ant**	**AntQ**	**Distance**	**EPG**	**Q-Learn**	**Rand**	**RandMin**
ArgoUML	21208	21278	**22205**	22063	21289	20950	21108
Buddi	15204	15122	**15948**	15897	14978	15008	15501
GanttProj	17674	17888	18195	**18322**	17336	17992	17661
TerpSpread	3933	3934	3936	**4019**	3941	4010	4011
TerpWord	7046	7222	7451	**7498**	7293	7076	7213
TimeSlot	6721	6832	6936	**6958**	6696	6837	6870
Total	71786	72276	74671	**74756**	71533	71874	72363

Table 6.4: Branch Coverage

	Algorithm						
Application	**Ant**	**AntQ**	**Distance**	**EPG**	**Q-Learn**	**Rand**	**RandMin**
ArgoUML	5332	5349	**5721**	5656	5399	5211	5277
Buddi	3624	3586	**3884**	3836	3520	3545	3698
GanttProj	3593	3595	3700	**3741**	3586	3586	3676
TerpSpread	1242	1239	1255	**1292**	1257	1282	1276
TerpWord	1942	1987	2034	**2056**	2015	1934	1966
TimeSlot	1371	1416	1489	**1504**	1341	1423	1440
Total	17104	17172	18083	**18085**	17118	16981	17333

The results of the Welch test and Games-Howell post-hoc test follow the same pattern as with the statement coverage. The Welch test showed that there was a significant difference in means at the $p < 0.05$ level. The Games-Howell test showed that both the distance reduction (M = 0.71, SD = 0.21) and event-pair greedy (M = 0.77, SD = 0.14) algorithms performed significantly better than all other algorithms: ant colony (M = 0.34, SD = 0.17), antq (M = 0.38, SD = 0.19), Q-Learning (M = 0.37, SD = 0.24), random (M = 0.36, SD = 0.24), and random minimum (M = 0.49, SD = 0.2). Random minimum also performed significantly better than ant colony.

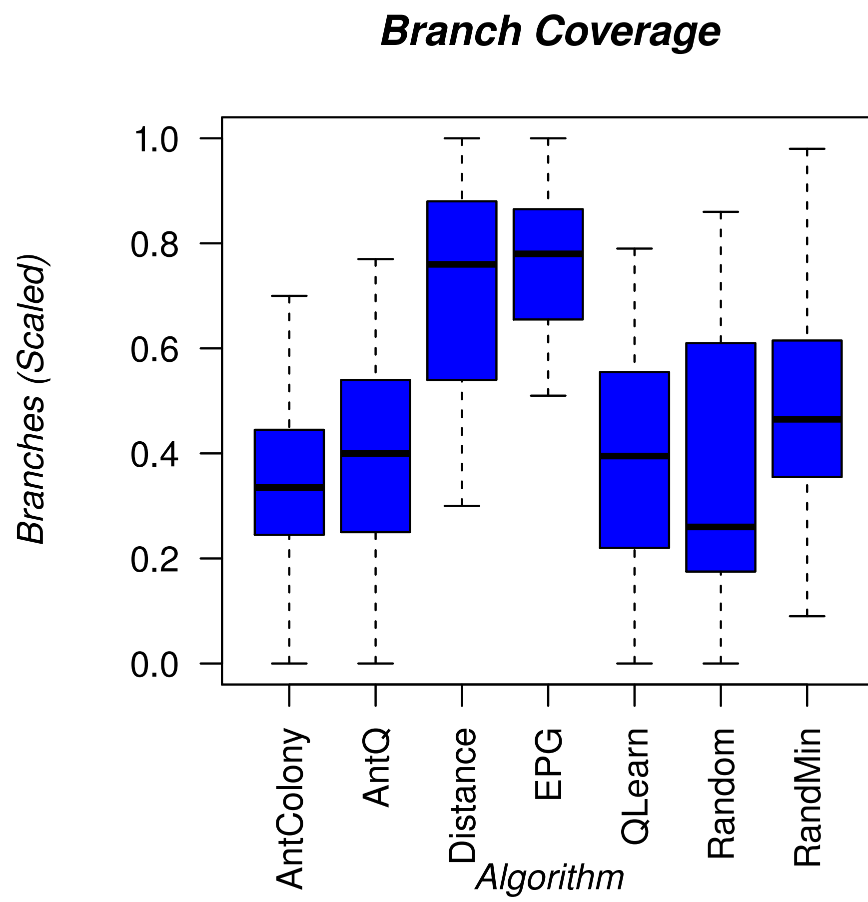

Figure 6.8: Branch Coverage

Table 6.5: Event Coverage

	Algorithm						
Application	**Ant**	**AntQ**	**Distance**	**EPG**	**Q-Learn**	**Rand**	**RandMin**
ArgoUML	387	384	718	**739**	384	412	434
Buddi	227	221	**387**	386	189	196	228
GanttProj	407	395	**817**	785	370	417	468
TerpSpread	92	92	97	111	96	153	**162**
TerpWord	282	284	**398**	396	289	318	344
TimeSlot	191	186	197	**199**	156	184	191
Total	1586	1562	2614	**2616**	1484	1680	1827

6.3 Event Metrics

In this section we look at three event metrics. First we look at the number of events each algorithm covered. Second, we analyze the number of event pairs covered by each algorithm, regardless of the pair's distance. Finally, we look at the number of event pairs covered with length one (that is, events that were executed one after the other). We call this event-1-pair coverage.

6.3.1 Event Coverage

We measured the event coverage by counting the number of events that appear in the event-pair graph. The summary of events for each application can be seen in Table 6.5. The box plot showing the scaled data is shown in Figure 6.9.

The Welch test showed that there was a significant difference in means at the $p < 0.05$ level. The Games-Howell test showed that both the distance reduction greedy (M = 0.69, SD = 0.31) and event-pair greedy (M = 0.73, SD = 0.27) algorithms performed signficantly better than all other algorithms: ant colony (M = 0.19, SD = 0.23), antq (M = 0.19, SD = 0.25), Q-Learning (M = 0.1, SD = 0.09), random (M = 0.36, SD = 0.29), and random minimum (M = 0.47, SD = 0.31). Random minimum also performed significantly better than both ant algorithms and Q-Learning. Finally, the random algorithm performed significantly better than Q-Learning.

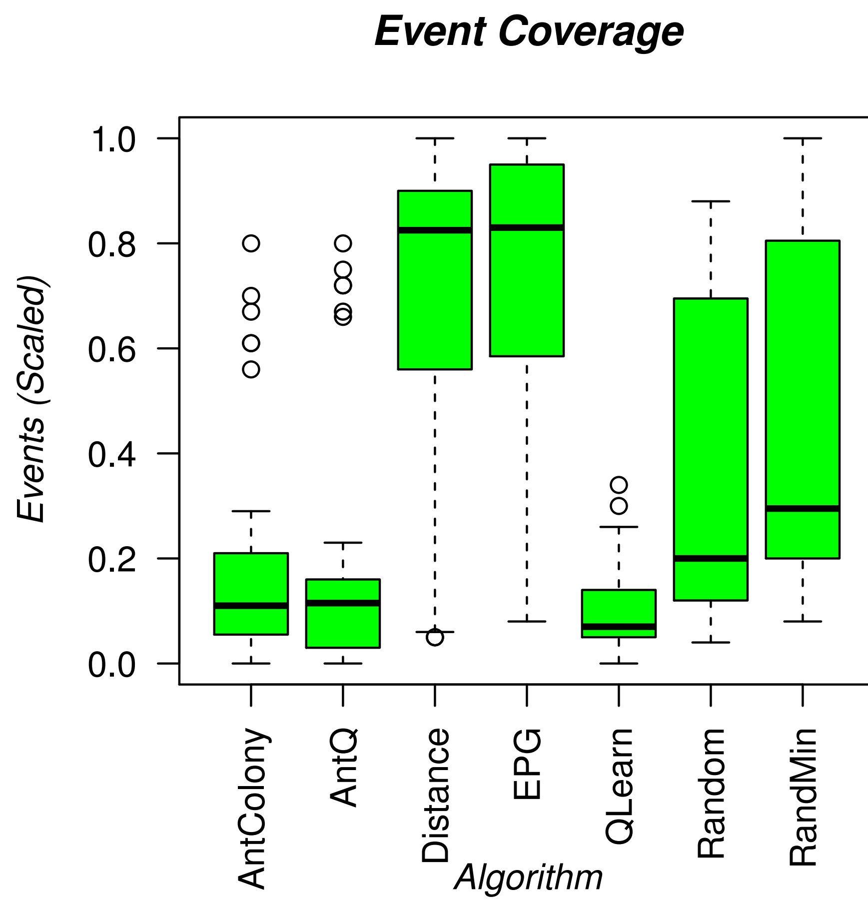

Figure 6.9: Event Coverage

Table 6.6: Event-pair Coverage

	Algorithm						
Application	**Ant**	**AntQ**	**Distance**	**EPG**	**Q-Learn**	**Rand**	**RandMin**
ArgoUML	44650	45196	71011	**76001**	39001	40139	42576
Buddi	15749	14442	**28368**	26926	11410	10291	15092
GanttProj	28861	30273	44733	**44867**	21888	25925	31046
TerpSpread	7379	7266	8586	9592	7834	10876	**11911**
TerpWord	32165	34230	47863	**48040**	34787	30019	33295
TimeSlot	13826	15001	**20351**	20157	9091	12013	14642
Total	142630	146408	220912	**225583**	124011	129263	148562

6.3.2 Event-pair Coverage

We measured the event-pair coverage by counting the number of edges that exist within the event-pair graph. The results can be seen in Table 6.6. As well, a box plot showing the scaled results is shown in Figure 6.10.

The results of the Welch test showed that there was a significant difference in means at the $p < 0.05$ level. The Games-Howell test showed that both the distance reduction greedy (M = 0.83, SD = 0.24) and event-pair greedy (M = 0.87, SD = 0.19) algorithms performed significantly better than the other algorithms: ant colony (M = 0.25, SD = 0.15), antq (M = 0.28, SD = 0.16), Q-Learning (M = 0.12, SD = 0.1), random (M = 0.24, SD = 0.26), and random minimum (M = 0.42, SD = 0.27). Random minimum also performed significantly better than ant colony and Q-Learning. Finally, both ant algorithms performed significantly better than Q-Learning.

6.3.3 Event-1-pair Coverage

We measured event-1-pair coverage by counting all edges with a distance of one in the event-pair graphs. Table 6.7 shows the results for each algorithm on each application. Furthermore, Figure 6.11 shows the box plot of the scaled results.

The results of the Welch test showed that there was a significant difference in means at the p

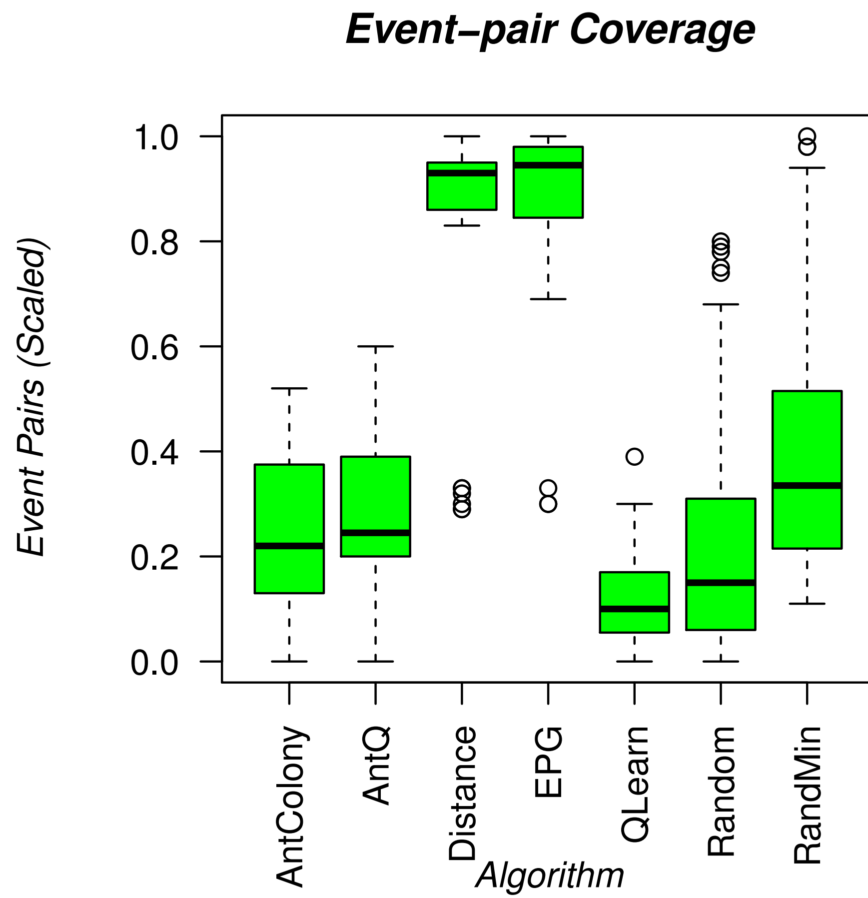

Figure 6.10: Event-pair Coverage

Table 6.7: Event-1-pair Coverage

Application	Algorithm						
	Ant	**AntQ**	**Distance**	**EPG**	**Q-Learn**	**Rand**	**RandMin**
ArgoUML	4572	4592	6092	**6104**	3009	5045	5913
Buddi	2468	2199	**4359**	3714	1694	1856	3472
GanttProj	2443	2859	4216	4032	2156	3313	**4923**
TerpSpread	1792	1784	2839	2549	2194	2334	**3132**
TerpWord	2753	3396	4883	4585	4029	3942	**5326**
TimeSlot	1381	1725	2492	2240	1036	1839	**2688**
Total	15409	16555	24881	23224	14118	18329	**25454**

< 0.05 level. The Games-Howell test showed that the distance reduction greedy (M = 0.85, SD = 0.09) and random minimum (M = 0.9, SD = 0.12) algorithms performed significantly better than all other algorithms: event-pair greedy (M = 0.74, SD = 0.13), ant colony (M = 0.25, SD = 0.2), antq (M = 0.32, SD = 0.16), Q-Learning (M = 0.19, SD = 0.2, and random (M = 0.45, SD = 0.18). The event-pair reduction algorithm performed significantly better than the remaining four algorithms. Lastly, the random algorithm performed significantly better than the two ant algorithms and Q-Learning.

6.4 Error Metrics

We counted the number of unique uncaught exceptions for each run of each algorithm on each application. Table 6.8 shows the average and total number of exceptions found across all six runs. Figure 6.12 shows a box plot of the scaled number of exceptions found.

We performed an ANOVA and Tukey post-hoc test on the results. The ANOVA showed that there was a significant effect at the $p < 0.05$ level [$f(6, 161) = 3.89$, $p = 0.001$]. The Tukey test showed that the distance reduction greedy (M = 0.53, SD = 0.28) algorithm found a significantly higher number of exceptions than random (M = 0.31, SD = 0.23) and random minimum (M = 0.29, SD = 0.18). As well, the event-pair greedy (M = 0.5, SD = 0.2) algorithm found a significantly higher number of exceptions than random minimum. There were no

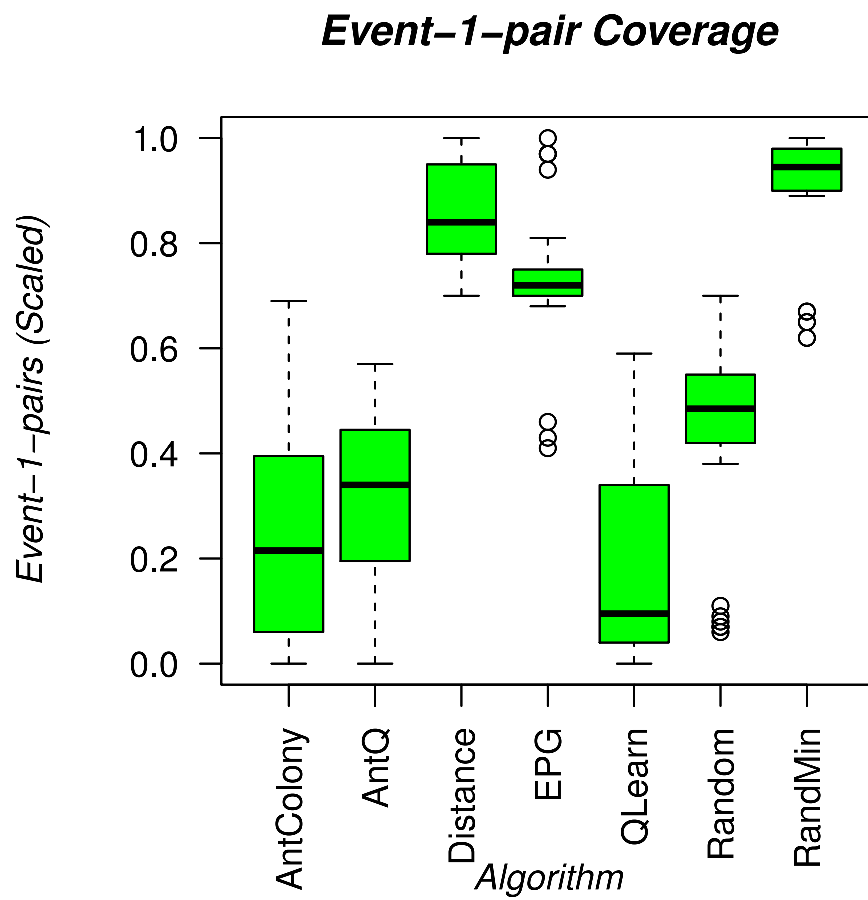

Figure 6.11: Event-1-pair Coverage

Table 6.8: Unique Uncaught Exceptions - Average (Total)

	Algorithm						
Application	**Ant**	**AntQ**	**Distance**	**EPG**	**Q-Learn**	**Rand**	**RandMin**
ArgoUML	10(19)	10(18)	**16(26)**	15(25)	10(20)	9(19)	10(18)
Buddi	0	0	0	0	0	0	0
GanttProj	0	0	0	0	0	0	0
TerpSpread	6(8)	6(8)	6(10)	7(8)	**7(15)**	6(10)	6(7)
TerpWord	11(16)	**12(21)**	11(15)	10(15)	11(15)	10(13)	9(14)
TimeSlot	4(9)	4(8)	**5(9)**	**5(9)**	3(7)	4(7)	4(7)
Total	31(52)	32 (55)	**38(60)**	37(57)	31(57)	29(49)	29(46)

significant differences between any other pairs of algorithms.

6.5 Cost

We measured the cost of an algorithm as its running time. The average running time for each algorithm on each application can be seen in Table 6.9. As well, Figure 6.13 shows the scaled running time in hours for each run across all applications. We can see that the two random algorithms performed the best, followed by the two greedy algorithms. The two ant algorithms performed moderately well, whereas Q-Learning performed the worst. Since part of the Q-Learning algorithms involves running steps from previous tests in order to place the system in a specific state, it follows that the algorithm would take longer to run.

The Welch test showed that there was a significant difference in means at the $p < 0.05$ level. The Games-Howell test showed that random ($M = 0.12$, $SD = 0.08$) and random minimum ($M = 0.11$, $SD = 0.13$) both performed significantly faster than the other five algorithms: ant colony ($M = 0.51$, $SD = 0.2$), antq ($M = 0.52$, $SD = 0.2$), distance reduction greedy ($M = 0.3$, $SD = 0.13$), event-pair greedy ($M = 0.3$, $SD = 0.14$), and Q-Learning ($M = 0.83$, $SD = 0.14$). The two greedy algorithms performed significantly better than the two ant algorithms and Q-Learning. Finally, the two ant algorithms performed significantly better than Q-Learning.

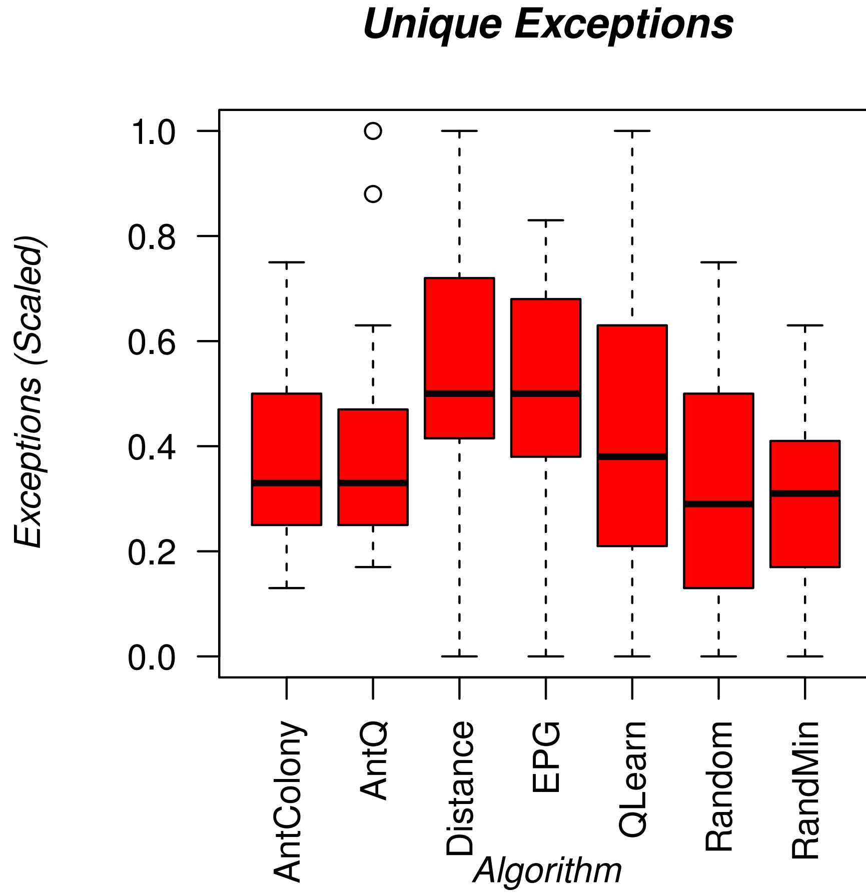

Figure 6.12: Unique Uncaught Exceptions

Table 6.9: Cost (Hours)

Application	Algorithm						
	Ant	**AntQ**	**Distance**	**EPG**	**Q-Learn**	**Rand**	**RandMin**
ArgoUML	5.56	5.63	5.91	5.85	6.11	**5.52**	5.56
Buddi	5.4	5.22	4.26	4.27	5.77	**3.96**	4.03
GanttProj	4.5	4.7	4.07	4.05	4.75	3.97	**3.8**
TerpSpread	3.92	3.89	3.7	3.83	4.51	**3.48**	3.49
TerpWord	4.13	3.7	3.55	3.51	4.15	3.38	**3.32**
TimeSlot	4.61	4.62	4.16	4.10	4.85	**3.94**	3.98
Total	28.12	27.77	25.65	25.61	30.15	24.25	**24.18**

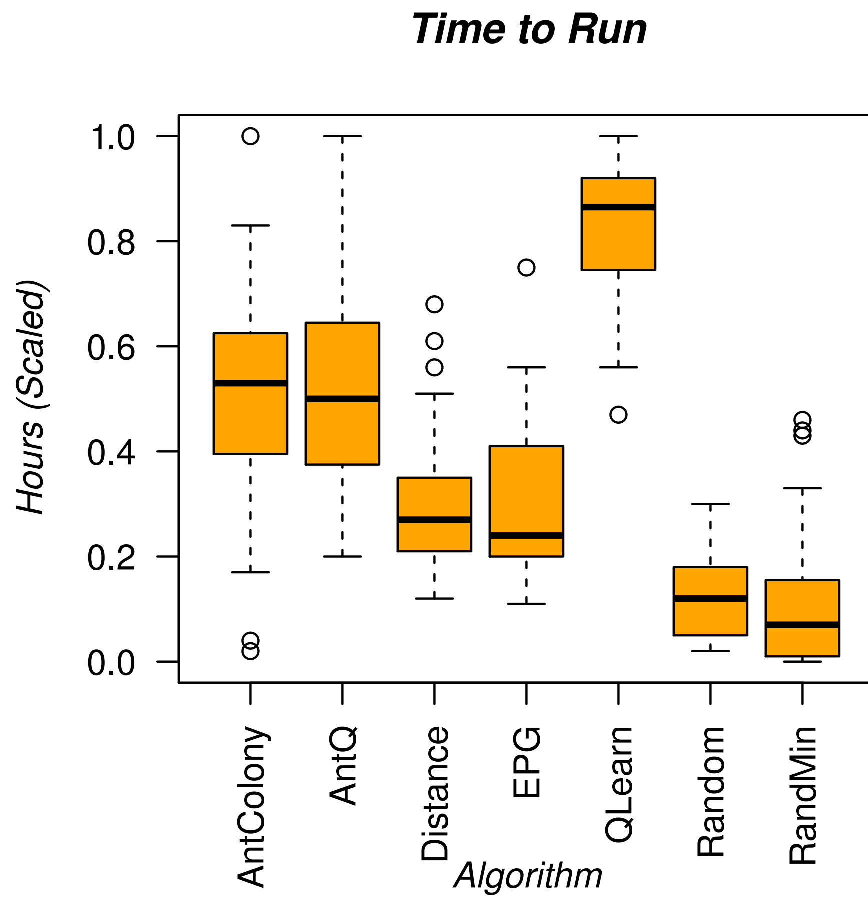

Figure 6.13: Cost

Table 6.10: Fitness Function Evaluation

Application	Pearson Value
ArgoUML	0.414
Buddi	-0.026
Gantt Project	0.339
TerpSpreadsheet	-0.213
TerpWord	0.228
TimeSlotTracker	0.497

6.6 Fitness Function Evaluation

In this section we aim to determine if there is a relationship between our antq fitness function and code coverage. We captured the fitness function data and the statement coverage data for each test case from the antq algorithm. We measured the Pearson correlation; the results are shown in Table 6.10. Figures 6.14 to 6.19 show the accompanying scatter plots.

ArgoUML, Gantt Project, and TimeSlotTracker had moderate to strong positive correlations. TerpWord had a weak positive correlation. Finally, Buddi and TerpSpreadsheet had weak negative correlations. The results show that there is a relationship between the state change ratio of a test and its statement coverage; however, it is application-dependent. Events that open windows will have the greatest impact on the state change ratio and will be favoured for selection. Therefore, applications with many useful windows will likely show a strong positive correlation between fitness and coverage. The applications that had strong positive correlations had multiple windows that affect the application's operation. TerpSpreadsheet, which did the poorest, has few windows. The windows it does have are used for open and save operations, font selection, and border selection for the cells. Since events that open windows will be favoured over other events as they have a larger impact on the fitness, applications with less useful windows will likely have a negative correlation between fitness and coverage.

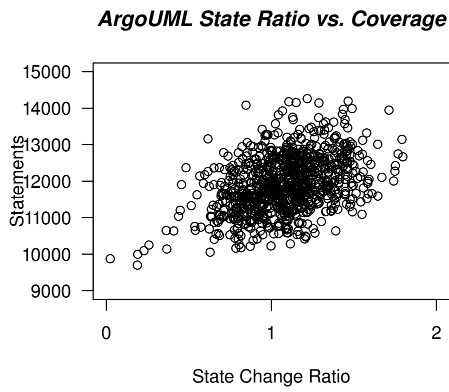

Figure 6.14: ArgoUML Scatter

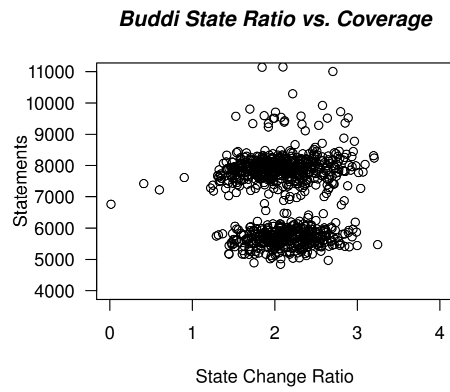

Figure 6.15: Buddi Scatter

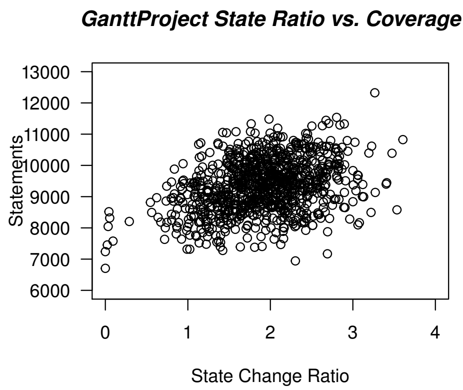

Figure 6.16: Gantt Project Scatter

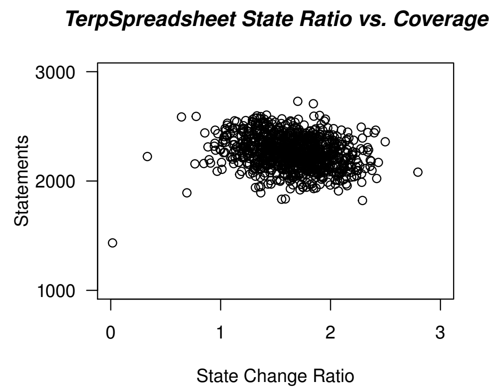

Figure 6.17: TerpSpreadsheet Scatter

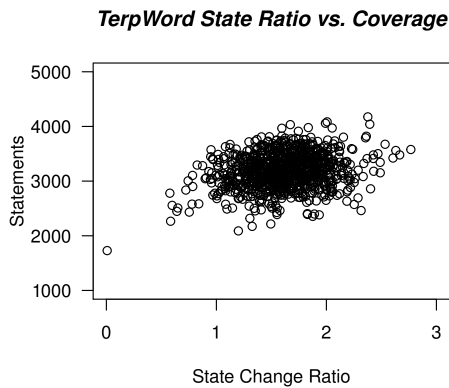

Figure 6.18: TerpWord Scatter

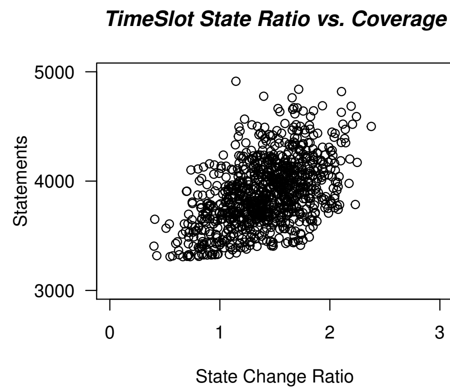

Figure 6.19: TimeSlot Scatter

6.7 Lessons Learned

The following are the lessons learned based on the results in this chapter.

1. We have discovered that the simpler greedy selection algorithms performed the best in terms of coverage and faults found. It appears that there is a relationship between event-pair coverage and code coverage, and therefore, since the greedy algorithms attempt to maximize evet-pair coverage, they performed better.

2. We also found that a large difference in code coverage does not correlate to a large difference in faults found. For example, though Q-Learning performed poorly in terms of coverage, it still found as many faults as the other algorithms.

3. Based on the results, it appears that event-1-pair coverage does not work well as a factor in test generation. The random minimum algorithm achieved high levels of event-1-pair coverage, but it did not achieve high levels of code coverage nor of faults found.

4. Looking at the results of the two ant algorithms, it appears that they only performed moderately well. They were not able to achieve high levels of event, event-pair, or event-1-pair coverage. This may be due to the fitness function used. Basing tests on the amount of state change may end up favouring the same few events, such as those that open windows. Though the algorithms did not perform as well as we had hoped, we believe they can be improved by using different fitness functions.

5. The analysis of the fitness function shows that it has a positive correlation in some instances but not in others. The functions' effectiveness depends heavily on the application under test, and therefore it does not make for a good general-purpose fitness.

6.8 Summary

To summarize, both of the greedy algorithms performed best in terms of statement, branch, event, and event-pair coverage. The distance reduction greedy algorithm found the most faults. The two random algorithms cost the least to run, with the random minimum algorithm performing the best. In the next section we investigate whether or not there is a relationship between any of the coverage metrics and the number of faults found.

Chapter 7

Test Adequacy Criteria

In this chapter we attempt to find relationships between the coverage measures and the faults found by the testing techniques we studied. As well, we attempt to find relationships between the GUI coverage measures and the statement and branch coverage measures. We use multiple regression analysis in order to determine if any of the independent measures can be used to predict the dependent values.

7.1 Results Analysis

We used the results from the previous experiments as shown in Chapter 6 in our multiple regression analysis. For all analyses, we used the normalized data and not the raw data.

7.1.1 Coverage Analysis

Our first set of analyses looks at the relationship between the GUI metrics and the statement and branch metrics. We want to know if it is possible to predict the statement and branch coverage values based on the GUI coverage values. We ran analyses for both statement and branch coverage and report on them separately. Figures 7.1 to 7.6 show scatter plots that compare the statement and branch coverage results against the events, event pairs, and event-1-pairs results.

The straight line in all scatter plots represents the regression line (line of best fit). For both statement and branch coverage, it appears that the number of events and event-pairs covered may have a a positive effect. As well, it appears that event-1-pair coverage does not have any positive effect on the statement and branch coverage values.

Statement Coverage

We ran a multiple linear regression with statement coverage as the dependent variable and event, event pair, and event-1-pair coverage as the independent variables. We used a stepwise regression, which works by adding the most significant variable to the model first, followed by the next most significant, and so on. Variables that are not significant are excluded from the model and are not used as predictors.

The results of the analysis showed that the variables event coverage and event-pair coverage are statistically significant in predicting the statement coverage, $F(2, 165) = 112.019$, $p < 0.001$, $R^2 = 0.576$[1]. Both variables added statistically significantly to the prediction, $p < 0.05$. Event-1-pair coverage was not found to be significant and was not included in the model.

The general from of the equation to predict statement coverage is as follows:

$$StatementCoverage = 0.369 + 0.338(\text{Event-Pair Coverage}) + 0.166(\text{Event Coverage}) \quad (7.1)$$

The R^2 values shows that 57.6% of the variation in coverage can be explained by both event-pair coverage and event coverage, though moreso by the former. Therefore, tests should aim to cover as many event pairs and events as possible in order to achieve higher levels of coverage.

[1] R^2 represents the percentage of variation explained by the model.

Branch Coverage

We performed a similar analysis for branch coverage as we did for statement coverage. The results showed that only event-pair coverage was statistically significant in predicting branch coverage, $F(1, 166) = 199.566$, $p < 0.001$, $R^2 = 0.546$. The variable added statistically significantly to the prediction, $p < 0.05$. Both event coverage and event-1-pair coverage were excluded from the model for not adding significantly to the prediction.

The general form of the equation to predict branch coverage is as follows:

$$BranchCoverage = 0.313 + 0.505(\text{Event-Pair Coverage}) \tag{7.2}$$

The R^2 value shows that 54.6% of the variation in coverage can be explained by event-pair coverage. Therefore, tests should aim to cover as many event pairs as possible in order to increase branch coverage.

7.1.2 Exception Analysis

Figures 7.7 to 7.11 show the scatter plots comparing the normalized number of uncaught exceptions found against each of the five coverage metrics. The plots for statement and branch coverage show that some relationship may exist between the coverage measures and the number of exceptions found. The plots for the GUI coverage metrics suggest that little to no relationship exists between the amount of GUI coverage and the number of exceptions found.

We performed a multiple linear regression analysis using the number of exceptions found as the dependent variable and the five coverage metrics as the independent variables using stepwise regression. The results showed that only branch coverage was statistically significant in predicting errors, $F(1, 166) = 14.929$, $p < 0.001$, $R^2 = 0.083$. The variable added statistically significantly to the prediction, $p < 0.05$. All other variables were excluded from the model as they did not add significantly to the prediction.

The general form of the equation to predict errors is as follows:

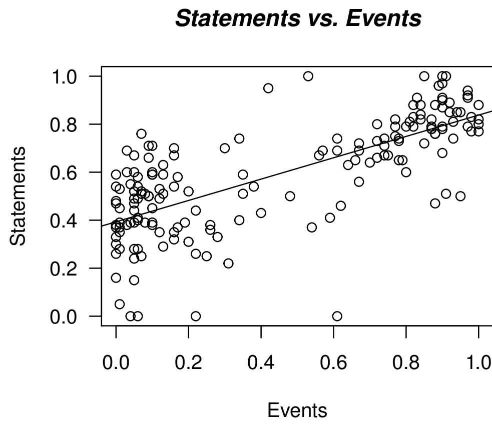

Figure 7.1: Statement vs. Events

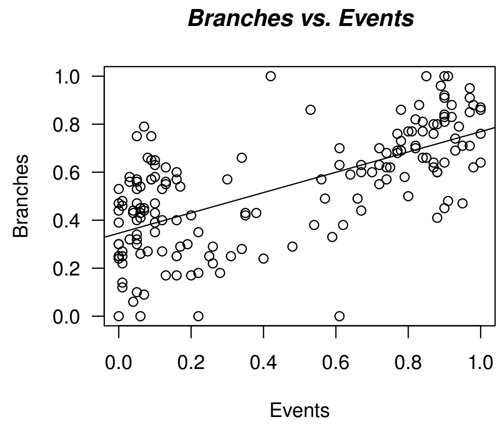

Figure 7.2: Branch vs. Events

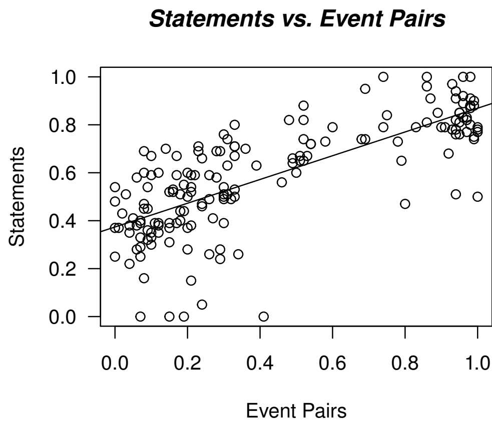

Figure 7.3: Statement vs. EPs

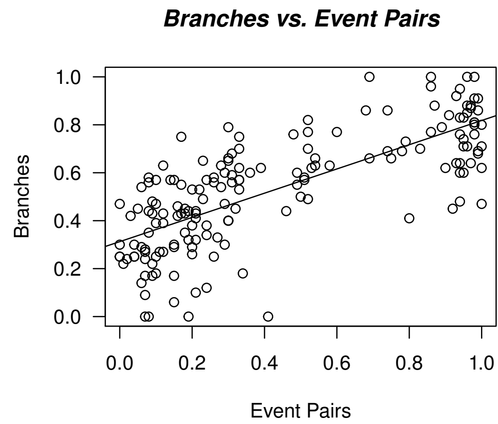

Figure 7.4: Branch vs. EPs

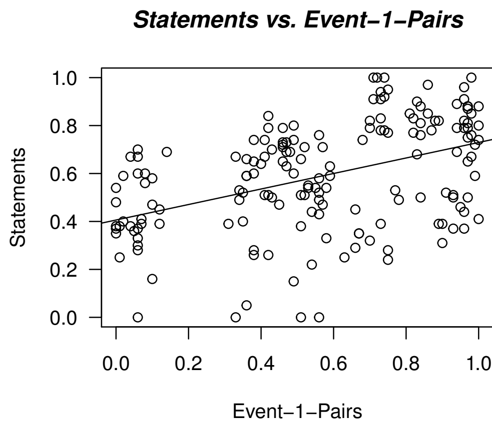

Figure 7.5: Statement vs. E1Ps

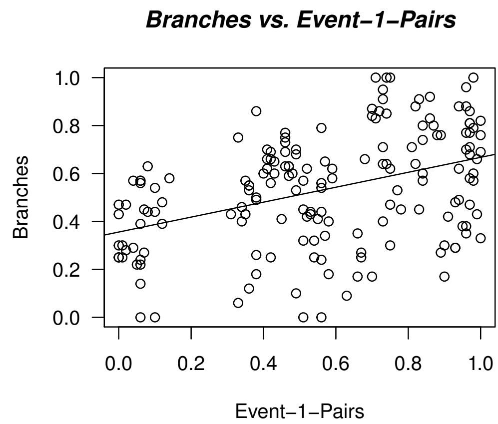

Figure 7.6: Branch vs. E1Ps

$$Exceptions = 0.247 + 0.289(\text{Branch Coverage}) \tag{7.3}$$

The R^2 states that only 8.3% of the variation can be explained by the branch coverage. Therefore, much of the variation in the exceptions found is unexplained. This is unsurpirsing as we saw in the previous chapter that Q-Learning did not achieve high levels of coverage but it performed well in terms of exceptions found. There may be other factors that affect the exceptions, such as the inputs used during the test, or the distance between event-pairs. Further study is required to determine good predictors of exceptions, if any.

7.2 Summary

The results of our analyses show that event-pair coverage is a good predictor of both statement and branch coverage. The analysis for the exceptions found show that branch coverage is the best predictor; however, it only explains a small percentage of the variation in the the number of exceptions found. Therefore, further study is required to determine a good predictor for exceptions.

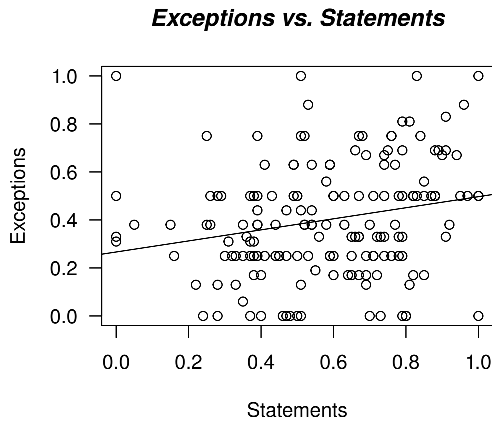

Figure 7.7: Exceptions vs. Statements

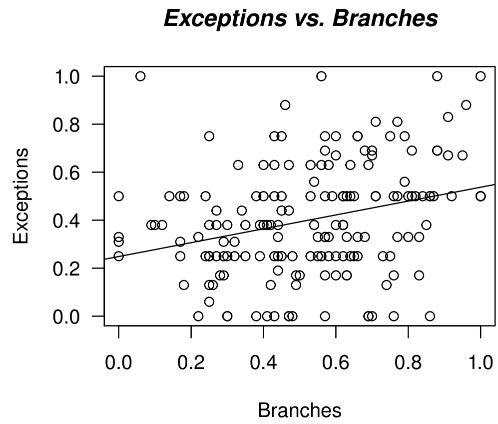

Figure 7.8: Exceptions vs. Branches

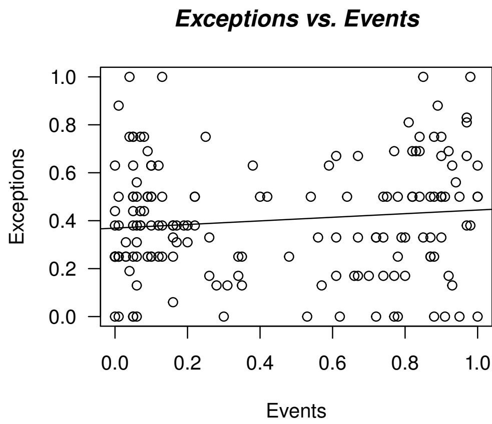

Figure 7.9: Exceptions vs. Events

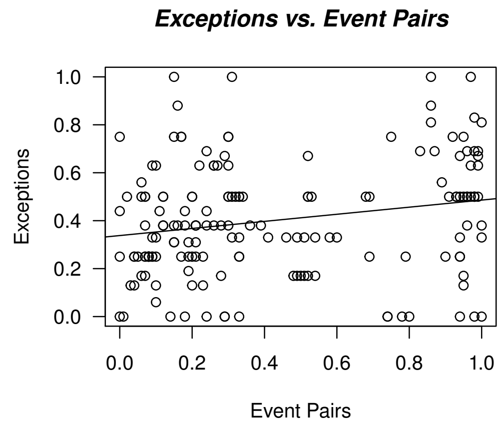

Figure 7.10: Exceptions vs. EPs

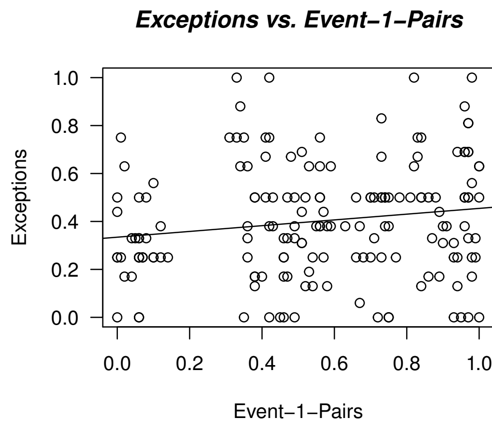

Figure 7.11: Exceptions vs. E1Ps

Chapter 8

Conclusion

In this thesis we investigated dynamic test generation for GUIs. We first introduced static testing systems, such as GUITAR, and showed how they generate infeasible test cases. We then introduced the concept of dynamic testing and discussed how it differs from static testing. We conducted experiments using our own dynamic testing system in order to determine good test lengths, which found that moderate to long tests perform better than short tests. Following that, we discussed seven dynamic testing algorithms and compared their performance on six applications. We found that the greedy selection algorithms performed the best in terms of coverage and faults found. Finally, we investigated test adequacy criteria and found that event-pair coverage is a good predictor of code coverage, and we found that branch coverage is the only predictor of faults found, albeit a weak one.

8.1 Summary of Contributions

The following are the major contributions of this thesis:

1. We created the concept of event-pair graphs and defined their creation and usage. Event-pair graphs can be used to represent pairs of events that have been executed in that order within the same test case. Each event is represented as a vertex and each directed edge

represents the relationship. The weight of the edge can be used to represent such things as the distance between the two events, or any other value the user wants. We showed how event-pair graphs can be used as both an input to a test generator, and as a measure of the quality of a test suite.

2. We conducted a study to determine if short or long test cases perform better. We used the event-pair greedy algorithm and generated test suites of different sizes and lengths. We compared the code coverage and fault finding abilities of each test suite and found that moderate to long tests perform better than short tests.

3. We created and implemented various algorithms for generating tests that consist of GUI events. We first implemented two random algorithms. The first algorithm makes choices completely randomly, whereas the second algorithm always chooses the least used edge. We then implemented two greedy algorithms. The first algorithm, event-pair greedy, attempts to cover as many event pairs as possible. The second algorithm, distance reduction greedy, attempts to reduce the distance between events. Following that, we implemented the Q-Learning algorithm, which attempts to find paths in the GUI that result in large amounts of change in the GUI's state. Finally, we implemented two ant colony algorithms. The first algorithm implemented the normal ant colony algorithm, whereas the second algorithm implemented antq, which combines ant colony optimization and Q-Learning.

4. We conducted a study comparing the seven algorithms on six subject applications. We measured the code coverage, GUI event coverage, run time, and faults found. In terms of the code coverage, GUI event coverage, and faults found, both of the greedy algorithms performed the best. In terms of run time, the two random algorithms performed the best.

5. Using the results from the previous experiments, we performed multiple linear regression analysis in order to determine which, if any, of the coverage metrics can be used as predictors of both code coverage and the faults found. We found that event-pair coverage

was a good predictor for statement and branch coverage. We also found that only branch coverage was a predictor of faults found, but that it only accounted for a small percentage of the variation.

8.2 Future Work

In the future we want to investigate the following:

1. We want to investigate the effect that various scalar inputs (e.g. integers and strings) have on coverage and faults found. Based on our studies, it appears that the event-pair greedy algorithm is sufficient at testing a GUI. However, it may benefit from a more sophisticated input generation algorithm. Currently our system only uses scalar inputs selected from a set of random values from a predefined pool. It may be possible to determine specific scalar inputs to the system that will cover new code or find more faults.

2. We want to investigate whether or not the ant algorithms can be improved upon. It may be possible to improve the algorithms by using different tuning parameters, different generation sizes and k_{best} values, a different fitness function, or by adding a heuristic. We believe the algorithms have a lot of potential and that some of these changes will improve the results.

Bibliography

[1] Osman Abul, Faruk Polat, and Reda Alhajj. Multiagent reinforcement learning using function approximation. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 30(4):485–497, 2000.

[2] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, New York, NY, USA, 2012. ACM.

[3] James H. Andrews, Alex Groce, Melissa Weston, and RuGang Xu. Random test run length and effectiveness. In *Proceedings of the IEEE/ACM Conference on Automated Software Engineering (ASE)*, pages Pages: 19–28, L'Aquila, Italy, September 2008.

[4] Andrea Arcuri. It does matter how you normalise the branch distance in search based software testing. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 205–214. IEEE, 2010.

[5] Andrea Arcuri. Longer is better: On the role of test sequence length in software testing. In *Int'l Conf. on Software Testing, Verification and Validation (ICST 2010)*, pages 469 – 478, April 2010.

[6] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Random testing: Theoretical results and practical implications. *Software Engineering, IEEE Transactions on*, 38(2):258–277, 2012.

[7] Andrea Arcuri and Xin Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008.

[8] Stephan Arlt, Andreas Podelski, Cristiano Bertolini, M Schaf, Ishan Banerjee, and Atif M Memon. Lightweight static analysis for GUI testing. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 301–310. IEEE, 2012.

[9] Gigon Bae, Gregg Rothermel, and Doo-Hwan Bae. On the relative strengths of model-based and dynamic event extraction-based GUI testing techniques: An empirical study. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 181–190. IEEE, 2012.

[10] Sebastian Bauersfeld, Stefan Wappler, and Joachim Wegener. A metaheuristic approach to test sequence generation for applications with a GUI. In *Search Based Software Engineering*, pages 173–187. Springer, 2011.

[11] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[12] Penelope A Brooks and Atif M Memon. Automated GUI testing guided by usage profiles. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 333–342. ACM, 2007.

[13] Penelope A Brooks and Atif M Memon. Introducing a test suite similarity metric for event sequence-based test cases. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 243–252. IEEE, 2009.

[14] Renée C Bryce, Sreedevi Sampath, and Atif M Memon. Developing a single model and test prioritization strategies for event-driven software. *Software Engineering, IEEE Transactions on*, 37(1):48–64, 2011.

[15] Santo Carino and James H. Andrews. Dynamically testing GUIs using ant colony optimization. In *Proceedings of the 2015 International Conference on Automated Software Engineering (ASE'15)*, Lincoln, Nebraska, November 2015.

[16] Santo Carino and James H. Andrews. Evaluating the effect of test case length on GUI test suite performance. In *Automation of Software Test (AST), 2015 10th International Workshop on*. IEEE, 2015.

[17] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499. Springer, 1999.

[18] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: Adaptive random testing for object-oriented software. In *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering (ICSE'08)*, pages 71–80, Leipzig, Germany, May 2008.

[19] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, pages 268–279, Montreal, Canada, September 2000.

[20] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering*, pages 281–290. ACM, 2008.

[21] J-L Deneubourg, Serge Aron, Simon Goss, and Jacques M Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of insect behavior*, 3(2):159–168, 1990.

[22] Marco Dorigo, Mauro Birattari, and Thomas Stützle. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1(4):28–39, 2006.

[23] Marco Dorigo and LM Gambardella. Ant-q: A reinforcement learning approach to the traveling salesman problem. In *Proceedings of ML-95, Twelfth Intern. Conf. on Machine Learning*, pages 252–260, 2014.

[24] Marco Dorigo and Luca Maria Gambardella. A study of some properties of Ant-Q. In *Parallel Problem Solving from NaturePPSN IV*, pages 656–665. Springer, 1996.

[25] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 1(1):53–66, 1997.

[26] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 26(1):29–41, 1996.

[27] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, July 1984.

[28] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[29] Gordon Fraser and Angelo Gargantini. Experiments on the test case length in specification based test case generation. In *Automation of Software Test, 2009. AST'09. ICSE Workshop on*, pages 18–26. IEEE, 2009.

[30] M. P. Gallaher and B. M. Kropp. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. National Institute of Standards, RTI-Health, Social, and Economics Research, May 2002.

[31] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, Chicago, June 2005.

[32] Florian Gross, Gordon Fraser, and Andreas Zeller. Search-based system testing: high coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 67–77. ACM, 2012.

[33] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[34] M. Harman and B.F. Jones. Search-based software engineering. *Journal of Information and Software Technology*, 43:833–839, December 2001.

[35] Si Huang, Myra B. Cohen, and Atif M. Memon. Repairing GUI test suites using a genetic algorithm. In *Int'l Conf. on Software Testing, Verification and Validation (ICST 2010)*, pages 245–254, 2010.

[36] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, pages 191–200. IEEE Computer Society Press, 1994.

[37] Shrinivas Joshi and Alessandro Orso. SCARPE: A technique and tool for selective capture and replay of program executions. In *Intl. Conf. on Software Maintenance (ICSM)*, pages 234–243, 2007.

[38] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[39] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[40] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[41] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. AutoBlack-Test: a tool for automatic black-box testing. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 1013–1015. IEEE, 2011.

[42] Scott McMaster and Atif M Memon. Call-stack coverage for GUI test suite reduction. *Software Engineering, IEEE Transactions on*, 34(1):99–115, 2008.

[43] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. What test oracle should I use for effective GUI testing? In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 164–173. IEEE, 2003.

[44] Atif M Memon. *A comprehensive framework for testing graphical user interfaces*. PhD thesis, University of Pittsburgh, 2001.

[45] Atif M Memon. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[46] Atif M Memon, Martha E Pollack, and Mary Lou Soffa. Using a goal-driven approach to generate test cases for GUIs. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 257–266. IEEE, 1999.

[47] Atif M Memon and Mary Lou Soffa. Regression testing of GUIs. *ACM SIGSOFT Software Engineering Notes*, 28(5):118–127, 2003.

[48] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[49] Ismail Bin Mohamad and Dauda Usman. Standardization and its effects on k-means clustering algorithm. *Res. J. Appl. Sci. Eng. Technol*, 6(17):3299–3303, 2013.

[50] Pablo Moscato et al. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, 826:1989, 1989.

[51] Glenford J. Myers. *The Art of Software Testing*. Wiley, New York, 1979.

[52] BaoN. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, pages 1–41, 2013.

[53] Carlos Pacheco, Shuvendru K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Intl. Conf. on Software Eng. (ICSE)*, pages 75–84, Minneapolis, MN, May 2007.

[54] Julie Pallant. *SPSS survival manual*. McGraw-Hill Education (UK), 2013.

[55] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[56] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 1995.

[57] SeleniumHQ: Web application testing system. http://seleniumhq.org/. Online. Accessed Feb. 2012.

[58] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, Lisbon, September 2005.

[59] Jaymie Strecker and Atif M Memon. Relationships between test suites, faults, and fault detection in GUI testing. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 12–21. IEEE, 2008.

[60] Thomas Stützle and Holger H Hoos. Max–min ant system. *Future generation computer systems*, 16(8):889–914, 2000.

[61] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for. net. In *Tests and Proofs*, pages 134–153. Springer, 2008.

[62] Bart Van Rompaey, Bert Du Bois, and Serge Demeyer. Characterizing the relative significance of a test smell. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 391–400. IEEE, 2006.

[63] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[64] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[65] Qing Xie and Atif Memon. Studying the characteristics of a 'good' GUI test suite. In *Intl. Symp. on Software Reliability Eng. (ISSRE)*, pages 159 –168, November 2006.

[66] Qing Xie and Atif M Memon. Using a pilot study to derive a GUI model for automated testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(2):7, 2008.

[67] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 359–368. IEEE, 2009.

[68] Xun Yuan, Myra Cohen, and Atif M. Memon. GUI interaction testing: Incorporating event context. *IEEE Trans. on Software Eng.*, 37:559–574, July/August 2011.

[69] Xun Yuan and Atif M. Memon. Using GUI run-time state as feedback to generate test cases. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 396–405. IEEE, 2007.

[70] Xun Yuan and Atif M Memon. Generating event sequence-based test cases using GUI runtime state feedback. *Software Engineering, IEEE Transactions on*, 36(1):81–95, 2010.

Curriculum Vitae

Name:	Santo Carino
Post-Secondary Education and Degrees:	University of Western Ontario London, ON 2005 - 2010 B.Sc.
	University of Western Ontario London, ON 2010 - 2012 M.Sc.
	University of Western Ontario London, ON 2012 - 2016 Ph.D.
Honours and Awards:	OGS 2014-2015
Related Work Experience:	Build/Release and Tool Developer RIM 2008 - 2009
	JVM Quality Assurance Engineer IBM Summer 2014
	Teaching Assistant The University of Western Ontario 2010 - 2015

Publications:

Carino, Santo, et al. "BlackHorse: creating smart test cases from brittle recorded tests." Software Quality Journal 22.2 (2014): 293-310.

Carino, Santo, and James H. Andrews. "Evaluating the effect of test case length on GUI test suite performance." Proceedings of the 10th International Workshop on Automation of Software Test. IEEE Press, 2015.

Carino, Santo and James H. Andrews. "Dynamically Testing GUIs Using Ant Colony Optimization." Proceedings of the 30th International Conference on Automated Software Engineering. 2015. (Accepted)