EMPIRICALLY EVALUATING GENETIC ALGORITHMS FOR GENERATING TEST SUITES FOR WEB APPLICATIONS

by

Hammad Ahmad

2019

© 2019 Hammad Ahmad
All Rights Reserved

TABLE OF CONTENTS

LIST OF TABLES . . . . **vi**
LIST OF FIGURES . . . . **vii**
ABSTRACT . . . . **ix**

Chapter

1 INTRODUCTION **1**

2 BACKGROUND **4**

2.1 Web Applications 4
2.2 Testing Web Applications 5
2.3 Genetic Algorithms 7

2.3.1 Model 8
2.3.2 Fitness Functions 9
2.3.3 Parent Selection Algorithms 9
2.3.4 Genetic Operators 10
2.3.5 Applying the Genetic Algorithm to Software Testing 11

2.4 Open Problems 12

2.4.1 Problems in Testing Web Applications 12
2.4.2 Evaluating Web Application Testing Approaches 13

3 GENERATING TEST SUITES FOR WEB APPLICATIONS USING THE GENETIC ALGORITHM **14**

3.1 Motivation 14
3.2 Model 14
3.3 Implementation 15

3.3.1 Initialization 15

3.3.2 Fitness Function . . . 16
3.3.3 Genetic Operators . . . 17
3.3.4 Parent Selection Algorithms . . . 18
3.3.5 Summary . . . 20

4 THE EVALUATION FRAMEWORK . . . 21

4.1 Design . . . 21

4.1.1 Configuring the Application . . . 22
4.1.2 Initializing the Persistent State . . . 23
4.1.3 Instrumenting the Source Code . . . 23
4.1.4 Executing the Test Suites . . . 24

4.2 Implementation . . . 24

4.2.1 Automating the Execution of Test Suites . . . 25

4.2.1.1 Instrumenting the Source Code . . . 25
4.2.1.2 Initializing the Persistent State . . . 25
4.2.1.3 Running the Web Applications Server . . . 26
4.2.1.4 Executing the Test Suites . . . 26

4.2.2 Evaluating the Results . . . 26
4.2.3 Summary . . . 27

5 EXPERIMENTAL STUDY . . . 28

5.1 Research Questions . . . 28
5.2 Subject Applications . . . 29
5.3 Methodology . . . 29

5.3.1 Gathering User Sessions . . . 29
5.3.2 Evaluation and Analysis Metrics . . . 30
5.3.3 Generating Test Suites . . . 30
5.3.4 Analyzing the Resulting Test Suites . . . 34
5.3.5 Evaluating the Resulting Test Suites . . . 34

5.4 Threats to Validity . . . 35

5.5 Results 36

5.5.1 Generating Test Suites 36

5.5.1.1 Comparing Parent Selection Algorithms 37
5.5.1.2 Comparing Genetic Operators and Genetic Algorithm Parameters 39
5.5.1.3 Comparing Fitness Weights 43
5.5.1.4 Comparing the Genetic Algorithm to the Hill Climbing Algorithm for Generating Cost-Effective Test Suites . 44

5.5.2 Evaluating Test Suites 47

5.6 Discussion 49

5.6.1 Comparing Parent Selection Algorithms 50
5.6.2 Comparing Genetic Operators and Genetic Algorithm Parameters 51
5.6.3 Comparing Fitness Weights 52
5.6.4 Evaluating the Performance of the Generated Test Suites . . . 52
5.6.5 Comparing the Genetic Algorithm to the Hill Climbing Algorithm for Generating Cost-Effective Test Suites 53

5.7 Recommendations for Testers 54

6 CONTRIBUTIONS AND FUTURE WORK 56

6.1 Contributions 56
6.2 Future Work 57

BIBLIOGRAPHY 59

LIST OF TABLES

5.1 Subject Application Characteristics 29

5.2 Subject User Session Set Characteristics 29

5.3 Genetic Operator Thresholds 32

5.4 Original Test Suite Coverage 47

5.5 Coverage for Test Suites Generated by the Genetic Algorithm . . . 48

5.6 Coverage for Test Suites Generated by the Hill Climbing Algorithm 48

LIST OF FIGURES

2.1 A Sample HTTP Request from the Ancient Graffiti Project 5

2.2 A Sample User Session from a Google Search 6

2.3 An Overview of the Genetic Algorithm Process 7

2.4 The Model for the Genetic Algorithm 8

3.1 Modeling Web Applications Test Suites for the Genetic Algorithm . 15

3.2 An Overview of the Genetic Algorithm For Generating Test Suites . 16

3.3 An Overview of the Mutation Operator 17

3.4 An Overview of the Variable Length Mutation Operator 17

3.5 An Overview of the One-point Genome-level Crossover Operator . . 19

3.6 An Overview of the Two-point Chromosome-level Crossover Operator 19

4.1 An Overview of the Evaluation Framework 22

5.1 Baseline Graphs for the Generated Test Suites. The x-axis is the generation number. The y-axes for the graphs in the left column are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis), and the y-axis for the graphs in the right column is the test suite's fitness. 36

5.2 A Comparison Parent Selection Algorithms for logic_fall2010. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis). . . . 37

5.3 A Comparison Parent Selection Algorithms for logic_2012_2013. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis). . . . 38

5.4 A Comparison Parent Selection Algorithms for logic_winter2016. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis). . . . 39

5.5 A Comparison Genetic Operator Thresholds for logic_2012_2013. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis). . . . 40

5.6 A Comparison the Maximum Number of Generations for logic_2012_2013. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis). 41

5.7 A Comparison the Tournament Sizes for logic_2012_2013. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis). . . . 42

5.8 A Comparison Fitness Weights for logic_2012_2013. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis). 43

5.9 A Comparison of Test Suite Generation by the Genetic and the Hill-Climbing Algorithms. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis). 45

ABSTRACT

As web applications increase in popularity, the need for extensively testing the web applications has become greater than ever. Developers are increasingly pressed to ensure that the number of faults in a web application is kept to a minimum to avoid a potential loss in the number of users of the web application. Despite the increasing importance of identifying faults and fixing them, testing web applications continues to be a very time-consuming task. As such, there exists a dire need for automating the process of testing to reveal potential faults. One such approach to testing is the generation of test suites representative of actual user behavior. However, systematic, empirical evaluation of test suites continues to be a largely unexplored area.

I propose the use of the genetic algorithm to generate test suites for web applications by first parsing user access logs to create a set of user sessions, and then modeling those user sessions as genes, chromosomes, and genomes to be used by the genetic algorithm to generate test suites representative of user behavior. I also explore the various possibilities with a genetic algorithm approach to generating test suites, and assess what effect tuning various parameters, such as genetic operator thresholds, has on the test suite produced at the end of the evolutionary run. I develop and use a framework to empirically evaluate the cost-effectiveness of the generated test suites output by the genetic algorithm. The framework employs code coverage as an evaluation metric to assess the quality of the generated test suites in particular, and the efficacy of the testing approach in general. I juxtapose the use of the genetic algorithm to generate test suites against another well-known, comparable approach. My results indicate that using the genetic algorithm can decrease the size of the test suite significantly while maintaining most of the testing functionality. In other words, the genetic algorithm can be used to create *cost-effective* test suites for web applications.

Chapter 1

INTRODUCTION

A web application is a software that usually resides on a remote server. Users typically access the software through a web browser over the Internet, although some applications do run only on local networks known as Intranets (e.g. a web application to log work hours for a company). Most web applications serve content that is dynamic in nature, i.e. the content that users see depends on certain states of the application. For example, users for a video streaming service have different recommended videos dependent on factors including their subscribed channels, their geographic locations, and their histories of videos watched, among other things. Web applications may also serve content that is static, i.e. the same content is delivered to every user, such as a log-in page for an application. Unfortunately, the dynamic and complex nature of web applications renders testing the applications more difficult than testing standalone software applications.

Web applications are increasingly becoming an integral part of our lives. We rely on web applications for day-to-day tasks, such as online banking, video streaming, social networking, and cloud computing. As our dependence on web applications increases, so does the need for web applications that are free of faults. According to an estimate, software testing process "accounts for approximately 50% of the cost of a software system development" [9]. This figure is likely higher for web applications testing for two reasons. Firstly, this estimate made in 1998 may not hold today, as software has gotten increasingly complex and so have software requirements. Secondly, this figure, which was estimate for software in general, may be an underestimation for web applications due to the increased difficulty associated with web applications testing.

Given the increase in popularity of web applications, and the greater need for extensively testing web applications, developers are increasingly pressed to ensure that the number of faults in a web application is kept to a minimum to avoid a potential loss in the number of users of the web application. It comes as no surprise then that efforts have been made to automate the process of web applications testing to minimize the time and resources spent in testing. An effective approach to testing web applications is the use of user sessions to generate test cases representative of actual user behavior [3]. However, user sessions can be very large, and this trait results in large test suites with redundant test cases. Attempts have been made to rectify this limitation, but to little effect. Peng et al. attempted to generate test cases using evolutionary computation [12], but their approach suffered from several limitations. In Peng et al.'s approach, the transition relations between pages and requests not present in the original user session set would not be covered in the generated test suite, regardless of the number of generations allowed for the evolutionary run. Additionally, this approach did not explore test case generation using other possibilities with the genetic algorithm, such as more sophisticated parent selection algorithms or different levels of crossover or mutation operations. Peng et al. also evaluated their approach on one small application using less than 100 user sessions–a sample size so small that the results might not be scalable.

I propose generating *cost-effective* test suites using the genetic algorithm by first parsing user access logs to create a set of user sessions, and then modeling those user sessions as genes, chromosomes, and genomes. I then use the genetic algorithm on an initial population of genomes over a maximum number of generations to generate a representative test suite for web application testing. I also explore what effect tuning various parameters, such as mutation and crossover thresholds and the number of generations, has on the test suite produced at the end of the evolutionary run. Additionally, I also implement different levels of genetic operations, such as genome-level and chromosome-level crossover operations, and sophisticated mutation operations, to assess the effect this variation has on the generated test suites.

I develop and use an extensible framework to empirically evaluate the effectiveness of the generated test suites output by the genetic algorithm. The framework employs code coverage as an evaluation metric to assess the quality of the generated test suites.

My contributions in this thesis:

1. explored test-suite generation by leveraging the genetic algorithm,
2. modeled test suites for web applications as components for the genetic algorithm: genes, chromosomes, and genomes,
3. implemented several genetic operators and parent selection algorithms to manipulate the genetic information of the test suites modeled as genomes,
4. produced an extensible evaluation framework that can be used to empirically evaluate testing approaches, and plan on making the evaluation framework available on GitHub for a straightforward install and an easy use,
5. empirically evaluated the test suites generated by the genetic algorithm, using code coverage as evaluation metrics, to assess their cost-effectiveness,
6. recommended the best parameters to tune the genetic algorithm for cost-effective test suite generation based on the results of my experimental study.

The organization of this thesis is as follows:

- Chapter 2 provides background knowledge about web applications, web applications testing, and genetic algorithms. It also outlines open problems involving testing and evaluating web applications.
- Chapter 3 describes my process for generating test suites for web applications using the genetic algorithm, including the motivation behind the process, the model for the algorithm, and the implementation details.
- Chapter 4 delineates the design and implementation details for the evaluation framework, including the motivation behind the metrics used for the evaluation.
- Chapter 5 explores my experimental study, including the research questions that I explore, the hypotheses I present, and the a discussion of the methodology I follow and the results I obtain.
- Chapter 6 discusses my contributions to this thesis in detail, the conclusions I draw, and the future work that I recommend.

Chapter 2

BACKGROUND

This chapter contains background information on web applications, web applications testing, and genetic algorithms. It also discusses some of the open problems in testing web applications.

2.1 Web Applications

A web application is a complex, dynamic client-server computer program that a user can access over the internet through a web browser. To interact with a web application, the client's web browser makes a HyperText Transfer Protocol (HTTP) request over a network to the server hosting the application. When the application server receives an HTTP request, it sends back an appropriate response to the client, typically in the form of a HyperText Markup Language (HTML) document. The client's web browser can process the HTML response and display it in a user-friendly format to user. The response may either be static, i.e., the content is the same for all users, or it may be dynamic, i.e., the content may depend on the user input or the state of the application.

Web applications are generally considered to be dynamic in nature. The content displayed by a web application in the web browser is often dependent on several factors. For example, the information being displayed may be pulled from a database. As the data is updated in the database, the content displayed to the user would reflect those changes. The content displayed may also depend on other factors, e.g., the type of user (administrators see information that typical users would not) and the location of the user (some content may only be accessible in certain regions). The database and

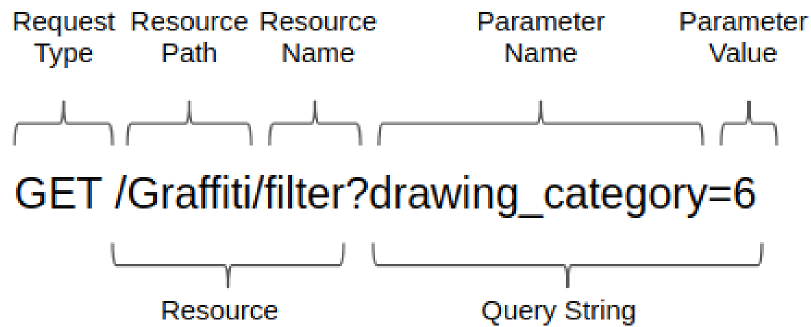


Figure 2.1: A Sample HTTP Request from the Ancient Graffiti Project

other data that a web application depends on is referred to as *persistent state*, in that the state outlives the process that initiated it.

When a user communicates with a web application, the application server records the user request and other identifying information, such as the user's IP address and the type of request, in an access log. Among other things, a user request contains the date and time of request, the type of request (typically a `GET` or `POST`), the path and name of the resource being requested, and the names and values of any parameters being passed. The *access log* contains all requests made by all users to the web application. Figure 2.1 breaks down the components of a user request made to the Ancient Graffiti Project[1], a web application.

A *user session* is the sequence of requests made by one user at one time to the application. Figure 2.2 shows a snapshot of a typical user session. User sessions are parsed from access logs, which contain all requests to the web application.

2.2 Testing Web Applications

Given their dynamic nature, composition of multiple components, and distributed architecture, it is more complicated to test web applications than standalone

[1] The Ancient Graffiti Project is a digital resource for locating and studying handwritten inscriptions of the early Roman empire.

Date and Time of Request
Request Type
Resource Requested
Request Parameter

```
16/July/2018:15:21:19 GET /Google/
16/July/2018:15:21:19 GET /Google/images/branding/googlelogo/2x/googlelogo_color?120x44dp.png
16/July/2018:15:21:19 GET /Google/static/searchengine.css
16/July/2018:15:21:20 GET /Google/adsid/google/ui
16/July/2018:15:22:02 POST /Google/search --post-data="&query=apache+tomcat"
16/July/2018:15:22:03 GET /Google/images/branding/googlelogo/2x/googlelogo_color?120x44dp.png
16/July/2018:15:22:03 GET /Google/images/phd/px.gif
16/July/2018:15:22:03 GET /Google/textinputassistant/tia.png
16/July/2018:15:22:03 GET /Google/static/searchengine.css
16/July/2018:15:22:03 GET /Google/adsid/google/ui
16/July/2018:15:22:59 POST /Google/search --post-data="&query=apache+tomcat+for+fedora"
16/July/2018:15:23:00 GET /Google/images/branding/googlelogo/2x/googlelogo_color?120x44dp.png
```

Figure 2.2: A Sample User Session from a Google Search

computer applications for functionality. Several web applications testing approaches have been shown to be effective at revealing faults in functionality [10], but each approach has its limitations.

Past attempts at using user sessions to test web applications have proven to be promising [3][4][2][16]. Employing user sessions in testing approaches is an effective technique because user sessions are representative of actual user behavior, and as such, can serve as useful test cases. This is because mimicking user behavior tends to produce test cases that focus on the parts of the web application that are used frequently, and as such, can be more effective at revealing higher priority faults (i.e., the faults that users are likely to encounter).

Another approach to testing web applications involves the use of bots, such as web crawlers, to browse through web pages. The use of crawlers to automate the process of testing web applications to reveal faults has also proven to be effective [14] at revealing faults in web applications. However, test cases from this approach do not represent how the web application is used on a daily basis by users since this approach merely visits pages in certain sequences to reveal faults.

Other widely-used post-deployment approaches to web applications testing include load testing [8][17], which assesses the robustness of the web application when

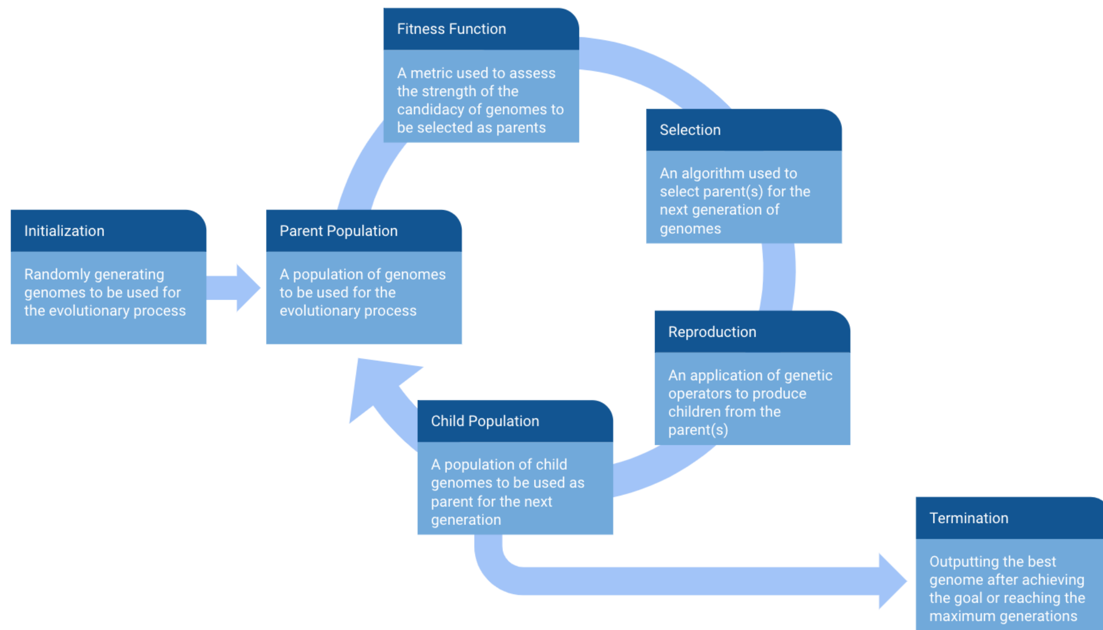


Figure 2.3: An Overview of the Genetic Algorithm Process

it is under heavy traffic (e.g., an abnormal number of users accessing the web application's resources simultaneously), and crowd-sourced testing, which involves a number of testers from different places—instead of the software developers—testing the web application application [21].

2.3 Genetic Algorithms

Genetic algorithms are adaptive problem-solving tools, primarily used to solve optimization problems. In a process that attempts to mimic the biological evolution process, genetic algorithms take as input a population of random individuals—also known as *genomes*—and manipulate these individuals over the course of the evolutionary run to achieve a certain goal. Similar to biological evolution, the individuals with high fitness survive and reproduce in genetic algorithms, attempting to produce individuals with higher fitness values. This process is continued until the goal is accomplished or a certain maximum number of generations is achieved, at which point the fittest individual is output. Genetic algorithms employ *fitness functions*, *parent selection algorithms*, and various *genetic operators* to accomplish this task. Figure 2.3

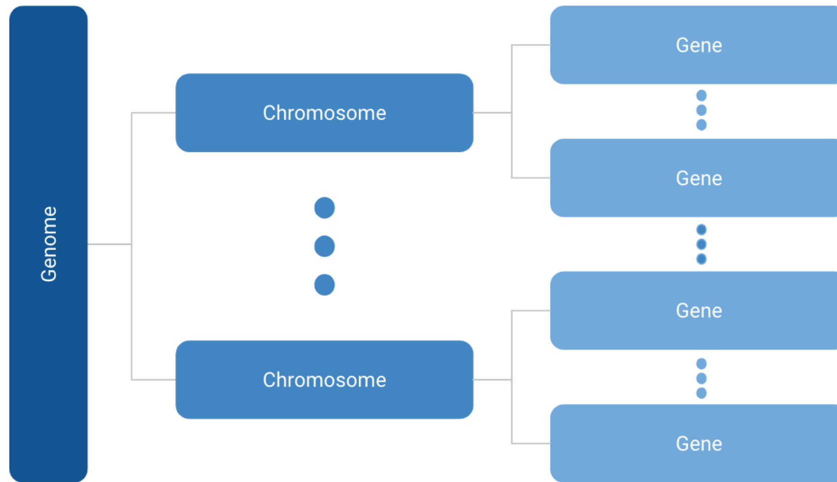


Figure 2.4: The Model for the Genetic Algorithm

delineates the steps involved in a genetic algorithm evolutionary process, which are discussed in detail in the remainder of this section.

2.3.1 Model

A genetic algorithm attempts to achieve a specified goal by evolving *genomes* to produce higher fitness genomes. To do so, a genetic algorithm framework requires that the input to the genetic algorithm be modeled as genes, chromosomes, and genomes. A *gene* serves as the building block of a genome. A sequence of genes represent a *chromosome*. A *genome*, then, encodes a sequence of chromosomes. Figure 2.4 outlines the model for the genetic algorithm.

The set of all genomes for a generation is known as the *population* for the generation. A genetic algorithm framework maintains a pool of chromosomes throughout the evolutionary process. This pool is used to initialize the random population of genomes before beginning the evolution and to swap random chromosomes into a genome during a mutation operation (see 2.3.4).

2.3.2 Fitness Functions

To evolve genomes, the concept of *fitness* for individuals in the problem's domain needs to be introduced. A *fitness function* is used as a metric to evaluate the strength of the candidacy of individuals to pass their genetic information to subsequent generations. A good fitness function assigns high fitness values to individuals that possess desirable behavior and penalizes the fitness of individuals that do not. This ensures that, for the most part, better characteristics are passed on as the evolution proceeds, while the less desirable ones are discouraged.

2.3.3 Parent Selection Algorithms

Parent selection algorithms work in conjunction with the fitness function to select an individual with genetic information that would benefit future generations of individuals. There are several parent selection algorithms that may be used with genetic algorithms, and each comes with its own benefits and drawbacks. Some of the more common parent selection algorithms are listed below:

1. **Random Selection**: This algorithm simply selects an individual from the population as the parent, regardless of the individual's fitness. Despite its runtime complexity of $\mathcal{O}(1)$ and its potential for increases in genetic diversity, this algorithm does little to steer the population towards individuals of higher fitness through the evolutionary process.

2. **Tournament Selection**: This algorithm selects a pool (or a sub-population) of n random individuals and returns the best individual in that pool. This algorithm has a runtime complexity of $\mathcal{O}(n)$ and, as such, is the most efficient approach for selecting a parent besides random selection. The simplicity of its algorithm, coupled with the remarkable extent to which it selects a parent with desirable characteristics, makes it a widely chosen default parent selection algorithm for genetic algorithm frameworks.

3. **Roulette Selection (Fitness Proportionate Selection)**: This algorithm functions as a roulette wheel in that each individual's probability of being selected is proportional to its fitness. Therefore, higher fitness individuals have a greater chance of being selected. However, there is still a chance that lower fitness individuals might get selected.

4. **Pareto Selection**: This algorithm makes use of Pareto fronts, as opposed to the fitness function, to select individuals as parents. Given a set of fitness objectives, an individual *Pareto-dominates* another if the former does not perform worse than the latter in all fitness objectives and performs better in at least one fitness objective [13]. The Pareto front is first computed as the set of all individuals that are not Pareto-dominated. A parent is then selected at random from the Pareto front. Pareto selection is well-suited to applications attempting to optimize multiple objectives for the individuals during the evolutionary process.

5. **Truncation Selection**: This algorithm orders individuals by fitness, and a proportion p of the fittest individuals are selected and reproduced $1/p$ times. Doing so ensures that only the fittest individuals pass genes to future generations and the genes from the less fit individuals are lost in evolution. Truncation selection is less sophisticated than most other parent selection algorithms but has less to offer and is therefore not widely used in practice.

6. **Reward-based Selection**: This algorithm requires a concept of rewards–which could be closely tied to fitness–for individuals. In this algorithm, the probability of an individual to be selected is proportional to its cumulative reward, which can be computed as the sum of the rewards of the genome and its parent(s). This algorithm works similarly to roulette selection, except that it also considers the parents' reward values in determining the individual's reward. This allows a weak child of strong parents to still have a fair chance at being selected as a parent in hopes for passing the desirable genes from the parents. This algorithm provides greater potential to increase the diversity of a population, at the expense of having to define a reward metric for the individuals.

2.3.4 Genetic Operators

Genetic operators manipulate the individuals' genetic information to guide the population towards a solution to the problem. Some of the more common genetic operators are listed below:

1. **Mutation**: This operator selects a chromosome at random from a genome and replaces it with a randomly selected chromosome from the pool of chromosomes.

2. **Variable Length Mutation**: Previously shown to be successful [6], this operator makes a single pass through a genome, inserting a random number of chromosomes, selected from the pool of available chromosomes, to the genome. It then makes another pass through the genome and selects and deletes a random number of chromosomes. The resulting variable-length genome is then returned.

3. **One-point Crossover**: This operator picks a split point for two genomes and slices the genomes at that point. It then combines the first part of the first

genome and the second part of the second genome to create a child. Similarly, it also combines the second part of the first genome and the first part of the second genome to create another child.

4. **Two-point Crossover**: This operator works in very much the same way as one-point crossover, except that it picks two split points—which are very likely different—and slices the genomes at their respective split points. It then combines a part of one genome and a part of another genome to create a child, using the other parts of the genomes to create the second child. Unlike its one-point counterpart that preserves the genome length after the crossover, the two-point crossover operator often results in variable length genomes after the crossover. Allowing the crossover operator to manipulate genome length offers the possibility of shortening genome length during the crossover, thereby increasing the fitness of the genome even more.

2.3.5 Applying the Genetic Algorithm to Software Testing

Employing the Genetic Algorithm for software testing is not a novel concept. For example, Srivasta and Kim proposed a method for optimizing software testing efficiency by employing the genetic algorithm to identify the most critical path clusters in a program [20], while Alander et al. used the genetic algorithm to automatically generate test data to find problematic situations, such as processing time extremes [1].

Peng and Lu [12] first proposed using the genetic algorithm to generate test suites from user sessions, using roulette selection as the parent selection algorithm and simple mutation and crossover as the genetic operators. Their findings indicated that using the genetic algorithm approach is more effective than the traditional user session based testing and attains higher path coverage and fault detection rate with a smaller sized test suite. However, for the approach proposed by Peng and Lu, transition relations between pages and requests not present in the original user session set would not be covered in the generated test suite, regardless of the number of generations allowed for the evolutionary run. Additionally, Peng and Lu did not explore test suite generation using other possibilities with the genetic algorithm, such as more sophisticated parent selection algorithms or different levels of crossover or mutation operations. They evaluated their approach on one small application using less than 100 user sessions.

2.4 Open Problems

As web applications increase in popularity, the need for extensively testing web applications has become greater than ever. Developers are increasingly pressed to ensure that the number of faults in a web application is kept to a minimum. Despite the increasing importance of identifying faults and fixing them, testing web applications continues to be a very time-consuming task. As such, there exists a dire need for automating the process of testing to reveal potential faults. Employing efficacious strategies to generate test suites for web applications, and then empirically evaluating them for effectiveness could significantly reduce the time it takes to identify faults in web applications. Unfortunately, little work has been done on evaluating test suites in particular, and testing strategies in general, for testing efficacy.

2.4.1 Problems in Testing Web Applications

Given the complex and dynamic nature of web applications, the process of testing web applications is more complicated than testing a standalone user application. Developers often need to ensure cross-browser compatibility to make the web application easier to access. This requirement has the potential to introduce unexpected complications (e.g., different browsers may render HTML in slightly different ways, so faults hidden in one browser may be revealed in another). Given that a web application has different components (e.g., the server, the clients, and the data store) working together, any of which may be physically separated from the rest, a fault in a single component is often difficult to isolate and fix. Furthermore, responses from web applications are dynamic and dependent on the application's state. As such, it is virtually impossible to probe the application with all possible user inputs on all possible states and test all possible responses from the application. This limitation necessitates the generation of test suites that represent normal use-case scenarios for the web application to ensure that parts that users rely on do not crash upon daily use. In other words, the test suite needs to at least test what the users did with the web application and what they are likely to do.

2.4.2 Evaluating Web Application Testing Approaches

Systematic, empirical evaluation of test suites continues to be a largely under-explored area. While there have been attempts to define metrics that can be used to assess the quality of test suites, this area is still relatively underdeveloped.

The reasons behind the lack of the existence of good evaluation frameworks for web applications testing are largely tied to the difficulty involving evaluating test suites for web applications. Most web applications are large—both in terms of the amount of back-end code that goes into the web application and in terms of the size of the user base of the web application. To make evaluating test suites even more complicated, virtually all web applications are dependent on certain configuration files and other software dependencies. Furthermore, the software dependencies may have different requirements, such as the version of the dependency.

To effectively evaluate test suites, a software developer would need to deploy the web application on a server separate from the actual production server so as not to interfere with the normal functionality of the web application. The application's persistent state (e.g., a database holding the information used by the application) would need to be initialized before the application can run and the test suites are executed. Once the process of execution of test suites is complete, the results of the executions would be evaluated. These steps are often done multiple times, so it is of paramount importance that they be handled automatically with minimal input from the software developer.

Chapter 3

GENERATING TEST SUITES FOR WEB APPLICATIONS USING THE GENETIC ALGORITHM

I apply the genetic algorithm to generate test suites for web applications using logged user sessions from web applications. My goal is to generate cost-effective test suites better suited for exposing functionality faults.

3.1 Motivation

Genetic algorithms are adaptive problem-solving tools primarily used to solve optimization problems, as discussed in section 2.3. Previous studies have demonstrated that search-based techniques have shown promise when applied to software testing [20]. However, previous attempts at generating test suites using the genetic algorithm failed to exploit the wide realm of possibilities that genetic algorithm offers, such as more sophisticated parent selection algorithms or genetic operators, to generate test suites. One such attempt involved evaluating the approach on one small application using less than 100 user sessions [12]. My proposed model addresses these limitations. Additionally, since I parse logged user sessions to web applications to generate test cases for the starting point of the genetic algorithm, the test suites produced are all representative of actual user behavior. This approach could greatly reduce the size of the test suite–when compared to the original set of user logged user sessions–while retaining most of the testing functionality.

3.2 Model

To model the research problem of generating test suites for the genetic algorithm, I break a web application test suite into the genetic algorithm input components: genes,

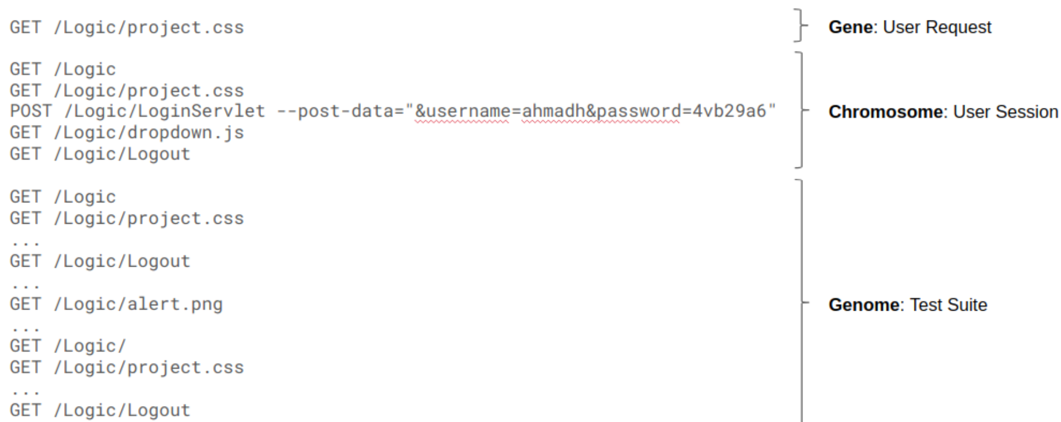


Figure 3.1: Modeling Web Applications Test Suites for the Genetic Algorithm

chromosomes, and genomes. A *gene* encodes a single user request. A *chromosome* encodes a user session, i.e., a sequence–or an ordered list–of genes. A *genome* encodes a test suite, i.e., a sequence of chromosomes. Figure 3.1 highlights this relationship between genes, chromosomes, and genomes for web application test suites.

3.3 Implementation

The following section discusses the implementation-specific details of my genetic algorithm framework. I use Python 3.6 to implement the framework for test suite generation. Figure 3.2 outlines the genetic algorithm framework for generating test suites for web applications.

3.3.1 Initialization

The user accesses to the subject applications are converted into user sessions using Sprenkle et. al framework [19]. I then parse these user sessions to produce a user session–or chromosome–pool to be input to the genetic algorithm framework. To initialize an evolutionary process, I generate genomes from this pool of chromosomes.

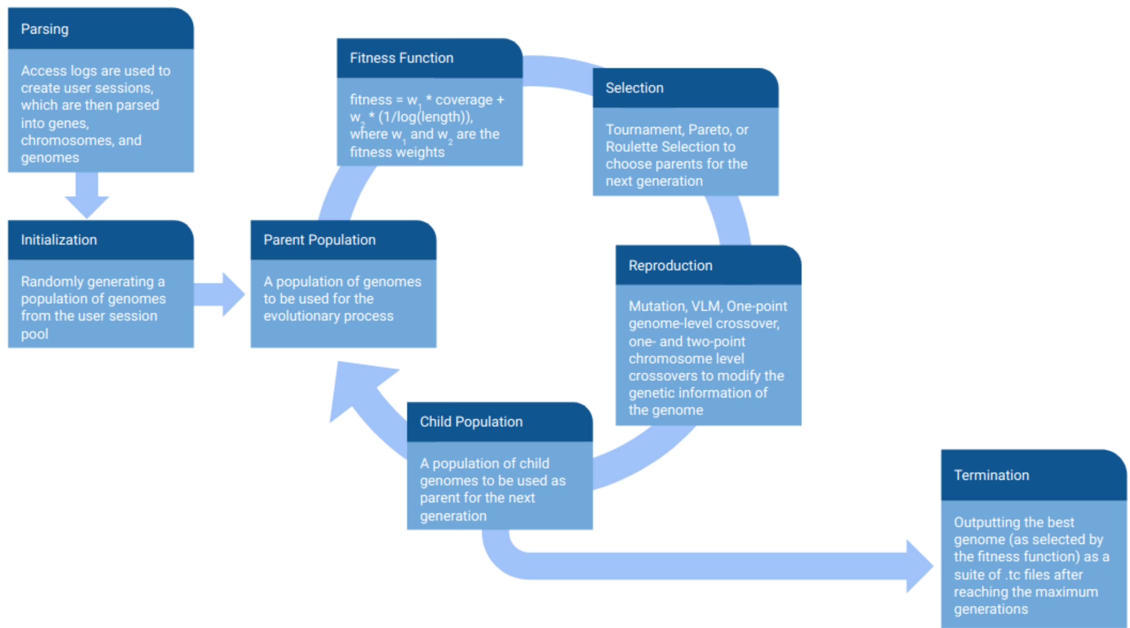


Figure 3.2: An Overview of the Genetic Algorithm For Generating Test Suites

By default, each genome is set to comprise of a hundred chromosomes, while chromosomes have a variable number of genes. These genomes serve as the parent population for the zeroth generation, or the start point, of the evolutionary process.

3.3.2 Fitness Function

As discussed in Section 2.3.2, a fitness function is used as a metric to evaluate the strength of the candidacy of genomes to pass their genes to the subsequent generations. A good fitness function should ensure that fit genes are preserved in an evolutionary run, while the unfit ones are discarded.

For the purpose of finding faults in a web application, a *good* test suite should provide extensive coverage of the code and must do efficiently (in a reasonable amount of time). As such, I use two metrics to evaluate the fitness of genomes: resource plus parameter names—or RRN—coverage and genome length. *RRN coverage* is defined as the percentage of a web application's unique requests (as represented by the request type, resource name, and parameter names) that a test suite executes [19]. *Genome*

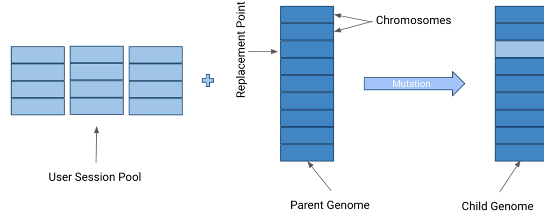


Figure 3.3: An Overview of the Mutation Operator

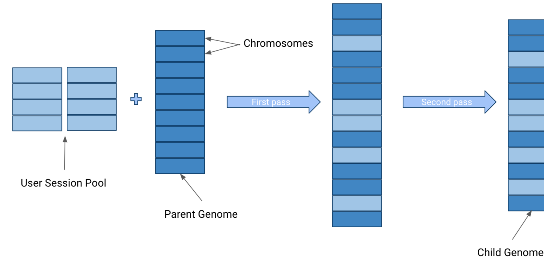


Figure 3.4: An Overview of the Variable Length Mutation Operator

length is defined as the total number of genes, i.e., user requests, in a genome. I then combine these metrics into the fitness function used by the genetic algorithm:

$$fitness = w_1 * coverage + w_2 * \frac{1}{\log(length)} \tag{3.1}$$

where w_1 and w_2 are weights used to provide more emphasis on one fitness metric over the other. Increasing the ratio $w_1 : w_2$ rewards coverage over length, while decreasing the ratio rewards length over coverage. A tester can tune these weights to find the optimal values of w_1 and w_2 that result in the most effective test suite.

3.3.3 Genetic Operators

As discussed in section 2.3.4, genetic operators are used to manipulate either chromosome sequences in a genome or gene sequences in a chromosome to steer the population of genomes towards a global optimum. In doing so, the genetic operators also attempt to increase the population's genetic diversity.

I use five different genetic operators. Each operator has its own benefits, and certain operators thrive more in select circumstances over the others:

- **Mutation**: Mutation can be useful for increasing diversity of the genomes by potentially pulling a chromosome from the user session pool that is not already represented in the current population. This could help the resulting test suite get higher resource coverage, since a resource that was not previously being accessed by a genome might be accessed after the genome mutation. Figure 3.3 describes the mutation operator.

- **Variable Length Mutation (VLM)**: This operator provides greater potential for increasing diversity compared to regular mutation because every pass through the genome swaps out *multiple* chromosomes instead of just one. However, this operator is computationally more expensive than regular mutation. Figure 3.4 describes the variable length mutation operator.

- **One-point Genome-level Crossover**: This operator is conceptually simpler to understand and keeps the genome lengths consistent, making the results easier to interpret. It is useful for crossing the chromosomes of two different genomes to yield a different chromosome sequence, allowing for greater potential to reveal faults. Figure 3.5 outlines the one-point genome-level crossover operator.

- **Two-point Genome-level Crossover**: This operator has the potential to decrease the genome length, creating the possibility for smaller genomes (and hence, test suites) that provide similar code coverage. It is important to note that two-point genome-level crossover is equally likely to increase genome length, but the fitness function would penalize an increase in genome length, so the increase will likely not affect the final test suite—unless it increases coverage sufficiently.

- **Chromosome-level Crossover**: This operator introduces the potential to create new chromosome sequences not present in the user session pool. This results in pages being visited and resources being requested in new sequences when the test suite is executed for evaluation, and this introduces more potential to reveal faults. However, chromosome-level crossover can be computationally more expensive, since the operator tries to find matching starting and ending points for chromosomes for crossover (see Section 2.3.4). Figure 3.6 outlines the two-point chromosome-level crossover. Note that the two-point genome-level crossover works in very much the same way, except it operates on genomes, not chromosomes.

3.3.4 Parent Selection Algorithms

As discussed in section 2.3.3, parent selection algorithms work in conjunction with the fitness function to select a parent with genes that would benefit future generations of genomes. Doing so ensures that good genes are passed on to the final test suite, and the bad ones are discarded in the evolution process.

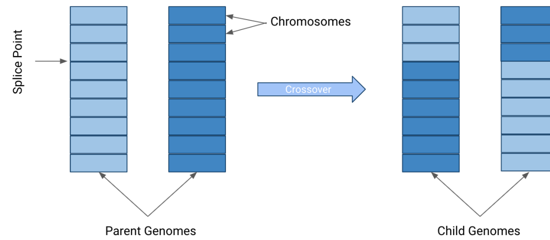


Figure 3.5: An Overview of the One-point Genome-level Crossover Operator

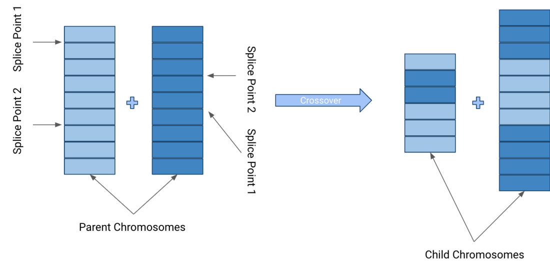


Figure 3.6: An Overview of the Two-point Chromosome-level Crossover Operator

I use three different parent selection algorithms out of the set of parent selection algorithms mentioned in section 2.3.3:

- **Tournament Selection**: This is a simple selection algorithm that is used as the default for most genetic algorithmic frameworks. It is a straightforward algorithm that does not attempt complex calculations to pick a parent, but at the same time performs better than sorting the population by the genomes' fitness values and only picking the best ones. In picking a pool, or a sub-population, from the population of genomes, tournament selection attempts to maintain some form of genetic diversity by potentially allowing less fit genomes to be selected for mutation or crossover.
- **Roulette Selection (Fitness Proportionate Selection)**: This algorithm, while still avoiding complex calculations, preserves genetic diversity better than tournament selection. Since every individual's probability of being selected for mutation or crossover is dependent on its fitness value, roulette selection has the potential to pick a less fit genome. In contrast, tournament selection always picks the best genome out of a pool of genomes. By picking a less fit genome, roulette selection allows for the possibility of preserving a good gene in a bad genome–this gene may prove useful later in the evolutionary run.
- **Pareto Selection**: This selection algorithm works well for optimizing multiple objectives, as is the case for test-suite generation, where we want test suites with

high code coverage *and* few resources executed. Pareto selection has been shown to perform well for multi-objective optimization [15], eliminating the need for a specific fitness function dependent on fitness weights to balance the prioritization of one objective over another.

3.3.5 Summary

This chapter highlighted how I apply the genetic algorithm to generate test suites using logged user sessions from web applications and delineated my goal to generate cost-effective test suites for web applications. The next chapter discusses my methodology for evaluating the test suites generated by the genetic algorithm framework.

Chapter 4

THE EVALUATION FRAMEWORK

I present an extensible evaluation framework that can be used to empirically evaluate web application test suites and testing approaches. This chapter contains the details about the design and the implementation of my evaluation framework.

4.1 Design

Due to the difficulty involving evaluating testing approaches for web applications testing (see section 2.4.2), there is a lack of frameworks that automate the process of evaluating test suites for web applications. Much of the problems in evaluating these test suites stem from the repetitive nature and the complications involving the tasks that need to be performed before a test suite can be evaluated. As such, automating the steps involved in evaluating test suites for a web application was an important requirement of the design of the evaluation framework.

At a high level, a single command starts the evaluation of test suites. This command executes a script used to automate the steps involved in evaluating web applications testing. The script starts a web application server locally to deploy a copy of the application and then executes the test suite. Before this can happen, however, there are certain tasks that need to be performed, including instrumentation of the source code, setting up the application's configuration files, and initializing the application's persistent state. Once this setup for the evaluation is done, the framework can proceed to executing test suites, before the results of the execution can be processed and displayed for evaluation. Figure 4.1 shows an overview of the evaluation framework.

It should be noted that there is an initial setup cost when the evaluation framework is initialized to be run on a new machine. This setup requires changes to the

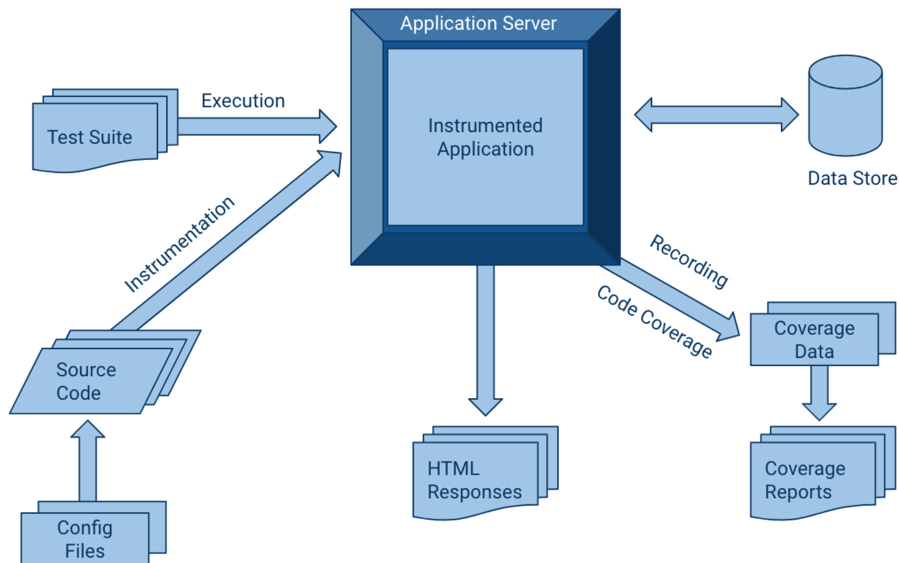


Figure 4.1: An Overview of the Evaluation Framework

scripts and configuration files, since the directories of the installation of the web server or the Java Runtime Environment (JRE), among other things, may be machine-dependent. However, the benefits of the evaluation framework far outweigh the costs of setting it up to work with an application.

4.1.1 Configuring the Application

Virtually all web applications are dependent on configuration files (see section 2.4.2). To ensure correct functionality of a web application, it is necessary to initialize the configuration files prior to running the application. Ideally, configuring the web application should be an automated process. In reality, however, some configurations are machine-dependent, e.g., setting up the class paths for the application. As such, these configurations need to be handled manually by the tester, which adds to the overhead cost of evaluating web applications.

My evaluation framework attempts to address this limitation by semi-automating the process of configuring an application. Prior to evaluating a new application, the

tester must modify the scripts used to run the evaluation process to set any configurations that may be needed, e.g., updating the class paths. Once this is done, however, the tester can use the same script multiple times without manually changing configuration files.

4.1.2 Initializing the Persistent State

The dynamic content generated by a web application is often largely dependent on the persistent state of the application, as discussed in section 2.4.1. The persistent state of a web application includes the database and other states that outlive the processes that initiated the states, and are external to the application itself.

Without the initialization of the persistent state of the web application, most of the executed test suites will result in error pages (e.g., HTTP Status 500) or pages with error messages (e.g., "Login failed"), and the test suites will not be able to be fairly evaluated. Given that the tester has gone through the initial steps of getting the persistent state of an application, storing it in a data store on the machine, and updating the evaluation scripts to point to the correct configuration files, my evaluation framework automates the process of setting up the persistent state for the application to run. While the initial steps to configuring the scripts for the persistent state result in an increased overhead cost of setting up the evaluation framework, the benefits to this approach far outweigh the drawbacks. The tester never has to worry about setting up, for example, the database for the web application, each time he or she executes test suites. This automation ends up saving the time and effort put into preparing the application before test suites can be executed.

4.1.3 Instrumenting the Source Code

Instrumentation, a widely used technique in software development, testing, and profiling, involves adding extra code to an application to monitor some behavior of the application. This monitoring can be static (i.e., performed at compile-time) or dynamic (i.e., performed at run-time) [7]. Instrumentation is common in evaluating software

testing techniques using code coverage because it provides the ability to monitor the lines and branches of code covered by the execution of test cases.

Prior to deploying an application to the local web application server, my evaluation framework can instrument the Java class files and the JavaServer Pages (JSP) files of the web application to provide code coverage on the back-end source code as well as the code for the web pages. Requiring no user input and reporting any errors as they occur, the instrumentation process is entirely automated.

4.1.4 Executing the Test Suites

Once the the application is instrumented and configured and its persistent state is initialized, it can be deployed to the server for the test suites to be executed. The automated scripts that drive my evaluation framework take care of this process in its entirety so that the tester only has to execute one command for the test suites to be executed. To execute the test suites, my evaluation framework uses a replay tool that simulates the execution of each user request to the web application and saves the response from the web server in a specified directory.

During the execution of the test suites, the instrumented source code is monitored for code coverage, and the coverage data are written in real time to a specified directory as the code is executed.

4.2 Implementation

This section delineates the implementation-specific details of the evaluation framework that I present.

I plan on making the evaluation framework available in a GitHub repository for easy installation on new machines. Provided that any necessary dependencies (e.g., JRE, Apache Tomcat, Apache Ant, etc.) have been installed on the machine, the framework can be set up and ready to run in a matter of minutes.

4.2.1 Automating the Execution of Test Suites

The evaluation framework uses scripts to automate the entire process of setting up the web application, executing test suites, and evaluating the results. More specifically, I use Bourne Again Shell (Bash) scripts to execute test suites for subject applications. The Bash scripts take as arguments the names of the application and the test suite to be executed. When a script is executed, it sets up certain parameters and the source and destination directories. It then instruments and compiles the source code, initializes the persistent state, and starts the Apache Tomcat server after deploying the application. To execute the test suites, it uses a Java replay tool to execute each test case on the application and save the response. It finally creates and saves the coverage reports for the tester to process.

4.2.1.1 Instrumenting the Source Code

My evaluation framework uses the freely available, open-source Atlassian Clover to instrument the Java and JSP source code to monitor and report code coverage.

I use Apache Ant, a tool for automating software build processes, to define and produce target builds for the web application. This automated process of building incorporates the instrumentation features provided by Clover to instrument and compile the application (i.e., compile the .java files and .jsp files to .class files) to prepare it for deployment.

4.2.1.2 Initializing the Persistent State

The process of initializing the persistent state of a web application usually involves setting up the database for the application. Certain applications may require more states to be set up, e.g., setting up Elasticsearch for a search engine application.

To set up the database for a web application, I use the object-relational database management system PostgreSQL. The process of setting up the database involves obtaining a data dump of the database on the production server (i.e., the server hosting the actual web application). I then use automated scripts to create a database on the

machine running the evaluation framework, and use the original data dump to replicate the data store for the web application for the execution of the test suites. These scripts are called by the main driver script so there is no need for the tester to manually set up the data store.

4.2.1.3 Running the Web Applications Server

I use Apache Tomcat 8 as the web applications server to host the subject applications being tested. The automated Bash driver script is responsible for shutting down Tomcat if it is already running, and then starting Tomcat once the web application has been deployed. As is the case with the other stages of the evaluation of test suites, no user input is required for this process.

4.2.1.4 Executing the Test Suites

To execute the test suites, I use a replay tool written in Java that establishes an HTTP connection to the server on the given host and port number before sequentially executing the test suites on the application. The replay tool, built on HttpClient, executes a test case by making a request to the server (typically, a GET or POST) and then saving the response from the server in a specified directory. This process is continued until all test cases in the test suite have been executes and their responses saved.

Since the source code is already instrumented at this point, each request made by the replay tool updates the code coverage data of the application as lines of code are executed.

4.2.2 Evaluating the Results

Once the execution of the test suites is complete and the code coverage data written to a directory, the tester has two options to evaluate the results. I use Apache Ant (see section 4.2.1.1) to generate two targets for the coverage reports: one as an XML report (the default format) and the other as an HTML report.

If the tester wishes to view a graphical representation of the coverage data, he or she can view the HTML version of the report. This report includes the overall code coverage, as well as other helpful information such as the top project risks, the most complex packages and classes, and the least tested methods. It displays various code metrics including the number of branches, statements, methods, classes, lines of code, etc. The tester also has the option to browse the packages and classes to view individual coverage percentages as well as a graphical representation of what lines of code were covered and what lines of code were not. This view of the report is helpful for the tester to glance at and superficially evaluate the test suite without needing to analyze large quantities of data.

Alternatively, if the tester wishes to analyze the report programmatically, there is also an Extensible Markup Language (XML) report that contains the coverage data in a format that is easier for a computer to understand.

I chose these formats for ease of use depending on the circumstances in which the reports are being evaluated. The XML report makes it easy for another automated script to process the report, while the HTML report makes it easy for a human user to view and assess the results from the test suites. Even though I generate the reports in XML and HTML formats, Clover's ease of use and readily available documentation makes it very straightforward to modify the Ant scripts to generate reports in different formats, e.g., as a PDF or a JavaScript Object Notation (JSON) file.

4.2.3 Summary

This chapter highlighted my methodology for evaluating the test suites generated by the genetic algorithm framework, and discussed the implementation specifics of the evaluation framework that I produced. The next chapter goes over the details of my experimental study, including my research questions, my experimental procedure, the analysis of my results, and conclusion of my findings.

Chapter 5

EXPERIMENTAL STUDY

In this chapter, I discuss the research questions I seek to answer experimentally, and then delineate the experiments I use to verify my hypotheses. I then provide an evaluation and analysis of the experimental results. I also provide any threats to validity, as well as a brief conclusion of the results.

5.1 Research Questions

The overarching research question I seek to answer with my thesis is can I generate cost-effective test suites using my genetic algorithm framework? To answer this question, I need to first answer the following questions:

- **Question 1:** What parent selection algorithm provides the best results when used in the genetic algorithm framework to generate test suites for web applications?
- **Question 2:** What genetic operators are most useful in this application of the genetic algorithm? What parameters (e.g. genetic operator thresholds, maximum number of generations, etc.) should be used to tune the framework for test suite generation?
- **Question 3:** What is the effect of fitness weights on the quality of the final test suite?
- **Question 4:** How well does the generated test suite perform on the evaluation framework, when compared to the original user session set?
- **Question 5:** How does the genetic algorithm approach to generate test suites for web applications compare to using the hill-climbing algorithm[1] to generate test suites?

[1] The hill-climbing algorithm is an iterative algorithm that starts with an arbitrary solution to a problem, and then makes iterative changes to the solution until a certain maximum number of iterations is reached. (See Section 5.3.3 for the hill-climbing implementation details.)

Table 5.1: Subject Application Characteristics

Subject Application	Branches	Statements	Methods	Classes	Files	NCLOC
Logic2	4,606	16,634	1,273	157	156	25,639
Logic5	5,722	20,330	1,438	177	174	30,977

Table 5.2: Subject User Session Set Characteristics

Subject	Subject Application	Test Cases	Total Requests	Unique Requests
logic_fall2010	Logic2	485	49431	138
logic_2012_2013	Logic5	2839	122149	181
logic_winter2016	Logic5	1460	63164	303

5.2 Subject Applications

The subject applications that I use to evaluate my approach are several versions of an online symbolic logic tutorial, which I will refer to as Logic. The applications are written in Java using servlets and JSPs. Logic2 is an earlier version of the Logic application, while Logic5 is an updated version of the Logic application. Table 5.1 summarizes the subject applications' characteristics. For the code metrics, NCLOC refers to the non-comment lines of code (i.e., the number of lines of code that are not documentation).

5.3 Methodology

This section discusses my experimentation process. To analyze the results from the experiments, I examine the trends spanning across the entire population of genomes for the entire iteration of the genetic algorithm evolutionary process. This allows me to understand the general trends that the populations of genomes follow, making it easier to verify hypotheses and draw conclusions.

5.3.1 Gathering User Sessions

The user accesses to the subject applications are converted into user sessions using Sprenkle et al.'s framework [19]. Prior to processing the access logs, accesses from IP addresses known to be bots or web crawlers are removed to reduce noise from non-human users and to ensure that the parsed user sessions are better representations of

user behavior. The parsing process involves recording web server accesses, generating user sessions from the access logs [18], removing static user sessions (i.e., user sessions that only request static web pages, and therefore do not access application code), and generating replayable user sessions.

The user sessions for Logic were collected during typical use of the application in academic semesters in the years 2010, 2012-2013, and 2016. Table 5.2 shows the characteristics of the sets of the collected user sessions. As indicated in the table, the subject logic_fall2010 were collected when the Logic2 version of the application was deployed, while the subjects logic_2012_2013 and logic_winter2016 were collected while Logic5 was deployed.

5.3.2 Evaluation and Analysis Metrics

I use the two metrics discussed in Section 3.3.2 to evaluate the fitness of individual genomes: RRN coverage and genome length. These metrics combine to form the fitness of a genome as follows:

$$fitness = w_1 * coverage + w_2 * \frac{1}{\log(length)},$$

where w_1 and w_2 are fitness weights.

To evaluate a genome population, I define a *diversity* metric as the number of unique genomes in a population. Diversity is imperative to the success of a genetic algorithm evolutionary run because it allows the genetic algorithm to explore an array of different genetic compositions.

5.3.3 Generating Test Suites

After parsing the access logs into user sessions, the user sessions are encoded as objects representing genes, chromosomes, and genomes, which are the input to test-suite generation.

To answer my research questions about generating test suites, I varied certain parameters while keeping others fixed. Due to the random nature of genetic algorithm

evolution processes, I ran each experiment twice. For both experiments, I calculated the percentage difference between the fitness values—as defined by the fitness function—of the fittest genomes at the end of the two experiments. If the percentage difference was greater than 3%, indicating that the results from the two experiments differed significantly, I re-ran the experiment a third time, ensuring that the third experiment was consistent with one of the first two experiments.

I use the following as baseline measures:

- *Tournament selection* as the parent selection algorithm with a *pool size of 5* genomes: across implementations of genetic algorithm applications, tournament selection is regarded as a good default parent selection algorithm [11].
- Coverage and length *fitness weights of 1 and 5* respectively: the results from my preliminary experiments suggested these fitness weights serve as reasonable default values.
- A *population size of 50* genomes evolved for *200 generations*: in having to choose between (a) a large population of genomes evolved for a smaller number of generations and (b) a smaller population evolved over a large number of generations, the latter option appears to better reap the benefits of evolution provided by the genetic algorithm.
- Mutation and one-point genome-level crossover *thresholds*[2] *of 0.3 and 0.7* respectively: these genetic operator thresholds often function as defaults for preliminary implementations of genetic algorithm applications.

For each experiment, I varied only one type of parameter to shed some light over the research question in consideration.

Question 1: *What parent selection algorithm provides the best results when used in the genetic algorithm framework to generate test suites for web applications?*

Across all subject user session sets, keeping other parameters at the baseline measures, I ran experiments with tournament selection, Pareto selection, and roulette (fitness-proportionate) selection as the parent selection algorithms. Note that while

[2] *A threshold of* x*, where* $0 \leq x \leq 1$*, for a genetic operator denotes the probability of the operator being selected to genetically modify a parent genome. The thresholds for all genetic operators must sum to 1.*

Table 5.3: Genetic Operator Thresholds

Experiment	Mutation	VLM	One-point	Two-point Genome	Two-point Chromosome
1	0.30	0.00	0.70	0.00	0.00
2	0.15	0.15	0.70	0.00	0.00
3	0.30	0.20	0.50	0.00	0.00
4	0.30	0.00	0.50	0.20	0.00
5	0.30	0.00	0.50	0.00	0.20
6	0.15	0.15	0.40	0.15	0.15

tournament selection and roulette selection both employ the fitness function to select parents, Pareto selection makes use of Pareto fronts (see Section 3.3.4) to select parents, using the fitness function only in the last generation to output the fittest test suite.

Question 2: *What genetic operators are most useful in this application of the genetic algorithm? What parameters (e.g. genetic operator thresholds, maximum number of generations, etc.) should be used to tune the framework for test suite generation?*

To find the effect of varying the genetic operator thresholds on the quality of the resulting test suite, I experimented with numerous combinations of genetic operator thresholds. Table 5.3 shows the sets of thresholds used for my experiments. Note that for the genetic operators, *VLM* refers to Variable Length Mutation, *One-point* refers to one-point genome-level crossover, *Two-point Genome* refers to two-point genome-level crossover, and *Two-point Chromosome* refers to two-point chromosome-level crossover.

To investigate how varying the number of generations affects the quality of the test suite, I ran experiments with limits of 100, 200, and 400 generations on each user session set, with 50 genomes in the population. While it would be most beneficial to run the genetic algorithm for more generations with more genomes in a population, the choices between population size and number of generations are often limited by the processing power and memory available. As such, to increase the number of genomes in the population, the maximum number of generations would need to be decreased, and vice versa.

To study the effect of changing the pool size for tournament selection on the resulting test suites, I varied the pool size between 2, 5, and 10. Note that the selection

process' pool size is only relevant when tournament selection is the parent selection algorithm; Pareto selection and roulette selection processes do not depend on pools of genomes for parent selection.

Question 3: *What is the effect of fitness weights on the quality of the final test suite?*

To evaluate the effects of varying the fitness weights on the resulting test suites, I ran experiments with the following pairs of fitness weights, keeping other parameters at the baseline measure:

$$(cvg_weight,\ len_weight) \in \{(1,1),\ (1,2),\ (2,1),\ (1,5),\ (0,1),\ (1,0)\}.$$

Question 5: *How does the genetic algorithm approach to generate test suites for web applications compare to using the hill-climbing algorithm to generate test suites?*

To compare the genetic algorithm and the hill-climbing approaches to generating test suites, I implemented a hill-climbing framework that, much like the genetic algorithm framework, initializes a user session pool from parsed user sessions and then creates a random parent genome from the user session pool. It then performs a mutation to the genome at every iteration and calculates the fitness of the child genome—using the same fitness function as the genetic algorithm framework. If the fitness of the child genome is greater than the fitness of the parent genome, the child is appointed as the new parent; otherwise, the parent remains the same for the next iteration.

For each subject user session set, I ran hill-climbing experiments with fitness weights $w_1 : w_2$ of $1 : 5$ for a maximum of 10,000 iterations. Assuming that a genetic algorithm population of 50 genomes is run for 200 generations (i.e., the baseline measures are used for the genetic algorithm framework experiments), an average of $50 \times 200 = 10,000$ genetic operations will be performed on this population over the course of the evolutionary run. This is computationally equivalent to performing one mutation on one genome for 10,000 iterations, to allow for fair comparison between the two approaches to generating test suites.

5.3.4 Analyzing the Resulting Test Suites

To analyze the test suites output from the genetic algorithm framework, I used the metrics discussed in Section 5.3.2 to assess the quality of the test suites. For each experiment:

1. I *computed the length and RRN coverage values for each genome in a population for each generation.* These values are written to two their respective .csv files and then subsequently used to generate graphs to analyze the results. I plot the length and RRN coverage values against the corresponding generation number to understand how the genetic algorithm framework prioritizes lower length over higher RRN coverage values as the fitness weights change.

2. I *computed the fitness value, as defined by the fitness function, for each genome in a population for each generation.* Like the RRN coverage and length values for populations, the fitness values are also written out to a .csv file. I then generate the graph of the overall fitness values of a population against the number of generations. While the fitness graphs are less telling in terms of how the quality of the genomes in a population changes as the evolutionary run proceeds, they do provide a good visualization of the overall gains in fitness accomplished by the genetic operators.

It is important to note that the magnitude of the fitness value is not significant in its own right; it is only used to compare the fitness of one genome over another. For example, using large values for the fitness function weights w_1 and w_2 results in a large magnitude of the fitness values for each genome, but this magnitude does not imply that the genomes are more fit than, say, the genomes whose fitness values are obtained by using smaller values for the fitness weights. For this reason, the fitness graphs for a population are less telling than the corresponding length and RRN coverage graphs for the population.

5.3.5 Evaluating the Resulting Test Suites

I designed and implemented an evaluation framework (see Chapter 4) to assess the quality of the resulting test suites. The evaluation framework executes a suite of test cases on a locally deployed, instrumented version of the web application to measure the code coverage of the test suite on the web application. The design and

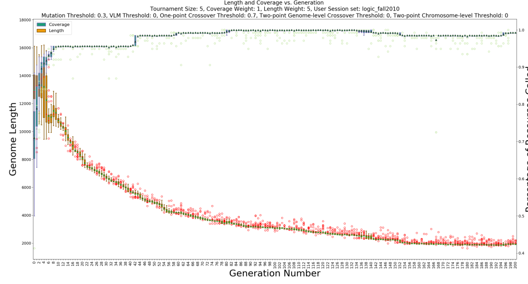
implementation specifics of the evaluation framework are discussed in detail in sections 4.1 and 4.2.

For each experiment that I run, I output the best genome in the population–in terms of fitness values, as selected by the fitness function–after the last generation as a test suite for the web application. This is done by looping through the chromosomes of the genome, and for each chromosome, every gene belonging to that chromosome is written to a .tc file as a user request. As such, each chromosome is output as a .tc file, and the collection of all .tc files output serves as the executable test suite for evaluation purposes. This executable test suite is then executed on the instrumented web application and the code coverage data are recorded. These code coverage data can be reported in HTML or XML formats, depending on the user's requirements.

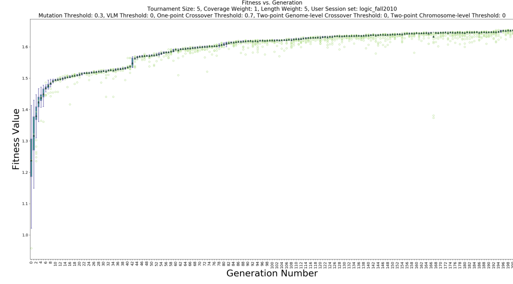
Since I run a number of experiments to generate test suites, varying a particular parameter for each experiment, there were a large number of test suites that could be evaluated on the evaluation framework. However, tuning the various parameters of the genetic algorithm framework yields different results. Due to the numerous combinations of parameters, I chose to focus on the subset of the experiments that yielded the best test suites in terms of the RRN coverage and length metrics, and then evaluate this subset of the test suites on the evaluation framework.

5.4 Threats to Validity

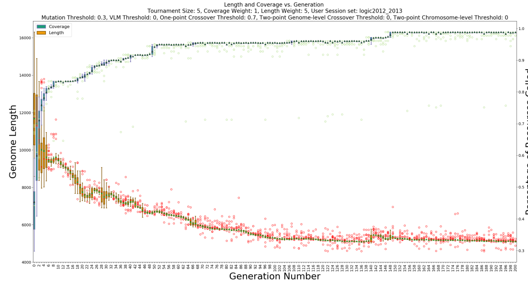
A potential source of threats to validity in my experimental results would be the relatively small size of the subject applications. While the applications are not *small* per se, they do not compare well to the size–both in terms of the code and the user base–of the larger, more widely used web applications out there, e.g. Kelley Blue Book, Ebay, etc. Additionally, the number of unique requests across the subject applications was small, implying that there is a large amount of redundant code between the versions of the application. To combat this threat, I will evaluate a larger web application with a wider user base in my future work.



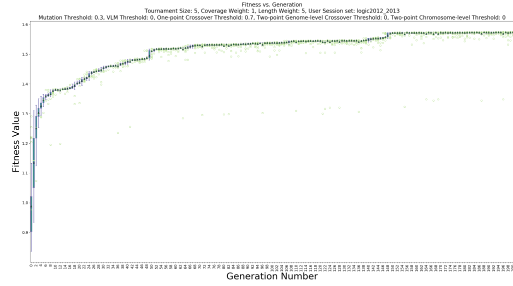
(a) Len-Cvg Graph for logic_fall2010



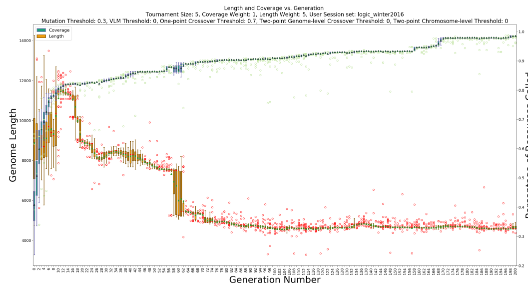
(b) Fitness Graph for logic_fall2010



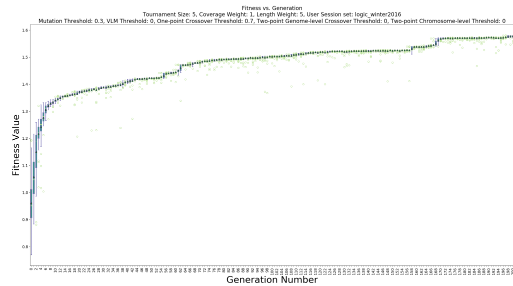
(c) Len-Cvg Graph for logic_2012_2013



(d) Fitness Graph for logic_2012_2013



(e) Len-Cvg Graph for logic_winter2016



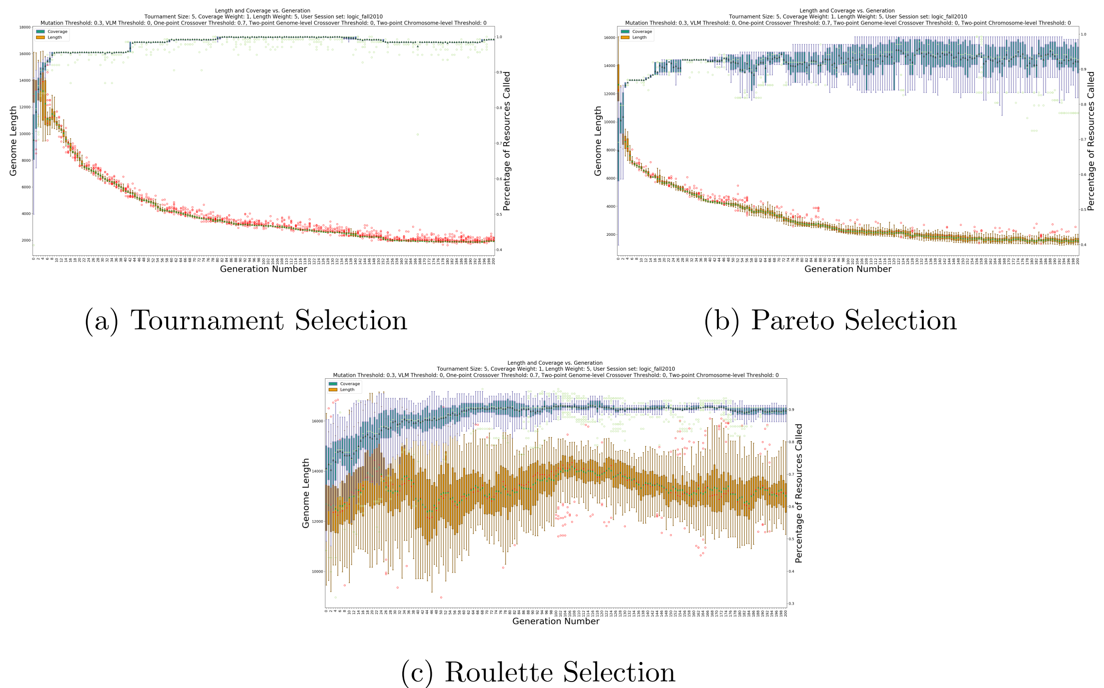
(f) Fitness Graph for logic_winter2016

Figure 5.1: Baseline Graphs for the Generated Test Suites. The x-axis is the generation number. The y-axes for the graphs in the left column are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis), and the y-axis for the graphs in the right column is the test suite's fitness.

5.5 Results

5.5.1 Generating Test Suites

Figure 5.1 shows the graphs for the test suites generated using the baseline values for the genetic algorithm framework parameters. Each graph uses box plots to illustrate the distribution of genome length and the RRN coverage values across the population for every generation. Each box represents the genomes between the 25th



(a) Tournament Selection

(b) Pareto Selection

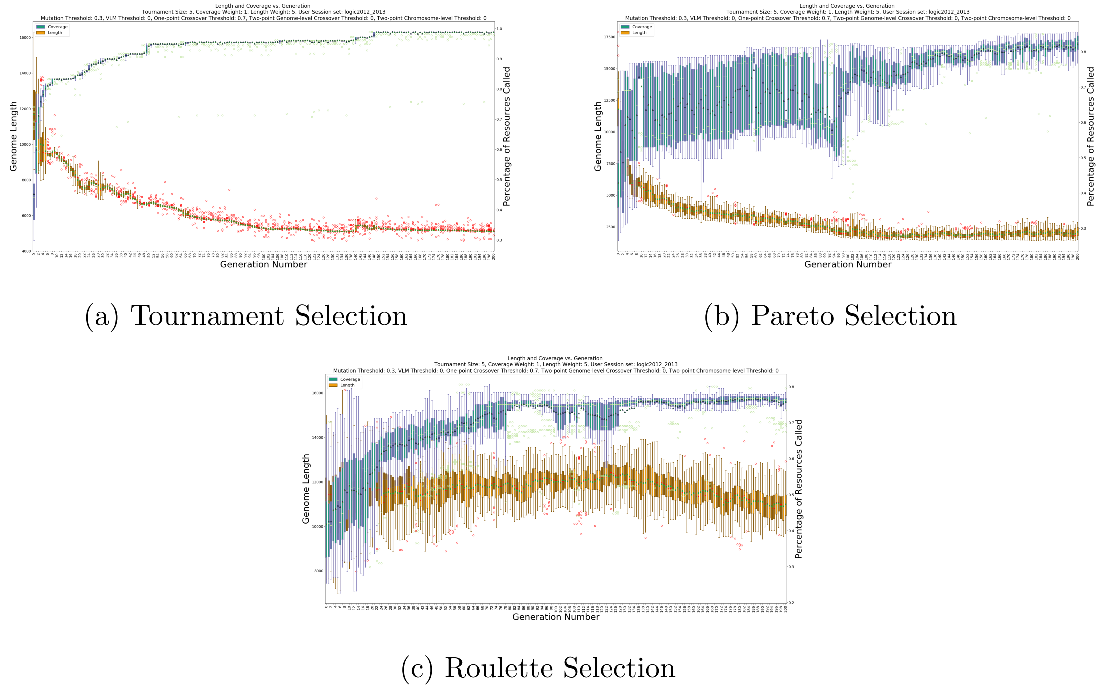
(c) Roulette Selection

Figure 5.2: A Comparison Parent Selection Algorithms for logic_fall2010. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis).

and 75$^{\text{th}}$ percentile for each heuristic. The whiskers show the range of all the values but the outliers, which are marked with circles. The left column of plots shows the distribution of length (in orange) and RRN coverage (in green) for each genome in a population for every generation. For each length and RRN coverage box plot, the x-axis represents the number of generations. For the graphs in the left column, the left and right y-axes represent the genome length and RRN coverage as a percentage, respectively. These graphs provide a template for comparing with the resultant test suites of other generation techniques.

5.5.1.1 Comparing Parent Selection Algorithms

To answer Question 1, I analyze the results from different parent selection algorithms, depicted in figures 5.2 through 5.4.



(a) Tournament Selection

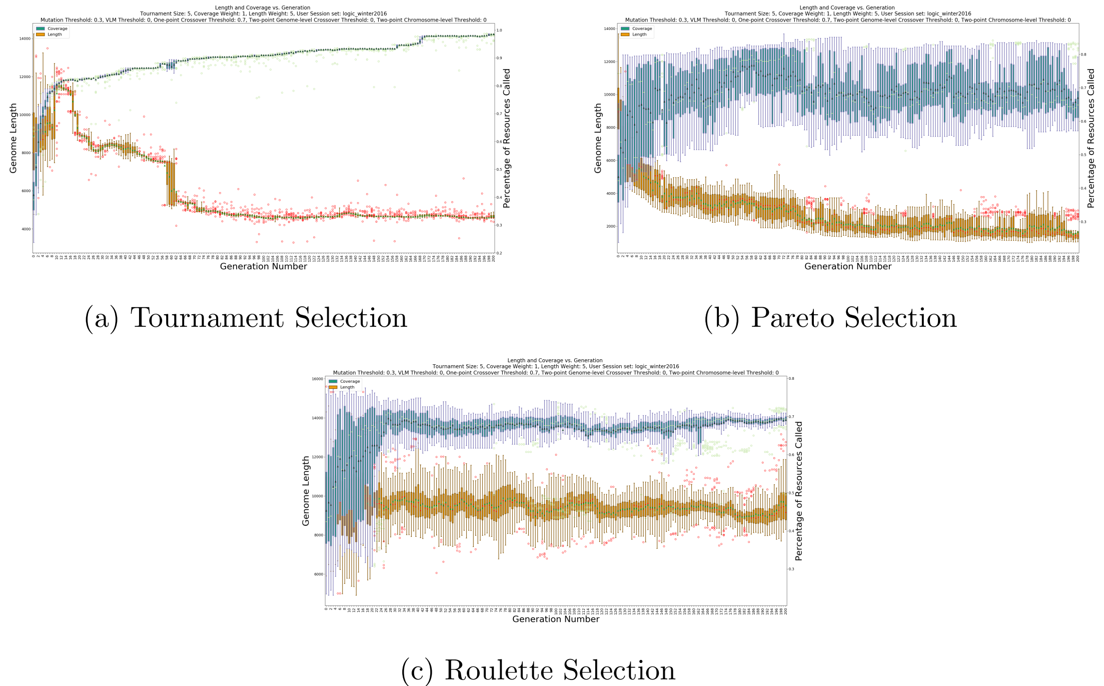
(b) Pareto Selection

(c) Roulette Selection

Figure 5.3: A Comparison Parent Selection Algorithms for logic_2012_2013. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis).

As illustrated by the trends in the graphs, tournament selection generally performed better than Pareto and roulette selection, reaching a higher RRN coverage and a lower length when compared to the other parent selection algorithms. For instance, for the logic_2012_2013 user session set, tournament selection got close to 96% RRN coverage on average, while Pareto selection and roulette selection got approximately 82% and 76% on average respectively. Similar trends were observed across the other user session sets. Roulette selection also had the worst performance across all subject user session sets when it comes to the lengths of the genomes across populations.

The box plots show a tendency of Pareto selection to result in populations with a higher spread of RRN coverage values, and a tendency of roulette selection to result in populations with a higher spread of length values. These higher spreads can be attributed to the selection algorithms' contributions towards maintaining a higher genetic diversity among populations.



(a) Tournament Selection

(b) Pareto Selection

(c) Roulette Selection

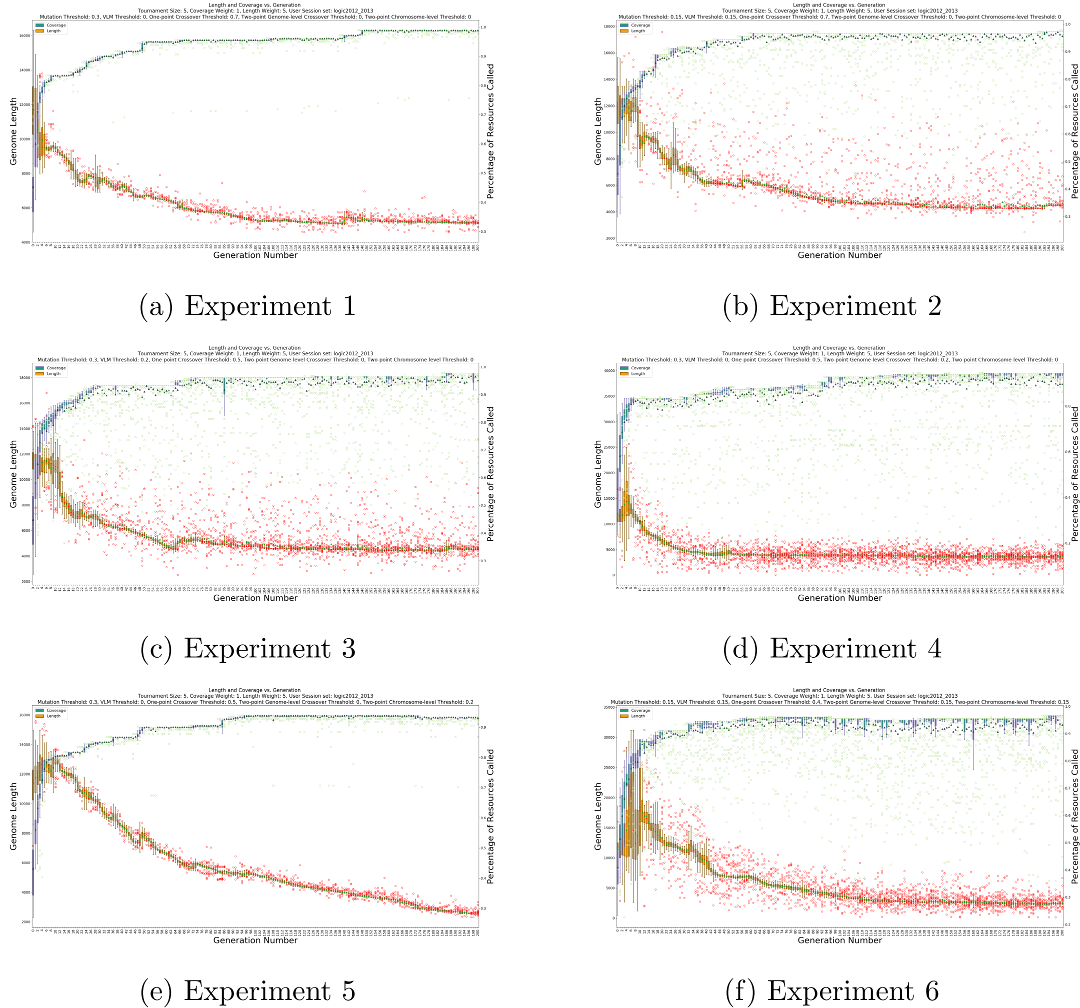
Figure 5.4: A Comparison Parent Selection Algorithms for logic_winter2016. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis).

5.5.1.2 Comparing Genetic Operators and Genetic Algorithm Parameters

To answer question 2, I analyze the results from the experiments varying the genetic operator thresholds (see table 5.3).

Figure 5.5 shows the resulting graphs for logic_2012_2013. The other subject user session sets followed similar trends.

As indicated by figures 5.5b, 5.5c, and 5.5f, the use of variable length mutation significantly increases the spread of the RRN coverage and length values across populations. This increase in spread shows up as red and green dots on the box plots as "outliers" to the data. These results are consistent with my initial hypothesis: that variable length mutation provides greater potential to increase genetic diversity that regular mutataion. The lack of large spreads of data for experiments corresponding to figures 5.5a, 5.5d, and 5.5e could be attributed to the use of genetic operators that do little to maintain genetic diversity.



(a) Experiment 1

(b) Experiment 2

(c) Experiment 3

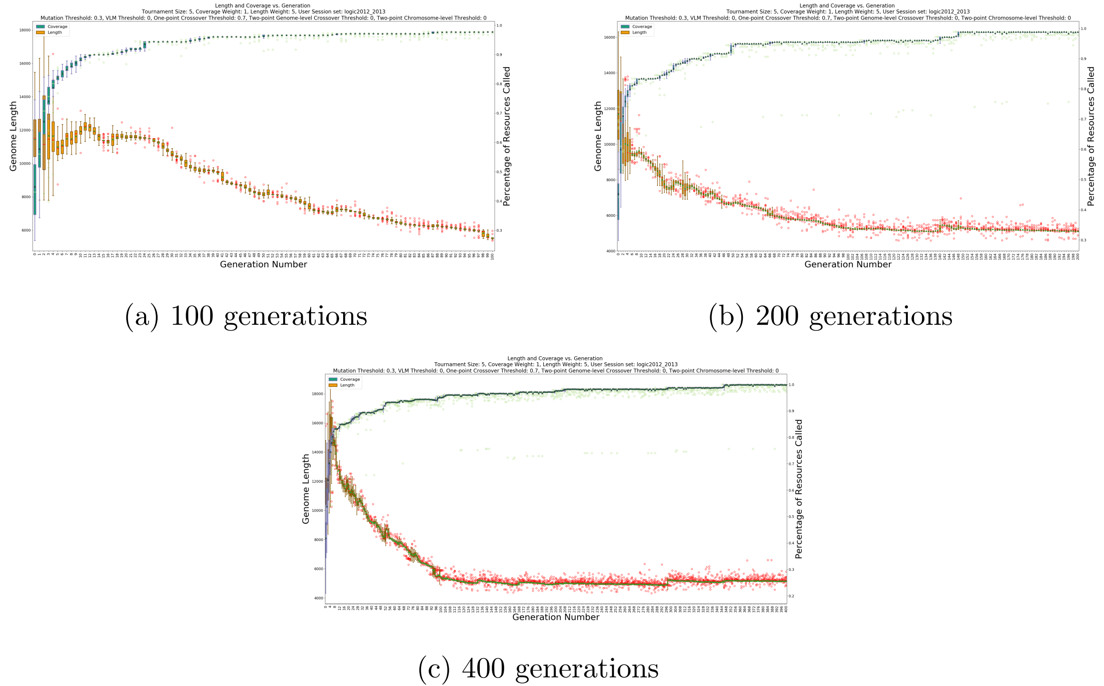
(d) Experiment 4

(e) Experiment 5

(f) Experiment 6

Figure 5.5: A Comparison Genetic Operator Thresholds for logic_2012_2013. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis).

Figure 5.5d shows that using two-point genome level crossover generally results in slightly smaller values of genome length across the populations. This is because the fitness function will naturally favor genomes in which the crossover replaces a large chromosome with a slightly smaller one, ideally not at the expense of losing RRN coverage. However, it is likely that in the process of replacing larger chromosomes with smaller ones, chromosomes that access a certain resource not accessed by other chromosomes might be replaced. As such, it was often observed that the use of two-point chromosome-level crossover resulted in test suites with smaller lengths at the



(a) 100 generations

(b) 200 generations

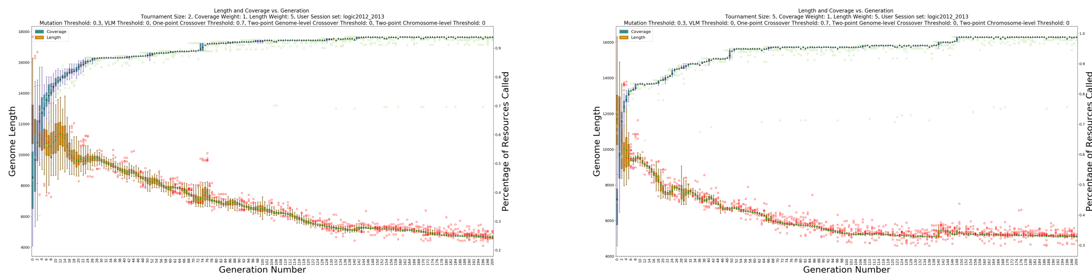
(c) 400 generations

Figure 5.6: A Comparison the Maximum Number of Generations for logic_2012_2013. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis).

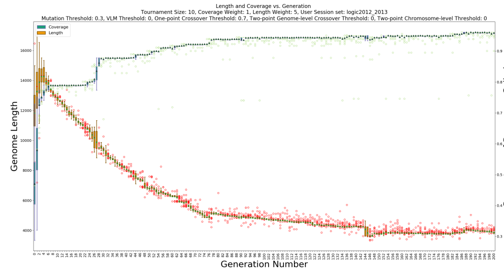
expense of slightly lower RRN coverage. Figure 5.5e shows that the use of two-point chromosome-level crossover does not do much to produce a test suite with better RRN coverage compared to a test suite produced using the baseline measures. It should be noted that the key idea behind the use of two-point chromosome-level crossover is to create new gene sequences, i.e. chromosomes, not already present in the user session pool. This introduces the potential for greater code coverage when the resulting test suite is evaluated on the evaluation framework.

I also analyze the results from the experiments varying the maximum number of generations. Figure 5.6 displays the results for these experiments on logic_2012_2013; the other subject user session sets exhibited similar trends.

As is evident in the graphs, running the genetic algorithm for a greater number of generations results in test suites that not only have higher RRN coverage but also have a significantly smaller length.



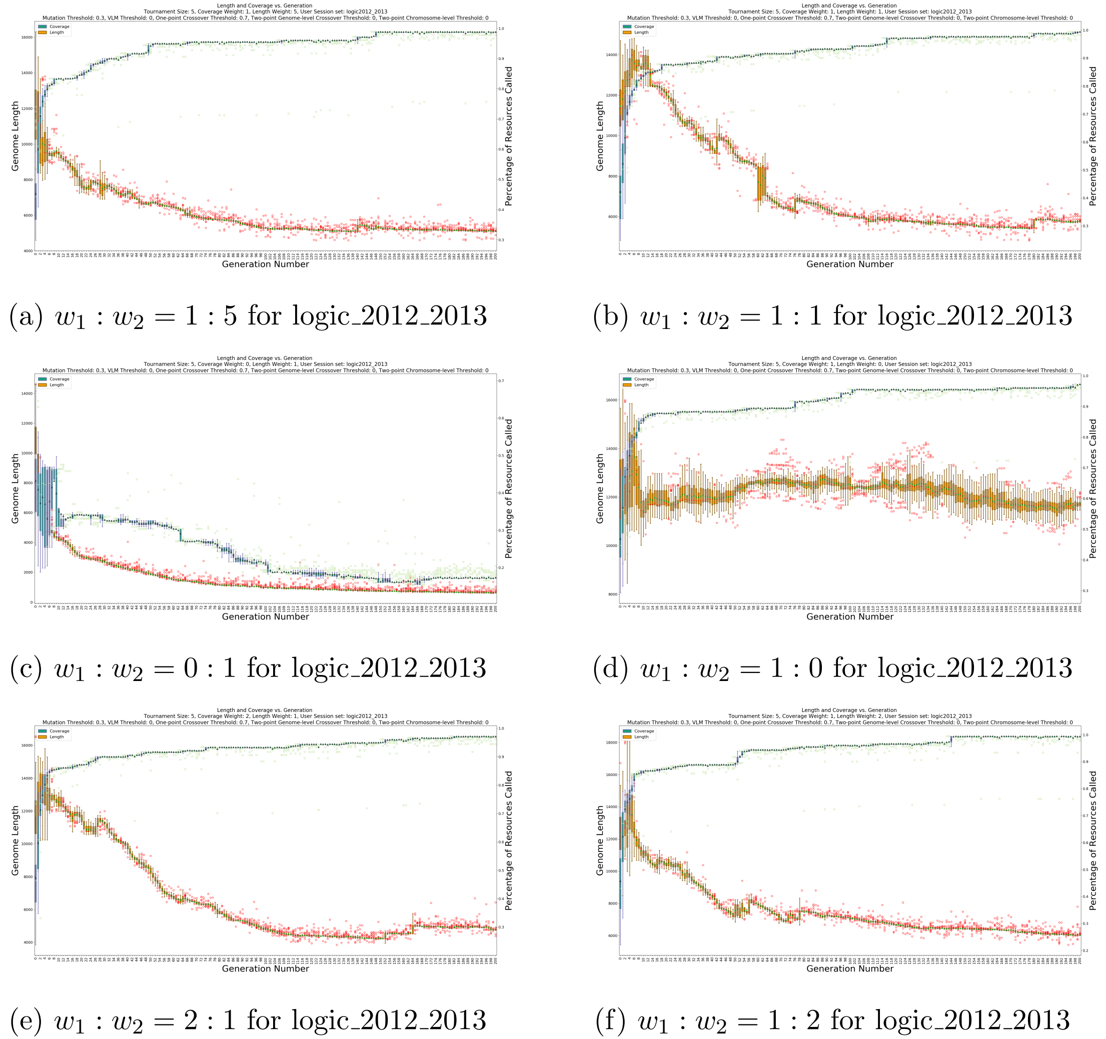
(a) Tournament Size = 2 (b) Tournament Size = 5



(c) Tournament Size = 10

Figure 5.7: A Comparison the Tournament Sizes for logic_2012_2013. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis).

Furthermore, the results from the experiments varying the tournament size for tournament selection were analyzed. Figure 5.7 shows the results for these experiments on logic_2012_2013. Other subject user session sets followed similar trends. Figures 5.7a and 5.7c show that as the tournament size gets larger, the spread of the length and RRN coverage values across populations decreases. With a large tournament pool for parent selection, the likelihood for selecting the same parent multiple times increases, since the chances of this parent's presence in the pool are higher. This in turn decreases the genetic diversity of the populations since the same parents are allowed to produce multiple children. Since a tournament pool size of 2 increases this spread of values too much, while a pool size of 10 decreases this spread too much, a pool size of 5 appears to be a reasonable compromise.



(a) $w_1 : w_2 = 1 : 5$ for logic_2012_2013

(b) $w_1 : w_2 = 1 : 1$ for logic_2012_2013

(c) $w_1 : w_2 = 0 : 1$ for logic_2012_2013

(d) $w_1 : w_2 = 1 : 0$ for logic_2012_2013

(e) $w_1 : w_2 = 2 : 1$ for logic_2012_2013

(f) $w_1 : w_2 = 1 : 2$ for logic_2012_2013

Figure 5.8: A Comparison Fitness Weights for logic_2012_2013. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis).

5.5.1.3 Comparing Fitness Weights

To answer question 3, I analyze the results from different experiments varying the fitness weights. Figure 5.8 shows the box plot graphs for these experiments. Similar trends were observed for the experiments for logic_fall2010 and logic_winter2016.

Figure 5.8c exhibits the length and RRN coverage trends for $w_1 : w_2 = 0 : 1$. As is evident in the graph, using a fitness weight $w_1 = 0$, which is equivalent to not considering the RRN coverage of a genome when calculating its fitness, results in a

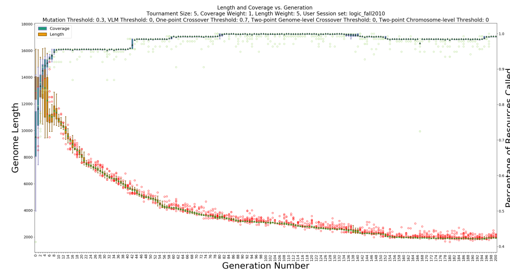
final test suite that, while having a small length, does little to cover the resources of the application. As such, the test suite would be fairly ineffective when it comes to testing capabilities and exposing faults, since a large proportion of the application's resources would be left untested. Similarly, Figure 5.8d, which exhibits the length and RRN coverage trends for $w_1 : w_2 = 1 : 0$, shows that not considering the genome length when calculating fitness values results in a final test suite that, while having a high RRN coverage, also has a large number of test cases.

Figures 5.8b, 5.8e, and 5.8f exhibit the length and RRN coverage trends for $w_1 : w_2$ values of $1 : 1$, $2 : 1$, and $1 : 2$ respectively. As is evident in the box plot graphs, trying to balance $w_1 : w_2$ such that slightly higher emphasis is put on length results in test suites that have smaller length with little to no compromise on the RRN coverage.

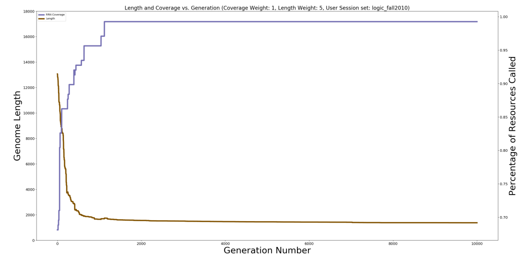
5.5.1.4 Comparing the Genetic Algorithm to the Hill Climbing Algorithm for Generating Cost-Effective Test Suites

To answer question 5, for each subject user session set, I compare the results from the experiments corresponding to the baseline parameters for the genetic algorithm to the results from running the hill-climbing algorithm for 10,000 iterations. Since the baseline parameters run the genetic algorithm on a population of 50 genomes for 200 generation, this approximately results in 10,000 genetic operations made on the genomes. In contrast, the hill-climbing algorithm is allowed to run for 10,000 iterations to allow for computational fairness.

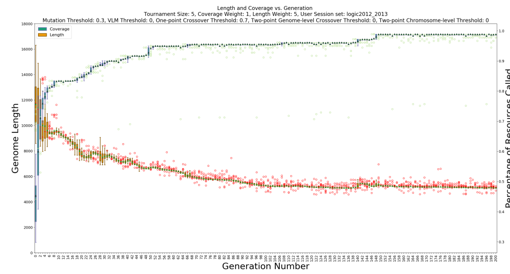
Figure 5.9 compares the length and RRN coverage graphs for test suite generation by the genetic algorithm and the hill-climbing algorithm approaches. The graphs corresponding to the hill-climbing framework are line graphs, and not box plots, since the hill-climbing algorithm on only one genome per iteration, and not a population of genomes. Figures 5.9b, 5.9d, and 5.9f indicate that the hill-climbing algorithm rapidly converges to a genome with a small length, and then works around this genome to get a higher RRN coverage by performing mutations. By contrast, the genetic algorithm



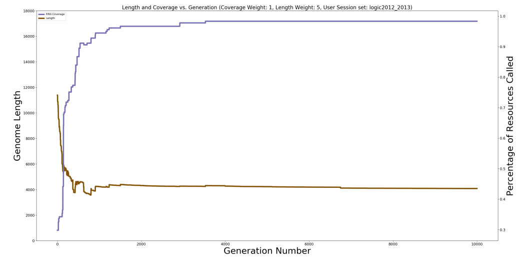
(a) GA Framework for logic_fall2010



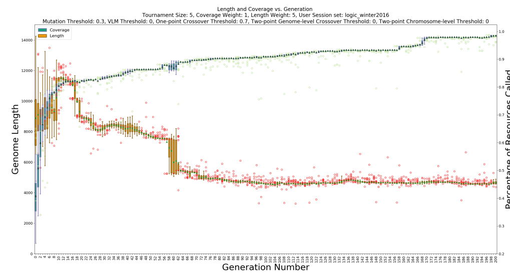
(b) HC Framework for logic_fall2010



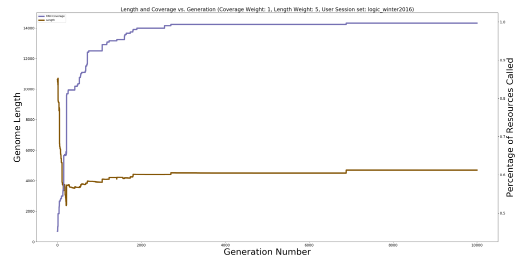
(c) GA Framework for logic_2012_2013



(d) HC Framework for logic_2012_2013



(e) GA Framework for logic_winter2016



(f) HC Framework for logic_winter2016

Figure 5.9: A Comparison of Test Suite Generation by the Genetic and the Hill-Climbing Algorithms. The x-axis is the generation number. The y-axes are test suite's number of requests (left y-axis) and percent RRN coverage (right y-axis).

converges to a population with smaller length genomes gradually, as indicated by figures 5.9a, 5.9c, and 5.9e. The biggest drawback of the hill-climbing algorithm is its likelihood of getting stuck in local optima and not being able to guide the solution space towards a global optimum. This is consistent with the results that I observe for hill-climbing experiments, where once the algorithm converges to a genome with a small length, it is unable to reverse this change in search of a genome that would allow for 100% RRN coverage. As a result, I did not observe any hill-climbing experiments

attain 100% RRN coverage across all subject user session sets. On the other hand, with the genetic algorithm, getting 100% RRN coverage was possible for each subject, provided that the parameters were tuned.

Table 5.4: Original Test Suite Coverage

Subject	RRN Coverage	Length	Code Coverage
logic_fall2010	100%	49431	51.9%
logic_2012_2013	100%	122149	58.0%

5.5.2 Evaluating Test Suites

To answer question 4, I assess the performance of a range of the test suites from the generation phase worth evaluating on the evaluation framework. As mentioned in section 5.3.5, I chose to not execute *all* generate test suites on the evaluation framework due to time constraints. This evaluation approach suffices to answer the research question adequately: it provides a comparison between the code coverage results obtained from lower quality test suites, average quality test suites, and high quality test suites. Note that the generated test suites with a low RRN coverage were not evaluated on the framework, since they fail to even access a majority of the resources of the application, and as such, would not make for good test suites.

I first executed the entire user session sets separately on the evaluation framework to get a benchmark for the code coverage metric. Table 5.4 shows the overall code coverage obtained by executing all test cases present in the user session for each user session set. The generated test suites are then evaluated and the resulting code coverage compared with the benchmark values. To answer the research question, for each test suite being evaluated, I consider the RRN coverage, length, and code coverage of the test suite. I use these values to subsequently calculate the *length reduction*, the percentage decrease in the length of the test suite when compared to the original user session set, and the *code coverage reduction*, the percentage decrease in the code coverage of the test suite when compared to the benchmark coverage, of the test suite. Table 5.5 shows the evaluation results for the test suites generated by the genetic algorithm, while table 5.6 shows the evaluation results for the test suites generated by the hill climbing algorithm run for 10,000 iterations with fitness weights of 1 : 5.

For the logic_fall2010 user session set:

Experiment	RRN Coverage	Length	Code Coverage	Length Reduction	Code Coverage Reduction
1	91.30%	11473	45.0%	76.79%	13.29%
2	98.55%	5729	45.6%	88.41%	12.14%
3	99.27%	1459	49.3%	97.00%	5.00%
4	100%	2187	50.5%	95.57%	2.70%

(a) logic_fall2010 on Logic2

Experiment	RRN Coverage	Length	Code Coverage	Length Reduction	Code Coverage Reduction
1	89.50%	5449	44.9%	95.54%	22.58%
2	93.92%	7318	48.3%	94.01%	16.72%
3	100%	5144	53.0%	95.79%	8.62%

(b) logic_2013_2013 on Logic5

Table 5.5: Coverage for Test Suites Generated by the Genetic Algorithm

Table 5.6: Coverage for Test Suites Generated by the Hill Climbing Algorithm

Subject	RRN Coverage	Length	Code Coverage	Length Reduction	Code Coverage Reduction
logic_fall2010	99.27%	1377	49.1%	97.21%	5.39%
logic_2012_2013	98.34%	4085	47.5%	96.66%	18.10%

- Experiment 1 corresponds to the test suite generated using the baseline parameters with roulette selection as the parent selection algorithm. These parameters result in a test suite with both a relatively lower RRN coverage and a relatively larger length.

- Experiment 2 corresponds to the test suite generated using the baseline parameters with a maximum of 50 allowed generations for the evolutionary run. While these parameters result in a final test suite with a relatively higher RRN coverage, the test suite also has a moderately large length.

- Experiment 3 corresponds to the test suite generated using the baseline parameters with a maximum of 400 allowed generations for the evolutionary run. These parameters not only result in a final test suite with a very high RRN coverage, but also a test suite with a very small length.

- Experiment 4 corresponds to the test suite generated using the baseline parameters with fitness weights of 1 : 1 for the evolutionary run. These parameters result in a final test suite with the highest possible RRN coverage and a relatively small length.

For the logic_2012_2013 user session set:

- Experiment 1 corresponds to the test suite generated by running the genetic algorithm for 100 generations, with fitness weights $w_1 : w_2 = 1 : 1$, and the other parameters at the baseline measure. While these parameters result in a test suite with a small length, the test suite also has a relatively poor RRN coverage.

- Experiment 2 corresponds to the test suite generated using the baseline parameters with a maximum of 50 allowed generations for the evolutionary run. These parameters result in a test suite with a relatively high RRN coverage but a relatively larger length.

- Experiment 3 corresponds to the test suite generated using the baseline parameters with a maximum of 400 allowed generations for the evolutionary run. These parameters result in a final test suite with the highest possible RRN coverage of 100% and a very small length.

As is evident by the results shown in tables 5.5 and 5.6, both approaches to generating test suites for web applications are effective in terms of reducing the number of test cases while keeping most of the testing functionality. For instance, table 5.5a shows that for experiment 4, the length of the test suite was reduced to 2,187 down from 49,431 while the code coverage obtained by the two test suites went down from 51.9% on the larger test suite (i.e., the entire user session set) to 50.5% on the generated test suite. Similarly, table 5.5b shows that for experiment 3, the length of the test suite was reduced to 5,144 down from 122,149 while the code coverage obtained by the two test suites went down from 58.0% on the larger test suite to 53.0% on the generated test suite. Both of these results small losses in code coverage–and hence, testing efficacy–for enormous decreases in the number of test cases.

Test suites generated using the hill-climbing approach tended to get lower code coverage when compared to test suites generated using the genetic algorithm. This difference can be attributed to the hill-climbing algorithm's tendency to converge on a genome with a small length, and failing to ever achieve 100% RRN coverage in the process. The genetic algorithm, by contrast, was able to get 100% RRN coverage for all subject user session sets.

5.6 Discussion

The overarching question that I hoped to answer with my experimental study was: *can the genetic algorithm be used to generate cost-effective test suites for web applications*? Due to the nature of the genetic algorithm, any output from a genetic algorithm application is subject to randomness–much like the output from hill-climbing

algorithm applications. However, which the hill-climbing algorithm ensures that the fitness of a genome never gets worse from one iteration to the next, the genetic algorithm has no such checks in place. With a genetic algorithm framework, it is very much possible that the fitness of the genomes in a population gets progressively worse. Therefore, it is imperative that the right combination of genetic operator thresholds and genetic algorithm parameters (including fitness weights) be used to encourage the fitness of genomes in a population to increase, or at worst, stays the same. Note that even then, there is no guarantee that the genetic algorithm with lead the populations towards a higher average fitness.

However, during my experimental study, I discovered that it was never the case that the average fitness of a population of genomes, for experiments run on a subject user session set, always decreases as the evolution proceeds. If certain combinations of the genetic algorithm parameters failed to yield desirable results, it was invariably the case that changing the parameters resulted in better results that were comparable to, if not better than, the results obtained from the hill-climbing approach.

5.6.1 Comparing Parent Selection Algorithms

My experimental results indicate that roulette selection tends to have the worst performance across all subject user session sets. While it is the case that roulette selection–much like Pareto selection–significantly increased the genetic diversity across populations, it generally resulted in both lower RRN coverage and larger lengths in the resulting test suites. On the other hand, Pareto selection performed significantly better in terms of the length of the genomes across populations, often outperforming tournament selection in the length metric. Tournament selection, however, was undoubtedly the best in getting higher RRN coverage–a trait that is the most desirable in a good quality test suite. Pareto selection will likely outperform tournament selection if more than two objectives are being optimized, since Pareto selection is inherently a multi-objective optimization parent selection algorithm. However, for the current

implementation of the genetic algorithm framework, tournament selection seems to exhibit the most promising results.

5.6.2 Comparing Genetic Operators and Genetic Algorithm Parameters

My experimental results indicate that, while there is no singleton set of genetic operator threshold values that works to generate the best quality cost-effective test suite across all subject user session sets, there are trends that can be extrapolated. For example, the use of variable length mutation significantly increases the spread of RRN coverage and length values across populations, i.e., increases the genetic diversity of the populations in an evolutionary run. The greater the threshold of variable length mutation, the greater the increase in genetic diversity of populations. Every other genetic operator does little to diversify the population in an evolutionary run. The use of two-point genome-level crossover generally decreases the genome lengths across populations by swapping larger chromosomes in a genome with smaller ones—often at the expense of losing RRN coverage slightly. The use of two-point chromosome-level crossover does little to produce a test suite with either a greater RRN coverage or a shorter length, compared to the baseline measures. However, its potential of introducing new gene sequences not present in the user session pool is reason enough to not discard two-point chromosome-level crossover from being considered a worthy genetic operator. It just means that smaller threshold values for two-point chromosome level crossover should be used in an evolutionary run.

On the contrary, my experimental study demonstrates that there are certain genetic algorithm parameters that appear to yield remarkable results across all subject user session sets. For instance, from my experiments varying the maximum number of generations while keeping the other parameters at the baseline measure, it was evident that running the genetic algorithm for a larger number of generations invariably increases the quality–both in terms of RRN coverage and genome length–of the resulting test suite (Figure 5.6). Furthermore, my experiments varying the tournament pool

size[3] clearly indicated that as the tournament size gets larger, the spread of the length and RRN coverage values decreases (Figure 5.7). This is because with larger tournament pools, the likelihood of the same parent's being selected for reproduction multiple times increases, and this in turn decreases the diversity among the populations. The tournament size should be considered in conjunction to the number of genomes in a population. If a larger number of genomes are present in a population, a larger tournament pool size can be used. However, I observed that with 50 genomes per population, a tournament pool size of 5 resulted in the best quality test suites.

5.6.3 Comparing Fitness Weights

Based on the results from my experiments varying the fitness weights while keeping the other genetic algorithm parameters at the baseline measure, it appears that using a fitness weight $w_1 = 0$, i.e. not considering the RRN coverage of genomes, when running the genetic algorithm results in test suites that, while having small lengths, have very poor RRN coverage. Such test suites would be fairly ineffective at exposing faults in the web application, because they would fail to even access most of the resources of the web application. On the contrary, using a fitness weight $w_2 = 0$, i.e. not considering the length of the genomes, when running the genetic algorithm results in test suites that have high RRN coverage values but large length values. Across all of my experiments, I found out that trying to balance $w_1 : w_2$ such that a slightly higher emphasis is put on length results in test suites that have smaller length with often no compromise on the RRN coverage (Figure 5.8). As such, fitness weights of $w_1 : w_2 = 1 : 5$ appear to have the best performance across all subject user session sets.

5.6.4 Evaluating the Performance of the Generated Test Suites

The results from my evaluation experiments were instrumental in answering the question of whether the genetic algorithm can be used to generate cost-effective test

---

[3] Tournament sizes are only applicable to evolutionary runs where tournament selection is the parent selection algorithm.

suites for web applications. My experimental results from the evaluation phase clearly indicate that while both approaches to generating test suites for web applications are very effective when it comes to reducing the number of test cases while minimizing losses in testing functionality, the best test suites generated using the genetic algorithm outperformed the best test suites generated using the hill-climbing algorithm. This difference could be attributed to the hill-climbing algorithm's failure to ever achieve 100% RRN for a subject user session set. I was able to substantially reduce the number of test cases in a test suite using the two approaches with minimal decreases in code coverage, indicating that both approaches can successfully generate cost-effect test suites. Furthermore, it can easily be concluded that the genetic algorithm produces test suites that perform better at code coverage when executed on the subject application.

5.6.5 Comparing the Genetic Algorithm to the Hill Climbing Algorithm for Generating Cost-Effective Test Suites

My experimental results indicate that hill-climbing algorithm rapidly converges to a genome with a small length, and then works around this genome with mutations to get a higher RRN coverage. On the other hand, the genetic algorithm gradually explores the search space and converges to populations with a small length. These results indicate that the hill-climbing algorithm is very prone to getting stuck in local optima and failing to reach a global optimum while exploring a variety of genetic compositions. Once the hill-climbing algorithm has fallen into a local optimum, it is impossible to recover from this change, since previous genomes–along with their genetic information–are discarded from one iteration to the next. As a result, the hill-climbing algorithm failed to achieve 100% RRN coverage across all subject user session sets. This was also reflected in the relatively poorer performance of hill-climbing generated test suites when the test suites were evaluated on the evaluation framework. In contrast, the genetic algorithm succeeded in achieving 100% RRN coverage for every subject for particular combinations of the genetic algorithm parameters.

5.7 Recommendations for Testers

An analysis of the results from my experiments allows me to understand the general trends across genetic algorithm populations and recommend the best parameters to tune the genetic algorithm for cost-effective test suite generation. For future testers using this framework to generate and evaluate test suites for a subject application, I have the following recommendations:

- If you value high RRN coverage in your resulting test suite more than a small length, use tournament selection as the parent selection algorithm, since it outperforms both Pareto and roulette selection. If you value a small length in your resulting test suite more than high RRN coverage, use Pareto selection as the parent selection algorithm. Furthermore, if tournament selection is used, use a tournament pool size that is approximately $\frac{1}{10}$ times the size of the population, as this allows for the ideal amount of genetic diversity across populations.

- Allowing the evolutionary process to run for a large number of generations almost always results in a higher-quality test suite, both in terms of length and RRN coverage. However, you must exercise caution when increasing the number of generations, as each increase of 100 generations on a population size of 50 results in $50 \times 100 = 5,000$ additional genetic operations. There is a practical limit to the number of generations allowed, depending on the machine executing the genetic algorithm, because these genetic operations can be processing- and memory-intensive. Since an increase in the number of generations by a factor of 2 requires a decrease in the population size by a factor of 2, the right balance needs to be maintained between the number of generations and the population size. My experimental results suggest that a population of 50 genomes be allowed to evolve for 200 generations to produce a good quality test suite and for 400 generations to produce an exceptional quality test suite. Since test suite generation is often not time-sensitive, running the genetic algorithm for 400 generations with a population of 50 genomes is recommended.

- Unfortunately, there are no universal values of genetic operator thresholds that perform well across all subjects. I recommend trying several different combinations until a combination that produces a good quality test suite is found. Often, a mutation threshold of 0.3 and a crossover threshold of 0.7 is a good starting point and will result in a decent test suite. If the initial evolutionary run does not result in a test suite with high RRN coverage, I recommend increasing the threshold for variable length mutation to allow for greater genetic diversity among populations. If the initial evolutionary run results in a test suite with a large length, I recommend increasing the threshold of two-point genome-level crossover. Keep in mind, however, that too large a value of the two-point

genome-level crossover threshold, while producing a shorter test suite, can hurt the RRN coverage of the test suite.

- For the fitness weights, I recommend using a ratio of $0.1 \leq w_1 : w_2 \leq 1$. In other words, use fitness weights that provide less emphasis on RRN coverage compared to length. While this may seem counter-intuitive at first—a high RRN value is arguably of utmost importance in a test suite—my experimental results suggest that using a higher value of w_2 than w_1 results in test suites that have smaller length with little to no compromise on RRN coverage. You must be careful not to make w_2 too large with respect to w_1 because that could indeed result in a test suite with a short length but a poor RRN coverage. I recommend using $w_1 : w_2 = 1 : 5$ to get the optimal performance out of the genetic algorithm.

- For evaluating test suites, I recommend finding parameters to the genetic algorithm that provide approximately 100% RRN coverage, even if that is at the expense of a slightly larger test suite. A good test suite should at least cover most, if not all, of the resources of an application. It is usually not worth evaluating test suites with a low RRN coverage, since such test suites will almost definitely perform poorly in terms of code coverage.

- If you are getting poor code coverage on test suites that have relatively high RRN coverage, the web application might be experiencing persistent state errors. Check the log files for the web application server to see if any exceptions that indicate persistent state (e.g., database) errors are reported. Additionally, for database errors, if the evaluation framework indicates that a certain request in a test case involving an entry in the database resulted in an error page, you could manually navigate the database to ensure that the entry being referenced is present in the database. Provided that the persistent state of the application is initialized correctly, a generated test suite with a high RRN coverage should result in code coverage that is comparable to the original set of user sessions used by the genetic algorithm to produce the test suite.

Chapter 6

CONTRIBUTIONS AND FUTURE WORK

In this chapter, I outline my contributions in the work I have presented and recommend future research directions.

6.1 Contributions

The primary goal of my thesis was to empirically evaluate genetic algorithms to generate cost-effective test suites for web applications. The contributions of my work are:

1. **explored test-suite generation by leveraging the genetic algorithm**: Chapter 3 highlights how I apply the genetic algorithm to generate test suites using logged user sessions from web applications, including an outline of the model that I use (Figure 3.2), and the implementation details of the framework that I propose. Chapter 5 delineates my experimental study procedure and analyzes and discusses the results that I obtain, comparing the genetic algorithm based approach to generating test suites with a hill-climbing algorithm based approach.

2. **modeled test suites for web applications as components for the genetic algorithm: genes, chromosomes, and genomes**: Section 3.2 highlights how I model test suites to be used within a genetic algorithm framework (Figure 3.1). Sections 5.2 and 5.3.1 outline how I convert user accesses to web applications into user sessions before parsing them into an input suitable for the genetic algorithm.

3. **implemented several genetic operators and parent selection algorithms to manipulate the genetic information of the test suites modeled as genomes**: Section 3.3.3 discusses my implementation of five genetic operators applied by the genetic algorithm to explore a variety of genetic compositions. Section 3.3.4 outlines my implementation of three parent selection algorithms used to select parent(s) for each genetic operator.

4. **produced an extensible evaluation framework that can be used to empirically evaluate testing approaches, and plan on making the evaluation framework available on GitHub for straightforward installation**

and easy use: Chapter 4 outlines the design and the implementation details of the evaluation framework that I produce to assess the cost-effectiveness of generated test suites.

5. **empirically evaluated the test suites generated by the genetic algorithm, using code coverage as evaluation metrics, to assess their cost-effectiveness**: Section 5.5.2 discusses the results of replaying the entire subject user session sets on the evaluation framework to obtain code coverage baseline measures, and then replaying the generated test suites on the evaluation framework. It assess the cost-effectiveness of the generated test suites by comparing their lengths and code coverage values with the original baseline measures obtained.

6. **recommended the best parameters to tune the genetic algorithm for cost-effective test suite generation based on the results of my experimental study**: Section 5.7 provides recommendations to testers using my framework to generate test suites for web applications. It highlights the recommended parent selection algorithm to use, suggests certain values for the genetic algorithm parameters, and provides guidelines to finding a good set of genetic operator thresholds, to assist testers in producing high-quality test suites from the genetic algorithm framework.

6.2 Future Work

The approach that I propose in this work can be further developed in the following ways:

1. Implementing more sophisticated multi-objective parent selection algorithms:

   The fitness function for the GA currently attempts to optimize multiple objectives, including resource coverage, genome length, length-two resource coverage, and length-three resource coverage. Certain parent selection algorithms have proved to work well with multi-objective optimizations, including the NSGA-II algorithm and Lexicase selection [5]. The framework that I propose could benefit from such parent selection algorithms.

2. Using additional subjects to verify the scalability of the approach:

   Larger subject applications accessed by more users may have different results with respect to the generated test suite's length and resource and code coverage.

3. Accounting for persistent state when genetic operators change the genetic information of genomes:

   Currently, a number of requests currently might result in error pages due to issues with persistent state, e.g., accessing a resource before logging on to the application or accessing data that does not exist yet. Test cases that take persistent

state into account may decrease the number of error pages accessed because of this limitation, thereby increasing the testing efficacy of test suites.

BIBLIOGRAPHY

[1] J. T. Alander, T. Mantere, and P. Turunen. Genetic algorithm based software testing. In *Artificial Neural Nets and Genetic Algorithms*, pages 325–328, Vienna, 1998. Springer Vienna.

[2] Mohammadreza Mollahoseini Ardakani and Mohammad Morovvati. A multi-agent system apporach for user-session-based testing of web applications. In *Proceedings of the 7th WSEAS International Conference on Distance Learning and Web Engineering*, DIWEB'07, pages 326–331, Stevens Point, Wisconsin, USA, 2007. World Scientific and Engineering Academy and Society (WSEAS).

[3] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, March 2005.

[4] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 49–59, Washington, DC, USA, 2003. IEEE Computer Society.

[5] T. Helmuth, L. Spector, and J. Matheson. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643, Oct 2015.

[6] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. Program synthesis using uniform mutation by addition and deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, pages 1127–1134, New York, NY, USA, 2018. ACM.

[7] Torsten Kempf, Kingshuk Karuri, and Lei Gao. *Software Instrumentation*. Wiley, September 2008.

[8] Rizwan Khan and Mohd Amjad. Performance testing (load) of web applications based on test case management. *Perspectives in Science*, 8, 04 2016.

[9] Bogdan Korel and Ali M. Al-Yami. Automated regression test generation. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '98, pages 143–152, New York, NY, USA, 1998. ACM.

[10] Giuseppe A. Di Lucca and Anna Rita Fasolino. Testing web-based applications: The state of the art and future trends. *Information and Software Technology*, 48(12):1172 – 1186, 2006. Quality Assurance and Testing of Web-Based Applications.

[11] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs (3rd Ed.)*. Springer-Verlag, Berlin, Heidelberg, 1996.

[12] Xuan Peng and Lu Lu. User-session-based automatic test case generation using GA. *International Journal of Physical Sciences*, 6(13):3232–3245, July 2011.

[13] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[14] Shikha Raina and Arun Prakash Agarwal. An automated tool for regression testing in web applications. *SIGSOFT Softw. Eng. Notes*, 38(4):1–4, July 2013.

[15] Je rey Horn, Nicholas Nafpliotis, and David E Goldberg. A niched pareto genetic algorithm for multiobjective optimization. In *Proceedings of the first IEEE conference on evolutionary computation, IEEE world congress on computational intelligence*, volume 1, pages 82–87. Citeseer, 1994.

[16] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, pages 132–141, Washington, DC, USA, 2004. IEEE Computer Society.

[17] S Sharmila and E Ramadevi. Analysis of performance testing on web applications. *International Journal of Advanced Research in Computer and Communication Engineering*, 2014.

[18] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated replay and fault detection for web applications. In *International Conference on Automated Software Engineering (ASE)*, pages 253–262, New York, NY, USA, November 2005. ACM Press.

[19] Sara Sprenkle, Lori Pollock, and Lucy Simko. A study of usage-based navigation models and generated abstract test cases for web applications. In *International Conference on Software Testing, Verification and Validation (ICST)*, ICST '11, pages 230–239, Washington, DC, USA, Mar 2011. IEEE Computer Society.

[20] Dr. Praveen Srivastava and Tai-Hoon Kim. Application of genetic algorithm in software testing. *International Journal of Software Engineering and Its Applications*, 3(4):87–95, November 2009.

[21] Shkodran Zogaj, Ulrich Bretschneider, and Jan Marco Leimeister. Managing crowdsourced software testing: a case study based insight on the challenges of a crowdsourcing intermediary. *Journal of Business Economics*, 84(3):375–405, 2014.