Learning How to Search: Generating Effective Test Cases Through Adaptive Fitness Function Selection

Hussein Khalid Almulla

Follow this and additional works at: https://scholarcommons.sc.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

This Open Access Dissertation is brought to you by Scholar Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact dillarda@mailbox.sc.edu.
Learning How to Search: Generating Effective Test Cases Through Adaptive Fitness Function Selection

by
Hussein Khalid Almulla

Bachelor of Science
University of Anbar 2008

Master of Science
University of Anbar 2011

Master of Science
University of South Carolina 2019

Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in Computer Science
College of Engineering and Computing
University of South Carolina
2021

Accepted by:
Gregory Gay, Major Professor
Csilla Farkas, Committee Member
John R. Rose, Committee Member
Pooyan Jamshidi, Committee Member
Benjamin Schooley, Committee Member
Tracey L. Weldon, Vice Provost and Dean of the Graduate School
© Copyright by Hussein Khalid Almulla, 2021
All Rights Reserved.
My dissertation is dedicated to my mother, who died in 2007. She always encouraged me to pursue a better education and continuous learning and put in a lot of effort to make sure I did so. I also dedicate this dissertation to my father, who always does everything he can for us to succeed. Without their continued support, I would not be where I am today. This is dedicated to my sisters for their love and support. This is dedicated to my wife for her patience, love, and encouragement that kept me moving. This is dedicated to my beloved kids.
ACKNOWLEDGMENTS

I would like to express my profound appreciation to my adviser, Dr. Gregory Gay, for support, encouragement, and guidance through the past four years. He has provided me with great opportunities to expand my knowledge in both research topics and related practical experiences. Additionally, I am especially appreciative of the words of wisdom that will I carry with me from here on.

The thanks extend to my colleagues from whom I learn about new cultures, thoughts, and ideas that equipped me with new and diverse perspective.

I would like to thank my committee members, Csilla Farkas, John R. Rose, Pooyan Jamshid, and Benjamin Schooley for their guidance during my research.

I express great appreciation for my country, Iraq, particularly the Higher Committee for Education Development in Iraq (HCED) for the financial and managerial support that made this opportunity possible.

Last, but not least, my appreciation is extended to all my family and friends for their endless love and support that light my way.
Abstract

Search-based test generation is guided by feedback from one or more fitness functions—scoring functions that judge solution optimality. Choosing informative fitness functions is crucial to meeting the goals of a tester. Unfortunately, many goals—such as forcing the class-under-test to throw exceptions, increasing test suite diversity, and attaining Strong Mutation Coverage—do not have effective fitness function formulations. We propose that meeting such goals requires treating fitness function identification as a secondary optimization step. An adaptive algorithm that can vary the selection of fitness functions could adjust its selection throughout the generation process to maximize goal attainment, based on the current population of test suites. To test this hypothesis, we have implemented two reinforcement learning algorithms in the EvoSuite framework, and used these algorithms to dynamically set the fitness functions used during generation for the three goals identified above.

We have evaluated our framework, EvoSuiteFIT, on a set of real Java faults. EvoSuiteFIT techniques attain significant improvements for two of the three goals, and show small improvements on the third when the number of generations of evolution is fixed. For all goals, EvoSuiteFIT detects faults missed by the other techniques. The ability to adjust fitness functions allows EvoSuiteFIT to make strategic choices that efficiently produce more effective test suites, and examining its choices offers insight into how to attain our testing goals. We find that AFFS is a powerful technique to apply when an effective fitness function does not exist for a testing goal.
# Table of Contents

Dedication ......................................................... iii

Acknowledgments ................................................ iv

Abstract ........................................................... v

List of Tables ..................................................... ix

List of Figures ................................................... xi

Chapter 1 Introduction .......................................... 1
  1.1 Motivation .................................................. 3
  1.2 Research Problem ........................................ 5
  1.3 Research Objective ...................................... 7
  1.4 Proposed Solution and Contribution .................. 7
  1.5 Dissertation Outline .................................. 12

Chapter 2 Background ........................................ 14
  2.1 Unit Testing ............................................... 14
  2.2 Search-Based Test Generation ......................... 16
  2.3 Reinforcement Learning ................................ 19

Chapter 3 Technical Approach and Implementation ........ 22
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Upper Confidence Bound (UCB) Algorithm</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>Differential Semi-Gradient Sarsa (DSG-Sarsa)</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>EvoSuite Overview</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>Implementation of AFFS within EvoSuite</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Related Work</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Hyperheuristics in Search-Based Software Testing</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Crash and Exception Discovery</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Test Suite Diversity</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Strong Mutation Coverage</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Methodology</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Case Examples</td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Test Suite Generation</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Data Collection</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Results and Discussion</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Goal: Exception Discovery</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>Goal: Test Suite Diversity</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>Goal: Strong Mutation Coverage</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>Discussion</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>Threats to Validity</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Conclusions and Future Work</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>7.1</td>
<td>Summary</td>
<td>110</td>
<td></td>
</tr>
</tbody>
</table>
List of Tables

Table 5.1 Data collected for each experiment ................................. 65

Table 6.1 Median count of exceptions thrown and exceptions discovered for each technique, along with the median ratio of thrown to discovered. Counts are normalized between 0-1 for each fault to allow comparison across case examples. Higher scores are better. ................................. 68

Table 6.2 P-Values for Mann-Whitney rank-sum test for exceptions thrown and discovered. Results are identical for both. ................................. 70

Table 6.3 Results of Vargha-Delaney A Measure for exceptions thrown and discovered. Large positive effect sizes are bolded. ................................. 70

Table 6.4 Percentage of faults detected by each approach. ................................. 71

Table 6.5 Median time per generation (in seconds) for the goal of exception discovery. EX+(1,2,3) = exception count + 1-3 fitness functions. The comparison data [37] lacked EX+1 and EX+3 data for Mockito. The fastest approach is bolded. ................................. 72

Table 6.6 Median diversity fitness score of the produced test suite. Score is between 0-1, with lower scores being better. Best approach is bolded. ................................. 78

Table 6.7 P-Values for Mann-Whitney rank-sum test for diversity score. ................................. 81

Table 6.8 Results of Vargha-Delaney A Measure for diversity score. Large positive effect sizes are bolded. ................................. 81

Table 6.9 Percentage of faults detected by each approach. ................................. 81

Table 6.10 Median time per generation (in seconds) for the goal of test suite diversity. The fastest approach is bolded. ................................. 82

Table 6.11 Percentage of Strong Mutation Coverage attained when all approaches execute for 10 minutes. Higher values are better. Best approach is bolded. ................................. 88
Table 6.12 P-Values for Mann-Whitney rank-sum test for Strong Mutation (all approaches run for 10 minutes). No p-values met the $\alpha = 0.05$ threshold. ................................. 89

Table 6.13 Results of Vargha-Delaney A Measure for Strong Mutation (all approaches run for 10 minutes). There is no large effect sizes were observed. ................................. 89

Table 6.14 Median time per generation (in seconds) for each technique for the Strong Mutation Coverage goal. Time is attained by dividing the number of generations completed by the total generation time (10 minutes). The fastest technique is bolded. ................................. 90

Table 6.15 Percentage of strong mutation coverage attained by test suites when the number of generations of evolution is fixed to that completed by DSG-Sarsa in 10 minutes. Higher values are better. Best approach is bolded. ................................. 92

Table 6.16 P-Values for Mann-Whitney rank-sum test for Strong Mutation (number of generations fixed). ................................. 93

Table 6.17 Results of Vargha-Delaney A Measure for Strong Mutation (number of generations fixed). There is no large positive effect sizes were observed. ................................. 93

Table 6.18 Percentage of faults detected by each approach for the Strong Mutation goal. $F\#G =$ fixed number of generations ................................. 94
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example of a unit test case written using the JUnit notation for Java.</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>A generic hyperheuristic search process.</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>An overview of the reinforcement learning process.</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>A simplified overview of EvoSuite’s test generation process.</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>How reinforcement learning fits into the test generation process.</td>
<td>34</td>
</tr>
<tr>
<td>6.1</td>
<td>Unique exceptions discovered and thrown by each technique. Counts are normalized between 0-1 for each fault to allow comparison across case examples.</td>
<td>68</td>
</tr>
<tr>
<td>6.2</td>
<td>Top ten function combinations chosen by DSG-Sarsa for each system for the exception discovery goal. E = Exception Count, B = Branch Coverage, CB = Direct Branch Coverage, L = Line Coverage, O = Output Coverage, M = Method Coverage, MNE = Method (No Exception), WM = Weak Mutation Coverage.</td>
<td>75</td>
</tr>
<tr>
<td>6.3</td>
<td>Top ten function combinations chosen by UCB for each system for the exception discovery goal. E = Exception Count, B = Branch Coverage, CB = Direct Branch Coverage, L = Line Coverage, O = Output Coverage, M = Method Coverage, MNE = Method (No Exception), WM = Weak Mutation Coverage.</td>
<td>76</td>
</tr>
<tr>
<td>6.4</td>
<td>Diversity fitness scores of the produced test suites. Score is between 0-1, with lower scores being better.</td>
<td>79</td>
</tr>
<tr>
<td>6.5</td>
<td>Top ten fitness function combinations chosen by DSG-Sarsa for the diversity goal. D = Diversity Score, B = Branch Coverage, CB = Direct Branch Coverage, O = Output Coverage, M = Method Coverage, MNE = Method (No Exception), WM = Weak Mutation Coverage.</td>
<td>85</td>
</tr>
</tbody>
</table>
Figure 6.6  Top ten fitness function combinations chosen by \textbf{UCB} for the diversity goal. D = Diversity Score, B = Branch Coverage, CB = Direct Branch Coverage, O = Output Coverage, M = Method Coverage, MNE = Method (No Exception), WM = Weak Mutation Coverage, E = Exception 

Figure 6.7  Strong Mutation Coverage attained by final test suites when (left) all approaches run for 10 minutes, and (right), when all approaches are fixed to the number of generations of evolution completed by DSG-Sarsa in 10 minutes.

Figure 6.8  Top ten fitness function combinations chosen by \textbf{DSG-Sarsa} approach for the Strong Mutation goal. SM = Strong Mutation Coverage, B = Branch Coverage, O = Output Coverage, MNE = Method (No Exception), WM = Weak Mutation Coverage, E = Exception Count

Figure 6.9  Top ten fitness function combinations chosen by \textbf{UCB} approach for the Strong Mutation goal. SM = Strong Mutation Coverage, B = Branch Coverage, O = Output Coverage, MNE = Method No Exception, WM = Weak Mutation Coverage, E = Exception Count
Chapter 1

Introduction

Complex software powers our society, underlying many of the rapid advancements made in recent years. The testing of software is a crucial step in the software development life cycle, as testing is our primary means of ensuring that complex software is robust and operates correctly [71]. However, testing is an expensive task that can consume much of the total development budget [71].

Testing requires the creation of test suites that are used to evaluate the system-under-test. Test creation is an effort-intensive task that requires the selection of sequences of program input and the creation of oracles that judge the correctness of the resulting execution [11]. If test generation could be even partially automated, the benefit to developers in terms of effort and cost would be immense. Naturally, a large body of research has been amassed around topics such as automated test input generation [8] and expected output (oracle) generation [11]. One area of automation that has shown great promise is search-based test generation [8, 66].

Test input generation—the selection of program features and parameter values—can naturally be seen as a search problem [49]. Testers approach input selection with a goal in mind—perhaps they would like to cause the program to crash, maximize code coverage, detect a set of known faults, or any number of other potential goals. Of the near-infinite number of possible inputs that could be provided to a program, the tester seeks input that meets a chosen goal. The search for test input that can maximize the goal can then be automated. Given a measurable goal, a metaheuristic optimization algorithm can systematically sample the space of possible test input,
guided by feedback from one or more **fitness functions**—numeric scoring functions that judge the optimality of the chosen input [76]. In other words: *algorithm + fitness functions* $\mapsto$ *goal*.

The effective use of search-based generation relies on the selection of the correct sampling mechanism—the right algorithm—and, perhaps more importantly, the proper feedback mechanisms— the correct fitness functions. Fitness functions shape the test suites generated by the search process to have certain properties promoted by those functions. The fitness functions used as part of test case generation, in normal use, are expected to embody the overall goals of the tester. By offering feedback on the quality of the generated solutions, they then ensure that test suites converge on these goals. The best fitness functions offer the information needed to rapidly increase attainment of the goal, differentiating *good* solutions from *bad* solutions, and offering the algorithm the feedback needed to locate better solutions.

Consider, for example, the concept of Branch Coverage. Branch Coverage is a measurement of *how much* of the code has been executed. For each program statement that can cause the execution path to diverge—such as *if* and *case* statements—test input should ensure that all potential outcomes are covered at least once [71].

If our goal is to achieve 100% Branch Coverage, there are multiple fitness functions that could be used to guide the optimization algorithm towards attaining coverage. A simple fitness function could simply measure the level of coverage attained by each test suite. A test suite that attains 75% Branch Coverage is inherently better than one that attains 50% Branch Coverage. This would alert the algorithm to which test suites to favor, leading over time to higher and higher attainment of Branch Coverage.

However, there is a better fitness function we could use to guide the search towards Branch Coverage attainment. Instead, we could take each subgoal we wish to cover—in this case, each branching outcome—and judge *how close* the chosen test input was to achieving that outcome. If we execute an expression “*if* (x == 5)” with the value
of x set to 3, and we seek a true outcome, then x needs to be incremented by 2. This suggests the magnitude of change needed to reach the desired outcome [66].

This concept, the branch distance [9], offers the algorithm additional feedback beyond a simple score—offering both a measurement of how much of the goal has been met and clues on how to attain full coverage. Similar notions underlie many successful applications of search-based test generation. Effective use of search-based test generation relies on the selection of fitness functions that give the optimization algorithm the feedback needed to attain the goals of the tester.

1.1 Motivation

As the earlier example illustrates, if our goal is to attain Branch Coverage, we have effective fitness functions that lead to rapid improvement in coverage. Unfortunately, many goals do not have a known, effective fitness function formulation. In fact, many goals do not inherently lend themselves to such a formulation.

To illustrate, consider the following three goals:

- **Exception Discovery**: “Causing the program to crash” is a common goal in testing. The number of crashes discovered is often measured by counting the number of exceptions—program-interrupting error messages—thrown during test execution [72]. Exceptions indicate the faults and abnormal operating conditions in programs. Thus, tests that trigger exceptions are valuable.

- **Test Suite Diversity**: When testing, it is generally impossible to try every input. It follows, then, that different test cases are more effective than similar ones [20, 77]. This intuition has led to effective automated test generation, prioritization, and reduction [20].

- **Strong Mutation Coverage**: Mutation testing is a practice where synthetic faults (called mutants) are injected into the code. If test suites detect these
“fake” faults, they are thought to be more robust to real faults. Mutation Coverage is often measured in two forms. In Weak Mutation Coverage, a mutant is considered detected if execution reaches the infected expression and the outcome of that expression differs from the original program—i.e., the state is infected. Strong Mutation Coverage requires an additional guarantee that the infected state propagate to the program output, offering clear evidence that the seeded fault was detected [61].

These three goals will be the focus of our research. All three are valid, measurable, goals for test suite generation. In principle, all three should be reasonable targets for search-based test generation. However, all three have properties that make them difficult to optimize directly:

- As we cannot know how many or what exceptions are possible to throw, “throw more exceptions” is not a goal that translates into an informative fitness representation. Prior work has proposed the use of a simple count of thrown exceptions as a fitness function [73]. Unfortunately, this count yields poor results in terms of both goal attainment and fault detection, as it offers the algorithm no guidance for improving its guesses [35, 36, 76].

- While numerous diversity metrics exist—for example, the Levenshtein distance [77] measures the number of operations needed to convert one string to another—these metrics tend to serve as poor fitness functions, as little feedback is offered to suggest how to gain more diversity.

- Weak Mutation Coverage can be optimized using a variant of the branch distance, where a cost function measures how close execution came to reaching the mutated line and corrupting the program state [32]. It is more difficult to offer fine-grained feedback on how to ensure that this corruption propagates to the program output—the condition required to attain Strong Mutation Coverage.
Current fitness functions offer probabilistic estimations of propagation [32, 70]. However, again, such estimations are generally too coarse-grained to accurately guide the search.

All three of these goals—and their current interpretation as fitness functions for search-based test generation—follow a common theme. They do not inherently lend themselves to formulation as feedback-offering fitness functions. This does not mean there is no way to effectively achieve such goals. Rather, we simply do not yet know what fitness functions will be effective.

There are many fitness functions available for use in search-based test generation. If we do not know of an effective fitness function that we can optimize to directly achieve a goal, it may be possible to identify fitness functions that indirectly achieve our goal. Careful selection of one or more of those functions could yield high goal attainment. For example, if optimizing the exception count fails to directly produce test suites that discover a large number of exceptions, optimizing the branch distance—or even simultaneously targeting branch distance and exception count—might instead indirectly achieve that goal. We simply need to identify that selection.

1.2 Research Problem

To summarize the high-level problem presented above, fitness functions are responsible for ensuring that test goals are met during search-based test generation by offering the feedback that the underlying metaheuristic optimization algorithm can use to reformulate its produced test suites. However, some testing goals—such as exception discovery, test suite diversity, and Strong Mutation Coverage—lack effective fitness functions. It may be possible to still achieve such goals, but this requires the selection of fitness functions adept at indirectly achieving such goals.

There are many possible combinations of fitness functions that could be selected, and the correct choices will be goal-specific. In fact, the “correct” choices may even
be specific to the system-under-test, and could even vary *during* test case generation, as search-based processes evolve test suites over time based on the state of the population of test suites during previous generations. Therefore, we seek a *systematic method of automatically identifying the appropriate fitness functions* that is appropriate for a variety of high-level testing goals.

We hypothesize that an *adaptive* algorithm—one that can vary the selection of fitness functions—could strategically adjust the chosen fitness functions throughout the generation process to maximize attainment of a desired goal.

A class of search-based test generation approaches are known as *hyperheuristic*, or self-adaptive, approaches [55, 44]. These approaches incorporate a learning phase in order to automatically tune the search strategy. Hyperheuristic search has been used, for example, to change parameters of the metaheuristic algorithm during evolution to change how solutions are mutated or combined between generations [55].

To evaluate this hypothesis, we propose a hyperheuristic search that optimizes the test generation process to adjust the set of chosen fitness functions [54]. Through the use of reinforcement learning [83], this approach is able to select the most appropriate set of fitness functions for the system-under-test and testing goal, and adjust that set as needed during generation. In this process, a measurement—representing the real goal of the search—is targeted as a high-level *reward function*. A reinforcement learning process is added to the search and selects the fitness functions that will be used to refine test suites. After evolving the population of test suites with the currently chosen fitness functions for a set number of generations, the change in the reward score will be evaluated and the reinforcement learning process will decide whether to continue using the set of fitness functions known to best improve the reward (*exploitation*) or to try a different set of fitness functions in order to refine expectations of the change in reward (*exploration*).
We refer to this hyperheuristic as **adaptive fitness function selection** (AFFS) [6, 5], and hypothesize that AFFS is the key to systematically and automatically generating test suites that achieve difficult-to-optimize goals. Over time, we hypothesize that this process will discover the fitness functions that are best at indirectly achieving goals such as exception discovery, test suite diversity, or Strong Mutation Coverage.

### 1.3 Research Objective

Our **long-range goal** is to improve the effectiveness of automated test generation. Test creation is an expensive, effort-intensive task. Automation of aspects of the test creation process, such as input creation, could lead to significant cost and effort reduction and could free developer attention for important tasks. A major hindrance in meeting this goal is the lack of understanding of how to generate tests for many valid, measurable testing goals.

This problem provides motivation for the **objective of this dissertation**, which is to explore whether such testing goals can be optimized through the intelligent selection of fitness functions by a hyperheuristic built into the existing search-based test generation process. The **central hypothesis** of this dissertation is that using reinforcement learning to strategically vary the chosen fitness functions will lead to higher attainment of goals than statically optimizing existing fitness function representations of such goals or optimizing functions naively chosen by a human developer.

### 1.4 Proposed Solution and Contribution

To that end, we have proposed methods of performing adaptive fitness function selection as part of the process of generating test suites. We have implemented two reinforcement learning algorithms—Upper Confidence Bound (UCB) and Differential Semi-Gradient Sarsa (DSG-Sarsa) [83]—in the EvoSuite unit test generation framework for Java [74]. We refer to the modified framework as EvoSuiteFIT.
We have evaluated EvoSuiteFIT for each of our three goals—exception discovery, test suite diversity, and Strong Mutation Coverage—on a set of Java case examples in terms of the ability of generated test suites to achieve the targeted goal and in terms of the ability of the generated suites to detect faults. In each case, we compare the two reinforcement learning approaches to two baselines. The first baseline is the current practice—a fitness function based on the goal that may not offer sufficient feedback. For exception discovery, this is a count of exceptions thrown [73]. For test suite diversity, this is the Levenshtein distance [77]. Finally, for Strong Mutation Coverage, we target the existing fitness function in EvoSuite, based on an estimated “impact distance” [32]. We additionally compare to a set of multiple fitness functions—the full set of functions that AFFS can choose among for that goal—that serves as a “best guess” a human might make at a combination of fitness functions that would produce effective test suites. We have found that:

- Both EvoSuiteFIT techniques discover and retain more exception-triggering input than the baseline techniques, with DSG-Sarsa yielding more consistent results. EvoSuiteFIT attains a 100% improvement in median exception discovery over the simple exception count and 33% over the default combination.

- Both EvoSuiteFIT techniques produce more diverse test suites, with UCB outperforming DSG-Sarsa. UCB attains 76.27% better median performance than diversity alone, 41.04% better than the default combination, and 22.72% better than DSG-Sarsa.

- For the goal of Strong Mutation Coverage, EvoSuiteFIT is slightly outperformed in median performance by the two baselines. Optimizing a baseline yields a median improvement of 2.56% over UCB and 5.26% over DSG-Sarsa. However, no technique demonstrates statistically significant improvements. When the search budget is a fixed number of generations rather than time, both Evo-
SuiteFIT techniques slightly outperform the baselines. UCB outperforms the default baseline by 8.33% and the Strong Mutation baseline by 5.41%. The effect size is small, but given additional time for test generation, we see some improvement from using AFFS over static approaches.

- For all goals, EvoSuiteFIT techniques detect faults missed by the other techniques. UCB detects 11.92 and 249.57% more faults than the baseline techniques, and 4.3% more than DSG-Sarsa, for the exception discovery goal. For the goal of test suite diversity, DSG-Sarsa detects 10.84 and 48.39% more faults than the baseline techniques, and 8.24% more than UCB. Finally, UCB is able to detect more faults than all other approaches for the Strong Mutation goal, outperforming the default baseline by 11.51%, Strong Mutation alone 12.32%, and DSG-Sarsa by 15.67%. DSG-Sarsa is outperformed by the baselines for this goal. However, when the number of generations is fixed, both EvoSuiteFIT approaches significantly outperform the baselines.

- We find that AFFS is an appropriate technique to apply when an effective fitness function does not already exist for the targeted goal. However, AFFS requires a reward function that is fast to calculate, or requires additional time for test generation. Further, the effect of AFFS is limited by the span of fitness functions available to choose from. If none of the chosen functions correlate to the goal of interest, then improvements in goal attainment will be limited.

- Improvements in fault detection may arise because of higher attainment of goals thought to have a positive relationship with fault detection likelihood, optimizing multiple fitness functions—but avoiding needlessly complex and conflicting functions—and changing fitness functions as the suite evolves rather than applying all functions at once.
• While reinforcement learning adds overhead to test generation, EvoSuiteFIT is often faster than the default static configuration because the ability to avoid calculation of unhelpful fitness functions mitigates this overhead. Further, feedback from effective fitness functions can help control computational costs.

• The ability to adjust the fitness functions at regular intervals allows EvoSuiteFIT to make strategic choices that refine the test suite and allows us to attain a deeper understanding of the properties that link to goal attainment and how fitness functions can work together to imbue those properties. Fitness function combinations that are ineffective in a static context may be effective when used by AFFS to diversify a pre-evolved population of suites.

Under the correct conditions, the use of AFFS allows EvoSuiteFIT to identify combinations of fitness functions effective at achieving our testing goals, and strategically vary that set of functions throughout the ongoing generation process. We hypothesize that other goals without known effective fitness function representations could also be maximized in a similar manner. We make EvoSuiteFIT available to others for use in test generation research or practice.

The intellectual merit of the project lies in its focus on automatically identifying the fitness functions adept at achieving a testing goal. Past hyperheuristic search-based test generation approaches have focused on tuning the crossover and mutation operators used by the metaheuristic optimization algorithm [55, 54, 90, 47, 44, 22, 28]. Fitness function selection has been performed by hyper-heuristic search in other domains, such as production scheduling [19, 68]. However, our approach is the first automated technique for optimizing the set of fitness functions used during test generation. This work is significant because it enables effective optimization without the need for manual fitness function selection or development. As a result, it could po-
tentially bring about major improvements to the cost and effort of test case creation, and could improve the quality of testing results.

This dissertation has made the following contributions to software testing:

1. **We have designed an automated self-adaptive method of selecting fitness functions to optimize difficult testing goals.** We propose adaptive fitness function selection (AFFS), a hyperheuristic that can be used to identify fitness functions appropriate for optimizing a high-level reward function—mapped to a desired testing goal.

2. **We have explored the types of goals that can be targeted for AFFS.** Not all testing goals can be indirectly met through AFFS. We have explored the use of AFFS for three goals—exception discovery, test suite diversity, and Strong Mutation Coverage. For each, we explore whether AFFS is a reasonable method of achieving that goal, whether it could be better optimized without AFFS, and whether there are limitations that affect the ability of AFFS to meet this goal. This exploration offers advice for applying AFFS to other goals.

3. **We have implemented AFFS within an automated testing framework.** This framework, EvoSuiteFIT, is able to make effective use of AFFS to optimize difficult testing goals for a wide range of Java classes.

4. **We have empirically evaluated the use of AFFS to detect real-world software faults.** In addition to determining the effectiveness of AFFS in terms of both goal achievement and fault detection, we examine the impact of AFFS on the test generation process as well as the behavior of AFFS when choosing fitness functions and reevaluating those choices.

The outcomes of this project are (1) an implementation of AFFS in the EvoSuite unit test generation framework for Java, and (2) authoritative studies on the
capabilities, effectiveness, and limitations of AFFS as applied to a large body of real-world Java case examples. The following publications have resulted from the research reported in this thesis:


1.5 Dissertation Outline

The remainder of this dissertation is organized as follows:

- Chapter 2 presents background material on software testing, search-based test generation, and reinforcement learning.

- Chapter 3 outlines our approach to adaptive fitness function selection, detailing the reinforcement learning algorithms used and their integration into the test generation process, as well as the fitness functions used and implemented and how goal attainment is measured during test generation.

- Chapter 4 surveys related work on hyperheuristic search-based test generation, as well as test generation for exception discovery, test suite diversity, and Strong Mutation Coverage.
• Chapter 5 details the research questions and experimental studies that we use to evaluate the effectiveness and applicability of AFFS.

• Chapter 6 reports the results of our evaluation and discusses their implications with regard to the effectiveness of AFFS. We also discuss the threats to the validity of this research.

• Finally, Chapter 7 concludes this dissertation and outlines future work.
This chapter presents background concepts necessary to understand core concepts of this dissertation. In particular, we present an overview of unit testing, search-based test generation, and reinforcement learning.

2.1 Unit Testing

Developing software is not just about writing code that provides functionality, but also requires verifying, at different levels, that code is functioning as intended. Verification is often performed using testing—the application of input to the system, and analysis of the resulting output to ensure correctness [71, 11]. Testing can be performed at various levels of granularity. In this research, we are focused on unit testing. Unit testing is the lowest level of testing in which individual units of software are tested. A unit in a piece of software represents the smallest segment of code that can be tested in isolation from the rest of the system, often a class [78]. Unit tests are written as executable code. When code changes, developers can re-execute the same unit tests to make sure the code works as expected after changing. Unit testing frameworks exist for many programming languages, such as JUnit for Java, and are integrated into most development environments.

An example of a test case, written in the JUnit notation for Java, is shown in Figure 2.1. Unit test cases can be defined as a set of instructions to verify that the examined code delivers the desired functionality. A unit test consists of a test sequence (or procedure)—a series of method calls to the CUT—with test input provided
to each method. The concept of “test inputs” is broad, and can include the entire spectrum of data that can be provided to method input, assignment of values to class variables, or adjustment of environmental conditions that can cause the system to act. When the test cases are written manually, test inputs are usually drawn from requirement specifications, documentation, or personal intuition. The choice of test input is critical in determining how to explore different parts of code under test. When selecting test input, particularly during test generation, input should be drawn from a variety of partitions within the input space.

Then, the test case will validate the output of the called methods and the class variables against a set of encoded expectations to determine whether the test passes or fails. The test oracle is the component that compares the actual behavior of the software with a set of encoded expectations [11]. In a unit test, the oracle is typically formulated as a series of assertions on the values of method output and class attributes. In the example in Figure 2.1, the test input consists of passing a user-provided string to the constructor of the ClassExample class, then calling its getText() method. From documentation or intuition, we know that this method should transform the provided string to upper-case. To ensure this is the case, our test oracle consists of an assertion that checks whether the string provided by the output of the getText() call is equal to an upper-case version of the provided string.

When performing unit testing, it is recommended to construct different test cases for different scenarios. Each of these scenarios may cover different portions of the
code, different functionality outcomes, or different system requirements. We refer to a group of test cases as a test suite. Grouping classes into a test suite allows easier management and analysis of the collected test cases.

2.2 Search-Based Test Generation

Manual test creation is extremely expensive in terms of time and effort. Automation has a critical role in controlling the cost of testing [69, 4]. Automated test generation is a popular topic [8], and achievements in automated generation have been named among the most significant advances in recent testing research [69]. One particular task that has seen great attention is the selection of test input. Exhaustively applying all possible inputs is infeasible due to enormous number of possibilities. Therefore, which input are tried becomes important. A promising method is search-based test input generation.

Test case creation can naturally be seen as a search problem [49]. Out of all of the test cases that could be generated for a class, we want to select—systematically and at a reasonable cost—those that meet our goals [66, 3]. Given a testing goal and a scoring function denoting closeness to the attainment of that goal—called a fitness function—optimization algorithms can sample from a large and complex set of options as guided by a chosen strategy (the metaheuristic) [14].

Metaheuristics are often inspired by natural phenomena, such as swarm behavior [24] or evolution [51]. While the particular details vary between algorithms, the general process employed by a metaheuristic is as follows: (1) One or more solutions are generated, (2), The solutions are scored according to the fitness function, and (3), this score is used to reformulate the solutions for the next round of evolution. This process continues over multiple generations, ultimately returning the best-seen solutions. The metaheuristic (genetic algorithm, simulated annealing, etc.) overcomes the shortcomings of a purely random selection when selecting test input by
using a deliberate strategy to traverse the input space, gravitating towards “good” input and discarding “bad” input—as determined by the fitness function—through the incorporation of fitness feedback and mechanisms for manipulating a population of solutions. By determining how solutions are evolved and selected over time, the choice of metaheuristic impacts the quality and efficiency of the search process [26].

In search-based test generation, the fitness function captures the testing objective and guides the search. Through this guidance, the fitness function has a major impact on the quality of the solutions generated. Functions must be efficient to execute, as they will be calculated thousands of times over a search. Yet, they also must provide enough detail to differentiate candidate solutions and guide the selection of optimal candidates. Structural coverage of the source code is a common optimization target for search-based test generation as such coverage criteria can be straightforwardly transformed into efficient, informative fitness functions [9]. Search-based generation often can achieve higher coverage than developer-created tests [34].

Due to the non-linear nature of software, resulting from branching control structures, a real-world program’s search space is large and complex [3]. Metaheuristic search—by strategically sampling from that space—can scale to larger problems than many other generation algorithms [64]. Such approaches have been applied to a wide variety of testing goals and scenarios [3].

2.2.1 Hyperheuristic Search

A special class of search-based approaches are known as hyperheuristic, or self-adaptive, approaches [55, 6, 5]. These approaches incorporate a learning phase in order to automatically tune the search strategy towards particular problem instances [10]. Hyperheuristic search has been used, for example, to change parameters of the metaheuristic during evolution [55].
Hyperheuristic search can, essentially, be thought of as “using a heuristic to choose a heuristic.” The general process employed by a hyperheuristic approach is shown in Figure 2.2. A hyperheuristic approach introduces an automated high-level search that can explore the lower-level space of options available to tune the metaheuristic algorithm, looking for the best options to solve the targeted problem. These options are low-level “heuristics”, and may include aspects of the metaheuristic such as population tuning mechanics (i.e., the crossover and mutation rates of a genetic algorithm) or the choice of fitness functions. The metaheuristic operates directly on the problem space, attempting to optimize fitness functions using the options selected by the high-level hyperheuristic layer. The hyperheuristic layer optimizes the metaheuristic itself, choosing the options that enable the low-level search to find better solutions [25, 60].

Hyperheuristic approaches can be divided into two types—selection and generation. Selection-based approaches aims to select the right low-level heuristics from a preexisting set based on the current state of the optimization process. Generation-based approaches create new heuristics using the components of existing heuristics as building blocks. Generation-based heuristics generally operate using genetic programming, a metaheuristic process used to generate code [17]. Selection-based hyperheuristics are more common, especially in software testing research, as they are
often easier to implement and are suited to a wider range of problems [10]. However, generation-based approaches may yield better solutions when applied successfully.

In this research, we use a selection-based hyperheuristic approach with online learning—the ideal settings are learned during test generation. This differs from offline learning, where the ideal option is learned separately, then applied later. We use online learning because of the cost required for offline learning—i.e., we would have to apply all options to a large variety of classes to learn a potentially inappropriate strategy that overfits to the targeted classes. The hyperheuristic that we propose uses reinforcement learning in the high-level space to tune the selection of fitness functions in the low-level metaheuristic space. We will discuss this approach further in Chapter 3.

2.3 Reinforcement Learning

Reinforcement learning is a form of machine learning that focuses on identifying an action that maximizes return (measured using a problem-specific numerical reward score). This return is gained after an agent interacts with the specified environment to reach the desired goal. This process is illustrated in Figure 2.3. To understand reinforcement learning, consider the $n$-armed bandit problem [59]. This problem describes a situation where you are repeatedly faced with a choice of $n$ different options. After each selection, you receive a reward chosen from a probability distribution dependent on the action selected. Reinforcement learning algorithms are designed to learn the optimal choice of action to maximize the reward earned [83].

Each action has an expected reward when it is selected. Over time, the reinforcement learning agent will try different actions and refine its estimations of their value. During each round, the agent will choose an action based on the expected reward of applying it in the current problem state. After applying the action, the agent will
Reinforcement learning manages the trade-off between two concepts—exploration and exploitation—to maximize the reward. An agent must explore—choosing different actions—until it reaches the point where it can exploit that knowledge—favoring the actions known to provide a higher reward. At any time, there will be a portfolio with the greatest estimated value. If the algorithm selects that portfolio, it exploits its current knowledge to gain immediate reward. If instead, it chooses a portfolio with an unknown or potentially lower reward, it is exploring the option space to improve its estimate of a portfolio’s value. Reinforcement learning is designed to effectively balance exploration and exploitation for different problem spaces [83, 55, 54].

In this work, we consider two different types of reinforcement learning—tabular solution methods and approximation solution methods. Tabular methods are generally used in cases where states and action space are small enough so they can be represented in table or array. For that, a method can find the exact solution for the given problem. However, finding an exact solution is not feasible when the state space is large or continuous. In this case, approximation methods attempt to find
an approximate solution rather than a specific one. An approximate solution is an estimated solution for a problem whose true optimal solution is never known. These type of methods depend on generalization, the concept that an approach can learn from previously encountered states and tries to apply what it has learned to similar situations. This is where the role of the function approximation comes into play. Function approximation is a function that is used to calculate the approximation. It provides an estimation of a state value based on similar states (feature) that have been seen. The estimation is used because it is impossible to find the true state's value, therefore; an the function approximates the value based on similarity [83].
Search-based test generation requires selecting one or more fitness functions to guide the search process. Careful selection is crucial, as the fitness functions act as strategies that shape the resulting test suite. In theory, fitness functions should be selected to maximize the attainment of the tester’s overall goals. However, this is not always straightforward. In practice, many goals do not translate cleanly to effective fitness function representations—ones that offer detailed feedback to the search process to enable rapid optimization.

Consider the three goals that we are focused on in this research: exception discovery, test suite diversity, and Strong Mutation Coverage. All three have existing fitness function representations. Exception discovery can be represented by a simple count of exceptions thrown. Test suite diversity can be calculated through a number of mechanisms, including the Levenshtein distance, which determines the number of operations needed to transform one string into another. Finally, Strong Mutation Coverage can be calculated by combining the informative distance function used for weak mutation coverage with a probabilistic estimation of propagation to the output.

However, all three of these fitness representations have weaknesses. In particular, the fitness function for exception discovery is worth examining. As we cannot know ahead of time how many or what exceptions are possible to throw, it is very difficult to estimate how to throw these unknown exceptions. Counting the ones that are thrown meets the technical requirement of a fitness function, in that it can distinguish a test suite that throws exceptions from one that does not. However, it offers no actionable
feedback to the search. Finding new exceptions requires blind guessing. Past research showed that general functions such as Branch Coverage were able to trigger more exceptions simply by guiding the search to visit more of the codebase [76].

The other two goals are also difficult to optimize. The Levenshtein distance can tell how different two test suites are. A detailed distance function can be very helpful when one wants to minimize the distance, and when the actions a test generation takes have a direct and learn-able impact on this score. This is the case with the branch distance used when optimizing Branch Coverage. It is less helpful when one wants to maximize the distance—to make the test suites more different—and when it is not clear how to cause the most effective change in this score by manipulating method calls to the class-under-test.

The final goal, Strong Mutation, should be the easiest to optimize of this set. It builds on the fitness function for Weak Mutation Coverage, which is—in turn—based on the branch distance. This means that it is effective for guiding test suites to trigger a seeded fault. However, Strong Mutation Coverage also requires that a triggered fault propagate to an observable failure in the output. Capturing this propagation as a fitness function is a more complex problem, and current approaches only provide course-grained estimations that insufficiently guide the search.

All three of these goals share common properties. All three contain elements that are either unknowable upfront, or are difficult to estimate. Optimization of these functions does not map to the actions available to the test case generator in a way that can be easily predicted, often requiring more specific actions not suggested by the fitness function feedback. Such properties are common when examining the goals a tester might have when creating test suites. In this research, our true aim is not to find a better way to meet these three specific goals. Rather, our aim to develop a systematic approach capable of better meeting any goal that does not already have an effective fitness function—especially goals that cannot be translated into a detailed
function like exception discovery. Even if existing fitness representations of such goals are insufficient to produce effective test suites, such goals can still be met. The existing fitness functions simply do not provide sufficient feedback. The problem to be solved is how to provide that feedback. An often overlooked benefit of search-based test generation over other test generation approaches is that search-based generation can simultaneously target multiple objectives [40]. Each fitness function further shapes the test suite, imbuing it with the properties encouraged by that function. This offers an opportunity to provide that missing feedback. We can augment—or even replace—the existing fitness representations with additional fitness functions that direct the search towards optimization of our core, high-level goal.

We hypothesize that careful selection—at different points in the generation process—of the set of optimized fitness functions could result in test suites that better meet our goals—exception discovery, suite diversity, or whatever our goal might be—than existing baselines. If this is true, identifying the correct sets of fitness functions becomes a secondary search problem, tackled as an additional hyperheuristic optimization stage within the normal test generation process [54].

We propose the use of reinforcement learning techniques to adapt the set of fitness functions over the generation process at regular intervals in service of matching the chosen CUT and a measurable testing goal. Adjusting the set of fitness functions could be considered as a reinforcement problem [59]. Given a measurable goal, each action—each choice of one or more fitness functions—has an expected reward when it is selected. If we use this function combination, we will increase attainment of our goal. Because test generation is a stateful process—the population of test suites at round \( N \) depends on the population from round \( N-1 \)—reinforcement learning affords not just an opportunity to identify effective fitness functions, but to strategically adjust the functions based on the changing population of test suites. We refer to this process as adaptive fitness function selection (AFFS).
In this work, we focus on *online* learning as a means to adjust fitness functions during the active test generation process. The policy for fitness function adjustment employed by the reinforcement learning algorithm is learned during test generation, and is not reused later. This is opposed to an *offline* approach, where we would learn a policy and then apply it during generation.

In this work, we have implemented AFFS by extending the EvoSuite test generation framework [74] with two online reinforcement learning algorithms—Upper Confidence Bound (UCB) and Differential Semi-Gradient Sarsa (DSG-Sarsa) [83]. EvoSuite is a search-based unit test generation framework for Java that uses a genetic algorithm to evolve test suites over a series of generations, forming new populations each generation by retaining, mutating, and combining the fittest solutions. It is actively maintained and has been successfully applied to a variety of projects [78]. In this study, we implemented AFFS in EvoSuite version 1.0.7.

We call our approach EvoSuiteFIT. At a user-defined interval, the reinforcement learning algorithm alters the current set of fitness functions and refines its estimation of their ability to increase the goal’s attainment. Throughout the test generation process, we will also retain an archive of tests that satisfy the given goal to ensure that the final test suite is effective. The modified procedure is illustrated in Figure 3.2, and will be explained in Section 3.4.

EvoSuiteFIT is available from

https://github.com/hukh/evosuite/tree/evosuitefit

In Sections 3.1-3.2, we will explain the UCB and DSG-Sarsa algorithms. In Section 3.3, we give an overview of the EvoSuite test generation framework. Finally, in Section 3.4, we explain how AFFS is implemented into EvoSuite and present an overview of new fitness and reward functions implemented as part of our approach.
Algorithm 1 Overview of the UCB algorithm.

1: **Initialization:**
2: \( \text{max} = 0 \)
3: \( \text{for} \ a = 0 \ldots \text{number of actions} \ \text{do} \)
4: \( \text{numberTimesSelected}_a = 0 \)
5: \( \text{sumReward}_a = 0 \)
6: \( \text{end for} \)
7: **Each time an action is selected:**
8: \( \text{if} \ \text{numberGenerations} < \text{numberActions} \ \text{then} \)
9: \( \text{action} = \text{getAction(\text{numberGenerations})} \) \( \triangleright \) Try all actions once before using RL
10: \( \text{else} \)
11: \( \text{for} \ a = 0 \ldots \text{number of actions} \ \text{do} \)
12: \( \text{upperBound} = 0 \)
13: \( \text{if} \ \text{numberTimesSelected}_a > 0 \ \text{then} \)
14: \( \text{avgReward} = \left( \frac{\text{sumReward}_a}{\text{numSelection}_a} \right) \)
15: \( \text{upperBound} = (\text{avgReward} + c \sqrt{\ln \frac{\text{numberGenerations}}{\text{numberTimesSelected}_a}}) \)
16: \( \text{else} \)
17: \( \text{upperBound} = \text{doubleMaxValue} \)
18: \( \text{end if} \)
19: \( \text{if} \ \text{upperBound} > \text{max} \ \text{then} \)
20: \( \text{max} = \text{upperBound} \) \( \triangleright \) Update estimation of best action.
21: \( \text{action} = \text{getAction(a)} \) \( \triangleright \) Choose the action with the highest estimation.
22: \( \text{end if} \)
23: \( \text{end for} \)
24: \( \text{end if} \)
25: \( \text{numberTimesSelected}_{\text{action}} = \text{numberTimesSelected}_{\text{action}} + 1 \)
26: \( \text{return action} \)

3.1 Upper Confidence Bound (UCB) Algorithm

In the \( n \)-armed bandit problem, an agent is presented with a machine with \( n \) arms. Each time the agent chooses an arm, they will get a reward. Naturally, this agent will seek to identify the arms that give them the most reward. Even if the reward earned is non-deterministic, it is likely that certain arms will give more reward “on average”. The problem, then, is to identify the arm that will give the greatest improvement in reward when chosen and to keep choosing that one until time runs out or the maximum reward is attained. This is challenging, of course, because one must decide to exploit their current knowledge—choose the arm that you currently think is the best—or to explore—to refine your expected reward by trying a new or previously suboptimal option. Exploitation will lead to short-term improvement, but risks missing out on potentially greater gains in the long-term. However, too much exploration also risks resulting in a low reward by repeatedly trying poor options in
the hope they improve. Approaches to the n-armed bandit problem seek to balance exploration and exploitation in an effective manner.

The Upper Confidence Bound (UCB) algorithm is well-suited to addressing n-armed bandit problems [82]. Each time a choice is made, UCB selects an action with a higher expected reward than the other possible actions. Each action returns a numerical value that is considered as the reward of taking that action. This means that a testing goal that is to be optimized using this approach requires the definition of a reward function representing the improvement attained in that goal by taking an action. In Section 3.4, we discuss the specific reward functions used for each of our three goals. In contrast to fitness functions, these can be relatively simple functions. One could even use existing fitness functions and measure reward as the change in that score from the score in the previous generation.

Algorithm 1 outlines the UCB algorithm. For a selected action $A$ at time step $t$ (represented as $A_t$), the reward $R_t$ represents the corresponding reward of taking action $A_t$. Using this notation, the expected reward of action $a$ is $q_e = E[R_t|A_t = a]$. We apply the Upper Confidence Bound to select the action [82]:

$$A_t = \text{max}[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}]$$

(3.1)

where $A_t$ represents the index of the combination that gives the highest expected reward. The $c$ term represents the confidence level, determining the balance between exploration and exploitation in the algorithm. The value of $c$ needs to be larger than 0. Otherwise, the algorithm will behave in a purely greedy manner. $Q_t(a)$ denotes the estimated value of choosing a combination of fitness functions $(a)$, which can be calculated as:

$$Q_t(a) = \frac{1}{N_t} \sum_{i=1}^{t-1} R_t(a)$$

(3.2)
This equation represents the total reward of a combination \( a \) divided by the number of times that combination had been selected until the time \( t \). In this project, \( t \) denotes the number of generations of evolution that have occurred.

Reinforcement learning approaches generally attempt to associate the reward of taking an action with a particular state. To control the size of the state space, we represent the state as a feature vector describing the current state of a test suite using a set of attributes. In this case, the state is represented using the current set of fitness functions, the current fitness value for that set of functions, the suite size, the coverage of the subgoals associated with the fitness function\(^1\). These are the features we consider relevant when describing a test suite.

UCB is an example of a tabular solution method, where it attempts to associate rewards with specific states. It logs those reward expectations in a table or list structure, and attempts to identify the exact reward that would be gained in that state. However, finding an exact solution is not feasible when the states space is large or continuous. This is a potential limitation of this approach during test generation, as the state space of even our limited representation is large, and our feature vector representation could potentially be met be a large number of actual test suites, as it is a summarization of facets of the suite rather than the exact suite itself (i.e., the set of test cases). To address this potential limitation, we also use a second algorithm, DSG-Sarsa, which generalizes expectations from previously-encountered states [83].

### 3.2 Differential Semi-Gradient Sarsa (DSG-Sarsa)

A special class of reinforcement learning algorithms is known as approximate solution methods [83]. Approximate methods generalize from previously encountered states, therefore, approximate methods are appropriate for problems with a large or

\(^1\)For example, in Strong Mutation Coverage, this would be the percent of mutants detected through an observed difference in output.
unconstrained state-space where finding exact solutions is not feasible with limited
time [18]. As test case generation has a potentially vast state space—even using
a feature vector to summarize that state—we have explored using an approximate
solution method, Differential Semi-Gradient Sarsa (DSG-Sarsa) [83].

DSG-Sarsa is semi-gradient, enabling continual and online learning. Relevant to
our application domain, the algorithm is well-suited to problems in which there is no
termination state. This is an “on-policy” method, which means that it tries to improve
the policy that the agent has in place to make decisions. The agent leverages from
past experiences to decide when to vary between exploitation and exploration [83].
On-policy methods may be better suited to our application domain than off-policy
methods. On-policy adjustment will allow more exploration than exploitation when
necessary—this may be beneficial, given a large number of potential combinations of
fitness functions that could be chosen.

An overview of DSG-Sarsa is presented in Algorithm 2. Each round, an action—a
choice of fitness functions—is applied, and the test suite evolves to a new state $S'$,
with observed reward $R$. We then use this information to choose a new action $A'$,
using the formula:

$$
\hat{q}(S, A, W) = W^T \cdot X(S, A) = \sum w_i x_i(S, A) \quad (3.3)
$$

This action-value function is calculated by the inner product of weights and feature
vectors. $X(S,A)$ is the feature vector: $X(s, A) = (x_1(S, A), x_2(S, A), \ldots, x_d(S, A))$.  

---

Algorithm 2: Overview of the DSG-Sarsa algorithm.

```
1: After action $A$ is taken, the test suite is in state $S$
2: if numberGenerations < numberActions then
3:     $A_{new} = \text{getAction(numberGenerations)}$  \hfill $\triangleright$ Try all actions once before using RL
4: else
5:     Select action $A_{new}$ as a function of $q(S_0, \ldots, w)$ using $\epsilon$-greedy policy
6: end if
7: Observe $S_{new}$ and the reward after taking the action
8: numSelection $A_{new} = \text{numSelection}A_{new} + 1$
9: $\delta = \text{reward} + \text{averageReward} + q(S_{new}, A_{new}, w) - q(S, A, w)$  \hfill $\triangleright$ Update the error function
10: averageReward $= \text{averageReward} + \beta \cdot \delta$  \hfill $\triangleright$ Update average change in reward
11: $w = w + \alpha \cdot \delta \cdot q(S, A, w)$  \hfill $\triangleright$ Update the weight vector
12: $A = A_{new}$
13: $S = S_{new}$
```
The feature vector describes the current state of a test suite using a set of attributes. In this case, the state is represented using the current set of fitness functions, the current fitness value for that set of functions, the suite size, the goal coverage. These are the features considered relevant when describing a test suite.

$W$ represents a weight vector, used to bias action selection [83]. A weight is provided for each feature, and illustrates the importance of each feature in respect to its contribution to the action value. The weight for an action is updated each round using the semi-gradient with delta, controlled by the learning rate:

$$W_{t+1} = W_t + \alpha \delta \nabla \hat{q}(S_t, A_t, W_t)$$  \hspace{1cm} (3.4)

To evaluate the chosen action, the algorithm calculates the error function ($\delta$), which represents the difference between the immediate reward $R$ and the average reward $\bar{R_t}$ and the difference between the value of a target $\hat{q}(S_{t+1}, A_{t+1}, W_t)$ and the value of the old estimate $\hat{q}(S_t, A_t, W_t)$ [83]. In each iteration, the current action—selection of fitness function—is used to generate a new state and reward. We use an action-value function to generate the action $A'$. In our case, we use $\varepsilon - greedy$ [83]. The reward return is calculated in terms of the difference between the current and the average reward. The corresponding value function that is used for this type of return called a differential value function [83]:

$$\delta_t = R_{t+1} - \bar{R_t} + \hat{q}(S_{t+1}, A_{t+1}, W_t) - \hat{q}(S_t, A_t, W_t)$$  \hspace{1cm} (3.5)

$\bar{R_t}$ is the estimated average reward at time $t$, calculated as:

$$\bar{R}_{t+1} = R_t + \beta \delta$$  \hspace{1cm} (3.6)

$\beta$ is an algorithm parameter that represents the step size of updating the average reward. The notation $t$ represent the the time step (the number of generations).

By using the average reward, we consider the immediate reward as important as a delayed one. This means that we treat all fitness function combinations impartially.
without bias toward combinations that were selected first. Thus, there is no priority for the chosen combinations other than effectiveness.

3.3 EvoSuite Overview

We have implemented both reinforcement learning algorithms in the EvoSuite unit test generation framework. EvoSuite targets classes written in the Java language, and produces complete JUnit test cases that initialize the class-under-test, calls its methods with generated input, and applies generated assertions to check the results. EvoSuite is a search-based test generation framework, and uses different metaheuristic search techniques to generate test input \cite{33, 31}. The general test generation process in EvoSuite is depicted in Figure 3.1.

EvoSuite takes, among other configuration options, a CUT, a set of chosen fitness functions, and a search budget—the time allocated to the test generation process. An initial population of test suites is generated randomly, then a metaheuristic algorithm evolves that test suite over a series of generations until the search budget

\footnote{Assertions are generated using the class-under-test, which means that generated assertions are not useful for fault detection in the tested code. Instead, assertions are used for regression testing scenarios or to check for differences between two versions of a class. Human-written assertions are needed for direct fault detection.}
is exhausted. In this research, we have integrated adaptive fitness function selection into the standard Genetic Algorithm (GA).

Each generation, the GA evaluates the members of the current population (each test suite) using the chosen fitness functions. Then, a new population is formed by retaining high-scoring solutions, mutating solutions, forming new solutions by combining elements of parent solutions (crossover), and generating a small number of new random solutions to maintain diversity. After the search budget has been exhausted, the best solution will go through a minimization process in which test cases that cover redundant goals are removed (using the goals set by the current fitness functions). For example, if one of the fitness functions represents the Branch Coverage, a test that does not cover additional goals not covered by already-selected tests will be removed. At the end, a small-but-effective test suite will be returned.

EvoSuite supports a large number of fitness functions for test generation [73]. We make use of nine of these functions in our work:

- **Exception Count**: A count of the unique exceptions thrown by a test suite.
- **Branch Coverage**: A test suite satisfies Branch Coverage if all control-flow branches are taken during test execution. For each program statement that can cause the execution path to diverge—such as if and case statements—test input should ensure that at all potential outcomes are covered at least once [71]. To guide the search, the fitness function calculates the branch distance from the point where the execution path diverged from the targeted branch. If an undesired branch is taken, the function describes how “close” the targeted predicate is to be true, using a cost function based on the predicate formula [9].
- **Direct Branch Coverage**: Branch Coverage may be attained by calling a method directly or indirectly—i.e., a method call within a method that was directly called. Direct Branch Coverage requires each branch to be covered
through a direct method call, while standard Branch Coverage allows indirect coverage. Each can detect faults missed by the other [38].

- **Line Coverage**: A test suite satisfies Line Coverage if it executes each non-comment source code line at least once. To cover each line, EvoSuite tries to ensure that each basic code block is reached. For each conditional statement that is a control dependency of some other line in the code, the branch leading to the dependent code must be executed.

- **Method Coverage (MC)**: Method Coverage requires that all the CUT’s methods are executed at least once, through direct or indirect calls.

- **Method Coverage (Top-Level, No Exception) (MNEC)**: Generated test suites sometimes achieve high levels of Method Coverage by calling methods in an invalid state or with invalid parameters. MNEC requires that all methods be called directly and terminate without throwing an exception.

- **Output Coverage (OC)**: Output Coverage rewards diversity in the method output by mapping return types to a list of abstract values [7]. A test suite satisfies Output Coverage if, for each public method in the CUT, at least one test yields a concrete return value characterized by each abstract value. For numeric data types, distance functions offer feedback using the difference between the chosen value and target abstract values.

- **Weak Mutation Coverage (WMC)**: A test suite satisfies weak mutation coverage if, for each mutated statement, at least one test detects the mutation. The infection distance guides the search, a variant of branch distance tuned towards reaching and discovering mutated statements [32].

- **Strong Mutation Coverage (SMC)**: Weak Mutation Coverage ensures that the mutated line of code is reached. However, it makes no guarantees that the infected program state is noticed by the tester. Strong Mutation Coverage
Figure 3.2: How reinforcement learning fits into the test generation process.

**Algorithm 3** Overview of the AFFS hyperheuristic. NORL = reinforcement learning is not used, skipIter = a user defined number of generations to improve the population before the fitness functions are changed (generally 3-5).

1: while searchBudget > 0 do
2:   evolvePopulation() ▷ Use fitness functions to evolve the population.
3:   sortPopulation() ▷ Sort by fitness score.
4:   if generation % skipIter = 0 and approach ≠ NORL then ▷ Update reward estimate.
5:     bestSolution = GetBestIndividualFromPopulation()
6:     reward = CalculateReward()
7:   end if
8:   if approach == DSGSARSA then
9:     DSGSarsa(iterNum, bestSolution.coverage, bestSolution.size(), reward, bestSolution.fitness)
10:   else if approach == UCB then
11:     UCB(reward, action)
12:   end if
13:   action = GetBestFFCombination() ▷ Determine new fitness functions.
14:   iterNum++
15:   if current_FF == previous_FF then
16:     generation++
17:   end if
18: end while

adds an estimation of the likelihood of propagation, the propagation distance, by estimating the impact of corrupted state [32].

Rojas et al. provide more details on each of these fitness functions [73]. We additionally implemented a Test Suite Diversity fitness function based on the Levenstein distance, which we will discuss in Section 3.4.2

34
3.4 Implementation of AFFS within EvoSuite

We have implemented both reinforcement learning algorithms in EvoSuiteFIT, and integrate their use into the standard GA. At a user-defined interval, the RL algorithm will choose a new set of one to four fitness functions. The specific sets of fitness functions are goal-dependent, and will be explained in the following subsections. The modified process is illustrated in Figure 3.2. Algorithm 3 provides an overview of the reinforcement learning implementation in EvoSuiteFIT.

AFFS is an online learning approach. The RL algorithm learns its policy during the test generation process, adapting to the CUT and the evolving state of the test suite. This stands in contrast to an offline process, which would attempt to apply a policy learned in an earlier process. We do not attempt to transfer learned policies to new classes in this work [52]. The differences between classes may result in poor transfer success. However, this is a topic we will consider in future work.

In the beginning, EvoSuiteFIT will make sure that all the actions have been tried once before it starts using the standard UCB or DSG-Sarsa selection mechanisms. This allows seeding of reward estimations. Before the initial selection occurs, the list of actions is randomized to avoid an ordering bias. This is important, as the population of test suites is shaped by the action used each generation. After this stage, every time the RL algorithm makes a selection, the set of chosen fitness functions will change unless the currently-selected combination is exploited.

After changing the fitness functions, EvoSuiteFIT will proceed through the normal population evolution mechanisms, judging solutions using the new set of fitness functions (lines 2-3 in Algorithm 3). We use the reformulated population to calculate the reward—the gain in goal attainment from choosing an action (line 6 in Algorithm 3). Reward functions, too, are goal-specific and will be explained in the following subsections. Then, we use this reward to update the expectations of the RL algorithm. For UCB, we store the accumulated reward of each combination alongside
the number of times each is selected $N_t$, so we can calculate the average reward (line 10). Over time, the combination that gains the highest reward will be more likely to be selected again until reaching convergence. For DSG-Sarsa, after getting the reward, the new combination is selected using the learned policy. Based on the new and current combination, the new and current state, and the reward, the average reward and the weight of the state is updated (line 8). Then the current fitness function combination will change to the new one (lines 12-13).

After experimentation, we found that changing the fitness functions every three to five generations allows enough time to adequately adjust reward expectations. Fewer generations do not allow sufficient time for the chosen fitness function combination to reshape the test suite. This means that the GA will have a short time to reshape the population before reward is evaluated (line 4 in Algorithm 3).

In EvoSuiteFIT, test cases that cover a set of chosen goals can be retained in a test archive during the search and optimization process to prevent loss in coverage as the test suites are reshaped. Normally, this archive is based on the goals of the static set of fitness functions chosen when test generation starts. However, as we use RL to change the fitness functions, we have altered how the test archive is used. Instead, we use a set of goals associated with high-level testing goal. In the following subsections, we will discuss the goals used. After the search process completes, the archive is used to help produce the final test suite. This prevents the loss of test cases that may contribute to effectiveness due to changes in fitness functions. After generation concludes, the best solution is minimized with respect to this set of goals. The archive then is used to supplement this suite with coverage of any missing goals.

In the following subsections, we will discuss specific adaptations made for the three high-level testing goals: exception discovery, test suite diversity, and Strong Mutation Coverage.
3.4.1 Adaptations for Goal: Exception Discovery

**Fitness Function Combinations:** EvoSuiteFIT chooses a combination of one to four of the following fitness functions: Exception Count, Branch Coverage, Direct Branch Coverage, Line Coverage, Method Coverage, MNEC, Output Coverage, and Weak Mutation Coverage. Initial experimentation revealed that many effective combinations include the exception count, even though the exception count is rarely effective on its own. Therefore, we filtered the initial set of combinations down to all combinations of one to four fitness functions that include the exception count as one of the choices. EvoSuiteFIT can choose from 64 actions (different sets of fitness functions) when attempting to optimize the high-level goal of exception discovery.

**Reward Function:** We measure reward as the sum of exceptions discovered during the entire generation process and the exceptions thrown by the current best test suite, encouraging discovery and retention of exceptions.

**Goals Used for Minimization and Archiving:** We use the set of discovered unique exceptions as the goals used in minimization and when archiving tests. A test that forces the CUT to throw a particular exception covers the “goal” for that exception. When the test suite is minimized, it is minimized to ensure that all discovered unique exceptions are covered. Tests detecting any exceptions no longer covered by that suite will be added from the archive, preventing loss of coverage.

3.4.2 Adaptations for Goal: Test Suite Diversity

**New Fitness Function:** EvoSuite does not already contain a fitness function intended to promote test suite diversity. Therefore, we have implemented a fitness function to measure test suite diversity based on the Levenshtein distance [77]. The Levenshtein distance is the minimal cost of the sum of individual operations—insertions, deletions, and substitutions—needed to convert one string to another (i.e., one test to another). We compare the text of test cases within a test suite.
The distance between two tests \((ta, tb)\) can be calculated as follows [77]:

\[
lev_{ta, tb}(i, j) = \begin{cases} 
  \max(i, j) & \text{if } \min(i, j) == 0 \\
  lev_{ta, tb}(i - 1, j) + 1 & \\
  \min(lev_{ta, tb}(i, j - 1) + 1) & \text{otherwise} \\
  lev_{ta, tb}(i - 1, j - 1) + 1_{(ta_i \neq tb_j)}
\end{cases}
\] (3.7)

where \(i\) and \(j\) are the letters of the strings representing \(ta\) and \(tb\). To calculate the diversity of a test suite \((TS)\), we calculate the sum of the Levenshtein distance between each pair of test cases:

\[
div(TS) = \sum_{ta, tb} lev_{ta, tb}
\] (3.8)

To attain a normalized value between 0-1 for use in a multi-fitness function environment, we then calculate and attempt to minimize the final fitness as:

\[
\frac{1}{1 + div(TS)}
\] (3.9)

Algorithms 4-6 explain how this fitness function is calculated, starting with the high-level fitness calculation in Algorithm 4. The process iterates through the test cases in a given test suite (line 2). Before calculating the distance, the variables and their values are extracted (line 5). This includes extracting numeric primitive variables, null variables, strings, arrays, instance and class fields, methods, and constructor statements. Our analysis also includes partial assessment of aliasing (lines
Algorithm 5 Calculate diversity value of extracted statements.

1: diversity = 0  \hfill ▷ Initialize fitness score.
2: if testCaseStatements.size() <= 1 then
3:  return 0  \hfill ▷ If there are only 0-1 tests, there is no diversity to measure.
4: else
5:  for index_i = 1, 2, ..., testCaseStatements.size() do
6:    for index_j = index_i + 1, ..., testCaseStatements.size() do  \hfill ▷ Compare pairs of tests.
7:      testStatements_i = testCaseStatements.get(index_i)
8:      testStatements_j = testCaseStatements.get(index_j)
9:      for statement_i in testStatements_i do
10:         for statement_j in testStatements_j do  \hfill ▷ Compare each pair of statements.
11:            diversity += LevenshteinDist(statement_i, statement_j)  \hfill ▷ See Algorithm 6.
12:         end for
13:      end for
14:   end for
15: end if
16: return diversity  \hfill ▷ Return fitness score.

Algorithm 6 Levenshtein Distance calculation.

1: distance = int[statement_i.length() + 1][statement_j.length() + 1]  \hfill ▷ Create matrix.
2: for index_i = 1, 2, ..., statement_i.length() do
3:   distance[index_i][0] = index_i  \hfill ▷ Initialize rows.
4: end for
5: for index_j = index_i + 1, ..., statement_j.length() do
6:   distance[0][index_j] = index_j  \hfill ▷ Initialize columns.
7: end for
8: for index_i = 1, 2, ..., statement_i.length() do
9:   for index_j = index_i + 1, ..., statement_j.length() do  \hfill ▷ Compare each pair of characters.
10:      if statement_i.charAt(index_j - 1) == statement_j.charAt(index_i - 1) then
11:         cost = 0  \hfill ▷ Characters are the same.
12:      else
13:         cost = 1  \hfill ▷ Characters are not the same.
14:      end if
15:      distance[index_i][index_j] = min(distance[index_i - 1][index_j] + 1, distance[index_i][index_j - 1] + 1, distance[index_i - 1][index_j - 1] + cost)  \hfill ▷ Distance is the least expensive option to change one character to the other (deletion, insertion, substitution).
16:  end for
17: end for
18: return distance[statement_i.length()][statement_j.length()]  \hfill ▷ Return final distance score.

6-7). Consider the following fragment: \texttt{String x = "var";} \texttt{String y = x;} \texttt{String z = y;} Variables \texttt{x, y,} and \texttt{z} are different, but are initialized with the same value. These should not be considered diverse, so we statically trace the reference to the original value when possible to attain a more accurate estimation of diversity. The list of filtered statements is then used to calculate fitness (line 11). The fitness is normalized (line 12), then stored (line 13).

Algorithm 5 explains the diversity calculation. Each pair of test cases is compared (lines 5-6). From each pair of tests, each pair of statements is compared (lines 9-10).
The Levenshtein distance is calculated between each of these pairs and added to the diversity score (line 11) and returned to the core process (line 17). The Levenshtein distance calculation is, then, explained in Algorithm 6. We use a classic matrix-based approach [67] where the characters in the two strings are compared, and the final value stored in the matrix is returned.

**Fitness Function Combinations:** EvoSuiteFIT chooses a combination of one to four of the following fitness functions: Diversity, Exception Count, Branch Coverage, Direct Branch Coverage, Method Coverage, MNEC, Output Coverage, and Weak Mutation Coverage. To constrain the number of combinations, we use only the combinations that include the diversity score and remove a small number of semi-overlapping combinations (i.e., Branch and Direct Branch). Ultimately, EvoSuiteFIT can choose from 44 combinations of fitness functions.

**Reward Function:** The change in the diversity fitness score is used as the reward function to identify the actions that best increase diversity.

**Goals Used for Minimization and Archiving:** Unlike exception discovery and Strong Mutation Coverage, test suite diversity lacks a natural set of discrete goals. Test suites can be diverse in many different ways, and coverage lacks a direct analogue. To support the archiving and minimization process, we adapt the set of goals from Method Coverage. This means that suites are minimized using their coverage of the source code. This is a low-cost calculation that does not have a noticeable effect on overhead, while retaining diversity in the final suite.

### 3.4.3 Adaptations for Goal: Strong Mutation Coverage

**Fitness Function Combinations:** EvoSuiteFIT chooses a combination of one to three of the following fitness functions: Strong Mutation, Exception Count, Branch Coverage, MNEC, Output Coverage, and Weak Mutation Coverage. This provides
EvoSuiteFIT with 31 combinations of fitness functions to choose from. Unlike the other two goals, not all combinations include the Strong Mutation Coverage.

**Reward Function:** We use the mutation score as the reward function. This is the percentage of mutants detected: \[
\frac{\text{DetectedMutants}}{\text{TotalNumberofMutants}} \times 100.
\] The mutation score can be calculated using either Strong or Weak Mutation Coverage. The difference is that, in Strong Mutation Coverage, we require a noticeable difference in class output between the original and mutated version. In Weak Mutation Coverage, the mutated statement simply must be reached and the internal state of the execution must be corrupted at that point.

Strong Mutation Coverage is much more expensive to calculate than Weak Mutation Coverage. To reduce the overhead that would occur when calculating Strong Mutation Coverage during reward estimation refinement, we iterate between Weak Mutation and Strong Mutation. The reward from choosing an action is the improvement in the mutation score.

**Goals Used for Minimization and Archiving:** We use the set of goals calculated in order to attain the final Strong Mutation Coverage score. That is, each mutant that can be detected is a discrete goal. Suites are minimized in terms of coverage of these mutants and tests from the archive are added to the final suite to detect any mutants missed by the unaugmented suite.
CHAPTER 4

RELATED WORK

This chapter will provide an overview of related work on hyperheuristic search-based software testing (SBST) approaches, as well as test generation research related to exception discovery, test suite diversity, and strong mutation coverage. The aim of this chapter is to give insight on what other researchers have done and their achievements in topics related to this dissertation.

4.1 HYPERHEURISTICS IN SEARCH-BASED SOFTWARE TESTING

Hyperheuristic search has been employed in addressing multiple several search-based software engineering problems. This section will review this work in the context of software testing. Broadly, we categorize this work into three common subject areas, as well as a fourth “other” category.

Fitness function selection has been performed by hyperheuristic search in other domains, such as production scheduling [19, 68]. However, our approach is the first automated technique for optimizing the set of fitness functions used during test generation. Related work, largely, uses the hyperheuristic to tune crossover and mutation operators used by an evolutionary algorithm.

4.1.1 COMBINATIONAL INTERACTION TESTING (CIT)

A system may have many input variables or components that can provide various configurations. Testing all of these configuration is costly. Combinatorial Interaction Testing (CIT) is used to systematically sample the configuration space by deriving
a set of test cases that covers all N-way interactions between parameter values [55]. Instead of attempting to test all combinations of values for all parameters, covering all N-way parameter interactions is thought to attain sufficient coverage. Commonly, this approach is used to cover all 2-way interactions between parameters values.

Jia et al. [54, 55] used reinforcement learning to tune the metaheuristic for Combinatorial Interaction Testing, improving performance by learning the best algorithm for test generation for targeted problems. Because each program is different, the choice of search heuristic can have a impact on the effectiveness of the results for that program. Therefore, finding the right heuristic to apply is essential. To solve this problem, they proposed an approach that can solve general SBST problems by using two layers of optimization. During each iteration, an outer and inner layer interact. The outer layer contains one or more metaheuristic search algorithms performing the overall optimization, while the inner layer contains heuristics that can tune those algorithms (i.e., heuristics for prioritization, minimization, and constraint solving). The reinforcement learning agent learns the low-level heuristics best expected to improve the high-level search.

Addressing the CIT problem, Jia et al. [55] proposed using the Simulated Annealing (SA) algorithm in the outer layer and using an n-Armed Bandit approach for learning and choosing the best operator(s) (out of six operators) to tune SA. In each iteration, SA asks the reinforcement learning to select the 'best' operator for the current state. The SA will apply the operator and update the fitness value accordingly. Their approach equaled or outperformed all other approaches for a set of 26 real-world subjects.

Zamli et al. used a hyper-heuristic approach for CIT [89]. They proposed using Tabu search as a high-level hyperheuristic (HHH) to select a low-level heuristic (LLH) from four algorithms. The HHH used three operators to choose an LLH that can be used in a next step. These operators are improvement (comparing the current
best individual in the population against the previous best individual), diversification (calculating the hamming distance of the best individual to evaluate how far the best are from the candidate solution), and intensification (calculated by taking the cumulative sum of the hamming distance for the population’s individuals of the best and compared the value against the intensification from the previous iteration). Their experiment executed HHH 30 times and comparing the results against other non-hyper-heuristic strategies. They found that HHH performed better than other strategies in term of test suite size and it was able to generate reasonable solutions at a low cost.

Later, Zamli et al. used hyperheuristic search to learn optimal selection and acceptance mechanisms used by the metaheuristic in CIT [90]. They experimented with four different selection mechanisms: Exponential Monte Carlo with Counter (EMCQ), Choice Function (CF), Improvement Selection Rules (ISR), and the newly developed Fuzzy Inference Selection (FIS). These selection mechanisms are used to select the “best” search operator that can be used in the current state. They observe from the experiment that FIS yields the best average test suite size. FIS and ISR were the slowest approaches, due to the computation overhead of calculating the Hamming distance, which is used to improve diversification. In general, the hyperheuristic approach outperforms the standard metaheuristic approach.

Din et al. also applied hyperheuristic search to CIT [21]. Their hyperheuristic is based on parameter-free choice functions, which use statistical information to rank low-level heuristics for selection. The proposed approach selects between four low-level heuristics that are used in generating pairwise tests. To make a decision, a choice function ranks the low-level heuristic dynamically based on the historical information. The ranking depends on calculating three measurements that provide effective exploration and balances between diversification and intensification. The diversification and intensification are more like exploration and exploitation. The approach
will focus on the current selection when it keep shown improvement. However, when there is no improvement the approach will diverse and explore more other options. To balance, they use two parameters (one for diversification and another for intensification) that change dynamically based on the previous results. The mechanism can decide how to focus on diversifying or intensifying the search to get a better solution. Their approach resulted in reduced test suite size.

Din and Zamli use Exponential Monte Carlo with Counter (EMCO) as a hyper-heuristic to select a low-level heuristic in CIT [22]. EMCO rewards a search operator by re-selecting the same operator in the next iteration. Operators penalized due to poor performance can still be chosen by EMCO randomly to refine expectations.

Ahmed et al. [1] compare EMCO against an improved version using Q-learning, called Q-EMCO, to select the best operator based on historical information. Q-learning is an off-policy reinforcement learning algorithm in which the learned action-value function (Q) is independent from the followed policy [82]. That. Q-EMCO takes into account historic information, while traditional EMCO only uses the previous iteration. Their evaluation was based on applying the proposed approach to 37 industrial control systems. The results show that Q-EMCO outperforms EMCO in terms of test suite size, but EMCO is faster as it lacks the overhead needed to maintain the Q-learning table. They found that Q-EMCO can provide high coverage over the code, but achieves low fault detection. Also, they found that it is cheaper for an experienced engineer to generate the test suite manually.

4.1.2 Integration Testing Order (ITO)

In an object-oriented system, classes often work together to complete tasks. However, when testing, we want assurance that errors are due to an issue in the class-under-test and not from its interaction with other classes. We can test a class in isolation using stubs—mock objects that return predictable values. However, creating stubs is
a high-effort task. If classes are tested in the correct order, we can reduce stubbing costs by using already-tested—and reliable—classes instead of creating stubs. The Integration Test Order (ITO) problem is the determination of the order in which classes are tested as they are integrated into the system [56, 86].

Guizzo et al. used a reinforcement learning-based hyper-heuristic search to tune the metaheuristic algorithm for optimizing the integration and test order problem [44, 47]. In this work, the authors used a hyper-heuristic to select an operator to be used in optimization. They utilized two different hyperheuristics, a Choice Function and Multi-Armed Bandit. The hyperheuristic then selects the lower-level heuristic operators, which will be used during optimization—tuning the optimization algorithm to the particular problem being solved. They selected from nine operators, which are the combination of three different crossover and two different mutation operators (with the third option of no mutation). To evaluate the approach, they tested it with seven systems. The results shows that the proposed approaches outperform NSGA-II, and that HTIO-CF preforms slightly better the HTIO-MAB based on the hypervolume indicator.

In later work, Guizzo et al. used hyperheuristic search to select an operator that can be executed by Multi-Objective Evolutionary Algorithms (MOEAs) to provide a solution for the ITO problem [46]. In this work, they again use Choice Function and Multi-Armed Bandit. These two approaches are used to select an operator (crossover or mutation) that can be executed next by the MOEA. The MOEA used two different sets of fitness functions. They used two objectives and four objectives that can be utilized to evaluate the generated solution. The solutions were compared against two standard MOEAs, MOEA/DD and NSGA-II. They ran each algorithm 30 times on a total of seven systems. They then compared the results by calculating the hypervolume and inverted generational distance. For hypervolume, the Choice Function outperforms NSGA-II for all the systems in both objective sets. However, in two sys-
tems in both objectives, the results of NSGA-II were very close to the hyperheuristic. For one system, NSGA-II was better than the Multi-Armed Bandit. Inverted Generational Distance shows mixed results of better or equal between MOEA/DD and the two hyperheuristic approaches. Vargha-Delaney’s A effect size shows that Choice Function and Multi-Armed Bandit performance is very similar. However, the Choice Function outperforms NSGA-II with a large effect size.

Guizzo et al. also applied a hyperheuristic to the NSGA-II MOEA to address ITO in Google Guava [43]. They used a hyperheuristic to select an operator from nine different heuristics for the evolutionary algorithm. They applied and compared three other heuristic selection methods: Random, Multi-Armed Bandit, and Choice Function. The results showed that the Choice Function outperforms the random approach and the standard NSGA-II, but there is no remarkable difference from Multi-Armed Bandit.

Mariani et al. introduced an approach that depends on an offline hyperheuristic named GEMOITO to generate MOEAs to solve the ITO problem [65]. GEMOITO is based on grammatical evolution, a type of genetic programming that uses a grammar to generate programs—generating variants of a MOEA tuned to the current problem to be solved. They found that their approach generated MOEAS that are better or equivalent to compared approaches. However, their approach required a high computational cost to generate the MOEAs. Their approach was less flexible than online hyper-heuristics, and required training data with a large variety of cases.

Guizzo et al. later used design patterns to improve the design of MOEA to reduce coupling and increase reusability of components [45]. They implemented the patterns into GEMOITO. To test the validity of their proposal, they compared GEMOITO with design patterns against the standard GEMOITO—applying both approaches to the ITO problem. They found that they were able to reuse MOEA components without decreasing the quality the algorithm results.
4.1.3 Software Product Line Testing

Ferreira et al. proposed the use of hyperheuristic search in software product line (SPL) testing [28]. Software Product Lines are sets of systems that share a common set of features that are customized for particular market segments or customers. SPL depends on a feature model, representing all variabilities and commonalities. The feature model is used to derive products for SPL testing. In practice, all products cannot be tested. Therefore, search-based approaches can be used to select “interesting” ones to focus on. Building on earlier work [23, 81], the authors proposed using a hyperheuristic MOEA to find a select product variants for testing. Their approach considers four objectives: the number of products, pairwise coverage, mutation score, and dissimilarity of products. UCB and a random selection were implemented to select suitable operators for crossover and mutation for each iteration. For better results, they used what they called a credit assignment, which uses a sliding window to store fitness improvement rate for recent operators. The fitness improvement rate is used later to calculate a reward that is also used during the operator selection.

For evaluation, they ran an experiment that includes evaluating four different versions MOEAs (NSGA-II, SPEA2, IBEA, and MOEA/D-DRA) with UCB and random selection. The results show that using NSGA-II with UCB provides the best solutions. With UCB, it can find higher number of solution when the number of products is large; however, it provide equal number of solutions when the number of products is small. It was also observed that NSGA-II with random selection could give better results when used for small instances, and that NSGA-II with UCB is more suitable for large instances. In general, hyperheuristic approach can provide better performance (due to high number of solution which dive more diversity) and larger effective size. However, this benefits come with expensive execution time.

Filho et al. proposed a hyperheuristic that uses grammatical evolution to generate MOEAs for SPL testing [30]. Their approach considers three factors—pairwise cover-
age, mutation score, and cost—and generates a MOEA using crossover and mutation operators tuned to the feature model being considered. Their evaluation includes a comparison against an online hyperheuristic and traditional NSGA-II. The proposed approach yields results better or statistically equivalent to similar approaches found in the literature.

Filho et al. extended this work [29, 62] to Preference-Based Evolutionary Multi-objective Algorithms (PEMOAs). PEMOAs consider user preferences during the search through Reference Points, resulting in a greater number of solutions in the region of interest. To do this, they Reference Points in NSGA-II in two different forms, Reference Point-based NSGA-II (R-NSGA-II) and Reference Solution-based NSGA-II (r-NSGA-II) [29]. In the second article, they then incorporated hyperheuristic selection based on Multi-Armed Bandit with Fitness Rate Rank to ensure solutions are in the region of interest [62]. The r-NSGA-II approach generates a reduced number of non-interesting solutions from the tester’s point of view.

4.1.4 Other Work

Kumari and Srinivas [60] used hyperheuristic search to tune software design—learning how to cluster classes for maximum cohesion and minimum coupling. This work applies reinforcement learning to select a low-level heuristic that will be used with an evolutionary algorithm to cluster software modules for further analysis.

Grechanik proposed an adaptive, feedback-driven approach to generating input designed to highlight performance issues [42]. Their technique, FOREPOST, initially generates test cases randomly, and the results are evaluated. Then, the results are fed to a machine learning classification algorithm, which will output a set of rules. These rules will be used in the next cycle as guidance to select input tests and generate test cases. With the time, the input tests will be modified and enhance until reaching the desired state in which the performance issue is located. To identify the bottleneck, the
authors suggest checking the invoked method and identify methods that are executed periodically in the bad tests, but not in the good tests. As a result, the proposed approach can generate input that can steer the application to consume more resource and increase the computation time, which helps in detecting the performance issues. This approach is not based on metaheuristic search, but can be considered similar, in that it uses feedback to improve test case generation.

In the area of performance testing, Moghadam et al. [50] worked on developing a framework that uses adaptive learning to generate test cases for stress testing. Performance testing is used to test the non-functionality aspect of the software in terms of efficiency of the software under test (SUT) perform well under normal operational conditions. Testers perform this type of testing looking for condition or configuration in which the software will break or perform poorly. Taking in consideration possible number of configuration or parameters, performance testing can be difficult and expensive to do. Moghadam et al. aim to develop an approach that utilize reinforcement learning to generate test cases that explore the tested software and find a breaking point. Their approach includes two phases, initial convergence and transfer learning. During these two phases the agent learn the basic experiences (of CPU, memory, and disk intensive) of the SUT. Then, it use what learn to generate test cases depending on the gain knowledge; in the same time, agent keeps learning going to update the experience with the new encounters. To achieve the learning, the approach includes four steps. First, the current state get detected by recording four measurements; memory, disk, and CPU utilization along with SUT response time. Then, agent applies an action base one possibility of that action to reduce utilize resources or increase workload. Applying action will continue until agent reach a break point. In third step, reward is calculated to be used in making decision. lastly, test cases generated using the experiences based on the previous steps. Moghadam et al. introduce a new adaptive approach to improve performance testing.
Bauersfeld et al. [12] introduced an automated testing approach for robustness testing of GUIs based on reinforcement learning. Building on their previous work [13], they introduce an approach to select input events for GUIs intended to improve coverage of deeply nested actions. They use Q-Learning to discover states and actions and learn the value function to maximize coverage of GUI actions. They do not evaluate the approach, instead focusing on proposing their idea.

4.2 Crash and Exception Discovery

One of the goals that we use hyperheuristic search to optimize is exception discovery, where we try to make the software crash by throwing an exception. Crashed software is not a desired outcome for users or developers. Therefore, past studies have examine how test generation can be used to encourage additional crashes or exceptions. This section reviews a relevant subset of this work.

Joffe et al. [57] use the results from an artificial neural network (ANN) classifier to construct a fitness function targeting crashes, which can be used in search-based test generation. They trained their ANN classifier on C programs to predict the likelihood of crashing, given a particular input. They modified American Fuzzy Lop—a search-based test generation tool—to consider the crash likelihood from the classifier. This work has limited generalization. They trained ANN in respect to execution traces of library calls to detect the conditions leading to a crash. However, this model does not work well on systems that do not use these libraries heavily. Reaching good generalization requires training the ANN on far more labeled data.

In contrast to the work by Joffe et al, which tries to predict a general likelihood of crashing, Romano et al. [75] focused on targeting null pointer exceptions. They aimed to provide an approach that can identify code that can cause this exception by looking at execution paths. The approach consists of four steps that work in sequence. The first step is using a program analyzer to generate the control flow graph, which will
be used in the second step to identify paths that could throw exceptions. Coverage of these paths is then targeted using search-based test generation. Although this approach is more likely to detect null pointer exceptions than a general test generation approach, coverage of these paths does not guarantee that a null pointer exception is triggered. Further, a fitness function based solely on null pointer exceptions is of very narrow use, compared to an effective approach that promotes all types of exceptions.

Due to inadequate detection of problematic exceptions in automated test case generation, Goffi et al. [41, 15] proposed the use of natural language processing to generate test oracles—assertions designed to assess the behavior of the system. Their approach extracts comments that are related to exceptional behaviors that can be thrown by a method or class. Then, these comments are translated into expression), which are used in generating test oracles. These assertions are used in test cases to improve detection of faults. To evaluate their approach, they integrate it into EvoSuite and Randop. The results show that the proposed approach can reduce false positive by about 33%. The approach was able to generate more failing tests, then EvoSuite or Randop alone. Extended work widens the range of behaviors that can be assessed by these oracles [15]. Their work, in contrast to ours, does not influence the selection of test inputs. Rather, it improves the likelihood of fault detection by existing inputs. Therefore, it could be combined with our approach, potentially improving fault detection further.

4.3 Test Suite Diversity

The second goal we use hyperheuristic search to optimize is test suite diversity. Providing diverse testing suites can improve code coverage and may lead to detecting new faults. Therefore, there is an interest in improving suite diversity. This section will provide an overview of attempts to improve diversity in search-based test generation.
Diversified test cases in unit test generation may influence the effectiveness of the test suite. Albunian investigated the impact of diversity on search-based test generation [2]. They proposed a phonotypic and genotypic representation to measure diversity. They studied the influence of five selection mechanisms and five fitness functions. They applied their approach to EvoSuite and selected branch coverage to be the target criteria with a 30-minute search budget. They ran the experiment on 347 classes, 10 times each. Their results show that the roulette wheel selection mechanism provided higher diversity than other mechanisms. However, their approach actually decreased code coverage. They did not assess the impact on fault detection.

Feldt et al. [27] proposed a new diversity fitness function based on normalized compression distance. They evaluated this fitness function on three Java projects and compared the results of their approach with random selection and a theoretical “best” coverage that could be achieved. The results show positive correlation between diversity and code coverage. The proposed approach does not provide higher code coverage than the “best” performance, but does provide higher code coverage and fault detection than random selection.

Diversity is also used in generating test cases for concurrent and multi-threaded systems. Ma et al. [63] proposed an adaptive approach that generated concurrent test cases targeting diversity metrics. They introduce two diversity metrics; static, which concerns diversity in structure, and dynamic, which is concerned with exposing untested thread schedules. They used these two metrics in three adaptive approaches—generating test cases using static, dynamic, and a combination of the two to select the optimal test cases among a finite number of candidates. They implemented their approach and studied nine open-source classes. The results show that, with the proposed approaches, they were able to generate test cases that provide high coverage in regard to thread schedules of concurrent data structures. Also, the dy-
namic approach can reduce the number of test cases required for exposing interleaving faults. However, the static approach requires higher computation time.

Vogel et al. [85] investigate using diversity metrics in search-based generation of test cases for Android mobile applications. They analyzed the fitness landscape of the default configuration of Sapienz, a search-based testing approach for Android. Their analysis showed that diversity decreases over time. As a result, they proposed an approach that used different metrics that focus on diversifying the population at the initialization and selection steps. These metrics are recalculated after each generation for analysis purposes. The feedback of these metrics is used to preserve and improve diversity during the search. To evaluate the approach, they assessed test generation on 76 apps. The final results show that Sapienz with diversity performs closely to Sapienz in respect of coverage. In regards to fault-finding, the proposed approach was able to detect more faults then without using diversity. However, with diversity, the test sequence was longer and test generation requires more time.

4.4 Strong Mutation Coverage

Many approaches to test generation for mutation coverage aim at satisfying weak mutation coverage, where the impact of a fault does not need to propagate to the output. Strong mutation coverage, which requires that the program output differs from the unmutated (correct) program, is harder to satisfy. Strong mutation coverage is the third goal that we attempt to optimize using our hyperheuristic search. Therefore, in this section, we examine related work on improving strong mutation coverage during test generation.

Fraser et al. [32] proposed a fitness function representation for strong mutation that is implemented in EvoSuite. In this work, they estimated propagation of change using an impact measurement, which measures the difference between control flow and data that results from running the tests on an original program and mutants. To
speed the process test execution, they only execute test cases that have reached the mutant and have zero “infection distance” (the impact estimation). They compare strong mutation with branch coverage in terms of coverage and mutation score. We use this fitness function in our work, and attempt to use hyperheuristic search to further improve optimization of this function.

Souza et al. [80] proposed an automated test generation approach for strong mutation using Hill Climbing, a simple local search algorithm. In this work, the author proposed a new approach that uses a fitness function based on weak and strong mutation. The proposed fitness function uses three metrics, called the Reach Distance, Mutation Distance, and the Impact Distance. These metrics are used to guide the search toward satisfying three goals; reaching the mutant, changing the program state, and propagating the state change to the program output. They perform their approach on C programs, and outperformed random testing by 19% in terms of detected mutants. However, the number of mutants they examined is quite small (maximum number of mutants is 166 for a case example), which makes it easy to achieve full coverage. They assess their approach on only a small number of programs (18 programs), which also makes it challenging to generalize to other programs.

Papadakis and Malevris [70] proposed using alternating variable method—a search algorithm—to generate tests to optimize a fitness function based on strong mutation. The proposed fitness function is composed of four parts. The first are the approach level and the branch distance, used in branch coverage to measure distance of the execution path from a targeted statement. They measure distance from covering the mutated line of code. The third is the mutation distance, which assesses how close program state is to being corrupted. Finally, the impact distance approximates the likelihood of the mutant impacting the output by quantifying how much of an effect the mutation had on the program state when exposed. They generated tests
targeting Java programs using their fitness function, and their approach was able to increase the number of detected mutants by about 7.6%.

An important note about all of these fitness function representations of strong mutation is that they are compatible with our approach, and could potentially be used within reward functions targeted by the hyperheuristic search. We used the strong mutation function proposed by Fraser et al. [32], as it was already implemented in EvoSuite. However, any of the other functions could have been implemented instead, and could be considered in future work.

In the domain of policy testing, Xu et al. [88] proposed using strong mutation to generate XACML policy tests automatically. Their approach is based on three constraints: reachability, necessity, and propagation. These constraints are used to capture the differences between mutants and original policies in terms of the responses to access requests. Their work includes evaluation of three methods Strong Mutation-based Testing (SMT), Non-Optimized Strong Mutation-based Testing (NO-SMT), and MC/DC, a metric based on assessing different outcomes of boolean expressions. The assessment was based on test size, time, and mutant detection. The results show that the approaches based on strong mutation have the biggest test sizes. They also provided better results by achieving 100% mutant detection, compared to 50%-70% for MC/DC. They were able to remove duplicate tests. However, SMT was also the slowest of the approaches. Strong mutation is not the most cost-effective approach, but it can help improve test cases.

Harman et al. proposed an approach that aims to achieve strong coverage of first and higher-order mutants [48]. Mutants that alter one line are “first-order” mutants, while higher-order mutants change multiple lines. Most mutation approaches are based on first-order mutants. Their approach, called SHOM, is a hybrid of dynamic symbolic execution (DSE) and search-based test generation aimed at overcoming limitations of earlier work with regard to higher-order mutants. The approach includes
applying three transformations to the program that reduce constraint and path analysis effort without impacting the semantics of programs under test. Their approach was able to detect up to 38% of first-order mutant and 48% of second-order mutants that were undetected by other approaches.

Strong mutation coverage is very expensive to measure, as it requires executing all test cases against all relevant mutants. Due to the high cost, there are many proposed methods to reduce computation cost. For example, Singh et al. [79] propose an approach that aims to reduce time cost by dynamically ending the execution of the mutant. Their approach falls between weak and strong mutation coverage. They suggested tracing the unit in two places (the beginning and the middle of the execution) to check for unplanned changes in state. They evaluate this approach on a small project, and they notice some improvement that can lead to better results. Zhu et al. [91] propose using compression strategies to reduce the cost. First, they suggested an enhancement of Formal Concept Analysis (FCA), which was used to derive maximal grouping. They used two new factors to enhance the grouping—mutation location and mutation operator type information. To do the compression, they clustered mutants based on similarity. These clusters were used to compress the mutant-test matrix to reduce the size of the matrix, i.e., removing some mutants from consideration. The resulting matrix is used for strong mutation analysis by executing tests on the remaining mutants. Their experiment includes evaluation of six different compression techniques on 20 open-source projects, and they compare the proposed approach against random sampling and weak mutation. They found that weak mutation produce higher absolute error (on average 23%) and 75% accuracy on estimating a strong mutation score. Using the compression techniques increases the speed of strong mutation analysis up to 94 times faster with accuracy larger than 90%. They also indicate that when mutation knowledge (location and operator type) are incorporated in compression techniques, they can reduce the error ratio and
accuracy. In general, the compression technique is better than random sampling and
weak mutation in terms of speeding up the strong mutation, accuracy on estimating
strong mutation score, and produce estimate error.
Chapter 5

Methodology

To better understand the effectiveness and applicability of adaptive fitness function selection, we have assessed EvoSuiteFIT using case examples from the Defects4J fault dataset [58] for each of our three goals—exception discovery, test suite diversity, and Strong Mutation Coverage. We wish to address the following research questions:

1. For each goal, is either EvoSuiteFIT approach more effective than test generation using static fitness function choices at attaining that goal?

2. For each goal, is either EvoSuiteFIT approach more effective than test generation using static fitness function choices in terms of attained fault detection?

3. What impact does the computational overhead from reinforcement learning have on the test generation process?

4. Are there observations that can be discerned in the combinations of fitness functions chosen by either EvoSuiteFIT approach that help explain the success (or lack of success) of an approach for a goal?

The first two questions provide us with an understanding of the effectiveness of EvoSuiteFIT compared to baseline approaches representing current practice. We hypothesize that adaptive fitness function selection is capable of increasing our attainment of difficult-to-optimize goals. We must evaluate whether that is true.

Increased goal attainment does not necessarily suggest higher likelihood of fault detection. However, each of the three goals we are maximizing are thought to be indicators of fault detection. That is, if the number of exceptions, suite diversity, or
Stron Mutation coverage are increased, it is theorized that the likelihood of fault
detection will rise as well. If EvoSuiteFIT is able to improve goal attainment, the
number of faults detected may increase as well. Note, however, that we are ask-
ing a broader question than whether increased goal attainment leads to increased
likelihood of fault detection. We are asking if any element of the adaptive fitness
function selection process increases that likelihood. AFFS is a complex process, and
other factors—like varying the fitness functions over time—could also impact fault
detection, either positively or negatively. We must examine this complex relationship
between AFFS and fault detection.

The third question will address the consequences of using reinforcement learning
during the test generation process. This question will focus on the computational
overhead of reinforcement learning. Test generation uses a time budget. Additional
overhead from reinforcement learning may impact the number of generations of evolu-
tion the population of test suites goes through during that time—potentially negating
the benefits of using reinforcement learning in the first place. At the same time, it
is also expensive to calculate certain fitness functions or large sets of functions, and
reinforcement learning may be able to avoid such functions. Therefore, we must
examine the relationship between reinforcement learning and the cost of computing
each generation of evolution.

To further understand adaptive fitness function selection, we will also examine
trends in the choices of fitness functions made by each AFFS approach for each of
our goals. We will also identify and discuss limitations of the current implementation.
The answers to these questions will help direct future research on this approach.

In order to investigate these questions, we have performed the following experi-
ment for each of the three goals:

1. **Collected Case Examples:*** We have used a collection of real faults, from the
Defects4J fault dataset, as test generation targets (Section 5.1).
2. Generated Test Suites: We target the classes affected by each fault for test generation. For each class, we generate 10 suites per approach. Approaches include the two reinforcement learning algorithms—UCB and DSG-Sarsa—and two baselines—an existing fitness function for that goal (current practice) and a combination of all fitness functions that AFFS can chose from (a “best guess”). A search budget of 10 minutes is used per suite (Section 5.2).

3. Removed Non-Compiling and Flaky Tests: Any tests that do not compile, or that return inconsistent results, are removed (Section 5.2).

4. Assessed Effectiveness: We measure goal attainment for each test suite, the number of faults detected by each approach, the likelihood of fault detection for each fault and approach, the number of generations of evolution that occur during the generation process, and other data that can be used to analyze the behavior of both AFFS and traditional test generation (Section 5.3).

We use the gathered data to analyze the performance of AFFS for each individual goal, as well as to analyze the general behavior of AFFS across all goals.

5.1 Case Examples

Defects4J is an extensible dataset of real faults extracted from Java projects [58]. For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose the fault, and a list of classes and lines of code modified by the patch that fixes the fault. To ease experiments, Defects4J provides test execution, generation, code coverage, and mutation analysis capabilities.

Each fault is required to meet three properties. First, a pair of code versions must exist that differ only by the minimum changes required to address the fault. The “fixed” version must be explicitly labeled as a fix to an issue, and changes imposed

\footnote{Available from \url{http://defects4j.org}}
by the fix must be to source code, not to other project artifacts such as the build system. Second, the fault must be reproducible—at least one test must pass on the fixed version and fail on the faulty version. Third, the fix must be isolated from unrelated code changes such as refactoring.

Our first goal, exception discovery, was assessed using Defects4J 1.4, which consists of 395 faults from six projects: Chart (26 faults), Closure (133 faults), Lang (65 faults), Math (106 faults), Mockito (38 faults), and Time (27 faults). Nine of the faults were excluded from our analysis—Closure faults 38, 44, 47, and 51, Math faults 13, 31, and 59, Mockito fault 6, and Time fault 21—as no technique caused exceptions to be thrown.

The other goals—suite diversity and Strong Mutation Coverage—were assessed later using the updated Defects4J 2.0. The experiments for exception discovery were not repeated due to experiment cost. However, as we already accounted for differences between Java 7 and 8—the primary semantic difference between Defects4J 1.4 and 2.0—results would not differ between versions of the dataset. To more easily compare results between the three high-level goals, we focus on the same projects from Defects4J.

In both the diversity and Strong Mutation Experiments, we use the following 434 faults: Chart (26 faults), Closure (174 faults), Lang (64 faults), Math (106 faults), Mockito (38 faults), and Time (26 faults). In addition, for the diversity goal, we also use the Gson project (18 faults), which was initially assessed in a separate experiment [5]—bringing the total case examples for the diversity experiment to 452.

5.2 Test Suite Generation

For all three goals, and for each class from each case example used from Defects4J, we have generated test suites using UCB and DSGSarsa. In addition, for each goal, we generate tests for two baseline approaches representing current practice:
• **Current Practice:** We use the existing fitness function representation of that goal. This would be the likely starting point for a tester interested in these goals, and thus, represent current practice. These are, as follows:

  – **Exception Count:** A common fitness function representation of the goal of throwing exceptions is to simply count the number of exceptions thrown by a test suite.

  – **Strong Mutation Coverage:** The existing fitness function for measuring Strong Mutation Coverage.

  – **Diversity Score (Levenshtein Distance):** A new fitness function based on the changes required to change one test case into another.

All three functions are explained in more detail in Chapter 3.

• **Combination of all Functions (“Default Approach”):** A combination of all of the individual fitness functions used in each experiment is used as a baseline as this combination attains reasonable fulfillment of each individual function, and in theory, will produce multifaceted test suites effective at fault-finding [73]. This configuration represents a “best guess” at what would produce effective test suites, and would be considered a reasonable approach in the absence of a known, informative fitness function or “best” combination.

Test suites are generated that target the classes reported as relevant to the fault by Defects4J. Tests are generated using the fixed version of the CUT and applied to the faulty version in order to eliminate the oracle problem. In practice, this translates to a regression testing scenario, where tests are generated using a version of the system understood to be “correct” in order to guard against future issues [78]. Tests that fail on the faulty version, then, detect behavioral differences between the two versions².

²Note that this is identical practice to other studies using EvoSuite with Defects4J, i.e. [78, 37]
To perform a fair comparison between approaches, each is allocated a ten minute search budget for test generation. In past work, 10 minutes was used as the maximum generation time and represented a point of “diminishing returns” for detection of the faults in Defects4J [37].

To control experiment cost, we deactivated assertion filtering—all possible regression assertions are included. All other settings were kept at their default values. As results may vary, we performed 10 trials for each fault and search budget. For the Exception experiment, this resulted in the generation of 15,800 test suites (ten trials, four approaches, 395 faults), representing over 2,633 hours of computation time. For the Diversity experiment, this resulted in the generation of 18,080 test suites (ten trials, four approaches, 452 faults), representing over 3,013 hours of computation time. Finally, in the Strong Mutation experiment, this resulted in the generation of 17,360 test suites (ten trials, four approaches, 434 faults), representing over 2893 hours of computation time. We performed experiments on Amazon EC2 infrastructure.

Generation tools may generate flaky (unstable) tests [78]. For example, a test case that makes assertions about the system time will only pass during generation. We automatically remove flaky tests. First, all non-compiling test suites are removed. Then, each remaining test suite is executed on the fixed version five times. If the test results are inconsistent, the test case is removed. This process is repeated until all tests pass five times in a row. On average, less than one percent of tests tends to be removed from each suite.

### 5.3 Data Collection

In order to address our research questions, we collect the following data for each test suite, based on the goal of the experiment:
Table 5.1: Data collected for each experiment

<table>
<thead>
<tr>
<th>Exception</th>
<th>Strong Mutation</th>
<th>Diversity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number Exceptions Discovered</td>
<td>Number of Mutants</td>
<td>Diversity Score</td>
</tr>
<tr>
<td>Number Exceptions Thrown</td>
<td>Strong Mutation Coverage</td>
<td></td>
</tr>
</tbody>
</table>

- Number of Faults Detected
- Number of Generations of Evolution
- Decisions Made by EvoSuiteFIT

- **Exception Experiment:**
  - Number of Unique Exceptions Discovered During Generation
  - Number of Unique Exceptions Thrown by the Final Test Suite:
    Tests that trigger an exception can be lost during the generation process.
    We calculate this number by monitoring test suite execution.

- **Strong Mutation Experiment:**
  - Number of Mutants: The number of mutants inserted into the CUT.
  - Strong Mutation Coverage: Percentage of mutants detected, meeting the conditions of Strong Mutation.

- **Diversity Experiment:**
  - Diversity Score: The diversity score (based on the Levenshtein Distance) for the final test suite.

- **All Experiments:**
  - Number of Faults Detected
  - Number of Generations of Evolution: The amount of time that it takes to complete one generation of evolution is not static, and each approach may complete a different number of generations during the test generation process based on the time needed to calculate each employed fitness function. Reinforcement learning will add additional overhead to this process, further decreasing the number of completed generations. We
collect the number of generations to assess the impact of fitness function choice and RL overhead.

- **Decisions Made by EvoSuiteFIT**: The reinforcement learning algorithms reformulate the fitness function combination in use at regular intervals. Each time a combination is selected, we log the decision made. This can assist in understanding how the reinforcement learning algorithms function, and how they make decisions in service of goal attainment.
Chapter 6
Results and Discussion

We are interested in understanding the effectiveness of EvoSuiteFIT in terms of attainment of our high-level goals—exception discovery, test suite diversity, and Strong Mutation Coverage—and in terms of detection of faults. We are also interested in the impact of the overhead of reinforcement learning on the generation process, how the approaches make their fitness function selections, and the limitations of adaptive fitness function selection. The following sections outline and discuss our observations.

6.1 Goal: Exception Discovery

6.1.1 Ability to Discover and Retain Exceptions

Our first question asks whether AFFS can be used to more effectively meet our goal of forcing the CUT to throw more exceptions than baseline static fitness function configurations. We assess this along two dimensions. First, we look at the number of unique exceptions discovered by each approach during test generation. Second, we look at the number of unique exceptions thrown by the final test suite produced by each method. Are exception-inducing tests cases both produced and retained by the generation framework? An effective approach must excel at both.

We do not know a priori which exceptions can be thrown by a CUT. However, we know that the number of possible exceptions varies from class to class. Therefore, it is not fair to compare raw counts of exceptions between each case example. If we discover thirty exceptions when testing one class, and five when testing another, we
Table 6.1: Median count of exceptions thrown and exceptions discovered for each technique, along with the median ratio of thrown to discovered. Counts are normalized between 0-1 for each fault to allow comparison across case examples. Higher scores are better.

<table>
<thead>
<tr>
<th>System</th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Exception Count</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Thrown</td>
<td>Discovered</td>
<td>Thrown</td>
<td>Discovered</td>
</tr>
<tr>
<td>Chart</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Closure</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Lang</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Math</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Mockito</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Time</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Overall</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Figure 6.1: Unique exceptions discovered and thrown by each technique. Counts are normalized between 0-1 for each fault to allow comparison across case examples.

The median count of exceptions thrown and discovered for each technique is listed in Table 6.1 for each project and overall. Boxplots of the exceptions discovered and thrown by each technique are shown in Figure 6.1. Higher scores are better.

The results show that both AFFS techniques have a higher median performance in both measurements than the two baselines. This is true across all systems, with EvoSuiteFIT attaining up to a 203.03% improvement in median exceptions discovered or thrown over the basic exception count and up to a 49.25% improvement over the
eight-function default configuration. Overall, EvoSuiteFIT attains a 100% improvement in median exception discovery over the simple exception count and 33% over the default combination. EvoSuiteFIT also tends to retain all discovered exceptions, while the default configuration may discard a small number of exception-triggering tests if offered improvements in the other fitness functions.

Figure 6.1 further shows the ability of AFFS to trigger unique exceptions. Both techniques not only offer a higher median than the competing techniques, but also have a narrower interquartile spread, showing relatively consistent performance. DSG-Sarsa yields more consistent performance, as shown by the decreased variance. Both techniques demonstrate superior ability to discover exception-triggering input over traditional baselines. Additionally, due to the use of a test archive, the two techniques do a better job of retaining exceptions.

We perform statistical analysis to assess our observations. For each pair of techniques and baselines, we formulate hypotheses and null hypotheses:

- $H_1$: Test suites generated using technique $A$ will have a different distribution of exception discovery results than suites generated using technique $B$.
- $H_2$: Test suites generated using technique $A$ will have a different distribution of exception retention results than suites generated using technique $B$.
- $H_{01}$: Observations of exception discovery for both techniques are drawn from the same distribution.
- $H_{02}$: Observations of exception retention for both techniques are drawn from the same distribution.

Our observations are drawn from an unknown distribution; To evaluate the null hypotheses without any assumptions on distribution, we use a one-sided (strictly greater) Mann-Whitney-Wilcoxon rank-sum test [87], a non-parametric test for de-
Table 6.2: P-Values for Mann-Whitney rank-sum test for exceptions thrown and discovered. Results are identical for both.

<table>
<thead>
<tr>
<th></th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Exception</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSG-Sarsa</td>
<td>-</td>
<td>&lt; 0.01</td>
<td>&lt; 0.01</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>UCB</td>
<td>0.99</td>
<td>-</td>
<td>&lt; 0.01</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>Exception</td>
<td>1.00</td>
<td>1.00</td>
<td>-</td>
<td>1.00</td>
</tr>
<tr>
<td>Default</td>
<td>1.00</td>
<td>1.00</td>
<td>&lt; 0.01</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.3: Results of Vargha-Delaney A Measure for exceptions thrown and discovered. Large positive effect sizes are bolded.

<table>
<thead>
<tr>
<th></th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Exception</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSG-Sarsa</td>
<td>-</td>
<td>0.53, 0.53</td>
<td>0.90, 0.90</td>
<td>0.90, 0.80</td>
</tr>
<tr>
<td>UCB</td>
<td>0.47, 0.47</td>
<td>-</td>
<td>0.89, 0.89</td>
<td>0.80, 0.77</td>
</tr>
<tr>
<td>Exception</td>
<td>0.10, 0.10</td>
<td>0.11, 0.11</td>
<td>-</td>
<td>0.31, 0.31</td>
</tr>
<tr>
<td>Default</td>
<td>0.19, 0.21</td>
<td>0.21, 0.32</td>
<td>0.69, 0.69</td>
<td>-</td>
</tr>
</tbody>
</table>

termining if one set of observations is drawn from a different distribution that another set. We apply the test for each pairing of techniques and baselines with $\alpha = 0.05$.

The resulting p-values are listed in Table 6.2. P-values are the same for both exceptions discovered and thrown. For DSG-Sarsa, we can reject both null hypotheses for UCB and two baselines. For UCB, we can reject the null hypotheses for the two baselines. For the default baseline, we can reject the null hypotheses for the exception count, but not for either EvoSuiteFIT technique.

We also used the Vargha-Delaney A measure to assess effect size [84]. The results for exception discovery and retention are listed in Table 6.3, with large effect sizes in bold ($\geq 0.80$). DSG-Sarsa outperforms the two baselines with a large effect size. UCB outperforms the exception count baseline with a large effect size in both exception discovery and retention, and outperforms the default baseline with a large effect size for retention and a medium effect size in discovery. DSG-Sarsa outperforms UCB, but with a negligible effect size.
Table 6.4: Percentage of faults detected by each approach.

<table>
<thead>
<tr>
<th>System</th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Exception Count</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>81%</td>
<td>85%</td>
<td>38%</td>
<td>65%</td>
</tr>
<tr>
<td>Closure</td>
<td>7%</td>
<td>6%</td>
<td>4%</td>
<td>15%</td>
</tr>
<tr>
<td>Lang</td>
<td>58%</td>
<td>65%</td>
<td>17%</td>
<td>52%</td>
</tr>
<tr>
<td>Math</td>
<td>69%</td>
<td>69%</td>
<td>12%</td>
<td>58%</td>
</tr>
<tr>
<td>Mockito</td>
<td>13%</td>
<td>16%</td>
<td>8%</td>
<td>13%</td>
</tr>
<tr>
<td>Time</td>
<td>59%</td>
<td>67%</td>
<td>19%</td>
<td>52%</td>
</tr>
<tr>
<td>Overall</td>
<td>41%</td>
<td>43%</td>
<td>12%</td>
<td>38%</td>
</tr>
</tbody>
</table>

Both EvoSuiteFIT techniques discover and retain more exception-triggering input than the baseline techniques, with DSG-Sarsa yielding more consistent results. Overall, EvoSuiteFIT attains a 100% improvement in median exception discovery over the simple exception count and 33% over the default combination.

6.1.2 Fault Detection Effectiveness

In theory, forcing the class-under-test to throw exceptions will help developers discover faults in the system. Therefore, our second research question revolves around the ability of the generated test suites to trigger and detect failures. Both DGS-Sarsa and UCB trigger more exceptions than the baseline fitness functions. Does this translate into greater fault detection?

Table 6.4 lists the number of faults detected by each technique. We can see that both EvoSuiteFIT techniques generate suites that are able to detect faults that are missed by suites generated using the baselines. UCB, in particular, detects the most faults—identifying seven more faults than DSG-Sarsa (4.32% improvement), 18 more than default (11.92% improvement), and 122 more than the exception count (259.57% improvement).

The default combination outperforms EvoSuiteFIT for one project—Closure—discovering 11 more faults than DSG-Sarsa. In such cases, it is likely that triggering
Table 6.5: Median time per generation (in seconds) for the goal of exception discovery. EX+(1,2,3) = exception count + 1-3 fitness functions. The comparison data [37] lacked EX+1 and EX+3 data for Mockito. The fastest approach is bolded.

<table>
<thead>
<tr>
<th></th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>EX+1</th>
<th>EX+2</th>
<th>EX+3</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>0.24</td>
<td>0.26</td>
<td>0.36</td>
<td>0.49</td>
<td>0.75</td>
<td>3.29</td>
</tr>
<tr>
<td>Closure</td>
<td>0.32</td>
<td>0.49</td>
<td>1.40</td>
<td>1.61</td>
<td>2.70</td>
<td>5.71</td>
</tr>
<tr>
<td>Lang</td>
<td>0.30</td>
<td>0.44</td>
<td>0.23</td>
<td>0.34</td>
<td>1.03</td>
<td>4.38</td>
</tr>
<tr>
<td>Math</td>
<td>0.14</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.42</td>
<td>3.03</td>
</tr>
<tr>
<td>Mockito</td>
<td>0.03</td>
<td>0.03</td>
<td>-</td>
<td>0.03</td>
<td>-</td>
<td>0.08</td>
</tr>
<tr>
<td>Time</td>
<td>0.33</td>
<td>0.43</td>
<td>0.32</td>
<td>0.50</td>
<td>0.97</td>
<td>3.72</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td><strong>0.22</strong></td>
<td>0.31</td>
<td>0.72</td>
<td>0.64</td>
<td>1.23</td>
<td>3.84</td>
</tr>
</tbody>
</table>

the fault requires producing incorrect output, rather than triggering an exception. The default configuration is outperformed across all other systems.

We previously found that DSG-Sarsa yielded more consistent performance. However, UCB detected more faults. The difference between the two may come down to how fitness functions are chosen. The reinforcement learning strategy, by impacting how and which fitness functions are selected, will impact how input is selected. Differences in how UCB and DSG-Sarsa make selections will influence the resulting likelihood of fault detection. Further analysis is required to understand the full impact that reinforcement learning strategy can have on fault detection capability. Still, the broad hypothesis that triggering exceptions can aid fault discovery appears to have some merit.

For the exception discovery goal, both EvoSuiteFIT techniques detect faults missed by the other techniques. UCB detects 11.92 and 259.57% more faults than the baseline techniques, and 4.3% more than DSG-Sarsa.

6.1.3 Impact of Reinforcement Learning Overhead

Search-based test generation approaches are generally benchmarked using a fixed time budget [78]. During this period, the amount of work completed by each algorithm
may not be equal. The number of generations of evolution will largely depend on total cost to calculate fitness. The addition of reinforcement learning will further impact this cost through the addition of reward score calculation and action selection mechanisms. We are interested in understanding whether the cost of reinforcement learning has more of an effect than the cost of fitness calculation, and the further impact of being able to change the set of fitness functions on the computational cost.

Table 6.5 lists the median time per generation for DSG-Sarsa, UCB, and the default combination. An issue in the version of EvoSuite deployed prevented us from collecting accurate generation times for the exception count alone, but—as it is an extremely simple count that does not require sophisticated instrumentation—it can be assumed that its computation is far less expensive than any other option. Using data from past experiments [37, 36], we can also compare the time per generation with a sample of 33,759 test suites generated using combinations of the exception count and 1-3 additional fitness functions—options that can be chosen by AFFS.

From Table 6.5, we can see that the median time per generation tends to increase as additional fitness functions are added to the calculation, with the time for the default combination often being many times higher than either EvoSuiteFIT approach. While reinforcement learning may add to the cost of generation, its overhead is less than that required to compute a larger number of fitness functions.

Most of the potential users of a test generation framework would, rightfully, not be interested in tinkering with fitness functions until they stumbled on the right approach. In the absence of perfect knowledge, using the “default”—all eight fitness functions—is a reasonable idea. It is also an expensive option. Reinforcement learning can reduce the time required to generate effective test cases.

Both AFFS approaches are similar in speed to the static EX+1 combinations, if not faster. Figure 6.2 helps explain why. In Figure 6.2, we show the ten actions chosen most often by DSG-Sarsa for each system. While the EvoSuiteFIT techniques
can choose combinations of up to four fitness functions, it rarely does so in practice. Often, EX+1 combinations are chosen, or even the simple exception count. Because EvoSuiteFIT can strategically change its fitness function selection, any overhead added by reinforcement learning is mitigated by the gain in speed from the ability to avoid calculating unhelpful fitness functions.

The ability to avoid calculation of unhelpful fitness functions mitigates reinforcement learning overhead. Both AFFS approaches are able to complete more generations of evolution during the search budget than the default combination, with DSG-Sarsa being the most efficient approach.

6.1.4 Actions Selected by AFFS

For the exception discovery goal, EvoSuiteFIT is able to freely alternate between 64 combinations of fitness functions. To help understand why reinforcement learning is effective, we should examine the actions chosen by AFFS techniques. In Figure 6.2, we display the ten actions chosen most often by DSG-Sarsa for each system, and in Figure 6.3, we do the same for UCB.

From Figure 6.2-6.3, we can see differences between projects in terms of which choices are made and how often choices are made. For example, DSG-Sarsa frequently used a combination of exception count, Direct Branch Coverage, Weak Mutation Coverage, and Output Coverage for the Time system, but not for others. Although the ordering differs, however, there are also a lot of commonalities in the choices.

For the most part, the combinations favored by DSG-Sarsa are simple—pairing the exception count with, at most, one additional fitness function. It is reasonable that simple combinations would be used frequently. Larger combinations introduce a risk of conflicting goals, and are harder to maximize. Simple combinations offer enough feedback to increase the exception count, without adding noise to the search.
Figure 6.2: Top ten function combinations chosen by DSG-Sarsa for each system for the exception discovery goal. E = Exception Count, B = Branch Coverage, CB = Direct Branch Coverage, L = Line Coverage, O = Output Coverage, M = Method Coverage, MNE = Method (No Exception), WM = Weak Mutation Coverage
Figure 6.3: Top ten function combinations chosen by UCB for each system for the exception discovery goal. E = Exception Count, B = Branch Coverage, CB = Direct Branch Coverage, L = Line Coverage, O = Output Coverage, M = Method Coverage, MNE = Method (No Exception), WM = Weak Mutation Coverage.
UCB chooses complex sets of actions, combinations of 3-4 fitness functions, somewhat more often than DSG-Sarsa. However, it does not necessarily do so significantly more often than it chooses simple combinations. The most noticeable factor about UCB, as seen in Figure 6.3, is that it heavily favors the simple exception count—applying it far more often than it does any other action.

Many of the fitness function combinations chosen heavily by DSG-Sarsa or UCB would yield poor results when used on their own, as static fitness functions for suite generation. Both often uses the pure exception count, when this yields poor results when used as the sole fitness function. Similarly, we know from past unpublished experiments that the EX-MNE combination produces poor results when used as a static choice, yet DSG-Sarsa applies it heavily.

However, it is important to remember that test generation is a stateful process. Each round of the generation process builds on the results of previous rounds. There are times where the choices that DSG-Sarsa makes are relevant given the state of generation, even if those choices yield poor results when used in a static context. For example, if a suite already has achieved a high level of code coverage, it would make sense to switch to pure use of the exception count to further tune the population of test suites. Similarly, the exception and MNE combination makes sense as a strategic choice because it adds a light feedback mechanism to the exception count. Method (No Exception) Coverage requires that methods execute without exception. This does not mean that the test itself cannot throw an exception. Rather, it means that test generation will be encouraged to execute some code before the exception is thrown. This combination may be ineffective in a static context, as it does not offer enough feedback to fully explore the code space. However, it can be very effective if chosen at the right stage of the generation process, as part of an adaptive process.

AFFS may use combinations early on that—for example—rapidly advance coverage of the source code. Combinations involving Branch Coverage could be used for
early gain, then a lightweight combination of exception count and MNE could further sculpt the test suite in a way that allows discovery of additional exceptions. Combinations like the exception count and Output Coverage would potentially be very useful in this same situation to diversify input selections after the suite has already evolved to achieve high code coverage.

The ability to adjust the fitness functions at regular intervals allows EvoSuiteFIT to make strategic choices that refine the test suite. Fitness function combinations that are ineffective in a static context may be effective when used by AFFS to diversify a pre-evolved population of suites.

6.2 Goal: Test Suite Diversity

6.2.1 Ability to Improve Suite Diversity

Again, our first question concerns the ability of AFFS to meet our goal of diverse test suites. We assess this by examining the diversity fitness score of the produced test suite. In this case, scores range between 0-1, and lower scores indicate higher levels of diversity. In Table 6.6, we indicate the median diversity score for each technique for each project and overall. In Figure 6.4, we show boxplots for each technique.

Table 6.6: Median diversity fitness score of the produced test suite. Score is between 0-1, with lower scores being better. Best approach is bolded.

<table>
<thead>
<tr>
<th>Project</th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Default</th>
<th>Diversity Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>7.45E-07</td>
<td>4.20E-07</td>
<td>1.11E-06</td>
<td>4.40E-06</td>
</tr>
<tr>
<td>Closure</td>
<td><strong>1.49E-06</strong></td>
<td>1.59E-06</td>
<td>1.82E-06</td>
<td>5.64E-06</td>
</tr>
<tr>
<td>Gson</td>
<td>1.43E-06</td>
<td><strong>9.51E-07</strong></td>
<td>1.86E-06</td>
<td>3.46E-06</td>
</tr>
<tr>
<td>Lang</td>
<td>5.21E-07</td>
<td><strong>3.05E-07</strong></td>
<td>1.11E-06</td>
<td>3.85E-06</td>
</tr>
<tr>
<td>Math</td>
<td>1.32E-06</td>
<td><strong>1.06E-06</strong></td>
<td>1.54E-06</td>
<td>4.02E-06</td>
</tr>
<tr>
<td>Mockito</td>
<td><strong>2.32E-06</strong></td>
<td>2.35E-06</td>
<td>3.69E-06</td>
<td>5.20E-06</td>
</tr>
<tr>
<td>Time</td>
<td>6.58E-07</td>
<td><strong>4.39E-07</strong></td>
<td>9.74E-07</td>
<td>3.51E-06</td>
</tr>
<tr>
<td>Overall</td>
<td>1.32E-06</td>
<td><strong>1.02E-06</strong></td>
<td>1.73E-06</td>
<td>4.30E-06</td>
</tr>
</tbody>
</table>
From Table 6.6 and Figure 6.4, we see that both AFFS techniques outperform the default combination and the diversity score alone. Diversity-alone serves as a poor fitness target, confirming our initial concerns. This fitness function—while representing a valid high-level goal—offers insufficient feedback to achieve that goal. This can be seen in both the worse median score, but also the wide variance in Figure 6.4.

The default combination attains better results than diversity-alone. However, both AFFS techniques outperform the default combination. Overall, the best technique, UCB, attains 76.27% better median performance than diversity alone, 41.04% better than the default combination, and 22.72% better than DSG-Sarsa.

Interestingly, from Figure 6.4, we see that DSG-Sarsa shows less variance in its results than UCB. UCB attains better results, but DSG-Sarsa is more consistent. While UCB yields better median results, the state approximation performed by DSG-Sarsa may result in less variance in performance.

We again perform statistical analysis to assess our observations. For each pair of AFFS technique and baseline, we formulate hypothesis and null hypothesis:

![Figure 6.4: Diversity fitness scores of the produced test suites. Score is between 0-1, with lower scores being better.](image)
• \textbf{H}: Test suites generated using technique \textit{A} will have a different distribution of diversity score results than suites generated using technique \textit{B}.

• \textbf{H0}: Observations of diversity score for both techniques are drawn from the same distribution.

We again use both the one-sided (strictly greater) Mann-Whitney-Wilcoxon rank-sum test, with $\alpha = 0.05$, and Vargha-Delaney A measure. The resulting p-values for the Mann-Whitney-Wilcoxon test are listed in Table 6.7. For DSG-Sarsa, we reject the null hypotheses for the two baselines. For UCB, we reject the null hypotheses for the two baselines as well as for DSG-Sarsa. For the default baseline, we can reject the null hypotheses for the diversity-only baseline, but not for the AFFS techniques.

The results for the Vargha-Delaney A measure are listed in Table 6.8, with large effect sizes in bold. The results of this test further confirm our observations. DSG-Sarsa outperforms the diversity-only baseline with a large effect size and the default combination with a medium effective size. UCB outperforms the diversity-only baseline with a large effect size, the default combination with medium effective size, and DSG-Sarsa but with a small effect size. The default combination outperforms diversity-only with medium effective size.

<table>
<thead>
<tr>
<th>Test Techniques</th>
<th>Median Performance Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCB</td>
<td>76.27%</td>
</tr>
<tr>
<td>DSG-Sarsa</td>
<td>41.04%</td>
</tr>
<tr>
<td>Default Baseline</td>
<td>22.72%</td>
</tr>
</tbody>
</table>

Both EvoSuiteFIT techniques produce more diverse test suites than static fitness function choices, with UCB outperforming DSG-Sarsa. Overall, UCB attains 76.27% better median performance than diversity alone, 41.04% better than the default combination, and 22.72% better than DSG-Sarsa.

### 6.2.2 Fault Detection Effectiveness

Proponents of test suite diversity have noted a positive relationship between diversity and the likelihood of fault detection. Logically, test suites that apply a larger variety of test cases are more likely to detect faults. This is because diversity helps to cover a wider range of scenarios and edge cases, increasing the chances of uncovering unseen bugs.

80
Table 6.7: P-Values for Mann-Whitney rank-sum test for diversity score.

<table>
<thead>
<tr>
<th></th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Default</th>
<th>Diversity Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSG-Sarsa</td>
<td>-</td>
<td>1.00</td>
<td>&lt; 0.01</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>UCB</td>
<td>&lt; 0.01</td>
<td>-</td>
<td>&lt; 0.01</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>Default</td>
<td>1.00</td>
<td>1.00</td>
<td>-</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>Diversity Score</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.8: Results of Vargha-Delaney A Measure for diversity score. Large positive effect sizes are **bolded**.

<table>
<thead>
<tr>
<th></th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Default</th>
<th>Diversity Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSG-Sarsa</td>
<td>-</td>
<td>0.47</td>
<td>0.63</td>
<td><strong>0.86</strong></td>
</tr>
<tr>
<td>UCB</td>
<td>0.53</td>
<td>-</td>
<td>0.66</td>
<td><strong>0.86</strong></td>
</tr>
<tr>
<td>Default</td>
<td>0.37</td>
<td>0.34</td>
<td>-</td>
<td>0.77</td>
</tr>
<tr>
<td>Diversity Score</td>
<td>0.14</td>
<td>0.14</td>
<td><strong>0.32</strong></td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.9: Percentage of faults detected by each approach.

<table>
<thead>
<tr>
<th></th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Default</th>
<th>Diversity Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>35%</td>
<td>42%</td>
<td>35%</td>
<td><strong>27%</strong></td>
</tr>
<tr>
<td>Closure</td>
<td><strong>11%</strong></td>
<td>9%</td>
<td>7%</td>
<td>6%</td>
</tr>
<tr>
<td>Gson</td>
<td>22%</td>
<td>17%</td>
<td>17%</td>
<td>11%</td>
</tr>
<tr>
<td>Lang</td>
<td>27%</td>
<td>22%</td>
<td>25%</td>
<td>19%</td>
</tr>
<tr>
<td>Math</td>
<td>31%</td>
<td>30%</td>
<td>30%</td>
<td>22%</td>
</tr>
<tr>
<td>Mockito</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>Time</td>
<td><strong>31%</strong></td>
<td>31%</td>
<td><strong>31%</strong></td>
<td>23%</td>
</tr>
<tr>
<td>Overall</td>
<td><strong>20%</strong></td>
<td>19%</td>
<td>18%</td>
<td>14%</td>
</tr>
</tbody>
</table>

of stimuli to the CUT will be more likely to detect faults just by virtue of not performing the same actions over and over again. We have observed that AFFS does increase suite diversity. Therefore, we also are curious about whether it increases the potential for fault detection as well. Table 6.9 lists the number of faults detected by each approach. Overall, the AFFS approaches detect more faults. DSG-Sarsa detects the most faults 92, it detected 7 more than UCB (8.24% improvement), 9 more than the default combination (10.84% improvement), and 30 more than optimizing for diversity alone (48.39% improvement).

Like with the exception count, targeting diversity alone results in suites that are lacking in diversity—due to limitations in the fitness function—and weak at detecting
Table 6.10: Median time per generation (in seconds) for the goal of test suite diversity. The fastest approach is **bolded**.

<table>
<thead>
<tr>
<th></th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Default</th>
<th>Diversity Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>5.26</td>
<td>3.39</td>
<td>8.38</td>
<td>6.93</td>
</tr>
<tr>
<td>Closure</td>
<td>8.63</td>
<td><strong>6.44</strong></td>
<td>12.58</td>
<td>12.37</td>
</tr>
<tr>
<td>Gson</td>
<td>4.09</td>
<td><strong>2.95</strong></td>
<td>4.70</td>
<td>4.64</td>
</tr>
<tr>
<td>Lang</td>
<td>3.65</td>
<td><strong>2.63</strong></td>
<td>7.09</td>
<td>4.60</td>
</tr>
<tr>
<td>Math</td>
<td>4.09</td>
<td><strong>3.05</strong></td>
<td>5.68</td>
<td>4.15</td>
</tr>
<tr>
<td>Mockito</td>
<td>5.98</td>
<td><strong>4.16</strong></td>
<td>6.22</td>
<td>5.96</td>
</tr>
<tr>
<td>Time</td>
<td>3.63</td>
<td><strong>2.77</strong></td>
<td>6.32</td>
<td>5.29</td>
</tr>
<tr>
<td>Overall</td>
<td>4.09</td>
<td><strong>3.05</strong></td>
<td>6.32</td>
<td>5.29</td>
</tr>
</tbody>
</table>

faults. The default combination is far more effective than targeting diversity alone. This is unlikely to be solely due to the improved diversity, but is also a natural product of generating test suites targeting multiple fitness functions [36]. However, both EvoSuiteFIT approaches detect additional faults missed by the default combination. To some extent, this may be due to the further improvement in diversity from AFFS.

UCB attained higher diversity than DSG-Sarsa. However, DSG-Sarsa detects more faults. The exact reason for this is not clear, but comes down to a combination of the stochastic nature of search-based generation and differences in the decision making processes for the two algorithms. Again, a goal in future work is a deeper examination of the impact of reinforcement learning approach on fault detection.

6.2.3 Impact of Reinforcement Learning Overhead

In Table 6.10, we display the median time per generation for each approach. This, again, allows us to compare the overhead introduced by reinforcement learning to the cost of calculating fitness. Immediately, we see that AFFS is faster than the
default combination. While reinforcement learning introduces overhead—including the calculation of diversity as part of the reward score—*this cost is less than naively calculating a large number of unnecessary fitness functions.*

A surprising result, however, was that *AFFS is faster than targeting diversity alone.* Intuitively, calculating multiple fitness functions should be more computationally expensive than calculating one fitness function. In examining this result, we made a valuable observation about the Levenshtein distance fitness function.

The cost of computing the Levenshtein distance is based on the quantity of text being compared. If a test suite is larger—containing a greater number of tests, longer tests with more interactions with the CUT, or both—then the fitness computation will be more expensive. In inspecting changes in the size of test suites throughout the generation process, we found that the test suites evolved when optimizing diversity alone were significantly larger than those being evolved by DSG-Sarsa, with the latter being 41% smaller on average for the studied examples.

In the absence of feedback from additional fitness functions, the genetic algorithm optimizing the diversity fitness function allowed unconstrained growth in its test suites. Creating longer tests is one *potential* path to improving diversity, but not a guaranteed one—it could still result in similar test cases. Ultimately, the diversity fitness function was not only limited in its ability to suggest means of improving fitness, but actually detrimental to goal attainment by limiting the number of generations that could be completed during the search budget. AFFS was able to both improve diversity and control the growth of test suites, in turn controlling the cost of fitness calculation as well.

Both AFFS approaches are able to complete more generations of evolution during the search budget than the default combination and the diversity fitness function. By incorporating feedback from additional fitness functions, AFFS is
not only more effective at achieving diversity, but also controls the cost of the
diversity fitness calculation by preventing uncontrolled test suite growth.

6.2.4 Actions Selected by AFFS

In Figure 6.5, we display the ten fitness function combinations chosen most often
by DSG-Sarsa for the diversity goal, and in Figure 6.6, we do the same for UCB.
Examining these choices may offer insight into the results attained by AFFS.

The first observation we make is that the combination of diversity score, exception
count, and Branch Coverage is chosen the most often. It is the top choice made
by DSG-Sarsa for four of the seven systems, and the top choice for UCB for all
projects. This specific combination provides several key ingredients for attaining
diversity. Branch Coverage encourages exploring the structure of the CUT, building
strong test suites that the other functions can tune. The exception count imbues the
suite with a wider spread of input choices, with specific test cases triggering unique
exceptions. Finally, the diversity score encourages further input diversification.

Each function is insufficient on its own. The diversity score needs external feed-
back to drive diversity. Branch Coverage and the exception count both offer this.
Branch Coverage alone will only result in as much diversity as is required to cover
more of the code. The other functions force diversification of the input choices. The
exception count could be a great driver of diversity, but needs Branch Coverage to aid
code exploration. Together, these three functions offer each other feedback, resulting
in more diversity than could be attained individually.

Unlike the exception discovery goal, both UCB and DSG-Sarsa favor complex
combinations of three-four fitness functions. For the goal of diversity, this makes
some sense. We seek test suites that try a lot of different things. Even if poor
coverage is attained of some of the fitness functions in a combination, and even if
Figure 6.5: Top ten fitness function combinations chosen by **DSG-Sarsa** for the diversity goal. D = Diversity Score, B = Branch Coverage, CB = Direct Branch Coverage, O = Output Coverage, M = Method Coverage, MNE = Method (No Exception), WM = Weak Mutation Coverage
Figure 6.6: Top ten fitness function combinations chosen by UCB for the diversity goal. D = Diversity Score, B = Branch Coverage, CB = Direct Branch Coverage, O = Output Coverage, M = Method Coverage, MNE = Method (No Exception), WM = Weak Mutation Coverage, E = Exception
conflicts exist, more functions could drive the generation process towards attempting to satisfy a huge variety of goals.

The combination of Branch Coverage, exception count, and diversity score seems effective at improving test suite diversity. These functions act in concert, enabling greater diversity together than they would alone. Other complex function combinations similarly collaborate to improve suite diversity.

We again see that UCB tends to exploit one combination above all others, while DSG-Sarsa will spend more time exploring different options. As UCB attains somewhat better results, it may be that heavier exploitation is a good idea for this goal.

UCB has a greater tendency for exploitation than DSG-Sarsa for both the exception discovery and suite diversity goals. This may enable better goal attainment, as less time is spent trying potentially weak function combinations.

Like we saw with the exception discovery goal, certain selections that would not work well in a static context may be useful to refine pre-evolved suites. We see this with DSG-Sarsa and the diversity score. Optimizing the diversity score in a static context yields poor results, but is used quite often by DSG-Sarsa to refine test suites that have been shaped by other function combinations. This allows diversification of test suites that have already been built up to do things like explore the code base.

6.3 Goal: Strong Mutation Coverage

6.3.1 Ability to Improve Coverage and Impact of Overhead

We assess the attainment of our third goal using the percent of mutants detected through an observable difference in CUT output. In Table 6.11, we note the median
Table 6.11: Percentage of Strong Mutation Coverage attained when all approaches execute for 10 minutes. Higher values are better. Best approach is bolded.

<table>
<thead>
<tr>
<th></th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Default</th>
<th>Strong Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>47.00</td>
<td>47.00</td>
<td>52.00</td>
<td>54.00</td>
</tr>
<tr>
<td>Closure</td>
<td>16.00</td>
<td>16.00</td>
<td>18.00</td>
<td>19.00</td>
</tr>
<tr>
<td>Lang</td>
<td>61.00</td>
<td>60.00</td>
<td>62.00</td>
<td>63.00</td>
</tr>
<tr>
<td>Math</td>
<td>73.00</td>
<td>74.00</td>
<td>73.00</td>
<td>73.00</td>
</tr>
<tr>
<td>Mockito</td>
<td>12.00</td>
<td>8.50</td>
<td>11.00</td>
<td>10.00</td>
</tr>
<tr>
<td>Time</td>
<td>67.00</td>
<td>66.00</td>
<td>67.00</td>
<td>68.00</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td>38.00</td>
<td>39.00</td>
<td><strong>40.00</strong></td>
<td>40.00</td>
</tr>
</tbody>
</table>

Figure 6.7: Strong Mutation Coverage attained by final test suites when (left) all approaches run for 10 minutes, and (right), when all approaches are fixed to the number of generations of evolution completed by DSG-Sarsa in 10 minutes.

Strong Mutation Coverage for each AFFS technique and static configuration for each system and overall, bolding the approach with the best results. In the left diagram in Figure 6.7, we show a boxplot of the Strong Mutation results for all approaches.

From Table 6.11, we see a notable contrast to our other two goals. For four of the six projects, both AFFS approaches and the default configuration attain worse results than simply optimizing the Strong Mutation Coverage directly. From Figure 6.7, we can see that all four approaches yield very similar boxplots. However, Strong Mutation alone and the default configuration both yield higher median performance. Overall, optimizing Strong Mutation alone or targeting the default configuration yields a median improvement of 2.56% over UCB and 5.26% over DSG-Sarsa.
We again perform statistical analysis to assess our observations, using the Mann-Whitney-Wilcoxon rank-sum test and Vargha Delaney A measure. For each pair of techniques and baselines, we formulate hypothesis and null hypothesis:

- \( H \): Test suites generated using technique \( A \) will have a different distribution of Strong Mutation Coverage results than suites generated using technique \( B \).
- \( H_0 \): Observations of Strong Mutation Coverage for the two considered techniques are drawn from the same distribution.

In Table 6.12, we display the p-values for the Mann-Whitney-Wilcoxon tests. No test yielded a p-value below the \( \alpha = 0.05 \) threshold. We fail to reject the null hypothesis for all pairs of techniques. In Table 6.13, we display effect sizes. The default and Strong Mutation baselines slightly outperform UCB and DSG-Sarsa, but with a negligible effect size. While both baselines yield a slightly higher median performance, neither outperform AFFS with significance.

<table>
<thead>
<tr>
<th></th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Default</th>
<th>Strong Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSG-Sarsa</td>
<td>-</td>
<td>0.60</td>
<td>0.84</td>
<td>0.94</td>
</tr>
<tr>
<td>UCB</td>
<td>0.40</td>
<td>-</td>
<td>0.78</td>
<td>0.90</td>
</tr>
<tr>
<td>Default</td>
<td>0.16</td>
<td>0.22</td>
<td>-</td>
<td>0.70</td>
</tr>
<tr>
<td>Strong Mutation</td>
<td>0.06</td>
<td>0.10</td>
<td>0.30</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.13: Results of Vargha-Delaney A Measure for Strong Mutation (all approaches run for 10 minutes). There is no large effect sizes were observed.

<table>
<thead>
<tr>
<th></th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Default</th>
<th>Strong Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSG-Sarsa</td>
<td>-</td>
<td>0.50</td>
<td>0.49</td>
<td>0.49</td>
</tr>
<tr>
<td>UCB</td>
<td>0.50</td>
<td>-</td>
<td>0.49</td>
<td>0.49</td>
</tr>
<tr>
<td>Default</td>
<td>0.51</td>
<td>0.51</td>
<td>-</td>
<td>0.50</td>
</tr>
<tr>
<td>Strong Mutation</td>
<td>0.51</td>
<td>0.51</td>
<td>0.50</td>
<td>-</td>
</tr>
</tbody>
</table>
Table 6.14: Median time per generation (in seconds) for each technique for the Strong Mutation Coverage goal. Time is attained by dividing the number of generations completed by the total generation time (10 minutes). The fastest technique is bolded.

<table>
<thead>
<tr>
<th></th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Default</th>
<th>Strong Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>6.57</td>
<td>7.90</td>
<td>7.11</td>
<td>5.86</td>
</tr>
<tr>
<td>Closure</td>
<td>2.44</td>
<td>2.92</td>
<td>4.63</td>
<td>3.98</td>
</tr>
<tr>
<td>Lang</td>
<td>24.69</td>
<td>17.88</td>
<td>12.65</td>
<td>12.36</td>
</tr>
<tr>
<td>Math</td>
<td>11.89</td>
<td>6.05</td>
<td>10.69</td>
<td>9.06</td>
</tr>
<tr>
<td>Mockito</td>
<td>0.21</td>
<td>0.13</td>
<td>0.21</td>
<td>0.23</td>
</tr>
<tr>
<td>Time</td>
<td>14.49</td>
<td>19.45</td>
<td>9.86</td>
<td>8.87</td>
</tr>
<tr>
<td>Overall</td>
<td>4.98</td>
<td>3.85</td>
<td>5.91</td>
<td>5.20</td>
</tr>
</tbody>
</table>

For the goal of Strong Mutation Coverage, AFFS is slightly outperformed on average by the baselines. Optimizing Strong Mutation alone or the default configuration yields a median improvement of 2.56% over UCB and 5.26% over DSG-Sarsa. No technique demonstrates statistically significant improvements.

To explain the lack of success of AFFS for this goal, there are two factors we will examine: (1) the reward function used by reinforcement learning and its impact on overhead, and (2), the fitness functions that can be combined by AFFS for this goal.

First, we can examine the impact of the overhead of reinforcement learning, particularly calculation of the reward function. As we explained earlier, test generation generally uses a fixed period of time—ten minutes, in our case—to generate test suites. Once that time budget is expired, the best test suite is returned. The amount of time that a single generation takes is not fixed, but depends on the fitness calculation. If multiple fitness functions, or expensive fitness functions are used, then fewer generations of evolution will take place over that time period. Reinforcement learning adds additional overhead on top of this calculation. There are multiple elements that contribute to this overhead, but like fitness functions, calculation of the reward function is a major element. An expensive reward function will further reduce the number of generations of evolution that can be completed.
With exception discovery, the reward function and many of common fitness function combinations were inexpensive, resulting in AFFS techniques being faster than the default configuration. In the case of diversity, the diversity score that was used as both a fitness function and to calculate reward could have been expensive—if the test suite was large—but remained inexpensive thanks to feedback from other fitness functions. Both illustrated the relationship between reward function, fitness function choices, and the overhead of reinforcement learning.

Strong Mutation Coverage is an expensive function to calculate. It requires the execution of the test suites against each mutant. The total cost of calculation depends on the number of mutants, but generally requires multiple test executions, rather than one, to calculate. We use this function not only as the reward function, but as part of many of the fitness function combinations. Although we alternate between Weak and Strong Mutation during reward calculation to control this cost, AFFS has a heavy reward calculation cost that the other approaches lack. This could have an impact on the resulting goal attainment.

In Table 6.14, we display the median time per generation for each approach. Overall, AFFS is again slightly faster than the baselines on average. However, the results vary by project. AFFS is faster than the baselines for three of the six projects. For the remaining three, the Strong Mutation baseline is significantly faster. As can be seen in Table 6.11, the Strong Mutation baseline also yields the highest goal attainment for those projects. The number of generations of evolution plays a role in the resulting goal attainment. For those projects, the slower performance of AFFS may have reduced effectiveness.

For three of the six projects, both AFFS techniques are slower than optimizing Strong Mutation alone, due to the overhead of calculating Strong Mutation...
Table 6.15: Percentage of strong mutation coverage attained by test suites when the number of generations of evolution is fixed to that completed by DSG-Sarsa in 10 minutes. Higher values are better. Best approach is bolded.

<table>
<thead>
<tr>
<th></th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Default</th>
<th>Strong Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>47.00</td>
<td>47.00</td>
<td>49.50</td>
<td>44.50</td>
</tr>
<tr>
<td>Closure</td>
<td>16.00</td>
<td>16.00</td>
<td>17.00</td>
<td>16.00</td>
</tr>
<tr>
<td>Lang</td>
<td><strong>61.00</strong></td>
<td>60.00</td>
<td>57.00</td>
<td>58.00</td>
</tr>
<tr>
<td>Math</td>
<td>73.00</td>
<td><strong>74.00</strong></td>
<td>70.00</td>
<td>71.00</td>
</tr>
<tr>
<td>Mockito</td>
<td>12.00</td>
<td>8.50</td>
<td>9.00</td>
<td>8.00</td>
</tr>
<tr>
<td>Time</td>
<td><strong>67.00</strong></td>
<td>66.00</td>
<td>64.00</td>
<td>64.00</td>
</tr>
<tr>
<td>Overall</td>
<td>38.00</td>
<td><strong>39.00</strong></td>
<td>36.00</td>
<td>37.00</td>
</tr>
</tbody>
</table>

Coverage as part of both fitness and reward. For these projects, optimizing Strong Mutation alone also results in improved goal attainment.

To investigate this possibility, we repeated our experiment for the two baselines, using a fixed number of generations as the search budget instead of a fixed period of time. We used the median number of generations completed by DSG-Sarsa (generally the slower reinforcement learning technique) in ten minutes as the search budget, and repeated generation for the Strong Mutation and default baselines. In Table 6.15, we indicate the median goal attainment for each technique when the number of generations of evolution is fixed. In the right diagram in Figure 6.7, we show box plots of results for all techniques.

For all systems, AFFS now attains equal or higher goal attainment than the Strong Mutation baseline and, for four of the six systems, outperforms the default baseline. Overall, AFFS outperforms both baselines when the number of generations is fixed. The best technique, UCB, outperforms the default baseline by 8.33% and the Strong Mutation baseline by 5.41%.

We repeat our statistical tests as well for this situation. The p-values for the Mann-Whitney rank-sum tests are shown in Table 6.16, and the effect sizes are shown in Table 6.17. We can reject the null hypothesis for AFFS versus both baselines.
Table 6.16: P-Values for Mann-Whitney rank-sum test for Strong Mutation (number of generations fixed).

<table>
<thead>
<tr>
<th></th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Default</th>
<th>Strong Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSG-Sarsa</td>
<td>-</td>
<td>0.60</td>
<td>&lt; 0.01</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>UCB</td>
<td>0.40</td>
<td>-</td>
<td>&lt; 0.01</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>Default</td>
<td>1.00</td>
<td>1.00</td>
<td>-</td>
<td>0.65</td>
</tr>
<tr>
<td>Strong Mutation</td>
<td>1.00</td>
<td>1.00</td>
<td>0.36</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.17: Results of Vargha-Delaney A Measure for Strong Mutation (number of generations fixed). There is no large positive effect sizes were observed.

<table>
<thead>
<tr>
<th></th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Default</th>
<th>Strong Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSG-Sarsa</td>
<td>-</td>
<td>0.50</td>
<td>0.52</td>
<td>0.52</td>
</tr>
<tr>
<td>UCB</td>
<td>0.50</td>
<td>-</td>
<td>0.52</td>
<td>0.52</td>
</tr>
<tr>
<td>Default</td>
<td>0.48</td>
<td>0.48</td>
<td>-</td>
<td>0.50</td>
</tr>
<tr>
<td>Strong Mutation</td>
<td>0.48</td>
<td>0.48</td>
<td>0.50</td>
<td>-</td>
</tr>
</tbody>
</table>

When the number of generations is fixed, the distribution of resulting goal attainment differs. However, all effect sizes remain small. If more time can be allocated to the generation process, AFFS can slightly increase attainment of Strong Mutation Coverage. However, the results for all techniques are still similar.

When the budget is fixed by number of generations rather than time, AFFS techniques outperform the baselines with small effect size. UCB outperforms the default baseline by 8.33% and the Strong Mutation baseline by 5.41%.

A second factor that helps explain our results is that we do not know whether any fitness functions exists that has a significant impact on attainment of Strong Mutation Coverage. The central hypothesis of AFFS is that certain combinations of fitness functions will provide the feedback that the existing fitness function fails to offer to the search. However, none of the functions used in our experiment offer feedback beyond that already offered by the Strong Mutation fitness function. There may be other functions that could offer this feedback, but we do not know what these are or whether they exist.
Table 6.18: Percentage of faults detected by each approach for the Strong Mutation
goal. F#G = fixed number of generations

<table>
<thead>
<tr>
<th></th>
<th>DSG-Sarsa</th>
<th>UCB</th>
<th>Default</th>
<th>SM</th>
<th>Default (F#G)</th>
<th>SM (F#Gen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>54%</td>
<td>69%</td>
<td>69%</td>
<td>54%</td>
<td>65%</td>
<td>50%</td>
</tr>
<tr>
<td>Closure</td>
<td>13%</td>
<td>25%</td>
<td>15%</td>
<td>20%</td>
<td>11%</td>
<td>10%</td>
</tr>
<tr>
<td>Lang</td>
<td>44%</td>
<td>45%</td>
<td>45%</td>
<td>44%</td>
<td>34%</td>
<td>30%</td>
</tr>
<tr>
<td>Math</td>
<td>53%</td>
<td>46%</td>
<td>49%</td>
<td>48%</td>
<td>41%</td>
<td>41%</td>
</tr>
<tr>
<td>Mockito</td>
<td>3%</td>
<td>3%</td>
<td>3%</td>
<td>3%</td>
<td>3%</td>
<td>3%</td>
</tr>
<tr>
<td>Time</td>
<td>50%</td>
<td>54%</td>
<td>50%</td>
<td>42%</td>
<td>46%</td>
<td>38%</td>
</tr>
<tr>
<td>Overall</td>
<td>31%</td>
<td>36%</td>
<td>32%</td>
<td>32%</td>
<td>26%</td>
<td>24%</td>
</tr>
</tbody>
</table>

If the number of generations are fixed, AFFS can offer mild improvements over
the default combination or targeting Strong Mutation on its own, but these improve-
ments are limited. The similarity of the boxplots in Figure 6.7 further demonstrates
the limited feedback offered by other fitness functions, as we do not see significant
reductions in variance like we did with the other two goals.

Limited improvement in median performance and variance indicates that fitness
functions considered by AFFS have limited impact on attainment of Strong
Mutation Coverage. Other functions—still unknown—may improve attainment.

6.3.2 Fault Detection Effectiveness

It is thought that improvements in Strong Mutation Coverage will also lead to im-
provements in detection of real faults as well. We examine fault detection for our
two AFFS approaches and for the two benchmarks in Table 6.18, where we list the
number of faults detected per project. In the last section, we saw that the attained
Strong Mutation Coverage was similar for all approaches, with a slight edge in median
going to directly optimizing Strong Mutation alone. Here, we see fairly similar fault
detection rates for the four approaches as well.
The top approach—despite slightly lower median coverage—was UCB, with 155 faults. It outperforms the default baseline by detecting 16 more faults (11.51% improvement), the Strong Mutation baseline by detecting 17 more faults (12.32% improvement), and DSG-Sarsa by detecting 21 more faults (15.67% improvement). However, DSG-Sarsa was outperformed by both baselines. DSG-Sarsa was also the technique with the lowest median Strong Mutation coverage, but as no significant differences were observed between result distributions, the reason for its slightly weaker performance is not immediately clear. From this, we cannot state that AFFS yields a clear improvement in fault detection over static fitness function choices.

When we fix the search budget in terms of the number of generations of evolution rather than the period of time, we do see that both AFFS techniques significantly outperform the two baselines. In this situation, UCB outperforms the default configuration by 34.78% and Strong Mutation by 49.04%. Some of the fault detection performance of the two baselines can be attributed to additional generations of evolution completed during the 10 minute search budget. When given additional time, AFFS may yield a higher likelihood of fault detection as well.

UCB is able to detect more faults than all other approaches for the Strong Mutation goal, outperforming the default baseline by 11.51%, Strong Mutation alone 12.32%, and DSG-Sarsa by 15.67%. However, DSG-Sarsa is outperformed by the baselines. When the number of generations is fixed, both AFFS approaches significantly outperform the baselines.

6.3.3 Actions Selected by AFFS

In Figure 6.8, we show the top fitness function combinations chosen by DSG-Sarsa for the Strong Mutation Coverage goal. In Figure 6.9, we do the same for UCB. For the first two goals, we saw that UCB more heavily favored exploitation of a
Figure 6.8: Top ten fitness function combinations chosen by **DSG-Sarsa** approach for the Strong Mutation goal. SM = Strong Mutation Coverage, B = Branch Coverage, O = Output Coverage, MNE = Method (No Exception), WM = Weak Mutation Coverage, E = Exception Count

particular action than DSG-Sarsa, which tended towards more exploration. Here, we see the reverse. DSG-Sarsa tends to choose the Strong Mutation fitness function far more often than any other option. UCB chooses Strong Mutation alone as the most common option for three of the six projects, but it spends more time exploring alternative options than DSG-Sarsa. In this case, UCB still attains slightly better median goal attainment, so DSG-Sarsa does not gain an advantage from heavier exploitation over exploration.
We see further evidence for the idea that none of the chosen fitness functions for this goal provide feedback that is sufficient to attain significant gains in Strong Mutation Coverage. The fitness function already designed for this goal, despite attaining relatively low levels of coverage, is still one of the best optimization targets.

Weak Mutation Coverage appears often as well in the most common options, and may help improve coverage of the stronger variant. Output Coverage and the exception count also appear frequently. Both offer potential means to improve Strong
Mutation Coverage, as both have a direct center around manipulation of output. Output Coverage increases the diversity of output response types, which may increase the likelihood of noticing a fault in that output. Similarly, encouraging triggering of exceptions may also increase the likelihood of a visible failure.

Choices made by AFFS approaches suggest that no fitness function combination provided feedback that significantly improved Strong Mutation Coverage. However, Output Coverage and the exception count both manipulate program output, and may lead to small improvements in Strong Mutation Coverage.

6.4 Discussion

In this section, we will summarize the results across all three goals and discuss our observations on the impact of AFFS on multiple aspects of the test generation process.

6.4.1 Impact of AFFS on Goal Attainment

Given a high-level testing goal with no known effective fitness function or a function that is difficult to optimize, our core hypothesis was that adaptive fitness function selection would result in greater attainment of that goal than optimizing the existing fitness function. We further hypothesized that the use of AFFS may result in even greater goal attainment than the optimization of a static set of fitness functions.

For two of the three studied testing goals, both hypotheses were confirmed, with significance. Both EvoSuiteFIT techniques discover and retain more exception-triggering input than the baseline techniques, with DSG-Sarsa yielding more consistent results. Overall, EvoSuiteFIT attains a 100% improvement in median exception discovery over the simple exception count and 33% over the default combination. Additionally, both EvoSuiteFIT techniques produce more diverse test suites than static fitness function choices, with UCB outperforming DSG-Sarsa. Overall, UCB
attains 76.27% better median performance than diversity alone, 41.04% better than the default combination, and 22.72% better than DSG-Sarsa.

For the goal of Strong Mutation Coverage, AFFS is slightly outperformed in terms of median performance by the two baselines. Optimizing Strong Mutation alone or the default configuration yields a median improvement of 2.56% over UCB and 5.26% over DSG-Sarsa. However, no technique demonstrates statistically significant improvements. However, when the search budget is a fixed number of generations rather than time, both AFFS techniques slightly outperformed the baselines. UCB outperforms the default baseline by 8.33% and the Strong Mutation baseline by 5.41%. The effect size is small, but given some additional time for test generation, we still see some improvement from using AFFS over static approaches.

Overall, these results indicate the potential of AFFS for performing test generation for difficult-to-optimize goals. In the future, we plan to explore the utility of AFFS for other goals and other types of testing. Reflecting on the experimental results, we can make the following observations:

**AFFS is an appropriate technique to apply when an effective fitness function does not already exist for the targeted goal.** In Chapter 1, we give the example of Branch Coverage as a goal with an effective fitness function, the branch distance. The branch distance offers clear guidance to the generation process and the means to attain high coverage over many classes. It is unlikely that AFFS would offer improved goal attainment, as other fitness functions are unlikely to offer particular feedback that would be better than the limitations in effectiveness instilled by the overhead of reinforcement learning.

Rather, AFFS can help improve goal attainment in situations where the existing fitness function offers no or little feedback, like the exception count. The exception count allows the retention of suites that trigger exceptions, but offers no feedback for finding new exceptions. The search is still essentially a blind guess. AFFS enables
the discovery of more exceptions by guiding test generation towards, for example, more exploration of the structure of the CUT.

AFFS can also help in situations where a fitness function can offer poor or misleading feedback. Consider the goal of test suite diversity, optimized using the Levenshtein distance. This fitness function rewards test suites that differ from each other, but does not particularly assist in suggesting how to create this difference. In the experiments, we made the surprising discovery that targeting this distance function alone resulted in uncontrolled test suite growth, without a correspondingly large gain in diversity. As this fitness function grows more expensive to calculate as suites get larger, the feedback from this function actually harmed the search process—limiting the number of generations of evolution possible during the ten-minute search budget. Rather than offering helpful feedback, the function actually led the algorithm astray. AFFS was able to produce more diverse test suites and—by keeping the test suite smaller—was substantially faster, despite overhead from reinforcement learning.

**AFFS requires a reward function that is fast to calculate, or requires additional time for test generation.** Reinforcement learning is an additional step in the generation algorithm. No matter how efficient it is, it will add some overhead to the process absent in the normal course of test generation. This overhead can be overcome by strategic selection of fitness functions. AFFS can be faster than the “default” multi-function combinations simply by virtue of calculating fewer fitness functions. However, the reward function must still be fast to calculate to gain the full benefits of using the approach.

For three of the six projects studied in the Strong Mutation experiment, both AFFS techniques are significantly slower than optimizing for Strong Mutation alone, due to the overhead of calculating Strong Mutation Coverage as part of both fitness and reward. For these projects, optimizing Strong Mutation alone also results in
improved goal attainment. In cases where selecting a faster reward function is not possible, more time should be given to the test generation process.

The effect of AFFS is limited by the span of fitness function selections available to choose from. AFFS can offer feedback to the search not offered by the existing fitness function for a goal. However, this relies on the assumption that some other unknown fitness functions will offer that feedback. This was the case for two of our three goals. For the third goal, Strong Mutation Coverage, limited improvement in median performance and variance indicate that the considered fitness functions had limited impact on goal attainment. Other functions—still unknown—may improve attainment of that goal, but there is no guarantee that such functions exist.

One may take from this the lesson that they should add more options for reinforcement learning to choose from. This is not the case. Reinforcement learning must try and retry options, continually refining its estimations of which will improve goal attainment the most. Reinforcement learning will see faster convergence and better results with fewer options to choose from. With too many options, AFFS will spend most of the search trying potentially suboptimal options without ever discovering the best ones. For all the goals, we actively removed some combinations before even starting the experiments. While we did not know which would yield the best results, we applied some care in removing those we knew would likely provide poor results. Therefore, we would recommend a similar process for additional testing goals of care in selecting functions in the first place and a small amount of manual pruning to eliminate options that are likely to provide little help.

6.4.2 Impact of AFFS on Fault Detection

Detection of faults is not a simple matter of maximizing some function, but of selecting the exact input that will trigger an observable failure [39]. The likelihood of fault detection is influenced by a number of factors [37]. The exact relationship of those
factors is not well understood, and detecting a fault is often more of a matter of blind luck than deliberate manipulation of test suites. Still, a major goal of test generation—and a major reason that we try to target many of these fitness functions—is to increase the likelihood that we detect faults with the generated test suites. Maximizing Branch Coverage is not the actual end goal of a tester. Rather, it is a measurable factor that may increase our likelihood of detecting a fault. Therefore, it is important to examine the impact that AFFS has on fault detection.

For the exception discovery goal, both EvoSuiteFIT techniques detect faults missed by the other techniques. UCB detects 11.92 and 249.57% more faults than the baseline techniques, and 4.3% more than DSG-Sarsa. For the goal of test suite diversity, both EvoSuiteFIT techniques also detect faults missed by the other techniques. DSG-Sarsa detects 10.84 and 48.39% more faults than the baseline techniques, and 8.24% more than UCB. Finally, UCB is able to detect more faults than all other approaches for the Strong Mutation goal, outperforming the default baseline by 11.51%, Strong Mutation alone 12.32%, and DSG-Sarsa by 15.67%. DSG-Sarsa is outperformed by the baselines. However, when the number of generations is fixed, both AFFS approaches significantly outperform the baselines.

AFFS approaches generally detect more faults than optimizing the existing fitness function based on the goal or the default combination of functions. We do not fully understand the impact of AFFS on fault detection, and will examine it more closely in future work. However, we have observed several factors that may lead to the higher likelihood of fault detection.

**AFFS results in higher attainment of goals thought to have a positive relationship with fault detection likelihood.** All three of the goals that we studied are hypothesized to improve the likelihood of fault detection. AFFS clearly results in improved attainment of exception discovery and test suite diversity. If hypotheses about these goals are correct, we would expect an increase in the likelihood
of fault detection as well. While this is unlikely to be the sole factor explaining improved fault detection rates, it is a factor that potentially correlates.

**Optimizing multiple fitness functions results in multifaceted test suites.** Each fitness function optimized will have an impact on the resulting test suite, shaping the test cases towards possessing the properties embodied by that fitness function. Naturally, then, optimizing multiple fitness functions can result in test suites that are multifaceted and better able to detect faults [73, 37]. This is not universally the case, and requires careful selection of fitness functions [36]. However, this is indicated by the significant improvement in fault detection between single-function and multi-function approaches in our experiments.

**Optimizing too many fitness functions at once can introduce conflicts between functions and reduce attainment of individual functions.** Optimizing a naively-chosen combination of fitness functions can have a detrimental impact on the resulting test suite. The goals of some fitness functions will conflict with the goals of others. Optimizing one fitness function may come at a significant cost in attainment of another. EvoSuite combines the scores of fitness functions into a single score, and will favor a test suite that highly maximizes one function over a test suite that carefully balances two functions at low levels of attainment. The default combination represents a naive combination of several functions, and there may be conflicts between some of those functions. By intelligently selecting smaller combinations of functions, AFFS may better navigate around such conflicts.

**Changing fitness functions as the suite evolves may result in better final test suites.** AFFS is able to respond to the evolving state of the population of test suites, choosing fitness functions that are best able to improve goal attainment given the current state. This means that certain fitness functions may be applied at certain stages of test generation, but not others. This may be a better method of producing multifaceted test suites than statically applying the same fitness functions.
the entire time. Rather, we may see a staggered approach, where certain properties are evolved into the test suite at different stages of evolution. This may be a more effective approach than trying to imbue many properties at once.

6.4.3 Impact of Reinforcement Learning Overhead

Reinforcement learning introduces overhead into the test generation process. As test generation is generally conducted using a fixed period of time, this overhead could result in a reduction of the number of generations of evolution that can be conducted during this period of time. If this reduction is significant, goal attainment could be reduced as well. We have made multiple observations regarding overhead.

The ability to avoid calculation of unhelpful fitness functions mitigates reinforcement learning overhead. For both the exception discovery and diversity, both AFFS approaches are able to complete more generations of evolution during the search budget than the default combination. An important factor in the number of generations that can be completed during a search budget is the cost of computing fitness. The more fitness functions to be calculated, the longer each generation takes. The default combination naively combines several fitness functions, some of which are likely unhelpful. The AFFS approaches learn to avoid calculating unhelpful functions, achieving speed gains that overcome the introduced overhead.

Feedback from effective fitness functions can help control computational costs. The diversity fitness function grows more expensive to calculate as test case length and suite size grows. By incorporating feedback from additional fitness functions, AFFS is able to prevent uncontrolled test suite growth. As a result, it is actually faster than optimizing diversity alone, as test suites grow rapidly when diversity is the sole fitness function.

Expensive reward functions negatively impact AFFS. For three of the six projects examined in the Strong Mutation experiment, both AFFS techniques are
significantly slower than optimizing Strong Mutation alone due to the overhead of calculating Strong Mutation Coverage as part of both fitness and reward. When we hold the number of generations at a fixed value instead of time, AFFS is more effective. In this situation, the overhead reduces the potential positive impact of AFFS. Either a less expensive reward function should be used, or more time should be allocated to AFFS, in such cases.

6.4.4 Actions Selected by AFFS

The ability to adjust the fitness functions at regular intervals allows EvoSuiteFIT to make strategic choices that refine the test suite. We can see this from examining the actions chosen by UCB and DSG-Sarsa as they attempt to maximize goal attainment. We can make two key observations in this area.

**AFFS enables deeper understanding of the properties that improve goal attainment and how fitness functions can imbue those properties.** The combination of Branch Coverage, exception count, and diversity score seems particularly effective at improving test suite diversity. Ahead of time, we did not know that these three specific functions would enable diversity when used together. Individually, none of these are as effective as they are in combination. These three functions each offer feedback to each other, enabling greater diversity when used in combination. Other complex function combinations similarly act in concert to improve suite diversity. AFFS enabled the discovery of these serendipitous combinations.

Similarly, the choices made by the AFFS approaches suggest that no fitness function combination provided enough feedback to significantly improve Strong Mutation Coverage. However, Output Coverage and the exception count both encourage deviations in program output, and may lead to small improvements in Strong Mutation Coverage. Ahead of time, we did not understand their potential impact on attain-
ment of Strong Mutation coverage, but inspecting the choices made by AFFS gave us insight into factors that could promote additional attainment of our goal.

**Fitness function combinations that are ineffective in a static context may be effective when used by AFFS to diversify a pre-evolved population of suites.** Many of the most common choices made by AFFS—particularly for the exception discovery and diversity goals—would result in poor test suites when used as the only fitness functions for the entire generation process. For example, the combination of exception count and Method Coverage (Top-Level, No Exception) was chosen very often for the exception discovery goal. Used in a static context, the produced suites are quite weak at both goal attainment (discovering exceptions) and fault detection. However, this combination is applied strategically by AFFS to suites evolved already using other functions, such as Branch Coverage. The suites are already robust at, for example, covering the code structure. Then, these combinations can be applied to reshape the suites into ones that discover new exceptions. A similar observation can be made in the other experiments. The diversity score is used quite a lot to shape existing suites, when it is a poor target in a static context. In the Strong Mutation experiment, Output Coverage and exception count offer some gain in coverage, but would yield weak coverage if used as the sole targets of generation.

Observation of the choices made by AFFS makes it clear how the stateful evolution of test suites can be harnessed to improve goal attainment. Fitness functions shape the test suites that emerge from search-based test generation. They imbue the suites with certain emphasized properties. *These properties do not need to be imbued at the same time.* Rather, fitness functions can be used to reshape a suite over time, and different functions may be best applied in different sequences or at different stages of this evolution. A future direction for this research will be to further understand this process, and how it can best be controlled to produce effective test suites. Little
research in search-based test generation has looked at the controlled staggering of fitness functions, but our observations indicate the potential importance it has.

6.4.5 Choice of Reinforcement Learning Approach

In this research, we implemented two reinforcement learning approaches—UCB and DSG-Sarsa. These approaches use different mechanisms for choosing actions and associating actions with particular states. It is natural, then, to compare the two in terms of their performance. In this regard, we can make the following observation: Overall, UCB attains a slight advantage over DSG-Sarsa. However, there are significant exceptions that rule out universal recommendation of UCB.

In terms of goal attainment, UCB is equivalent or better than DSG-Sarsa in performance. The median performance of UCB is better than the median performance of DSG-Sarsa for the exception discovery and Strong Mutation goals. For the exception discovery goal, the two attain equivalent median performance. However, for both the exception discovery and diversity goals, DSG-Sarsa returns more consistent results—with less of a difference between the first and third quartile results for both goals. DSG-Sarsa would be the recommendation for the exception discovery goal, if we look purely at goal attainment.

In terms of fault detection, UCB outperforms DSG-Sarsa for both the exception discovery and Strong Mutation Coverage goals. DSG-Sarsa attains better fault detection for the diversity goal, even though UCB attains better coverage. Finally, in terms of speed, UCB is faster for the diversity and Strong Mutation goals, but slower than DSG-Sarsa for the exception discovery goal.

In general, we lack enough evidence to recommend one approach over the other. UCB attains a slight lead in multiple categories for multiple goals, but is outperformed by DSG-Sarsa in enough cases to rule out its use. Overall, both approaches appear useful, and more observations will be needed to make any sort of conclusive
judgement. Given the success of the two approaches, it may even make sense to execute both and pool their test cases.

6.5 Threats to Validity

External Validity: Our study has focused on six systems (seven for the diversity goal)—a relatively small number. Nevertheless, we believe that such systems are representative of, at minimum, other small to medium-sized Java systems. We believe that Defects4J offers enough fault examples that our results are generalizable to other, sufficiently similar, projects. As Defects4J is used across multiple research fields, the use of this dataset also allows comparisons of our approach with other research, and allows others to replicate our experiments.

We have implemented our reinforcement learning techniques in a single test generation framework. There are many search-based methods of generating tests and these methods may yield different results. Unfortunately, no other generation framework offers the same number and variety of fitness functions. Therefore, a more thorough comparison of tool performance cannot be made at this time. By using the same framework to generate all test suites, we can compare our approach to the baselines on an equivalent basis.

Similarly, we have chosen two reinforcement learning algorithms to implement, out of the many that have been proposed. We chose these two specifically because (a) they are well-understood and widely-used, and (b) they represent different approaches to handling state (tabular versus approximate). Because these approaches have substantial differences in how they work, we believe we present a reasonable portrait of how AFFS would work. Still, different reinforcement learning techniques may lead to different outcomes.

To control experiment cost, we have only generated ten test suites for each combination of fault, budget, and configuration. It is possible that larger sample sizes
may yield different results. However, given the consistency of our experiment results, we believe that this is a sufficient number of repetitions to draw stable conclusions.

**Conclusion Validity:** When using statistical analyses, we have attempted to ensure the base assumptions behind these analyses are met. We have favored non-parametric methods, as distribution characteristics are not generally known a priori, and normality cannot be assumed.
CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this chapter, we summarize the research to date and offer potential directions for future research.

7.1 Summary

Search-based test generation is guided by feedback from one or more fitness functions—scoring functions that judge solution optimality. Choosing informative fitness functions is crucial to meeting the goals of a tester. Unfortunately, many goals—such as forcing the class-under-test to throw exceptions, increasing test suite diversity, and attaining Strong Mutation Coverage—do not have effective fitness function formulations. We propose that meeting such goals requires treating fitness function identification as a secondary optimization step. An adaptive algorithm that can vary the selection of fitness functions could adjust its selection throughout the generation process to maximize goal attainment, based on the current population of test suites.

To test this hypothesis, we have implemented two reinforcement learning algorithms in the EvoSuite framework, and used these algorithms to dynamically set the fitness functions used during generation for the three goals identified above.

We have evaluated EvoSuiteFIT for each of our three goals on a set of Java case examples in terms of the ability of generated test suites to achieve the targeted goal and in terms of the ability of the generated suites to detect faults. In each case, we compare the two reinforcement learning approaches to two baselines. The first baseline is the current practice—a fitness function based on the goal that may not offer
sufficient feedback. We additionally compare to a set of multiple fitness functions—the full set of functions that AFFS can choose among for that goal—that serves as a “best guess” a human might make at a combination of fitness functions that would produce effective test suites. We have found that:

- Both EvoSuiteFIT techniques discover and retain more exception-triggering input than the baseline techniques, with DSG-Sarsa yielding more consistent results. EvoSuiteFIT attains a 100% improvement in median exception discovery over the simple exception count and 33% over the default combination.

- Both EvoSuiteFIT techniques produce more diverse test suites, with UCB outperforming DSG-Sarsa. UCB attains 76.27% better median performance than diversity alone, 41.04% better than the default combination, and 22.72% better than DSG-Sarsa.

- For the goal of Strong Mutation Coverage, EvoSuiteFIT is slightly outperformed in median performance by the two baselines. Optimizing a baseline yields a median improvement of 2.56% over UCB and 5.26% over DSG-Sarsa. However, no technique demonstrates statistically significant improvements. When the search budget is a fixed number of generations rather than time, both EvoSuiteFIT techniques slightly outperform the baselines. UCB outperforms the default baseline by 8.33% and the Strong Mutation baseline by 5.41%. The effect size is small, but given additional time for test generation, we see some improvement from using AFFS over static approaches.

- For all goals, EvoSuiteFIT techniques detect faults missed by the other techniques. UCB detects 11.92 and 249.57% more faults than the baseline techniques, and 4.3% more than DSG-Sarsa, for the exception discovery goal. For the goal of test suite diversity, DSG-Sarsa detects 10.84 and 48.39% more faults than the baseline techniques, and 8.24% more than UCB. Finally, UCB is able
to detect more faults than all other approaches for the Strong Mutation goal, outperforming the default baseline by 11.51%, Strong Mutation alone 12.32%, and DSG-Sarsa by 15.67%. DSG-Sarsa is outperformed by the baselines for this goal. However, when the number of generations is fixed, both EvoSuiteFIT approaches significantly outperform the baselines.

- We find that AFFS is an appropriate technique to apply when an effective fitness function does not already exist for the targeted goal. However, AFFS requires a reward function that is fast to calculate, or requires additional time for test generation. Further, the effect of AFFS is limited by the span of fitness functions available to choose from. If none of the chosen functions correlate to the goal of interest, then improvements in goal attainment will be limited.

- Improvements in fault detection may arise because of higher attainment of goals thought to have a positive relationship with fault detection likelihood, optimizing multiple fitness functions—but avoiding needlessly complex and conflicting functions—and changing fitness functions as the suite evolves rather than applying all functions at once.

- While reinforcement learning adds overhead to test generation, EvoSuiteFIT is often faster than the default static configuration because the ability to avoid calculation of unhelpful fitness functions mitigates this overhead. Further, feedback from effective fitness functions can help control computational costs.

- The ability to adjust the fitness functions at regular intervals allows EvoSuiteFIT to make strategic choices that refine the test suite and allows us to attain a deeper understanding of the properties that link to goal attainment and how fitness functions can work together to imbue those properties. Fitness function combinations that are ineffective in a static context may be effective when used by AFFS to diversify a pre-evolved population of suites.
The use of AFFS allows EvoSuiteFIT to identify combinations of fitness functions effective at achieving our testing goals, and strategically vary that set of functions throughout the ongoing generation process. We hypothesize that other goals without known effective fitness function representations could also be maximized in a similar manner. We make EvoSuiteFIT available to others for use in test generation research or practice.

7.2 Future Work

The research performed to date suggests the potential for adaptive fitness function selection to affect test generation effectiveness. There are many directions we could take to expand the capabilities and applicability of the technique. Some areas we may pursue in future work include:

- **Extensive analysis of fault detection:** AFFS approaches generally discovered more faults than baseline approaches. However, the exact reasons for this are unclear. While each goal is thought to improve the probability of fault detection with higher attainment, we are not certain if gains are due to higher attainment or other elements of the AFFS process. For example, AFFS techniques did not necessarily attain higher goal attainment for the Strong Mutation goal, but UCB still detected more faults. We intended to perform a more thorough analysis of the faults detected exclusively by AFFS to determine why those particular faults were detected. In addition, we will perform analyses to see if there is a correlation between higher goal attainment and higher fault detection.

- **Additional goals:** We will examine whether AFFS can be applied to additional goals. For example, we recently proposed a coverage criterion intended to address masking in bytecode, based on Multiple Condition Coverage [16]. This
criterion was ineffective as a fitness function, offering insufficient feedback to
the search. This and other goals may benefit from the use of AFFS to provide
missing feedback.

• **New testing scenarios:** We have applied AFFS to generation of unit tests
  for Java. We wish to expand our focus to differing levels of granularity, such as
  system or GUI testing, and additional technological platforms, like Python.

• **Empirical investigation of controlled application of fitness functions:**
  Our results indicate that staggered application of fitness functions can be more
  effective than attempting to use one set of functions throughout the generation
  process. We seek understanding of how controlled application can shape test
  suites, increase goal attainment, and improve fault detection.

• **Exploration of transfer learning:** In reinforcement learning, efforts are often
  made to learn policies for action selection from one set of case examples and
  apply those to others. In this work, we did not investigate this process—known
  as transfer learning [53]. In the future, we would like to see if policies learned
  from a set of classes can be applied to new classes to improve AFFS results.

• **Generation of fitness functions:** We use pre-chosen sets of fitness functions
  as the actions that the reinforcement learning algorithm can select from. Al-
  ternatively, we could attempt to construct new fitness functions out of a set
  of ingredients—such as the subgoals of existing fitness functions. We could
  attempt to learn custom fitness functions for a CUT and goal.

• **Automated suggestion of fitness function combinations:** We manually
  constructed a set of fitness function combinations for each goal. Many of these
  options are likely poor for goal attainment, and we almost certainly omitted po-
  tentially effective options. We would like to build a model that can predict, for a
class, which sets of fitness functions are most likely to improve goal attainment. This is a learning problem, where we could train a model on a set of classes and metadata collected from each class, such as past test generation performance, mutation testing results, or source code metrics [76]. Such a model could be used to prune fitness functions that may be ineffective.

We will explore some or all of these areas in upcoming research projects.
BIBLIOGRAPHY


