Noname manuscript No.
(will be inserted by the editor)

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

Hussein Almulla · Gregory Gay

Received: date / Accepted: date

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 unit test generation 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 Java case examples. EvoSuiteFIT techniques attain significant improvements for two of the three goals, and show limited improvements on the third when the number of generations of evolution is fixed. Additionally, for two of the three goals, EvoSuiteFIT detects faults missed by the other techniques. The ability to adjust fitness functions allows strategic choices that efficiently produce more effective test suites, and examining these choices offers insight into how to attain our testing goals. We find that adaptive fitness function selection is a powerful technique to apply when an effective fitness function does not already exist for achieving a testing goal.

Keywords Automated Test Generation · Search-Based Test Generation · Reinforcement Learning · Hyperheuristic Search

This research was supported by Vetenskapsrådet grant 2019-05275.

Hussein Almulla
University of South Carolina, Columbia, SC, USA
E-mail: halmulla@email.sc.edu

Gregory Gay
Chalmers and the University of Gothenburg, Gothenburg, Sweden
E-mail: greg@greggay.com

1 Introduction

The testing of software is crucial, as testing is our primary means of ensuring that complex software is robust and operates correctly [1]. However, testing is an expensive task that can consume much of the development budget [1]. 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 [2]. If test creation could be even partially automated, the effort and cost of testing could be significantly reduced.

One promising method of automating test creation is search-based test generation [3,4]. Test input selection can naturally be seen as a search problem [5]. 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 those that meets their chosen goal. This search can then be automated. Given a measurable goal, a metaheuristic optimization **algorithm** can systematically sample the space of possible test input and manipulate those samples, guided by feedback from one or more **fitness functions**—scoring functions that judge the optimality of the chosen input [6]. In other words: ***algorithm + fitness functions*** $\Longrightarrow$ ***goal***.

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

Consider, for example, Branch Coverage—a measurement of *how many parts* of the code have 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 executed [1]. If our *goal* is to achieve 100% Branch Coverage, there are multiple fitness functions that could be used to guide the algorithm towards meeting that goal. A simple fitness function could measure the attained coverage. A test suite that attains 75% Branch Coverage is inherently better than one that attains 50% Branch Coverage. This tells the algorithm which test suites to favor, leading to higher and higher attainment of Branch Coverage.

However, there is a more informative fitness function that leads to faster attainment. Instead, we could take each branch outcome we wish to cover 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 [4]. Fitness functions based on this concept, the *branch distance* [7], offer additional feedback by offering both a measurement of *how much* of the goal has been met as well as clues on how to *attain further coverage*. If our goal is Branch Coverage, we have known and effective fitness functions that enable attainment of that goal. Unfortunately, many goals *do not* have a known or effective fitness function. 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 [8]. Exceptions indicate fault 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 *diverse* test input is more effective than similar input [9,10]. This intuition has led to effective automated test generation, prioritization, and reduction [9].
– **Strong Mutation Coverage:** Mutation testing is a practice where synthetic faults (mutants) are injected into the code. If test suites detect these faults, they are thought to be more robust to real faults. In *Weak* Mutation Coverage, a mutant is 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 that the infected state propagates to the program output, offering clear evidence that the fault was detected [11].

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 counting thrown exceptions as a fitness function [12]. Unfortunately, this count yields poor results, as it offers the algorithm no guidance for improving its guesses [6,13,14].
– While numerous diversity metrics exist—for example, the *Levenshtein distance* [10] 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, which measures how close execution came to reaching the mutated line and corrupting the program state [15]. It is more difficult to offer feedback on how to propagate corruption to the output. Current fitness functions offer probabilistic estimations of propagation [15,16]. However, such estimations are generally too coarse-grained to accurately guide the search.

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 produce test suites that trigger exceptions, optimizing different functions (e.g., targeting both branch distance and exception count) might achieve that goal.

We simply need to identify that selection. There are many combinations of fitness functions that could be selected, and the "correct" choices may be specific to the goal and system/class-under-test (SUT/CUT). In fact, the "correct" choices may even vary *during* test case generation, as search-based processes evolve test suites over time in a stateful process. Therefore, **we seek a systematic method of automatically identifying and adapting the selection of fitness functions** that is appropriate for a variety of high-level testing goals.

A class of search-based test generation approaches are known as *hyperheuristic*, or self-adaptive, approaches [17,18]. 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 search algorithm during evolution to improve solution quality [17]. We propose a hyperheuristic search that strategically adjusts the chosen fitness functions throughout the generation process to maximize attainment of a desired goal.

Through the use of reinforcement learning [19], this approach is able to select the most appropriate fitness functions for a CUT 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 agent selects fitness functions, and after evolving test suites using these functions for a set number of generations, the change in the reward score will be evaluated and the agent decides whether to continue using the set of functions known to best improve the reward (*exploitation*) or to try different functions in order to refine expectations on the change in reward (*exploration*). We refer to this hyperheuristic as **adaptive fitness function selection** (AFFS).

We have implemented two reinforcement learning algorithms—Upper Confidence Bound (UCB) and Differential Semi-Gradient Sarsa (DSG-Sarsa) [19]—in the EvoSuite unit test generation framework for Java [20]. We refer to the modified framework as **EvoSuiteFIT**. We have evaluated EvoSuiteFIT for each of the three goals listed above on a set of Java case examples in terms of (a) the ability to produce test suites that achieve the targeted goal and (b) the ability of the generated suites to detect real faults. In each case, we compare the two reinforcement learning approaches to three baselines: (a) current practice—a fitness function based on the goal that may not offer sufficient feedback, (b) 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, and (c), a set of fitness functions randomly selected from the choices available to AFFS. We have found:

– Both EvoSuiteFIT techniques outperform all baselines with at least medium effect size for the goals of exception discovery and suite diversity—attaining improvements of up to 107.14% in goal attainment. For the goal of Strong Mutation Coverage, no technique demonstrates significant improvements. When the search budget is a fixed number of generations rather than time, both EvoSuiteFIT techniques slightly outperform the baselines (up to 8.33% improvement). However, the effect size is still negligible.
– Both EvoSuiteFIT techniques detect faults missed by the other techniques for the exception discovery goal (up to 259.90% improvement). UCB is able to detect more faults for the Strong Mutation goal (12.50% improvement), and when the number of generations is fixed, both EvoSuiteFIT approaches

outperform the baselines (up to 50.00% improvement). Both techniques are outperformed by the random baseline for the diversity goal (34.74% difference), but outperform the other 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. However, higher goal attainment does not ensure fault detection.
- 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 (up to 94.27% faster than baselines). 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.

We have previously proposed adaptive fitness function selection, and demonstrated its potential for increasing the number of discovered exceptions [21]. We also have published a small pilot study for the Gson case examples and the diversity goal [22]. This publication extends both studies in significant ways:

- We perform more extensive experiments and analyses for the exception discovery goal, and perform the first full experiments for the diversity goal.
- We add a third testing goal—Strong Mutation Coverage.
- We have added a third baseline—random selection of fitness functions.
- We perform cross-goal analyses to better understand the capabilities of AFFS, leading to a richer discussion than in the previous studies.

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[1] and our empirical data[2] available to others for use in research or practice.

[1] EvoSuiteFIT is available from `https://github.com/hukh/evosuite/tree/evosuitefit`.
[2] We make our research data available at `https://doi.org/10.5281/zenodo.4524786`.

```
@Test
public void testPrintMessage() {
    String str = "Test␣Message";
    TransformCase tCase = new TransformCase(str);
    String upperCaseStr = str.toUpperCase();
    assertEquals(upperCaseStr, tCase.getText());
}
```

Fig. 1: Example of a unit test case written using the JUnit notation for Java.

2 Background

2.1 Unit Testing

Testing can be performed at various levels of granularity. In this research, we are focused on *unit testing* [1]. Unit testing is where the smallest segment of code that can be tested in isolation from the rest of the system—often a class [23]—is tested. Unit tests are written as executable code. We refer to a purposefully grouped set of test cases as a *test suite*. When code changes, developers can re-execute the test suite 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 unit test, written in JUnit, is shown in Figure 1. 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. Then, the test case will validate the output of the called methods and the class variables against a set of encoded expectations—the *test oracle*—to determine whether the test passes or fails. In a unit test, the oracle is typically formulated as a series of assertions on the values of method output and class attributes [2]. In the example in Figure 1, the *test input* consists of passing a string to the constructor of the `TransformCase` class, then calling its `getText()` method. This method should transform the string to upper-case. To ensure this is the case, we use an assertion to check whether the output of the call is equal to an upper-case version of the provided string.

2.2 Search-Based Test Generation

Automation has a critical role in controlling the cost of testing [24,25]. 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 input selection can naturally be seen as a search problem [5]. 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 [4,26]. 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*) [27].

Metaheuristics are often inspired by natural phenomena, such as swarm behavior [28] or evolution [29]. 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 [30].

In search-based test generation, the fitness functions capture testing objectives 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, as coverage criteria can be straightforwardly transformed into efficient, informative fitness functions [7]. Search-based generation often can achieve higher coverage than developer-created tests [31]. Due to the non-linear nature of software, resulting from branching control structures, a real-world program's search space is large and complex [26]. Metaheuristic search—by strategically sampling from that space—can scale to larger problems than many other generation algorithms [32]. Such approaches have been applied to a wide variety of testing goals and scenarios [26].

A special class of search-based approaches are known as *hyperheuristic*, or self-adaptive, approaches [17,21,22]. These approaches incorporate a learning phase in order to automatically tune the search strategy towards particular problem instances [33]. Hyperheuristic search has been used, for example, to change parameters of the metaheuristic during evolution [17].

Hyperheuristic search can, essentially, be thought of as "using a heuristic to choose a heuristic." 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 may include aspects of the metaheuristic such as population tuning mechanics (e.g., the crossover and mutation rates of a genetic algorithm) or, in this study, the choice of fitness functions. The metaheuristic operates directly on the problem space, attempting to optimize its own effectiveness using the options selected by the high-level hyperheuristic layer [34,35].

Hyperhueristic approaches can be divided into two types—selection and generation. Selection-based approaches select low-level heuristics from a preexisting set. Generation-based approaches create new heuristics using the components of existing heuristics as building blocks [36]. Selection-based hyperheurstics are more common, especially in software testing research, as they are often easier to implement and are suited to a wider range of problems [33]. However, generation-based approaches may yield better solutions when applied successfully. In this research, we use a selection-based hyperheuristic approach, but may explore generation-based approaches in future work.

2.3 Reinforcement Learning

Reinforcement learning 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. To understand reinforcement learning, consider the n-armed bandit problem [37]. 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 [19].

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 receive a reward value. The agent will update the expected reward for the chosen set using the new information—updating the *policy* it uses to choose the next action.

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 [17,19,38].

In this work, we consider two different types of reinforcement learning—tabular solution methods and approximation solution methods [19]. Tabular methods are generally used in cases where states and action apace 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 by generalizing from previously encountered states [19].

3 Technical Approach and Implementation

In theory, fitness functions should be selected to maximize 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—a simple count of exceptions thrown, the Levenshtein distance, and a probabilistic estimation of state propagation to output. All three of these fitness representations have weaknesses. Consider the fitness function for exception discovery. Counting the thrown exceptions 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.

The other two goals are also difficult to optimize. The Levenshtein distance can effectively *minimize* the distance between two strings, as the actions a test generation takes have a direct and learn-able impact on this score. It is less helpful when one wants to *maximize* the distance—to make the test suites more diverse—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. Similarly, Strong Mutation Coverage requires that a triggered fault propagate to an observable failure in the output. Offering feedback on the likelihood of propagation is a complex problem, and current approaches only provide course-grained estimations that insufficiently guide the search [15, 16].

All three of these goals 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 specific actions not suggested by fitness function feedback. Such properties are common when examining the goals a tester might have when creating test suites. In this research, our 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.

Even if existing fitness functions are insufficient, *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.* Search-based generation can simultaneously target multiple fitness functions [39]. Each fitness function further shapes the test suite, imbuing it with additional properties. This offers an opportunity to provide that missing feedback. We can augment—or even replace—the existing fitness representations with fitness functions that better direct the search towards optimization of our core, high-level goal.

We propose that careful selection—at different points in the generation process—of the set of fitness functions could result in test suites that better meet our goals than existing baselines. If this is true, identifying these fitness functions becomes a secondary search problem, tackled as an additional hyperheuristic optimization within the normal test generation process [38]. 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. 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 have implemented AFFS by extending the EvoSuite test generation framework [20] with two online reinforcement learning algorithms—Upper Confidence Bound (UCB) and Differential Semi-Gradient Sarsa (DSG-Sarsa) [19]. 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

Algorithm 1 Overview of the UCB algorithm.

```
1: Initialization:
2: max = 0
3: for a = 0 ... number of actions do                    ▷ Initialize for all actions
4:     numberTimesActionSelected_a = 0
5:     sumReward_a = 0
6: end for
7: Each time an action is selected:
8: if numberGenerationsElapsed < numberActionsTried then
9:     action = getAction(numberGenerationsElapsed)    ▷ Try all actions once before using RL
10:    numberActionsTried = numberActionsTried + 1
11: else
12:    for a = 0 ... number of actions do
13:        upperBound = 0
14:        if numberTimesSelected_a > 0 then
15:            avgReward = $(\frac{sumReward_a}{numSelection_a})$
16:            upperBound = (avgReward + c $\sqrt{\frac{\ln(numberGenerationsElapsed)}{numberTimesActionSelected_a}}$)
17:        else
18:            upperBound = doubleMaxValue
19:        end if
20:        if upperBound > max then
21:            max = upperBound                        ▷ Update estimation of best action.
22:            action = getAction(a)         ▷ Choose the action with the highest estimation.
23:        end if
24:    end for
25: end if
26: numberTimesActionSelected_action = numberTimesActionSelected_action + 1
27: return action
```

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 [23]. In this study, we implemented AFFS in EvoSuite version 1.0.7. We call our approach **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.

3.1 Upper Confidence Bound (UCB) Algorithm

In the n-armed bandit problem [19], 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 whether 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 [40]. 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 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. In test generation, the time step is the current number of generations that have elapsed. Using this notation, the expected reward of action a is $q_* \doteq E[R_t|A_t = a]$. We apply the Upper Confidence Bound to select the action [40]:

$$A_t \doteq max[Q_t(a) + c\sqrt{\frac{ln(t)}{N_t(a)}}] \tag{1}$$

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. The confidence level is multiplied by the square root of the natural log of the time step divided by the number of times the action has been selected. $Q_t(a)$ denotes the estimated value of choosing a combination of fitness functions (a), which can be calculated as:

$$Q_t(a) = \frac{1}{N_t}\sum_{i=1}^{t-1} R_i(a) \tag{2}$$

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. In this algorithm, we first ensure that all actions are tried once in a random order (lines 8-10 in Algorithm 1). This allows us to seed expected rewards of applying actions before using the standard selection procedure. We when proceed to apply the set of equations defined above to update our estimation of gain in reward and select the action with the highest estimation.

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 containing the current set of fitness functions, the current fitness value for that set of functions, the test suite size, and the coverage of the subgoals associated with the fitness functions.[3]

[3] For example, in Strong Mutation Coverage, this would be the percent of mutants detected through an observed difference in output.

Algorithm 2 Overview of the DSG-Sarsa algorithm.

```
1: After action A is taken, the test suite is in state S
2: if numberGenerationsElapsed < numberActionsSelected then
3:     A_new = getAction(numberGenerations)          ▷ Try all actions once before using RL
4:     numberActionsSelected = numberActionsSelected + 1
5: else
6:     Select action A_new as a function of $q(S_0, ., w)$ using $\epsilon - greedy$ policy
7: end if
8: Observe S_new and the reward after taking the action
9: numTimesActionSelected_{A_new} = numTimesActionSelected_{A_new} + 1
10: $\delta$ = reward + averageReward + q(S_new, A_new, w) - q(S, A, w)     ▷ Update the error function
11: averageReward = averageReward + $\beta$ * $\delta$     ▷ Update the average change in the reward score
12: w = w + $\alpha$ * $\delta$ * q(S, A, w)                        ▷ Update the weight vector
13: A = A_new
14: S = S_new
```

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 by a large number of actual test suites as is a summarization of facets of the suite. To address this potential limitation, we also implemented a second algorithm, DSG-Sarsa, which generalizes expectations from previously-encountered states [19].

3.2 Differential Semi-Gradient Sarsa (DSG-Sarsa)

Approximate solution methods generalize from previously encountered states [19]. Therefore, approximate methods are appropriate for problems with a large or unconstrained state-space where finding exact solutions is not feasible with limited time [41]. 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) [19].

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 [19]. 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 generation, an action—a choice of fitness functions—is applied, and the test suite evolves to a new state S‘, with observed reward R. Again, we start by trying each action once in a random order to seed estimates (lines 2-4). Then, we choose a new action A‘, using the formula:

$$\hat{q}(S, A, W) \doteq W^{\top} \cdot X(S, A) = \sum w_i x_i(S, A) \tag{3}$$

This action-value function is calculated by the inner product of weights and feature vectors. X(S,A) is the feature vector: $X(s, A) = (x_1(S, A), x_2(S, A), \dots x_d(S, A))$. As noted previously, the feature vector describes the current state of a test suite as the current set of fitness functions, the current fitness value for that set of functions, the test suite size, and the coverage of function goals.

W represents a weight vector, used to bias action selection [19]. 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:

$$W_{t+1} \doteq W_t + \alpha\delta\nabla\hat{q}(S_t, A_t, W_t) \tag{4}$$

To evaluate the chosen action, the algorithm calculates the error function (δ), 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 $\hat{q}(S_t, A_t, W_t)$ [19]. 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 $\varepsilon - greedy$ [19]. 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 is called a differential value function [19]:

$$\delta_t = R_{t+1} - \bar{R}_t + \hat{q}(S_{t+1}, A_{t+1}, W_t) - \hat{q}(S_t, A_t, W_t) \tag{5}$$

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

$$\bar{R}_{t+1} = R_t + \beta\delta \tag{6}$$

β 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 [42,43]. 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.[4]

The general test generation process in EvoSuite is depicted in Figure 2. 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.

[4] 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.

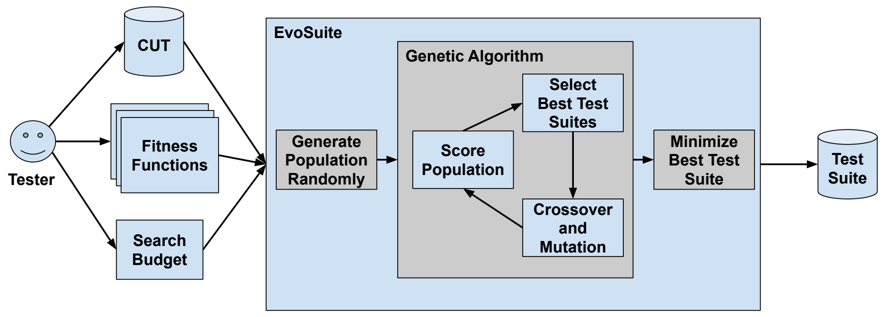


Fig. 2: A simplified overview of EvoSuite's test generation process.

An initial population of test suites is generated randomly, then a metaheuristic algorithm evolves that population until the search budget is exhausted. In this research, we have integrated AFFS into the standard Genetic Algorithm (GA).

Each generation, the GA evaluates the current population (a collection of test suites) using the chosen fitness functions. The score is calculated for each fitness function, the scores are normalized to a 0-1 scale, then the scores are summed into a single score. Lower scores are preferred. The standard GA in EvoSuite is not a true multi-objective approach, i.e., it does not try to balance each fitness function. Sufficient improvements to one of the chosen functions will result in a suite being favored, even if it attains worse scores in other functions than other test suites.

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 [12]. We make use of nine of these functions in our work:

– **Exception Count:** A count of the unique exceptions thrown by a test suite. Exceptions are tracked using the name of the Exception class and the method where the exception was thrown. In addition, exceptions are separated into those that are declared (in method signature), explicit (developer used a `throw` expression), and implicit (unplanned exceptions).
– **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 [1]. 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 [7].

- **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 [44].
- **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 [45]. 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 [15].
- **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 adds an estimation of the likelihood of propagation, the propagation distance, by estimating the impact of corrupted state [15].

Rojas et al. provide more details on each of these fitness functions [12]. We additionally implemented a Test Suite Diversity fitness function based on the Levensthein distance, which we will discuss in Section 3.4.2

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. 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

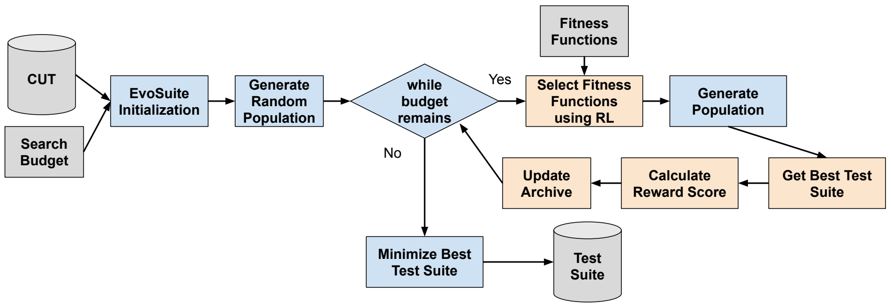


Fig. 3: The modified test generation process. Steps in orange are introduced or modified by AFFS.

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:         if approach == DSGSARSA then
 8:             DSGSarsa(iterNum, bestSolution.coverage, bestSolution.size(), reward, bestSolution.fitness)
 9:         else if approach == UCB then
10:             UCB(reward, action)
11:         end if
12:         action = GetBestFFCombination()   ▷ Determine new fitness functions.
13:         UpdateCurrentFitnessFunction()    ▷ Update fitness functions.
14:         iterNum++
15:     end if
16:     generation++
17: end while
```

policies to new classes in this work [46]. 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 effective combinations include the exception count, even though the 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 different sets of fitness functions.

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 goals for minimization and 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 [10]. 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 and tb) can be calculated as follows [10]:

$$lev_{ta,tb}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) == 0 \\ min \begin{cases} lev_{ta,tb}(i-1,j)+1 \\ lev_{ta,tb}(i,j-1)+1 \\ lev_{ta,tb}(i-1,j-1)+1_{(ta_i \neq tb_j)} \end{cases} & otherwise \end{cases} \tag{7}$$

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:

$$div(TS) = \sum_{ta,tb}^{TS} lev_{ta,tb} \tag{8}$$

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:

$$\frac{1}{1+div(TS)} \tag{9}$$

The fitness function calculation iterates through the test cases in a given test suite. Before calculating the distance, the variables and their values are extracted from the test cases. 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. Consider the following fragment: `String x = "var"; String y = x; String z = y;`. Variables `x`, `y`, and `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.

To calculate diversity, each pair of test cases is compared. From each pair of tests, each pair of statements is compared. The Levenshtein distance is calculated between each of these pairs and added to the diversity score, then it is returned to the core process. The Levenshtein distance calculation uses a classic matrix-based approach [47] 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. This is a smaller pool of actions than was used for the other two goals. This is because the calculation of Strong Mutation Coverage requires more time than calculating other fitness functions. Attaining a clear estimation of the expected reward of choosing an action requires that each action be tried multiple times. If it is expensive to calculate fitness, however, the total number of generations that can be completed within a time period may be restricted. This reduces the time that can be spent exploring different actions. To compensate for this cost, we have reduced the number of possible actions by (1) limiting combinations to three fitness functions, and (2), removing potentially redundant fitness functions (Line Coverage, Direct Branch Coverage, Method Coverage). Unlike the other two goals, not all combinations include Strong Mutation Coverage. Instead, we conducted a small experiment and utilized the best combinations found in that experiment.

Reward Function: We use the mutation score as the reward function. This is the percentage of mutants detected: $\frac{DetectedMutants}{TotalNumberofMutants} * 100$. 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.

4 Related Work

This section will provide an overview of related work on hyperheuristic search-based software testing approaches, as well as test generation research related to exception discovery, test suite diversity, and strong mutation coverage to give insight into past research in topics related to this study.

4.1 Hyperheuristics in Search-Based Software Testing

Hyperheuristic search has been employed in addressing multiple several search-based software engineering problems. Fitness function selection has been performed by hyperheuristic search in other domains, such as production scheduling [48,49]. 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. We briefly give an overview of this work below to illustrate how hyperheuristics have been used to improve other aspects of the search algorithm.

Jia et al. [17,38] used reinforcement learning to tune the metaheuristic for Combinatorial Interaction Testing, using the Simulated Annealing algorithm in the outer layer and using an n-Armed Bandit approach for learning and choosing the best operator(s) (out of six) to tune the performance of the algorithm. Zamli et al. also used a hyper-heuristic approach for CIT [50], using Tabu search as a high-level hyperheuristic to select a low-level heuristic from four algorithms. Later, Zamli et al. used hyperheuristic search to learn optimal selection and acceptance mechanisms used by the metaheuristic in CIT [51]. Din et al. also applied hyperheuristic search to CIT [52], using parameter-free choice functions to rank low-level heuristics for selection. Din and Zamli use Exponential Monte Carlo with Counter (EMCO) as a hyperheuristic to select a low-level heuristic in CIT [53]. Ahmed et al. [54] compare EMCO against an improved version using Q-learning, called Q-EMCO, to select the best operator based on historical information.

Guizzo et al. used a reinforcement learning-based hyper-heuristic search to tune the metaheuristic algorithm for optimizing the integration and test order problem [18,55]. 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 [56]. Guizzo et al. also applied a hyperheuristic to the NSGA-II MOEA to address ITO in Google Guava [57]. Mariani et al. introduced an approach that depends on an offline hyperheuristic named GEMOITO to generate MOEAs to solve the ITO problem [58]. Guizzo et al. later used design patterns to improve the design of MOEA to reduce coupling and increase reusability of components [59]. They implemented the patterns into GEMOITO. They found that they were able to reuse MOEA components without decreasing the quality the algorithm results.

Ferreira et al. proposed the use of hyperheuristic search in software product line (SPL) testing [60]. Software Product Lines are sets of systems that share a common set of features that are customized for particular market segments or customers. 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 [61,62], 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. Filho et al. also proposed a hyperheuristic that uses grammatical evolution to generate MOEAs for SPL testing [63]. Their approach considers three factors—pairwise coverage, mutation score, and cost—and generates a MOEA using crossover and mutation operators tuned to the feature model being considered. Filho et al. extended this work [64, 65] to Preference-Based Evolutionary Multi-objective Algorithms, which consider user preferences during the search.

Kumari and Srinivas [35] 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.

Moghadam et al. [66] have proposed a framework that uses adaptive learning to generate test cases for stress testing. Bauersfeld et al. [67] introduced an automated testing approach for robustness testing of GUIs based on reinforcement learning. Building on their previous work [68], 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. Grechanik proposed an adaptive, feedback-driven approach to generating input designed to highlight performance issues [69]. Their technique, FOREPOST, initially generates test cases randomly, and the results are evaluated. Then, the results are feed 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. This approach is not based on metaheuristic search, but still uses feedback to improve test case generation.

4.2 Crash and Exception Discovery

Joffe el al. [70] 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. Romano et al. [71] focused on targeting null pointer exceptions, providing an approach that can identify code that can cause this exception by looking at execution paths. The approach generates a control flow graph, which is used 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.

Due to inadequate detection of exceptions in automated test case generation, Goffi et al. [72, 73] 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 assertions, which are used in test cases to improve detection of faults. Extended work widens the range of behaviors that can be assessed by these oracles [73]. 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

Albunian investigated the impact of diversity on search-based test generation [74], proposing a phonetypic and genotypic representation to measure diversity. They studied the influence of five selection mechanisms and five fitness functions. Feldt et al. [75] proposed a new diversity fitness function based on normalized compression distance. Ma et al. [76] 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. Vogel et al. [77] investigated using diversity metrics in search-based generation of test cases for Android mobile applications. They proposed an approach that diversifies the population at the initialization and selection steps, then preserves and improvse diversity during the search. All of these approaches are complementary to our proposed approach, and could potentially be used in combination with our approach to yield improved suite diversity.

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. Fraser el al. [15] proposed a fitness function representation for strong mutation that is implemented in EvoSuite. This function estimates 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. We use this fitness function in our work, and attempt to use hyperheuristic search to further improve optimization of this function. Souza el al. [78] proposed an automated test generation approach for strong mutation using Hill Climbing, a simple local search algorithm. 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. Papadakkis and Malevris [16] 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.

Like with suite diversity, all of these fitness function representations 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. [15], 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. [79] 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. Harman et al. proposed an approach that aims to achieve strong coverage of first and higher-order mutants [80]. 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 limtiations 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.

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 benchmark [81] for each of our goals—exception discovery, test suite diversity, and Strong Mutation Coverage. We will 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 Strong 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 asking a broader question than whether increased goal attainment leads to increased likelihood of fault detection. We are asking if any element of the AFFS process increases that likelihood. AFFS is a complex process, and other factors—like varying the fitness functions over time—could also impact 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 evolution 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. Finally, to better understand AFFS, we will also examine trends in the fitness functions choices. We will also identify and discuss limitations of the current implementation.

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

1. **Collected Case Examples:** We have used a collection of case examples, from the Defects4J fault benchmark, 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 three baselines—an existing fitness function for that goal (current practice), a combination of all fitness functions that AFFS can chose from (a "best guess"), and random selection from the choices available to AFFS. 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 a benchmark of real faults extracted from Java projects [81].[5] 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. 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.

[5] Available from `http://defects4j.org`

The "fixed" version must be explicitly labeled as a fix to an issue, and changes imposed 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 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 benchmark. To compare results between the three high-level goals, we focus on the same projects. 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 pilot study [22]—bringing the total case examples for the diversity experiment to 452.

5.2 Test Suite Generation

For all three goals, and for each bug-affected class from each case example used from Defects4J, we have generated test suites using UCB and DSG-Sarsa. In addition, we generate tests for 2-3 baseline approaches representing current practice:

- **Current Practice**: We use the existing fitness function representation of that goal.[6] 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 count of the number of unique exceptions thrown by a test suite.
  - **Strong Mutation Coverage**: The existing fitness function in EvoSuite for measuring Strong Mutation Coverage, based on an estimated propagation of corrupted state [15].
  - **Diversity Score (Levenshtein Distance)**: A new fitness function based on the textual changes required to transform one test case into another.
- **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 [12]. 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.

[6] All three functions are explained in more detail in Section 3.

– **Random Selection of Functions**: The final baseline is a random selection of fitness functions, chosen from the combinations available to AFFS. For each fault, we make a random selection and use that selection for all trials for that fault. We employ this baseline for the exception and diversity goals, but omit it for the Strong Mutation goal in order to control experiment costs, and due to limited value from adding this baseline for that goal (as we will discuss further in Section 6.3).

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 [23]. Tests that fail on the faulty version, then, detect behavioral differences between the two versions.[7]

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 [82].

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 19,750 test suites (ten trials, five approaches, 395 faults), representing over 3,291 hours of computation time. For the Diversity experiment, this resulted in the generation of 22,600 test suites (ten trials, five approaches, 452 faults), representing over 3,766 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 2,893 hours of computation time. We performed experiments on Amazon EC2 infrastructure, where all VMs shared an identical hardware and software configuration (t2.large instances, with two CPU threads and 8GB of RAM, running Amazon Linux).

Generation tools may generate flaky (unstable) tests [23]. 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:

– **Exception Experiment:**
  – **Number of Unique Exceptions Discovered During Generation**

[7] This is identical practice to other studies using EvoSuite and Defects4J, e.g. [23,82].

  - **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.

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 makes their fitness function selections, and the limitations of adaptive fitness function selection. The following subsections outline and discuss our observations.

6.1 Goal: Exception Discovery

6.1.1 Ability to Discover Exceptions

Our first question asks whether AFFS can be used to more effectively meet our goal of generating tests that trigger more unique exceptions than baseline static fitness function configurations. 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

Table 1: Median count of exceptions discovered for each technique. Counts are normalized between 0-1 for each fault to allow comparison across case examples. Higher scores are better. The highest median is **bolded**.

System	DSG-Sarsa	UCB	Exception Count	Default	Random
Chart	**0.83**	0.82	0.33	0.63	**0.83**
Closure	**0.83**	0.82	0.30	0.55	0.59
Lang	**0.92**	0.91	0.61	0.83	**0.92**
Math	**0.89**	**0.89**	0.54	0.67	0.67
Mockito	0.83	**0.86**	0.50	0.50	0.50
Time	**0.87**	0.83	0.39	0.63	0.71
Overall	**0.87**	0.86	0.42	0.63	0.68

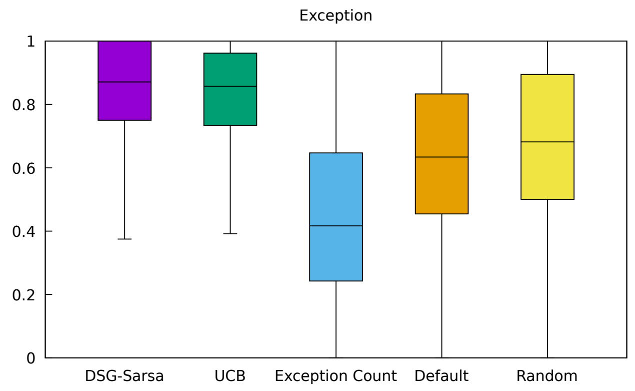


Fig. 4: Unique exceptions discovered by each technique. Counts are normalized between 0-1 for each fault to allow comparison across case examples.

when testing another, we should not compare five to thirty. Instead, we *normalize* exception counts between 0-1 for each class-under-test, using the formula:

$$\left(\frac{\text{Number of Exceptions Observed In This Trial For CUT}}{\text{Maximum Number of Observed Exceptions In Any Trial For CUT}}\right) \tag{10}$$

This normalization allows fair comparison between case examples.

The median count of unique exceptions discovered for each technique is listed in Table 1 for each project and overall. Boxplots of the exceptions discovered are shown in Figure 4. **Higher scores are better.** Overall, both AFFS techniques have a higher median performance in both measurements than all three baselines. In particular, both approaches outperform current practices—attaining up to a 176.67% improvement in median exceptions discovered over the basic exception count and up to a 72.00% improvement over the eight-function default configuration. Overall, DSG-Sarsa attains a 107.14% improvement in median exception discovery over the simple exception count and 38.10% 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.

On a per-system basis, the third baseline—a random selection—ties in median performance with DSG-Sarsa for Chart and Lang. However, it is outperformed by both AFFS approaches for the other systems. Overall, DSG-Sarsa outperforms the

Table 2: Results of Vargha-Delaney A Measure for exceptions discovered. Large positive effect sizes are **bolded**. Medium positive effect sizes are *italicized.*

	DSG-Sarsa	**UCB**	**Exception**	**Default**	**Random**
DSG-Sarsa	-	0.52	**0.86**	**0.74**	*0.69*
UCB	0.48	-	**0.85**	*0.73*	*0.67*
Exception	0.14	0.15	-	0.33	0.30
Default	0.26	0.27	*0.67*	-	0.45
Random	0.32	0.33	*0.70*	0.55	-

random baseline by 27.94% in median performance, and UCB outperforms it by 26.47%. Notably, the random baseline outperforms both of the other baselines—using the existing fitness function and combining several fitness functions. Testers would be better served by choosing a small random set of fitness functions than by blindly combining all options available. We will discuss the primary reasons for this shortly.

Figure 4 also shows that both techniques not only offer a higher median than the baselines, but also have a narrower interquartile spread, showing relatively consistent performance. UCB yields more consistent performance, as shown by the decreased variance. However, DSG-Sarsa has a slightly higher median performance and third quartile.

We perform statistical analysis to assess our observations. First, we are interested in establishing whether there are differences in performance between the different AFFS techniques and the baselines. If so, we are then interested in examining the magnitude of the differences. For each pair of techniques and baselines, we formulate hypothesis and null hypotheses:

- H: Generated test suites have different distributions of exception discovery results depending on the technique used to generate the suite.
- $H0$: Observations of exception discovery for all techniques are drawn from the same distribution.

Our observations are drawn from an unknown distribution. To evaluate the null hypothesis without any assumptions on distribution, we use the Friedman test, a non-parametric test for determining whether there are any statistically significant differences between the distributions of three or more paired groups. For each fault, we have a set of paired observations based on the exception discovery performance of the suites generated for that fault by each technique. We apply the test with $\alpha = 0.05$. This test yielded a p-value < 0.001, indicating differences in performance between the techniques.

Therefore, we then used the Vargha-Delaney A measure to assess effect size [83]. The results for exception discovery are listed in Table 2, with large effect sizes in bold ($E \geq 0.80$) and medium effect sizes in italics ($0.80 > E \geq 0.70$). DSG-Sarsa outperforms the default and exception count baselines with large effect size, and the random baseline with medium effect size. UCB outperforms the exception count baseline with a large effect size, and outperforms the default and random baselines with a medium effect size. DSG-Sarsa also outperforms UCB, but with a negligible effect size.

Table 3: Percentage of faults detected by each approach for the exception goal. The best approach is **bolded**.

System	DSG-Sarsa	UCB	Exception Count	Default	Random
Chart	80.77%	**84.62%**	38.46%	65.39%	69.23%
Closure	6.77%	6.02%	3.76%	15.04%	**15.79%**
Lang	58.46%	**64.62%**	16.92%	52.31%	53.85%
Math	**68.87%**	**68.87%**	12.26%	57.55%	41.51%
Mockito	13.16%	**15.79%**	7.89%	13.14%	7.89%
Time	59.25%	**66.67%**	18.52%	51.85%	55.56%
Overall	41.01%	**42.78%**	11.90%	38.23%	34.43%

Both EvoSuiteFIT techniques discover more exceptions than the baseline techniques with significance. DSG-Sarsa outperforms the exception count and default baselines with large effect size (107.14%, 38.10% improvement in median) and the random baseline with medium effect (27.94%).

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. Table 3 lists the percentage 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—4.32% more than DSG-Sarsa, 11.90% more than default, 24.25% more than the random baseline, and 259.50% more than the exception count.

The default and random baselines outperform EvoSuiteFIT for one project—Closure. It is likely that triggering these faults requires incorrect output, rather than an exception. The baselines are outperformed on all other systems.

DSG-Sarsa yielded slightly better performance at goal attainment. 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.

As an initial assessment of the connection between goal attainment and fault detection, we calculated the point-biserial correlation coefficient between the normalized goal attainment and whether the fault was detected (the dichotomous variable). This calculation yielded a coefficient of only 0.22, indicating only a weak correlation between fault detection and goal attainment. This suggests that, while improving the ability of the suite to throw exceptions has a positive relationship with detection of the specific faults used in this study, the relationship is far from the only factor influencing fault detection. Rather, factors such as the fitness functions applied in service of improving the goal may also increase the likelihood of selecting input that triggers a particular fault. 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 may have merit.

Table 4: Median time per generation (in seconds) for exception discovery goal. The lowest median is **bolded**.

	DSG-Sarsa	**UCB**	**Random**	**Default**
Chart	**0.24**	0.26	1.38	3.29
Closure	**0.32**	0.49	0.45	5.71
Lang	**0.30**	0.44	2.02	4.38
Math	**0.14**	0.22	2.64	3.03
Mockito	**0.03**	**0.03**	0.05	0.08
Time	0.33	0.43	2.67	3.72
Overall	**0.22**	0.31	0.91	3.84

For the exception discovery goal, both EvoSuiteFIT techniques detect faults missed by the other techniques. UCB detects up to 259.90% more faults than the baselines.

6.1.3 Impact of Reinforcement Learning Overhead

Search-based test generation approaches are generally benchmarked using a fixed time budget [23]. 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 due to 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 this cost.

Table 4 lists the median time per generation for DSG-Sarsa, UCB, and the default and random baselines. 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.

From Table 4, 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 large 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 random selection, if not faster. Figure 5 helps explain why. In Figure 5, we show the ten actions chosen most often by DSG-Sarsa for each system. While DSG-Sarsa can choose combinations of up to four fitness functions, it rarely does so in practice. Often, either the simple exception count is used, or the combination of the exception count and method (no exception) coverage. The latter is a count of methods called without throwing an exception, which can be calculated efficiently. Because EvoSuiteFIT can strategically change its fitness function selection, overhead added by reinforcement learning is mitigated by the gain in speed from the ability to avoid calculating unhelpful fitness functions.

The need to calculate these fitness scores helps explain the difference in performance. The default baseline requires more time per generation than AFFS techniques. In turn, this means that AFFS techniques are able to refine test suites further during the time allocated to the search than the default baseline, or even the random baseline. Reinforcement learning adds overhead to the generation process, but the ability to vary the generation strategy can mitigate the impact of that overhead.

The ability to avoid unhelpful fitness functions mitigates reinforcement learning overhead. Both AFFS approaches are able to complete more generations of evolution during than the default and random baselines, with DSG-Sarsa being 94.27% and 75.82% faster on average.

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 5, we display the ten actions chosen most often by DSG-Sarsa for each system, and in Figure 6, we do the same for UCB.

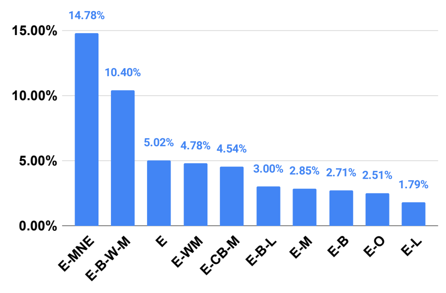
From Figure 5-6, we can see differences between projects in terms of which choices are made and how often choices are made. For example, DSG-Sarsa frequently used a combination of exception count, Direct Branch Coverage, Weak Mutation Coverage, and Output Coverage for the Time system, but not for others. 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 exception count with 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.

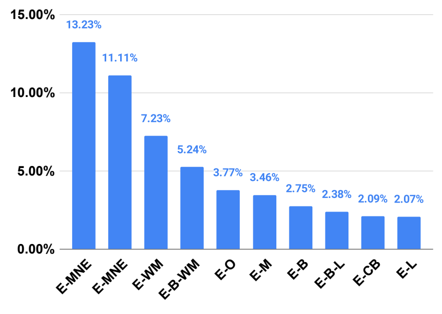
UCB chooses complex sets of actions, combinations of 3-4 fitness functions, somewhat more often than DSG-Sarsa. However, it does not necessarily do so significantly more often than it chooses simple combinations. The most noticeable factor about UCB, as seen in Figure 6, is that it heavily favors the 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 use 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.

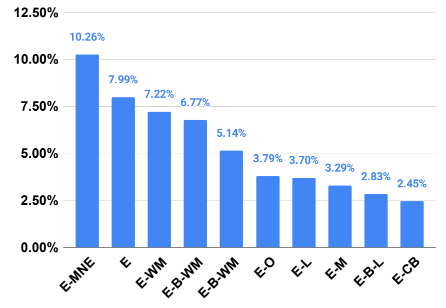
The EX-MNE combination appears to be contradictory at first. MNE rewards calling each method of the CUT and it executing without throwing an exception.



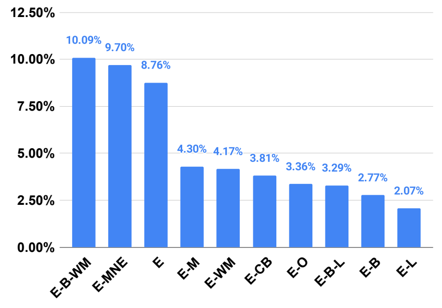
(a) Chart



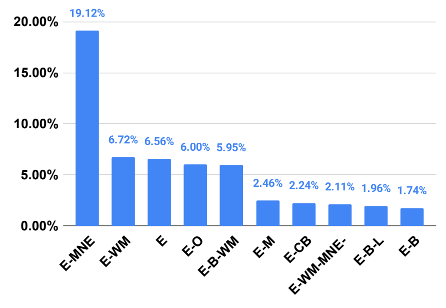
(b) Closure



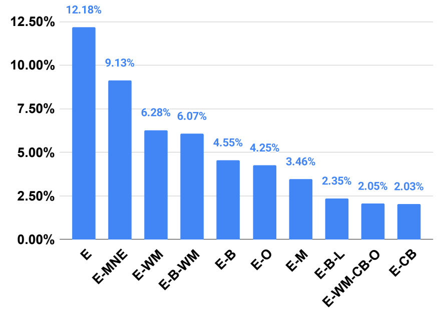
(c) Lang



(d) Math



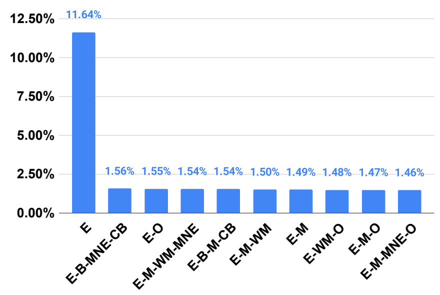
(e) Mockito



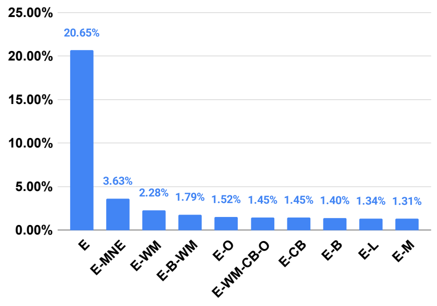
(f) Time

Fig. 5: 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

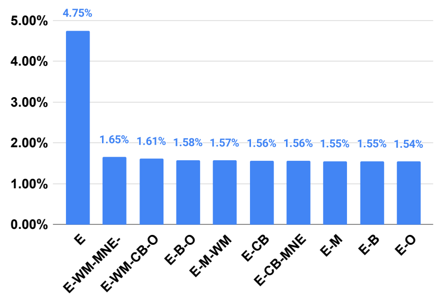
However, it is possible to attain high fitness in both functions at the same time, as each test case can call multiple methods (and there are multiple test cases in a suite). A test case might call method `X()` twice. If it executed once without an exception and threw an exception in the second call, it would increase fitness for both functions.



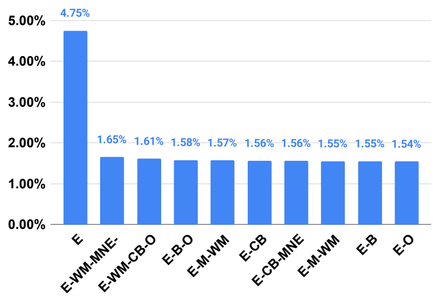
(a) Chart



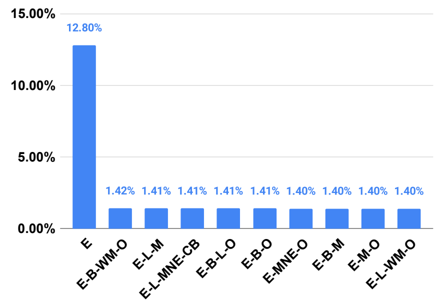
(b) Closure



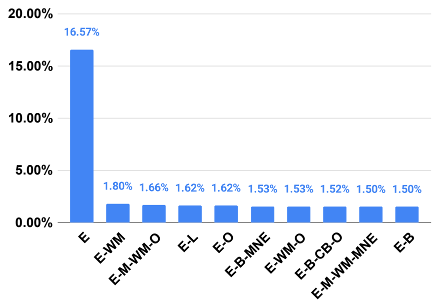
(c) Lang



(d) Math



(e) Mockito



(f) Time

Fig. 6: Top ten function combinations chosen by **UCB** 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 Coverage

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 pop-

Table 5: Median diversity fitness score of the produced test suite. Score is between 0-1, with **lower scores being better**. The lowest median is **bolded**.

	DSG-Sarsa	**UCB**	**Default**	**Diversity Score**	**Random**
Chart	7.45E-07	**4.20E-07**	1.11E-06	4.40E-06	1.12E-06
Closure	**1.49E-06**	1.59E-06	1.82E-06	5.64E-06	2.40E-06
Gson	1.43E-06	**9.51E-07**	1.86E-06	3.46E-06	2.28E-06
Lang	5.21E-07	**3.05E-07**	1.11E-06	3.85E-06	1.32E-06
Math	1.32E-06	**1.06E-06**	1.54E-06	4.02E-06	1.88E-06
Mockito	**2.32E-06**	2.35E-06	3.69E-06	5.20E-06	4.30E-06
Time	6.58E-07	**4.39E-07**	9.74E-07	3.51E-06	9.00E-07
Overall	1.32E-06	**1.02E-06**	1.73E-06	4.30E-06	1.87E-06

ulation 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. When the combination is employed, new exceptions may be discovered, but the simple count of methods called might prevent loss of code coverage as other fitness functions are explored. The EX-MNE combination may be ineffective in a static context, as it does not offer enough feedback to fully explore the code structure. 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. In this case, scores range between 0-1, and **lower scores** indicate higher levels of diversity. In Table 5, we indicate the median diversity score for each technique for each project and overall. In Figure 7, we show boxplots for each technique.

From Table 5 and Figure 7, we see that both AFFS techniques outperform all three baselines. 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 the worse median score and the wide variance in Figure 7.

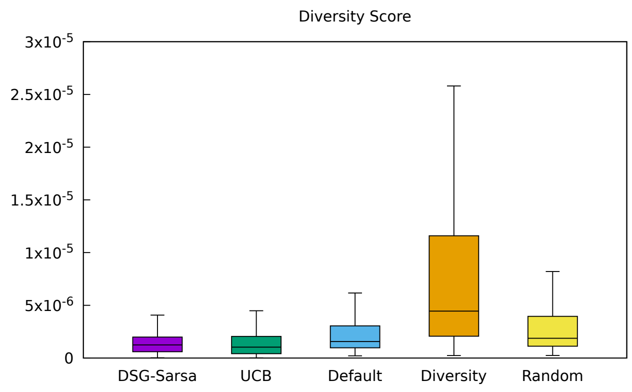


Fig. 7: Diversity fitness scores of the produced test suites. Score is between 0-1, with lower scores being better.

Table 6: Results of Vargha-Delaney A Measure for diversity score. Large positive effect sizes are **bolded**. Medium effect sizes are *italicized*.

	DSG-Sarsa	**UCB**	**Default**	**Diversity Score**	**Random**
DSG-Sarsa	-	0.47	*0.63*	**0.87**	*0.68*
UCB	0.53	-	*0.66*	**0.86**	*0.71*
Default	0.36	0.34	-	**0.78**	0.56
Diversity Score	0.13	0.13	0.23	-	0.28
Random	0.32	0.29	0.44	*0.72*	-

The default baseline attains better results than diversity-alone or the random baseline. However, both AFFS techniques outperform it. Overall, the best technique, UCB, attains 76.27% better median performance than diversity alone, 45.45% better than the random baseline, 41.04% better than the default combination, and 22.72% better than DSG-Sarsa. From Figure 7, we also see that DSG-Sarsa also shows less variance in its results than UCB. UCB attains *better* results, but DSG-Sarsa is more consistent. The state approximation performed by DSG-Sarsa may result in less variance in performance.

We again perform statistical analysis to assess our observations. We formulate hypothesis and null hypothesis:

- H: Generated test suites have different distributions of diversity score results depending on the technique used to generate the suite.
- $H0$: Observations of diversity score for all techniques are drawn from the same distribution.

The Friedman test confirms, with p-value < 0.001, that there are significant differences between the distributions of the AFFS approaches and baselines. The results for the Vargha-Delaney A measure are listed in Table 6, with large effect sizes in bold and medium effect sizes in italics. The results of this test further confirm our observations. Both techniques outperform the diversity score baseline with large effect size and the other baselines with medium effect size. The default combination outperforms diversity-only with medium effective size, and the random baseline outperforms the diversity-only baseline with medium effect size.

Table 7: Percentage of faults detected by each approach for the diversity goal. The best approach is **bolded**.

	DSG-Sarsa	**UCB**	**Default**	**Diversity Score**	**Random**
Chart	34.61%	42.31%	34.61%	26.92%	**53.85%**
Closure	10.80%	8.52%	7.34%	5.68%	**12.50%**
Gson	**22.00%**	16.67%	16.67%	11.11%	16.67%
Lang	26.15%	21.54%	24.61%	18.46%	**32.31%**
Math	31.13%	30.19%	30.19%	21.69%	**46.22%**
Mockito	5.26%	5.26%	5.26%	5.26%	**10.53%**
Time	29.63%	29.63%	29.63%	22.22%	**40.74%**
Overall	20.18%	18.64%	18.20%	13.60%	**27.19%**

Both EvoSuiteFIT techniques produce more diverse test suites than static baselines with significance. UCB outperforms the diversity score with large effect size (76.27% improvement in median) and the default and random baselines with medium effect (41.04%, 45.45%).

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 stimuli* to the CUT should be more likely to detect faults just by virtue of not performing the same actions over and over again. AFFS does increase suite diversity. Therefore, we also are curious about whether it increases the potential for fault detection. Table 7 lists the percentage of faults detected by each approach.

Overall, the AFFS approaches detect more faults than the diversity score and default baselines. DSG-Sarsa detects 8.26% more faults than UCB, 10.88% more than the default combination, and 48.38% more more than optimizing for diversity alone. However, the random baseline detects significantly more faults than either the other baselines or the AFFS techniques. The random baseline detects 34.74% more faults than DSG-Sarsa and 45.87% more than UCB.

We again calculated the point-biserial correlation coefficient between diversity and fault detection. The calculated coefficient was 0.01—indicating a practically non-existent relationship between goal attainment and fault detection in this experiment. This should not be interpreted as a conclusion that improved diversity will not improve the likelihood of fault detection *in general*. However, in this specific experiment, the diversity of a test suite was not a significant factor in whether faults were detected. Highly-specific input is needed to trigger many of the faults in Defects4J, and improving suite diversity does little to locate those faults.

This can also be seen by comparing the percentage of faults detected between this experiment and the exception discovery experiment. Many more faults were detected in that experiment, suggesting that other fitness functions may offer better guidance for locating the specific input that is needed to trigger those faults. Again, we stress that this does not mean that test suite diversity is in unimportant goal, or that it is not helpful in general. However, it may be less helpful for the specific examples in Defects4J than other fitness functions.

The fitness function combinations that could be selected for the random baseline are the same that AFFS techniques can choose from when attempting to

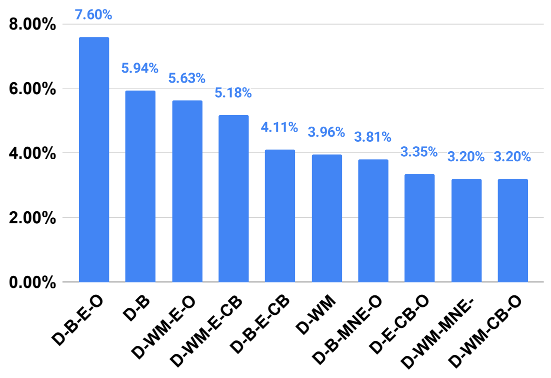


Fig. 8: The ten fitness function combinations chosen most often when generating tests for the random baseline when a fault was detected.

improve diversity. The AFFS techniques attain higher diversity, but the functions that best improve diversity may differ from those that are most likely to result in detection of these specific faults. In Figure 9, we show the ten fitness function combinations chosen most often by DSG-Sarsa for each system, and in Figure 10, we do the same for UCB. To contrast, we show the ten most-selected fitness function combinations for the random baseline when a fault was detected in Figure 8. We see little overlap between these figures, further indicating that there is little connection between improved diversity and the likelihood of fault detection for these case examples.

DSG-Sarsa detects more faults than UCB, even though UCB attains higher diversity. Given the observations above, the difference is likely to be due to a combination of the stochastic nature of search-based generation and differences in the decision making processes for the two algorithms. DSG-Sarsa is more likely to choose fitness functions that are better for detecting the studied faults than UCB.

The random baseline detects 34.74% more faults than DSG-Sarsa and 45.87% more than UCB. Improved diversity does not lead to improved likelihood of fault detection for these case examples.

6.2.3 Impact of Reinforcement Learning Overhead

In Table 8, 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 and random baselines. While reinforcement learning introduces overhead—including the calculation of diversity as part of the reward score—*this cost is less than naively calculating 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. However, the cost of

Table 8: Median time per generation (in seconds) for the goal of test suite diversity. The lowest median is **bolded**.

	DSG-Sarsa	UCB	Default	Diversity Score	Random
Chart	5.26	**3.39**	8.38	6.93	7.73
Closure	8.63	**6.44**	12.58	12.37	11.84
Gson	4.09	**2.95**	4.70	4.64	4.33
Lang	3.65	**2.63**	7.09	4.60	6.27
Math	4.09	**3.05**	5.68	4.15	4.64
Mockito	5.98	**4.16**	6.22	5.96	5.90
Time	3.63	**2.77**	6.32	5.29	5.28
Overall	4.09	**3.05**	8.39	6.81	7.56

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 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 targeting diversity alone were significantly larger than those being evolved by DSG-Sarsa, with the latter being 41% smaller on average in the studied examples.

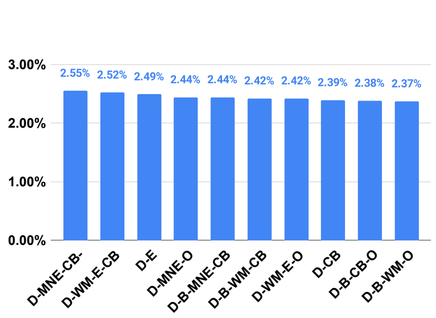
In the absence of feedback from additional fitness functions, optimizing the diversity fitness function alone led to unconstrained growth in test suites. Creating longer tests is one *potential* path to improving diversity, but not a guaranteed one—it could still result in similar test cases. Ultimately, 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 complete more generations of evolution during the search than the baselines, with UCB being 63.65% faster than the default baseline, 59.66% faster than random, and 55.22% faster than diversity alone. By incorporating additional feedback, AFFS controls the cost of the diversity calculation by preventing unconstrained test suite growth.

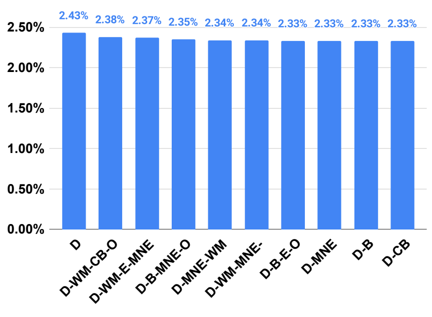
6.2.4 Actions Selected by AFFS

In Figure 9, we display the ten fitness function combinations chosen most often by DSG-Sarsa for the diversity goal, and in Figure 10, we do the same for UCB. Examining these choices may offer insight into the results attained by AFFS.

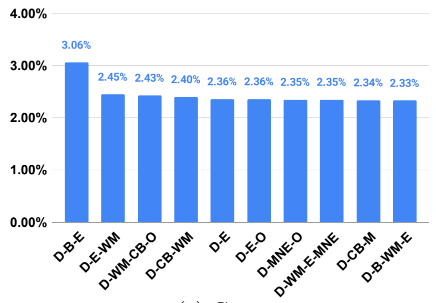
The first observation we make is that the combination of diversity score, exception count, and Branch Coverage is chosen the most often. It is the top choice made by DSG-Sarsa for four of the seven systems, and the top choice for UCB for all projects. This combination provides several key ingredients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



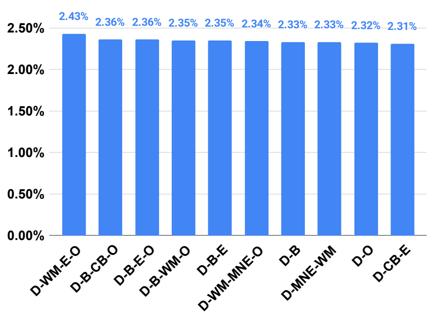
(a) Chart



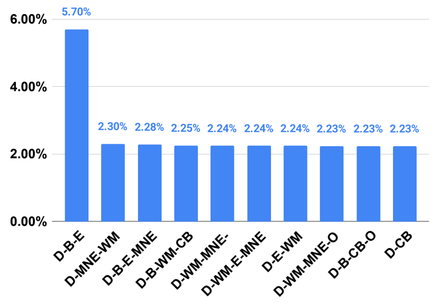
(b) Closure



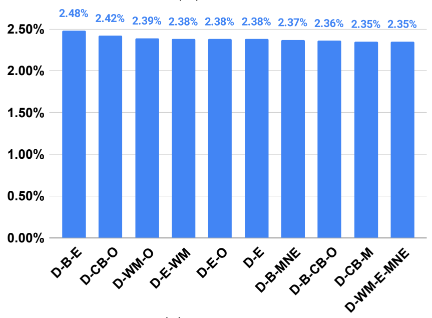
(c) Gson



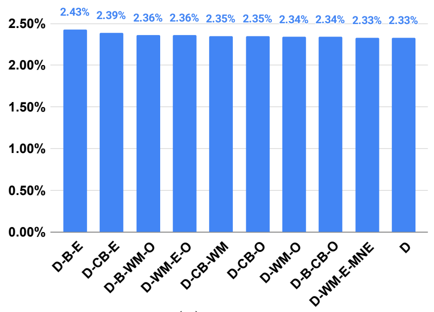
(d) Lang



(e) Math

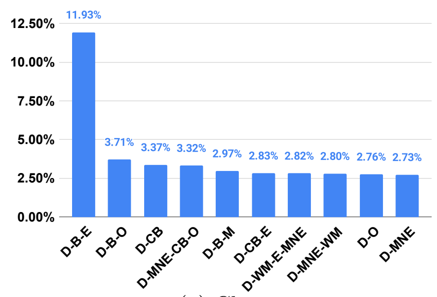


(f) Mockito

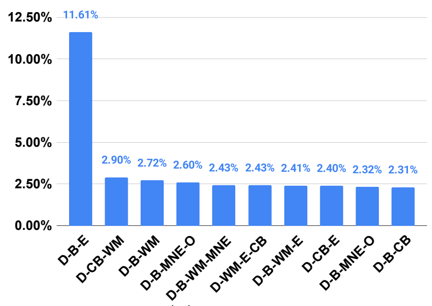


(g) Time

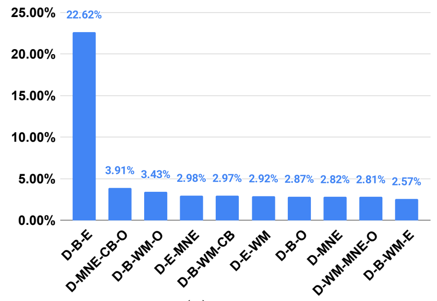
Fig. 9: Top ten function combinations chosen by **DSG-Sarsa** for diversity. D = Diversity Score, B = Branch, CB = Direct Branch, O = Output, M = Method, MNE = Method (No Exception), WM = Weak Mutation



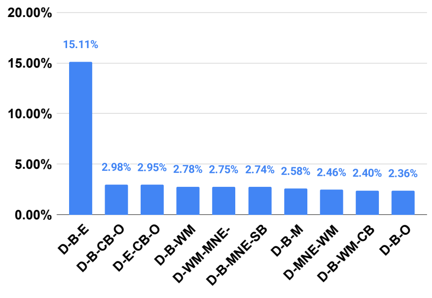
(a) Chart



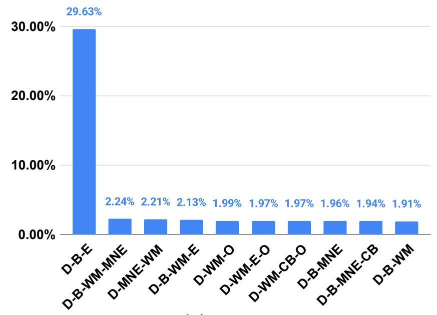
(b) Closure



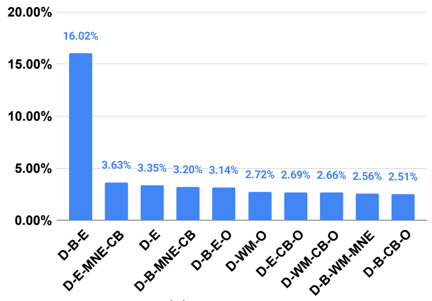
(c) Gson



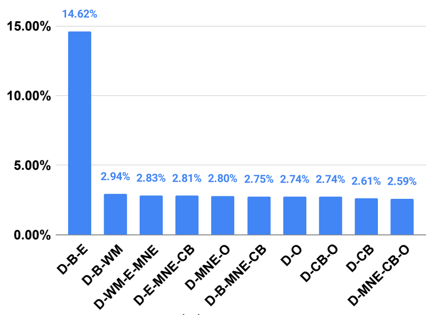
(d) Lang



(e) Math



(f) Mockito



(g) Time

Fig. 10: Top ten fitness function combinations chosen by **UCB** for the diversity goal. D = Diversity Score, B = Branch, CB = Direct Branch, O = Output, M = Method, MNE = Method (No Exception), WM = Weak Mutation

Table 9: Percentage of Strong Mutation Coverage attained when all approaches execute for 10 minutes. **Higher values are better**. The highest median is **bolded**.

	DSG-Sarsa	**UCB**	**Default**	**Strong Mutation**
Chart	47.00	47.00	52.00	**54.00**
Closure	16.00	16.00	18.00	**19.00**
Lang	61.00	60.00	62.00	**63.00**
Math	73.00	**74.00**	73.00	73.00
Mockito	**12.00**	8.50	11.00	10.00
Time	67.00	66.00	67.00	**68.00**
Overall	38.00	39.00	**40.00**	**40.00**

Each function is insufficient on its own. The diversity score needs external feedback 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 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 (and other combinations) act in concert, providing feedback to the other functions.

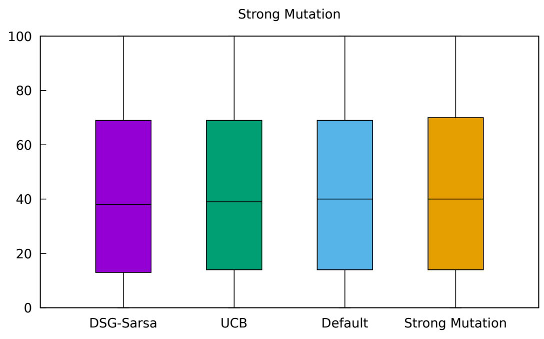
We again see that UCB tends to exploit one combination above all others, while DSG-Sarsa will spend more time exploring different options. As UCB attains better results, it may be that heavier exploitation is a good idea for this goal. A greater tendency towards exploitation may enable better goal attainment, 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 attainment of our third goal using the attained percentage of Strong Mutation Coverage. In Table 9, we note the median Strong Mutation Coverage



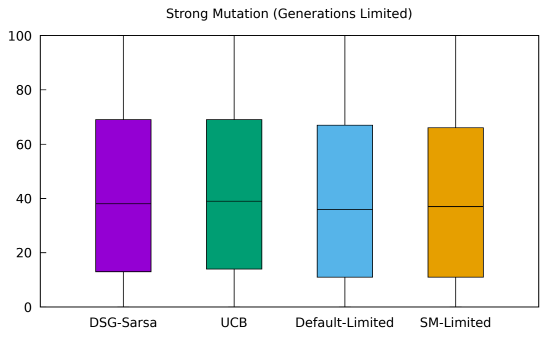


Fig. 11: Strong Mutation Coverage attained by final test suites when (top) all approaches run for 10 minutes, and (bottom), when all approaches are fixed to the number of generations of evolution completed by DSG-Sarsa in 10 minutes.

for each AFFS technique and baseline configuration[8] for each system and overall, bolding the approach with the highest median result. In the top diagram in Figure 11, we show a boxplot of the Strong Mutation results for all approaches.

From Table 9, we see that, for four of the six projects, *both AFFS approaches and the default configuration attain worse median results than simply optimizing Strong Mutation Coverage directly.* From Figure 11, we can see that all four approaches yield very similar boxplots, with Strong Mutation having a slightly higher third-quartile than the other approaches. 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 Friedman test and Vargha Delaney A measure. We formulate hypothesis and null hypothesis:

- H: Generated test suites have different distributions of Strong Mutation Coverage results depending on the technique used to generate the suite.
- $H0$: Observations of Strong Mutation Coverage for all techniques are drawn from the same distribution.

[8] We omit the random baseline for this experiment to (a) control the cost of running experiments, and (b), because of the similarity of results for the AFFS approaches and the other two baselines. It is likely that the random baseline would attain similar results as well.

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

	DSG-Sarsa	**UCB**	**Default**	**Strong Mutation**
DSG-Sarsa	-	0.50	0.49	0.49
UCB	0.50	-	0.49	0.49
Default	0.51	0.51	-	0.50
Strong Mutation	0.51	0.51	0.50	-

Table 11: Median time per generation (in seconds) for the Strong Mutation Coverage goal. The fastest technique is **bolded**.

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

The Friedman test, at p-value < 0.001, suggests differences in the distributions of results for the techniques. In Table 10, 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, we cannot say that any technique outperforms AFFS with significance.

Optimizing Strong Mutation alone or the default baseline yields median improvement of 2.56%/5.26% over UCB/DSG-Sarsa. However, no technique demonstrates significant performance differences (negligible effect sizes).

To explain the lack of success of AFFS for this goal, we examine two factors: (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 examine the impact of the overhead of reinforcement learning, particularly calculation of the reward function. In this experiment, a search budget of ten minutes is allocated to generate test suites. The amount of time that a single generation takes is not fixed, but depends on 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. 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 due to feedback from other fitness functions.

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,

Table 12: 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**. The highest median is **bolded**.

	DSG-Sarsa	**UCB**	**Default**	**Strong Mutation**
Chart	47.00	47.00	**49.50**	44.50
Closure	16.00	16.00	**17.00**	16.00
Lang	**61.00**	60.00	57.00	58.00
Math	73.00	**74.00**	70.00	71.00
Mockito	**12.00**	8.50	9.00	8.00
Time	**67.00**	66.00	64.00	64.00
Overall	38.00	**39.00**	36.00	37.00

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 11, we display the median time per generation for each approach. AFFS is again slightly faster than the baselines on average. However, the results vary by project. *For three projects, the Strong Mutation baseline is significantly faster than AFFS.* As seen in Table 9, 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, AFFS techniques are up to 49.94% slower per generation than optimizing Strong Mutation alone. For these projects, optimizing Strong Mutation alone also results in improved goal attainment.

To investigate this possibility, we repeated our experiment, 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 rather than a fixed period of time. In Table 12, we indicate the median goal attainment for each technique when the number of generations of evolution is fixed. In the bottom diagram in Figure 11, we show box plots of results for all techniques.

For all systems, AFFS now attains equal or higher median goal attainment than the Strong Mutation baseline and, for four of the six systems, outperforms the default baseline. Overall, AFFS outperforms both baselines in median performance when the number of generations is fixed. The best technique, UCB, outperforms the default baseline in median performance by 8.33% and the Strong Mutation baseline by 5.41%. The box plots are still similar, but show a slight shift, with DSG-Sarsa and UCB now yielding higher third-quartile and median values than the two baselines.

We repeat our statistical tests as well. The effect sizes are shown in Table 13. While the effect sizes now show slight improvements from AFFS over the default and Strong Mutation baselines, the effect sizes are still negligible. There is some indication that, if more time can be allocated to the generation process, AFFS can

Table 13: Results of Vargha-Delaney A Measure for Strong Mutation (number of generations fixed). No large positive effect sizes were observed.

	DSG-Sarsa	**UCB**	**Default**	**Strong Mutation**
DSG-Sarsa	-	0.50	0.52	0.52
UCB	0.50	-	0.52	0.52
Default	0.48	0.48	-	0.50
Strong Mutation	0.48	0.48	0.50	-

Table 14: Percentage of faults detected by each approach for the Strong Mutation goal. F#G = fixed number of generations

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

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 in median performance. UCB outperforms the default by 8.33% and Strong Mutation alone by 5.41%. However, effect sizes still remain negligible.

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.

If the number of generations are fixed, AFFS may be able to offer mild improvements over the default combination or targeting Strong Mutation on its own, but these improvements are very limited. The similarity of the boxplots in Figure 11 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. Improvement in attainment of Strong Mutation coverage requires further experimentation and discovery of new fitness functions.

The similar performance of all techniques may indicate that the fitness functions considered by AFFS have limited impact on attainment of Strong Mutation Coverage. Other unknown functions may be more effective.

6.3.2 Fault Detection Effectiveness

We examine fault detection for our two AFFS approaches and for the two benchmarks in Table 14, 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. Here, we see fairly similar fault detection rates for the four approaches as well. The top approach was UCB, with 36% of the faults. It outperforms both baseline by 12.50%, and DSG-Sarsa by 16.12%.

We again calculated the point-biserial correlation coefficient between Strong Mutation Coverage and fault detection, resulting in a coefficient of 0.31. This was the strongest correlation of our three goals to fault detection, but is still only a weak correlation. Higher Strong Mutation Coverage has a positive impact on the likelihood of fault detection, but is not—in itself—enough to ensure success.

DSG-Sarsa was outperformed by both baselines, and was also the technique with the lowest median Strong Mutation coverage. However, (a) as no significant differences were observed between result distributions for the approaches, and (b), the weak correlation result, lower Strong Mutation Coverage does not explain its slightly weaker performance. We cannot state that AFFS will always result in improved fault detection over static fitness function choices for this goal.

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 38.46% and Strong Mutation by 50.00%. 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 to develop suites, AFFS may yield a higher likelihood of fault detection as well.

UCB detects 12.50% more faults than both baselines and 16.12% more than DSG-Sarsa for the Strong Mutation goal. When the number of generations is fixed, both AFFS approaches outperform the baselines by up to 50.00%.

6.3.3 Actions Selected by AFFS

In Figure 12, we show the top fitness function combinations chosen by DSG-Sarsa for the Strong Mutation Coverage goal. In Figure 13, we do the same for UCB. For the first two goals, we saw that UCB more heavily favored exploitation of a 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

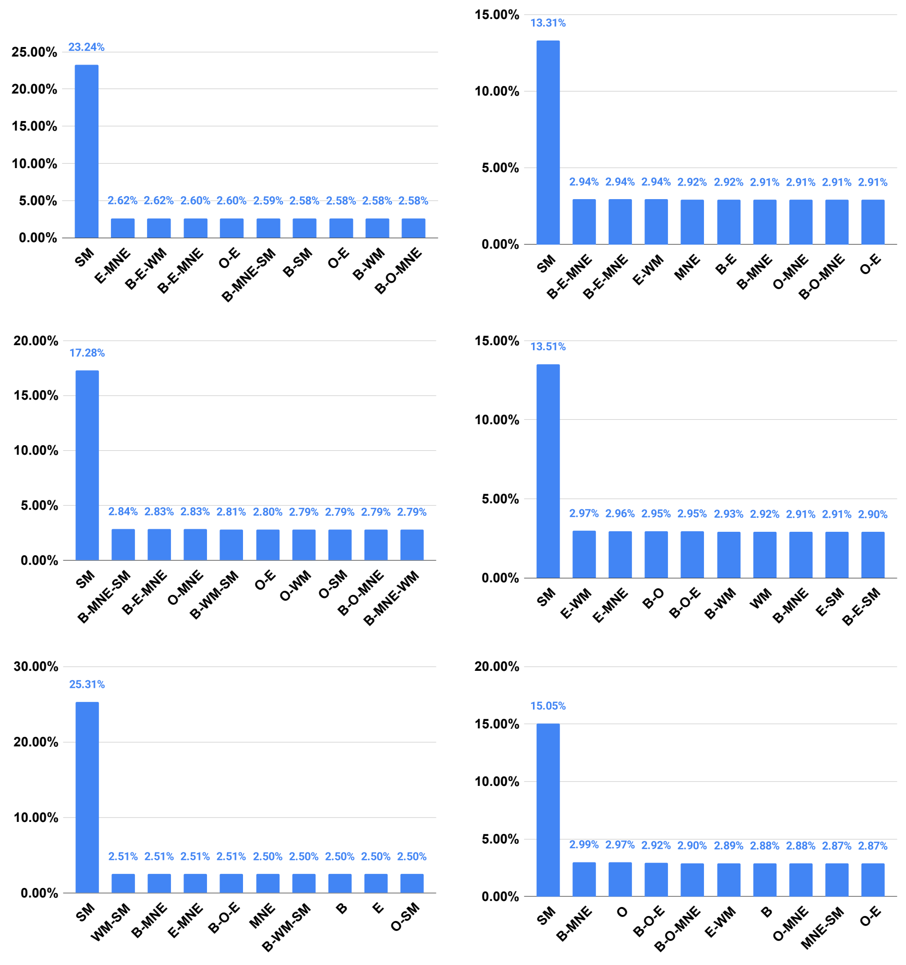


Fig. 12: Top ten fitness function combinations chosen by **DSG-Sarsa** approach for the Strong Mutation goal. SM = Strong Mutation Coverage, B = Branch Coverage, O = Output Coverage, MNE = Method (No Exception), WM = Weak Mutation Coverage, E = Exception Count

attaining relatively low levels of coverage, is still one of the most common optimization targets chosen by AFFS.

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.

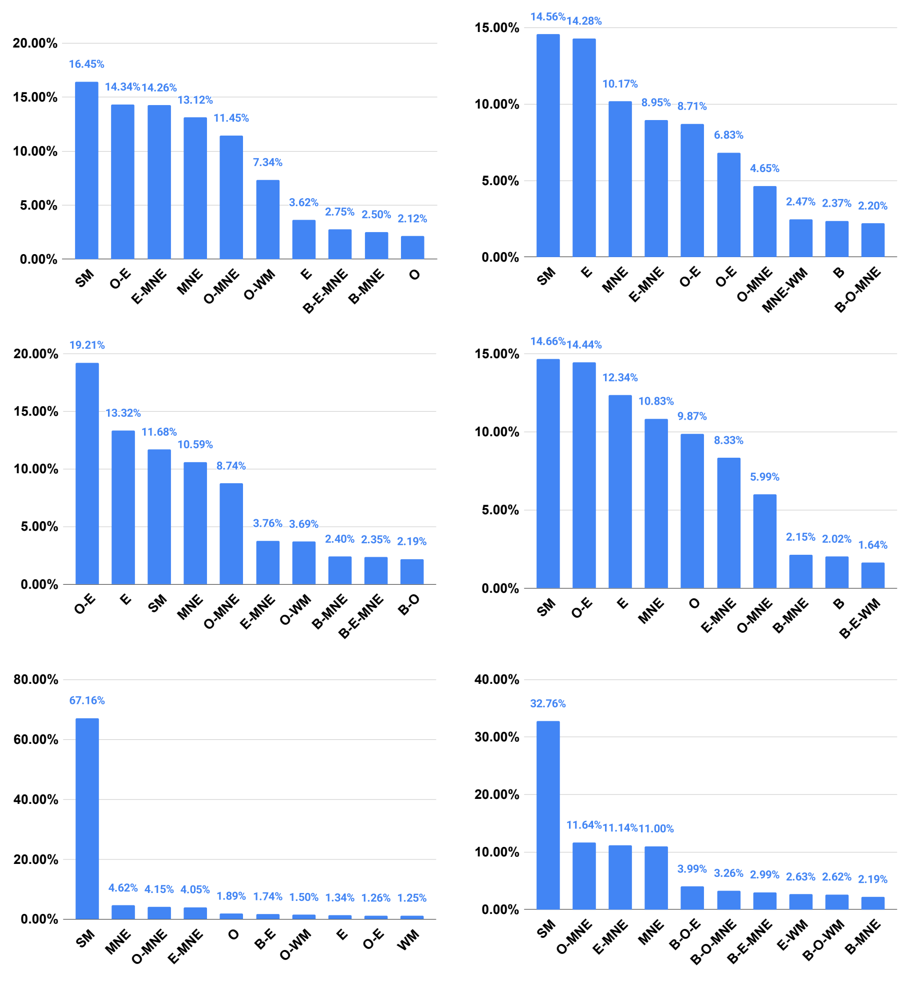


Fig. 13: Top ten fitness function combinations chosen by **UCB** approach for the Strong Mutation goal. SM = Strong Mutation Coverage, B = Branch Coverage, O = Output Coverage, MNE = Method No Exception, WM = Weak Mutation Coverage, E = Exception Count

Choices made by AFFS approaches suggest that no fitness functions 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.

7 Discussion

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

7.1 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 at least medium effect size. Both EvoSuiteFIT techniques discover and retain more exception-triggering input than the baseline techniques, with DSG-Sarsa yielding better results. Additionally, both EvoSuiteFIT techniques produce more diverse test suites than static fitness function choices, with UCB outperforming DSG-Sarsa.

For the goal of Strong Mutation Coverage, no technique demonstrates statistically significant improvements. When the search budget is a fixed number of generations rather than time, both AFFS techniques slightly outperformed the baselines in medium performance, but effect sizes remain negligible. Given some additional time for test generation, we see some potential for improvement from using AFFS over static approaches. However, these improvements were limited.

For the first two goals in particular, 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 Section 1, we gave 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.[9] It is unlikely that AFFS would offer improved goal attainment, as other fitness functions are unlikely to offer additional feedback sufficient to overcome the introduced overhead of reinforcement learning. Rather, AFFS can help improve goal attainment in situations where the existing fitness function offers no or little feedback to improve fitness, like the exception count. AFFS enables the discovery of more exceptions by guiding test generation towards, for example, exploration of CUT structure.

[9] As an aside, it is possible that AFFS could improve Branch Coverage in situations where the branch distance is not informative enough to guide the search, and some learned combination of fitness functions would help. We conducted a small pilot study to examine this question. In a small number of situations, AFFS did attain higher coverage than targeting Branch Coverage directly. However, in the majority of cases, targeting Branch Coverage directly yielded better results due to the overhead of reinforcement learning. Hence, our recommendation is to apply AFFS when an informative fitness function does not already exist, or if that function is known not to work well for a particular problem instance.

AFFS can also help in situations where a fitness function offers misleading feedback. Consider the Levenshtein distance used in promoting suite diversity. This fitness function rewards test suites that differ from each other, but does not assist in suggesting *how* to instill this difference. We observed that targeting this 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.

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 Strong Mutation alone due to the overhead of calculating Strong Mutation Coverage as part of both fitness and reward. 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 functions available to choose from. AFFS can only offer feedback to the search if some combination of the functions it can choose from actually offers the missing feedback. This was the case for two of our three goals. For 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 as many options as possible 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 best improve goal attainment. Reinforcement learning will see faster convergence and better results with fewer options to choose from to start. 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 that we knew or suspected would be suboptimal before even starting the experiments. We would recommend a similar process for additional testing goals—*start by pruning functions and combinations that you suspect will produce weak results.*

7.2 Impact of AFFS on Fault Detection

Fault detection is not a simple matter of maximizing some function, but of selecting the exact input that will trigger an observable failure [84]. The likelihood of fault detection is influenced by a number of factors [82]. 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 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. Thus, 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 is able to detect more faults than all other approaches for the Strong Mutation goal, while DSG-Sarsa is outperformed by the baselines. However, when the number of generations is fixed, both AFFS approaches outperform the baselines. For the diversity goal, the random baseline outperformed all other approaches. The AFFS approaches, however, outperformed both of the other baselines.

AFFS approaches can detect more faults than optimizing static baselines. However, this is not guaranteed. Higher goal attainment does not lead always lead to improved fault detection, and can actually mean the opposite if goal attainment does not actually have a positive correlation with the likelihood of fault detection. We do not fully understand the impact of AFFS on fault detection yet, and will examine it more closely in future work. However, we have observed several factors that may lead to a higher likelihood of fault detection.

AFFS results in higher attainment of goals thought to have a positive relationship with 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. We do see this in the exception experiment, but not in the diversity experiment. The calculated correlation coefficients for all three goals do not indicate strong connections between goal attainment and fault detection in this experiment. Still, it is a factor that may contribute to improvements in fault detection.

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 [12,82]. This is not universally the case, and requires careful selection of fitness functions [14]. 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 avoid such conflicts.

Changing fitness functions as the suite evolves may result in better 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 more effective than trying to imbue many properties at once.

7.3 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.

The ability to avoid calculation of unhelpful fitness functions mitigates reinforcement learning overhead. For both 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 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. In such cases, either a less expensive reward function should be used or more time should be allocated to AFFS.

7.4 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 function combinations similarly act in concert to improve suite diversity. AFFS enabled the discovery of these serendipitous combinations.

Similarly, the choices made by AFFS suggest that no fitness function combination provided feedback needed 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 attainment 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.

7.5 Choice of Reinforcement Learning Approach

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 attains higher median performance for the diversity and Strong Mutation goals than DSG-Sarsa, while the reverse is true for the diversity goal. 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 goal.

We lack enough evidence to recommend one approach over the other. UCB attains a slight lead in multiple categories, but is outperformed by DSG-Sarsa in enough cases to rule out an unqualified recommendation. 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.

7.6 When AFFS Harms Goal Attainment

While we observed that AFFS *generally* enables greater attainment of testing goals, there are times where it not only fails to improve attainment—it actively attains worse results than all of the baselines. To gain a greater perspective on the limitations of AFFS, we manually examined situations where either DSG-Sarsa or UCB attained worse results in all, or almost all, trials than the single fitness function baseline.

We focus on the *diversity* goal in this analysis, as it offers the clearest performance differentials between AFFS and the single-function baseline. We identified the ten faults where AFFS techniques attained the worst results relative to the diversity score alone. We examined the classes-under-test and the generated suites, and attempted to identify factors that explain the worse performance of AFFS.

For the diversity goal, the faults where AFFS attained the worst performance relative to the diversity score baseline were, in order, Gson-12, Chart-24, Lang-25, Math-35, Lang-55, Math-34, Closure 39, Math 56, Math 89, and Mockito-12. Examining the classes and tests generated for this fault, we observed two primary factors limiting performance of AFFS—the first being a factor that can affect any goal, and the second being specific to the diversity goal.

Fitness functions are merged into a single score: In EvoSuite's genetic algorithm, all selected fitness functions are merged into a single score. This means that large improvements to a single function will be accepted, even if they come with a small drop in another fitness function. This creates potential conflicts between fitness functions, and allows particular functions to be dominated if they are difficult to optimize in comparison to other functions, or if they do not rise in conjunction with another function.

In some cases, diversity and functions like code coverage may rise in conjunction with each other. For example, increasing diversity may also increase the attained code coverage. In such cases, using code coverage as one of the fitness functions may offer feedback that is not offered by optimizing diversity alone.

In other cases, input diversity may have little impact on the code coverage. For example, only a small set of values may cause control-flow to take different paths, or only a small number of methods might accept a large selection of input values. In these cases, the "best" test suites may be those that cover a large span of the code, even if they lack diversity. The reward function used by AFFS will still encourage some diversity, but its impact may be lessened because the fitness functions employed prioritize large gains in coverage over diversity, and those fitness functions are the ones that ultimately evolve the test suites.

Chart-24 offers an example of this, where the test suites generated by AFFS call a wider variety of methods than those generated targeting diversity alone. However, the latter apply a wider range of input to a smaller set of methods. The suites generated by AFFS may be "better". They cover more of the code, and could detect faults missed by the other suites. At the same time, they are "worse" with regard to the goal of diversity.

This factor can impact the results of AFFS for not just diversity, but for other testing goals as well. The success of AFFS relies on the existence of fitness functions that can improve attainment of our goal of interest. In cases where we can identify those functions, AFFS works well. However, if we cannot identify fitness functions that improve goal attainment, the end result may be worse than just trying to optimize the existing weak fitness function for that goal. In many cases, AFFS was able to identify such functions. However, for some classes, there may be no effective fitness functions for increasing diversity.

Methods with limited or no input: In multiple cases, the CUT had a large number of methods where there were no, or limited, means of interacting through input parameters. Consider, for example:

- **Gson-12:** The CUT parses elements from a Json structure. The only "input" provided is to the constructor. EvoSuite cannot generate arbitrary Json input, so it provides an empty file to the constructor. The other methods of the class interact with this structure, and many have no input parameters.
- **Lang-55:** The CUT is a stopwatch. The constructor initializes the object, and it can be interacted with through methods that stop, pause, and reset the stopwatch. There are no input parameters.
- **Math-34:** The CUT represents the population of a genetic algorithm in list form. The constructor initializes the population, and the methods can be used to analyze or interact with that population. Many methods have no input (e.g., getting the fittest chromosome or getting a list of chromosomes), and their result depends on the contents of the population. The primary means of introducing diversity through input are when controlling the size of the population, i.e., a method with numeric input.

One of the primary means of improving diversity is to provide a wider range of input to method parameters. When methods lack parameters, gaining diversity becomes more difficult. Instead, other means of gaining diversity must be employed, including increasing the diversity of output—the generated assertion statements include the output of calling methods that offer concrete output, generating tests

```
public void test0()  throws Throwable  {
    ElitisticListPopulation elitisticListPopulation0 =
        new ElitisticListPopulation(208, 0.0);
    elitisticListPopulation0.addChromosome((Chromosome) null);
    assertEquals(208, elitisticListPopulation0.getPopulationLimit());
}

public void test2()  throws Throwable  {
    ElitisticListPopulation elitisticListPopulation0 =
        new ElitisticListPopulation(208, 0.0);
    elitisticListPopulation0.getChromosomeList();
    assertEquals(208, elitisticListPopulation0.getPopulationLimit());
}

public void test4()  throws Throwable  {
    LinkedList<Chromosome> linkedList0 = new LinkedList<Chromosome>();
    ElitisticListPopulation elitisticListPopulation0 =
        new ElitisticListPopulation(linkedList0, 1135, 0.0);
    // Undeclared exception!
    try {
      elitisticListPopulation0.setPopulationLimit((-1485));
      fail("Expecting␣exception:␣IllegalArgumentException");

    } catch(IllegalArgumentException e) {
       //
       // population limit has to be positive
       //
       verifyException("org.apache.commons.math3.genetics.ListPopulation",
           e);
    }
}
```

Fig. 14: Two similar test cases generated by DSG-Sarsa for Math-34, followed by one generated when targeting only diversity.

that call highly different sequences of input, and triggering exceptions—as the `try/catch` block included in the test case will give the test a very different body than many other tests. However, it can be difficult to "discover" test cases that exploit these routes to diversity.

In these cases, other fitness functions—i.e., code coverage—will dominate the diversity fitness function. As a result, for these classes, AFFS tends to generate a large number of highly-similar test cases, while targeting diversity-alone yields a small number of test cases that are very different from each other. Consider, for example, the first two test cases in Figure 14. These two tests were among those generated by DSG-Sarsa for the CUT. The test suite contains many of this form, where the first and third lines (set-up, and assertion on the output) are identical. The only difference is the second line, where different methods are called. These two test cases cover different parts of the code, but are not very different in terms of the resulting diversity score. Test like these are added to the suite because they have a small positive impact on diversity, but—more importantly—because they have a large impact on other fitness functions like the code coverage. This impact comes more easily than improvements in diversity, and has a greater impact on the resulting test suite.

In contrast, targeting diversity alone prioritizes test cases like the third one in Figure 14—longer test cases where exceptions are thrown and diversity is introduced through the available method calls with input parameters. As a result, the suites generated by AFFS are less diverse than those generated targeting diversity-only. Again, the suites generated by AFFS may be better for fault detection, but they are technically worse for the stated "goal" of the tester.

8 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 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 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.

9 Conclusions

Search-based test generation is guided by feedback from one or more fitness functions. 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 a set of baselines.

We have found that both EvoSuiteFIT techniques outperform all baselines with at least medium effect size for the goals of exception discovery and suite diversity—attaining improvements of up to 107.14% in goal attainment. For the goal of Strong Mutation Coverage, no technique demonstrates significant improvements. When the search budget is a fixed number of generations rather than time, both EvoSuiteFIT techniques slightly outperform the baselines (up to 8.33% improvement). However, the effect size is still negligible.

Additionally, both EvoSuiteFIT techniques detect faults missed by the other techniques for the exception discovery goal (up to 259.90% improvement). UCB is able to detect more faults for the Strong Mutation goal (12.50% improvement), and when the number of generations is fixed, both EvoSuiteFIT approaches outperform the baselines (up to 50.00% improvement). Both techniques are outperformed by the random baseline for the diversity goal (34.74% difference), but outperform the other baselines. Improvements in fault detection may arise because of higher attainment of these goals, optimizing multiple fitness functions—but avoiding needlessly complex and conflicting functions—and changing fitness functions as the suite evolves. However, higher goal attainment does not ensure fault detection.

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.

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. Additionally, 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.

In future work, we plan apply AFFS to new goals and testing scenarios (e.g., system testing) and integrate it into metaheuristic algorithms beyond standard Genetic Algorithms. We also will perform expanded empirical studies to better understand the relationship between AFFS and fault detection and how the staggered application of fitness functions can improve goal attainment and suite effectiveness. We will also explore the generation of new fitness functions—i.e., a generative hyperheuristic rather than a selective one—and how learned policies can be transferred to new classes and systems. Finally, we will examine the application of AFFS to multiple high-level goals simultaneously.

References

1. Pezze, M., Young, M.: Software Test and Analysis: Process, Principles, and Techniques. John Wiley and Sons (2006)
2. Barr, E., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Transactions on Software Engineering **41**(5), 507–525 (2015). DOI 10.1109/TSE.2014.2372785
3. Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P.: An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software **86**(8), 1978–2001 (2013)
4. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability **14**, 105–156 (2004)
5. Harman, M., Jones, B.: Search-based software engineering. Journal of Information and Software Technology **43**, 833–839 (2001)
6. Salahirad, A., Almulla, H., Gay, G.: Choosing the fitness function for the job: Automated generation of test suites that detect real faults. Software Testing, Verification and Reliability **29**(4-5), e1701 (2019). DOI 10.1002/stvr.1701. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1701`. E1701 stvr.1701
7. Arcuri, A.: It really does matter how you normalize the branch distance in search-based software testing. Software Testing, Verification and Reliability **23**(2), 119–147 (2013)
8. Robillard, M.P., Murphy, G.C.: Designing robust java programs with exceptions. In: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications, SIGSOFT '00/FSE-8, pp. 2–10. ACM, New York, NY, USA (2000). DOI 10.1145/355045.355046. URL `http://doi.acm.org/10.1145/355045.355046`
9. De Oliveira Neto, F.G., Feldt, R., Erlenhov, L., Nunes, J.B.D.S.: Visualizing test diversity to support test optimisation. In: 2018 25th Asia-Pacific Software Engineering Conference (APSEC), pp. 149–158 (2018)
10. Shahbazi, A.: Diversity-based automated test case generation. Ph.D. thesis, University of Alberta (2015)
11. Lindstrom, B., Mrki, A.: On strong mutation and subsuming mutants. In: 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 112–121 (2016)
12. Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A.: Combining multiple coverage criteria in search-based unit test generation. In: M. Barros, Y. Labiche (eds.) Search-Based Software Engineering, *Lecture Notes in Computer Science*, vol. 9275, pp. 93–108. Springer International Publishing (2015). DOI 10.1007/978-3-319-22183-0_7. URL `http://dx.doi.org/10.1007/978-3-319-22183-0_7`
13. Gay, G.: The fitness function for the job: Search-based generation of test suites that detect real faults. In: Proceedings of the International Conference on Software Testing, ICST 2017. IEEE (2017)
14. Gay, G.: Generating effective test suites by combining coverage criteria. In: Proceedings of the Symposium on Search-Based Software Engineering, SSBSE 2017. Springer Verlag (2017)
15. Fraser, G., Arcuri, A.: Achieving scalable mutation-based generation of whole test suites. Empirical Software Engineering **20**(3), 783–812 (2014)
16. Papadakis, M., Malevris, N.: Searching and generating test inputs for mutation testing. SpringerPlus **2**(1), 121 (2013). DOI 10.1186/2193-1801-2-121. URL `https://doi.org/10.1186/2193-1801-2-121`

17. Jia, Y., Cohen, M.B., Harman, M., Petke, J.: Learning combinatorial interaction test generation strategies using hyperheuristic search. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, pp. 540–550. IEEE Press, Piscataway, NJ, USA (2015). URL `http://dl.acm.org/citation.cfm?id=2818754.2818821`
18. Guizzo, G., Fritsche, G.M., Vergilio, S.R., Pozo, A.T.R.: A hyper-heuristic for the multi-objective integration and test order problem. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO '15, pp. 1343–1350. ACM, New York, NY, USA (2015). DOI 10.1145/2739480.2754725. URL `http://doi.acm.org/10.1145/2739480.2754725`
19. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
20. Rojas, J.M., Vivanti, M., Arcuri, A., Fraser, G.: A detailed investigation of the effectiveness of whole test suite generation. Empirical Software Engineering **22**(2), 852–893 (2017). DOI 10.1007/s10664-015-9424-2. URL `https://doi.org/10.1007/s10664-015-9424-2`
21. Almulla, H., Gay, G.: Learning how to search: Generating exception-triggering tests through adaptive fitness function selection. In: 13th IEEE International Conference on Software Testing, Validation and Verification (2020)
22. Almulla, H., Gay, G.: Generating diverse test suites for Gson through adaptive fitness function selection. In: Proceedings of the Symposium on Search-Based Software Engineering, SSBSE 2020. Springer Verlag (2020)
23. Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMinn, P., Arcuri, A.: Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), ASE 2015. ACM, New York, NY, USA (2015)
24. Orso, A., Rothermel, G.: Software testing: A research travelogue (2000–2014). In: Proceedings of the on Future of Software Engineering, FOSE 2014, pp. 117–132. ACM, New York, NY, USA (2014). DOI 10.1145/2593882.2593885. URL `http://doi.acm.org/10.1145/2593882.2593885`
25. Almasi, M.M., Hemmati, H., Fraser, G., Arcuri, A., Benefelds, J.: An industrial evaluation of unit test generation: Finding real faults in a financial application. In: Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)—Software Engineering in Practice Track (SEIP), ICSE 2017. ACM, New York, NY, USA (2017)
26. Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A systematic review of the application and empirical investigation of search-based test case generation. Software Engineering, IEEE Transactions on **36**(6), 742–762 (2010)
27. Bianchi, L., Dorigo, M., Gambardella, L., Gutjahr, W.: A survey on metaheuristics for stochastic combinatorial optimization. Natural Computing **8**(2), 239–287 (2009). DOI 10.1007/s11047-008-9098-4. URL `http://dx.doi.org/10.1007/s11047-008-9098-4`
28. Dorigo, M., Gambardella, L.M.: Ant colony system: a cooperative learning approach to the traveling salesman problem. Evolutionary Computation, IEEE Transactions on **1**(1), 53–66 (1997)
29. Holland, J.H.: Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. MIT press (1992)
30. Feldt, R., Poulding, S.: Broadening the search in search-based software testing: It need not be evolutionary. In: Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th International Workshop on, pp. 1–7 (2015). DOI 10.1109/SBST.2015.8
31. Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated white-box test generation really help software testers? In: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA, pp. 291–301. ACM, New York, NY, USA (2013). DOI 10.1145/2483760.2483774. URL `http://doi.acm.org/10.1145/2483760.2483774`
32. Malburg, J., Fraser, G.: Combining search-based and constraint-based testing. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, pp. 436–439. IEEE Computer Society, Washington, DC, USA (2011). DOI 10.1109/ASE.2011.6100092. URL `http://dx.doi.org/10.1109/ASE.2011.6100092`
33. Balera, J.M., de Santiago Júnior, V.A.: A systematic mapping addressing hyper-heuristics within search-based software testing. Information and Software Technology **114**, 176–189 (2019)
34. Drake, J.H., Kheiri, A., Özcan, E., Burke, E.K.: Recent advances in selection hyper-heuristics. European Journal of Operational Research **285**(2), 405–428 (2020)
35. Kumari, A.C., Srinivas, K.: Hyper-heuristic approach for multi-objective software module clustering. Journal of Systems and Software **117**, 384 – 401 (2016). DOI https://doi.org/

10.1016/j.jss.2016.04.007. URL http://www.sciencedirect.com/science/article/pii/S0164121216300231
36. Burke, E.K., Hyde, M.R., Kendall, G., Ochoa, G., Özcan, E., Woodward, J.R.: A Classification of Hyper-Heuristic Approaches: Revisited, pp. 453–477. Springer International Publishing, Cham (2019). DOI 10.1007/978-3-319-91086-4_14. URL https://doi.org/10.1007/978-3-319-91086-4_14
37. Katehakis, M.N., Veinott Jr, A.F.: The multi-armed bandit problem: decomposition and computation. Mathematics of Operations Research **12**(2), 262–268 (1987)
38. Jia, Y.: Hyperheuristic search for sbst. In: Proceedings of the Eighth International Workshop on Search-Based Software Testing, SBST '15, pp. 15–16. IEEE Press, Piscataway, NJ, USA (2015). URL http://dl.acm.org/citation.cfm?id=2821339.2821343
39. Gay, G., Whalen, M.W., Heimdahl, M.P., Staats, M.: The risks of coverage directed test case generation. Currently under submission, draft available from http://greggay.com/pdf/14risks.pdf (2014)
40. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction, vol. 1. MIT press Cambridge (1998)
41. Buşoniu, L., Ernst, D., De Schutter, B., Babuška, R.: Approximate reinforcement learning: An overview. In: 2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL), pp. 1–8 (2011). DOI 10.1109/ADPRL.2011.5967353
42. Fraser, G., Arcuri, A.: Evosuite at the sbst 2017 tool competition. In: 10th International Workshop on Search-Based Software Testing (SBST'17) at ICSE'17, pp. 39–42 (2017)
43. Fraser, G.: A tutorial on using and extending the evosuite search-based test generator. In: Search-Based Software Engineering, pp. 106–130. Springer (2018)
44. Gay, G.: To call, or not to call: Contrasting direct and indirect branch coverage in test generation. In: Proceedings of the 11th International Workshop on Search-Based Software Testing, SBST 2018. ACM, New York, NY, USA (2018)
45. Alshahwan, N., Harman, M.: Coverage and fault detection of the output-uniqueness test selection criteria. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, pp. 181–192. ACM, New York, NY, USA (2014). DOI 10.1145/2610384.2610413. URL http://doi.acm.org/10.1145/2610384.2610413
46. Iqbal, M.S., Kotthoff, L., Jamshidi, P.: Transfer learning for performance modeling of deep neural network systems. In: 2019 {USENIX} Conference on Operational Machine Learning (OpML 19), pp. 43–46 (2019)
47. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. **33**(1), 31–88 (2001). DOI 10.1145/375360.375365. URL https://doi.org/10.1145/375360.375365
48. Crawford, B., Soto, R., Monfroy, E., Palma, W., Castro, C., Paredes, F.: Parameter tuning of a choice-function based hyperheuristic using particle swarm optimization. Expert Systems with Applications **40**(5), 1690 – 1695 (2013). DOI https://doi.org/10.1016/j.eswa.2012.09.013. URL http://www.sciencedirect.com/science/article/pii/S0957417412010676
49. Ochoa, G., Vazquez-Rodriguez, J.A., Petrovic, S., Burke, E.: Dispatching rules for production scheduling: A hyper-heuristic landscape analysis. In: 2009 IEEE Congress on Evolutionary Computation, pp. 1873–1880 (2009). DOI 10.1109/CEC.2009.4983169
50. Zamli, K.Z., Alkazemi, B.Y., Kendall, G.: A tabu search hyper-heuristic strategy for t-way test suite generation. Appl. Soft Comput. **44**(C), 57–74 (2016). DOI 10.1016/j.asoc.2016.03.021. URL https://doi.org/10.1016/j.asoc.2016.03.021
51. Zamli, K.Z., Din, F., Kendall, G., Ahmed, B.S.: An experimental study of hyper-heuristic selection and acceptance mechanism for combinatorial t-way test suite generation. Information Sciences **399**, 121 – 153 (2017). DOI https://doi.org/10.1016/j.ins.2017.03.007. URL http://www.sciencedirect.com/science/article/pii/S0020025517305820
52. Din, F., Alsewari, A.R.A., Zamli, K.Z.: A parameter free choice function based hyper-heuristic strategy for pairwise test generation. In: 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 85–91 (2017)
53. Din, F., Zamli, K.Z.: Hyper-heuristic based strategy for pairwise test case generation. Advanced Science Letters **24**(10), 7333–7338 (2018). DOI doi:10.1166/asl.2018.12938. URL https://www.ingentaconnect.com/content/asp/asl/2018/00000024/00000010/art00068
54. Ahmed, B.S., Enoiu, E., Afzal, W., Zamli, K.Z.: An evaluation of monte carlo-based hyper-heuristic for interaction testing of industrial embedded software applications. Soft Computing (2020). DOI 10.1007/s00500-020-04769-z. URL http://dx.doi.org/10.1007/s00500-020-04769-z

55. Guizzo, G., Vergilio, S.R., Pozo, A.T.R.: Evaluating a multi-objective hyper-heuristic for the integration and test order problem. In: 2015 Brazilian Conference on Intelligent Systems (BRACIS), pp. 1–6 (2015). DOI 10.1109/BRACIS.2015.11
56. Guizzo, G., Vergilio, S.R., Pozo, A.T., Fritsche, G.M.: A multi-objective and evolutionary hyper-heuristic applied to the integration and test order problem. Appl. Soft Comput. **56**(C), 331–344 (2017). DOI 10.1016/j.asoc.2017.03.012. URL `https://doi.org/10.1016/j.asoc.2017.03.012`
57. Guizzo, G., Bazargani, M., Paixão, M., Drake, J.H.: A hyper-heuristic for multi-objective integration and test ordering in google guava. In: SSBSE (2017)
58. Mariani, T., Guizzo, G., Vergilio, S.R., Pozo, A.T.: Grammatical evolution for the multi-objective integration and test order problem. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016, GECCO '16, p. 1069–1076. Association for Computing Machinery, New York, NY, USA (2016). DOI 10.1145/2908812.2908816. URL `https://doi.org/10.1145/2908812.2908816`
59. Guizzo, G., Vergilio, S.R.: A pattern-driven solution for designing multi-objective evolutionary algorithms. Natural Computing pp. 1–14 (2018)
60. Ferreira, T.N., Lima, J.A.P., Strickler, A., Kuk, J.N., Vergilio, S.R., Pozo, A.: Hyper-heuristic based product selection for software product line testing. IEEE Computational Intelligence Magazine **12**(2), 34–45 (2017)
61. do Nascimento Ferreira, T., Kuk, J.N., Pozo, A., Vergilio, S.R.: Product selection based on upper confidence bound moea/d-dra for testing software product lines. In: 2016 IEEE Congress on Evolutionary Computation (CEC), pp. 4135–4142 (2016)
62. Strickler, A., Prado Lima, J.A., Vergilio, S.R., Pozo, A.T.: Deriving products for variability test of feature models with a hyper-heuristic approach. Applied Soft Computing **49**, 1232 – 1242 (2016). DOI https://doi.org/10.1016/j.asoc.2016.07.059. URL `http://www.sciencedirect.com/science/article/pii/S1568494616303994`
63. Filho, H.L.J., Lima, J.A.P., Vergilio, S.R.: Automatic generation of search-based algorithms applied to the feature testing of software product lines. In: Proceedings of the 31st Brazilian Symposium on Software Engineering, SBES'17, p. 114–123. Association for Computing Machinery, New York, NY, USA (2017). DOI 10.1145/3131151.3131152. URL `https://doi.org/10.1145/3131151.3131152`
64. Filho, H.L.J., Ferreira, T.N., Vergilio, S.R.: Multiple objective test set selection for software product line testing: Evaluating different preference-based algorithms. In: Proceedings of the XXXII Brazilian Symposium on Software Engineering, SBES '18, p. 162–171. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3266237.3266275. URL `https://doi.org/10.1145/3266237.3266275`
65. Luiz Jakubovski Filho, H., Nascimento Ferreira, T., Regina Vergilio, S.: Incorporating user preferences in a software product line testing hyper-heuristic approach. In: 2018 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8 (2018)
66. Helali Moghadam, M., Saadatmand, M., Borg, M., Bohlin, M., Lisper, B.: Machine learning to guide performance testing: An autonomous test framework. In: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 164–167 (2019)
67. Bauersfeld, S., Vos, T.: A reinforcement learning approach to automated gui robustness testing. In: Fast abstracts of the 4th symposium on search-based software engineering (SSBSE 2012), pp. 7–12 (2012)
68. Bauersfeld, S., Vos, T.E.J.: Guitest: a java library for fully automated gui robustness testing. In: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 330–333 (2012). DOI 10.1145/2351676.2351739
69. Grechanik, M., Fu, C., Xie, Q.: Automatically finding performance problems with feedback-directed learning software testing. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 156–166 (2012)
70. Joffe, L., Clark, D.: Directing a search towards execution properties with a learned fitness function. IEEE (2019)
71. Romano, D., Penta, M.D., Antoniol, G.: An approach for search based testing of null pointer exceptions. 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation pp. 160–169 (2011)
72. Goffi, A., Gorla, A., Ernst, M.D., Pezzè, M.: Automatic generation of oracles for exceptional behaviors. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, p. 213–224. Association for Computing Machinery, New York, NY, USA (2016). DOI 10.1145/2931037.2931061. URL `https://doi.org/10.1145/2931037.2931061`

73. Blasi, A., Goffi, A., Kuznetsov, K., Gorla, A., Ernst, M.D., Pezzè, M., Castellanos, S.D.: Translating code comments to procedure specifications. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, p. 242–253. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3213846.3213872. URL `https://doi.org/10.1145/3213846.3213872`
74. Albunian, N.M.: Diversity in search-based unit test suite generation. In: T. Menzies, J. Petke (eds.) Search Based Software Engineering, pp. 183–189. Springer International Publishing, Cham (2017)
75. Feldt, R., Poulding, S., Clark, D., Yoo, S.: Test set diameter: Quantifying the diversity of sets of test cases. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 223–233 (2016)
76. Ma, L., Wu, P., Chen, T.Y.: Diversity driven adaptive test generation for concurrent data structures. Information and Software Technology **103**, 162 – 173 (2018). DOI https://doi.org/10.1016/j.infsof.2018.07.001. URL `http://www.sciencedirect.com/science/article/pii/S0950584918301356`
77. Vogel, T., Tran, C., Grunske, L.: Does diversity improve the test suite generation for mobile applications? In: International Symposium on Search Based Software Engineering, pp. 58–74. Springer (2019)
78. Souza, F.C.M., Papadakis, M., Le Traon, Y., Delamaro, M.E.: Strong mutation-based test data generation using hill climbing. In: Proceedings of the 9th International Workshop on Search-Based Software Testing, pp. 45–54. ACM (2016)
79. Xu, D., Shrestha, R., Shen, N.: Automated strong mutation testing of xacml policies. In: Proceedings of the 25th ACM Symposium on Access Control Models and Technologies, SACMAT '20, p. 105–116. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3381991.3395599. URL `https://doi.org/10.1145/3381991.3395599`
80. Harman, M., Jia, Y., Langdon, W.B.: Strong higher order mutation-based test data generation. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, p. 212–222. Association for Computing Machinery, New York, NY, USA (2011). DOI 10.1145/2025113.2025144. URL `https://doi.org/10.1145/2025113.2025144`
81. Just, R., Jalali, D., Ernst, M.D.: Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, pp. 437–440. ACM, New York, NY, USA (2014). DOI 10.1145/2610384.2628055. URL `http://doi.acm.org/10.1145/2610384.2628055`
82. Gay, G.: Choosing the fitness function for the job: Automated generation of test suites that detect real faults. Under revision, Journal of Software Testing, Verification, and Reliability **X**(Y), 1–20 (2018). Draft available from `http://greggay.com/pdf/18fitness.pdf`
83. Vargha, A., Delaney, H.D.: A critique and improvement of the cl common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics **25**(2), 101–132 (2000). DOI 10.3102/10769986025002101. URL `https://doi.org/10.3102/10769986025002101`
84. Gay, G., Staats, M., Whalen, M., Heimdahl, M.: The risks of coverage-directed test case generation. Software Engineering, IEEE Transactions on **PP**(99) (2015). DOI 10.1109/TSE.2015.2421011