Generating Diverse Test Suites for Gson Through Adaptive Fitness Function Selection

Downloaded from: https://research.chalmers.se, 2026-04-04 17:29 UTC

Citation for the original published paper (version of record):

Almulla, H., Gay, G. (2020). Generating Diverse Test Suites for Gson Through Adaptive Fitness Function Selection. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), SSBSE 2020: 246-252.
http://dx.doi.org/10.1007/978-3-030-59762-7_18

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Generating Diverse Test Suites for Gson Through Adaptive Fitness Function Selection

Hussein Almulla[1] and Gregory Gay[2]

[1] University of South Carolina, Columbia, SC, USA, halmulla@email.sc.edu
[2] Chalmers and the University of Gothenburg, Gothenburg, SE, greg@greggay.com

Abstract. Many fitness functions—such as those targeting test suite diversity—do not yield sufficient feedback to drive test generation. We propose that diversity can instead be improved through *adaptive fitness function selection* (AFFS), an approach that varies the fitness functions used throughout the generation process in order to strategically increase diversity. We have evaluated our AFFS framework, EvoSuiteFIT, on a set of 18 real faults from Gson, a JSON (de)serialization library. Ultimately, we find that AFFS creates test suites that are more diverse than those created using static fitness functions. We also observe that increased diversity may lead to small improvements in the likelihood of fault detection.

Keywords: Search-Based Test Generation, Fitness Function, Reinforcement Learning

1 Introduction

In search-based test generation, testers seek input that attains their testing *goal*. An optimization algorithm systematically samples the space of possible test input in search of a solution to that goal, guided by feedback from one or more **fitness functions**—numeric scoring functions that judge the optimality of the chosen input [8].

During this search, the fitness functions embody the high-level goals of the tester [1]. Feedback from the fitness function will shape the resulting test suite, and we choose fitness functions based on their suitability for those goals. For example, a common goal is to attain Branch Coverage—to ensure that all outcomes of control-diverging statements have been executed. Search-based test generation generally attains this goal using a fitness function based on the branch distance—a fine-grained measurement of how close we came to covering each branch outcome. If our goal is Branch Coverage, we know how to attain it through search-based test generation. For other goals, we may not be so fortunate. Many goals *do not* have an effective fitness function formulation.

One such goal is *test suite diversity*. When testing, it is generally impossible to try every input. It follows, then, that different test cases are more effective than similar ones [2,9]. This intuition has led to effective automated test generation, prioritization, and reduction [2]. While numerous diversity metrics exist—for example, the *Levenshtein distance* [9]—these metrics serve as poor fitness functions, as they may no present sufficient actionable feedback to optimize.

This does *not* mean that test suite diversity cannot be attained. Rather, we do not *yet* know what fitness functions will be effective. There are many fitness functions used in search-based test generation for attainment of other goals. Careful selection of one or more of *those* functions—paired with, or even excluding, a diversity metric—could provide that missing feedback. In fact, we may even attain higher diversity by reevaluating our choice of fitness functions over time based on the test suite population.

We previously introduced *adaptive fitness function selection* (AFFS)—a hyperheuristic that adapts the test generating process, using reinforcement learning [7], to adjust the fitness functions in service of optimizing attainment of a higher-level goal [1]. We used AFFS to increase the number of exceptions thrown by test suites [1]. In this study, we extend our AFFS framework, EvoSuiteFIT, to a new goal—increasing test suite diversity. We have implemented the Levenshtein distance as both a fitness function and as a target for reinforcement learning in EvoSuiteFIT, and made improvements to the two implemented RL approaches, UCB and DSG-Sarsa [7].

As a case study to evaluate the ability of AFFS to increase test suite diversity, we perform a case study on the Gson library[1]. Gson is an open-source library for serializing and deserializing JSON input, and is an essential tool of Java and Android development [5]. A set of 18 real-world faults from Gson are available in the Defects4J fault database [3]. Previous work has found that this framework is a challenging target for test generation [3]. We evaluate EvoSuiteFIT on these examples in terms of attained test suite diversity and fault detection, comparing against two static baselines.

Ultimately, we find that EvoSuiteFIT creates test suites that are more diverse than those created using static fitness functions. While results on fault detection are inconclusive, we observe that diversity may lead to small improvements in the likelihood of fault detection. We make EvoSuiteFIT available for use in research and practice.

2 Adaptive Fitness Function Selection

Careful selection and reevaluation of fitness functions could result in test suites that are more diverse than those generated targeting a single diversity metric alone or a naively chosen set of fitness functions. Identifying this set of fitness functions is a secondary search problem—one that can be tackled as an additional step within the normal test generation process using reinforcement learning (RL). This process is known as *adaptive fitness function selection* (AFFS) [1].

Due to space constraints, we present only a brief overview of AFFS[2]. Adjusting the set of fitness functions can be considered as an instance of the n-armed bandit problem [7]. Given a measurement of diversity, each action—a choice of fitness functions—has an expected reward—increase in diversity—when it is selected. The modified test generation process is illustrated in Figure 1. At a defined interval, the RL agent will reevaluate the set of fitness functions and refine its estimation of their ability to increase diversity. Because the population of test suites at round N depends on the population from round $N-1$, RL can not only choose effective fitness functions, but can strategically adjust that choice based on the test suite state.

AFFS has been implemented within the standard Genetic Algorithm in the EvoSuite test generation framework [8]. Two RL algorithms—Upper Confidence Bound (UCB) and Differential Semi-Gradient Sarsa (DSG-Sarsa) [7]—have been implemented. UCB is a classic reinforcement learning approach that works well for many problems, while DSG-Sarsa is an approximate approach adapted for problems with large state spaces [7]—i.e., test generation. We refer to the overall framework as **EvoSuiteFIT**[3].

[1] `https://github.com/google/gson`
[2] For full details, refer to our prior work [1].
[3] EvoSuiteFIT is available from `https://github.com/HusseinAlmulla/evosuite/tree/evosuitefit`.

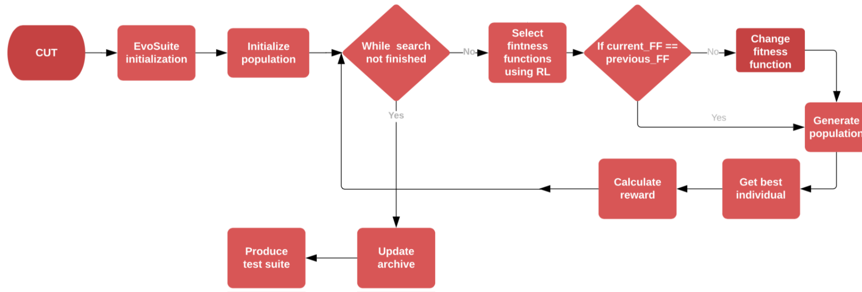


Fig. 1: Overview of the test generation process with AFFS.

Diversity-Based Fitness Function: We have implemented a fitness function to measure test suite diversity based on the Levenshtein distance [9]. 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). The distance between two tests (ta and tb) can be calculated as follows [9]:

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

where i and j are the letters of the strings representing t_a and t_b. To calculate the diversity of a test suite (TS), we calculate the sum of the Levenshtein distance between each pair of test cases: $div(TS) = \sum_{ta,tb}^{TS} lev_{ta,tb}$. 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 $\frac{1}{1+div(TS)}$.

RL Implementation in EvoSuiteFIT: Every five generations, EvoSuiteFIT will select one to four fitness functions from the following: Diversity, Exception Count, Branch Coverage, Direct Branch Coverage, Method Coverage, Method Coverage (Top-Level, No Exception), Output Coverage, and Weak Mutation Coverage. Rojas et al. provide more details on each [8]. To constrain the number of combinations, we (1) use only the combinations that include the diversity score, and (2), remove a small number of semi-overlapping combinations (i.e., Branch and Direct Branch). Ultimately, the RL agent can choose from 44 combinations of fitness functions. To seed reward estimates, EvoSuiteFIT will make sure that all the actions have been tried (in a random order) before it starts using the UCB or DSG-Sarsa selection mechanisms. After selecting fitness functions, EvoSuiteFIT will proceed through the normal population evolution mechanisms. After five generations, the reformulated population is used to calculate the reward and update the expected reward. This allows sufficient population evolution to judge the effect of changing fitness functions. Over time, the combination that gains the highest reward will be more likely to be selected again until reaching convergence.

Related Work: Hyperheuristic search—often based on reinforcement learning—has been employed to improve search-based test generation. For instance, it has been used to tune the metaheuristic during Combinatorial Interaction Testing [6] or for addressing

the test ordering problem [4]. In all of these cases, the hyperheuristic is used to tune the algorithm, and not the fitness functions. Although AFFS has been performed in other domains, such as production scheduling, we were the first to apply this concept in test generation [1]. We extend our earlier approach to test suite diversity.

3 Case Study

We have assessed EvoSuiteFIT using 18 real faults from the Gson project in order to address the following research questions: **(1)** *Is EvoSuiteFIT able to yield more diverse test suites than static fitness function choices?* **(2)** *Does increased test suite diversity lead to greater likelihood of fault detection?*
In order to investigate these questions, we have performed the following experiment:
(1) Collected Case Examples: We use 18 real faults from the Gson project from the Defects4J fault database. We target the affected classes for each fault for test generation.
(2) Generated Test Suites: For each class, we generate 10 suites per approach (the two reinforcement learning algorithms—UCB and DSG-Sarsa—and two baselines—generation guided by diversity score alone and a combination of all eight fitness functions). A search budget of 10 minutes is used per suite.
(3) Removed Non-Compiling and Flaky Tests: Any tests that do not compile, or that return inconsistent results, are removed.
(4) Assessed Results: We measure the diversity of each test suite and the likelihood of fault detection (proportion of failing suites to the number generated) for each fault.

Case Examples: Defects4J is an extensible database of real faults extracted from Java projects[4]. For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose each fault, and a list of classes and lines of code modified to fix the fault. The current version contains 18 faults mined from the Gson framework [3]. We target the classes affected by faults for test generation.

Baselines: We have generated test suites using both reinforcement learning approaches, UCB and DSG-Sarsa. In addition, we generate tests for two baseline approaches representing current practice. The first is the diversity score alone. This would be the likely starting point for a tester interested in improving suite diversity. The second is the combination of all eight functions that are used in this study. 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.

Test Generation: Tests are generated using the fixed version of the class and applied to the faulty version because EvoSuite generates assertions for use as oracles. Tests that fail on the faulty version display behavioral differences between the two versions[5]. To perform a fair comparison between approaches, each is allocated a ten minute search budget. 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 approach and fault. We automatically remove tests that return inconsistent results over five executions and non-compiling test suites. On average, less than one percent of tests are removed from each suite.

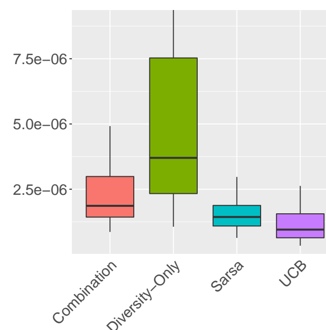
[4] Available from `http://defects4j.org`
[5] This is the same practice followed by other studies using EvoSuite, i.e. [1,3]

4 Results & Discussion

Below, we illustrate the final diversity fitness score attained by each test suite, the median fitness for each technique, and the median likelihood of fault detection for the four detected faults (faults 4, 9, 12, and 17). Because we minimize fitness, *lower* scores indicate *higher* diversity.

	DSG-Sarsa	UCB	Combination	Diversity-Only
Diversity	1.47E-06	9.99E-07	1.93E-06	3.91E-06
Fault Detection Likelihood	0.60	0.75	0.60	0.05

We can clearly see that the diversity score is a difficult fitness function to optimize on its own. When the sole target of test generation, the diversity score gives the worst final results. There is also a wide variance (distance between first and third quartile) in its results. Diversity alone not offer actionable feedback to the test generation algorithm. The combination of all eight fitness functions attains more stable results, with a better median. By adding additional fitness functions, we offer the test generation framework detailed feedback, leading to improvements in the final diversity.



Both RL approaches outperform the static approaches. DSG-Sarsa attains a 62% median improvement over diversity-only and 24% improvement over the combination. UCB is even better, attaining a 75% median improvement over diversity-only and 49% over the combination. UCB also attains a 33% median improvement over DSG-Sarsa. Both RL approaches also demonstrate lower variance in the results they attain.

A one-sided (strictly lesser) Mann-Whitney-Wilcoxon rank-sum ($\alpha = 0.05$) confirms (p-values < 0.01) that UCB outperforms all other approaches and that DSG-Sarsa outperforms the static approaches. The Vargha-Delaney A measure indicates that UCB outperforms DSG-Sarsa with small, the combination with medium, and diversity-only with large effect size. DSG-Sarsa outperforms the combination with small and diversity-only with large effect size. ***Both EvoSuiteFIT techniques—particularly UCB—increase test suite diversity over static fitness function choices with significance.***

There is limited evidence that we can use to assess the impact of diversity on the likelihood of fault detection. Only four faults are detected by any of the approaches—faults 4, 9, 12, and 17. We identified several factors impacting fault detection in Gson, including faults that are easier to expose through system-level testing, complex datatypes, and faults exposed through stronger test requirements [3]. Increased diversity may not overcome these broader issues. However, our observations suggest a positive impact from increased diversity.

Examining fault detection across the four faults, we see that (1) all approaches outperform diversity-alone, and (2) the approach with the greatest diversity—UCB—also attains the highest likelihood of fault detection. In addition, Fault 17 is uniquely detected by DSG-Sarsa. This fault is based around serialization of date values[6]. Increased

[6] See `https://github.com/google/gson/issues/1096`

diversity could lead to a wider range of attempted input and method calls. ***AFFS may improve likelihood of fault detection. However, more examples are needed to draw clear conclusions on the impact of diversity.***

5 Conclusions

We find that AFFS creates test suites that are more diverse than those created using static fitness functions. While results on fault detection are inconclusive, we also observe that increased diversity may lead to small improvements in the likelihood of fault detection. However, additional research is needed to tackle the particular challenges presented by Gson. Future work on AFFS will extend this study beyond Gson to a greater pool of faults and examine other goals that can be optimized in a similar manner. We make the EvoSuiteFIT available for use in test generation research or practice.

References

1. 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)
2. 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)
3. Gay, G.: Detecting real faults in the Gson library through search-based unit test generation. In: Proceedings of the Symposium on Search-Based Software Engineering. SSBSE 2018, Springer Verlag (2018)
4. 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. pp. 1343–1350. GECCO '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2739480.2754725
5. Idan, H.: The top 100 java libraries in 2017 - based on 259,885 source files (2017), https://blog.takipi.com/the-top-100-java-libraries-in-2017-based-on-259885-source-files/
6. 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. pp. 540–550. ICSE '15, IEEE Press, Piscataway, NJ, USA (2015), http://dl.acm.org/citation.cfm?id=2818754.2818821
7. Richard S. Sutton and Andrew G. Barto: Reinforcement Learning, Second Edition An Introduction (2018)
8. Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A.: Combining multiple coverage criteria in search-based unit test generation. In: Barros, M., Labiche, Y. (eds.) Search-Based Software Engineering, Lecture Notes in Computer Science, vol. 9275, pp. 93–108. Springer International Publishing (2015), http://dx.doi.org/10.1007/978-3-319-22183-0_7
9. Shahbazi, A.: Diversity-based automated test case generation. Ph.D. thesis, University of Alberta (2015)