Leveraging Program Invariants to Promote Population Diversity in Search-Based Automatic Program Repair

Anonymous Author(s)

ABSTRACT

Search-based automatic program repair has shown promise in reducing the cost of defects in real-world software. However, to date, such techniques have typically been most successful when constructing short or single-edit repairs. This is true even when techniques make use of heuristic search strategies, like genetic programming, that in principle support the construction of patches of arbitrary length. One key reason is that the fitness function traditionally depends entirely on test cases, which are poor at identifying partially correct solutions and lead a to highly step-wise fitness landscape with many plateaus. We propose a novel fitness function that optimizes for both functionality and semantic diversity, characterized using learned invariants over intermediate behavior. Our early results show that this new fitness function improves semantic diversity and fitness granularity, but does not yield a statistically significant change in repair performance.

KEYWORDS

genetic programming, automated program repair, program semantics, invariant analysis, fitness function

1 INTRODUCTION

Software bugs are problematic: in 2017 alone, just 606 software bugs affected $1.7 trillion USD in assets and impacted half of the world's population [2]. Moreover, programmers spend, on average, roughly half of their time finding and repairing bugs [1], which reduces the amount of time available for implementing new functionality.

Research in automated program repair attempts to automate the bug repair process, reducing these costs. Most modern repair techniques take as input program source and a test suite, and seek a patch that leads all input tests to pass, including those that are initially failing (identifying the bug). One dominant class of repair approaches relies on heuristic search strategies like random search [32], a deterministic heuristic walk of the search repair space [25, 26, 38], or genetic programming [19, 22]. These techniques are promising in their application to both historical bugs in real-world programs as well as, more recently, industrial settings [28]. However, to date, most techniques are limited to finding single-edit repairs. For some techniques, this is by design [19, 32, 38], and avoids the challenges of traversing a combinatorial multi-edit search space. However, even techniques like GenProg, which uses genetic programming and can in principle construct multi-edit repairs, still in practice produce patches that reduce to single edits [33]. This inability to construct multi-edit repairs limits the scope of such repair techniques to relatively simple, single-line bugs.

One major factor that limits the ability of genetic programming to evolve multi-edit bug repairs is a fitness function that relies on test cases. Ideally, a candidate patch that contains a partial solution — such as one of several required edits — would cause the program to pass more test cases than one that contains no useful edits. The test-based fitness function could then usefully identify partial solutions, which could then be selected and recombined for evolution towards a complete solution. Unfortunately, this does not typically hold in practice. Test case-based fitness functions instead tend to assign the same fitness score to many different candidate solutions, producing large *fitness plateaus* [5, 13]. An insufficiently granular fitness function that fails to distinguish between distinct candidate solutions prevents the search from identifying promising edits, and instead simply prevents the introduction of destructive edits (i.e., edits that cause test failures) [36]. Moreover, tests inherently lack information about intermediate program behavior. They are therefore poorly suited to guiding a search towards semantically diverse mechanisms to accomplishing the same repair, a possibly fruitful application for program repair [12].

We seek to overcome these limitations by providing additional information about intermediate behavior to help the genetic programming search strategy distinguish between promising candidate patches. One potential source of additional information is likely program invariants over intermediate program values, like `x = 0` or `y > a`. There exist mature dynamic invariant detection tools, such as Daikon [10], that can infer likely invariants observed as holding over multiple program executions. These invariants offer semantic information about intermediate program behavior. We propose to infer a set of likely invariants by observing the *original* program behavior on its *initially passing* test cases, which demonstrate good behavior. We then evaluate the behavior of the program as modified by candidate patch solutions on those inferred invariants. We use this observed behavior over the entire population of candidate patches to determine which patch variants appear to be more *semantically diverse*. We then use semantic diversity as an optimization objective, along with the number of passing test cases, to determine variant fitness. Beyond the potential utility of a technique that can find diverse solutions to a program repair problem, diversity among variants in a genetic programming (GP) population discourages genetic drift and premature convergence towards local optima [9].

Our goal is a repair search strategy with fewer fitness plateaus, and that can generate potentially useful semantic diversity in its population of candidate solutions. Our technique is designed to be more general than previous attempts to either examine intermediate memory values [5], or to use intermediate program predicates [11]

GI'19, May 28, 2019, Montréal, QC, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

to characterize repair fitness, and further does not require a training step.

This paper's main contributions are:

- A new program repair technique that uses learned program invariants to promote semantic diversity of variant programs.
- A compact, invariant-based representation of program semantics. Our representation facilitates an approximation of the semantics difference between programs.
- An evaluation of an implementation of our technique's performance across sampled IntroClassJava [8] bugs.
- An evaluation of our technique's scalability to Defects4j [14], a set of larger, real-world buggy programs.

The rest of this paper proceeds as follows. Section 2 outlines background helpful to understanding our approach. We motivate the approach with an illustrative example in Section 3 and then describe the approach in Section 4. Section 5 describes our evaluation on both large and small programs, showing our technique's potential to reduce fitness plateaus and increase population diversity. We describe threats to the validity of our experiments and possible limitations of our approach in Section 6. Section 7 puts our work in context with respect to the prior literature; we conclude with a discussion and vision for future directions in Section 8.

2 BACKGROUND

This section provides background helpful to understanding our technique and contributions.

Automatic Program Repair (APR). Given a program and an oracle that determines its correctness, an Automatic Program Repair tool attempts to find a *patch* that modifies the program in a way the oracle deems correct. The majority of modern repair techniques use test cases as an oracle, as a widely available but incomplete proxy for formal correctness specifications. Test cases that fail on the original program (*negative* tests) expose the bug to be repaired; test cases that pass on the original program (*positive* tests) indicate desirable behavior that should be maintained. Thus, the goal of APR is to find a patch such that all provided tests pass on the modified program. We describe the previous alternatives to this approach in Section 7. Repair techniques typically also use the test cases to perform fault localization (generally using an off-the-shelf technique [18, 22, 29]), constraining the search to a smaller set of candidate repair locations.

Two dominant approaches to APR are *semantics-based* and *heuristic* or *search-based*. Semantic-based repair tools [18, 29, 39] seek to synthesize a repair by deducing intended program semantics. Heuristic or search-based approaches explore search spaces of templated repairs applied to the abstract syntax tree of the program. Heuristic techniques [25, 26, 38] use predefined schemas and probabilistic models or other heuristics to determine how to apply candidate changes. We focus in this work on search-based techniques like GenProg [22] and RSRepair [32] that explicitly use stochastic techniques to traverse the search space.

Techniques in both classes have been successfully applied to real-world programs. However, they vary in the type of code they can handle and the types of patches they can create. Semantics-based approaches are typically restricted by their underlying reasoning to to synthesizing new conditionals or, less commonly, the right-hand side of assignments. Search-based techniques can provide a broader set of syntactic repair templates, but must still be restricted in the type and variety that they explore (lest the search problem become intractable). We give concrete details for GenProg, below. Additionally, note that many techniques are expressly limited to constructing or exploring single-edit repairs. Even techniques (like GenProg, Angelix, or S3) that can in principle construct multi-edit repairs rarely do, and seemingly multi-edit patches often reduce to a single edit [33].

GenProg. GenProg [22] is a search-based APR tool that uses genetic programming [16] to traverse the space of possible patches. Patches are comprised of a variable-length sequence of *statement-level* edits to the original program (typically *deletion*, *replacement*, or *insertion* of one statement after another). Edits requiring new code (insertion or replacement) restrict the domain of new code to elsewhere in the same buggy program. New patches are created via recombination (using *crossover*) and mutation (by appending a new randomly-instantiated edit to an existing patch candidate); intermediate solutions are *selected* to continue evolution to subsequent generations, typically with a selection scheme weighted by variant *fitness*, or suitability. Like most other repair techniques, GenProg optimizes for patched program performance on the input test cases, with a fitness function comprised of a weighted sum of the number of positive and negative tests that a modified program passes. However, empirically, this fitness function lacks granularity and often fails to precisely distinguish between partial solutions [5, 11, 13]. GenProg was initially implemented to repair C programs[1] and has been reimplemented for Java.[2]

Automatic Learning of Likely Program Invariants. There are tools that automatically infer likely program invariants, or predicates, to characterize expected or abnormal program behaviors. Such techniques have been used to characterize either normal or abnormal program executions, useful in applications like specification mining [10], N-variant systems [31], and fault localization [23]. We use Daikon [10], a mature dynamic analysis technique that infers likely program invariants by running the program on a provided set of inputs or tests; observing the intermediate values the program computes; and then matching those values to template predicates to report properties that were true over all observed executions. Daikon's released toolset[3] works on a variety of languages, including Java. We discuss Daikon's limitations in Section 6.

3 MOTIVATING EXAMPLE

We begin with a short example to motivate our proposed approach. Consider the Java function implementing Euclid's GCD algorithm shown in Figure 1. This code contains a two-part bug: when `a=0`, it will return 0 instead of `b` (the correct answer); when `b=0` and `a > 0`, it will return `a` (instead of 0). A simple patch for this bug requires two changes: deletion of line 16, and a replacement of line 4 with line 8 (or appending line 8 before or after line 4). Note that GenProg can in principle construct this patch using statement-level modifications:

[1] https://github.com/squaresLab/genprog-code
[2] https://github.com/squaresLab/genprog4java
[3] https://plse.cs.washington.edu/daikon/

```
1  public int gcd(int a, int b) {
2    int result = 1;
3    if (a == 0) {
4      b = b - a;
5    } else {
6      result=a;
7      while (b != 0) {
8        result = b;
9        if (a > b) {
10         a = a - b;
11       } else {
12         b = b - a;
13       }
14     }
15   }
16   result=a;
17   return result;
18 }
```

Figure 1: A buggy implementation of Euclid's GCD algorithm. This function is incorrect when `a=0` (returning 0 instead of `b`).

GenProg's mutation operators can for example delete all of line 16 in Figure 1, or replace the statement at line 4 with the one at line 8 (but could not modify the expression in any of the if conditions on lines 3 or 9).

Although test cases usefully characterize program behavior, they by design do not offer intermediate information on partial program correctness. More importantly, two very different programs can easily pass the same number of test cases. For example, a candidate patch that simply deletes line 16 would result in a program that passes the same tests as the original buggy code. However, this deletion is in fact one of two required edits. Ideally, the fitness function would rank it more highly than the empty patch.

One potential source of additional semantic information that could distinguish between such patches is program invariants. Programs that display different behavior with respect to a set of likely invariants may be interestingly different from one another in terms of intermediate program semantics. For example, given a small set of positive test cases, Daikon can infer likely invariants such as **`a % gcd(a,b)= 0`** and **`b % gcd(a,b)= 0`** . These invariants describe important properties of the **`gcd`** function.

Note that the buggy code in Figure 1 violates these properties on an initially failing test where **`a = 0`**. On the other hand, the candidate patch that deletes line 16 produces a program to correctly preserve those invariants. This suggests that the change may be worth maintaining in the search.

Note that we do not know a priori which possible properties are most important, and Daikon can produce noisy output with trivial or overfitting properties. Fortunately, genetic programming [17, 30] is robust to noisy fitness functions. Thus, we propose in this initial work to use inferred likely invariants as one component of a genetic programming-based repair search to promote diversity and reduce fitness plateaus.

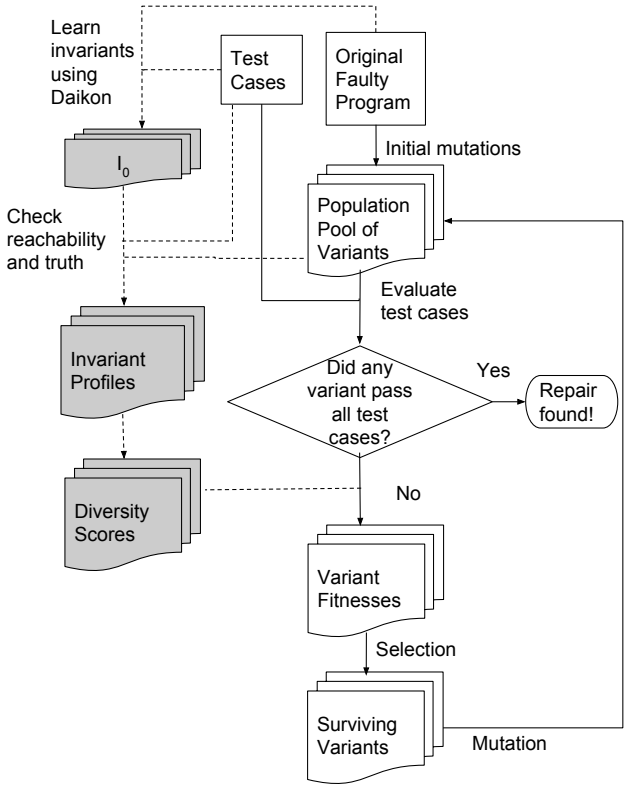

Figure 2: A flow-chart of our approach: White boxes and solid lines show GenProg's architecture, grey boxes and dotted lines are our approach's additions.

4 APPROACH

In this section, we describe our new genetic programming-based repair technique that optimizes for both *functionality* (measured using test cases) and *variant diversity*. We measure diversity by using inferred program invariants to characterize modified program (and thus candidate patch) behavior. We call this its *invariant profile* (Section 4.1). We then quantify a candidate patch's diversity by comparing its invariant profile to the rest of the patches in the search population (Section 4.2). We use this invariant-based diversity score, along with the number of initially passing and failing test cases passed, as optimization objectives in the multi-objective genetic algorithm NSGA-II [6] (Section 4.3).

Figure 2 shows the high-level architecture of our approach compared to GenProg. We add the following procedure to the core GenProg approach: First, we run Daikon on the original faulty program with positive test cases to obtain a set of inferred possible invariants, denoted by I_0. Then, in each generation, we run tests and instrumented code through each variant in the population to obtain invariant profiles. We calculate diversity scores for each variant in the population based on this profile, and use the diversity scores in the fitness function in the multi-objective search.

4.1 Invariant Profiles

For each candidate patch, we construct an invariant profile containing information about the patch's behavior. This profile consists of the reachability and truth value of a set of potentially interesting invariants inferred by Daikon [10]. Due to Daikon's high computational cost, we do not learn invariants from each patch. Instead, we infer potentially interesting invariants from the originally buggy program to create a starting set of invariants I_0. For each candidate patch, we then construct an invariant profile from a subset of I_0.

Daikon can learn potential invariants at multiple levels of program abstraction. However, because patches can manipulate code arbitrarily, predicates that are sensible in describing original intermediate program behavior may no longer be evaluable on patched code. We therefore include only *method-level* invariants in I_0, which enables evaluation of whether an invariant continues to hold in an arbitrary candidate patch.

Additionally, we only use positive test cases to inform the construction of I_0. This is due to the practical limitations of running Daikon on failing test cases that crash the program. We believe that learning properties over negative test cases may also be interesting, and leave the engineering and conceptual approach to doing so to future work. We do, however, check whether invariants in I_0 hold over both positive test cases and negative test cases when constructing invariant profiles for candidate patches.

Assuming the following definitions:

$$T = \text{Set of all test cases in a program's test suite}$$
$$T^+ = \text{Set of all positive test cases in } T$$
$$T^- = \text{Set of all negative test cases in } T$$
$$I_0 = \text{Set of inferred method-level invariants}$$

The invariant profile for a particular patch v is constructed as follows. We apply v to the original program code to produce a variant program. We then instrument the variant program's code to evaluate, for every invariant p in I_0, whether the method associated with p is ever reached, and if so, whether p holds for the variant or not. We collect the aforementioned data separately for T^+ and T^-, and track and incorporate the invariant data derived from the two sets of tests separately.

We represent the invariant profile of a variant m, Π_m, as a ternary vector of length $2|I_0|$, where for all i between 0 and $|I_0|$, the $(2i)$-th and $(2i+1)$-th characters of Π_m are:

$$\Pi_m[2i] = \begin{cases} 0 & p_i \text{ is always true for all tests in } T^+ \\ 1 & p_i \text{ is false at least once in } T^+ \\ 2 & p_i \text{ is never reached by any test in } T^+ \end{cases}$$

$$\Pi_m[2i+1] = \begin{cases} 0 & p_i \text{ is always true for all tests in } T^- \\ 1 & p_i \text{ is false at least once in } T^- \\ 2 & p_i \text{ is never reached by any test in } T^- \end{cases}$$

4.2 Quantifying Invariant-based Diversity

To score the invariant-based diversity of candidate patches, we compare a patch's invariant profile to that of every other patch in the population. The goal is to determine the uniqueness of a patch's invariant profile with respect to the other candidate patches. To measure the difference in invariant behavior between two patches m_1 and m_2, we use the Hamming distance of their invariant profiles Π_{m_1} and Π_{m_2}:

$$\Delta(m_1, m_2) = \text{HammingDistance}(\Pi_{m_1}, \Pi_{m_2})$$

The diversity score of a patch m in population P, assuming that m compiles and terminates within a specified time limit, is the sum of the Hamming distances between m's invariant profile and the profiles of every other patch in P whose variant programs compile and terminate:

$$\text{diversity}(m) = \sum_{n \in P} \begin{cases} \Delta(m, n) & m \text{ and } n \text{ compile and terminate} \\ 0 & \text{otherwise} \end{cases}$$

In general, diversity among variants in a search population discourages premature convergence towards local optima [9]. By promoting semantic diversity, we aim to encourage a repair-oriented genetic algorithm to explore niches in semantic space.

4.3 Promoting Diversity

We use variant diversity score as an optimization objective, alongside the number of positive and negative tests passed This treats program repair as a multi-objective optimization problem. We use the NSGA-II algorithm [6] for this purpose. NSGA-II is a genetic algorithm that optimizes for multiple objectives using Pareto optimality [6]. NSGA-II also employs mechanisms to encourage variant diversity over the course of its search.

Note that our use of NSGA-II optimizes for initially passing and failing tests as separate objectives rather than using a weighted sum of positive/negative tests as a single objective, as GenProg does.

5 EVALUATION

To analyze the effects of promoting invariant-based diversity, we implemented our proposed technique as a extension of GenProg4J[4] and used the original version of GenProg4J as a control. Using our modified version of GenProg, we answer the following questions:

RQ1 Does optimizing for invariant-based diversity change the effectiveness in searching for a repair with respect to the number of repairs found, the performance of the search for repairs, and the percentage of seeds that succeeds for each repaired bug?

RQ2 Does optimizing for invariant-based diversity increase the diversity of patches in a population?

RQ3 Does invariant-based diversity provide more granular fitness (and therefore less plateauing)?

RQ4 Does our diversity-enhanced extension of GenProg scale to large, real-world programs?

5.1 Efficiency, Repairability and Diversity

To answer RQs 1, 2, and 3, we use a modified version of the IntroClassJava [8] dataset of bugs (see Section 6 for a discussion of the changes). IntroClassJava is a set of small Java programs (< 30 lines) derived from IntroClass [21], a set of buggy C programs written by students in an introductory programming course. We use IntroClassJava as a benchmark since the small size of its programs

[4] https://github.com/squaresLab/genprog4java

Bug Scenario	GenProg	Our Approach
SMALLEST_AF81_000	1 / 10	1 / 10
SMALLEST_8839_002	10 / 10	10 / 10
DIGITS_0CDF_006	10 / 10	10 / 10
DIGITS_0CDF_007	9 / 10	10 / 10
DIGITS_D120_001	8 / 10	9 / 10
DIGITS_5B50_000	1 / 10	0 / 10
Total	39 / 60	40 / 60

Table 1: A comparison of repair success rates for each bug repaired by either GenProg or our approach.

reduce the cost of searching for repairs, making it more amenable to a large number of experiments.

We ran GenProg and our technique on a random sample of 59 out of a possible 297 bugs from IntroClassJava, and repeated each experiment 10 times to allow a statistical comparison of the results. To ensure a fair comparison between GenProg and our technique, we use an identical configuration for both techniques, based on the default settings for GenProg4J. We used the append, replace, and delete repair operators, one-point crossover, tournament selection ($k = 8$), a population size of 40, and ran for 10 generations. We ran each experiment on Amazon EC2 using a *c5.l* instance with two vCPUs, 4 GB of RAM, and a 100 GB *gp2* storage volume, running Ubuntu 16.04 LTS. We intend to make an Amazon Machine Image for reproducing our experiments available as part of our replication package after double-blind review.

5.1.1 RQ1 – Performance. Across both GenProg and our approach, a total of six unique bugs were successfully repaired at least once. Table 1 provides the repair success rate (i.e., the likelihood that a given attempt to repair the bug will be successful) for each of the six bugs.

We find that, with one exception, our approach and GenProg fix the same set of bugs. In the one exceptional case, GenProg is able to repair DIGITS_5B50_000 for one out of ten seeds. Using a two-sided version of Fisher's exact test, we fail to demonstrate a statistically significant difference ($p < 0.05$) in the success rate between GenProg and our approach for each individual bug and for across all bugs in aggregate. Given the low success rate and lack of statistical significance for DIGITS_5B50_000, the inability of our technique to find a repair may be due to a sampling error.

Figure 3 compares the *efficiency* of our approach against GenProg for the five bugs that were repaired by both techniques, where we define efficiency as the number of unique candidate patch evaluations. Using a one-sided Mann-Whitney U test [27], we fail to find a statistically significant improvement ($p < 0.05$) over GenProg across all bugs.

Table 2 reports the number of syntactically unique patches that were found for each bug. Using a two-sided version of Fisher's exact test, we were unable to demonstrate a statistically significant difference at the $\alpha = 0.05$ confidence level ($p = 0.249$).

In summary, our results fail to demonstrate a significant difference, be it negative or positive, to efficiency, repair success rate, and the number of unique patches.

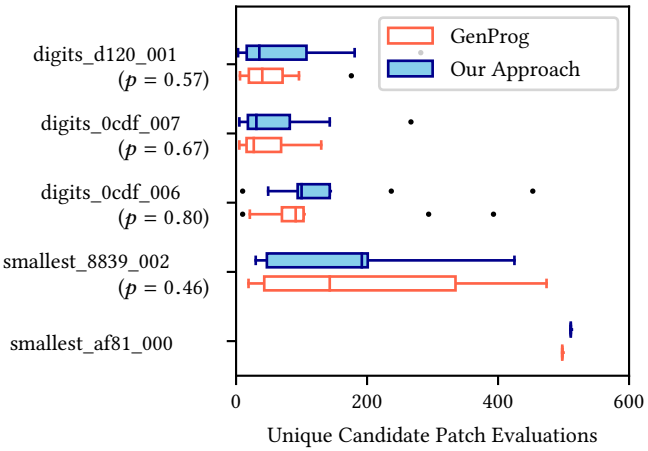

Figure 3: A comparison of the number of unique candidate patch evaluations before a repair was found for the five bugs fixed by both our approach and GenProg. Using a one-sided Mann-Whitney U test, we fail to demonstrate a statistically significant difference ($p < 0.05$) for any bug.

Bug Scenario	GenProg	Our Approach
SMALLEST_AF81_000	1 / 1	1 / 1
SMALLEST_8839_002	8 / 10	8 / 10
DIGITS_0CDF_006	9 / 10	10 / 10
DIGITS_0CDF_007	8 / 9	9 / 10
DIGITS_D120_001	4 / 8	6 / 9
DIGITS_5B50_000	1 / 1	0 / 0
Total	31 / 39	34 / 40

Table 2: A comparison of the number of unique repairs found for each bug across all seeds.

5.1.2 RQ2 – Diversity. To determine whether our multi-objective search algorithm is an effective means of promoting semantic diversity within the population, we compare the *normalized population diversity* between GenProg and our approach across all 59 bugs sampled from IntroClassJava. We define normalized population diversity (NPD) of a given population P as the sum of diversity scores of the individual candidate patches belonging to that population, divided by the number of individual patches in the population squared and the length of invariant profile (Π) for a given bug.

$$NPD(P) = \frac{\Sigma_{m \in P} \text{diversity}(m)}{|P|^2 \cdot |\Pi|}$$

Figure 4 shows the distribution of normalized population diversity scores for both our approach and GenProg at each generation prior to selection. Our results show significantly higher levels of diversity within the population (after the initial population) when our approach is used, which suggests that our algorithm is effective at promoting diversity. We observe low levels of population diversity in the initial population of the search (i.e., Generation 0) for

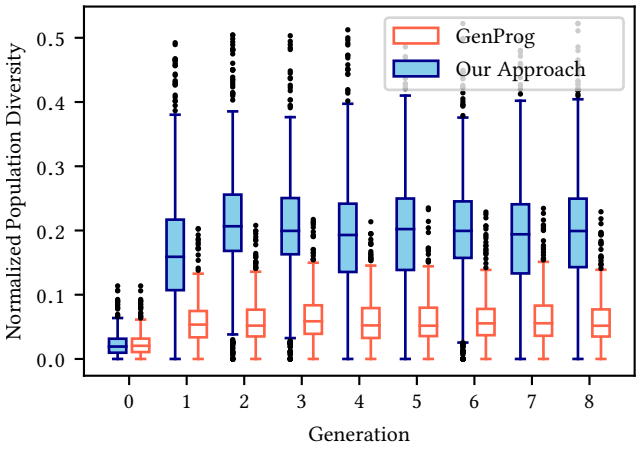

Figure 4: A comparison of normalized population diversity over the course of the search. Except in generation 0, there is a statistically significant difference ($p < 0.001$), computed using a one-sided Mann-Whitney U test, between GenProg and our approach.

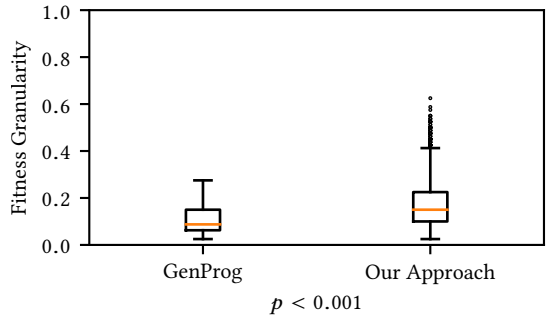

Figure 5: A comparison of fitness granularity within the population prior to selection in each generation.

both our approach and GenProg; this result is to be expected as the population has not been subject to selection. When our approach is used, we see diversity levels increase until the second generation of the search, at which point diversity remains stable.

For GenProg, we observe a significant increase in diversity between the initial population and all subsequent populations, which immediately stabilizes after the first generation. This result suggests that GenProg's test-based fitness function implicitly selects for diversity within the population, albeit relatively weakly.

5.1.3 RQ3 – Fitness Granularity. We measure the granularity of fitness information available to each repair method by analyzing the phenotypic diversity [4, 7] of populations. We use a ratio of unique fitness scores to the size of the population as a measure of phenotypic diversity.

Figure 5 compares fitness granularity between GenProg and our approach. Using a one-sided Mann-Whitney U test, we find that our approach significantly ($p < 0.001$) increases fitness granularity. We observe a 71% increase in median fitness granularity from 0.088 to 0.150 with our approach.

Bug	GenProg	Our Approach	Difference
LANG11	3586 s	4914 s	1.37 X
LANG29	1783 s	2781 s	1.56 X
LANG36	2088 s	3256 s	1.56 X
LANG8	5850 s	6239 s	1.07 X
LANG9	3304 s	7170 s	2.17 X
MATH30	5356 s	5433 s	1.01 X
MATH44	5906 s	14424 s	2.44 X
MATH46	4023 s	51195 s	12.73 X
MATH79	6033 s	7903 s	1.31 X
MATH86	3751 s	5325 s	1.42 X
Median	3887 s	5386 s	1.50 X
Mean	4168 s	10864 s	2.61 X

Table 3: A comparison of the wall-clock time taken by GenProg and our approach to run to completion for a single seed across 10 bugs from the Defects4J benchmark, in seconds.

5.2 Scalability

To answer RQ4, we compare the wall-clock time taken to run to completion against GenProg on 10 bugs from the Defects4J [14] dataset: a collection of real-world bugs in open-source Java projects. To accommodate the larger test suites of Defects4J programs, we no longer infer invariants I_0 from the full set of positive tests T^+. Instead, we use a subset of T^+ that only includes positive tests that co-occur in JUnit test classes that also contain negative tests. We use a near-identical configuration to our IntroClassJava experiments with two changes: (1) we use a population size of 20, and (2) we evaluate each variant on a random sample of 10% of the positive tests [11], along with all of the negative tests. We ran all experiments on the same machine with the following specifications: Ubuntu 16.04 LTS, 4 x Intel Xeon E7-4820 (40 cores), 128 GB RAM, and 2 TB storage.

Table 3 presents a summary of our results. We find that the median run-time is 50% higher when our approach is used. We expect our approach to be naturally slower than GenProg since it introduces two notable overheads: (1) prior to the search, we use Daikon on the original program to infer a set of likely predicates that are later used by invariant profiles; (2) we instrument each variant program and run the positive test cases again to obtain an invariant profile. The static overhead from (1) is necessary only once per buggy program, and can be amortized across multiple (or longer) runs. The dynamic overhead from (2) could be reduced by an optimized version of our prototype implementation (e.g., by avoiding redundant test executions).

The order-of-magnitude increase in the run-time for MATH46 is likely explained by the abnormally large number of invariants inferred by Daikon. Daikon infers 390 method-level likely invariants for MATH46: a substantially greater number of invariants compared to other math bugs for which Daikon infers between 36 and 148

invariants. Our implementation adds instrumentation for each individual predicate in I_0 to the program. As a result, a large amount of instrumentation was added to MATH46, creating a substantial static overhead. This overhead may be reduced by reducing the overhead of our instrumentation and by identifying and filtering spurious invariants.

In summary, our results show that our prototype implementation, in most cases, introduces a tractable run-time computational overhead (approximately 50%) over base GenProg, suggesting it may plausibly be applied to programs of realistic scale.

6 THREATS AND LIMITATIONS

Manual Code Modification. In porting IntroClass for C [21] to Java, the IntroClassJava authors elected to include all primary functionality in a single Java method. Input values are retrieved using I/O in the middle of this single method. Daikon struggles to identify interesting invariants over code organized this way. We thus mechanically refactored IntroClassJava code to relocate core logic into a separate method and more clearly delineate input and output values for the different problems. This decouples our technique performance from Daikon performance. We anticipate that dataflow analysis or slicing techniques can automate the task of identifying potentially interesting input and output values, as has been done in different program repair technique contexts [15].

Repair quality. Repair quality is a key unsolved concern in program repair. Both search- and semantics-based approaches can create patches that *overfit* to the provided test cases, or fail to generalize to the desired but unwritten program specification [20, 35]. We did not evaluate the quality of the repairs that GenProg and our repair technique produced, relying instead on performance on the provided test cases. We leave assesment of repair quality to future work.

Invariant quality. While Daikon can detect various types of invariants, Daikon's limited set of supported invariants and scopes restrict the tool's flexibility. Daikon's inability to detect invariants relating to the state of variables at intermediate statements of a method lead us to refactor IntroClassJava.

Daikon may also produce trivial invariants such as `this != null`. These invariants provide no useful semantic information. Unnecessarily evaluating these invariants consumes computational resources when constructing invariant profiles. Inferred invariants may also overfit. For example, if a program executes the function `sin(x)` only ever with input `x = pi * n`, Daikon might infer `sin(x) = 0` as an invariant. Nonetheless, these overfit invariants may sometimes be useful on our approach as breaking them with the same input does indicate behavior change in a program. We might attempt to mitigate these limitations in the future by implementing a filter to Daikon's output attempts to identify and remove trivial or overfitting invariants.

7 RELATED WORK

Fast et al. [11] propose an alternative fitness function for GenProg that uses learned invariants to identify and promote partial solutions. Their fitness function is implemented as a linear model that accepts inferred invariants and test cases as its features. Existing patches are used to create training data that is used to construct a linear model for a given program and to learn associations between invariants and patch correctness. In contrast to Fast et al.'s approach, our approach does not require training data.

de Souza et al. [5] aimed to reduce fitness plateaus by injecting source code checkpoints to track pre-determined variables. Based on changes in these variables and the results of fault localization, they incorporate a *checkpoints* metric into the fitness function. de Souza et al. found that their approach reduced plateauing, fixed more bugs, and improved efficiency. Our approach provides an orthogonal method for reducing plateauing that uses automatically learned invariants instead of the values of pre-determined variables.

Timperley et al. [37] aim to reuse variant test results collected over the course of the search to determine fix locations via a dynamic mutation-based fault localization. They found that modifications to correct statements are more likely to result in the failure of previously passing tests, but that few previously passing tests covered the faulty area of the program. Ultimately, they were unable to demonstrate a significant improvement to the accuracy of fault localization when variant test cases were analyzed. Our approach might address the lack of sufficient test coverage necessary to perform effective mutation analysis — at no additional cost — by observing changes to invariant profiles rather than test cases.

Beadle and Johnson [3] investigated promoting semantic diversity during population initialization in genetic programming. They developed semantically driven initialization algorithms for Boolean and artificial ant problems. Using their algorithms often resulted in better performance compared to RHH [16], a standard initialization technique. Their results suggest that promoting invariant diversity during initialization may yield greater performance.

8 CONCLUSIONS

Our early results demonstrate that inferred invariants can be used to effectively increase semantic diversity within the population, and to reduce fitness plateauing. Using Defects4J, we demonstrate that a prototype implementation of our technique scales to bugs in large-scale, real-world programs. In our experiments on IntroClassJava, we fail to demonstrate a statistically significant improvement to efficiency, the number of bugs fixed, unique patches found, and repair success rate.

A possible explanation for this result is that the search space contains few, if any, acceptable repairs. Our approach uses the same statement-level repair operators as GenProg, and so it shares the limitations of GenProg's search space. To obtain a better idea of how our approach improves search performance, we plan to evaluate our approach with a richer search space, generated using a larger number of mutation operators.

Going forward, we also plan to collect more data on our approach's performance on a larger number of bugs, including the rest of the IntroClassJava and Defects4J datasets, along with JaConTeBe [24] and Bugs.jar [34]. We may also attempt to improve our approach's run time by pruning invariants and by optimizing the code instrumentation process.

To encourage further investigation and ensure reproducibility, we provide a prototype implementation of our technique, along with

the results and raw data from our study, as part of our replication package at:

https://bit.ly/2HNQ5g7

REFERENCES

[1] [n. d.]. Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year. http://www.prweb.com/releases/2013/1/prweb10298185.htm. Accessed December, 2018.

[2] [n. d.]. Software Fail Watch: 5th Edition. https://www.tricentis.com/software-fail-watch/. Accessed December, 2018.

[3] L. Beadle and C. Johnson. 2009. Semantic analysis of program initialisation in genetic programming. *Genetic Programming and Evolvable Machines* 10 (2009), 307–337.

[4] E. Burke, S. Gustafson, and G. Kendall. 2002. A Survey and Analysis of Diversity Measures in Genetic Programming. In *Genetic and Evolutionary Computation Conference (GECCO'02)*. 716–723.

[5] E. F. de Souza, C. Le Goues, and C. G. Camilo-Junior. 2018. A Novel Fitness Function for Automated Program Repair Based on Source Code Checkpoints. In *Genetic and Evolutionary Computation Conference (GECCO '18)*.

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Comp.* 6, 2 (2002), 182–197.

[7] P. D'haeseleer. 1994. Context preserving crossover in genetic programming. In *Conference on Evolutionary Computation. World Congress on Computational Intelligence*, Vol. 1. 256–261.

[8] T. Durieux and M. Monperrus. 2016. *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs.* Research Report. Universite Lille 1.

[9] A. E. Eiben and J. E. Smith. 2011. *Introduction to evolutionary computing*. Springer.

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1–3 (2007), 35–45.

[11] E. Fast, C. Le Goues, S. Forrest, and W. Weimer. 2010. Designing Better Fitness Functions for Automated Program Repair. In *Genetic and Evolutionary Computation Conference (GECCO '10)*. 965–972.

[12] R. Feldt. 1998. Generating diverse software versions with genetic programming: an experimental study. *IEEE Proceedings-Software* 145, 6 (1998), 228–236.

[13] S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues. 2009. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference (GECCO '09)*. 947–954.

[14] R. Just, D. Jalali, and M. D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis (ISSTA)*. 437–440.

[15] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. 2015. Repairing Programs with Semantic Code Search. In *Automated Software Engineering (ASE '15)*. 295–306.

[16] J. R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press.

[17] J. R. Koza. 1994. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing* 4, 2 (01 Jun 1994), 87–112.

[18] X. D. Le, D. H. Chu, D. Lo, C. Le Goues, and W. Visser. 2017. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE '17)*. 593–604.

[19] X. D. Le, D. Lo, and C. Le Goues. 2016. History Driven Program Repair. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, Vol. 1. 213–224.

[20] X. D. Le, F. Thung, D. Lo, and C. Le Goues. 2018. Overfitting in semantics-based automated program repair. *Empirical Software Engineering* 23, 5 (2018), 3007–3033.

[21] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Trans. Software Eng.* 41, 12 (2015), 1236–1256.

[22] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38 (2012), 54–72.

[23] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. 2005. Scalable statistical bug isolation. In *Conference on Programming Language Design and Implementation (PLDI '05)*. 15–26.

[24] Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao. 2015. JaConTeBe: A Benchmark Suite of Real-World Java Concurrency Bugs. In *Conference on Automated Software Engineering (ASE '15)*.

[25] F. Long and M. Rinard. 2015. Staged program repair with condition synthesis. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. 166–178.

[26] F. Long and M. Rinard. 2016. Automatic patch generation by learning correct code. In *Principles of Programming Languages (POPL '16)*. 298–312.

[27] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50–60.

[28] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. 2019. SapFix: Automated End-to-End Repair at Scale *(ICSE-SEIP '19)*. (to appear).

[29] S. Mechtaev, J. Yi, and A. Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE '16)*. 691–701.

[30] B. L. Miller and D. E. Goldberg. 1996. *Optimal Sampling for Genetic Algorithms.* Technical Report. University of Illinois at Urbana-Champaign.

[31] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. 2009. Automatically patching errors in deployed software. In *Symposium on Operating Systems Principles*. 87–102.

[32] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. 2014. The Strength of Random Search on Automated Program Repair. In *International Conference on Software Engineering (ICSE '14)*. 254–265.

[33] Z. Qi, F. Long, S. Achour, and M. Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *International Symposium on Software Testing and Analysis (ISSTA '15)*. 24–36.

[34] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad. 2018. Bugs.Jar: A Large-scale, Diverse Dataset of Real-world Java Bugs. In *International Conference on Mining Software Repositories (MSR '18)*. 10–13.

[35] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun. 2015. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. 532–543.

[36] C. S. Timperley. 2017. *Advanced Techniques for Search-Based Program Repair.* Ph.D. Dissertation. University of York.

[37] C. S. Timperley, S. Stepney, and C. Le Goues. 2017. An Investigation into the Use of Mutation Analysis for Automated Program Repair. In *International Symposium on Search Based Software Engineering (SSBSE '17)*. 99–114.

[38] W. Weimer, Z. P. Fry, and S. Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *International Conference on Automated Software Engineering (ASE '13)*. 356–366.

[39] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. R. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Software Eng.* 43, 1 (2017), 34–55.