A Novel Fitness Function for Automated Program Repair Based on Source Code Checkpoints

Eduardo Faria de Souza
Instituto de Informática
Universidade Federal de Goiás
Goiânia, Goiás, Brazil
eduardosouza@inf.ufg.br

Claire Le Goues
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
clegoues@cs.cmu.edu

Celso Gonçalves Camilo-Junior
Instituto de Informática
Universidade Federal de Goiás
Goiânia, Goiás, Brazil
celso@inf.ufg.br

ABSTRACT

Software maintenance, especially bug fixing, is one of the most expensive problems in software practice. Bugs have global impact in terms of cost and time, and they also reflect negatively on a company's brand. GenProg is a method for Automated Program Repair based on an evolutionary approach. It aims to generate bug repairs without human intervention or a need for special instrumentation or source code annotations. Its canonical fitness function evaluates each variant as the weighted sum of the test cases that a modified program passes. However, it evaluates distinct individuals with the same fitness score (*plateaus*). We propose a fitness function that minimizes these plateaus using dynamic analysis to increase the granularity of the fitness information that can be gleaned from test case execution, increasing the diversity of the population, the number of repairs found (expressiveness), and the efficiency of the search. We evaluate the proposed fitness functions on two standard benchmarks for Automated Program Repair: IntroClass and ManyBugs. We find that our proposed fitness function minimizes plateaus, increases expressiveness, and the efficiency of the search.

CCS CONCEPTS

• **Computing methodologies → Genetic algorithms**; • **Software and its engineering → Search-based software engineering**;

KEYWORDS

genetic programming, software engineering, program repair, fitness function

ACM Reference Format:
Eduardo Faria de Souza, Claire Le Goues, and Celso Gonçalves Camilo-Junior. 2018. A Novel Fitness Function for Automated Program Repair Based on Source Code Checkpoints. In *GECCO '18: Genetic and Evolutionary Computation Conference, July 15–19, 2018, Kyoto, Japan,* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205455.3205566

1 INTRODUCTION

Regardless of the size or experience in a group of developers, the code they produce is susceptible to bugs. Critical bugs are present in everything from small to mature projects and from proprietary to open-source software [3]. Such bugs compromise functionality, expose sensitive data, or even grant privilege escalation, among other harmful effects. At least 3.7B people in the world were affected by bugs in calendar year 2017. These bugs led to vehicle recalls, malware in mobile phones, undelivered paychecks, hacked accounts, and so on. Overall, such losses amounted to 1.7T USD in losses from software failure in 2017 [22]. Bug fixing, an action traditionally addressed by humans, is costly and time-consuming; worse, human developers often introduce new defects over the course of repairing others [28].

Automated Program Repair is a research area that seeks to automatically fix bugs efficiently and accurately [11]. One way to subdivide this domain is into behavioral and state-based repair approaches. *Behavioral* approaches seeks to address bugs by directly modifying source code, while *state-based* techniques aim to change the environment or otherwise apply modifications at run time [18].

GenProg is a classic behavioral technique for automated program repair [13]. It is based on the principles of Genetic Programming [2], which it uses to evolve a set of changes (a *patch*) to buggy source code. Given as input a source code and a test suite, it seeks such a patch that leads all input tests to pass, including those that are initially failing (identifying the bug). It does so without special annotations or any other form of human intervention.[1]

A patch is represented as a sequence of edit operations to the source code. Each edit operation is composed of an action (append, swap/replace, or delete) and one or two statements, depending on the action selected. The first indicates the *location* at which the edit will be applied; the second, when applicable (i.e., for append, swap, or replace operations, indicates the code to be inserted). This collection of sequences of edit operations comprises the search space. Naively, this space is very large. For instance, a faulty program with 100 statements has a search space of $2.7 \cdot 10^{13}$ possible 3-edit patch solutions [15]. As this search space grows exponentially with considered lines of code, any technique proposing to traverse this space must do so efficiently, distinguishing between possible solutions to identify promising search areas and, ideally, a fix. Heuristic techniques like fault localization [1, 7] and simple semantic reasoning [24] can reduce this search space somewhat, but overall the problem remains difficult.

One important and underexplored area for possible improvement to the GP for repair search problem is the fitness function. GenProg's *canonical fitness function* scores each individual as the weighted sum of the test cases that the modified program passes.

[1] GenProg is available at https://squareslab.github.io/genprog-code/.

GECCO '18, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5618-3/18/07…$15.00
https://doi.org/10.1145/3205455.3205566

For efficiency, initially only a random subsample of the positive (initially passing) tests and all the negative (initially failing) tests are applied [5]. If the individual succeeds on all the tests in the subsample, the remaining positive test cases are executed. A variant is considered a repair when all test cases pass [10].

Regardless of the random sample, two individuals passing the same proportion of passing versus failing test cases receive the same fitness score. This fundamentally produces fitness score *plateaus*, where fitness fails to truly distinguish between individuals of otherwise varying quality. This has long been observed in the program repair domain [6]. These plateaus can paralyze the search, or render it purely random, since many points in the search space have the same observed fitness value. An objective function that does not effectively differentiate between possible solutions accordingly cannot guide the search for promising regions.

We propose to minimize these plateaus, increasing the granularity of the fitness information, by instrumenting the program to collect information about the intermediate program state throughout test case execution. We call this dynamically collected information *checkpoints*, and we use them to distinguish between the behavior of different potential solutions. We conjecture that this approach to evaluate fitness can, in a novel way, reduce the number of fitness plateaus in GP-based Automated Program Repair and increase search efficiency.

The development of our research was guided by the following research questions:

- **RQ1:** Does aggregating checkpoints' data in the fitness function successfully reduce the incidence of plateaus in the patch-based program repair search space?
- **RQ2:** Does the checkpoints-based fitness function improve the expressiveness and the efficiency of GenProg, a GP-based automated program repair technique?

Our primary contribution is thus a novel fitness function for GP-based Automated Program Repair problems that collects intermediate state along test case executions. While reducing plateaus is no guarantee of improving the search landscape, our experimental results showed that our novel fitness function reduces plateaus while increasing, with statistical significance, the expressiveness (finding more repairs) and the efficiency (lowering the number of evaluations to find a repair) of the search. We show that this fitness function increases the granularity of the fitness evaluation, better differentiating between individuals, and reducing the number of plateaus in the search space. We also show that, practically, our proposed fitness function increases the expressiveness and efficiency of GenProg.

The remainder of this paper is organized as follows. Section 2 introduces key concepts that are necessary to understand our contribution and presents a set of related work that improved GenProg's algorithm. Section 3 presents the main contribution of this paper, using an illustrative example. Section 5 presents and analyzes our experiments. We conclude and discuss opportunities for future work in Section 6.

2 BACKGROUND AND RELATED WORK

This section introduces key background to understanding our primary contributions, and presents related work, primarily focussed on previous research on improving GenProg's underlying GP-based program repair algorithm.

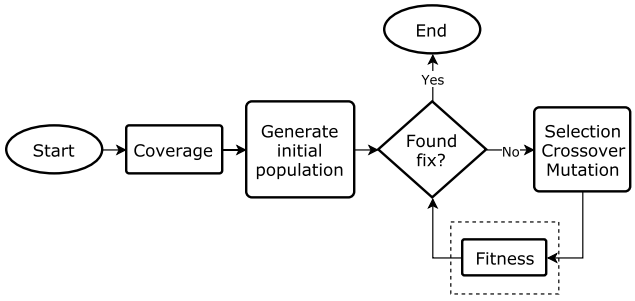

Figure 1: GenProg's algorithm.

2.1 Automated Program Repair

Automated Program Repair aims to fix software bugs with minimal or no human intervention. As a domain, such techniques seek to lower maintenance costs and enable systems to be more resilient to bugs and unexpected situations [4]. The fix is often encoded as a set of incremental changes to the source code [10, 16]. These changes aim to transform an inadequate behavior (the bug) to the expected behavior. A test suite is commonly used as an oracle to the repair process. We refer to the *negative test cases* as those that encode the bug and are failing on the initial (unmodified program). We refer to the *positive test cases* as those the unmodified program initially pass, and that encode desired behavior that should not regress [13, 23]. A variety of techniques for automatic program repair have been proposed in the literature, especially in the past 5–10 years (e.g., Long and Rinard [14], Mechtaev et al. [17], Perkins et al. [21], Xuan et al. [27], among others). We focus on GenProg (described next), one of a class of techniques for program repair that rely primarily on stochastic search (e.g., among others, Kim et al. [8], Le et al. [9], Le Goues et al. [13]).

2.2 GenProg

GenProg takes as input program source code and a test suite with at least one failing test case. Given this input, GenProg performs an initial coverage analysis, applying a standard statistical fault localization [1, 7] to identify the statements that are more likely to contain the bug in question. It uses this information to weight the statements for modification. The genetic algorithm proceeds over multiple generations. Each individual is a collection of edit operations, or a *patch*. A *fix* is patch that, when applied to the source code, produces a program that passes all provided test cases. That is, such a patch transforms the inadequate behavior into the expected one. New individuals and populations are created using domain-specific instantiation of common GP operators, including Selection, Mutation, and Crossover. Figure 1 shows the algorithm expressed as a flowchart.

The fitness function evaluates each variant as the weighted sum of the test cases that a modified program passes, i.e., it applies each test case and it verifies whether the modified program displays the desired behavior. This evaluation only considers the final output

of the program. A variant is considered a repair when all test cases pass [10]. We refer to this function as GenProg's *canonical fitness*.

To date, GenProg's representation and fitness function have been improved to increase efficiency and scalability, but the overall model remains the same. The two primary improvements include (1) replacing the Abstract Syntax Tree (AST) representation with a patch representation, and (2) introducing a fitness subsampling technique:

(1) **Patch representation.** The initial representation of a variant in this GP domain [25] was the entire AST of the original program with modifications. However, this representation does not scale to handle large programs. This motivated the patch-based representation, which treats an individual as a list of changes with respect to the original program. To evaluate a variant, the changes are applied to a copy of the input program, and producing a modified version that is run on the provided test cases. The patch representation is slightly but measurably more effective than its AST predecessor [10].
(2) **Fitness sampling.** Many programs have a test suite consisting of many test cases; executing the whole suite for every variant is costly and naive, since most test behavior is unaffected by a given change. Thus, the other major improvement to the canonical fitness function in prior work introduced a subsampling technique where initially only a subsample of the positive (initially passing) tests and all the negatives (initially failing) tests are applied [5]. If the individual succeeds on all the tests in the subsample, the remaining positive test cases are executed.

We can model the GenProg search space as a search-based problem as follows. Let op be an operation search space composed of three elements, append, swap,[2] and delete; p is the pool of statements used to compose a fix; s are the statements in the source code; and n the number of edit operations that are necessary to constitute a fix. The search space cardinality is then given by:

$$(|op| \cdot |p| \cdot |s|)^n$$

As an example of how big this search is space is, a program with one million statements (p), of which 5% may possibly embody a fix (s) constituted by 3 edit operations (n), results in a search space with $3.375 \cdot 10^{30}$ possible solutions.

Given the size of the search space and the cost involved in testing each possible solution (compiling and applying test cases), even with domain-specific restrictions and probabilities (informed by fault localization, for example), ideally, the genetic algorithm search should be carefully guided to avoid local maxima, or search *plateaus*.

2.3 Related Work

To the best of our knowledge, the only previous technique that considers an alternative fitness function besides test cases is HDRepair [9]. HDRepair explores single-mutation fixes and scores intermediate solutions via similarity to a database of historical patches, in the interest of improving patch quality. While promising, this approach does not address the established problem of fitness plateaus in evolutionary program repair for patches of arbitrary length [6]. Beyond this, we focus on previous work that improves other elements of GenProg's genetic algorithm, or discusses guidelines for fitness functions in the Program Repair context.

Oliveira et al. [20] presents new crossover operators that explores the locality of the search spaces in the patch, increasing GenProg expressiveness. Subsequent work further reformulated the representation (subpatch), crossover, and mutation operators, with experimental results showing improvement in search success and efficiency [19].

As noted above, Fast et al. [5] first presented the subsampling approach for fitness evaluation (now canonical in GenProg applications). They proposed a dynamic predicate-based fitness function to smooth GenProg's search landscape. It was designed to approximate the distance between an individual and a potential repair. The distance metric was based on a linear regression model that was trained with 1.772 points and used fourteen known repairs that served as reference. They further sought to establish general guidelines for a generic fitness function based on formal specifications.

Le Goues et al. [10] demonstrated the importance of the patch-based representation over the AST-based version used in the previous work [25], discussed above.

None of these works are concerned with modeling a new fitness function for GenProg. The Checkpoints fitness function could be used to aggregate other GA operators, such as those proposed by Oliveira et al. [20]. The modifications in the fitness function presented by Le Goues et al. [10] are related to performance and it does not reside within our scope. Although Fast et al. [5] proposed a method to smooth the search landscape, it does not generalize beyond a single case study and it relies on linear regression and an excess of training data to create a metric to differentiate individuals. Our proposed fitness function generalizes the high-level idea of a source code inspection to differentiate among individuals without relying on a model that needs an excess of training data. It also scales from small to real-world and large programs.

3 CHECKPOINTS

Instrumenting source code to collect run time data when applying test cases is a form of dynamic analysis. We refer to these special instructions as *checkpoints*. The data collected by these checkpoints allow us to inspect dynamic memory state, values that variables assume through the course of the program execution. Our high-level goal is to "smooth" GenProg's search landscape by composing a new fitness function that leverages data from dynamic analysis to better differentiate among individuals. This section presents our main contribution: an improved fitness function based on source code checkpoints that seeks to mitigate plateaus formed by the canonical fitness function.

Our novel fitness function instruments every individual's phenotype, its source code, with debug statements we call *checkpoints*. We use these checkpoints to track the values of numeric local variables that are involved in control-flow statements, such as variables inside loops, conditionals, or any other variables that might lead the execution of the program to different paths in the AST.

[2] Some literature introduces a comparable replace operator instead of swap.

Algorithm 1: Checkpoints' Algorithm.

Input: individual
Output: totalScore

1: $faultyStmts \leftarrow$ [*GetListOfFaultyStmts*]
2: $debugVariables \leftarrow$ [*individual.debugVariables*]
3: $totalScore \leftarrow 0$
4: $posScore \leftarrow 0$
5: $negScore \leftarrow 0$
6: **for** $tc \in testCases$ **do**
7: $score \leftarrow 0$
8: $faultScore \leftarrow 0$
9: $maxScore \leftarrow |debugVariables|$
10: **for** $var \in debugVariables$ **do**
11: **if** $tc.isPositive$ **and** $var.out = var.origOut$ **then**
12: $score \leftarrow score + 1$
13: **end if**
14: **if** $tc.isNegative$ **and** $var.out \neq var.origOut$ **then**
15: $score \leftarrow score + 1$
16: **end if**
17: **if** $var.out \neq var.origOut$ **and** $var.stmt \in faultyStmts$ **then**
18: $faultScore \leftarrow faultScore + 1$
19: **end if**
20: **end for**
21: **if** $tc.isPositive$ **then**
22: $posScore \leftarrow posScore + max(\frac{score}{maxScore}, tc.canonicalOut)$
23: **else**
24: **if** $score \neq maxScore$ **then**
25: $wr = 0.4 * \frac{faultScore}{|faultyStmts|}$
26: $negScore \leftarrow negScore + max(0.5 + wr, tc.canonicalOut)$
27: **else**
28: $negScore \leftarrow negScore + tc.canonicalOut$
29: **end if**
30: **end if**
31: **end for**
32: $totalScore \leftarrow \frac{posScore + negScore}{|tc.posTcs| + |tc.negTcs|}$
33: **return** $totalScore$

These locations are determined statically in a pre-processing analysis pass. For every test case then executed in fitness evaluation, the checkpoints logs both statement coverage information and the memory state at each instrumented point. This provides complete information for both program control- and data-flow.

Algorithm 1 shows how we use the set of faulty statements (identified by the fault localization step) and the logged coverage and memory state information to compose the *checkpoints metric*, representing individual fitness. For each test case, our metric counts changes in tracked variable values. We expect that the changes made by an individual to the original source code generally do not affect variables touched by the passing (positive) test cases. However, we do expect the variables in faulty statements to change. Line 11 counts changes on the positive test cases; Line 14, the negative. Changes in statements covered by the failing test cases are also tracked (Line 18).

Listing 1: Buggy prime computation.

```
1  int is_prime( int n ) {
2      if ( n % 2 == 0 ) || (n % 3 == 0) {
3          return n == 2 || n == 3;
4      }
5      int d = 5;
6      while ( d * d <= n ) {
7          d += 2;
8          if ( n % d == 0 )
9              return 0;
10         d += 4;
11 +       d += 2; // candidate insertion
12     }
13     return 1;
14 }
```

After counting the changes, we score the behavior for every considered test case. If the test case passed, we do not need to look the internals of the individual, so we score it with 1. Positive test cases are scored according to the proportion of variable values that did not change with respect to the unmodified program (Line 22). For negative test cases, we provide a bonus score (of 0.5) for individuals that only change variable values along the faulty path (Line 26). We complete the negative score with an additional bonus of 0.4+ *% of change* that measures how much change was made in the desired location (Line 25). The positive and the negative scores are summed. Our goal in selecting these constants (0.4 and 0.5) is to induce some change in the faulty code without leading to new bugs. These constants thus heuristically seek to balance the competing goals of changing undesired behavior (0.4) while reinforcing patches that do not break previously desired behavior (0.5).

Finally, the canonical fitness score is calculated with information from the output of each test case (Line 32). Our proposed fitness calculates the score of an individual as the weighted sum of the canonical fitness and the checkpoints score ($0.7 \cdot canonical + 0.3 \cdot checkpoints$). These weights are chosen to induce the search to privilege the macro behavior (i.e., if the individual passed the test cases), while including the internal differentiation element.

4 ILLUSTRATIVE EXAMPLE

We now present an example of how to calculate fitness for a single checkpoint in an illustrative example[3]. Consider the example buggy code in Listing 1, which should return 0 if input n is prime and 1 otherwise. One way to fix this code is to copy the if-block on line X and insert it at line Y. We will perform the fitness computation for a variant that does not fix the bug (copying the computation on line N to line M, shown as an insertion in the code sample). We consider two test cases: (1) $n = 123 \Rightarrow 0$ and (2) $n = 125 \Rightarrow 0$. Both the original program and the considered variant pass (1) but fail (2).

4.1 Check pointing and applying test cases

The first step is to place checkpoints relative to variables of interest, namely those effecting control-flow statements. Such statements

[3]The concept generalizes naturally to an arbitrary number of checkpoints.

Listing 2: A sample statement with *checkpoints*.

```
// log_checkpoint(stmt_id, var_name, var_state);
log_checkpoint("7", "d", d);
log_checkpoint("7", "n", n);
while( d * d <= n ) {
    { ... }
}
log_checkpoint("7", "d", d);
log_checkpoint("7", "n", n);
```

Table 1: Reference values and Memory states for the sample variables.

	Var	Neg. Ref.	Neg. Out
Before	*d*	5	5
	n	125	125
After	*d*	17	13
	n	125	125

are logged with checkpoints before and after execution. Listing 2 shows such debug instructions inserted in one statement of our example.

Table 1 provides the intermediate values tracked by the checkpoints. Column *Neg. Ref.* shows intermediate values of the original source code for the negative test case. Column *Neg. Out* shows the intermediate values of the (incorrect) variant for the negative test case. The positive test case does not produce checkpoints because it does not touch the instrumented statement.

The canonical fitness uses only test case success or failure computed from program output. So, as we apply the test cases and gather data from the internal states, we can also store the outputs and compute the canonical score (the *tc.canonicalOut* variables in the checkpoints algorithm).

4.2 Calculating the fitness score

With the data collected in the previous step, we are able to apply Algorithm 1. Assume fault localization has identified lines 8 and 9 as faulty. The algorithm tracks variables *d* and *n*, implicated/used in these lines.

The first loop of the algorithm (Line 6) iterates over the result of each test case. When analyzing the positive test cases (Line 11), *score* is incremented twice because the statement did not deviate from the reference values. For the negative test case (Line 14), *score* will be incremented only by 1, because it is different from the reference values in one variable. The variant also presented different values of a variable inside a faulty statement, so *faultScore* is incremented to 1.

Now we calculate the *posScore* and *negScore* variables, that will compose the final score (*totalScore*). In our example, as we only have one positive test case and it kept positive in our snippet, the *posScore* will receive 1 as its value (corresponding to the *tc.canonicalOut* variable). However, as the negative score remained negative, we do not score it with canonical output. We bonus the snippet with 0.5+ *wr* (Line 25). The *wr* bonus will receive $0.4 \cdot \frac{1}{2}$. The final *negScore* will be $0.5 + (0.4 \cdot \frac{1}{2}) = 0.7$. The final checkpoints score (*totalScore*) is thus

$$\frac{1 + 0.7}{1 + 1} = 0.85$$

The individual fitness score is a weighted sum of the canonical score and the checkpoints score:

$$0.7 \cdot 1 + 0.3 \cdot 0.85 = 0.955$$

The more lines of code, the more likely our approach is to better differentiate among individuals, as it will leverage the internal states to compare them.

5 EXPERIMENTS

This section describes our evaluation and results. The results of our experiments can be found at https://github.com/eduardodx/gecco2018-checkpoints.

5.1 Setup

Dataset. Our goal in benchmark selection is to have a diversity of projects and project size, to support arguments of applicability and generalizability. We conducted our experiments on subsets of two well-known benchmarks for Automated Program Repair, IntroClass and ManyBugs [12], shown in Table 2. IntroClass consists of six small programming assignments developed by undergraduate students of a C programming course. ManyBugs is a collection of nine large and mature open source software. In the IntroClass dataset, the course grading system stored all intermediate versions of code produced by students. In our experiments we used only the last version that contained a bug and had at least one failing and one passing test case. ManyBugs was introduced in 2015, but the scenarios were extracted from 2012 in an environment that can be hard to reproduce. We selected three projects we were best able to reproduce: Gzip, a data compression utility; Libtiff, an image processing library; and Wireshark, a network packet analyzer.

Settings. We run GenProg on each IntroClass bug with 20 seeds. Each execution (seed) was configured to run 30 generations with 40 individuals and 4 elitists each generation. Each ManyBugs scenario had 10 executions with 10 generations, 40 individuals each generation, and 4 elitists per generation. Those are same parameters as Le Goues et al. [12], with exception of the elitism component. The search budget (1,200 fitness evaluations for IntroClass executions and 400 evaluations for ManyBugs executions) is due the prohibitive cost of each fitness evaluation, where it is necessary to compile and run an entire test suite for each individual (especially prohibitive on the ManyBugs scenarios).

We ran our experiments on an Intel® Xeon® CPU E5-2660 v3 @ 2.60GHz with 40 Threads and 128GB RAM.

Research Questions. We seek to answer the following questions:

- **RQ1:** Does aggregating checkpoints' data in the fitness function successfully reduce search space plateaus?
- **RQ2:** Does the checkpoints-based fitness function improve the expressiveness and the efficiency of GenProg, a GP-based automated program repair technique?

Table 2: IntroClass and Manybugs benchmarks
With information from [12].

Benchmark	Program	LOC	Bugs	Description
IntroClass	Checksum	13	19	checksum of a string
	Digits	15	21	digits of a number
	Grade	19	30	grade from score
	Median	24	25	median of 3 numbers
	Smallest	20	25	min of 4 numbers
	Syllables	23	22	count vowels
ManyBugs	Gzip	491k	5	data compression utility
	Libtiff	77k	24	image processing library
	Wireshark	2,814k	8	network packet analyzer

5.2 RQ1: Plateaus

To answer our first Research Question, we analyzed each fitness function's ability to differentiate (via fitness score) individuals with distinct genetic material, thereby reducing the number of fitness plateaus. Table 3 shows the percentage of individuals in a population that share a fitness value with at least one other individual in that population. Lower is better, signifying a function that more effectively differentiates distinct individuals. We run Wilcoxon signed-rank test [26] on every set of results to test for statistical significance ($p < 0.05$) whether checkpoints fitness is lesser than canonical fitness.

Because our proposed fitness function leverages data from internal program state rather than test case passing/failure alone, it more effectively differentiates candidate solutions. Table 3 demonstrates: in a statistically significant manner, the checkpoints fitness function more effectively differentiated individuals on 5 out of 6 IntroClass problems and 2 out of 3 ManyBugs problems.

Overall, the checkpoints-based fitness better differentiated 83.3% of the problems in IntroClass with statistical significance. From this subset, the best case was Smallest, where it could differentiate 14.88% more than the canonical fitness function. The worst set of problems was taken from the Grade program, where it only differentiated 0.72% more than the canonical fitness. On average, our proposed fitness reduced the number of fitness plateaus on IntroClass problems by 5.52% overall.

The results for real-world programs in Manybugs showed that checkpoints fitness reduced plateaus in 67% of the problems. It reduced plateaus in Gzip and Wireshark by 18.73% and 12.71%, respectively, with statistical significance. Although not presenting statistical significance for Libtiff, its results presented a reduction of 5.21% in the number of plateaus.

Overall, our new fitness function reduces the number of fitness plateaus in most of the considered problems, ranging from small and synthetic problems to large and real-world programs.

5.3 RQ2: Expressiveness and Efficiency

We define *expressiveness* as the search algorithm's ability to find repairs as a proportion of the successful random seeds[4]. For *efficiency*, we consider the number of fitness evaluations required to find a fix. So, we answer the second Research Question by comparing the expressiveness and efficiency of our proposed fitness function against the canonical function. Table 4 shows results.

Expressiveness. The "Bugs fixed" and "Runs w. fix" columns in Table 4 measure search expressiveness; higher is better. The proposed fitness increases the number of repairs found for bugs in 50% of the IntroClass problems. It also increases the number of repairs per seed in 67% of the cases, ranging from an increase of 3.65% (Smallest) to 76.92% (Syllables), with an average of 35.68% of increase per problem.

Our most interesting results are on ManyBugs. As they are real-world programs with thousands of lines of code, their bugs are more complex to solve automatically. Our fitness function leverages the characteristics of these benchmarks, problems with more test cases and more lines of code, to generate more data that composes the final score of each variant in a more detailed manner and with fewer plateaus. The resulting population is better differentiable by the selection operator and individuals with good genetic material are carried to the next generations more frequently. The search is, then, less likely to stay stagnant in certain areas of the search space and can traverse it more efficiently.

Our proposed function found a repair to a Gzip bug where the canonical fitness failed. This means a 100% increase in the expressiveness to the Gzip results. Gzip also saw the number of repairs per run increased as well, up to a factor of 150%. For the other scenarios, the checkpoints fitness found repairs for the same bugs as the canonical function, but with more repairs per run. The average increase in repairs per seed for our proposed fitness function is 13.29%. These results show that our proposed fitness can guide the search to new areas of the search space (finding repairs to previously unsolved bugs) and it is more robust in terms that it increases the likelihood of finding a repair given the same set of seeds.

[4]We use the term expressiveness because effectiveness does not apply in our context. Effectiveness would apply if we had a semantic oracle that could ensure a repair has the correct semantic with no overfitting. As we are bounded by test cases as an oracle, and they are of varying quality, we cannot measure effectiveness.

Table 3: IntroClass and Manybugs: the average percentage of individuals in a population that have at least one other with the same fitness score.

Benchmark	Program	Canonical		Checkpoints		*p*-value
		Avg	StdDev	Avg	StdDev	
IntroClass	Checksum	96.55%	15.46%	95.75%	17.46%	0.1815
	Digits	91.75%	20.48%	90.23%	22.23%	0.0026
	Grade	99.92%	0.17%	99.20%	0.49%	$p < 0.001$
	Median	94.52%	18.26%	89.30%	23.42%	$p < 0.001$
	Smallest	76.73%	32.46%	61.85%	34.47%	$p < 0.001$
	Syllables	98.13%	8.58%	92.88%	15.07%	$p < 0.001$
ManyBugs	Gzip	99.16%	2.57%	80.43%	19.05%	$p < 0.001$
	Libtiff	78.26%	30.65%	73.05%	31.31%	0.0752
	Wireshark	93.62%	16.73%	80.91%	19.81%	$p < 0.001$

Efficiency The "Avg. Evals" columns in Table 4 shows the average number of evaluations to find a repair for each fitness function. Lower is better. The checkpoints fitness improved the search efficiency for every problem in IntroClass. The speedups ranged from 2.01 (Digits) to 5.98 (Smallest). The average speed up of our proposed fitness function over the canonical function is 3.9.

The speedups are even more significant on ManyBugs. Our proposed fitness function achieved a 63.10 speedup for Wireshark, a program with hundreds of test cases and over two million lines of code, and 40.58 for Libtiff, another program with 70 thousand lines of code. This achievement is important for GP-based Automated Program Repair techniques because it makes it possible to work with larger programs or to increase the number of generations and the population, thus better exploring the search space.

Our results show that both expressiveness and efficiency are improved when using checkpoints fitness function. Its use leads the search algorithm to be more expressive (repairing more bugs), robust (more likely to find a repair in a run), and efficient (lower evaluations necessary to find a repair).

6 CONCLUSION

GenProg, a behavioral method for Automated Program Repair, has a canonical fitness function that is not able to properly differentiate between individuals, evaluating variants with distinct genetic material with the same fitness score (plateaus). This can paralyze the search in certain regions of the search space. We propose a new fitness function based on intermediate code checkpoints, aiming to better differentiate between individuals. Our experimental results showed, with statistical significance, that our proposed fitness function reduces the occurrence of plateaus, better guiding the genetic algorithm; finds more repairs, both in number of bugs and runs; and increases the efficiency of the search.

As for future work, we aim to reduce the noise in the Checkpoints fitness by placing debug statements only in statements that are more likely to contain bugs, as shown by the fault-localization process. We also plan to include another term to the fitness function that asses source code quality based on static analysis.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC '06)*. Washington, DC, USA, 39–46. https://doi.org/10.1109/PRDC.2006.18

[2] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. 1998. *Genetic programming: an introduction*. Vol. 1. Morgan Kaufmann San Francisco.

[3] MITRE Corporation. 2018. The Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/. (2018). (Visited on 01/10/2018).

[4] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic Repair of Buggy if Conditions and Missing Preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA 2014)*. ACM, New York, NY, USA, 30–39. https://doi.org/10.1145/2593735.2593740

[5] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2010. Designing Better Fitness Functions for Automated Program Repair. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO '10)*. ACM, New York, NY, USA, 965–972. https://doi.org/10.1145/1830483.1830654

[6] Stephanie Forrest, Westley Weimer, ThanhVu Nguyen, and Claire Le Goues. 2009. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference*, Franz Rothlauf (Ed.). ACM, 947–954.

[7] James A. Jones, Mary Jean Harrold, and John T. Stasko. 2001. Visualization for Fault Localization. In *in Proceedings of ICSE 2001 Workshop on Software Visualization*. Toronto, ON, Canada, 71–75.

[8] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *International Conference on Sofware Engineering (ICSE '13)*. 802–811.

[9] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *Software Analysis, Evolution, and Reengineering (SANER '16)*. 213–224.

[10] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 3–13. https://doi.org/10.1109/ICSE.2012.6227211

[11] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *Software quality journal* 21, 3 (2013), 421–443.

[12] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering (TSE)* 41, 12 (December 2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513 http://dx.doi.org/10.1109/TSE.2015.2454513DOI: 10.1109/TSE.2015.2454513.

[13] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on* 38, 1 (Jan 2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[14] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Symposium on Principles of Programming Languages (POPL '16)*. 298–31.

[15] Matias Martinez and Martin Monperrus. 2012. *Mining repair actions for guiding automated program fixing*. Ph.D. Dissertation. Inria.

Table 4: IntroClass and Manybugs: Expressiveness and Efficiency.

Benchmark	Program	Bugs	Runs	Canonical fixes			Checkpoints fixes		
				Bugs fixed	Runs w. fix	Avg. evals.	Bugs fixed	Runs w. fix	Avg. evals.
IntroClass	Checksum	19	380	2	27	240.37	**3**	**32**	**60.68**
	Digits	21	420	7	55	189.63	**8**	**79**	**93.93**
	Grade	30	600	0	0	0	0	0	0
	Median	25	500	6	**76**	164.68	6	75	**31.13**
	Smallest	25	500	25	466	637.52	25	**483**	**106.56**
	Syllables	22	440	1	13	156.92	**2**	**23**	**69.39**
ManyBugs	Gzip	5	50	1	4	**54.75**	**2**	**10**	66.80
	Libtiff	24	240	17	114	2220.65	17	**128**	**54.71**
	Wireshark	8	80	2	19	1104.42	2	**20**	**17.5**
Total		179	3,210	61	774	4,768.94	65	850	500.7

[16] Matias Martinez and Martin Monperrus. 2013. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. *Empirical Software Engineering* Online First (Sept. 2013). https://doi.org/10.1007/s10664-013-9282-8 Accepted for publication on Sep. 11, 2013.

[17] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *International Conference on Software Engineering (ICSE '16)*. 691–701.

[18] Martin Monperrus. 2018. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 17.

[19] Vinicius Paulo L. Oliveira, Eduardo Faria de Souza, Claire Le Goues, and Celso G. Camilo-Junior. 2018. Improved representation and genetic operators for linear genetic programming for automated program repair. *Empirical Software Engineering* (25 Jan 2018). https://doi.org/10.1007/s10664-017-9562-9

[20] Vinicius Paulo L. Oliveira, Eduardo F. D. Souza, Claire Le Goues, and Celso G. Camilo-Junior. 2016. Improved Crossover Operators for Genetic Programming for Program Repair. In *8th International Symposium Search Based Software Engineering (SSBSE 2016)*, Federica Sarro and Kalyanmoy Deb (Eds.). Springer International Publishing, 112–127. https://doi.org/10.1007/978-3-319-47106-8_8

[21] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. 2009. Automatically patching errors in deployed software. In *ACM Symposium on Operating Systems Principles (SOSP '09)*. 87–102.

[22] Tricentis. 2017. Software Fail Watch: 5th Edition. (2017). https://www.tricentis.com/software-fail-watch/

[23] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. 2010. Automatic Program Repair with Evolutionary Computation. *Commun. ACM* 53, 5 (May 2010), 109–116. https://doi.org/10.1145/1735223.1735249

[24] W. Weimer, Z. P. Fry, and S. Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 356–366. https://doi.org/10.1109/ASE.2013.6693094

[25] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *International Conference on Software Engineering (ICSE '09)*. 364–374.

[26] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics bulletin* 1, 6 (1945), 80–83.

[27] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55.

[28] Michael Zhivich and Robert K Cunningham. 2009. The Real Cost of Software Errors. *IEEE Security & Privacy* 7, 2 (March 2009), 87–90. https://doi.org/10.1109/MSP.2009.56