Refining Fitness Functions for Search-Based Program Repair

Anonymous

***Abstract*—Debugging is a time-consuming task for software engineers. Automated Program Repair (APR) has proved successful in automatically fixing bugs for many real-world applications. Search-based APR generates program variants that are then evaluated on the test suite of the original program, using a fitness function. In the vast majority of search-based APR work only the Boolean test case result is taken into account when evaluating the fitness of a program variant. We pose that more fine-grained fitness functions could lead to a more diverse fitness landscape, and thus provide better guidance for the APR search algorithms.**

We thus present 2Phase, a fitness function that also incorporates the output of test case failures, and compare it with ARJAe, that shares the same principles, and the standard fitness, that only takes the Boolean test case result into consideration. We conduct the comparison on 16 buggy programs from the QuixBugs benchmark using the Gin genetic improvement framework. The results show no significant difference in the performance of all three fitness functions considered. However, Gin was able to find 8 correct fixes, more than any of the APR tools in the recent QuixBugs study.

***Index Terms*—Software Engineering, Genetic Programming, Genetic Improvement, Program Repair, Fitness Function**

I. Introduction

Software engineers commonly make mistakes in designing and writing pieces of code. In 2016 alone, software failures cost the worldwide economy approximately 1.1 trillion dollars [1]. Software bugs are faults, flaws or errors in the software system that lead to unexpected behaviour or incorrect outputs. Debugging software is extremely expensive — it costs half of the programming time [2] — raising the need for effective and cheap bug fixing techniques.

Automated Program Repair (APR) [3], [4] has become a popular research field in recent years, due to a growing number of available tools and concrete examples of real-world repairs. There are two main types of APR approaches: semantic-based APR [5], [6], that synthesises repairs based on semantic information of the program (via symbolic execution and constraint solving), and search-based APR, which we focus on in this work.

A typical search-based APR tool, such as GenProg [7], [8], generates program variants to fix a given faulty program using its test suite as an oracle for software quality. It iteratively generates sets of variants through source code modifications until a variant is found that passes every test case. The search space of software variants is typically navigated using a search heuristic, such as genetic programming or local search. This type of approach is also known as *generate-and-validate*. In the vast majority of search-based APR work the fitness function only takes the number of passing and failing test cases into account. This can lead to a large number of equivalent program variants sharing the same fitness values.

We present a more fine-grained fitness function using not only the binary result of test cases but also their output in case of a failure. Our goal is to create a more diverse fitness landscape with fewer plateaus, to provide better guidance for the underlying search algorithm. Indeed, most unit testing frameworks, such as JUnit, report the expected and actual value of a test case failure. Our new fitness function will thus differentiate between program variants that fail due to compilation or runtime errors and those that simply produce different output, the latter being considered better. We also measure the difference between the expected and actual output value of the failing test cases.

In this paper, we compare our proposed fitness function, which we call 2Phase, with two other: the often used fitness function of GenProg and the fitness function of the very recent ARJAe tool [9], which also implements a more fine-grained fitness variant. Experiments were conducted using 16 programs of the QuixBugs benchmark, as for these test-suite adequate patches were found by existing APR tools [10]. We also generated additional test cases using EvoSuite for validation purposes. The experiments were conducted using the Gin framework [11], [12], as suggested in [13], which inspired our work.

Our results show that overall there is no significant difference between effectiveness and efficiency of the three fitness functions on our dataset. Moreover, we observed no significant difference in the distribution of fitness values between the three fitness functions. This is in contrast to the results obtained by authors of ARJAe on the Defects4J benchmark set [9]. However, our APR framework was able to find test-suite adequate patches for 11 programs, with 8 being correct fixes (more than any of the APR tools in the recent QuixBugs study [10][1]).

II. Fitness Functions in Search-Based APR

Ever since the introduction of GenProg [8] search-based APR tools have used Boolean test-case results in fitness calculations for program variant evaluation. We will call such a fitness function as *standard*. Our literature review revealed only two papers that proposed and evaluated more fine-grained fitness functions, which we describe below. We also present our new 2Phase fitness function in this section.

[1]We note here that 10 correct fixes were reported in [14]. However, details of the programs and the fixes were not provided, so we could not make a direct comparison.

A. ARJAe

ARJA [15] is a state-of-art search-based APR tool. It combines multi-objective genetic programming, lower-granularity patch representation and test-filtering techniques to improve on a GenProg-based APR approach.

ARJAe [9] extends the ARJA framework with a new fitness function. In addition to the number of test case failures, ARJAe also tracks all assertion methods invoked during the execution of each test case. It then measures the distance between the expected and actual assertion values for each assertion failure. ARJAe defines a distance metric for each assertion type supported by the JUnit `Assertion` class. For example, the `String` comparison function uses the Levenshtein distance (minimal number of single-character insertions, deletions, or substitutions), while for numeric types (e.g., `int`, `double`) absolute difference is used. All distance values are normalised to $[0, 1]$ by the function $v(x) = x/(x+1)$ [16]. ARJAe defines the error ratio $h(x,t)$ as the degree of a program variant x that violates a test case t, with $d(e)$ representing the normalised distance of an assertion e and $E(x,t)$ is the set of all executed assertions in t.

$$h(x,t) = \frac{\sum_{e \in E(x,t)} d(e)}{|E(x,t)|} \tag{1}$$

The fitness function $f(x)$ is finally defined as a weighted sum of all assertion distances, with T_{pos} and T_{neg} the subsets of positive and negative test cases, and $w \in (0,1]$ the weight parameter that introduces bias against the latter.

$$f(x) = \frac{\sum_{t \in T_{pos}} h(x,t)}{|T_{pos}|} + w * \frac{\sum_{t \in T_{neg}} h(x,t)}{|T_{neg}|} \tag{2}$$

B. 2Phase

A fitness function similar to the ARJAe one has been independently proposed [13], however without being properly implemented and analysed. In order to provide a comparison between the standard fitness and the more fine-grained variants, we provide an implementation for the ftiness function proposed in [13], and call it 2Phase.

2Phase is based on the idea that patches with more passing test cases should always be preferred, while variants with the same number of passing test cases should be compared through *assertion distances*, i.e., the distances between the expected and actual outputs.

Equation (3) details the fitness function, with n and m the number of passing and failing tests, and $\sum_{e \in E} d(e)$ the sum of assertion distances for failing tests. Assertion distances are normalised to $[0, 1)$ using $v(x) = x/(x+1)$, following ARJAe.

$$f(x) = n + \Big(1 - \frac{\sum_{e \in E} d(e)}{m}\Big) \tag{3}$$

The main difference between ARJAe and 2Phase is that 2Phase prioritises the number of test case failures, while ARJAe penalises the assertion distances computed from the original passing test cases with the intuition being to actively avoid breaking test cases that passed for the original software.

```java
class GCD {
    public static int gcd(int a, int b) {
        if (b == 0) {
            return 0;
        } else {
            return gcd(a % b, b); // fix: gcd(b, a % b)
        }
    }
}
```

Listing 1. GCD.java

TABLE I
TEST CASES FOR GCD, WITH OUTPUTS FROM BUGGY CODE (LISTING 1) AND VARIANT `GCD(A % B, B)-> GCD(A % B, A % B)`

Inputs	Output		
(a, b)	buggy	variant	expected
17, 0	17	17	17
13, 13	infinite loop	0	13
37, 600	infinite loop	0	1
20, 100	infinite loop	0	20
624129, 2061517	infinite loop	0	18913
3, 12	infinite loop	0	3

Specifics of all assertion distances are detailed hereafter. For tests leading to exceptions or mismatched types, a maximal penalty `max_int` is applied; otherwise, the assertion distance depends on the output type. 2Phase and ARJAe use the same distance assertion distance metrics. For strings the Levenshtein distance is computed and for numerical outputs the absolute difference is used. For different Boolean values the maximal penalty `max_int` is applied. Finally, for arrays, the assertion distance combines both the length difference and the respective element distances, as given in Equation (4), with len_s and len_l the lengths of the shortest and longest arrays and $d(i)$ the assertion distance between the values at index i of the expected and actual array.

$$d_{arr}(x) = (len_l - len_s) + \frac{\sum_{i=0}^{len_s} d(i)}{len_s} \tag{4}$$

For example, the distance between the arrays [0,1,2] and [1,3,5,7] would be $(4-3) + \frac{1}{3} \times \big(v(1) + v(2) + v(3)\big) \approx 1.639$.

We illustrate 2Phase on the faulty program *GCD* shown in Listing 1, for which five of the six test cases given in Table I fails. The fitness is then $1 + (1 - \frac{5}{5} \times v(\texttt{max\_int})) \approx 1.000$. Let us consider a program variant in which the second argument of the recursive call, `b` is replaced with `a % b`. The fitness is then $1 + (1 - \frac{1}{5} \times (v(13) + v(1) + v(20) + v(18913) + v(3))) \approx 1.174$. With a higher fitness, this variant can be preferred to the original program and, as an intermediate step, may lead to the fix in which the first argument of the recursive call is in turn modified to `b`.

C. Checkpoints

De Souza et al. [17] introduced Checkpoints, an approach that tracks all numerical values, which are then evaluated during fitness computation. The proposed fitness aims to provide a more fine-grained technique for search-based APR approaches to distinguish between software variants and avoid plateaus in the search landscape by monitoring intermediate program states. Checkpoints are placed around statements that contain numerical variables and, in case of a failing test case, distances between the computed values and the values of the original faulty program are computed. The final fitness function combines the original test results and the checkpoints distance information.

We chose not to evaluate this approach, as it has shown little to no improvement over GenProg's default fitness function. Moreover, our benchmark set contains few instances of useful checkpoint application, and thus the overhead of keeping track of numeric values (mostly those being for/while loop indices) would have been wasteful.

III. Methodology

Our goal is to evaluate the effectiveness of various fitness functions in search-based APR for finding bug fixes. In particular, we compare the standard one, as implemented in GenProg, with two more-fine grained variants: ARJAe and 2Phase. We aim to answer the following research questions:

RQ1: How effective are the various fitness functions at providing a diverse set of patches, thus avoiding large fitness plateaus?
In RQ1, we compare the diversity of patches generated using the three fitness functions.

RQ2: How effective and efficient are the various fitness functions at finding test-suite adequate patches?
In RQ2, we compare the effectiveness and efficiency of the three fitness functions in finding test-suite adequate patches. More specifically, we investigate: (1) the number of experimental trials producing at least one test-suite adequate patch, (2) the number of total and unique test-adequate patches produced, (3) the number of evaluations to produce the first test-suite adequate patches (if any exists), and (4) the minimum number of edits to find a test-suite adequate patch.

Metrics (1) and (2) represent the effectiveness of a fitness function. For (2) uniqueness is determined using equality between sequences of edits, as the very large number of patches makes semantic manual investigation of every patch impractical. Metrics (3) and (4) measure the efficiency of a fitness function, i.e., how quickly test-suite adequate patches are found.

RQ3: How well do the test-suite adequate patches generalise to unseen tests?
Recent research [18], [19] reports that a patch that passes all test cases might still be incorrect if test cases fail to exploit all unexpected software behaviour. This is often regarded as the patch overfitting problem [20].

In RQ3, we validate the correctness of test-suite adequate patches in two steps. First, we use EvoSuite to automatically generate new test cases for each of our programs, using the correct program versions provided in the QuixBugs benchmark. Then, we manually evaluate a selection of generated patches that pass both the training set (used during the repair process) and the test set (held-out EvoSuite tests).

Algorithm 1: Gin's GP Search Algorithm

```
Input: faulty program OrigProgram
Input: number of generations Gen
Input: population size PopSize
Output: the set of patches produced Set
Pop ← ∅
repeat PopSize times
    add mutate (clone of OrigProgram) into Pop
repeat Gen times
    parents ← tournSelect (Pop, OrigProgram)
    offspr ← crossover (parents, OrigProgram)
    newPop ← ∅
    foreach patch in offspr do
        newPatch ← mutate (patch)
        if newPatch.fitness > 0 then
            add newPatch into newPop
        add newPatch into Set
    Pop ← newPop
return Set
```

A. Genetic Improvement Search Process

All three fitness functions (i.e., GenProg, ARJAe, and 2Phase) are implemented using the Gin framework [11], [12], an extensible and modifiable toolbox for GI experimentation.

Algorithm 1 shows the logic of Gin's genetic algorithm for the APR task, derived from GenProg's search function [7]. The mutation operator either appends a random new edit to a patch or removes one from a non-empty patch. The crossover operator combines two parent solutions by concatenating, in both orders, the two sequences of edits before each edit is removed with 50% probability to create the two children patches. The tournament selection selects parent patches from the population based on their fitness.

Gin supports source code modifications at the line, statement and expression levels. The experiments will use all types of statement and expression edits available in Gin[2]: deletion, swap, copy, or replacement of statements; swap and replacement of expressions; and replacement of binary (e.g., addition +, Boolean "and" `&&`) and unary (e.g., increment ++) operators. We chose Gin as it contains a richer set of mutation operators than used in previous search-based APR work. Moreover, it also supports non-functional software improvement, hence the experiments can be easily extended to that domain in the future.

B. QuixBugs Benchmark

We use QuixBugs [22] as the program repair benchmark. It consists of 40 Java programs with associated test cases, each faulty program containing a single bug on a single line. A recent empirical study [10] on QuixBugs reports that test-suite adequate patches were found by existing APR tools for 16 of the 40 buggy programs. The features of the 16 chosen problems are given in Table II. The problem sets include eight different types of bugs, such as incorrect array slices and variable swaps.

[2]More edits have been introduced since [21] we ran our experiments.

TABLE II
SELECTED 16 QUIXBUGS PROGRAMS, WITH SIZE OF THE TEST SUITE (ORIGINAL+EVOSUITE) AND BUG AND FAILURE TYPES

Program	Tests	Bug type	Failure type
depth_first_search	5+3	missing line	stack overflow
detect_cycle	5+4	missing condition	null pointer
find_in_sorted	7+9	off-by-one error	stack overflow
get_factors	11+3	wrong array slice	wrong output
hanoi	7+6	incorrect variable	wrong output
is_valid_parenthesization	3+4	incorrect variable	wrong output
knapsack	10+6	incorrect operator	wrong output
levenshtein	7+4	off-by-one error	wrong output
lis	28+4	missing expression	wrong output
mergesort	13+8	incorrect operator	stack overflow
next_permutation	8+8	incorrect operator	wrong output
powerset	5+3	incorrect variable	wrong output
quicksort	13+6	incorrect operator	wrong output
rpn_eval	6+7	variable swap	wrong output
shortest_path_lengths	4+10	variable swap	wrong output
sqrt	7+5	incorrect arithmetic	infinite loop

We noticed that not all programs in the benchmark had passing tests, thus we extended the test suite, which was later also added to the official QuixBugs benchmark[3].

C. Experimental Setup

For a given QuixBugs program and fitness function (i.e., either GenProg, ARJAe, or 2Phase), the search algorithm from Gin is executed as detailed in Algorithm 1. The GP algorithm is run with a budget of 10 generations with a population of 40 individuals, meaning a total of 400 patches evaluated for each trial, following recent work using the GenProg algorithm [23]. Additionally, a cutoff of 1000 millisecond is used for every test case to kill potentially non-terminating variants — long enough so that non-buggy code can terminate successfully.

Experiments were repeated 20 times and conducted independently on the 16 selected QuixBugs faulty programs. All experiments were conducted on a MacBook Air, 1.7 GHz Dual-Core Intel Core i7, 8GB RAM.

IV. RESULTS AND ANALYSIS

In this section we present and analyse our results, providing answers to our three research questions.

A. RQ1: Fitness Plateaus

We first investigate each fitness function's ability to differentiate between the fitness values of all generated patches. Table III shows the average and standard deviation of numbers of patches for which there is at least another distinct patch with the same fitness value. Lower figures are better as they imply that there is more diversity between fitness function values for the generated patches.

The results show that for most programs, the differences between the three fitness functions are within 3%. *shortest_path_lengths* is the only exception, with 2Phase producing a more diverse set of fitness values.

[3]In case of acceptance, the link to the relevant pull request will be provided.

TABLE III
AVERAGE PERCENTAGE OF VARIANTS WITH NON-UNIQUE FITNESS

Program	GenProg	ARJAe	2Phase
depth_first_search	99.9 ±0.1	99.9 ±0.2	99.9 ±0.2
detect_cycle	99.9 ±0.2	100.0 ±0.0	100.0 ±0.0
find_in_sorted	99.9 ±0.1	97.6 ±0.6	97.6 ±0.6
get_factors	99.9 ±0.2	99.9 ±0.2	99.9 ±0.2
hanoi	100.0 ±0.1	100.0 ±0.1	100.0 ±0.1
is_valid_parenthesization	99.9 ±0.1	100.0 ±0.0	100.0 ±0.0
knapsack	99.6 ±0.3	96.3 ±0.8	96.6 ±1.1
levenshtein	99.9 ±0.2	96.9 ±1.0	97.0 ±1.2
lis	98.9 ±0.5	97.2 ±0.8	97.4 ±0.6
mergesort	100.0 ±0.0	100.0 ±0.0	100.0 ±0.0
next_permutation	99.4 ±0.3	99.8 ±0.2	99.8 ±0.2
powerset	100.0 ±0.1	100.0 ±0.1	100.0 ±0.1
quicksort	99.5 ±0.4	99.6 ±0.4	100.0 ±0.3
rpn_eval	99.9 ±0.1	99.4 ±0.4	99.6 ±0.5
shortest_path_lengths	99.6 ±0.3	97.4 ±1.8	88.5 ±20.7
sqrt	99.9 ±0.1	99.6 ±0.3	99.5 ±0.2

TABLE IV
AVERAGE PERCENTAGE OF VARIANTS WITH BETTER OR EQUAL FITNESS

Program	GenProg		ARJAe		2Phase	
	$<$	$=$	$<$	$=$	$<$	$=$
depth_first_search	25.6	21.9	0.0	35.7	0.0	34.8
detect_cycle	43.6	6.3	0.4	24.9	0.6	24.7
find_in_sorted	49.6	5.3	13.3	19.9	13.0	20.0
get_factors	25.1	24.2	24.9	23.6	32.2	20.0
hanoi	0.0	100.0	1.7	36.6	1.9	36.1
is_valid_parenthesization	59.5	2.0	0.0	33.6	0.0	31.2
knapsack	1.7	11.3	2.4	11.1	3.2	11.8
levenshtein	38.2	20.4	49.0	5.6	50.0	4.7
lis	10.9	6.0	8.7	6.8	9.3	6.5
mergesort	0.0	48.4	0.0	47.9	0.0	48.8
next_permutation	19.4	25.2	21.8	25.2	26.3	22.1
powerset	1.2	27.6	3.2	25.0	0.8	29.7
quicksort	11.0	16.5	23.6	5.8	23.3	5.9
rpn_eval	9.6	19.3	11.0	16.5	4.0	21.4
shortest_path_lengths	57.8	1.4	43.9	3.3	42.1	4.5
sqrt	67.6	3.2	71.0	2.3	67.8	3.2

Table IV shows the average percentage of program variants with a fitness value either strictly better ($<$) or equal ($=$) to the fitness value of the unmodified buggy program. Higher numbers of fitter variants are preferred, as they imply that the search is better guided towards more test-suite adequate patches. The results show that all three fitness functions produce roughly the same percentages of better and equal fitness values (within 5%), except for five programs (*depth_first_search*, *detect_cycle*, *find_in_sorted*, *is_valid_parenthesization*, and *shortest_path_lengths*), for which the standard fitness function produces at least 10% more patches of better or equal fitness as the original, and one (*levenshtein*) for which the standard function produces 10% less.

Answer to RQ1: Overall, neither of the two analysis conducted shows any significant difference between the three fitness functions with regard to the general distribution of fitness values.

B. RQ2: Effectiveness and Efficiency

Table V shows the results yielded by each fitness function over the 20 repeated trials. More specifically, it indicates the number of successful trials (i.e., that resulted in a test-suite adequate patch), the total number of test-suite adequate patches obtained, the number of such *unique* patches obtained, and finally the associated ratio.

We first compare the number of successful trials for each fitness function. For 11 of the 16 faulty programs at least one test-adequate patch was found using all three fitness functions. For *lis*, 2Phase was unsuccessful while the two other produced several test-suite adequate patches. For both *is_valid_parenthesization* and *mergesort*, GenProg's fitness function is the only one to find a test-suite adequate patch. Finally, no test suite-adequate patch was found for the five remaining programs. For programs for which all three fitness functions succeeded in producing test-suite adequate patches, there is no significant difference in the number of trials in which a patch was found.

Next, we compare the uniqueness of test-suite adequate patches generated by three fitness functions. For all three fitness functions uniqueness ratio ranges from 67% to 100% (for *is_valid_parenthesization* the 91 patches, all generated using the standard fitness function, are all unique). When test-suite adequate patches are found there is no significant difference in uniqueness between fitness functions.

Finally, Table VI reports on two metrics used to represent the minimum effort to find a test-suite adequate patch: the smallest number of evaluations before finding a test-suite adequate patch in any trial, and the size of the shortest test-suite adequate patch found across all trials. Excluding programs for which no test-suite adequate patch was found, most GI runs produced a test-suite adequate patch within 200 evaluations (i.e., 5 generations). Furthermore, most test-suite adequate patches consist of at most three edits, with many requiring a single source code modification.

Answer to RQ2: Overall there is no significant difference in effectiveness or efficiency between the three fitness functions. While the standard one led to test-suite adequate patches for two more faulty programs, it did so in a single trial.

C. RQ3: Patch Correctness

The third and final research question relates to the ability of a test-suite adequate patch to generalise to additional test cases. Because manually checking all test-suite adequate patches is impractical (see Table V), to ease the manual verification effort we only focus on a single test-suite adequate patch for each program: the patch with lowest number of edits that also passes the held-out EvoSuite test suite.

Results are shown in Table VII. Data related to other state-of-the-art APR tools is based on the QuixBugs study [10]. Additionally, Table VIII reports the percentage of unique test-suite adequate patches that are not killed by the new EvoSuite-generated test cases. On seven of the eleven programs with test-suite adequate patches, more than 99% of all unique patches also passes the EvoSuite test cases; on the remaining

```
    Node hare = node;
    Node tortoise = node;
    while (true) {
        // if (null == hare || hare.getSuccessor() == null)
        if (hare.getSuccessor() == null)
            return false;
-       tortoise = tortoise.getSuccessor();
-       hare = hare.getSuccessor().getSuccessor();
-       if (hare == tortoise)
+       hare = hare.getSuccessor();
+       if (hare.getSuccessor() == tortoise)
            return true;
    }
}
```

Listing 2. DETECT_CYCLE

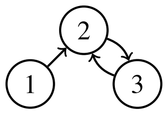

Fig. 1. Counterexample for Listing 2

five programs, the percentage of killed patches greatly varies, especially for *sqrt* on which most patches were killed.

With regards to results reported in the recent QuixBugs study [10], Gin fixes more QuixBugs programs than any of the reported APR tools. The reason might lie in the use of the binary operator replacement edit type, crucial to several of the buggy programs. However, Gin cannot fix buggy programs that are missing an ingredient such as a line or a parameter, a drawback of most search-based APR tools [18].

Answer to RQ3: For 7 out of 11 programs at least 99% of unique patches generalised to the held-out EvoSuite tests, regardless of the fitness function used. Overall there is little difference between the three fitness functions.

D. Details of Manual Patch Analysis

In the following section we analyse a selection of test-suite adequate patches that we used to answer RQ3. Edits to the buggy code are shown following the *diff* format while the ground truth fix is shown as a comment.

1) detect_cycle: [Listing 2]. This patch passes both the original and the EvoSuite test suites, yet is incorrect. Figure 1 shows a counterexample: with 1 as input to the program the cycle is not detected and instead the program runs into an infinite loop.

2) get_factors: [Listing 3] All test-suite adequate patches for *get_factors* were first killed by the EvoSuite test suite. After investigation, this was due to a difference to the oracle for an invalid input value (`get_factors(0)`). Without this faulty test case every test-suite adequate patches pass the EvoSuite test suite.

Listing 3 shows a patch in which the two expressions `n` and `Math.sqrt(n)` have been swapped. While it removes the optimisation that only factors up to `sqrt(n)` are checked, it leads to the check `n%n==0` making so that the faulty return statement is never used. Because this work's focus is on functional fixes rather than optimisation, we report this patch as a functionally correct fix.

TABLE V
RESULTS OVER 20 TRIALS FOR ALL THREE FITNESS FUNCTIONS: NUMBER OF TRIAL YIELDING AT LEAST ONE TEST-ADEQUATE PATCH; TOTAL NUMBER OF TEST-ADEQUATE PATCHES; TOTAL NUMBER OF UNIQUE TEST-ADEQUATE PATCHES AND ASSOCIATED RATIO

Program	**GenProg**				**ARJAe**				**2Phase**			
	success	patches	unique	ratio	success	patches	unique	ratio	success	patches	unique	ratio
depth_first_search	0	0	0	-	0	0	0	-	0	0	0	-
detect_cycle	3	153	123	80.39	1	35	23	65.71	1	48	36	75.00
find_in_sorted	0	0	0	-	0	0	0	-	0	0	0	-
get_factors	8	239	236	98.74	10	392	388	98.98	9	341	335	98.24
hanoi	0	0	0	-	0	0	0	-	0	0	0	-
is_valid_parenthesization	1	91	91	100.00	0	0	0	-	0	0	0	-
knapsack	11	89	79	88.76	8	72	65	90.28	11	99	87	87.88
levenshtein	6	729	630	86.42	5	488	404	82.79	9	1003	827	82.45
lis	5	118	112	92.94	3	63	60	95.24	0	0	0	-
mergesort	1	1	1	100.00	0	0	0	-	0	0	0	-
next_permutation	16	452	439	97.12	15	366	351	95.90	17	748	739	98.80
powerset	0	0	0	-	0	0	0	-	0	0	0	-
quicksort	20	1822	1300	71.35	20	1894	1292	68.22	20	1870	1249	66.79
rpn_eval	12	596	427	71.64	15	737	530	71.91	8	285	214	75.09
shortest_path_lengths	0	0	0	-	0	0	0	-	0	0	0	-
sqrt	5	217	194	89.40	3	111	99	89.19	2	22	17	77.27

TABLE VI
LOWEST NUMBER OF STEPS TO FIND THE FIRST TEST-ADEQUATE PATCH; AND SIZE OF THE SHORTEST TEST-ADEQUATE PATCH

Program	**GenProg**		**ARJAe**		**2Phase**	
	steps	size	steps	size	steps	size
depth_first_search	-	-	-	-	-	-
detect_cycle	41	2	171	2	81	2
find_in_sorted	-	-	-	-	-	-
get_factors	4	1	9	1	9	1
hanoi	-	-	-	-	-	-
is_valid_parenthesization	42	2	-	-	-	-
knapsack	2	1	16	1	1	1
levenshtein	2	1	3	1	7	1
lis	14	1	121	1	-	-
mergesort	379	4	-	-	-	-
next_permutation	9	1	2	1	1	1
powerset	-	-	-	-	-	-
quicksort	1	1	1	1	1	1
rpn_eval	5	1	1	1	39	1
shortest_path_lengths	-	-	-	-	-	-
sqrt	34	1	212	8	300	9

```
  public static ArrayList<Integer> get_factors(int n) {
-     if (n == 1) {
+     if (Math.sqrt(n) == 1) {
          return new ArrayList<Integer>();
      }
-     int max = (int) (Math.sqrt(n) + 1.0);
+     int max = (int) (n + 1.0);
      for (int i = 2; i < max; i++) {
          if (n % i == 0) {
              ArrayList<Integer> prepend = new ArrayList<Integer
                  >(0);
              prepend.add(i);
              prepend.addAll(get_factors(n / i));
              return prepend;
          }
      }
      // return new ArrayList<Integer>(Arrays.asList(n));
      return new ArrayList<Integer>();
  }
```

Listing 3. GET_FACTORS

```
      if (arr.size() == 0) { // if (arr.size() <= 1) {
          return arr;
      } else {
-         int middle = arr.size() / 2;
+         int middle = 2 / 2;
          ArrayList<Integer> left = new ArrayList<Integer>(100);
          left.addAll(arr.subList(0, middle));
-         left = mergesort(left);
          ArrayList<Integer> right = new ArrayList<Integer>(100);
          right.addAll(arr.subList(middle, arr.size()));
          right = mergesort(right);
        return merge(left, right);
      }
```

Listing 4. MERGESORT

3) mergesort: [Listing 4] This patch transforms the buggy merge sort procedure to a weirdly implemented, yet fully functional, insertion sort. Unfortunately this leads to a time complexity of $O(N^2)$ as compared to the expected $O(N \log(N))$. As for *get_factors*, despite the difference in complexity to the oracle, this patch is reported as a correct fix.

4) is_valid_parenthesization: [Listing 5] This patch passes both original and EvoSuite test suites. However, when manually checking the patch we find it to be incorrect. The patched program correctly passes the initially failing "((" test case, but fails the manually constructed "())" test case. Most EvoSuite test cases are unhelpful as they mostly include unrelated non-parentheses characters, such as digits and letters.

5) levenshtein: [Listing 6] The buggy program exposes an off-by-one error in its return statement. Rather than eliminating the superfluous "1+" operation, the Gin-produced patch replaced the addition "+" with "$*$" for an equivalent expression.

6) rpn_eval: [Listing 7] The reference patch swaps the names of the two variables that hold values popped out from the stack. The Gin-produced patch swaps the two statements containing the calls to `stack.pop()` instead, resulting in a semantically equivalent function.

7) lis: [Listing 8] This non-obvious patch passes both the original and the EvoSuite-generated test suites; manual investigation revealed it to be functionally correct. The bug lies in the update of the `longest` variable, which value should

TABLE VII

RESULTS AGAINST ORIGINAL AND EVOSUITE-AUGMENTED TEST SUITES FOR MANUALLY INVESTIGATED PATCHES

Program	**Failure**	**Original**	**EvoSuite**	**Manual**	**Known fix**
depth_first_search	missing line	-	-	-	-
detect_cycle	missing condition	✓	✓	×	NPEFix
find_in_sorted	off-by-one error	-	-	-	-
get_factors	wrong array slice	✓	×	✓	-
hanoi	incorrect variable	-	-	-	-
is_valid_parenthesization	incorrect variable	✓	✓	×	-
knapsack	incorrect operator	✓	✓	✓	jMutRepair
levenshtein	off-by-one error	✓	✓	✓	Cardumen
lis	missing expression	✓	✓	✓	Arja, jGenProg, Nopol, Cardumen, RSRepair, Tibra
mergesort	incorrect operator	✓	✓	✓	Cardumen
next_permutation	incorrect operator	✓	✓	✓	-
powerset	incorrect variable	-	-	-	-
quicksort	incorrect operator	✓	✓	✓	Arja, Dynamoth, jKali, jMutRepair, Nopol, RSRepair
rpn_eval	variable swap	✓	✓	✓	Cardumen
shortest_path_lengths	variable swap	-	-	-	-
sqrt	incorrect arithmetic	✓	✓	×	-

"✓": patch validated; "×": patch killed; "-": no patch found.

TABLE VIII

PERCENTAGE OF UNIQUE TEST-SUITE ADEQUATE PATCHES THAT ALSO PASS THE EVOSUITE-GENERATED TEST CASES

Program	**GenProg**	**ARJAe**	**2Phase**
depth_first_search	-	-	-
detect_cycle	100.00	100.00	100.00
find_in_sorted	-	-	-
get_factors	100.00	100.00	100.00
hanoi	-	-	-
is_valid_parenthesization	67.03	-	-
knapsack	99.79	100.00	100.00
levenshtein	100.00	99.46	100.00
lis	66.24	100.00	-
mergesort	100.00	-	-
next_permutation	100.00	99.14	100.00
powerset	-	-	-
quicksort	100.00	99.44	98.15
rpn_eval	79.72	83.28	77.02
shortest_path_lengths	-	-	-
sqrt	49.40	10.62	0.00

```
public static Boolean is_valid_parenthesization(String parens) {
    int depth = 0;
    for (int i = 0; i < parens.length(); i++) {
        Character paren = parens.charAt(i);
        if (paren.equals('(')) {
            depth++;
        } else {
            depth--;
-           if (depth < 0)
-               return false;
+           if (depth > 0)
+               return true;
        }
    }
    // return depth == 0;
-   return true;
+   return false;
}
```

Listing 5. IS_VALID_PARENTHESIZATION

only increase. In the ground truth version this is done using the `Math.max()` function; in this patch, that statement is replaced by the `if` statement in which it is enclosed.

First, the duplicated line `ends.put(length+1, i);` can be ignored as that value is already set in the outer `if` statement. Thanks to that the right-hand part of the inner `if` condition can be simplified: it is equivalent to `val < arr[i]`, therefore to `val < val`, which is always false. Overall, the inserted condition is semantically equivalent to `if (length==longest){ longest = length+1;}` and in turn equivalent to the provided oracle.

```
  if (source.isEmpty() || target.isEmpty()) {
    return source.isEmpty() ? target.length() : source.length();
  } else if (source.charAt(0) == target.charAt(0)) {
    // return lvshtn(source.substring(1), target.substring(1));
-   return 1 + lvshtn(source.substring(1), target.substring(1));
+   return 1 * lvshtn(source.substring(1), target.substring(1));
```

Listing 6. LEVENSHTEIN

```
        token = (String) token;
        Double c = 0.0;
-       Double b = (Double) stack.pop();
        Double a = (Double) stack.pop();
+       Double b = (Double) stack.pop();
        BinaryOperator<Double> bin_op = op.get(token);
        c = bin_op.apply(a, b); // c = bin_op.apply(b, a);
        stack.push(c);
```

Listing 7. RPN_EVAL

V. THREATS TO VALIDITY

Internal validity. ARJAe and GenProg fitness functions were reimplemented within the Gin framework to ensure a cohesive comparison environment. Some implementation details, in particular the undocumented multiple normalisation functions appearing in the ARJAe source code that were not described in ARJAe's paper might not have been perfectly replicated. To mitigate this threat and encourage high code quality our implementation is released as an open-source repository available for code review.

Another internal threat is the parameters used in the experiment that may lead to GP runs not representative of the performance of each fitness function. Finally, experiments are repeated 20 times in order to minimise bias due to the heuristic nature of the GP algorithm.

```
 for (int val : arr) {
   // ... code elided for concision ...
   int length = !prefix_lengths.isEmpty() ? Collections.max(
        prefix_lengths) : 0;
   if (length == longest || val < arr[ends.get(length + 1)]) {
     ends.put(length + 1, i);
     // longest = Math.max(longest, length + 1);
-    longest = length + 1;
+    if (length == longest || val < arr[ends.get(length + 1)]){
+        ends.put(length + 1, i);
+        longest = length + 1;
+    }
   }
   i++;
 }
 return longest;
```

Listing 8. LIS

External validity. The results of our research and ARJAe's paper are surprisingly different. Specifically, ARJAe's fitness function was found to outperform the GenProg's one, whereas we observe the latter being slightly more effective than the other two investigated fitness functions. Since experiments are conducted on a single benchmark, there is a threat that results do not generalise for the other common benchmarks such as Defects4J. However, it should be noted that QuixBugs is regarded as a hard dataset for APR tools to find test-suite adequate patches for [18].

Construct validity. Correctness of the test-suite adequate patches is manually investigated, and uses as an oracle semantic equivalence to a provided reference program.

VI. Conclusions and Future Work

In this paper, we present 2Phase, a new fitness function for search-based automated program repair (APR). 2Phase considers the difference between the expected and actual value of a test case failure to rank program variants. In this work, 2Phase was implemented, alongside two other state-of-the-art fitness functions, within the Gin genetic improvement framework and empirically evaluated over 16 buggy programs, for which test-suite adequate patches were found in previous work. Furthermore, the EvoSuite tool was used to supplement the original set of test suites to study patch generalisation. Overall, our experiments have produced test-suite adequate patches for 11 of the 16 selected programs. Our manual investigation revealed that 8 patches were true fixes. This is more than any of the APR tools in recent empirical evaluation of APR tools on the QuixBugs benchmark.

Regarding effectiveness and efficiency of the implemented fitness functions, we were unable to show any major significant difference between the standard GenProg fitness function, the more fine-grained ARJAe and our proposed 2Phase. We plan to extend our empirical evaluation to the other programs of the QuixBugs benchmark, as well as to other APR benchmarks, to see if indeed more fine-grained fitness functions could lead to improvement of efficacy and efficiency of APR tooling, and what are the characteristics of programs for which more fine-grained fitness variants work well.

Artefacts: *GitHub links to the code, raw data, including generated EvoSuite and manual test cases, and analysis scripts will be added in case of acceptance.*

References

[1] R. Azevedo, "What is the cost of a bug?" Apr. 2018. [Online]. Available: https://azevedorafaela.com/2018/04/27/what-is-the-cost-of-a-bug/

[2] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software," *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 2013.

[3] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018.

[4] ——, "The Living Review on Automated Program Repair," HAL archives-ouvertes.fr, Tech. Rep. hal-01956501, 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01956501

[5] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

[6] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[7] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 3–13.

[8] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A Generic Method for Automatic Software Repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, Jan. 2012.

[9] Y. Yuan and W. Banzhaf, "Toward Better Evolutionary Program Repair: An Integrated Approach," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 1, Jan. 2020.

[10] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the quixbugs benchmark," *J. Syst. Softw.*, vol. 171, p. 110825, 2021.

[11] A. E. I. Brownlee, J. Petke, B. Alexander, E. T. Barr, M. Wagner, and D. R. White, "Gin: genetic improvement research made easy," in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, Jul. 2019, pp. 985–993.

[12] D. R. White, "GI in No Time," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery, 2017, pp. 1549–1550.

[13] J. Petke and A. Blot, "Refining fitness functions in test-based program repair," in *The First International Workshop on Automated Program Repair (APR@ICSE 2020)*, 2020.

[14] M. Asad, K. K. Ganguly, and K. Sakib, "Impact of similarity on repairing small programs: A case study on quixbugs benchmark," in *ICSE '20: 42nd International Conference on Software Engineering, Workshops*. ACM, 2020, pp. 21–22.

[15] Y. Yuan and W. Banzhaf, "ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[16] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing," *Software Testing, Verification and Reliability*, vol. 23, no. 2, pp. 119–147, 2013.

[17] E. F. de Souza, C. L. Goues, and C. G. Camilo-Junior, "A Novel Fitness Function for Automated Program Repair Based on Source Code Checkpoints," in *Proceedings of the Genetic and Evolutionary Computation Conference*. Association for Computing Machinery, 2018, pp. 1443–1450.

[18] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of Java program repair tools: A large-scale experiment on 2, 141 bugs and 23, 551 repair attempts," *CoRR*, vol. abs/1905.11973, 2019.

[19] D. X. Bach, "Overfitting in automated program repair: Challenges and solutions," Ph.D. dissertation, Singapore Management University, 2018.

[20] R. Doornbosch and R. Steenblik, "Biofuels: Is the cure worse than the disease," *Revista Virtual REDESMA*, vol. 2, pp. 63–100, 2008.

[21] A. E. I. Brownlee, J. Petke, and A. F. Rasburn, "Injecting shortcuts for faster running Java code," in *CEC*. IEEE, 2020, pp. 1–8.

[22] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge," in *SPLASH (Companion Volume)*. ACM, 2017, pp. 55–56.

[23] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 12, pp. 1236–1256, December 2015.