Refining Fitness Functions in Test-Based Program Repair

Justyna Petke
University College London
London, United Kingdom
j.petke@ucl.ac.uk

Aymeric Blot
University College London
London, United Kingdom
a.blot@cs.ucl.ac.uk

ABSTRACT

Genetic improvement has proved to be a successful technique in optimising various software properties, such as bug fixing, runtime improvement etc. It uses automated search to find improved program variants. Usually the evaluation of each mutated program involves running a test suite, and then calculating the fitness based on Boolean test case results. This, however, creates plateaus in the fitness landscape that are hard for search to efficiently traverse. Therefore, we propose to consider a more fine-grained fitness function that takes the output of test case assertions into account.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**.

KEYWORDS

Automated Program Repair, Genetic Improvement, Search-Based Software Engineering

ACM Reference Format:
Justyna Petke and Aymeric Blot. 2020. Refining Fitness Functions in Test-Based Program Repair. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3387940.3392180

1 INTRODUCTION

With the release of GenProg [5], the first scalable automated program repair (APR) tool, the field of APR has blossomed. There are currently over 30 tools in the field[1]. The technique since has been adapted to improve other software properties, such as runtime or energy or memory consumption, and the field of genetic improvement (GI) has emerged [7]. It uses automated, usually metaheuristic, search to improve software. Regardless of the properties of choice for improvement, in GI-based approaches, a set of test cases is used to evaluate a candidate software variant. The result of running these tests is then used to estimate how 'good' a software variant is. In the automated program repair field this amounts to the number of test cases passed.

[1]http://program-repair.org/tools.html

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392180

Using Boolean test cases in fitness evaluation in search-based APR approaches may lead to fitness landscapes with very large plateaus. Indeed, many software modifications have by themselves very little impact on the overall software [3, 8] thus leaving the fitness unchanged. These plateaus make guidance over the search space very difficult, especially when a repair involves multiple simultaneous modifications.

Therefore, we propose to change how test cases are evaluated, by taking the output of test cases into account at the fitness evaluation stage. More specifically, we propose to evaluate fitness by using the differences between the expected and the actual results.

2 MOTIVATING EXAMPLE

We present a motivating example in which a buggy piece of software is to be fixed following a provided test suite, though the technique can be applied to any genetic improvement framework. The QuixBugs benchmark suite[2] [6, 9] provides many simple buggy programs together with associated test cases. We focus on the buggy version of the greatest common divisor (GCD) Java program, presented in Listing 1. Note that the arguments' order in the recursive call to `gcd()` is wrong and should be reversed. Table 1 presents the six test cases provided by the QuixBugs benchmark suite, together with the output results on the buggy code and the expected output. While the buggy code correctly passes the first test case, it induces an infinite loop in each of the following five. Under the usual Boolean fitness evaluation assumption, this would, for example, mean a fitness value of five, to be minimised.

Listing 1: Buggy GCD.java program.

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

Suppose now that our APR framework operates at the level of abstract syntax tree (AST) nodes and that among mutation operators we have a replacement operator. In the generate-and-validate APR (and GI in general) this is a common setup [7]. Suppose that during search the second argument of the recursive call, `b`, is replaced with `a % b`. The output of this mutant on the test suite is shown in Table 2. In particular, while the first output is unchanged, the five infinite loops have been replaced by wrong outputs. Again, using Boolean fitness evaluation the fitness of this mutant would be the same as the original program, albeit it being much closer to the intended solution.

[2]https://jkoppel.github.io/QuixBugs/

Table 1: Test cases for GCD, with outputs from Listing 1.

Inputs (a,b)	Output on Buggy Code	Expected Output
17, 0	17	17
13, 13	infinite loop	13
37, 600	infinite loop	1
20, 100	infinite loop	20
624129, 2061517	infinite loop	18913
3, 12	infinite loop	3

Table 2: Test cases for GCD with outputs for mutant `gcd(a % b, b) -> gcd(a % b, a % b)`.

Inputs (a,b)	Output on Buggy Code	Expected Output
17, 0	17	17
13, 13	0	13
37, 600	0	1
20, 100	0	20
624129, 2061517	0	18913
3, 12	0	3

Therefore, we propose to have a two-step fitness function. At first mutants are compared using the usual Boolean fitness evaluation: mutants with more passing test cases should always be preferred. However, when two mutants have the same Boolean fitness then a more fine-grained fitness is used, based on the distance between individual expected and actual outputs. Essentially, this allows consideration of more rugged fitness landscapes thus enabling more guidance in the search space of APR and GI approaches, which otherwise have to deal with very flat landscapes.

The tests that lead to a crash, infinite loops etc. (essentially any behaviour that does not produce an output) would contribute the maximum penalty to the fitness function. On the other hand, the tests that yield an inaccurate output would be assigned a penalty that is based on some distance metric. For example, in our case of GCD this could simply be a difference between integers. In this case the fitness value based on outputs in Table 1 could be $5 * (MAX\_INT)$, while the fitness value based on outputs in Table 2 could be $13 + 1 + 20 + 18913 + 3 = 18950$. Therefore, it is the second variant that would be preferred and selected for mutation, thus increasing the chances of the second mutation (that replaces the first `a % b` with `b`) to take place that leads to the required solution. Otherwise, search would have had to find the two changes simultaneously, essentially by chance, as it has no guidance that the first mutation `b -> a % b` is a step towards finding the right solution.

3 TECHNICAL ASPECTS

Modification of the fitness function in existing generate-and-validate APR and GI frameworks should be straightforward. For example, the Gin [1] genetic improvement framework has a `UnitTestResult` class that allows for automatic extraction of the expected and actual test result. This is achieved with the following two methods: `getAssertionExpectedValue()` & `getAssertionActualValue()`. For numeric values, one could calculate the sum (as in the case of the motivating example) or the maximum of the differences of the expected and actual test case outputs. For strings one could use Hamming distance, or any other distance metric of choice.

4 RELATED WORK

More fine-grained fitness functions have been proposed by Souza et al. [2], where they propose to exploit intermediate program states (called checkpoints). However, the technique involves instrumenting source code and there's an additional overhead of collecting and using checkpoints.

Jang et al. [4] propose a fitness function that, additionally to test case failure, records the number of modification points that have been touched by those tests. The modification points have been previously calculated, using coverage and fault localisation criteria. This is a more lightweight approach than that of Souze et al., but it does not take the test case output itself into consideration.

5 CONCLUSIONS AND FUTURE WORK

In this paper we propose to use a new fine-grained fitness function for improvement of software properties such as bug fixing. We presented a motivating example from the QuixBugs program repair benchmark suite and discussed technical aspects of our proposed approach. We pose that in the generate-and-validate automated software improvement approaches by employing a fitness function that takes into account the differences between actual and expected output values of test cases we will be able to guide the search towards effective solutions. Therefore, we intend to implement the proposed approach and conduct an empirical study to verify this claim in our future work.

ACKNOWLEDGMENTS

This work is supported by UK EPSRC Fellowship EP/P023991/1.

REFERENCES

[1] Alexander E. I. Brownlee, Justyna Petke, Brad Alexander, Earl T. Barr, Markus Wagner, and David R. White. 2019. Gin: genetic improvement research made easy. In *Genetic and Evolutionary Computation Conference*. ACM, 985–993.

[2] 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 *Genetic and Evolutionary Computation Conference*. ACM, 1443–1450.

[3] Nicolas Harrand, Simon Allier, Marcelino Rodriguez-Cancio, Martin Monperrus, and Benoit Baudry. 2019. A journey among Java neutral program variants. *Genet. Program. Evolvable. Mach.* 20, 4 (2019), 531–580.

[4] Yoowon Jang, Quang-Ngoc Phung, and Eunseok Lee. 2019. Improving the Efficiency of Search-Based Auto Program Repair by Adequate Modification Point. In *International Conference on Ubiquitous Information Management and Communication (Advances in Intelligent Systems and Computing)*, Vol. 935. Springer, 694–710.

[5] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Softw. Eng.* 38, 1 (2012), 54–72.

[6] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, 55–56.

[7] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Trans. Evol. Comput.* 22, 3 (2018), 415–432.

[8] Eric M. Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2014. Software mutational robustness. *Genet. Program. Evolvable. Mach.* 15, 3 (2014), 281–312.

[9] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark. In *Intelligent Bug Fixing*. IEEE, 1–10.