Combining Online Learning with Mutation-Based Stochastic Search to Repair Buggy Programs

Joseph Renzullo
Arizona State University
Tempe, Arizona, USA
renzullo@asu.edu

Westley Weimer
University of Michigan
Ann Arbor, Michigan, USA
weimerw@umich.edu

Stephanie Forrest*
Arizona State University
Tempe, Arizona, USA
steph@asu.edu

ABSTRACT

This article summarizes recent work in the field of Automated Program Repair that was published in Transactions on Evolutionary Learning and Optimization as *Evolving Software: Combining Online Learning with Mutation-Based Stochastic Search.* Automated Program Repair is a subfield of software engineering that has the goal of repairing defects in software with minimal human involvement. A popular approach combines random mutation with some form of search, but these methods are highly conservative, because most mutations are deleterious and can damage the program. We describe a method inspired by neutral mutations in biological systems that splits the problem of finding useful mutations into two stages. First, before a bug is identified, we generate mutations and screen them for safety, discarding any that break required functionality of the program. Then, when a software bug is reported, we rapidly and dynamically test large subsets of the earlier-discovered pool of mutations to find those that repair the defect. We implement this method in an algorithm called MWRepair, which uses online learning to guide the aggressiveness of the search process. MWRepair extends the reach of existing mutation-based techniques to repair harder and more complex defects in programs.

CCS CONCEPTS

• Theory of computation → Evolutionary algorithms; **Online learning algorithms**; **• Software and its engineering → Search-based software engineering**; **Software fault tolerance**.

KEYWORDS

Neutral Mutations, Automated Program Repair, Software Mutational Robustness, Multiplicative Weights Update

ACM Reference Format:
Joseph Renzullo, Westley Weimer, and Stephanie Forrest. 2024. Combining Online Learning with Mutation-Based Stochastic Search to Repair Buggy Programs. In *Genetic and Evolutionary Computation Conference (GECCO '24 Companion), July 14–18, 2024, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3638530.3664082

*Also at the Santa Fe Institute

1 INTRODUCTION

Some of the earliest program repair tools [4] and some of the most recent [5] use population-based evolutionary search. The surprising success of population-based evolutionary algorithms in repairing bugs in software spawned a subfield of software engineering called Automated Program Repair [2]. Earlier work in APR has historically used conservative search processes, because it is much easier to break a program than it is to repair one [1]. Despite their successes, current APR algorithms are still quite limited, which motivates our work. Our work addresses these challenges using an algorithm, MWRepair, designed with three key features. First, we combine many *neutral mutations* to enable evaluation of more than one mutation at a time, which reduces the cost of searching for repairs. Second, we incorporate *online learning* to guide the search to the optimal region of the search space, as characterized by our model. Third, we *precompute* neutral mutations to reduce the cost of the online search process, refactoring some of the expense normally paid when the repair is needed to an offline, parallel process, which can be reused for multiple bugs. This paper extends prior work that evaluated parallel computation of the inputs (neutral mutations) and studied several implementations of online learning to identify the most efficient for this problem.

The main contributions of the paper are:

- An empirical analysis of the search space for mutation-based program repair. Repairs are on average 42.4 times more frequent in the optimal region of the search space for C programs (6.9 times more frequent for Java programs) as they are one mutation away from the original.
- An evaluation of MWRepair with GenProg's mutation operators on the Defects4J benchmark. MWRepair repairs significantly more defects than two reference evolutionary tools for Java, jGenProg and GenProg-A, and it discovers some repairs that have not been repaired by any earlier tool, including some multi-edit repairs.
- A quantification and visualization of all published Java repair algorithms applied to Defects4J. We discuss the overlap and uniqueness of the repairs generated by each algorithm.

2 METHODS

In this paper we are particularly interested in the search space for evolutionary APR algorithms, focusing on the effect of combining neutral mutations, which have previously been validated against the test suite. We define the set of *neutral mutations* as $\{m(p) : f(m(p)) = f(p)\}$, where p is a program, $f(.)$ assigns a fitness value to a program, and $m(.)$ applies a mutation to a program. Mutations are generated relative to the original version of the program and evaluated for neutrality using all positive test cases. These positive

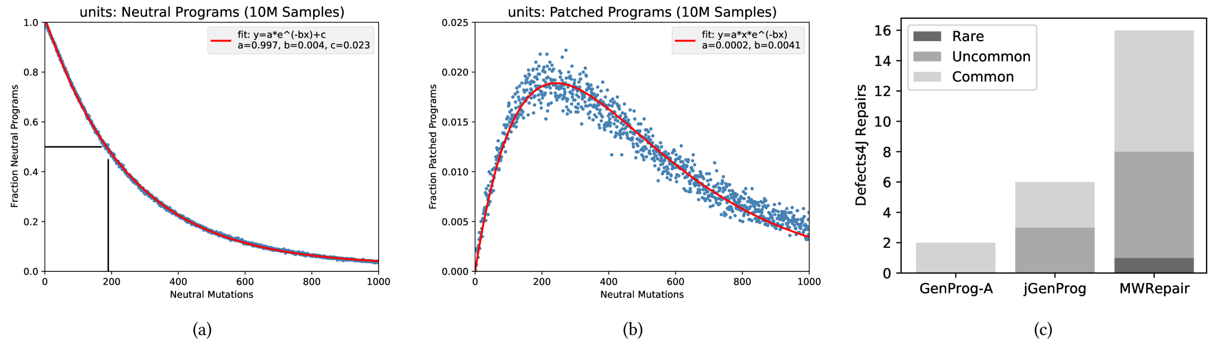

Figure 1: 10,000 independent random samples each of n = 1 to 1,000 pre-computed neutral mutations. Y-axis: fraction of resulting programs that are (a) neutral or (b) patch the defect. Empirical data shown in blue, best-fit model in red. Shades in (c) indicate repair novelty: rare (found by 1 algorithm), uncommon (2–4 algorithms), or common (5+ algorithms). Data from all published results for Defects4J.

tests define the required behavior of the program; only mutations passing all positive tests are deemed neutral; they have the same fitness as the original version of the program.

We assume that later, one or more (negative) test cases are added that surface a latent defect in the program. These negative tests initially fail, and a successful program variant must pass them (while retaining positive test behavior) to be deemed a patch. Thus, we define patches as $\{m(p) : f'(m(p)) = f'(p*)\}$, where $f'(.)$ is the fitness function augmented with new, negative tests and $p*$ is the reference human repair in the benchmark.

3 SUMMARY OF RESULTS

As a case study, we consider the program `units`, a Unix command-line utility for converting measurements into different units. `units` is a C program with ≈ 1,000 lines of code (1 kLoC). We generated all possible 41,344 atomic mutations to the buggy version of `units` using GenProg's mutation operators, i.e. all possible applications of the *append*, *delete*, and *swap* operators applied to the parts of the program covered by the original test suite. Figure 1(a) illustrates the success rate (fraction of programs behaving normally) when evaluating n individually neutral mutations at a time. Figure 1(b) plots the probability of finding patches (y-axis) for each choice of n. This established that it is safe and effective to search by combining large numbers of neutral mutations. A fuller discussion of this that includes evidence from other programs appears in the full paper.

We evaluated MWRepair's performance on the popular Defects4J set of benchmark Java (buggy) programs. For each Defects4J scenario, we precomputed a pool of neutral mutations within a maximum budget of 10k fitness function evaluations (usually resulting in ≈3000 neutral mutations). We then ran MWRepair with a limit of 1k test suite evaluations for testing combinations of this precomputed subset and summarize the outcome in Figure 1(c).

Here, we ask how often MWRepair finds repairs for bugs, relative to the two independent implementations of the GenProg algorithm for Java that have been evaluated against Defects4J: jGenProg [3] and GenProg-A [6]. MWRepair identifies more than twice the number of repairs to bugs in the benchmark, and it discovers repairs to harder bugs (measured by the proxy of repair rarity across all known evaluations). Some of the repairs MWRepair finds are not only unique relative to GenProg's earlier results, but also relative to all published algorithm evaluations on the Defects4J benchmark.

4 CONCLUSIONS

This paper investigated how to more effectively and efficiently search the space created by a set of mutation operators inherited from GenProg. MWRepair explores the search space differently from other algorithms: its ambitious combination of mutations is enabled by the use of neutral mutations. To apply this approach efficiently, we precompute a set of neutral mutations for subsequent use in online repair scenarios. We first established that combining mutations can be safe and productive, and then showed that this is effective for repairing harder and more complex bugs in programs.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the partial support of NSF (CCF 2211750, 2211749) and ARPA-H (SP4701-23-C-0074). We also thank ASU Research Computing for enabling these experiments.

REFERENCES

[1] N. Harrand, S. Allier, M. Rodriguez-Cancio, M. Monperrus, and B. Baudry. 2019. A Journey among Java Neutral Program Variants. *Genetic Programming and Evolvable Machines* 20, 4 (June 2019), 531–580. https://doi.org/10.1007/s10710-019-09355-3
[2] C. Le Goues, M. Pradel, and A. Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (Nov. 2019), 56–65. https://doi.org/10.1145/3318162
[3] M. Martinez and M. Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *International Symposium on Software Testing and Analysis*. ACM, Saarbrücken, Germany, 441–444. https://doi.org/10.1145/2931037.2948705
[4] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *International Conference on Software Engineering*. IEEE, Vancouver, BC, Canada, 364–374. https://doi.org/10.1109/ICSE.2009.5070536
[5] C.-P. Wong, P. Santiesteban, C. Kästner, and C. Le Goues. 2021. VarFix: Balancing Edit Expressiveness and Search Effectiveness in Automated Program Repair. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Athens, Greece, 354–366. https://doi.org/10.1145/3468264.3468600
[6] Y. Yuan and W. Banzhaf. 2020. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *Transactions on Software Engineering* 46, 10 (2020), 1040–1067. https://doi.org/10.1109/TSE.2018.2874648