Evolving Software: Combining Online Learning with Mutation-Based Stochastic Search

JOSEPH RENZULLO, Arizona State University, USA
WESTLEY WEIMER, University of Michigan, USA
STEPHANIE FORREST, Arizona State University, USA

Evolutionary algorithms and related mutation-based methods have been used in software engineering, with recent emphasis on the problem of repairing bugs. In this work, programs are typically not synthesized from a random start. Instead, existing solutions—which may be flawed or inefficient—are taken as starting points, with the evolutionary process searching for useful improvements. This approach, however, introduces a challenge for the search algorithm: what is the optimal number of neutral mutations that should be combined? Too much is likely to introduce errors and break the program while too little hampers the search process, inducing the classic tradeoff between exploration and exploitation.

In the context of software improvement, this work considers MWRepair, an algorithm for enhancing mutation-based searches, which uses online learning to optimize the tradeoff between exploration and exploitation. The aggressiveness parameter governs how many individual mutations should be applied simultaneously to an individual between fitness evaluations. MWRepair is evaluated in the context of automated program repair problems, where the goal is repairing software bugs with minimal human involvement. The article analyzes the search space for automated program repair induced by neutral mutations, finding that the greatest probability of finding successful repairs often occurs when many neutral mutations are applied to the original program. Moreover, repair probability follows a characteristic, unimodal distribution. MWRepair uses online learning to leverage this property, finding both rare and multi-edit repairs to defects in the popular Defects4J benchmark set of buggy Java programs.

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

Additional Key Words and Phrases: Neutral mutations, automated program repair, software mutational robustness, multiplicative weights update

ACM Reference format:
Joseph Renzullo, Westley Weimer, and Stephanie Forrest. 2023. Evolving Software: Combining Online Learning with Mutation-Based Stochastic Search. *ACM Trans. Evol. Learn.* 3, 4, Article 13 (December 2023), 32 pages.
https://doi.org/10.1145/3597617

S. Forrest is also at the Santa Fe Institute.

S. Forrest received partial support from NSF (CCF 2211749, CCF 2211750, CICI 2115075), DARPA (FA8750-19C-0003, N6600120C4020), AFRL (FA8750-19-1-0501), and the Santa Fe Institute.

Authors' addresses: J. Renzullo and S. Forrest, Arizona State University, 727 E. Tyler St., Tempe, Arizona, 85281, USA; emails: renzullo@asu.edu, steph@asu.edu; W. Weimer, University of Michigan, 2260 Hayward St., Ann Arbor, Michigan, 48109-2121, USA; email: weimerw@umich.edu.

2688-3007/2023/12-ART13 $15.00
https://doi.org/10.1145/3597617

1 INTRODUCTION

Some of the earliest program repair tools [Weimer et al. 2009] and some of the most recent [Wong et al. 2021; Yuan and Banzhaf 2020b] use population-based evolutionary search. These tools generate initial populations by applying a small number of mutations to each of many copies of the original program. Then the fitness of each program variant in the population is evaluated by executing it on a set of test inputs and comparing the result to a set of expected outputs. Next, selection and crossover use this fitness information to construct the subsequent generation, and the algorithm iterates. The termination condition is either a time limit or the discovery of a program that passes all of the test cases, including those that demonstrated the defect in the program.

The surprising success of population-based evolutionary algorithms in repairing bugs in software spawned a subfield of software engineering called **Automated Program Repair (APR)** [Gazzola et al. 2019; Le Goues et al. 2019; Monperrus 2018]. As it has matured, diverse approaches have been proposed for generating mutated programs, most of which are not evolutionary algorithms. These algorithms include brute-force search over single mutations [Weimer et al. 2013], synthesis methods based on program semantics [Mechtaev et al. 2016; Xiong et al. 2017], the use of templated mutation operators [Liu et al. 2019b], and the prediction of code by neural networks [Chen et al. 2019; Jiang et al. 2021].

This 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 [Harrand et al. 2019; Schulte et al. 2014]. This motivates our investigation of the search space of neutral mutations that, unlike untested mutations, have been filtered by their impact on program behavior. 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. In this article, we explore how neutral mutations can be employed to enhance exploration of the search space for program repair.

Despite their successes, current APR algorithms are still quite limited, which motivates our work. In particular, they repair only a fraction of the presented defects [Durieux et al. 2019; Le et al. 2019; Le Goues et al. 2015; Long and Rinard 2016]. For example, Java APR algorithms individually repair between 1% and 15% of programs in the Defects4J benchmark. Our survey of all published Java APR algorithms finds that only 130 out of 357 defects (39.2%) in the popular Defects4J benchmark [Just et al. 2014] are reported to have been repaired by any algorithm. In addition, APR searches are expensive because they execute the test suite on each step of the search [Monperrus 2018].

Our work addresses these challenges using an algorithm 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 (Section 4). 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 (Section 6.5). These innovations extend the reach of evolutionary software repair algorithms (e.g., GenProg) by enhancing their ability to explore the search space effectively.

This article 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. Here, we present additional detail characterizing the search space, extend the evaluation from 5 to 357 bugs in the Defects4J Java APR benchmark, and situate our results in the broader context of the field.

In summary, the main contributions of this work are as follows:

- *An empirical analysis of the search space for mutation-based program repair*: Repairs are on average 42.4 times as frequent in the optimal region of the search space for C programs (6.9 times as frequent for Java programs), as they are one mutation away from the original, which is where current methods search most intensively.
- *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 the success of all published Java repair algorithms applied to Defects4J*: We discuss the relative overlap and uniqueness of the repairs generated by each algorithm.

Section 2 outlines the repair model we assume and briefly motivates the research questions that structure the article. In Section 3, we review related work with a focus on distinguishing MWRepair from comparable algorithms. Section 4 characterizes the search space; Section 5 details the MWRepair algorithm, which Section 6 evaluates on reference benchmarks. Section 7 discusses the results of our evaluation, and Section 8 details promising directions for future work. Finally, Section 9 concludes the article.

2 REPAIR MODEL AND RESEARCH QUESTIONS

In this article, 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. As mentioned in Section 1, 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 we are trying to repair. Mutations are evaluated for neutrality using all of the positive tests in the test suite. These positive tests define the required behavior of the program; only mutations passing this positive test suite are deemed to be neutral, and they have the same fitness value as the original version of the program on these tests, which may contain latent bugs.

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. This reflects how test suites evolve in practice, with new test cases added to formalize behavior in response to bug reports [Pinto et al. 2012]. Under this definition, some semantically correct patches will not be labeled as repairs if they repair the defect differently from the reference human repair. In the remainder of the article, we refer to neutral mutations as those generated at time t with reference to the original test suite $f(.)$ and evaluate their utility when deployed in a repair process at time t' with reference to the augmented test suite $f'(.)$.

With this framework in mind, our first cluster of research questions (1–3) centers around characterizing the search space in terms of neutral mutations:

- *RQ 1*: How many neutral mutations can be safely combined?
- *RQ 2*: How often do neutral mutations generate a patch?
- *RQ 3*: Can we describe the effects of combining neutral mutations in a general model?

Answers to RQ 1 and RQ 2 will shed light on the structure of the search space, but they are not immediately relevant because the location of the optimum varies with the particular program and

defect. In addition, APR searches are typically terminated after a single patch is discovered, so in most cases we do not expect to observe the full distribution of patch frequency. We return to these topics in Section 4, where the answer to RQ 3 provides the requisite information for designing an adaptive, online algorithm.

We hypothesize that improving search processes, in light of our analysis of the search space, will allow evolutionary algorithms to repair a wider variety of defects than they do today. To investigate this, the next cluster of research questions (4–8) centers around an evaluation of search-based repair algorithms, which was designed to exploit these insights:

- *RQ 4*: Does MWRepair improve the ability of Java-based GenProg tools to discover patches and (unique) repairs?
- *RQ 5*: Do patches found by MWRepair minimize to single mutations or can MWRepair discover multi-edit patches?
- *RQ 6*: How many of the patches MWRepair identifies are correct repairs?
- *RQ 7*: Do neutral mutations remain neutral or are they invalidated over time by changes to the test suite as a software project matures?
- *RQ 8*: How does MWRepair-enhanced GenProg compare to the state of the art?

These topics are addressed in Section 6.

3 BACKGROUND

3.1 Online Learning Algorithms

Online learning algorithms are designed to efficiently incorporate information obtained while a system is running, without requiring expensive model training. **Multiplicative Weights Update (MWU)** is a popular example [Arora et al. 2012] that has many implementations that have been devised for different applications. In earlier work [Renzullo et al. 2021], the three most relevant versions of MWU were studied in the context of APR. The algorithm formulations vary in their use of memory (globally shared or distributed), communication requirements (full synchronization or partial), the number of trials required per timestep, and the time to convergence. For problem domains like APR that are characterized by high resource cost per trial and low communication cost, an implementation that uses globally shared memory and full synchronization (e.g., the standard MWU formulation [Arora et al. 2012]) was found to be optimal. We incorporate MWU in MWRepair (and inherit part of its name) due to its time and space efficiency and its attractive mathematical bound on *regret*, which in our application area corresponds to unnecessary test suite evaluations. The time complexity required for MWU to converge is $O(\frac{ln(n)}{\varepsilon^2})$, where n is the number of options (in our context, this is the number of combined mutations) and ε is the error tolerance. We discuss the finer-grained components in Section 5. MWRepair reduces the dominant cost associated with search-based APR.

3.2 Evolving Software

Evolutionary algorithms have been applied in many domains [Bäck et al. 1997], particularly where the creativity of the evolutionary search process allows it to navigate problem spaces that are infeasible to optimize explicitly. The idea of evolving software has a long history [Petke et al. 2019], originating with ideas about automatic programming that predate modern computers, and continuing through decades of research on genetic programming [Banzhaf and Leier 2006; Koza 1992]. The goal of genetic programming is to evolve programs from scratch that satisfy a specification. This is challenging because the space of possible programs is enormous and fitness functions typically provide only coarse guidance. Two key ideas that enabled genetic programming to successfully evolve software were (1) focusing on improving an existing program rather than starting from

scratch and (2) using existing test suites from programs to define a fitness function [Arcuri 2008; Weimer et al. 2009].

3.2.1 Neutral Mutations. Our proposed approach rests on the idea of neutral mutations—that is, mutations that do not change the measured fitness of the program. This term *neutral mutations* has been used in software engineering [Jia and Harman 2011; Schulte et al. 2014], where they are sometimes referred to as *sosies* [Baudry et al. 2015], and in evolutionary computation (e.g., in genetic programming [Banzhaf and Leier 2006] and genetic improvement [Veerapen et al. 2017]). Neutral mutations have long been observed in biology [Kimura 1968, 1983], where they are posited to be a key mechanism for introducing and maintaining genetic diversity. They have also been measured to be important for search processes in genetic programming [Banzhaf and Leier 2006].

Mutations applied to tested regions of C programs [Schulte et al. 2014] and Java programs [Baudry et al. 2014; Harrand et al. 2019] are neutral ≈30% of the time and mutations are harmful ≈70% of the time. Assuming no interactions, the probability of applying n random mutations together and still passing the positive tests is $\approx (1 - 0.70)^n = 0.3^n$. This quickly approaches zero as n grows, which Langdon et al. [2010] observed empirically. This motivates our focus on neutral mutations because naively combining random mutations does not allow the search to explore very far.

However, mutations that are neutral on their own are not guaranteed to be neutral when combined, because they may interact. In the field of software optimization, this interaction is referred to as *synergy* [Bruce et al. 2019], and in evolutionary biology and evolutionary computation, it is called *epistasis* [Phillips 2008]. Some earlier work in program repair augments fitness functions to rank mutants and reduce neutrality [De Souza et al. [2018]. Instead, MWRepair uses neutral mutations as the building blocks of repairs. Interactions between individually neutral mutations lead to a tradeoff between search aggressiveness and search efficiency. To understand and quantify this effect, Section 4 presents empirical evidence and a mathematical model.

3.2.2 Search Landscapes. All search algorithms confront the problem of balancing *exploration* with *exploitation*. Our approach enhances exploration by composing large numbers of neutral mutations. This relates to *neutral landscapes*: networks of equally fit mutational neighbors. The topology of neutral landscapes affects a population's ability to find high-fitness innovations [Banzhaf and Leier 2006; Wagner 2007]. Neutrality and robustness to mutation are key to biological evolution [Kimura 1968, 1983], perhaps because they allow population-based search processes to preserve diversity while retaining functionality.

Earlier studies of search spaces for mutating programs [Petke et al. 2019] use diverse algorithms. Some have characterized the mutation-behavior relationship of the triangle problem [Langdon et al. 2017], sorting programs [Schulte et al. 2014], and the TCAS (Traffic Collision Avoidance System) avionics software [Veerapen et al. 2017]. Others sample the search space of patches for APR [Long and Rinard 2016; Renzullo et al. 2018] or formalize it [Martinez and Monperrus 2015]. However, these earlier studies generate single mutations and analyze their individual effects. Langdon et al. [2010] extended this to higher-order (multiple) mutations. They report that the chance of a single untested mutation being neutral is 16% (lower than several other estimates), but that the chance of combining four and retaining behavior drops sharply to 0.06%. Our work extends these earlier results in two ways: (1) by analyzing combinations of hundreds of neutral mutations and (2) incorporating our analysis into an algorithm.

Because MWRepair searches by combining many mutations, it is likely to generate patches with many mutations, only some of which are needed to address the defect. There are two shortcomings inherent to large patches such as these. First, each time a new neutral mutation is added, there is

a risk that it changes semantics on some untested aspect of the code, and in some cases those side effects could be harmful. Second, more mutations negatively affects readability, increasing the difficulty for a human to understand how the patch operates. To minimize the mutation set to only those that are necessary, we use a standard minimization algorithm. The procedure, known as delta debugging [Zeller 1999], is used widely by the GenProg family [Weimer et al. 2009] and other APR algorithms, producing a guaranteed minimal set of mutations that maintains desired behavior (e.g., repairs the defect).

3.3 Automated Program Repair

3.3.1 Test Oracle Problem. It is infeasible to exactly specify the desired semantics of large programs [Liu et al. 2021]. The most common approach to this problem is to use test suites as a proxy for correctness. Used widely in industry and in APR, this approach relies on a set of individual tests, each consisting of an input and the expected output that corresponds to it. It is difficult to test all of the desired behavior of a program, and even extensively tested programs (like those in the Siemens Software-artifact Infrastructure Repository) admit neutral mutations [Schulte et al. 2014], either because the test suite is incomplete or because the mutation is semantically equivalent. Because of this, there is often a gap between what the test suite measures about program behavior and what the developer intended. Some recent work has moved toward automating the assessment of patch correctness [Cashin et al. 2019; Wang et al. 2020; Ye et al. 2021].

3.3.2 Terminology. Different papers have defined different terms for programs that have been modified to pass all of the available tests in a test suite, including *final repair* [Le Goues et al. 2012], *plausible patch* [Qi et al. 2015], and *test-suite adequate patch* [Ye et al. 2020], among others. Determining whether a program that passes a given test suite actually repairs a defect is challenging. The most popular method used today involves comparing the code generated by the APR algorithm to human-written code supplied as the reference in the defect benchmark [Just et al. 2014; Qi et al. 2015]. In this approach, only code that is semantically equivalent to the reference code is considered *correct*.

For simplicity, in this article we refer to a set of mutations that passes the entire test suite as a *patch*. Similarly, an algorithm *patches* a defect if it produces a patch for it. A *repair* is a type of patch: one that is semantically equivalent to the reference human-written repair. An algorithm *repairs* a defect if it produces a repair for it.

3.4 MWU and Regret

MWRepair uses MWU to guide its search. MWU, reproduced from Arora et al. [Arora et al. 2012] as Algorithm 1, maintains a weight vector over its options. In our application, each option is a bin that specifies a range of how many mutations to apply.

MWU bounds *regret*: the gap between choices made while learning and the best choice in hindsight. Following Theorem 2.1 in the work of Arora et al. [Arora et al. 2012]:

$$Regret = \sum_{t=1}^{T} m^{(t)} \cdot p^{(t)} \leq \sum_{t=1}^{T} m_i^{(t)} + \eta \sum_{t=1}^{T} |m_i^{(t)}| + \frac{ln(n)}{\eta}.$$

The vector of option costs is $m^{(t)}$ and the probability vector over the options is $p^{(t)}$, both indexed by time t. The learning parameter, which quantifies how much attention is paid to new information at each timestep, is η and the number of options is n. Regret has a tight upper bound determined by the best choice plus two smaller factors that depend on the learning parameter and the number of options.

ALGORITHM 1: The MWU Algorithm

Initialization: *Select learning parameter* $\eta \leq 1/2$.
For each option i, set its weight $w_i^{(1)} \leftarrow 1$.
for $t \leftarrow 1$ **to** T **do**
    1. option $\leftarrow$ select($w^{(t)}$)
    2. $\mathbf{m}^{(t)} \leftarrow$ observe_cost(option)
    3. $w^{(t+1)} \leftarrow w^{(t)} * (1 + \eta * \mathbf{m}^{(t)})$

MWU can trade off runtime for accuracy according to the following relationship [Arora et al. 2012]:

$$T < \frac{4\rho^2 \cdot ln(n)}{\varepsilon^2}.$$

The time limit is T, the success rate of the best option is ρ, the number of options is n, and the error tolerance is ε. Lower error tolerance increases the runtime, and higher tolerance speeds convergence. Adding options increases runtime by a logarithmic factor. We normalize by empirical success, so $\rho = 1$, and we use a standard error tolerance of $\varepsilon = 0.1$. The optimal learning rate is $\eta = \varepsilon/2 \cdot \rho$, which for our settings is 0.05.

4 CHARACTERIZING THE SEARCH SPACE

Ignoring recombination, the search space of an evolutionary algorithm is defined in terms of the problem representation and the genetic operators that the algorithm can apply. We focus on the space of possible mutations to a given defective program. In particular, we consider only mutations that are neutral with respect to the program's positive test suite. As mentioned earlier, any set of mutations that repairs a defect must also be neutral (pass the test suite).

We are interested in searching beyond single-mutation changes without breaking the program's functionality to improve the chances of finding a repair. To accomplish this, we study the impact of combining multiple neutral mutations on the probability of finding repairs in the space—that is, the chance that a random sample of mutations, in combination, repair the defect. Although many of the defects we study can be repaired with one or two mutations, they can be found more effectively by considering many neutral mutations in combination. The extra mutations can be removed later using standard patch minimization techniques. This section extends a prior analysis [Renzullo et al. 2021] by extending it to a larger set of programs, including Java programs sampled randomly from the Defects4J benchmark. First, we study the probability of negative interactions among individually neutral mutations (negative epistasis), then we ask if the number of mutations is correlated with the probability of finding repairs, and finally we present a theoretical model that accounts for our empirical results.

4.1 Combining Neutral Mutations

RQ 1: How Many Neutral Mutations Can Be Safely Combined? 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). A well-studied early benchmark program in APR [Le Goues et al. 2012], `units` has a defect that crashes with a segmentation fault on some inputs because of an incorrect buffer size.

We generated all possible 41,344 atomic mutations to the buggy version of `units` using GenProg's mutation operators—that is, all possible applications of the *append*, *delete*, and *swap* operators applied to the parts of the program covered by the test suite. Of these, 14,726 of the 41,344

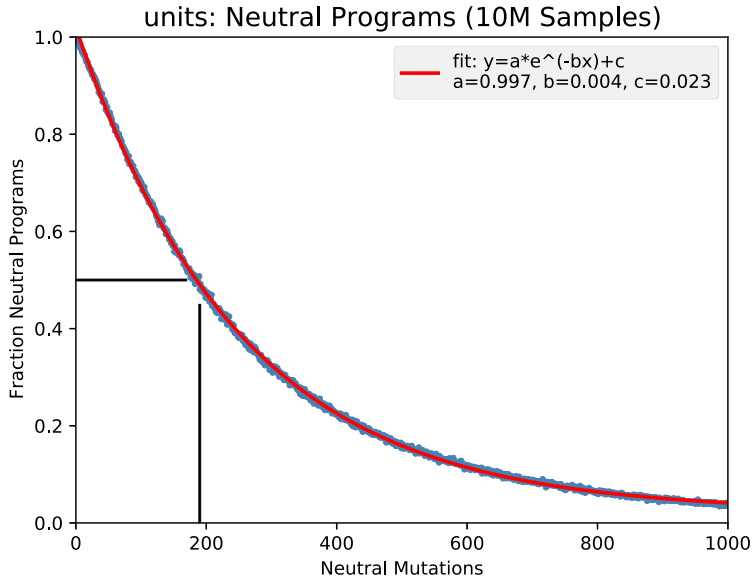

Fig. 1. Effect of combining neutral mutations. Each point (blue dots) is the mean of 10,000 independent random samples. Each sample combines n (x-axis) individually neutral mutations. The y-axis is the fraction of samples that remain neutral after the n mutations are applied. Empirical data shown in blue and the best-fit model in red. $R^2 = 0.999$.

mutations were neutral (35.6%). Next, we studied the effect of applying n randomly chosen neutral mutations from this pool to the same program, for n ranging from 1 to 1,000. Each of the samples is independent, so for each value of n, the observed fraction of variants that remains neutral (still pass the positive tests) approximates the probability that a program with n random neutral mutations applied will remain neutral. The results are shown in Figure 1. Remarkably, even when 190 mutations are applied (annotated point), half of the resulting program variants remain neutral. This means that we could test 190 neutral mutations at once, with an expected overhead factor of only around 2. For the simplest case, a defect that can be patched with a single mutation, a search that tested 190 individually neutral mutations together would be about 95 times faster than a naive strategy that tests one mutation at a time. Additionally, if the naive strategy is not assumed to benefit from a precomputed pool of neutral mutations, it would be ≈3 times slower still (because ≈ 30% of mutations are neutral). In other words, for this particular defect and program, a search that sampled 190 individually neutral mutations together would be 95×3 times faster than the naive approach.

It is reasonable to wonder, at this point, about the strength of test suites. After all, if the mutations combined here are only neutral because the test suite is too weak to detect them, then the diversity they introduce will not be useful, because it will lead to overfitting patches. Other work has examined the phenomenon of neutral mutations in software, finding in Java [Harrand et al. 2019] and in a variety of other languages [Schulte et al. 2014] that neutral mutations are not only the result of test weakness. In the following section, we examine whether combinations of neutral mutations are able to patch defects in software. We consider the problem of correctness (also known as patch overfitting) in Section 6.4.

4.2 Optimal Combinations of Neutral Mutations

RQ 2: How Often Do Neutral Mutations Generate a Patch? Figure 1 illustrates the success rate when evaluating n mutations at a time. It also shows the cost incurred by negative epistasis

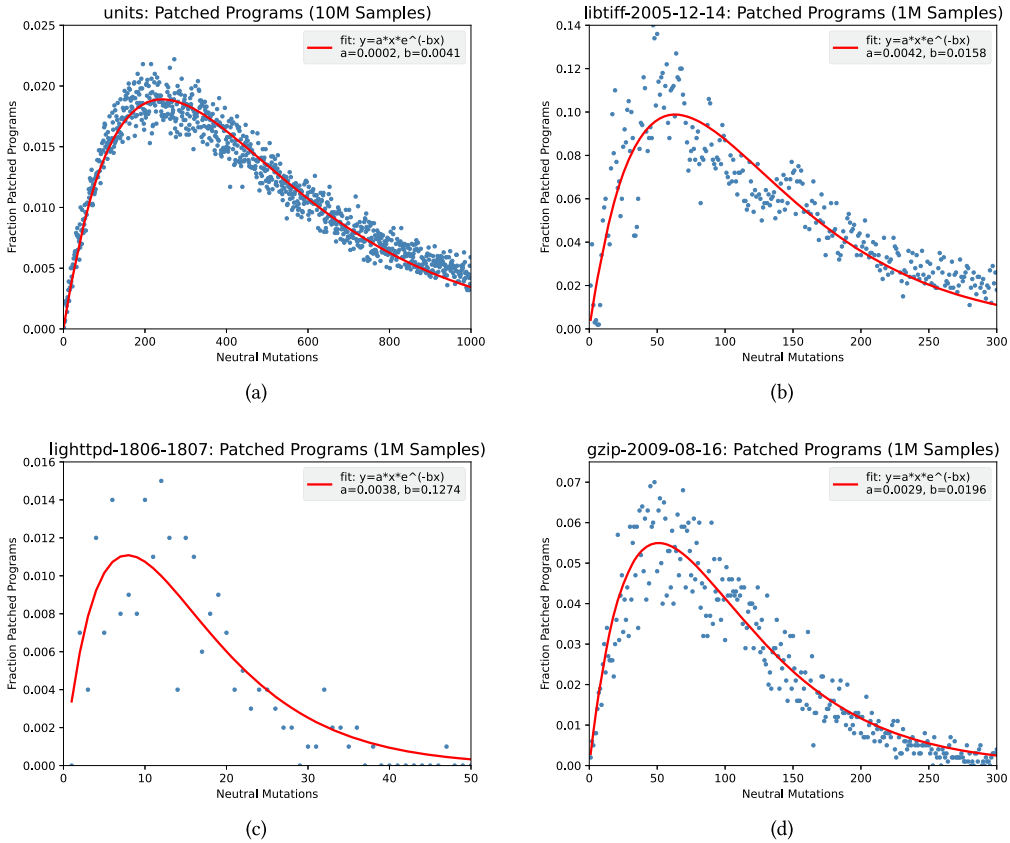

Fig. 2. Probability of finding patches. For each value of n, 10,000 (a) or 1,000 (b–d) independent random samples of n precomputed neutral mutations were applied, for n = 1 to 1,000. The y-axis reports the fraction of resulting program variants that patch the defect. Empirical data shown in blue and the best-fit model in red. (a) $R^2 = 0.975$. (b) $R^2 = 0.797$. (c) $R^2 = 0.795$. (d) $R^2 = 0.907$.

among the sampled mutations. The naive benefit (i.e., the gain in efficiency from sampling many mutations simultaneously) is a linear function in n. Combining the naive benefit with the effect of interaction between mutations yields the probability of finding a patch.

Figure 2 plots the probability of finding patches (y-axis) for each choice of n for four different programs. As before, each choice of n is sampled independently, so the empirically observed fraction of sets of n mutations approximates the probability that a program combining n neutral mutations will patch the defect. Programs containing unnecessary mutations (likelier as n increases) can be minimized to the necessary mutation set using standard techniques, as discussed further in Section 6.2. Beginning with the `units` example, Figure 2(a) shows a single-peaked distribution, with the optimum estimated by a best-fit model (red curve) at $n = 269$. This is close to the empirically observed maximum (blue dots) at $n = 271$. The fraction of patches at the peak is 0.022, compared to 0.00022 for single mutations: two orders of magnitude better (Table 1).

Figure 2(b) through (d) shows similar results for three other defects in large open source C programs taken from the ManyBugs benchmark [Le Goues et al. 2015]: `libtiff-2005-12-14`, `lighttpd-1806-1807`, and `gzip-2009-08-16`. These programs were selected because they expose different fault types and each is from a different large open source library. The defect in `libtiff-2005-12-14` causes the program to behave incorrectly if given specific inputs (although

Table 1. Patch Probability as a Function of the Number of Neutral Mutations

Program	Description	Language	kLoC	Single Mutation Patch Probability	Patch Probability at Peak	Optimal N (No. of Mutations)	Advantage
gzip-2009-08-16	Compression	C	480	0.270%	7.70%	48	28.4
gzip-2009-09-26	Compression	C	480	0.028%	0.90%	34	32.0
libtiff-2005-12-14	Image Processing	C	77	0.810%	14.00%	46	17.2
lighttpd-1806-1807	Web Server	C	62	0.041%	1.50%	11	36.2
ccrypt	Encryption	C	7	0.980%	39.50%	43	39.9
look	Dictionary	C	1	1.580%	69.60%	110	43.9
units	Conversion	C	1	0.022%	2.20%	271	99.5
Chart 13	Plotting	Java	96	0.100%	0.10%	1	1.0
Chart 26	Plotting	Java	96	0.050%	1.50%	85	30.0
Closure 13	JavaScript Compiler	Java	90	1.050%	5.15%	13	4.9
Closure 19	JavaScript Compiler	Java	90	0.150%	2.15%	91	14.3
Closure 22	JavaScript Compiler	Java	90	0.500%	0.50%	1	1.0
Closure 107	JavaScript Compiler	Java	90	0.140%	0.23%	2	1.67
Lang 10	Java Language	Java	22	0.100%	0.10%	1	1.0
Math 8	Math Utilities	Java	85	0.500%	5.45%	75	10.9
Math 28	Math Utilities	Java	85	0.850%	0.85%	1	1.0
Math 80	Math Utilities	Java	85	0.090%	0.32%	9	3.5

10^6 mutants were generated for each C defect except units, which used 10^7. All Java defects show results from 10^4 mutants. "Single Mutation Patch Probability" is the percentage of single mutations ($N = 1$) that patch the defect. "Patch Probability at Peak" is the percentage of samples that patched the defect at the optimal number of mutations. "Optimal N (No. Mutations)" is the number of combined mutations at the peak. "Advantage" is the quotient of the optimal and single-mutation patch densities, which quantifies the advantage of finding and exploiting the optimum.

it does not crash); the defect in `lighttpd-1806-1807` causes some webserver connections to hang indefinitely, even when servers are configured according to the correct specification; and the defect in `gzip-2009-08-16` causes the program to crash when parsing exactly one trailing NUL byte. The location of the peak varies with each example, but in these examples it always exists and occurs a large number of mutations away from the original program. Each of the four distributions in Figure 2 is unimodal, and statistical fitting suggests a common model for all four. If the search can be guided to these optima, it find repairs faster and perhaps more often.

4.3 Modeling the Observed Distribution

RQ 3: Can We Describe the Effects of Combining Neutral Mutations in a General Model? The preceding experiments shed light on the structure of the search space, but they cannot be used directly. First, the location of the peak varies with the particular program and defect. Second, an APR search is typically terminated after a single patch is discovered, so in most cases we do not expect to observe the distribution of patch probability. We can address the first issue using online estimation and the second by using the probability of finding neutral mutations (easily measured) as a proxy for patch probability. We propose a mathematical model for the expected fraction of patched programs as a function of n, the number of mutations to be combined. Negative interactions (epistasis) are much more frequent than positive ones, but interactions are rare [Bruce et al. 2019; Renzullo et al. 2018]. We assume (conservatively) that all interactions are negative, ignoring positive epistasis. Because patches are rare, it is often impractical to directly measure enough to show a pattern. However, we can use the number of neutral mutations in, for example, Figure 1 to predict the optimum probability of finding patches. Note that the coefficient of the negative exponential (the parameter labeled b) in Figure 1 closely matches that in Figure 2(a). This relationship allows us to use online estimation to predict the optimal probability of finding patches.

Our model assumes that patches and epistatic interactions are uniformly distributed—that is, all neutral mutations are equally likely to produce a patch. Thus, the benefit of adding one randomly selected neutral mutation is an additive constant. But the cost increases with n because

the probability of negative epistasis increases. Near the original program (low n), the benefit of adding a mutation outweighs the cost. As we accumulate mutations, however, the marginal cost rises, matching the benefit at the optimum and then dominating after the peak.

We model this using a linear function for benefit and a negative exponential for cost, as shown in Figure 2. The form $a * x * e^{-bx} + c$ provides an excellent fit for our empirical data. We estimated a, b, and c using Levenberg-Marquardt nonlinear least squares regression [Marquardt 1963], and the coefficients of determination for each defect are reported in the caption.

4.4 Do We Observe Similar Dynamics in Other Programs?

We next consider the behavior of other programs by running the same experiments described earlier and observing where the peak probability of finding patches occurs (see Table 1). We chose a representative, stratified random sample of programs that vary in size (1k to 480k lines of code), language (C vs. Java), and repair difficulty (rarity of patches). From the ManyBugs benchmarks [Le Goues et al. 2015], we evaluated four defects in three large C programs: `libtiff`, `lighttpd`, and `gzip`. From the GenProg benchmarks [Le Goues et al. 2012], we evaluated one defect each in three small C programs: `ccrypt`, `look`, and `units`. From the Defects4J benchmark [Just et al. 2014], we evaluated 10 randomly selected defects from four open source Java projects. As in the `units` example, mutations to the buggy version of each program were generated and filtered for neutrality, then combined.

We find that each studied program has a single peak in its distribution of patch probability (see Table 1). Most optima occur many mutations away from the original program. In Table 1, the "Advantage" column shows that patches are between 17.2 and 99.5 times more frequent at the optimum value of n for C programs (average: 42.4). For Java programs, the range is from 1.0 to 30.0 (average: 6.9). Informally, these values represent the theoretical advantage of a search algorithm that can exploit this information. The location of the optimum does not correlate with obvious structural properties such as program size, test suite size, and number of functions. So, for a given defect, we have not been able to identify an easy-to-measure program property that predicts the optimal number of mutations to combine. However, we can use what we have learned about the unimodal distribution of patch probability in the search space. We use a well-known statistical learning algorithm: MWU [Arora et al. 2012]. MWU estimates a distribution by making predictions based on a model, then evaluating the results, iteratively updating its model. By incorporating MWU into our search algorithm, it can estimate the location of the optimum while it searches.

5 THE MWREPAIR ALGORITHM

We present MWRepair, a search-based algorithm that discovers patches by adaptively balancing exploration and exploitation. MWRepair's inputs are a program with a defect, the program's test suite (consisting of positive and negative tests), and a precomputed set of program-specific neutral mutations. MWRepair's output is a minimized set of mutations that patches the defect. If the search fails, the set is empty. MWRepair guarantees that the search minimizes regret (i.e., wasted computational effort).

APR algorithms are computationally expensive, and the dominant cost is running test cases. To address this, we precompute a large pool of neutral mutations on a per-program basis. The pool can then be applied whenever a new defect emerges in that program. Precomputation removes a computationally expensive step from the inner loop of the search algorithm and is easily parallelized [Renzullo et al. 2021].

Algorithm 2 precomputes the set, or pool, of neutral mutations. The algorithm first generates a set of N mutations (summarized in Figure 3), and then it validates each mutation against the test suite. This is the dominant cost and is trivially parallel [Renzullo et al. 2021]. It can be computed

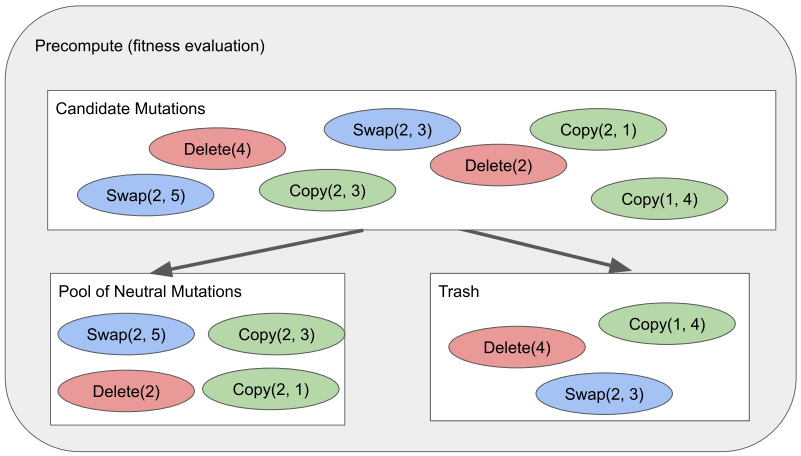

Fig. 3. MWRepair filters mutations that use GenProg's operators based on their fitness. Mutations, shown here in shorthand, correspond to code edits. They are screened for fitness via test cases. Neutral mutations are retained in a pool to be used during the search, whereas non-neutral mutations are discarded.

ALGORITHM 2: Precomputation: Until the limit T is reached, the algorithm selects a random, untested mutation, applies it to the original program, and retains it if it passes the positive tests.

Input: *A program,* ***P****, a set of positive test cases,* ***S****, a fitness function* ***f(P, S)*** $\rightarrow \mathbb{N}$*, and a limit,* ***T****.*
Output*: A set,* ***M****, of neutral mutations.*

```
N ← get_mutation_list(P)
M ← ∅
for lim ← 1 to T do
    m ← pop_random(N)
    P' ← apply_mutation(P, m)
    if f(P', S) == f(P, S) then
        M ← M ∪ m
Return M
```

at different points of the software lifecycle, including before development or during continuous integration testing. Later, when a defect surfaces, the pool of neutral mutations supplies the atoms that MWRepair combines. Over time, developers can update the set of neutral mutations as needed whenever the test suite changes, eliminating mutations that no longer pass the new tests or adding new mutations to replenish or expand the pool (see Section 6.5). The storage cost for the pool is small: using the popular patch representation [Le Goues et al. 2012] requires only 100 kb to store 10,000 mutations (i.e., 10 bytes per mutation). This storage requirement grows proportionally to the logarithm of the number of nodes in the program's abstract syntax tree.

Initially, MWRepair models the search space using an exponentially decreasing probability as the number of mutations (step size) increases (Section 4). In other words, as n (the number of combined mutations) grows, the algorithm is exponentially less likely to sample that n. MWRepair also employs an exponential binning strategy to reduce the number of options by a logarithmic factor. Instead of each option having its own probability, options are binned by the logarithm of their value. MWRepair selects a bin and then uniformly samples a value from within that bin. In

ALGORITHM 3: MWRepair samples a number of mutations, evaluates the resulting program, and updates its model, terminating if a patch is found or the time limit is reached.

Initialization: *Select a learning parameter $\eta \leq 1/2$, and a time limit, **T**. Set option weights $w_i^{(1)}$ to initial model.*
Input: *a set, **M**, of neutral mutations, a program, **P**, a set of test cases, **S**, and a fitness function **f(P, S)** $\rightarrow \mathbb{N}$.*
Output: *A set of mutations which patch the program.*
for $t \leftarrow 1$ **to** T **do**
    step_size $\leftarrow$ select($w_i^{(t)}$)
    muts $\leftarrow$ random_sample(**M**, step_size)
    P' $\leftarrow$ apply_mutations(**P**, muts)
    if ***f(P', S)*** $== \|S\|$ **then**
        Return minimize(muts).
    else if ***f(P', S)*** $\geq$ ***f(P, S)*** **then**
        $w_{step\_size}^{(t+1)} \leftarrow w_{step\_size}^{(t)} * (1.0 + \eta))$

principle, MWRepair could start with a uniform probability distribution or any other model, but Section 4 and our preliminary experiments suggest that exponential binning is a good fit for our problem domain. Choosing a good initial model reduces the runtime, which is proportional to the disagreement between the initial model and the ground truth search landscape.

MWRepair uses its current model to select an n, then chooses n mutations randomly from the pool, and applies them to the original program to generate a program variant. Each mutated program is then evaluated to determine its fitness, and the model is updated according to the MWU rule. Our implementation of MWU uses a reward of 1.0 for a set of mutations that performs as well as the original program and 0.0 otherwise. As the weight vector is updated, it biases decisions toward the empirically observed optimum in the search space. Figure 4 gives a schematic overview of MWRepair, and Algorithm 3 gives the pseudocode.

5.1 Summary

In summary, MWRepair is an online search algorithm that estimates the optimal number of mutations to combine while searching for software patches. It uses neutrality as a proxy for patch probability (Section 4.3), which allows MWRepair to focus attention on the part of the search space with the highest chance to find patches. To the best of our knowledge, three aspects of MWRepair are novel compared to existing search-based repair algorithms: (1) decomposition of the search problem into one step that can be precomputed and easily parallelized and one that implements the search, (2) considering massive numbers of mutations simultaneously, and (3) the use of online learning to change *how* the search is conducted rather than *what* it considers. All three are relevant to the dominant cost of search-based APR algorithms: validating solutions by running program variants on test cases. These aspects enable low-latency exploration of a large region of the search space, and we hypothesize that they will enable MWRepair to find novel patches. In the next section, we evaluate MWRepair's performance empirically.

6 EXPERIMENTAL RESULTS

We evaluated MWRepair's performance on the popular Defects4J set of benchmark Java (buggy) programs. We first analyzed MWRepair's performance in detail[1] and then compared its

[1]Results data and the code used to generate them are available at https://doi.org/10.5281/zenodo.7906893.

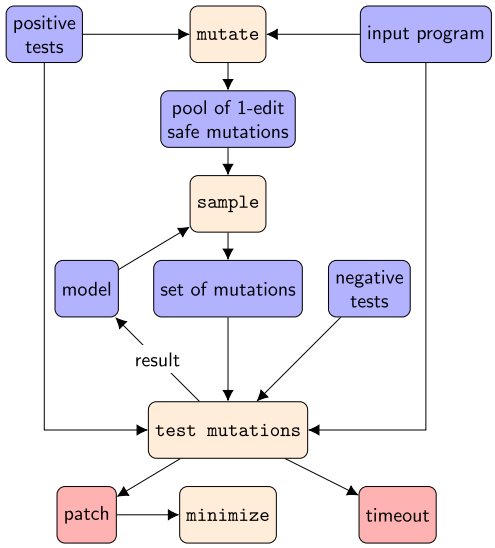

Fig. 4. MWRepair as defined in Algorithm 3. First, `mutate` uses test information to generate safe (neutral) mutations. `Sample` combines them and then they are evaluated in `test_mutations`. Repairs are added to a database, non-repairs are discarded, and evaluation results are used to update the weights. Termination conditions are in red, data in blue, and subroutines in tan.

performance to state-of-the art APR methods. Many of these do not use evolutionary methods, although most use some form of random mutation. Our implementation uses mutation operators modeled after GenProg, an evolutionary APR repair method, so our first analysis compares MWRepair directly to GenProg. After describing our experimental design, we ask how many patches MWRepair finds compared to GenProg (Section 6.1), the study patch size (Section 6.2), consider MWRepair's ability to find multi-edit repairs (Section 6.3), and examine the correctness of MWRepair patches (Section 6.4). We also examine the life span of generated neutral mutations (Section 6.5). Finally, we compare MWRepair's performance to all published APR algorithms that target Defects4J (Section 6.6).

6.1 RQ 4: Does MWRepair Improve the Ability of Java-Based GenProg Tools to Discover Patches and (Unique) Repairs?

For each Defects4J scenario (program/defect pair), we precomputed a pool of neutral mutations. For smaller programs, this pool is exhaustive; for larger programs, we computed a sample, limited to a total of 10k test suite evaluations (usually resulting in 3,000 neutral mutations). We then ran MWRepair 100 times with a limit of 1,000 test suite evaluations and a learning rate of $\eta = 0.05$. These parameters (in particular, the budget of 1,000 test suite evaluations) were chosen to provide a fair comparison with published results for other algorithms taken from the papers cited. We acknowledge that comparing to published results run on other hardware precludes an exact comparison, but we mitigate this by focusing on repairs found, not execution time or memory usage. We report the number of scenarios for which MWRepair found patches. We also studied each patch for correctness by comparing it manually to reference human patches that are supplied as part of the of benchmark. In Section 6.6, we then place the repairs found by MWRepair in the context of the results of 34 other published algorithms. However, many of these other algorithms make stronger assumptions about the information available to the APR algorithm than others—one reason that we focus first on comparing to GenProg. The algorithms that make stronger assumptions

Table 2. Patch (Repair) Discovery

Algorithm	No. of Defects	Plausible Patches (Repairs)	Average Success Rate	Variance	*p*-Value
MWRepair	357	54 (16)	0.151 (0.045)	0.129 (0.043)	
jGenProg	357	21 (6)	0.059 (0.017)	0.056 (0.017)	5.87E-7 (7.35E-3)
GenProg-A	357	30 (2)	0.084 (0.006)	0.077 (0.006)	8.16E-4 (1.62E-4)

MWRepair, jGenProg, and GenProg-A were each tested on the 357 defects in Defects4J v1.0. Variance reports the uncertainty of the success rate measurement. The *p*-values are reported for two-tailed *t*-tests (df = 356) evaluating the null hypothesis that the performance of the algorithm is equal to the performance of MWRepair.

are annotated as such in Table 3 (presented later) because they assume perfect fault localization, either by requiring that the buggy method be given as input or that the exact location where the human repair was applied be identified.

Evaluating MWRepair's Ability to Discover Repairs. There are two independent implementations of the GenProg algorithm for Java that have been evaluated against Defects4J: jGenProg [Martinez and Monperrus 2016] and GenProg-A [Yuan and Banzhaf 2020a]. Each experiment was run 100 times, and the union of these results is presented in Table 2, which reports results comparing MWRepair's performance to both. This methodology follows that used in the papers we compare to here, as well as in Appendix A.

One concern about manually verifying the correctness of proposed patches is human fallibility [Liu et al. 2021]. To control against this potential source of bias, we cross validated all patches that we annotated against repairs generated by algorithms that use similar base mutation operators (i.e., jGenProg, GenProg-A, jKali, and RSRepair-A). In two cases (Closure 21 and Closure 22), we were more strict than the prior literature and annotated as patches two programs that other authors had labeled as repairs. For consistency with prior work, we report these as repairs; in all other cases, our annotations agree. Table 2 shows these results. Single factor ANOVA on all three algorithms shows a substantial difference in mean repair performance (p = .00127), with MWRepair outperforming the other two methods. This advantage is statistically significant: using the Sidak-Bonferroni method to control for the effect of multiple hypothesis testing and setting a familywise error tolerance of 0.05, p-critical = .0253. We conclude that MWRepair *repairs* significantly more defects than comparable implementations of GenProg.

Evaluating MWRepair's Ability to Repair New Defects. Here, we ask how often MWRepair finds repairs to defects that other comparable algorithms have missed. Figure 5 visualizes the data summarized in Table 2 and situates it in the context of the APR literature, which is summarized in Table 3 and discussed in Section 6.6. Each repair is categorized as rare (repaired by only one of the 35 algorithms summarized in the table), uncommon (repaired by two to four algorithms), or common (repaired by five or more algorithms). GenProg-A finds only common repairs and jGenProg finds a combination of common and uncommon repairs, but no rare ones. MWRepair not only finds more repairs than comparable algorithms, but it also finds rarer ones as well.

6.2 RQ 5: Do Patches Found by MWRepair Minimize to Single Mutations or Can MWRepair Discover Multi-Edit patches?

Because MWRepair searches by combining multiple mutations aggressively, it is likely to report patches that consist of many mutations, only some of which are needed to address the defect. Figure 6 illustrates the size of this effect, showing on the horizontal axis the number of mutations in each patch before and after minimization. Consistent with earlier results [Weimer et al. 2013],

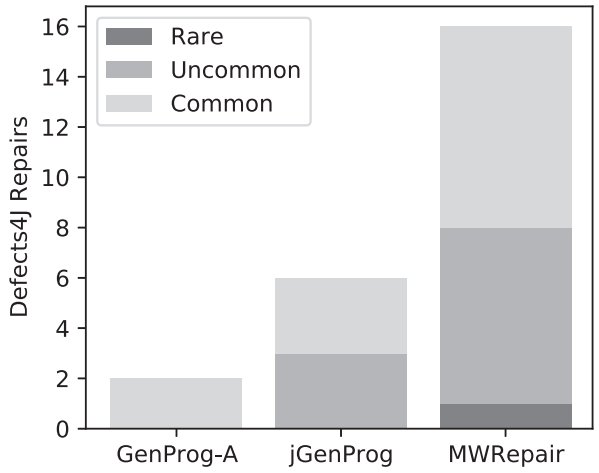

Fig. 5. Repair novelty. Different shades indicate rare (unique to that algorithm), uncommon (found by two to four algorithms), or common (found by five or more algorithms). Data compiled from MWRepair results and all published results for Defects4J (Section 6.6).

most patches are small and require only one or two mutations. However, this is not usually how MWRepair discovers them (see "Before Minimization"in the figure). For example, one of the patches to the Closure 19 defect was found by evaluating 127 mutations, whereas the minimized set contains only 1 of them. A similar phenomenon has been discovered in evolutionary biology [Kimura 1983], where the accumulation of neutral mutations is thought to help populations explore the high-dimensional space of mutations and maintain diversity. At the other end of the spectrum, the Lang 50 patch contains two mutations, both of which are necessary. The minimization procedure effectively controls patch size, retaining the necessary mutations while eliminating any that are not necessary.

6.3 Multi-Edit Repairs

Multi-edit repairs are important for advancing the APR field. For example, 71.77% of the reference patches in the Defects4J benchmark contain multiple chunks (contiguous regions of edited code) [Sobreira et al. 2018]. Although simpler solutions are sometimes possible, the edits that human developers make when repairing most of these defects are more complex than those that are expressible by single mutations. With this context, we examined the MWRepair results for multi-edit patches and found four that had more than one required mutation. For each defect (Closure 85, Lang 50, Math 46, and Math 35), the mutations applied to different locations in the program. Three of these four patches were correct repairs.

MWRepair Finds Two Multi-Edit Patches That Other Algorithms Do Not. Lang 50 is repaired by MWRepair, which makes two mutations to two similar yet distinct functions in a library that handles date formatting. No other APR algorithm has repaired this defect. The small dissimilarities between the functions where the mutations are needed presents challenges for algorithms like Hercules [Saha et al. 2019], which target multi-edit defects but depend on making similar or identical mutations at related locations. The multi-hunk nature of the bug means that it is out of scope for all algorithms that make only single mutations (most of them). We note that the code presented in Listing 1 is not semantically equivalent to the human repair: it modifies the `key` variable later than the human repair. This causes a value to be regenerated rather than fetched from cache (which has performance implications but generates the correct values).

Table 3. Summary Data for Java APR Algorithms on the Defects4J Benchmark

Algorithm	Chart (26)	Lang (65)	Math (106)	Time (27)	Closure (133)	Total (357)
ACS	2 (2)	3 (3)	16 (11)	1 (1)	0 (0)	22 (17)
ARJA-p	15 (8)	22 (9)	42 (17)	5 (1)	0 (0)	84 (35)
AVATAR	12 (5)	13 (4)	17 (3)	0 (0)	15 (7)	57 (19)
CapGen	4 (4)	5 (5)	13 (13)	0 (0)	0 (0)	22 (22)
Cardumen	4 (2)	0 (0)	6 (1)	0 (0)	2 (0)	12 (3)
CoCoNuT†	7 (7)	7 (7)	16 (16)	1 (1)	9 (9)	40 (40)
ConFix	13 (4)	15 (5)	37 (6)	6 (1)	21 (6)	92 (22)
CURE†	10 (10)	9 (9)	19 (19)	1 (1)	14 (14)	53 (53)
DEAR	8 (8)	8 (8)	21 (21)	3 (3)	7 (7)	47 (47)
DLFix	5 (5)	5 (5)	12 (12)	1 (1)	6 (6)	29 (29)
DynaMoth	6 (0)	2 (0)	13 (1)	1 (0)	0 (0)	22 (1)
FixMiner	14 (5)	2 (0)	14 (7)	1 (0)	2 (0)	33 (12)
GenPat	8 (4)	11 (4)	13 (4)	1 (0)	7 (6)	40 (18)
HDRepair*	2 (0)	6 (2)	7 (4)	1 (0)	7 (0)	23 (6)
Hercules	9 (8)	15 (10)	29 (20)	5 (3)	14 (9)	72 (50)
JAID*	11 (5)	22 (6)	28 (9)	2 (1)	27 (13)	90 (34)
jKali	4 (0)	4 (2)	8 (1)	0 (0)	8 (3)	24 (6)
jMutRepair	4 (1)	2 (0)	11 (2)	0 (0)	5 (2)	22 (5)
kPAR	13 (3)	18 (1)	21 (4)	1 (0)	10 (2)	63 (10)
LSRepair	3 (3)	8 (8)	7 (7)	0 (0)	0 (0)	18 (18)
Nopol	6 (0)	6 (1)	18 (0)	1 (0)	0 (0)	31 (1)
NPEFix	5 (0)	0 (0)	3 (0)	0 (0)	0 (0)	8 (0)
RSRepair-A	4 (0)	3 (0)	12 (0)	0 (0)	22 (4)	41 (4)
Restore	4 (4)	6 (6)	9 (9)	1 (1)	21 (21)	41 (41)
RewardRepair	4 (4)	3 (3)	14 (14)	1 (1)	7 (7)	29 (29)
SelfAPR†	9 (6)	12 (6)	24 (16)	3 (0)	23 (16)	71 (44)
SequenceR†	3 (3)	2 (2)	9 (6)	0 (0)	5 (3)	19 (14)
SimFix	8 (3)	16 (5)	25 (10)	0 (0)	19 (7)	68 (25)
SketchFix*	8 (6)	4 (3)	8 (7)	1 (0)	5 (3)	26 (19)
ssFix	7 (2)	12 (5)	26 (6)	4 (0)	11 (1)	60 (14)
TBar	16 (7)	21 (6)	23 (8)	0 (0)	12 (3)	72 (24)
VarFix	0 (0)	0 (0)	24 (11)	0 (0)	11 (6)	35 (17)
GenProg-A	5 (0)	1 (0)	9 (0)	0 (0)	15 (2)	30 (2)
jGenProg	5 (0)	2 (0)	12 (4)	0 (0)	2 (2)	21 (6)
↓	↓	↓	↓	↓	↓	↓
MWRepair	7 (1)	10 (3)	20 (5)	1 (0)	16 (7)	54 (16)
Combined Algs.	24 (17)	43 (29)	75 (48)	11 (5)	72 (41)	225 (140)

*Data from experiments where the buggy method was provided to the algorithm.
†Data from experiments where the exact line with the bug was provided to the algorithm.
The benchmark consists of five different programs (columns in the table), each of which has multiple defects (shown in parentheses). Each row reports the number of defects patched (correctly repaired in parentheses) by each algorithm. Data from APR algorithms given additional input (e.g., perfect fault localization) are indicated with a dagger (†) or asterisk (*). "Combined Algs." reports the union over all presented algorithms as an ensemble success rate.

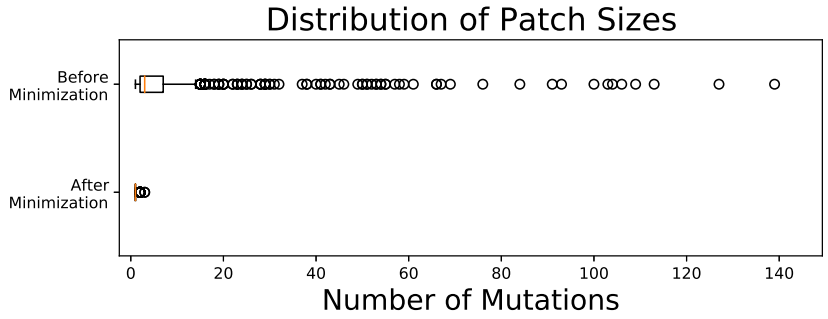

Fig. 6. Patch size before and after minimization.

```
--- FastDateFormat.java.original
+++ FastDateFormat.java.mwrepair
@@ -281,6 +281,9 @@
         if (timeZone != null) {
             key = new Pair(key, timeZone);
         }
+        if (locale == null) {
+            locale = Locale.getDefault();
+        }

         if (locale != null) {
             key = new Pair(key, locale);
@@ -471,6 +474,7 @@
             if (locale == null) {
                 locale = Locale.getDefault();
             }
+            key = new Pair(key, locale);
             try {
                 SimpleDateFormat formatter = (SimpleDateFormat)
                     DateFormat.getDateTimeInstance(dateStyle,
                     timeStyle, locale);
```

Listing 1. Multi-Edit Repair to Lang 50

Closure 85 is an interesting case: there were three potential locations for mutations that led to patches, and each MWRepair patch mutated exactly two of them. Other APR algorithms have not patched this defect. For the other three defects that required multi-edit patches, there were exactly two locations involved in each patch. In each case, all of the patches that MWRepair discovered touched the same locations, although they used different mutations in some of the independent MWRepair runs. Further, the conserved locations (and sometimes also the applied mutations) are identical to the modifications made by the reference human repair.

6.4 RQ 6: How Many of the Patches MWRepair Identifies Are Correct Repairs?

One of the challenges of using test cases to validate patch correctness is that test cases are often incomplete or may not test for undesirable side effects—that is, the proposed patch may be overfit to the test cases and not generalize to other inputs. A popular way to compensate for potential

Table 4. Granular Analysis of MWRepair Repairs

	Same Location	Different Location	Total
Same Modification	56.25%	6.25%	62.5%
Different Modification	31.25%	6.25%	37.5%
Total	87.50%	12.5%	

Each column reports the percentage of repairs that agree/disagree with the reference human repair's location (by line of code). Each row indicates the percentage of repairs that agree/disagree with the human repair's code modification.

overfitting involves human inspection, usually a comparison to a human-written reference patch (e.g., [Qi et al. 2015; Ye et al. 2020]). This is not a perfect solution either, since are many ways to remedy a defect—not all deviations from a pre-existing example are incorrect—and since human-written code often introduces further defects or does not fully resolve the root cause of a given defect [Gu et al. 2010]. Despite these issues, we report the results of that comparative standard here, annotating programs with identical semantics to the human-written reference patch as correct. We note one exception, Lang 50, which was discussed in detail in Section 6.3.

Wang et al. [2019] dissected a corpus of repairs from Defects4J. They introduced four categories of repairs, focusing on location in the code and the particular modification. In their terminology, these are SLSM (same location same modification, SLDM (same location different modification), DLSM (different location same modification), and DLDM (different location different modification). Table 4 shows how this classification applies to MWRepair: the location of MWRepair repairs agrees with the reference human repair in 87.5% of cases, and the modification agrees in 62.5% of cases. This result demonstrates the importance of fault localization [Moon et al. 2014], and the ability of mutations to help pinpoint this localization is an area of active research referred to as mutation-based fault localization [Moon et al. 2014].

To see why this level of detail can assist analysis, consider the collection of patches suggested by MWRepair for the Lang 50 defect. Across multiple runs of MWRepair, nine patches were generated. Of these, each modified the same pair of functions that the human patch modified. However, not every proposed modification was correct or located at the identical line of code as the human-modified version. One of the proposed patches is correct and applies the same modification at the same location (SLSM), identically to the human version. In another case, the MWRepair patch applies the same modification at a slightly different location (DLSM). Additionally, a third MWRepair patch is partially correct: the first function is modified in the same location in the same way (SLSM), but the second function is modified at the same location in a different way, which introduces a side effect (SLDM). Each of the other seven patches are located in the same two functions, but the modifications are overfit to the test cases (e.g., replacing the code that correctly throws an error with an innocuous initialization statement). This avoids the tested problematic behavior but does not patch the underlying defect.

6.5 RQ 7: Do Neutral Mutations Remain Neutral or Are They Invalidated over Time by Changes to the Test Suite as a Software Project Matures?

MWRepair can take advantage of a set of precomputed neutral mutations as the seeds of repair when a defect emerges. As such, it is important to quantify how far in advance this work can be done. To answer this question, we consider the space of neutral mutations that TBar makes on each scenario in Defects4J 1.0, excluding Chart, which does not provide a git history. We selected TBar because it has a smaller set of mutations that can be exhaustively generated and analyzed. Although the TBar mutation operators are somewhat different from those that GenProg uses, both

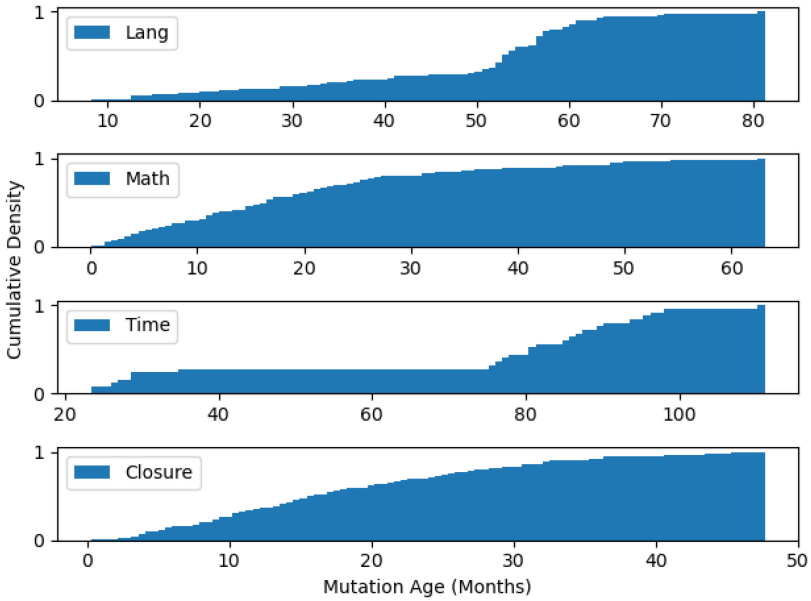

Fig. 7. Each subplot shows a cumulative density graph of the average age of neutral mutations for defects in the labeled Defects4J project. Mutation age is the duration elapsed between the modified code's last change and the date of the developer repair. It represents the durability of generated mutations over time.

tools use spectrum-based fault localization to prioritize where mutations are applied, and it is these locations that determine the durability of applied mutations.

For each of these mutations, we analyze the repository's history to determine when the part of the program touched by the mutation was last modified. This is done using PyDriller, a Python interface for analyzing git repositories. It checks, for each commit, the lines changed in that commit and uses `git blame` to determine the most recent prior modification of that line. The mutation's age is the time between the code's last modification and the time that the developer repair was applied. The neutral mutation could be generated at any time in that interval.

Our data show that this age is often large—even though a project is under active development, individual segments of it are stable for quite some time—even segments containing defects that are later fixed. We summarize the results of this analysis in Figure 7, showing the average age in months of each mutation in the neutral pool. For all projects, more than half of the mutations in the pool are at least 1 year old. For the defects in the Time library, most mutations are more than 6 years old. Because of the large time span available to a precompute process, even a modest deployment should be able to keep up with changes in the codebase as it evolves.

6.6 RQ 8: How Does MWRepair-Enhanced GenProg Compare to the State of the Art?

Table 3 compiles data taken from results published for 34 other Java APR algorithms on the Defects4J v1.0 benchmark: ACS [Xiong et al. 2017], ARJA-p [Yuan and Banzhaf 2020b], AVATAR [Liu et al. 2019b], CapGen [Wen et al. 2018], Cardumen [Martinez and Monperrus 2018], CoCoNuT [Lutellier et al. 2020], ConFix [Kim and Kim 2019], CURE [Jiang et al. 2021], DEAR [Li et al. 2022], DLFix [Li et al. 2020], DynaMoth [Durieux and Martinez 2016], ELIXIR [Saha et al. 2017], FixMiner [Koyuncu et al. 2020], GenPat [Jiang et al. 2019], GenProg-A [Yuan and Banzhaf 2020a], HDRepair [Le et al. 2016], Hercules [Saha et al. 2019], JAID [Chen et al. 2017], jGenProg [Martinez and Monperrus 2016], jKali [Martinez and Monperrus 2016], jMutRepair [Martinez and Monperrus

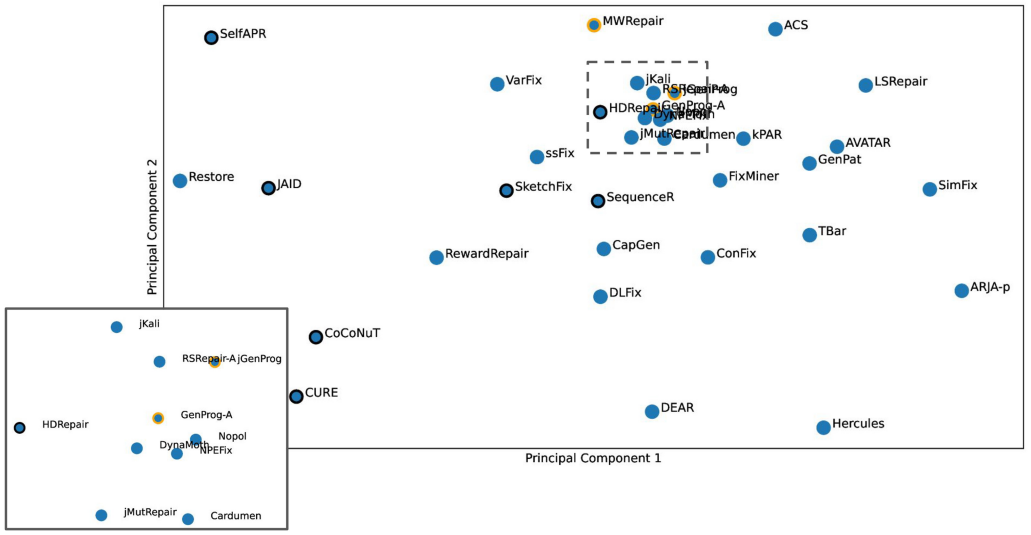

Fig. 8. APR Algorithms Repair Different Defects. In this Principal component analysis (PCA) embedding of Java APR algorithms, programs that correctly repair similar subsets of the Defects4J benchmark appear close to one another. Dense center region shown in lower left (inset).

2016], kPAR [Liu et al. 2019a], LSRepair [Liu et al. 2018], Nopol [Xuan et al. 2017], NPEFix [Cornu et al. 2015], Restore [Xu et al. 2020], RewardRepair [Ye et al. 2022a], SelfAPR [Ye et al. 2022b], SequenceR [Chen et al. 2019], SimFix [Jiang et al. 2018], SketchFix [Hua et al. 2018], ssFix [Xin and Reiss 2017], TBar [Liu et al. 2019c], and VarFix [Wong et al. 2021]. We include each of these for completeness but note that there are many commonalities and overlaps among the algorithms encoded by the different tools. As discussed earlier, several of the tools also make stronger assumptions about how the repair problem is presented than others (requiring method or exact line fault localization as input), which complicates comparison. Where possible, we compare to published data generated from experiments that make assumptions roughly similar to MWRepair, specifically those that do not require *perfect* knowledge of where the human repair was applied. For some algorithms (e.g., CURE), these data are not available since the authors report only results from experiments that do supply perfect fault localization information. With those caveats in mind, the total number of defects with repairs reported across all of the tools is 140, and the ensemble success rate ("Combined Algs." in Table 3) is 39.2%. Each tool repairs a different subset of these 140 defects, although some have more in common with one another than others.

This large set of program repair tools encompasses different designs and overall performance. Many of the older tools (e.g., Nopol, GenProg, Kali, and RSRepair) repair a similar set of defects correctly. Some of the next generation of algorithms are limited in scope: Cardumen, for example, finds plausible patches for 12 defects but correctly repairs only 3 of them. Other recent algorithms are more generic (and thus potentially more broadly applicable) and have higher success rates. For example, TBar finds plausible patches for 72 defects and correctly repairs 24 of them. Recent deep learning approaches are currently popular, with CURE topping the list with 53 correct repairs. Some other recent methods, like Hercules (50 correct repairs), perform similarly, but they do not rely on deep learning to synthesize code modifications.

To better understand how the different tools compare to one another, we developed a spatial embedding of the data from Table 3, which is shown in Figure 8. Pairwise distance was computed between each algorithm by computing the number of defects that either algorithm correctly

repairs but the other does not. Cases where both algorithms repair or fail to repair a defect do not contribute to this distance, so it is symmetric. These distances form a stress matrix, which is embedded in two dimensions for visualization. Similar embeddings have been used to analyze the defects in a variety of benchmarks, which are then highlighted based on whether a particular tool repairs a particular defect [Aleti and Martinez 2021]. Instead, we use the pattern of which tools repair which defects to embed the tools themselves, directly.

Tools that are nearby in the embedding have found repairs for mostly the same defects, whereas those that are farther apart repaired different ones. GenProg clusters mostly with other older APR algorithms, which repair relatively few defects and mostly the same ones. MWRepair is distant from algorithms like TBar and DLFix because it patches defects that other algorithms miss.

Recent algorithms such as Hercules [Saha et al. 2019], Restore [Xu et al. 2022], and ARJA-p [Yuan and Banzhaf 2020b] also explore new ideas for repair (appearing on the perimeter of the figure). Although all three algorithms use deep learning models, we note that DEAR [Li et al. 2022] embeds near Hercules (another method that targets multi-hunk bugs); RewardRepair [Ye et al. 2022a], which uses neural machine translation, embeds near CoCoNuT and CURE (which use similar architectural ideas); and SelfAPR [Ye et al. 2022b], which uses a novel training procedure, appears on the frontier. Algorithms that require additional fault localization information are shown with a black border. MWRepair contributes different repairs from other algorithms with similar success (e.g., SequenceR, AVATAR, and CapGen), and it complements both these and algorithms that are more performant in terms of the absolute number of repairs generated (e.g., CURE and Hercules).

6.7 Summary of Results

We evaluated MWRepair using GenProg's mutation operators and the search space they induce. Because it searches the space differently, MWRepair boosts GenProg's ability to find both patches and repairs. 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. These are composed of multiple edits, even when minimized to reduce potential side effects and improve readability. MWRepair complements other modern algorithms by repairing a different subset of the scenarios in Defects4J.

7 DISCUSSION

MWRepair differs from most of the algorithms used in APR along a few key dimensions. We discuss the potential use of the online learning component of MWRepair (MWU) in other domains, what it would take to generalize MWRepair to other algorithms than GenProg and why this might be desirable, and discuss MWRepair's exploration of the APR search space. Finally, we note limitations, with particular emphasis on those inherited by the field through flaws in common benchmarks.

7.1 Applicability of MWU to Other Domains

The unimodal distribution of patch probability in the search space (Section 4) arose in every program/defect pair that we investigated. We conjecture that our model captures this regularity—that is, the probability of finding a patch increases approximately linearly with the number of atomic mutations considered, and the probability of failure increases asymptotically faster, but has a smaller coefficient. As a result, there is a benefit to combining multiple atomic mutations to enhance search effectiveness, and the benefit is reflected in the unimodal curve, suggesting that there is an optimal number of mutations to combine. MWU proved useful for estimating the optimum number of mutations to combine, using information obtained as the search proceeds. We suspect that similar optima might exist in other mutation-based search problems than the APR domain we studied and that other evolutionary and optimization algorithms in these domains could benefit from incorporating online learning in a similar way.

7.2 Generalization of MWRepair to Other APR Algorithms

We implemented MWRepair using GenProg's mutation operators because they are simple and generic, and many subsequent tools use variations of these operators. Thus, we expect that our results will generalize to these settings. Reference implementations of GenProg are available in both C [Le Goues et al. 2012] and Java [Martinez and Monperrus 2016], and GenProg's patch notation facilitated our evaluation of compound mutations.

However, MWRepair makes no assumptions about mutation operators or the patching framework. MWRepair is algorithm agnostic by design and can be applied to algorithms that use other mutation operators or that target other languages and benchmarks. Even algorithms that by default consider only single mutations are in principle compatible with MWRepair, which would dynamically manage how many neutral mutations to consider simultaneously. So, we expect that MWRepair can help many algorithms search more broadly, including those that use more sophisticated heuristic operators.

We argue for this generality by examining the defects in the Defects4J benchmark [Just et al. 2014], where repairs should be in the search space induced by existing algorithms' mutation operators. For example, Time 3 requires that simple preconditions be checked in different parts of the program; this operation is in the search space for Nopol [Xuan et al. 2017] and ACS [Xiong et al. 2017]. Lang 4 requires that a function call be modified in unrelated contexts, which Hercules [Saha et al. 2019] or ELIXIR [Saha et al. 2017] could handle if augmented. Additionally, Chart 25 consists of four changes, each of which is a templated operator used by TBar [Liu et al. 2019b]. None of these defects has been repaired by any algorithm, to date, because the transformations required are more complex than those expressed by single mutations. As this demonstrates, making progress in searching for multi-edit repairs is an important problem, and MWRepair takes a step in this direction.

7.3 Search Space Exploration

It is tempting to assume that when existing APR algorithms fail to patch a defect that it is because the repair requires complex code transformations. However, this is not always the case, and sometimes these failures occur because the search space is large and the successful configurations are rare. The `units` utility is an instructive example. In the case of GenProg, a repair is found on 7% of the runs [Le Goues et al. 2012]; MWRepair repaired units in 100% of our trials. Further, some defects that GenProg has never repaired are in its search space: MWRepair finds repairs for them by sampling with replacement to search for epistasis.

MWRepair explores the search space differently from other algorithms because of its at-times aggressive combination of mutations, which is enabled by the use of neutral mutations. This promotes the discovery of multi-edit patches, which are formed by epistatic interaction between individually neutral mutations. Multi-edit patches are ruled out by the many algorithms that only ever consider single mutations in isolation and are unlikely to be found by algorithms that, although they use multiple mutations, are conservative in their combination. For example, none of the multi-edit defects we repair here (some for the first time) were repaired by GenProg. Additionally, finding multi-edit patches is crucial for improving the performance of APR algorithms; as mentioned earlier, 71.77% of the reference patches in the Defects4J benchmark contain edits in multiple locations [Sobreira et al. 2018].

7.4 Limitations

The cost of evaluating a sizable test suite on a complex software project can be high: even on modern hardware, running an extensive test suite can take on the order of hours. Generating a large pool of neutral mutations for such a project may seem expensive. But this is a cost paid by any

algorithm that uses test suites as acceptance criteria for patches. Additionally, in most cases, these algorithms test many mutated variant programs before finding a suitable patch. These algorithms have to pay that cost online; we partially address the expense by refactoring it out of the repair algorithm itself so that it can be amortized by precomputation. This amortization is particularly important for real-time scenarios [Highnam et al. 2016].

As software undergoes continuing evolution, once-neutral mutations may be invalidated. This decay is not immediate or absolute, as we measured in Section 6.5. Changes to large projects often have limited impact on the test suite. Indeed, this is an assumption underlying change-aware impact analysis [Ren et al. 2004]. We can amortize the cost of generating and updating a set of neutral mutations and employ them when a defect emerges.

7.5 Benchmark Quality

Algorithms for APR tend to be evaluated on a small set of bugs in widely used benchmarks, so their design may be overfit to these [Liu et al. 2021; Wang et al. 2020]. MWRepair is evaluated against common benchmarks, and so, like the algorithms we compare to, it inherits the limitations and vulnerabilities of these benchmarks. *Defect isolation*, the problem of separating the code that introduces or repairs a defect from other code, is a known problem in this context [Herbold et al. 2020]. This problem can be serious: Defects4J maintainers manually separated what they considered to be code that was related to the defect from code that was not. This approach means that any code that was not deemed to directly relevant to repairing the defect is included in the reference defect. This has benefits—for example, by reducing large change sets that include unrelated program refactoring to a smaller set. But the approach is error prone. Consider, for example, the Math 24 defect, which requires changes to multiple code locations to ensure that an optimizer can handle both maximization and minimization scenarios. The human programmer wrote an auxiliary function from scratch to handle these cases and then inserted invocations of it in two locations. The Defects4J maintainers committed the bespoke function the human developer created as part of the defective program; only the two invocations are segmented to the repaired version. A recent paper [Saha et al. 2019] reports a repair to this defect—and highlights it as the motivating example—because the code the repair algorithm synthesized is an exact match to the reference human repair. But the repair algorithm did not write the function that does most of the work of repairing the defect; instead, it uses template matching to synthesize invocations of it. This is an extreme example, but others are common: it is frequently the case that human-written comments about the defect repair are not correctly segmented and are included in the defective version. As algorithms are developed that use language models and program context, the importance of this increases. Recent work has sought to apply algorithmic techniques to better and more systematically perform this isolation [Yang et al. 2021]. Defect isolation itself cannot easily be solved by the field, except by the development and adoption of new benchmarks that are explicitly constructed with this in mind. However, the more general concern of overfitting to the types of defect in a dataset can be partially addressed by evaluating against more than one benchmark, as some large-scale studies [Durieux et al. 2019] and replication studies [Guizzo et al. 2021] have recently reported. This is one potential direction for future work.

8 FUTURE WORK

8.1 Efficiency Considerations

An underexplored metric for Java program repair is the efficiency with which patches are generated [Liu et al. 2020]. Most studies instead focus on the number or quality of patches. MWRepair uses MWU to guarantee that it explores in a way that minimizes regret. When a patch proves

elusive, MWRepair learns the richest part of the search space. Because it pools mutations to test them, it has an advantage over brute-force algorithms. MWU's performance depends on the learning parameter η and the number of options. In this work, we used $\eta = 0.05$ and a maximum of 300 mutations for C programs (128 mutations for Java programs). A detailed characterization of the relationship between MWU parameters and APR performance remains an open problem.

To apply this approach efficiently, we precompute a set of neutral mutations for subsequent use in online repair scenarios. This precomputation is trivial for algorithms that do not require semantic information about the defect. But for algorithms like Angelix [Mechtaev et al. 2016] that do require this type of information, this screening could instead be performed on demand while the repair algorithm runs. This would likely affect the runtime of such an approach, but it does not prevent its application.

8.2 Modeling the Search Space

Section 4.3 presents a model that explains the unimodal distribution of patches (see Table 1) using a single functional form. But we cannot be certain that the model applies universally without additional experimentation. Additionally, if we had a principled way to predict the optimum's location ahead of time, we would not need to integrate online learning: we could instead target the optimal region from the start.

Although we tested several structural metrics for programs, including the number of test cases and the program's size, we did not find any that correlate with the location of the optimum. However, it is possible that other metrics, such as code complexity or revision histories, may be predictive.

8.3 Other Search Space Formulations

In this article, we study one of several possible APR search spaces. Our space builds on the set of all single neutral mutations that apply to the original program. Most existing algorithms use this space as well. An alternative approach might iteratively apply mutations to already-mutated code.

We do not consider retaining deleterious mutations in this work. But prior work has shown that $\approx 5\%$ of patches include deleterious mutations [Renzullo et al. 2018]. These contribute to the diversity of solutions. Future studies may relax the neutrality constraint we impose in this work to trade off time for repair diversity.

9 CONCLUSION

Evolutionary algorithms face the general problem of balancing exploration and exploitation when refining existing solutions. The field of APR has advanced rapidly, and a significant body of recent work has investigated new and improved mutation operators and code transformation techniques. Most of this work is conservative in how it applies transformations, and many published APR algorithms repair a similar subset of the defects presented in common benchmarks. It is also typical to consider the computationally expensive task of generating and evaluating mutations as part of the inner loop of the search process.

This article investigated how to more effectively and efficiently search the space created by a set of arbitrary mutation operators. We refactored the process of mutation generation and screening to a separate phase, which can be precomputed and amortized. We then applied a search strategy using dynamically learned step sizes (number of mutations to combine) based on our analysis of the search space. The resulting algorithm leveraged the relative strengths of online learning and evolutionary computation. MWRepair is a meta-algorithm in the sense that it can be instantiated with the mutation operators of any search algorithm. We evaluated MWRepair, instantiated with the GenProg mutation operators, on the ManyBugs and Defects4J benchmarks and found more

repairs—including multi-edit repairs that GenProg missed in its search space—more consistently. Where jGenProg found 6 correct repairs, MWRepair found 16. Three of these required multiple edits, and jGenProg repaired none of those, although the answers exist in its search space. These results show the promise of exploring search spaces more aggressively, both for APR and more speculatively for other problem domains.

APPENDIX

A MWREPAIR EVALUATION SUPPLEMENT: MANYBUGS

We selected C defects from the popular ManyBugs benchmark that highlight differences (i.e., defects that not all algorithms patch). For each ManyBugs experiment, we ran MWRepair 100 times with a time limit of 1,000 test suite evaluations and a learning rate of $\eta = 0.05$. We report the fraction of evaluations in which MWRepair patches the defect, as well as the average number of test suite evaluations. We used BugZoo [Timperley et al. 2018] to orchestrate evaluation.

The algorithms targeting the ManyBugs benchmark are GenProg [Le Goues et al. 2012], TrpAutoRepair [Qi et al. 2013], Kali [Qi et al. 2015], **Staged Program Repair (SPR)** [Long and Rinard 2015b], Prophet [Long and Rinard 2015a], Angelix [Mechtaev et al. 2016], F1X [Mechtaev et al. 2018], SOSRepair [Afzal et al. 2021], and Novelty [Villanueva et al. 2020].

A.1 Can We Find the Optimum with Online Learning?

MWRepair guarantees a bounded estimate, at worst 10% from the empirical optimum. We list the location of the optimum for each program we evaluated in Table 1. For each of these programs, the probability of discovering patches at the optimum is high. For example, patches are 38.5 times more common at the optimum for gzip-2009-08-16 than they are at a mutation distance of 1.

We compare MWRepair's cost to GenProg's cost, measured in test suite evaluations. For any algorithm that explores one random mutation at a time, the cost is a simple random variable. In expectation, this is equal to the reciprocal of the single-mutation patch rate. For example, consider a case where there are 1,000 mutations in the search space and 4 of them patch the defect. We expect the algorithm to test 250 mutations by the time it identifies its first patch, noting that the time taken by any actual execution is random. The algorithm could, in principle, get lucky and terminate at the first iteration. Or, it could get unlucky and need more than 800 failed attempts before a success.

The analysis for MWRepair must include that it can test combinations of mutations. That analysis can decompose the *regret* from the cost. As first introduced in Section 3.4, we define regret as the gap between the choices MWRepair would make and the best it could have. The underlying distribution of patches determines the best choice. Regret = $\sum_{t=1}^{T} m^{(t)} \cdot p^{(t)}$. Here, $m^{(t)}$ is the relative cost of each option: the gap between it and the best option in hindsight. Additionally, $p^{(t)}$ is the probability distribution over each option at each timestep.

Applying this analysis to the concrete case of gzip-2009-08-16 shows its utility. GenProg tests $1/0.0027 \approx 370$ mutations in expectation to identify a patch. MWRepair tests 71. MWRepair's cost is lower because it searches where patches are more frequent: the regret is only ≈ 4.24, showing that it pinpoints the optimum accurately with online learning.

A.2 ManyBugs

Table 5 compares the success of MWRepair, GenProg, and eight other repair algorithms. For the sample of programs we evaluated, MWRepair patches more defects than any other algorithm.

Table 6 compares the success and cost of MWRepair directly to the success and cost of GenProg. MWRepair patches 3 out of 15 defects in this dataset that GenProg has not patched. MWRepair's

Table 5. Comparison Across Search-Based Algorithms

Program	MWRepair	GenProg	TrpAutoRepair	Kali	SPR	Prophet	Angelix	F1X	SOSRepair	Novelty
gzip-2009-08-16	✓	✓	✓	✓	✓	✓		✓		
gzip-2009-09-26	✓				✓	✓	✓	✓	✓	✓
libtiff-2005-12-14	✓	✓	✓				✓	✓		
libtiff-2005-12-21	✓	✓	✓	✓	✓	✓	✓		✓	
libtiff-2006-02-27	✓	✓	✓	✓			✓			
libtiff-2007-11-02	✓	✓	✓	✓			✓	✓	✓	
libtiff-2007-11-23	✓	✓	✓					✓		✓
libtiff-2009-08-28	✓	✓	✓				✓	✓	✓	✓
libtiff-2010-12-13	✓	✓	✓				✓	✓		✓
lighttpd-1806-1807	✓	✓	✓	✓	✓			✓		✓
python-69223-69224	✓				✓	✓		✓	✓	
python-70098-70101	✓			✓	✓	✓				

MWRepair is compared to published results for other algorithms: GenProg [Le Goues et al. 2012], TrpAutoRepair [Qi et al. 2013], Kali [Qi et al. 2015], SPR [Long and Rinard 2015b], Prophet [Long and Rinard 2015a], Angelix [Mechtaev et al. 2016], F1X [Mechtaev et al. 2018], SOSRepair [Afzal et al. 2021], and Novelty [Villanueva et al. 2020]. For each APR algorithm and defect, a check indicates that the algorithm patched the defect at least once.

Table 6. Defect Patch Efficiency

Defect	MWRepair Consistency	GenProg Consistency	MWRepair Cost	GenProg Cost
gzip-2009-08-16	**100%**	30%	**35.64**	130.7
gzip-2009-09-26	**100%**	0%	**340.45**	∞
python-69223-69224	**27%**	0%	**101.1**	∞
python-70098-70101	**3%**	0%	**132.0**	∞
libtiff-2005-12-14	**100%**	90%	**4.0**	20.8
libtiff-2005-12-21	100%	100%	22.41	20.8
libtiff-2006-02-27	100%	100%	**4.33**	20.8
libtiff-2007-11-02	**100%**	80%	**3.05**	20.8
libtiff-2007-11-23	**100%**	80%	**9.80**	20.8
libtiff-2009-08-28	100%	100%	**2.89**	20.8
libtiff-2010-12-13	100%	100%	**11.08**	20.8
lighttpd-1806-1807	**64%**	50%	251.2	44.1
ccrypt	100%	100%	53.1	32.3
look	**100%**	99%	**1.6**	20.1
units	**100%**	7%	83.13	61.7

We ran MWRepair 100 times on each defect. Program names refer to ManyBugs defect IDs. We compare two key factors: patch consistency and patch cost. Patch consistency is a success percentage within a fixed budget. Patch cost is the average number of fitness evaluations required to patch the defect. Data for GenProg is taken from previously published papers [Le Goues et al. 2012, 2015].

patch rate ranges from 3% to 100%, but it is always greater than or equal to that of GenProg's. One motivation for MWRepair is to extend the reach of search-based algorithms, and the above results demonstrate this for our example C defects.

Table 6 reports search efficiency using two metrics: how expensive the searches are in terms of test suite evaluations and how many searches are successful. Returning to our earlier example, `units` is a small program with a defect that GenProg patches only 7% of the time. MWRepair's average cost on this defect was 90.2 test suite evaluations, which is higher than GenProg's 61.7.

However, it found a patch in 100% of our trials, where GenProg succeeded in only 7%. Thus, the expected cost for GenProg, per success, is $61.7 * 0.07 + 400 * 0.93 = 376.3$ since it uses its entire evaluation budget in failed cases. Because of this, reporting costs based only on successful runs can be misleading.

Composing many neutral mutations and evaluating them together has other advantages. Consider `libtiff-2005-12-14` as reported in Table 6, a defect in the libtiff image processing library that returns an incorrect success code. GenProg took 20.8 test case evaluations on average to patch it [Le Goues et al. 2015] and MWRepair took 4.0. MWRepair found 17 of these patches on the first program evaluation. The number of fitness evaluations MWRepair uses ranges from ≈ 1.6 to ≈ 340.45. In general, MWRepair's cost per success is about an order of magnitude lower than that of GenProg.

ACKNOWLEDGMENTS

The authors are grateful to ASU Research Computing, whose support enabled these experiments.

REFERENCES

A. Afzal, M. Motwani, K. T. Stolee, Y. Brun, and C. Le Goues. 2021. SOSRepair: Expressive semantic search for real-world program repair. *Transactions on Software Engineering* 47, 10 (2021), 2162–2181. https://doi.org/10.1109/TSE.2019.2944914

A. Aleti and M. Martinez. 2021. E-APR: Mapping the effectiveness of automated program repair techniques. *Empirical Software Engineering* 26, 5 (Sept. 2021), Article 99, 30 pages.

A. Arcuri. 2008. On the automation of fixing software bugs. In *Proceedings of the International Conference on Software Engineering*. ACM, New York, NY, 1003.

S. Arora, E. Hazan, and S. Kale. 2012. The multiplicative weights update method: A meta-algorithm and applications. *Theory of Computing* 8, 6 (2012), 121–164.

T. Bäck, D. B. Fogel, and Z. Michalewicz. 1997. *Handbook of Evolutionary Computation*. CRC Press, Boca Raton, FL.

W. Banzhaf and A. Leier. 2006. Evolution on neutral networks in genetic programming. In *Genetic Programming Theory and Practice III*, T. Yu, R. Riolo, and B. Worzel (Eds.). Vol. 9. Springer, Boston, MA, 207–221.

B. Baudry, S. Allier, and M. Monperrus. 2014. Tailored source code transformations to synthesize computationally diverse program variants. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, New York, NY, 149–159. https://doi.org/10.1145/2610384.2610415

B. Baudry, S. Allier, M. Rodriguez-Cancio, and M. Monperrus. 2015. Automatic software diversity in the light of test suites. *arXiv:1509.00144* (2015).

B. R. Bruce, J. Petke, M. Harman, and E. T. Barr. 2019. Approximate oracles and synergy in software energy search spaces. *Transactions on Software Engineering* 45, 11 (2019), 1150–1169. https://doi.org/10.1109/TSE.2018.2827066

P. Cashin, C. Martinez, W. Weimer, and S. Forrest. 2019. Understanding automatically-generated patches through symbolic invariant differences. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Los Alamitos, CA, 411–414. https://doi.org/10.1109/ASE.2019.00046

L. Chen, Y. Pei, and C. A. Furia. 2017. Contract-based program repair without the contracts. In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Los Alamitos, CA, 637–647.

Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus. 2019. SequenceR: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (Sept. 2019), 1–17. https://doi.org/10.1109/TSE.2019.2940179

B. Cornu, T. Durieux, L. Seinturier, and M. Monperrus. 2015. NPEFix: Automatic runtime repair of null pointer exceptions in Java. *arXiv:1512.07423* (2015).

E. F. De Souza, C. Le Goues, and C. G. Camilo-Junior. 2018. A novel fitness function for automated program repair based on source code checkpoints. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, New York, NY, 1443–1450.

T. Durieux, F. Madeiral, M. Martinez, and R. Abreu. 2019. Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, 302–313. https://doi.org/10.1145/3338906.3338911

T. Durieux and M. Martinez. 2016. DynaMoth: Dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*. ACM, New York, NY, 85–91. https://doi.org/10.1109/AST.2016.021

L. Gazzola, D. Micucci, and L. Mariani. 2019. Automatic software repair: A survey. *Transactions on Software Engineering* 45, 1 (2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. 2010. Has the bug really been fixed? In *Proceedings of the International Conference on Software Engineering*, Vol. 1. ACM, New York, NY, 55–64. https://doi.org/10.1145/1806799.1806812

G. Guizzo, A. Blot, J. Callan, J. Petke, and F. Sarro. 2021. Refining fitness functions for search-based automated program repair: A case study with ARJA and ARJA-e. In *Search-Based Software Engineering*, Vol. 12914. Springer International Publishing, Cham, Switzerland, 159–165.

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

S. Herbold, A. Trautsch, and B. Ledel. 2020. Large-scale manual validation of bugfixing changes. In *Proceedings of the 17th International Conference on Mining Software Repositories*. ACM, New York, NY, 611–614. https://doi.org/10.1145/3379597.3387504

K. Highnam, K. Angstadt, K. Leach, W. Weimer, A. Paulos, and P. Hurley. 2016. An uncrewed aerial vehicle attack scenario and trustworthy repair architecture. In *Proceedings of the Dependable Systems and Networks Workshop*. IEEE, Los Alamitos, CA, 222–225.

J. Hua, M. Zhang, K. Wang, and S. Khurshid. 2018. SketchFix: A tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, 888–891. https://doi.org/10.1145/3236024.3264600

Y. Jia and M. Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* 37, 5 (Sept. 2011), 649–678. https://doi.org/10.1109/TSE.2010.62

J. Jiang, L. Ren, Y. Xiong, and L. Zhang. 2019. Inferring program transformations from singular examples via big code. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Los Alamitos, CA, 255–266. https://doi.org/10.1109/ASE.2019.00033

J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'18)*. ACM, New York, NY, 298–309. https://doi.org/10.1145/3213846.3213871

N. Jiang, T. Lutellier, and L. Tan. 2021. CURE: Code-aware neural machine translation for automatic program repair. In *Proceedings of the International Conference on Software Engineering*. IEEE, Los Alamitos, CA, 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

R. Just, D. Jalali, and M. D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, New York, NY, 437–440. https://doi.org/10.1145/2610384.2628055

J. Kim and S. Kim. 2019. Automatic patch generation with context-based change application. *Empirical Software Engineering* 24, 6 (Dec. 2019), 4071–4106. https://doi.org/10.1007/s10664-019-09742-5

M. Kimura. 1968. Evolutionary rate at the molecular level. *Nature* 217, 5129 (Feb. 1968), 624–626. https://doi.org/10.1038/217624a0

M. Kimura. 1983. *The Neutral Theory of Molecular Evolution*. Cambridge University Press, Cambridge, England.

A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* 25, 3 (May 2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z

J. R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Complex Adaptive Systems, Vol. 1. MIT Press, Cambridge, MA.

W. B. Langdon, M. Harman, and Y. Jia. 2010. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software* 83, 12 (Dec. 2010), 2416–2430. https://doi.org/10.1016/j.jss.2010.07.027

W. B. Langdon, N. Veerapen, and G. Ochoa. 2017. Visualising the search landscape of the triangle program. In *Genetic Programming*, James McDermott, Mauro Castelli, Lukas Sekanina, Evert Haasdijk, and Pablo García-Sánchez (Eds.), Vol. 10196. Springer, 96–113. https://doi.org/10.1007/978-3-319-55696-3_7

X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu. 2019. On reliability of patch correctness assessment. In *Proceedings of the International Conference on Software Engineering*. IEEE, Los Alamitos, CA, 524–535. https://doi.org/10.1109/ICSE.2019.00064

X.-B. D. Le, D. Lo, and C. Le Goues. 2016. History driven program repair. In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, Los Alamitos, CA, 213–224. https://doi.org/10.1109/SANER.2016.76

C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. 2015. The ManyBugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering* 41, 12 (Dec. 2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan. 2012), 54–72. https://doi.org/10.1109/TSE.2011.104

C. Le Goues, M. Pradel, and A. Roychoudhury. 2019. Automated program repair. *Communications of the ACM* 62, 12 (Nov. 2019), 56–65. https://doi.org/10.1145/3318162

Y. Li, S. Wang, and T. N. Nguyen. 2020. DLFix: Context-based code transformation learning for automated program repair. In *Proceedings of the International Conference on Software Engineering*. ACM, New York, NY, 602–614. https://doi.org/10.1145/3377811.3380345

Y. Li, S. Wang, and T. N. Nguyen. 2022. DEAR: A novel deep learning-based approach for automated program repair. In *Proceedings of the International Conference on Software Engineering*. ACM, New York, NY, 511–523.

K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon. 2019a. You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems. In *Proceedings of the IEEE Conference on Software Testing, Validation, and Verification*. IEEE, Los Alamitos, CA, 102–113.

K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. 2019b. TBar: Revisiting template-based automated program repair. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, New York, NY, 31–42. https://doi.org/10.1145/3293882.3330577

K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. 2019c. AVATAR: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, Los Alamitos, CA, 456–467. https://doi.org/10.1109/SANER.2019.8667970

K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé. 2018. LSRepair: Live search of fix ingredients for automated program repair. In *Proceedings of the Asia-Pacific Software Engineering Conference*. IEEE, Los Alamitos, CA, 658–662. https://doi.org/10.1109/APSEC.2018.00085

K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software* 171 (Jan. 2021), 110817. https://doi.org/10.1016/j.jss.2020.110817

K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, K. Kim, P. Wu, J. Klein, X. Mao, and Y. Le Traon. 2020. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for Java programs. In *Proceedings of the International Conference on Software Engineering*. ACM, New York, NY, 615–627. https://doi.org/10.1145/3377811.3380338

F. Long and M. Rinard. 2015a. *Prophet: Automatic Patch Generation via Learning from Successful Patches*. Technical Report MIT-CSAIL-TR-2015-027. CSAIL, Massachusetts Institute of Technology, Cambridge, MA.

F. Long and M. Rinard. 2015b. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, New York, NY, 166–178. https://doi.org/10.1145/2786805.2786811

F. Long and M. Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the International Conference on Software Engineering*. ACM, New York, NY, 702–713. https://doi.org/10.1145/2884781.2884872

T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan. 2020. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, New York, NY, 101–114. https://doi.org/10.1145/3395363.3397369

D. W. Marquardt. 1963. An algorithm for least-squares estimation of nonlinear parameters. *Society for Industrial and Applied Mathematics* 11, 2 (June 1963), 431–441. https://doi.org/10.1137/0111030

M. Martinez and M. Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering* 20, 1 (Feb. 2015), 176–205. https://doi.org/10.1007/s10664-013-9282-8

M. Martinez and M. Monperrus. 2016. ASTOR: A program repair library for Java. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, New York, NY, 441–444. https://doi.org/10.1145/2931037.2948705

M. Martinez and M. Monperrus. 2018. Ultra-large repair search space with automatically mined templates: The Cardumen mode of Astor. In *Search Based Software Engineering*. Lecture Notes in Computer Science, Vol. 11036. Springer, 65–86.

S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury. 2018. Test-equivalence analysis for automatic patch generation. *ACM Transactions on Software Engineering and Methodology* 27, 4 (2018), 15. https://doi.org/10.1145/3241980

S. Mechtaev, J. Yi, and A. Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the International Conference on Software Engineering*. ACM, New York, NY, 691–701. https://doi.org/10.1145/2884781.2884807

M. Monperrus. 2018. Automatic software repair: A bibliography. *ACM Computing Surveys* 51, 1 (Jan. 2018), 1–24. https://doi.org/10.1145/3105906

S. Moon, Y. Kim, M. Kim, and S. Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*. IEEE, Los Alamitos, CA, 153–162. https://doi.org/10.1109/ICST.2014.28

J. Petke, B. Alexander, E. T. Barr, A. E. I. Brownlee, M. Wagner, and D. R. White. 2019. A survey of genetic improvement search spaces. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, New York, NY, 1715–1721. https://doi.org/10.1145/3319619.3326870

P. C. Phillips. 2008. Epistasis—The essential role of gene interactions in the structure and evolution of genetic systems. *Nature Reviews Genetics* 9, 11 (Nov. 2008), 855–867. https://doi.org/10.1038/nrg2452

L. S. Pinto, S. Sinha, and A. Orso. 2012. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, New York, NY, Article 33, 11 pages.

Y. Qi, X. Mao, and Y. Lei. 2013. Efficient automated program repair through fault-recorded testing prioritization. In *Proceedings of the International Conference on Software Maintenance*. IEEE, Los Alamitos, CA, 180–189. https://doi.org/10.1109/ICSM.2013.29

Z. Qi, F. Long, S. Achour, and M. Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, New York, NY, 24–36. https://doi.org/10.1145/2771783.2771791

X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. 2004. Chianti: A tool for change impact analysis of Java programs. In *ACM SIGPLAN Notices* 39, 10 (2004), 432–448. https://doi.org/10.1145/1028976.1029012

J. Renzullo, W. Weimer, and S. Forrest. 2021. Multiplicative weights algorithms for parallel automated software repair. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE, Los Alamitos, CA, 984–993. https://doi.org/10.1109/IPDPS49936.2021.00107

J. Renzullo, W. Weimer, M. Moses, and S. Forrest. 2018. Neutrality and epistasis in program space. In *Proceedings of the Genetic Improvement Workshop*. ACM, New York, NY, 1–8. https://doi.org/10.1145/3194810.3194812

R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. 2017. ELIXIR: Effective object-oriented program repair. In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Los Alamitos, CA, 648–659. https://doi.org/10.1109/ASE.2017.8115675

S. Saha, R. K. Saha, and M. R. Prasad. 2019. Harnessing evolution for multi-hunk program repair. In *Proceedings of the International Conference on Software Engineering*. IEEE, Los Alamitos, CA, 13–24. https://doi.org/10.1109/ICSE.2019.00020

E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest. 2014. Software mutational robustness. *Genetic Programming and Evolvable Machines* 15, 3 (Sept. 2014), 281–312. https://doi.org/10.1007/s10710-013-9195-8

V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, Los Alamitos, CA, 130–140. https://doi.org/10.1109/SANER.2018.8330203

C. S. Timperley, S. Stepney, and C. Le Goues. 2018. BugZoo: A platform for studying software bugs. In *Proceedings of the International Conference on Software Engineering*. ACM, New York, NY, 446–447. https://doi.org/10.1145/3183440.3195050

N. Veerapen, F. Daolio, and G. Ochoa. 2017. Modelling genetic improvement landscapes with local optima networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, New York, NY, 1543–1548. https://doi.org/10.1145/3067695.3082518

O. M. Villanueva, L. Trujillo, and D. E. Hernandez. 2020. Novelty search for automatic bug repair. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, New York, NY, 1021–1028. https://doi.org/10.1145/3377930.3389845

A. Wagner. 2007. *Robustness and Evolvability in Living Systems*. Princeton University Press, Princeton, NJ.

S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao. 2019. How different is it between machine-generated and developer-provided patches? An empirical study on the correct patches generated by automated program repair techniques. In *Proceedings of the 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, Los Alamitos, CA, 1–12. https://doi.org/10.1109/ESEM.2019.8870172

S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin. 2020. Automated patch correctness assessment: How far are we? In *Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, 968–980. https://doi.org/10.1145/3324884.3416590

W. Weimer, Z. P. Fry, and S. Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Los Alamitos, CA, 356–366. https://doi.org/10.1109/ASE.2013.6693094

W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the International Conference on Software Engineering*. IEEE, Los Alamitos, CA, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the International Conference on Software Engineering*. ACM, New York, NY, 1–11. https://doi.org/10.1145/3180155.3180233

C.-P. Wong, P. Santiesteban, C. Kästner, and C. Le Goues. 2021. VarFix: Balancing edit expressiveness and search effectiveness in automated program repair. In *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, 354–366. https://doi.org/10.1145/3468264.3468600

Q. Xin and S. P. Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Los Alamitos, CA, 660–670. https://doi.org/10.1109/ASE.2017.8115676

Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the International Conference on Software Engineering*. IEEE, Los Alamitos, CA, 416–426. https://doi.org/10.1109/ICSE.2017.45

T. Xu, L. Chen, Y. Pei, T. Zhang, M. Pan, and C. A. Furia. 2022. Restore: Retrospective fault localization enhancing automated program repair. *IEEE Transactions on Software Engineering* 48, 1 (2022), 309–326. https://doi.org/10.1109/TSE.2020.2987862

J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. 2017. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Transactions on Software Engineering* 43, 1 (Jan. 2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

D. Yang, Y. Lei, X. Mao, D. Lo, H. Xie, and M. Yan. 2021. Is the ground truth really accurate? Dataset purification for automated program repair. In *Proceedings of the 2021 IEEE International Conference onSoftware Analysis, Evolution, and Reengineering*. IEEE, Los Alamitos, CA, 96–107. https://doi.org/10.1109/SANER50967.2021.00018

H. Ye, M. Martinez, T. Durieux, and M. Monperrus. 2020. A comprehensive study of automatic program repair on the QuixBugs benchmark. *Journal of Systems and Software* 171 (Sept. 2020), Article 110825, 15 pages. https://doi.org/10.1016/j.jss.2020.110825

H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus. 2022b. SelfAPR: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 2022 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, 1–13.

H. Ye, M. Martinez, and M. Monperrus. 2021. Automated patch assessment for program repair at scale. *Empirical Software Engineering* 26, 2 (March 2021), Article 20, 38 pages. https://doi.org/10.1007/s10664-020-09920-w

H. Ye, M. Martinez, and M. Monperrus. 2022a. Neural program repair with execution-based backpropagation. In *Proceedings of the International Conference on Software Engineering*. ACM, New York, NY, 1506–1518.

Y. Yuan and W. Banzhaf. 2020a. ARJA: Automated repair of Java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering* 46, 10 (2020), 1040–1067. https://doi.org/10.1109/TSE.2018.2874648

Y. Yuan and W. Banzhaf. 2020b. Making better use of repair templates in automated program repair: A multi-objective approach. In *Evolution in Action: Past, Present and Future: A Festschrift in Honor of Erik d. Goodman*, Wolfgang Banzhaf, Betty H.C. Cheng, Kalyanmoy Deb, Kay E. Holekamp, Richard E. Lenski, Charles Ofria, Robert T. Pennock, William F. Punch, and Danielle J. Whittaker (Eds.). Springer, Cham, Switzerland, 385–407.

A. Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software Engineering Notes* 24, 6 (1999), 253–267. https://doi.org/10.1007/3-540-48166-4_16

Received 1 February 2022; revised 3 May 2023; accepted 12 May 2023