2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-4066-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPS49936.2021.00107

Multiplicative Weights Algorithms for Parallel Automated Software Repair

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

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

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

***Abstract*—Multiplicative Weights Update (MWU) algorithms are a form of online learning that is applied to multi-armed bandit problems. Such problems involve allocating a fixed number of trials among multiple options to maximize cumulative payoff. MWU is a popular and effective method for dynamically balancing the trade-off between exploring the value of new options and exploiting the information already gained. However, no clear strategy exists to help practitioners choose which of the several algorithmic designs within this family to deploy. In this paper, three variants of parallel MWU algorithms are considered: Two parallel variants that rely on global memory, and one variant that uses distributed memory. The three variants are first analyzed theoretically, and then their effectiveness is assessed empirically on the task of estimating distributions in the context of stochastic search for repairs to bugs in software. Earlier work on APR suffers from various inefficiencies, and the paper shows how to decompose the problem into two stages: one that is embarrassingly parallel and one that is amenable to MWU. We then model the cost of each MWU variant and derive the conditions under which it is likely to be preferred in practice. We find that all three MWU algorithms achieve accuracy above 90% but that there are significant differences in runtime and total cost. When 90% accuracy is sufficient and evaluating options is expensive, such as in our use case, we find that the algorithm that uses global memory and has high communication cost outperforms the other two. We analyze the reasons for this surprising result.**

I. Introduction

Many important problems are challenging because the structure of the solution space is poorly understood. An example is Automated Program Repair (APR), where stochastic search of the space of possible program edits is a popular approach [1], [2]. Online learning algorithms are often used in such settings because they can acquire and incorporate information about the solution space while they run, thus improving their performance over time, although they have not been previously applied to APR.

APR typically starts with a defective computer program and a test suite, where the defect is indicated by one or more tests that the program fails to pass. In search-based APR [1], random mutations (probes) are applied to the program, which is then compiled and executed on the test cases to determine if the mutation constitutes a repair. It is not known in advance which mutations (if any) will repair the defect, either individually or in combination with other mutations. In addition, testing the functionality of a large-scale software project can take minutes to hours; this step occurs in the inner loop and is the dominant cost.

Online learning algorithms must balance exploration of new information with exploitation of previously-learned information. Multi-armed bandit problems [3], [4] capture the essence of this trade-off, framing the problem as one of allocating scarce resources (samples or probes) to options of varying but unknown quality. Sampling to estimate the quality of each option can be expensive. For example, in APR each trial requires patching and compiling a program, running it on a test suite, and assessing the results. In addition, many trials may be required to estimate an option's value. Multi-armed bandit problems emerge in many domains, including economics [5], web design [6], and optimal search [7]. Multiplicative Weights Update (MWU) [8] is a meta-algorithm that operates on bandit problems and is optimal (maximizes cumulative gain) in the asymptotic case [9], [10]. The theoretical literature outlines many concrete realizations of MWU. However, there is little guidance about when each realization is most appropriate.

Some MWU realizations target bandit problems explicitly, e.g., stock balancing problems [11]. Others have emerged as models of natural behavior, e.g., social learning [12]. Still others focus on parallelizing the evaluation of a subset of options, e.g., in Internet advertising markets [13]. When faced with large problem instances that have expensive probes, it is desirable to use many compute devices, but we lack a careful characterization of the formal or observed trade-offs that MWU design choices instantiate.

In this paper, we assess three realizations of MWU in terms of their convergence rate and accuracy at selecting valuable options. Next, we recast search-based APR as a two-phase process, where the first phase is embarrassingly parallel and the second phase is a multi-armed bandit problem. We study the performance of the different MWU realizations empirically in the context of this reformulation.

Our approach to APR addresses the inherent inefficiency and conservatism of existing methods, which apply mutations one at a time, incurring the high cost of testing for each mutation. Further, some defects cannot be repaired by a single mutation, even though most APR algorithms consider very few mutations in combination [14], [15]. What remains unsolved is determining how to navigate the super-exponential space of possible combinations of mutations. In our formulation, the

978-1-6654-4066-0/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPS49936.2021.00107

number of mutations to try in combination are the arms of the bandit, and we use MWU to choose which arm to sample at each iteration of the search. Because most mutations lower fitness (causing the program to fail on more test cases than the original), there is a trade-off between sampling a small number of mutations at once (less efficient and may not be sufficient to repair the bug at all) and sampling a large number (more efficient but higher chance of one or more mutations breaking test cases). We know of no method to predict the number of mutations that optimizes this trade-off for a given program and defect, so we use MWU to explore the set of feasible options while the repair algorithm executes.

Because the cost of running APR on large programs is high, we first introduce the idea of precomputing *safe* mutations at the time software is deployed, making it trivial to parallelize an expensive component of the search and to reuse mutations for multiple bug repairs. Next, we consider three MWU realizations to estimate how many safe mutations should be tested simultaneously on a particular bug scenario, in terms of both efficiency and likelihood of finding a repair. We give theoretical results for the three MWU realizations and report empirical results on representative bug repair scenarios. Finally, we analyze and model the cost to highlight the trade-offs among the approaches. We find that the communication cost and cost of evaluating options determine which MWU algorithm is recommended.

In summary, the main contributions of the paper are:

- A *formal comparison of three MWU algorithms*. We consider memory overheads, communication costs, convergence times, and the minimum number of agents required.
- A *parallel, online learning algorithm for APR*. We recast APR as a bandit problem, explicitly accounting for multiple mutations and precomputation.
- An *empirical evaluation of three MWU algorithms*. We compare algorithm performance on a number of program repair scenarios and two generic distributions.
- *Explicit cost models for MWU* in a practical application. Our model incorporates the number of agents, the number of options, and the learning rate.

II. Multiplicative Weights Update

Multiplicative Weights Update (MWU) [8] is a meta-algorithm that operates on bandit problems. Candidate options are assigned initial weights that are used to select a subset of options to evaluate, and the weights are updated multiplicatively over time based on how the selected options performed. We first describe the standard MWU algorithm, consider two popular variations, and summarize differences between them.

A. Standard MWU

The standard MWU algorithm [8] (*Standard*), sometimes called the *weighted majority algorithm*, is summarized in Figure 1. It takes as input a set of options, $Opts$, that have an unknown benefit. The algorithm samples the cost, *m*, of a set of options: The cost (reward) is 1 if the sample is correct and 0 otherwise. The algorithm is typically limited to a maximum number of iterations T. MWU associates a weight, w, with each individual option, updating the weights on each iteration, seeking to optimize the cost function. The rate at which new evidence is incorporated into the weights, e.g., increasing the weights of options found to be useful, is controlled by the parameter η. Together with other algorithmic decisions, η influences how the algorithm balances exploration and exploitation. In Figure 1, the weights are initialized to one, and the main loop implements the iterative sampling and weight updating. Each of the n parallel threads is assigned an option to evaluate in the *Sample* step. Then each thread evaluates its assigned option in parallel in the *Update* step. When this evaluation is complete, the shared model weights are updated, which requires all threads to communicate.

Input: option set $Opts$, function returning the cost m of an option, iteration limit T, learning rate $\eta \leq 1/2$, and number of parallel threads n.
Output: per-option weight vector w

1: $w_i^{(1)} \leftarrow 1,\ 1 \leq i \leq |Opts|$ ▷ Initialize
2: **for** $t \leftarrow 1$ to T **do**
3: **for** $j \leftarrow 1$ to n **do** ▷ Sample
4: $decision_j \leftarrow i \ \propto w_i^{(t)}$.
5: **end for**
6: **for** $j \leftarrow 1$ to n **do** ▷ Update
7: Observe cost $m_j^{(t)}$ of $decision_j$
8: $w_j^{(t+1)} = w_j^{(t)} * (1 + \eta * m_j^{(t)})$
9: **end for**
10: **end for**

Fig. 1. Standard: The Multiplicative Weights Update Algorithm

Input: option set $Opts$, method returning the cost m of an option, iteration limit T, and parallel threads n.
Output: per-option weights w

1: $w_i^{(1)} \leftarrow 1,\ 1 \leq i \leq |Opts|$ ▷ Initialize
2: $\gamma = \sqrt{\frac{(|Opts|/n)\cdot\ln(|Opts|/n)}{T}}$
3: $\eta = \sqrt{\frac{(1-\gamma)\cdot n\cdot\ln(|Opts|/n)}{|Opts|\cdot T}}$
4: **for** $t \leftarrow 1$ to T **do**
5: $p^{(t)} = w^{(t)} / \sum_i w_i^{(t)}$ ▷ Sample
6: $p'^{(t)} = (1-\gamma)\cdot p^{(t)} + \frac{\gamma}{|Opts|}\cdot 1_{|Opts|}$
7: Decompose $s \cdot p'^{(t)} = \sum_S q_S \cdot 1_S$, such that $q_S > 0$, $\sum_S q_S = 1$, and $S \subseteq |Opts|$, $|S| = n$
8: Select slate $S \propto q_S$
9: $m_j'^{(t)} = m_j^{(t)} / (n \cdot p_j'^{(t)})$ if $j \in S$ else 0 ▷ Update
10: $w_j^{(t+1)} = w_j^{(t)} * (1 + \eta * m_j'^{(t)}),\ 1 \leq j \leq |Opts|$
11: **end for**

Fig. 2. Slate Multiplicative Weights Update Algorithm. 1_S is a vector of length $|Opts|$, set to 1 if $j \in S$ and 0 otherwise.

B. Slate and Distributed MWU Variations

Standard assumes full visibility of the quality of each option on each iteration. That is, it makes a global, centralized

Given option set $Opts$, a method returning the cost m of an option, an iteration limit T, a learning rate $\eta \leq 1/2$, a number of parallel threads (agents) pop_size, and attention parameters $0 \leq \alpha \leq \beta \leq 1$

1: **for** $i \leftarrow 1$ to $|Opts|$ **do**
2: **for** $j \leftarrow 1$ to $(pop\_size/|Opts|)$ **do**
3: $C_{i \cdot j+j} = i$ ▷ Initialize
4: **end for**
5: **end for**
6: **for** $t \leftarrow 1$ to T **do**
7: **for** $j \leftarrow 1$ to pop_size **do** ▷ Sample
8: **if** $random() < \mu$ **then**
9: $k \leftarrow random\_int(|Opts|)$
10: $O_j \leftarrow Opt_k$ ▷ Pick a random option.
11: **else**
12: $k \leftarrow random\_int(pop\_size)$
13: $O_j \leftarrow C_k$ ▷ Observe a random neighbor.
14: **end if**
15: **end for**
16: **for** $j \leftarrow 1$ to pop_size **do** ▷ Update
17: **if** $m^t_{(O_j)} == 1$ && $random() < \beta$ **then**
18: $C_j \leftarrow O_j$
19: **else if** $m^t_{(O_j)} == 0$ && $random() < \alpha$ **then**
20: $C_j \leftarrow O_j$
21: **end if**
22: **end for**
23: **end for**

Fig. 3. Distributed Multiplicative Weights Update Algorithm. α is the chance of adopting a failed option; β is the chance of adopting a successful option; C_j is the option chosen by individual j; O_j is the option observed by individual j; and μ is the probability of choosing an option at random.

decision based on all available information. This visibility requirement is relaxed in the slate selection variation [13] (*Slate*), developed for online advertising and shown in Figure 2. In Slate, a subset of options is selected at each iteration, and only the weights of the subset are updated. Slate is specialized for problems that require selecting and evaluating a fixed-size set of options (like advertisements on a web page). As with Standard, each of the n parallel threads is assigned an option to evaluate: this is the slate S selected in Figure 2, line 8. In the *Update* step, these options are evaluated in parallel, and then the model is updated, which requires a communication block before the next iteration begins.

A distributed, memoryless variant of MWU (*Distributed*) was developed to model social learning dynamics [12]. Distributed is shown in Figure 3. It does not require full communication. Instead, each agent (thread) samples either a random option to observe or selects a single *neighbor* to observe, as can be seen on lines 7-15. Each thread then evaluates its selected option in parallel. Then each thread decides whether to adopt the observed option (lines 16-22). Distributed is specialized for situations where its memoryless property is advantageous and requires less intense communication.

TABLE I
ASYMPTOTIC COMPLEXITY OF MWU ALGORITHMS

	Standard	Distributed	Slate
Communication Cost	$O(n)$	$O(\frac{\ln(n)}{\ln(\ln(n))})*$	$O(n)$
Memory Overhead	$O(k)$	$O(1)$	$O(k)$
Convergence Time	$O(\frac{\ln(k)}{\epsilon^2})$	$O(\frac{\ln(k)}{\delta^2})$	$O(\frac{\frac{k}{n}\cdot\ln(k)}{\epsilon^2})$
Minimum Agents	$O(n)$	$O(k^{(\frac{1}{\delta^2})})$	$O(n)$

C. Algorithmic Analysis

The three MWU variations we consider have different complexity properties. It is critical to understand these complexity properties because focusing only on the asymptotic convergence time does not reflect the trade-offs faced in practice. First, we establish these properties formally; we return to this issue in Section IV and evaluate them empirically.

These properties are summarized in Table I. Memory overhead is the per-node cost, communication cost is the expected congestion of the heaviest hit node, and convergence time is the number of update cycles required for the weights to converge. *k* is the number of options available, *n* is the number of nodes, $\delta = \ln(\beta/(1-\beta))$ where β controls the attention paid to the most recent observation, and ϵ is the error tolerance. Starred bounds hold with probability at least $1 - \frac{1}{n}$.

Although Table I summarizes the properties of the three MWU algorithms, it can be challenging to compare them directly using the existing literature. For example, convergence of Standard is presented in terms of algorithm iterations [8, Sec. 3.1], while the convergence of Slate is presented in terms of regret [13, Sec. 2]. While it is possible to solve for one in terms of the other, as we have done for clarity in Table I, the relationships may not be obvious to practitioners, especially if they are given in terms of parameters that can appear opaque, such as δ (which depends on β) or ϵ (which depends on η).

Communication. In this context, communication cost refers to congestion: the maximum number of agents that any one agent must communicate with. Congestion is the crux of communication overhead because it determines how long the system must spend synchronizing at each iteration. Distributed has much lower expected congestion than the other two, and this bound holds with high probability.

To see why, we observe that the worst-case communication congestion for Distributed is $O(n)$: all agents could select the same agent to observe, but the probability of this decreases as the population size increases. Each agent selects uniformly at random from the entire population to decide which other agent to treat as its neighbor, so it is possible to derive a high-probability bound on the worst congested node. This is a classic instance of the *balls into bins* capacity problem, where the number of agents choosing neighbors equals the number of potential neighbors. So, with high probability, the worst congested node is no worse than $O(\frac{\ln(n)}{\ln(\ln(n))})$ [16].

By contrast, for Slate a fixed-size slate (subset) of options is chosen at each iteration. This is achieved naively by projecting the current weight vector onto each subset of options and

choosing sets according to the value of the projection. This is prohibitively expensive, even for moderately-sized problem instances. For example, if we had 1000 options and wanted to choose what each of 16 threads should evaluate, the number of combinations is $\binom{1000}{16} = 4.2 \times 10^{34}$. However, because each weight vector used in MWU can be capped and normalized to fall strictly within the probability simplex whose vertices correspond to the slates, it is possible to decompose the weight vector into a convex combination of these vertices. This requires $O(k^2)$ time [17].

Memory. Slate and Standard require $O(k)$ memory, but Distributed is $O(1)$. This is because the popularity of each option encodes the weight vector implicitly, and agents observe random neighbors to access this information.

Convergence. Convergence time is an important property of any learning algorithm, and in MWU it is the number of iterations required for the weights to converge to values within the error tolerance. Slate converges more slowly than Standard because it selects a subset of options on each iteration. Distributed converges under similar asymptotics to Standard, but this requires fixing the value of δ, which exponentially increases the number of agents.

Agents. In Distributed, the minimum number of agents is higher than for the other two because the weight vector is implicitly stored in the popularity of each option in the population, which must be large enough to avoid premature decay of diversity.

D. MWU Summary

MWU is used in many application domains to iteratively learn the quality of various options when measurement is expensive. MWU is popular because of its convergence guarantees and low overhead costs. MWU variations, such as Distributed and Slate, have attractive properties such as lower memory overheads or less intense communication congestion, but the relative costs and benefits of the different approaches are challenging to assess in the context of particular applications. In the next section, we consider one such application, the automated search for repairs to software defects.

III. Stochastic Search for Program Repairs

APR tools aim to remedy defects in software without human involvement [1], [18]. A popular approach is search-based [19], in which candidate patches are generated through random mutation and recombination of existing code, and then validated against regression test suites to determine if they retain required functionality (pass required tests) and remedy the bug (pass a bug-inducing test). Most search-based methods are inspired by evolutionary computation [20]–[26], where mutations are edit operations to program statements and fitness is assessed by running test cases. Randomly mutating program code seems likely to introduce errors, and it is therefore surprising that $\approx 30\%$ of whole-statement mutations to programs in C [27] and Java [28] preserve required functionality. We refer to these as *safe* mutations and note that any mutation that constitutes a bug repair must also be safe. To be counted as a repair, it must pass all required tests and the bug-inducing test. We further note that all mutations, safe or otherwise, are restricted to lines of code that are executed by the regression test suite to avoid mutations applied to dead or untested code. Because $\approx 70\%$ of single mutations fail at least one required test, search-based APR methods are conservative, and even those that are capable of applying multiple mutations [20] typically do so only one at a time [14], [15]. A recent study of eleven such repair tools on five common Java program repair benchmarks found that between 53% and 90% of the defects in each benchmark were not fixed by any repair tool [29] . Thus, there is a need to improve the effectiveness of APR, both by expanding the scope of its search and by improving the efficiency to allow more extensive searches.

A. Recasting APR as an online statistical estimation problem

Current methods search only in a one- or two-edit distance from the original program [14], [15], [18], focusing on mutation. This limits the exploration of the search and its effectiveness at finding repairs (cf. [25]). In this paper, we focus on increasing the parallelism of the search and on combining safe mutations in a way that maximizes the chances of finding a repair. To do this, we (1) step out of the evolutionary computation paradigm by eliminating selection and recombination, (2) precompute a large sample of individually safe mutations which can be applied to multiple bug scenarios for a single program, and (3) use online learning to identify the best way to combine safe mutations to search for repairs. *The unknown, which must be determined online, is how many safe mutations to combine to balance the size of the step in the search space with the failure rate of each attempt.*

To implement this approach requires answering these questions: (1) How does the probability of finding a repair change with the number of mutations considered? (2) How likely are mutations to interact negatively? and (3) How do the answers to (1) and (2) change with different programs and bug scenarios? The answers to these three questions motivate our use of online learning and investigation of MWU realizations.

B. Number of Mutations

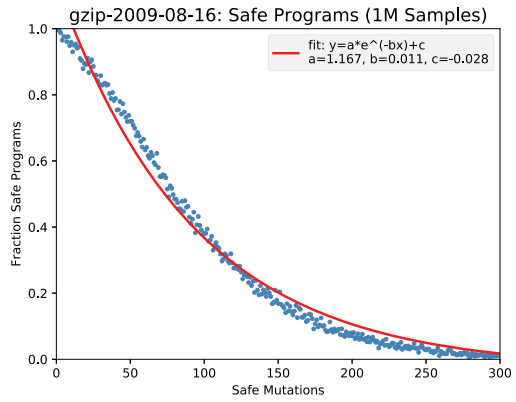
Any mutation that repairs a defect must pass all available test cases, i.e., be safe. This defines the search space. Assuming that mutations are generated randomly, that each mutation is tested independently, and that we have no prior information about which mutations are more likely to be beneficial, we can assume that any individual mutation has an equally likely chance of repairing a bug. The chance of finding a repair is thus expected to increase linearly with the number of independent mutations considered, which implies a linear relationship between safe mutations and repair probability.

Testing each mutation separately limits efficiency, but combinations of individually safe mutations can interact negatively and break the program. Figure 4a shows the strength of this effect, when x random safe mutations are applied to the program `gzip` [30]. The figure shows how the fraction of programs that pass the test suite changes as a function of

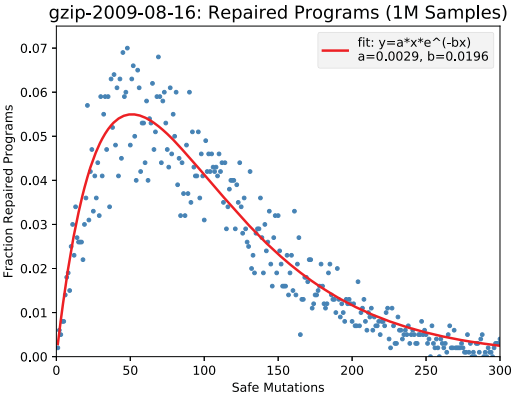
the number of mutations that have been applied. Each point depicted is the average of 1,000 independent random trials of x safe mutations. Although the curve descends quickly as x increases, indicating negative interactions, many mutations can be combined safely. Even when 80 safe mutations are applied together, on average, over 50% of the resulting programs retain their original functionality with respect to the test suite. In comparison, only two random, untested mutations (not guaranteed to be safe) can be applied before more than 50% of the resulting programs lose functionality. Although we show data for only one program, this pattern holds for all of the programs we have tested.

Next, we investigate how many mutations should be combined to maximize the chance of finding a repair. There are two factors: maximizing how many mutations we can test simultaneously (to reduce test suite execution costs) and minimizing the chance that multiple mutations interfere negatively (Fig. 4a). As we accumulate safe mutations, we expect the density of repaired programs to increase until the effect of negative interactions outweighs the benefit of testing more mutations together. Figure 4b confirms the prediction for `gzip`, demonstrating a unimodal distribution with the optimum occurring at 48 combined mutations. We have confirmed this unimodal pattern in thirteen other programs, with the optimum found anywhere from 11 to 271 mutations. Importantly, for each program/bug combination, *the optimal density of repairs occurs at a different place on the x-axis*. We can use MWU to adaptively bias samples of x (how many mutations to combine) towards regions of the space that are near the optimum.

In a bug repair scenario, the repair process typically terminates after the first repair is found, so the online search can not sample repair density directly. Instead, we use the density of safe mutations, which the search does sample, as a proxy.



(a)



(b)

Fig. 4. Effect of Mutation Accumulation on Program Behavior

C. Precomputing Safe Mutations

All search-based APR methods rely on generating safe mutations. To date, all such algorithms generate new mutations on demand, and the process of finding safe mutations is embedded in the inner loop of the search [1]. We propose a new approach, which precomputes a large pool of safe mutations, a one-time cost that is easily run in parallel and can be amortized over the cost of repairing multiple bugs in a given program. This technique is useful in general and it minimizes a fundamental problem which arises in translating our observations about the search space into an efficient APR algorithm.

A search process that combines a varying number of safe mutations and generates them on the fly encounters a performance issue. In algorithms that require synchronization blocks (e.g., for communication), all threads must wait for the slowest one to complete its work. This problem is exacerbated as the number of threads increases, because the highest cost is paid with increasing probability. The probability of the worst k values in a range of n values being selected in at least one of m trials is $1 - (\frac{n-k}{n})^m$. In the example of 64 threads choosing between 1 and 100 mutations to evaluate, the process would need to wait for a thread to identify 91 or more safe mutations (worst 10% of outcomes) with probability $1 - (\frac{90}{100})^{64} \approx 99.9\%$. *Thus, almost all iterations incur high cost, and the naive system operates at about half the efficiency of threads requiring no synchronization blocks.* An additional cost occurs when identical mutations are generated and evaluated repeatedly by different probes, which is common.

Precomputation avoids this bottleneck. If a pool of known safe mutations is available, then each thread can select and compose them independently, evaluating the resulting program with a single call to the test suite. That is, precomputing safe mutations linearizes this part of the online algorithm by removing the dependence on the maximum number of mutations any thread must evaluate.

Most deployed software has an associated regression test suite. New tests may be added over time as the program's source code evolves, and the safe mutation pool can be updated incrementally whenever this occurs. As defects are repaired, the failing test(s) that exposed the defect can be added to the test suite, the precomputed pool can be run on the new test(s),

and any that fail can be replaced with a new safe mutation. This process is embarrassingly parallel.

D. The MWRepair Algorithm

We combine these two insights (precomputing a pool of safe mutations and searching for repairs where density in the search space is highest) into the MWRepair algorithm (Fig. 5), which recasts APR as an online statistical estimation problem that is naturally parallel. The increase in efficiency mediated by effective online learning motivates our use of MWU.

In the precompute phase, MWRepair takes a program and regression test suite as input and generates a pool (labeled database in the figure) of safe mutations. Each parallel process applies a mutation to the original program and evaluates its test cases. The online phase uses MWU to bias the search towards the optimum (Fig. 4b) while iteratively testing combinations of mutations to see if they repair the bug. When a repair is found, the algorithm terminates. We have implemented MWRepair and confirmed that it finds bug repairs successfully in both Java and C (results summarized in Sec. IV-G).

IV. Empirical evaluation

Our empirical evaluation focuses on the performance of the three MWU algorithms in the context of MWRepair, reporting the number of iterations for MWU to converge and its accuracy. Section IV-A describes the datasets used in the evaluation; Section IV-B outlines the experimental design; Section IV-C reports performance in terms of number of iterations to convergence; Section IV-D reports accuracy of the estimates; Section IV-E develops a cost model for MWU algorithms based on our results; and Section IV-G summarizes MWRepair's ability to repair bugs compared to previous work.

A. Datasets

Each algorithm is evaluated on four distributions: two that are generic and two that are empirically derived from well-known APR benchmarks. The two generic distributions are `random` and `unimodal`. We selected `random` as a proxy for the class of distributions where the value of each option is not correlated with surrounding options. We selected `unimodal` for generality because we have strong evidence that most bug repair scenarios are unimodal (Section III-B). The `random` and `unimodal` datasets are composed of synthetic data. Each scenario in the `random` dataset contains a variable number of entries (2^8 to 2^{14}), each of which is independently and uniformly sampled from the unit interval. The larger the instance (number of entries), the harder it is for the algorithm to converge, and it is likelier that multiple options have similar values. The `unimodal` dataset is constructed similarly, except the distribution is defined by the form $a * x * e^{-bx} + c$, where a, b, and c are chosen independently and uniformly at random from the unit interval.

The remaining datasets are constructed from popular program repair benchmarks for C and Java. The C dataset contains four scenarios from the ManyBugs benchmark [30] and `units`, a small command-line utility from an older benchmark set [20]. We selected examples that range over different sizes, runtimes, and test suite complexity. The Java benchmark contains five scenarios from the Defects4J benchmark [32]. Each of the five Java scenarios have the same number of options, but vary in the distribution of values over them.

B. Experimental design

Each algorithm was executed on each dataset with 100 unique random seeds, and Table II and Table III report the mean and standard deviation of these experiments. All experiments share the same input datasets and are limited to 10,000 iterations. Algorithm parameters were selected to ensure maximum comparability: the probabilities of selecting a random choice (μ in Distributed, and γ in Slate) were set equal, at 0.05. The error threshold in Standard, ϵ, was also set to 0.05. This fixes all other parameters (e.g., the size of the population in Distributed, and the size of the subset in Slate), since the iteration limit and the option set are fixed inputs.

C. Convergence Time

Table II presents the average and standard deviation of update cycles until convergence for 6,000 experiments: 100 executions of each algorithm on each of twenty datasets. Convergence is defined by the probability of the highest weight option at each time step. For Standard and Slate, this was defined by a tolerance of 10^{-5} relative to the maximum possible. For Distributed, a threshold was set to 30% of the population choosing the same option. This is a less demanding threshold, but reflects the maximum achievable given the inherent noise of the finite-population approximation of the weight vector and the probability of choosing a random option.

For Standard, the number of iterations until convergence is closely related to the instance size. Compare, for instance, the results for `random` and `unimodal`. The performance of Standard is also consistent across all five Java datasets, which have instance size 100 but vary in their data entries. Distributed behaves similarly, although the exponential dependence of the population size on the scenario size led to two intractable computations. It neither dominates nor is dominated by Standard. For all five `random` scenarios, Distributed converges most quickly. Slate does not always converge within the allotted budget. It is always the most expensive algorithm in terms of number of iterations until convergence.

D. Accuracy

A key goal of MWU algorithms is convergence to an accurate estimate of which options minimize cost. Table III shows the absolute percent error between the best possible solution in hindsight and the converged solution of each algorithm. Standard deviation (over 100 replications) is shown in parentheses. For scenarios in which an algorithm did not converge within the allotted time (represented as $>$10000 in Table II), we show the option with the highest weight when the time limit is reached.

The mean accuracy of each algorithm is always at least 90%. In domains where a reasonable estimate is good enough, such

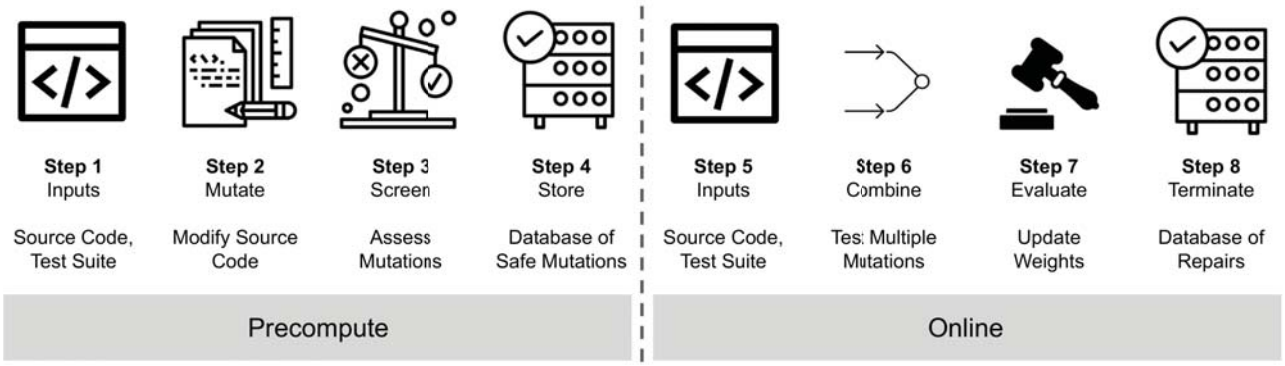


Fig. 5. Algorithm overview for MWRepair. Icons under Creative Commons license [31].

Input: M, a set of precomputed safe single mutations, P, a program with a defect, S, a set of test cases, $Opts$, a set of options, and a fitness function $f(P, S) \to \mathbb{N}$

```
MWU_Init()
for t ← 1 to T do
    probes^(t) ← MWU_Sample(probabilities^(t))
    for j ← 1 to n do                          ▷ Parallel Evaluation
        muts ← Random_Subset(M, probes_j^(t))
        P' ← Apply_Mutations(P, muts)
        if f(P', S) == |S| then
            Return P'                          ▷ Terminate Early
        else if f(P', S) ≥ f(P, S) then
            results_j^(t) ← 1
        else
            results_j^(t) ← 0
        end if
    end for
    probabilities^(t+1) ← MWU_Update(results^(t))
end for
Return null                                    ▷ Terminate
```

Fig. 6. MWRepair samples the number of mutations to apply, applies them, evaluates the resulting program, observes its cost (fitness), and uses that information to update the model for the next iteration. $MWU\_Init()$, $MWU\_Sample()$ and $MWU\_Update()$ are generic interfaces.

as our use case, any of the three would be acceptable, and the fastest running should be selected. For problem domains that require a high degree of accuracy, Standard is worse than the other two. If the availability of parallel compute resources is not a limiting factor, Distributed offers the best accuracy with the fewest update cycles. If resources are constrained, then there is a trade-off between Standard and Slate. Standard uses fewer update cycles, but has lower accuracy.

E. Cost Model

The asymptotic analysis discussed in Section II-C is useful as a general guide, but it abstracts away detail that is often relevant in practice. For example, the convergence complexity of Standard and Distributed is comparable, but the required settings to achieve this also imply that the communication cost and number of CPUs will asymptotically differ. We next combine the asymptotic analysis from Section II-C with our empirical observations from Section IV to model the real-world cost of the three MWU algorithms.

TABLE II
MEAN ALGORITHM ITERATIONS UNTIL CONVERGENCE. THE "SCENARIO" COLUMN IDENTIFIES THE INPUT DATASET, "SIZE" IS THE NUMBER OF OPTIONS IN EACH SCENARIO. THE VALUES IN THE "STANDARD", "DISTRIBUTED", AND "SLATE" COLUMNS ARE THE MEAN (AND STANDARD DEVIATION) OF 100 REPLICATED RUNS OF EACH ALGORITHM.

Scenario	Size	Standard	Distributed	Slate
random	64	20.8 (3.4)	9.9 (2.8)	956.5 (69.0)
random	256	59.5 (7.3)	21.6 (5.8)	1183.5 (108.1)
random	1024	206.8 (20.4)	37.8 (9.2)	1194.0 (98.4)
random	4096	732.7 (63.1)	91.0 (23.2)	1044.1 (69.4)
random	16384	2661.3 (207.8)	—	963.9 (49.3)
unimodal	64	20.8 (2.8)	16.4 (6.6)	>10000
unimodal	256	60.5 (5.7)	63.8 (22.6)	>10000
unimodal	1024	199.5 (18.3)	288.2 (119.4)	>10000
unimodal	4096	699.0 (52.8)	836.2 (228.4)	>10000
unimodal	16384	2527.7 (181.7)	—	>10000
units	1000	203.2 (19.6)	253.6 (94.8)	>10000
gzip-2009-08-16	500	101.2 (10.2)	81.8 (25.2)	>10000
gzip-2009-09-26	200	51.6 (6.3)	66.4 (24.9)	>10000
libtiff-2005-12-14	100	30.1 (3.4)	27.0 (9.6)	>10000
lighttpd-1806-1807	50	19.0 (2.9)	18.1 (7.3)	>10000
Chart26	100	29.1 (3.5)	29.0 (12.5)	>10000
Closure13	100	29.5 (3.5)	24.5 (8.9)	>10000
Closure22	100	29.7 (3.4)	27.4 (10.5)	>10000
Math8	100	28.8 (3.6)	28.5 (10.1)	>10000
Math80	100	30.1 (3.6)	36.3 (15.6)	>10000

F. CPU Cost

Table II shows how many iterations of the algorithm are required to converge on each dataset, but it omits information about the number of required CPUs required for each iteration, assuming that this is not a constraint. Table IV shows the cost in CPU-iterations required by each algorithm. The table highlights that, while Distributed often requires the fewest iterations to converge, it uses a large number of CPUs. Slate looked prohibitively expensive when considering only iteration cycles, but when viewed by CPU-iteration cost, it is sometimes more cost-efficient than Distributed. A similar argument

TABLE III
ACCURACY: PERCENT OF OPTIMAL VALUE ACHIEVED BY THE HIGHEST-WEIGHT OPTION AT TIME OF CONVERGENCE. THE COLUMNS LABELED "SCENARIO" AND "SIZE" IDENTIFY EACH BENCHMARK DATASET. THE VALUES IN THE "STANDARD", "DISTRIBUTED", AND "SLATE" COLUMNS ARE THE MEAN (AND STANDARD DEVIATION) OF 100 REPLICATED RUNS OF EACH ALGORITHM.

Scenario	Size	Standard	Distributed	Slate
random	64	93.4 (11.8)	95.7 (10.1)	100.0 (0.0)
random	256	92.2 (7.9)	98.6 (2.5)	99.2 (1.9)
random	1024	90.0 (7.0)	99.4 (0.9)	98.8 (1.8)
random	4096	91.7 (6.1)	99.9 (0.3)	97.8 (2.0)
random	16384	91.7 (5.5)	—	97.6 (2.3)
unimodal	64	94.5 (5.6)	96.3 (4.6)	99.0 (1.6)
unimodal	256	94.7 (4.6)	99.1 (0.8)	98.8 (1.2)
unimodal	1024	95.3 (4.4)	99.8 (0.2)	99.1 (1.0)
unimodal	4096	95.1 (4.3)	100.0 (0.1)	99.2 (0.7)
unimodal	16384	95.9 (4.0)	—	99.3 (0.7)
units	1000	94.9 (3.6)	99.6 (0.8)	98.1 (1.0)
gzip-2009-08-16	500	93.9 (4.3)	99.4 (0.8)	98.8 (1.2)
gzip-2009-09-26	200	92.6 (5.5)	98.2 (1.7)	98.1 (2.1)
libtiff-2005-12-14	100	92.1 (5.5)	97.1 (2.9)	98.9 (1.8)
lighttpd-1806-1807	50	93.8 (5.5)	97.0 (3.0)	98.8 (1.5)
Chart26	100	95.2 (5.7)	98.0 (1.9)	99.0 (1.3)
Closure13	100	93.7 (6.2)	98.1 (2.1)	98.9 (1.3)
Closure22	100	93.0 (5.8)	97.2 (2.7)	98.8 (1.8)
Math8	100	93.7 (6.5)	98.8 (1.0)	99.0 (0.9)
Math80	100	96.4 (4.9)	98.5 (1.2)	98.9 (0.9)

TABLE IV
COST: TOTAL CPU-ITERATIONS REQUIRED BY EACH ALGORITHM UNTIL CONVERGENCE. THE COLUMNS LABELED "SCENARIO" AND "SIZE" IDENTIFY EACH BENCHMARK DATASET. THE VALUES IN THE "STANDARD", "DISTRIBUTED", AND "SLATE" COLUMNS ARE THE MEAN OF 100 REPLICATED RUNS OF EACH ALGORITHM.

Scenario	Size	Standard	Distributed	Slate
random64	64	2080	1783	5701
random256	256	5947	22108	28350
random1024	1024	20679	219019	115431
random4096	4096	73268	2980905	402370
random16384	16384	266131	—	1507483
unimodal64	64	2078	2972	60377
unimodal256	256	6050	65352	241509
unimodal1024	1024	19948	1669375	966038
unimodal4096	4096	69896	27401758	3864151
unimodal16384	16384	252765	—	15456604
units	1000	20316	1426098	943396
gzip-2009-08-16	500	10124	193333	471698
gzip-2009-09-26	200	5156	49911	188679
libtiff-2005-12-14	100	3014	8541	94340
lighttpd-1806-1807	50	1900	2404	47170
Chart26	100	2906	9174	94340
Closure13	100	2947	7738	94340
Closure22	100	2970	8674	94340
Math8	100	2878	9022	94340
Math80	100	3012	11470	94340

reveals that although Slate required ≈ 1000 iterations on each of the `random` scenarios (implied by fixed γ setting the k/n ratio to a constant), the hidden cost of the scaling number of CPUs required to achieve that iteration bound is significant.

1) Weighted Asymptotic Model: Processes involving a large number of CPU cores must communicate, and this cost is not included in convergence asymptotics alone, although it is paid on each iteration. The size of the option set may be large enough that memory overhead is relevant. And in extremely CPU-constrained environments the minimum number of CPUs is important for determining which algorithm to prefer.

A decision model combining these features assigns weights to encode the relative importance of each feature. Then it is straightforward to predict which algorithm is preferred. As a simple example, consider the trade-off between communication cost and convergence time: cost = $\alpha \cdot \ communication\_cost \ + \ \beta \cdot \ convergence\_time$. Comparing Standard to Distributed, we then have: $cost_{Standard} = \alpha \cdot \ O(n) \ + \ \beta \cdot \ O(\ln(k)/\epsilon^2)$, $cost_{Distributed} = \alpha \cdot \ O(\frac{\ln(n)}{\ln(\ln(n))}) + \beta \cdot \ O(\ln(k)/\delta^2)$. Since neither ϵ nor δ depend asymptotically on the size of the option set or the number of CPUs, this analysis clearly favors Distributed. However, a model in which the number of CPUs used in each iteration is weighted (e.g., when parallel resources are constrained) will prefer Standard instead.

2) Concrete Recommendations: If the problem is characterized by low communication cost, and the computation required to evaluate an option is high (e.g., $\alpha << \beta$ above), then the benefit of Distributed on reducing communication cost is not enough to compensate for its higher CPU demand, and either Standard or Slate should be used. APR has exactly this feature: the information communicated by each process is a small packet containing the option index and the binary success or failure of the evaluation; however, evaluating a single option requires compiling and running a program on test cases. Conversely, if the application domain is severely memory-constrained and bandwidth-limited (as often occurs in embedded devices and sensor networks), then Distributed will be advantageous.

G. MWRepair and Other Baselines

We have shown how the three parallel MWU algorithms compare to one another, but one might ask how they compare to the baseline GenProg algorithm. In particular, it is important to establish whether the addition of online learning to a program repair process meaningfully improves effectiveness and efficiency. Our use of established defect scenarios allows us to compare MWRepair to earlier repair algorithms. First, in terms of expressive power, we note that MWRepair repairs all the C and Java defect scenarios, while previous algorithms such as GenProg, RSRepair, jGenProg, and AE repair 4/5, 3/5, 3/5 and 4/5 respectively [14], [20], [33], [34]. In single-threaded scenarios, APR algorithm costs are often measured in terms of fitness or test suite evaluations required to produce a repair. Including the overhead of the online learning process, MWRepair requires about half (52%) of the fitness evaluations of GenProg and jGenProg total. Because of the parallel nature of MWRepair, the total latency is $\approx 40\times$ less than those approaches. MWRepair uses the same mutation operators as all four of the algorithms mentioned above, so the search space it explores is the same. But, due to the nature of how it composes mutations, it explores that search space in a way

that improves both effectiveness (finding more repairs) and efficiency (paying lower costs to do so).

V. Related Work

A. Multiplicative Weights Update

Multiplicative Weights Update (MWU) is an efficient, versatile meta-algorithm for online learning. It has been discovered independently in multiple fields, for example as "fictitious play" in game theory [35] and as "winnow" [36] or "hedge" [10] in machine learning. It has also been an active area of theoretical development for online learning [6], [37], [38]. Our work applies three MWU algorithms [8], [12], [13] to generic scenarios and the concrete problem of program repair, and discusses conditions in which a practitioner should prefer one MWU algorithm over another.

B. Automated Program Repair

APR [18] is a subfield of software engineering which aims to repair defects in software without human involvement in the repair process. Our work augments existing APR approaches that use stochastic search in two ways.

First, we explicitly consider the value of multiple mutations. Previous work investigated this theoretically but has not proposed a concrete algorithm [27]. Although search-based APR was inspired by evolutionary computation, in practice most multi-edit repairs are redundant and can be minimized to one or two single-statement edits [20], which is illustrated by algorithms that focus explicitly on single-edit repairs [14], [15]. Other impediments to multi-edit repairs include the combinatorial explosion of the search space as the number of mutations increases [39] and the risk of breaking the program with each additional mutation [27], [28]. There is increasing interest in multi-edit repairs, but few approaches have been proposed. Hercules [25] mutates only similar or identical regions of code to reduce the combinatorial search space. By contrast, our use of MWU allows us to apply a large number of unrelated mutations, an approach that to our knowledge is unexplored in the program repair literature [1], [2].

Second, we consider parallelization directly in a principled way that leverages the structure of the problem. Previous algorithms parallelized the evaluation of a set of test cases on a single program [20], [23] or used naive random search that is parallel because no information is shared between threads [14], [33]. One exception is the Schulte-DiLorenzo distributed algorithm [40], which uses a distributed genetic algorithm to coordinate exploration, but the search space is explicitly partitioned among the processors. Finally, many semantics-based approaches cast the problem as one of constraint solving [14], [15], [41], but modern constraint solvers, like Z3 [42], are not well suited to parallel, much less distributed, operation.

VI. Discussion and Threats to Validity

In the evaluation, we selected parameters for each MWU algorithm to make as fair a comparison as possible. However, each algorithm has multiple interacting parameters (e.g., learning rate, iteration limit, and the chance of choosing an option randomly instead of obeying the weight distribution). There is also a large space of potential search characteristics: size, the relationship between the value of an option and the values of its neighbors, and how much option values vary across the distribution. Future research could characterize the interaction between parameters more carefully, facilitating extensions to other search problems.

Although our examination of MWU was conducted in the context of a concrete problem, our overall findings are generally applicable to other MWU settings (Section IV-E), and our analysis extends easily to other dataset distributions. Because our problem domain is unimodal, it is less important to find the exact best option than it is to bias the search towards high-density regions of the distribution. This is likely not the case for all problem domains.

An important aspect of APR is the fact that some bugs are easier to repair than others, and it is not well understood what determines the difficulty of finding repairs. The early termination condition of MWRepair, which returns the first candidate repair identified, means that for easy problems the overhead of online learning sometimes may not be required. For harder scenarios, where the cost of identifying a valid solution is often prohibitive, the choice of algorithm matters a great deal. We evaluated against 10 APR scenarios, five each from well-known C and Java benchmarks. In the future, the efficiency gains achieved with MWU on a larger set of benchmarks will be informative, especially for bugs that are currently not repairable by current methods. In preliminary studies (Sec. IV-G), we have already found examples in which MWRepair finds repairs for bugs that have not been repaired with other methods. However, a systematic study on a large corpus of bugs is required to confirm these results.

VII. Summary

Our formal analysis of MWU algorithms represents their asymptotic properties uniformly in terms of the same variables, easing the comparison between them. Our proposed MWRepair algorithm is naturally parallel and based on insights involving the precomputation of a large pool of safe mutations, the ability to compose multiple mutations to enhance exploration in a cost-effective manner, and the use of MWU to learn the optimal number of safe compositions. Our empirical cost model helps explain why an MWU algorithm that has global memory and high communication cost outperforms the other two on our problem domain. For this unimodal problem, our evaluation found that all three MWU algorithms achieve accuracy above 90%.

Acknowledgments

We gratefully acknowledge the partial support of NSF (CCF 1763674, CCF 1908633, IOS 2029696), DARPA (FA8750-19C-0003, N6600120C4020), AFRL (FA8750-19-1-0501), and the Santa Fe Institute.

REFERENCES

[1] L. Gazzola, D. Micucci, and L. Mariani, "Automatic Software Repair: A Survey," *Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[2] M. Monperrus, "Automatic Software Repair: A Bibliography," *ACM Computing Surveys*, vol. 51, no. 1, pp. 1–24, Jan. 2018.

[3] D. A. Berry and B. Fristedt, *Bandit Problems: Sequential Allocation of Experiments*, ser. Monographs on Statistics and Applied Probability. Springer, 1985, vol. 5.

[4] A. Mahajan and D. Teneketzis, "Multi-Armed Bandit Problems," in *Foundations and Applications of Sensor Management*, A. O. Hero, D. A. Castañón, D. Cochran, and K. Kastella, Eds. Boston, MA: Springer, 2008, pp. 121–151.

[5] H. E. Posen and D. A. Levinthal, "Chasing a Moving Target: Exploitation and Exploration in Dynamic Environments," *Management Science*, vol. 58, no. 3, pp. 587–601, Mar. 2012.

[6] M. Babaioff, Y. Sharma, and A. Slivkins, "Characterizing Truthful Multi-armed Bandit Mechanisms," *SIAM Journal on Computing*, vol. 43, no. 1, pp. 194–230, Jan. 2014.

[7] K. Glazebrook and A. Washburn, "Shoot-Look-Shoot: A Review and Extension," *Operations Research*, vol. 52, no. 3, pp. 454–463, Jun. 2004.

[8] S. Arora, E. Hazan, and S. Kale, "The Multiplicative Weights Update Method: A Meta-Algorithm and Applications," *Theory of Computing*, vol. 8, no. 6, pp. 121–164, 2012.

[9] A. Jafari, A. Greenwald, D. Gondek, and G. Ercal, "On No-Regret Learning, Fictitious Play, and Nash Equilibrium," in *International Conference on Machine Learning*, vol. 1, Williamstown, MA, USA, 2001, pp. 226–233.

[10] Y. Freund and R. E. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[11] D. P. Helmbold, R. E. Schapire, Y. Singer, and M. K. Warmuth, "On-Line Portfolio Selection Using Multiplicative Updates," *Mathematical Finance*, vol. 8, no. 4, pp. 325–347, Oct. 1998.

[12] L. E. Celis, P. M. Krafft, and N. K. Vishnoi, "A Distributed Learning Dynamics in Social Groups," in *Principles of Distributed Computing*. Washington, DC, USA: ACM, 2017, pp. 441–450.

[13] S. Kale, L. Reyzin, and R. E. Schapire, "Non-Stochastic Bandit Slate Problems," in *Neural Information Processing Systems*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, Eds. Vancouver, BC, Canada: Curran Associates, Inc., 2010, pp. 1054–1062.

[14] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results," in *Automated Software Engineering*, Palo Alto, California, Nov. 2013, pp. 356–366.

[15] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program Repair via Semantic Analysis," in *International Conference on Software Engineering*. San Francisco, CA, USA: IEEE, May 2013, pp. 772–781.

[16] G. H. Gonnet, "Expected Length of the Longest Probe Sequence in Hash Code Searching," *Journal of the ACM*, vol. 28, no. 2, pp. 289–304, Apr. 1981.

[17] M. K. Warmuth and D. Kuzmin, "Randomized Online PCA Algorithms with Regret Bounds that are Logarithmic in the Dimension," *Machine Learning Research*, vol. 9, no. 75, pp. 2287–2320, 2008.

[18] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated Program Repair," *Communications of the ACM*, vol. 62, no. 12, p. 10, 2019.

[19] M. Harman and B. F. Jones, "Search-based Software Engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

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

[21] Y. Yuan and W. Banzhaf, "ARJA: Automated Repair of Java Programs via Multi-objective Genetic Programming," *Transactions on Software Engineering*, vol. 46, no. 10, pp. 1040–1067, 2020.

[22] G. An, J. Kim, and S. Yoo, "Comparing Line and AST Granularity Level for Program Repair Using PyGGI," in *ICSE Genetic Improvement Workshop*. Gothenburg, Sweden: ACM, 2018, pp. 19–26.

[23] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic Patch Generation Learned from Human-written Patches," in *International Conference on Software Engineering*. San Francisco, California: IEEE, May 2013, pp. 802–811.

[24] R. B. Abdessalem, A. Panichella, S. Nejati, L. C. Briand, and T. Stifter, "Automated Repair of Feature Interaction Failures in Automated Driving Systems," in *International Symposium on Software Testing and Analysis*. Los Angeles, California: ACM, 2020, pp. 88–100.

[25] S. Saha, R. K. Saha, and M. R. Prasad, "Harnessing Evolution for Multi-Hunk Program Repair," in *International Conference on Software Engineering*, Montreal, Canada, May 2019, pp. 13–24.

[26] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting Template-based Automated Program Repair," in *International Symposium on Software Testing and Analysis*. Beijing, China: ACM, 2019, pp. 31–42.

[27] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest, "Software Mutational Robustness," *Genetic Programming and Evolvable Machines*, vol. 15, no. 3, pp. 281–312, Sep. 2014.

[28] N. Harrand, S. Allier, M. Rodriguez-Cancio, M. Monperrus, and B. Baudry, "A Journey among Java Neutral Program Variants," *Genetic Programming and Evolvable Machines*, vol. 20, no. 4, pp. 531–580, Jun. 2019.

[29] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical Review of Java Program Repair Tools: A Large-scale Experiment on 2,141 Bugs and 23,551 Repair Attempts," in *Foundations of Software Engineering*. Tallinn, Estonia: ACM, 2019, pp. 302–313.

[30] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs," *Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, Dec. 2015.

[31] "Noun Project: Free Icons & Stock Photos for Everything," https://thenounproject.com/.

[32] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," in *International Symposium on Software Testing and Analysis*. San Jose, CA, USA: ACM, 2014, pp. 437–440.

[33] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The Strength of Random Search on Automated Program Repair," in *International Conference on Software Engineering*. Hyderabad, India: ACM, 2014, pp. 254–265.

[34] M. Martinez and M. Monperrus, "ASTOR: A Program Repair Library for Java," in *International Symposium on Software Testing and Analysis*, Saarbrücken, Germany, 2016, pp. 441–444.

[35] G. W. Brown, "Iterative Solution of Games by Fictitious Play," *Activity Analysis of Production and Allocation*, vol. 13, no. 1, pp. 374–376, 1951.

[36] N. Littlestone, "Learning Quickly When Irrelevant Attributes Abound: A New Linear-threshold Algorithm," *Machine Learning*, vol. 2, pp. 285–318, 1988.

[37] R. Kleinberg, G. Piliouras, and E. Tardos, "Multiplicative Updates Outperform Generic No-regret Learning in Congestion Games: Extended Abstract," in *Symposium on Theory of Computing*. Bethesda, MD, USA: ACM, 2009, p. 533.

[38] G. Palaiopanos, I. Panageas, and G. Piliouras, "Multiplicative Weights Update with Constant Step-Size in Congestion Games: Convergence, Limit Cycles and Chaos," in *Neural Information Processing Systems*. Long Beach, CA, USA: Curran Associates, Inc., 2017, pp. 5874–5884.

[39] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis," in *International Conference on Software Engineering*. Austin, TX: ACM, 2016, pp. 691–701.

[40] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest, "Automated Repair of Binary and Assembly Programs for Cooperating Embedded Devices," in *Architectural Support for Programming Languages and Operating Systems*. Houston, Texas, USA: ACM, 2013, pp. 317–328.

[41] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the Cure Worse than the Disease? Overfitting in Automated Program Repair," in *Foundations of Software Engineering*. Bergamo, Italy: ACM, 2015, pp. 532–543.

[42] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Budapest, Hungary: Springer, 2008, pp. 337–340.