Enhancing Software Runtime with Reinforcement Learning-Driven Mutation Operator Selection in Genetic Improvement

Damien Bose, Carol Hanna, Justyna Petke
Dept. of Computer Science, University College London, London, United Kingdom

***Abstract*—Genetic Improvement employs heuristic search algorithms to explore the search space of program variants by modifying code using mutation operators. This research focuses on operators that delete, insert and replace source code statements. Traditionally, in GI an operator is chosen uniformly at random at each search iteration. This work leverages Reinforcement Learning to intelligently guide the selection of these operators specifically to improve program runtime.**

We propose to integrate RL into the operator selection process. Four Multi-Armed bandit RL algorithms (Epsilon Greedy, UCB, Probability Matching, and Policy Gradient) were integrated within a GI framework, and their efficacy and efficiency were benchmarked against the traditional GI operator selection approach. These RL-guided operator selection strategies have demonstrated empirical superiority over the traditional GI methods of randomly selecting a search operator, with UCB emerging as the top-performing RL algorithm. On average, the UCB-guided Hill Climbing search algorithm produced variants that compiled and passed all tests 44% of the time, while only 22% of the variants produced by the traditional uniform random selection strategies compiled and passed all tests.

***Index Terms*—reinforcement learning, genetic improvement**

I. Introduction

In today's digital era, software permeates every aspect of our lives, driving critical systems, powering consumer applications, and facilitating key business operations. Its omnipresence underscores the importance of its efficiency. Though software correctness is essential, non-functional program properties like runtime must also be considered, especially as software systems scale to cope with increasing demand. Furthermore, the sustainability of software systems is a growing concern as the environmental impact of technology becomes increasingly significant. Efficient and sustainable software can significantly reduce energy consumption and carbon footprint, aligning with global efforts towards environmental responsibility. Advancements in processor speeds and computing power have historically driven performance enhancements. However, as we hit the physical limits of microchip manufacturing and Moore's Law ends, the emphasis shifts towards software solutions that can extract the utmost performance from existing hardware. As software systems grow in complexity and size, the cost of manually improving and maintaining them escalates rapidly, highlighting the need for automated techniques for software enhancement.

A successful approach to this is Genetic Improvement (GI) [1], a subfield of Search Based Software Engineering [2], which uses heuristic search techniques to explore the space of software variants in order to find improved software. These variants are generated by applying mutation operators to the existing source code, e.g., deleting a statement, inserting a new statement, or replacing it with another. Intuitively, the search space is enormous, and enumerating through all possible variants is intractable. Thus, GI is concerned with efficiently traversing the search space using heuristic search algorithms such as Genetic Programming and Hill Climbing [3]. This is done by optimising using a fitness function which decides if a program is better than another, such as runtime or the number of tests passed. There has been success in applying genetic improvement to source code to improve its various properties. GI tools like GenProg [4], which evolve populations of program variants, were shown to be scalable to large open-source projects for Automated Program Repair [5]. There have also been successful applications of GI for program repair in the industry, including JM for Janus Rehabilitation Centre [6] and SapFix at Meta [7]. Finally, GI has been successfully applied for runtime improvements. For example, Haraldsson et al. [8] have shown the ability of GI to provide runtime improvements for ProbABEL, a C++ program used for genome-wide association analysis.

Previous Genetic Improvement research has shown the benefits of treating software engineering as a search problem. However, what is evident is the need for efficient search. This is especially important for runtime improvements since often, the software whose runtime is being optimised is very inefficient, and as a result, evaluating population variants is an expensive task. Thus, the search algorithm needs to be incredibly selective with the population variants it selects; Reinforcement Learning achieves just that.

This paper explores this form of self-learning during search using bandit algorithms [9]. These algorithms adjust the mutation operator probabilities during search so that operators more likely to produce improved software variants are more likely to be explored. These are simple enough to be integrated easily into most search strategies for GI; however, later in this report, it is highlighted how this research can be extended forward very quickly by using more novel Machine Learning techniques such as Large Language Models.

This work investigates the validity of using Reinforcement Learning techniques for intelligent operator selection in search-based software optimisation. We chose to focus on runtime optimisation to focus this research topic further. Hanna et al. [10] have already investigated the use of bandit algorithms for Automated Program Repair. However, runtime likely provides a smoother fitness landscape that is better suited for reinforcement learning. For example, in Automated Program Repair, if one test fails, multiple others do too. Furthermore, some smaller programs may only have a few test cases in their test suite. This means it is difficult to quantify how well an improvement is in relation to the other. On the other hand, runtime provides a continuous fitness function.

By integrating our RL approach into an existing GI tool, we conduct an empirical study comparing the RL approaches to each other and the baseline (uniform random selection). We test for the efficacy and efficiency of the different search strategies. Among others, our results show that Bandit algorithms outperform the Uniform Selector used in GI literature. The UCB algorithm, in particular, consistently outperformed others in producing a higher percentage of successful software variants, showcasing its theoretical robustness in efficiently balancing the exploration and exploitation trade-off.

II. Background

Reinforcement Learning (RL) is a machine learning paradigm that teaches agents how to make decisions by interacting with an environment to achieve a goal. The fundamental concept of RL involves an agent learning to make decisions through trial and error, receiving rewards or penalties based on the actions it takes in response to the state of the environment. This learning process aims to maximise the cumulative reward it receives. This learning process mimics how humans and animals learn, providing a mathematical framework.

We now outline the Multi-Armed Bandit problem [9]. This will help motivate the Multi-Armed Bandit algorithms within the context of GI. These algorithms train an agent to select actions that maximise the cumulative sum of observed rewards. The MAB problem is named after the slot machine analogy, where a gambler must decide which machines to play, how many times to play each machine, and in which order to play them to maximise their total rewards. Each machine provides a random reward from a probability distribution specific to that machine. The goal is to maximise the rewards earned through a sequence of lever pulls.

For this research, the agent is our operator selector, the actions are the mutations operators to apply, and the rewards are given by some function by the correctness and runtime of the program variant generated by that operator. Further detail of how GI search is modelled as a MAB problem is outlined in a later section. More formally, the Multi-Armed Bandit problem is a set of distributions:

$$\{\mathcal{R}_a \mid a \in \mathcal{A}\}$$

Where $\mathcal{A}$ is the set of all actions our agent can take and $\mathcal{R}_a$ is a distribution of rewards received after the selection of that action a.

At each time step t, the agent selects an action $A_t \in \mathcal{A}$. The agent is then rewarded for that action. This reward is some Random Variable $R_t \sim \mathcal{R}_{A_t}$.

The MAB problem assumes that the distribution of rewards at time step t only depends on the action taken at time t, and previous actions will not affect future rewards. This assumption is invalid in most RL problems (and GI search algorithms). However, our approach proposal outlines how this holds for Neighbourhood Search. We also assume this to be approximately true for Neighbourhood Search and empirically measure performance gains.

Now, the goal of bandit algorithms is to maximise cumulative rewards during search; more formally, the goal of the agent is to choose a sequence of actions $A_0, A_1, A_2, \ldots$ that maximises the expected cumulative reward:

$$\mathbb{E}[R_0 + R_1 + R_2 + \ldots]$$

There are two main approaches to this: value-based and policy-based. One approach is to estimate the value of each action, that is, the expected reward. Formally, the RL agent should learn the mapping $q : \mathcal{A} \to \mathbb{R}$, defined as:

$$q(a) := \mathbb{E}[R_t \mid A_t = a]$$

This is called the action value. If the agent knows this mapping, it could always choose the action that maximises this expected reward.

However, action values, $q(a)$, are given by expectations of Random Variables. Hence, these need to be approximated. One estimator $Q_t(a) \approx q(a)$, with a statistical guarantee of convergence, is the sample average of the reward:

$$Q_t(a) := \frac{\text{Total reward for action } a \text{ till time step } t}{\text{\#Times action } a \text{ has been selected till time step } t}$$

However, this presents a dual-objective problem. First, the agent should select action a as often as possible to get the values $Q_t(a)$ as close to $\mathbb{E}[R_t \mid A_t = a]$ as possible. However, the agent should simultaneously select actions that maximise the estimate for the expected reward $Q_t(a)$. Algorithms that balance these two objectives using action values $Q_t(a)$ are highlighted below.

The first algorithm used is Epsilon Greedy. Hanna et al. [10] seemed to get the best performance from this algorithm. Also, it is one of the simplest and most widely used Bandit algorithms, which made it a good candidate for this research. It works by greedily choosing the operator/action (a) with the highest estimated expected reward $(Q(a))$, but with a probability of epsilon ϵ (hyper-parameter), it chooses an operator entirely at random. This algorithm has an issue with over-exploring actions/operators of low value. That is, even if it is pretty sure that the estimated action-value $Q(a)$ for an operator a is relatively low, it still picks that operator with prob. $\frac{\epsilon}{|\mathcal{A}|}$.

The Upper Confidence Bound (UCB) algorithm mitigates this issue by taking into account the relative uncertainty of estimated expected reward $Q(a)$ and picking the operator which maximises $Q(a) + U(a)$ where $U(a)$ is a metric of uncertainty. Ideally, we want to pick an operator with a high estimated action value Q. However, if an operator has Q lower, but the uncertainty U is much higher, then it should be picked because that might instead be the optimal operator. Finally, operators with low Q and low U, those with low action value and low uncertainty, should not be picked.

In this work, the uncertainty of an operator's expected reward is given by the following quality [9]:

$$U_t(a) = c\sqrt{\frac{\ln t}{N_t(a)}} \tag{1}$$

In Eq. 1, The quantity c tightens and loosens the uncertainty bounds. This is a Hyperparameter, which dictates how much exploration of the different types of operators the operator selector algorithm should do. $N_t(a)$ is the number of times that an action a was chosen by time t.

The next operator selector algorithm was probability matching. This was another algorithm implemented by Hanna et al. [10], which was proposed by Thierens et al. [11]. The motivation for this algorithm is simple: it selects an action with a higher expected reward with a higher probability. The probability of choosing action/operator a at time t is given by $\pi_t(a)$ in Eq. 2.

$$\pi_t(a) = P_{min} + (1 - P_{min}|\mathcal{A}|)\frac{Q_t(a)}{\sum_b Q_t(b)} \tag{2}$$

The intuition behind the Probability Matching algorithm is simple; the other algorithms presented (Epsilon Greedy and UCB) try to converge to a single operator with maximum reward. This may be suboptimal if one wants to explore the space of program variants extensively. On the other hand, the Probability Matching algorithm will continually choose all operators. However, it will value them proportionally to their average reward $Q(a)$. Furthermore, a hyperparameter is introduced P_{min}. This gives the minimum probability with which each operator is picked, which reduces the likelihood of the operator selector strategy pushing that to 0. This ensures that all $Q(a)$ values are continually updated over time.

An agent's policy is a mapping $\pi : \mathcal{A} \to [0, 1]$. π is a probability mass function over the action space the agent can take. The Policy drives the decision-making of the agent; in fact, even the value-based algorithms described above contain an implicit policy (the Probability Matching algorithm makes it explicit), but this Policy uses the current average rewards $Q_t(a)$.

However, some bandit approaches try to learn the policy π directly without trying to compute any estimates $Q_t(a)$ to do so. These are called policy-based approaches. The realisation of such an algorithm is highlighted below.

The Gradient Bandit algorithm [9] maps each operator to a numerical preference $H_t(a)$. For this research, the probability with which each operator is selected at a time is given by $\pi_t(a)$ in the soft-max equation:

$$\pi_t(a) = \frac{e^{H_t(a)}}{\sum_b e^{H_t(b)}} \tag{3}$$

Then, stochastic gradient ascent updates the preferences $H_t(a)$ to maximise performance. The metric for performance is given by the expected reward $\mathbb{E}[R_t]$ at time t. This gradient ascent update is given by:

$$H_{t+1}(a) = H_t(a) + \alpha\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}, \forall a \tag{4}$$

If we simplify Eq. 4 using our definition of π in Eq. 3, then the following update rule is obtained for stochastic gradient ascent upon selecting action A_t at and obtaining reward R_t at time step t.

$$\begin{gathered} H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \overline{R_t})(1 - \pi_t(A_t)), and \\ H_{t+1}(a) = H_t(a) - \alpha(R_t - \overline{R_t})\pi_t(a), \forall a \neq A_t \end{gathered} \tag{5}$$

Note that α is the learning rate hyperparameter, and $\overline{R_t}$ is the average reward seen so far (across all actions).

Hanna et al. [10] focus on value-based algorithms, but this research explores policy-based as it may be an exciting avenue to extend this research to deep learning. For example, the preferences $H_t(a)$ can be given by the output of neural networks, such as transformers. This is discussed in the future work section. The formulation outlined will likely not lead to better performance since it may converge to a local maximum (for example, the greedy algorithm). However, this algorithm is interesting nonetheless.

III. Research Questions

We now define research questions used to focus the empirical research. Of the five operator selection techniques, four bandit in Section II + Uniform Random (baseline), we ask, in the context of Genetic Improvement:

RQ1: Which operator selection strategy leads to the best *efficacy* of search for Neighbourhood Search and Hill Climbing?

RQ2: Which operator selection strategy leads to the best *efficiency* of search for Neighbourhood Search and Hill Climbing?

IV. Experimental Protocol

This section specifies the experimental protocol to ensure reliable and reproducible results. It is important to note that our approach is agnostic of the RL algorithm used for runtime improvement.

Training/Validation Split We now discuss how we will split 1000 test instances that come with the MiniSAT benchmark [12]. Blot et al. [3] have empirically shown that a training set of 20 instances was enough to mitigate overfitting; hence, we do the same. However, they randomly select 20 cases. However, here we try to reduce the impact of overfitting further by choosing the 20 test instances that maximise *branch coverage*. In software, a branch refers to a point in the code where the program's execution can take different paths based on a condition. For example, an if-else statement creates two branches: one for when the condition is true and another for when it is false. We use the gcov [13] tool for `C++` to get branch coverage data for a training set of 20 test instances. We selected 20 test instances such that 2 of each type were present and such that when combined, the 20 test instances produced 100% coverage.

Repetitions We discussed how we split our 1000 instances into a training set of size 20 (used to measure fitness in the search algorithms) and a validation set of size 980 (used to measure the true runtime improvement). This is done four more times for a total of five repetitions. Each repetition contains a training set mutually exclusive to the others and has a branch coverage of 100%. This ensures generalisable and reliable results modulo test suite; of course, the test suite does not guarantee correctness, but serves as a good proxy. It is also important to note that each operator selector technique runs the same five training/test splits to allow for a fair comparison.

Training Budget We allow for a 2-hour budget for the Neighbourhood Search. This allows for ample exploration of the single-edit fitness landscape. 2 hours is likely too long, but it helps us motivate the search budget for the Hill Climbing experiment. From our experiments on Neighbourhood Search, we noticed that 3000 software variants could be evaluated in 2 hours. Hence, 3.5 hours was the settled time budget of Hill Climbing, as this would allow us to explore 5250 variants and create an edit list of size $\sim$10, assuming we only apply an edit after searching a neighbourhood size of 475. The justification for using a neighbourhood size of 475 is discussed in the Results section.

Hardware Specifications Experiments were run on a dedicated Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz machine with eight cores and 16 GB RAM running Ubuntu 22.04.4. 5/8 were Performance Cores (P-cores), and 3/8 were Efficient Cores (E-cores). P-cores are used for compute-intensive jobs, while E-cores are used for low-intensity jobs. For this experiment to be valid, all experiments ran on the P-cores by only parallelising across four processes.

V. Results

In this section, we discuss the results of our experiment.

1) Neighbourhood Search: The first question is which operator selector techniques have the best efficacy. We investigate four operator selector algorithms: Uniform Selector (our baseline and the standard which all other GI approaches have used) and the four Bandit algorithms, Epsilon Greedy, UCB and Probability Matching and Policy Gradient discussed in Section II. This section tries to answer the **RQ1** and **RQ2**. The efficiency and efficacy of each operator selection algorithm to guide the neighbourhood search algorithm are analysed.

Efficacy We now address **RQ1**. The following two metrics are used to measure the efficacy of the operator selector algorithms:

- **Metric 1**: measures the best runtime improvement found (ratio of new runtime to original runtime)
- **Metric 2**: measures the percentage of *unique* variants evaluated that are *successful*

The variant with the highest decrease in runtime gives the first metric. This is measured using the 980 validation set test instances not used during the search (training stage). In Fig. 1 across the five trials, all the operators seem to find a patch with a variant that leads to a variant whose runtime is $\sim 32\%$ of the original. Minimal variance is seen across the search operators. This is likely because the 2-hour search budget is large compared to the local neighbourhood of the original MiniSAT program. As a result, the search is almost exhaustive, and all 25 runs lead to finding four unique best patches across them. Concluding the relative efficacy of the different operator selector algorithms for Neighbourhood Search using this metric is impossible since they all perform similarly.

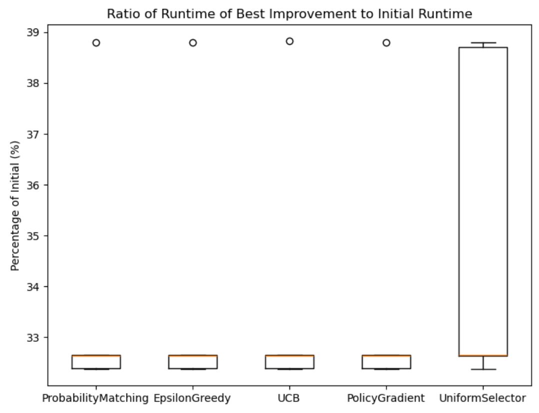

Fig. 1. Ratio of Runtime of Best Software Variant to the Original Runtime

Metric 2 is the percentage of **unique** variants evaluated that are **successful**. "Successful" is defined as a variant that both complies and passes all tests. The graph in Fig. 2 shows visible differences between the different operator selection strategies. UniformSelector (baseline) and Epsilon Greedy seem to perform the worst. PolicyGradient is marginally better. Finally, UCB has the highest success rate, and Probability Matching is second.

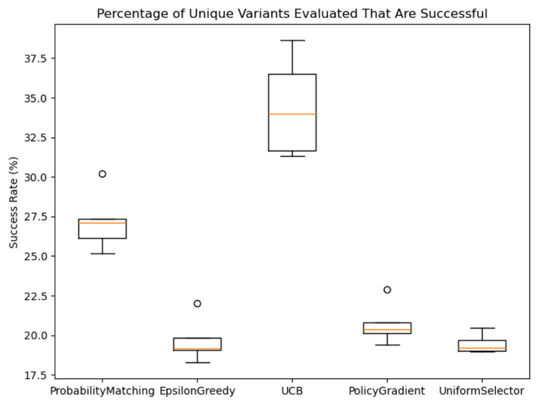

Fig. 2. Percentage of Variants That Compile and Pass All Tests

Efficiency The following metric is used to measure the efficiency, addressing **RQ2**:

- **Metric**: the number of iterations before finding the best patch/variant.

From the graph in Fig. 3, we see that most of the box plots overlap, i.e., most of the algorithms need a similar number of iterations till they find one of the four patches. There seems to be a significant amount of variance for EpsilonGreedy. This is likely because Epsilon Greedy explores a lot more variants overall. This is because Epsilon Greedy selects the operator a with the highest average reward $Q(a)$ most of the time, so the search space of variants it explores is relatively small. As a result, a lot of the variants are duplicates. However, this is not an issue since the fitness/runtime evaluations are cached

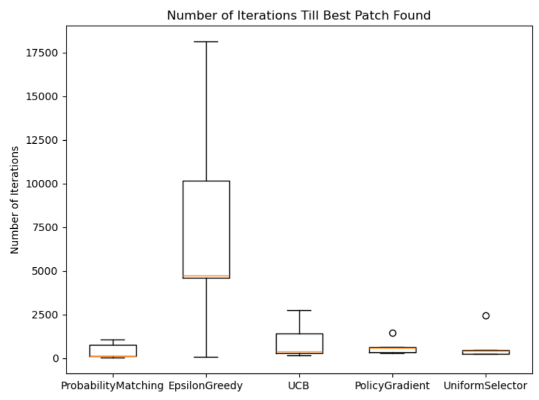

Fig. 3. Number of Neighbourhood Search Iterations Completed Before the Variant of Best Fitness Is Found

for the variants, so there is little overhead to repeat selections. More analysis of how these algorithms behave is discussed in the subsequent section.

Apart from Epsilon Greedy, all the search trials find the best variant in around 434 iterations (median value). We thus added a 10% buffer to the 434 number of iterations. Hence, we selected the local neighbourhood size for Hill Climbing to be 475.

Next, we break down individual runs of each operator selector algorithm and see what each algorithm learns.

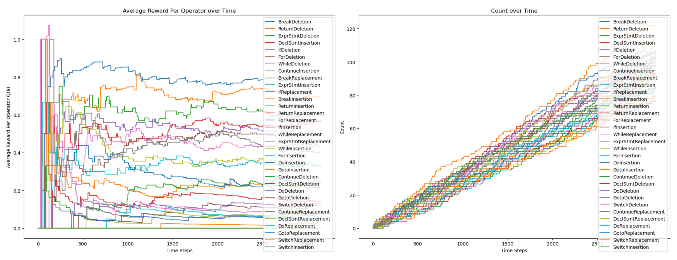

Fig. 4. Run Statistics For Uniform Selector

The Uniform Selector run for repetition 0 is in Fig. 4. From the graph showing the average reward for each operator, we see that averages stabilise quickly, around 400 iterations. Note that the Uniform Selector does not adapt its selection strategy based on the average reward of each operator. However, this statistic is tracked to get an understanding of the rewards of each operator. This is likely because there are only three break statements in the MiniSAT program – two of the three lead to a program that compiles and passes all tests. Therefore, the expected reward should lie around 66%. Most other operators create variants that do not compile or pass tests. Hence, the reward for those is closer to 0. The Uniform Selector picks each operator uniformly at random; Hence, the count over time in Fig. 4 is roughly the same across all operators. The average slope of the count over time graph is $1/33$ since that is the probability of picking each operator at any time step.

Fig. 5 shows the sample graphs for the Epsilon Greedy runs. The graph shows that the average reward per operator converges to similar values over time as those in Fig. 4. This makes intuitive sense. For example, `BreakDeletion` is still the highest; however, even within the 2-hour time-

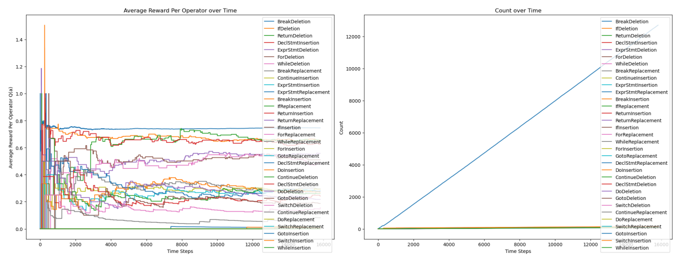

Fig. 5. Run Statistics For Epsilon Greedy

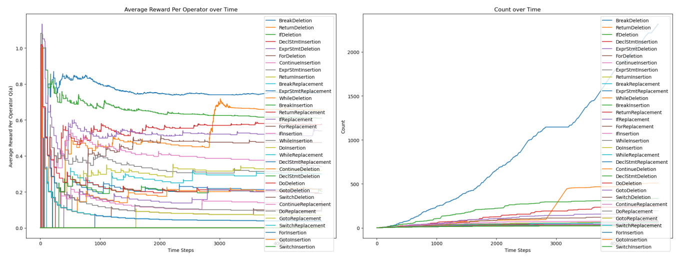

Fig. 6. Run Statistics For UCB

frame, some estimates, like those for `ReturnDeletion` and `IfDeletion`, do not match precisely across the two figures. The noisy estimates are likely because some operators are under-sampled in Epsilon Greedy and cannot converge to their correct average reward value ($Q(a)$)

`BreakDeletion` has been sampled the most, around 12,000 times (out of the 16,000) total time steps. It is important to note that most of the search's computational resources come from evaluating the program variant. That is running it against the training test cases. Hence, even though Epsilon Greedy has evaluated $\sim$ 16000 variants, only $\sim$ 2200 of those were unique, while Uniform Selection evaluated $\sim$ 3000 variants on average, and $\sim$ 2150 were unique. Hence, the number of unique variants explored between the two search algorithms was comparable.

Note also that there are three break statements in the MiniSAT program [12]. Hence, after only a couple of iterations, Epsilon Greedy has explored all three possible ways of applying `BreakDeletion`. As a result, for each subsequent time step, when Epsilon Greedy selects `BreakDeletion` greedily, the program would return a cached fitness evaluation of that variant since it has already been evaluated previously. There is some wasted computation here that can be addressed in future implementations; however, this is minimal since explored cached variants are computationally very cheap compared to rerunning against the test suite.

Finally, the efficacy of Epsilon Greedy given by Metric 1 and Metric 2 in Fig. 2 and Fig. 1 is similar to that of the Uniform Selector. The explanation is simple: Epsilon Greedy will behave the same as Uniform Selection after exploring all the `BreakDeletion` variants. This is because all the greedy selections of `BreakDeletion` will take a negligible amount of computation since the results are cached. Epsilon Greedy will employ the Uniform Selection policy the other 20 per cent of the time ($\epsilon = 0.2$).

From Metric 2 in Fig. 2, we see that UCB performs much better than the other search operators. This is justified through graphs in Fig. 6 and the algorithms' design. UCB is designed to act greedily, and it does just that. Like Epsilon Greedy, it prioritises `BreakDeletion` since that is the operator with the highest average reward.

However, unlike Epsilon Greedy, which chooses the operators with the highest average reward 80% of the time and the other operators uniformly random 20% of the time, UCB creates an internal ranking based not only on their average reward but also on how often they were explored. As a result, UCB will select other operators like `ReturnDeletion`, `IfDeletion` and `DeclareStmtInsertion` proportionally to their reward and uncertainty.

Furthermore, UCB seldom selects operators like `DeclareStmtDeletion` and `DoDeletion` when it becomes pretty sure that they are very bad operators. The other Bandit algorithms, like Epsilon Greedy and Probability Matching, will still explore operators like `DeclareStmtDeletion` well into the search even when the average reward for those is close to 0.

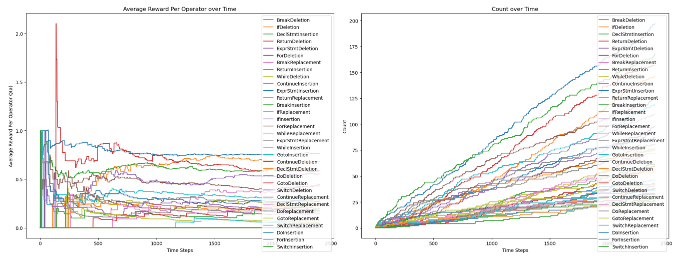

Fig. 7. Run Statistics For Probability Matching

While Probability Matching does not perform as well as UCB, it still outperforms Epsilon Greedy and Uniform Selector. Like UCB, Probability Matching (PM) selects operators proportionally to their average reward. Hence, it selects operators like `BreakDeletion` and `IfDeletion` more frequently than others. This leads to a more intelligent search.

However, PM has a hyperparameter P_{min}, which was set to $\frac{1}{\text{number of operators}}$, which for this experiment is 1/33. Hence, $\sim$ 3% of the time, it selects very poor operators like `DeclareStmtReplacement`, which have an average reward (Q) close to 0. On the other hand, UCB would select this operator a handful of times and then, subsequently, rarely once the uncertainty (U) on the average reward becomes low.

This hyperparameter setting minimum probability $P_{min} > 0$ is important. If $P_{min} = 0$, operators with $Q(a) = 0$ will never be selected. Hence, they will never be updated. Hence, this hyperparameter P_{min} is needed to ensure sufficient exploration.

The final algorithm discussed is Policy Gradient. The results are seen in Fig. 8. Note that the Policy Gradient algorithm does not try to learn the average reward $Q(a)$; instead, it learns the preferences $H(a)$ and then chooses an operator based on the probabilities given by Eq. 3.

From Fig. 8, we see that again, operators such as `BreakDeletion` and `ReturnDeletion` have the highest quality, similar to the Q rankings for the Uniform Selector in Fig. 4. This would intuitively make sense; through the gradient

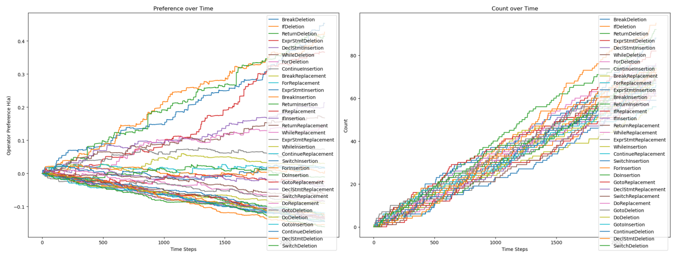

Fig. 8. Run Statistics For Policy Gradient

ascent step in Eq. 5, the operators with higher rewards get a higher preference, and those with a lower reward get a lower preference.

We also see that the operators with the highest reward are selected more often; for example, `IfDeletion` is most often selected. Also, the slope of each operator's count graphs is similar. Hence, it is apparent that Policy Gradient with $\alpha = 0.01$ behaves similarly to the Uniform Selector. This is because $\alpha = 0.01$ is a small learning rate; hence, it takes time for Policy Gradient to become greedy and explore more than Epsilon Greedy, UCB, and PM. This is good as it explores a wider variety of variants. However, it frequently selects operators like `DeclareStmtDeletion`, which may lead to wasted computation. For this reason, UCB is still a better Bandit algorithm.

2) Hill Climbing: The same two metrics used for Neighbourhood Search are also used for Hill Climbing to answer **RQ1**. These are analysed below.

Metric 1 measures the relative runtime against the unseen test cases in the validation set. The results are seen in Tab. I and visualised in Fig. 9. From Tab. I, we see that some of the variants found during the search overfit to the training set. That is, they passed all the 20 tests used during training but failed on one of the 980 used for validation. This is an unfortunate outcome, but most repetitions seem to lead to variants that pass all tests and are deemed correct by proxy. Hence, when compiling results, ones that break the tests are ignored.

The results in Fig. 9 seem to favour the bandit algorithms. We note that in the figure, lower is better. The result for Uniform Selector was 37.10%, while the median result for Probability Matching was $\sim$ 30.5%. The other Bandit Algorithms seem similar, and it is hard to make conclusive analyses due to the significant variance in results across repetitions.

We also note that Epsilon Greedy manages to produce a variant which runs 26.89% of the original runtime. Our trials consistently beat or match the results from the Local Search implementation by Blot et al. [3]. The best result they achieve is 32%, and they average 36% for their Local Search implementations. However, this is not a totally fair comparison since their search budget is lower at 1000 time steps, while ours is 5000-6000. However, they added an additional validation step to reduce software bloat; this would likely reduce runtime further in our approach.

Metric 1 is quite a noisy metric, as random chance is often a more significant predictor of the best variant than the ability of the search strategy. Instead, Metric 2, which measures the

TABLE I
RUNTIME IMPROVEMENTS OF THE OPERATOR SELECTOR ALGORITHMS. GIVEN BY THE RATIO OF CPU INSTRUCTION COUNT OF THE BEST PROGRAM VARIANT FOUND VS THE ORIGINAL PROGRAM. LOWER IS BETTER. "TEST ERROR" = THE VARIANT FOUND RAN ON THE 20 TEST CASES IN THE TRAIN SET, BUT FAILED IN THE VALIDATION SET.

RL algorithm	instruction count of initial (%)
ProbabilityMatching	Test Error
	30.70
	36.06
	30.06
	Test Error
EpsilonGreedy	Test Error
	33.92
	36.35
	32.69
	27.24
UCB	31.18
	31.49
	31.90
	34.09
	36.89
PolicyGradient	35.56
	31.84
	31.58
	30.08
	Test Error
UniformSelector	38.09
	31.85
	37.10
	Test Error
	Test Error

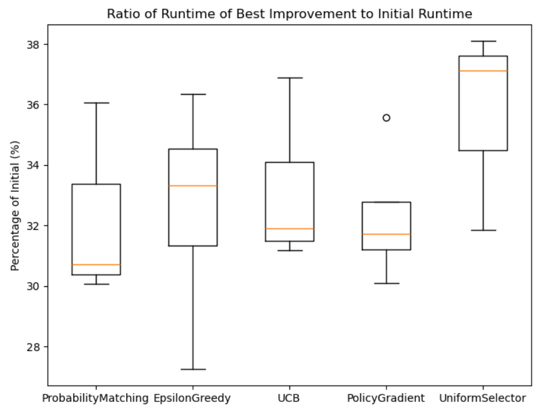

Fig. 9. Ratio of Runtime of Best Software Variant to the Original Runtime

number of variants that compile and pass all tests, is likely a much better performance measure.

Again, the results in Fig. 10 favour the bandit algorithms.

All the bandit algorithms are able to explore a greater number of successful variants, UCB especially; 44% of the UCB guided Hill Climbing steps produced variants which passed all tests, while that number was around 22% for UniformSelector. UCB was the best, followed by Probability Matching then Epsilon Greedy, Policy Gradient and then finally the Uniform Selector. This is consistent with the results summary and analysis for the simple Neighbourhood Search.

Efficiency As stated earlier, the metric for efficiency (**RQ2**) is the number of iterations till the best variant is found. This

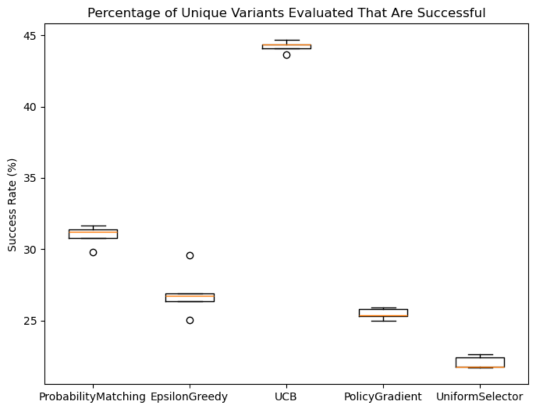

Fig. 10. Percentage of Variants That Compile and Pass All Tests

was more important in determining the neighbourhood size in a Neighbourhood Search; however, it is still interesting to analyse for comparison purposes. It is not easy to produce an effective conclusion due to the high variance in the box plots. Probability Matching seems the best, likely because it covers a larger breadth of variants compared to algorithms like UCB and Epsilon Greedy. However, at the same time, Probability Matching still prioritises those operators with higher average rewards. Uniform selector seems to be the second fastest to find the best variant, followed by UCB and finally Epsilon Greedy. Again, UCB and Epsilon Greedy act greedier and cover a less diverse set of variants. As a result of this greediness, they often reexplore the same variant twice. However, this is not necessarily bad because software variant evaluations are cached in the Magpie implementation. So, revisiting an existing variant does not have a significant overhead cost.

Hence, a better efficiency measure in this setting would be the percentage of the 3.5-hour search budget needed to find the best variant. This is given in Fig. 11. This graph gives a more representative view of performance; we see it is difficult to compare the algorithms due to the high variance in results across trials. However, they all seem to perform similarly. It is evident that they all seem to find their best variant toward the end of their 3.5-hour search. This might indicate that training beyond the 3.5-hour budget would be beneficial and that software variants with even more edits would run faster. For example, there are many `print` and `assert` statements in the source code, so removing all of them likely leads to quicker code. However, there is also a risk of overfitting to the 20 test instances in the training set. Since a variant that runs faster on the training set may not do so on the 980 other test instances in the validation set. We see some examples of this in Table I.

Further Discussion The results for Hill Climbing are similar to those with the Neighbourhood Search experiments. For example, `BreakDeletion` still has the highest average reward alongside others like `ReturnDeletion`. Hence, it is apparent that even as edits are added to the software, the search landscape (i.e., operator qualities) does not change much. Hence, it is evident that many edits need to be present for the search landscape to be different. The best edit found

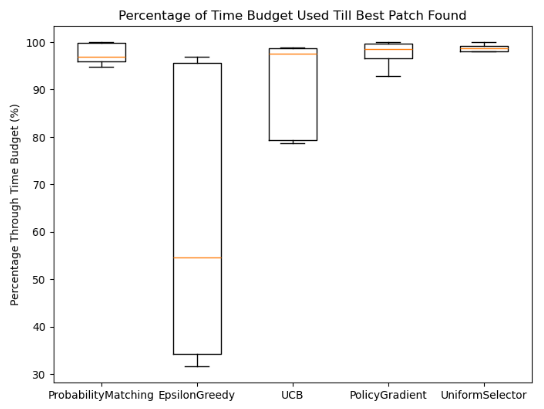

Fig. 11. Percentage Through Time Budget Till Best Variant Is Found

took only 27.24% of the original runtime to evaluate the 980 test instances in the validation split. Manually analysing the correctness is beyond this work's scope; however, they pass all 1000 test instances and are deemed correct in that regard. The Epsilon Greedy and all other operator selectors heavily value code deletion. This is a common occurrence with GI for runtime improvement. It is very easy to reduce runtime, cutting test suite redundant code. For example, none of the test cases checks for exceptions, so the assert statements are redundant and are, hence, deleted during the search.

VI. Threats to Validity

The first threat to validity is the generalizability of this research arising from using MiniSAT, which may not fully represent large-scale software systems. However, it is an important real-world tool. Therefore, expanding our research to cover multiple languages and tools could provide a more comprehensive perspective on the generalizability of the results. Secondly, we were not able to tune our approach's hyperparameters due to the risks of overfitting to our GI benchmark. However, hyperparameter tuning could lead to better results. Furthermore, it is also possible that the hyperparameters used for this research would not generalise well to other software search landscapes. Finally, relying on the testing in the first place is a crucial threat to GI. Testing only serves as a proxy to correctness, but it is impossible for a finite test suite to cover all the input cases. One possible extension could be formal verification methods, which are computationally expensive and beyond this work's scope.

VII. Conclusions

In this research, we have thoroughly explored the application of various reinforcement learning strategies to enhance the effectiveness of genetic improvement techniques in optimising program runtime. We implemented a series of experiments involving different operator selection algorithms, including Uniform Selector, Epsilon Greedy, UCB (Upper Confidence Bound), Probability Matching and Policy Gradient, to guide the mutation operators within both Neighbourhood Search and Hill Climbing search algorithms.

Our empirical investigation centred on the efficiency and efficacy of the software search algorithms. Regarding efficacy, for the Hill Climbing local search, we found that all the Bandit algorithms tests outperformed the Uniform Selector used in GI literature, both in Metric 1 (relative runtime improvement) and Metric 2 (percentage of successful variants). The UCB algorithm, in particular, consistently outperformed others in producing a higher percentage of successful software variants, showcasing its theoretical robustness in efficiently balancing the exploration and exploitation trade-off. However, we found it relatively difficult to compare the search efficiency of the search algorithms. What seemed apparent is that if the experiments were allowed to run longer, better patches would be found; from Fig. 11, almost all search trials find the best variants towards the end of the 3.5-hour time budget.

This comprehensive analysis not only reinforces the viability of using reinforcement learning techniques in genetic improvement but also highlights their potential to significantly refine the search process for runtime optimisation. In the future, we hope to validate our technique on larger software systems, use broader contextual operators and search techniques, refine the reward functions, and integrate deep learning models for handling complex data.

The replication package and full results are available at: https://github.com/SOLAR-group/RL_Runtime_Optimization.

References

[1] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic Improvement of Software: A Comprehensive Survey," *IEEE Transactions on Evolutionary Computation*, vol. 22, pp. 415–432, June 2018.

[2] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo, "Search Based Software Engineering: Techniques, Taxonomy, Tutorial," in *Empirical Software Engineering and Verification* (B. Meyer and M. Nordio, eds.), vol. 7007, (Berlin, Heidelberg), pp. 1–59, Springer Berlin Heidelberg, 2012.

[3] A. Blot and J. Petke, "Empirical Comparison of Search Heuristics for GI of Software," *IEEE Transactions on Evolutionary Computation*, vol. 25, pp. 1001–1011, Oct. 2021.

[4] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, (New York, NY, USA), pp. 947–954, ACM, July 2009.

[5] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *ICSE*, pp. 3–13, June 2012.

[6] S. O. Haraldsson, J. R. Woodward, A. E. I. Brownlee, and K. Siggeirsdottir, "Fixing bugs in your sleep: How gi became an overnight success," in *GECCO Companion*, GECCO '17, (New York, NY, USA), pp. 1513–1520, ACM, July 2017.

[7] "SapFix: Automated End-to-End Repair at Scale | IEEE Conference Publication | IEEE Xplore." https://ieeexplore.ieee.org/document/8804442.

[8] "Gi of runtime and its fitness landscape in a bioinformatics application," in *GECCO Companion*.

[9] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning Series, Cambridge, Massachusetts: The MIT Press, second edition ed., 2018.

[10] C. Hanna, A. Blot, and J. Petke, "Reinforcement Learning for Mutation Operator Selection in Automated Program Repair," June 2023.

[11] D. Thierens, "An adaptive pursuit strategy for allocating operator probabilities," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, (New York, NY, USA), pp. 1539–1546, ACM, June 2005.

[12] "MiniSat Page." http://minisat.se/MiniSat.html.

[13] "Gcov (Using the GNU Compiler Collection (GCC))." https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.