Effective Adaptive Mutation Rates for Program Synthesis

Andrew Ni
ani24@amherst.edu
Amherst College
Amherst, MA, USA

Lee Spector
lspector@amherst.edu
Amherst College
Amherst, MA, USA

ABSTRACT

The problem-solving performance of many evolutionary algorithms, including genetic programming systems used for program synthesis, depends on the values of hyperparameters including mutation rates. The mutation method used to produce some of the best results to date on software synthesis benchmark problems, Uniform Mutation by Addition and Deletion (UMAD), adds new genes into a genome at a predetermined rate and then deletes genes at a rate that balances the addition rate, producing no size change on average. While UMAD with a predetermined addition rate outperforms many other mutation and crossover schemes, we do not expect a single rate to be optimal across all problems or all generations within one run of an evolutionary system. However, many current adaptive mutation schemes such as self-adaptive mutation rates suffer from pathologies like the vanishing mutation rate problem, in which the mutation rate quickly decays to zero. We propose an adaptive bandit-based scheme that addresses this problem and essentially removes the need to specify a mutation rate. Although the proposed scheme itself introduces hyperparameters, we either set these to good values or ensemble them in a reasonable range. Results on software synthesis and symbolic regression problems validate the effectiveness of our approach.

CCS CONCEPTS

• **Computing methodologies → Genetic programming**.

KEYWORDS

Uniform Addition by Mutation and Deletion, Adaptive Mutation Rate, Max K-armed Bandits, Genetic Programming, Software Synthesis

ACM Reference Format:
Andrew Ni and Lee Spector. 2024. Effective Adaptive Mutation Rates for Program Synthesis. In *Genetic and Evolutionary Computation Conference (GECCO '24), July 14–18, 2024, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3638529.3654135

1 INTRODUCTION

Genetic Programming (GP) is a problem-solving tool that uses concepts from biological evolution to find a program that solves a specific target problem. When applied to software synthesis, GP evolves a population of programs to satisfy a set of user-defined test cases, and successful individuals are then tested for generalization to a set of previously unseen test cases.

Many of the best results to date on benchmark program synthesis problems have been produced by PushGP, a stack-based GP system that uses linear genomes [6, 12, 13, 17, 30]. While many mutation and crossover schemes have been tested for program synthesis with PushGP, the variation scheme that has produced the best results recently is Uniform Mutation by Addition and Deletion (UMAD) [13]. In contrast to uniform mutation, which replaces genes with random genes with some probability, UMAD decouples the replacement step into an addition step and a deletion step. Genes are first added into a linear genome at a certain rate, and then deleted from the augmented genome at a rate that balances out the addition. This makes mutation more flexible compared to random replacement, and results in better problem solving performance. While the authors show that reasonable performance can be obtained with a relatively wide range of mutation rates, and that a UMAD rate of 0.1 generally works well for many problems, they find that doubling or halving the mutation rate can sometimes improve or greatly hinder the performance of the GP system.

Since the aim of software synthesis is to find a single solution to a set of test cases, in practice we expect end users to conduct a single, or at most a few PushGP runs until a single solution to their problem has been obtained. At that point the user will have no interest in conducting many more runs on the same problem in order to tune hyperparameters. Therefore, in order to tune hyperparameters to each individual problem, we must use an adaptive method to find optimal hyperparameter values as a run proceeds.

In the process of biological evolution, the mutation rate of organisms' DNA is evolved alongside the organisms themselves [26]. Self-Adaptation of Mutation Rates (SAMR) is an adaptive technique inspired by the biological evolution of mutation rates. In its first formulation by Bäck [3], mutation rates were represented alongside the genome itself as a bitvector. During variation, the mutation rate section of the bitvector was decoded, and then the decoded rate was used to mutate the entire genome, including the mutation bitvector. While this scheme is directly analogous to the mutation of DNA and its repair mechanisms in biological evolution, recent research in real-coded systems have shifted to using real values to represent mutation rates, and using a special meta mutation rate to vary the mutation rate.

However, in contrast to the sheer diversity produced by biological evolution, these SAMR schemes often struggle with premature convergence due to the vanishing mutation rate problem [7, 10, 19]. As most mutations in a difficult problem are deleterious, the only individuals which can survive for many generations in a row are those with very small mutation rates.

GECCO '24, July 14–18, 2024, Melbourne, VIC, Australia
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0494-9/24/07…$15.00
https://doi.org/10.1145/3638529.3654135

Here we propose an adaptive controller based on multi-armed bandits and tile-codings [34] to adapt the UMAD rate to each software synthesis problem. To address the vanishing mutation rate problem, we use extreme value-based credit assignment [9, 36], which allows mutation operators to be mostly deleterious as long as they occasionally produce very beneficial mutations. The use of tile-codings enables us to take advantage of the continuous domain of mutation rates, generalizing the credit assignment to a range of mutation rates for improved sample complexity. While our work is mainly focused on the current best mutation operator for software synthesis in PushGP, this scheme is not tied to a specific genetic operator and can be used to adapt other hyperparameters beyond the UMAD rate.

This paper is organized as follows: In section 2, we give a brief overview of the PushGP system for software synthesis and the UMAD mutation operator, as well as various adaptive mutation rate schemes in the literature. In section 3, we present our adaptive mutation rate scheme. In section 4, we present our experiments building up from initial validation experiments to the final performance on software synthesis and symbolic regression problems.

2 BACKGROUND AND RELATED WORK

In this section, we give a brief overview of the PushGP system and the UMAD mutation operator, as well as a brief review on the strategies used for adaptive mutation rates.

2.1 PushGP

PushGP[29, 30] is a linear-genome GP system that evolves Push programs for solving software synthesis problems. Push [29] is a stack-based language with a separate stack for each datatype, including one stack for the program code itself. Instructions in the program code will read off one or more values from the top of one or more datatype stacks, and push one or more values onto one or more datatype stacks. When there are not enough items on the required datatype stacks, instructions can no-op, producing no change to the program state. At the end of execution, the output of the program is simply the top one or more items from a predetermined datatype stack (e.g. the top integer for the `Vector Average`), or a special token if the requested stack does not have enough elements. With the addition of instructions for moving items from the middle of a stack to the top of the stack, as well as code-modifying commands to allow for complex control structures, the Push language becomes Turing-complete. While Push programs have a hierarchical structure, they are constructed from linear plushy genomes, where genes encode the instructions and constants in a Push program, as well as the opening and closing of nested Push code blocks.

2.2 UMAD

Uniform Mutation by Addition and Deletion (UMAD) [13] is a mutation operator for variable-length, symbolic, linear genomes such as those evolved using the PushGP system. The key concept of UMAD is to extend the expressibility of traditional *replacement* based mutation operators by separating the deletion of an existing gene from the addition of a new gene. In UMAD, more specifically size-neutral UMAD, new genes are first inserted into a genome at a predetermined rate, usually set to 0.1. Then, genes are deleted from this augmented genome at a rate that brings the expected length of the mutated genome back to the length of the starting genome. For an addition rate of μ, the deletion rate needed to balance the addition rate is $\frac{1}{1+\mu}$. In general, when we refer to the UMAD rate, we mean the addition rate, and leave the deletion rate to be implicitly determined. In this paper, we extend UMAD to be defined for all positive-valued rates as follows: given a UMAD rate μ, during the addition step, for every instruction in the current genome we add $\lfloor \mu \rfloor$ new instructions with probability $\{\mu\} = \mu - \lfloor \mu \rfloor$, and $\lfloor \mu \rfloor + 1$ new instructions with probability $1 - \{\mu\}$. The deletion probability and deletion step remain the same as before. UMAD with a rate of 0.1 has been shown to outperform or equal a variety of other mutation operators and combinations of mutation and crossover operators on software synthesis problems using the PushGP system. However, even though Helmuth et al. [13] show that the UMAD rate is robust and performs well at different rate values, there is no single mutation rate that works optimally across all problems, and doubling or halving the default mutation rate can sometimes improve performance. We hypothesize that more "fundamentals-based" problems, in which a clever solution idea is the crux of the problem, may require a larger UMAD rate to quickly search over the possible solution ideas. In contrast, more "implementation-based" problems, which have many tricky details and edge cases that need to be carefully considered, may require a lower UMAD rate for a slow, methodical descent to a solution.

2.3 Adaptive Mutation Rates

Research on mutation rates is an extensively studied subfield of genetic algorithms [1, 3, 4, 21, 24, 25]. As in Aleti et al.[1], we can classify mutation rate schemes into the *fixed*, *self-adaptive*, and *adaptive* categories.

In the fixed mutation rate scheme the mutation rate remains constant over all problems and all generations. While hyperparameter optimization can be used to find the optimal fixed mutation rate, research has also shown that the optimal mutation rate can change over the course of evolution [11, 31], making any fixed mutation rate suboptimal. The current mutation scheme used in PushGP is that of a fixed UMAD mutation rate, with a default rate of 0.1.

The self-adaptive mutation rate (SAMR) scheme aims to simultaneously evolve both individuals and their mutation rates [27]. In SAMR, each individual is associated with a mutation rate. During selection, individuals together with their mutation rates are chosen based on the individual's fitness function. During variation, each individual's genome is varied according to their mutation rate, and their mutation rate according to some preset meta-mutation rate. The SAMR scheme is attractive because of its elegance and its similarity to biological evolution [8]. However, SAMR schemes often suffer from the vanishing mutation rate problem [7, 10, 19], in which mutation rates decay to 0 and cause premature convergence. This is often attributed to the observation that SAMR schemes are myopic, only optimizing in the short term [21]. The extent to which this short-term optimization is beneficial or detrimental can be dependent on the problem domain, the selection strength, or the solution representation [10]. However, in general SAMR struggles with more difficult problems that require optimizing for long-term improvement, often producing mutation rates which are far below optimal [7]. Some strategies for alleviating this pathology include limiting the mutation rate to within a certain interval [4], adding a constant to the mutation rate during variation [19], or high selection pressure [25].

The adaptive mutation rate scheme attempts to explicitly optimize the mutation rate for better performance. Various adaptive mutation rate algorithms have been proposed to counteract the vanishing mutation rate problem. Fialho et al. [9] propose to select mutation rates based on their extreme value statistics, using a dynamic multi-armed bandit and the upper confidence bound algorithm [2] to balance exploration and exploitation. In contrast to the short-sightedness of SAMR, they directly optimize for the long-term effectiveness of mutation rates by choosing mutation rates with the best maximum improvement over many sampled individuals. On the other hand, Kumar et al. [21] propose to use a co-evolutionary scheme to optimize the mutation rate, which enables them to exploit the continuous nature of mutation rates. At each generation, they assign each mutation rate to a group of selected parents, producing a set of children. From that set of parent-child pairs, they then calculate the fitness of the mutation rate as the best change in fitness from parent to child in the group. They show that this maximum value-based credit assignment is effective at avoiding premature convergence compared to SAMR, and is able to converge precisely to the optimal mutation rate.

Our work lies in the category of *adaptive* mutation rate schemes, and is adjacent to both the aforementioned adaptive schemes. Like Fialho et al. [9], we use a windowing method to track the maximum change in fitness obtained by a mutation rate over some number of sampled individuals, and also use a multi-armed bandit controller to balance exploration and exploitation. However, in order to take advantage of the continuous domain, we propose to use tile codings [34] to learn the bandit controller's weights. Similar to Kumar et al.'s [21] use of a meta-mutation rate to explore the range of possible mutation rates, we use sampling noise to choose mutation rates located around the current best rate. However, their meta-evolutionary scheme is tied to the underlying evolutionary scheme, as the suggested population size of the mutation rates approaches $N^{\frac{3}{4}}$ at large N for an underlying population size N. This ties the sampling horizon of their max-improvement credit assignment to the population size, and makes their scheme less suited to algorithms that sample only a few individuals at a time, such as steady-state GAs or hill climbers. In contrast, our bandit scheme is able to optimize for maximum improvement over an arbitrary number of sampled individuals, and only needs the underlying scheme to repeatedly generate and evaluate children from selected parents.

3 METHODS

In this section, we present our adaptive scheme for controlling the UMAD rate. Our method uses multi-armed bandits and tile codings for better sample complexity, and optimizes for the expected maximum improvement in fitness over many mutations, which we find to be a better indicator of the problem-solving performance of a mutation rate.

3.1 Reward Function

Lexicase selection[18] is a selection algorithm for multiobjective genetic algorithms that has been shown to outperform selection methods based on aggregated fitness measures on program synthesis problems. The success of lexicase selection is commonly attributed to its ability to select "elites," individuals which may have poor overall performance but which have excellent performance on a subset of the training cases[14, 15]. To encapsulate this idea of focusing on the individual's best performances, we optimize the UMAD rate with respect to the symmetrical log scale-transformed error function. For each error value x_i in an individual's error vector, we compute the transformed error on that test case as

$$f(x_i) = \text{sgn}(x_i)\log(c + |x_i|) \tag{1}$$

We base our error transformation off of the log function so that very poor performances on certain test cases will not overwhelm good performances on other test cases, following the lexicase idea. The parameter c roughly represents the resolution of our proxy error function, as it behaves linearly in the region $[-c, c]$ and logarithmically outside. Therefore, we set it equal to the lowest non-negative error we expect to see from the system. Since the function minimization problems have one-dimensional error vectors, we do not use this transformation in those experiments. For more details on the parameter values used, see appendix D.

Given a parent with errors $[e_0, \cdots, e_m]$ that was mutated to produce a child with errors $[e'_0, \cdots, e'_m]$, we compute the immediate reward obtained as the amount of improvement (decrease) in the average transformed error from parent to child

$$r = \frac{1}{m}\sum_{i=0}^{m}\log(1 + e_i) - \frac{1}{m}\sum_{i=0}^{m}\log(1 + e'_i) \tag{2}$$

Similarly to Fialho et al.[9], we track the maximum reward $r_{\max}$ obtained over the last *len_history* samples from the same UMAD rate range and use that value to update our adaptive controller.

3.2 Multi-Armed Bandits

Multi-armed bandits[34] are one commonly used controller for adaptive mutation rates[9]. The (stationary) multi-armed bandit problem is formulated as such[5]: Given k slot machines, each with an unknown payout distribution, and a maximum number of pulls of a slot machine, how do we allocate pulls to each slot machine to maximize our expected total payout? In our case, the slot machines correspond to different UMAD rate ranges and the payouts to the reward function defined above. Different strategies have been devised for balancing exploration and exploitation in multi-armed bandits, including simple strategies like epsilon-greedy or boltzmann exploration as well as more complex strategies with better theoretical properties like upper confidence bound (UCB) based algorithms[20]. In this work, we use an epsilon-greedy strategy combined with sampling noise for exploration. We anneal the epsilon value from 1 to 0.01 over the first 5 generations, at which point it is kept constant at 0.01 for the rest of the run. Our sampling noise takes the form of gaussian perturbation, where we sample UMAD rate intervals in the vicinity of the best rate interval according to a normal distribution.

3.3 Tile Codings

Current works using multi-armed bandits for adaptive mutation rate control do not take advantage of the continuous nature of this domain, instead partitioning possible rates into separate intervals[9, 36]. Intervals which are too large will mask the performance of good mutation rates with the poor performance of significantly different mutation rates. On the other hand, intervals which are too small suffer from poor sample complexity, as each interval needs to be sampled many times in order to obtain an accurate estimate of the expected reward. In order to combine generalization with precision, we use tile-codings[34] to learn the expected $r_{\max}$ associated with different UMAD rates. In tile codings, we create many tilings of our desired parameter range with different tile offsets and widths. When we observe a reward associated with a specific point, we update all the tiles covering that point using that reward. Then, when we want to assess the value of a point, we average the values of all the tiles covering that point. With enough random tilings, we can distinguish between very small UMAD rate ranges with high precision, while each sampled reward will still update tiles covering a large range of UMAD rates for good generalization and low sample complexity.

Given the very stochastic rewards in our system, we use SGD with nesterov momentum[33] as well as ensembling to stabilize learning. A tile coding defined on the range $[l, r]$ with tile offset o and width w will track the value (expected $r_{\max}$) and momentum associated with the intervals $[l, l+o)$, $[l+o, l+o+w)$, $\cdots$, $[l+o+\lfloor \frac{r-l-o}{w} \rfloor \cdot w, r)$. When a tile coding with values $[v_0, \cdots, v_n]$ and momentum $[m_0, \cdots, m_n]$ observes a reward r at position x, it will calculate the index of the tile that covers x as $i = \lfloor \frac{x-o-l}{w} \rfloor + 1$. Then, using the learning rate γ and momentum factor μ, it will calculate the gradient $g = 2(v_i - r)$, update the momentum $m_i \leftarrow \mu m_i + g$, and update the value using the updated momentum $v_i \leftarrow v_i - \gamma(g + \mu m_i)$.

In practice, we create multiple multi-armed bandits each with a randomized learning rate. During variation, a multi-armed bandit is chosen at random to sample for a mutation rate. Then, once we have computed the immediate reward r for that sampled mutation rate, all of the bandit controllers are updated with that information.

Algorithm 1: Sampling UMAD rate from a single bandit

```
Data:
    • Search range $[l, r]$
    • Search resolution $res$
    • Exploration noise $\sigma$
    • Epsilon-greedy exploration rate $\epsilon$
    • Number of tile codings $num\_codings$
    • Base coding $base\_coding = \text{Coding}(l, r, 0, res)$
    • Tile codings $tile\_codings = \{\text{Coding}_i(l, r, o_i, w_i)\}$
    • $V = \{v_{i,j}\}$ the value of the $j^{th}$ tile in the $i^{th}$ tile coding
Result: $\rho$, the UMAD rate to use for mutation
$weights \longleftarrow []$
$num\_base\_tiles \longleftarrow \lfloor \frac{r-l}{res} \rfloor$
/* Compute tile weights in the base coding */
for $i \leftarrow 0$ to $num\_base\_tiles$ do
    /* Average the associated values in each tile
       coding */
    $total \longleftarrow 0$
    for $j \leftarrow 0$ to $num\_codings$ do
        $idx \longleftarrow \lfloor \frac{i \cdot res - o_j}{w_j} \rfloor + 1$
        $total += v_{j,idx}$
    end
    $weights$.append($\frac{total}{num\_codings}$)
end
if (rand) < $\epsilon$ then
    /* Sample a random tile */
    $best\_tile \sim \lfloor \mathcal{U}([0, num\_base\_tiles]) \rfloor$
else
    /* Sample around the best tile */
    $best\_tile \sim \text{argmax}(weights) + \lfloor \mathcal{N}(0, \sigma) \rfloor$
    $best\_tile \longleftarrow \max(\min(best\_tile, num\_base\_tiles - 1), 0)$
end
/* Uniformly sample a log UMAD rate from within
   the selected tile */
$log\_rate \sim \mathcal{U}([l + res \cdot best\_tile, l + res \cdot (best\_tile + 1)])$
/* Convert from log UMAD rate to UMAD rate */
return $e^{log\_rate}$
```

3.4 Adaptive UMAD Rate

We combine the previous sections into an adaptive controller for UMAD rates in problem synthesis. The algorithm for sampling a UMAD rate from a single bandit is given in Algorithm 1, and the algorithm for updating a single bandit is given in Algorithm 2. In practice, we have multiple bandits, updating them all with the same rewards and randomly choosing a bandit to sample from.

4 EXPERIMENTS

In this section we present our experiments validating the effectiveness our algorithm, exploring the fitness landscape of software synthesis problems, and demonstrating the performance of our controller on genetic programming and symbolic regression problems.

4.1 Function Minimization

We first validate the effectiveness of our algorithm on some function minimization problems. In these problems, a real-valued n-dimensional vector is evolved to minimize a synthetic fitness function. We use the common test functions `Ackley`, `Greiwank`, `Rastrigin`, `Rosenbrock`, `Sphere`, and `Linear` [21, 23, 32]. However, due to differences in genetic algorithm hyperparameters such as the truncation size, our results are not comparable to those in the literature. The test problem definitions can be found in appendix A. We run each problem with a 100-dimensional test function using a population size of 100 + 1 elite, and a generation limit of 1000. We initialize the GA population according to the normal distribution $\mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$ where the standard deviation σ is on roughly the same order of magnitude as the dimensions recommended in [32], or 1 if such a recommendation is not available. For specific details, see

Algorithm 2: Updating a single bandit

```
Data:
    • Search range $[l, r]$
    • Search resolution $res$
    • Learning rate $\gamma$
    • Momentum factor $\mu$
    • $len\_history$ the number of past rewards to max over
    • $tile\_coding = \text{Coding}(l, r, o, w)$ the tile coding
    • $V = \{v_i\}$ the value of the $i^{th}$ tile in the tile coding
    • $M = \{m_i\}$ the momentum of the $i^{th}$ tile in the tile coding
    • $R = \{r_i\}$ the history of rewards obtained by tiles in the tile
      coding, represented as deques with max length $len\_history$
    • $x$ the sampled UMAD rate
    • $E = \{e_i\}$ the parent error vector
    • $E' = \{e'_i\}$ the child error vector
Result:
    • $\{v_i\}$ the updated tile coding values
    • $\{m_i\}$ the updated tile coding momentum
/* Locate the tile containing the UMAD rate    */
$idx \longleftarrow \lfloor \frac{\log x - l - o}{w} \rfloor + 1$
/* Compute immediate reward                    */
$reward = (\log(1 + E') - \log(1 + E))$.mean()
/* Compute max over reward history             */
$r_{idx}$.push($reward$)
$max\_reward \longleftarrow \max(r_{idx}[0], \cdots, r_{idx}[len\_history - 1])$
/* Compute gradient and update parameters      */
$g \longleftarrow 2(v_{idx} - max\_reward)$
$m_{idx} \longleftarrow \mu \cdot m_{idx} + g$
$v_{idx} \longleftarrow v_{idx} - \gamma(g + \mu \cdot m_{idx})$
return $\{m_i\}, \{v_i\}$
```

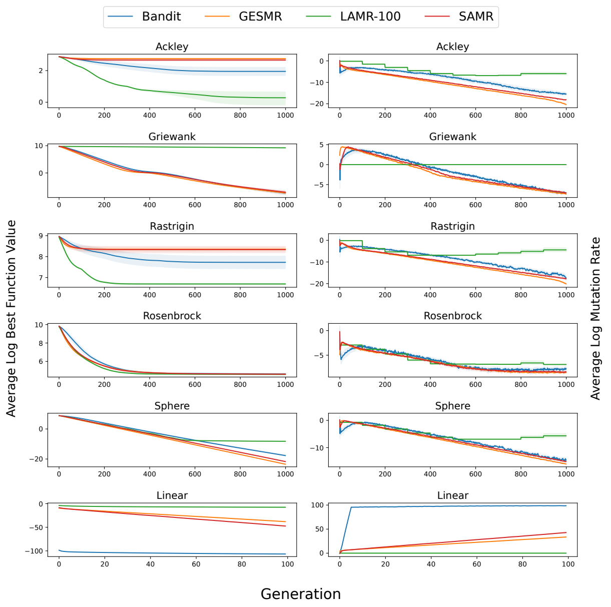

Figure 1: Performance of four adaptive mutation rate schemes on function minimization problems. 95% confidence intervals obtained by bootstrapping. The bandit controller is competitive with GESMR, with slightly slower adaptation on easy problems like `Rosenbrock` and `Sphere`, but better final fitness on rugged domains like `Ackley` and `Rastrigin`.

Table 1: Average final function value of adaptive mutation rate schemes on function minimization problems. The LAMR-100 oracle is not used for comparison. The best non-oracle value is shown in bold. Results significantly worse than the best scheme with $p < 0.05$ using Welch's t-test are underlined.

Problem	Bandit	GESMR	SAMR	LAMR-100
Ackley	**10.1**	<u>15.8</u>	<u>14.8</u>	2.0
Griewank	**4.69e-3**	4.95e-3	5.24e-3	1.02e+4
Rastrigin	**3686**	4505	<u>4772</u>	815
Rosenbrock	105	**102**	**102**	98
Sphere	<u>5.56e-8</u>	**6.86e-11**	<u>4.52e-10</u>	2.74e-4
Linear	**-2.91e+46**	<u>-1.12e+17</u>	<u>-2.10e+21</u>	-1949

appendix A. The one exception is the `Linear` problem which is only run for 100 generations. We compare our adaptive bandit-based controller against the GESMR, SAMR, and LAMR-100 mutation rate schemes from Kumar et al.[21]. GESMR is the the novel evolutionary mutation rate adaptation proposed in Kumar et al.[21], which coevolves a population of mutation rates alongside the main population. SAMR is a simple self-adaptive mutation rate scheme that attaches mutation rates to individuals and evolves mutation rates and genomes together. The LAMR-100 scheme, which is our "oracle," determines the optimal mutation rate every 100 generations by "looking ahead," running each mutation value in a preset range for 100 generations and choosing the best one. Therefore LAMR-100 is able to directly optimize for future behavior. For these experiments, LAMR-100 searches over a log-spaced range of values from 10^{-3} to 10^{0}. To showcase the sample efficiency of our controller, we let the bandit controller search over a very large log-range of mutation rates from -100 to 100. In addition, we use an amount of sampling noise roughly equal to the meta mutation strength of GESMR. For specific parameter details, see appendix D. All results are averaged over 50 runs.

The results of our experiment is displayed in figure 1. Despite the wide range of mutation standard deviations searched by our controller, we still have high precision and good sample complexity due to our use of tile codings, which efficiently exploit the continuous nature of mutation rates. As we have set $len\_history$ to 100 in this paper, our bandit controller optimizes for more long-term performance than GESMR. This is because, following the trend reported in Kumar et al.[21], we have set the GESMR meta population size to 10. Therefore, the fitness function of a mutation rate is the maximum improvment in fitness of 10 sampled individuals for GESMR, whereas we choose mutations based on the expected maximum over 100 individuals. For this reason, our controller outperforms GESMR which outperforms SAMR on more rugged domains due to their better ability to ignore local minima in favor of global minima. However, since our bandit controller directly estimates this order statistic empirically, it requires more samples to learn and takes more time to adapt the mutation rate. This results in weaker performance on easier problems like Rosenbrock and Sphere.

Since the LAMR-100 oracle optimizes for performance over 100 generations, or 10,000 individuals, it is able to achieve the best performance on the rugged problems Ackley and Rastrigin. However, the limitations of the LAMR-100 method are also clear. Because LAMR-100 only searches over mutation rates from 10^{-3} to 10^{0}, it is unable to perform as well as the other methods on the Linear and Griewank problems, which both require higher mutation rates. In addition, the sampled mutation rates flatten out at later generations on the Ackley, Rastrigin, and Sphere problems at the minimum value available to LAMR-100, 10^{-3}, whereas the optimal mutation rate likely continues decreasing according to the trend seen in earlier generations.

Out of the six problems tested, the Linear problem is unique in its unbounded and smooth fitness function. While GESMR and SAMR will eventually outperform our bandit controller on the Linear problem due to their unbounded growth of mutation rates, our bandit controller is able to learn much more quickly than GESMR or SAMR, essentially achieving its maximum mutation rate within the 5 exploration generations.

4.2 Software Synthesis

For our experiments in software synthesis, we use the propeller implementation of PushGP[1] and tackle several problems from the PSB1 and PSB2 benchmarks. These problems were chosen to represent a range of difficulties and consist of three problems from each benchmark suite.

4.2.1 Fitness Landscape. Compared to the continuous test functions, software synthesis is a much less regular domain. There are fewer beneficial mutations, and the changes in total fitness from parent to child can vary significantly. We therefore expect SAMR to suffer from the vanishing mutation rate problem on this domain. Before conducting problem-solving expeirments, we first validate our choice of a max-value based credit assignment $r_{\max}$ and show that it can help avoid the vanishing mutation rate problem.

In these experiments, we run PushGP on each software synthesis problem with the default UMAD rate of 0.1. We use the default settings[6, 12, 15], with a population size of 1000 and generation limit of 300, and lexicase selection[18] as the selection operator. At each generation, we additionally sample 1000 individuals using each UMAD rate $\mu \in \{0.01, 0.03, 0.1, 0.3, 1\}$. These additional individuals have no impact on the genetic algorithm, and are solely used to evaluate the performance of each mutation rate under our defined reward function. For each individual sampled this way, we compute its immediate reward as described in equation 2. For each UMAD rate, we combine all of these sampled rewards over the entire GP run into a linear array, apply 1-dimensional max pooling with kernel size *len_history* and stride 1 to transform it into the maximum reward $r_{\max}$, and then take an exponentially weighted average with a learning rate of 0.01 to smooth out the plot. We also do

[1]https://github.com/lspector/propeller

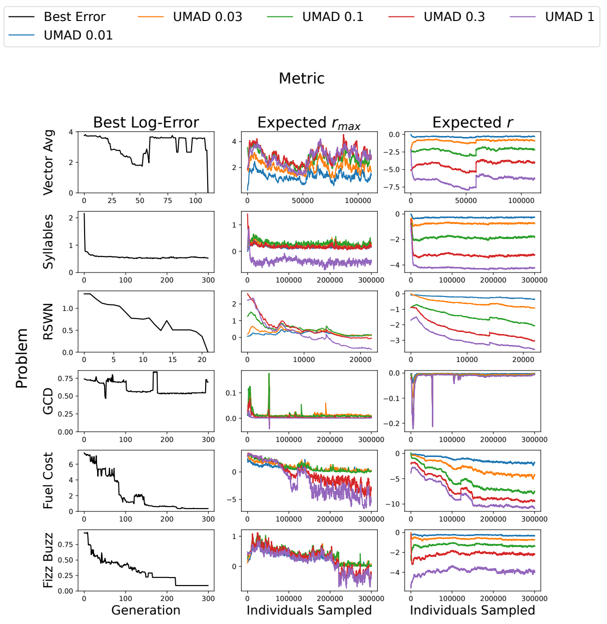

Figure 2: Rewards obtained by different UMAD rates on software synthesis problems. Naively considering only the expected reward makes the lowest UMAD rate appear to be the best. Considering the expected maximum reward over *len_history* samples paints a much more realistic picture.

the same process without max-pooling, essentially plotting the immediate reward r. Finally, we also display the best log-error in the current population at each generation to get an idea of how far evolution has progressed. For all the experiments in this paper, we use *len_history* = 100. All problems were run for 3 runs each, and at each generation the average statistic was taken over all runs which had not yet found a solution.

The results are depicted in figure 2. If we naively consider only the immediate reward, it would appear that on all problems, the lower the mutation rate the better. In fact, these expected rewards are all negative throughout all generations of all problems, so all the mutation rates studied have a worse expected r than zero mutation. Therefore, we expect using the expected immediate reward as a metric to result in mutation rates converging to zero. However, if we instead consider the max-value based reward $r_{\max}$, we see that larger mutation rates are generally better when the best solution found is still poor, but as evolution proceeds and the solutions improve, lower mutation rates become better. We therefore expect the $r_{\max}$ metric to be a much more realistic predictor of mutation rate performance, and we expect that controllers based on maximum value statistics will avoid the vanishing mutation rate problem.

4.2.2 Software Synthesis Performance. As in the previous section, we use the propeller implementation of PushGP and run on the same software synthesis problems with the same settings. In this section, we compare the performance of a fixed UMAD rate of 0.1 with our adaptive bandit controller as well as a self-adaptive mutation rate scheme. Due to computational limitations, instead of

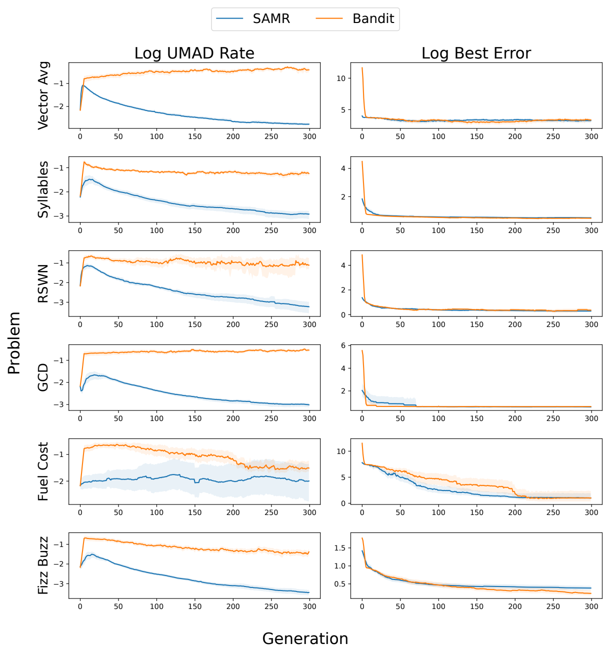

Figure 3: Best error and average log-UMAD rate sampled by three different adaptive controllers on Software Synthesis problems. 95% confidence intervals obtained by bootstrapping. The bandit controller is able to effectively avoid the vanishing mutation rate problem.

Table 2: Number of successes out of 50 runs of fixed and adaptive mutation rate schemes on Software Synthesis problems. The best performance among adaptive results is shown in bold. The performance of the fixed UMAD rate of 0.1 is bolded when best but does not prevent bolding of the adaptive schemes. Statistical significance was calculated using a two-proportion z-test. Results significantly worse than the bandit scheme with $p < 0.05$ are underlined. No results from adaptive schemes were significantly better than the bandit scheme.

Problem	UMAD 0.1	Bandit	SAMR
Vector Average	**45**	**27**	16
Syllables	10	**18**	6
Replace Space With Newline	**47**	**43**	23
GCD	**6**	1	**2**
Fuel Cost	21	**29**	23
Fizz Buzz	8	**12**	1

reporting the number of successes out of 100 as recommended in [12], we report the number of successes out of 50.

The results are shown in table 2. Although the bandit controller is not always able to perform as well as the fixed UMAD rate, it is often able to significantly outperform the SAMR scheme. To see why, we plotted the best log-error and average log-UMAD rate of each adaptive scheme for each problem over 300 generations. As before, we average each statistic at each generation over all the runs which have not yet terminated. The results are shown in figure 3. Although SAMR is able to increase the UMAD rate in the outset of evolution, the UMAD rate quickly decays to very low values, resulting in premature convergence and a low solution rate. On the other hand, our bandit controller is better able to avoid the vanishing mutation rates, chooses more optimal UMAD rates, and achieves better solution rates.

Table 3: Number of successes out of 50 runs of fixed and adaptive mutation rate schemes on Symbolic Regression problems. UMAD 0.1 is bolded when best but does not prevent bolding of the adaptive schemes. Statistical significance was calculated using a two-proportion z-test. Results significantly worse than the bandit scheme with $p < 0.05$ are underlined. No results from adaptive schemes were significantly better than the bandit scheme.

Problem	UMAD 0.1	Bandit	GESMR	SAMR
Nguyen1	**50**	**50**	47	47
Nguyen2	**50**	**50**	48	34
Nguyen3	**48**	**43**	12	9
Nguyen4	34	**43**	14	2
Nguyen5	**50**	**42**	24	23
Nguyen6	**50**	**30**	6	23
Nguyen7	**8**	2	0	**3**
Nguyen8	0	0	0	0

4.3 Symbolic Regression Performance

As in the previous section, we use the propeller implementation of PushGP. In this experiment, we run on the eight 1-dimensional synthetic regression problems from Nguyen et al.[35]. The target functions are detailed in appendix B. As the details of our experiment differ from those in the literature, our results only serve to draw comparisons between the adaptive mutation rate schemes studied here. We define a successful function as one which completely replicates the given input-output pairs, to within some small constant. In addition, we use an instruction set composed of the input instruction `input`, the constant `1.0`, basic arithmetic operations (+, −, ×, ÷), and the additional functions (`Sin`, `Cos`, `Log`). We protect `Log` and ÷ to return 0.0 for undefined inputs, and clamp all numbers in the execution of our program to the interval $[-1.0\times10^6, 1.0\times10^6]$. For our test cases on problems `Nguyen1` through `Nguyen6`, we use a range of evenly spaced inputs from -4 to 4 with a step size of 0.1. For `Nguyen7` and `Nguyen8`, to avoid undefined regions of the domain, we shift the inputs to range from 0 to 8. We find that the larger range of inputs gives more information about the shape of the function, leading to higher quality runs. We use the epsilon-lexicase selection method [22] and run PushGP with a population size of 1000 for 300 generations. We report our results as the number of successful runs out of 50 total.

The results are displayed in table 3. The bandit based scheme consistently outperforms the SAMR and GESMR schemes on the symbolic regression problems, although it is not always able to perform as well as the preset UMAD rate. The best log-error and

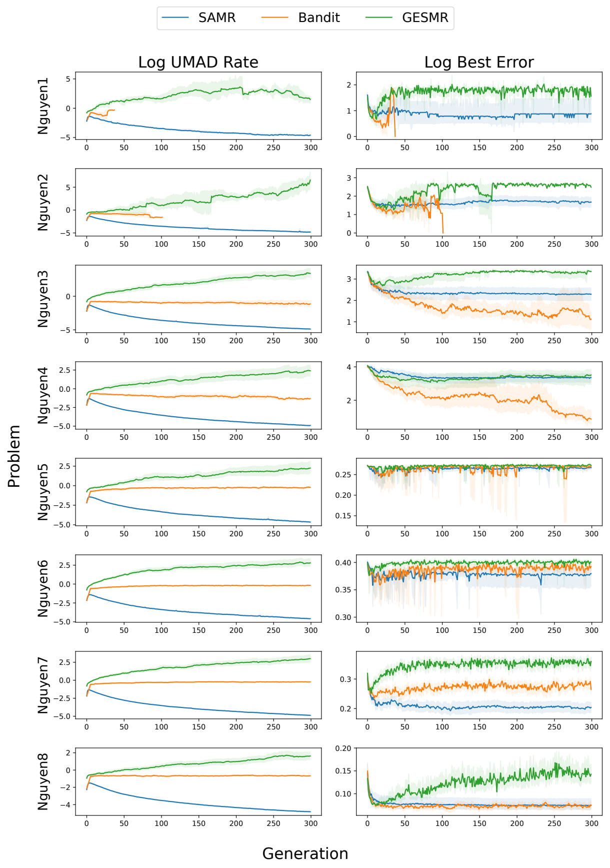

Figure 4: Best error and average log-UMAD rate sampled by three different adaptive controllers on SR problems. 95% confidence intervals obtained by bootstrapping. SAMR suffers from a vanishing mutation rate, while GESMR suffers from an exploding mutation rate.

average log-UMAD rate sampled by these three adaptive methods are displayed in figure 4. As before, we average each statistic at each generation over all the runs that have not yet terminated. Therefore, for problems like Nguyen1 and Nguyen2 where the bandit method always terminates within 50 or 100 generations, there is no data for later generations. In all problems, the SAMR method results in continually decreasing mutation rates which often results in premature convergence, such as in the Nguyen3 and Nguyen4 problems. On the other hand, GESMR samples ever increasing UMAD rates. We hypothesize that this is due to the low degree of differentiation between large UMAD rates. As the UMAD rate tends towards infinity, UMAD mutation does not tend towards infinitely large mutations as with gaussian mutation, but rather towards completely random mutation. Therefore, the difference between mutation rates is much less pronounced, especially for large mutation rates, in the symbolic regression domain than it is in the function minimization domain. As GESMR only utilizes local information about mutation rates, we hypothesize that it gets stuck in this uninformative region of high mutation rates and is unable to realize when significant progress has been made and large mutation rates are no longer optimal. In contrast, the bandit controller is able to escape these pathologies and robustly determine a good, moderate UMAD rate at each generation.

5 CONCLUSION AND FUTURE WORK

We have presented an adaptive scheme for controlling the rate of UMAD mutation in genetic programming problems. The proposed method uses multi-armed bandits to optimize the mutation rate, tile codings to improve generalization, and optimizes for the maximum improvement over many mutations, which is a more accurate predictor of the usefulness of a given mutation rate.

In the future, we could extend our adaptive controller to other mutation and selection operators in addition to the UMAD rate. For example, we could learn various combinations of tree-based mutation operators at various strengths for tree-based GP systems[28], or various bit mutation operators for systems with binary representations[9]. One attractive domain for adaptive optimization is the genetic source[16]. An effective adaptive scheme for genetic source optimization not only has the potential for greatly improved solution rates[16], but also can relieve the end user of the need to specify which datatypes and instructions are needed for each individual problem.

Beyond genetic programming, this system could also be used to adapt hyperparameters such as learning rate or weight decay over the course of training a neural network. To do this, we frame the task of choosing hyperparameters as the maximization of the expected improvement in the training loss, which can be estimated by the improvement in training loss from the current training batch to the next. While we could also use the expected improvement of the validation loss, this formulation does not require any additional forward passes, and has almost no overhead cost. Since we are interested in the continuous improvement of a single model in this domain, we would maximize the expected reward, and not the expected maximum reward over *len_history* samples as we have done in the evolutionary domains.

However, the scope of possible applications of our controller is limited by the curse of dimensionality, as the number of tiles to be evaluated grows exponentially with the number of parameters to be optimized. One simple fix would be to keep a separate bandit controller for each hyperparameter. However, this assumes that the hyperparameters can be optimized independently, ignoring the possible interplay between hyperparameters. In the future, we could extend our adaptive scheme to higher dimensions, such as by replacing the tile coding with a neural network.

ACKNOWLEDGMENTS

This work was performed in part using high-performance computing equipment at Amherst College obtained under National Science Foundation Grant No. 2117377. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation. The authors would like to thank Ryan Boldi, Bill Tozier, Tom Helmuth, Edward Pantridge and other members of the PUSH lab for their insightful comments and suggestions.

REFERENCES

[1] Aldeida Aleti and Irene Moser. 2016. A Systematic Literature Review of Adaptive Parameter Control Methods for Evolutionary Algorithms. *ACM Comput. Surv.* 49, 3, Article 56 (Oct 2016), 35 pages. https://doi.org/10.1145/2996355

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47, 2 (01 May 2002), 235–256. https://doi.org/10.1023/A:1013689704352

[3] Thomas Bäck et al. 1992. Self-adaptation in genetic algorithms. In *Proceedings of the first european conference on artificial life.* MIT press Cambridge, 263–271.

[4] Thomas Bäck and Martin Schütz. 1996. Intelligent mutation rate control in canonical genetic algorithms. In *Foundations of Intelligent Systems*, Zbigniew W. Raś and Maciek Michalewicz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 158–167.

[5] Donald A Berry and Bert Fristedt. 1995. *Bandit problems: sequential allocation of experiments (Monographs on statistics and applied probability).* Springer.

[6] Ryan Boldi, Martin Briesch, Dominik Sobania, Alexander Lalejini, Thomas Helmuth, Franz Rothlauf, Charles Ofria, and Lee Spector. 2024. Informed Down-Sampled Lexicase Selection: Identifying Productive Training Cases for Efficient Problem Solving. *Evolutionary Computation* (03 2024), 1–31. https://doi.org/10.1162/evco_a_00346

[7] Jeff Clune, Dusan Misevic, Charles Ofria, Richard E Lenski, Santiago F Elena, and Rafael Sanjuán. 2008. Natural selection fails to optimize mutation rates for long-term adaptation on rugged fitness landscapes. *PLoS Computational Biology* 4, 9 (2008), e1000187.

[8] Benjamin Doerr, Carsten Witt, and Jing Yang. 2018. Runtime Analysis for Self-Adaptive Mutation Rates. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Kyoto, Japan) *(GECCO'18)*. Association for Computing Machinery, New York, NY, USA, 1475–1482. https://doi.org/10.1145/3205455.3205569

[9] Álvaro Fialho, Luís Da Costa, Marc Schoenauer, and Michèle Sebag. 2008. Extreme Value Based Adaptive Operator Selection. In *Parallel Problem Solving from Nature – PPSN X*, Günter Rudolph, Thomas Jansen, Nicola Beume, Simon Lucas, and Carlo Poloni (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 175–184.

[10] M.R. Glickman and K. Sycara. 2000. Reasons for premature convergence of self-adapting mutation rates. In *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No.00TH8512)*, Vol. 1. 62–69. https://doi.org/10.1109/CEC.2000.870276

[11] Brian W. Goldman and Daniel R. Tauritz. 2011. Meta-evolved empirical evidence of the effectiveness of dynamic parameters. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation* (Dublin, Ireland) *(GECCO'11)*. Association for Computing Machinery, New York, NY, USA, 155–156. https://doi.org/10.1145/2001858.2001945

[12] Thomas Helmuth and Peter Kelly. 2021. PSB2: the second program synthesis benchmark suite. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Lille, France) *(GECCO'21)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/3449639.3459285

[13] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2018. Program synthesis using uniform mutation by addition and deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Kyoto, Japan) *(GECCO'18)*. Association for Computing Machinery, New York, NY, USA, 1127–1134. https://doi.org/10.1145/3205455.3205603

[14] Thomas Helmuth, Edward Pantridge, and Lee Spector. 2019. Lexicase Selection of Specialists. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Prague, Czech Republic) *(GECCO'19)*. Association for Computing Machinery, New York, NY, USA, 1030–1038. https://doi.org/10.1145/3321707.3321875

[15] Thomas Helmuth, Edward Pantridge, and Lee Spector. 2020. On the importance of specialists for lexicase selection. *Genetic Programming and Evolvable Machines* 21, 3 (01 Sep 2020), 349–373. https://doi.org/10.1007/s10710-020-09377-2

[16] Thomas Helmuth, Edward Pantridge, Grace Woolson, and Lee Spector. 2020. Genetic Source Sensitivity and Transfer Learning in Genetic Programming. In *Proceedings of ALIFE 2020: The 2020 Conference on Artificial Life.* 303–311. https://doi.org/10.1162/isal_a_00326

[17] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation* (Madrid, Spain) *(GECCO'15)*. Association for Computing Machinery, New York, NY, USA, 1039–1046. https://doi.org/10.1145/2739480.2754769

[18] Thomas Helmuth, Lee Spector, and James Matheson. 2015. Solving Uncompromising Problems With Lexicase Selection. *IEEE Transactions on Evolutionary Computation* 19, 5 (2015), 630–643. https://doi.org/10.1109/TEVC.2014.2362729

[19] Johannes W. Kruisselbrink, Rui Li, Edgar Reehuis, Jeroen Eggermont, and Thomas Bäck. 2011. On the log-normal self-adaptation of the mutation rate in binary search spaces. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation* (Dublin, Ireland) *(GECCO'11)*. Association for Computing Machinery, New York, NY, USA, 893–900. https://doi.org/10.1145/2001576.2001699

[20] Volodymyr Kuleshov and Doina Precup. 2014. Algorithms for multi-armed bandit problems. arXiv:1402.6028 [cs.AI]

[21] Akarsh Kumar, Bo Liu, Risto Miikkulainen, and Peter Stone. 2022. Effective mutation rate adaptation through group elite selection. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Boston, Massachusetts) *(GECCO'22)*. Association for Computing Machinery, New York, NY, USA, 721–729. https://doi.org/10.1145/3512290.3528706

[22] William La Cava, Lee Spector, and Kourosh Danai. 2016. Epsilon-Lexicase Selection for Regression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016* (Denver, Colorado, USA) *(GECCO'16)*. Association for Computing Machinery, New York, NY, USA, 741–748. https://doi.org/10.1145/2908812.2908898

[23] Katherine M. Malan and Andries P. Engelbrecht. 2009. Quantifying ruggedness of continuous landscapes using entropy. In *2009 IEEE Congress on Evolutionary Computation.* 1440–1447. https://doi.org/10.1109/CEC.2009.4983112

[24] Stefano Marsili Libelli and P Alba. 2000. Adaptive mutation in genetic algorithms. *Soft computing* 4 (2000), 76–80.

[25] Michael Kurtis Maschek. 2010. Intelligent Mutation Rate Control in an Economic Application of Genetic Algorithms. *Computational Economics* 35, 1 (01 Jan 2010), 25–49. https://doi.org/10.1007/s10614-009-9190-6

[26] David Metzgar and Christopher Wills. 2000. Evidence for the adaptive evolution of mutation rates. *Cell* 101, 6 (2000), 581–584.

[27] Silja Meyer-Nieberg and Hans-Georg Beyer. 2007. Self-adaptation in evolutionary algorithms. In *Parameter setting in evolutionary algorithms.* Springer, 47–75.

[28] Ricardo Poli, William B. Langdon, and Nicholas F. McPhee. 2008. *A Field Guide to Genetic Programming.* Lulu Enterprises.

[29] Lee Spector, Chris Perry, Jon Klein, and Maarten Keijzer. 2004. *Push 3.0 programming language description.* Retrieved Jan 30, 2024 from https://faculty.hampshire.edu/lspector/temp/HC-CSTR-2004-02.pdf

[30] Lee Spector and Alan Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines* 3, 1 (01 Mar 2002), 7–40. https://doi.org/10.1023/A:1014538503543

[31] C. R. Stephens, I. García Olmedo, J. Mora Vargas, and H. Waelbroeck. 1998. Self-Adaptation in Evolving Systems. *Artificial Life* 4, 2 (1998), 183–201. https://doi.org/10.1162/106454698568512

[32] Sonja Surjanovic and Derek Bingham. 2013. *Optimization Test Functions and Datasets.* Retrieved Jan 28, 2024 from https://www.sfu.ca/~ssurjano/optimization.html

[33] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. 2013. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 28)*, Sanjoy Dasgupta and David McAllester (Eds.). PMLR, Atlanta, Georgia, USA, 1139–1147. https://proceedings.mlr.press/v28/sutskever13.html

[34] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction.* MIT press.

[35] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, R. I. McKay, and Edgar Galván-López. 2011. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines* 12, 2 (01 Jun 2011), 91–119. https://doi.org/10.1007/s10710-010-9121-2

[36] James M. Whitacre, Tuan Q. Pham, and Ruhul A. Sarker. 2006. Use of statistical outlier detection method in adaptive evolutionary algorithms. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation* (Seattle, Washington, USA) *(GECCO'06)*. Association for Computing Machinery, New York, NY, USA, 1345–1352. https://doi.org/10.1145/1143997.1144205

[37] David R. White, James McDermott, Mauro Castelli, Luca Manzoni, Brian W. Goldman, Gabriel Kronberger, Wojciech Jaśkowski, Una-May O'Reilly, and Sean Luke. 2013. Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines* 14, 1 (01 Mar 2013), 3–29. https://doi.org/10.1007/s10710-012-9177-2

A TEST FUNCTION DEFINITIONS

In this section we describe the test functions used in our function minimization experiment as well as the standard deviation used to initialize our population. Aside from the Linear function, which is unbounded, and the Rosenbrock function, which achieves its minimum at $\mathbf{x} = \mathbf{1}$, all test functions are constructed to have a global minimum of 0 at $\mathbf{x} = \mathbf{0}$. In these definitions, d is the dimension of the vector $\mathbf{x}$ and the x_i are all 1-indexed.