Program Synthesis using Uniform Mutation by Addition and Deletion

Thomas Helmuth
Hamilton College
Clinton, New York, USA
thelmuth@hamilton.edu

Nicholas Freitag McPhee
University of Minnesota, Morris
Morris, Minnesota, USA
mcphee@morris.umn.edu

Lee Spector
Hampshire College
Amherst, Massachusetts, USA
lspector@hampshire.edu

ABSTRACT

Most genetic programming systems use mutation and crossover operators to create child programs from selected parent programs. Typically, the mutation operator will replace a randomly chosen subprogram in the parent with a new, randomly generated subprogram. In systems with linear genomes, a uniform mutation operator can be used that has some probability of replacing any particular gene with a new, randomly chosen gene. In this paper, we present a new uniform mutation operator called Uniform Mutation by Addition and Deletion (UMAD), which first adds genes with some probability before or after every existing gene, and then deletes random genes from the resulting genome. In UMAD it is not necessary that the new genes *replace* old genes, as the additions and deletions can occur in different locations. We find that UMAD, with relatively high rates of addition and deletion, results in significant increases in problem-solving performance on a range of program synthesis benchmark problems. In our experiments, we compare this method to a variety of alternatives, showing that it equals or outperforms all of them. We explore this new mutation operator and other well-performing high-rate mutation schemes to determine what traits are crucial to improved performance.

CCS CONCEPTS

• **Computing methodologies → Genetic programming**;

KEYWORDS

mutation, genetic operators, Push, program synthesis

ACM Reference Format:
Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2018. Program Synthesis using Uniform Mutation by Addition and Deletion. In *GECCO '18: Genetic and Evolutionary Computation Conference, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205455.3205603

1 INTRODUCTION

Genetic programming (GP) is a problem solving tool that begins with a population of random programs, and produces a succession of populations, each derived from previous, with the aim of finding a program that solves a given target problem. Each new program is produced by passing one or more selected parents to *genetic operators* that construct children, incorporating material from the parents while also introducing variation. The traditional approach produces some children for each population with a *mutation* operator that uses material only from a single parent, and others with a *recombination* or *crossover* operator that uses material from two parents. The traditional mutation operator replaces a randomly chosen subprogram of the single parent program with a new, randomly generated subprogram. The traditional crossover operator does something similar but obtains the replacement subprogram from a second parent, rather than generating it randomly. Many variants of these methods have been explored in the literature; for example, see the "Mutation Cookbook" section of [14].

Here we describe a novel single-parent mutation operator, which we demonstrate in the context of genetic programming on benchmark software synthesis problems [6]. We show that by using this novel mutation operator, without any form of crossover, we achieve significant and general improvements in program synthesis, improving the state of the art over all previous known approaches.

This new operator, which we call Uniform Mutation by Addition and Deletion (UMAD), stems from the idea that mutations based on *replacement*, such as the traditional mutation operator, are overly constrained. They delete parts of a parent program and add new parts, but the additions are always in the same places as the deletions. With UMAD we dissociate the additions from the deletions, so that we can get one without the other and/or both but in different parts of the parent program. Intriguingly, the high levels of problem-solving performance that we demonstrate with UMAD—the highest ever demonstrated for the general program synthesis benchmarks considered here—are achieved without recombination.

One explanation for improved performance with dissociated additions and deletions, relative to replacements, is that the former can provide more paths through the search space from a parent to a descendant than can the latter. For example, if we have a parent `ABC` and we seek a child `ADC`, in a context in which the loss of either `A` or `C` would be fatal, then there is only one path to `ADC` by replacement: replacing `B` with `D`. With dissociated additions and deletions, it would still be possible to go directly from `ABC` to `ADC` but there would also be additional paths, including paths through `AC`, `ABDC`, and `ADBC`. Depending on the details of the program representation and the problem's fitness function, some of these alternative paths might either be more likely to arise randomly or more likely to be favored, at each step, by selection.

Here we study UMAD in the context of the PushGP genetic programming system [15, 16], recent versions of which generate

GECCO '18, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5618-3/18/07…$15.00
https://doi.org/10.1145/3205455.3205603

and manipulate programs in the form of linear genomes [8]. It should be directly applicable, however, to any genetic programming system that uses linear program representations, and the general idea may be applicable to non-linear representations as well.

Among our findings is that UMAD works well with mutation rates that are considerably higher than those generally used in the field. For this reason, we also present the results of experiments using replacement-based mutation with the same high rates; we find that this also works surprisingly well, but not as well as UMAD.

Because gene addition and deletion are dissociated in UMAD, the rates of addition and deletion can be set independently. While some settings are size neutral on average, others produce a bias toward smaller or larger programs. In the experiments described here, we explore the effects of these settings.

In the following section we describe the PushGP system in which our experiments are conducted, along with the genetic operators against which we will later compare UMAD. We then describe UMAD itself, our experimental design, and the results of our experiments. After discussing results we describe related work and draw general conclusions, including suggestions for future research.

2 PUSHGP

PushGP is a GP system that evolves Push programs [15, 16]. Push is a stack-based language that uses a separate stack for each data type, including a stack for program code that remains to be executed. These features facilitate the evolution of programs that involve multiple data types and non-trivial control structures, including for general program synthesis benchmark problems [6].

Although Push programs can have a nested, hierarchical structure, recent versions of PushGP represent the programs in their populations as linear sequences of genes. These sequences, which are called "Plush genomes" (the "l" is for "linear"), are translated into Push programs prior to execution [8]. Each gene in a Plush genome specifies an instruction or a constant, and when certain instructions (such as conditional or looping instructions) are processed for translation to a Push program, one or more code blocks may be opened, resulting in a hierarchical Push program. Genes in a Plush genome also include epigenetic `close` markers that specify where these code blocks end. We initialize random Plush genomes by simply creating sequences of random genes, with each gene having a random instruction and `close` marker.

When using linear Plush genomes, we can vary individuals using methods similar to those developed for genetic algorithms that operate on linear genomes. The primary genetic operators used in prior work are uniform mutation, alternation and uniform close mutation. In uniform mutation, each gene in a single parent has a specified probability of being replaced by a new, randomly generated gene in the child program that is otherwise copied directly from the parent. Note that the child will always be the same size as its parent. In alternation, a crossover operator, two selected parents serve as source material for the child; initially, genes are copied from one of the parents, but copying alternates between the two parents with some probability at each step as it proceeds. Furthermore, the index from which genes are being copied may randomly deviate either forward or backward when switching between parents. In uniform close mutation, each epigenetic `close` marker has some probability of being incremented or decremented. These are described in more detail in [8].

Table 1: Genetic operator parameter settings and the rates at which we use them in our *benchmark genetic operators*.

Parameter	Value
alternation rate	0.01
alignment deviation	10
uniform mutation rate	0.01
uniform close mutation rate	0.1
Genetic Operator Rates	**Prob**
alternation	0.2
uniform mutation	0.2
uniform close mutation	0.1
alternation followed by uniform mutation	0.5

Our experiments refer to a genetic operator setting used in previous papers on the problems studied here [6]; we call this setting the *benchmark genetic operators* (see Table 1), and we note that experiments have shown that 100% uniform mutation, with a rate of 0.01, can be almost equally effective [8].

3 UNIFORM MUTATION BY ADDITION AND DELETION

Uniform Mutation by Addition and Deletion (UMAD) could be applied to any variable-length linear genome representation in evolutionary computing. For each gene in a parent's genome, UMAD considers whether to add a new random gene before or after it with probability given by the parameter *addition rate*. Then, in this augmented genome, UMAD considers whether to delete each gene with probability given by the parameter *deletion rate*. This process is formalized in the Algorithm 1. UMAD is uniform in that every gene in the parent has equal chance of having a new gene added before or after it, and also has equal chance of being deleted. During the addition step of UMAD, we generate new genes using the same process used to generate random genomes, as described above.

The addition and deletion rate parameters affect not only how much of the parent is changed in the child, but also whether this mutation will, on average, increase size, decrease size, or remain size-neutral. For a particular addition rate, using the same deletion rate will actually result in smaller child programs on average. In general, for UMAD to be size-neutral, the deletion rate needs to be set using the formula:

$$\text{deletionRate} = \frac{\text{additionRate}}{1 + \text{additionRate}} \tag{1}$$

For example, in the extreme case of an addition rate of 1.0, after the addition step the genome will be twice as large as the parent; in order to reduce it to the parent's size, the deletion rate should be 0.5. In some of our experiments we test UMAD variants that are not size-neutral, and therefore grow or shrink children on average, by varying the deletion rate while holding the addition rate constant.

Algorithm 1: Uniform Mutation by Addition and Deletion

```
Input: genome, additionRate, deletionRate
// Additions step
for gene ∈ genome do
    if rand() < additionRate then
        newGene ← randomGene() ;
        if rand() < 0.5 then
            Add newGene to genome before gene;
        else
            Add newGene to genome after gene;
        end
    end
end
// Deletions step
for gene ∈ genome do
    if rand() < deletionRate then
        Delete gene from genome;
    end
end
return genome
```

4 EXPERIMENTAL DESIGN

In our experiments we explore a variety of factors related to UMAD. Some of these experiments test variants of UMAD with different parameters or combined with other genetic operators. Others try to understand why UMAD works so well compared to our previously-used benchmark genetic operators, and ask questions about whether a large population is necessary even without crossover to recombine individuals.

To explore these questions, we focus on general program synthesis problems, where the aim is to generate programs sharing characteristics with programs that humans write. In particular, the problems tested here require programs to utilize a variety of data types and control flow structures. These experiments use problems from a recent program synthesis benchmark suite, composed of 29 problems from introductory computer science textbooks [6].

We use various subsets of this benchmark suite for different experiments, using more problems for more important experiments or when differences in performance are not obvious from a few problems. In our main comparisons, we use 25 of the problems, only leaving off 4 problems that have never been previously solved. Every experiment includes, at minimum, the three problems Replace Space With Newline, Syllables, and Vector Average, as these are problems that have previously shown varying levels of difficulty without being either trivial or impossible. To give some idea of the types of problems in the suite, we briefly describe these three problems; more details and the remaining problems are described in the benchmark suite [6]:

- Replace Space With Newline (RSWN): Given a string input, the program must do two things: it must print the input string after replacing each space character with a newline character, and it must functionally return the number of non-whitespace characters in the string.
- Syllables: Given a string input, the program must print `"The number of syllables is X"`, where `X` is the number of vowels in the string.
- Vector Average: Given a vector of floats, functionally return the average of those floats.

For our experiments, we used Clojush,[1] an implementation of the PushGP system written in Clojure. For each PushGP run we used a population size of 1000 and at most 300 generations, except for the Median, Number IO, and Smallest problems, which used at most 200 generations. The maximum Plush genome size varies per problem between 200 and 1000 instructions, as given with the original benchmark suite [6]. The experiments here used lexicase selection for parent selection [7], which has shown to produce significantly better results than tools like tournament selection on these problems [6].

In our experiments, we measure problem solving performance as the number of GP runs out of 100 that solve the problem, which we call the *success rate*. In particular, we only count programs that return the correct answer on test cases used for training as well as on a withheld set of unseen data, to ensure that solutions generalize instead of simply memorizing answers [6]. Before testing for generalization, we automatically simplify the evolved programs that pass all of the training cases to create smaller programs with identical outputs on every training case; this was recently shown to improve generalization on the problems in the benchmark suite used here [3]. To test for significant differences in success rates, we use a pairwise chi-square test with Holm correction and a 0.05 significance level. These tests are applied independently on each row of every results table in Section 5; no comparisons or corrections were performed across rows or across tables.

5 RESULTS

We designed a variety of experiments to learn more about UMAD and compare it to related genetic operators. While we performed numerous comparisons, there are many more that we could have performed. For some experiments we limited the number of problems tested once initial results made outcomes clear.

5.1 Initial Comparisons to UMAD

Table 2 presents our first set of results, comparing

- Three variants of UMAD (see below)
- HAHD (Half Addition, Half Deletion), which half of the time uses addition alone to generate a child genome, and half of the time uses deletion alone.
- Uniform mutation like that used in earlier work (e.g, [6]), but with a substantially higher mutation rate than that used before (0.09 instead of 0.01).
- Benchmark genetic operators described in Section 2.

The three UMAD variants each use an addition rate of 0.09, but differ in their deletion rates:

- UMAD (or size-neutral UMAD) uses a deletion rate of 0.0826, computed using Eq. 1 so that on average, a child genome will have the same size as its parent genome.

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

Problem	UMAD	UMAD↓	UMAD↑	HAHD	Uni. mut.	Benchmark
Checksum	1	5	0	4	2	1
Compare String Lengths	**32**	**42**	**26**	**22**	**28**	5
Count Odds	8	12	6	10	7	5
Digits	19	11	10	15	14	10
Double Letters	**19**	**20**	5	**18**	**16**	1
Even Squares	0	0	0	0	0	0
For Loop Index	2	1	0	1	1	0
Grade	0	0	0	0	0	1
Last Index of Zero	**62**	**56**	**62**	49	**72**	29
Median	55	48	66	60	55	54
Mirror Image	**100**	**100**	**100**	**100**	**100**	87
Negative To Zero	80	**82**	73	72	79	62
Number IO	98	100	100	100	100	100
Pig Latin	0	0	0	0	0	0
Replace Space With Newline	**87**	**87**	**88**	**83**	**92**	58
Scrabble Score	13	**20**	3	11	10	4
Small or Large	7	4	9	4	6	5
Smallest	100	100	98	99	86	97
String Lengths Backwards	**94**	86	80	82	62	74
Sum of Squares	**21**	**26**	13	10	13	3
Super Anagrams	4	0	2	3	1	0
Syllables	38	**48**	21	**51**	28	24
Vector Average	**88**	**92**	**84**	**77**	**82**	43
Vectors Summed	11	9	3	9	7	1
X-Word Lines	**61**	**59**	12	**63**	25	18
# Sig. Better than Bench.	9	11	5	6	6	-

Table 2: Number of successes out of 100 independent runs for three variations of UMAD, half addition and half deletion (HAHD), high-rate uniform mutation (“Uni. mut.”), and the benchmark genetic operations on a range of software synthesis problems. “UMAD” is size neutral UMAD, shrinking UMAD (i.e. “UMAD↓”) is UMAD with a bias toward smaller genomes, and growing UMAD (i.e. “UMAD↑”) is UMAD with a bias toward larger genomes. Bold results are significantly better than Benchmark using a pairwise chi-squared test (see Section 4 for details). No method was ever significantly worse than Benchmark. Underlined results are significantly *worse* than UMAD; no results were significantly better.

- Shrinking UMAD (“UMAD↓”) uses a deletion rate of 0.1. This leads to a bias towards child genomes that are, on average, smaller than their parent genomes.
- Growing UMAD (“UMAD↑”) uses a deletion rate of only 0.0652, causing child genomes to, on average, be larger than their parent genomes.

We discuss the effects on genome sizes in Section 5.5.

Where UMAD performs a combination of additions and deletions as a single step to generate a child genome, HAHD does either addition or deletion to create a child, but not both. Each new individual has a 50% chance of being constructed via addition, and a 50% chance of being constructed via deletion. We compare UMAD with HAHD to determine whether it is important for UMAD to do additions and deletions in the same step. Over time HAHD can in principle accomplish the same changes as UMAD, but it is possible that the intermediate individuals that HAHD adds to the population might be selected either for or against in ways that would affect HAHD’s efficacy. The version of HAHD used here uses the same addition and deletion rates as UMAD.

The uniform mutation operator used in earlier work [6] replaces genes in the parent with probability set by the mutation rate. Uniform mutation is similar to UMAD, except that a gene is always replaced by a gene *in the same location*, where with UMAD insertions and deletions are not correlated in position or number of changes. With our standard rates for addition and deletion, UMAD makes considerably more changes to a genome than the uniform mutation operator used in earlier work, where the default mutation rate was 0.01, i.e., one gene in a hundred would be replaced with a randomly generated gene. Of note, a previous experiment showed that only using uniform mutation with rate of 0.01 was approximately as good as using our benchmark genetic operators [8]. To gauge the importance of UMAD’s ability to add genes in one location and remove them in another, we compare it to uniform mutation with a much higher mutation rate (0.09) than has been used in the past in the Uni. Mut. column of Table 2.

Table 2 shows the number of successes out of 100 independent runs for each of the three variants of UMAD, HAHD, high-rate uniform mutation, and the benchmark set of genetic operators. Here we see that the all of the purely mutational genetic operators outperform the benchmark set on almost every problem, frequently by wide margins. In quite a few cases, for example, one or more of the mutation-only approaches leads to over twice the number of successes achieved with the benchmark set of genetic operators.

Of the various operators in Table 2, size-neutral UMAD and shrinking UMAD consistently have the best success rates, with one ranked first and the other ranked second for most of the test problems. The differences in success rates between UMAD and shrinking UMAD are typically quite small, and one does not appear to have a meaningful advantage over the other, with no differences ever being significant. Both UMAD and shrinking UMAD are significantly better than the benchmark operators on more than one third of the problems here, where the other three operator settings are not.

Growing UMAD was typically comparable to or better than the benchmark operator results, but was less successful than UMAD, and was significantly worse on two problems. These results likely reflect issues related to code growth, as we discuss when examining average program sizes in Section 5.5.

HAHD appears somewhat weaker than UMAD and shrinking UMAD, having fewer instances of significant improvement over the benchmark operator. However, it was never significantly worse than UMAD.

In general, UMAD outperformed high-rate uniform mutation, including by a significant amount on three problems. Additionally, uniform mutation was significantly better than the benchmark operators on only six problems, three fewer than UMAD.

To our knowledge, this is the first time that GP has created generalizing solutions to the Super Anagrams problem. We have previously seen solutions that do not generalize, but never generalizing ones. Thus it is quite interesting that four different mutation-based methods found such solutions.

Overall, the results in Table 2 suggest that size-neutral UMAD and shrinking UMAD are generally similar to each other in performance, and generally better than any of the other options. Both HAHD and uniform mutation with a mutation rate of 0.09 also generally outperformed the benchmark set of genetic operators. It is important to note, however, that all of these approaches depend on particular parameter values, none of which were tuned extensively. These experiments, for example, only looked at one particular addition rate (0.09) for all three UMAD variants, and one mutation rate (0.09) for uniform mutation. It is thus certainly possible that there are other parameter settings that could further improve the performance of some of these approaches.

5.2 UMAD Addition and Deletion Rates

While size-neutral UMAD, with addition rate 0.09 and deletion rate 0.0826, performs significantly better than our benchmark genetic operators, it was not clear whether these mostly arbitrary settings for addition and deletion rates were good choices. To explore whether higher or lower rates would produce even better results, we conducted sets of runs with addition rates both halved and doubled compared to our standard choices, with deletion rates computed using Eq. 1. Table 3 gives the results for these runs.

While for the most part the results with both higher and lower rates of addition and deletion resulted in similar success rates, those with higher rates were a bit worse. With addition rate of 0.18 and deletion rate of 0.1525, UMAD produced significantly worse results on the Syllables and X-Word Lines problems. On the other hand, with addition rate of 0.045 and deletion rate of 0.0431, UMAD performed similarly to standard UMAD on all problems, and only significantly worse than UMAD with addition rate of 0.18 on one problem. These results suggest that UMAD is robust to large changes in the addition and deletion rates, since a 4-times difference in rates produced very similar performance.

Problem	**0.045**	**0.09**	**0.18**
Double Letters	16	19	20
Mirror Image	96	100	100
Replace Space With Newline	84	87	93
Sum of Squares	9	21	16
Syllables	50	38	13
Vector Average	*75	88	91
X-Word Lines	56	61	17

Table 3: Number of success out of 100 runs for size-neutral UMAD with different addition rates (given in each column). The corresponding deletion rates for each set of runs can be calculated using Eq. 1. The underlined results are significantly *worse* than the other results in their rows. The Vector Average result of 75 for 0.045 is significantly worse than the result of 91 for 0.18, but not significantly different from the result of 88 for 0.09.

5.3 Population Size

An important question raised by the use of a mutation-only system is what role (if any) do populations play? Some sort of population is necessary when using recombination to provide a pool from which to draw pairs of parents. There is also evidence that populations provide a diverse pool of individuals for evolutionary search to act on [4, 5]. When using just a single mutation operator like UMAD, it is less clear how important populations are to the process; in principle there’s no reason UMAD could not be used in a single-individual system such as a (1 + 1)-ES [2].

To help assess the importance of populations when using UMAD alone, we conducted runs with population sizes of 10, 100, 1,000, and 10,000; these results are summarized in Table 4. In each case we adjusted the number of generations so that the total number of individuals processed remained constant. For example, since our runs with population size of 1,000 were run for at most 300 generations, the runs with population size 100 had a maximum of 3,000 generations.

Table 4 indicates there is some flexibility in population size; for Replace Space With Newline, results are very similar with population sizes across three orders of magnitude (100, 1,000, and 10,000). Both Vector Average and Syllables show a drop in successes when using a population size of 10,000; it is possible here that the real issue is only having 30 generations, and that these problems often require more than 30 “evolutionary steps” to reach a solution.

Problem	10	100	1,000	10,000
Replace Space With Newline	7	**95**	**87**	**93**
Syllables	0	2	**38**	**31**
Vector Average	2	**89**	**88**	74

Table 4: Number of successes out of 100 runs for four different population sizes using UMAD on three problems. In each case the total number of generations was adjusted so that the total number of individuals processed remained constant across each treatment. Bold success rates are significantly better than non-bolded rates in the same row. For Vector Average, the success rate with population size 10,000 is also significantly better than with population size 10.

Problem	UMAD	UMAD + Alt
Replace Space With Newline	**87**	62
Syllables	38	40
Vector Average	**88**	74

Table 5: Successes out of 100 run for UMAD and a 50/50 mix of UMAD and alternation. UMAD is significantly better on the two problems with bold success rates.

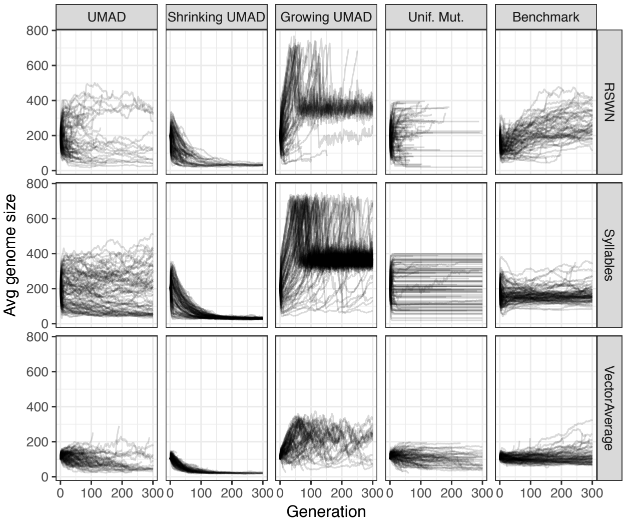

Figure 1: Average genome sizes over time for Replace Space with Newline (RSWN), Syllables, and Vector Average for UMAD, shrinking UMAD, growing UMAD, uniform mutation, and the benchmark genetic operators.

All three problems have very poor success rates with a population size of 10; this might suggest that they need a larger population to explore a variety of different partial solutions to the target problem. With only 10 individuals, the cost of even a few "fatal" mutations is very high (and such mutations occur quite regularly), and it would be easy for one or two strong individuals to swamp the entire population with their progeny, eliminating necessary diversity. Additionally, this poor performance may be related to our use of lexicase selection, which focuses on different test cases during each selection This emphasis on different cases may necessitate a population size above some threshold in order to ensure that many areas of the search space are considered in each generation.

5.4 Using Recombination Alongside UMAD

Previous work has shown that uniform mutation alone, without alternation, is approximately as powerful as the benchmark genetic operators [8]. The present work raises additional questions about the utility of recombination. We therefore conducted runs that used a 50/50 combination of alternation and UMAD, in which an independent choice of alternation vs. UMAD was made for each new child. The results in Table 5 show that using alternation alongside UMAD decreased its performance on all three problems, with two of those differences significant.

5.5 Genome Sizes

While it is clear that several of these mutation-based systems outperform the benchmark set of genetic operators on the bulk of our test problems, it is less clear why they do so and what other effects they have on evolutionary dynamics. To better understand their impact, we plotted the average genome size in Figure 1. Each subplot contains one plot line representing the average genome size over the generations for each of the 100 runs for a particular combination of benchmark problem and genetic operator. Because runs are terminated as soon as a solution is found, many of the plot lines end before reaching generation 300. Both Replace Space with Newline and Syllables start with an average genome size of 200 and have a maximum size of 800, where the starting genomes in Vector Average runs have an average length of 100 with an overall maximum of 400.

The impact of size-neutral UMAD on genome size (the left-most column in Figure 1) is scattered on all three problems, with some runs showing a growth in average genome size to over 500, while other runs shrink to average sizes around 50.

Shrinking UMAD sharply drives down the average genome size on all three problems to sizes well below any of the other trends. With an addition rate of 0.09 and deletion rate of 0.1, the average ratio of size between parent and child will be $(1 + 0.09)(1 - 0.1) = 0.981$, meaning that a child will have approximately 2% fewer genes than its parent. The plots of average size for shrinking UMAD follow this expected decrease in size until they reach some size above zero. This suggests there may be a "floor" beyond which Plush genomes are unlikely to encode programs that have a reasonable chance of being selected, meaning that children that randomly grow or stay the same size are the only ones that receive selections.

Just as shrinking UMAD drives the sizes down, growing UMAD drives the sizes up at a similar rate, where there are then complex interactions between the growth bias and the cap on the maximum genome length. In these runs, if an offspring is generated with a genome that exceeds the maximum allowed length, then it is replaced by a newly generated random individual. This likely accounts for the oscillations of the average genome length between a ceiling near the maximum allowed and a strong band caused by the injection of new random programs. Since the random individuals are likely considerably worse than the large ones, this process accelerates while larger genomes receive most of the selections, until substantial proportions of the population are being replaced by random programs.

The uniform mutation plots display a peculiar trait, particularly on the Syllables problem, where many runs have the same average genome size for hundreds of consecutive generations. Remember that a child produced by uniform mutation will always have the same size as its parent, since genes are simply replaced, meaning that changes in average genome size simply reflect which sizes become more prevalent in the population. That said, if one or more genomes of the same size dominate parent selections over a few generations, genomes of that size may make up the entire population in a generation. Once that happens, every single new child will also be that size, resulting in the horizontal lines in the plot. Even with this strange trait, uniform mutation performed about as well as UMAD on these problems.

6 DISCUSSION

The choices of the particular addition, deletion, and mutation rates for UMAD and uniform mutation used here were somewhat arbitrary. The fact that both size-neutral UMAD and shrinking UMAD did well (see Table 2) suggests that there is some flexibility in setting those rates. Additionally, our experiment in Section 5.2 with both doubled and halved addition rates (and corresponding deletion rates) showed that UMAD has considerable robustness to changes in these weights. However, the weaker performance of growing UMAD suggests that parameter settings that actively encourage genome growth can lead to code bloat and poor performance.

We designed many of our comparisons between UMAD and variants or other operators to tell us what characteristics of UMAD are important to its improved performance compared to the benchmark operators. HAHD does additions and deletions like UMAD, but not both in one step. HAHD performed slightly worse in this comparison, showing that the ability of UMAD to perform both additions and deletions in one step helps, although it is not crucial.

Similarly, we compared UMAD to high-rate uniform mutation to see if the dissociation of adding and deleting genes contributes to UMAD's success. As noted in Section 5.5, UMAD also has the ability to make genomes larger or smaller, which uniform mutation lacks. UMAD performed significantly better than uniform mutation on three problems, and better than the benchmark on three problems more than uniform mutation. These differences in performance suggest that the ability to add and delete genes in different places, along with the ability to change genome sizes, is somewhat important for the performance of UMAD.

Our experiments with different population sizes clearly show that UMAD performs poorly if the population is too small. Since a population is not necessary for providing individuals to crossover when we only use UMAD, why do we need a population? The other way in which individuals interact is through parent selection. In all of our experiments we use lexicase selection [7]. Lexicase selection first shuffles the list of test cases; it then considers the cases one at a time, discarding any individuals that are not tied for best on the case, until a single individual remains. Lexicase selection emphasizes different test cases, and therefore different parts of the search space, with each selection. In this way, we hypothesize that larger population sizes allow lexicase selection to prefer individuals with different strengths in different selections, maintaining exploration in numerous areas in the search space. All of the test problems used here have several hundred test cases, so with a population size as small as 10, lexicase selection cannot consistently emphasize all of the test cases, which may mean that during some generations it never selects the individuals that would best move it toward a solution. Thus a reasonably sized population is necessary in order for lexicase selection to maintain search lines in multiple interesting parts of the search space.

7 RELATED WORK

Explorations of mutation-only tree-based GP systems date back several decades, although mutation-only systems never became a popular approach in the field. In fact the question posed was often not whether mutation was *sufficient* on its own, but whether it was *necessary*; section 5.2.1 of *A field guide to genetic programming* [14] is, for example, titled “Is Mutation Necessary?”

O'Reilly's PhD dissertation [13] proposed a set of mutation operators for tree-based GP, and showed that some mutation-only systems are competitive with tree-based GP with sub-tree crossover. For example, a system using simulated annealing with the proposed mutation operators consistently outperformed GP with sub-tree crossover on the test problems used. Chellapilla [15] also explored the efficacy of mutation-only tree-based GP sytems on 14 test problems, and concluded that “crossover is not essential for successful program evolution”.

Much of this early work was connected to the question of whether sub-tree crossover was in fact mixing and spreading building blocks. An alternative theory was that sub-tree crossover was really a form of macro-mutation; while a sub-tree from one parent was replaced by a subtree from the other parent, the effect was similar to replacing the sub-tree with a randomly generated subtree. Angeline [1] showed that two forms of *headless chicken crossover* [9], where a subtree was replaced with a randomly generated subtree, were as effective or better than sub-tree crossover on three test problems.

Over a decade later, White and Poulding [18] did a comparison of sub-tree crossover and sub-tree mutation based GP systems on six test problems, independently tuning the parameters for each approach, and found that “crossover does not significantly outperform mutation on most of the problems examined”.

More recently, much of the work in the area of genetic improvement [10, 11, 17] has been substantially or entirely mutation based. Here researchers are modifying existing code that was typically generated by humans, and mutation is a natural approach to making the small, local changes that are necessary.

8 CONCLUSIONS & FUTURE WORK

Many researchers in GP assume to some degree that crossover is necessary, or at least beneficial, for combining elements from parents in different areas in the search space, while mutation primarily fine tunes programs that are nearly correct. Our results defy these assumptions in multiple ways, at least in the context of PushGP with Plush genomes applied to these software synthesis problems.

The results in Table 2 make it clear that mutation alone can be a powerful genetic operator. Two versions of UMAD (size-neutral and shrinking) regularly outperformed the benchmark set of genetic operators used in previous work, and two other mutation-only systems (HAHD and high-rate uniform mutation) also outperformed the benchmark set in many instances. In all of these systems, mutation is not simply fine-tuning programs, but is entirely in charge of exploring the search space, and succeeding. A 50/50 combination of UMAD with a recombination operator (alternation) did not improve performance and in two cases harmed it; while this does not suggest that adding recombination is strongly detrimental, it also does not indicate that it is particularly helpful.

These experiments, especially when combined with the results of earlier studies, raise important questions about the role, necessity, and efficacy of recombination in this type of genetic search.

It is always possible, however, that no one has found the "right" recombination operator or operator parameters for PushGP and Plush genomes. As far as we know, no one has ever comprehensively explored the parameters and combinations of operators used in previous PushGP work. While those choices allowed PushGP to discover solutions to a variety of interesting software synthesis problems, there was never a claim that those were the "best" options; some limited parameter tuning work [12] definitely indicated that there was potential for improvements in these parameter settings. Thus there might exist different parameters or operators that outperform the mutation-based systems presented here, in which recombination plays a vital role. It is also possible, however, that different settings or different mutation operators might improve the results here even further. While we explored quite a few variations here, our exploration was by no means exhaustive.

This suggests the broad range of potential future work. These new mutation operators imply a large parameter space which we have only begun to explore, including alternative combinations of mutation operators like UMAD and crossovers like alternation. It would also be worth trying to identify problems where recombination is demonstrably important, as well as looking for more effective recombination operators.

Lastly, while all the work here was on PushGP using Plush genomes, mutation operators like UMAD could easily be used with other variable-length linear genome systems. It would be valuable to explore mutation-only systems in those domains as well to see if these results are in some way specific to PushGP and Plush, or are a more general phenomena. Previous work has shown that mutation-only systems can be very effective in tree-based GP systems; our results suggest that more attention should be paid to this area, and that additional exploration of the space of mutation operators would be valuable throughout GP.

ACKNOWLEDGMENTS

Thanks to the members of the Hampshire College Computational Intelligence Lab and Shawn Saliyev at the University of Minnesota, Morris for discussions that helped shape this work, and to Josiah Erikson for systems support. This material is based upon work supported by the National Science Foundation under Grant No. 1617087. 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.

REFERENCES

[1] Peter J. Angeline. 1997. Comparing subtree crossover with macromutation. In *Evolutionary Programming VI*, Peter J. Angeline, Robert G. Reynolds, John R. McDonnell, and Russ Eberhart (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 101–111.

[2] Hans-Georg Beyer and Hans-Paul Schwefel. 2002. Evolution Strategies – A Comprehensive Introduction. 1, 1 (May 2002), 3–52. https://doi.org/10.1023/A:1015059928466

[3] Thomas Helmuth, Nicholas Freitag McPhee, Edward Pantridge, and Lee Spector. 2017. Improving Generalization of Evolved Programs Through Automatic Simplification. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '17)*. ACM, Berlin, Germany, 937–944. https://doi.org/doi:10.1145/3071178.3071330

[4] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2015. Lexicase Selection For Program Synthesis: A Diversity Analysis. In *Genetic Programming Theory and Practice XIII (Genetic and Evolutionary Computation)*. Springer, Ann Arbor, USA. http://www.springer.com/us/book/9783319342214

[5] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2016. Effects of Lexicase and Tournament Selection on Diversity Recovery and Maintenance. In *GECCO '16 Companion: Proceedings of the Companion Publication of the 2016 Annual Conference on Genetic and Evolutionary Computation*. ACM, Denver, Colorado, USA, 983–990. https://doi.org/doi:10.1145/2908961.2931657

[6] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. In *GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, Madrid, Spain, 1039–1046. https://doi.org/doi:10.1145/2739480.2754769

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

[8] Thomas Helmuth, Lee Spector, Nicholas Freitag McPhee, and Saul Shanabrook. 2017. Linear Genomes for Structured Programs. In *Genetic Programming Theory and Practice XIV*. Springer.

[9] Terry Jones et al. 1995. Crossover, macromutation, and population-based search. In *Proceedings of the Sixth International Conference on Genetic Algorithms*. 73–80.

[10] William B Langdon and Mark Harman. 2015. Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation* 19, 1 (2015), 118–135.

[11] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 3–13.

[12] Nicholas Freitag McPhee, Thomas Helmuth, and Lee Spector. 2017. Using Algorithm Configuration Tools to Optimize Genetic Programming Parameters: A Case Study. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '17)*. ACM, New York, NY, USA, 243–244. https://doi.org/10.1145/3067695.3076097

[13] Una-May O'Reilly. 1995. *An Analysis of Genetic Programming*. Ph.D. Dissertation. Carleton University, Ottawa-Carleton Institute for Computer Science, Ottawa, Ontario, Canada. http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/oreilly/abstract.ps.gz

[14] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. 2008. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. http://www.gp-field-guide.org.uk (With contributions by J. R. Koza).

[15] Lee Spector, Jon Klein, and Maarten Keijzer. 2005. The Push3 execution stack and the evolution of control. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, Vol. 2. ACM Press, Washington DC, USA, 1689–1696. https://doi.org/doi:10.1145/1068009.1068292

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

[17] David R White, Andrea Arcuri, and John A Clark. 2011. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation* 15, 4 (2011), 515–538.

[18] David R. White and Simon Poulding. 2009. A Rigorous Evaluation of Crossover and Mutation in Genetic Programming. In *Genetic Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 220–231.