Preprint of: Pantrdige, E., Helmuth, T., and Spector, L. 2020. Comparison of Linear Genome Representations For Software Synthesis. In Genetic Programming Theory and Practice XVII. New York: Springer.

Comparison of Linear Genome Representations For Software Synthesis

Edward Pantrdige, Thomas Helmuth, Lee Spector

Abstract In many genetic programming systems, the program variation and execution processes operate on different program representations. The representations on which variation operates are referred to as genomes. Unconstrained linear genome representations can provide a variety of advantages, including reduced complexity of program generation, variation, simplification and serialization operations. The Plush genome representation, which uses epigenetic markers on linear genomes to express nonlinear structures, has supported the production of state-of-the-art results in program synthesis with the PushGP genetic programming system. Here we present a new, simpler, non-epigenetic alternative to Plush, called Plushy, that appears to maintain all of the advantages of Plush while providing additional benefits. These results illustrate the virtues of unconstrained linear genome representations more generally, and may be transferable to genetic programming systems that target different languages for evolved programs.

1 Introduction

Inductive program synthesis is the field of producing executable programs from a set of input-output examples [7, 14, 16]. General software synthesis refers to the subfield of inductive program synthesis in which the programs produced are expected to be capable of manipulating a variety of data types, control structures, and data structures. The field of genetic programming has produced some the most capable

Edward Pantridge
Swoop, Inc. e-mail: ed@swoop.com

Thomas Helmuth
Hamilton College e-mail: thelmuth@hamilton.edu

Lee Spector
Amherst College, Hampshire College, and The University of Massachusetts, Amherst e-mail: lspector@hampshire.edu

general software synthesis methods, such as PushGP [5], Grammar Guided Genetic Programming [1], and SignalGP [10]. The experiments and discussion in this paper focuses on PushGP.

PushGP synthesizes programs in the Push programming language. Push is a stack-based programming language designed for genetic programming, in which arguments for instructions are taken from typed stacks and return values are placed on the stacks [19]. A Push program is a sequence that may contain instructions, literals, and code blocks. A code block is also a sequence that may contain instructions, literals, and code blocks, allowing for hierarchically nested program structures.

When executed by a Push interpreter, the program itself is pushed onto the `exec` stack, a special stack that keeps track of the executing program. During execution, items from the `exec` stack are consumed from the program and evaluated sequentially. Literal values are placed onto stacks corresponding to their data types. Instructions are evaluated as functions that pop their arguments from the stacks and push their return values back onto the stacks. When code blocks are interpreted, their contents are unpacked and inserted at the start of the `exec` stack [19].

Push implementations typically provide instructions and stacks for common data types such as integers, floating point numbers, Boolean values, and strings. It is also possible for users to provide stacks and instructions for any other data types they choose [19, 12]. Since the executing program itself is stored on a stack, instructions can manipulate the executing code itself *as it runs*; this functionality is used to implement both standard and exotic control flow structures using the `exec` stack.

Like all genetic programming methods, PushGP manipulates "genome" data structures that correspond to executable programs. In its initial design, the PushGP genome structure was also the program structure of nested code blocks. With this representation, it straightforward to implement tree-based genetic operators like those used in tree-based genetic programming [8], but less straightforward to implement operators that act uniformly on program elements at all levels of nesting [17].

More recent implementations of PushGP use the Plush linear genome representation , which can be translated into the hierarchical code block program structure before execution of the program [6]. This layer of indirection provides flexibility with respect to which genetic operators can be applied to genomes, specifically with uniform mutation and crossover operators, which in turn has produced better search performance [3, 6].

This work compares two linear genome representations, each of which takes a different approach to the problem of specifying a nested structure in a flat, linear form. While both Plush and the new genome structure, Plushy, can represent the same set of Push programs, there are trade-offs between them that may affect evolvability and the dynamics of program size and structure.

We begin by detailing the two genome representations, Plush and Plushy. We then present experimental results that allow us to compare their search performance and to examine the effects of each on program structure over evolutionary time. We discuss factors relevant to choosing among the representations in practice, and

conclude by recommending the adoption of Plushy genomes in future PushGP work.

2 Linear Genomes: Plush vs. Plushy

Plush [6] and Plushy [13] are two different linear data structures that have been used to represent genomes in PushGP systems. Both genome representations encode the nested structure of Push programs, and can be translated into executable programs. The process of translating a genome into a program will determine which individual genes should be placed within nested code blocks to produce the structured Push program.

```
(5 x int_gt exec_if (3 x int_sub) (x 2 int_mult))
```

Fig. 1 A simple Push program that takes an integer input x and returns $(3-x)$ if $x > 5$, or $2x$ otherwise. Note that the program contains two code blocks (`(x)` and `(x 2 int_mult)`), exemplifying the nested, non-linear structure of Push programs.

Every instruction gene has a defined number of code blocks expected to follow the instruction, which is the same number in both representations. For example, the `int_add` instruction sums the top two integers on the integer stack, and thus opens no code blocks. The `exec_if` instruction takes two code blocks from the `exec` stack as arguments: one for holding the body of the "then" clause and one for holding the body of the "else" clause. If the top value on the Boolean stack is true, then the code block for the "else" clause is ignored. If the top value on the Boolean stack is false, then the code block for the "then" clause is ignored. Figure 1 shows a simple Push program that utilizes this `exec_if` instruction.

Just as the `exec_if` is defined to require two code blocks to follow it, other instructions also require specific numbers of code blocks as arguments. Note that this is a feature of the genome specifications we are discussing, not a requirement of the underlying Push programs. In fact, when using Push programs as genomes, there was nothing guaranteeing the presence of code blocks after instructions that made use of them. Often `exec_if` and similar instructions would just be followed by single instructions instead of code blocks; this can sometimes be useful, but it is often advantageous to have larger code blocks in these positions. When designing Plush , and afterward when designing Plushy , we chose to force instructions that can use code blocks to be followed by them, to increase the use of code blocks in evolved programs [6].

Given that instruction definitions are used to determine where code blocks are opened, it is left up to the genome representation to determine how to store the information denoting where each code block is closed.

The Plush genome representation is a flat sequence of instructions and literals. Each of these tokens is considered a gene of the genome. Each gene also has epi-

genetic markers that store information that is used when translating the genome into a program—these are "epigenetic" in the sense that they affect translation into Push programs, but do not appear in programs themselves. The distinction between genetic and epigenetic information raises the possibility that the two kinds of information could be varied in different ways or at different times during evolution. While in most prior work with Plush, including the experiments described in this chapter, epigenetic information was only varied during the production of offspring from parents (like genetic information), previous work has used hill-climbing search over variation of epigenetic markers to "learn" during an individual's lifetime [9]. The two kinds of epigenetic markers that have been used in PushGP systems are "close" and "silent" markers, though others could be created [6] .

The "close" marker is an integer denoting how many code blocks should be closed directly following that particular gene. This allows the genome to indicate where code blocks are closed using epigenetic markers attached to specific genes. If there are no code blocks open at that location, the close maker value is ignored; if the number of open code blocks is less than the "close" value, then all open blocks are closed. If some code blocks are left open after the entire program has been translated, it is assumed the code blocks are closed at the end of the program.

The "silent" marker is a Boolean flag denoting if the gene is silenced. If true, the gene is skipped during genome translation. Using these markers a genome can hold genetic material that does not influence the resulting program and potentially pass it on to it children.

Gene:	5	x	int_gt	exec_if	x	int_sqr	x	2	int_mult
Closes:	0	0	2	0	1	0	0	0	0
Silent:	false	false	false	false	false	true	false	false	false

Fig. 2 One potential Plush genome that produces the program from Figure 1 after translation. The definition of the `exec_if` instruction specifies the opening of two code blocks; one for the "then" clause and one for the "else" clause. The "close" epigenetic marker on the `x` gene denotes the end of the "then" clause for the `exec_if`. There is no gene with a non-zero close marker to denote the end of the "else" clause, and thus it is assumed to be at the end of the sequence. Notice that the `int_gt` instruction closes 2 code blocks despite no code blocks being opened by the previous genes, and thus these close markers are ignored by translation. The `int_sqr` instruction is not translated into the program because it has a true silent marker.

Figure 2 shows a Plush genome that produced the program from Figure 1 when translated. Due to the separation of genes and their epigenetic markers under this representation, the Plush data structure can be thought of as a tabular structure since every gene has a value for every epigenetic marker.

The Plushy genome representation is also a sequence of instruction and literal genes, however there are additional genes used solely for translation. Plushy genomes do not use epigenetic markers, but are instead simply flat sequences of genes. The two additional kinds of genes introduced thus far in Plushy genomes are `CLOSE` genes and `SKIP` genes [13].

The `CLOSE` gene denotes the end of a code block. If there are no code blocks open at that location, the `CLOSE` is a no-op. If some code blocks are left open after the entire program has been translated, translation continues as if additional `CLOSE` genes are present until all code blocks are closed.

The `SKIP` gene causes genome translation to ignore the subsequent gene. Much like the silent epigenetic markers used in Plush genomes, these `SKIP` genes can be used to suppress genetic material such that it does not appear in the resulting program, yet potentially can be passed down to children. `SKIP` genes also cause a following `SKIP` or `CLOSE` gene to be ignored.

```
5 x int_gt CLOSE exec_if x SKIP int_sqrt CLOSE x 2 int_mult
```

Fig. 3 One potential Plushy genome that produces the program from Figure 1 after translation. The definition of the `exec_if` instruction specifies the opening of two code blocks; one for the "then" clause and one for the "else" clause. The end of the "then" clause is denoted by the final `CLOSE` gene. There is no `CLOSE` gene to denote the end of the "else" clause, and thus it is assumed to be at the end of the sequence. There is no gene that opens a code block before the first `CLOSE` and thus it has no effect on translation. The `SKIP` gene specifies the following gene should not be included in the translation, which explains why `int_sqrt` does not appear in the translated program.

Figure 3 shows a Plushy genome that produces the program from Figure 1 when translated.

2.1 Random Genome Generation

Random genomes are used to seed the initial population of genetic programming runs. Each genome type is generated differently to ensure the logic, structure, and size of the programs in the initial population is diverse.

The instructions and literals in random Plush genomes are typically chosen with a uniform distribution. The close epigenetic markers in Plush are initialized using a probability distribution. Sampling the distribution will give the value of the "close" marker. In previous PushGP research a binomial distribution with $n = 4$ and $p = 1/16$ is used. This yields the following probabilities for assigning values for the "close" marker.

Closes	Probability
0	0.772
1	0.206
2	0.021
3	0.001

We do not use "silent" markers in this work.

When generating Plushy genomes , a set of Push instructions and literals provided by the user is available. Plushy simply adds additional elements to this set for

the CLOSE and SKIP genes. As described above, the definition of each instruction in the set denotes how many code blocks are opened by the instruction, based on the number of arguments it takes from the exec stack.

If the set of genes were randomly sampled with uniform probability, the CLOSE gene would occur in genomes at a rate of $\frac{1}{|S|}$, where $|S|$ is the number of available genes. This likely provides too few CLOSE genes compared to the number of code blocks opened. Instead, to generate Plushy genomes with a larger proportion of CLOSE genes, we set the probability of sampling a CLOSE gene proportionally to the sum of all "open" counts across all instructions. For example, if there are 10 instructions that each open 1 code block, the CLOSE gene is given a 10 times the probability of being added to the Plushy genome compared to any other instruction. This results in an average of one CLOSE for every code block opened.

2.2 Genetic Operators

When using Plush genomes , the genes and their epigenetic markers are two separate values corresponding to the same location in the genome. Genetic operators can affect either the genes, their epigenetic markers, or both. Uniform crossover and mutation operators that keep genes with their epigenetic markers have typically been used [17, 3]. Additionally, specialized genetic operators that do not effect gene values can be applied to epigenetic markers. For example, a *uniform close mutation* operator changes the "close" epigenetic markers by incrementing or decrementing them by one [6]. This close-marker mutation operator is applied to each gene in the genome with some configurable probability.

Plushy genomes do not contain epigenetic markers. Genetic operators that manipulate the genes in the genome are modifying both logic and structure. The uniform genetic operators that are applied to Plush can be applied to Plushy, with the exception of the epigenetic-marker operations. Genetic operators commonly used in the field of genetic algorithms can also be applied to Plushy genomes.

Genetic operators that add random genes to a Plushy genome , such as a mutation, will utilize the same increased probability of adding a CLOSE gene as seen in random genome generation discussed in section 2.1.

3 Impact on Search Performance

A genome is the data structure manipulated by genetic operator throughout evolution. Different genome structures yield different landscapes to search over. Some landscapes may be more difficult to search through and thus search performance could be degraded when using certain genome representations.

3.1 Benchmarks

To evaluate the impact of Plush vs Plushy genomes on search performance, we tested each on 25 problems from the general software synthesis benchmark suite [5] . These benchmark problems come from coding assignments traditionally given to human programmers in introductory computer science classes. For our detailed analysis of the affects of each genome on evolved program structure, we selected a representative subset of 10 problems, which are:

- **Compare String Lengths**. Given three strings (s_1, s_2, and s_3) return true if $length(s_1) < length(s_2) < length(s_3)$, and false otherwise.
- **Double Letters**. Given a string, print the string, doubling every letter character, and tripling every exclamation point. All other non-alphabetic and non-exclamation characters should be printed a single time each.
- **Last Index of Zero**. Given a vector of integers of length ≤ 50, each integer in the range $[-50, 50]$, at least one of which is 0, return the index of the last occurrence of 0 in the vector.
- **Mirror Image**. Given two lists of integers of the same length ≤ 50, return true if one list is the reverse of the other, and false otherwise.
- **Negative To Zero**. Given a vector of integers in $[-1000, 1000]$ with length ≤ 50, return the vector where all negative integers have been replaced by 0.
- **Replace Space With Newline**. Given a string input, print the string, replacing spaces with newlines. Also, the program should return the integer count of the non-whitespace characters.
- **String Lengths Backwards**. Given a vector of strings with length ≤ 50, where each string has length ≤ 50, print the length of each string in the vector starting with the last and ending with the first.
- **Sum of Squares**. Given an integer $0 < n \leq 100$, return the sum of squaring each positive integer between 1 and n inclusive.
- **Syllables**. Given a string (max length 20, containing symbols, spaces, digits, and lowercase letters), count the number of occurrences of vowels in the string and print that number as X in "The number of syllables is X."
- **Vector Average**. Given a vector of floats with length in $[1, 50]$, with each float in $[-1000, 1000]$, return the average of those floats. Results are rounded to 4 decimal places.

For each benchmark problem, 100 runs were performed with each genome type. All runs were performed with the same configuration of the PushGP system , with the exception of the genome type used. The hyperparameter values used to configure the PushGP system are presented in Figure 4. We use size-neutral uniform mutation with addition and deletion (UMAD) to make all children for both Plush and Plushy [3]. UMAD, with an addition rate of 0.09, adds a new random instruction before or after each instruction with 0.09 probability; it then deletes each instruction in the program with probability $\frac{1}{\frac{1}{0.09}+1} \approx 0.08257$ to remain size-neutral on average. Note that we do not use any crossover here; while crossover may play some

Parameter	Value
Runs per setting	100
Population size	1000
Max number of generations	300
Genetic operator	UMAD, used to make all children
UMAD addition rate	0.09
max genome size	varies per problem, but same for Plush and Plushy

Fig. 4 The configuration of the Clojush PushGP system for the experimental runs performed for this research. We leave the tuning of these configurations for each genome type as future research.

role in deciding which representation to use, UMAD by itself has outperformed any crossover technique we have tried, so we used it here. The initial and maximum genome sizes vary per problem, and follow the recommendations from the benchmark suite's technical report [4].

3.2 Benchmark Results

Figure 5 shows the number of solutions found by the genetic programming runs using Plush or Plushy genomes for all problems. Only one of the differences in success rate is significant using a Chi-squared test at $a = 0.05$: the Syllables problem. All other problems show no significance in the difference in numbers of successes. This shows that, at least for these program synthesis problems and hyperparameter settings, the choice of genome between Plush and Plushy has little to no effect on performance. Thus the choice to use Plush or Plushy should be based not on their effects on performance, but instead on other considerations such as flexibility with respect to genetic operators and the required amount of hyperparameter tuning.

4 Genome and Program Structure

Push programs have meaningful structure organized by code blocks, which affect the semantics of programs, particularly with respect to control flow. When evaluating genome representations , we therefore consider program structure in addition to search performance.

4.1 Sizes

Figure 6 shows a comparison of genome lengths produced during PushGP runs for each genome representation for 10 representative problems. Plushy genomes tend to be slightly longer than Plush genomes . This is to be expected because Plushy

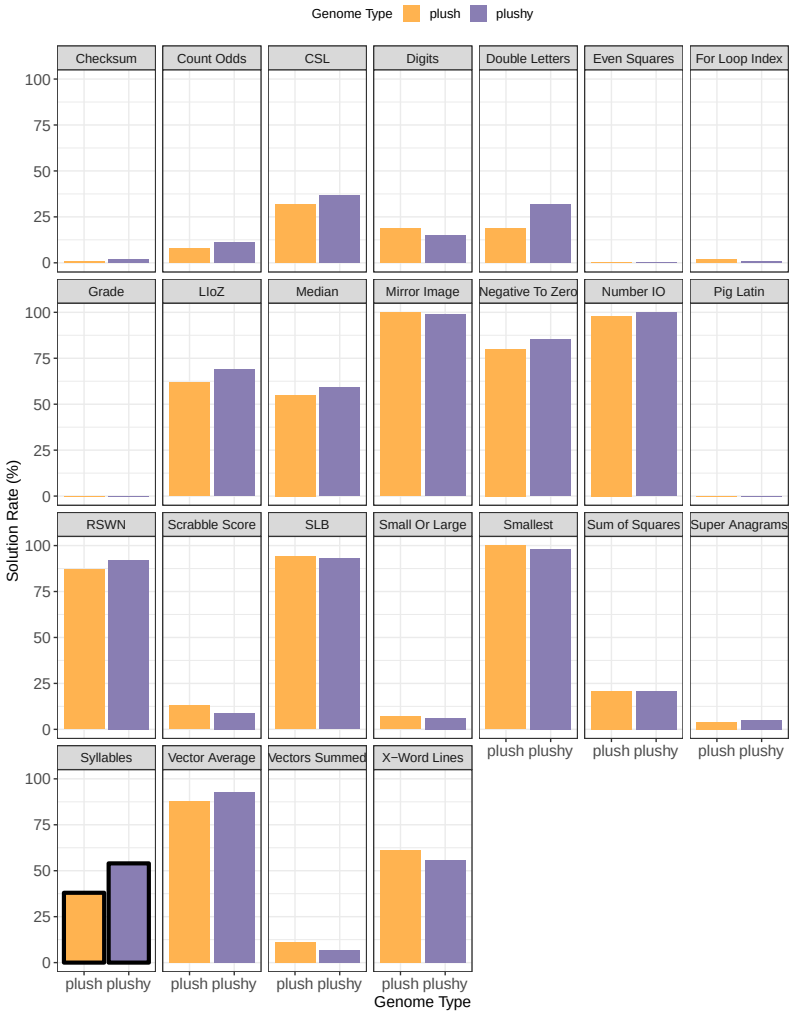

Fig. 5 The rate of solutions found using Plush genomes versus Plushy genomes. A genome is a solution if it receives an error of zero on all cases in a previously unseen test set after being simplified by an automatic simplification algorithm [2]. Each genome representation was evaluated across 100 runs for each problem. The difference in solution rates is only significant for one problem, "Syllables", shown with a black outline. On the "Syllables" problem the Plushy genome produces more solutions.

genomes require explicit `close` genes, increasing the size of genomes, whereas Plush stores "close" markers as epigenitic markers that do not contribute to genome size.

The examples from previous sections also illustrate the difference in size. The Plush genome in Figure 2 contains 9 genes. The Plushy genome in Figure 3 contains

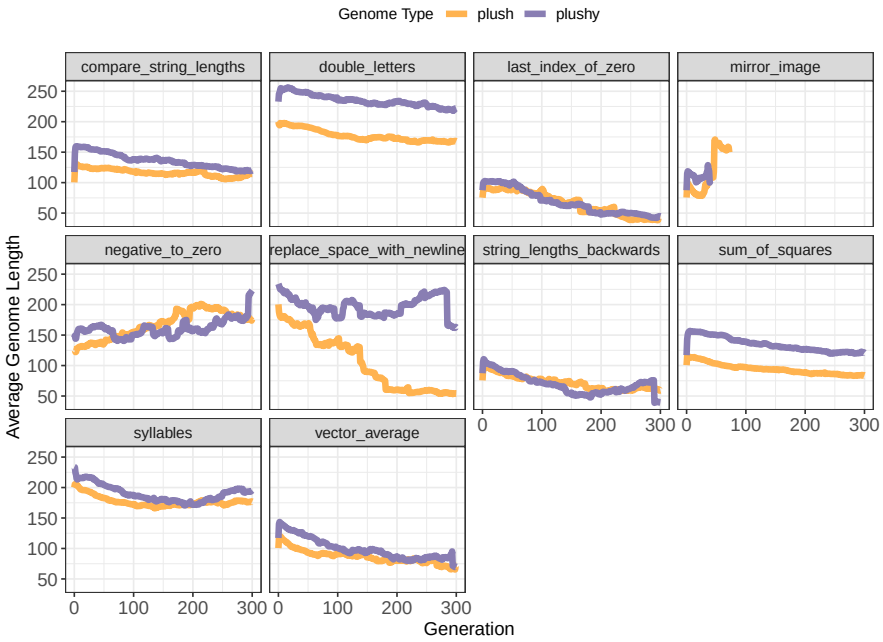

Fig. 6 Average Plush and Plushy genome lengths for all benchmark problems, averaged across all benchmark runs.

12 genes. Both genomes translate into program in Figure 1, and both genomes only silence/skip one gene.

Figure 7 shows a comparison of program sizes produced during PushGP runs for both genome representations. Despite producing longer genomes, the Plushy data structure tends to translate into slightly smaller programs. This further confirms that the difference lengths was due to `CLOSE` genes, which affect genome lengths but not program sizes.

It seems as though the Replace Space With Newline problem is an outlier for both genome and program sizes. Genome and program sizes tend to be similar for Plush and Plushy experiments in the early generations. In later generations, the Plushy genomes and programs far exceed the size of their Plush counterparts. Since a large number of runs had finished by that point (by finding a solution), this can likely be attributed to a small number of outliers for each drastically changing the average of the remaining runs.

Push programs are nested structures of code blocks. It is possible to measure the maximum depth of a program. We refer the maximum depth of a program as the program's depth. Figure 8 shows that Plush genomes and Plushy genomes tend to produce similar program depths. The Sum of Squares problem is a drastic outlier here, with programs translated from Plush genomes tending to have roughly twice the depth of the programs translated from Plushy genomes.

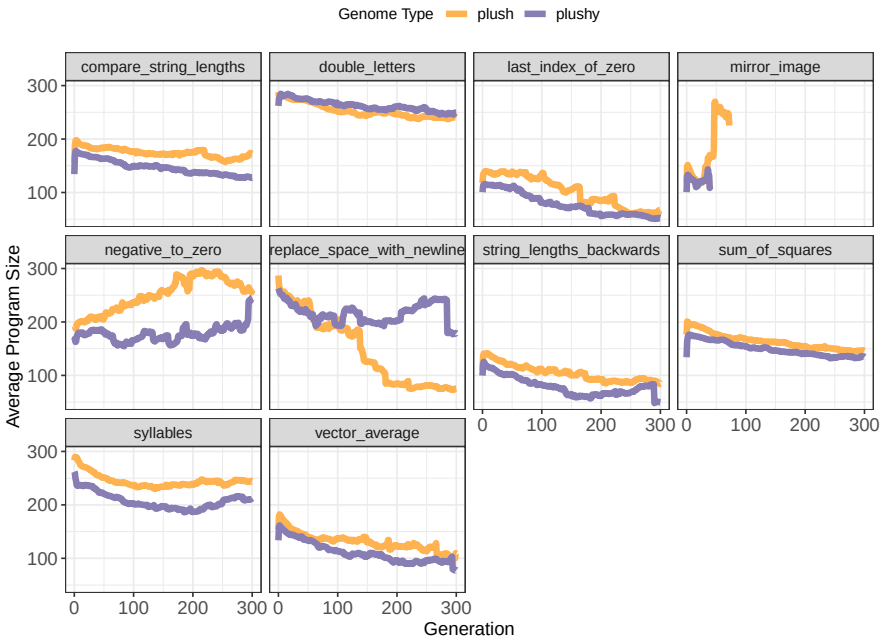

Fig. 7 Average Push program sizes produced using Plush and Plushy genomes for all benchmark problems, averaged across all benchmark runs.

It is important to note that these genome representations do not have direct effects on program depth, as only the instructions they contain dictate where and how many code blocks are opened. Thus any differences here come about by evolutionary pressures. So, it may be the case that for the Sum of Squares problem the way in which Plushy genomes close code blocks made it evolutionarily advantageous to have more nested instructions than with Plush.

4.2 Presence of "Closing" Genes

As PushGP searches for solution programs, it manipulates genomes such that the logic and structure of the resulting programs is varied generation to generation. Figure 9 shows the prevalence of "closing" genes in both kinds of genomes as evolution progresses. For Plush genomes, this is the percentage of genes in the genome with a non-zero close epigenetic marker. For Plushy genomes, this is the percentage of `CLOSE` genes in the genome.

The levels of "closing" genes stay relatively stable for both Plush and Plushy throughout evolutionary time, often ending with approximately the same percentage of closing genes as in the initial generation. These flat trends indicate that the levels

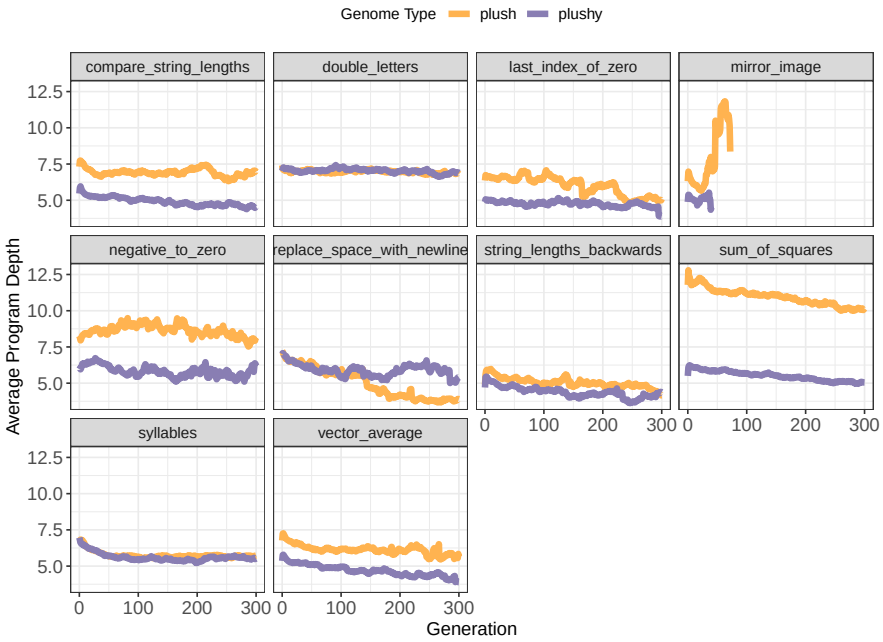

Fig. 8 Average program depths produced using Plush and Plushy genomes for all benchmark problems, averaged across all benchmark runs.

of closing genes are largely dictated by the percentage of close epigenetic markers / `CLOSE` genes present in random code created during initialization and mutation, and are not reflective of evolutionary pressures toward higher or lower levels. The percentage of close markers with Plush starts around the same level (around 0.25) for every problem, as would be expected with the hard-coded probabilities of close markers as described earlier. The percentage of `CLOSE` genes in random Plushy genomes depends on the instruction set, and will therefore be different for these different problems, which use differing instruction sets. This explains the high level of `CLOSE` genes for the Sum of Squares problem, which uses a higher percentage of `exec` stack instructions (those responsible for opening code blocks) compared to the other problems here. Despite the adaptive prevalence of `CLOSE` genes offered by Plushy as discussed in section 2.1, it is interesting to recall the lack of significantly different solution rates reported in Figure 5. This suggests that the performance of evolution is not particularly sensitive to the prevalence of "closeing" genes.

As demonstrated in Section 2, when using either Plush or Plushy it is possible to have "closing" genes that have no impact of the structure of the resulting program because they occur at locations in the genome where no code blocks are open. In order to compare the number of close genes that are having impact on program structure, we must compare the number of code blocks found in programs translated from each genome representation.

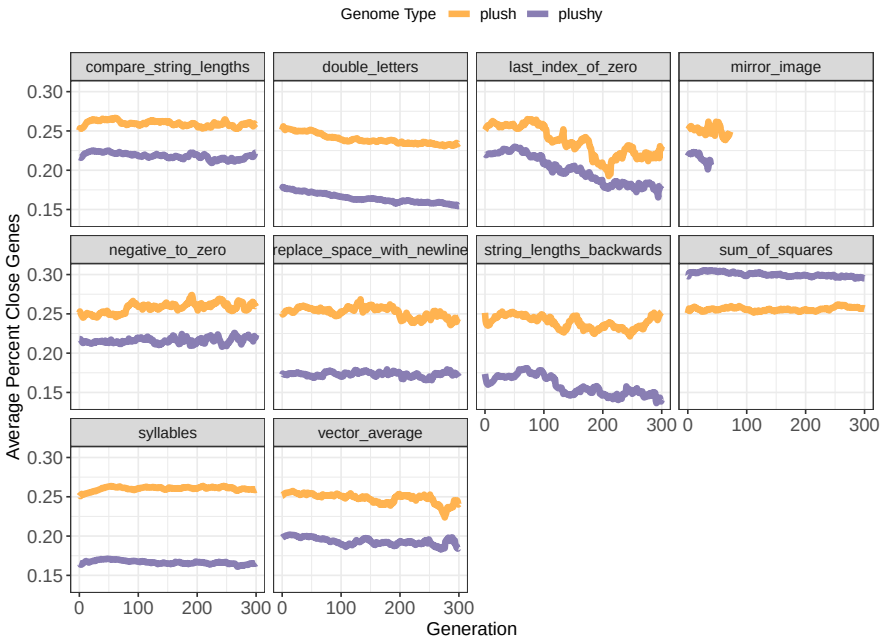

Fig. 9 The percentage of "closing" genes observed when using Plush and Plushy genomes for all benchmark problems, averaged across all benchmark runs. For Plush genomes, this is the percentage of genes in the genome with a non-zero close epigenetic marker. For Plushy genomes, this is the percentage of `CLOSE` genes in the genome.

Figure 10 shows the average number of code blocks in translated programs divided by the program size.

All problems show that Plush genomes tend to have a slightly higher concentration of code blocks in the translated programs. The range of differences between experiments using Plush genomes and experiments using Plushy genomes is very narrow, suggesting that the genome representation has very little bearing on the concentration of code blocks in a program. The small differences here likely reflect the fact that even though the genome sizes are the same, the Plush genomes will contain more actual instructions compared to Plushy genomes, for which use some genes are `CLOSE` genes, leading to slightly higher numbers of instructions that open code blocks in Plush genomes.

5 Other Considerations

Section 3 discussed how Plush and Plushy genomes have nearly identical search performance. The various measurements presented in Section 4 show that programs

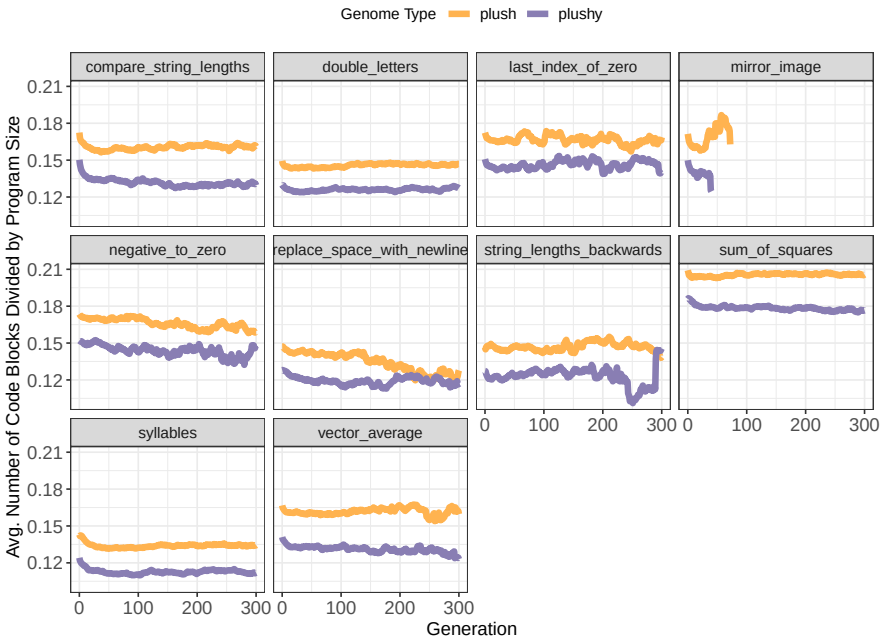

Fig. 10 The average number of code blocks divided by the average program size observed when using Plush and Plushy genomes for all benchmark problems, averaged across all benchmark runs. The plot shows a clear similarity between genome representations, especially considering the narrow range of the y-axis.

produced while using the different genome representations are usually similar. This may seem to indicate the choice between Plush and Plushy genomes is inconsequential, but in practice there are important differences regarding their effects on usability and ease of implementation.

5.1 Hyperparameter Fitting

Most machine learning systems have a collection of hyperparameters that can be tuned to problem-specific values that improve performance. Typically hyperparameters for genetic programming systems include the population size, mutation rates, and parent selection methods. Grid search is a common method of tuning hyperparameters by exhaustively evaluating sets of values taken from a grid of hyperparameter values.

As mentioned in Section 2.1, the Plush genome representation requires a probability distribution to generate epigenetic marker values for random code for ini-

tialization and mutation. Probability distributions are difficult hyperparameters to tune.

All previous research using Plush genomes assumes a binomial distribution of initial values for epigenetic markers, although this has not been proven to be optimal via theoretical analysis nor empirical experimentation, and in fact has never been tuned. We believe that they are relatively robust to moderate change. However, it is possible that better-tuned values may lead to better performance than has been seen previously. Even if the optimal distribution is a binomial distribution in all cases, there are two hyperparameters to tune (n and p) for initial close marker assignment alone. If the optimal type of probability distribution is problem specific, the number of hyperparameters is unknown. This further complicates the tuning of hyperparameters that is required when using the Plush genome representation. Typically, the computational cost of tuning all hyperparameters drastically increases as the number of hyperparameters increases.

In contrast, when using Plushy genomes the choice of which instructions can appear in the instruction set determines both structure and logic. No additional hyperparameters are required specifically to initialize the `CLOSE` genes. Furthermore, when using Plushy genomes the proportional rate of `CLOSE` genes presented in Section 2.1 agrees with the intuition on how many `CLOSE` genes should appear in a random program and is suitable for most cases. Using this method of generating random genomes, there are no hyperparameters to tune when using Plushy genomes.

Section 2.2 discussed the separate set of genetic operators that can be used to vary the epigenetic markers on genes in Plush genomes. These mutation operations often require their own hyperparameter tuning for values such as mutation rate. When using Plushy genomes, there is no need for genetic operators that vary epigenetic markers, and thus no additional tuning is required.

It is possible that future research will produce genetic operators that specifically target the `CLOSE` and `SKIP` genes of Plushy genomes. These operators may expand the space of hyperparameters.

5.2 Applicable Search Methods

The field of inductive program synthesis has a large variety of methods undergoing active research across many problem domains. There is no clear superior family of algorithms that dominate the field. It is in the best interest of the field to compare and evaluate as many systems as possible to gain a better understanding of their behaviors.

Nearly every program synthesis method has a different approach to representing programs. This heterogeneity makes comparisons of different search methods on the same problems difficult [11] . The simplicity of the Plushy genome facilitates such comparisons better than the Plush genome because any search method capable of making changes to a sequence of tokens can be used to search over the space

of Plushy genomes. Some examples of algorithms that could be used to search for solution Plushy genomes are:

- Evolutionary algorithms such as genetic algorithms and evolution strategies.
- Traditional local search methods such as simulated annealing and hill climbing.
- "Sequence to Sequence" neural architectures that are commonly used to synthesize sentences of natural language.
- Brute force combinatorics.

In contrast, Plush genomes require that each step in the search account for both gene tokens and their epigenetic markers. It is not immediately clear how a given search procedure should coordinate searching through genome space and epigenetic marker space in tandem, illustrating the complexity added by the epigenetic markers compared to `CLOSE` genes.

5.3 Automatic Simplification

Previous research on PushGP has detailed algorithms for automatically simplifying Push programs and Plush genomes [20, 15, 2]. This process uses hill climbing on program or genome size by randomly removing a small set of genes and testing that the program's outputs remain unchanged on the training data. If not, the genes are not removed, and a new random set of genes is removed, and the program is tested again. This process typically reaches a local size optimum within a few thousand iterations [18].

Automatic simplification was originally intended to yield programs that are easier for humans to understand [18, 20, 15], however it has also been shown that applying automatic simplification after evolution often produces programs that generalize better to unseen data [2]. In this sense, automatic simplification can be thought of as a regularization step for evolutionary program synthesis methods.

The solution rates reported in Figure 5 are for simplified programs on a held-out test set (that is, a test set not used during evolution). In this case, we automatically simplified the Push programs, not the genomes, so there should be no difference in simplification for those results. However, previous work described alternative methods for simplifying Plush genomes directly before translation [2], and we could also imagine automatically simplifying Plushy genomes.

When simplifying Plush genomes, we can randomly turn on a small number of "silent" epigenetic markers during a hill-climbing step, effectively removing the genes without losing their information. This allows for backtracking of the hill-climbing by unsilencing those genes at a later time, potentially allowing the process to escape local optima. This leads to smaller Push programs than non-backtracking approaches, though it only produces negligible gains in generalization [2].

When applying the same automatic simplification process to Plushy genomes, it is possible for a set of `CLOSE` or `SKIP` genes to be removed without the modifica-

tion of any genes that encode instructions. We leave it to future research to perform a rigorous study on the impact this has on generalization or interpretability.

5.4 Serialization

The main artifact of inductive program synthesis systems is the solution program found by the search. In PushGP, this artifact is typically either an executable Push program or a genome that can be translated into a Push program. In practice these solutions need to be serialized, stored, and recalled for later use.

Serializing Push programs requires the serialization of a nested structure of code blocks, literals, and instructions. One benefit of using a linear genome representation is that the solution's genome is often easier to serialize and de-serialize than the program.

Serializing Plush genomes requires denoting the gene value and the value for all epigenetic markers at every location in the genome. Serializing Plushy genomes only requires serializing the gene values. The simplicity of Plushy cuts down on the size of serialized genomes and improves interpretation and ease of de-serialization.

5.5 New Epigenetic Markers for Plush

One of the inspirations for the development of Plush genomes was the ability to add new epigenetic markers to add new data to the genome. We have discussed two such epigenetic markers that have easy translations to the Plushy representation: "close" and "silent" markers. However, we could imagine (and have experimented with) other epigenetic markers that would be more difficult to add to Plushy genomes. For example, we have experimented with the idea of adding "crossover hot-spots", which are locations in the genome where crossover is more likely to occur than in other locations. This is easy to envision as a new epigenetic marker, whereas it is not obvious how this feature would be added to Plushy genomes.

However, we have yet to find a specific use of a new epigenetic marker that actually improves performance of PushGP in practice. We therefore recommend keeping this ability in mind as a possible advantage of Plush—a context in which the complexity of Plush could add to its utility in comparison to Plushy.

6 Conclusion

We have compared two genome representations for evolving Push programs, Plush and Plushy . Experiments using the Clojush implementation of PushGP showed that the choice of representation has little effect on the problem-solving power of the ge-

netic programming system, making it impossible to recommend one representation over the other on the basis of problem-solving performance alone. We also explored other qualities of the programs evolved using each representation, and found some minor differences in genome/program sizes, numbers of closing genes, and numbers of code blocks in the translated Push programs. While these differences are interesting and potentially could impact problem-solving performance on other problems, they appear incidental in the problem-solving performance results in this study.

We then discussed at the qualitative aspects of each representation. Plushy requires fewer hyperparameters to be tuned than Plush, since the number of `CLOSE` genes to include is determined from the instruction set, whereas Plush requires at least one to use (and potentially to tune) hyperparameters that determine the distribution of "close" epigenetic markers in randomly-generated genes. Additionally, the simplicity of Plushy compared to Plush makes it easier to apply non-genetic-programming search methods and to serialize genomes.

After comparing the two representations, we recommend using Plushy genomes for the evolution of Push programs in most settings. Since both representations achieve similar problem-solving performance, Plushy's simplicity makes it more versatile and easier to use.

Acknowledgements

Feedback and discussions that improved this work were provided by other members of the Hampshire College Institute for Computational Intelligence, and by participants in the Genetic Programming Theory and Practice workshop.

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. Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. A grammar design pattern for arbitrary program synthesis problems in genetic programming. In Mauro Castelli, James McDermott, and Lukas Sekanina, editors, *EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming*, volume 10196 of *LNCS*, pages 262–277, Amsterdam, 19-21 April 2017. Springer Verlag.
2. Thomas Helmuth, Nicholas Freitag McPhee, Edward Pantridge, and Lee Spector. Improving generalization of evolved programs through automatic simplification. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 937–944, Berlin, Germany, 15-19 July 2017. ACM.
3. Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. Program synthesis using uniform mutation by addition and deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, pages 1127–1134, New York, NY, USA, 2018. ACM.

4. Thomas Helmuth and Lee Spector. Detailed problem descriptions for general program synthesis benchmark suite. Technical Report UM-CS-2015-006, Computer Science, University of Massachusetts, Amherst, June 2015.
5. Thomas Helmuth and Lee Spector. General program synthesis benchmark suite. In *GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1039–1046, Madrid, Spain, 11-15 July 2015. ACM.
6. Thomas Helmuth, Lee Spector, Nicholas Freitag McPhee, and Saul Shanabrook. Linear genomes for structured programs. In *Genetic Programming Theory and Practice XIV*. Springer, 2017.
7. Emanuel Kitzelmann. Inductive programming: A survey of program synthesis techniques. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming*, pages 50–73, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
8. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
9. William La Cava, Thomas Helmuth, Lee Spector, and Kourosh Danai. Genetic programming with epigenetic local search. In *GECCO '15: Proceedings of the 2015 conference on Genetic and Evolutionary Computation Conference*, pages 1055–1062, Madrid, Spain, 11-15 July 2015. ACM.
10. Alexander Lalejini and Charles Ofria. Evolving event-driven programs with signalgp. *CoRR*, abs/1804.05445, 2018.
11. Edward Pantridge, Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. On the difficulty of benchmarking inductive program synthesis methods. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 1589–1596, New York, NY, USA, 2017. ACM.
12. Edward Pantridge and Lee Spector. PyshGP: PushGP in python. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 1255–1262, Berlin, Germany, 15-19 July 2017. ACM.
13. Edward Pantridge and Lee Spector. Plushi: An embeddable, language agnostic, push interpreter. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '18, pages 1379–1385, New York, NY, USA, 2018. ACM.
14. Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. *ACM SIGPLAN Notices*, 49(6):408–418, 6 2014.
15. Alan Robinson. Genetic programming: Theory, implementation, and the evolution of unconstrained solutions. Division iii thesis, Hampshire College, May 2001.
16. Christopher D. Rosin. Stepping stones to inductive synthesis of low-level looping programs. *CoRR*, abs/1811.10665, 2018.
17. Lee Spector and Thomas Helmuth. Uniform linear transformation with repair and alternation in genetic programming. In *Genetic Programming Theory and Practice XI*, Genetic and Evolutionary Computation, chapter 8, pages 137–153. Springer, Ann Arbor, USA, 9-11 May 2013.
18. Lee Spector and Thomas Helmuth. Effective simplification of evolved push programs using a simple, stochastic hill-climber. In *GECCO Comp '14: Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion*, pages 147–148, Vancouver, BC, Canada, 12-16 July 2014. ACM.
19. Lee Spector, Jon Klein, and Maarten Keijzer. The push3 execution stack and the evolution of control. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA, 25-29 June 2005. ACM Press.
20. Lee Spector and Alan Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, March 2002.