# Flow of Control in Linear Genetic Programming

Justin Shonfeld Member IEEE, Daniel Ashlock, Senior Member IEEE

***Abstract*—Traditional flow of control for linear genetic programming includes structures such as if-then-else statements combined with gotos. In this study we examine additional classes of flow of control structures. The first is called the alternator. This is a deterministically variable flow of control that executes a goto every other time it is accessed. We demonstrate that evolution can use alternators that jump past one another to create solutions with significantly more complexity than those created by solutions without alternators for a simple binary string generation problem. The alternator, while clearly useful, would be difficult for human programmers to use effectively. The alternator thus demonstrates a strong disjunction between human-friendly and evolution-friendly programming languages. Domain specific flow of control structures tailored to the environment being studied are also examined. These are statements carefully designed for the problems being solved. Allowing controllers solving the Tartarus task to change the flow of control based on knowledge of their position in the interior boundary of a world substantially enhances the performance of the controllers. Comparison of the three different fitness functions used demonstrates that the benefit of the alternate flow-of-control is domain specific.**

## I. Introduction

Genetic programming uses evolutionary computation to evolve computer programs or, more commonly, useful fragments of code that solve a specific problem [7], [8], [9]. The original efforts at genetic programming used a parse tree representation. Linear genetic programming (LGP) comprises any type of evolvable code representation arranged as a sequence of commands [3]. It is possible to linearize the representation for tree structured programs, e.g. as with context free grammar genetic programming [4].

In this study we will work with linear representations that execute commands in sequence except when one of the commands changes the flow of control. The execution of commands in linear sequence can be considered the default or null flow of control state for LGP. For a parse tree representation a left based post-order traversal is the default flow of control. It is also useful to distinguish flow of control from the flow of information. Cartesian Genetic Programming [11], for example, allows for a non-linear flow of information but maintains a linear flow of control.

We examine two categories of flow of control statements. The first is the *alternator* which is simply a goto that executes every other time it is accessed. Implementation of the alternator requires a single bit of state information be saved but this means that its use adds a type of state conditioning to a genetic programming environment. A simple example of an environment where an alternator might be useful is in computing the roots of a quadratic equation. The alternator would permit implementation of the $\pm$ appearing in the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

for solving the equation $ax^2 + bx + c = 0$.

The second type of flow of control we study is a simple variation of the $if(condition)goto$ semantic but the condition is chosen to be highly relevant to the problem being solved. This simply explores the corollary to the no free lunch theorem [13] that tell us that algorithms should be specialized to the problem they are solving as much as possible. The incorporation of domain specific information has been shown to be helpful in a wide variety of evolutionary computation problems [5], [6].

The remainder of this study is structured as follows. Section II gives the design of the experiments performed. Section III gives and discusses results while Section IV draws conclusions and outlines possible next steps.

## II. Experimental Design

This section specifies several different environments where LGP can be applied and describes the flow of control variables used. Testing is performed by changing which statements are available in the LGP system and comparing the quality of resulting solutions. We use extremely simple LGP systems that can be stored as strings for this initial exploration.

Even though structured programming seeks to avoid the use of the *goto* statement to enhance the clarity of code, almost all flow of control structures can be built with if and goto statements. When evolving code, as with linear genetic programming, there is no need to provide the formalisms of while, repeat-until, or for loops unless the code is to later be mined for ideas by human experts. In this study we will simply use goto with various trigger conditions. We now define the first of our nonstandard control operators.

*Definition 1:* An **alternator** counts how many times it has been executed. If, in a given execution, it has been accessed an even number of times it does nothing. The alternator stores an address to goto. When it is accessed an odd number of times it executes a goto to the specified address.

### A. Complex Binary String Generators

The *complex binary string fitness function* (CBS-fitness) takes strings of length $n$ and tabulates the number of times each possible substring of length $k$ occurs. The counts are normalized to provide an empirical probability distribution. The fitness function is the entropy of this distribution, to be maximized. In essence we are searching for strings with the maximum possible diversity of substrings for a given length.

TABLE I
SHOWN IS THE LOGARITHM BASE 10 OF THE RATIO OF THE NUMBER OF LGP CHROMOSOMES IN COMPARISON TO BINARY CHROMOSOMES FOR SEVERAL LENGTHS.

| Length | 10 | 15 | 20 | 25 | 30 | 35 | 40 |
|---|---|---|---|---|---|---|---|
| $Log_{10}(Ratio)$ | 7.78 | 13.9 | 20.8 | 28.3 | 36.1 | 44.4 | 52.9 |

Two representations are compared for this problem. The first is strings of length $m$ in $\{0,1\}^m$ and the second is a simple LGP representation of length $m$ consisting of the characters 0 and 1 and alternator statements. Both the strings and the LGP are circularized and executed until they generate a 1024 character output string to which the CBF-fitness function is applied. This length gives a maximum fitness of 10. Examples of evolved simple LGPs with alternators are given in Figure 1. Using material presented in [10] it is not difficult to prove that almost all simple strings of length $m$ attain the maximum fitness *that the string representation can obtain*, making the problem of evolution to satisfy the CBS-fitness function trivial. This is because almost all strings of length $m$ are aperiodic, i.e. none of their rotations are equal to the original string. As we will see the alternator representation achieves much higher fitness.

```
Fit=7.45359          Fit=4.4594
 0) 0                 0) 0
 1) A 5               1) A 6
 2) A 5               2) 1
 3) 1                 3) A 0
 4) A 8               4) A 0
 5) A 8               5) 1
 6) A 0               6) A 8
 7) A 0               7) A 0
 8) A 3               8) 1
 9) A 3               9) A 5
```

Fig. 1. Examples of simple LGPs with alternators for generating complex strings. The theoretical maximum fitness is 10.

In the experiments, we will compare the evolution of base units of a periodic string with strings generated by simple LGPs using alternators. These structures will be used again in the experiments with fractals. For that reason it is worth obtaining a formula for the size of the search space. For simple strings of length $n$ this is easy - there are $2^n$ binary strings of length $n$. For the LGP's a loci may be 0, 1, or an alternator which can stores one of $n$ possible addresses so there are $(n+2)^2$ possible LGPs. Much of the increased expressive power of the LGPs probably arises from the larger search space. Table I gives the relative sizes of the search spaces for different lengths in logarithmic form.

### B. Generalized Julia Sets

Julia sets are a type of fractal in the complex plane defined by a complex parameter $\mu$. Given a point $z$ in the complex plane a series is computed using the rule:

$$\begin{aligned} a_0 &= z \\ a_{n+1} &= a_n^2 + \mu \end{aligned}$$

If the absolute value of the members of this sequence never exceed 2 then the point $z$ is a member of the Julia set, otherwise it is not. For points that are not members of the Julia set, the index of the first term of the series that has an absolute value of at least 2 is called the *iteration number* of that point.

Figure 2 shows two Julia sets for the parameters $\mu = 0.26$ and $\mu = 0.65i$. It is possible to generalize the notion of Julia sets by permitting the parameter $\mu$ to vary.

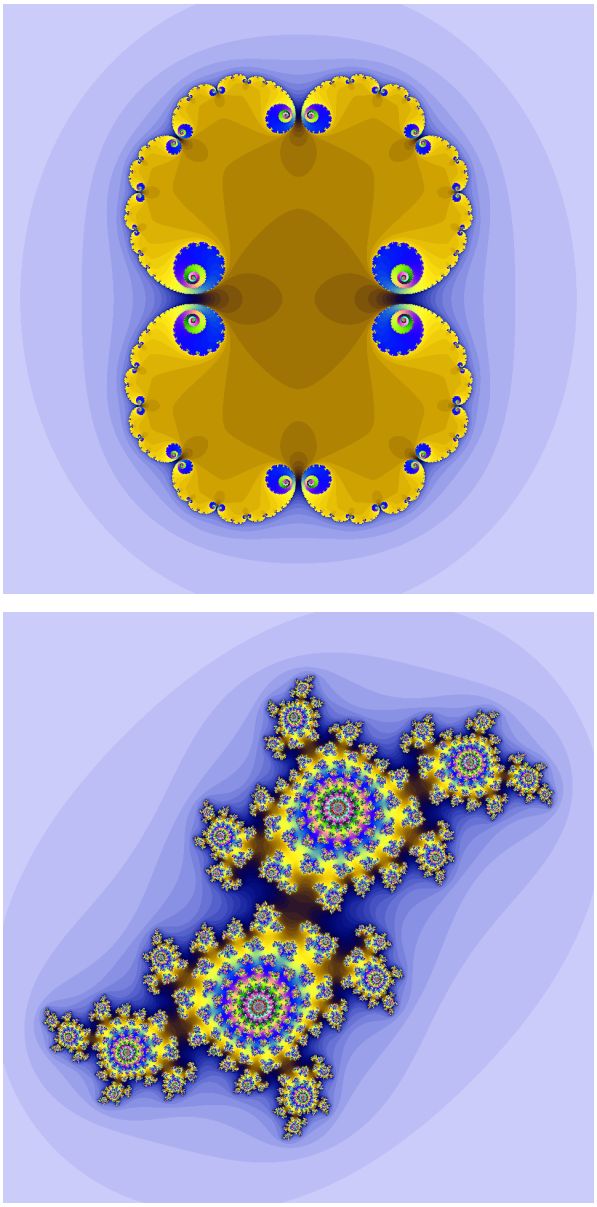

Fig. 2. Julia sets for the parameters $\mu = 0.26$ (top) and $\mu = 0.65i$ (bottom). These Julia parameters form the basis for the evolved generalized Julia sets.

*Definition 2:* Let $B$ be a binary string that is repeated periodically to make an infinite binary string $S_B$. If $\mu_1$ and $\mu_2$ are complex numbers then the *Generalized Julia Set* for sequence $B$ with parameters $\mu_1$ and $\mu_2$ is computed in the same manner as a standard Julia set except that, in order to produce term $k$ of the sequence, $\mu_1$ is added to the squared term if the $k$th bit of $S_B$ is 0 and $\mu_2$ is added if the $k$th bit of $S_B$ is 1.

The complex binary string generator gives us a source of binary strings that can be used to create generalized Julia sets. Rather than simply rendering complex strings as Julia sets, we will use a fitness function that can locate interesting Julia sets, or generalized Julia sets, and evolve binary strings that are judged by the quality of the generalized Julia sets they produce.

When running Julia sets in practice it is, of course, impossible to detect that a sequence satisfies the conditions for membership in a Julia set. What is done is to simply set an iteration limit and any point that makes it to this iteration limit without its sequence obtaining an absolute value in excess of 2 is considered to be in the Julia set. Such a point is necessarily *close* to points in the Julia set. For the generalized Julia set experiments in this study the iteration limit is set to 256 iterations. We now define the fitness function.

The fitness function uses an evenly spaced $31 \times 31$ grid of points in a square subset of the complex plane with corners $-1.6-1.6\imath$ and $1.6+1.6\imath$. At each point the iteration number of that point is computed or it is identified as being a point in the Julia set. The iteration numbers, which are in the range 0-255, are placed in 32 bins with the number of iteration numbers in each bin counted. The counts are normalized to an empirical distribution on the bins and the number of members $m$ of the Julia set found are counted. Fitness is:

$$fitness = E/(m+1) \tag{1}$$

Where $E$ is the entropy of the distribution among the bins. This function is to be maximized and so encourages a broad variety of iteration numbers and as few points as possible in the Julia set. To render a Julia set as a picture the iteration numbers, or Julia set membership, of a $960 \times 960$ grid are computed. The points correspond to pixels in the image with members of the Julia set rendered in white and other points colored according to a periodic pallet indexed by their iteration number. The images in Figure 2 were colored in this fashion.

### C. Tartarus

The Tartarus task was defined in [12]. It takes place on a $6 \times 6$ grid bounded by walls. Figure 3 shows a valid initial state for the Tartarus task. Six blocks are placed so that they neither touch the walls nor form a close $2 \times 2$ group of blocks. The robot is given 80 moves in which it may advance, turn left, or turn right. Turning is done in place and so is always possible. The robot can push one box ahead of it but cannot push two boxes. The robot cannot advance through a wall nor can it push a box through a wall. After 80 moves the robot is rewarded for the number of box sides against a wall. A box in a corner is thus worth two points, and a box against a wall but not in a corner is worth one point. The maximum score for a board is thus 10. In this study the fitness of a robot is the average over 100 instances of the Tartarus problem. New random instances are generated, with replacement, for each generation.

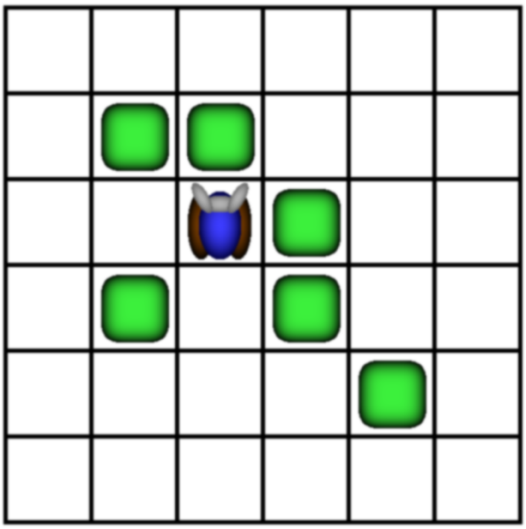

Fig. 3. A valid initial condition for the Tartarus task.

### D. Simple Linear Controller

The simplest representation for Tartarus is to evolve a fixed string of moves. We call this type of representation a simple linear controller (SLC). This representation can be improved by the use of *adaptive characters* that make a simple Boolean test on the squares adjacent to the robot. These represent flow-of-control restricted to a single, local decision. The adaptive and non-adaptive characters used are given in Table II. Finally we may modify the character string by adding characters for hard coded gotos, a type of flow of control called an alternator, and a Tartarus specific flow of control called *interiorP*. The addition of the flow-of-control variables yields a simple type of LGP representation.

*Definition 3:* A **goto** stores an index into the string and sets the execution position to that index when it is executed.

The goto command permits evolution to skip over parts of the code, in essence removing large parts of the code with a single mutation.

*Definition 4:* The **interiorP** test also stores an index into the string. When an interiorP test is executed then if the robot is adjacent to a wall, nothing is done. If the robot is not adjacent to a wall the interiorP test sets the execution position in the string to this index for the next time step.

We test several forms of the simple linear controller,

1) Just the non-adaptive characters.
2) Non-adaptive characters and goto.
3) Non-adaptive characters and the alternator.
4) Non-adaptive characters and the interiorP test.

TABLE II
THE FIRST THREE CHARACTERS LISTED BELOW ARE THE NON-ADAPTIVE CHARACTERS USED IN THE SIMPLE LINEAR CONTROLLER. THE LAST FOUR ARE THE ADAPTIVE CHARACTERS.

| Character | Meaning |
|---|---|
| L | Turn left |
| R | Turn right |
| F | Attempt to go forward |
| X | If there is a wall ahead of the robot turn left, otherwise attempt to go forward |
| Y | If there is a wall ahead of the robot turn right, otherwise attempt to go forward |
| U | If there is a box to your left, turn toward it, otherwise attempt to go forward |
| V | If there is a box to your right, turn toward it, otherwise attempt to go forward |

5) All characters but no flow of control.
6) All characters and goto.
7) All characters and the alternator.
8) All characters and the interiorP test.
9) All characters and all flow of control statements.

### E. Evaluation Tools

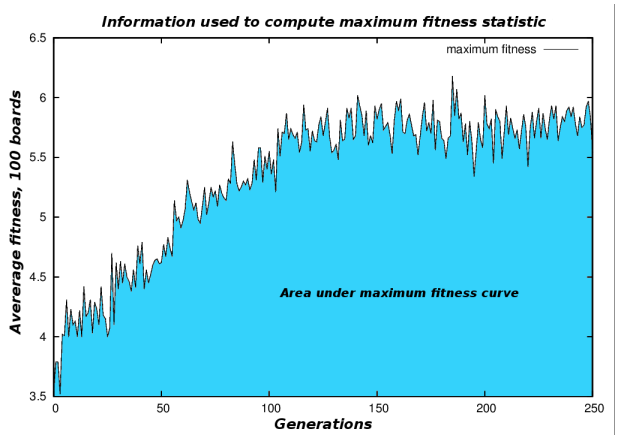


Fig. 4. An example of a maximum fitness plot with the area under the curve shaded.

Figure 4 shows a shaded plot of the maximum fitness of any population member at each generation of an evolving population. The jaggedness of the curve is the result of sampling new cases of the Tartarus problem to evaluate fitness in each generation. This stocasticity of maximum fitness makes comparison of results somewhat problematic. To combat this problem the *total maximum fitness* statistic is used for comparison.

*Definition 5:* The **total maximum fitness** (TMF) statistic for a single run of an evolutionary algorithm is the area under the plotted maximum fitness curve divided by a normalizing factor equal to the product of the known or estimated maximum fitness and the number of generations.

The normalization of the TMF makes its minimum value zero and its maximum one, assuming the maximum fitness used in normalization is correct. In this study we normalize Tartarus fitness using the true maximum fitness of 10. The TMF statistic contains information about both how high the maximum fitness curve moved and how fast it rose. Its meaning is not obvious beyond "larger is better" but it is useful for comparison of different experiments. It is also worth noting that if the experiments are run well past the point where maximum fitness is achieved the statistic converges to simple comparison of maximum fitness.

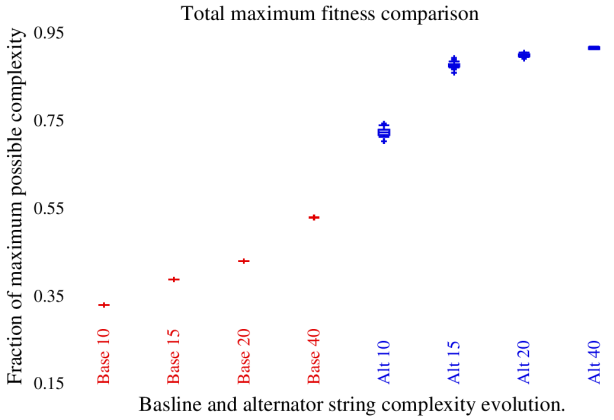


Fig. 5. Results for the simple string and the alternator equipped string representations for the CBS problem. Numbers denote length of the representation, runs without alternators are denoted by 'Base' while runs with alternators are denoted 'Alt'.

### F. Experiments Performed

Three collections of experiments are performed, the CBS experiments, the generalized Julia set experiments, and the SLC experiments for Tartarus.

*1) Complex Binary String Generators:* Both strings and simple LGP with alternators are evolved for lengths of $L \in \{10, 15, 20, 40\}$. The algorithm uses a population size of 120. Selection and replacement are performed with size seven single tournament selection. Mutation is performed on each new string. The string has from 1-3 loci replaced with new loci. Both the new loci and the number of loci replaced are generated uniformly at random. Evolution continues for 10,000 tournament selections with summary statistics, including maximum current fitness, saved every 100 selections. This number was chosen by preliminary experimentation, looking at when fitness curves leveled off. A collection of 30 independent runs of the evolutionary algorithm are performed for lengths 10, 15, 20, and 40 using both LGP systems with the alternators enabled and simple periodic strings. The results, in the form of boxplots for total maximum fitness are given in Figure 5.

*2) Generalized Julia Sets:* The algorithm used to evolve the generalized Julia sets is based on the one used in the CBS experiments. Runs are performed for the same representation lengths both with and without alternators. The differences are as follows. The fitness function used is the one specified for generalized Julia sets, the population size is increased from 120 to 1,000, and the algorithm is run for 100,000 tournament

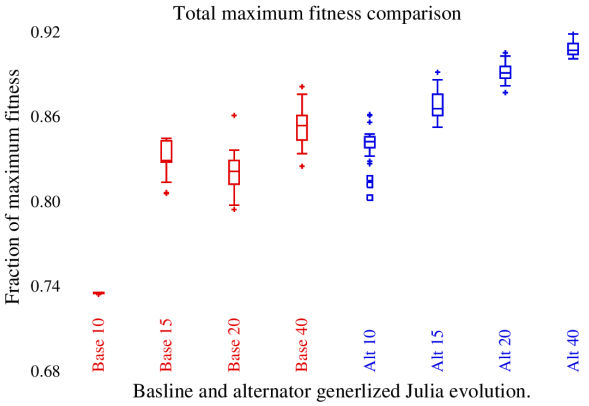


Fig. 6. Results for simple and alternator equipped representations for the generalized Julia set problem. Numbers denote length of the representation, runs without alternators are denoted by 'Base' while runs with alternators are denoted 'Alt'.

selections saving statistics every 1,000 selections. The increase in population size and length of evolution were based on preliminary experimentation. The values used for the complex parameters are $\mu_1 = 0.26$ and $\mu_2 = 0.65i$.

*3) Tartarus SLC Experiments:* All nine sets of commands for the SLC representation for Tartarus were tested separately. Initial populations were filled uniformly at random. The algorithm uses a gene of length 80, equal to the nominal number of moves. Since the various flow of control operations can cause the algorithm to reach the end of the string prematurely, the string was treated as being circular. In addition the potential exists for a string that specifies an empty set of moves, e.g. "0: Goto 0". Strings were thus permitted to generate 80 Tartarus moves (L,R,F) subject to the limit that they examine no more than 1000 characters. Strings that reached the limit of 1000 characters examined on a particular example of the Tartarus problem were interrupted and awarded fitness for their current board position. In effect they were penalized by being allowed fewer moves.

The algorithm was run using size four single tournament selection for 250 generations. During reproduction parents generated children via two-point crossover of their SLCs. The resulting SLCs then had 1-3 characters changed to new values selected uniformly at random. The number of new characters, and their position, was also selected uniformly at random.

## III. Results and Discussion

### A. Complex Binary String Results

Looking at the results in Figure 5 we see that the use of alternators enormously increases the complexity of strings that can be generated. In addition, the variability and fitness trajectories (not shown) indicate that with the alternators present there was non-trivial work for evolution to do.

The story for the simple string representation is different. As noted in the problem description, the probability that a gene generated uniformly at random for the pure string experiments will be aperiodic and hence generate the maximum fitness possible with that representation is $1 - \epsilon$ where $\epsilon$ is small and decreases as string length increases. The fitness results shown are box plots for 30 experiments. In the simple string representations values equal to the theoretical maximum appeared in the initial population in 119 of the 120 evolutionary runs performed for the four lengths. The theoretical maximum for this representation results when every ten character substring of the circularization of the string being evolved is distinct. If the strings being evolved have length $n$ then this value is $Log_2(n)$, which is what happened.

The CBS experiments show that the use of alternators can substantially increase the complexity, and hence the expressive power, of a simple linear representation. We will revisit these effects in the fractal experiments, described subsequently. Almost all the length 10 runs with alternators used 8 alternators and single examples of each digit. This effect was not present in solutions for representation lengths of greater than 10.

In the shortest alternator equipped runs of the CBS problem, length 10, an additional interesting phenomenon occurred. Examine Figure 1. The highest fitness example has all of its alternators paired. When this happens, the control bits of the alternator count to four in binary with the results that the first time they are encountered they do nothing but the next three times they generate the "goto" result. In essence they create a pattern **0111** of gotos. The phenomenon occurred for the longer representation lengths, but far less often. The four occurrences in the example shown in Figure 1 represents the largest number of occurrences and it is worth noting that they are nested.

### B. Generalized Julia Results

Figure 6 shows that the availability of alternators substantially increased the fitness of generalized Julia sets that were located. While all of the runs that could use alternators did, they used a lower fraction of them than the CBS experiments. The very small range of fitness values for the length 10 baseline experiment merited careful examination of the evolved results. It was found that the algorithm found the same optima 30 times in the form of a periodic string based on the length 10 string **0101100011**. This is not too surprising as there are only 1024 genes in the space and so it is quite plausible that an evolutionary algorithm simply found the global optima every time.

Following on the 30-fold duplicate answer for length 10 without alternators, the generalized Julia sets produced were examined for duplicates, the results are summarized in Table III. The notation used is an exponential one for sequences that takes the number of repetitions and exponetiated it with the number of times that number of repetitions occurred. So, for example $20^1 2^2 1^6$ means one image repeated 20 times, two images repeated twice, and six unique images.

The pattern of image repetition in Table III shows that the use of alternators also substantially enhances image diversity. Figure 7 shows the most common image, or one of the most common images, for each of the eight experiments. The

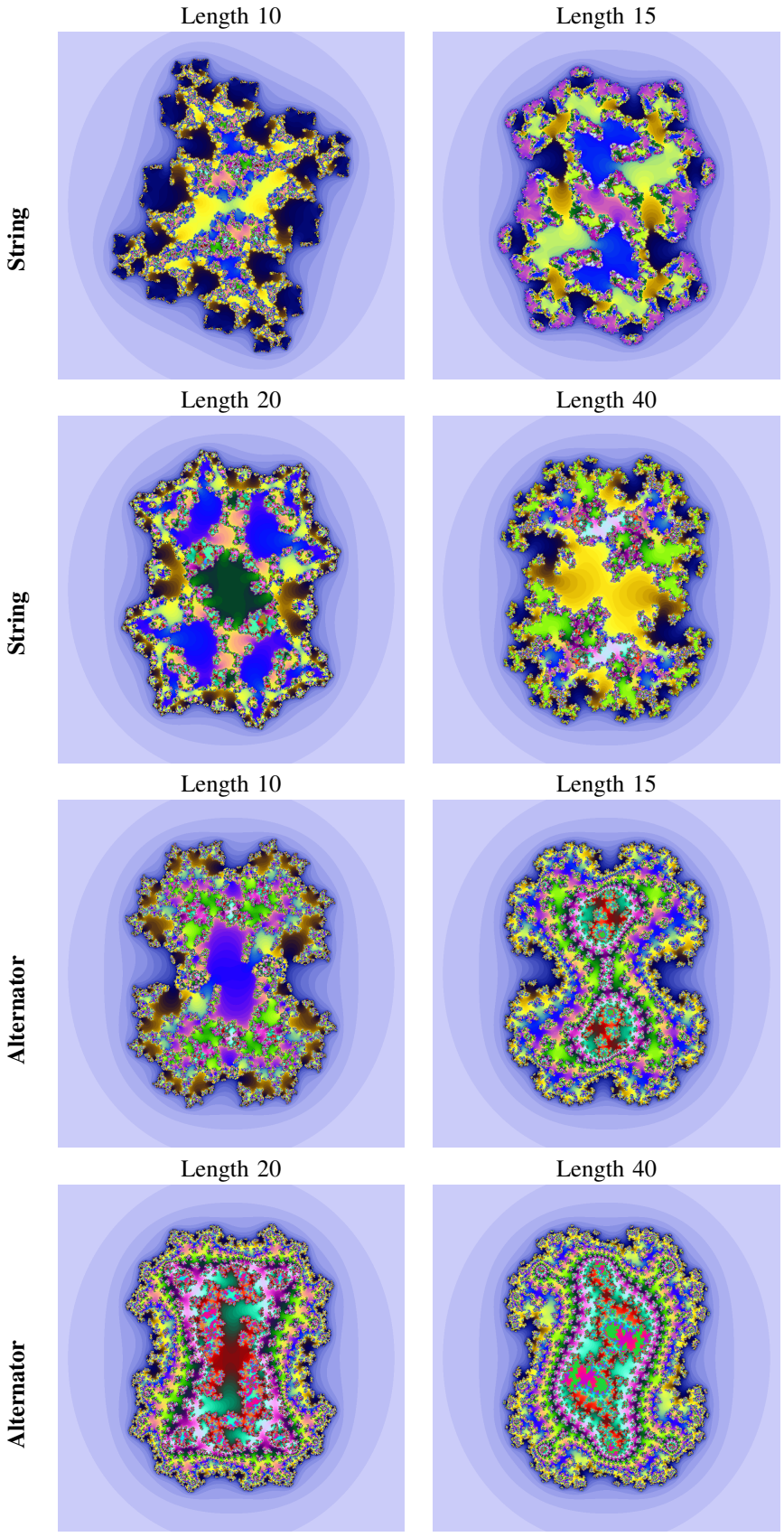


Fig. 7. Images that are the most common, or selected from among the most common, for the eight experiments performed with generalized Julia sets.

TABLE III
NUMBER OF REPEATED IMAGES FOR THE BASELINE AND ALTERNATOR EQUIPPED GENERALIZED JULIA SET EXPERIMENTS.

| Len. | Baseline | Alternator |
|---|---|---|
| 1 | $30^1$ | $16^1 10^1 2^1 1^2$ |
| 2 | $16^1 10^1 2^1 1^2$ | $1^{30}$ |
| 3 | $3^2 2^5 1^{11}$ | $1^{30}$ |
| 4 | $2^1 1^{28}$ | $1^{30}$ |

alternator based images for length 15 and above seem, to the authors, to be more complex and esthetically pleasing.

### C. Tartarus results

The results for the experiments with SLCs and LGPs are shown in Figure 8 which compares the distributions of maximum total fitness statistics for collections of 30 runs of the evolutionary algorithm for each of nine sets of commands used to build SLCs. Changing the available characters and sets of commands clearly causes significant variation in performance. The vertical scale of Figure 8 also demonstrates that we are achieving good (but far from optimal) results even among the best outcomes in these experiments.

When only non-adaptive characters are used the overall results are inferior. In contrast with a pure string-based controller any flow of control causes a significant improvement in the results. The use of the interiorP test causes the largest improvement among the experiments without adaptive characters, yielding results intermediate between the other non-adaptive results and the results with adaptive characters.

The adaptive character results are all significantly superior to those with non-adaptive characters. The importance of specific alternate flow of control commands has a different pattern than it did with non-adaptive characters. When adaptive characters were used the goto command added the most value rather than the interior operator. The alternator remained in last place for both types of character sets. The loss of value of the interiorP command almost certainly results from the ability of the four adaptive characters, X, Y, U, and V to produce similar effects to those available by using the interior flow of control. The results for permitting all characters and flow of control operators to be used did not significantly differ from runs with access to only the goto or interior operators.

The alternator flow of control, which permits more complex patterns of characters, was only beneficial in improving the fitness of the non-adaptive characters. This is consistent with the results for the CBS where no adaptive characters were available and the alternator substantially enhanced performance.

## IV. CONCLUSION AND NEXT STEPS

This study has shown that the novel flow of control operators proposed have a positive and substantial impact on all three test problems used. The inclusion of alternators in the CBS and generalized Julia set experiments enormously increased the expressiveness of the systems. This is not surprising given the differences in the sizes of the spaces encoded by the two different types of representations, shown in Table I. When the representation length is 40 we generate either 1024 (CBS) or 256 (generalized Julia sets) bits. The baseline does this by repeating a 40 bit string; the LGP by executing a simple LGP with alternators until it generates sufficient bits. From the perspective of algorithm design or run time the two experiments are very similar. The LGP space, however, encodes slightly more than fifty-two orders of magnitude more strings. This is ameliorated by the fact that the simple binary string representation is 1:1 while the LGP representation is many-one, but only slightly.

The baseline results for smaller representation length on the generalized Julia sets show that an evolutionary algorithm is too powerful for the search problem - the same optima was located 30 times for representation length 10. The LGP representation, with its astronomically larger space, is more appropriate to an evolutionary algorithm and the algorithm was able to locate better results. Intuition suggests that directly evolving binary control strings of length 256 or 1024 would be more difficult than evolving the alternator based representations, but it might be interesting to make the attempt.

The alternator was much less valuable for the Tartarus experiments than the other two flow of control statements. In these experiments the flow of control that gave the LGP access to the information that it was in the interior of the Tartarus world was more valuable. Even more valuable than any of the flow of control operators explored here was the use of adaptive characters. Each of these contains very local flow of control type information, limited to the choice of a single action. The alternator was also incorporated into a full LGP representation called an ISAc list [2] where it did not change fitness at all. We conclude that the alternator is more valuable in simple LGP environments, though a great deal more experimentation remains to be done.

### A. Discretization of Fractal Search

Every Julia set presented in this study used only two complex parameters with values 0.26 and $0.65_{\mathbb{I}}$ with the resulting fractal selected by a binary string. In past work on evolving Julia sets [1] the very irregular continuous dependence of a Julia set on its parameter meant that the effect of mutation was quite unpredictable. The system presented in this study removes this feature of evolving Julia sets and transforms it into a simple binary string problem that does not exhibit these stability issues. It is worth noting that this study used two particular Julia parameter - an infinite selection of other pairs of parameters is available.

### B. Next Steps

The paired alternators shown in Figure 1 suggest an interesting feature that could be added to the CBS and generalized Julia environments: doing this as a design feature. A macro pre-processor is used in programming languages like C and

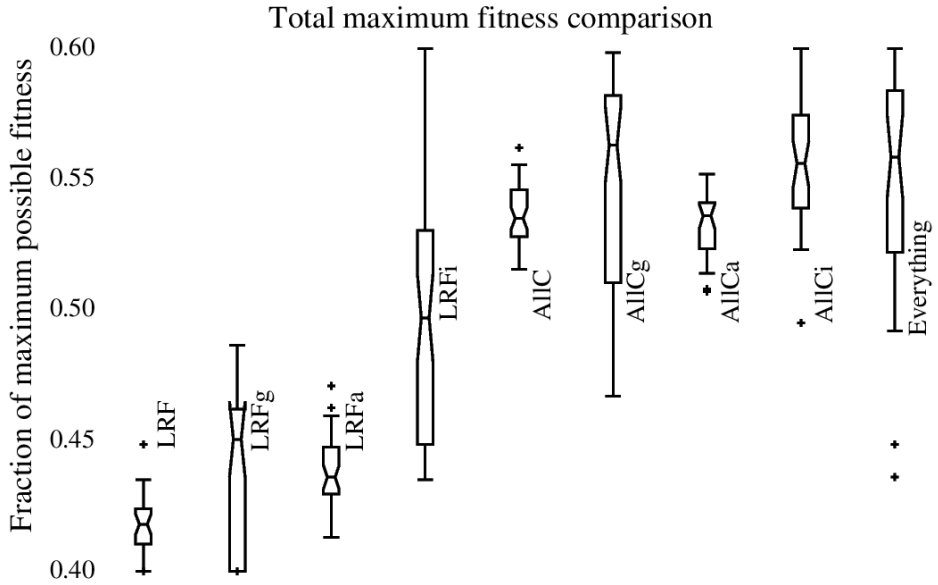


Fig. 8. Shown are wasp plots depicting the maximum total fitness statistic for the SLC experiments using nine different command sets. The letters g, a, and i denote the use of the flow of control commands *goto*, the *alternator*, and the *interiorP* test. The notation All means all characters, adaptive and non-adaptive, are permitted. Statistics for a collection of 30 evolutionary runs are shown for each set of commands.

C++ to increase expressive power. Adding a macro of the form "make $k$ copies of the next statement" could be added to the simple LGP formalism. This idea of macro-prepossessing enabled LGP has many other potential applications and is an early priority for future research. An LGP system would incorporate, potentially, a variety of macros that would be processed to obtain the LGP to be evaluated. This is a form of generative representation for LGPs.

An alternate method of examining this self-organized feature is to add a mutation that picks a statement and duplicates it over the next statement. This is a special-purpose mutation operator intended to increase the occurrence of paired alternators. It is not hard to see that serial alternators use their control bits to count in binary; the effect of different patterns of nesting of alternators is something that merits additional investigation.

*1) More General Julia Sets:* The two complex numbers chosen as Julia parameters in this study were chosen because, first, they generate interesting but unexceptional Julia sets and second, because one is real and the other is purely imaginary, meaning they form an orthogonal basis for the complex plane. This means that displacement in any direction, after squaring of the complex parameter, can be achieved. In addition to the already noted possibility of choosing other Julia parameters there is no reason not to replace the binary alphabet with a $k$-ary alphabet that uses $k$ different Julia parameters. This would create an additional, astronomical increase in the size of the search space for generalized Julia sets.

Another issue that arises is the choice of fitness function for the Julia sets. The function used in this study strongly favors complex images with few-to-no portions of the image actually in the Julia set. It would be simple to modify the function to favor larger fractions of the image being devoted to points that are in the Julia set. There are many potential fitness functions and exploring them is a distinct possibility for future work.

## REFERENCES


[1] D. Ashlock and B. Jamieson. Evolutionary computation to search mandelbrot sets for aesthetic images. *Journal of Mathematics and Art*, 3(1):147–158, 2008.

[2] Daniel Ashlock and Mark Joenks. ISAc lists, a different representation for program induction. In *Genetic Programming 98, proceedings of the third annual genetic programming conference.*, pages 3–10, San Francisco, 1998. Morgan Kaufmann.

[3] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming : An Introduction : On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann, San Francisco, 1998.

[4] J. J. Freeman. A linear representation for genetic programming using context free grammars. In J. R. Koza *et. al.*, editor, *Genetic Programming 1998*, pages 72–77. The MIT Press, 1998.

[5] Yaochu Jin. *Knowledge Incorporation in Evolutionary Computation*. Springer-Verlag, Berlin, Heidelberg, 2004.

[6] K. J. Kim and Cho Sung-Bae. Systematically incorporating domain-specific knowledge into evolutionary speciated checkers players. *IEEE Tansactions on Evolutionary Computation*, 9(6):615–627, 2005.

[7] John R. Koza. *Genetic Programming*. The MIT Press, Cambridge, MA, 1992.

[8] John R. Koza. *Genetic Programming II*. The MIT Press, Cambridge, MA, 1994.

[9] William B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer, New York, 2001.

[10] M. Lothaire. *Combinatorics on words*. Number 17 in Encyclopedia of Mathematics and Its Applications. Cambride University Press, London, 1997.

[11] J. F. Miller and S. L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, 2006.

[12] Astro Teller. The evolution of mental models. In Kenneth Kinnear, editor, *Advances in Genetic Programming*, chapter 9. The MIT Press, 1994.

[13] D. H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Computation*, 8(7):13411390, 1996.