Presented February 13, 1993 at the Symposium on Pattern Formation (SPF-93) at Claremont, California

Discovery of Rewrite Rules in Lindenmayer Systems and State Transition Rules in Cellular Automata via Genetic Programming

John R. Koza
Computer Science Department
Margaret Jacks Hall
Stanford University
Stanford, California 94305
Koza@Cs.Stanford.Edu
415-941-0336

Abstract: *It is difficult to write programs for both Lindenmayer systems and cellular automata. This paper demonstrates the possibility of discovering the rewrite rule for Lindenmayer systems and the state transition rules for cellular automata by means of genetic programming. Genetic programming is an extension of the genetic algorithm in which computer programs are genetically bred to solve problems. We demonstrate the use of genetic programming to discover the rewrite rules for a Lindenmayer system for the quadratic Koch island using a pattern matching measure as the driving force for the evolutionary process. We also demonstrate the use of genetic programming to discover the state transition rules for a one-dimensional and two-dimensional cellular automata using entropy as the driving force for the evolutionary process.*

1 Introduction and Overview

Interesting behavior often emerges from the repetitive application of seemingly simple rules. Both Lindenmayer systems and cellular automata often produce interesting emergent behavior. Lindenmayer systems are grammatical systems controlled by an initial condition and one or more rewriting rules. Cellular automata are dynamical systems controlled by an initial condition and a locally applied state transition rule. However, it is difficult to discover the rules that produce desired behavior in both Lindenmayer systems and cellular automata. This problem is called the “inverse” problem for Lindenmayer systems and cellular automata. This paper demonstrates the possibility of discovering the rewrite rule for Lindenmayer systems and the state transition rules for cellular automata by means of genetic programming. Genetic programming provides a way to search the space of all possible programs composed of certain terminals and primitive functions to find a function which solves, or approximately solves, a problem.

Section 2 of this paper provides background on genetic algorithms and genetic programming. Section 3 describes genetic programming. Section 4 describes Lindenmayer systems. Section 5 demonstrates the use of genetic programming to discover the rewrite rules for a Lindenmayer system for the quadratic Koch island using a pattern matching measure as the driving force for the evolutionary process. Section 6 demonstrates the use of genetic programming to discover the state transition rules for a one-dimensional and two-dimensional cellular automata using entropy as the driving force for the evolutionary process.

2 Background on Genetic Algorithms and Genetic Programming

John Holland's pioneering 1975 *Adaptation in Natural and Artificial Systems* described how the evolutionary process in nature can be applied to artificial systems using the genetic algorithm operating on fixed length character strings [Holland 1975]. Holland demonstrated that a population of fixed length character strings (each representing a proposed solution to a problem) can be genetically bred using the Darwinian operation of fitness proportionate reproduction and the genetic operation of recombination. The recombination operation combines parts of two chromosome-like fixed length character strings, each selected on the basis of their fitness, to produce new offspring strings. Holland established, among other things, that the genetic algorithm is a near optimal approach to adaptation in that it maximizes expected overall average payoff when the adaptive process is viewed as a multi-armed slot machine problem requiring an optimal allocation of future trials given currently available information. The genetic algorithm has proven successful at searching nonlinear multidimensional spaces in order to solve, or approximately solve, a wide variety of problems [Goldberg 1989, Davis 1987, Davis 1991, Davidor 1991, Forrest 1991, Michalewicz 1992]. Recent conference proceedings provide an overview of current work in the field [Schaffer 1989, Forrest 1990, Belew and Booker 1991, Rawlins 1991, Meyer and Wilson 1991, Schwefel et al. 1991, Langton et al. 1992, Whitley 1993].

It is difficult, unnatural, and overly restrictive to attempt to represent hierarchies of dynamically varying size and shape with fixed length character strings. For many problems, the most natural representation for a solution is a hierarchical composition of primitive functions and terminals (i.e., a computer program) of indeterminate size and shape, as opposed to character strings whose size has been determined in advance.

Genetic programming provides a way to find a computer program of unspecified size and shape to solve, or approximately solve, a problem. The book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* [Koza 1992] describes genetic programming in detail. A videotape visualization of applications of genetic programming can be found in the *Genetic Programming: The Movie* [Koza and Rice 1992]. Specifically, genetic programming has been successfully applied to problems such as

- classification and pattern recognition (e.g., distinguishing two intertwined spirals),
- evolution of a subsumption architecture for robotic control,

- discovering control strategies for backing up a tractor-trailer truck, centering a cart, and balancing a broom on a moving cart,
- generation of maximal entropy sequences of random numbers,
- empirical discovery (e.g., rediscovering the well-known non-linear econometric "exchange equation" $MV = PQ$ from actual, noisy time series data for the money supply, the velocity of money, the price level, and the gross national product of an economy),
- symbolic "data to function" regression, symbolic integration, symbolic differentiation, symbolic solution to general functional equations (including differential equations with initial conditions, and integral equations) and sequence induction,
- Boolean function learning (e.g., learning the Boolean 11-multiplexer function and 11-parity functions),
- planning (e.g., navigating an artificial ant along a trail),
- discovering inverse kinematic equations to control the movement of a robot arm to a designated target point,
- induction of decision trees for classification,
- optimization problems (e.g., finding an optimal food foraging strategy for a lizard),
- emergent behavior (e.g., discovering a computer program which, when executed by all the ants in an ant colony, enables the ants to locate food, pick it up, carry it to the nest, and drop pheromones along the way so as to produce cooperative emergent behavior),
- finding minimax strategies for games (e.g., differential pursuer-evader games, discrete games in extensive form) by both evolution and co-evolution, and
- simultaneous architectural design and training of neural networks.

3 Steps Required to Execute Genetic Programming

Genetic programming, like the conventional genetic algorithm, is a domain independent method. Genetic programming proceeds by genetically breeding populations of compositions of the primitive functions and terminals (i.e., computer programs) to solve problems by executing the following three steps:

(1) Generate an initial population of random computer programs composed of the primitive functions and terminals of the problem.

(2) Iteratively perform the following sub-steps until the termination criterion has been satisfied:

  (a) Execute each program in the population and assign it a fitness value according to how well it solves the problem.

  (b) Create a new population of programs by applying the following two primary operations. The operations are applied to program(s) in the population selected with a probability based on fitness (i.e., the fitter the program, the more likely it is to be selected).

    (i) *Reproduction*: Copy existing programs to the new population.

    (ii) *Crossover*: Create two new offspring programs for the new population by genetically recombining randomly chosen parts of two existing programs. The genetic crossover (sexual recombination) operation (described below) operates on two parental computer programs and produces two offspring programs using parts of each parent.

(3) The single best computer program in the population produced during the run is designated as the result of the run of genetic programming. This result may be a solution (or approximate solution) to the problem.

3.1 Crossover Operation

The crossover operation for the genetic programming paradigm is a sexual operation that operates on two parental LISP S-expressions and produces two offspring S-expressions using parts of each parent. Typically the two parents are hierarchical compositions of functions of different size and shape. In particular, the crossover operation starts by selecting a random crossover point in each parent and then creates two new offspring S-expressions by exchanging the sub-trees (i.e. sub-lists) between the two parents. Because entire sub-trees are swapped, this genetic crossover (recombination) operation produces syntactically and semantically valid LISP S-expressions as offspring regardless of which point is selected in either parent.

For example, consider the parental LISP S-expression:

```
(OR (NOT D1) (AND D0 D1))
```

And, consider the second parental S-expression below:

```
(OR (OR D1 (NOT D0))
    (AND (NOT D0) (NOT D1)))
```

These two LISP S-expressions can be depicted graphically as rooted, point-labeled trees with ordered branches. Assume that the points of both trees are numbered in a depth-first way starting at the left. Suppose that the second point (out of 6 points of the first parent) is randomly selected as the crossover point for the first parent and that the sixth point (out of 10 points of the second parent) is randomly selected as the crossover point of the second parent. The crossover points are therefore the `NOT` in the first parent and the `AND` in the second parent.

The two parental LISP S-expressions are shown are shown in figure 1. The numbers on the points of the trees are for reference only.

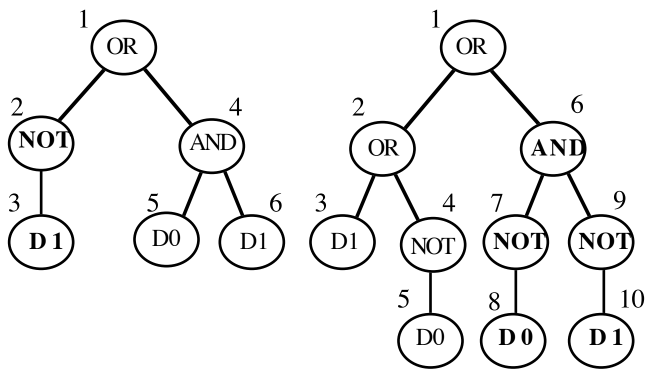

Figure 1 Two parental computer programs.

The two crossover fragments are two sub-trees shown in figure 2 and are underlined in the two parents above.

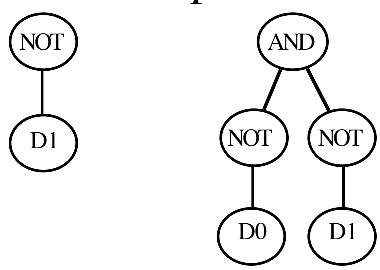

Figure 2 The crossover fragments resulting from selection of point 2 of the first parent and point 6 of the second parent as crossover points.

These two crossover fragments correspond to the bold, underlined sub-expressions (sub-lists) in the two parental LISP S-expressions shown above.

The first offspring S-expression is

(OR **(AND (NOT D0) (NOT D1))** (AND D0 D1)),

and happens to be the even-2-parity function.

The second offspring is

(OR (OR D1 (NOT D0)) **(NOT D1)**).

The two offspring resulting from crossover are shown in figure 3. Details can be found in Koza [1992].

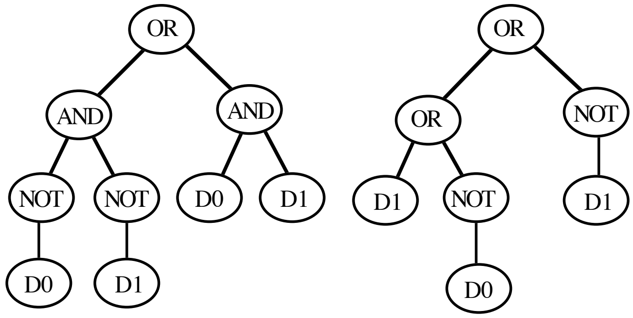

Figure 3 The two offspring produced by crossover.

4 Lindenmayer Systems

Lindenmayer systems (L-Systems) were conceived as a mathematical theory of plant development [Lindenmayer 1968]. L-systems are grammatical rewriting systems wherein a complex object can be defined by successively replacing parts of a simple initial object (the axiom) using one or more rewriting rules (productions) [Prusinkiewicz and Hanan 1980, Prusinkiewicz and Lindenmayer 1990].

In a deterministic context-free L-system, the predecessor string (left side) of each rewriting rule consists of only a single symbol of the alphabet and a particular symbol from the alphabet appears on the left side of only one rule. When the current string is rewritten, all occurrences of each symbol which can be rewritten are simultaneously replaced by its successor string (the right side of its rewriting rule). L-systems acquire their local and distributed character because of this simultaneous and parallel rewriting.

The complex objects generated by L-systems are often visualized by means of a geometric interpretation of the strings. In the turtle interpretation of strings, one symbol from the alphabet (say `F`) might be interpreted as a straight line segment and might be visualized by moving the turtle forward by one unit. The operation + (called `L+` later) might be interpreted by rotating the turtle in a positive direction (counterclockwise) by a specified angle ϕ whereas the operation − (called `L-` later) would be interpreted as a clockwise rotation by angle ϕ.

Without loss of generality, we assume the original straight line is of unit length and is initially oriented north throughout this paper.

4.1 Example of an L-System – The von Koch Snowflake

For example, consider the L-system for the von Koch snowflake [Prusinkiewicz and Lindenmayer 1990]. Three items need to specified. The axiom consists of the string F++F++F. The set of productions consists of the single rewriting rule

F → F-F++F-F.

The angle ϕ is 60°.

Figure 4 shows the turtle interpretation of the axiom (iteration 0) F++F++F for the von Koch snowflake.

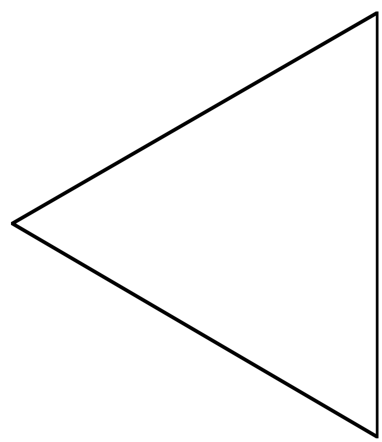

Figure 4. Iteration 0 (the axiom) F++F++F *for the von Koch snowflake*

Creation of iteration 1 requires three applications of the rewriting rule to the axiom (iteration 0) in order to produce the string for iteration 1, namely

F-F++F-F++F-F++F-F++F-F++F-F.

Figure 5 shows the turtle interpretation for iteration 1 of the von Koch snowflake. The size of the objects at later iterations generally grow, so we rescale all figures to the original size.

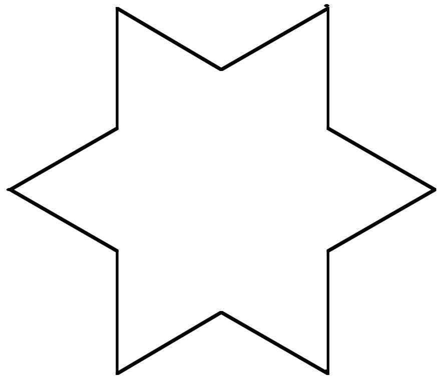

Figure 5 Iteration 1 of the von Koch snowflake

Figure 6 shows the turtle interpretation for the string produced when this rewriting process for the von Koch snowflake is continued to iteration 4.

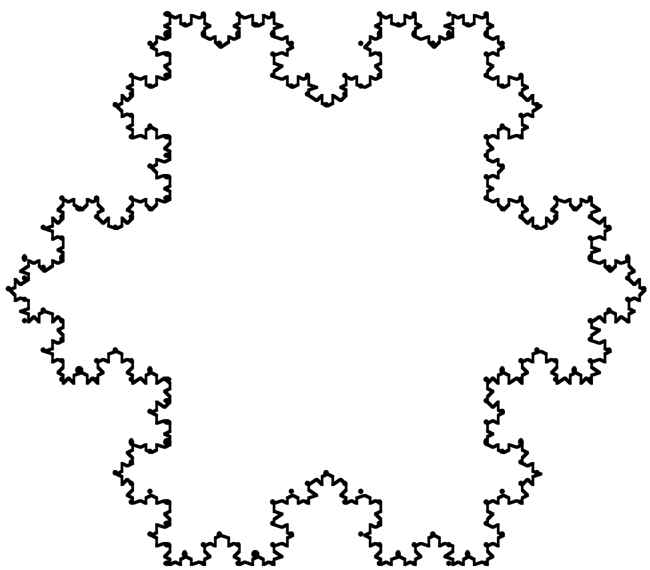

Figure 6 Iteration 4 of the von Koch snowflake

4.2 Example of a Plant Defined Using a Bracketed L-System

Bracketed L-system employ the additional feature of bracket. Upon completion of execution of a substring in a rewriting rule that is enclosed in a pair of brackets, the turtle is rubber-banded back to its position and orientation at the start of the execution of the bracketed substring.

For example, consider the L-system for a plant [Prusinkiewicz and Lindenmayer 1990]. The axiom consists merely of F. The angle ϕ is 26°. The single production is

```
F  →  F[+F]F[−F]F.
```

Figure 7 shows the turtle interpretation for iteration 1 of a plant using this bracketed L-system rule. The axiom (iteration 0) consists merely on a single vertical line segment. Iteration 1 is created by replacing the axiom with five line segments. The first `F` in the rewriting rule is interpreted as a line segment going forward in the current direction (i.e., north). The `[+F]` is interpreted as a counterclockwise (i.e., left) turn of 26° and a line segment in the new northwesterly direction. Because of the brackets, the turtle is rubber-banded back to its position and restored to its orientation prior to the left turn (i.e., north). The third `F` is interpreted as a line segment going forward in the now-restored northerly direction. The `[−F]` is interpreted as a clockwise (i.e., right) turn of 26° and a line segment in the new northeasterly direction. Similarly, because of the brackets, the turtle is rubber-banded back, so that the fifth and final `F` is interpreted as a line segment going forward in the now-restored northerly direction.

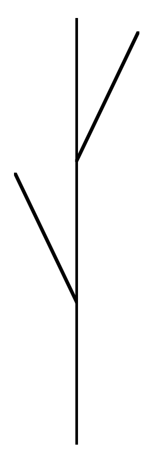

Figure 7 Iteration 1 of a plant using a bracketed L-system rule

Figure 8 shows the turtle interpretation for the string produced when this rewriting process for a plant using this bracketed L-system rule is continued to iteration 4.

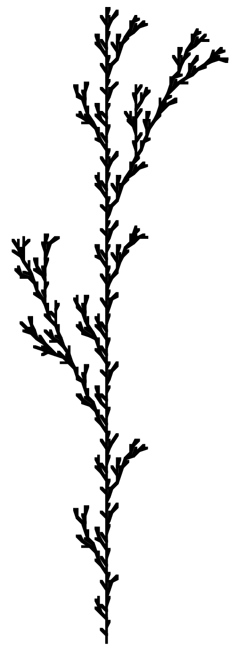

Figure 8 Iteration 4 of a plant using a bracketed L-system rule

5 The Inverse Problem for L-Systems

The "inverse" or "inference" problem for Lindenmayer systems involves finding the rewriting rules for a given structure or sequence of structures. In this paper, we consider a version of this inverse problem wherein the goal is to discover the rewriting rule, given the axiom and given the angle. Notice that we can, in general, determine the angle by measurement of the target object. In other words, we will be seeking a composition of the primitive functions and terminals (i.e., a computer program) that solves the problem. We will use genetic programming to discover the desired rewriting rule.

5.1 The Quadratic Koch Island

We will use the quadratic Koch island as the illustrative problem. The angle is 90° for the quadratic Koch island. The axiom is `F+F+F+F` (a square). The rewriting rule is

`F → F-F+F+FF-F-F+F.`

Figure 9 shows iteration 1 for the quadratic Koch island.

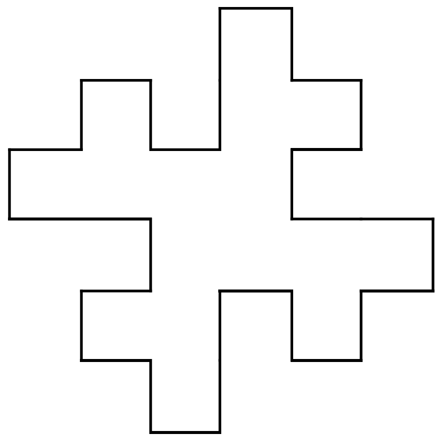

Figure 9 Iteration 1 for the Quadratic Koch island

Figure 10 shows iteration 2 for the quadratic Koch island.

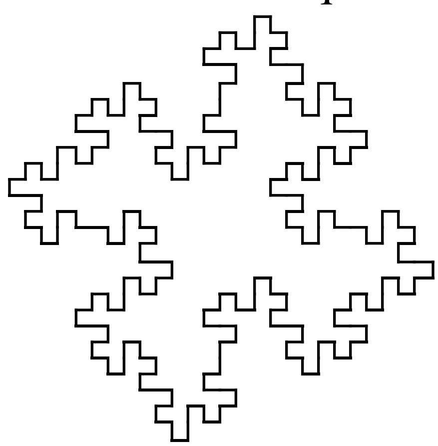

Figure 10 Iteration 2 for the Quadratic Koch island

5.2 Preparatory Steps for Using Genetic Programming

There are five major steps in preparing to use genetic programming, namely determining

(1) the set of terminals,
(2) the set of primitive functions,
(3) the fitness measure,
(4) the parameters for controlling the run, and

(5) the method for designating a result and the criterion for terminating a run.

The terminal set $\mathcal{T}$ for this problem can be viewed as consisting of three zero-argument functions of this problem.

$\mathcal{T}$ = {L+, L-, F}.

The function set $\mathcal{F}$ for this problem consists of

$\mathcal{F}$ = {BRACKET, PROGN},

taking one and two arguments, respectively.

The two-argument PROGN function is the ordinary LISP connective that evaluates (executes) both of its arguments in sequence.

The one-argument BRACKET function causes the turtle to execute the single argument of BRACKET and then rubber-bands the turtle to its position and orientation at the start of the evaluation of the BRACKET function. The rewriting rule for the quadratic Koch island does not necessarily require the use of brackets; however, as it happens, brackets appear in the solution discovered herein.

Each function-defining branch is a composition of primitive functions from the function set $\mathcal{F}$ and terminals from the terminal set $\mathcal{T}$.

The third major step in preparing to use genetic programming is the identification of the fitness measure for evaluating the goodness of each individual in the population. For this problem, fitness is measured according to how well a particular individual in the population matches the target quadratic Koch island. A minimal enclosing square is placed around the quadratic Koch island after iteration 2 and the enclosing square is divided into a 100 by 100 grid. The fitness is the number of cells in this grid (out of 10,000) for which the individual in the population behaves differently than the quadratic Koch island. The smaller this number, the better. A 100% correct individual would have a fitness of zero. Specifically, the fitness is incremented for each cell (1) entered by the quadratic Koch island on iteration 2 but not entered by the object on iteration 2, and (2) entered by the object on iteration 2 but not entered by quadratic Koch island on iteration 2. The magnitude of these increments are selected to be 9 and 1 respectively, so that an individual that draws no lines scores the same as an individual that paints lines in all 10,000 cells. The worst possible value of fitness is 17,808. This worst possible value is also assigned to any object that attempts to draw more than 1,280 line segments.

Although the above fitness measure is based on the matching of two patterns, genetic programming of Lindenmayer systems lends itself to fitness measures computed by means of biologically meaningful simulations involving scarce resources (e.g., sunlight falling on a plant) or to implicit fitness measures based on the competitive interactions of the growing objects in a simulated environment.

The fourth major step in preparing to use genetic programming is the selection of values for certain parameters. Our choice of 4,000 as the population size and our choice of 51 as the maximum number of generations to be run reflect an estimate on our part as to the likely difficulty of this problem and the practical limitations on available computer time and memory. Our choice of values for the various secondary parameters that control a run of genetic programming are the same default values as we have consistently used on numerous other problems [Koza 1992], except that we continue our recently adopted practice of using tournament selection (with a group size of seven) as the selection method (as opposed to fitness proportionate reproduction).

Finally, the fifth major step in preparing to use genetic programming is the selection of the criterion for terminating a run and the selection of the method for designating a result. We will terminate a given run if we encounter a 100% correct individual or after 51 generations. We designate the best individual obtained during the run (the best-so-far individual) as the result of the run.

5.3 Results for the Quadratic Koch island

A review of one particular successful run will serve to illustrate how genetic programming operates to solve this problem.

Genetic programming starts by randomly generating 4,000 individual compositions of the terminals form the terminal set and the functions from the function set for this problem. As one would expect, none of the 4,000 randomly generated individuals in the initial generation of the population (generation 0) are very good. The worst 12% of the population for generation 0 score the worst possible value, namely 17,808. They either do not draw anything at all or they exceed the maximum allowable number of line segments.

However, even in a randomly created population of programs, some individuals are better than others. For example, the next better individual from the 13% percentile of the population for generation 0 scores 9,252 and is

```
F → F+[-]-F-++--F-FF-.
```

Figure 11 shows this individual from the 13% percentile of the population for generation 0 for the quadratic Koch island.

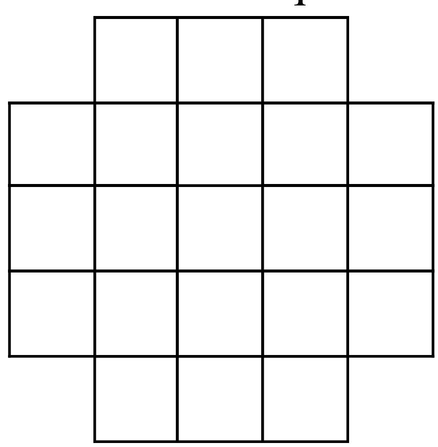

Figure 11 Individual from the 13th percentile of generation 0 for the quadratic Koch island

The second-best individual from generation 0 scores 7,776, and, when written in the style of L-systems, is

```
F → FFF+F-F-.
```

and is shown in figure 12.

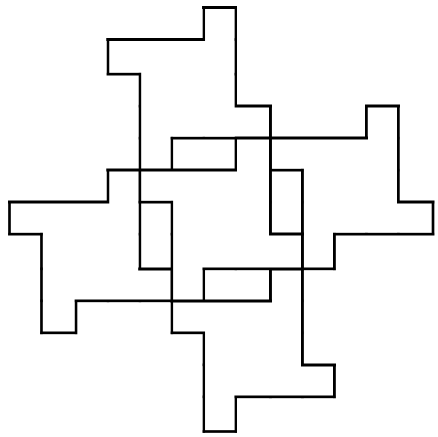

Figure 12 Second-Best individual from generation 0 for the quadratic Koch island.

The best-of-generation individual from generation 0 scores 6,522 and is

```
(PROGN (PROGN (PROGN (PROGN (F) (L+)) (PROGN (F) (F))) (PROGN (PROGN (L-) (L-)) (PROGN
(L+) (L+)))) (PROGN (PROGN (PROGN (L+) (F)) (PROGN (F) (L-))) (BRACKET (PROGN (F) (F))))).
```

When written in the style of L-systems, this best-of-generation individual from generation 0 is

```
F → F+FF--+++FF-[FF].
```

Figure 13 shows this best-of-generation individual from generation 0 for the quadratic Koch island.

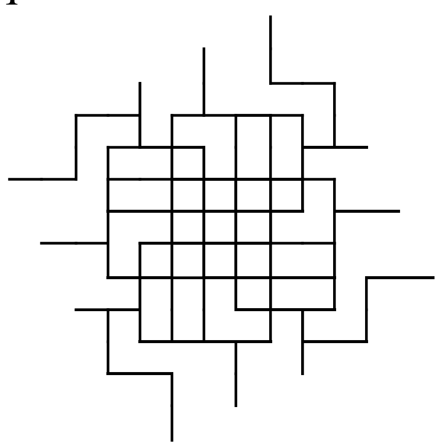

Figure 13 Best-of-generation individual from generation 0 for the quadratic Koch island

After applying the Darwinian reproduction operation and the genetic recombination (crossover) operations to the 4,000 individuals in the population, we find that the best-of-generation individual in generation 1 has an improved fitness of

6,446. When written in the style of L-systems, this individual is

```
F  →  F+FF[F]+FF-[FF].
```

Figure 14 shows this best-of-generation individual from generation 1 for the quadratic Koch island.

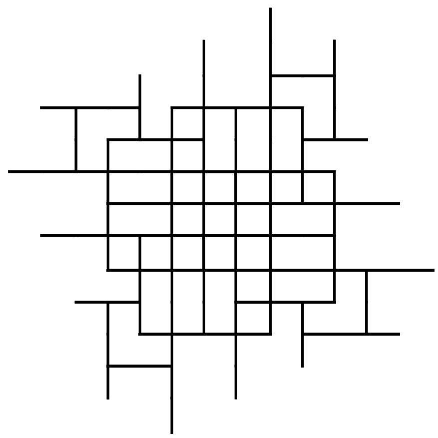

Figure 14 Best-of-generation individual from generation 1

As genetic programming proceeds from generation to generation, the population tends to improve.

Figure 15 shows that the best-of-generation individual from generation 3 has no lines in the central area of the object. The emergence of the empty central area is one step in the progress toward the eventual solution of this problem. This individual has fitness of 6,254.

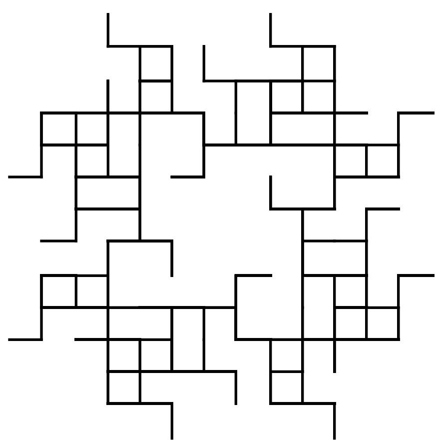

Figure 15 Best-of-generation individual from generation 3 with an empty central area

The best-of-generation individual from generation 5 has fitness of 4,488 and, when written in the style of L-systems, is

```
F → [-+F-F++---FF+-FFFF+++FF+]-F-FF-F+FF[+F].
```

Figure 16 shows this best-of-generation individual from generation 5. While this object does not yet bear much resemblance to the target quadratic Koch island, it is nevertheless better than its predecessors.

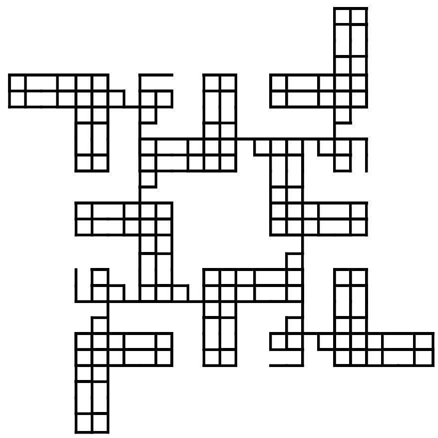

Figure 16 Best-of-generation individual from generation 5

The best-of-generation individual from generation 19 has fitness of 520 and, when written in the style of L-systems, is

```
F → [F][F-F+F--+++FF-[F][[-F]F]+-F+--F]FFFF.
```

Figure 17 shows this best-of-generation individual from generation 19. Notice the empty area in the central area now resembles a St. Andrew's cross (as it does in the target quadratic Koch island).

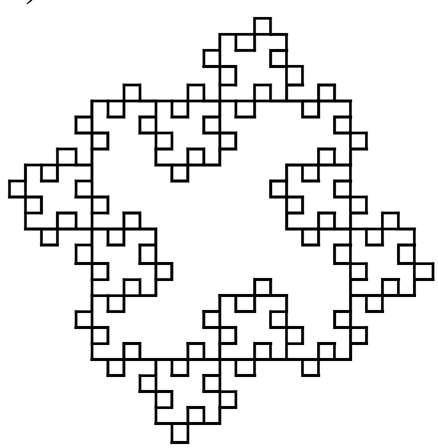

Figure 17 Best-of-generation individual from generation 19

Figure 18 shows, by generation, the fitness of the best-of-generation individual and the average fitness of the population as a whole. As can be seen, the fitness of the best-of-generation individual and the average fitness of the population as a whole tend to improve (i.e., drop) from generation to generation.

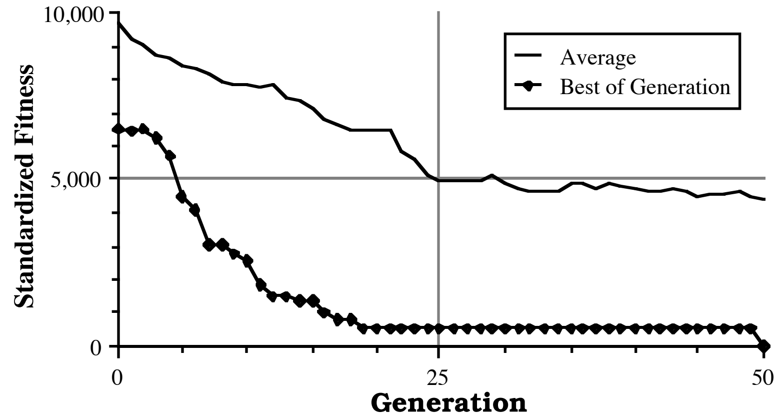

Figure 18 Fitness curves

Figure 19 shows, by generation, the structural complexity (i.e., number of functions and terminals) of the best-of-generation individual and the average structural complexity of the population as a whole for the LISP S-expressions (i.e., not the equivalent rules written in the form of a Lindenmayer system).

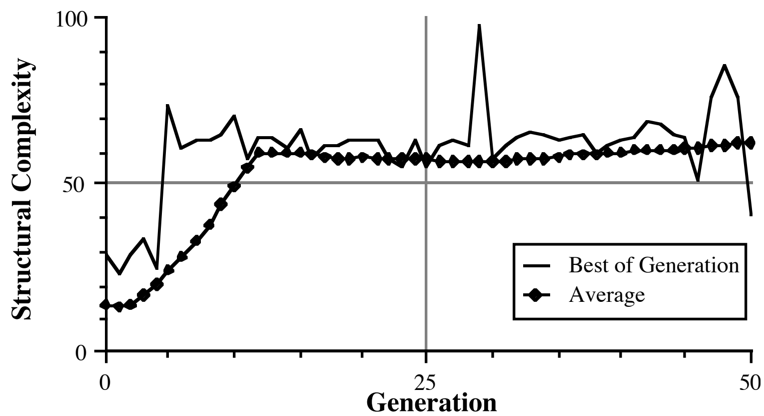

Figure 19 Structural complexity curves

The hits histogram is a useful monitoring tool for visualizing the progressive learning of the population as a whole during a run. To compute the number of hits awarded to an individual, we subtract the fitness (as defined above) of the individual from 17,808 (the worst possible fitness value) and express the result as a rounded percentage of the range 0 through 17,808. Thus, a perfect individual is awarded 100 hits, and the worst possible individual is awarded 0 hits. The horizontal axis of this histogram represents hits after they have been gathered into eleven bins. The first ten bins represent ten consecutive hits values each and the 11th bin represents 100 hits. The vertical axis represents the number of individuals in the population (0 to 4,000) scoring that number of hits.

Figure 20 shows the hits histograms for generations 0, 10, and 50 of this run. Notice the left-to-right undulating movement of both the high point and the center of mass of these histograms. This “slinky” movement reflects the improvement of the population as a whole. The number "1" on the third panel of this figure indicates that on generation 50 there was one individual that perfectly solved the problem (i.e., had fitness of zero and scored 100 hits).

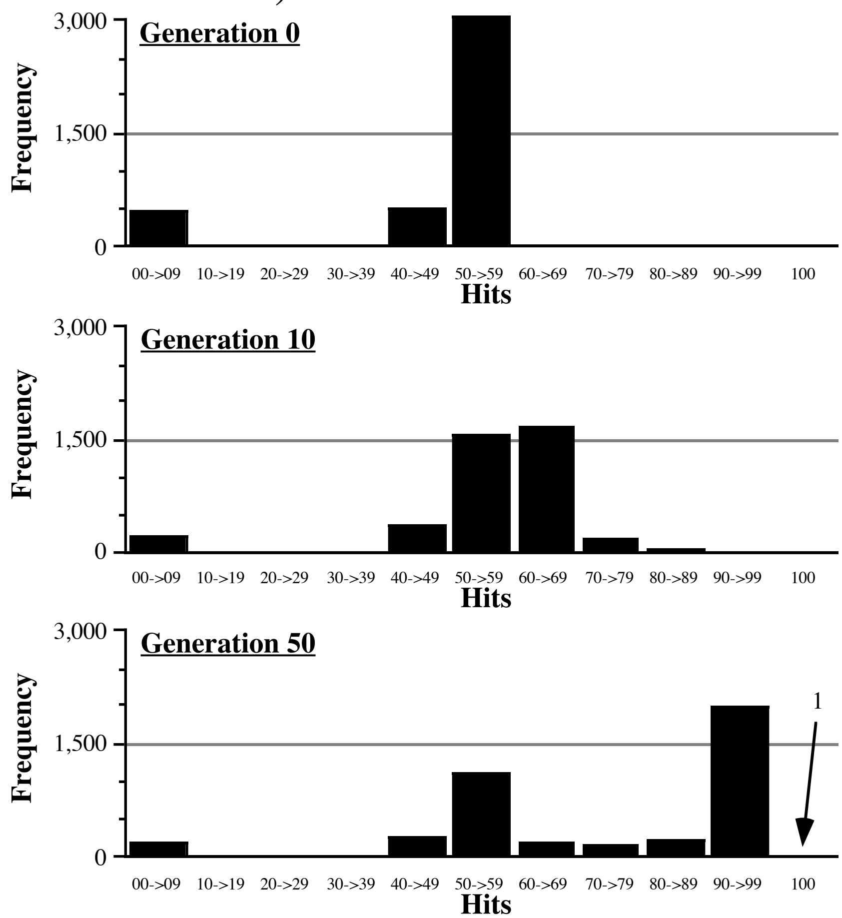

Figure 20 Hits histograms for generations 0, 10, and 50

Between generations 19 and 50, there is a different rewriting rule for each generation, each having almost perfect fitness.

By generation 50, the best-of-generation individual has the perfect fitness value of zero and is shown below:

```
(PROGN2 (PROGN2 (LM1-ADF0) (LM-)) (PROGN2 (PROGN2 (PROGN2 (PROGN2 (PROGN2 (PROGN2 (LM1-
ADF0) (LM+)) (LM1-ADF0)) (PROGN2 (PROGN2 (LM-) (LM-)) (PROGN2 (LM+) (LM+)))) (PROGN2
(PROGN2 (PROGN2 (LM+) (LM1-ADF0)) (PROGN2 (LM1-ADF0) (LM-))) (LM1-ADF0))) (HOMING-LM1
(PROGN2 (LM-) (LM1-ADF0)))) (PROGN2 (LM-) (PROGN2 (PROGN2 (LM1-ADF0) (LM+)) (LM1-
ADF0))))).
```

When written in the style of L-systems, this individual is

```
F  →  F-F+F--+++FF-F[-F]-F+F.
```

In this rewriting rule, the substring `--++` changes, but immediately restores, the turtle’s orientation and thus can be deleted from the above string. In addition, the bracketed sequence `[-F]` turns the turtle and draws a forward line; however, because it is bracketed and followed immediately by an identical `-F`, there is an overwriting and this bracketed string can be deleted. After these two deletions, the above string is seen to be a 100% correct string for the rewriting rule for the quadratic Koch island.

6 The Inverse Problem for Cellular Automata

In a cellular automaton, each cell in a cellular space is occupied by an automaton that is identical except for its initial state. The next state of each automaton depends on its own current state and on the current states of the automata in a specified set of neighboring cells. For example, for a one-dimensional cellular automaton, the next state of a given automaton might depend on the current state of that automaton and the current states of its two neighbors at distance 1. We denote these three states as `X` (for the automaton at the center), `W` (west), and `E` (east). Similarly, for a two-dimensional cellular automaton, the next state of a given automaton might depend on the current state of that automaton and the current states of its four neighbors at distance 1 in the two-dimensional space, namely `X`, `W`, `E`, north (`N`), and south (`S`). Cellular spaces typically have periodic boundary conditions (i.e., are toroidal) so that every cell has the same number of neighbors.

Cellular automata are the discrete counterparts of continuous dynamical systems defined by partial differential equations and the physicist's concept of field [Gutowitz 1991, Toffoli and

Margolus 1987]. If the automaton located in each cell happens to have only two states, the state-transition function of the automaton is merely a Boolean function. For a one-dimensional cellular space with von Neumann neighbors, the Boolean function has three inputs and one output. For a two-dimensional cellular space with von Neumann neighbors, the Boolean function has five inputs and one output. Cellular automata with Boolean state-transition functions are dynamical systems that are discrete in time, space (their cells), and site value (Boolean).

Complex overall behavior is often produced by cellular automata as the result of the repetitive application (at each cell in the cellular space) of seemingly simple transition rules contained in each cell.

The problem of designing a state-transition rule that, when it operates in each cell of the cellular space, produces a desired overall emergent behavior is called the "inverse" problem for cellular automata.

The "inverse" problem for cellular automata involves finding the state-transition rule that, when it operates in each cell of the cellular space, produces a desired overall behavior. In this paper, we consider a version of this inverse problem wherein the goal is to discover the state-transition rule, given the initial condition of the cellular space. In other words, we will be seeking a composition of the primitive functions and terminals (i.e., a computer program) that solves the problem. We will use genetic programming to discover the desired state-transition rule.

6.1 One-Dimensional Cellular Automata

In this section, we use genetic programming to evolve a state-transition rule that enables a cellular automaton to produce certain desired emergent behavior. In particular, we evolve a state-transition rule that produces temporal random behavior in a cellular automaton.

Wolfram [1986] showed that a particular two-state automaton depending only on itself and its two immediate neighbors (W and E) in a one-dimensional cellular space was capable of producing a pseudo-random temporal stream of bits. In particular, Wolfram showed random temporal behavior (using several frequently used tests for randomness) from the state-transition rule

```
(XOR W (OR X E)).
```

This Boolean function with three inputs is rule 30 using the usual numbering scheme for Boolean functions. It is, under reflection, equivalent to rule 86.

We used a one-dimensional cellular space of width 32. The initial state of cell 15 was 1 (True) and the initial state of all other cells was 0 (NIL). The initial state of the cellular space used by Wolfram consisted of one cell in state 1 (True) and all the other 31 cells in state 0 (NIL). In other words, the initial state contained a minimal amount of activity. The temporal stream of random bits was taken from the single cell that started in state 1 (i.e., cell 15).

In this section, we demonstrate how genetic programming can rediscover Wolfram's two-state automaton using only the overall goal (i.e., to produce a high-entropy stream of bits over time) to guide the discovery process.

The terminal set for this problem consisted of the three inputs available to each automaton in the one-dimensional cellular space, namely

```
T = {X, W, E}.
```

Since we are considering functions of three Boolean arguments, the function set for this problem can consist of the following computationally complete and convenient set of three Boolean functions:

```
F = {AND, OR, NOT}
```

taking two, two, and one argument, respectively.

Fitness is measured by means of entropy. We examined the time series over 4,096 time steps at cell 15 and considered the entropy associated with the probability of occurrence of each of the $2^4 = 16$ possible temporal subsequences of length 4. That is, there were 4,096 fitness cases. If each of the 16 subsequences of length 4 occurred exactly $\frac{4{,}096}{16} = 256$ times in 4,096 time steps, entropy would attain the maximal value of 4.000 bits. Fitness was measured via entropy using a lookback of 4. Maximum raw fitness was 4.000 bits. A hit for this problem was defined as 1,000 times raw fitness and thus ranged from 0 to 4,000.

The population is 500 for this cellular automata problem.

The genetically produced S-expressions in the population are often large and complex. Nonetheless, they involve only the three independent variables `X`, `W`, and `E`, and therefore they necessarily correspond to one of the 256 possible Boolean functions with three arguments and one output.

In one run, the best-of-generation individual from generation 0 had an entropy of 1.832 and 32 points:

```
(AND (AND (NOT (AND (NOT E) (OR E X))) (NOT (AND (AND X E) (NOT X)))) (NOT (AND (OR (OR X
X) (OR W W)) (AND (OR W W) (AND W W))))).
```

In figure 21, the horizontal axis ranges over the $2^4 = 16$ possible temporal subsequences of length 4 for generation 0 (i.e., from 0000 to 1111). The vertical axis of this histogram ranges over the number of occurrences of each of the 16 subsequences. As can be seen, the most frequent two of the possible temporal subsequences of length 4 occur 1,792 and 1,664 times each (out of 4,096 times), and many of the possible subsequences are unrepresented for generation 0.

Generation 0

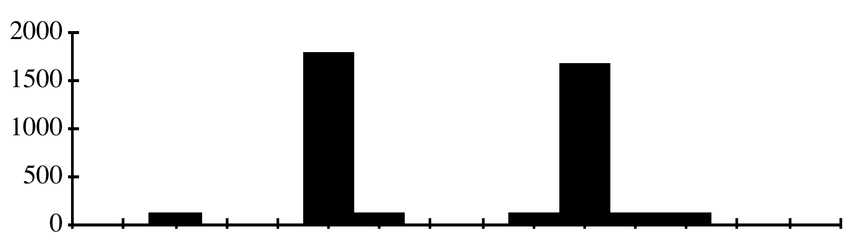

Figure 21 Subsequence histogram for the best-of-generation individual for generation 0 for the one-dimensional cellular automaton problem.

For generation 6, the best-of-generation individual had an entropy of 3.494 and 20 points:

```
(AND (OR E X) (NOT (AND (AND (AND X X) (OR W X)) (AND (AND W W) (AND W E))))).
```

Figure 22 is the histogram for the 16 possible temporal subsequences for generation 6. Six of the possible temporal subsequences of length 4 occur between 502 and 504 times each for generation 6. Generation 6 is the first generation of this particular run for which there was at least one occurrence of each of the 16 possible temporal subsequences (although the fact that the number of occurrences of the several of the rarer subsequences is non zero is not discernible on this histogram, because of its scale).

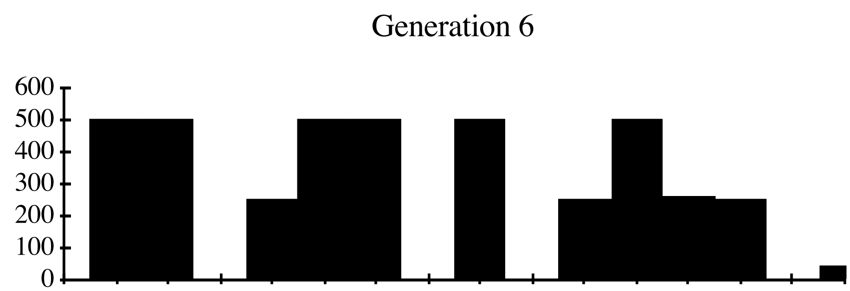

Figure 22 Subsequence histogram for the best-of-generation individual for generation 6 for the one-dimensional cellular automaton problem.

For generation 7, the best-of-generation individual had an entropy of 3.645 and 28 points:

```
(OR (NOT (OR (NOT W) (NOT (OR (NOT W) (OR E X))))) (NOT (OR (OR (NOT (NOT X)) (OR E W))
(OR (OR X W) (AND W X))))).
```

Figure 23 is the histogram for the 16 possible temporal subsequences for the best-of-generation 7 individual. As can be seen, there has been a substantial improvement in the uniformity of the distribution between generations 6 and 7. Fifteen of the 16 subsequences in generation 7 have between 214 and 289 occurrences, and one of the subsequences has 425 occurrences.

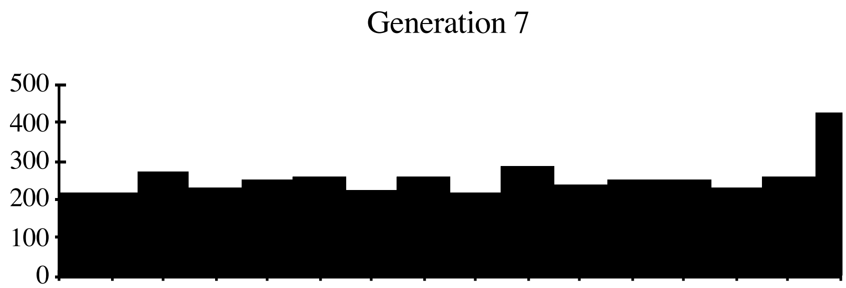

Figure 23 Subsequence histogram for the best-of-generation individual for generation 7 for the one-dimensional cellular automaton problem.

For generation 10, the best-of-generation individual had an entropy of 3.982 and 43 points:

```
(OR (AND (AND (NOT (NOT (NOT X))) (OR (NOT (AND E W)) (OR (NOT (OR (NOT E) (AND X X)))
(NOT E)))) (NOT (AND (OR (NOT E) (NOT W)) (OR (OR X W) (AND E E))))) (NOT (OR (NOT X) (AND
X W)))).
```

Figure 24 is the histogram for the best-of-generation 10 individual. The numbers of occurrences of all 16 of the subsequences lie in the relatively narrow range of 232 to 275.

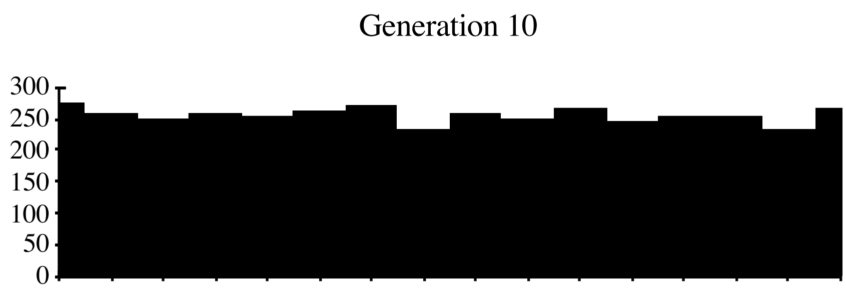

Figure 24 Subsequence histogram for the best-of-generation individual for generation 10 for the one-dimensional cellular automaton problem.

The best-of-generation individual for generation 25 had entropy of 3.996. Its histogram is similar to, but smoother than, the histogram in figure 24 for generation 10. This best-of-run individual has 83 points and an entropy of 3.996:

```
(AND (OR (OR (NOT (OR E E)) (NOT (OR (OR (AND W W) (OR E (NOT (NOT X)))) (NOT (AND (AND
(AND X X) X) (AND (AND W W) (AND W E))))))) (OR (AND W W) (AND E E))) (OR (NOT (OR (NOT
(OR (NOT W) (NOT (OR (NOT W) (OR E X))))) (NOT (OR (OR (NOT (NOT X)) (OR E W)) W)))) (NOT
(OR E (OR (OR (NOT W) (AND (OR X X) (NOT E))) (AND (OR X X) (AND X E))))))).
```

Table 1 shows that this S-expression is rule 30 (00011110 in binary) and is therefore equivalent to Wolfram's cellular automaton randomizer.

Table 1 Truth table of best-of-run individual from generation 25 for the one-dimensional cellular automaton problem.

West	X	East	Result
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

The question arises as to whether the above S-expression, which was genetically bred using $4{,}096 = 2^{12}$ temporal sequence steps, is generalizable to other numbers of steps. When we retested the genetically produced randomizer over $65{,}536 = 2^{16}$ steps, we got an even better value of entropy: 4.000 bits (as compared to the original 3.996).

On an earlier generation of this same run, we also encountered Boolean rule 45 that Wolfram (1986) identified as the second-best randomizer of this type (when inserted into a one-dimensional cellular automaton). The S-expression for rule 45 (which is, upon reflection, equivalent to rule 75) is

```
(XOR W (OR X (NOT E))).
```

6.2 Two-Dimensional Cellular Automata

We can genetically breed a randomizing computer program for a two-dimensional cellular automaton in a similar manner.

The terminal set for this problem consisted of the five inputs from the von Neumann neighborhood available to each automaton in the two-dimensional cellular space, namely

```
T = {X, W, N, E, S}.
```

The function set and fitness measure are the same as for the one-dimensional cellular automata problem previously described.

We used a two-dimensional cellular space of size 8 × 8. The initial state of cell (3, 3) was 1 (True) and the initial state of all other 63 cells was 0 (NIL). We examined the time series over 16,384 time steps and considered the entropy associated with the probability of occurrence of each of the $2^7 = 128$ possible subsequences of length 7. Maximum entropy is now 7.000 bits.

In one run, the best-of-generation individual from generation 0 had an entropy of 3.202 (out of 7.000 bits) and has 4 points:

```
(NOT (OR X E)).
```

Figure 25 is a histogram showing the number of occurrences of each of the $2^7 = 128$ possible subsequences of length 7 occurring temporally at cell (3,3) for the best-of-generation individual from generation 0. The horizontal axis represents the possible temporal subsequences from 0000000 to 1111111 at cell (3,3).

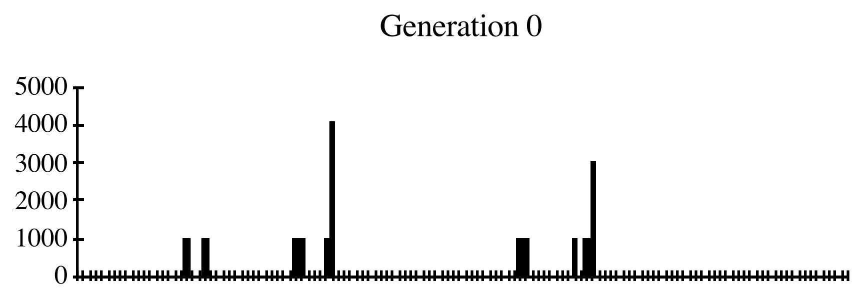

Figure 25 Subsequence histogram for the best-of-generation individual for generation 0 for the two-dimensional cellular automaton problem.

For generation 7, the best-of-generation individual had 51 points and an entropy of 6.711:

```
(OR (OR (AND (OR (NOT E) (NOT S)) (OR (OR E (AND X W)) S)) (AND (NOT (OR N X)) (OR (NOT W)
(AND W (NOT (OR (AND X (NOT (NOT N))) (NOT S))))))) (OR (AND (NOT (NOT N)) (NOT (OR N X)))
(AND (AND (NOT W) (NOT X)) (NOT E)))).
```

Figure 26 is the histogram for the 128 possible temporal subsequences for generation 7, the first generation of this particular run for which there was at least one occurrence of each of the 128 possible temporal subsequences.

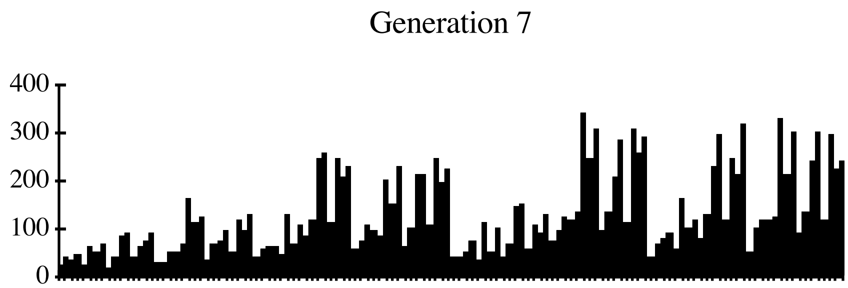

Figure 26 Subsequence histogram for the best-of-generation individual for generation 7 for the two-dimensional cellular automaton problem.

The best-of-generation individual for generation 10 had 67 points and an entropy of 6.995:

```
(OR (OR (AND (OR (NOT E) (NOT S)) (OR (AND (OR (NOT E) (NOT S)) (OR (AND W X) E)) (NOT (OR
(AND (OR X N) (NOT E)) (OR X W))))) (AND (NOT (OR X S)) (OR (NOT W) (AND W (NOT (OR (AND X
(NOT (NOT N))) (NOT S))))))) (OR (AND (NOT (NOT N)) (NOT (OR N X))) (AND (AND (NOT W) (NOT
X)) (NOT E))))).
```

Figure 27 is the histogram for generation 10.

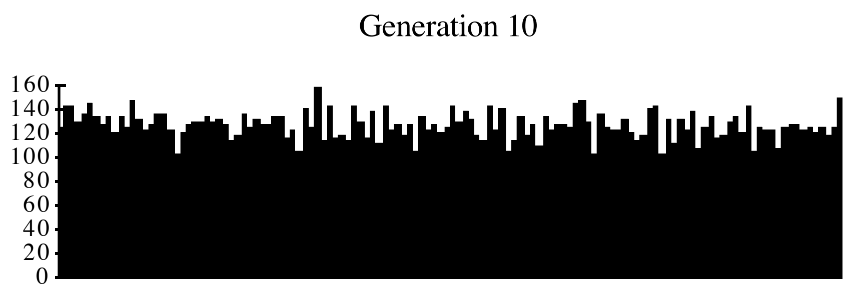

Figure 27 Subsequence histogram for the best-of-run individual from generation 10 for the two-dimensional cellular automaton problem.

The question arises as to whether the above S-expression, which was genetically bred using $16{,}384 = 2^{14}$ temporal sequence steps, is generalizable to other numbers of steps. When we retested the genetically produced randomizer over $65{,}536 = 2^{16}$ steps, we got an even better value of entropy: 6.998 bits (as compared to the original 6.995).

Table 2 shows the rule number and entropy for six selected genetically discovered high-entropy rules for a two-dimensional cellular automaton. For example, line A of this table shows the best-of-generation individual with entropy of 6.995 for generation 10 of the run described above. The S-expression for this rule is equivalent to rule number $2{,}857{,}758{,}960_{10}$ in the numbering scheme for two-dimensional cellular automata used in Toffoli and Margolus [1987]. In this numbering scheme, the bits are presented in the order `EWSNX` and the inputs are taken starting with 00000 (i.e., the opposite to the order employed for one-dimensional cellular automata described in the previous section). The decimal and hexadecimal identifications of each rule are shown in columns 2 and 3 of this table and the entropy is shown in column 4.

Table 2 Selected high-entropy rules for the two-dimensional cellular automaton problem.

	Rule number in decimal notation	**Rule number in hexadecimal notation**	**Entropy**
A	2,857,758,960	`AA55F0F0`	6.995
B	4,042,268,190	`F0F01E1E`	6.997
C	3,435,935,286	`CCCC3636`	6.997
D	4,027,577,610	`F00FF50A`	6.997
E	3,435,947,622	`CCCC6666`	6.995
F	3,140,699,340	`BB3344CC`	6.995

Table 3 is the truth table for these same six genetically discovered high-entropy rules for a two-dimensional cellular automaton. The first five columns of this table present the specific combination of inputs in the order `E`, `W`, `S`, `N`, and `X`. The next six columns show the value of the six rules for each of the 32 combinations of values of the inputs.

Table 3 Truth table for selected genetically produced high-entropy rules for the two-dimensional cellular automaton problem.

E	W	S	N	X	A	B	C	D	E	F
0	0	0	0	0	1	1	1	1	1	1
0	0	0	0	1	0	1	1	1	1	0
0	0	0	1	0	1	1	0	1	1	1
0	0	0	1	1	0	1	0	1	1	1
0	0	1	0	0	1	0	1	0	0	1
0	0	1	0	1	0	0	1	0	0	0
0	0	1	1	0	1	0	0	0	0	1
0	0	1	1	1	0	0	0	0	0	1
0	1	0	0	0	0	1	1	0	0	0
0	1	0	0	1	1	1	1	0	0	0
0	1	0	1	0	0	1	0	0	0	1
0	1	0	1	1	1	1	0	0	0	1
0	1	1	0	0	0	0	1	1	1	0
0	1	1	0	1	1	0	1	1	1	0
0	1	1	1	0	0	0	0	1	1	1
0	1	1	1	1	1	0	0	1	1	1
1	0	0	0	0	1	0	0	1	1	0
1	0	0	0	1	1	0	0	1	1	1
1	0	0	1	0	1	0	1	1	1	0
1	0	0	1	1	1	1	1	1	1	0
1	0	1	0	0	0	1	0	0	0	0
1	0	1	0	1	0	1	1	1	1	1
1	0	1	1	0	0	1	1	0	0	0
1	0	1	1	1	0	0	0	1	1	0
1	1	0	0	0	1	0	0	0	0	1
1	1	0	0	1	1	0	0	0	0	1
1	1	0	1	0	1	0	1	0	0	0
1	1	0	1	1	1	1	1	0	0	0
1	1	1	0	0	0	1	0	1	1	1
1	1	1	0	1	0	1	1	0	0	1
1	1	1	1	0	0	1	1	1	1	0
1	1	1	1	1	0	0	0	0	0	0

Interestingly, rule E in this table (i.e., rule 3,435,947,622) was produced on two different runs (out of 11 runs that produced rules with entropy of 6.995 or better). The S-expression obtained on one of those two runs was

```
(NOT (AND (OR (AND (OR (OR (AND (OR (OR (OR S (OR (OR W (NOT E)) N)) N) N) (AND (OR (OR
(OR (OR (AND (OR W N) (NOT E)) (NOT (NOT X))) (AND S X)) (NOT (NOT X))) (AND S X)) (NOT
(NOT (AND (OR W N) (NOT E)))))) (NOT (NOT X))) (AND S X)) (NOT (NOT N))) (NOT (OR (OR X N)
(NOT E)))) (OR (AND (NOT (OR (OR W (NOT (NOT X))) N)) (OR (OR (AND (OR (OR W (NOT (NOT
X))) N) (AND S X)) (AND (AND S S) (OR (OR (AND (OR W N) (NOT E)) (NOT (NOT X))) (AND S
X)))) (NOT (OR (AND (AND (OR (AND S X) N) (AND S X)) (NOT E)) (AND (AND S S) (OR (OR W (OR
W W)) N)))))) (OR (OR W (OR W W)) N)))).
```

7 Conclusions

We used genetic programming to find a rewriting rule for a Lindenmayer system for the quadratic Koch island and to find a high entropy state transition rule for both a one-dimensional and two-dimensional cellular automata.

ACKNOWLEDGEMENTS

James P. Rice of the Knowledge Systems Laboratory at Stanford University programmed the above on the Texas Instruments Explorer II+ computer.

REFERENCES

Belew, Richard and Booker, Lashon (editors) *Proceedings of the Fourth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc. 1991.

Davidor, Yuval. *Genetic Algorithms and Robotics*. Singapore: World Scientific 1991.

Davis, Lawrence (editor) *Genetic Algorithms and Simulated Annealing* London: Pittman l987.

Davis, Lawrence. *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold 1991.

Forrest, Stephanie (editor). *Emergent Computation: Self-Organizing, Collective, and Cooperative Computing Networks*. Cambridge, MA: The MIT Press 1990.

Forrest, Stephanie. *Parallelism and Programming in Classifier Systems*. London: Pittman 1991.

Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley l989.

Gutowitz, Howard (editor). *Cellular Automata: Theory and Experiment*. Cambridge, MA: The MIT Press 1991.

Holland, John H. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press 1975. Revised Second Edition 1992 from The MIT Press.

Koza, John R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: The MIT Press 1992. 1992.

Koza, John R. and Rice, James P. *Genetic Programming: The Movie*. Cambridge, MA: The MIT Press 1992.

Langton, Christopher, Taylor, Charles, Farmer, J. Doyne, and Rasmussen, Steen (editors). *Artificial Life II, SFI Studies in the Sciences of Complexity*. Volume X. Redwood City, CA: Addison-Wesley 1992.

Lindenmayer, Aristid. Mathematical models for cellular interactions in development, I & II. *Journal of Theoretical Biology*. 18: 280–315. 1968.

Meyer, Jean-Arcady and Wilson, Stewart W. *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*. Paris. September 24-28, 1990. Cambridge, MA: MIT Press 1991.

Michalewicz, Zbignlew. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag 1992.

Prusinkiewicz, Przemyslaw and Hanan, James. *Lindenmayer Systems, Fractals, and Plants*. New York: Springer-Verlag 1980.

Prusinkiewicz, Przemyslaw, and Lindenmayer, Aristid. 1990. *The Algorithmic Beauty of Plants*. New York: Springer-Verlag 1990.

Rawlins, Gregory (editor). Proceedings of Workshop on the Foundations of Genetic Algorithms and Classifier Systems. Bloomington, Indiana. July 15-18, 1990. San Mateo, CA: Morgan Kaufmann 1991.

Schaffer, J. D. (editor). *Proceedings of the Third International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc. 1989.

Schwefel, Hans-Paul and Maenner, Reinhard (editors). *Parallel Problem Solving from Nature*. Berlin: Springer-Verlag. 1991. Pages 124-128. 1991b.

Toffoli, T. and Margolus, N. *Cellular Automata Machines*. Cambridge, MA: The MIT Press, 1987.

Wolfram, Stephen. Random sequence generation by cellular automata. In Wolfram (editor). *Theory and Applications of Cellular Automata*. Singapore: World Scientific 1986.