Revised July 4, 1993 for Proceedings of Third Workshop on Artificial Life (ALIFE-3).

Spontaneous Emergence of Self-Replicating and Evolutionarily Self-Improving Computer Programs

John R. Koza

Computer Science Department
Margaret Jacks Hall
Stanford University
Stanford, CA 94305-2140 USA
Koza@CS.Stanford.Edu
415-941-0336
FAX 415-941-9430

ABSTRACT

This chapter reports on the spontaneous emergence of computer programs exhibiting the ability to asexually reproduce, to reproduce by combining parts from two parents, and to improve their performance through evolution from a primordial ooze of primitive computational elements. The computational elements are a computationally complete set and compositions of them are capable of agglomerating themselves to one another.

1. Introduction

John von Neumann (Burks 1987), E. F. Codd (1968), Thatcher (1970), John Devore (Devore and Hightower 1992), Christopher Langton (1983), Thomas Ray (1991a, 1991b, 1991c, 1991d), Skipper (1992), Rasmussen et. al. (1990, 1991) and others have designed various self-reproducing automata and computer programs. Each of these self-replicating entities required considerable human ingenuity to design. None of these self-reproducing entities were small enough or simple enough that one could expect to conduct an experiment in any reasonable amount of time in which the entity might spontaneously emerge by means of a blind generative search. The reasonableness of the amount of computer time required to witness spontaneous emergence is important because it sheds some light on the origins of self-reproducing entities and, as a practical matter, because it determines whether it is possible to conduct computer experiments concerning the phenomenon of spontaneous emergence (either now or in the foreseeable future).

In addition, although several of the above self-reproducing entities designed by humans are computationally universal, none of them are programmed to exhibit evolutionarily self-improving behavior nor do they have the ability to learn. In contrast, living organisms exhibit both self-improving behavior and the ability to learn.

As Farmer and Belin (1991) state,

"Discovering how to make ... self-reproducing patterns more robust so that they can evolve to increasingly more complex states is probably the central problem in the study of artificial life."

Moreover, all of the above self-reproducing entities are very brittle in that they do not correctly function if there is almost any perturbation from the structure that their human programmer specified. In contrast, living organisms exhibit considerable ability to continue to function under perturbation.

This chapter addresses the question as to whether self-replicating computer programs can spontaneously emerge from a primordial ooze of primitive computational elements by means of a blind random generative process within a reasonable amount of computer time. And, if so, can such spontaneously emergent self-replicating programs exhibit evolutionarily self-improving behavior and learning ability? In this chapter, we answer both questions affirmatively.

2. Overview

In nature, the evolutionary process occurs when the following four conditions are satisfied:

- An entity must have the ability to reproduce itself.
- There must be a population of such self-reproducing entities.
- There must be some variety among the population.
- There must be some difference in ability to survive in the environment associated with the variety.

In a population of entities, the presence of some variability that is associated with some difference in the survival rate is almost inevitable. As Charles Darwin observed in *On the Origin of Species by Means of Natural Selection* (1859),

"I think it would be a most extraordinary fact if no variation ever had occurred useful to each being's own welfare ... . But if variations useful to any organic being do occur, assuredly individuals thus characterised will have the best chance of being preserved in the struggle for life; and from the strong principle of inheritance they will tend to produce offspring similarly characterised. This principle of preservation, I have called, for the sake of brevity, Natural Selection."

Thus, the first of the above four conditions (the ability to self-replicate) is, in practice, the crucial one for starting the evolutionary process. Once the entity is self-reproducing, evolution is the inevitable result of naturally occurring variations that are correlated with different rates of survival and reproduction. Of course, evolution is not purposive in the sense that it seeks to reach some predefined goal. Instead, the different rates of survival and reproduction of individuals in their environment produce, over time, changes in the genetic makeup of the population (Buss 1987, Dawkins 1987, John Maynard Smith 1989).

This chapter presents an example of a population of computer programs learning to perform a simple task wherein the individuals in the population are active (as they are in nature) and where the fitness measure is implicit (as it is in nature). The essential ingredient for the individuals to be active is that they have the ability to reproduce themselves (either alone or in conjunction with a mate). Specifically, this chapter explores the organizational complexity required for a computational structure to engage in three levels of behavior: asexual reproduction (i.e., self-replicability), sexual reproduction (wherein parts of two parents genetically recombine to produce a new offspring which inherit traits from both parents), and evolutionarily self-improving behavior (i.e., learning).

If an individual has the ability to perform or partially perform some task beyond self-reproduction and if performance or partial performance of that task increases its probability of survival and reproduction (or increases its number of offspring), then the processes of natural

selection and evolution exert selective pressure in favor of that individual. Thus, populations of self-reproducing individual computer programs are able to learn to perform tasks.

3. Background on Self-Reproducing Automata and Programs

In his 1949 lectures at the University of Illinois, John von Neumann explored the abstract question of what level of organizational complexity is required for self-replication to occur. Von Neumann did not finish or publish his work on this subject prior to his death, but Arthur Burks, a colleague of von Neumann, extensively edited and completed many of von Neumann's manuscripts on this subject. Burks (1970) states that von Neumann in his 1949 lectures inquired as to

"What kind of logical organization is sufficient for an automaton to reproduce itself? This question is not precise and admits to trivial versions as well as interesting ones. Von Neumann ... was not trying to simulate the self-reproduction of a natural system at the level of genetics and biochemistry. He wished to abstract from the natural self-reproduction problem its logical form."

By abstracting out the chemical, biological, and mechanical details of the molecular structures that successfully engage in self-replication, von Neumann was able to highlight the essential requirements for the self-replicability of a structure. In particular, von Neumann demonstrated the possibility of self-replicability of a computational structure by actually designing a self-reproducing automaton consisting of a two-dimensional cellular arrangement containing a large number of individual 29-state automata cells (Burks 1966, 1970, 1987). The next state of each 29-state automaton was a function of its own current state and the state of its four neighbors (`N`, `E`, `W`, and `S`) in the two-dimensional cellular space.

Von Neumann designed his self-reproducing automaton to perform the functions of a universal Turing machine and showed how to embed both a computation-universal automaton and its associated input tape into the two-dimensional cellular space. His computation-universal automaton was *a fortiori construction universal* in the sense that it was capable of reading the input tape (composed of cells of the cellular space), interpreting the data on the tape, and using a constructing arm to construct the configuration described on the tape in an unoccupied part of the cellular space. His automaton was also capable of backspacing the tape, constructing a copy of the tape, attaching the copy to the configuration just constructed, signaling to the configuration that the construction process had finished, and retracting the constructing arm. By putting a description of the constructing automaton itself on the input tape, von Neumann created a self-reproducing automaton (Kampis 1991; Arbib 1966; Myhill 1970; Thatcher 1970; Alvy Ray Smith 1991, Kemeny 1955). Although never implemented, von Neumann's self-reproducing automaton was very large (involving many millions of cells).

Von Neumann's self-reproducing automaton treats the information on the input tape to the Turing machine in two distinct ways (Langton 1984, 1986). First, the information on the input tape is actively interpreted as instructions to be executed by the constructor in order to cause the construction of a particular configuration in an unoccupied part of the cellular space. Second, the information on the input tape is interpreted as data which is to be passively copied, in an uninterpreted way, to become the tape of the new machine. In other words, the information on the tape is interpreted as both a program and as data at different stages of the overall process.

Although von Neumann's 1949 lectures predated the discovery and elucidation by Watson and Crick in 1953 of the self-replicating structure and genetic role of deoxyribonucleic acid (DNA), the dual role of the information found in von Neumann's cellular automaton has a direct analogy in nature. In particular, the process of actively constructing a configuration based on the information contained on the input tape is analogous to the translation of the nucleotide bases

into a chain of amino acids constituting a protein. And, the process of passively copying the input tape is analogous to the transcription of the nucleotide bases of DNA to messenger ribonucleic acid (mRNA) or to a complementary strand of DNA (Schuster 1985; Bagley and Farmer 1991). As Rasmussen et al. (1991) observes, in chemical reactions involving molecules, entities interact in such a way that the entity can serve as both the entity executing a reaction (the program) and the entity being acted upon by the reaction (the data).

E. F. Codd (1968) reduced the number of states required for a computationally universal, self-reproducing automaton from 29 to only eight. Codd's self-reproducing automaton occupied about 100,000,000 cells (Devore and Hightower 1992, Codd 1992). In the early 1970s, John Devore's self-reproducing automaton simplified Codd's automaton and occupied only about 94,794 cells (Devore and Hightower 1992). Thatcher (1970) designed another self-reproducing automaton.

Langton (1983) observed that the capability of computational universality found in the self-reproducing automata of von Neumann, Codd, Devore, and Thatcher is not known to be present in any self-replicating molecules (which may be the building blocks of the earliest and simplest forms of life) or in the biological structure of any known living entity. Consequently, Langton achieved a substantial reduction in size by abandoning computation universality. In particular, Langton designed a self-reproducing cellular automaton that occupied an area of only 100 cells in its cellular space. Like Codd's and Devore's automata, each cell in Langton's automaton had only eight states. Like the designs of von Neumann, Devore, and Codd, each cell in Langton's self-reproducing automaton communicated only with its four neighbors. However, unlike the other self-reproducing automata, the coded description of Langton's self-reproducing automaton was not on a static tape, but instead endlessly circulated in a manner reminiscent of the delay-line storage devices used in early computers. A computer simulation of Langton's automaton can be seen on videotape (Langton 1991b).

All of the above self-reproducing cellular automata take advantage of the particular transition function applied to all cells in the cellular space (just as molecular self-replication takes advantage of the particular physical laws governing the molecules involved). If the particular transition function is too powerful, the issue of self-reproduction becomes trivial. Langton's design, like those of von Neumann, Codd, Devore, and Thatcher avoided triviality by requiring that the responsibility for construction of the copy be actively directed by instructions residing primarily in the configuration itself. In addition, Langton's design shared an important characteristic with the other designs and with DNA, namely that the stored information has dual roles in that it is treated both as instructions to be actively interpreted and executed and as data to be passively copied into the new automaton. In addition, triviality is avoided because the crucial copying event copies only one primitive element at a time.

Ray (1991a, 1991b, 1991c) wrote a clever 80 line self-reproducing computer program in assembly code and demonstrated how his program could evolve over time as a consequence of mutation. A visualization of this work appears as a videotape in Ray (1991d). Ray wrote his program in a special assembly language (Tierra) for a special virtual machine. Ray's virtual machine was intentionally imperfect and introduced random mutations to his original hand-written 80-line program (called the "ancestor"). Ray observed the emergence, over a period of hundreds of millions of time steps, of an impressive variety of different entities (some self-reproducing, some parasitically self-reproducing, some symbiotically self-reproducing, and some not self-reproducing) and a dazzling array of biological phenomena including parasitism, defenses against parasitism, hyperparasitism, symbiosis, and social parasitism.

In Ray's Tierra system, certain assembly-code instructions can search memory backwards or forwards from their location for the first occurrence of a sequence of consecutive `NOP` (No-

Operation) instructions having a specified sequence of bits in their one-bit data fields. The bit pattern used in such searches is called a *template*. As in nature, complementary matching is performed with the template. In writing his self-reproducing program, Ray put a certain identifying four-bit template at the beginning of his program and another identifying four-bit template at its end. Ray's program is capable of performing a self-examination to locate these two templates. His program is then able to calculate its own size by subtracting (using registers of his computer) the memory addresses of the two templates discovered by this self-examining search. Ray then wrote a loop in which each instruction between the now-located beginning and now-located end of his program was moved (copied), one instruction at a time, from its location in memory into an unoccupied area of memory. Thus, Ray's program is able to make an exact copy of itself. A special assembly-code instruction `MAL` ("Memory Allocation") causes the operating system to allocate an unoccupied area of memory to an offspring and a special cell division instruction activates that new area as an independent new entity. Ray's program uses data in the same dual way used by DNA and the designs of von Neumann, Codd, Thatcher, Devore, and Langton in that it first actively executes its 80 assembly-code instructions as instructions at one point and it passively copies its 80 instructions as data at another point. However, in contrast to DNA and the other designs, Ray's program does not use a separate coded description of the entity to be reproduced; instead, it uses the program itself as its own description. Laing (1976, 1977) discusses such "automata introspection" where there is no separate description of the entity being reproduced.

An important design feature of Ray's Tierra system is that it is never necessary to refer to a numerical memory address (either absolute or relative) in the operand of an instruction. Ray achieves this in two ways.

First, he avoids numerical memory addresses entirely as the operands of his assembly-code instructions by designing his virtual machine with only four registers, a flag bit, a program counter, and one small stack. Thus, both the operation code of each assembly-code instruction and its associated operand (if any) fits into only five bits. Some of Ray's operation codes occupy three of the five bits while others occupy four bits. For the operations whose operation code occupies three bits, the remaining two bits designate a particular one of four registers to be used in conjunction with the operation. For the operations whose operation code occupies four bits, the remaining bit is a data bit.

Second, Ray's biologically-motivated scheme for template matching eliminates the need to refer to a location in memory by means of a numerical memory address. Specifically, a template consists of consecutive `NOP` instructions, each with a one-bit data field. A searching template is deemed to have found a match in memory if it finds consecutive `NOP` instructions whose data bits are complementary to the data bits of the searching template. The search is limited to a specified small bounded area of memory. Thus, it is possible to transfer control (i.e., jump) to a particular location in memory without ever referring to a specific numerical address by searching memory for the nearest occurrence in one direction of a matching (i.e., complementary) template. Similarly, a particular location in memory (such as the beginning and end of the program) may be located without referring to its specific numerical memory address in the operand of an assembly-code instruction.

Holland's ECHO system (1990, 1992) for exploring evolution in a miniature world of co-evolving creatures employs a similar template-matching scheme. In Holland's system, a search template (consisting of all or part of a binary string) is deemed to match another binary string if the bits match (starting from, say, the left) for the number of positions they have in common.

Skipper (1992) has presented a model of evolution incorporating several of the above features, including template matching.

Self-reproducing computer programs have been written in FORTRAN, C, LISP, PASCAL, and many other languages (Bratley and Millo 1972; Burger, Brill, and Machi 1980). Computer viruses and worms are self-reproducing programs that operate within various computers (Dewdney 1985, 1987, 1989; Spafford 1991). Programs entered into the Core Wars tournaments typically have self-replicating features (Dewdney 1984, 1987; Rasmussen et al. 1990, 1991).

In nature, carbon-based life forms exploit energy available from the environment (primarily the sun) to organize matter available from the environment in order to survive, reproduce, and evolve. This exploitative and organizational process operates within the constraints of the rules of physical interaction governing the matter involved, notably the rules of physical interaction of atoms of carbon, hydrogen, oxygen, nitrogen, and other elements found in organic molecules.

The field of artificial life contemplates alternatives to the single carbon-based paradigm for life found on earth. One such alternative involves computational structures (such as programs within the digital environment of a computer) that exploit computer time to organize memory in order to survive, reproduce, and improve themselves (Langton 1989; Langton et al. 1991; Langton 1991a, 1991b). This exploitative and organizational process operates within the constraints of the rules of interaction governing the milieu of computational structures. In this analogy, computer time corresponds to energy available from the environment and computer memory corresponds to matter available from the environment.

4. Common Features of Self-Reproducing Automata and Programs

All of the foregoing self-reproducing automata and programs have the crucial common feature of creating a copy of a given item at one critical point in their operation. In each instance, they create the copy by exploiting the ability of the milieu in which they are operating to change the state of some part of the computer memory (in the manner permitted by the rules of interaction of the milieu) to a specified desired new state, which is, in each instance, a copy of something. That is, there is an assumption that there is sufficient free matter and free energy available in the milieu to create a copy of a specified thing.

In the self-reproducing cellular automata of von Neumann, Codd, Devore, and Thatcher, the copy is created by interpreting the coded description on the tape as a sequence of instructions which cause the construction, cell by cell, of the new machine in an unoccupied part of the cellular space. The copying actually occurs at the moment when the state of some previously quiescent cell immediately adjacent to the tip of the constructing arm is changed to the desired new state. This crucial state transition can occur because the rules of the milieu permit it and because there is sufficient free matter and free energy available in the milieu to permit the creation of a copy of the desired thing. The free matter arises from the plasticity of the state of each cell in the cellular space. That is, every cell is free to change to any desired new state in accordance with the underlying physics (i.e., the state transition rules of the cellular space). The force that causes the change of state corresponds to the free energy of the system (i.e., computer time). Free energy must be supplied to the system to enable it to change states. The copying is done without affecting or depleting the original coded description in any way because of this availability of free matter and free energy. The fact that the automaton is copied at a different time than the coded description on the tape emphasizes the dual use of the coded description on the tape.

In Langton's self-reproducing cellular automaton, the coded description circulates endlessly in a channel of cells. For most points along the channel, there is only one cell that is poised (either immediately ahead or around a corner) to receive whatever was emitted by its immediately

adjacent predecessor in the channel. However, at one particular point in the channel, there are two automata that are poised to receive whatever is emitted by the immediately adjacent predecessor. At this fan-out point in the channel, one copy of the coded description continues circulating around the original channel while another copy of the original description goes off toward an unoccupied part of the cellular space. The copying actually occurs at the moment when the states of the two previously quiescent cells adjacent to the cell at the fan-out point are changed to the specified new state. This crucial state transition can occur because the rules of the milieu permit it and because there is sufficient free matter (i.e., states of the cellular space) and free energy (i.e., the force that causes the change of state) available in the milieu to create a copy of a specified thing (i.e., the copying is done without affecting or depleting the original coded description).

In Ray's self-reproducing program, the 80-instruction program serves as its own description and the `MOVE-INDIRECT` instruction inside an iterative loop make a copy of what the program finds from a self-examination. Inside the iterative loop, each assembly-code instruction of the program is copied, one by one, from its location in memory into an unoccupied area of memory. Copying occurs when the state of a memory location in an unoccupied (or inactive) area of memory is changed to the specified new state by the `MOVE-INDIRECT` instruction. This crucial state transition can occur because the rules of the milieu permit it and because there is sufficient free matter and free energy available in the milieu to create a copy of a specified thing without affecting or depleting that which is copied.

In each instance, the copying of the entire entity is done cell by cell over many time steps.

Fontana (1991a, 1991b) and Rasmussen et. al. (1990, 1991) have addressed similar issues.

In nature, an individual strand of DNA is able to replicate itself because there is a plentiful supply of nucleotide bases available in the milieu to bind (in a complementary manner) to the bases of both of the existing strands of DNA when the DNA unwinds. Similarly, messenger RNA (mRNA) can be translated by a ribosome into a string of amino acids (a protein) by means of the genetic code because there is a plentiful supply of *transfer RNA* (tRNA) available in the milieu to bind (in a complementary manner) to the codons of the messenger RNA.

5. Organizational Complexity of Existing Self-Reproducing Automata and Programs

Each of the above self-reproducing computer programs was conceived and written by a human programmer using an impressive amount of ingenuity. None is small and none is simple.

Von Neumann recognized that a certain "minimum number of parts" was a necessary precondition for self-reproduction. See also Dyson 1985. Von Neumann observed that a machine tool that stamps out parts is an example of a synthesizer that is more complex than that which it synthesizes. The machine tool is

"an organization which synthesizes something [that is] necessarily more complicated ... than the organization it synthesizes," so that "complication, or [re]productive potentiality in an organization, is degenerative" (Burks 1987).

Von Neumann also recognized that living organisms, unlike the machine tool, can produce things as complicated as themselves (by means of reproduction) and can produce things more complicated than themselves (by means of evolution). Von Neumann concluded

"there is a minimum number of parts below which complication is degenerative, in the sense that if one automaton makes another, the second is less complex than the first, but above which it is possible for an automaton to construct other automata of equal or higher complexity" (Burks 1987).

It would appear that the "minimum number of parts" to which von Neumann referred must be exceedingly large (Holland 1976). Certainly von Neumann's, Codd's, Thatcher's, and Devore's self-reproducing automata each had a very large number of parts. If the minimum number of parts for a self-reproducing entity is large, that entity is going to be very rare in the space of possible entities of which it is an instance. Therefore, the probability of finding that entity in a random search of that space of entities is very small. Accordingly, the probability of spontaneous emergence of that entity is very small.

The self-reproducing programs written by von Neumann, Codd, Thatcher, Devore, Langton, and Ray are each just one program from an enormous space of possible computer programs. It would not be possible to find any of these programs in any reasonable amount of time by means of any kind of blind random search of the space of programs.

We can get a rough idea of the probability of finding one of their programs in a blind random search if we make four simplifying assumptions. First, suppose that we ignore the milieu itself in making this rough calculation (e.g., the rules of the cellular space or the rules governing the computer and operating system executing the assembly-code instructions). Second, suppose we limit the search to only entities whose size is no larger than the precise size that the human programmer involved actually used. In other words, since Langton's design occupies 100 cells in a cellular space, we consider only entities with 100 or fewer cells. Third, suppose we limit the search to spaces that do not contain any features beyond the minimum set of features actually used by the programmer involved. Fourth, suppose that we ignore all symmetries and equivalences in functionality (i.e., we assume that there is only one such solution).

Von Neumann's 29-state self-reproducing automaton has many millions of cells. An apparently grossly erroneous underestimate of 200,000 cells is cited in Kemeny (1955). Using this underestimate, the probability of discovering this particular automaton in a blind random generative search is still one in $29^{200,000} \approx 10^{292,480}$.

Codd's eight-state self-reproducing automaton has about 10^8 cells (Devore and Hightower 1992). Using the above simplifying assumptions, the probability of discovering this particular automaton in a blind random generative search is about one in $8^{10,000,000} \approx 10^{9,000,000}$. Thatcher's self-reproducing automaton is also very large.

Devore's version of Codd's program fits into a rectangle of 259 cells by 366 cells (i.e., 94,794 cells).. The probability of randomly generating this particular automaton is about one in $8^{94,794} \approx 10^{85,315}$.

Langton's program is much smaller and is known to occupy 100 cells. Therefore, the probability of randomly generating this automaton is one in $8^{100} \approx 10^{91}$.

Ray's hand-written program consists of 80 lines of 5-bit assembly-code instructions. Therefore, the probability of randomly generating this automaton is one in $32^{80} \approx 10^{120}$.

Interestingly, Ray observed the evolution, over hundreds of millions of time steps, of a self-reproducing program consisting of only 22 assembly code instructions in the Tierra language. The probability of randomly generating this evolutionarily-derived self-reproducing program is one in $32^{22} \approx 10^{33}$.

The rough calculations above suggest that the probability of creating a self-reproducing computer program at random must be exceedingly small. That is, the chance is remote that a self-reproducing computer entity can spontaneously emerge merely by trying random compositions of the available ingredients (i.e., states of an automaton or assembly-code instructions). The probability of randomly generating Langton's automaton and the probability of randomly generating the evolutionarily-derived 22-line program are the least remote. However, even these probabilities (i.e., 10^{91} and 10^{33}, respectively) are far too remote to permit

spontaneous emergence with existing available computer resources or any foreseeable computer resources or other resources for conducting simulations and experiments. For example, even if it were possible to test a billion (10^9) points per second and if a blind random search had been running since the beginning of the universe (i.e., about 15 billion years), it would be possible to have searched only about 10^{27} points.

However, once spontaneous emergence of a self-replicating entity occurs in a particular milieu, the situation changes. As von Neumann recognized, once created, self-reproducing structures can not only reproduce, but multiply. He noted,

"living organisms are very complicated aggregations of elementary parts, and by any reasonable theory of probability or thermodynamics, highly improbable ... [however] if by any peculiar accident there should ever be one of them, from there on the rules of probability do not apply, and there will be many of them, at least if the milieu is reasonable"

Von Neumann further noted,

"produc[ing] other organisms like themselves ... is [the] normal function [of living organisms]. [T]hey wouldn't exist if they didn't do this, and it's plausible that this is the reason why they abound in the world" (Burks 1987).

Darwin recognized that once a population of different, self-replicating structures has been created, the ones that are fitter in grappling with the problems posed by the environment tend to survive and reproduce at a higher rate. This natural selection causes the evolution of structures that are improvements over previous structures.

In nature, individuals in the population that do not respond effectively to a particular combination of environmental conditions suffer a diminished probability of survival to the age of reproduction. That is, the environment communicates a negative message (in the extreme, immediate death) to unfit entities. If entities in the population have a finite lifetime (which is the case in this chapter), this means that the potential offspring of entities that do not respond effectively to the environment do not become a part of the population for the next generation.

The size, complexity, and intricacy of von Neumann's, Codd's, Thatcher's, Devore's, Langton's, and Ray's self-reproducing entities suggests that the organizational complexity required for self-replicability is very high, and, therefore, the possibility of spontaneous emergence of self-replicability (much less sexual reproduction and evolutionary self-improving behavior) is very low.

In the remainder of this chapter, we show that the probability of spontaneous emergence of self-reproducing and self-improving computer programs is very much more likely than the foregoing discussion would suggest. In fact, we will demonstrate experimentally that spontaneous emergence of self-reproducing and self-improving computer programs is possible, in an appropriate milieu, with a fast workstation.

The design of this experiment is severely constrained by considerations of the available computer resources. Accordingly, certain compromises must be made to improve the probability of spontaneous emergence. Each compromise is accompanied by an explanation of the approach that could be pursued if there were no such considerations of the available computer resources. However, it will be seen that each of the rare events in which we are interested would nevertheless have occurred within a reasonably small amount of time (albeit more time than we have available on our computer), even if the compromises had not been made.

The set of basic computational operations that we chose are not the most elementary, and they do not purport to accurately model the chemical, biological, or mechanical details of the actual organic molecules that carry out asexual reproduction, sexual reproduction, and evolutionary self-improving behavior in carbon-based life forms. However, these basic computational operations perform functions that bear some resemblance to the functions performed by organic

molecules in nature, and they perform these functions in a way that bears some resemblance to the way organic molecules perform in nature.

Spontaneous emergence of computational structures capable of complex behavior might occur in a sea of randomly created computer program fragments that interact in a random way. Such a sea and the turbulent intermixing of entities in this sea of entities has some degree of biological plausibility since biological macromolecules randomly move and randomly come into contact in their cellular milieu.

One of the basic operations permits the incorporation of one entire computer program into another. That is, larger entities can grow by means of an agglomeration of smaller entities. This agglomerative operation is a highly simplified form of recombination (crossover). Another basic operation permits the emission of new entities into the sea.

The approach used here bears some relation to both the genetic algorithm (Holland 1975, 1992) and genetic programming (Koza 1989, 1991, 1992) as illustrated on videotape (Koza and Rice 1991a, 1992). In genetic methods, the calculation of the fitness measure is usually an explicit calculation and the individuals in the population are passive. The fitness of each individual is measured by means of some explicit calculation and the controller of the genetic process applies various genetic operations to the passive individuals on the basis of that computed fitness. The controller exists outside of the individuals. The user of a genetic method may think of the fitness of a living organism as a numerical quantity reflecting the probability of reproduction weighted by the number of offspring produced (the fecundity); however, this numerical fitness measure is not known to the individual or explicitly used by the individual. Instead, this fitness value is used by the outside controller to execute the genetic process.

In contrast, in nature, fitness is implicit and the individuals are active. There is no explicit numerical calculation of fitness. Individuals act independently without centralized control or direction. If the individual is successful in grappling with its environment, it may survive to the age of reproduction and reproduce. Moreover, the individuals in the population are active in that each individual has the capacity for reproduction (either alone or in conjunction with a mate). The fact that they reproduce means that they are fit, and their fecundity indicates the degree of their fitness.

6. Primitive Functions and Terminals from which the Computational Entities are Composed

Our focus here will be on a primordial ooze of primitive computational entities composed of functions from a function set $\mathcal{F}$ and of terminals from a terminal set $\mathcal{T}$.

The terminal set consists of three terminals and is

$\mathcal{T}$ = {`D0`, `D1`, `D2`}.

The three terminals `D0`, `D1`, and `D2` can be thought of as sensors of the environment. They are variables that can evaluate to either `NIL` (i.e., false) or something other than `NIL` (i.e., `T` = true).

The function set contains eight functions and is

$\mathcal{F}$ = {`NOT`, `SOR`, `SAND`, `ADD`, `SST`, `DWT`, `LR`, `SR`},

taking one, two, two, one, one, two, zero, and zero arguments, respectively.

The first three of the functions in $\mathcal{F}$ (NOT, SOR, and SAND) together constitute a computationally complete set of Boolean functions. They have no side effects.

The function NOT takes one argument and performs an ordinary Boolean negation.

The function SOR ("Strict OR") takes two arguments and performs a Boolean disjunction in a way that combines the behavior of the OR macro from Common LISP and the PROGN special form from Common LISP. In particular, SOR begins by unconditionally evaluating *both* of its arguments in the style of a PROGN function (thereby executing the side effects, if any, of *both* arguments) and then returns the result of evaluating the first of its two arguments that evaluates to something other than NIL. If both arguments evaluate to NIL, SOR returns NIL. In contrast, once the ordinary OR in Common LISP finds an argument that evaluates to something other than NIL, it does not evaluate any remaining argument(s) and therefore does not execute any side effects of those remaining argument(s).

The function SAND ("Strict AND") takes two arguments and performs a Boolean conjunction in a "strict" manner similar to SOR.

Each of the other five functions (ADD, SST, DWT, LR, and SR) have return values; however, as we will see, their real functionality lies in their side effects. The ADD ("agglomerate") and SST ("Search Sea for Template") functions act as identify functions insofar as their return values are concerned (i.e., they return the result of evaluating their one argument). The DWT ("Do «WORK» until «TRAILING-TEMPLATE» matches") function returns the result of evaluating its second argument (i.e., it acts as an identity function on its second argument); its return value does not depend at all on its first argument. The return value of the LR ("Load REGISTER") function is T and the return value of the SR ("Store REGISTER") function is NIL.

Three global variables are used in conjunction with the SST, DWT, LR and SR functions: MATCHED-TREE, OPEN-SPACE, and REGISTER. Each is initially NIL when evaluation of a given program begins.

There is a sea (population) of computer programs each composed of ingredients from $\mathcal{F}$ and $\mathcal{T}$. As a program circulates in the sea of computer programs, both the ADD function and the SST function search the ever-changing sea for a program whose top part matches the top part of the argument subtree of the ADD or the SST, much as a freely-moving biological macromolecule circulates in its milieu by means of random diffusion and seeks other molecules with a receptor site that matches (in a complementary way) its own receptor site. If the search is successful, the ADD function and the SST function then perform their specified side effects.

Only the top part of the argument subtree of the ADD function and SST function is used in the search; that part is called the *template*. This method of template matching has a degree of biological plausibility and is an extension into the space of program trees from the space of the linear bit strings in Holland's ECHO templates and the space of linear strings of assembly-code instructions in Ray's Tierra templates. See also Skipper (1992).

Two points from the argument subtree of the ADD function and the SST function are used in the comparison with the other programs in the sea. The template consists of the top point of the argument subtree of the ADD or SST and the point immediately below the top point on the leftmost branch of the argument subtree. If the argument subtree of an ADD or SST contains only one point (i.e., is insufficient), the search fails and no side effect is produced.

If there is a side-effecting ADD or SST in the template of an ADD or SST, the ADD or SST in the template is not executed. Instead, the ADD or SST being executed merely uses the ADD or SST in the template symbolically in trying to make a match. The points in the argument subtree of an ADD or SST other than the two points constituting the template are not relevant to the

template matching process in any way. In particular, if there is another ADD or SST located inside the argument subtree, but beneath the template, they do not initiate any searching.

The comparison is made only to the top part (the *site*) of the program being searched (consisting of the root of the program tree being searched and the point immediately below the root on the leftmost branch).

The sea (called a *Turing gas* by Fontana 1991a, 1991b) is turbulent in the sense that the immediate neighbor of a given program is constantly changing. The search of the sea initiated by an ADD or SST begins with the program itself and then proceeds to the other programs in the sea. A template is never allowed to match to itself (i.e., to the argument subtree of the very ADD or SST currently being executed). The search of the sea continues until 100% of the programs in the sea have been searched.

When the SST function ("Search Sea for Template") searches the sea of programs, and the SST function finds a match, the global variable MATCHED-TREE is bound to the entire matching program.

If the template of the ADD function of the program being executed successfully matches the top part of a program in the sea, the ADD function has the side effect of substituting the entire matching program into the program being executed. Specifically, the entire matching program replaces the ADD being executed and the entire argument subtree of the ADD (i.e., not just the template of the ADD). In other words, the entire matching program is permanently incorporated into the physical structure of the program being executed, so the physical size of the program being executed usually increases as a consequence of the ADD function. The inserted program is not executed at the time of its agglomeration. However, since it has now become part of a larger program, its parts may be executed on the next occasion when the program as a whole is executed. The execution of the ADD function is finished as soon as a single match and substitution occurs. If an ADD function cannot find a match, the search fails and the ADD has no side effect.

The ADD function is the basic tool for creating structures capable of more complex behavior. The ADD function operates by agglomerating free-floating material from the sea. In the initial generation of the process, this agglomerative process merely combines programs that were initially in the sea; however, in later generations, this process combines programs that are themselves the products of earlier combinations. This agglomerative process thereby creates, over time, large structures that contain a hierarchy of smaller structures. In nature, when the active receptor sites of two freely moving biological molecules match (in a complementary way), a chemical reaction takes place to which both entities contribute. The specific reaction generally would not have taken place had they not come together. Thus, there is some resemblance between the operation of the ADD function and the biochemical reactions that occur in nature. As will be seen, the physical growth by means of agglomeration yields a growth in functionality from generation to generation.

The number of ADD or SST functions in the program being executed generally increases as execution of ADDs incorporate programs into the program being executed. We call these new occurrences of the ADD or SST functions *acquired* functions.

The function LR ("Load REGISTER") works in conjunction with the global variables MATCHED-TREE and REGISTER. This function views the program (i.e., parse tree, LISP S-expression) in MATCHED-TREE as an unparenthesized sequence of symbols. For example, the program (i.e., parse tree)

```
(SAND (SOR (NOT D1) D0) D2)
```

is viewed as the sequence

```
{ SAND SOR NOT D1 D0 D2 }.
```

When each function has a specific number of arguments (as is the case here with the function set $\mathcal{F}$), an unparenthesized sequence of symbols is unambiguously equivalent to the original program (i.e., parse tree, LISP S-expression). This view is similar to that employed in the FORTH programming language, where functions are applied to a specified number of arguments, but where parentheses are not used or needed.

The function LR removes the first element (i.e., function or terminal) from MATCHED-TREE (viewed as an unparenthesized sequence of symbols) and moves it into the REGISTER. For example, if LR were to act once on the above program, the unparenthesized sequence associated with MATCHED-TREE would be reduced to

```
{ SOR NOT D1 D0 D2 }
```

and the REGISTER would contain SAND.

The function SR ("Store REGISTER") works in conjunction with the global variables REGISTER and OPEN-SPACE. The function SR takes the current contents of REGISTER (which is the name of a single function or terminal) and places it in OPEN-SPACE. REGISTER is not changed by the SR. Thus, an LR and an SR together can move a primitive element of a program from MATCHED-TREE to OPEN-SPACE.

The two-argument iterative operator DWT iteratively evaluates its first argument, «WORK», until its entire second argument, «TRAILING-TEMPLATE», matches the contents of MATCHED-TREE. The DWT operator is similar to the iterative looping operator found in many programming languages in that it performs certain work until its termination predicate is satisfied. The DWT operator is terminated by means of a biologically-motivated template-matching scheme.

If MATCHED-TREE contains a program because of the action of a previous SST ("Search Sea for Template"), a DWT ("Do «WORK» until «TRAILING-TEMPLATE» matches") that happens to contain both an LR ("Load REGISTER") and an SR ("Store REGISTER") in that order in its first argument subtree will iteratively load the REGISTER with the individual primitive elements of the program in MATCHED-TREE and successively store them into OPEN-SPACE one at a time. This iterative process will continue until the «TRAILING-TEMPLATE» of the DWT matches the current part of MATCHED-TREE.

If DWT fails to find a subtree that matches «TRAILING-TEMPLATE» while it is traversing MATCHED-TREE, the contents of OPEN-SPACE are discarded. If the tree placed into the OPEN-SPACE is not syntactically valid, the contents of OPEN-SPACE disintegrate, i.e., are discarded.

When the DWT operator terminates, the string of symbols in OPEN-SPACE is once more should be viewed as the equivalent parse tree.

Because of limited computer time, it is a practical necessity that deeply nested loops be avoided. Therefore, a DWT contained in the first argument of a DWT is not executed. If DWT fails to terminate for any reason within a reasonable number of steps or the symbols placed into OPEN-SPACE by the side effects of the «WORK» do not form a valid program, the contents of OPEN-SPACE are discarded for that DWT.

If there is an occurrence of a DWT, SST, or ADD function in the second argument subtree of a DWT, the inner function is never used to perform a search; the outer function merely uses it in trying to make a match.

7. Spontaneous Emergence

We start by creating a sea (population) of 12,500,000 computer programs consisting of independently created random compositions of the eight functions from the function set $\mathcal{F}$ and the three terminals from the terminal set $\mathcal{T}$ described above.

Figure 1 shows one of the programs that was actually generated by this random generative process of 12,500,000 programs. The program is shown as a rooted, point-labeled tree with ordered branches. In the tree representation of a program, the external points of the tree are associated with terminals (i.e., inputs to the program) or functions taking no arguments (i.e., LR or SR here) and the internal points are associated with functions taking one or more arguments. There are as many lines emanating downward from each internal point as arguments taken by the function. The particular program shown has 36 points (i.e., functions and terminals).

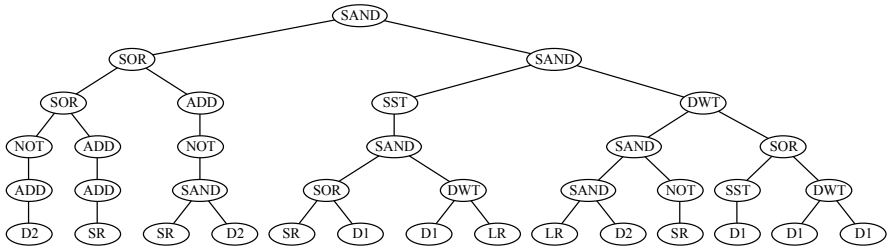

Figure 1 Spontaneously emergent 36-point program.

This 36-point program tree can be presented in "prefix" form as used in the Common LISP programming language:

```
(SAND (SOR (SOR (NOT (ADD D2)) (ADD (ADD (SR))))
           (ADD (NOT (SAND (SR) D2))))
      (SAND (SST (SAND (SOR (SR) D1) (DWT D1 (LR))))
            (DWT (SAND (SAND (LR) D2) (NOT (SR)))
                 (SOR (SST D1) (DWT D1 D1)))).
```

When this 36-point program is executed with one of the eight combinations of the three terminals D0, D1, and D2 as its inputs, it produces a certain Boolean value as its output. Each of the three logical functions (NOT, SOR, and SAND) return a specified Boolean output based on their one or two arguments; each of the five side-effecting functions (ADD, SST, LR, SR, and DWT) return T, return NIL, or act as the identity function. When this 36-point program is executed with all eight combinations of the three terminals D0, D1, and D2 as its inputs, it is discovered to perform the three-argument Boolean function shown in table 1. We say that this program performs three-argument Boolean rule 204 because 11001100_2 is 204_{10}.

Table 1 Truth table for the spontaneously emergent 36-point program.

	D2	D1	D0	Rule 204
0	NIL	NIL	NIL	NIL
1	NIL	NIL	T	NIL
2	NIL	T	NIL	T
3	NIL	T	T	T
4	T	NIL	NIL	NIL
5	T	NIL	T	NIL
6	T	T	NIL	T
7	T	T	T	T

As we will shortly see, this spontaneously emergent 36-point program is a self-replicator. We use the term self-replicator to mean that a program reproduces itself exactly, reproduces itself approximately, reproduces itself mutually as part of a group of two or more programs, or reproduces itself in any of the three forgoing ways with changes due to agglomeration of new material.

The italicized portion of this 36-point program contains an executable ADD function. This ADD function has an argument consisting of four points of which the functions NOT and SAND constitute its template. Each time this program is evaluated, the ADD function searches the sea for a program with the function NOT at its root and SAND as the point below the root in the leftmost branch. The search starts with the program itself. The only match found within this program is with the template of the argument of the very ADD function being executed. Since this kind of match is disallowed, the result is that no match is found within this program. The search therefore continues to the neighbor of this program in the sea. A match may occur between the template of this ADD (i.e., the NOT and SAND) and some neighboring program in the sea having a NOT at its root and a SAND immediately below its root (i.e., having a site consisting of NOT and SAND). If and when this happens, the ADD function in the first program will perform the side effect of substituting the entire second program into the first program. The five (of the 36) points of the first program consisting of the ADD and the four points below the ADD in the argument subtree of the ADD will be replaced by the entire second program.

Note that the SR function within the subtree argument to this ADD is not executable. Note the second of the two ADDs in the first underlined portion of the program is not executable because its template is too small. Note that the first of these two ADDs is executable, but its search will generally be fruitless since any potential matching program having an ADD at its root would have metamorphosed itself into something else.

The underlined and italicized portion of this program contains an SST subtree that searches the sea with a template of SAND and SOR. The SST function ("Search Sea for Template") searches the sea of programs for a program whose top part (i.e., site) matches the top part (i.e., template) of the argument subtree of the SST function. When an SST function finds a match, the global variable MATCHED-TREE is bound to the entire matching program. Note that the SR and DWT functions within the subtree argument to this SST are not executable.

The first argument to the executable two-argument DWT function on line 4 is in boldface, while the second argument is underlined in boldface. The first argument to this DWT executes both the LR ("Load REGISTER") and the SR ("Store REGISTER") functions, thereby causing the copying, one piece at a time, of the contents of MATCHED-TREE to the REGISTER. The six-item second argument to this DWT is the «TRAILING-TEMPLATE». Since this DWT function appears at the far right of the program, a match between the trailing template and the contents of MATCHED-TREE occurs precisely on completion of the copying of the entire contents of MATCHED-TREE.

This 36-point program is not the only self-replicator that we found. In fact, we found 604 self-replicators in this random search of 12,500,000 programs, so that the probability of finding a self-replicator from a blind random search of programs composed of terminals from the terminal set $\mathcal{T}$ and functions from the function set $\mathcal{F}$ is about one in 20,695.

The discovery of this 36-point program and the 603 other self-replicators in the 12,500,000 programs verifies that spontaneous emergence of self-reproducing computer programs is possible.

If we had unlimited computer resources, we could proceed using all 12,500,000 programs, however, proceeding with the 12,500,000 programs would be prohibitive with the available

computer resources. Moreover, proceeding with the 604 self-replicators would also be prohibitive since they contain many excessively time-consuming and memory-consuming features. Many of the randomly created self-replicators contain multiple occurrences of the ADD function that cause the permanent incorporation of various matching programs into the program being executed, thereby increasing the size of the program at an exponential rate. Moreover, in practice, this growth in program size coincides with multiple occurrences of the DWT function. The combined effect of multiple occurrences of both the ADD and DWT functions in the initial generation is that the available computer memory is almost immediately exhausted. Therefore, we decided to screen the 604 self-replicators so that only the less prolific individuals would be included in our sea of programs. If we had unlimited computer resources, we would not need to do this screening.

As initial screening criteria, we require that each program be capable of asexual reproduction, have exactly one executable ADD function (i.e., one with a non-trivial template) and that the executable ADD function appears before (i.e., to the left of) exactly one executable DWT function. It was necessary to require that the one executable occurrence of the ADD function appear to the left of the one executable occurrence of the DWT function so that the ADD function would always be executed prior to the DWT function. For purposes of this screening, a function is considered executable provided it is not in the argument subtree of an SST function or an ADD function. The reason is that, if there were only one DWT function in a program and if it were executed before execution of the ADD function, the emission produced by the DWT would necessarily always occur before the substitution caused by the ADD. Consequently, the substitution would never be permanently incorporated into a program for the next generation, and it would be impossible to evolve an improvement requiring more than one generation to evolve. Even with this screening of the initial population, the exponential growth of the programs caused by the ADD function and the large number of emissions produced by the DWT functions kept the entire process on the edge of collapse. After applying these initial screening criteria, there are only eight self-replicators satisfying these initial screening criteria, so that the probability of finding a screened self-replicator with the function set $\mathcal{F}$ is about one in 1,562,500.

The following five additional criteria make a self-replicator even more suitable for our purposes here. First, there is exactly one executable SST function. Second, the template of the one executable ADD and SST function must not refer to a program that would never be present in a population of self-replicators. Third, the ADD, SST, and DWT functions must appear in that order when the LISP S-expression is evaluated in the usual way. Fourth, the executable ADD and SST functions must not be part of the first argument to the DWT function. Fifth, the first argument to the DWT function must have one executable LR and one executable SR function appearing in that order.

If we impose these additional five screening criteria, then there are only two fully screened self-replicators in 12,500,000, so that the probability of finding a fully screened self-replicator is about one in 6,250,000. The 36-point self-replicator shown above is one of these two.

As was seen above, finding 500 fully screened self-replicators by means of blind random search would require processing about 3,125,000,000 individuals (i.e., 500 times 6,250,000). Moreover, since numerous separate runs will be required in this chapter, many different initial random populations of 500 fully screened self-replicators will be necessary. Even if a single pool of, say, 3,000 replicators were used (with a different random drawing of 500 individuals being made for each particular run), 18,750,000,000 individuals would still have to be processed to yield the desired pool of 3,000. Both of these approaches are totally prohibitive.

The spontaneous emergence of 604 self-replicators, the 8 screened self-replicators, and 2 fully screened self-replicators out of 12,500,000 randomly generated individuals already established that the spontaneous emergence of self-reproducing computer programs is possible. Finding additional self-replicators is merely a matter of computer time. Since the goal now is to perform experiments involving the emergence of evolutionary self-improving behavior, we decided to synthesize a pool of 3,000 self-replicators modeled after the fully screened self-replicators that were actually discovered. In synthesizing the pool of 3,000, we made certain that it contained a wide variety of different sizes and shapes representative of the sizes and shapes of the self-replicators that were actually discovered. No duplicate checking was done.

8. Evolution of Self-Improving Behavior

Actual biological entities interact with their environment. Each program in the sea is an entity which must grapple with its environment. The determination of whether an individual in the population has learned a task is made by evaluating how well it performs the task for all possible situations that it may encounter in the environment.

The domain of Boolean functions provides a relatively simple domain in which learning can take place. We represent these situations by means of combinations of the environmental sensors (i.e., inputs to the program). Specifically, the environment consists of all $2^3 = 8$ possible combinations of the three Boolean variables `D2`, `D1`, and `D0`. Each program in the population is evaluated once for each of the eight possible combinations of the environmental sensors `D2`, `D1`, and `D0`. If the program performs correctly (i.e., emulates the behavior of the target Boolean function to be learned) for a particular combination of environmental sensors, its emission, if any, will be admitted into the `OPEN-SPACE`. If the emission then survives random culling (i.e., decimation unrelated to its performance), the emitted entity will become part of the sea for the next generation. For example, if an entity emulates a particular target Boolean function for six of the eight possible combinations of the environmental sensors, it has six opportunities to produce emissions. In contrast, if an entity emulates the target Boolean function for three of the eight possible combinations, it will have only half as many opportunities to emit items. If the individual entity happens to be a self-replicator, it will emit a copy of itself into the `OPEN-SPACE` if it successfully performs the unknown task for a particular combination of environmental conditions. A self-replicator that performs the unknown task for six combinations of environmental conditions will have twice the chance of becoming part of the sea for the next generation than a self-replicator that correctly performs the task for only three combinations.

The programs that are in the sea in one generation of the population do not automatically get into the next generation. They have a limited life (i.e., one generational time-step). The only way to get into the new generation is by means of a `SR` function that places symbols of a program into the `OPEN-SPACE`. In a self-replicator, there is at least one `DWT` function with appropriate invocations of `LR` and `SR` that together copy the program into the `OPEN-SPACE`. In addition, getting into the next generation requires correctly emulating the performance of a target three-argument Boolean function for one or more combinations of the environmental sensors. In other words, an entity survives only if it correctly grapples with the problems posed by its environment. Fitter programs successfully emulate the target function for more combinations of the environmental sensors and therefore are more likely to appear in the next generation.

Note that there is no explicit numerical calculation of fitness here. Instead, the entities in the sea are independently acting and self-replicating entities. Each entity is presented with the entire spectrum of possible combinations of environmental sensors. If an entity successfully performs

the task for a combination of environmental conditions and the entity is also capable of self-reproduction, it has a chance of being admitted into the next generation. Effectiveness in grappling with the unknown environment (i.e., correct performance of the unknown task) is required for an emission to have a chance to be admitted to the next generation. Since the entities have a finite lifetime (one generational time step), the failure of an entity to respond effectively to a combination of environmental conditions means that its emission does not become a part of the population for the next generation. Thus, the requirements for continued viability in the population are both the ability to self-replicate (exactly, approximately, mutually, or with changes due to agglomeration) and the ability to effectively grapple with the environment. In the context of these self-replicating entities in this sea, natural selection operates on individuals in the population that are more or less successful in grappling with their environment.

Because the sea is constantly changing, the neighbor of a given program changes as each combination of the environmental sensors is presented to each program for testing.

If we had unlimited computer resources, we would allow 100% of the emissions to move from the OPEN-SPACE into the sea for any combination of environmental sensors for which the program emulates the target function. However, since our computer resources are limited and we intend to present each computer program in the sea with all combinations of its environmental sensors, we need to randomly cull these emissions. In particular, only 30% of the emissions coming from programs that correctly emulate the behavior of the target function to be learned for a particular combination of environmental sensors are actually admitted to the sea from. This culling is conducted using a uniform random probability distribution and is not related to performance, so an entity that performs better will have a greater chance of having its emissions end up in the sea of the next generation.

Because the template size is 2, and because the function set has eight functions, and because the terminal set has three terminals, there are 40 (i.e., $5 \infty 8$) possible templates. Since the neighbor of a given program changes as each different combination of environmental sensors is presented to a program, it is desirable that the sea contain at least as many possible matches as there are combinations of its environmental sensors (i.e., eight) for the typical template, and preferably more. This suggests a population somewhat larger than 320. In fact, we settled on a population size of 500.

If the population is to consist of about 500 individuals capable of emission and if a randomly created computer program emulates the performance of a target three-argument Boolean function for about half of the eight possible combinations of the three environmental sensors, then there will be about 2,000 emissions as a result of testing 500 individuals as to their ability to grapple with their environment. If the programs in the sea improve so that after a few generations the typical program emulates the performance of a target three-argument Boolean function for about six of the eight possible combinations of environmental sensors, then, the number of emissions rises to about 3,000. When these emissions are randomly culled so as to leave only 30% of their original number on each generation, about 900 remain.

In addition, programs generally acquire additional occurrences of the DWT function when an ADD function makes a substitution, thereby causing the number of occurrences of DWT functions to increase rapidly. These occurrences of acquired functions are identified with the suffix -ACQ herein. The templates of these acquired DWT functions are rarely helpful or relevant to self-replication. If we had unlimited computer resources, we could retain all of these acquired DWT functions. In practice, we heavily cull the emissions produced by these acquired DWT functions so that only 0.01% of the emissions from programs that correctly emulate the behavior of the target function are admitted to the sea.

The population is monitored with the goal of maintaining it at 500. The culling percentages selected for this problem are intended to initially yield somewhat more than 500 emissions. If the number of emissions after this culling is still above 500, the remaining emissions are randomly culled again so that exactly 500 emissions are placed into the sea. Occasionally, the sea contains fewer than 500 programs. If we had unlimited computer resources, there would be no need to carefully control the size of the population in this way.

The ADD function bears some resemblance to the crossover (recombination) operation of genetic programming. The two operations are similar in that they permanently incorporate a program tree from one program into another program. They differ in four ways. First, the ADD function here produces only one offspring. Second, the sea changes as each new combination of environmental sensors is considered, so the matching program is generally not constant between such combinations. Third, there is no random selection of a crossover point within the program; the entire matching program is always inserted into the program being executed. Fourth, the incorporation of genetic material into the program is initiated from inside the program itself. The program being executed is active, and its code causes the incorporation of genetic material into itself and causes the reproduction of the program into the next generation.

Computer memory as well as computer time was limited. The initial sea containing the desired 500 fully screened self-replicators would have come from a run containing about 3,124,999,500 programs that were not fully screened self-replicators. Note that programs that are not even self-replicators may nevertheless contain numerous occurrences of the DWT function and may produce numerous emissions. Since we did not have sufficient memory to handle these 3,124,999,500 other programs, much less their emissions, and we did not have sufficient computer time to execute 6,249,999 useless programs for each potentially interesting program, we started each run with 500 fully screened individuals.

In implementing the ADD function, the COPY-TREE function is used to make a copy of both the matching program and the program being executed. Only a copy of the matching program is permanently incorporated into a copy of the program being executed, thus permitting the unchanged original program to be exposed to all possible combinations of its environmental sensors within the current generation and permitting the matching program to interact more than once with the program being executed on the current generation. If we had unlimited computer resources, the sea could be sufficiently large and rich to permit a matching program to be devoured by the program into which it is incorporated. In that event, essentially the same process would occur progressively over many generations, thereby producing some extraordinarily large programs.

Even with both types of culling mentioned above and the full screening of generation 0, our experience is that the growth in the physical size of the programs and the growth in emissions is such that the process can be run for only about six generations. Therefore, it is necessary to find a task that can be learned in fewer than about six generations with an initial population of size 500, but which is not so simple that it is likely to be produced at random.

Among three-argument Boolean functions generated at random using the function set {AND, OR, NAND, NOR}, Boolean rule 235 lies roughly in the middle of all three-argument Boolean functions in difficulty of discovery by genetic programming (Koza 1992). Rule 235 is equivalent to

```
(OR (EQV D2 D1) D0),
```

and is a moderately difficult rule to learn because it contains an even-2-parity (i.e., not-exclusive-or) function.

9. Results of One Run

We proceed using Boolean function 235 as the environment for the programs in the sea and a pool of 3,000 synthesized programs patterned after the actually-discovered fully-screened self-replicators.

In one run, one of the individuals in generation 0 of our sea of programs was the following 28-point program, which performs the three-argument Boolean function 250:

```
(SOR (NOT (SAND (NOT D0) (ADD (NOT (SOR D0 D2)))))
     (SAND (SST (SOR (NOT D2) (ADD D2)))
           (DWT (SOR (LR) (SR))
                (SOR (SAND D0 D2) (SAND D0 D0)))).
```

The ADD function and its entire argument subtree are underlined in the program above; the ADD has a template of NOT and SOR. The SST function has a template of SOR and NOT. The SST subtree is shown in italics. The DWT function, its first argument (SOR (LR) (SR)), and its second «TRAILING-TEMPLATE» argument consisting of (SOR (SAND D0 D2) (SAND D0 D0)) are shown in boldface.

This 28-point program is a self-replicator, because its SST function has a template consisting of SOR and NOT, thereby matching its own root and causing MATCHED-TREE to be bound to itself. The «TRAILING-TEMPLATE» of the DWT function matches itself, thereby causing the copying of the entire program into the sea (subject to culling) for the next generation for those combinations of environmental sensors that emulate rule 235.

Figure 2 graphically depicts this 28-point program.

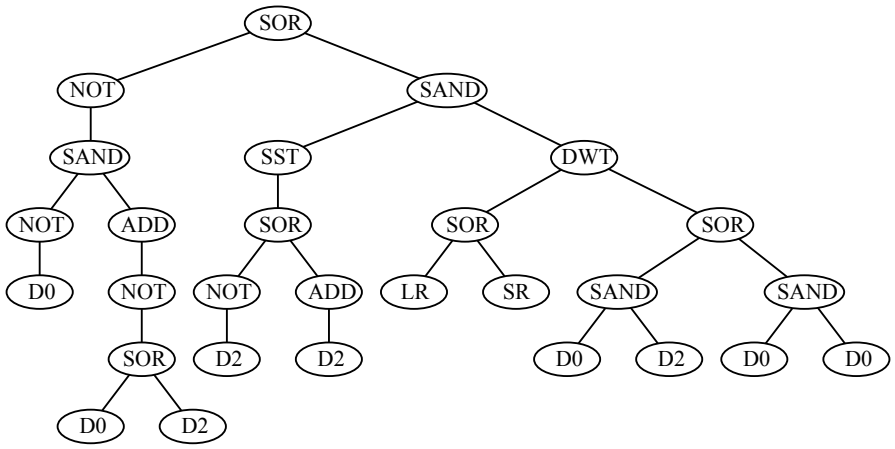

Figure 2 28-point self-replicator performing rule 250 from generation 0.

Note that a human programmer could write a much smaller and simpler self-replicator composed of the functions from the function set $\mathcal{F}$ and terminals from the terminal set $\mathcal{T}$. However, the random compositions that spontaneously emerge in a blind random search of this space of compositions are unlikely to be minimal in size.

The first neighbor in the sea in generation 0 that matched the template of NOT and SOR of the ADD function was the following 31-point program, which performs the three-argument Boolean rule 004:

```
(NOT (SOR (SAND (SOR D0 D2) (SOR (ADD (SOR (SAND D0 D1) D2)) D0)) (SOR (SST
(NOT (SOR D2 D1))) (DWT (SAND (NOT (LR)) (SR)) (SAND (SAND D2 D1) (SST
D0)))))).
```

This 31-point program is also a self-replicator whose root is NOT and whose point immediately below and to the left of the root is SOR, as shown in boldface above.

Figure 3 graphically depicts this 31-point program.

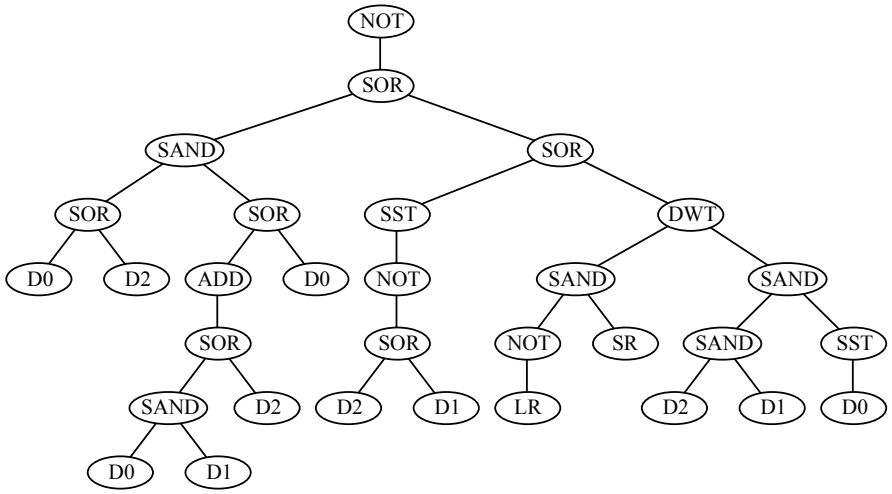

Figure 3 31-point self-replicator performing rule 004 from generation 0.

The side effect of the execution of the ADD function in the 28-point individual is that the following 54-point individual, which performs the three-argument rule 251, is created:

(SOR (NOT (SAND (NOT D0) **(NOT (SOR (SAND (SOR D0 D2) (SOR (ADD (SOR (SAND D0 D1) D2)) D0)) (SOR (SST (NOT (SOR D2 D1))) (DWT-ACQ (SAND (NOT (LR)) (SR)) (SAND (SAND D2 D1) (SST D0))))))**)) (SAND (SST (SOR (NOT D2) (ADD D2))) (DWT (SOR (LR) (SR)) (SOR (SAND D0 D2) (SAND D0 D0))))).

The 31-point subtree in boldface (with a NOT at its root and with SOR immediately below and to the left of the root) was inserted by the ADD function from the 28-point individual. The 28-point individual lost 5 points and gained 31 points, thus producing the 54-point individual. Note that this new 54-point individual has an acquired DWT-ACQ function that came from the 31-point individual.

In this run, this 54-point individual eventually enters the sea and becomes part of generation 1.

Figure 4 graphically depicts this 54-point program.

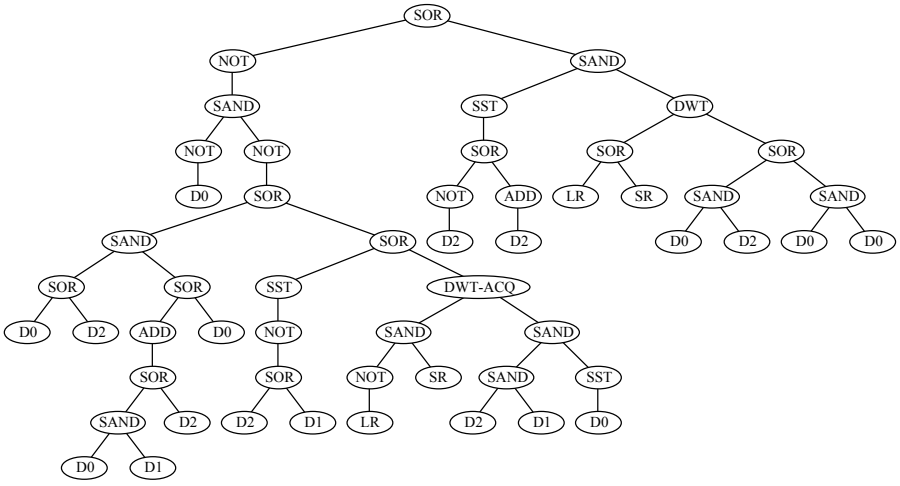

Figure 4 54-point self-replicator performing rule 251 from generation 1 with SOR at its root and with NOT immediately below and to the left of the root.

Another of the individuals in generation 0 of our sea of programs was the following 41-point program, which performs the three-argument Boolean function 238:

```
(SOR (SAND (NOT (ADD (SOR D0 (SAND D2 D0)))) (SAND (NOT (SOR (SAND D0 D1) D2))
(SST (SOR (SAND D1 D2) D0)))) (SOR D1 (DWT (SOR (SAND (NOT D0) (NOT (LR)))
(SAND D0 (SR))) (SOR (SOR D0 D1) (SAND D2 D0))))).
```

The ADD function and its argument subtree in this program is shown in boldface. Its template consists of SOR and D0. As it happens, there is no program in the sea that matches this template. However, because this individual is a self-replicator that emulates rule 235 for some combinations of the environmental sensors, this 41-point individual is copied (subject to culling) into the sea of generation 1.

Figure 5 graphically depicts this 41-point program.

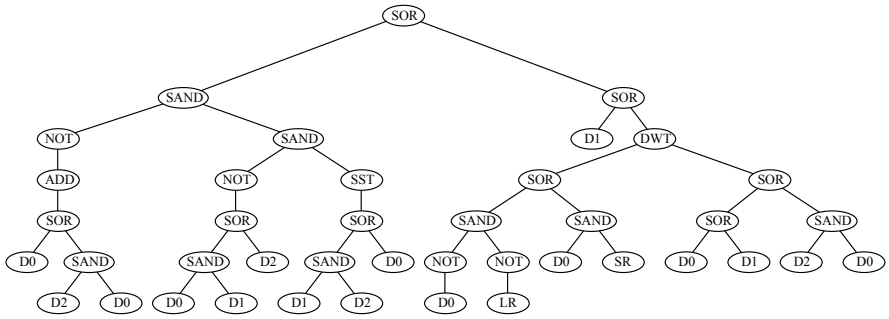

Figure 5 41-point self-replicator performing rule 238 in generation 0.

The above 54-point individual contains an ADD function with a template of SOR and SAND. When it was evaluated during generation 1, it encountered the 41-point individual (which has SOR at its root and SAND immediately below and to the left of the root) as a neighbor. Therefore, the ADD template of the above 54-point program matched the 41-point neighbor. The side effect of the match is the creation of an 89-point individual (i.e., 54 points + 41 points – 6 points for the subtree argument of the ADD function being executed). This 89-point program performs rule 235:

```
(SOR (NOT (SAND (NOT D0) (NOT (SOR (SAND (SOR D0 D2) (SOR (SOR (SAND (NOT (ADD
(SOR D0 (SAND D2 D0)))) (SAND (NOT (SOR (SAND D0 D1) D2)) (SST (SOR (SAND D1
D2) D0)))) (SOR D1 (DWT (SOR (SAND (NOT D0) (NOT (LR))) (SAND D0 (SR))) (SOR
(SOR D0 D1) (SAND D2 D0))))) D0)) (SOR (SST (NOT (SOR D2 D1))) (DWT (SAND (NOT
(LR)) (SR)) (SAND (SAND D2 D1) (SST D0)))))))) (SAND (SST (SOR (NOT D2) (ADD
D2))) (DWT (SOR (LR) (SR)) (SOR (SAND D0 D2) (SAND D0 D0))))).
```

The 41-point subtree that was inserted by the ADD function is shown in boldface in the above program.

This 89-point individual performing three-argument Boolean rule 235 eventually enters the sea for generation 2.

Figure 6 graphically depicts this 89-point program.

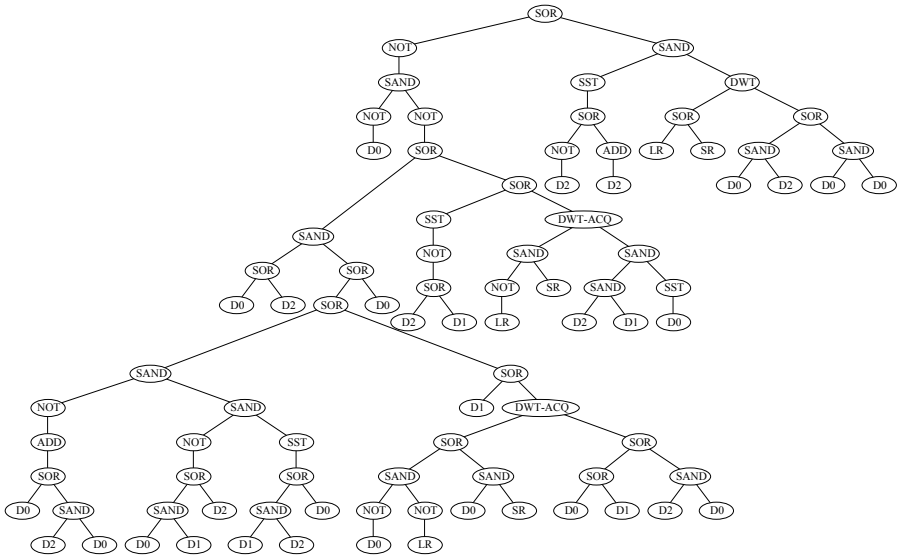

Figure 6 89-point self-replicator performing rule 235 in generation 2 .

Figure 7 is a genealogical tree showing the three grandparents from generation 0, the two parents from generation 1, and the final 89-point individual performing rule 235 in generation 2 from figure 29. The 28-point grandparent performing rule 250 comes from figure 2. The 31-point grandparent performing rule 004 comes from figure 3. The 54-point parent performing rule 251 comes from figure 4.

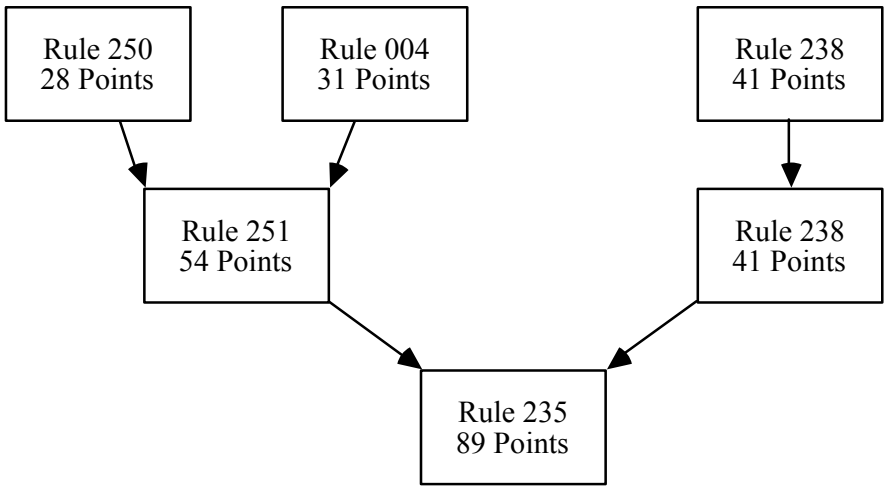

Figure 7 Genealogical tree for the 89-point self-replicator performing rule 235 in generation 2 .

Figure 8 shows, by generation, the progressive improvement for the run in which the 89-point self-replicator performing rule 235 was found in generation 2. In this figure, standardized fitness is the zero-based (i.e., zero is best) measure of fitness based on the number of errors (out of eight) made by a particular program. In particular, it shows the average standardized fitness of the population as a whole and the value of the standardized fitness for the best-of-generation individual and the worst-of-generation individual.

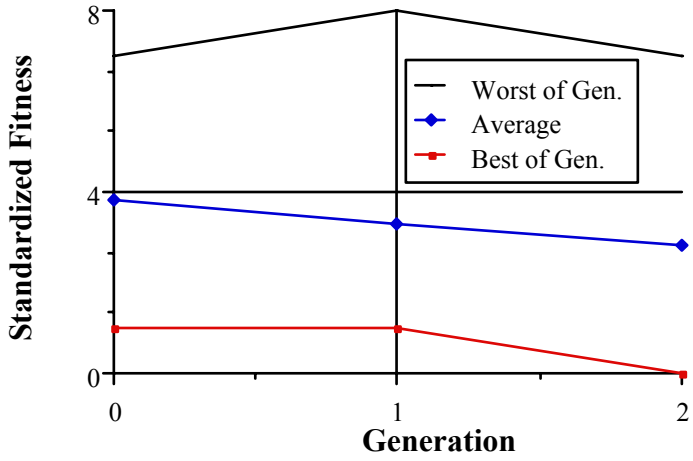

Figure 8 Fitness curves.

Figure 9 shows, by generation, the average structural complexity of the population as a whole and the values of structural complexity for the best-of-generation individual. Because of the way that the ADD function incorporates entire individuals in the population within existing individuals, structural complexity rises very rapidly.

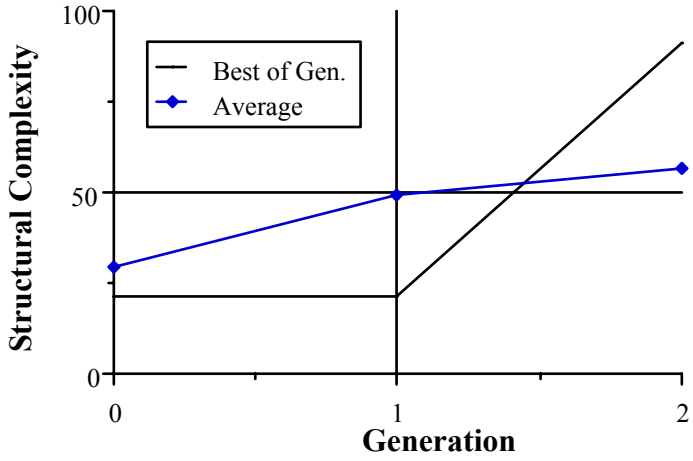

Figure 9 Structural complexity curves.

Figure 10 shows the hits histograms for generations 0, 1, and 2 of this run. One can see that the population as a whole has improved over even the small number of generations shown. The arrow points to the two individuals scoring 8 hits on generation 2. In this figure, "hits" is the fitness measure based on the number of correct responses made by a program to the combinations of environmental inputs it experiences.

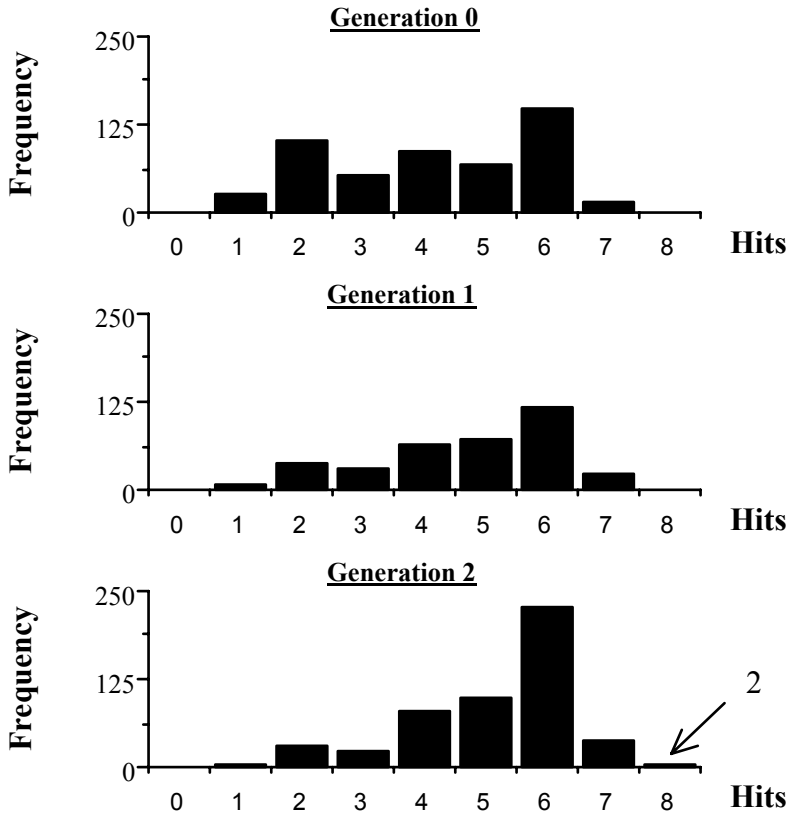

Figure 10 Hits histograms for generations 0, 1, and 2 .

10. Results of 242 Runs

Figure 11 presents the performance curves showing, by generation, the cumulative probability of success $P(M,i)$ and the number of individuals that must be processed $I(M,i,z)$ to yield, with 99% probability, at least one program producing the correct value of rule 235 for all eight combinations of environmental sensors. The graph is based on 242 runs. The population was 500 and the maximum number of generations to be run was 2.

The rising curve in figure 11 shows, by generation, the experimentally observed cumulative probability of success, $P(M,i)$, of solving the problem by generation i (i.e., finding at least one program in the population which produces the correct value for all combinations of environmental sensors). The falling curve shows, by generation, the number of individuals that must be processed, $I(M,i,z)$, to yield, with $z = 99\%$ probability, a solution to the problem by generation i. $I(M,i,z)$ is derived from the experimentally observed values of $P(M,i)$ and is the product of the population size M, the generation number i, and the number of independent runs $R(z)$ necessary to yield a solution to the problem with probability z by generation i. The number of runs $R(z)$ is, in turn, given by

$$R(z) = \left\lceil \frac{\log(1-z)}{\log(1-P(M,i))} \right\rceil,$$

where the brackets indicate the ceiling function for rounding up to the next highest integer.

The cumulative probability of success, $P(M,i)$, is 2.9% by generation 1 and 4.5% by generation 2. The numbers in the oval indicate that if this problem is run through to generation 2,

processing a total of 148,500 (i.e., 500 ∞ 3 generations ∞ 99 runs) individuals is sufficient to yield a solution to this problem with 99% probability.

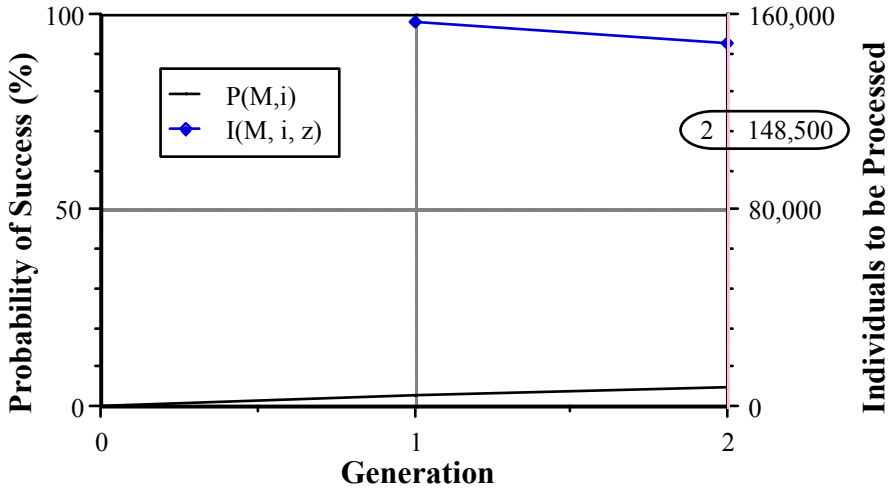

Figure 11 Performance curves show that 148,500 individuals must be processed to yield a solution to the problem of learning rule 235 .

When we tested rule 235 with the function set $\mathcal{F}$, we found that it has a probability of about one in 33,177 of being generated at random. This means that the processing of 158,251 individuals is sufficient to yield a random program realizing rule 235 with 99% probability. Thus, for a population size of 500, we would be very unlikely to encounter a solution to rule 235 merely as a consequence of blind random search on any one run. Thus, the selection of rule 235 as the "environment" was a reasonable one.

The 148,500 individuals required to be processed here is less than the 158,251 individuals that must be processed to find an program performing rule 23 at random.

It is interesting that rule 235 can be learned with this agglomerative ADD function.

11. Summary

In summary, in spite of a number of simplifications and compromises necessitated by the limited available computer resources, we have demonstrated the spontaneous emergence of self-replicating computer programs, sexual reproduction, and evolutionary self-improving behavior among randomly created computer programs using a computationally complete set of logical functions.

Table 2 summarizes the key features of this experiment.

Table 2 Summary.

Probability of a program performing rule 235	1:33,177
Number of individuals that must be processed to find a program performing rule 235 with 99% probability	158,251
Probability of a self-replicator	1:20,695
Probability of a screened self-replicator	1:1,562,500
Probability of a fully screened self-replicator	1:6,250,000
500 times probability of a fully screened self-replicator	3.125×10^9

Number of self-replicators that must be processed to evolve a self-replicator performing rule 235 with 99% probability	148,500

The above experiment is well within the range of currently available computational resources. The fact that spontaneous emergence can occur with a probability of the order of 10^{-6} to 10^{-9} with a function set such as $\mathcal{F}$ suggests that many fruitful experiments on spontaneous emergence and evolutionary self-improvement can be conducted at this time.

12. Acknowledgments

James P. Rice of the Knowledge Systems Laboratory at Stanford University made numerous contributions in connection with the computer programming of the above.

Simon Handley made helpful comments on the above.

13. References

Arbib, Michael A. Simple self-reproducing universal automata. *Information and Control*. 9: 177–189. 1966.

Bagley, Richard J. and Farmer, J. Doyne. Spontaneous emergence of a metabolism. In 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 1991. Pages 93-140.

Bratley, Paul, and Millo, Jean. Computer recreations: Self-reproducing programs. *Software Practice and Experience*. 2: 397–400. 1972.

Burger, John, Brill, David, and Machi, Filip. Self-reproducing programs. *Byte*. 5(August) 72-–74. 1980.

Burks, Arthur W. Theory of Self Reproducing Automata. Urbana, IL: University of Illinois Press, 1966.

Burks, Arthur W. *Essays on Cellular Automata*. Urbana, IL: University of Illinois Press, 1970.

Burks, Arthur W. Von Neumann's self-reproducing automata. In Aspray, William and Burks, Arthur (editors). *Papers of John von Neumann on Computing and Computer Theory*. Cambridge, MA: The MIT Press 1987. Pages 491-552.

Buss, Leo W. *The Evolution of Individuality*. Princeton, NJ: Princeton University Press 1987.

Codd, Edgar. F. *Cellular Automata*. New York: Academic Press 1968.

Codd, Edgar F. 1992. Private communication June 12, 1992.

Dawkins, Richard. *The Blind Watchmaker*. New York: W. W. Norton 1987.

Devore, John and Hightower, Ronald. The Devore variation of the Codd self-replicating computer. Draft November 30, 1992. Presentation at Third Workshop on Artificial Life, Santa Fe, New Mexico. **NOTE TO THE EDITOR: IS THIS IN THE SAME VOLUME NOW???**

Dewdney, A. K. In a game called core war hostile programs engage in a battle of bits. *Scientific American*, 250 (May) 14–22. 1984.

Dewdney, A. K. A core war bestiary of viruses, worms and other threats to computer memories. *Scientific American*, 252 (March) 14-23. 1985.

Dewdney, A. K. A program called MICE nibbles its way to victory at the first core war tournament. *Scientific American*, 256(January) 14–20. 1987.

Dewdney, A. K. Of worms, viruses, and Core War. *Scientific American*, 260 (March) 110–113. 1989.

Dyson, Freeman. *Origins of Life*. Cambridge: Cambridge University Press. 1985.

Farmer, J. Doyne and Belin, Alletta d'A. Artificial life: The coming evolution. In 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 1991. Pages 815-838.

Fontana, Walter. Functional self-organization in complex systems. In Nadel, Lynn and Stein, Daniel L. (editors). *1990 Lectures in Complex Systems*. Redwood City, CA: Addison-Wesley 1991. Pages 407-426. 1991a.

Fontana, Walter. Algorithmic Chemistry. In 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 1991. Pages 159-209. 1991b.

Holland, John H. *Adaptation in Natural and Artificial Systems.* Ann Arbor, MI: University of Michigan Press 1975. See also Holland 1992.

Holland, John H. Studies of the spontaneous emergence of self-replicating systems using cellular automata and formal grammars. In Lindenmayer, Aristid, and Rozenberg, G. (editors). *Automata, Languages, Development*. Amsterdam: North-Holland 1976. Pages 384-404.

Holland, John H. ECHO: Explorations of evolution in a miniature world. Paper presented at Second Workshop on Artificial Life in Santa Fe, New Mexico, February 1990.

Holland, John H. Second edition of *Adaptation in Natural and Artificial Systems.* Cambridge, MA: The MIT Press 1992.

Kampis, George. *Self-Modifying Systems in Biology and Cognitive Science*. Oxford: Pergamon Press 1991.

Kemeny, John G. Man viewed as a machine, *Scientific American*, 192(4), Pages 58–67, April 1955.

Koza, John R. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann 1989. Volume I. Pages 768-774.

Koza, John R. Genetic evolution and co-evolution of computer programs. In 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. Pages 603–629. 1991.

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

Koza, John R., and Rice, James P. A genetic approach to artificial intelligence. In Langton, C. G. (editor).*Artificial Life II Video Proceedings*. Redwood City, CA: Addison-Wesley 1991a.

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

Laing, Richard. Automaton Introspection. *Journal of Computer and System Sciences*. 13:172-183. 1976.

Laing, Richard. Automaton Models of Reproduction by Self-Inspection. *Journal of Theoretical Biology*. 66: 437-456. 1977.

Langton, Christopher G. Self-reproduction in cellular automata. In Farmer, Doyne, Toffoli, Tommaso, and Wolfram, Stephen (editors). *Cellular Automata: Proceeding of an*

Interdisciplinary Workshop, Los Alamos, New Mexico, March 7-11, 1983. Amsterdam: North-Holland Physics Publishing, 1983. Also in *Physica D*, volume 10, pages 135-144, 1984.

Langton, Christopher G. Studying artificial life with cellular automata. In Farmer, Doyne, Lapedes, Alan, Packard, Norman, and Wendroff, Burton (editors). *Evolution, Games, and Learning*. Amsterdam: North-Holland 1986. Pages 120-149. Also in *Physica D* 22D (1986), pages 120-149.

Langton, Christopher G. (editor). *Artificial Life, Santa Fe Institute Studies in the Sciences of Complexity*. Volume VI. Redwood City, CA: Addison-Wesley. 1989.

Langton, Christopher G., 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 1991.

Langton, Christopher G. (editor). *Artificial Life II Video Proceedings*. Addison-Wesley 1991a.

Langton, Christopher G. Self-reproducing loops and virtual ants. In Langton, Christopher G. (editor). *Artificial Life II Video Proceedings*. Addison-Wesley 1991a. 1991b.

Myhill, John. The abstract theory of self-reproduction. In Burks, Arthur W. *Essays on Cellular Automata*. Urbana, IL: University of Illinois Press, 1970. Pages 206-218.

Newman, R. C. Self-reproducing automata and the origin of life. *Perspectives on Science and Christian Faith*. 40(1) 24-31. 1988.

Rasmussen, Steen, Knudsen, Carsten, Feldberg, Rasmus, and Hindsholm, Morten. The coreworld: Emergence and evolution of cooperative structures in a computational chemistry. In Forrest, Stephanie (editor). *Emergent Computation: Self-Organizing, Collective, and Cooperative Computing Networks*. Cambridge, MA: The MIT Press 1990. Pages 111-134. Also in *Physica D* , Volume 42D, 1990.

Rasmussen, Steen, Knudsen, Carsten, and Feldberg, Rasmus. Dynamics of programmable matter. In 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 1991. Pages 211-254.

Ray, Thomas S. Evolution and optimization of digital organisms. In Brown, H. (editor). *Proceedings of the 1990 IBM Supercomputing Competition: Large Scale Computing Analysis and Modeling Conference* . Cambridge, MA: The MIT Press 1991a.

Ray, Thomas S. An approach to the synthesis of life. In 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 1991. Pages 371-408. 1991b.

Ray, Thomas S. Is it alive or is it GA? In Belew, Rik and Booker, Lashon (editors). *Proceedings of the Fourth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann. Pages 527-534. 1991c.

Ray, Thomas S. Population dynamics of digital organisms. In C. G. Langton (editor). *Artificial Life II Video Proceedings*. Redwood City, CA: Addison-Wesley 1991a. 1991d.

Schuster, P. Evolution of self-replicating molecules – a comparison of various models for selection. In Demongeot, J., Goles, E., Tchuente, M. (editors) *Dynamical Systems and Cellular Automata*. London: Academic Press, 1985.

Skipper, Jakob. The complete zoo evolution in a box. In Varela, Francisco J., and Bourgine, Paul (editors). *Toward a Practice of Autonomous Systems: Proceedings of the first European Conference on Artificial Life*. Cambridge, MA: The MIT Press 1992. Pages 355-364

Smith, Alvy Ray. Simple non-trivial self-reproducing machines. In Langton, C. G. (editor).*Artificial Life II*. Redwood City, CA: Addison-Wesley 1991a. Pages 709-725. 1991.

Smith, John Maynard. *Evolutionary Genetics*. Oxford: Oxford University Press. 1989.

Spafford, Eugene H. Computer viruses - a form of artificial life? In 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 1991. Pages 727-745.

Thatcher, J. W. Self-describing Turing machines and self-reproducing cellular automata. In Burks, Arthur W. (editor) *Essays on Cellular Automata*. Urbana, IL: University of Illinois Press, 1970.