Life and Evolution in Computers

Melanie Mitchell[*]

Biophysics Group

Los Alamos National Laboratory

1 Computers and Life

Can we build computers that are intelligent and alive? This question has been on the minds of computer scientists since the dawn of the computer age and remains a most compelling line of inquiry. Some would argue that the question makes sense only if we put scare quotes around "intelligent" and "alive," since we're talking about computers, after all, not biological organisms. My own view is that the answer is unequivocally yes, no scare quotes or other punctuation needed, but that to get there our notions of life, intelligence, and computation will have to be deepened considerably.

You can ask ten biologists what are the ten (or 20 or 100) key requisites for life and you'll get a different list each time. But most are likely to include autonomy, metabolism, self-reproduction, survival instinct, and evolution and adaptation. As a start, can we understand these processes mechanistically and capture them in computers?

Many people have argued a vehement "no" for the following reasons:

Autonomy: "A computer can't do anything on its own; it can do only what humans program it to do."

Metabolism: "Computers can't create or gather their own energy from their environment like living organisms do; they have to be fed energy (e.g., electricity) by humans."

Self-reproduction: "A computer can't reproduce itself; to do so it would have to contain a description of itself, and that description would have to contain a description of itself, and so on *ad infinitum.*"

Survival instinct: "Computers don't care whether they survive or not." (For example, from an editorial in the *Boston Globe*: "Deep Blue may have beat Kasparov, but it didn't get any joy out of it.")

---

[*]Address: P-21, MS D454, LANL, Los Alamos, NM 87545. Email: mmitchell@lanl.gov

Evolution and adaptation: "A computer can't evolve or adapt on its own; it is restricted to change only in ways specified ahead of time by its programmer."

Although these arguments are still believed by a great many people, all of them have been disproven in one way or another in the field of Artificial Life (Langton 1989; Langton et al., 1992; Langton 1993; Langton, 1995; Langton & Shimohara, 1997; Brooks & Maes, 1994). In this paper I will focus on those issues most closely related to Darwinism—self-reproduction and evolution.

2   Self-Reproduction in Computers

The "self-reproduction" argument above is the most mathematical one: it states that reproduction in a computer would lead to an infinite regress. Consider, for example, the simplest version of the computer self-reproduction problem: write a computer program to print out an exact copy of itself and nothing else. Consider the following attempt. I start out with the name of the program:

```
program copy
```

Now I need to add an instruction to print out the name of the program:

```
program copy
    print("program copy");
```

(In this computer language, every instruction has to end with a semicolon.) Now I need to add a line to print out that second line:

```
program copy
    print("program copy");
    print("    print("program copy");");
```

Note that the second print command has to include the four spaces before the first print command. Now I need another line to print out that third line:

```
program copy
    print("program copy");
    print("    print("program copy");");
    print("    print("    print("program copy");");");
```

and so on. You can see how I am getting into an infinite regress. How can this be avoided? Before reading on, you might spend a few moments trying to solve this puzzle.

This simple sounding problem turns out to be a non-trivial conceptual exercise with echos in the work of Kurt Gödel and Alan Turing, which profoundly shook up the world of mathematics earlier in the 20th century. The solution also contains the essence of how biological

instruction
pointer

```
1 | program test
2 |     print("Hello, world!);
3 |     print("Goodbye.");
4 | end
5 |
. |
. |
. |
```
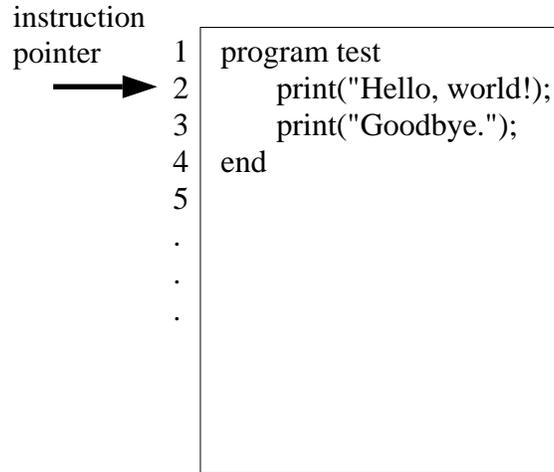
Figure 1: A simplified picture of computer memory, with numbered locations 1–5 and beyond, four of which contain lines of a program. The instruction pointer points to the instruction currently being executed by the computer.

systems themselves get around the infinite regress. The solution was originally found, in the context of a more complicated problem, by the mathematician John von Neumann.

Von Neumann was a pioneer in fields ranging from quantum mechanics to economics, and a designer of one of the earliest electronic computers. His design consisted of a central processing unit which communicates with a random access memory in which both programs and data can be stored. It remains the basic design of all standard computers today. He was also one of the first scientists who thought deeply about connections between computation and biology. He dedicated the last years of his life to solving the problem of how a computer might be able to reproduce itself; his solution was the first complete design for a self-reproducing machine. The self-copying computer program I will show you was inspired by his "self-reproducing automaton" and illustrates its fundamental principle in a simplified way.

Before showing you the self-copying program, I need to explain a few things about the programming language I will be using. It is actually a simplified pseudo-language. You won't need to know much about computer programming to understand the program.

Consider the picture of computer memory given in Figure 1. Computer memory, in our highly simplified case, consists of numbered locations or "addresses," numbered 1–5 and beyond. Here each location contains some text. These lines of text can be interpreted by the computer as commands in a program or as data to be used by a program. The program currently stored in memory, when executed, will print

```
Hello, world!
Goodbye.
```

At any given time during the program's execution, the computer's "instruction pointer" contains the memory location of the instruction currently being executed by the computer. In our pseudo-language, there is a variable `mem` which gives that location. For example, in Figure 1, the instruction pointer is pointing to the line "print("Hello, world!");" and `mem` is equal to 2.

A second variable, `line`, is equal to the string of characters in memory location n. For example, the command `print(line(2));` will print

```
print("Hello, world!");
```

Finally, our language contains a `loop` command. For example, the following lines of computer code,

```
x = 0;
loop until x = 4
{
    print("Hello, world!");
    x = x + 1;
}
```

will print

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

The variable `x` is used as a counter—it starts off at zero and is incremented each time the loop is performed. When it gets to 4, the loop is exited.

Now we are ready for the self-copying program, which appears in Figure 2. The best way to understand a computer program is to hand-simulate it; that is, to go through it line by line keeping track of what it does.

When the user types "copy", this signals the computer to start interpreting the program called `copy`. The interpreter sets the instruction pointer to location 1, containing the name of the program. It then moves down, line by line, executing each instruction.

In location 2 a variable `L` is set to `mem`+1. Recall that `mem` is the location of the instruction currently being executed—here, 2. So `L` is set to $2 + 1 = 3$. Next, the program prints out the first two lines of the program:

```
program copy
    L = mem + 1;
```

```
 1  program copy
 2      L = mem + 1;
 3      print("program copy");
 4      print("    L = mem + 1;");
 5      loop until line[L] = "end"
 6      {
 7          print(line[L]);
 8          L = L + 1;
 9      }
10      print("end");
11 end
```

Figure 2: A self-copying program.

Next, a loop is entered which will be iterated until `line(L)` is equal to "end". Remember that `line(L)` is equal to the string located in memory location L. Right now, L is set to 3, so `line(L)` is equal to "   print("program copy");". This is not equal to "end", so the loop is entered. In the loop, `line[L]` is printed and L is incremented. First, with L= 3,

```
    print("program copy");
```

is printed, and L is set to 4. Now, `line[L]` is the fourth line of the program:

```
    print("    L = mem + 1;");
```

Again, this is not equal to "end", so the loop is continued. In this way, each line of the program is printed out. A particularly interesting line is line 7 itself: when line 7 is being executed with L= 7, the instruction "print(line[L])" prints itself out. When L= 11 and `line[L]` is equal to "end", the loop ends. At this point, lines 1–10 have been printed. The instruction pointer moves to line 10 (the instruction immediately following the loop), which, when executed, prints out the string "end", completing the self-copying.

The essence of self-copying in this program was to use the same information stored in memory in two ways: first as instructions to be executed, and second as data to be used (i.e., printed) by those instructions. This dual use of information is what allows us to avoid an infinite regress of the kind illustrated earlier by our first attempt at a self-copying program. It is also at the essence of biological self-reproduction. DNA is made up of strings of nucleotides, which encode amino acids making up proteins, including the enzymes which effect the splitting of the double helix and the copying of each strand via messenger RNA, transfer RNA, ribosomes, and so on. In a very crude analogy, the DNA encoding the enzymes that effect copying roughly correspond to the lines of code in the self-copying program. These

"lines of code" in DNA are executed when the enzymes are created and act on the DNA itself, interpreting it as data to be split up and copied.

A major difference between the self-copying program above and DNA self-reproduction is that the self-copying program required an interpreter to execute it: an instruction pointer to move one by one down the lines of computer code and a computer operating system to carry them out (e.g., actually perform the storing and retrieving of internal variables like `mem` and L, actually print strings of characters, and so on). The interpreter is completely external to the program itself. However, in the case of DNA, the instructions for building the "interpreter"—the messenger RNA, transfer RNA, ribosomes, and all the other machinery of protein synthesis—are encoded along with everything else in the DNA. Von Neumann's original self-reproducing automaton also contained not only a self-copying program but also the machinery needed for its own interpretation. Thus it was truly a *self*-reproducing machine. That it was formulated in the 1950s, before the details of biological self-reproduction were well understood, is testament to von Neumann's insight. Von Neumann's design and mathematical proofs of its correctness were eventually published in 1966 as a book, *Theory of Self Reproducing Automata* (von Neumann, 1966), completed and edited by his colleague Arthur Burks. (See Burks, 1970 and Mitchell, 1998 for descriptions of von Neumann's self-replicating automaton. See Chapter 16 of Hofstadter, 1979 for an account of self-replication in DNA and how it relates to mathematical logic and self-copying computer programs.)

Von Neumann's design for a self-reproducing automaton was one of the first real advances in the science of artificial life. He recognized it as such, and accordingly took it very seriously, saying that he wanted no mention of the "reproductive potentialities of the machines of the future" made to the mass media.

3   Evolution in Computers

After he answered the question "Can a machine reproduce itself?" in the affirmative, von Neumann wanted to take the next logical step and have computers (or computer programs) reproduce themselves with mutations and compete for resources to survive in some environment. This would counter the "survival instinct" and "evolution and adaptation" arguments mentioned above. However, von Neumann died in 1957 without being able to work on the evolution problem.

Others quickly took up where he left off. By the early 1960s, several groups of researchers were experimenting with evolution in computers. Such work has come to be known collectively as "evolutionary computation" (Fogel, 1995). The most widely known of these efforts today was the work on genetic algorithms done by John Holland and his students and colleagues at the University of Michigan (Holland, 1992; Goldberg, 1989; Mitchell, 1996).

A genetic algorithm (GA) is an idealized computational version of Darwinian evolution. In Darwinian evolution, organisms reproduce at differential rates, with fitter organisms producing more offspring than less fit ones. Offspring inherit traits from their parents; those traits are inherited with variation via random mutation, sexual recombination, and other sources of variation. Thus traits that lead to higher reproductive rates get preferentially spread in

the population, and new traits can arise via variation. In GAs, computer "organisms"—e.g., computer programs encoded as strings of ones and zeros (bit strings)—reproduce in proportion to their fitness in the environment, where fitness is a measure of how well an organism solves a given problem. Offspring inherit traits from their parents with variation coming from random mutation, in which parts of an organism are changed at random, and sexual reproduction, in which an organism is made up of recombined parts coming from its parents.

Assume the individuals in the population are computer programs encoded as bit strings. The following is a simple genetic algorithm.

1. Generate a random initial population of M individuals.

Repeat the following for N generations:

2. Calculate the fitness of each individual in the population. (The user must define a function assigning a numerical fitness to each individual. For example, if the individuals represent computer programs, the fitness of an individual is calculated by running the corresponding computer program and seeing how well it does on a given task.)

3. Repeat until the new population has M individuals:

   (a) Choose two parent individuals from the current population probabilistically as a function of fitness.

   (b) Cross them over at a randomly chosen locus to produce an offspring. That is, choose a position in each bit string, form one offspring by taking the bits before that position from one parent and after that position from the other parent.

   (c) Mutate each locus in the offspring with a small probability.

   (d) Put the offspring in the new population.

4. Go to step 2 with the new population.

This process is iterated for many generations, at which point hopefully one or more high-fitness individuals have been created.

Notice that reproduction in the simple GA consists merely of copying parts of bit strings—von Neumann's complicated design for self-reproduction is avoided by having an external copying routine. This is because research on GAs is typically aimed at solving computational problems via evolution and at investigating evolutionary dynamics in an idealized setting rather than trying to capture all the facets of evolution in biological systems (self-reproduction, complex genotype-to-phenotype mappings via development, and so on).

The simple GA given above is simple indeed, but versions of it that are only slightly more complex have been used to solve problems in many scientific and engineering disciplines. GAs in one form or another have been used for numerical optimization, circuit design, factory scheduling, drug design, telecommunications network optimization, robot navigation, financial-market prediction, and models of the immune system, the economy, and population genetics, to name a few of the areas in which these algorithms have been applied.

4 Evolving Parallel, Decentralized Computing Systems

A major effort in computer science is the development of massively parallel computing systems. A goal is to eventually build systems with thousands or millions of relatively simple processors working in parallel, each following its own set of instructions and communicating with only a small number of other processors, and no central controller telling each processor what to do. In principle the *collective* behavior of such a system would allow problems to be solved very quickly that no single processor or small group of processors could solve efficiently alone.

Such parallel, decentralized systems contrast with the traditional architecture for computers in which a single central processing unit (CPU) communicating with a random-access memory runs a single program, one step at a time. This traditional architecture is known as a "von-Neumann-style architecture" since it follows von Neumann's and colleagues' original design for programmable computers. It has been extended to parallel processing, where typically one processor plays the role of a central controller for the other processors.

Massively parallel, decentralized computing systems as described above are more like biological systems than are traditional computers. For example, the brain is a massively parallel system of billions of tiny processors (neurons) with limited communication with one another and no central controller running the show. Instead the collective actions of the neurons (and their attendant synapses, neurotransmitters, etc.) lead to the states we call "thinking", "consciousness", and so on in ways we do not yet understand very well. Other complex systems in the living world, such as ant colonies or the immune system, have similar decentralized architectures and similarly "emergent" collective abilities.

These biological systems have many properties that are desired in computational systems. First, such systems, particularly neural systems, are extremely fast and capable at performing tasks such as recognizing complex patterns and responding in appropriate ways that are still far beyond the abilities of current computers. Second, biological systems are robust, in that if some processors (neurons, ants, white blood cells) fail, the whole system does not come to a crashing halt like a traditional computer would, but is able to keep running, usually quite satisfactorily. Third, biological systems are very good at adapting to changing circumstances. Such adaptability is essential for intelligence and autonomy in the real world.

For these reasons, building computer systems that are, like living systems, massively parallel and decentralized, is an important goal of computer science. The difficulty, however, is that such systems are very difficult to design, program, and control, precisely because of their decentralized nature. Some computer scientists, including myself, believe that a promising avenue is to go one step further in mimicking nature: to use evolution, in the form of genetic algorithms or other evolutionary computation approaches, to design such systems rather than to engineer them ourselves by hand.
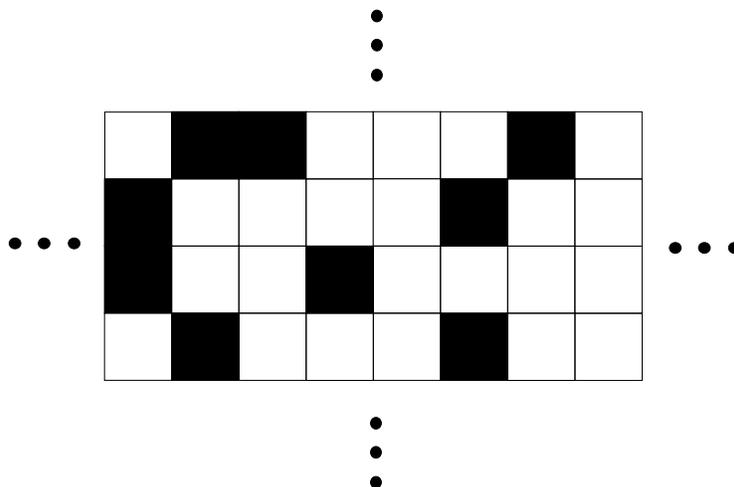
Figure 3: Illustration of a 2D cellular automaton. (a) The two dimensional lattice. (b) Illustration of a cell's neighborhood: here cell 0 and its 8 neighbors.

5   Cellular Automata

My colleagues and I have attempted an idealized version of this long-term project by using genetic algorithms to design cellular automata to perform computations. Sections 6–9 below summarize some of our previously published work (Crutchfield & Mitchell, 1995; Das, 1998; Das, Mitchell, & Crutchfield, 1994; Mitchell, Crutchfield, & Das, 1996).

Cellular automata (CAs) are idealized versions of the massively parallel, decentralized computing systems I described above. A CA is a large network of simple processors, with limited communication among the processors and no central control, that can produce very complex dynamics from simple rules of operation. A two-dimensional cellular automaton is illustrated in Figure 3. It consists of a two-dimensional lattice of "cells", represented as squares in the figure, each of which communicates with the 8 cells that surround it. These 8 cells along with the cell itself is known as the "neighborhood" of a cell. Each cell starts out at time $t = 0$ as either in the black state ("on") or in the white state ("off"). The whole pattern of states at $t = 0$ is called the "initial configuration". The cellular automaton iterates over a series of discrete time steps ($t = 1, 2, 3, \ldots$). At each time step, each cell updates its state as a function of its current state and the current state of the 8 neighbors to which it is connected. (The boundary cells are dealt with by having the whole lattice wrap around into a torus—thus boundary cells are connected to "adjacent" cells on the opposite boundary.) Each cell uses the same function for updating its state and all cells update synchronously in parallel at each time step. For example, the well-known Game of Life (Berlekamp, Conway, & Guy, 1982) is a two-dimensional cellular automaton with the following update function followed by each cell:

> If the cell is white at time t, it will turn black at $t + 1$ if exactly three of its eight neighbors are black at time t; otherwise it will stay white at $t + 1$.
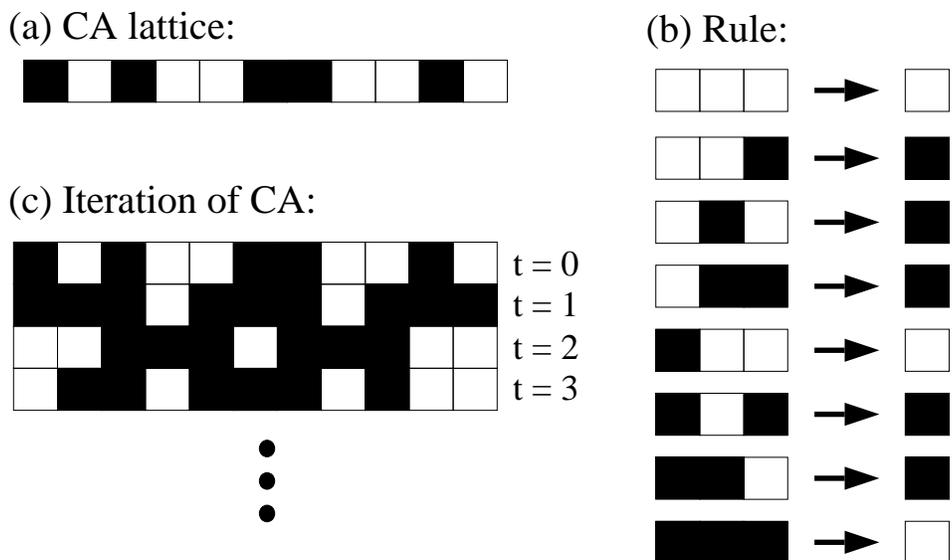
Figure 4: Illustration of a one-dimensional CA. (a) CA lattice at $t = 0$. (b) The rule table for this CA. (c) Iteration of the CA for three time-steps.

> If the cell is black at time t, it will stay black at $t + 1$ if two or three of its eight neighbors are black at time t. Otherwise it will turn white at $t + 1$.

As readers who have seen the Game of Life on their computer screen-savers (or other places) will know, these simple rules can lead to very complicated behavior.

Being massively parallel decentralized systems, cellular automata are quintessential examples of a "non-von-Neumann-style" computer architecture, as it is called in the computer science community. This is ironic, since von Neumann himself was one of the inventors of cellular automata. In fact, von Neumann's self-reproducing automaton was itself embedded in a cellular automaton.

In our project on evolving cellular automata with genetic algorithms, we focussed our attention on the simplest type of cellular automaton—one dimensional arrays of cells. A one-dimensional cellular automaton and its rule table (or "rule") are illustrated in Figure 4. A one-dimensional lattice of 11 cells is illustrated in Figure 4a. Here each cell is connected to its two immediate neighbors, and the ends of the lattice wrap around in a circle, so the right-hand neighbor of the rightmost cell is the leftmost cell and vice versa. Let a "neighborhood" of a cell be defined as the cell itself and the other neighbors to which it is connected. The update rule is given as a table which lists the eight possible neighborhood configurations formed by a neighborhood of three cells, and the state that the center cell takes on at time $t + 1$ if it is in the corresponding configuration at time t (Figure 4b). For example, a white cell surrounded by two black cells at time t will turn black at time $t + 1$. The iteration of the rule over four time steps, starting from a random initial configuration of cell states at $t = 0$, is illustrated in Figure 4c.

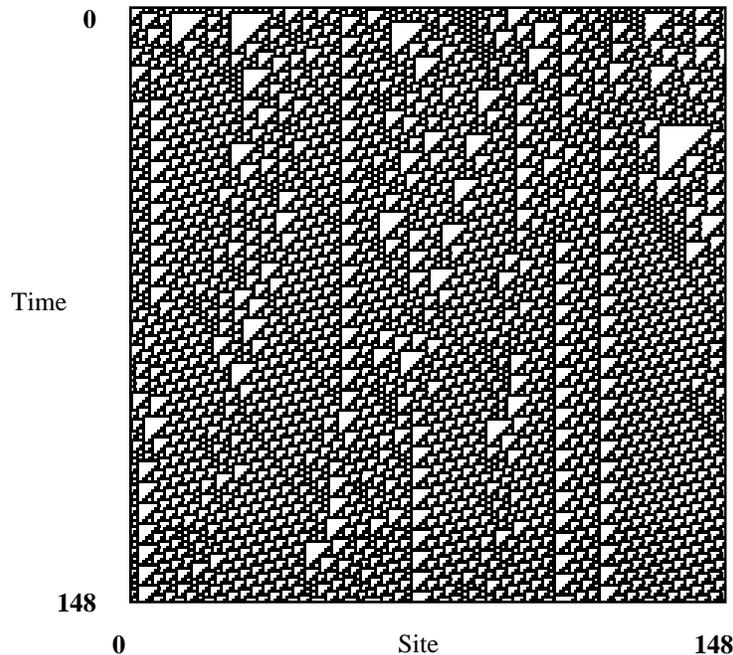Figure 5 displays a "space-time diagram" illustrating the behavior of this cellular au-

Figure 5: A space-time diagram, illustrating the behavior of the CA rule given in Figure 4b on a lattice of 149 cells, iterated from a random initial configuration for 148 time steps. This diagram and all other space-time diagrams in this paper were generated with la1d, a cellular automaton simulation and graphics package written by James P. Crutchfield.

tomata on a larger array (149 cells), iterated for 148 time steps. As before, the cells in the one-dimensional CA are arrayed horizontally, with black representing the "on" state and white the "off" state, and time increases down the page. At this larger scale, complicated-looking patterns start to emerge—white triangles of different sizes appearing in irregular-seeming ways throughout the space-time diagram. In principle, given the eight update rules of Figure 4b, this cellular automaton's behavior is completely specified in a very simple way. But in practice it is very difficult to predict the long-term behavior of the cellular automaton starting from a given initial configuration—e.g., when a triangle of a certain size will appear—without actually iterating it.

Thus in cellular automata very simple rules can produce very complicated dynamics. Our group's question was, can the complex dynamics be harnessed by evolution to perform sophisticated collective information processing, as it is in living systems?

6 A Task for Cellular Automata Requiring Collective Information Processing

To answer this question, we looked at a task for cellular automata—the "density classification" task—that requires collective information processing. The task is to decide whether
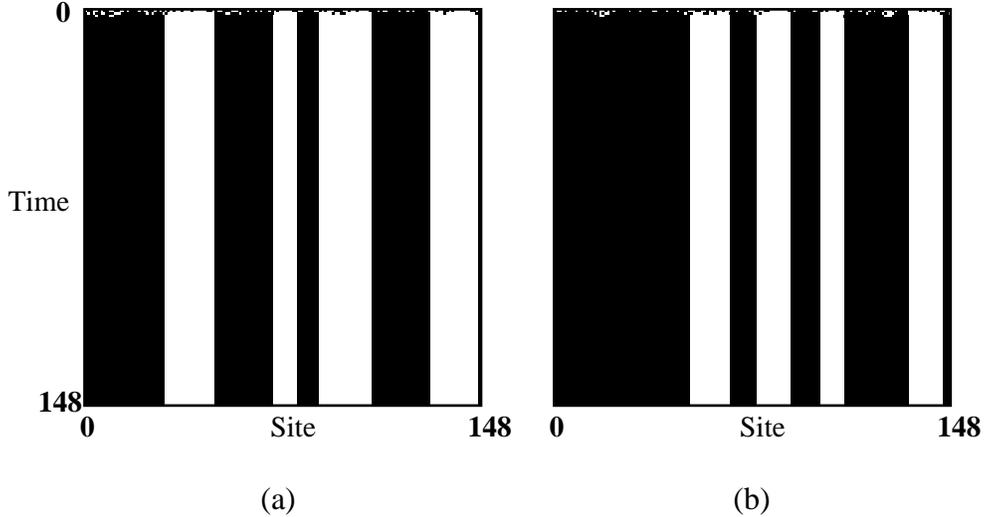
Figure 6: Space-time behavior of the majority CA from two random initial configurations: (a) majority white; (b) majority black.

or not the initial configuration of cell states has a majority of black cells ("high density"). If yes, then after some number of iterations all cells should turn black and stay black. If no, then after some number of iterations all cells should turn white and stay white. This task was first used for studying collective computation in cellular automata by Packard (1988). The goal is to design a cellular automaton that can perform this task for a large set of initial configurations. For this task we used one-dimensional CAs where each cell is connected to three cells on either side, so the number of cells in each neighborhood is 7.

This task would be trivial for a system with a central controller or central storage of some kind, such as a von Neumann-style computer: it would be a simple matter for the central controller to count up the number of black states, saving the intermediate sum in a register, and then divide the total by the lattice length. However, it is nontrivial to design a decentralized system with limited communication among components to perform this task. It has been proven that no CA of the type we are using can perform this task perfectly (Land & Belew, 1995), but even to perform this task reasonably well requires more powerful computation than can be performed by a single cell or any linear combination of cells. Since the black cells can be distributed throughout the lattice, the CA must transfer information over large distances and process information collected from different parts of the lattice. To do this requires the global coordination of cells that are separated by large distances and that cannot communicate directly.

The need for such coordination is illustrated in Figure 6, which displays the space-time behavior of a "naive" hand-designed candidate solution for this task—the "majority" CA rule, called ϕ_{maj}, which simply takes a majority vote in each neighborhood. The state each cell becomes at time t is simply the state that is in the majority for that cell and its neighbors at time $t - 1$. As can be seen, over time local neighborhoods with majority of black states converge to regions of all black and similarly for white, but when a black region and a white

12

CA rule:

Representation of CA rule in GA population:
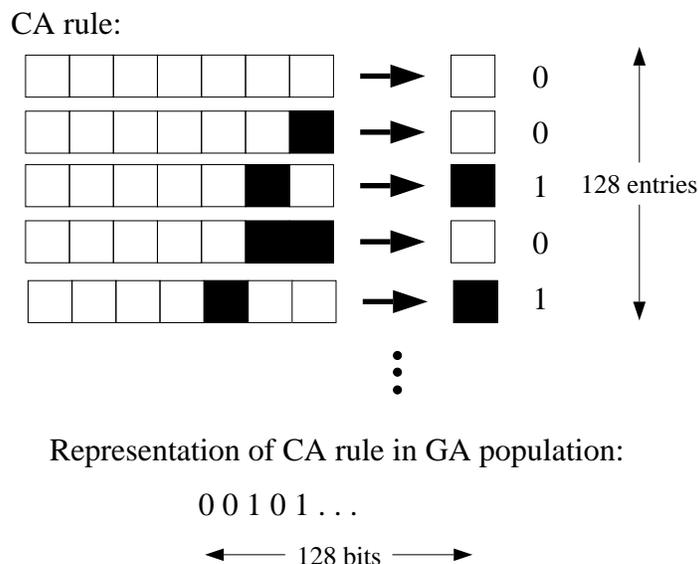
0 0 1 0 1 . . .

◀━━ 128 bits ━━▶

Figure 7: Illustration of how a seven-cell-neighborhood 1D CA is represented as a bit string for the GA. The CA rule table has $2^7 = 128$ entries. The GA bit string has 128 bits, representing the output states in the rule table, in lexicographic order of neighborhood configuration.

region border each other, there is no way to decide between them, and both persist. Thus ϕ_{maj} does not perform the density classification task.

Instead, more sophisticated coordination and information transfer must be achieved. This coordination must, of course, happen in the absence of any central processor or central memory directing the coordination.

7 Evolving Cellular Automata with Genetic Algorithms

We used a genetic algorithm to evolve cellular automata perform the density classification task. Each individual in the population is a bit string representing a cellular automaton rule table. This is illustrated in Figure 7. We list all the 128 possible neighborhood configurations of seven cells in lexicographic order, starting with the neighborhood of all white cells and ending with the neighborhood of all black cells. Each neighborhood configuration is listed with an update state that the center cell becomes at the next time step. If we call the black state "1" and the white state "0", the update states, in order, form a bit string of length 128. Such bit strings are the individuals that undergo GA evolution.

In the initial generation a population of such bit strings is generated at random. The fitness of each is then calculated by translating the bit string into a CA rule and trying the resulting CA out on a number of initial configurations of states. In a typical simulation, the GA's population was 100 CA rules. Each corresponding CA had 149 cells. Each CA

rule was tested on 100 initial configurations of cell states. A "trial" consisted of starting at a given initial configuration and then updating the 149 cells in parallel, using the given rule table, until a fixed configuration was reached or until 298 time steps (twice the lattice length) had elapsed, whichever happened first. The fitness of the CA rule was the fraction of the 100 trials on which the CA reached the correct answer (a fixed configuration of all black for initial majority black, all white for initial majority white) before the maximum number of time steps elapsed.

After the fitnesses of all the CA rules in the population were calculated, the genetic algorithm selected the highest fitness individuals to be parents and created a new population via reproduction from those parents using crossover and mutation. The whole process was then repeated for the new generation. The GA was run for 100 generations.

The algorithms involved and the parameters they use are explained in more detail in Das, Mitchell, and Crutchfield (1994) and Crutchfield, Mitchell, and Das (1998). In our runs, parallelism is simulated on a serial computer. To put the running time into perspective: the GA evaluates a total of 10,000 CA rules (100 per generation times 100 generations). This may seem like a lot, but consider that the number of *possible* CA rules with 7 cells in a neighborhood is 2^{128}, one of those numbers that far outstrips the number of particles in the universe or anything else you probably care to imagine. Each of these 10,000 CA rules are tested on 100 initial configurations, for a total of one million (10^6) CA trials during a run of the GA. Each CA trial consists of (at most) 298 time steps. At each time step 149 cells are updated. So the maximum number of cell updates during a GA run is $10^6 \times 298 \times 149$, which is approximately 44 billion. Such a simulation would be far beyond what von Neumann's computers of the 1940s and 1950s would have been capable of in reasonable time. However, with today's fast hardware, this kind of extensive evolutionary simulation is eminently feasible: each GA run takes less than two minutes on my 500 MH Pentium III workstation.

8   Results of Evolution

We performed thousands of runs of the GA on this problem, and on many runs the GA found high-performing CA rules that used sophisticated collective computation to perform the density classification task. (Detailed results of our simulations are described in Crutchfield, Mitchell, & Das, 1998.) The behavior of one of the best CA rules evolved by the GA is illustrated in Figure 8. Call this rule ϕ_{GA}. In Figure 8(a), the initial configuration has a majority of black cells and eventually all cells become black and remain in that state; in Figure 8(b), the initial configuration has a majority of white cells and eventually all cells become white and remain in that state. In both cases, ϕ_{GA} reaches the correct classification. When tested on a set of 10,000 randomly generated initial configurations on 149-cell lattices, ϕ_{GA} reaches the correct classification approximately 80% of the time. Such random initial configurations are the hardest cases, since a randomly generated string of white and black cells will have close to 50% of each state. The performance of ϕ_{GA} improves as the density of black cells in the initial configuration increases or decreases away from 50%.
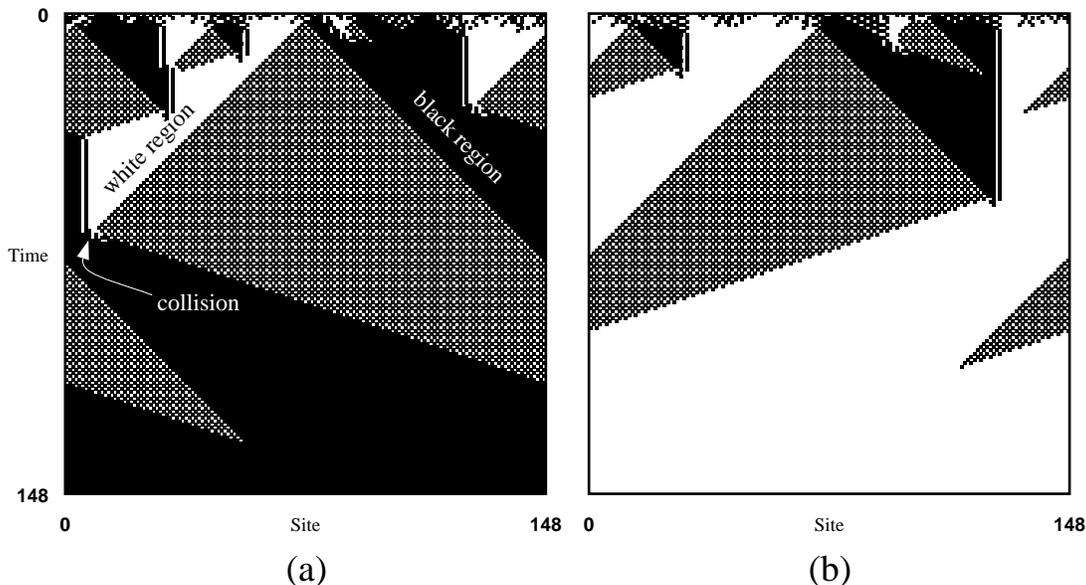
Figure 8: Space-time behavior of ϕ_{GA}, one of the best evolved CA rules for the density classification task. In (a), the initial condition contains a majority of black cells and the CA iterates to a fixed configuration of all black. In (b), the initial condition contains a majority of white cells and the CA iterates to a fixed configuration of all white.

How does computation occur in ϕ_{GA}? In particular, how do the cells collectively obtain and process the information in the initial configuration? I will first describe qualitatively how ϕ_{GA} performs its density classification, and then sketch our approach towards a more quantitative description of computation in decentralized spatially extended systems like ϕ_{GA}.

In Figure 8 it can be seen that, under ϕ_{GA}, between time $t = 0$ and when all cells have turned black or white there is a "transient" phase during which patterns are formed. It is these patterns that transfer information about the density of black cells in local regions. Roughly, over short times, ϕ_{GA}'s behavior is similar to that of ϕ_{maj} in that local high-density regions are mapped to all black, local low-density regions are mapped to all white, with a vertical boundary in between them. This is what happens when a region of black cells on the left meets a region of white cells on the right. However, there is a crucial difference from ϕ_{maj}: when a region of white cells on the left meets a region of black cells on the right, rather than a vertical boundary being formed, a checkerboard region (alternating black and white cells) is propagated. When the propagating checkerboard region collides with the vertical black-white boundary, the inner region is cut off and the outer region is allowed to propagate. For example, in Figure 8a, the marked white region is smaller than the marked black region that wraps around the edges—thus the propagating checkerboard pattern reaches the black-white boundary on the white side before it reaches it on the black side; the former is cut off, and the latter is allowed to propagate. In this way, the CA uses local interactions and geometry to determine the relative sizes of adjacent large low and high density regions.

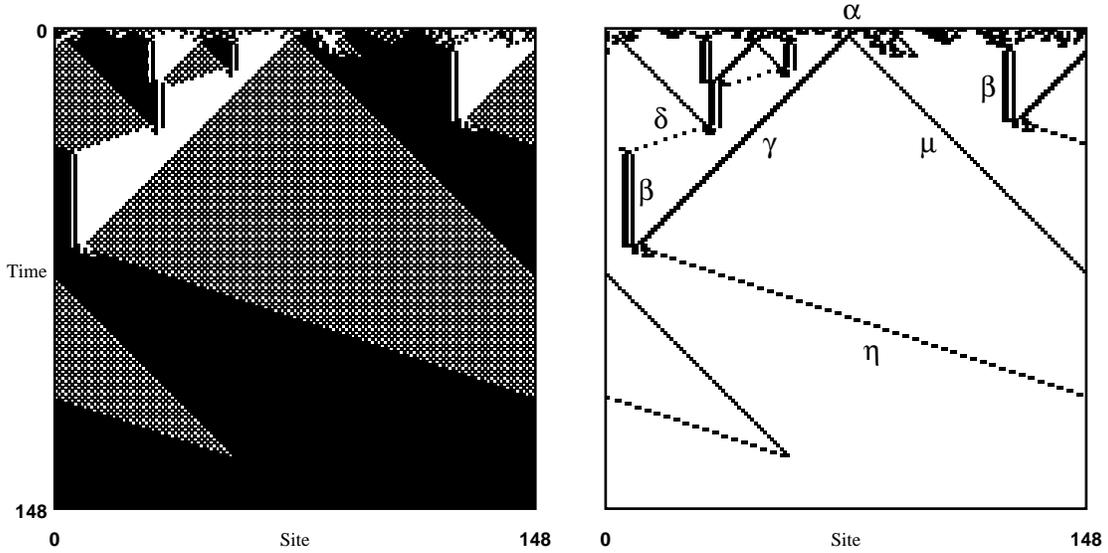The black-white boundary and the checkerboard region can be thought of as signals indi-

Figure 9: The space-time diagram of Figure'8(a) and its filtered version, in which the regular domains are colored white, leaving the boundaries between these domains ("particles").

cating ambiguous regions. The creation and interactions of these signals can be interpreted as collective information processing performed by the CA.

The above explanation of how ϕ_{GA} performs the density classification task is informal and incomplete. Can we understand more rigorously how the evolved CAs perform the desired classification? Understanding the products of GA evolution is a general problem—typically the GA is asked to find individuals that achieve high fitness but is not told what traits the individuals should have to attain such fitness. This is analogous to the difficulty biologists have in understanding the products of natural evolution. In the case of cellular automata, the collective computation performed by a given CA is determined by its overall space-time behavior, and is thus almost always impossible to extract from the rule table.

A more rigorous and quantitative description of information processing in CAs can be obtained via Crutchfield and Hanson's "computational mechanics" approach. (Crutchfield & Hanson, 1993; Hanson & Crutchfield, 1992). Crutchfield and Hanson give a general method for describing the "intrinsic computation" embedded in space-time behavior in terms of "regular domains," "particles," and "particle interactions." A detailed discussion of their method requires knowledge of formal language theory and automata theory, and is beyond the scope of this paper. Very briefly, regular domains are simple-to-describe patterns such as "all black", "all white", and "checkerboard." Particles are the localized boundaries between these patterns. Figure 9 shows how the regular domains created by ϕ_{GA} can be filtered out, leaving the particles (here labeled with Greek letters). Particles are interpreted as information carriers, and collisions between particles are interpreted as the loci of information processing.

For example, the α and β particles encode different types of ambiguity in the initial configuration (large black and white regions bordering one another). α decays into γ and μ. γ

16

carries the information that it borders a white region and μ carries the information that it borders a black region. These two particles, by having the same velocity, also carry the mutual information of having come from the same ambiguous region in the initial configuration. When γ collides with β before μ does, the information contained in β and γ is combined to deduce that the large initial white region was smaller than the large initial black region it bordered. This new information is encoded in a newly created particle η, whose job is to catch up with and annihilate the μ (and itself).

In Hordijk, Crutchfield, and Mitchell (1998) and Hordijk (1999), it is shown how a description of information processing at the level of particles and their interactions can explain and predict both qualitative and quantitative aspects of CA performance on the density-classification task and other tasks.

9   Evolutionary History of ϕ_{GA}

It is of interest to see how high-performance CAs like ϕ_{GA} come about under GA evolution. Figure 10(a) plots the best fitness in the population as a function of generation for the run in which ϕ_{GA} evolved. Space-time diagrams illustrating the behaviors of the best CA rules at generations 7, 8, 13, 17, and 18 are given in Figure 10(b)–(f). These were generations at which particular innovations were discovered by the GA. At generation 7, the highest-fitness CA always immediately converges to all black from any initial configuration. This, of course, results in getting half the cases (the majority-black ones) correct, sort of like answering "yes" to all yes/no questions. By generation 8, this "default" solution is improved upon a bit—the CA will converge to all white if given a very low density initial configuration (not shown here). In addition, at generation 8 a checkerboard pattern begins to appear. At this generation, the checkerboard pattern is not actually being used to increase fitness; when the CA rule is changed by hand to eliminate the checkerboard, the fitness stays roughly the same. The checkerboard-like pattern seen in generation 13 is likewise adaptively neutral. However, evolution tinkers with this pattern (via crossover and mutation) and by generation 17 the checkerboard is central to the higher-fitness strategy of the highest-fitness CA at that generation. Thus in this idealized world of a GA evolving CA rules, we see a key phenomenon of evolution: adaptively neutral traits arise by accident and are later modified by evolution to have adaptive significance.

Between generation 17 and 18 there is a big jump in fitness, partially due to the fact that the black-to-white particle β (see Figure 9) changes from having a positive velocity (slanted to the right) to a zero velocity (vertical). In generation 17, because β has a positive velocity it expands black regions and shrinks white regions, allowing black an advantage over white. This is seen in Figure 10(d), where, as it happens, the initial configuration had a majority of white cells, but the CA incorrectly converges to all black. In generation 18 (Figure 10(e)), β has velocity zero so neither black nor white expand unfairly and the correct classification is reached. By generation 18, the basic computational strategy described for ϕ_{GA} has been found, and from generation 18 on, no large improvements were made in fitness.

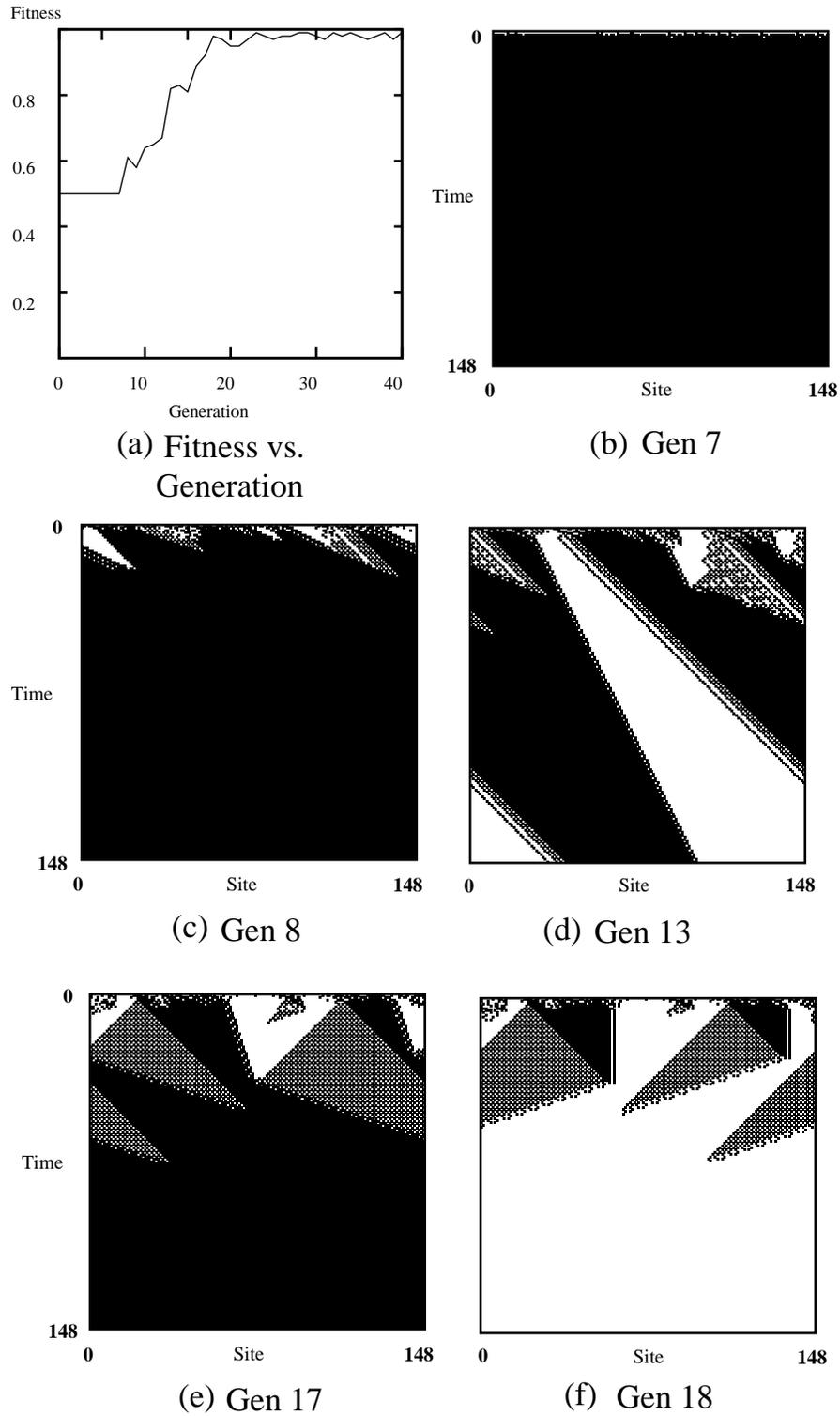This evolutionary history is given in more detail in terms of particles in Das, Mitchell,

Figure 10: (a) Plot of best fitness in the population at each generation. (b)–(f): Space-time behavior of the highest-fitness CA in the population at generations 7, 8, 13, 17, and 18.

and Crutchfield (1994), Crutchfield, Mitchell, and Das (1998), and Hordijk (1999).

10 Conclusion

In this paper I've highlighted some approaches to building life-like computing systems: von Neumann's design of a self-reproducing automaton, illustrated by a self-copying program; Holland and others' invention of evolutionary computation methods such genetic algorithms; and recent work by myself and colleagues on evolving cellular automata with collective computing abilities. This work, along with other research on artificial life and artificial intelligence, demonstrates that we can create some of the attributes of life and intelligence, such as self-reproduction, evolution, and adaptation, in machines, albeit in rudimentary ways. Such *in silico* experiments give us what the philosopher Daniel Dennett has called "intuition pumps" (Dennett, 1991): simple models that help us make sense of difficult ideas in a concrete rather than abstract way. I believe that many concrete and progressively realistic examples of such systems will be necessary for us to make sense of questions like "Can we build computers that are intelligent and alive?" and to extend our intuitions about what life is and what computers are and what they can be.

11 Acknowledgments

12 References

Adami, C., 1998, *Artificial Life VI*, Cambridge, MA: MIT Press.

Berlekamp, E., Conway, J.H., Guy, R., 1982, *Winning ways for your mathematical plays*, Volume 2, New York, NY: Academic Press.

Brooks, R.A., Maes, P., 1994, *Artificial Life IV*, Cambridge, MA: MIT Press.

Burks, A.W., 1970, 'Von Neumann's Self-Reproducing Automata'. In: Burks, A.W. (ed.) *Essays on Cellular Automata*, Urbana, IL: Univerity of Illinois Press.

Crutchfield, J.P., Hanson, J.E., 1993, 'Turbulent Pattern Bases for Cellular Automata', *Physica D*, 69, 279-301.

Crutchfield, J.P., Mitchell, M., 1995, 'The Evolution of Emergent Computation', *Proceedings of the National Academy of Science U.S.A.*, 92, 10742–10746.

Available at www.santafe.edu/projects/evca/Papers/EvEmComp.html.

Crutchfield, J.P., Mitchell, M., Das, R., 1998, 'The Evolutionary Design of Collective Computation in Cellular Automata', Working Paper 98-09-080, Santa Fe Institute, Santa Fe, NM. Available at www.santafe.edu/projects/evca/Papers/EvDesign.html.

Das, R., 1998, *The Evolution of Emergent Computation*, Ph.D. Thesis, Colorado State University, Fort Collins, Colorado.

Das, R., Mitchell, M., Crutchfield, J.P., 1994, 'A Genetic Algorithm Discovers Particle-Based Computation in Cellular Automata." In Davidor, Y., Schwefel, H.-P., Männer, R. (eds.), *Parallel Problem Solving From Nature—PPSN III*, Berlin: Springer-Verlag (Lecture Notes in Computer Science), 344-353. Available at www.santafe.edu/projects/evca/Papers/GA-Particle.html.

Dennett, D.C., 1991, *Consciousness explained*, Boston: Little, Brown and Co.

Fogel, D.B., 1995, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, New York: IEEE Press.

Goldberg, D.E., 1989, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Reading, MA: Addison-Wesley.

Hanson, J.E., Crutchfield, J.P., 1992, 'The Attractor-Basin Portrait of a Cellular Automaton', *Journal of Statistical Physics*, 66 (5/6), 1415-1462.

Hofstadter, D.R., 1979, *Gödel, Escher, Bach: an Eternal Golden Braid*, New York: Basic Books.

Holland, J.H., 1992, *Adaptation in Natural and Artificial Systems*, Cambridge, MA: MIT Press. Second edition (First edition, 1975).

Hordijk, W., 1999. *Dynamics, Emergent Computation, and Evolution in Cellular Automata*, Ph.D. Thesis, University of New Mexico, Albuquerque, New Mexico.

Hordijk, W., Crutchfield, J.P., Mitchell, M., 1998, 'Mechanisms of Emergent Computation in Cellular Automata.' In Eiben, A.E. (ed.), *Parallel Problem Solving from Nature—Proceedings Vth Workshop PPSN V*, New York: Springer.
Available at www.santafe.edu/projects/evca/Papers/mecca.html.

Land, M., Belew, R.K., 1995, 'No Perfect Two-State Cellular Automata for Density Classification Exists', *Physical Review Letters*, 74 (25), 5148.

Langton, C.G. (editor), 1989, *Artificial Life*, Reading, MA: Addison-Wesley.

Langton, C.G. (editor), 1993, *Artificial Life III*, Reading, MA: Addison-Wesley.

Langton, C.G., 1995, *Artificial Life: An Overview* Cambridge, MA: MIT Press.

Langton, C.G., Shimohara, K., 1997 *Artificial Life V*, Cambridge, MA: MIT Press.

Langton, C.G., Taylor, C., Farmer, J. D., Rasmussen, S. (editors), 1992, *Artificial Life II*, Reading, MA: Addison-Wesley.

Mitchell, M., 1996, *An Introduction to Genetic Algorithms*, Cambridge, MA: MIT Press.

Mitchell, M., 1998, 'Computation in Cellular Automata: A Selected Review'. In Gramss, T. et al. (eds.), *Nonstandard Computation*, Weinheim, Germany: Wiley-VCH. Available at www.santafe.edu/ mm/ca-review.html.

Mitchell, M., Crutchfield, J.P., Das, R., 1996, 'Evolving Cellular Automata to Perform Computations: A Review of Recent Work.' In *Proceedings of the First International Conference on Evolutionary Computation and its Applications (EvCA '96)*, Moscow, Russia: Russian Academy of Sciences. Available at www.santafe.edu/projects/evca/Papers/evca-review.html

Packard, N.H., 1988, 'Adaptation Toward the Edge of Chaos.' In Kelso, J.A.S., Mandell, A.J., Shlesinger, M.F. (eds.), *Dynamic Patterns in Complex Systems*, Singapore: World Scientific, 293–301.

von Neumann, J., 1966, *Theory of Self-Reproducing Automata* (Edited and completed by A. W. Burks), Urbana, IL: University of Illinois Press