CHAPTER 10

Cellular Automata, Life, and the Universe

Computation in Nature

A recent article in *Science* magazine, called "Getting the Behavior of Social Insects to Compute," described the work of a group of entomologists who characterize the behavior of ant colonies as "computer algorithms," with each individual ant running a simple program that allows the colony as a whole to perform a complex computation, such as reaching a consensus on when and where to move the colony's nest.

This would be an easy computation for me to program on my computer: I could just appoint one (virtual) ant as leader and decision maker. All the other ants would simply observe the leader's decision and follow it. However, as we have seen, in ant colonies there is no leader; the ant-colony "computer" consists of millions of autonomous ants, each of which can base its decisions and actions only on the small fraction of other ants it happens to interact with. This leads to a kind of computation very different from the kind our desktop computers perform with a central processing unit and random-access memory.

Along the same lines, a 1994 article by three prominent brain researchers asked, "Is the brain a computer?" and answered, "If we embrace a broader concept of computation, then the answer is a definite Yes." Like ant colonies, it is clear that the way the brain computes—with billions of neurons working in parallel without central control—is very different from the way current-day digital computers work.

In the previous two chapters we explored the notion of life and evolution occurring *in* computers. In this part of the book, we look at the opposite notion; the extent to which computation itself occurs in nature. In what sense do natural systems "compute"? At a very general level, one might say that *computation* is what a complex system does with *information* in order to succeed or adapt in its environment. But can we make this statement more precise? Where is the information, and what exactly does the complex system do with it?

In order to make questions like this more amenable to study, scientists generally will *idealize* the problem—that is, simplify it as much as possible while still retaining the features that make the problem interesting.

In this spirit of simplification, many people have studied computation in nature via an idealized model of a complex system called a *cellular automaton*.

Cellular Automata

Recall from chapter 4 that Turing machines provide a way of formalizing the notion of "definite procedure"—that is, computation. A computation is the transformation of the input initially on a Turing machine's tape, via the machine's set of rules, to the output on its tape after the **halt** state is reached. This abstract machine inspired the design of all subsequent digital computers. Because of John von Neumann's contribution to this design, our current-day computers are called "von-Neumann-style architectures."

The von-Neumann-style architecture consists of a random access memory (RAM) that stores both program instructions and data, and a central processing unit (CPU) that fetches instructions and data from memory and executes the instructions on the data. As you probably know, although programmers write instructions in high-level programming languages, instructions and data are actually stored in the computer as strings of 1s and 0s. Instructions are executed by translating such bit strings into simple logic operations applied to the data, which are then computed by the CPU. Only a few types of simple logic operators are needed to perform any computation, and today's CPUs can compute billions of these logic operations per second.

A cellular automaton, being an idealized version of a complex system, has a very different kind of architecture. Imagine a grid of battery-powered lightbulbs, as shown in figure 10.1. Each lightbulb is connected to all of its neighboring lightbulbs in the north, south, east, west, and diagonal directions. In the figure, these connections are shown for only one of the lightbulbs, but imagine that all the other ones have corresponding connections.

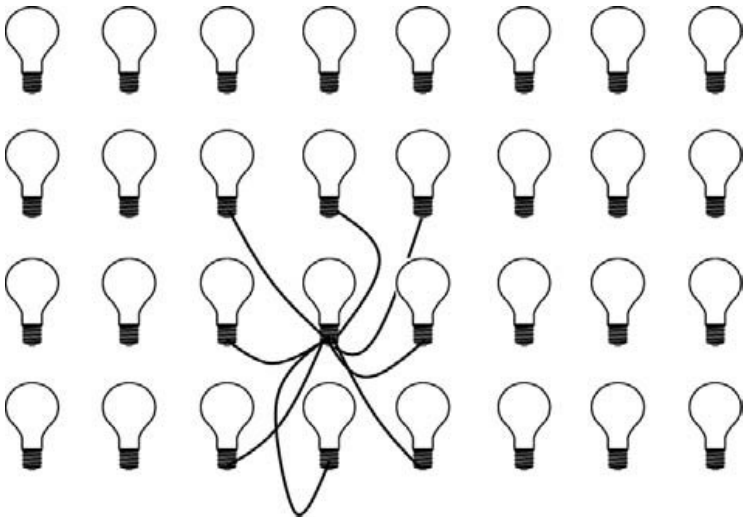

FIGURE 10.1. An array of lightbulbs, each of which is connected to its neighbors in the north, south, east, west, and diagonal directions, as is illustrated for one of the lightbulbs. Each lightbulb can either be in state **on** or state **off**. Imagine that all four edges wrap around in a circular fashion—for example, the upper left bulb has the upper right bulb as its western neighbor and the lower left bulb as its northern neighbor.

In figure 10.2 (left box), some of the lightbulbs have been turned on (to make the figure simpler, I didn't draw the connections). After this initial configuration of on and off lightbulbs has been set up, each lightbulb will run a clock that tells it when to "update its state"— that is, turn on or off; and all the clocks are synchronized so all lightbulbs update their states at the same time, over and over again. You can think of the grid as a model of fireflies flashing or turning off in response to the flashes of nearby fireflies, or of neurons firing or being inhibited by the actions of close-by neurons, or, if you prefer, simply as a work of abstract art.

How does a lightbulb "decide" whether to turn on or off at each time step? Each bulb follows a rule that is a function of the states in its *neighborhood*—that is, its own state (i.e., **on** or **off**) and those of the eight neighbors to which it is connected.

For example, let's say that the rule followed by each lightbulb is, "If the majority of bulbs in my neighborhood (including myself) are on, turn on (or stay on, if already on), otherwise turn off (or stay off, if already off)." That is, for each neighborhood of nine bulbs, if five or more of them are on, then the middle one is on at the next time step. Let's look at what the lightbulb grid does after one time step.

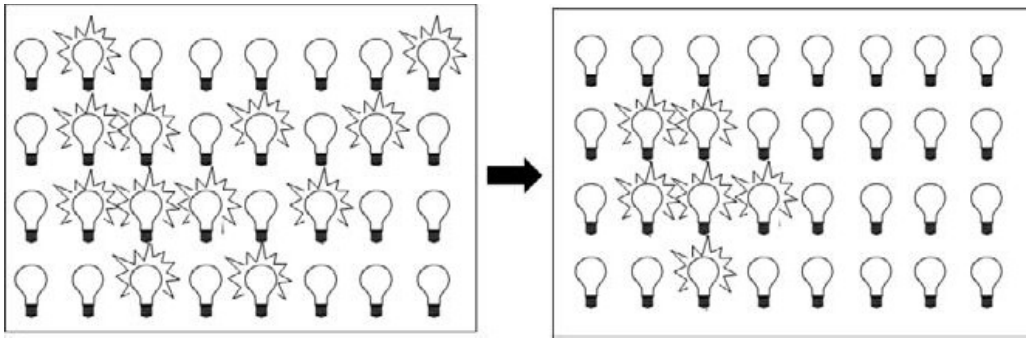

FIGURE 10.2. Left: The same array of lightbulbs as in figure 10.1, set up in an initial configuration of **on** and **off** states. Connections between lightbulbs are not shown. Right: each bulb's state has been updated according to the rule "take on whichever state is a majority in my local neighborhood."

As explained in the caption to figure 10.1, to make sure that each lightbulb indeed has eight neighbors, we will give the grid circular boundaries. Imagine that the top edge is folded over and touches the bottom edge, and the left edge is folded over and touches the right edge, forming a donut shape. This gives every lightbulb eight neighbors.

Now let's go back to the rule defined above. Figure 10.2 shows the initial grid and its configuration after following the rule for one time step.

I could have defined a more complicated rule, such as, "If at least two but no more than seven bulbs in my neighborhood are on, then turn on, otherwise turn off," and the updated grid would have looked different. Or "if exactly one bulb is off or exactly four bulbs are on in my neighborhood, turn off, otherwise turn on." There are lots of possibilities.

Exactly how many possible different rules could be defined? "Lots" is really understating it. The answer is "two raised to the power 512" (2^{512}), a huge number, many times larger than the number of atoms in the universe. (See the notes to find out how this answer was derived.)

This grid of lightbulbs is a cellular automaton. More generally, a cellular automaton is a grid (or *lattice*) of *cells*, where a cell is a simple unit that turns on or off in response to the states in its local neighborhood. (In general, cells can be defined with any number of states, but here we'll just talk about the on/off kind.) A cellular automaton *rule*—also called a *cell update rule*—is simply the identical rule followed by each cell, which tells the cell what its state should be at the next time step as a function of the current states in its local neighborhood.

Why do I say that such a simple system is an idealized model of a complex system? Like complex systems in nature, cellular automata are composed

of large numbers of simple components (i.e., cells), with no central controller, each of which communicates with only a small fraction of the other components. Moreover, cellular automata can exhibit very complex behavior that is difficult or impossible to predict from the cell update rule.

Cellular automata were invented—like so many other good ideas—by John von Neumann, back in the 1940s, based on a suggestion by his colleague, the mathematician Stan Ulam. (This is a great irony of computer science, since cellular automata are often referred to as **non**-*von-Neumann-style* architectures, to contrast with the *von-Neumann-style* architectures that von Neumann also invented.) As I described in chapter 8, von Neumann was trying to formalize the *logic* of self-reproduction in machines, and he chose cellular automata as a way to approach this problem. In short, he was able to design a cellular automaton rule, in his case with twenty-nine states per cell instead of just two, that would create a perfect reproduction of any initial pattern placed on the cellular automaton lattice.

Von Neumann also was able to show that his cellular automaton was equivalent to a universal Turing machine (cf. chapter 4). The cell update rule plays the role of the rules for the Turing machine tape head, and the configuration of states plays the role of the Turing machine tape—that is, it encodes the *program* and *data* for the universal machine to run. The step-by-step updates of the cells correspond to the step-by-step iteration of the universal Turing machine. Systems that are equivalent in power to universal Turing machines (i.e., can compute anything that a universal Turing machine can) are more generally called *universal computers*, or are said to be *capable of universal computation* or to *support universal computation*.

The Game of Life

Von Neumann's cellular automaton rule was rather complicated; a much simpler, two-state cellular automaton also capable of universal computation was invented in 1970 by the mathematician John Conway. He called his invention the "Game of Life." I'm not sure where the "game" part comes in, but the "life" part comes from the way in which Conway phrased the rule. Denoting **on** cells as *alive* and **off** cells as *dead*, Conway defined the rule in terms of four life processes: *birth*, a dead cell with exactly three live neighbors becomes alive at the next time step; *survival*, a live cell with exactly two or three live neighbors stays alive; *loneliness*, a live cell with fewer than two neighbors dies and a dead cell with fewer than three neighbors stays dead;

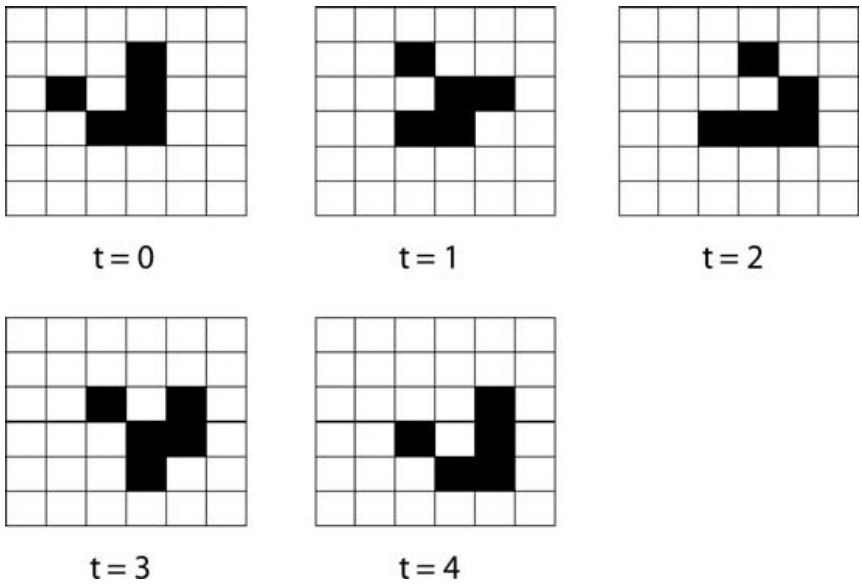

FIGURE 10.3. Close-up picture of a glider in the Game of Life. After four time steps, the original glider configuration has moved in the southeast direction.

and *overcrowding*, a live or dead cell with more than three live neighbors dies or stays dead.

Conway came up with this cellular automaton rule when looking for a rule that would create *interesting* (or perhaps life-like) behavior. Indeed the Game of Life has plenty of interesting behavior and there is a whole community of Life aficionados whose main hobby is to discover initial patterns that will create such behavior.

One simple pattern with interesting behavior is the *glider*, illustrated in figure 10.3. Here, instead of lightbulbs, I simply represent **on** (live) states by black squares and **off** (dead) states by white squares. The figure shows a glider "moving" in a southeast direction from its initial position. Of course, it's not the cells that move; they are all fixed in place. The moving entities are **on** states that form a coherent, persisting shape. Since, as I mentioned earlier, the cellular automaton's boundaries wrap around to create a donut shape, the glider will continue moving around and around the lattice forever.

Other intricate patterns that have been discovered by enthusiasts include the *spaceship*, a fancier type of glider, and the *glider gun*, which continually shoots out new gliders. Conway showed how to simulate Turing machines in Life by having the changing **on**/**off** patterns of states simulate a tape head that reads and writes on a simulated tape.

John Conway also sketched a proof (later refined by others) that Life could simulate a universal computer. This means that given an initial configuration

of **on** and **off** states that encodes a program and the input data for that program, Life will run that program on that data, producing a pattern that represents the program's output.

Conway's proof consisted of showing how glider guns, gliders, and other structures could be assembled so as to carry out the *logical operations* **and**, **or**, and **not**. It has long been known that any machine that has the capacity to put together all possible combinations of these logic operations is capable of universal computation. Conway's proof demonstrated that, in principle, all such combinations of logical operations are possible in the Game of Life.

It's fascinating to see that something as simple to define as the Life cellular automaton can, in principle, run any program that can be run on a standard computer. However, in practice, any nontrivial computation will require a large collection of logic operations, interacting in specific ways, and it is very difficult, if not impossible, to design initial configurations that will achieve nontrivial computations. And even if it were possible, the ensuing computation would be achingly slow, not to mention wasteful, since the huge parallel, non-von-Neumann-style computing resources of the cellular automaton would be used to simulate, in a very slow manner, a traditional von-Neumann-style computer.

For these reasons, people don't use Life (or other "universal" cellular automata) to perform real-world computations or even to model natural systems. What we really want from cellular automata is to harness their parallelism and ability to form complex patterns in order to achieve computations in a nontraditional way. The first step is to characterize the kinds of patterns that cellular automata can form.

The Four Classes

In the early 1980s, Stephen Wolfram, a physicist working at the Institute for Advanced Study in Princeton, became fascinated with cellular automata and the patterns they make. Wolfram is one of those legendary former child prodigies whom people like to tell stories about. Born in London in 1959, Wolfram published his first physics paper at age 15. Two years later, in the summer after his first year at Oxford, a time when typical college students get jobs as lifeguards or hitchhike around Europe with a backpack, Wolfram wrote a paper in the field of "quantum chromodynamics" that caught the attention of Nobel prize–winning physicist Murray Gell-Mann, who invited Wolfram to join his group at Caltech (California Institute of Technology). Two years later, at age twenty, Wolfram received a Ph.D. in theoretical physics.

Stephen Wolfram. (Photograph courtesy of Wolfram Research, Inc.)

(Most students take at least five years to get a Ph.D., *after* graduating from college.) He then joined the Caltech faculty, and was soon awarded one of the first MacArthur Foundation "genius" grants. A couple of years later, he was invited to join the faculty at the Institute for Advanced Study in Princeton.

Whew. With all that fame, funding, and the freedom to do whatever he wanted, Wolfram chose to study the dynamics of cellular automata.

In the spirit of good theoretical physics, Wolfram set out to study the behavior of cellular automata in the simplest form possible—using one-dimensional, two-state cellular automata in which each cell is connected only to its two nearest neighbors (figure 10.4a). Wolfram termed these "elementary cellular automata." He figured that if he couldn't understand what was going on in these seemingly ultra-simple systems, there was no chance of understanding more complex (e.g., two-dimensional or multistate) cellular automata.

Figure 10.4 illustrates one particular elementary cellular automaton rule. Figure 10.4a shows the lattice—now just a line of cells, each connected to its nearest neighbor on either side. As before, a cell is represented by a square—black for **on**, and white for **off**. The edges of the lattice wrap around to make a circle. Figure 10.4b shows the rule that each cell follows: for each of the eight possible state configurations for a three-cell neighborhood, the update state for the center cell is given. For example, whenever a three-cell neighborhood consists of all **off** states, the center cell should stay **off** at the next time step. Likewise, whenever a three-cell neighborhood has the configuration **off-off-on**, the center cell should change its state to **on** at the next time step. Note that

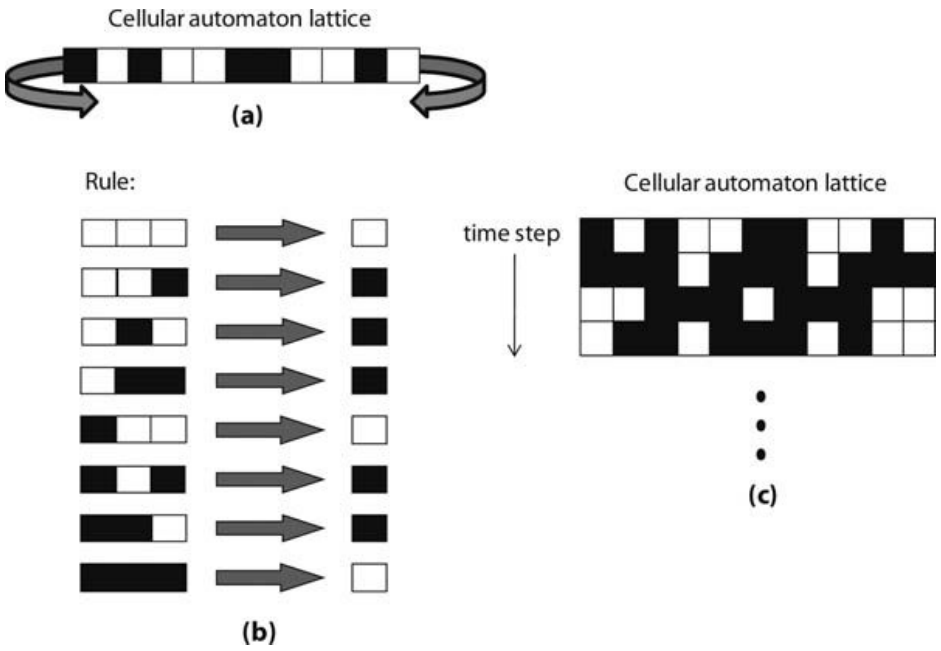

FIGURE 10.4. (a) An illustration of a one-dimensional lattice whose ends wrap around in a circle; (b) A particular elementary cellular automaton rule (Rule 110) expressed as a list of three-cell configurations and corresponding update states for the configuration's center cell; (c) A space-time diagram, showing four successive configurations of the cellular automaton.

the term *rule* refers to the entire list of configurations and update states, not to the individual lines in the list. Figure 10.4c shows a space-time diagram for this cellular automaton. The top row of cells is the one-dimensional lattice set up with a particular initial configuration of **on** and **off** states. Each successive row going down is the updated lattice at the next time step. Such plots are called space-time diagrams because they track the spatial configuration of a cellular automaton over a number of time steps.

Since there are only eight possible configurations of states for a three-cell neighborhood (cf. figure 10.4b) and only two possible ways to fill in the update state (**on** or **off**) for each of these eight configurations, there are only 256 (2^8) possible rules for elementary cellular automata. By the 1980s, computers were powerful enough for Wolfram to thoroughly investigate every single one of them by looking at their behavior starting from many different initial lattice configurations.

Wolfram assigned an identifying number to each elementary cellular automaton rule as illustrated in figure 10.5. He called the **on** state "1" and the **off** state "0," and wrote the rule's update states as a string of 1s and 0s, starting with the update state for the all **on** neighborhood and ending with the update state for the all **off** neighborhood. As shown, the rule given in

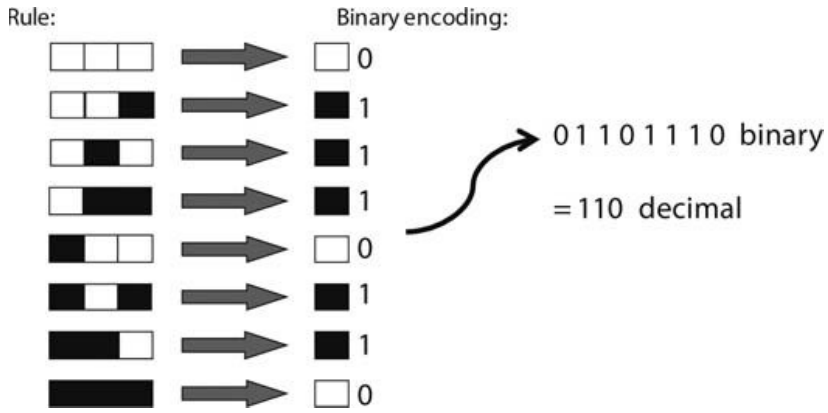

FIGURE 10.5. An illustration of the numbering system for elementary cellular automata used by Stephen Wolfram.

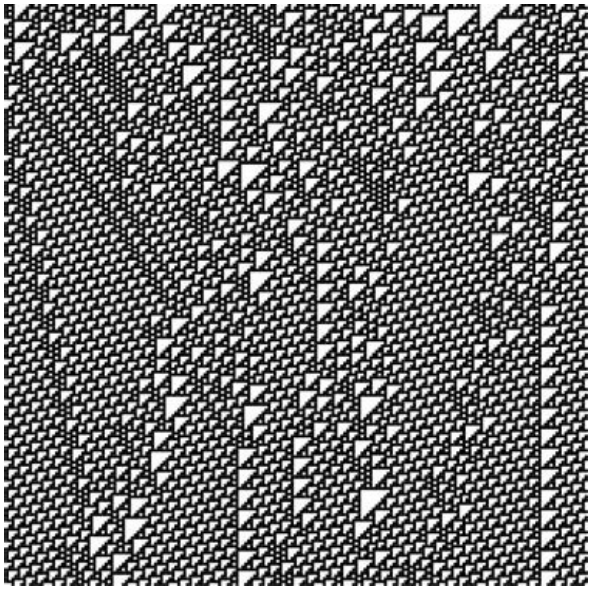

FIGURE 10.6. Rule 110 space-time diagram. The one-dimensional cellular automaton lattice has 200 cells, which are shown starting with a random initial configuration of states, and updating over 200 time steps.

figure 10.4 is written as 0 1 1 0 1 1 1 0. Wolfram then interpreted this string as a number written in binary (i.e., base 2). The string 0 1 1 0 1 1 1 0 in binary is equal to the number 110 in decimal. This rule is thus called "Rule 110." As another example, the rule with update states 0 0 0 1 1 1 1 0 is "Rule 30." (See the notes for a review on how to convert base 2 numbers to decimal.)

Wolfram and his colleagues developed a special programming language, called Mathematica, designed in part to make it easy to simulate cellular automata. Using Mathematica, Wolfram programmed his computer to run elementary cellular automata and to display space-time diagrams that show their behavior. For example, figures 10.6 and 10.7 are plots like that given in figure 10.4, just on a larger scale. The top horizontal row of figure 10.6 is a random initial configuration of 200 black and white cells, and each successive row is the result of applying Rule 110 to each cell in the previous row, for

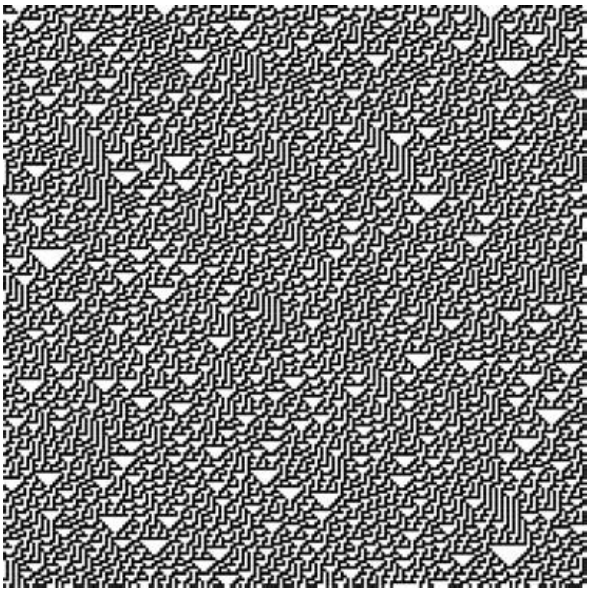

FIGURE 10.7. Rule 30 space-time diagram, with an initial configuration of random states.

200 time steps. The plot in figure 10.7 shows the pattern created by Rule 30, starting from a random initial configuration.

Looking at figures 10.6 and 10.7, perhaps you can sense why Wolfram got so excited about elementary cellular automata. How, exactly, did these complex patterns emerge from the very simple cellular automaton rules that created them?

Seeing such complexity emerge from simple rules was evidently an epiphany for Wolfram. He later said, "The Rule 30 automaton is the most surprising thing I've ever seen in science.... It took me several years to absorb how important this was. But in the end, I realized that this one picture contains the clue to what's perhaps the most long-standing mystery in all of science: where, in the end, the complexity of the natural world comes from." In fact, Wolfram was so impressed by Rule 30 that he patented its use as part of a pseudo-random number generator.

In Wolfram's exhaustive survey of all 256 elementary cellular automata, he viewed the behavior over time of each one starting from many different initial configurations. For each elementary cellular automaton and each initial configuration, he applied the cellular automaton rule to the lattice for a number of time steps—until the cellular automaton exhibited a stable type of behavior. He observed the behavior to fall into four classes:

> *Class 1:* Almost all initial configurations settle down to the same uniform final pattern. Rule 8 is an example of this class; for all initial configurations, all cells in the lattice quickly switch to the **off** state and stay that way.

Class 2: Almost all initial configurations settle down to either a uniform final pattern or a cycling between a few final patterns. Here, the specific final pattern or patterns depends on the initial configuration.

Class 3: Most initial configurations produce random-looking behavior, although triangles or other regular structures are present. Rule 30 (figure 10.7) is an example of this class.

Class 4: The most interesting class. As described by Wolfram: "class 4 involves a mixture of order and randomness: localized structures are produced which on their own are fairly simple, but these structures move around and interact with each other in very complicated ways." Rule 110 (figure 10.6) is an example of this class.

Wolfram speculated that, because of this complexity of patterns and interactions, all class 4 rules are capable of universal computation. However, in general it is hard to prove that a particular cellular automaton, Turing machine, or any other device is universal. Turing's proof that there exists a universal Turing machine was a triumph, as was von Neumann's proof that his self-replicating automaton was also a universal computer. Since then several researchers have proved that simple cellular automata (such as the Game of Life) are universal. In the 1990s, Matthew Cook, one of Wolfram's research assistants, finally proved that Rule 110 was indeed universal, and is perhaps the simplest known example of a universal computer.

Wolfram's "New Kind of Science"

I first heard about Cook's result in 1998 when he spoke at a workshop at the Santa Fe Institute. My own reaction, like that of many of my colleagues, was "Very cool! Very ingenious! But not of much practical or scientific significance." Like the Game of Life, Rule 110 is an example of a very simple deterministic system that can create unpredictable complex behavior. But in practice it would be very difficult to design an initial configuration that would result in a desired nontrivial computation. Moreover, Rule 110 would be an even slower computer than the Game of Life.

Wolfram's view of the result is very different. In his 2002 book, *A New Kind of Science*, Wolfram interprets the universality of Rule 110 as strong evidence for "a new law of nature," namely, his Principle of Computational Equivalence. Wolfram's proposed principle consists of four parts:

1. The proper way to think about processes in nature is that they are *computing*.