PART II | Life and Evolution in Computers

Nature proceeds little by little from things lifeless to animal life in such a way that it is impossible to determine the exact line of demarcation.

—Aristotle, *History of Animals*

[*W*]*e all know intuitively what life is. It is edible, lovable, or lethal.*

—James Lovelock, *The Ages of Gaia*

CHAPTER 8 | Self-Reproducing Computer Programs

What Is Life?

CHAPTER 5 DESCRIBED SOME OF THE HISTORY of ideas about how life has evolved. But a couple of things were missing, such as, how did life originate in the first place? And what exactly constitutes being alive? As you can imagine, both questions are highly contentious in the scientific world, and no one yet has definitive answers. Although I do not address the first question here, there has been some fascinating research on it in the complex systems community.

The second question—what is life, exactly?—has been on the minds of people probably for as long as "people" have existed. There is still no good agreement among either scientists or the general public on the definition of life. Questions such as "When does life begin?" or "What form could life take on other planets?" are still the subject of lively, and sometimes vitriolic, debate.

The idea of creating artificial life is also very old, going back at least two millennia to legends of the Golem and of Ovid's Pygmalion, continuing in the nineteenth-century story of Frankenstein's monster, all the way to the present era of movies such as *Blade Runner* and *The Matrix*, and computer games such as "Sim Life."

These works of fiction both presage and celebrate a new, technological version of the "What is life?" question: Is it possible for computers or robots to be considered "alive"? This question links the previously separate topics of computation and of life and evolution.

You can ask ten biologists what are the ten key requisites for life and you'll get a different list each time. 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 and they don't care how successful they are. (For example, in a lecture I attended by a prominent psychologist, the speaker, discussing the success of computer chess programs, asserted that "Deep Blue may have beat Kasparov, but it didn't get any joy out of it.")

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 many people, all of them have been claimed to be disproven in one way or another in the field of *artificial life,* whose purview is the simulation and "creation" of life on computers. In this chapter and the next I focus on those issues most closely related to Darwinism—self-reproduction and evolution.

Self-Reproduction in Computers

The self-reproduction argument is the most mathematical one: it states that self-reproduction in a computer would lead to an infinite regress.

Let's investigate this issue via the simplest version of the computer self-reproduction problem: write a computer program that prints out an exact copy of itself and nothing else.

I've written the following programs in a simple computer language that even nonprogrammers should be able to understand. (It's actually a pseudo

language, with a few commands that most real languages wouldn't have, but still plausible and thrown in to make things simpler.)

Here's a first 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")
```

The `print` command simply prints on the computer screen the characters between the first and last quotation marks, followed by a carriage return. Now I need to add an instruction to print out that second line:

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

Note that, since I want the program to print out an *exact* copy of itself, the second print command has to include the four spaces of indentation I put before the first print command, plus the two quotation marks, which are now themselves being quoted (the `print` command prints anything, including quotation marks, between the outermost quotation marks). Now I need another line to print out that third line:

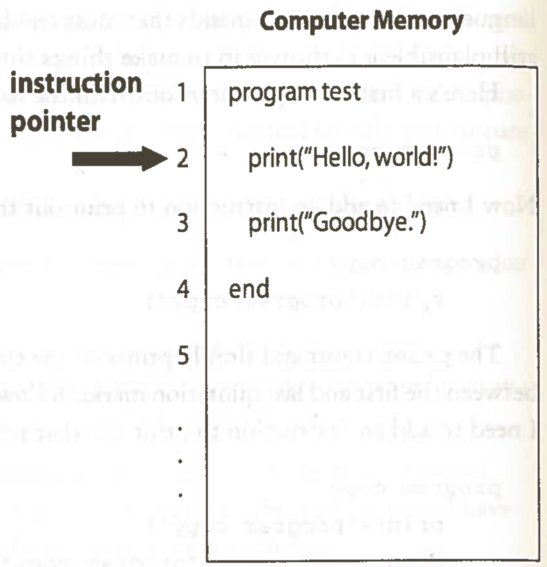
```
program copy
    print("program copy")
    print("    print("program copy")")
    print("    print("    print("program copy")")")
```

and so forth. By now you can probably see how this strategy, in which each command prints an exact copy of the command preceding it, leads to 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 have echos in the work of Kurt Gödel and Alan Turing, which I described in chapter 4. The solution also contains an essential means by which biological systems themselves get around the infinite regress. The solution was originally found, in the context of a more complicated problem, by the twentieth-century Hungarian 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 that communicates with a random

FIGURE 8.1. 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. The lines sometimes contain leading spaces, which are ignored when the instruction is executed.



access memory in which both programs and data can be stored. It remains the basic design of all standard computers today. Von Neumann 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 machine 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 more things about the programming language I will be using.

Consider the picture of computer memory given in figure 8.1. Computer memory, in our highly simplified case, consists of numbered locations or "addresses," here numbered 1–5 and beyond. 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.
```

To accomplish this, the computer has an "instruction pointer"—a number also stored in memory, which always is equal to the memory location of the instruction currently being executed by the computer. The instruction pointer—let's call it `ip` for short—is initially set to the memory location

containing the program's first line. We say it "points to" that instruction. At each step in the computation the instruction pointed to by `ip` is carried out, and `ip` is increased by 1.

For example, in figure 8.1, the value of `ip` is 2, and we say that `ip` is pointing to the line `print("Hello, world!")`.

We call `ip` a *variable* since its value changes ("varies") as the computation is carried out.

We also can define another variable, `line[n]`, as 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 commands inside the two curly brackets get repeated until the loop condition (here `x = 4`) is true. The variable `x` is used as a counter—it starts off at zero and is increased by 1 each time the loop is performed. When it gets to 4, the loop stops.

Now we are ready for the self-copying program, which appears in its entirety in figure 8.2. 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.

Suppose this program has been loaded into computer memory as shown in figure 8.2, and suppose a user comes along and types `selfcopy` on the computer's command line. This signals the computer to start interpreting the program called `selfcopy`. The interpreter—part of the computer's

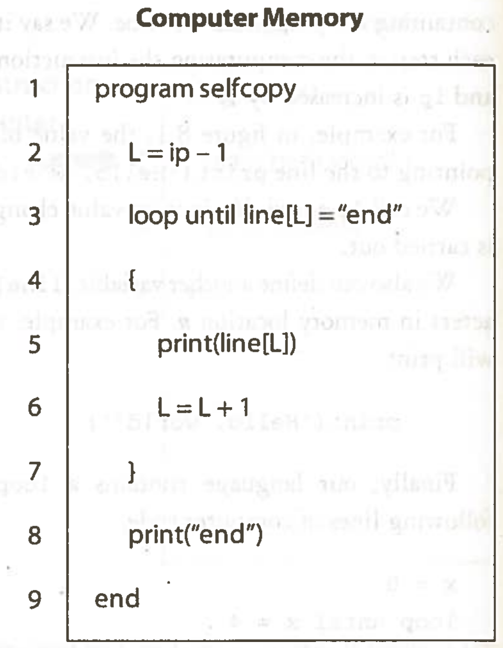


FIGURE 8.2. A self-copying program.

Computer Memory

```
1  program selfcopy
2    L = ip - 1
3    loop until line[L] = "end"
4    {
5        print(line[L])
6        L = L + 1
7    }
8    print("end")
9  end
```

operating system—sets the instruction pointer to 1, which points to the name of the program. The `ip` then moves down, line by line, executing each instruction.

In memory location 2 a variable `L` is set to `ip`−1. Recall that `ip` is the location of the instruction currently being executed. So when line 2 is executed, `ip` is set to 2 and `L` is set to $2 - 1 = 1$. (Note that `L` will now stay equal to 1 until it is reset, even though `ip` changes as each instruction is executed.)

Next, a loop is entered, which will be iterated until `line[L]` is equal to the character string `end`. Remember that `line[L]` is equal to the string located in memory location `L`. Right now, `L` is set to 1, so `line[L]` is equal to the string `program selfcopy`. This is not equal to the string `end`, so the loop is continued. In the loop, `line[L]` is printed and `L` is incremented. First, with `L = 1`, `program selfcopy` is printed; then `L` is set to 2.

Now, `line[L]` is the second line of the program, namely `L = ip - 1`. Again, this string 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 5: when line 5 is being executed with `L` = 5, the instruction `print(line[L])` prints itself out. When `L` = 9 and `line[L]` is equal to `end`, the loop ends. At this point, lines 1–8 have been printed. The instruction pointer moves

to line 8 (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 is 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 my first attempt at a self-copying program.

The Deeper Meaning of the Self-Reproducing Computer Program

The dual use of information is also at the heart of Gödel's paradox, embodied by his self-referential sentence "This statement is not provable."

This is a bit tricky to understand. First, let's note that this sentence, like any other English sentence, can be looked at in two ways: (1) as the literal string of letters, spaces, and punctuation contained in the sentence, and (2) as the *meaning* of that literal string, as interpreted by an English speaker.

To be very clear, let's call the literal string of characters S. That is, S = "This statement is not provable." We can now state facts about S: for example, it contains twenty-six letters, four spaces, and one period.

Let's call the meaning of the sentence M. We can rewrite M as follows: "Statement S is not provable." In a way, you can think of M as a "command" and of S as the "data" for that command. The weird (and wonderful) thing is that the data S is the same thing as the command M. The chief reason Gödel was able to translate his English sentence into a paradox in mathematics was that he was able to phrase M as a mathematical statement and S as a number that encoded the string of characters of that mathematical statement.

This is all very tricky. This kind of distinction between literal strings of characters and the meaning of those strings, and the paradoxes that self-reference can produce, are discussed in a detailed and very entertaining way in Douglas Hofstadter's book *Gödel, Escher, Bach: an Eternal Golden Braid*.

Similarly, this kind of dual use of information is key to Turing's proof of the undecidability of the Halting problem. Remember H and H' from chapter 4? Do you recall how H' ran on its own code? That is, just like our self-reproducing computer program above, H' was used in two ways: as an interpreted program and as the input for that program.

Self-Replication in DNA

At this point you may be groaning that we're back in the abstract realm of logical headaches. But give me a minute to bring us back to reality. The really amazing thing is that this dual use of information is key to the way DNA replicates itself. As I described in chapter 6, DNA is made up of strings of nucleotides. Certain substrings (*genes*) encode amino acids making up proteins, including the enzymes (special kinds of proteins) that effect the splitting of the double helix and the copying of each strand via messenger RNA, transfer RNA, ribosomes, et cetera. In a very crude analogy, the DNA strings encoding the enzymes that perform the 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.

However, you may have noticed something I have so far swept under the rug. There is a major difference between my self-copying program and DNA self-reproduction. The self-copying program required an interpreter to execute it: an instruction pointer to move 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 such as ip and L, actually print strings of characters, and so on). The interpreter is completely external to the program itself.

In contrast, 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. That is, DNA not only contains the code for its self-replicating "program" (i.e., the enzymes that perform the splitting and copying of DNA) but also it encodes its own interpreter (the cellular machinery that translates DNA into those very enzymes).

Von Neumann's Self-Reproducing Automaton

Von Neumann's original self-reproducing automaton (described mathematically but not actually built by von Neumann) similarly contained not only a self-copying program but also the machinery needed for its own interpretation. Thus it was truly a *self*-reproducing machine. This explains why von Neumann's construction was considerably more complicated than my simple self-copying program. That it was formulated in the 1950s, before the details of biological self-reproduction were well understood, is testament

John von Neumann, 1903–1957
(AIP Emilio Segre Visual Archives)

to von Neumann's insight. Von Neumann's design of this automaton and mathematical proof of its correctness were mostly completed when he died in 1957, at the age of 53, of cancer possibly caused by his exposure to radiation during his work on the atomic bomb. The proof was completed by von Neumann's colleague, Arthur Burks. The complete work was eventually published in 1966 as a book, *Theory of Self-Reproducing Automata*, edited by Burks.

Von Neumann's design for a self-reproducing automaton was one of the first real advances in the science of artificial life, demonstrating that self-reproduction by machine was indeed possible in principle, and providing a "logic" of self-reproduction that turned out to have some remarkable similarities to the one used by living systems.

Von Neumann recognized that these results could have profound consequences. He worried about public perception of the possibilities of self-reproducing machines, and said that he did not want any mention of the "reproductive potentialities of the machines of the future" made to the mass media. It took a while, but the mass media eventually caught up. In 1999, computer scientists Ray Kurzweil and Hans Moravec celebrated the possibility of super-intelligent self-reproducing robots, which they believe will be built in the near future, in their respective nonfiction (but rather far-fetched) books *The Age of Spiritual Machines* and *Robot*. In 2000 some of the possible perils of self-reproducing nano-machines were decried by Bill Joy, one of the founders of Sun Microsystems, in a now famous article in *Wired* magazine called "Why the Future Doesn't Need Us." So far none of these predictions has come to pass. However, complex self-reproducing machines may soon be a reality: some simple self-reproducing robots have already been constructed by roboticist Hod Lipson and his colleagues at Cornell University.

John von Neumann

It is worth saying a few words about von Neumann himself, one of the most important and interesting figures in science and mathematics in the twentieth century. He is someone you should know about if you don't already. Von Neumann was, by anyone's measure, a true genius. During his relatively short life he made fundamental contributions to at least six fields: mathematics, physics, computer science, economics, biology, and neuroscience. He is the type of genius whom people tell stories about, shaking their heads and wondering whether someone that smart could really be a member of the human species. I like these stories so much, I want to retell a few of them here.

Unlike Einstein and Darwin, whose genius took a while to develop, Hungarian born "Johnny" von Neumann was a child prodigy. He supposedly could divide eight-digit numbers in his head at the age of six. (It evidently took him a while to notice that not everyone could do this; as reported in one of his biographies, "When his mother once stared rather aimlessly in front of her, six-year-old Johnny asked: 'What are you calculating?' ") At the same age he also could converse with his father in ancient Greek.

At the age of eighteen von Neumann went to university, first in Budapest, then in Germany and Switzerland. He first took the "practical" course of studying chemical engineering but couldn't be kept away from mathematics. He received a doctorate in math at the age of twenty-three, after doing fundamental work in both mathematical logic and quantum mechanics. His work was so good that just five years later he was given the best academic job in the world—a professorship (with Einstein and Gödel) at the newly formed Institute for Advanced Study (IAS) in Princeton.

The institute didn't go wrong in their bet on von Neumann. During the next ten years, von Neumann went on to invent the field of game theory (producing what has been called "the greatest paper on mathematical economics ever written"), design the conceptual framework of one of the first

programmable computers (the EDVAC, for which he wrote what has been called "the most important document ever written about computing and computers"), and make central contributions to the development of the first atomic and hydrogen bombs. This was all before his work on self-reproducing automata and his exploration of the relationships between the logic of computers and the workings of the brain. Von Neumann also was active in politics (his positions were very conservative, driven by strong anti-communist views) and eventually became a member of the Atomic Energy Commission, which advised the U.S. president on nuclear weapons policy.

Von Neumann was part of what has been called the "Hungarian phenomenon," a group of several Hungarians of similar age who went on to become world-famous scientists. This group also included Leo Szilard, whom we heard about in chapter 3, the physicists Eugene Wigner, Edward Teller, and Denis Gabor, and the mathematicians Paul Erdös, John Kemeny, and Peter Lax. Many people have speculated on the causes of this improbable cluster of incredible talent. But as related by von Neumann biographer Norman MacRae, "Five of Hungary's six Nobel Prize winners were Jews born between 1875 and 1905, and one was asked why Hungary in his generation had brought forth so many geniuses. Nobel laureate Wigner replied that he did not understand the question. Hungary in that time had produced only one genius, Johnny von Neumann."

Von Neumann was in many ways ahead of his time. His goal was, like Turing's, to develop a general theory of information processing that would encompass both biology and technology. His work on self-reproducing automata was part of this program. Von Neumann also was closely linked to the so-called cybernetics community—an interdisciplinary group of scientists and engineers seeking commonalities among complex, adaptive systems in both natural and artificial realms. What we now call "complex systems" can trace its ancestry to cybernetics and the related field of systems science. I explore these connections further in the final chapter.

Von Neumann's interest in computation was not always well received at the elite Institute for Advanced Study. After completing his work on the EDVAC, von Neumann brought several computing experts to the IAS to work with him on designing and building an improved successor to EDVAC. This system was called the "IAS computer"; its design was a basis for the early computers built by IBM. Some of the "pure" scientists and mathematicians at IAS were uncomfortable with so practical a project taking place in their ivory tower, and perhaps even more uncomfortable with von Neumann's first application of this computer, namely weather prediction, for which he brought a team of meteorologists to the IAS. Some of the purists didn't think this kind of activity fit in with the institute's theoretical charter. As IAS physicist Freeman Dyson put it, "The [IAS] School of Mathematics has a permanent establishment which is divided into three groups, one consisting of pure mathematics, one consisting of theoretical physicists, and one consisting of Professor von Neumann." After von Neumann's death, the IAS computer project was shut down, and the IAS faculty passed a motion "to have no experimental science, no laboratories of any kind at the Institute." Freeman Dyson described this as, "The snobs took revenge."

CHAPTER 9 | Genetic Algorithms

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 before he was 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*. The most widely known of these efforts today is the work on *genetic algorithms* done by John Holland and his students and colleagues at the University of Michigan.

John Holland is, in some sense, the academic grandchild of John von Neumann. Holland's own Ph.D. advisor was Arthur Burks, the philosopher, logician, and computer engineer who assisted von Neumann on the EDVAC computer and who completed von Neumann's unfinished work on self-reproducing automata. After his work on the EDVAC, Burks obtained a faculty position in philosophy at the University of Michigan and started the Logic of Computers group, a loose-knit collection of faculty and students who were interested in the foundations of computers and of information processing in general. Holland joined the University of Michigan as a Ph.D. student, starting in mathematics and later switching to a brand-new program called "communication sciences" (later "computer and communication sciences"), which was arguably the first real computer science department in the world. A few years later, Holland became the program's first Ph.D.

John Holland. (Photograph copyright © by the Santa Fe Institute. Reprinted by permission.)

recipient, giving him the distinction of having received the world's first Ph.D. in computer science. He was quickly hired as a professor in that same department.

Holland got hooked on Darwinian evolution when he read Ronald Fisher's famous book, *The Genetical Theory of Natural Selection*. Like Fisher (and Darwin), Holland was struck by analogies between evolution and animal breeding. But he looked at the analogy from his own computer science perspective: "That's where genetic algorithms came from. I began to wonder if you could breed programs the way people would say, breed good horses and breed good corn."

Holland's major interest was in the phenomenon of adaptation—how living systems evolve or otherwise change in response to other organisms or to a changing environment, and how computer systems might use similar principles to be adaptive as well. His 1975 book, *Adaptation in Natural and Artificial Systems*, laid out a set of general principles for adaptation, including a proposal for genetic algorithms.

My own first exposure to genetic algorithms was in graduate school at Michigan, when I took a class taught by Holland that was based on his book. I was instantly enthralled by the idea of "evolving" computer programs. (Like Thomas Huxley, my reaction was, "How extremely stupid not to have thought of that!")

A Recipe for a Genetic Algorithm

The term *algorithm* is used these days to mean what Turing meant by *definite procedure* and what cooks mean by *recipe*: a series of steps by which an input is transformed to an output.

In a *genetic* algorithm (GA), the desired output is a solution to some problem. Say, for example, that you are assigned to write a computer program that controls a robot janitor that picks up trash around your office building. You decide that this assignment will take up too much of your time, so you want to employ a genetic algorithm to evolve the program for you. Thus, the desired output from the GA is a robot-janitor control program that allows the robot to do a good job of collecting trash.

The input to the GA has two parts: a *population* of candidate programs, and a *fitness function* that takes a candidate program and assigns to it a *fitness* value that measures how well that program works on the desired task.

Candidate programs can be represented as strings of bits, numbers, or symbols. Later in this chapter I give an example of representing a robot-control program as a string of numbers.

In the case of the robot janitor, the fitness of a candidate program could be defined as the square footage of the building that is covered by the robot, when controlled by that program, in a set amount of time. The more the better.

Here is the recipe for the GA.

Repeat the following steps for some number of *generations*:

1. Generate an initial population of candidate solutions. The simplest way to create the initial population is just to generate a bunch of random programs (strings), called "individuals."
2. Calculate the fitness of each individual in the current population.
3. Select some number of the individuals with highest fitness to be the *parents* of the next generation.
4. Pair up the selected parents. Each pair produces offspring by recombining parts of the parents, with some chance of random mutations, and the offspring enter the new population. The selected parents continue creating offspring until the new population is full (i.e., has the same number of individuals as the initial population). The new population now becomes the current population.
5. Go to step 2.

Genetic Algorithms in the Real World

The GA described above is simple indeed, but versions of it have been used to solve hard problems in many scientific and engineering areas, as well as in art, architecture, and music.

Just to give you a flavor of these problems: GAs have been used at the General Electric Company for automating parts of aircraft design, Los Alamos National Lab for analyzing satellite images, the John Deere company for automating assembly line scheduling, and Texas Instruments for computer chip design. GAs were used for generating realistic computer-animated horses in the 2003 movie *The Lord of the Rings: The Return of the King*, and realistic computer-animated stunt doubles for actors in the movie *Troy*. A number of pharmaceutical companies are using GAs to aid in the discovery of new drugs. GAs have been used by several financial organizations for various tasks: detecting fraudulent trades (London Stock Exchange), analysis of credit card data (Capital One), and forecasting financial markets and portfolio optimization (First Quadrant). In the 1990s, collections of artwork created by an interactive genetic algorithm were exhibited at several museums, including the Georges Pompidou Center in Paris. These examples are just a small sampling of ways in which GAs are being used.

Evolving Robby, the Soda-Can-Collecting Robot

To introduce you in more detail to the main ideas of GAs, I take you through a simple extended example. I have a robot named "Robby" who lives in a (computer simulated, but messy) two-dimensional world that is strewn with empty soda cans. I am going to use a genetic algorithm to evolve a "brain" (that is, a control strategy) for Robby.

Robby's job is to clean up his world by collecting the empty soda cans. Robby's world, illustrated in figure 9.1, consists of 100 squares (sites) laid out in a 10 × 10 grid. You can see Robby in site 0,0. Let's imagine that there is a wall around the boundary of the entire grid. Various sites have been littered with soda cans (but with no more than one can per site).

Robby isn't very intelligent, and his eyesight isn't that great. From wherever he is, he can see the contents of one adjacent site in the north, south, east, and west directions, as well as the contents of the site he occupies. A site can be empty, contain a can, or be a wall. For example, in figure 9.1, Robby, at site 0,0, sees that his current site is empty (i.e., contains no soda cans), the "sites" to the north and west are walls, the site to the south is empty, and the site to the east contains a can.

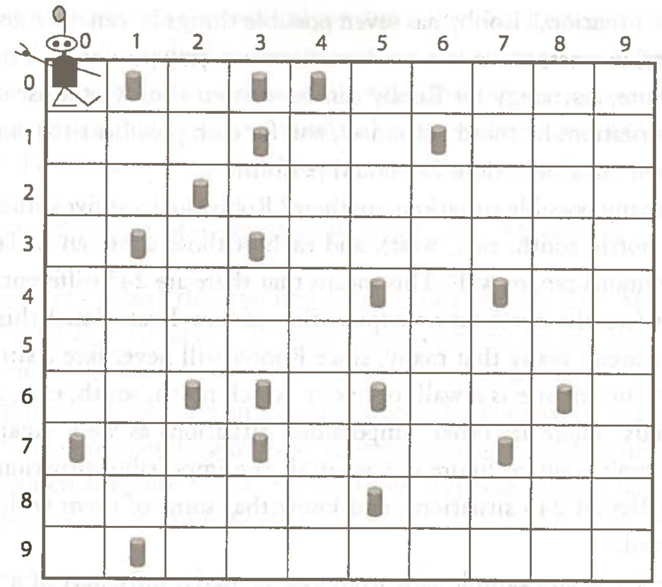


FIGURE 9.1. Robby's world. A 10 x 10 array, strewn with soda cans.

For each cleaning session, Robby can perform exactly 200 actions. Each action consists of one of the following seven choices: move to the north, move to the south, move to the east, move to the west, choose a random direction to move in, stay put, or bend down to pick up a can. Each action may generate a reward or a punishment. If Robby is in the same site as a can and picks it up, he gets a reward of ten points. However, if he bends down to pick up a can in a site where there is no can, he is fined one point. If he crashes into a wall, he is fined five points and bounces back into the current site.

Clearly, Robby's reward is maximized when he picks up as many cans as possible, without crashing into any walls or bending down to pick up a can if no can is there.

Since this is a simple problem, it would probably be pretty easy for a human to figure out a good strategy for Robby to follow. However, the point of genetic algorithms is that humans, being intrinsically lazy, don't have to figure out anything; we just let computer evolution figure it out for us. Let's use a genetic algorithm to evolve a good strategy for Robby.

The first step is to figure out exactly what we are evolving; that is, what exactly constitutes a *strategy*? In general, a strategy is a set of rules that gives, for any situation, the action you should take in that situation. For Robby, a "situation" is simply what he can see: the contents of his current site plus the contents of the north, south, east, and west sites. For the question "what to do in each situation," Robby has seven possible things he can do: move north, south, east, or west; move in a random direction; stay put; or pick up a can.

Therefore, a strategy for Robby can be written simply as a list of all the possible situations he could encounter, and for each possible situation, which of the seven possible actions he should perform.

How many possible situations are there? Robby looks at five different sites (current, north, south, east, west), and each of those sites can be labeled as empty, contains can, or wall. This means that there are 243 different possible situations (see the notes for an explanation of how I calculated this). Actually, there aren't really that many, since Robby will never face a situation in which his current site is a wall, or one in which north, south, east, and west are all walls. There are other "impossible" situations as well. Again, being lazy, we don't want to figure out what all the impossible situations are, so we'll just list all 243 situations, and know that some of them will never be encountered.

Table 9.1 is an example of a strategy—actually, only part of a strategy, since an entire strategy would be too long to list here.

Robby's situation in figure 9.1 is

North	*South*	*East*	*West*	*Current Site*
Wall	Empty	Can	Wall	Empty

To decide what to do next, Robby simply looks up this situation in his strategy table, and finds that the corresponding action is MoveWest. So he moves west. And crashes into a wall.

I never said this was a *good* strategy. Finding a good strategy isn't our job; it's the job of the genetic algorithm.

TABLE 9-1

Situation					Action
North	*South*	*East*	*West*	*Current Site*	
Empty	Empty	Empty	Empty	Empty	MoveNorth
Empty	Empty	Empty	Empty	Can	MoveEast
Empty	Empty	Empty	Empty	Wall	MoveRandom
Empty	Empty	Empty	Can	Empty	PickUpCan
⋮	⋮	⋮	⋮	⋮	⋮
Wall	Empty	Can	Wall	Empty	MoveWest
⋮	⋮	⋮	⋮	⋮	⋮
Wall	Wall	Wall	Wall	Wall	StayPut

I wrote the code for a genetic algorithm to evolve strategies for Robby. In my GA, each individual in the population is a strategy—a listing of the actions that correspond to each possible situation. That is, given a strategy such as the one in table 9.1, an individual to be evolved by the GA is just a listing of the 243 actions in the rightmost column, in the order given:

MoveNorth MoveEast MoveRandom PickUpCan ... MoveWest ... StayPut

The GA remembers that the first action in the string (here MoveNorth) goes with the first situation ("Empty Empty Empty Empty Empty"), the second action (here MoveEast) goes with the second situation ("Empty Empty Empty Empty Can"), and so on. In other words, I don't have to explicitly list the situations corresponding to these actions; instead the GA remembers the order in which they are listed. For example, suppose Robby happened to observe that he was in the following situation:

North	*South*	*East*	*West*	*Current Site*
Empty	Empty	Empty	Empty	Can

I build into the GA the knowledge that this is situation number 2. It would look at the strategy table and see that the action in position 2 is MoveEast. Robby moves east, and then observes his next situation; the GA again looks up the corresponding action in the table, and so forth.

My GA is written in the programming language C. I won't include the actual program here, but this is how it works.

1. **Generate the initial population.** The GA starts with an initial population of 200 *random* individuals (strategies).

   A random population is illustrated in figure 9.2. Each individual strategy is a list of 243 "genes." Each gene is a number between 0 and 6, which stands for an action (0 = *MoveNorth*, 1 = *MoveSouth*, 2 = *MoveEast*, 3 = *MoveWest*, 4 = *StayPut*, 5 = *PickUp*, and 6 = *RandomMove*). In the initial population, these numbers are filled in at random. For this (and all other probabilistic or random choices), the GA uses a pseudo-random-number generator.

   Repeat the following for 1,000 generations:
2. **Calculate the *fitness* of each individual in the population.** In my program, the fitness of a strategy is determined by seeing how well the strategy lets Robby do on 100 different cleaning sessions. A cleaning session consists of putting Robby at site 0, 0, and throwing down a bunch of cans at random (each site can contain at most one can; the

```
Individual 1:
23300323421630343530546006102562515114162260435654334066511514
15650220640642051006643216161521652022364433363346013326503000
40622050243165006111305146664232401245633345524126143441361020
150630642551654043264463156164510543665346310551646005164

Individual 2:
16411343121025360340361241431201104235462525304202044516433665
61035322153105131440622120614631432154610256523644422025340345
30502005620634026331002453416430151631210012214400664012665246
351650154123113132453304433212634555005314213064423311000

Individual 3:
20423344402411226132136452632464212206122122252660626144436125
32512664061335340153411110206164226653145522540234051155031302
22020065445125062206631426135532010000400031640130154160162006
134440626160505641421553133236021503355131253632642630551

        .
        .
        .

Individual 200:
34632525136001012225612106043301135205155320130656005322235043
32425064124255265534635345523053326612010632124554423440613654
30246240160663016464641103026540006334126150352262106063624260
550616616344255124354464110023463330440102533212142402251
```

FIGURE 9.2. Random initial population. Each individual consists of 243 numbers, each of which is between 0 and 6, and each of which encodes an action. The location of a number in a string indicates to which situation the action corresponds.

probability of a given site containing a can is 50%). Robby then follows the strategy for 200 actions in each session. The score of the strategy in each session is the number of reward points Robby accumulates minus the total fines he incurs. The strategy's *fitness* is its average score over 100 different cleaning sessions, each of which has a different configuration of cans.

3. **Apply evolution** to the current population of strategies to create a new population. That is, repeat the following until the new population has 200 individuals:

   (a) Choose two parent individuals from the current population probabilistically based on fitness. That is, the higher a strategy's fitness, the more chance it has to be chosen as a parent.

   (b) Mate the two parents to create two children. That is, randomly choose a position at which to split the two number strings; form one child by taking the numbers before that position from parent A and after that position from parent B, and vice versa to form the second child.

   (c) With a small probability, mutate numbers in each child. That is, with a small probability, choose one or more numbers and replace them each with a randomly generated number between 0 and 6.

   (d) Put the two children in the new population.

4. Once the new population has 200 individuals, return to step 2 with this new generation.

The magic is that, starting from a set of 200 random strategies, the genetic algorithm creates strategies that allow Robby to perform very well on his cleaning task.

The numbers I used for the population size (200), the number of generations (1,000), the number of actions Robby can take in a session (200), and the number of cleaning sessions to calculate fitness (100) were chosen by me, somewhat arbitrarily. Other numbers can be used and can also produce good strategies.

I'm sure you are now on the edge of your seat waiting to find out what happened when I ran this genetic algorithm. But first, I have to admit that before I ran it, I overcame my laziness and constructed my own "smart" strategy, so I could see how well the GA could do compared with me. My strategy for Robby is: "If there is a can in the current site, pick it up. Otherwise, if there is a can in one of the adjacent sites, move to that site. (If there are multiple adjacent sites with cans, I just specify the one to which Robby moves.) Otherwise, choose a random direction to move in."

This strategy actually isn't as smart as it could be; in fact, it can make Robby get stuck cycling around empty sites and never making it to some of the sites with cans.

I tested my strategy on 10,000 cleaning sessions, and found that its average (per-session) score was approximately 346. Given that at the beginning of each session, about 50%, or 50, of the sites contain cans, the maximum possible score for any strategy is approximately 500, so my strategy is not very close to optimal.

Can the GA do as well or better than this? I ran it to see. I took the highest-fitness individual in the final generation, and also tested it on 10,000 new and different cleaning sessions. Its average (per-session) score was approximately 483—that is, nearly optimal!

How Does the GA-Evolved Strategy Solve the Problem?

Now the question is, what is this strategy doing, and why does it do better than my strategy? Also, how did the GA evolve it?

Let's call my strategy *M* and the GA's strategy *G*. Below is each strategy's genome.

```
M: 656353656252353252656353656151353151252353252151353151656353656
   252353252656353656050353050252353252050353050151353151252353252
   151353151050353050252353252050353050656353656252353252656353656
   151353151252353252151353151656353656252353252656353454

G: 254355153256235251056355461151336154151034156110550150052030256
   256132252350325112052333054055231255051336154150665264150266506
   012264453605631520256431054354632404350334153250253251352352045
   150130156213436252353223135051260513356201524514343432
```

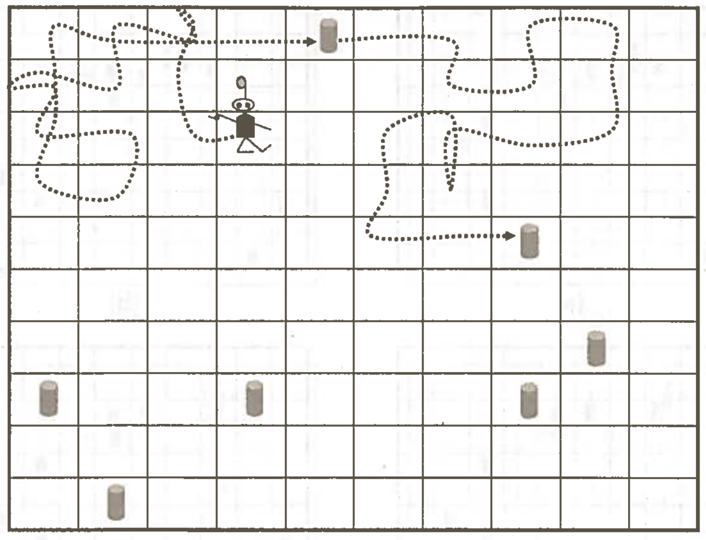
Staring at the genome of a strategy doesn't help us too much in understanding how that strategy works. We can see a few genes that make sense, such as the important situations in which Robby's current site contains a can, such as the second situation ("Empty Empty Empty Empty Can"), which has action 5 (*PickUp*) in both strategies. Such situations always have action 5 in *M*, but only most of the time in *G*. For example, I managed to determine that the following situation

North	*South*	*East*	*West*	*Current Site*
Empty	Can	Empty	Can	Can

has action 3 (*MoveWest*), which means Robby doesn't pick up the can in his current site. This seems like a bad idea, yet *G* does better than *M* overall! The key, it turns out, is not these isolated genes, but the way different genes interact, just as has been found in real genetics. And just as in real genetics, it's very difficult to figure out how these various interactions lead to the overall behavior or fitness.

It makes more sense to look at the actual behavior of each strategy—its *phenotype*—rather than its genome. I wrote a graphics program to display Robby's moves when using a given strategy, and spent some time watching the behavior of Robby when he used strategy *M* and when he used strategy *G*. Although the two strategies behave similarly in many situations, I found that strategy *G* employs two tricks that cause it to perform better than strategy *M*.

First, consider a situation in which Robby does not sense a can in his current site or in any of his neighboring sites. If Robby is following strategy



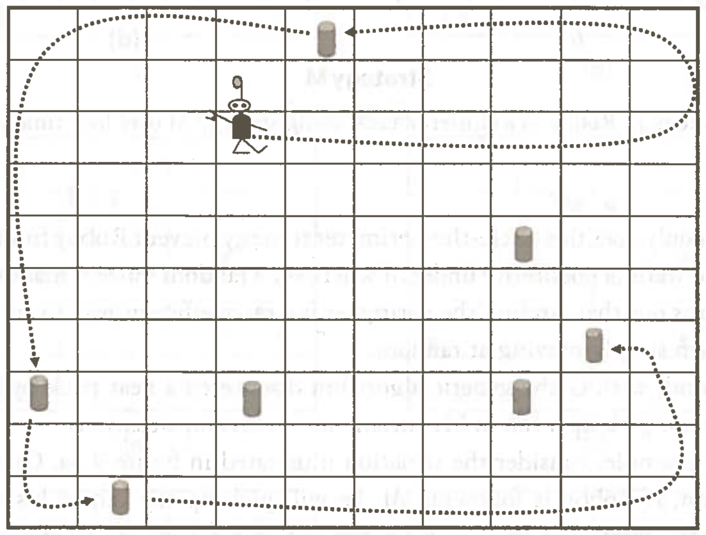


FIGURE 9.3. Robby in a "no-can" wilderness. The dotted lines show the paths he took in my simulation when he was following strategies M (top) and G (bottom).

M, he chooses a random move to make. However, if he is following strategy *G*, Robby moves to the east until he either finds a can or reaches a wall. He then moves north, and continues to circle the edge of the grid in a counterclockwise direction until a can is encountered or sensed. This is illustrated in figure 9.3 by the path Robby takes under each strategy (dotted line).

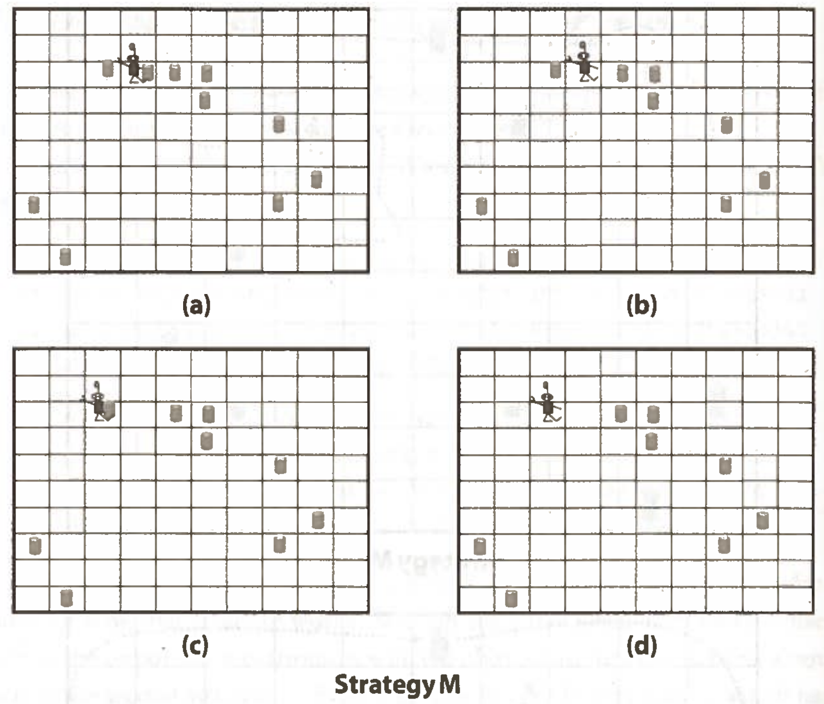


FIGURE 9.4. Robby in a cluster of cans, using strategy M over four time steps.

Not only does this circle-the-perimeter strategy prevent Robby from crashing into walls (a possibility under *M* whenever a random move is made), but it also turns out that circling the perimeter is a more efficient way to encounter cans than simply moving at random.

Second, with *G* the genetic algorithm discovered a neat trick by having Robby not pick up a can in his current site in certain situations.

For example, consider the situation illustrated in figure 9.4a. Given this situation, if Robby is following *M*, he will pick up the can in his current site, move west, and then pick up the can in his new site (pictures b–d). Because Robby can see only immediately adjacent sites, he now cannot see the remaining cluster of cans. He will have to move around at random until he encounters another can by chance.

In contrast, consider the same starting situation with *G*, illustrated in figure 9.5a. Robby doesn't pick up the can in his current site; instead he moves west (figure 9.5b). He then picks up the western-most can of the cluster (figure 9.5c). The can he didn't pick up on the last move acts as a marker so Robby can "remember" that there are cans on the other side of it. He goes on to pick up all of the remaining cans in the cluster (figure 9.5d–9.5k).

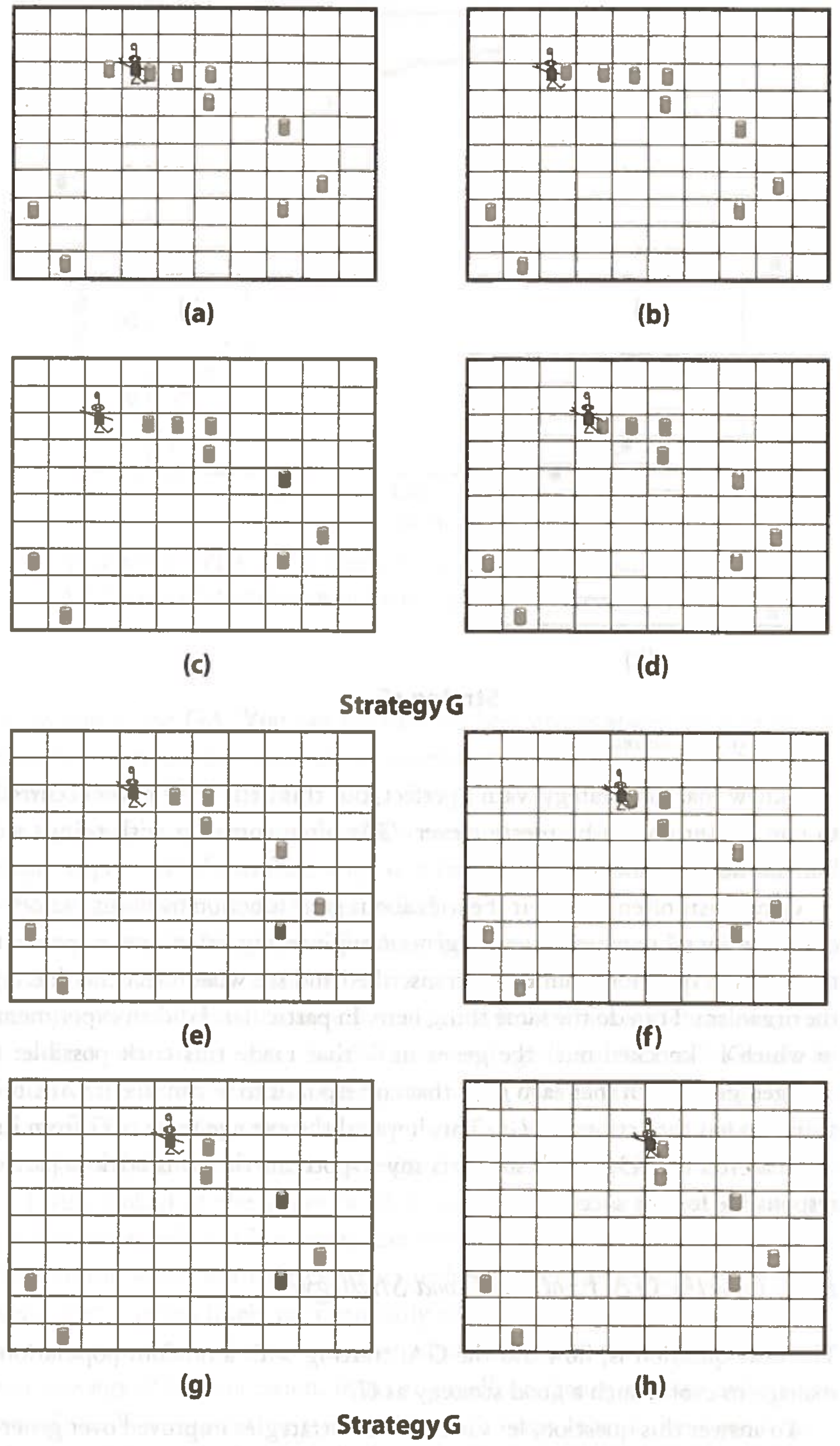


FIGURE 9.5. Robby in the same cluster of cans, using strategy G over eleven time steps. (*Continued on next page*)

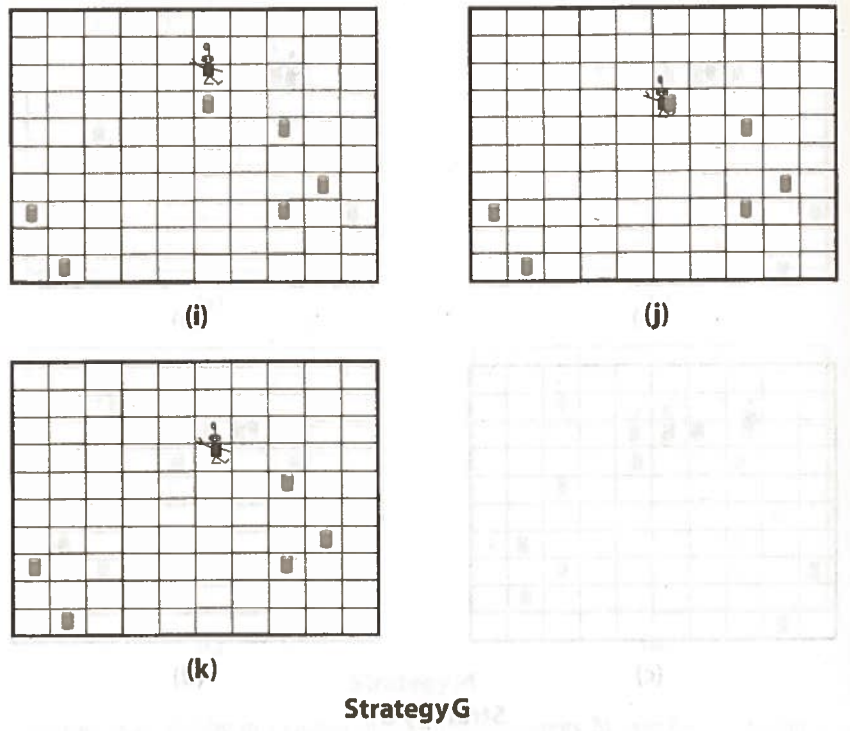


FIGURE 9.5. (*Continued*)

I knew that my strategy wasn't perfect, but this little trick never occurred to me. Evolution can be pretty clever. GAs often come up with things we humans don't consider.

Geneticists often test their theories about gene function by doing "knock-out mutations," in which they use genetic engineering techniques to prevent the genes in question from being transcribed and see what effect that has on the organism. I can do the same thing here. In particular, I did an experiment in which I "knocked out" the genes in *G* that made this trick possible: I changed genes such that each gene that corresponds to a "can in current site" situation has the action *PickUp*. This lowered the average score of *G* from its original 483 to 443, which supports my hypothesis that this trick is partly responsible for *G*'s success.

How Did the GA Evolve a Good Strategy?

The next question is, how did the GA, starting with a random population, manage to evolve such a good strategy as *G*?

To answer this question, let's look at how strategies improved over generations. In figure 9.6, I plot the fitness of the best strategy in each generation

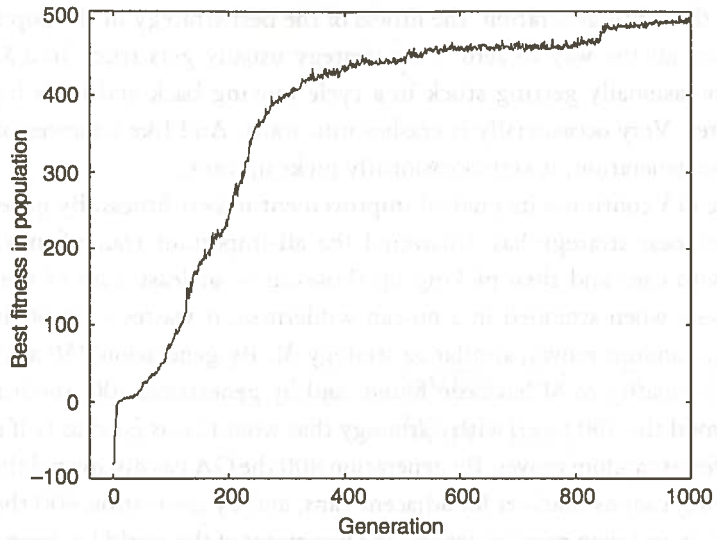


FIGURE 9.6. Plot of best fitness in the population versus generation for the run of the GA in which strategy G was evolved.

in my run of the GA. You can see that the best fitness starts out way below zero, rises very quickly until about generation 300, and then improves more slowly for the rest of the run.

The first generation consists of 200 randomly generated strategies. As you might expect, all of them are very, very bad. The best one has fitness of only −81 and the worst one has fitness −825.

I looked at the behavior of Robby when using the worst strategy of this generation, on several sessions, each starting with a different environment (configuration of cans). In some environments, Robby makes a few moves, then gets stuck, executing action *StayPut* again and again, for the entire session. In others he spends the session crashing into a wall over and over again. In others he spends his whole time trying to pick up a nonexistent can in his current site. No surprise that evolution weeded out this strategy quite early on.

I also looked at the behavior of Robby using the best strategy of this generation, which is still a pretty bad one that gets stuck in ways similar to those in the worst strategy. However, it has a couple of advantages over the worst one: it is less likely to continually crash into a wall, and it occasionally moves into a site with a can and actually picks up the can! This being the best strategy of its generation, it has an excellent chance of being selected to reproduce. When it indeed is selected, its children inherit these good traits (along with lots of bad traits).

By the tenth generation, the fitness of the best strategy in the population has risen all the way to zero. This strategy usually gets stuck in a *StayPut* loop, occasionally getting stuck in a cycle moving back and forth between two sites. Very occasionally it crashes into walls. And like its ancestor from the first generation, it very occasionally picks up cans.

The GA continues its gradual improvement in best fitness. By generation 200 the best strategy has discovered the all-important trait of moving to sites with cans and then picking up those cans—at least a lot of the time. However, when stranded in a no-can wilderness, it wastes a lot of time by making random moves, similar to strategy *M*. By generation 250 a strategy equal in quality to *M* has been found, and by generation 400, the fitness is up beyond the 400 level, with a strategy that would be as good as *G* if only it made fewer random moves. By generation 800 the GA has discovered the trick of leaving cans as markers for adjacent cans, and by generation 900 the trick of finding and then moving around the perimeter of the world has been nearly perfected, requiring only a few tweaks to get it right by generation 1,000.

Although Robby the robot is a relatively simple example for teaching people about GAs, it is not all that different from the way GAs are used in the real world. And as in the example of Robby, in real-world applications, the GA will often evolve a solution that works, but it's hard to see *why* it works. That is often because GAs find good solutions that are quite different from the ones humans would come up with. Jason Lohn, a genetic algorithms expert from the National Astronautical and Space Administration (NASA), emphasizes this point: "Evolutionary algorithms are a great tool for exploring the dark corners of design space. You show [your designs] to people with 25 years' experience in the industry and they say 'Wow, does that really work?'.... We frequently see evolved designs that are completely unintelligible."

In Lohn's case, unintelligible as it might be, it does indeed work. In 2005 Lohn and his colleagues won a "Human Competitive" award for their GA's design of a novel antenna for NASA spacecraft, reflecting the fact that the GA's design was an improvement over that of human engineers.

PART III | Computation Writ Large

The proper domain of computer science is information processing writ large across all of nature.

—Chris Langton (Quoted in Roger Lewin, *Complexity: Life at the Edge of Chaos*)