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.

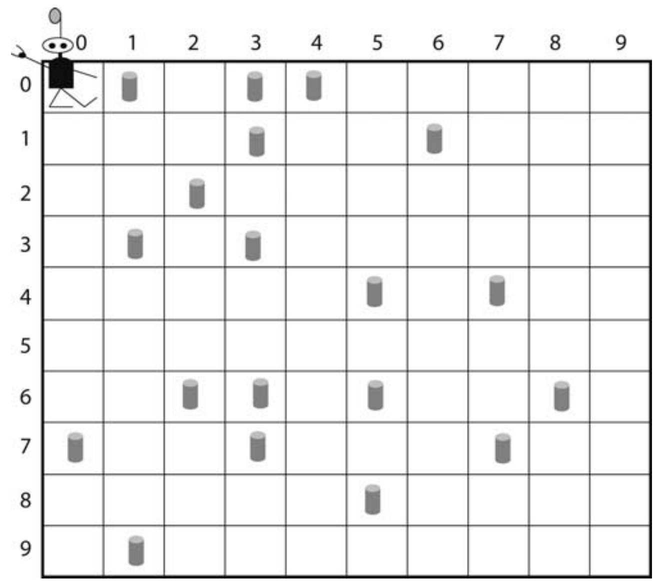


FIGURE 9.1. Robby's world. A 10 x 10 array, strewn with soda cans.

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:
2330032342163034353054600610256251511141622604356543340665115 14
1565022064064205100664321616152165202236443336334601332650300 0
406220502431650061113051466642324012456333455241261434413610 20
15063064255165404326446315616451054366534631055164600 5164

Individual 2:
16411343121025360340361241431201104235462525304202044516433665
61035322153105131440622120614631432154610256523644422025340345
30502005620634026331002453416430151631210012214400664012665246
35165015412311313245330443321263455500531421306442331 1000

Individual 3:
20423344402411226132136452632464212206122122252660626144436125
32512664061335340153411110206164226653145522540234051155031302
22020065445125062206631426135532010000400031640130154160162006
13444062616050564142155313323602150335513125363264263055 1

                .
                .
                .

Individual 200:
34632525136001012225612106043301135205155320130656005322235043
32425064124255265534635345523053326612010632124554423440613654
30246240160663016464641103026540006334126150352262106063624260
5506166163442551243544641100234633304401025332121424022 51
```

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
```

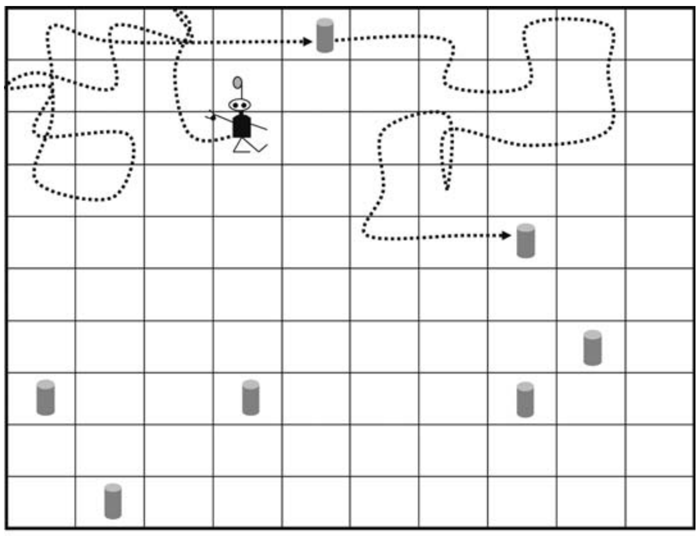
Staring at the genome of a strategy doesn't help us too much in understanding how that strategy works. We can see a few genes that make sense, such as the important situations in which Robby's current site contains a can, such as the second situation ("Empty Empty Empty Empty Can"), which has action 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



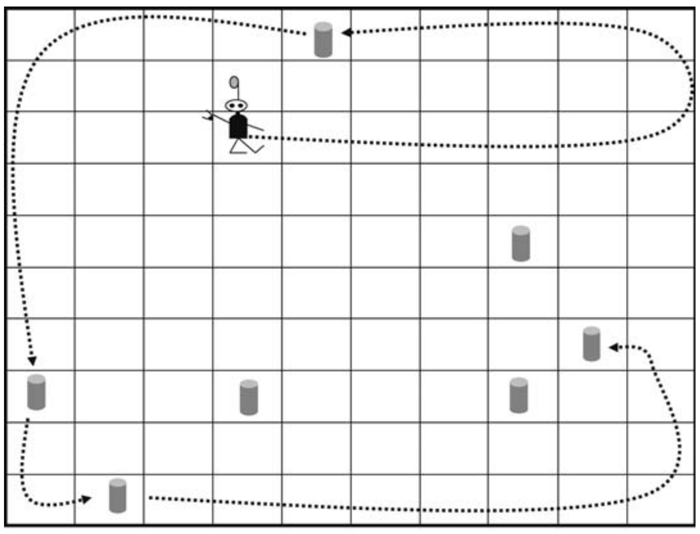


FIGURE 9.3. Robby in a "no-can" wilderness. The dotted lines show the paths he took in my simulation when he was following strategies M (top) and G (bottom).

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).

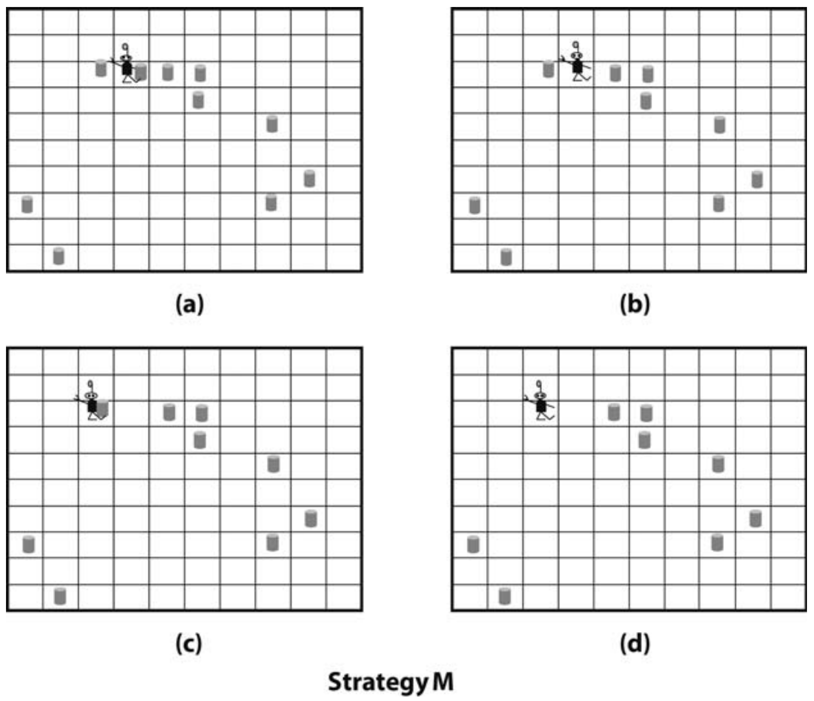


FIGURE 9.4. Robby in a cluster of cans, using strategy M over four time steps.

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).

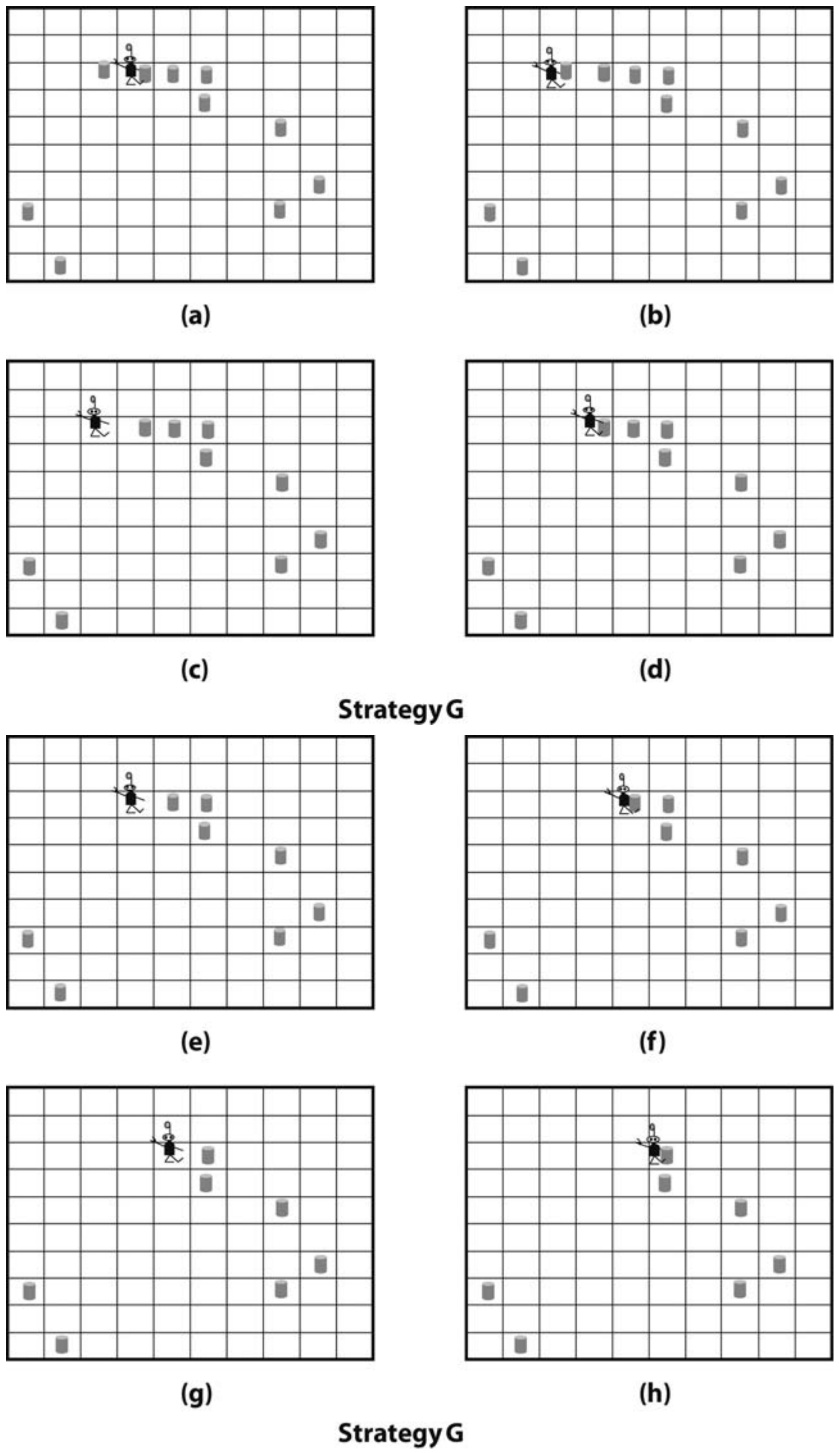


FIGURE 9.5. Robby in the same cluster of cans, using strategy G over eleven time steps. (*Continued on next page*)

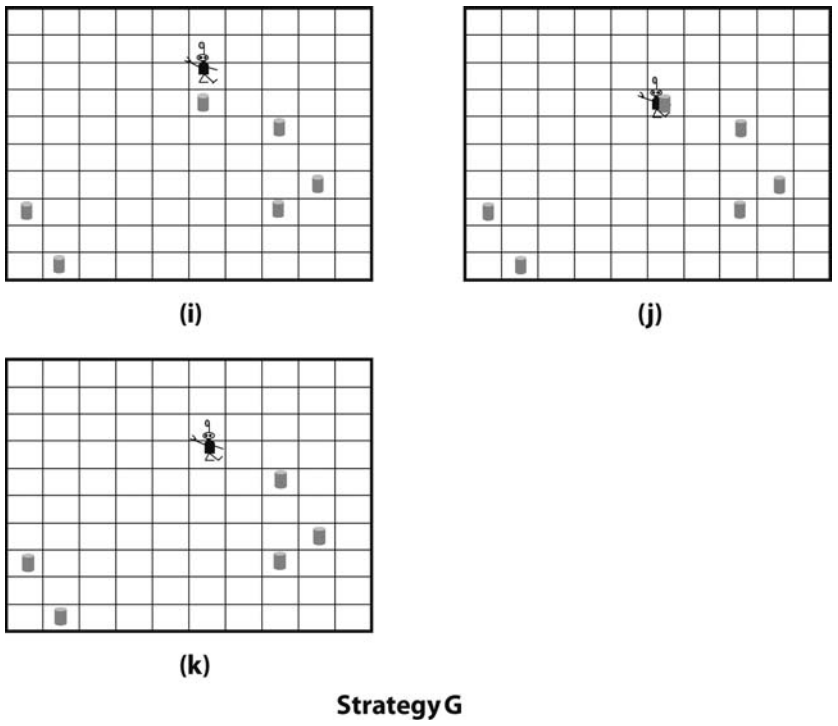


FIGURE 9.5. (*Continued*)

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

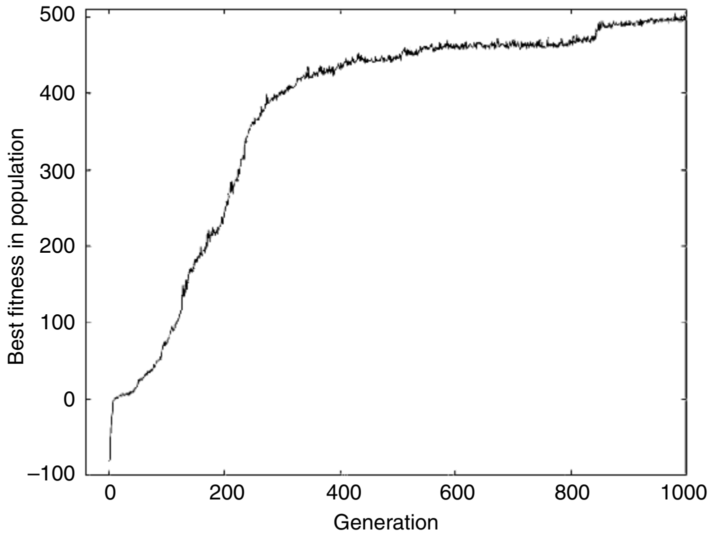


FIGURE 9.6. Plot of best fitness in the population versus generation for the run of the GA in which strategy G was evolved.

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.