CHAPTER 9

GENETIC ALGORITHMS

Genetic algorithms provide an approach to learning that is based loosely on simulated evolution. Hypotheses are often described by bit strings whose interpretation depends on the application, though hypotheses may also be described by symbolic expressions or even computer programs. The search for an appropriate hypothesis begins with a population, or collection, of initial hypotheses. Members of the current population give rise to the next generation population by means of operations such as random mutation and crossover, which are patterned after processes in biological evolution. At each step, the hypotheses in the current population are evaluated relative to a given measure of fitness, with the most fit hypotheses selected probabilistically as seeds for producing the next generation. Genetic algorithms have been applied successfully to a variety of learning tasks and to other optimization problems. For example, they have been used to learn collections of rules for robot control and to optimize the topology and learning parameters for artificial neural networks. This chapter covers both genetic algorithms, in which hypotheses are typically described by bit strings, and genetic programming, in which hypotheses are described by computer programs.

9.1 MOTIVATION

Genetic algorithms (GAs) provide a learning method motivated by an analogy to biological evolution. Rather than search from general-to-specific hypotheses, or from simple-to-complex, GAs generate successor hypotheses by repeatedly mutating and recombining parts of the best currently known hypotheses. At each step,

a collection of hypotheses called the current *population* is updated by replacing some fraction of the population by offspring of the most fit current hypotheses. The process forms a generate-and-test beam-search of hypotheses, in which variants of the best current hypotheses are most likely to be considered next. The popularity of GAs is motivated by a number of factors including:

- Evolution is known to be a successful, robust method for adaptation within biological systems.
- GAs can search spaces of hypotheses containing complex interacting parts, where the impact of each part on overall hypothesis fitness may be difficult to model.
- Genetic algorithms are easily parallelized and can take advantage of the decreasing costs of powerful computer hardware.

This chapter describes the genetic algorithm approach, illustrates its use, and examines the nature of its hypothesis space search. We also describe a variant called genetic programming, in which entire computer programs are evolved to certain fitness criteria. Genetic algorithms and genetic programming are two of the more popular approaches in a field that is sometimes called evolutionary computation. In the final section we touch on selected topics in the study of biological evolution, including the Baldwin effect, which describes an interesting interplay between the learning capabilities of single individuals and the rate of evolution of the entire population.

9.2 GENETIC ALGORITHMS

The problem addressed by GAs is to search a space of candidate hypotheses to identify the best hypothesis. In GAs the "best hypothesis" is defined as the one that optimizes a predefined numerical measure for the problem at hand, called the hypothesis *fitness*. For example, if the learning task is the problem of approximating an unknown function given training examples of its input and output, then fitness could be defined as the accuracy of the hypothesis over this training data. If the task is to learn a strategy for playing chess, fitness could be defined as the number of games won by the individual when playing against other individuals in the current population.

Although different implementations of genetic algorithms vary in their details, they typically share the following structure: The algorithm operates by iteratively updating a pool of hypotheses, called the population. On each iteration, all members of the population are evaluated according to the fitness function. A new population is then generated by probabilistically selecting the most fit individuals from the current population. Some of these selected individuals are carried forward into the next generation population intact. Others are used as the basis for creating new offspring individuals by applying genetic operations such as crossover and mutation.

GA($Fitness$, $Fitness\_threshold$, p, r, m)

Fitness: A function that assigns an evaluation score, given a hypothesis.

Fitness_threshold: A threshold specifying the termination criterion.

p: The number of hypotheses to be included in the population.

r: The fraction of the population to be replaced by Crossover at each step.

m: The mutation rate.

- *Initialize population:* $P \leftarrow$ Generate p hypotheses at random
- *Evaluate:* For each h in P, compute $Fitness(h)$
- While $[\max_h Fitness(h)] < Fitness\_threshold$ do

  Create a new generation, P_S:

  1. *Select:* Probabilistically select $(1-r)p$ members of P to add to P_S. The probability $\Pr(h_i)$ of selecting hypothesis h_i from P is given by

$$\Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)}$$

  2. *Crossover:* Probabilistically select $\frac{r \cdot p}{2}$ pairs of hypotheses from P, according to $\Pr(h_i)$ given above. For each pair, $\langle h_1, h_2 \rangle$, produce two offspring by applying the Crossover operator. Add all offspring to P_s.
  3. *Mutate:* Choose m percent of the members of P_s with uniform probability. For each, invert one randomly selected bit in its representation.
  4. *Update:* $P \leftarrow P_s$.
  5. *Evaluate:* for each h in P, compute $Fitness(h)$
- Return the hypothesis from P that has the highest fitness.

TABLE 9.1
A prototypical genetic algorithm. A population containing p hypotheses is maintained. On each iteration, the successor population P_S is formed by probabilistically selecting current hypotheses according to their fitness and by adding new hypotheses. New hypotheses are created by applying a crossover operator to pairs of most fit hypotheses and by creating single point mutations in the resulting generation of hypotheses. This process is iterated until sufficiently fit hypotheses are discovered. Typical crossover and mutation operators are defined in a subsequent table.

A prototypical genetic algorithm is described in Table 9.1. The inputs to this algorithm include the fitness function for ranking candidate hypotheses, a threshold defining an acceptable level of fitness for terminating the algorithm, the size of the population to be maintained, and parameters that determine how successor populations are to be generated: the fraction of the population to be replaced at each generation and the mutation rate.

Notice in this algorithm each iteration through the main loop produces a new generation of hypotheses based on the current population. First, a certain number of hypotheses from the current population are selected for inclusion in the next generation. These are selected *probabilistically*, where the probability of selecting hypothesis h_i is given by

$$\Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)} \tag{9.1}$$

Thus, the probability that a hypothesis will be selected is proportional to its own fitness and is inversely proportional to the fitness of the other competing hypotheses in the current population.

Once these members of the current generation have been selected for inclusion in the next generation population, additional members are generated using a crossover operation. Crossover, defined in detail in the next section, takes two parent hypotheses from the current generation and creates two offspring hypotheses by recombining portions of both parents. The parent hypotheses are chosen probabilistically from the current population, again using the probability function given by Equation (9.1). After new members have been created by this crossover operation, the new generation population now contains the desired number of members. At this point, a certain fraction m of these members are chosen at random, and random mutations all performed to alter these members.

This GA algorithm thus performs a randomized, parallel beam search for hypotheses that perform well according to the fitness function. In the following subsections, we describe in more detail the representation of hypotheses and genetic operators used in this algorithm.

9.2.1 Representing Hypotheses

Hypotheses in GAs are often represented by bit strings, so that they can be easily manipulated by genetic operators such as mutation and crossover. The hypotheses represented by these bit strings can be quite complex. For example, sets of if-then rules can easily be represented in this way, by choosing an encoding of rules that allocates specific substrings for each rule precondition and postcondition. Examples of such rule representations in GA systems are described by Holland (1986); Grefenstette (1988); and DeJong et al. (1993).

To see how if-then rules can be encoded by bit strings, first consider how we might use a bit string to describe a constraint on the value of a single attribute. To pick an example, consider the attribute *Outlook*, which can take on any of the three values *Sunny*, *Overcast*, or *Rain*. One obvious way to represent a constraint on *Outlook* is to use a bit string of length three, in which each bit position corresponds to one of its three possible values. Placing a 1 in some position indicates that the attribute is allowed to take on the corresponding value. For example, the string 010 represents the constraint that *Outlook* must take on the second of these values, or $Outlook = Overcast$. Similarly, the string 011 represents the more general constraint that allows two possible values, or $(Outlook = Overcast \vee Rain)$. Note 111 represents the most general possible constraint, indicating that we don't care which of its possible values the attribute takes on.

Given this method for representing constraints on a single attribute, conjunctions of constraints on multiple attributes can easily be represented by concatenating the corresponding bit strings. For example, consider a second attribute, *Wind*, that can take on the value *Strong* or *Weak*. A rule precondition such as

$$(Outlook = Overcast \vee Rain) \wedge (Wind = Strong)$$

can then be represented by the following bit string of length five:

Outlook	*Wind*
011	10

Rule postconditions (such as $PlayTennis = yes$) can be represented in a similar fashion. Thus, an entire rule can be described by concatenating the bit strings describing the rule preconditions, together with the bit string describing the rule postcondition. For example, the rule

$$\text{IF } Wind = Strong \quad \text{THEN } PlayTennis = yes$$

would be represented by the string

Outlook	*Wind*	*PlayTennis*
111	10	10

where the first three bits describe the "don't care" constraint on *Outlook*, the next two bits describe the constraint on *Wind*, and the final two bits describe the rule postcondition (here we assume *PlayTennis* can take on the values *Yes* or *No*). Note the bit string representing the rule contains a substring for each attribute in the hypothesis space, even if that attribute is not constrained by the rule preconditions. This yields a fixed length bit-string representation for rules, in which substrings at specific locations describe constraints on specific attributes. Given this representation for single rules, we can represent sets of rules by similarly concatenating the bit string representations of the individual rules.

In designing a bit string encoding for some hypothesis space, it is useful to arrange for every syntactically legal bit string to represent a well-defined hypothesis. To illustrate, note in the rule encoding in the above paragraph the bit string 111 10 11 represents a rule whose postcondition does not constrain the target attribute *PlayTennis*. If we wish to avoid considering this hypothesis, we may employ a different encoding (e.g., allocate just one bit to the *PlayTennis* postcondition to indicate whether the value is *Yes* or *No*), alter the genetic operators so that they explicitly avoid constructing such bit strings, or simply assign a very low fitness to such bit strings.

In some GAs, hypotheses are represented by symbolic descriptions rather than bit strings. For example, in Section 9.5 we discuss a genetic algorithm that encodes hypotheses as computer programs.

9.2.2 Genetic Operators

The generation of successors in a GA is determined by a set of operators that recombine and mutate selected members of the current population. Typical GA operators for manipulating bit string hypotheses are illustrated in Table 9.1. These operators correspond to idealized versions of the genetic operations found in biological evolution. The two most common operators are *crossover* and *mutation*.

The *crossover operator* produces two new offspring from two parent strings, by copying selected bits from each parent. The bit at position i in each offspring is copied from the bit at position i in one of the two parents. The choice of which parent contributes the bit for position i is determined by an additional string called the *crossover mask*. To illustrate, consider the *single-point crossover* operator at the top of Table 9.2. Consider the topmost of the two offspring in this case. This offspring takes its first five bits from the first parent and its remaining six bits from the second parent, because the crossover mask 11111000000 specifies these choices for each of the bit positions. The second offspring uses the same crossover mask, but switches the roles of the two parents. Therefore, it contains the bits that were not used by the first offspring. In single-point crossover, the crossover mask is always constructed so that it begins with a string containing n contiguous 1s, followed by the necessary number of 0s to complete the string. This results in offspring in which the first n bits are contributed by one parent and the remaining bits by the second parent. Each time the single-point crossover operator is applied,

	Initial strings	*Crossover Mask*	*Offspring*
Single-point crossover:	11101001000 00001010101	11111000000	11101010101 00001001000
Two-point crossover:	11101001000 00001010101	00111110000	11001011000 00101000101
Uniform crossover:	11101001000 00001010101	10011010011	10001000100 01101011001
Point mutation:	11101001000	⟶	11101011000

TABLE 9.2
Common operators for genetic algorithms. These operators form offspring of hypotheses represented by bit strings. The crossover operators create two descendants from two parents, using the crossover mask to determine which parent contributes which bits. Mutation creates a single descendant from a single parent by changing the value of a randomly chosen bit.

the crossover point n is chosen at random, and the crossover mask is then created and applied.

In *two-point crossover*, offspring are created by substituting intermediate segments of one parent into the middle of the second parent string. Put another way, the crossover mask is a string beginning with n_0 zeros, followed by a contiguous string of n_1 ones, followed by the necessary number of zeros to complete the string. Each time the two-point crossover operator is applied, a mask is generated by randomly choosing the integers n_0 and n_1. For instance, in the example shown in Table 9.2 the offspring are created using a mask for which $n_0 = 2$ and $n_1 = 5$. Again, the two offspring are created by switching the roles played by the two parents.

Uniform crossover combines bits sampled uniformly from the two parents, as illustrated in Table 9.2. In this case the crossover mask is generated as a random bit string with each bit chosen at random and independent of the others.

In addition to recombination operators that produce offspring by combining parts of two parents, a second type of operator produces offspring from a single parent. In particular, the *mutation* operator produces small random changes to the bit string by choosing a single bit at random, then changing its value. Mutation is often performed after crossover has been applied as in our prototypical algorithm from Table 9.1.

Some GA systems employ additional operators, especially operators that are specialized to the particular hypothesis representation used by the system. For example, Grefenstette et al. (1991) describe a system that learns sets of rules for robot control. It uses mutation and crossover, together with an operator for specializing rules. Janikow (1993) describes a system that learns sets of rules using operators that generalize and specialize rules in a variety of directed ways (e.g., by explicitly replacing the condition on an attribute by "don't care").

9.2.3 Fitness Function and Selection

The fitness function defines the criterion for ranking potential hypotheses and for probabilistically selecting them for inclusion in the next generation population. If the task is to learn classification rules, then the fitness function typically has a component that scores the classification accuracy of the rule over a set of provided training examples. Often other criteria may be included as well, such as the complexity or generality of the rule. More generally, when the bit-string hypothesis is interpreted as a complex procedure (e.g., when the bit string represents a collection of if-then rules that will be chained together to control a robotic device), the fitness function may measure the overall performance of the resulting procedure rather than performance of individual rules.

In our prototypical GA shown in Table 9.1, the probability that a hypothesis will be selected is given by the ratio of its fitness to the fitness of other members of the current population as seen in Equation (9.1). This method is sometimes called *fitness proportionate selection*, or roulette wheel selection. Other methods for using fitness to select hypotheses have also been proposed. For example, in

tournament selection, two hypotheses are first chosen at random from the current population. With some predefined probability p the more fit of these two is then selected, and with probability $(1 - p)$ the less fit hypothesis is selected. Tournament selection often yields a more diverse population than fitness proportionate selection (Goldberg and Deb 1991). In another method called *rank selection*, the hypotheses in the current population are first sorted by fitness. The probability that a hypothesis will be selected is then proportional to its rank in this sorted list, rather than its fitness.

9.3 AN ILLUSTRATIVE EXAMPLE

A genetic algorithm can be viewed as a general optimization method that searches a large space of candidate objects seeking one that performs best according to the fitness function. Although not guaranteed to find an optimal object, GAs often succeed in finding an object with high fitness. GAs have been applied to a number of optimization problems outside machine learning, including problems such as circuit layout and job-shop scheduling. Within machine learning, they have been applied both to function-approximation problems and to tasks such as choosing the network topology for artificial neural network learning systems.

To illustrate the use of GAs for concept learning, we briefly summarize the GABIL system described by DeJong et al. (1993). GABIL uses a GA to learn boolean concepts represented by a disjunctive set of propositional rules. In experiments over several concept learning problems, GABIL was found to be roughly comparable in generalization accuracy to other learning algorithms such as the decision tree learning algorithm C4.5 and the rule learning system AQ14. The learning tasks in this study included both artificial learning tasks designed to explore the systems' generalization accuracy and the real world problem of breast cancer diagnosis.

The algorithm used by GABIL is exactly the algorithm described in Table 9.1. In experiments reported by DeJong et al. (1993), the parameter r, which determines the fraction of the parent population replaced by crossover, was set to 0.6. The parameter m, which determines the mutation rate, was set to 0.001. These are typical settings for these parameters. The population size p was varied from 100 to 1000, depending on the specific learning task.

The specific instantiation of the GA algorithm in GABIL can be summarized as follows:

- **Representation.** Each hypothesis in GABIL corresponds to a disjunctive set of propositional rules, encoded as described in Section 9.2.1. In particular, the hypothesis space of rule preconditions consists of a conjunction of constraints on a fixed set of attributes, as described in that earlier section. To represent a set of rules, the bit-string representations of individual rules are concatenated. To illustrate, consider a hypothesis space in which rule preconditions are conjunctions of constraints over two boolean attributes, a_1 and a_2. The rule postcondition is described by a single bit that indicates the predicted

value of the target attribute c. Thus, the hypothesis consisting of the two rules

$$\text{IF } a_1 = T \wedge a_2 = F \text{ THEN } c = T; \quad \text{IF } a_2 = T \text{ THEN } c = F$$

would be represented by the string

a_1	a_2	c	a_1	a_2	c
10	01	1	11	10	0

Note the length of the bit string grows with the number of rules in the hypothesis. This variable bit-string length requires a slight modification to the crossover operator, as described below.

- **Genetic operators.** GABIL uses the standard mutation operator of Table 9.2, in which a single bit is chosen at random and replaced by its complement. The crossover operator that it uses is a fairly standard extension to the two-point crossover operator described in Table 9.2. In particular, to accommodate the variable-length bit strings that encode rule sets, and to constrain the system so that crossover occurs only between like sections of the bit strings that encode rules, the following approach is taken. To perform a crossover operation on two parents, two crossover points are first chosen at random in the first parent string. Let d_1 (d_2) denote the distance from the leftmost (rightmost) of these two crossover points to the rule boundary immediately to its left. The crossover points in the second parent are now randomly chosen, subject to the constraint that they must have the same d_1 and d_2 value. For example, if the two parent strings are

	a_1	a_2	c	a_1	a_2	c
h_1:	10	01	1	11	10	0

and

	a_1	a_2	c	a_1	a_2	c
h_2:	01	11	0	10	01	0

and the crossover points chosen for the first parent are the points following bit positions 1 and 8,

	a_1	a_2	c	a_1	a_2	c
h_1:	1[0	01	1	11	1]0	0

where "[" and "]" indicate crossover points, then $d_1 = 1$ and $d_2 = 3$. Hence the allowed pairs of crossover points for the second parent include the pairs of bit positions $\langle 1, 3\rangle$, $\langle 1, 8\rangle$, and $\langle 6, 8\rangle$. If the pair $\langle 1, 3\rangle$ happens to be chosen,

	a_1	a_2	c	a_1	a_2	c
h_2:	0[1	1]1	0	10	01	0

then the two resulting offspring will be

	a_1	a_2	c
h_3:	11	10	0

and

	a_1	a_2	c	a_1	a_2	c	a_1	a_2	c
h_4:	00	01	1	11	11	0	10	01	0

As this example illustrates, this crossover operation enables offspring to contain a different number of rules than their parents, while assuring that all bit strings generated in this fashion represent well-defined rule sets.

- **Fitness function.** The fitness of each hypothesized rule set is based on its classification accuracy over the training data. In particular, the function used to measure fitness is

$$Fitness(h) = (correct(h))^2$$

where $correct(h)$ is the percent of all training examples correctly classified by hypothesis h.

In experiments comparing the behavior of GABIL to decision tree learning algorithms such as C4.5 and ID5R, and to the rule learning algorithm AQ14, DeJong et al. (1993) report roughly comparable performance among these systems, tested on a variety of learning problems. For example, over a set of 12 synthetic problems, GABIL achieved an average generalization accuracy of 92.1 %, whereas the performance of the other systems ranged from 91.2 % to 96.6 %.

9.3.1 Extensions

DeJong et al. (1993) also explore two interesting extensions to the basic design of GABIL. In one set of experiments they explored the addition of two new genetic operators that were motivated by the generalization operators common in many symbolic learning methods. The first of these operators, *AddAlternative*, generalizes the constraint on a specific attribute by changing a 0 to a 1 in the substring corresponding to the attribute. For example, if the constraint on an attribute is represented by the string 10010, this operator might change it to 10110. This operator was applied with probability .01 to selected members of the population on each generation. The second operator, *DropCondition* performs a more drastic generalization step, by replacing all bits for a particular attribute by a 1. This operator corresponds to generalizing the rule by completely dropping the constraint on the attribute, and was applied on each generation with probability .60. The authors report this revised system achieved an average performance of 95.2% over the above set of synthetic learning tasks, compared to 92.1% for the basic GA algorithm.

In the above experiment, the two new operators were applied with the same probability to each hypothesis in the population on each generation. In a second experiment, the bit-string representation for hypotheses was extended to include two bits that determine which of these operators may be applied to the hypothesis. In this extended representation, the bit string for a typical rule set hypothesis would be

a_1	a_2	c	a_1	a_2	c	AA	DC
01	11	0	10	01	0	1	0

where the final two bits indicate in this case that the *AddAlternative* operator may be applied to this bit string, but that the *DropCondition* operator may not. These two new bits define part of the search strategy used by the GA and are themselves altered and evolved using the same crossover and mutation operators that operate on other bits in the string. While the authors report mixed results with this approach (i.e., improved performance on some problems, decreased performance on others), it provides an interesting illustration of how GAs might in principle be used to evolve their own hypothesis search methods.

9.4 HYPOTHESIS SPACE SEARCH

As illustrated above, GAs employ a randomized beam search method to seek a maximally fit hypothesis. This search is quite different from that of other learning methods we have considered in this book. To contrast the hypothesis space search of GAs with that of neural network BACKPROPAGATION, for example, the gradient descent search in BACKPROPAGATION moves smoothly from one hypothesis to a new hypothesis that is very similar. In contrast, the GA search can move much more abruptly, replacing a parent hypothesis by an offspring that may be radically different from the parent. Note the GA search is therefore less likely to fall into the same kind of local minima that can plague gradient descent methods.

One practical difficulty in some GA applications is the problem of *crowding*. Crowding is a phenomenon in which some individual that is more highly fit than others in the population quickly reproduces, so that copies of this individual and very similar individuals take over a large fraction of the population. The negative impact of crowding is that it reduces the diversity of the population, thereby slowing further progress by the GA. Several strategies have been explored for reducing crowding. One approach is to alter the selection function, using criteria such as tournament selection or rank selection in place of fitness proportionate roulette wheel selection. A related strategy is "fitness sharing," in which the measured fitness of an individual is reduced by the presence of other, similar individuals in the population. A third approach is to restrict the kinds of individuals allowed to recombine to form offspring. For example, by allowing only the most similar individuals to recombine, we can encourage the formation of clusters of similar individuals, or multiple "subspecies" within the population. A related approach is to spatially distribute individuals and allow only nearby individuals to recombine. Many of these techniques are inspired by the analogy to biological evolution.

9.4.1 Population Evolution and the Schema Theorem

It is interesting to ask whether one can mathematically characterize the evolution over time of the population within a GA. The schema theorem of Holland (1975) provides one such characterization. It is based on the concept of *schemas*, or patterns that describe sets of bit strings. To be precise, a schema is any string composed of 0s, 1s, and *'s. Each schema represents the set of bit strings containing the indicated 0s and 1s, with each "*" interpreted as a "don't care." For example, the schema 0*10 represents the set of bit strings that includes exactly 0010 and 0110.

An individual bit string can be viewed as a representative of each of the different schemas that it matches. For example, the bit string 0010 can be thought of as a representative of 2^4 distinct schemas including 00**, 0*10, ****, etc. Similarly, a population of bit strings can be viewed in terms of the set of schemas that it represents and the number of individuals associated with each of these schema.

The schema theorem characterizes the evolution of the population within a GA in terms of the number of instances representing each schema. Let $m(s, t)$ denote the number of instances of schema s in the population at time t (i.e., during the tth generation). The schema theorem describes the expected value of $m(s, t+1)$ in terms of $m(s, t)$ and other properties of the schema, population, and GA algorithm parameters.

The evolution of the population in the GA depends on the selection step, the recombination step, and the mutation step. Let us start by considering just the effect of the selection step. Let $f(h)$ denote the fitness of the individual bit string h and $\bar{f}(t)$ denote the average fitness of all individuals in the population at time t. Let n be the total number of individuals in the population. Let $h \in s \cap p_t$ indicate that the individual h is both a representative of schema s and a member of the population at time t. Finally, let $\hat{u}(s, t)$ denote the average fitness of instances of schema s in the population at time t.

We are interested in calculating the expected value of $m(s, t+1)$, which we denote $E[m(s, t+1)]$. We can calculate $E[m(s, t+1)]$ using the probability distribution for selection given in Equation (9.1), which can be restated using our current terminology as follows:

$$\Pr(h) = \frac{f(h)}{\sum_{i=1}^{n} f(h_i)} = \frac{f(h)}{n\bar{f}(t)}$$

Now if we select one member for the new population according to this probability distribution, then the probability that we will select a representative of schema s is

$$\Pr(h \in s) = \sum_{h \in s \cap p_t} \frac{f(h)}{n\bar{f}(t)} = \frac{\hat{u}(s, t)}{n\bar{f}(t)} m(s, t) \tag{9.2}$$

The second step above follows from the fact that by definition,

$$\hat{u}(s,t) = \frac{\sum_{h \in s \cap p_t} f(h)}{m(s,t)}$$

Equation (9.2) gives the probability that a single hypothesis selected by the GA will be an instance of schema s. Therefore, the expected number of instances of s resulting from the n independent selection steps that create the entire new generation is just n times this probability.

$$E[m(s,t+1)] = \frac{\hat{u}(s,t)}{\bar{f}(t)} m(s,t) \tag{9.3}$$

Equation (9.3) states that the expected number of instances of schema s at generation $t+1$ is proportional to the average fitness $\hat{u}(s,t)$ of instances of this schema at time t, and inversely proportional to the average fitness $\bar{f}(t)$ of all members of the population at time t. Thus, we can expect schemas with above average fitness to be represented with increasing frequency on successive generations. If we view the GA as performing a virtual parallel search through the space of possible schemas at the same time it performs its explicit parallel search through the space of individuals, then Equation (9.3) indicates that more fit schemas will grow in influence over time.

While the above analysis considered only the selection step of the GA, the crossover and mutation steps must be considered as well. The schema theorem considers only the possible negative influence of these genetic operators (e.g., random mutation may decrease the number of representatives of s, independent of $\hat{u}(s,t)$), and considers only the case of single-point crossover. The full schema theorem thus provides a lower bound on the expected frequency of schema s, as follows:

$$E[m(s,t+1)] \geq \frac{\hat{u}(s,t)}{\bar{f}(t)} m(s,t) \left(1 - p_c \frac{d(s)}{l-1}\right) (1 - p_m)^{o(s)} \tag{9.4}$$

Here, p_c is the probability that the single-point crossover operator will be applied to an arbitrary individual, and p_m is the probability that an arbitrary bit of an arbitrary individual will be mutated by the mutation operator. $o(s)$ is the number of *defined bits* in schema s, where 0 and 1 are defined bits, but * is not. $d(s)$ is the distance between the leftmost and rightmost defined bits in s. Finally, l is the length of the individual bit strings in the population. Notice the leftmost term in Equation (9.4) is identical to the term from Equation (9.3) and describes the effect of the selection step. The middle term describes the effect of the single-point crossover operator—in particular, it describes the probability that an arbitrary individual representing s will still represent s following application of this crossover operator. The rightmost term describes the probability that an arbitrary individual representing schema s will still represent schema s following application of the mutation operator. Note that the effects of single-point crossover and mutation increase with the number of defined bits $o(s)$ in the schema and with the distance $d(s)$ between the defined bits. Thus, the schema theorem can be roughly interpreted as stating that more fit schemas will tend to grow in influence, especially schemas

containing a small number of defined bits (i.e., containing a large number of *'s), and especially when these defined bits are near one another within the bit string.

The schema theorem is perhaps the most widely cited characterization of population evolution within a GA. One way in which it is incomplete is that it fails to consider the (presumably) positive effects of crossover and mutation. Numerous more recent theoretical analyses have been proposed, including analyses based on Markov chain models and on statistical mechanics models. See, for example, Whitley and Vose (1995) and Mitchell (1996).

9.5 GENETIC PROGRAMMING

Genetic programming (GP) is a form of evolutionary computation in which the individuals in the evolving population are computer programs rather than bit strings. Koza (1992) describes the basic genetic programming approach and presents a broad range of simple programs that can be successfully learned by GP.

9.5.1 Representing Programs

Programs manipulated by a GP are typically represented by trees corresponding to the parse tree of the program. Each function call is represented by a node in the tree, and the arguments to the function are given by its descendant nodes. For example, Figure 9.1 illustrates this tree representation for the function $\sin(x) + \sqrt{x^2 + y}$. To apply genetic programming to a particular domain, the user must define the primitive functions to be considered (e.g., sin, cos, $\sqrt{}$, +, −, exponentials), as well as the terminals (e.g., x, y, constants such as 2). The genetic programming algorithm then uses an evolutionary search to explore the vast space of programs that can be described using these primitives.

As in a genetic algorithm, the prototypical genetic programming algorithm maintains a population of individuals (in this case, program trees). On each iteration, it produces a new generation of individuals using selection, crossover, and mutation. The fitness of a given individual program in the population is typically determined by executing the program on a set of training data. Crossover operations are performed by replacing a randomly chosen subtree of one parent

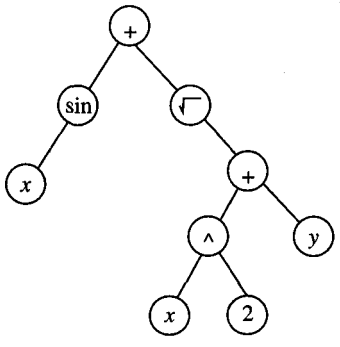

FIGURE 9.1
Program tree representation in genetic programming. Arbitrary programs are represented by their parse trees.

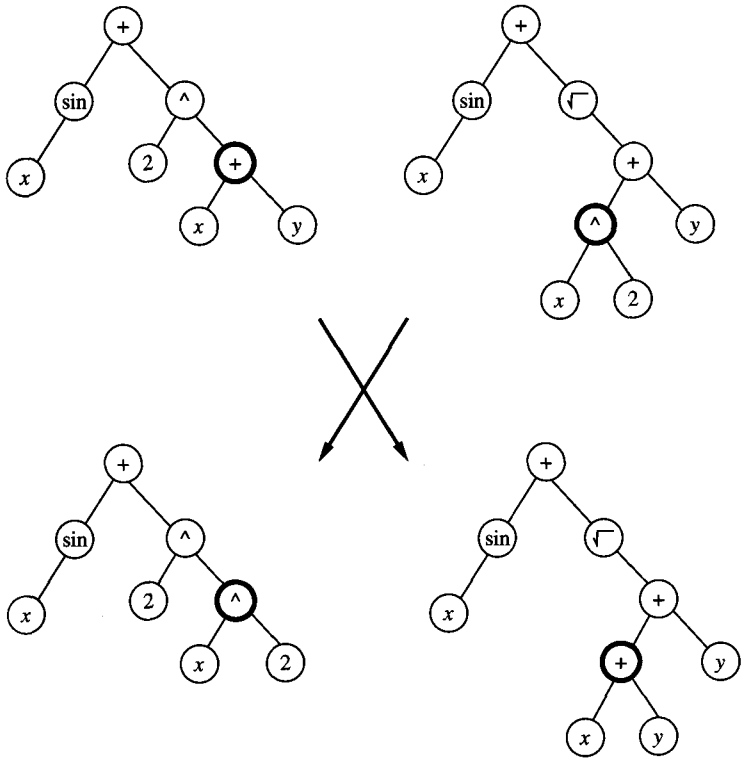

FIGURE 9.2
Crossover operation applied to two parent program trees (top). Crossover points (nodes shown in bold at top) are chosen at random. The subtrees rooted at these crossover points are then exchanged to create children trees (bottom).

program by a subtree from the other parent program. Figure 9.2 illustrates a typical crossover operation.

Koza (1992) describes a set of experiments applying a GP to a number of applications. In his experiments, 10% of the current population, selected probabilistically according to fitness, is retained unchanged in the next generation. The remainder of the new generation is created by applying crossover to pairs of programs from the current generation, again selected probabilistically according to their fitness. The mutation operator was not used in this particular set of experiments.

9.5.2 Illustrative Example

One illustrative example presented by Koza (1992) involves learning an algorithm for stacking the blocks shown in Figure 9.3. The task is to develop a general algorithm for stacking the blocks into a single stack that spells the word "universal,"

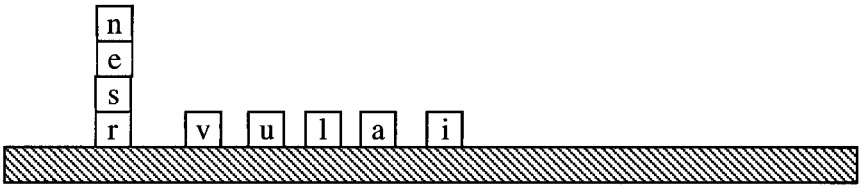

FIGURE 9.3
A block-stacking problem. The task for GP is to discover a program that can transform an arbitrary initial configuration of blocks into a stack that spells the word "universal." A set of 166 such initial configurations was provided to evaluate fitness of candidate programs (after Koza 1992).

independent of the initial configuration of blocks in the world. The actions available for manipulating blocks allow moving only a single block at a time. In particular, the top block on the stack can be moved to the table surface, or a block on the table surface can be moved to the top of the stack.

As in most GP applications, the choice of problem representation has a significant impact on the ease of solving the problem. In Koza's formulation, the primitive functions used to compose programs for this task include the following three terminal arguments:

- CS (current stack), which refers to the name of the top block on the stack, or F if there is no current stack.
- TB (top correct block), which refers to the name of the topmost block on the stack, such that it and those blocks beneath it are in the correct order.
- NN (next necessary), which refers to the name of the next block needed above TB in the stack, in order to spell the word "universal," or F if no more blocks are needed.

As can be seen, this particular choice of terminal arguments provides a natural representation for describing programs for manipulating blocks for this task. Imagine, in contrast, the relative difficulty of the task if we were to instead define the terminal arguments to be the x and y coordinates of each block.

In addition to these terminal arguments, the program language in this application included the following primitive functions:

- (MS x) (move to stack), if block x is on the table, this operator moves x to the top of the stack and returns the value T. Otherwise, it does nothing and returns the value F.
- (MT x) (move to table), if block x is somewhere in the stack, this moves the block at the top of the stack to the table and returns the value T. Otherwise, it returns the value F.
- (EQ x y) (equal), which returns T if x equals y, and returns F otherwise.
- (NOT x), which returns T if $x = F$, and returns F if $x = T$.

- (DU x y) (do until), which executes the expression x repeatedly until expression y returns the value T.

To allow the system to evaluate the fitness of any given program, Koza provided a set of 166 training example problems representing a broad variety of initial block configurations, including problems of differing degrees of difficulty. The fitness of any given program was taken to be the number of these examples solved by the algorithm. The population was initialized to a set of 300 random programs. After 10 generations, the system discovered the following program, which solves all 166 problems.

(EQ (DU (MT CS)(NOT CS)) (DU (MS NN)(NOT NN)))

Notice this program contains a sequence of two DU, or "Do Until" statements. The first repeatedly moves the current top of the stack onto the table, until the stack becomes empty. The second "Do Until" statement then repeatedly moves the next necessary block from the table onto the stack. The role played by the top level EQ expression here is to provide a syntactically legal way to sequence these two "Do Until" loops.

Somewhat surprisingly, after only a few generations, this GP was able to discover a program that solves all 166 training problems. Of course the ability of the system to accomplish this depends strongly on the primitive arguments and functions provided, and on the set of training example cases used to evaluate fitness.

9.5.3 Remarks on Genetic Programming

As illustrated in the above example, genetic programming extends genetic algorithms to the evolution of complete computer programs. Despite the huge size of the hypothesis space it must search, genetic programming has been demonstrated to produce intriguing results in a number of applications. A comparison of GP to other methods for searching through the space of computer programs, such as hillclimbing and simulated annealing, is given by O'Reilly and Oppacher (1994).

While the above example of GP search is fairly simple, Koza et al. (1996) summarize the use of a GP in several more complex tasks such as designing electronic filter circuits and classifying segments of protein molecules. The filter circuit design problem provides an example of a considerably more complex problem. Here, programs are evolved that transform a simple fixed seed circuit into a final circuit design. The primitive functions used by the GP to construct its programs are functions that edit the seed circuit by inserting or deleting circuit components and wiring connections. The fitness of each program is calculated by simulating the circuit it outputs (using the SPICE circuit simulator) to determine how closely this circuit meets the design specifications for the desired filter. More precisely, the fitness score is the sum of the magnitudes of errors between the desired and actual circuit output at 101 different input frequencies. In this case, a population of size 640,000 was maintained, with selection

producing 10% of the successor population, crossover producing 89%, and mutation producing 1%. The system was executed on a 64-node parallel processor. Within the first randomly generated population, the circuits produced were so unreasonable that the SPICE simulator could not even simulate the behavior of 98% of the circuits. The percentage of unsimulatable circuits dropped to 84.9% following the first generation, to 75.0% following the second generation, and to an average of 9.6% over succeeding generations. The fitness score of the best circuit in the initial population was 159, compared to a score of 39 after 20 generations and a score of 0.8 after 137 generations. The best circuit, produced after 137 generations, exhibited performance very similar to the desired behavior.

In most cases, the performance of genetic programming depends crucially on the choice of representation and on the choice of fitness function. For this reason, an active area of current research is aimed at the automatic discovery and incorporation of subroutines that improve on the original set of primitive functions, thereby allowing the system to dynamically alter the primitives from which it constructs individuals. See, for example, Koza (1994).

9.6 MODELS OF EVOLUTION AND LEARNING

In many natural systems, individual organisms learn to adapt significantly during their lifetime. At the same time, biological and social processes allow their species to adapt over a time frame of many generations. One interesting question regarding evolutionary systems is "What is the relationship between learning during the lifetime of a single individual, and the longer time frame species-level learning afforded by evolution?"

9.6.1 Lamarckian Evolution

Lamarck was a scientist who, in the late nineteenth century, proposed that evolution over many generations was directly influenced by the experiences of individual organisms during their lifetime. In particular, he proposed that experiences of a single organism directly affected the genetic makeup of their offspring: If an individual learned during its lifetime to avoid some toxic food, it could pass this trait on genetically to its offspring, which therefore would not need to learn the trait. This is an attractive conjecture, because it would presumably allow for more efficient evolutionary progress than a generate-and-test process (like that of GAs and GPs) that ignores the experience gained during an individual's lifetime. Despite the attractiveness of this theory, current scientific evidence overwhelmingly contradicts Lamarck's model. The currently accepted view is that the genetic makeup of an individual is, in fact, unaffected by the lifetime experience of one's biological parents. Despite this apparent biological fact, recent computer studies have shown that Lamarckian processes can sometimes improve the effectiveness of computerized genetic algorithms (see Grefenstette 1991; Ackley and Littman 1994; and Hart and Belew 1995).