set loose in a domain and allowed to act as parallel asynchronous cooperating agents, we sometimes witness the evolution of seemingly independent "life forms."

As another example, Rodney Brooks (1986, 1987) and his students have designed and built simple robots that interact as autonomous agents solving problems in a laboratory situation. There is no central control algorithm; rather cooperation emerges as an artifact of the distributed and autonomous interactions of individuals. The a-life community has regular conferences and journals reflecting their work (Langton 1995).

In Section 12.1 we introduce evolutionary or biology-based models with *genetic algorithms* (Holland 1975), an approach to learning that exploits parallelism, mutual interactions, and often a bit-level representation. In Section 12.2 we present *classifier systems* and *genetic programming*, relatively new research areas where techniques from genetic algorithms are applied to more complex representations, such as to build and refine sets of production rules (Holland et al. 1986) and to create and adapt computer programs (Koza 1992). In Section 12.3 we present *artificial life* (Langton 1995). We begin 12.3 with an introduction to "The Game of Life." We close with an example of emergent behavior from research at the Santa Fe Institute (Crutchfield and Mitchell 1995).

Chapter 13 presents stochastic and dynamic forms of machine learning.

12.1 The Genetic Algorithm

Like neural networks, genetic algorithms are based on a biological metaphor: They view learning as a competition among a population of evolving candidate problem solutions. A "fitness" function evaluates each solution to decide whether it will contribute to the next generation of solutions. Then, through operations analogous to gene transfer in sexual reproduction, the algorithm creates a new population of candidate solutions.

Let P(t) define a population of candidate solutions, x_i^t, at time t:

$$P(t) = \{x_1^t, x_2^t, \ldots x_n^t\}$$

We now present a general form of the genetic algorithm:

```
procedure genetic algorithm;

    begin
        set time t:= 0;
        initialize the population P(t);
        while the termination condition is not met do
            begin
                evaluate fitness of each member of the population P(t);
                select members from population P(t) based on fitness;
                produce the offspring of these pairs using genetic operators;
                replace, based on fitness, candidates of P(t), with these offspring;
                set time t := t +1
            end
    end.
```

This algorithm articulates the basic framework of genetic learning; specific implementations of the algorithm instantiate that framework in different ways. What percentage of the population is retained? What percentage mate and produce offspring? How often and to whom are the genetic operators applied? The procedure "replace the weakest candidates of $P(t)$" may be implemented in a simple fashion, by eliminating a fixed percentage of the weakest candidates. More sophisticated approaches may order a population by fitness and then associate a probability measure for elimination with each member, where the probability of elimination is an inverse function of its fitness. Then the replacement algorithm uses this measure as a factor in selecting candidates to eliminate. Although the probability of elimination would be very low for the fittest members of the society, there is a chance that even the best individuals could be removed. The advantage of this scheme is that it may save some individuals whose overall fitness is poor but that include some component that may contribute to a more powerful solution. This replacement algorithm has many names, including *Monte Carlo, fitness proportionate selection,* and *roulette wheel*.

Although the examples of Section 12.1.3 introduce more complex representations, we will introduce the representation issues related to genetic algorithms using simple bit strings to represent problem solutions. For example, suppose we want a genetic algorithm to learn to classify strings of 1s and 0s. We can represent a population of bit strings as a pattern of 1s, 0s, and #s, where # is a "don't care," that may match with either 0 or 1. Thus, the pattern 1##00##1 represents all strings of eight bits that begin and end with 1 and that have two 0s in the middle.

The genetic algorithm initializes $P(0)$ to a population of candidate patterns. Typically, initial populations are selected randomly. Evaluation of candidate solutions assumes a fitness function, $f(x_i^t)$ that returns a measure of the candidate's fitness at time t. A common measure of a candidate's fitness tests it on a set of training instances and returns the percentage of correct classifications. Using such a fitness function, an evaluation assigns each candidate solution the value:

$$f(x_i^t)/m(P, t)$$

where m(P,t) is the average fitness over all members of the population. It is also common for the fitness measure to change across time periods, thus fitness could be a function of the stage of the overall problem solution, or $f(x_i^t)$.

After evaluating each candidate, the algorithm selects pairs for recombination. Recombination uses *genetic operators* to produce new solutions that combine components of their parents. As with natural evolution, the fitness of a candidate determines the extent to which it reproduces, with those candidates having the highest evaluations being given a greater probability of reproducing. As just noted, selection is often probabilistic, where weaker members are given a smaller likelihood of reproducing, but are not eliminated outright. That some less fit candidates survive is important since they can still contain some essential component of a solution, for instance part of a bit pattern, and reproduction may extract this component.

There are a number of genetic operators that produce offspring having features of their parents; the most common of these is *crossover*. Crossover takes two candidate solutions and divides them, swapping components to produce two new candidates. Figure 12.1 illustrates crossover on bit string patterns of length 8. The operator splits them in the

middle and forms two children whose initial segment comes from one parent and whose tail comes from the other. Note that splitting the candidate solution in the middle is an arbitrary choice. This split may be at any point in the representation, and indeed, this splitting point may be randomly adjusted or changed during the solution process.

For example, suppose the target class is the set of all strings beginning and ending with a 1. Both the parent strings in Figure 12.1 would have performed relatively well on this task. However, the first offspring would be much better than either parent: it would not have any false positives and would fail to recognize fewer strings that were actually in the solution class. Note also that its sibling is worse than either parent and will probably be eliminated over the next few generations.

Mutation is another important genetic operator. Mutation takes a single candidate and randomly changes some aspect of it. For example, mutation may randomly select a bit in the pattern and change it, switching a 1 to a 0 or #. Mutation is important in that the initial population may exclude an essential component of a solution. In our example, if no member of the initial population has a 1 in the first position, then crossover, because it preserves the first four bits of the parent to be the first four bits of the child, cannot produce an offspring that does. Mutation would be needed to change the values of these bits. Other genetic operators, for example *inversion*, could also accomplish this task, and are described in Section 12.1.3.

The genetic algorithm continues until some termination requirement is met, such as having one or more candidate solutions whose fitness exceeds some threshold. In the next section we give examples of genetic algorithm encodings, operators, and fitness evaluations for two situations: the CNF constraint satisfaction and the traveling salesperson problems.

12.1.3 Two Examples: CNF Satisfaction and the Traveling Salesperson

We next select two problems and discuss representation issues and fitness functions appropriate for their solutions. Three things should be noted: first, all problems are not easily or naturally encoded as bit level representations. Second, the genetic operators must preserve crucial relationships within the population, for example, the presence and uniqueness of all the cities in the traveling salesperson tour. Finally, we discuss an important relationship between the fitness function(s) for the states of a problem and the encoding of that problem.

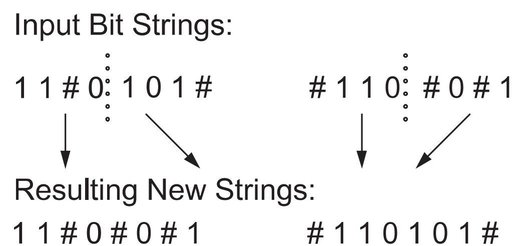

Figure 12.1 Use of crossover on two bit strings of length eight. # is don t care.

EXAMPLE 12.2.1: THE CNF-SATISFACTION PROBLEM

The conjunctive normal form (CNF) satisfiability problem is straightforward: an expression of propositions is in conjunctive normal form when it is a sequence of clauses joined by an and ($\wedge$) relation. Each of these clauses is in the form of a disjunction, the or ($\vee$), of literals. For example, if the literals are, a, b, c, d, e, and f, then the expression

$$(\neg a \vee c) \wedge (\neg a \vee c \vee \neg e) \wedge (\neg b \vee c \vee d \vee \neg e) \wedge (a \vee \neg b \vee c) \wedge (\neg e \vee f)$$

is in CNF. This expression is the conjunction of five clauses, each clause is the disjunction of two or more literals. We introduced propositions and their satisfaction in Chapter 2. We discussed the CNF form of propositional expressions, and offered a method of reducing expressions to CNF, when we presented resolution inferencing in Section 14.2.

CNF satisfiability means that we must find an assignment of true or false (1 or 0) to each of the six literals, so that the CNF expression evaluates to true. The reader should confirm that one solution for the CNF expression is to assign false to each of a, b, and e. Another solution has e false and c true.

A natural representation for the CNF satisfaction problem is a sequence of six bits, each bit, in order, representing true (1) or false (0) for each of the six literals, again in the order of a, b, c, d, e, and f. Thus:

1 0 1 0 1 0

indicates that a, c, and e are true and b, d, and f are false, and the example CNF expression is therefore false. The reader can explore the results of other truth assignments to the literals of the expression.

We require that the actions of each genetic operator produce offspring that are truth assignments for the CNF expression, thus each operator must produce a six-bit pattern of truth assignments. An important result of our choice of the bit pattern representation for the truth values of the literals of the CNF expression is that any of the genetic operators discussed to this point will leave the resulting bit pattern a legitimate possible solution. That is, crossover and mutation leave the resulting bit string a possible solution of the problem. Even other less frequently used genetic operators, such as *inversion* (reversing the order of the bits within the six-bit pattern) or *exchange* (interchanging two different bits in the pattern) leave the resulting bit pattern a legitimate possible solution of the CNF problem. In fact, from this viewpoint, it is hard to imagine a better suited representation than a bit pattern for the CNF satisfaction problem.

The choice of a fitness function for this population of bit strings is not quite as straightforward. From one viewpoint, either an assignment of truth values to literals will make the expression true or else the expression will be false. If a specific assignment makes the expression true, then the solution is found; otherwise it is not. At first glance it seems difficult to determine a fitness function that can judge the "quality" of bit strings as potential solutions.

There are a number of alternatives, however. One would be to note that the full CNF expression is made up of the conjunction of five clauses. Thus we can make up a rating

system that will allow us to rank potential bit pattern solutions in a range of 0 to 5, depending on the number of clauses that pattern satisfies. Thus the pattern:

1 1 0 0 1 0 has fitness 1,
0 1 0 0 1 0 has fitness 2,
0 1 0 0 1 1 has fitness 3, and
1 0 1 0 1 1 has fitness 5, and is a solution.

This genetic algorithm offers a reasonable approach to the CNF satisfaction problem. One of its most important properties is the use of the implicit parallelism afforded by the population of solutions. The genetic operators have a natural fit to this representation. Finally, the solution search seems to fit naturally a parallel "divide and conquer" strategy, as fitness is judged by the number of problem components that are satisfied. In the chapter exercises the reader is encouraged to consider other aspects of this problem.

EXAMPLE 12.2.2: THE TRAVELING SALESPERSON PROBLEM

The traveling salesperson problem (TSP) is classic to AI and computer science. We introduced it with our discussion of graphs in Section 3.1. Its full state space requires the consideration of N! states where N is the number of cities to be visited. It has been shown to be NP-hard, with many researchers proposing heuristic approaches for its solution. The statement of the problem is simple:

> A salesperson is required to visit N cities as part of a sales route. There is a cost (e.g., mileage, air fare) associated with each pair of cities on the route. Find the least cost path for the salesperson to start at one city, visit all the other cities exactly once and return home.

The TSP has some very nice applications, including circuit board drilling, X-ray crystallography, and routing in VLSI fabrication. Some of these problems require visiting tens of thousands of points (cities) with a minimum cost path. One very interesting question in the analysis of the TSP class of problems is whether it is worth running an expensive workstation for many hours to get a near optimal solution or run a cheap computer for a few minutes to get "good enough" results for these applications. TSP is an interesting and difficult problem with many ramifications of search strategies.

How might we use a genetic algorithm to solve this problem? First, the choice of a representation for the path of cities visited, as well as the creation of a set of genetic operators for this path, is not trivial. The design of a fitness function, however, is very straightforward: all we need do is evaluate the path length cost. We could then order the paths by their cost, the cheaper the better.

Let's consider some obvious representations that turn out to have complex ramifications. Suppose we have nine cities to visit,1, 2,..., 9, so we make the representation of a path the ordered listing of these nine integers. Suppose we simply make each city a four-bit pattern, 0001, 0010, . . . 1001. Thus, the pattern:

0001 0010 0011 0100 0101 0110 0111 1000 1001

represents a visit to each city in the order of its numbering. We have inserted blanks into the string only to make it easier to read. Now, what about the genetic operators? Crossover is definitely out, since the new string produced from two different parents would most probably not represent a path that visits each city exactly once. In fact, with crossover, some cities could be removed while others are visited more than once. What about mutation? Suppose the leftmost bit of the sixth city, 0110, is mutated to 1? 1110, or 14, is no longer a legitimate city. Inversion, and the swapping of cities (the four bits in the city pattern) within the path expression would be acceptable genetic operators, but would these be powerful enough to obtain a satisfactory solution? In fact, one way to look at the search for the minimum path would be to generate and evaluate all possible permutations of the N elements of the city list. The genetic operators must be able to produce all permutations.

Another approach to the TSP would be to ignore the bit pattern representation and give each city an alphabetic or numeric name, e.g., 1, 2, ..., 9; make the path through the cities an ordering of these nine digits, and then select appropriate genetic operators for producing new paths. Mutation, as long as it was a random exchange of two cities in the path, would be okay, but the crossover operator between two paths would be useless. The exchange of pieces of a path with other pieces of the same path, or any operator that shuffled the letters of the path (without removing, adding, or duplicating any cities) would work. These approaches, however, make it difficult to combine into offspring the "better" elements of patterns within the paths of cities of the two different parents.

A number of researchers (Davis 1985, Oliver et al. 1987) have created crossover operators that overcome these problems and let us work with the ordered list of cities visited. For example, Davis has defined an operator called *order crossover*. Suppose we have nine cities, 1, 2, ..., 9, and the order of the integers represents the order of visited cities.

Order crossover builds offspring by choosing a subsequence of cities within the path of one parent. It also preserves the relative ordering of cities from the other parent. First, select two cut points, indicated by a "|", which are randomly inserted into the same location of each parent. The locations of the cut points are random, but once selected, the same locations are used for both parents. For example, for two parents p1 and p2, with cut points after the third and seventh cities:

p1 = (1 9 2 | 4 6 5 7 | 8 3)
p2 = (4 5 9 | 1 8 7 6 | 2 3)

two children c1 and c2 are produced in the following way. First, the segments between cut points are copied into the offspring:

c1 = (x x x | 4 6 5 7 | x x)
c2 = (x x x | 1 8 7 6 | x x)

Next, starting from the second cut point of one parent, the cities from the other parent are copied in the same order, omitting cities already present. When the end of the string is reached, continue on from the beginning. Thus, the sequence of cities from p2 is:

2 3 4 5 9 1 8 7 6

Once cities 4, 6, 5, and 7 are removed, since they are already part of the first child, we get the shortened list 2, 3, 9, 1, and 8, which then makes up, preserving the ordering found in p2, the remaining cities to be visited by c1:

c1 = (2 3 9 | 4 6 5 7 | 1 8)

In a similar manner we can create the second child c2:

c2 = (3 9 2 | 1 8 7 6 | 4 5)

To summarize, in order crossover, pieces of a path are passed on from one parent, p1, to a child, c1, while the ordering of the remaining cities of the child c1 is inherited from the other parent, p2. This supports the obvious intuition that the ordering of cities will be important in generating the least costly path, and it is therefore crucial that pieces of this ordering information be passed on from fit parents to children.

The order crossover algorithm also guarantees that the children would be legitimate tours, visiting all cities exactly once. If we wished to add a mutation operator to this result we would have to be careful, as noted earlier, to make it an exchange of cities within the path. The inversion operator, simply reversing the order of all the cities in the tour, would not work (there is no new path when all cities are inverted). However, if a piece within the path is cut out and inverted and then replaced, it would be an acceptable use of inversion. For example, using the cut | indicator as before, the path:

c1 = (2 3 9 | 4 6 5 7 | 1 8),

becomes under inversion of the middle section,

c1 = (2 3 9 | 7 5 6 4 | 1 8)

A new mutation operator could be defined that randomly selected a city and placed it in a new randomly selected location in the path. This mutation operator could also operate on a piece of the path, for example, to take a subpath of three cities and place them in the same order in a new location within the path. Other suggestions are in the exercises.

12.1.4 Evaluating the Genetic Algorithm

The preceding examples highlight genetic algorithm's unique problems of knowledge representation, operator selection, and the design of a fitness function. The representation selected must support the genetic operators. Sometimes, as with the CNF satisfaction problem, the bit level representation is natural. In this situation, the traditional genetic operators of crossover and mutation could be used directly to produce potential solutions. The traveling salesperson problem was an entirely different matter. First, there did not seem to be any natural bit level representations for this problem. Secondly, new mutation and crossover operators had to be devised that preserved the property that the offspring

had to be legal paths through all the cities, visiting each only once.

Finally, genetic operators must pass on "meaningful" pieces of potential solution information to the next generation. If this information, as in CNF satisfiability, is a truth value assignment, then the genetic operators must preserve it in the next generation. In the TSP problem, path organization was critical, so as we discussed, components of this path information must be passed on to descendants. This successful transfer rests both in the representation selected as well as in the genetic operators designed for each problem.

We leave representation with one final issue, the problem of the "naturalness" of a selected representation. Suppose, as a simple, if somewhat artificial, example, we want our genetic operators to differentiate between the numbers 6, 7, 8, and 9. An integer representation gives a very natural and evenly spaced ordering, because, within base ten integers, the next item is simply one more than the previous. With change to binary, however, this naturalness disappears. Consider the bit patterns for 6, 7, 8, and 9:

0110 0111 1000 1001

Observe that between 6 and 7 as well as between 8 and 9 there is a 1 bit change. Between 7 and 8, however, all four bits change! This representational anomaly can be huge in trying to generate a solution that requires any organizing of these four bit patterns. A number of techniques, usually under the general heading of *gray coding,* address this problem of non-uniform representation. For instance, a gray coded version of the first sixteen binary numbers may be found in Table 12.1. Note that each number is exactly one bit different from its neighbors. Using gray coding instead of standard binary numbers, the genetic operator's transitions between states of near neighbors is natural and smooth.

Binary	Gray
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

Table 12.1 The gray coded bit patterns for the binary numbers 0, 1, . . ., 15.

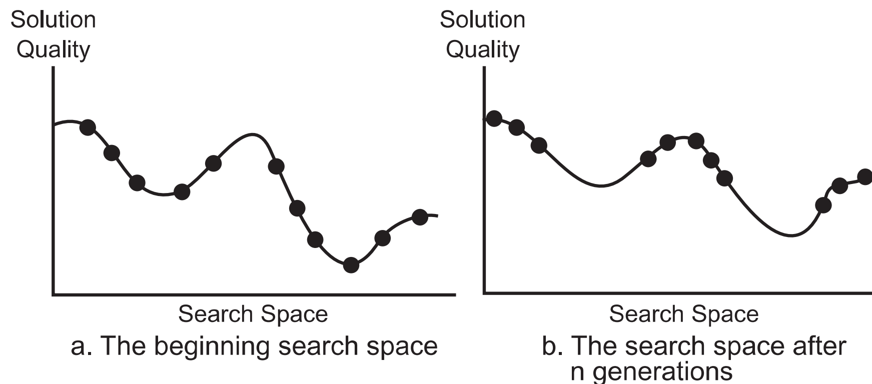

Figure 12.2 Genetic algorithms visualized as parallel hill climbing, adapted from Holland (1986).

An important strength of the genetic algorithm is in the parallel nature of its search. Genetic algorithms implement a powerful form of hill climbing that maintains multiple solutions, eliminates the unpromising, and improves good solutions. Figure 12.2, adapted from Holland (1986), shows multiple solutions converging toward optimal points in a search space. In this figure, the horizontal axis represents the possible points in a solution space, while the vertical axis reflects the quality of those solutions. The dots on the curve are members of the genetic algorithm's current population of candidate solutions. Initially, the solutions are scattered through the space of possible solutions. After several generations, they tend to cluster around areas of higher solution quality.

When we describe our genetic search as "hill climbing" we implicitly acknowledge moving across a "fitness landscape." This landscape will have its valleys, peaks, with local maxima and minima. In fact, some of the discontinuities in the space will be artifacts of the representation and genetic operators selected for the problem. This discontinuity, for example, could be caused by a lack of gray coding, as just discussed. Note also that genetic algorithms, unlike sequential forms of hill climbing, as in Section 4.1, do not immediately discard unpromising solutions. Through genetic operators, even weak solutions may continue to contribute to the makeup of future candidate solutions.

Another difference between genetic algorithms and the state space heuristics presented in Chapter 4 is the analysis of the present-state/goal-state difference. The information content supporting the A* algorithm, as in Section 4.2, required an estimate of "effort" to move between the present state and a goal state. No such measure is required with genetic algorithms, simply some measure of fitness of each of the current generation of potential solutions. There is also no strict ordering required of next states on an open list as we saw in state space search; rather, there is simply a population of fit solutions to a problem, each potentially available to help produce new possible solutions within a paradigm of parallel search.

An important source of the genetic algorithm's power is the *implicit parallelism* inherent in evolutionary operators. In comparison with state space search and an ordered open list, search moves in parallel, operating on entire families of potential solutions. By

restricting the reproduction of weaker candidates, genetic algorithms may not only eliminate that solution, but all of its descendants. For example, the string, 101#0##1, if broken at its midpoint, can parent a whole family of strings of the form 101#____. If the parent is found to be unfit, its elimination can also remove all of these potential offspring and, perhaps, the possibility of a solution as well.

As genetic algorithms are more widely used in applied problem solving as well as in scientific modeling, there is increasing interest in attempts to understand their theoretical foundations. Several questions that naturally arise are:

1. Can we characterize types of problems for which GAs will perform well?

2. For what problem types do they perform poorly?

3. What does it even "mean" for a GA to perform well or poorly for a problem type?

4. Are there any laws that can describe the macrolevel of behavior of GAs? In particular, are there any predictions that can be made about the changes in fitness of subgroups of the population over time?

5. Is there any way to describe the differential effects of different genetic operators, crossover, mutation, inversion, etc., over time?

6. Under what circumstances (what problems and what genetic operators) will GAs perform better than traditional AI search methods?

Addressing many of these issues goes well beyond the scope of our book. In fact, as Mitchell (1996) points out, there are still more open questions at the foundations of genetic algorithms than there are generally accepted answers. Nonetheless, from the beginning of work in GAs, researchers, including Holland (1975), have attempted to understand how GAs work. Although they address issues on the macro level, such as the six questions just asked, their analysis begins with the micro or bit level representation.

Holland (1975) introduced the notion of a *schema* as a general pattern and a "building block" for solutions. A schema is a pattern of bit strings that is described by a template made up of 1, 0, and # (don't care). For example, the schema 1 0 # # 0 1, represents the family of six-bit strings beginning with a 1 0 and ending with a 0 1. Since, the middle pattern # # describes four bit patterns, 0 0, 0 1, 1 0, 1 1, the entire schema represents four patterns of six 1s and 0s. Traditionally, each schema is said to describe a hyperplane (Goldberg 1989); in this example, the hyperplane cuts the set of all possible six-bit representations. A central tenet of traditional GA theory is that schemata are the building blocks of families of solutions. The genetic operators of crossover and mutation are said to manipulate these schemata towards potential solutions. The specification describing this manipulation is called the *schema theorem* (Holland 1975, Goldberg 1989). According to Holland, an adaptive system must identify, test, and incorporate structural properties hypothesized to give better performance in some environment. Schemata are meant to be a formalization of these structural properties.

Holland's schema analysis suggests that the fitness selection algorithm increasingly

focuses the search on subsets of the search space with estimated best fitness; that is, the subsets are described by schemas of above average fitness. The genetic operator crossover puts high fitness building blocks together in the same string in an attempt to create ever more fit strings. Mutation helps guarantee that (genetic) diversity is never removed from the search; that is, that we continue to explore new parts of the fitness landscape. The genetic algorithm can thus be seen as a tension between opening up a general search process and capturing and preserving important (genetic) features in that search space. Although Holland's original analysis of GA search focused at the bit level, more recent work has extended this analysis to alternate representational schemes (Goldberg 1989). In the next section we apply GA techniques to more complex representations.

12.2 Classifier Systems and Genetic Programming

Early research in genetic algorithms focused almost exclusively on low-level representations, such as strings of {0, 1, #}. In addition to supporting straightforward instantiations of genetic operators, bit strings and similar representations give genetic algorithms much of the power of other subsymbolic approaches, such as connectionist networks. There are problems, however, such as the traveling salesperson, that have a more natural encoding at a more complex representational level. We can further ask whether genetic algorithms can be defined for still richer representations, such as *if... then...* rules or pieces of computer code. An important aspect of such representations is their ability to combine distinct, higher level knowledge sources through rule chaining or function calls to meet the requirements of a specific problem instance.

Unfortunately, it is difficult to define genetic operators that capture the syntactic and semantic structure of logical relationships while enabling effective application of operators such as crossover or mutation. One possible way to marry the reasoning power of rules with genetic learning is to translate logical sentences into bit strings and use the standard crossover operator. Unfortunately, under many translations most of the bit strings produced by crossover and mutation will fail to correspond to meaningful logical sentences. As an alternative to representing problem solutions as bit strings, we may define variations of crossover that can be applied directly to higher level representations such as *if... then...* rules or chunks of code in a higher level programming language. This section discusses examples of each approach to extending the power of genetic algorithms.

12.2.1 Classifier Systems

Holland (1986) developed a problem-solving architecture called *classifier systems* that applies genetic learning to rules in a production system. A classifier system (Figure 12.3) includes the familiar elements of a production system: production rules (here called classifiers), working memory, input sensors (or decoders), and outputs (or effectors). Unusual features of a classifier system include the use of competitive bidding for conflict resolution, genetic algorithms for learning, and the *bucket brigade algorithm* to assign credit and blame to rules during learning. Feedback from the outside environment