12 MACHINE LEARNING: GENETIC AND EMERGENT

What limit can we put to this power, acting during long ages and rigidly scrutinizing the whole constitution, structure and habits of each creature – favoring the good and rejecting the bad? I can see no limit to this power in slowly and beautifully adapting each form to the most complex relations of life.

—CHARLES DARWIN, *On the Origin of Species*

The First Law of Prophecy:
When a distinguished but elderly scientist states that something is possible, he is almost certainly right. When he states that something is impossible, he is very probably wrong.

The Second Law:
The only way of discovering the limits of the possible is to venture a little way past them into the impossible.

The Third Law:
Any sufficiently advanced technology is indistinguishable from magic.

—ARTHUR C. CLARKE, *Profiles of the Future*

12.0 Social and Emergent Models of Learning

Just as connectionist networks received much of their early support and inspiration from the goal of creating an artificial neural system, so also have a number of other biological analogies influenced the design of machine learning algorithms. This chapter considers learning algorithms patterned after the processes underlying evolution: shaping a population of individuals through the survival of its most fit members. The power of selection across a population of varying individuals has been demonstrated in the emergence of species in natural evolution, as well as through the social processes underlying cultural change. It has also been formalized through research in cellular automata, genetic

algorithms, genetic programming, artificial life, and other forms of emergent computation.

Emergent models of learning simulate nature's most elegant and powerful form of adaptation: the evolution of plant and animal life forms. Charles Darwin saw "...no limit to this power of slowly and beautifully adapting each form to the most complex relations of life...". Through this simple process of introducing variations into successive generations and selectively eliminating less fit individuals, adaptations of increasing capability and diversity *emerge* in a population. Evolution and emergence occur in populations of *embodied* individuals, whose actions affect others and that, in turn, are affected by others. Thus, selective pressures come not only from the outside environment, but also from interactions between members of a population. An ecosystem has many members, each with roles and skills appropriate to their own survival, but more importantly, whose cumulative behavior shapes and is shaped by the rest of the population.

Because of their simplicity, the processes underlying evolution have proven remarkably general. Biological evolution produces species by selecting among changes in the genome. Similarly, cultural evolution produces knowledge by operating on socially transmitted and modified units of information. Genetic algorithms and other formal evolutionary analogs produce increasingly capable problem solutions by operating on populations of candidate problem solutions.

When the genetic algorithm is used for problem solving, it has three distinct stages: first, the individual potential solutions of the problem domain are encoded into representations that support the necessary variation and selection operations; often, these representations are as simple as bit strings. In the second stage, mating and mutation algorithms, analogous to the sexual activity of biological life forms, produce a new generation of individuals that recombine features of their parents. Finally, a *fitness* function judges which individuals are the "best" life forms, that is, most appropriate for the eventual solution of the problem. These individuals are favored in survival and reproduction, shaping the next generation of potential solutions. Eventually, a generation of individuals will be interpreted back to the original problem domain as solutions for the problem.

Genetic algorithms are also applied to more complex representations, including production rules, to evolve rule sets adapted to interacting with an environment. For example, genetic programming combines and mutates fragments of computer code in an attempt to evolve a program for solving problems such as capturing the invariants in sets of data.

An example of learning as social interaction leading to survival can be found in games such as *The Game of Life*, originally created by the mathematician John Horton Conway and introduced to the larger community by Martin Gardner in *Scientific American* (1970, 1971). In this game, the birth, survival, or death of individuals is a function of their own state and that of their near neighbors. Typically, a small number of rules, usually three or four, are sufficient to define the game. In spite of this simplicity, experiments with the game of life have shown it to be capable of evolving structures of extraordinary complexity and ability, including self replicating, multi-cellular "organisms" (Poundstone 1985).

An important approach for *artificial life*, or *a-life*, is to simulate the conditions of biological evolution through the interactions of finite state machines, complete with sets of states and transition rules. These automata are able to accept information from outside themselves, in particular, from their closest neighbors. Their transition rules include instructions for birth, continuing in life, and dying. When a population of such automata is

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.

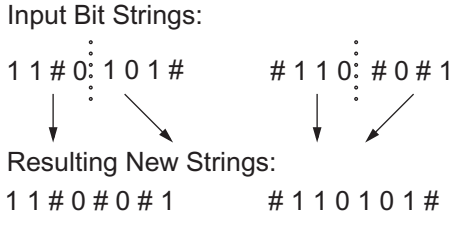

Figure 12.1 Use of crossover on two bit strings of length eight. # is don t care.

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.

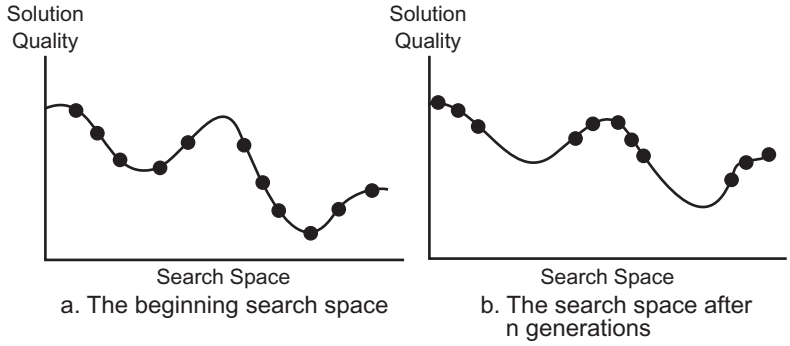

Figure 12.2 Genetic algorithms visualized as parallel hill climbing, adapted from Holland (1986).

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

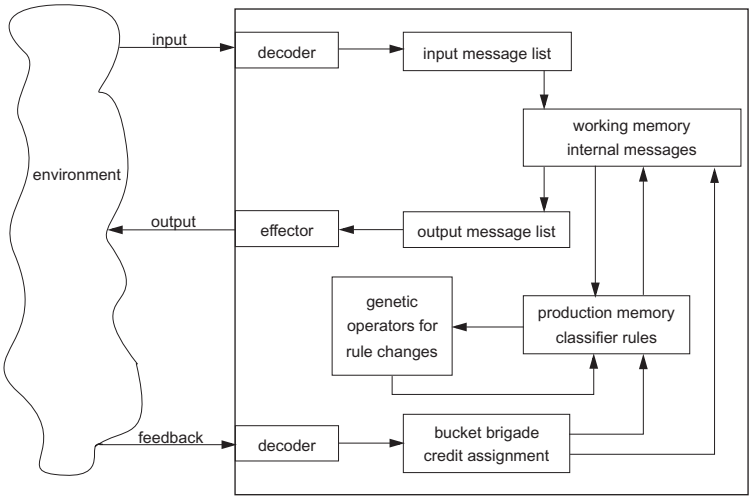

Figure 12.3 A classifier system interacting with the environment, adapted from Holland (1986).

provides a means of evaluating the fitness of candidate classifiers, as required in genetic learning. The classifier system of Figure 12.3 has the following major components:

1. Detectors of input messages from the environment.
2. Detectors of feedback messages from the environment.
3. Effectors translating results of rule applications back to the environment.
4. A production rule set made up of a population of classifiers. Each classifier has an associated fitness measure.
5. A working memory for the classifier rules. This memory integrates the results of production rule firing with input information.
6. A set of genetic operators for production rule modification.
7. A system for giving credit to rules involved in producing successful actions.

In problem solving, the classifier performs as a traditional production system. The environment sends a message, perhaps a move in a game, to the classifier system's detectors. This event is decoded and placed as a pattern on the internal message list, the working memory for the production system. These messages, in the normal action of data-driven production system, match the condition patterns of the classifier rules. The selection of the "strongest activated classifiers" is determined by a bidding scheme, where

a bid is a function of both the accumulated fitness of the classifier and the quality of the match between the input stimulus and its condition pattern. The classifiers with the closest match add messages (the action of the fired rules) to working memory. The revised message list may send messages to the effectors which act upon the environment or activate new classifier rules as the production system processing continues.

Classifier systems implement a form of reinforcement learning, Section 10.7. Based on feedback from a teacher or fitness evaluation function, the learner computes the fitness of a population of candidate rules and adapts this population using a variation of genetic learning. Classifier systems learn in two ways. First, there is a reward system that adjusts the fitness measures of the classifier rules, rewarding successful rule firings and penalizing errors. The credit assignment algorithm passes part of the reward or penalty back to any classifier rules that have contributed to the final rule firing. This distribution of differential rewards across interacting classifiers, as well as those that enabled their firing, is often implemented in a *bucket brigade* algorithm. The bucket brigade algorithm addresses the problem of assigning credit or blame in situations where the system's output may be the product of a sequence of rule firings. In the event of an error, how do we know which rule to blame? Is the responsibility that of the last rule to fire, or of some previous rule that provided it with faulty information? The bucket brigade algorithm allocates both credit and blame across a sequence of rule applications according to measures of each rule's contribution to the final conclusion. (An analogous assignment of blame for error was described with the backpropagation algorithm of Section 11.3; see Holland (1986) for more details.)

The second form of learning modifies the rules themselves using genetic operators such as mutation and crossover. This allows the most successful rules to survive and combine to make new classifiers, while unsuccessful rule classifiers disappear.

Each classifier rule consists of three components: the rule's condition matches data in the working memory in the typical production system sense. In learning, genetic operators can modify both the conditions and the actions of the production rules. The second component of the rule, the action, can have the effect of changing the internal message list (the production memory). Finally, each rule has a fitness measure. This parameter is changed, as just noted, both by successful as well as by unsuccessful activity. This measure is originally assigned to each rule on its creation by the genetic operators; for example, it may be set as the average fitness of its two parents.

A simple example illustrates the interactions of these components of a classifier system. Assume that a set of objects to be classified are defined by six attributes (conditions c1, c2, ..., c6), and further suppose that each of these attributes can have five different values. Although the possible values of each attribute are of course different (for example, the value of c3 might be color, while c5 might describe the weather) we will, without loss of generality, give each attribute an integer value from {1, 2, ..., 5}. Suppose the conditions of these rules place their matching object in one of four classes: A1, A2, A3, A4.

Based on these constraints, each classifier will have the form:

(c1 c2 c3 c4 c5 c6) $\rightarrow$ Ai, where i = 1, 2, 3, 4.

where each ci in the condition pattern denotes the value {1, 2, ..., 5} of the ith attribute of the condition. Usually, conditions can also assign a value of # or "don't care" to an

attribute. Ai denotes the classification, A1, A2, A3, or A4. Table 12.2 presents a set of classifiers. Note that different condition patterns can have the same classification, as in rules 1 and 2, or the same patterns, as in rules 3 and 5, can lead to different classifications.

Condition (Attributes)	Action (Classification)	Rule Number
(1 # # # 1 #)	→ A1	1
(2 # # 3 # #)	→ A1	2
(# # 4 3 # #)	→ A2	3
(1 # # # # #)	→ A2	4
(# # 4 3 # #)	→ A3	5
etc.		

Table 12.2 A set of condition → action classifiers to be learned.

As described so far, a classifier system is simply another form of the ubiquitous production system. The only really novel feature of classifier rules in this example is their use of strings of digits and #s to represent condition patterns. It is this representation of conditions that constrains the application of genetic algorithms to the rules. The remainder of the discussion describes genetic learning in classifier systems.

In order to simplify the remainder of the example, we will only consider the classifier system's performance in learning the classification A1. That is, we will ignore the other classifications, and assign condition patterns a value of 1 or 0 depending on whether or not they support classification A1. Note that there is no loss of generality in this simplification; it may be extended to problems of learning more than one classification by using a vector to indicate the classifications that match a particular condition pattern. For example, the classifiers of Table 12.2 may be summarized by:

(1 # # # 1 #) → (1 0 0 0)
(2 # # 3 # #) → (1 0 0 0)
(1 # # # # #) → (0 1 0 0)
(# # 4 3 # #) → (0 1 1 0)

In this example, the last of these summaries indicates that the condition attributes support classification rules A2 and A3 and not A1 or A4. By replacing the 0 or 1 assignment with these vectors, the learning algorithm can evaluate the performance of a rule across multiple classifications.

In this example, we will use the rules in Table 12.2 to indicate the correct classifications; essentially, they will function as teachers or evaluators of the fitness of rules in the learning system. As with most genetic learners, we begin with a random population of rules. Each condition pattern is also assigned a *strength*, or *fitness*, parameter (a real number between 0.0, no strength, and 1.0, full strength. This strength parameter, s, is computed from the fitness of each rule's parents, and measures its historical fitness.

At each learning cycle, the rules attempt to classify the inputs and are then ranked by the teacher or fitness metric. For example, assume that at some cycle, the classifier has the

following population of candidate classification rules, where the conclusion of 1 indicates that the pattern led to a correct classifcation and 0 that it did not:

(# # # 2 1 #) → 1 s = 0.6
(# # 3 # # 5) → 0 s = 0.5
(2 1 # # # #) → 1 s = 0.4
(# 4 # # # 2) → 0 s = 0.23

Suppose a new input message arrives from the environment: (1 4 3 2 1 5), and the teacher (using the first rule of Table 12.2) classifies this input vector as a positive example of A1. Let's consider what happens when working memory receives this pattern and the four candidate classifier rules try to match it. Rules 1 and 2 match. Conflict resolution is done through competitive bidding among matching rules. In our example, bidding is a function of the sum of the matches of the attribute values times the strength measure of the rule. "Don't care" matches have the value 0.5, while exact matches have value 1.0. To normalize we divide this result by the length of the input vector. Since the input vector matches the first classifier with two exact and four "don't cares," its bid is ((4 * 0.5 + 2 * 1) * 0.6) / 6, or 0.4. The second classifier also matches two attributes and has four "don't cares," so its bid is 0.33. In our example, only the classifier making the highest bid fires, but in more complex situations, it may be desirable for a percentage of the bids to be accepted.

The first rule wins and posts its action, a 1, indicating that this pattern is an example of A1. Since this action is correct, the fitness measure of rule 1 is increased to between its present value and 1.0. Had the action of this rule been incorrect, the fitness measure would have been lowered. If the system required multiple firings of the rule set to produce some result on the environment, all the rules responsible for this result would receive some proportion of the reward. The exact procedure by which the rule's fitness is calculated varies across systems and may be fairly complex, involving the use of the bucket brigade algorithm or some similar credit assignment technique. See Holland (1986) for details.

Once the fitness of the candidate rules has been computed, the learning algorithm applies genetic operators to create the next generation of rules. First, a selection algorithm will decide the most fit members of the rule set. This selection is based on the fitness measure, but may also include an additional random value. The random value gives rules with a poor fitness the opportunity to reproduce, helping to avoid a too hasty elimination of rules that, while performing poorly overall, may incorporate some element of the desired solution. Suppose the first two classifier rules of the example are selected to survive and reproduce. Randomly selecting a crossover position between the fourth and fifth elements,

(# # # 2 | 1 #) → 1 s = 0.6
(# # 3 # | # 5) → 0 s = 0.5

produces the offspring:

(# # 3 # | 1 #) → 0 s = 0.53
(# # # 2 | # 5) → 1 s = 0.57

The fitness measure of the children is a weighted function of the fitness of the parents. The weighting is a function of where the crossover point lies. The first offspring has 1/3 of the original 0.6 classifier and 2/3 of the original 0.5 classifier. Thus, the first offspring has strength of (1/3 * 0.6) + (2/3 * 0.5) = 0.53. With a similar calculation, the fitness of the second child is 0.57. The result of firing the classifier rule, always 0 or 1, goes with the majority of the attributes, thus preserving the intuition that these patterns are important in the outcomes of the rules. In a typical classifier system these two new rules, along with their parents, would make up the subset of classifiers for the operation of the system at the next time step.

A mutation operator may also be defined. A simple mutation rule would be to randomly change any attribute pattern to some other valid attribute pattern; for example, a 5 could be mutated to 1, 2, 3, 4 or #. Again, as noted in our discussion of GAs, mutation operators are seen as forcing diversity into the search for classifiers, while crossover attempts to preserve and build new children from successful pieces of parental patterns.

Our example was simple and intended primarily for illustrating the main components of the classifier system. In an actual system, more than one rule might fire and each pass their results along to the production memory. There is often a taxation scheme that keeps any classifier from becoming too prominent in the solution process by lowering its fitness each time it wins a bid. We also did not illustrate the bucket brigade algorithm, differentially rewarding rules supporting successful output messages to the environment. Also, the genetic operators do not usually rework the classifiers at every operation of the system. Rather, there is some general parameter for each application that decides, perhaps on analysis of feedback from the environment, when the classifiers should be evaluated and the genetic operators applied.

Finally, our example is taken from the classifier systems that Holland (1986) at the University of Michigan proposed. The Michigan approach can be viewed as a computational model of cognition, where the knowledge, (the classifiers), of a cognitive entity are exposed to a reacting environment and as a result undergo modification over time. We evaluate the success of the entire system over time, while the importance of the individual classifier is minimal. Alternative classifier systems have also been investigated, including work at the University of Pittsburgh (Michalski et al. 1983). The Pittsburgh classifier focuses on the roles of individual rules in producing new generations of classifiers. This approach implements a model of inductive learning proposed by Michalski.

In the next section we consider a different and particularly exciting application for GAs, the evolution of computer programs.

12.2.2 Programming with Genetic Operators

Through the last several subsections we have seen GAs applied to progressively larger representational structures. What began as genetic transformations on bit strings evolved to operations on *if* . . . *then* . . . rules. It can quite naturally be asked if genetic and evolutionary techniques might be applied to the production of other larger scale computational tools. There have been two major examples of this: the generation of computer programs and the evolution of systems of finite state machines.

Koza (1991, 1992, 2005) has suggested that a successful computer program might evolve through successive applications of genetic operators. In genetic programming, the structures being adapted are hierarchically organized segments of computer programs. The learning algorithm maintains a population of candidate programs. The fitness of a program will be measured by its ability to solve a set of tasks, and programs are modified by applying crossover and mutation to program subtrees. Genetic programming searches a space of computer programs of varying size and complexity; in fact, the search space is the space of all possible computer programs composed of functions and terminal symbols appropriate to the problem domain. As with all genetic learners, this search is random, largely blind and yet surprisingly effective.

Genetic programming starts with an initial population of randomly generated programs made up of appropriate program pieces. These pieces, suitable for a problem domain, may consist of standard arithmetic operations, other related programming operations, and mathematical functions, as well as logical and domain-specific functions. Program components include data items of the usual types: boolean, integer, floating point, vector, symbolic, or multiple-valued.

After initialization, thousands of computer programs are genetically bred. The production of new programs comes with application of genetic operators. Crossover, mutation, and other breeding algorithms must be customized for the production of computer programs. We will see several examples shortly. The fitness of each new program is then determined by seeing how well it performs in a particular problem environment. The nature of the fitness measure will vary according to the problem domain. Any program that does well on this fitness task will survive to help produce the children of the next generation.

To summarize, *genetic programming* includes six components, many very similar to the requirements for GAs:

1. A set of structures that undergo transformation by genetic operators.

2. A set of initial structures suited to a problem domain.

3. A fitness measure, again domain dependent, to evaluate structures.

4. A set of genetic operators to transform structures.

5. Parameters and state descriptions that describe members of each generation.

6. A set of termination conditions.

In the following paragraphs we address each of these topics in more detail.

Genetic programming manipulates hierarchically organized program modules. Lisp was (and still remains) the primary representation for the programming language components: Koza represents program segments as Lisp symbol expressions, or *s-expressions*. (See the auxiliary material for a discussion of s-expressions, their natural representation as tree structures, and their evaluation as programs.)

Genetic operators manipulate s-expressions. In particular, operators map tree structures of s-expressions, (Lisp program segments), into new trees, (new Lisp program segments). Although this s-expression is the basis for Koza's early work, other researchers have more applied this approach to different programming languages/paradigms.

Genetic programming will construct useful programs, given that the atomic pieces and evaluable predicates of the problem domain are available. When we set up a domain for the generation of a programs sufficient to address a set of problems, we must first analyze what terminals are required for units in its solution as well as what functions are necessary to produce these terminals. As Koza notes (1992, p.86) "... the user of genetic programming should know ... that some composition of the functions and terminals he supplies can yield a solution of the problem."

To initialize the structures for adaptation by genetic operators, we must create two sets: F, the set of functions and T, the set of terminal values required for the domain. F can be as simple as {+, *, –, /} or may require more complex functions such as sin(X), cos(X), or functions for matrix operations. T may be the integers, reals, matrices, or more complex expressions. The symbols in T must be closed under the functions defined in F.

Next, a population of initial "programs" is generated by randomly selecting elements from the union of sets F and T. For example, if we begin by selecting an element of T, we have a degenerate tree of a single root node. More interestingly, when we start with an element from F, say +, we get a root node of a tree with two potential children. Suppose the initializer next selects * (with two potential children) from F, as the first child, and then terminal 6 from T as the second child. Another random selection might yield the terminal 8, and then the function + from F. Assume it concludes by selecting 5 and 7 from T.

The program we have randomly produced is represented in Figure 12.4. Figure 12.4a gives the tree after the first selection of +, 15.4b after selecting the terminal 6, and 15.4c the final program. A population of similar programs is created to initialize the genetic programming process. Sets of constraints, such as the maximum depth for programs to evolve, can help prune this population. Descriptions of these constraints, as well as different methods for generating initial populations, may be found in Koza (1992).

The discussion to this point addresses the issues of representation (s-expressions) and the set of tree structures necessary to initialize a situation for program evolution. Next, we require a fitness measure for populations of programs. The fitness measure is problem domain dependent and usually consists of a set of tasks the evolved programs must address. The fitness measure itself is a function of how well each program does on these tasks. A simple *raw fitness* score would add the differences between what the program produced and the results that the actual task from the problem domain required. Thus, raw fitness could be seen as the sum of errors across a set of tasks. Other fitness measures are possible, of course. Normalized fitness divides raw fitness by the total sum of possible errors and thus puts all fitness measures within the range of 0 to 1. Normalization can have an advantage when trying to select from a large population of programs. A fitness measure can also include an adjustment for the size of the program, for example, to reward smaller, more parsimonious programs.

Genetic operators on programs include both transformations on a tree itself as well as the exchange of structures between trees. Koza (1992) describes the primary transformations as *reproduction* and *crossover*. Reproduction simply selects programs from the

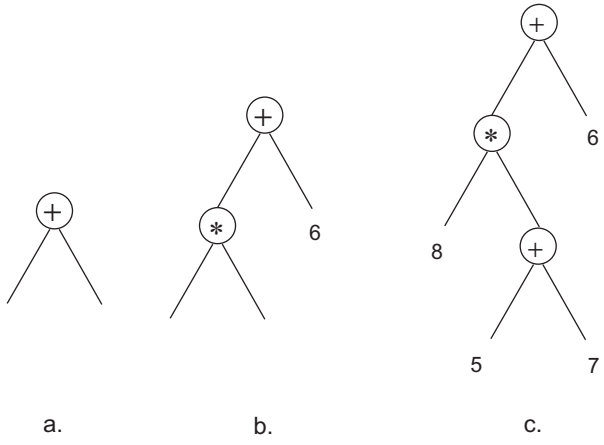

Figure 12.4 The random generation of a program to initialize. The circled nodes are from the set of functions.

present generation and copies them (unchanged) into the next generation. Crossover exchanges subtrees between the trees representing two programs. For example, suppose we are working with the two parent programs of Figure 12.5, and that the random points indicated by | in parent a and parent b are selected for crossover. The resulting children are shown in Figure 12.6. Crossover can also be used to transform a single parent, by interchanging two subtrees from that parent. Two identical parents can create different children with randomly selected crossover points. The root of a program can also be selected as a crossover point.

There are a number of secondary, and much less used, genetic transforms of program trees. These include *mutation*, which simply introduces random changes in the structures of a program. For example, replacing a terminal value with another value or a function subtree. The *permutation* transform, similar to the inversion operator on strings, also works on single programs, exchanging terminal symbols, or subtrees, for example.

The state of the solution is reflected by the current generation of programs. There is no record keeping for backtrack or any other method for skipping around the fitness landscape. In this aspect genetic programming is much like the hill-climbing algorithm described in Section 4.1. The genetic programming paradigm parallels nature in that the evolution of new programs is a continuing process. Nonetheless, lacking infinite time and computation, termination conditions are set. These are usually a function both of program fitness and computational resources.

The fact that genetic programming is a technique for the computational generation of computer programs places it within the automatic programming research tradition. From the earliest days of AI, researchers have worked to automatically produce computer programs from fragmentary information (Shapiro 1992). Genetic programming can be seen as another tool for this important research domain. We conclude this section with a simple example of genetic programming taken from Mitchell (1996).

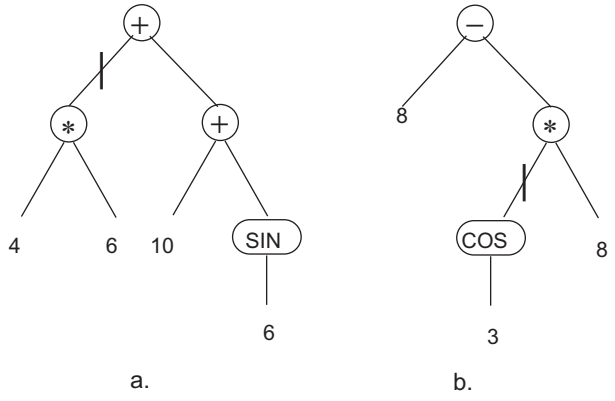

Figure 12.5 Two programs, selected on fitness for crossover. Point | from a and b are randomly selected for crossover.

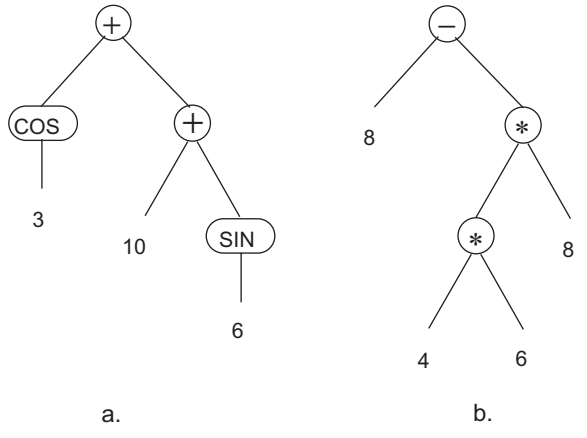

Figure 12.6 The child programs produced by crossover of the points in Figure 12.5.

EXAMPLE 3.2.1: EVOLVING A PROGRAM FOR KEPLER'S THIRD LAW OF PLANETARY MOTION

Koza (1992, 2005) describes many applications of genetic programming to solve interesting problems, but most of these examples are large and too complex for our present purposes. Mitchell (1996), however, has created a simple example that illustrates many of the concepts of genetic programming. Kepler's Third Law of Planetary Motion describes the functional relationship between the orbital period, P, of a planet and its average distance, A, from the sun.

The function for Kepler's Third Law, with c a constant is:

$$P^2 = cA^3$$

If we assume that P is expressed in units of earth years, and A is expressed in units of earth's average distance from the sun, then c = 1. The s-expression of this relationship is:

```
P = (sqrt (* A (* A A)))
```

And so the program we want to evolve is represented by the tree structure of Figure 12.7.

The selection of the set of terminal symbols in this example is simple; it is the single real value given by A. The set of functions could be equally simple, say {+, –, *, /, sq, sqrt}. Next we will create a beginning random population of programs. The initial population might include:

```
(* A (– (* A A) (sqrt A)))      fitness: 1
(/ A (/ (/ A A) (/ A A)))      fitness: 3
(+ A (* (sqrt A) A))           fitness: 0
```

(We explain the attached fitness measures shortly). As noted earlier in this section this initializing population often has a priori limits both of size and search depth, given knowledge of the problem. These three examples are described by the program trees of Figure 12.8.

Next we determine a suite of tests for the population of programs. Suppose we know some planetary data we want our evolved program to explain. For example, we have the planetary data in Table 12.3, taken from Urey (1952), which gives a set of data points that the evolving programs must explain.

Planet	A (input)	P (output)
Venus	0.72	0.61
Earth	1.0	1.0
Mars	1.52	1.87
Jupiter	5.2	11.9
Saturn	9.53	29.4
Uranus	19.1	83.5

Table 12.3 A set of fitness cases, with planetary data taken from Urey (1952). A is Earth s semi-major axis of orbit and P is in units of earth-years.

Since the fitness measure is a function of the data points we want to explain, we define fitness as the number of outputs of the program that come within 20 per cent of the correct output values. We use this definition to create the fitness measures of the three programs of Figure 12.8. It remains for the reader to create more members of this initial population, to build crossover and mutation operators that can produce further generations of programs, and to determine termination conditions.

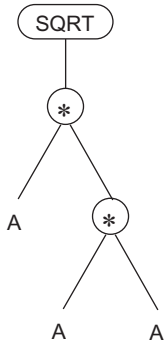

Figure 12.7 The target program relating orbit to period for Kepler s Third Law.

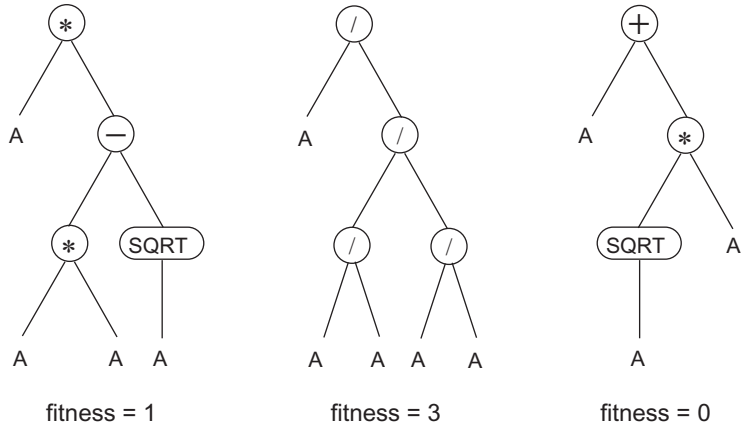

Figure 12.8 Members from the initial population of programs to solve the orbital period problem.

12.3 Artificial Life and Society-Based Learning

Earlier in this chapter, we described a simplified version of "The Game of Life." This game, most effectively shown in computational visual simulations where succeeding generations rapidly change and evolve on a screen display, has a very simple specification. It was first proposed as a board game by the mathematician John Horton Conway, and made famous through Martin Gardner's discussion of it in *Scientific American* (1970, 1971). The Game of Life is a simple example of a model of computation called *cellular automata* (CA). Cellular automata are families of simple, finite-state machines that exhibit interesting, emergent behaviors through their interactions in a population.

DEFINITION

FINITE-STATE MACHINE or CELLULAR AUTOMATON

1. A set I called the *input alphabet*.
2. A set S of *states* that the automaton can be in.
3. A designated state s_0, the *initial state*.
4. A *next state function* N: S x I $\rightarrow$ S, that assigns a next state to each ordered pair consisting of a current state and a current input.

The output of a Finite State Machine, as presented previously in Section 3.1, is a function of its present state and input values. The cellular automata makes the input to the present state a function of its "neighbor" states. Thus, the state at time (t + 1) is a function of its present state and *the state of its neighbors* at time t. It is through these interactions with neighbors that collections of cellular automata achieve much richer behaviors than simple finite state machines. Because the output of all states of a system is a function of their neighboring states, we can describe the evolution of a set of neighboring FSMs as society-based adaptation and learning.

For the societies described in this section, there is no explicit evaluation of the fitness of individual members. Fitness results from interactions in the population, interactions that may lead to the "death" of individual automata. Fitness is implicit in the survival of individuals from generation to generation. Learning among cellular automata is typically unsupervised; as occurs in natural evolution, adaptation is shaped by the actions of other, co-evolving members of the population.

A global, or society-oriented viewpoint also allows an important perspective on learning. We no longer need to focus exclusively on the individual, but can rather see invariances and regularities emerging within the society as a whole. This is an important aspect of the Crutchfield–Mitchell research presented in Section 12.3.2.

Finally, unlike supervised learning, evolution need not be "intentional". That is, the society of agents need not be seen as "going somewhere", say to some "omega" point. We did have a convergence bias when we used the explicit fitness measures in the earlier sections of this chapter. But as Stephen Jay Gould (1977, 1996) points out, evolution need not be viewed as making things "better", rather it just favors survival. The only success is continued existence, and the patterns that emerge are the patterns of a society.

12.3.1 The "Game of Life"

Consider the simple two-dimensional grid or game board of Figure 12.9. Here we have one square "occupied" - having bit value 1 - in black, with its eight nearest neighbors indicated by gray shading. The board is transformed over time periods, where the state of each square at time t + 1 is a function of its state and the state of these indicated nearest neighbors at time t. Three simple rules can drive evolution in the game: First, if any square, occupied or not, has exactly three of its nearest neighbors occupied, it will be

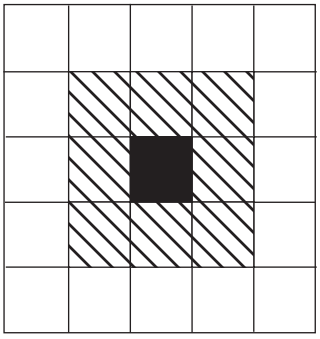

Figure 12.9 The shaded region indicates the set of neighbors for the game of life.

occupied at the next time period. Second, if any occupied square has exactly two of its nearest neighbors occupied, it will be occupied in the next time period. Finally, for all other situations the square will not be occupied at the next time period.

One interpretation of these rules is that, for each generation or time period, life at any location, i.e., whether or not the square is occupied (has state value 1) is a result of its own as well as its neighbors' life during the previous generation. Specifically, too dense a population of surrounding neighbors (more than three) or too sparse a neighboring population (less than two) at any time period will not allow life for the next generation.

Consider, for example, the state of life for Figure 12.10a. Here exactly two squares, indicated by an x, have exactly three occupied neighbors. At the next life cycle Figure 12.10b will be produced. Here again there are exactly two squares, indicated by y, with exactly three occupied neighbors. It can be seen that the state of the world will cycle back and forth between Figures 12.10a and 12.10b. The reader can determine what the next state will be for Figures 12.11a and 12.11b and examine other possible "world" configurations. Poundstone (1985) describes the extraordinary variety and richness of the structures that can emerge in the game of life, such as *gliders*, patterns of cells that move across the world through repeated cycles of shape changes, as is seen in Figure 12.12.

Because of their ability to produce rich collective behaviors through the interactions of simple cells, cellular automata have proven a powerful tool for studying the mathematics of the emergence of life from simple, inanimate components. *Artificial life* is defined as *life made by human effort rather than by nature*. As can be seen in the previous example, artificial life has a strong "bottom up" flavor; that is, the atoms of a life-system are defined and assembled and their physical interactions "emerge". Regularities of this life form are captured by the rules of the finite state machine.

But how might a-life constructs be used? In biology, for example, the set of living entities provided by nature, as complex and diverse as they may be, are dominated by accident and historical contingency. We trust that there are logical regularities at work in the creation of this set, but there need not be, and it is unlikely that we will discover many of the total possible regularities when we restrict our view to the set of biological entities that nature actually provides. It is critical to explore the full set of possible biological

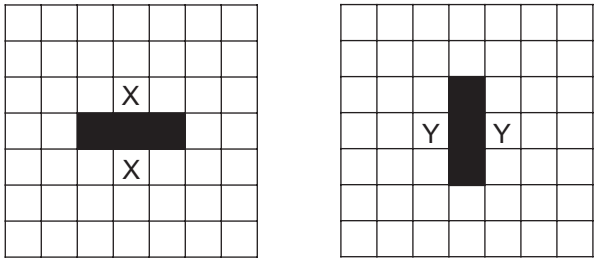

Figure 12.10 A set of neighbors generating the blinking light phenomenon.

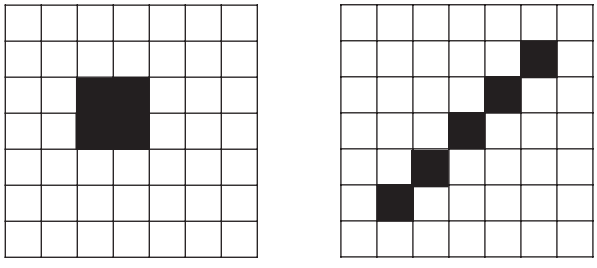

Figure 12.11 What happens to these patterns at the next time cycle?

regularities, some of which may have been eliminated by historical accident. We can always wonder what the present world would be like had not the dinosaurs' existence been peremptorily terminated. To have a theory of the actual, it is necessary to understand the limits of the possible.

Besides the determined effort of anthropologists and other scientists to fill in the gaps in knowledge of our actual evolution, there is continued speculation about rerunning the story of evolution itself. What might happen if evolution started off with different initial conditions? What if there were alternative intervening "accidents" within our physical and biological surroundings? What might emerge? What would remain constant? The evolutionary path that actually did occur on earth is but one of many possible trajectories. Some of these questions might be addressed if we could generate some of the many biologies that are possible. (See as an example PACE, Programmable Artificial Cell Evolution in Section 12.3.3.)

A-life technology is not just an artifact of computational or biological domains. Research scientists from areas as diverse as chemistry and pharmacology have built synthetic artifacts, many related to the knowledge of actual entities existing in our world. For example, in the field of chemistry, research into the constitution of matter and the

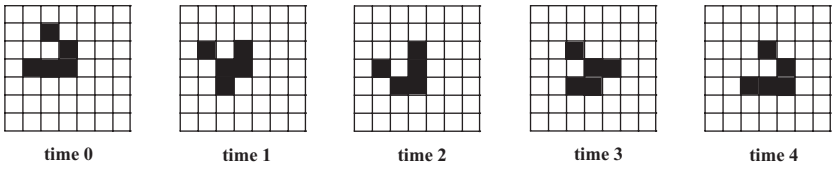

Figure 12.12 A glider moves across the display.

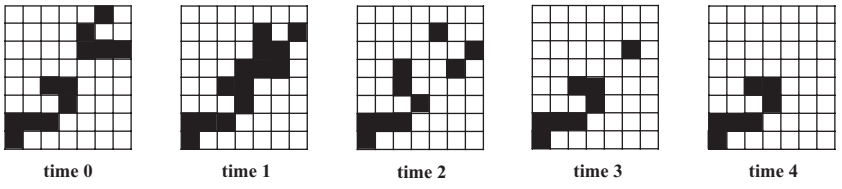

Figure 12.13 A glider is consumed by another entity.

many compounds that nature provides has led to analysis of these compounds, their constituent pieces, and their bonds. This analysis and recombination has led to the creation of numerous compounds that do not exist naturally. Our knowledge of the building blocks of nature has led us to our own synthetic versions, putting components of reality together in new and different patterns. It is through this careful analysis of natural chemical compounds that we come to some understanding of the set of possible compounds.

One tool for understanding possible worlds is to simulate and analyze society based movement and interaction effects. We have simple examples of this in the Game of Life. The sequence of time cycles demonstrated in Figure 12.12 implements the *glider* that was mentioned earlier. The glider sweeps across the game space by cycling among a small number of patterns. Its action is simple as it moves, in four time periods, to a new location one row further to the right and one row closer to the bottom of the grid.

An interesting aspect of the game of life is that entities such as the glider persist until interacting with other members of their society; what then happens can be difficult to understand and predict. For example, in Figure 12.13, we see the situation where two gliders emerge and engage. After four time periods, the glider moving down and to the left is "consumed" by the other entity. It is interesting to note that our ontological descriptions, that is, our use of terms such as "entity", "blinking light", "glider", "consumed", reflects our own anthropocentric biases on viewing life forms and interactions, whether artificial or not. It is very human of us to give names to regularities as they emerge within our social structures.

12.3.2 Evolutionary Programming

The "Game of Life" is an intuitive, highly descriptive example of cellular automata. We can generalize our discussion of cellular automata by characterizing them as finite state

machines. We now discuss societies of linked FSMs and analyze them as emergent entities. This study is sometimes called *evolutionary programming*.

The history of evolutionary programming goes back to the beginning of computers themselves. John von Neumann, in a series of lectures in 1949, explored the question of what level of organizational complexity was required for self-replication to occur (Burks 1970). Burks cites von Neumann's goal as "...not trying to simulate the self-reproduction of a natural system at the level of genetics and biochemistry. He wished to abstract from the natural self-reproduction problem its logical form".

By removing chemical, biological, and mechanical details, von Neumann was able to represent the essential requirements for self-replication. von Neumann went on to design (it was never built) a self-reproducing automaton consisting of a two-dimensional cellular arrangement containing a large number of individual 29-state automata, where the next state for each automaton was a function of its current state and the states of its four immediate neighbors (Burks 1970, 1987).

Interestingly, von Neumann designed his self-replicating automaton, estimated to contain at least 40,000 cells, to have the functionality of a Universal Turing Machine. This universal computation device was also *construction universal*, in the sense that it was capable of reading an input tape, interpreting the data on the tape, and, through use of a construction arm, building the configuration described on the tape in an unoccupied part of the cellular space. By putting a description of the constructing automaton itself on the tape, von Neumann created a self-reproducing automaton (Arbib 1966).

Later Codd (1968) reduced the number of states required for a computationally universal, self-reproducing automaton from 29 to 8, but required an estimated 100,000,000 cells for the full design. Later Devore simplified Codd's machine to occupy only about 87,500 cells. In modern times, Langton created a self-replicating automaton, without computational universality, where each cell had only eight states and occupied just 100 cells (Langton 1986, Hightower 1992, Codd 1992). Current descriptions of these research efforts may be found in the proceedings of the a-life conferences (Langton 1989, Langton et al. 1992).

Thus, the formal analysis of self-replicating machines has deep roots in the theory of computation. Perhaps even more exciting results are implicit in empirical studies of a-life forms. The success of these programs is not indicated by some *a priori* fitness function, but rather by the simple fact that they can survive and replicate. Their mark of success is that they survive. On the darker side, we have experienced the legacy of computer viruses and worms that are able to work their way into foreign hosts, replicate themselves (usually destroying any information in the memory required for replication), and move on to infect yet other foreign hosts.

We conclude this section by discussing several research projects that can be seen as examples of a-life computing. The final section of Chapter 12 is a detailed example of emergent computation by Melanie Mitchell and her colleagues from the Sante Fe Institute. We first present two projects already discussed in our earlier chapters, that of Rodney Brooks at MIT and Nils Nilsson and his students at Stanford. In the earlier presentations, Brooks' work came under the general heading of alternative representations, Section 6.3, and Nilsson's under the topic of planning, Section 7.4.3. In the context of the present chapter, we recast their work in the context of artificial life and emergent phenomena.

Rodney Brooks (1991*a*, *b*) at MIT has built a research program based on the premise of a-life, namely that intelligence emerges through the interactions of a number of simple autonomous agents. Brook's describes his approach as "intelligence without representation", building a series of robots able to sense obstacles and move around the offices and hallways at MIT. Based on the *subsumption architecture* these agents can wander, explore, and avoid other objects. The intelligence of this system is an artifact of simple organization and embodied interactions with their environment. Brooks states "We wire finite state machines together into layers of control. Each layer is built on top of existing layers. Lower level layers never rely on the existence of higher level layers." Further references include McGonigle (1990, 1998), Brooks (1987, 1991a); Lewis and Luger (2000).

Nils Nilsson and his students (Nilsson 1994, Benson 1995, Benson and Nilsson 1995) designed a *teleo-reactive* (T-R) program for agent control, a program that directs an agent towards a goal in a manner that continuously takes into account changing environmental circumstances. This program operates very much with the flavor of a production system but also supports *durative action*, or action that takes place across arbitrary time periods, such as *go forward until....* Thus, unlike ordinary production systems, conditions must be continuously evaluated, and the action associated with the current highest true condition is always the one being executed. For further detail, the interested reader is directed to Nilsson (1994), Benson (1995), Benson and Nilsson (1995), Klein et al. (2000).

The European commission is supporting the PACE, Programmable Artificial Cell Evolution, project. This research has produced a new generation of synthetic chemical cells. The focus is that life evolves around living computational systems, that are self-organizing as well as self-repairing. The complete artificial cells are also self-reproducing, capable of evolution, can maintain their complex structure using simpler environmental resources. For more details see the PACE website http://www.istpace.org/research_overview/artificial_cells_in_pace.html.

The John Koza research effort in genetic programming at Stanford University continues to publish its many successes. Recent advances include the automatic development of circuits and controllers, the synthesis of circuit topology, sizing, placement and routing, as well as other engineering artifacts. References may be found in Koza (2005).

These four research efforts are samples from a very large population of automaton-based research projects. These projects are fundamentally experimental. They ask questions of the natural world. The natural world responds with survival and growth for successful algorithms as well as the annihilation of a system incapable of adaptation.

Finally, we consider research from the Santa Fe Institute: a case study in emergence.

12.3.3 A Case Study in Emergence (Crutchfield and Mitchell 1995)

Crutchfield and Mitchell explore the ability of evolution and interaction within simple systems to create higher-level collective information processing relationships. Their

research offers an example of the emergence of instances of global computation across a spatial system consisting of distributed and locally interacting processors. The term *emergent computation* describes the appearance of global information processing structures in these systems. The goal of the Crutchfield and Mitchell research is to describe an architecture and mechanisms sufficient to evolve and support methods for emergent computation.

Specifically, a cellular automaton (CA) is made up of a number of individual cells; in fact, there are 149 cells in each automaton of the examples we present. These binary-state cells are distributed across a one-dimensional space with no global coordination. Each cell changes state as a function of its own state and the states of its two immediate neighbors. The CA forms a two-dimensional lattice as it evolves across time periods. The lattice starts out with an initial randomly generated set of N cells. In the example of Figure 12.14, there are 149 cells and 149 time steps of their evolution. (There is a zero time period and the cells are numbered from 0, 1, ..., 148). Two examples of these cellular automata's behavior may be seen in the space-time diagrams of Figure 12.14. In these diagrams the 1s are given as black cells and the 0s as white cells. Of course, different rules for the cell neighborhoods will produce different patterns in the space-time diagram of the CA.

Next we describe the rule set that determines the activity of the cells that make up each CA. Figure 12.15 presents a one-dimensional binary-state nearest neighbor CA, with N = 11 cells. Both the lattice and the rule table for updating the lattice are presented. The lattice is shown changing across one time step. The lattice is actually a cylinder, with the left end and the right end of the lattice at each time period being neighbors (this is important for applying the rule set). The rule table supports the local *majority vote* rule: if a local neighborhood of three cells has a majority of ones, then the center cell becomes a one at the next time step; otherwise, it becomes a zero at the next time step.

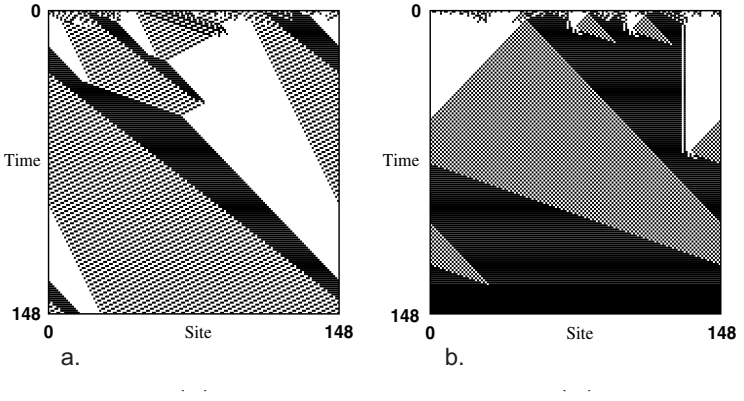

Figure 12.14 Space-time diagrams showing the behavior of two As, discovered by the genetic algorithm on different runs. They employ embedded particles for the non local computation or general emerging patterns seen. Each space-time diagram iterates over a range of time steps, with 1s given as black cells, 0s as white cells; time increases down the page, from Crutchfield and Mitchell (1995).

Crutchfield and Mitchell want to find a CA that performs the following collective computation, here called *majority wins*: if the initial lattice contains a majority of ones, the CA should evolve across time to all ones; otherwise, it should evolve to all zeros. They use CAs with neighborhoods containing seven cells, a center cell with three neighbors on each side. An interesting aspect of this research is that it is difficult to design a CA rule that performs the majority wins computation. In fact, in Mitchell et al. (1996), they show that the simple seven-neighbor "majority vote" rule does not perform the "majority wins" computation. The GA is used to search for a rule that will.

The genetic algorithm (GA), Section 12.1, is used to create the rule tables for different experiments with the CAs. Specifically, a GA is used to evolve the rules for the one-dimensional binary-state cell population that makes up each CA. A fitness function is designed to reward those rules that support the majority wins result for the CA itself. Thus, over time, the GA built a rule set whose fitness was a function of its eventual success in enforcing global majority rules. The fittest rules in the population were selected to survive and randomly combined by crossover to produce offspring, with each offspring subject to a small probability of mutation. This process was iterated for 100 generations, with fitness estimated for a new set of initial cells at each generation. Full details may be found in Crutchfield and Mitchell (1995).

How can we quantify the emergent computation the more successful CAs are supporting? Like many spatially extended natural processes, the cell configurations often organize over time into spatial regions that are dynamically homogenous. Ideally, the analysis and determination of underlying regularities should be an automated process. In fact, Hanson and Crutchfield (1992) have created a language for minimal deterministic finite automaton and use it for describing the attractor-basins within each cellular automaton. This language can be used to describe our example.

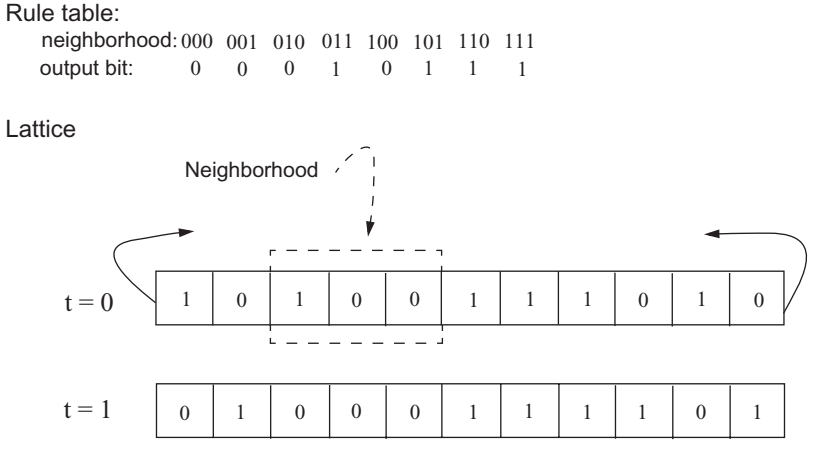

Figure 12.15 Illustration of a one-dimensional, binary-state, nearest-neighbor cellular automaton with N = 11. Both the lattice and the rule table for updating the lattice are illustrated. The lattice configuration is shown over one time step. The cellular automaton is circular in that the two end values are neighbors.

Regular Domains		
$\Lambda^0 = 0^*$	$\Lambda^1 = 1^*$	$\Lambda^2 = (10001)^*$
Particles (Velocities)		
$\alpha \sim \Lambda^1\Lambda^0$ (1)	$\beta \sim \Lambda^0\Lambda^1$ (0)	
$\gamma \sim \Lambda^2\Lambda^0$ (–2)	$\delta \sim \Lambda^0\Lambda^2$ (1/2)	
$\eta \sim \Lambda^2\Lambda^1$ (4/3)	$\mu \sim \Lambda^1\Lambda^2$ (3)	
Interactions		
decay	$\alpha \rightarrow \gamma + \mu$	
react	$\alpha + \delta \rightarrow \mu,\ \eta + \alpha \rightarrow \gamma,\ \mu + \gamma \rightarrow \alpha$	
annihilate	$\eta + \mu \rightarrow \varnothing_1\ ,\ \gamma + \delta \rightarrow \varnothing_0$	

Table 12.4 Catalog of regular domains, particles (domain boundaries), particle velocities (in parentheses), and particle interactions of the space-time behavior of the CA of Figure 12.14a. The notation p ~ $\Lambda^x\Lambda^y$ means that p is the particle forming the boundary between regular domains Λ^x and Λ^y.

Sometimes, as in Figure 12.14a, these regions are obvious to the human viewer as invariant domains, that is, regions in which the same pattern recurs. We will label these domains as Λ values, and then filter out the invariant elements, in order to better describe the interactions (computations) effected by the intersections of these domains. Table 12.4 describes three Λ regions: Λ^0, the repeated 0s; Λ^1, the repeated 1s; and Λ^2, the repeated pattern 10001. There are other Λ regions in Figure 12.14a, but we will only discuss this subset.

With the filtering out of invariant elements of the Λ domains, we can see the interactions of these domains. In Table 12.4, we describe the interaction of six Λ areas, for example, the particles at the frontiers of the Λ^1 and Λ^0 domains. The frontier, where the all 1 domain meets the all 0 domain, is called the *embedded particle* α. Crutchfield and Mitchell claim that the collection of embedded particles is a primary mechanism for carrying information (or signals) over long space-time continua. Logical operations on these particles or signals occur when they collide. Thus the collection of domains, domain walls, particles, and the particle interactions for a CA represent the basic information-processing elements that are embedded in the CAs behavior, that is, that make up the CA's intrinsic computation.

As an example, Figure 12.16 describes the emergent logic of Figure 12.14a. The Λ domain areas have been filtered of their invariant content to allow the domain wall particles to be easily observed. Each of the magnified regions of Figure 12.16 demonstrates the logic of two of the interacting embedded particles. The particle interaction $\alpha + \delta \rightarrow \mu$,

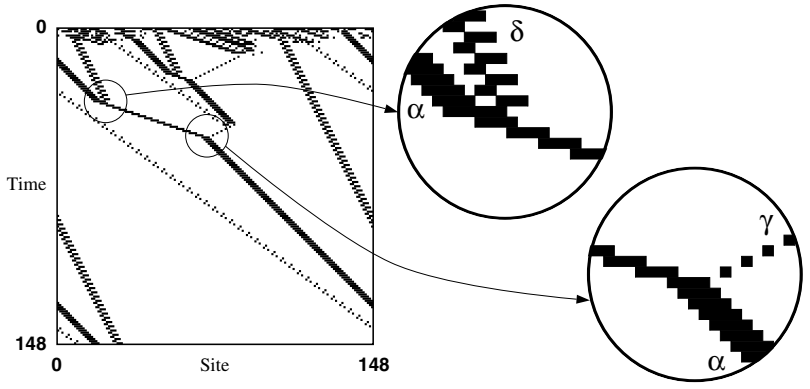

Figure 12.16 Analysis of the emergent logic for density classification of 12.14a. This CA has three domains, six particles, and six particle iterations, as noted in Table 12.3. The domains have been filtered out using an 18-state nonlinear transducer; adapted from Crutchfield and Mitchell (1995).

shown in the upper right, implements the logic of mapping a spatial configuration representing signals α and δ into the signal μ. Similar detail is shown for the particle interaction $\mu + \gamma \rightarrow \alpha$, that maps a configuration representing μ and γ to the signal α. A more complete listing of the particle interactions of Figure 12.16 may be found in Table 12.4.

To summarize, an important result of the Crutchfield–Mitchell research is the discovery of methods for describing emergent computation within a spatially distributed system consisting of locally interacting cell processors. The locality of communication in the cells imposes a constraint of global communication.The role of the GAs is to discover local cell rules whose effect is to perform information processing over "large" space-time distances. Crutchfield and Mitchell have used ideas adopted from formal language theory to characterize these space-time patterns.

For Crutchfield and Mitchell, the result of the evolving automaton reflects an entirely new level of behavior that is distinct from the lower level interactions of the distributed cells. Global particle-based interactions demonstrate how complex coordination can emerge within a collection of simple individual actions. The result of the GA operating on local cell rules showed how an evolutionary process, by taking advantage of certain nonlinear pattern-forming actions of cells, produced a new level of behavior and the delicate balance necessary for effective emergent computation.

The results of the Crutchfield–Mitchell research are important in that they have, with GA support, demonstrated the emergence of higher level invariances within a cellular automaton world. Furthermore, they present computational tools, adapted from formal language theory, that can be used to describe these invariances. Continued research in this domain has the potential to elucidate the emergence of complexity: the defining characteristic of living things, and fundamental to understanding the origins of minds, species, and ecosystems.

12.4 Epilogue and References

Research on genetic algorithms and biology-based learning began with John Holland's design of genetic algorithms. His early research includes *Adaptation in Natural and Artificial Systems* (1975), a study on emergent phenomena in self-replicating systems (1976), and *Escaping Brittleness: The possibilities of general purpose learning algorithms applied to parallel rule-based systems* (1986), introducing *classifiers*. Some examples of work on the analysis of genetic systems can be found in Forrest and Mitchell (Forrest 1990, Mitchell 1996, Forrest and Mitchell 1993a, 1993b). Other researchers, especially Goldberg (1989), Mitchell (1996), and Koza (1992, 1994) have continued the formal analysis of genetic algorithms and learning.

As noted above, Holland (1986) was also responsible for the original design of classifier systems. Classifiers create a macro or complete-system viewpoint of learning. Another similar view is represented by the SOAR project (Laird et al. 1986a, 1986b; Rosenbloom and Newell 1987; Rosenbloom et al. 1993).

A number of popular books, including *Chaos*, by James Gleick (1987) have helped establish the importance of emergent and chaotic systems. Concepts from these systems, such a strange attractors, have found a role in connectionist architectures (Chapter 11).

John Koza is the primary designer of the genetic programming research area. His major contributions are described in: *Genetic Programming: On the programming of computers by means of natural selection* (1992) and *Genetic Programming II: Automatic discovery of reusable programs* (1994). Kosa (2005) uses genetic programming to build engineering tools, including control systems. The example of using genetic programs to learn Kepler's Third Law, Section 12.2.2, was suggested by Mitchell (1996).

The Game of Life was originally presented by the mathematician John Horton Conway, but made famous by Martin Gardner's discussion of it in *Scientific American* (1970, 1971). Research in the computational power of finite state machines goes back to the design of the first digital computers. John von Neumann was very active in this research, and in fact was the first to show that the FSM had the computational power of Turing's Universal Machine. Most of von Neumann's early research is presented in the writings of Arthur Burks (1966, 1970, 1987). Other researchers (Hightower 1992, Koza 1992) describe how a-life research evolved from this early work on FSMs. Other researchers in artificial life include Langton (1986) and Ackley and Littmann (1992). Proceedings of the early a-life conferences were edited by Langton (1989, 1990).

Dennett, *Darwin's Dangerous Ideas* (1995) and *Sweet Dreams* (2006) and other philosophers have addressed the importance of evolutionary concepts in philosophy. We also recommend *Full House: The Spread of Excellence from Plato to Darwin* (Gould 1996).

Besides the brief descriptions of agent research is Section 12.3 (Brooks 1986, 1987, 1991*a*, 1991*b*; Nilsson 1994; Benson and Nilsson 1995), there are many other projects in this domain, including Maes (1989, 1990) model of spreading activation in behavior networks, and the extension of the blackboard architecture by Hayes-Roth et al. (1993).The proceedings of the AAAI, IJCAI, and a-life conferences contain multiple articles from this important research domain. Crutchfield and Mitchell (1995) supported our presentation of Section 12.3.3.

12.5 Exercises

1. The genetic algorithm is intended to support the search for genetic diversity along with the survival of important skills (represented by genetic patterns) for a problem domain. Describe how different genetic operators can simultaneously support both these goals.
2. Discuss the problem of designing representations for genetic operators to search for solutions in different domains? What is the role of *inductive bias* here?
3. Consider the CNF-satisfaction problem of Section 12.1.3. How does the role of the number of disjuncts in the CNF expression bias the solution space? Consider other possible representations and genetic operators for the CNF-satisfaction problem. Can you design another fitness measure?
4. Build a genetic algorithm in the language to solve the CNF-satisfaction problem.
5. Consider the traveling salesperson problem of Section 12.1.3. Discuss the problem of selecting an appropriate representation for this problem. Design other appropriate genetic operators and fitness measures for this problem.
6. Build a genetic algorithm to search for a solution for the traveling salesperson problem.
7. Discuss the role of techniques such as gray coding for shaping the search space of the genetic algorithm. Describe two other areas where similar techniques may be important.
8. Read Holland's Schema Theorem (Mitchell 1996, Koza 1992). How does Holland's schema theory describe the evolution of the GA solution space? What does it have to say about problems not encoded as bit strings?
9. How does the Bucket Brigade Algorithm (Holland 1986) relate to the backpropagation algorithm (Section 14.3)?
10. Write a program to solve Kepler's Third Law of Motion Problem, described with a preliminary representation offered in Section 12.2.2.
11. Discuss the constraints (presented in Section 12.2.2) on using genetic programming techniques to solve problems. For example, what components of a solution cannot be evolved within the genetic programming paradigm?
12. Read the early discussion of the Game of Life in Gardner's column of *Scientific American* (1970, 1971). Discuss other a-life structures, similar to the *glider*, of Section 12.3.1.
13. Write an a-life program that implements the functionality of Figures 12.10–12.13.
14. The area of agent-based research was introduced in Section 12.3. We recommend further reading on any of the projects mentioned, but especially the Brooks, Nilsson and Benson, or Crutchfield and Mitchell research. Write a short paper on one of these topics.
15. Discuss the role of inductive bias in the representations, search strategies, and operators used in the models of learning presented in Chapter 12. Is this issue resolvable? That is, does the genetic model of learning work solely because of its representational assumptions or can it be translated into broader domains?
16. For further insights into evolution and complexity, read and discuss *Darwin's Dangerous Idea* (Dennett 1995) or *Full House: The Spread of Excellence from Plato to Darwin* (Gould 1996).