Classifier Systems and Genetic Algorithms

L.B. Booker, D.E. Goldberg and J.H. Holland
Computer Science and Engineering, 3116 EECS Building, The University of Michigan, Ann Arbor, MI 48109, U.S.A.

ABSTRACT

Classifier systems are massively parallel, message-passing, rule-based systems that learn through credit assignment (the bucket brigade algorithm) and rule discovery (the genetic algorithm). They typically operate in environments that exhibit one or more of the following characteristics: (1) perpetually novel events accompanied by large amounts of noisy or irrelevant data; (2) continual, often real-time, requirements for action; (3) implicitly or inexactly defined goals; and (4) sparse payoff or reinforcement obtainable only through long action sequences. Classifier systems are designed to absorb new information continuously from such environments, devising sets of competing hypotheses (expressed as rules) without disturbing significantly capabilities already acquired. This paper reviews the definition, theory, and extant applications of classifier systems, comparing them with other machine learning techniques, and closing with a discussion of advantages, problems, and possible extensions of classifier systems.

1. Introduction

Consider the simply defined world of checkers. We can analyze many of its complexities and with some real effort we can design a system that plays a pretty decent game. However, even in this simple world novelty abounds. A good player will quickly learn to confuse the system by giving play some novel twists. The real world about us is much more complex. A system confronting this environment faces perpetual novelty—the flow of visual information impinging upon a mammalian retina, for example, never twice generates the same firing pattern during the mammal's lifespan. How can a system act other than randomly in such environments?

It is small wonder, in the face of such complexity, that even the most carefully contrived systems err significantly and repeatedly. There are only two cures. An outside agency can intervene to provide a new design, or the system can revise its own design on the basis of its experience. For the systems of most interest here—cognitive systems or robotic systems in realistic environments, ecological systems, the immune system, economic systems, and so on—the first option is rarely feasible. Such systems are immersed in continually changing

Artificial Intelligence **40** (1989) 235–282
0004-3702/89/$3.50 © 1989, Elsevier Science Publishers B.V. (North-Holland)

environments wherein timely outside intervention is difficult or impossible. The only option then is learning or, using the more inclusive word, adaptation.

In broadest terms, the object of a learning system, natural or artificial, is the expansion of its knowledge in the face of uncertainty. More directly, a learning system improves its performance by generalizing upon past experience. Clearly, in the face of perpetual novelty, experience can guide future action only if there are relevant regularities in the system's environment. Human experience indicates that the real world abounds in regularities, but this does not mean that it is easy to extract and exploit them.

In the study of artificial intelligence the problem of extracting regularities is the problem of discovering useful representations or categories. For a machine learning system, the problem is one of constructing relevant categories from the system's primitives (pixels, features, or whatever else is taken as given). Discovery of relevant categories is only half the job; the system must also discover what kinds of action are appropriate to each category. The overall process bears a close relation to the Newell–Simon [40] problem solving paradigm, though there are differences arising from problems created by perpetual novelty, imperfect information, implicit definition of the goals, and the typically long, coordinated action sequences required to attain goals.

There is another problem at least as difficult as the representation problem. In complex environments, the actual attainment of a goal conveys little information about the overall process required to attain the goal. As Samuel [42] observed in his classic paper, the information (about successive board configurations) generated during the play of a game greatly exceeds the few bits conveyed by the final win or a loss. In games, and in most realistic environments, these "intermediate" states have no associated payoff or direct information concerning their "worth." Yet they play a stage-setting role for goal attainment. It may be relatively easy to recognize a triple jump as a critical step toward a win; it is much less easy to recognize that something done many moves earlier set the stage for the triple jump. How is the learning system to recognize the implicit value of certain stage-setting actions?

Samuel points the way to a solution. Information conveyed by intermediate states can be used to construct a model of the environment, and this model can be used in turn to make predictions. The verification or falsification of a prediction by subsequent events can be used then to improve the model. The model, of course, also includes the states yielding payoff, so that predictions about the value of certain stage-setting actions can be checked, with revisions made where appropriate.

In sum, the learning systems of most interest here confront some subset of the following problems:

(1) a perpetually novel stream of data concerning the environment, often noisy or irrelevant (as in the case of mammalian vision),

(2) continual, often real-time, requirements for action (as in the case of an organism or robot, or a tournament game),

(3) implicitly or inexactly defined goals (such as acquiring food, money, or some other resource, in a complex environment),

(4) sparse payoff or reinforcement, requiring long sequences of action (as in an organism's search for food, or the play of a game such as chess or go).

In order to tackle these problems the learning system must:

(1) invent categories that uncover goal-relevant regularities in its environment,

(2) use the flow of information encountered along the way to the goal to steadily refine its model of the environment,

(3) assign appropriate actions to stage-setting categories encountered on the way to the goal.

It quickly becomes apparent that one cannot produce a learning system of this kind by grafting learning algorithms onto existing (nonlearning) AI systems. The system must continually absorb new information and devise ranges of competing hypotheses (conjectures, plausible new rules) without disturbing capabilities it already has. Requirements for consistency are replaced by competition between alternatives. Perpetual novelty and continual change provide little opportunity for optimization, so that the competition aims at satisficing rather than optimization. In addition, the high-level interpreters employed by most (nonlearning) AI systems can cause difficulties for learning. High-level interpreters, by design, impose a complex relation between primitives of the language and the sentences (rules) that specify actions. Typically this complex relation makes it difficult to find simple combinations of primitives that provide plausible generalizations of experience.

A final comment before proceeding: Adaptive processes, with rare exceptions, are far more complex than the most complex processes studied in the physical sciences. And there is as little hope of understanding them without the help of theory as there would be of understanding physics without the attendant theoretical framework. Theory provides the maps that turn an uncoordinated set of experiments or computer simulations into a cumulative exploration. It is far from clear at this time what form a unified theory of learning would take, but there are useful fragments in place. Some of these fragments have been provided by the connectionists, particularly those following the paths set by Sutton and Barto [98], Hinton [23], Hopfield [36] and others. Other fragments come from theoretical investigations of complex adaptive systems such as the investigations of the immune system pursued by Farmer, Packard and Perelson [14]. Still others come from research centering on genetic algorithms and classifier systems (see, for example, [28]). This paper focuses on contributions deriving from the latter studies, supplying some

illustrations of the interaction between theory, computer modeling, and data in that context. A central theoretical concern is the process whereby structures (rule clusters and the like) emerge in response to the problem solving demands imposed by the system's environment.

2. Overview

The machine learning systems discussed in this paper are called *classifier systems*. It is useful to distinguish three levels of activity (see Fig. 1) when looking at learning from the point of view of classifier systems:

At the lowest level is the *performance system*. This is the part of the overall system that interacts directly with the environment. It is much like an expert system, though typically less domain-dependent. The performance systems we will be talking about are rule-based, as are most expert systems, but they are message-passing, highly standardized, and highly parallel. Rules of this kind are called classifiers. The performance system is discussed in detail in Section 3; Section 4 relates the terminology and procedures of classifier systems to their counterparts in more typical AI systems.

Because the system must determine which of its rules are effective, a second level of activity is required. Generally the rules in the performance system are of varying usefulness and some, or even most, of them may be incorrect. Somehow the system must evaluate the rules. This activity is often called *credit assignment* (or apportionment of credit); accordingly this level of the system will be called the *credit assignment system*. The particular algorithms used here for credit assignment are called *bucket brigade algorithms*; they are discussed in Section 5.

The third level of activity, the *rule discovery system*, is required because,

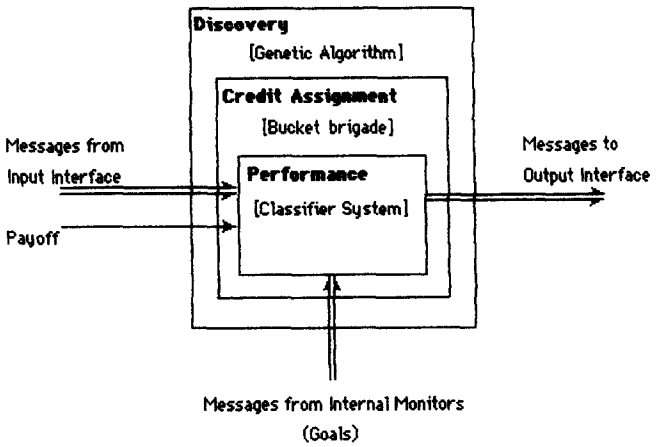

Fig. 1. General organization of a classifier system.

even after the system has effectively evaluated millions of rules, it has tested only a minuscule portion of the plausibly useful rules. Selection of the best of that minuscule portion can give little confidence that the system has exhausted its possibilities for improvement; it is even possible that none of the rules it has examined is very good. The system must be able to generate new rules to replace the least useful rules currently in place. The rules could be generated at random (say by "mutation" operators) or by running through a predetermined enumeration, but such "experience-independent" procedures produce improvements much too slowly to be useful in realistic settings. Somehow the rule discovery procedure must be biased by the system's accumulated experience. In the present context this becomes a matter of using experience to determine useful "building blocks" for rules; then new rules are generated by combining selected building blocks. Under this procedure the new rules are at least plausible in terms of system experience. (Note that a rule may be plausible without necessarily being useful of even correct.) The rule discovery system discussed here employs *genetic algorithms*. Section 6 discusses genetic algorithms. Section 7 relates the procedures implicit in genetic algorithms to some better-known machine learning procedures.

Section 8 reviews some of the major applications and tests of genetic algorithms and classifier systems, while the final section of the paper discusses some open questions, obstacles, and major directions for future research.

Historically, our first attempt at understanding adaptive processes (and learning) turned into a theoretical study of genetic algorithms. This study was summarized in a book titled *Adaptation in Natural and Artificial Systems* (Holland [28]). Chapter 8 of that book contained the germ of the next phase. This phase concerned representations that lent themselves to manipulation by genetic algorithms. It built upon the definition of the *broadcast language* presented in Chapter 8, simplifying it in several ways to obtain a standardized class of parallel, rule-based systems called *classifier systems*. The first descriptions of classifier systems appeared in Holland [29]. This led to concerns with apportioning credit in parallel systems. Early considerations, such as those of Holland and Reitman [34], gave rise to an algorithm called the *bucket brigade algorithm* (see [31]) that uses only local interactions between rules to distribute credit.

3. Classifier Systems

The starting point for this approach to machine learning is a set of rule-based systems suited to rule discovery algorithms. The rules must lend themselves to processes that extract and recombine "building blocks" from currently useful rules to form new rules, and the rules must interact simply and in a highly parallel fashion. Section 4 discusses the reasons for these requirements, but we define the rule-based systems first to provide a specific focus for that discussion.

3.1. Definition of the basic elements

Classifier systems are parallel, message-passing, rule-based systems wherein all rules have the same simple form. In the simplest version all messages are required to be of a fixed length over a specified alphabet, typically k-bit binary strings. The rules are in the usual *condition/action* form. The condition part specifies what kinds of messages satisfy (activate) the rule and the action part specifies what message is to be sent when the rule is satisfied.

A classifier system consists of four basic parts (see Fig. 2).

- The *input interface* translates the current state of the environment into standard messages. For example, the input interface may use property detectors to set the bit values (1: the current state has the property, 0: it does not) at given positions in an incoming message.
- The *classifiers*, the rules used by the system, define the system's procedures for processing messages.
- The *message list* contains all current messages (those generated by the input interface and those generated by satisfied rules).
- The *output interface* translates some messages into effector actions, actions that modify the state of the environment.

A classifier system's basic execution cycle consists of the following steps:

Step 1. Add all messages from the input interface to the message list.

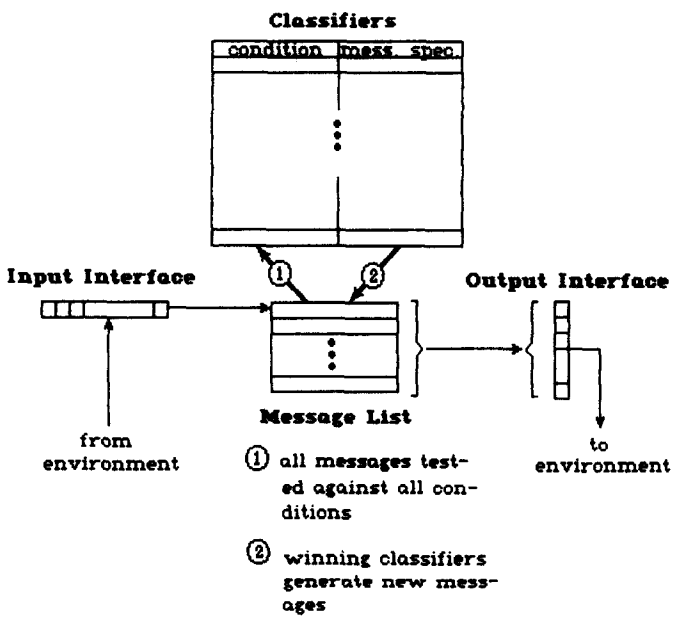

Fig. 2. Basic parts of a classifier system.

Step 2. Compare all messages on the message list to all conditions of all classifiers and record all matches (satisfied conditions).

Step 3. For each set of matches satisfying the condition part of some classifier, post the message specified by its action part to a list of new messages.

Step 4. Replace *all* messages on the message list by the list of new messages.

Step 5. Translate messages on the message list to requirements on the output interface, thereby producing the system's current output.

Step 6. Return to Step 1.

Individual classifiers must have a simple, compact definition if they are to serve as appropriate grist for the learning mill; a complex, interpreted definition makes it difficult for the learning algorithm to find and exploit building blocks from which to construct new rules (see Section 4).

The major technical hurdle in implementing this definition is that of providing a simple specification of the condition part of the rule. Each condition must specify exactly the set of messages that satisfies it. Though most large sets can be defined only by an explicit listing, there is one class of subsets in the message space that can be specified quite compactly, the hyperplanes in that space. Specifically, let $\{1, 0\}^k$ be the set of possible k-bit messages; if we use "#" as a "don't care" symbol, then the set of hyperplanes can be designated by the set of all ternary strings of length k over the alphabet $\{1, 0, \#\}$. For example, the string $1\#\# \ldots \#$ designates the set of all messages that start with a 1, while the string $00 \ldots 0\#$ specifies the set $\{00 \ldots 01, 00 \ldots 00\}$ consisting of exactly two messages, and so on.

It is easy to check whether a given message satisfies a condition. The condition and the message are matched position by position, and if the entries at all non-# positions are identical, then the message satisfies the condition. The notation is extended by allowing any string c over $\{1, 0, \#\}$ to be prefixed by a "$-$" with the intended interpretation that $-c$ is satisfied just in case *no* message satisfying c is present on the message list.

3.2. Examples

At this point we can introduce a small classifier system that illustrates the "programming" of classifiers. The sets of rules that we'll look at can be thought of as fragments of a simple simulated organism or robot. The system has a vision field that provides it with information about its environment, and it is capable of motion through that environment. Its goal is to acquire certain kinds of objects in the environment ("targets") and avoid others ("dangers"). Thus, the environment presents the system with a variety of problems such as "What sequence of outputs will take the system from its present location to a visible target?" The system must use classifiers with conditions sensitive to messages from the input interface, as well as classifiers that integrate the messages from other classifiers, to send messages that control the output interface in appropriate ways.

In the examples that follow, the system's input interface produces a message for each object in the vision field. A set of *detectors* produces these messages by inserting in them the values for a variety of properties, such as whether or not the object is moving, whether it is large or small, etc. The detectors and the values they produce will be defined as needed in the examples.

The system has three kinds of *effectors* that determine its actions in the environment. One effector controls the VISION VECTOR, a vector indicating the orientation of the center of the vision field. The VISION VECTOR can be rotated incrementally each time step (V-LEFT or V-RIGHT, say in 15-degree increments). The system also has a MOTION VECTOR that indicates its direction of motion, often independent of the direction of vision (as when the system is scanning while it moves). The second effector controls rotation of the MOTION VECTOR (M-LEFT or M-RIGHT) in much the same fashion as the first effector controls the VISION VECTOR. The second effector may also align the MOTION VECTOR with the VISION VECTOR, or set it in the opposite direction (ALIGN and OPPOSE, respectively), to facilitate behaviors such as pursuit and flight. The third effector sets the rate of motion in the indicated direction (FAST, CRUISE, SLOW, STOP). The classifiers process the information produced by the detectors to provide sequences of effector commands that enable the system to achieve goals.

For the first examples let the system be supplied with the following property detectors:

$$d_1 = \begin{cases} 1, & \text{if the object is moving,} \\ 0, & \text{otherwise;} \end{cases}$$

$$(d_2, d_3) = \begin{cases} (0,0), & \text{if the object is centered in the vision field,} \\ (1,0), & \text{if the object is left of center,} \\ (0,1), & \text{if the object is right of center;} \end{cases}$$

$$d_4 = \begin{cases} 1, & \text{if the system is adjacent to the object,} \\ 0, & \text{otherwise;} \end{cases}$$

$$d_5 = \begin{cases} 1, & \text{if the object is large,} \\ 0, & \text{otherwise;} \end{cases}$$

$$d_6 = \begin{cases} 1, & \text{if the object is striped,} \\ 0, & \text{otherwise.} \end{cases}$$

Let the detectors specify the rightmost six bits of messages from the input interface, d_1 setting the rightmost bit, d_2 the next bit to the left, etc. (see Fig. 3).

Example 3.1. A simple *stimulus-response* classifier.

IF there is "prey" (*small*, *moving*, *nonstriped object*), centered in the vision field (*centered*), and not adjacent (*nonadjacent*),
THEN move toward the object (ALIGN) rapidly (FAST).

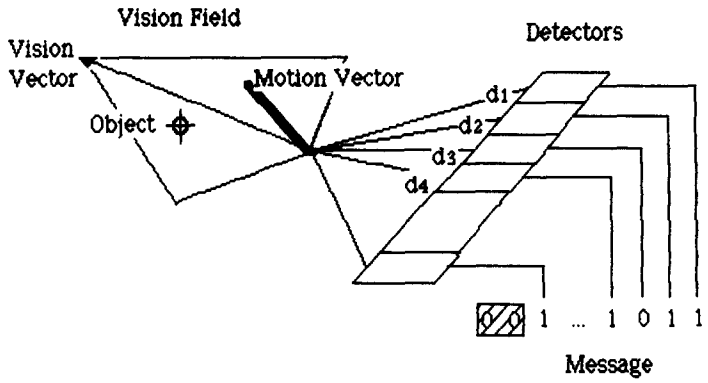

Fig. 3. Input interface for a simple classifier system.

Somewhat fancifully, we can think of the system as an "insect eater" that seeks out small, moving objects unless they are striped ("wasps"). To implement this rule as a classifier we need a condition that attends to the appropriate detector values. It is also important that the classifier recognize that the message is generated by the input interface (rather than internally). To accomplish this we assign messages a prefix or *tag* that identifies their origin—a two-bit tag that takes the value $(0, 0)$ for messages from the input interface will serve for present purposes (see Example 3.5 for a further discussion of tags). Following the conventions of the previous subsection the classifier has the condition

$$00\#\#\#\#\#\#\#\#000001\,,$$

where the leftmost two loci specify the required tag, the # specify the loci (detectors) not attended to, and the rightmost 6 loci specify the required detector values ($d_1 = 1 = \textit{moving}$, being the rightmost locus, etc.). When this condition is satisfied, the classifier sends an outgoing message, say

$$0100000000000000\,,$$

where the prefix 01 indicates that the message is *not* from the input interface. (Though these examples use 16-bit messages, in realistic systems much longer messages would be advantageous.) We can think of this message as being used directly to set effector conditions in the output interface. For convenience these *effector settings*, ALIGN and FAST in the present case, will be indicated in capital letters at the right end of the classifier specification. The complete specification, then, is

$$00\#\#\#\#\#\#\#\#000001 \,/\, 0100000000000000,\ \text{ALIGN},\ \text{FAST}\,.$$

Example 3.2. A set of classifiers detecting a compound object defined by the *relations* between its parts.

The following pair of rules emits an identifying message when there is a moving T-shaped object in the vision field.

> IF there is a centered object that is large, has a long axis, and is moving *along* the direction of that long axis,
> THEN move the vision vector FORWARD (along the axis in the direction of motion) and record the presence of a moving object of type "I".
>
> IF there was a centered object of type "I" observed on the previous time step, and IF there is currently a centered object in contact with "I" that is large, has a long axis, and is moving *crosswise* to the direction of that long axis,
> THEN record the presence of a moving object of type "T" (blunt end forward).

The first of these rules is "triggered" whenever the system "sees" an object moving in the same direction as its long axis. When this happens the system scans forward to see if the object is preceded by an attached cross-piece. The two rules acting in concert detect a compound object defined by the relation between its parts (cf. Winston's [53] "arch"). Note that the pair of rules *can* be fooled; the moving "cross-piece" might be accidentally or temporarily in contact with the moving "I". As such the rules constitute only a first approximation or default, to be improved by adding additional conditions or exception rules as experience accumulates. Note also the assumption of some sophistication in the input and output interfaces: an effector "subroutine" that moves the center of vision along the line of motion, a detector that detects the absence of a gap as the center of vision moves from one object to another, and beneath all a detector "subroutine" that picks out moving objects. Because these are intended as simple examples, we will not go into detail about the interfaces—suffice it to say that reasonable approximations to such "subroutines" exist (see, for example, [37]).

If we go back to our earlier fancy of the system as an insect eater, then moving T-shaped objects can be thought of as "hawks" (not too farfetched, because a "T" formed of two pieces of wood and moved over newly hatched chicks causes them to run for cover, see [43]).

To redo these rules as classifiers we need two new detectors:

$$d_7 = \begin{cases} 1, & \text{if the object is moving in the direction of its long axis,} \\ 0, & \text{otherwise;} \end{cases}$$

$$d_8 = \begin{cases} 1, & \text{if the object is moving in the direction of its short axis,} \\ 0, & \text{otherwise.} \end{cases}$$

We also need a command for the effector subroutine that causes the vision vector to move up the long axis of an object in the direction of its motion, call it V-FORWARD. Finally, let the message 0100000000000001 signal the detection of the moving "I" and let the message 0100000000000010 signal the detection of the moving T-shaped object. The classifier implementing the first rule then has the form

00######01#1#001 / 0100000000000001, V-FORWARD .

The second rule must be contingent upon *both* the just previous detection of the moving "I", signalled by the message 0100000000000001, and the current presence of the cross-piece, signalled by a message from the environment starting with tag 00 and having the value 1 for detector d_8.

0100000000000001, 00######10#1#001 / 0100000000000010 .

Example 3.3. Simple memory.

The following set of three rules keeps the system on alert status if there has been a moving object in the vision field recently. The duration of the alert is determined by a timer, called the ALERT TIMER, that is set by a message, say 0100000000000011, when the object appears.

IF there is a moving object in the vision field ,
THEN set the ALERT TIMER and send an alert message .

IF the ALERT TIMER is not zero ,
THEN send an alert message .

IF there is *no* moving object in the vision field and the ALERT TIMER is not zero ,
THEN decrement the ALERT TIMER .

To translate these rules into classifiers we need an effector subroutine that sets the alert timer, call it SET ALERT, and another that decrements the alert timer, call it DECREMENT ALERT. We also need a detector that determines whether or not the alert timer is zero.

$$d_9 = \begin{cases} 1, & \text{if the ALERT TIMER is } \textit{not} \text{ zero ,} \\ 0, & \text{otherwise .} \end{cases}$$

The classifiers implementing the three rules then have the form

00############1 / 0100000000000011, SET ALERT
00#####1######## / 0100000000000011
00#####1#######0 / DECREMENT ALERT .

Note that the first two rules send the same message, in effect providing an OR of the two conditions, because satisfying either the first condition *or* the second will cause the message to appear on the message list. Note also that these rules check on an *internal* condition via the detector d_9, thus providing a system that is no longer driven solely by external stimuli.

Example 3.4. Building blocks.

To illustrate the possibility of combining several active rules to handle complex situations we introduce the following three pairs of rules.

(A) IF there is an alert and the moving object is near,
THEN move at FAST in the direction of the MOTION VECTOR.

IF there is an alert and the moving object is far,
THEN move at CRUISE in the direction of the MOTION VECTOR.

(B) IF there is an alert, and a small, nonstriped object in the vision field,
THEN ALIGN the motion vector with the vision vector.

IF there is an alert, and a large T-shaped object in the vision field,
THEN OPPOSE the motion vector to the vision vector.

(C) IF there is an alert, and a moving object in the vision field,
THEN send a message that causes the vision effectors to CENTER the object.

IF there is an alert, and *no* moving object in the vision field,
THEN send a message that causes the vision effectors to SCAN.

(Each of the rules in pair (C) sends a message that invokes additional rules. For example "centering" can be accomplished by rules of the form,

IF there is an object in the left vision field,
THEN execute V-LEFT.

IF there is an object in the right vision field,
THEN execute V-RIGHT.

realized by the pair of classifiers

00##########10# / V-LEFT
00##########01# / V-RIGHT.)

Any combination of rules obtained by activating one rule from each of the three subsets (A), (B), (C) yields a potentially useful behavior for the system. Accordingly the rules can be combined to yield behavior in eight distinct situations; moreover, the system need encounter only two situations (involving

disjoint sets of three rules) to test all six rules. The example can be extended easily to much larger numbers of subsets. The number of potentially useful combinations increases as an *exponent* of the number of subsets; that is, n subsets, of two alternatives apiece, yield 2^n distinct combinations of n simultaneously active rules. Once again, only two situations (appropriately chosen) need be encountered to provide testing for *all* the rules.

The six rules are implemented as classifiers in the same way as in the earlier examples, noticing that the system is put on alert status by using a condition that is satisfied by the alert message 0100000000000011. Thus the first rule becomes

$$0100000000000011,\ 00\#\#\#\#0\#\#\#\#\#\#\#\#1\ /\ \text{FAST},$$

where a new detector d_{10}, supplying values at the tenth position from the right in environmental messages, determines whether the object is far (value 1) or near (value 0).

It is clear that the building block approach provides tremendous combinatorial advantages to the system (along the lines described so well by Simon [45]).

Example 3.5. Networks and tagging.

Networks are built up in terms of *pointers* that couple the elements (nodes) of the network, so the basic problem is that of supplying classifier systems with the counterparts of pointers. In effect we want to be able to couple classifiers so that activation of a classifier C in turn causes activation of the classifiers to which it points. The passing of activation between coupled classifiers then acts much like Fahlman's [13] marker-passing scheme, except that the classifier system is passing, and processing, messages. In general we will say a classifier C_2 is *coupled* to a classifier C_1 if some condition of C_2 is satisfied by the message(s) generated by the action part of C_1. Note that a classifier with very specific conditions (few #) will be coupled typically to only a few other classifiers, while a classifier with very general conditions (many #) will be coupled to many other classifiers. Looked at this way, classifiers with very specific conditions have few incoming "branches," while classifiers with very general conditions have many incoming "branches."

The simplest way to couple classifiers is by means of *tags*, bits incorporated in the condition part of a classifier that serve as a kind of identifier or address. For example, a condition of the form 1101## . . . # will accept any message with the prefix 1101. Thus, to send a message to this classifier we need only prefix the message with the tag 1101. We have already seen an example of this use of tags in Example 3.1, where messages from the input interface are "addressed" only to classifiers that have conditions starting with the prefix 00. Because b bits yield 2^b distinct tags, and tags can be placed anywhere in a

condition (the component bits need not even be contiguous), large numbers of conditions can be "addressed" uniquely at the cost of relatively few bits.

By using appropriate tags one can define a classifier that attends to a specific *set* of classifiers. Consider, for example, a pair of classifiers C_1 and C_2 that send messages prefixed with 1101 and 1001, respectively. A classifier with the condition 1101#### . . . # will attend only to C_1, whereas a classifier with condition 1#01#### . . . # will attend to both C_1 and C_2. This approach, in conjunction with recodings (where the prefix of the outgoing message differs from that of the satisfying messages), provides great flexibility in defining the sets of classifiers to which a given classifier attends. Two examples will illustrate the possibilities:

Example 3.5.1. Producing a message in response to an arbitrarily chosen subset of messages.

An arbitrary logical (boolean) combination of conditions can be realized through a combination of couplings and recodings. The primitives from which more complex expressions can be constructed are AND, OR, and NOT. An AND-condition is expressed by a single multi-condition classifier such as $M_1,M_2/M$, for M is only added to the message list if *both* M_1 *and* M_2 are on the list. Similarly the *pair* of classifiers M_1/M and M_2/M express an OR-condition, for M is added to the message list if *either* M_1 *or* M_2 is on the list. NOT, of course, is expressed by a classifier with the condition $-M$. As an illustration, consider the boolean expression

$$(M_1 \text{ AND } M_2) \text{ OR } ((\text{NOT } M_3) \text{ AND } M_4) .$$

This is expressed by the following set of classifiers with the message M appearing if and only if the boolean expression is satisfied.

$$M_1,M_2/M , \qquad -M_3,M_4/M .$$

The judicious use of # and recodings often substantially reduces the number of classifiers required when the boolean expressions are complex.

Example 3.5.2. Representing a network

The most direct way of representing a network is to use one classifier for each pointer (arrow) in the network (though it is often possible to find clean representations using one classifier for each node in the network).

As an illustration of this approach consider the following network fragment (Fig. 4). In marker-passing terms, the ALERT node acquires a marker when there is a MOVING object in the vision field. For the purposes of this example, we will assume that the conjunction of arrows at the TARGET node is a requirement that all three nodes (ALERT, SMALL, and NOT STRIPED) be marked

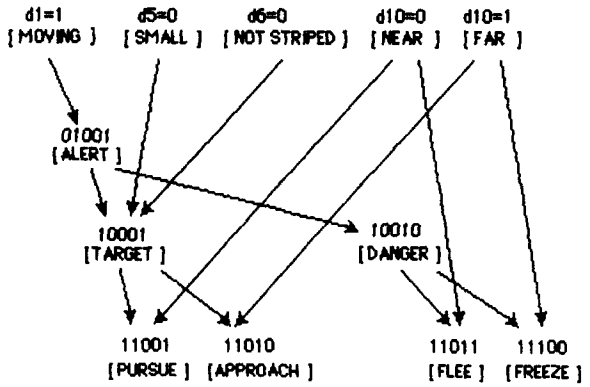

Fig. 4. A network fragment.

before TARGET is marked. Similarly, PURSUE will only be marked if both TARGET and NEAR are marked, etc.

To transform this network into a set of classifiers, begin by assigning an identifying tag to each node. (The tags used in the diagram are 5-bit prefixes). The required couplings between the classifiers are then simply achieved by coordinating the tags used in conditions with the tags on the messages ("markers") to be passed. Henceforth, we extend the notation to allow #'s in the action part of classifiers, where they designate *pass-throughs*: Wherever the message part contains a #, the bit value of the outgoing message is identical to the bit value of the message satisfying the classifier's first condition. That is, the bit value of the incoming (satisfying) message is "passed through" to the outgoing message.

On the basis, assuming that the MOVING node is marked by the detector d_1, the arrow between MOVING and ALERT would be implemented by the classifier

$$00\#\#\#\#\#\#\#\#\#\#\#\#1 \;/\; 01001\#\#\#\#\#\#\#\#\#\#\#,$$

while the arrows leading from SMALL, NOT STRIPED, and ALERT to TARGET could be implemented by the single classifier

$$00\#\#\#\#\#\#\#00\#\#\#\#,\quad 01001\#\#\#\#\#\#\#\#\#\#\;/$$
$$10001\#\#\#\#\#\#\#\#\#\#.$$

In turn, the arrows from NEAR and TARGET to PURSUE could be implemented by

$$00\#\#\#\#0\#\#\#\#\#\#\#\#,\quad 10001\#\#\#\#\#\#\#\#\#\#\;/$$
$$11001\#\#\#\#\#\#\#\#\#\#.$$

The remainder of the network would be implemented similarly.

Some comments are in order. First, the techniques used in Example 3.5.1 to implement boolean connectives apply equally to arrows. For example, we could set conditions so that TARGET would be activated if *either* MOVING and SMALL *or* MOVING and NOT STRIPED were activated. Relations between categories can be introduced following the general lines of Example 3.2. Second, tags can be assigned in ways that provide direct information about the structure of the network. For example, in the network above the first two bits of the tag indicate the level of the corresponding category (the number of arrows intervening between the category and the input from the environment). Finally, effector-oriented categories such as PURSUE would presumably "call subroutines" (sets of classifiers) that carry out the desired actions. For instance, the message from PURSUE would involve such operations as centering the object (see the classifiers just after (C) in Example 3.4), followed by rapid movement toward the object (see the classifier in Example 3.1).

Forrest [15] has produced a general complier for producing coupled classifiers implementing any semantic net specified by KL-ONE expressions.

A final comment on the use of classifiers: Systems of classifiers, when used with learning algorithms, are *not* adjusted for consistency. Instead individual rules are treated as partially confirmed hypotheses, and conflicts are resolved by competition. The specifics of this competition are presented in Section 5.

4. The Relation of Classifier Systems to Other AI Problem Solving Systems

As noted previously, many of the problem solving and learning mechanisms in classifier systems have been motivated by broad considerations of adaptive processes in both natural and artificial systems. This point of view leads to a collection of computation procedures that differ markedly from the symbolic methods familiar to the AI community. It is therefore worthwhile to step back from the details of classifier systems and examine the core ideas that make classifier systems an important part of machine learning research.

When viewed solely as rule-based systems, classifier systems have two apparently serious weaknesses. First, the rules are written in a language that lacks descriptive power in comparison to what is available in other rule-based systems. The left-hand side of each rule is a simple conjunctive expression having a limited number of terms. It clearly cannot be used to express arbitrary, general relationships among attributes. Even though sets of such expressions are adequate in principle, most statements in the classifier language can be expressed more concisely or easily as statements in LISP or logic. Second, because several rules are allowed to fire simultaneously, control issues are raised that do not come up in conventional rule-based systems. Coherency

can be difficult to achieve in a distributed computation. Explicit machinery is needed for insuring a consistent problem solving focus, and the requisite control knowledge may be hard to come by unless the problem is inherently parallel to begin with. These two properties suggest an unconventional approach if a classifier system is to be used to build a *conventional* expert system, though the computational completeness of classifier systems assures it could be done in the usual way.

The key to understanding the advantages of classifier systems is to understand the kind of problems they were designed to solve. A perpetually novel stream of data constitutes an extremely complex and uncertain problem solving environment. A well-known strategy for resolving uncertainty is exemplified by the blackboard architecture (see [12]). By coordinating multiple sources of hierarchically organized knowledge, hypotheses and constraints, problem solving can proceed in an opportunistic way, guided by the summation of converging evidence and building on weak or partial results to arrive at confident conclusions. However, managing novelty requires more than this kind of problem solving flexibility. A system must dynamically construct and modify the representation of the problem itself! Flexibility is required at the more basic level of concepts, relations, and the way they are organized. Classifier systems were designed to make this kind of flexibility possible.

Building blocks are the technical device used in classifier systems to achieve this flexibility. The message list is a global database much like a blackboard, but the possibilities for organizing hypotheses are not predetermined in advance. Messages and tags are building blocks that provide a flexible way of constructing arbitrary hierarchical or heterarchical associations among rules and concepts. Because the language is simple, modifying these associations can be done with local syntactic manipulations that avoid the need for complex interpreters or knowledge-intensive critics. In a similar way, rules themselves are building blocks for representing complex concepts, constraints and problem solving behaviors. Because rules are activated in parallel, new combinations of existing rules and rule clusters can be used to handle novel situations. This is tantamount to building knowledge sources as needed during problem solving.

The apparently unsophisticated language of classifier systems is therefore a deliberate tradeoff of descriptive power for adaptive efficiency. A simple syntax yields building blocks that are easy to identify, evaluate, and recombine in useful ways. Moreover, the sacrifice of descriptive power is not as severe as it might seem. A complex environment will contain concepts that cannot be specified easily or precisely even with a powerful logic. For example, a concept might be an equivalence class in which the members share no common features. Or it might be a relation with a strength that is measured by the distance from some prototype. Or it might be a network of relationships so variable that there are no clearly defined concept boundaries. Rather than construct a syntactically complex representation of such a concept that would

be difficult to use or modify, a classifier system uses *groups* of rules as the representation. The structure of the concept is *modeled* by the organization, variability, and distribution of strength among the rules. Because the members of a group compete to become active (see Section 5), the appropriate aspects of the representation are selected only when they are relevant in a given problem solving context. The modularity of the concept thereby makes it easier to use as well as easier to modify.

This distributed approach to representing knowledge is similar to the way complex concepts are represented in connectionist systems (see [24]). Both frameworks use a collection of basic computing elements as epistemic building blocks. Classifier systems use condition/action rules that interact by passing messages. Connectionist systems use simple processing units that send excitatory and inhibitory signals to each other. Concepts are represented in both systems by the simultaneous activation of several computing elements. Every computing element is involved in representing several concepts, and the representations for similar concepts share elements. Retrieval of a concept is a constructive process that simultaneously activates constituent elements best fitting the current context. This technique has the important advantage that some relevant generalizations are achieved automatically. Modifications to elements of one representation automatically affect all similar representations that share those elements.

There are important differences between classifier systems and connectionist systems, however, that stem primarily from the properties of the building blocks they use. The interactions among computing elements in a connectionist system make "best-fit" searches a primitive operation. Activity in a partial pattern of elements is tantamount to an incomplete specification of a concept. Such patterns are automatically extended into a complete pattern of activity representing the concept most consistent with the given specification. Content-addressable memory can therefore be implemented effortlessly. The same capability is achieved in a classifier system using pointers and tags to link related rules. A directed spreading activation is then required to efficiently retrieve the appropriate concept.

Other differences relate to the way inductions are achieved. Modification of connection strengths is the only inductive mechanism available in most connectionist systems (see [36, 48]). Moreover, the rules for updating strength are part of the initial system design that cannot be changed except perhaps by tuning a few parameters. Classifier systems, on the other hand, permit a broad spectrum of inductive mechanisms ranging from strength adjustments to analogies. Many of these mechanisms can be controlled by, or can be easily expressed in terms of, inferential rules. These inferential rules can be evaluated, modified and used to build higher-level concepts in the same way that building blocks are used to construct lower-level concepts.

Classifier systems are like connectionist systems in emphasizing micro-

structure, multiple constraints and the emergence of complex computations from simple processes. However, classifier systems use rules as a basic epistemic unit, thereby avoiding the reduction of all knowledge to a set of connection strengths. Classifier systems thus occupy an important middle ground between the symbolic and connectionist paradigms.

We conclude this section by comparing classifier systems to SOAR (see [38]), another system architecture motivated by broad considerations of cognitive processes. SOAR is a general-purpose architecture for goal-oriented problem solving and learning. All behavior in SOAR is viewed as a search through a problem space for some state that satisfies the goal (problem solution) criteria. Searching a problem space involves selecting appropriate operators to transform the initial problem state, through a sequence of operations, into an acceptable goal state. Whenever there is an impasse in this process, such as a lack of sufficient criteria for selecting an operator, SOAR generates a subgoal to resolve the impasse. Achieving this subgoal is a new problem that SOAR solves recursively by searching through the problem space characterizing the subgoal. SOAR's knowledge about problem states, operators, and solution criteria is represented by a set of condition/action rules. When an impasse is resolved, SOAR seizes the opportunity to learn a new rule (or set of rules) that summarizes important aspects of the subgoal processing. The new rule, or chunk of knowledge, can then be used to avoid similar impasses in the future. The learning mechanism that generates these rules is called *chunking*.

There are some obvious points of comparison between classifier systems and the SOAR architecture. Both emphasize the flexibility that comes from using rules as a basic unit of representation, and both emphasize the importance of tightly coupling induction mechanisms with problem solving. However, classifier systems do not enforce any one particular problem solving regime the way SOAR does. At a broader level, these systems espouse very different points of view about the mechanisms necessary for intelligent behavior. SOAR emphasizes the sufficiency of a single problem solving methodology coupled with a single learning mechanism. The only way to break a problem solving impasse is by creating subgoals, and the only way to learn is to add rules to the knowledge base by chunking. Classifier systems, on the other hand, place an emphasis on flexibly modeling the problem solving environment. A good model allows for prediction-based evaluation of the knowledge base, and the assignment of credit to the model's building blocks. This, in turn, makes it possible to modify, replace, or add to existing rules via inductive mechanisms such as the recombination of highly rated building blocks. Moreover, a model can provide the constraints necessary to generate plausible reformulations of the representation of a problem. To resolve problem solving impasses, then, classifier systems hypothesize new rules (by recombining building blocks), instead of recompiling (chunking) existing rules.

We will make comparisons to other machine learning methods (Section 7),

after we have defined and discussed the learning algorithms for classifier systems.

5. Bucket Brigade Algorithms

The first major learning task facing any rule-based system operating in a complex environment is the *credit assignment* task. Somehow the performance system must determine both the rules responsible for its successes and the representativeness of the conditions encountered in attaining the successes. (The reader will find an excellent discussion of credit assignment algorithms in Sutton's [47] report.) The task is difficult because overt rewards are rare in complex environments; the system's behavior is mostly "stage-setting" that makes possible later successes. The problem is even more difficult for parallel systems, where only some of the rules active at a given time may be instrumental in attaining later success. An environment exhibiting perpetual novelty adds still another order of complexity. Under such conditions the performance system can *never* have an absolute assurance that any of its rules is "correct." The perpetual novelty of the environment, combined with an always limited sampling of that environment, leaves a residue to uncertainty. Each rule in effect serves as a hypothesis that has been more or less confirmed.

The *bucket brigade* algorithm is designed to solve the credit assignment problem for classifier systems. To implement the algorithm, each classifier is assigned a quantity called its *strength*. The bucket brigade algorithm adjusts the strength to reflect the classifier's overall usefulness to the system. The strength is then used as the basis of a competition. Each time step, each satisfied classifier makes a *bid* based on its strength, and only the highest bidding classifiers get their messages on the message list for the next time step.

It is worth recalling that there are no consistency requirements on posted messages; the message list can hold any set of messages, and any such set can direct further competition. The only point at which consistency enters is at the output interface. Here, different sets of messages may specify conflicting responses. Such conflicts are again resolved by competition. For example, the strengths of the classifiers advocating each response can be summed so that one of the conflicting actions is chosen with a probability proportional to the sum of its advocates.

The bidding process is specified as follows. Let $s(C, t)$ be the strength of classifier C at time t. Two factors clearly bear on the bidding process: (1) relevance to the current situation, and (2) past "usefulness." Relevance is mostly a matter of the specificity of the rule's condition part—a more specific condition satisfied by the current situation conveys more information about that situation. The rule's strength is supposed to reflect its usefulness. In the simplest versions of the competition the bid is a product of these two factors, being 0 if the rule is irrelevant (condition not satisfied) or useless (strength 0),

and being high when the rule is highly specific to the situation (detailed conditions satisfied) and well confirmed as useful (high strength).

To implement this bidding procedure, we modify Step 3 of the basic execution cycle (see Section 3.1).

Step 3. For each set of matches satisfying the condition part of classifier C, *calculate a bid* according to the following formula,

$$B(C, t) = bR(C)s(C, t) ,$$

where $R(C)$ is the specificity, equal to the number of non-# in the condition part of C divided by the length thereof, and b is a constant considerably less than 1 (e.g., $\frac{1}{8}$ or $\frac{1}{16}$). The size of the bid determines the *probability* that the classifier posts its message (specified by the action part) to the new message list. (E.g., the probability that the classifier posts its message might decrease exponentially as the size of the bid decreases.)

The use of probability in the revised step assures that rules of lower strength sometimes get tested, thereby providing for the occasional testing of less-favored and newly generated (lower strength) classifiers ("hypotheses").

The operation of the bucket brigade algorithm can be explained informally via an economic analogy. The algorithm treats each rule as a kind of "middleman" in a complex economy. As a "middleman," a rule only deals with its "suppliers"—the rules sending messages satisfying its conditions—and its "consumers"—the rules with conditions satisfied by the messages the "middleman" sends. Whenever a rule wins a bidding competition, it initiates a transaction wherein it pays out part of its strength to its suppliers. (If the rule does not bid enough to win the competition, it pays nothing.) As one of the winners of the competition, the rule becomes active, serving as a supplier to its consumers, and receiving payments from them in turn. Under this arrangement, the rule's strength is a kind of capital that measures its ability to turn a "profit." If a rule receives more from its consumers than it paid out, it has made a profit; that is, its strength has increased.

More formally, when a winning classifier C places its message on the message list it pays for the privilege by having its strength $s(C, t)$ reduced by the amount of the bid $B(C, t)$,

$$s(C, t + 1) = s(C, t) - B(C, t) .$$

The classifiers $\{C'\}$ sending messages matched by this winner, the "suppliers," have their strengths increased by the amount of the bid—it is shared among them in the simplest version—

$$s(C', t + 1) = s(C', t) + aB(C, t) ,$$

where $a = 1/(\text{no. of members of } \{C'\})$.

A rule is likely to be profitable only if its consumers, in their local transactions, are also (on the average) profitable. The consumers, in turn, will be profitable only if *their* consumers are profitable. The resulting chains of consumers lead to the ultimate consumers, the rules that directly attain goals and receive payoff directly from the environment. (Payoff is added to the strengths of all rules determining responses at the time the payoff occurs.) A rule that regularly attains payoff when activated is of course profitable. The profitability of other rules depends upon their being coupled into sequences leading to these profitable ultimate consumers. The bucket brigade ensures that early acting, "stage-setting" rules eventually receive credit if they are coupled into (correlated with) sequences that (on average) lead to payoff.

If a rule sequence is faulty, the final rule in the sequence loses strength, and the sequence will begin to disintegrate, over time, from the final rule backwards through its chain of precursors. As soon as a rule's strength decreases to the point that it loses in the bidding process, some competing rule will get a chance to act as a replacement. If the competing rule is more useful than the one displaced, a revised rule sequence will begin to form using the new rule. The bucket brigade algorithm thus searches out and repairs "weak links" through its pervasive local application.

Whenever rules are coupled into larger hierarchical knowledge structures, the bucket brigade algorithm is still more powerful than the description so far would suggest. Consider an abstract rule C^* of the general form, "if the goal is G, and if the procedure P is executed, then G will be achieved." C^* will be active throughout the time interval in which the sequence of rules comprising P is executed. If the goal is indeed achieved, this rule serves to activate the response that attains the goal, as well as the stage-setting responses preceding that response. Under the bucket brigade C^* will be strengthened immediately by the goal attainment. On the very next trial involving P, the earliest rules in P will have their strengths substantially increased under the bucket brigade. This happens because the early rules act as suppliers to the strengthened C^* (via the condition "if the procedure P is executed"). Normally, the process would have to be executed on the order of n times to backchain strength through an n-step process P. C^* circumvents this necessity.

6. Genetic Algorithms

The rule discovery process for classifier systems uses a *genetic algorithm* (GA). Basically, a genetic algorithm selects high strength classifiers as "parents," forming "offspring" by recombining components from the parent classifiers. The offspring displace weak classifiers in the system and enter into competition, being activated and tested when their conditions are satisfied. Thus, a genetic algorithm crudely, but at high speed, mimics the genetic processes underlying evolution. It is vital to the understanding of genetic algorithms to

know that even the simplest versions act much more subtly than "random search with preservation of the best," contrary to a common misreading of genetics as a process primarily driven by mutation. (Genetic algorithms have been studied intensively by analysis, Holland [28] and Bethke [4], and simulation, DeJong [11], Smith [46], Booker [6], Goldberg [18], and others.)

Though genetic algorithms act subtly, the basic execution cycle, the "central loop," is quite simple:

Step 1. From the set of classifiers, select pairs according to strength—the stronger the classifier, the more likely its selection.

Step 2. Apply *genetic operators* to the pairs, creating "offspring" classifiers. Chief among the genetic operators is *cross-over*, which simply exchanges a randomly selected segment between the pairs (see Fig. 5).

Step 3. Replace the weakest classifiers with the offspring.

The key to understanding a genetic algorithm is an understanding of the way it manipulates a special class of building blocks called *schemas*. In brief, under a GA, a good building block is a building block that occurs in good rules. The GA biases future constructions toward the use of good building blocks. We will soon see that a GA rapidly explores the space of schemas, a very large space, implicitly rating and exploiting schemas according to the strengths of the rules employing them. (The term *schema* as used here is related to, but should not be confused with, the broader use of that term in psychology).

The first step in making this informal description precise is a careful definition of *schema*. To start, recall that a condition (or an action) for a classifier is defined by a string of letters $\alpha_1\alpha_2 \ldots \alpha_j \ldots \alpha_k$ of length k over the 3-letter alphabet $\{1, 0, \#\}$. It is reasonable to look upon these strings as built up from the component letters $\{1, 0, \#\}$. It is equally reasonable to look upon certain combinations of letters, say 11 or 0##1, as components. All such possibilities can be defined with the help of a new "don't care" symbol "*." To define a given *schema*, we specify the letters at the positions of interest, filling out the rest of the string with "don't cares." (The procedure mimics that for defining conditions, but we are operating at a different level now.) Thus, *0##1** . . . * focuses attention on the combination 0##1 at positions 2 through 5. Equivalently, *0##1** . . . * specifies a *set* of conditions, the set of *all* conditions that can be defined by using the combination 0##1 at positions 2 through 5. Any condition that has 0##1 at the given positions is an *instance* of schema *0##1** . . . *. The set of all *schemas* is just the set $\{1, 0, \#, *\}^k$ of all strings of length k over the alphabet $\{1, 0, \#, *\}$. (Note that a *schema* defines a subset of the set of all possible conditions, while each condition defines a subset of the set of all possible messages.)

A classifier system, at any given time t, typically has many classifiers that contain a given component or schema σ; that is, the system has many instances of σ. We can assign a value $s(\sigma, t)$ to σ at time t by averaging the strengths of

its instances. For example, let the system contain classifier C_1, with condition 10##110...0 and strength $s(C_1, t) = 4$, and classifier C_2, with condition 00##1011...1 and strength $s(C_2, t) = 2$. If these are the only two instances of schema $\sigma = *0##1**...*$ at time t, then we assign to the schema the value

$$s(\sigma, t) = \tfrac{1}{2}[s(C_1, t) + s(C_2, t)] = 3 ,$$

the average of the strengths of the two instances. The general formula is

$$s(\sigma, t) = (1/[\text{no. of instances of } \sigma]) \sum_{C \text{ an instances of } \sigma} s(C, t) .$$

$s(\sigma, t)$ can be looked upon as an *estimate* of the mean value of σ, formed by taking the average value of the samples (instances) of σ present in the classifier system at time t. It is a crude estimate and can mislead the system; nevertheless it serves well enough as a heuristic guide if the system has procedures that compensate for misleading estimates. This the algorithm does, as we will see, by evaluating additional samples of the schema; that is, it constructs new classifiers that are instances of the schema and submits them to the bucket brigade.

Consider now a system with M classifiers that uses the observed averages $\{s(\sigma, t)\}$ to guide the construction of new classifiers from schemas. Two questions arise: (1) How many schemas are present (have instances) in the set of M classifiers? (2) How is the system to calculate and use the $\{s(\sigma, t)\}$?

The answer to the first question has important implications for the use of schemas as building blocks. A single condition (or action) is an instance of 2^k schemas! (This is easily established by noting that a given condition is an instance of *every* schema obtained by substituting an "*" for one or more letters in the definition of the condition.) In a system of M single-condition classifiers, there is enough information to calculate averages for somewhere between 2^k and $M2^k$ schemas. Even for very simple classifiers and a small system, $k = 32$ and $M = 1000$, this is an enormous number, $M2^k \sim 4$ trillion.

The natural way to use the averages would be to construct more instances of above-average schemas, while constructing fewer instances of below-average schemas. That is, the system would make more use of above-average building blocks, and less use of below-average building blocks. More explicitly: Let $s(t)$ be the average strength of the classifiers at time t. Then schema σ is above average if $s(\sigma, t)/s(t) > 1$, and vice versa. Let $M(\sigma, t)$ be the number of instances of schema σ in the system at time t, and let $M(\sigma, t + T)$ be the number of instances of σ after M new classifiers (samples) have been constructed. The simplest heuristic for using the information $s(\sigma, t)/s(t)$ would be to require that the number of instances (uses) of σ increase (or decrease) at time $t + T$ according to that ratio,

$$M(\sigma, t + T) = c[s(\sigma, t)/s(t)]M(\sigma, t) ,$$

where c is an arbitrary constant. It is even possible, *in principle*, to construct the new classifiers so that *every* schema σ *with at least a few instances present at* t receives the requisite number of samples. (This is rather surprising since there are so many schemas and only M new classifiers are constructed; however a little thought and some calculation, exploiting the fact that a single classifier is an instance of 2^k distinct schemas, shows that it is possible.)

A generating procedure following this heuristic, setting aside problems of implementation for the moment, has many advantages. It samples each schema with above-average instances with increasing intensity, thereby further confirming (or disconfirming) its usefulness and exploiting it (if it remains above average). This also drives the overall average $s(t)$ upward, providing an ever-increasing criterion that a schema must meet to be above average. Moreover, the heuristic employs a distribution of instances, rather than working only from the "most recent best" instance. This yields both robustness and insurance against being caught on "false peaks" (local optima) that misdirect development. Overall, the power of this heuristic stems from its rapid accumulation of better-than-average building blocks. Because the strengths underlying the $s(\sigma, t)$ are determined (via the bucket brigade) by the regularities and interactions in the environment, the heuristic provides a sophisticated way of exploiting such regularities and interactions.

Though these possibilities exist in principle, there is no feasible *direct* way to calculate and use the large set of averages $\{s(\sigma, t)/s(t)\}$. However, genetic algorithms do *implicitly* what is impossible explicitly. To see this, we must specify exactly the steps by which a genetic algorithm generates new classifiers.

The algorithm acts on a set $B(t)$ of M strings $\{C_1, C_2, \ldots, C_M\}$ over the alphabet $\{1, 0, \#\}$ with assigned strengths $s(C_j, t)$ via the following steps:

Step 1. Compute the average strength $s(t)$ of the strings in $B(t)$, and assign the normalized value $s(C_j, t)/s(t)$ to each string C_j in $B(t)$.

Step 2. Assign each string in $B(t)$ a probability proportional to its normalized value. Then, using this probability distribution, select n pairs of strings, $n \ll M$, from $B(t)$, and make copies of them.

Step 3. Apply *cross-over* (and, possibly, other *genetic operators*) to each copied pair, forming $2n$ new strings. *Cross-over* is applied to a pair of strings as follows: Select at random a position i, $1 \leq i \leq k$, and then exchange the segments to the left of position i in the two strings (see Fig. 5).

Step 4. Replace the $2n$ lowest strength strings in $B(t)$ with the $2n$ strings newly generated in Step 3.

Step 5. Set t to $t+1$ in preparation for the next use of the algorithm and return to Step 1.

Figure 6 illustrates the operation of the algorithm. In more sophisticated versions of the algorithm, the selection of pairs for recombination may be biased toward classifiers active at the time some triggering condition is satisfied. Also Step 4 may be modified to prevent one kind of string from

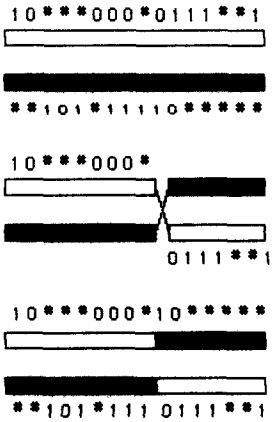

Fig. 5. Example of the cross-over operator.

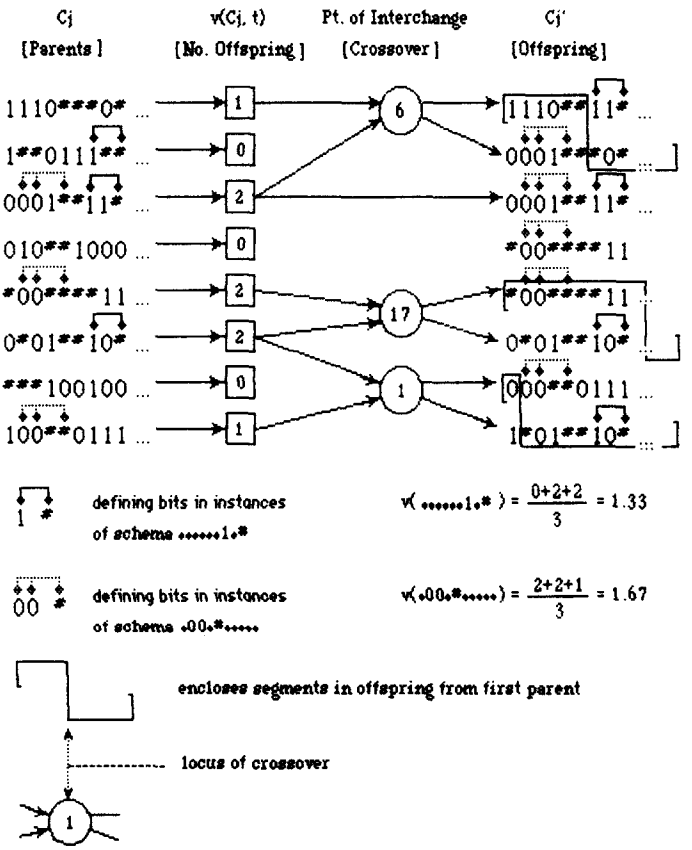

Fig. 6. Example of a genetic algorithm acting on schemas.

"overcrowding" $B(t)$ (see [4, 11] for details).

Contiguity of constituents, and the building blocks constructed from them, are significant under the cross-over operator. Close constituents tend to be exchanged together. Operators for rearranging the atomic constituents defining the rules, such as the genetic operator *inversion*, can bias the rule generation process toward the use of certain *kinds* of building blocks. For example, if "color" is nearer to "shape" than to "taste" in a condition, then a particular "color"-"shape" combination will be exchanged as a unit more often than a "color"-"taste" combination. *Inversion*, by rearranging the positions of "shape" and "taste," could reverse this bias. Other genetic operators, such as *mutation*, have lesser roles in this use of the algorithm, mainly providing "insurance" (see [28, Chapter 6, Sections 2–4] for details).

To see how the genetic algorithm implicitly carries out the schema search heuristic described earlier, it is helpful to divide the algorithm's action into two phases: phase 1 consists of Steps 1–2; phase 2 consists of Steps 3–4.

First consider what would happen if phase 1 were iterated, without the execution of phase 2, but with the replacement of strings in $B(t)$. In particular, let phase 1 be iterated $M/2n$ times (assuming for convenience that M is a multiple of $2n$). Under $M/2n$ repetitions of phase 1, *each instance* C of a given schema σ can be expected to produce $s(\sigma, t)/s(t)$ "offspring" copies. The total number of instances of schema σ after the action of phase 1 is just the *sum* of the copies of the individual instances. Dividing this total by the original number of instances, $M(\sigma, t)$, gives the average rate of increase, and is just $s(\sigma, t)/s(t)$ as required by the heuristic. This is true of every schema with instances in $B(t)$, as required by the heuristic.

Given that phase 1 provides just the emphasis for each schema required by the heuristic, why is phase 2 necessary? Phase 2 is required because phase 1 introduces no *new* strings (samples) into $B(t)$, it merely introduces copies of strings already there. Phase 1 provides emphasis but no new trials. The genetic operators, applied in phase 2, obviously modify strings. It can be proved (see [28, Theorem 6.2.3]) that the genetic operators of Step 3 leave the emphasis provided by phase 1 largely undisturbed, while providing *new* instances of the various schemas in $B(t)$ in accord with that emphasis. Thus, phase 1 combined with phase 2 provides, implicitly, just the sampling scheme suggested by the heuristic.

The fundamental theorem for genetic algorithms [28, Theorem 6.2.3] can be rewritten as a procedure for progressively biasing a probability distribution over the space $\{1, 0, \#\}^k$:

Theorem 6.1. *Let* P_{cross} *be the probability that a selected pair will be crossed, and let* P_{mut} *be the probability that a mutation will occur at any given locus. If* $p(\sigma, t)$ *is the fraction of the population occupied by the instances of* σ *at time* t,

then

$$p(\sigma, t+1) \geq [1 - \lambda(\sigma, t)][1 - P_{\text{mut}}]^{d(\sigma)}[u(\sigma, t)/u(t)]\, p(\sigma, t)$$

gives the expected fraction of the population occupied by instances of σ *at time* $t+1$ *under the genetic algorithm.*

The right-hand side of this equation can be interpreted as follows: $[u(\sigma, t)/u(t)]$, the ratio of the observed average value of the schema s compared to the overall population average, determines the rate of change of $p(\sigma, t)$, subject to the "error" terms $[1 - \lambda(\sigma, t)][1 - P_{\text{mut}}]^{d(\sigma)}$. If $u(\sigma, t)$ is above average, then schema σ tends to increase, and vice versa.

The "error" terms are the result of breakup of instances of σ because of cross-over and mutation, respectively. In particular, $\lambda(\sigma, t) = P_{\text{cross}}(l(\sigma)/k)p(\sigma, t)$ is an upper bound on the loss of instances of σ resulting from crosses that fall within the interval of length $l(\sigma)$ determined by the outermost defining loci of the schema, and $[1 - P_{\text{mut}}]^{d(\sigma)}$ gives the proportion of instances of σ that escape a mutation at one of the $d(\sigma)$ defining loci of σ.

(The underlying algorithm is stochastic so the equation only provides a bound on expectations at each time step. Using the terminology of mathematical genetics, the equation supplies a *deterministic model* of the algorithm under the assumption that the expectations are the values actually achieved on each time step.)

In any population that is not too small—from a biological view, a population not so small as to be endangered from a lack of genetic variation—distinct schemas will almost always have distinct subsets of instances. For example, in a randomly generated population of size 2500 over the space $\{1, 0\}^k$, any schema defined on 8 loci can be expected to have about 10 instances. (For ease of calculation, we consider populations of binary strings in the rest of this section, but the same results hold for n-letter alphabets.) There are

$$\binom{2500}{10} \simeq 3 \times 10^{26}$$

ways of choosing this subset, so that it is extremely unlikely that the subsets of instances for two such schemas will be identical. (Looked at another way, the chance that two schemas have even *one* instance in common is less than $10 \times 2^{-8} \simeq \frac{1}{25}$ if they are defined on disjoint subsets of loci.) Because the sets of instances are overwhelmingly likely to be distinct, the observed averages $\hat{u}(\sigma, t)$, will have little cross-correlation. As a consequence, the rate of increase (or decrease) of a schema σ under a genetic algorithm is largely uncontaminated by the rates associated other such schemas. Loosely, the rate is uninfluenced by "cross-talk" from the other schemas.

To gain some idea of how many schemas are so processed consider the following:

Theorem 6.2. *Select some bound e on the transcription error under reproduction and cross-over, and pick l such that* $l/k \leq \frac{1}{2}e$. *Then in a population of size* $M = c_1 2^{l/k}$, *obtained as a uniform random sample from* $\{1, 0\}^k$, *the number of schemas propagated with an error less than e greatly exceeds* M^3.

Proof. (1) Consider a "window" of $2l$ contiguous loci in a string of length k such that $2l/k = e$. Clearly any schema having all its defining loci within this window will be subject to a transcription error less than e under cross-over.

(2) There are

$$\binom{2l}{l} \simeq 2^{2l}/[\pi l]^{-1/2}$$

ways of selecting l defining positions in the window, and there are 2^l different schemas that can be defined using any given set of l of defining loci. Therefore, there are approximately $2^{3l}/[\pi l]^{-1/2}$ distinct schemas with l defining positions that can be defined in the window.

(3) A population of size $M = c_1 2^l$, for c_1 a small integer, obtained by a uniform random sampling of $\{1, 0\}^k$ can be expected to have c_1 instances of *every* schema defined on l defining positions. Therefore, for the given window, there will be approximately $M^3/(c_1)^3[\pi l]^{-1/2}$ schemas having instances in the population and defined on some set of l loci in the window.

(4) The same argument can be given for schemas of length $l-1, l-2, \ldots,$ and for $l+1, l+2, \ldots,$ with values of

$$\binom{2l}{l \pm j}$$

decreasing in accord with the binomial distribution. There are also $k - l - 1$ distinct positionings of the window on strings of length k. It follows that many more than M^3 schemas, with instances in the population of size M, increase or decrease at a rate given by their observed marginal averages with a transcription error less than e. □

From the point of view of sampling theory, 20 or 30 instances of a schema σ constitute a sample large enough to give some confidence to the corresponding estimate of $u(\sigma)$. Thus, for such schemas, the biases $p(\sigma, t)$ produced by a genetic algorithm over a succession of generations are neither much distorted by sampling error nor smothered by "cross-talk."

It is important to recognize that the genetic algorithm only manipulates M strings while implicitly generating and testing the new instances of the very

large number of schemas involved ($\gg M^3$, early on). Moreover, during this procedure, samples (instances) of schemas not previously tried are generated. This implicit manipulation of a great many schemas through operations on $2n$ strings per step is called *implicit parallelism* (it is called *intrinsic parallelism* by Holland [28]).

7. Comparison with Other Learning Methods

The rule discovery procedures in a classifier system—genetic algorithms—are just as unconventional as the problem solving procedures. Here again, it is important to look beyond the details and examine the core ideas. In terms of the weak methods familiar to the AI community, a genetic algorithm can be thought of as a complex hierarchical generate-and-test process. The generator produces building blocks which are combined into complete objects. At various points in the procedure, tests are made that help weed out poor building blocks and promote the use of good ones. The information requirements of the process are modest: a generator for building blocks and objects, and an evaluator that allows them to be tested and compared with alternatives. It is altogether appropriate to label the procedure a weak method, if one is referring to its lack of domain-dependent requirements.

On closer examination, though, it is apparent that there are important differences between genetic algorithms and the standard assortment of weak methods. The differences are centered on the formulation of the search for useful rules. The familiar weak methods focus on managing the *complexity* of the search space, emphasizing ways to avoid computationally prohibitive exhaustive searches. Such methods use small amounts of knowledge to focus the search and prune the space of alternatives. Genetic algorithms proceed by managing the *uncertainty* of the search space. Uncertainty enters in the sense that the desirability of an element of the search space as a solution or partial solution is unknown until it has been tested. Managing complexity reduces uncertainty as more of the search space is explored; on the other hand, it is also clear that reducing uncertainty makes the search more effective, with complexity becoming more manageable in the process.

This shift of viewpoint is subtle, but has important consequences for the way the search is carried out. Typical AI search procedures use heuristic evaluation functions to prune search paths, and it often suffices that they provide bounds on test outcomes. On the other hand, uncertainty management requires the use of test outcomes (samples) to estimate regularities in the search space. Acquiring and using this knowledge as the search proceeds requires that more attention be paid to the *distribution* of test outcomes over the search space. The focus is on subspaces and the kinds of elements they contain, rather than on paths and their ultimate destinations. That is, the emphasis is on sample-based induction [33].

This point of view is captured by a weak method we will call *sample-select-and-recombine*. Assume that the elements of the search space are structured, that is, that they are constructed of components or building blocks. To find an element in this space:

Step 1. Draw a sample from the space.
Step 2. Order the elements in the sample according to some preference criterion related to the goals of the search.
Step 3. Use this ranking to estimate the usefulness of the building blocks present in the sample's elements.
Step 4. Generate a new sample by selecting building blocks on the basis of this evaluation, recombining them to construct new elements.
Step 5. Repeat Steps 1–4 until the desired element is found.

This method has the obvious advantage that the memory requirements are small and it can be used when a conventional generator or heuristic evaluation function is hard to find. All that is needed is a set of building blocks and a capability to order a sample in terms of a goal-relevant preference. Just as generate-and-test procedures are made more effective by incorporating as much of the test as possible into the generation process, genetic algorithms derive their power by tightly coupling the sampling and selection process. It is important that there are theoretical results that show that genetic algorithms implement the sample-select-and-recombine method in a near-optimal way.

We can make a direct comparison of this approach with more familiar AI learning procedures. This is most easily accomplished in the realm of concept learning tasks where the problem is to find a concept description consistent with a given set of (positive and negative) examples of the concept. Wilson [52] gives a detailed account of the way in which genetic algorithms acting on classifier systems learn complex multiple disjunctive concepts. Here we will refer to the description of genetic algorithms given above, and will briefly examine two well-known learning algorithms: the interference matching algorithm (Hayes-Roth and McDermott [22]), and the candidate elimination algorithm (Mitchell [39]).

7.1. Interference matching

Interference matching is a general technique for inferring the common attributes of several positive examples. (Interference matching is closely related to techniques, such as those examined by Valiant [49], for inferring boolean functions from true and false instances, but interference matching makes less stringent requirements on the match between problem, algorithm and representation.) A schema describing the shared characteristics is constructed using the attribute value for attributes shared, and a place holder symbol (essentially a "don't care" symbol) for attributes that differ over the examples. For

example, interference matching of the two descriptors [RED, ROUND, HEAVY] and [RED, SQUARE, HEAVY] yields the schema [RED, HEAVY]. This technique can be used to compute a set of schemas accounting for all positive examples of a concept—called a *maximal decomposition*—by using the following algorithm:

Let S be the list of schemas, initially empty.
Let $\{E_1, \ldots, E_N\}$ be the set of N examples.
For $i = 1$ to N
  For $j = 1$ to $|S|$
    Form a schema s by interference matching E_i with the jth element of S.
    Form a new schema s' by interference matching all examples that satisfy (are instances of) s.
    Add s' to the list S if it is not already there.
  Repeat until no new schemas are created (for the given value of i).
  Add E_i to S unaltered.

A simple example is given in Fig. 7.

The list of schemas comprising a maximal decomposition is the minimal complete set of nonredundant schemas that occur in the examples. For instance, in Fig. 7, the schemas ***1 and **0* are redundant because they designate the same subset of examples; they are therefore summarized by the more restrictive schema **01. The algorithm is more complicated when there is more than one concept to be learned or, equivalently, negative examples are available. A separate maximal decomposition is computed for each concept, but, in addition, a performance value is computed for each schema. The performance value rates each schema according to its ability to discriminate instances of a concept from noninstances in the set of examples.

Examples	Maximal Decomposition
1001	1001
1110	1***
	1110
0101	**01
	****
	*1**
	0101
0010	*0**
	**10
	0***
	0010

Fig. 7. Example of a maximal decomposition.

Similar algorithms have been suggested for inferring the structure of boolean functions from presentations of true instances (see, for example, Valiant [49]).

Because a maximal decomposition is an exhaustive list of the structural characteristics of a concept, the size of the list becomes unmanageable as the number of examples grows. Hayes-Roth suggests discarding schemas with low performance values to keep the size of the list under control. This strategy can only work, though, if all the examples are available at once. If the examples occur incrementally, the performance value assigned to a schema at any given time is an estimate subject to error. The only obvious way to recover from a mistakenly discarded schema is to recompute the entire maximal decomposition.

Therein lies the major difference between interference matching and genetic algorithms. Genetic algorithms implicitly work with the building blocks for a decomposition. Iterative application of a genetic algorithm produces a population of concept descriptions in which the number of occurrences of each building block is proportional to the observed average performance of its carriers. In this sense, the population is a database that compactly and usefully summarizes the examples so far encountered. If a new example is introduced, it is assimilated by an automatic revision of the proportions of the relevant building blocks. This updating occurs without keeping explicit or exhaustive records about performance, thereby avoiding the large computational burdens associated with updating a maximal decomposition.

7.2. Candidate elimination

The candidate elimination algorithm is similar to genetic algorithms in that it cleverly implements a procedure that would be intractable if attempted by brute force. The basic idea is to enumerate the set of all possible concept descriptions and, for each example, remove from consideration any description that is inconsistent with that example. When there is only one description left the problem is solved. Mitchell [39] makes this idea tractable by ordering the set of possible descriptions according to *generality*. One description is more general than another if it includes as instances all the instances specified by the other description. Thus the schema **0* is more general than the schema **01. This is a partial ordering because not all descriptions are comparable—there can be several maximally general or maximally specific descriptions in the space. The key to this approach is the observation that the set of most specific descriptions and the set of most general descriptions consistent with an example *bound* the set of all descriptions consistent with the example. An algorithm therefore need only keep track of these bounds to converge to the description consistent with all examples.

In more detail, the candidate elimination algorithm maintains two sets that bound the space of consistent descriptions: the set S of most specific possible

descriptions, and the set G of most general possible descriptions. Given a new positive example, the elements of S are generalized the smallest amount that allows inclusion of the new example as an instance. Any element of G that is inconsistent with this example is removed. Similarly, given a new negative example, the elements of G are specialized the smallest amount that precludes the example as an instance. Any element of S that includes this example as an instance is removed. This process is repeated for each new example, the set S becoming more general and the set G becoming more specific, until S and G are identical. The concept description remaining is the one that is consistent with all the examples.

This algorithm obviously has no problems assimilating new examples and it converges to a solution quickly. The basic limitations are: (i) the S and G can be quite large, even for relatively simple concepts, (ii) only conjunctive concepts can be learned, and (iii) the algorithm usually fails if the data are noisy (some instances incorrect). Genetic algorithms avoid these limitations by characterizing the search space in a fundamentally different way: (i) the data set corresponding to the S and G sets is carried implicitly in the proportions of the building blocks, (ii) disjunctions are handled by the parallelism of the rule set, and (iii) noise is handled effortlessly because uncertainty reduction is at the heart of the procedure. The price paid by the genetic algorithm is that it robustly samples the space without concern for the difficulty of the problem; it cannot use an obvious path to a solution to curtail its search unless there are strong building blocks that can be combined to construct that path.

8. Applications

Research on genetic algorithms has paralleled work in mainstream artificial intelligence in the sense that simpler studies of search and optimization in straightforward problem domains have preceded the more complex investigations of machine learning. This is no surprise. Search and optimization applications, with their well-defined problems, objective functions, constraints and decision variables provide a tame environment where alternatives may be compared easily. By contrast, machine learning problems, with their ill-defined goal statements, subjective evaluation criteria and multitudinous decision options, constitute an unwieldy environment not easily given to comparison or analysis. The application of GAs in search and optimization has both tested and improved GAs, and it has encouraged their successful application to search problems that have not succumbed to more traditional procedures. Accordingly this review of applications starts by examining GA applications in search and optimization.

8.1. Genetic algorithms in search and optimization

(Because much of the inspiration for early studies of genetic algorithms came

from genetics, much of the work described in this section was set forth using the terminology of genetics. We have not explained these terms in detail, but we have eliminated any terms that would not appear in a high-school biology text.)

The first *application* of a GA—in fact, the first published use of the words "genetic algorithm"—came in Bagley's [3] pioneering dissertation. At that time there was much interest in game playing computer programs, and in that spirit Bagley devised a controllable testbed of game tasks modeled after the game hexapawn. Bagley's GA operated successfully on "diploid chromosomes" (paired strings) which were decoded to construct parameter sets for a game board evaluation function. The GA contained the three basic operators—reproduction, cross-over, and mutation—along with dominance and inversion. At about the same time, Rosenberg [41] was completing his Ph.D. study of the simulated growth and genetic interaction of a population of single-celled organisms. His organisms were characterized by a simple rigorous biochemistry, a permeable membrane, and a classical, one-gene/one-enzyme structure. He introduced an interesting adaptive cross-over scheme that associated linkage factors with each gene, thereby permitting different linkages between adjacent genes. Rosenberg's work is sometimes overlooked by GA researchers because of its emphasis on biological simulation, but its nearness to root finding and function optimization make it an important contribution to the search domain.

In 1971 Cavicchio [7] investigated the application of GAs to a subroutine selection task and a pattern recognition task. He adopted the pixel weighting scheme of Bledsoe and Browning [5] and used a GA to search for good sets of detectors (subsets of pixels). His GA found good sets of detectors more quickly than a competing "hill-climbing" algorithm. Cavicchio was one of the first to implement a scheme for maintaining population diversity.

The first dissertation to apply GAs to well-posed problems in mathematical optimization was Hollstien's [35] which used a testbed of 14 functions of two variables. The work is notable in its use of allele dominance and schemes of mating preference adopted from traditional breeding practices. Hollstien's GA located optima for his functions much more rapidly than traditional algorithms, but it was difficult to draw general conclusions because he used very small populations ($n = 16$). Frantz [17] studied positional effects on function optimatization. Specifically, he considered functions wherein the value assigned to an argument string could not be well approximated by assigning a least mean squares estimate to each component bit of the argument. (From the point of view of a geneticist, this amounts to saying there are strong epistatic interactions between the genes.) He tested the hypothesis that an inversion (string permutation) operator might improve the efficiency of a GA for such functions. Because the standard GA found near-optimal results quickly in all cases, the inversion operator had little effect. However, for substantially more difficult

problems, such as the traveling salesman problem, job shop scheduling and bin packing, Frantz's hypothesis remains a fruitful avenue of research (see Davis [8, 9], Goldberg and Lingle [19], and Grefenstette, Gopal, Rosmaita and van Gucht [21]). More recently, Bethke [4] has added rigor to the study of functions that are hard for GAs through his investigation of schema averages using Walsh transforms (following a suggestion of Andrew Barto). Goldberg [19] has also contributed to the understanding of GA-hard functions with his definition and analysis of the *minimal deceptive problem*.

De Jong's [11] dissertation was particularly important to subsequent applications of genetic algorithms. He recognized the importance of carefully controlled experimentation in an uncluttered function optimization setting. Varying population size, mutation and cross-over probabilities, and other operator parameters, he examined GA performance in a problem domain consisting of five test functions ranging from a smooth, unimodal function of two variables to functions characterized by high dimensionality (30 variables), great multimodality, discontinuity and noise. To quantify GA performance he defined online and offline performance measures, emphasizing interim performance and convergence, respectively. He also defined a measure of robustness of performance over a range of environments and demonstrated by experiment the robustness of GAs over the test set.

In Appendix A we display a representative group of GA search and optimization applications ranging from an archeological model of the transition from hunting and gathering to agriculture, through VLSI layout problems and medical image registration, to structural optimization. (A complete bibliography covering the entries in Appendices A and B is available from any one of the authors.) The broad successes in these domains have encouraged experiments with GAs in machine learning problems.

8.2. Machine learning using genetic algorithms

The goals for GAs in the context of machine learning have always been clear:

> The study of adaptation involves the study of both the adaptive system and its environment. In general terms, it is a study of how systems can generate procedures enabling them to adjust efficiently to their environments. If adaptability is not to be arbitrarily restricted at the outset, the adapting system must be able to generate any method or procedure capable of an effective definition. (Holland [25])

The original intent, and the original outline of the attendant theory, encompassed a class of adaptive systems much broader than those concerned with search and optimization. The theoretical foundation was used as a basis for defining a series of increasingly sophisticated *schemata processors* (Holland

[26]). Although the 1968 conference at which this paper was presented predates the first application of a classifier system by a full decade [34], schemata processors resemble modern day classifier systems in both outline and detail. The 1978 implementation of Holland and Reitman [34], called CS-1 (Cognitive System Level One), was trained to learn two maze-running tasks. It used (i) a performance system with a message list and simple classifiers, (ii) a credit assignment algorithm that retained information about all classifiers active between successive payoffs and adjusted their strengths at the time of payoff, and (iii) a GA with reproduction, cross-over, mutation and crowding that generated new classifiers. The main result demonstrated that the system could transfer its experience in a simpler maze to improve its rate of learning in a more complex maze.

Smith's [46] study of a classifier system used a purely GA approach, sidestepping the need for a credit assignment algorithm. He represented a rule *set* by a single string, obtained by stringing the rules end to end. He then devised a micro-level cross-over operator, for exchanging segments of individual rules, and a macro-level cross-over operator, for exchanging segments of rule strings (equivalent to exchanging subsets of rules). Smith successfully applied this system, LS-1 (Learning System One) to the Holland and Reitman maze-running task and to a draw poker betting task. In the draw poker task, Smith's system learned to beat Waterman's [50] adaptive poker playing program consistently, a substantial achievement given the amount of domain-specific knowledge in Waterman's program.

The next major application of classifier systems was Booker's [6] study. Booker concentrated on the formal connections between cognitive science and classifier systems. His computer simulations investigated the adaptive behavior of an artificial creature, moving about in a two-dimensional environment containing "food" and "poison," controlled by a classifier system "brain." Booker's classifier system contained a number of innovations including the use of sharing to promote "niche" exploitation, and the use of mating restrictions to reduce the production of ineffective offspring (lethals).

In 1983, Goldberg [18] applied a classifier system to the control of two engineering systems: the pole-balancing problem and a natural gas pipeline-compressor system. The simulations were in the SR (stimulus-response) format, with payoff being presented at each computational time step by a critic. In both cases Goldberg observed the formation of stable subpopulations of rules serving as *default hierarchies*. In a default hierarchy, fairly general rules cover the most frequent cases and more specific rules (that typically contradict the default rules) cover exceptions.

Wilson [51, 52], working along different lines, studied a number of applications of classifier systems. While at Polaroid, he was able to construct and test a classifier system that learned to focus and center a moveable videocamera on an object placed in its field of vision. These experiments, though successful,

caused him to turn to a simpler environment and a simpler version of the classifier system to better understand its behavior. In the later experiments, performed at the Rowland Institute for Science, a classifier system called ANIMAT operated in a two-dimensional environment, searching for food hidden behind obstacles. ANIMAT did not use a message list and hence could not employ a standard bucket brigade algorithm. Instead all classifiers contributing to a chosen action, the *action set*, received strength increments derived either from subsequent environmental payoff or from the bids of the next action set. This bucket brigade-like algorithm successfully propagated credit to early stage-setting rules under conditions of intermittent and noisy payoff.

A number of GA-based machine learning applications and extensions have followed the early works. A representative list, ranging from the evolution of cooperation (Axelrod [2]) and prediction of international events (Schrodt [44]) to VLSI compaction (Fourman [16]), is presented in Appendix B. There are now standard software tools for exploring these systems, including Forrest's [15] KL-ONE-to-classifier-system translator and Riolo's general-purpose, classifier system C-package.[1]

Recent work on classifier systems and genetic algorithms may be found in the books *Genetic Algorithms and Simulated Annealing* (Davis [10]) and *Genetic Algorithms and Their Applications* (Grefenstette [20]), the latter book containing papers presented at a conference held at MIT in the summer of 1987.

9. The Future: Advantages, Problems, Techniques, and Prospects

Up to this point we have reviewed and commented upon established aspects of classifier systems and their learning algorithms. Now we want to look to the future. Section 9.1, as a prologue, reviews some properties of classifier systems that afford future opportunities, while Section 9.2 points up some of the problems that currently impede progress. Section 9.3 outlines some untried techniques that broaden the possibilities for classifier systems, and Section 9.4 offers a look at some of the directions we think will be productive for future research.

9.1. Advantages

When it comes to describing advantages, pride of place goes to the genetic algorithm. The genetic algorithm operating on classifiers discovers potentially useful building blocks, tests them, and recombines them to form plausible new classifiers. It does this at the large "speedup" implied by Theorem 6.2 on

[1]Available on request from R. Riolo, Division of Computer Science and Engineering, 3116 EECS Building, The University of Michigan, Ann Arbor, MI 48109, U.S.A.

implicit parallelism, searching through and testing large numbers of building blocks while manipulating relatively few classifiers.

Competition based on rule strength, in conjunction with the parallelism of classifier systems provides several additional advantages. New rules can be added without imposing the severe computational burden of checking their consistency with all the extant rules. Indeed the system can retain large numbers of mutually contradictory, partially confirmed rules, an important advantage because these rules serve as alternative hypotheses to be invoked when currently favored rules prove inadequate. Moreover, this approach in conjunction with the genetic algorithm provides the overall system with a robust incremental means of handling noisy data. The system has no need of an archival memory of all past examples; its memory resides in the sets of competing alternatives.

9.2. Problems

To this point in time our problems are largely those attending a new approach wherein the experimental landmarks only sparsely cover the landscape of possibilities.

The most serious problem we have encountered concerns the stability of emergent default hierarchies. The hierarchies *do* emerge (see, for example, Goldberg [18], a first as far as we know), but in long runs there may be a catastrophic collapse in which whole subsets of good rules are lost. The rules, or rules similar in effect, are then reacquired, but this instability is highly undesirable.

Forrest [15] has demonstrated that semantic nets can be implemented simply and directly with coupled classifiers, but the question of how such structures can emerge in response to experience has been barely touched. This is, of course, more a research objective than a fault.

We also have only the faintest guidelines as to the functioning of the bucket brigade when the rule sequences are long and intertwined. Again, we have uncovered no faults, we simply have very little knowledge.

9.3. Techniques

There are several new techniques that should substantially increase the power and robustness of classifier systems. Chief among these is the *triggering* of genetic operators. For example, when an input message receives only weak bids from very general classifiers, it is a sign that the system has little specific information for dealing with the current environmental situation. A cross between the input message and the condition parts of some of the active general rules will yield plausible new rules with more specific conditions. This amounts to a bottom-up procedure for producing candidate rules that will automatically be tested for usefulness when similar situations recur. As another

example, when a rule makes a large profit under the bucket brigade, this can be used as a signal to cross it with rules active on the immediately preceding time step. An appropriate cross between the message part of the precursor and the condition part of the profit making successor can produce a new pair of *coupled* rules. (The trigger is only activated if the precursor is *not* coupled to the active profit maker.) The coupled pair models the state transition mediated by the original pair of (uncoupled) rules. Such coupled rules can serve as the building blocks for models of the environment. Because the couplings serve as "bridges" for the bucket brigade, these building blocks will be assigned credit in accord with the efficacy of the models constructed from them. Interestingly enough there seems to be a rather small number of robust triggering conditions (see Holland et al. [33]), but each of them would appear to add substantially to the responsiveness of the classifier system.

Support is another technique that adds considerably to the system's flexibility. Basically, support is a technique that enables the classifier system to integrate many pieces of partial information (such as several views of a partially obscured object) to arrive at strong conclusions. Support is a quantity that travels *with* messages, rather than being a counterflow as in the case of bids. When a classifier is satisfied by several messages from the message list, each such message adds its support into that classifier's *support counter*. Unlike a classifier's strength, the support accrued by a classifier lasts for only the time step in which it is accumulated. That is, the support counter is reset at the end of each time step (other techniques are possible, such as a long or short half-life). Support is used to modify the size of the classifier's bid on that time step; large support increases the bid, small support decreases it. If the classifier wins the bidding competition, the message it posts carries a support proportional to the size of its bid. The propagation of support over sets of coupled classifiers acts somewhat like spreading activation (see [1]), but it is much more directed. Like spreading activation, support can serve to bring associations (coupled rules) into play; but, as mentioned at the outset, it is meant to act primarily as a means of integrating partial information (as when several weakly bidding, general rules bearing on the same topic are activated simultaneously).

9.4. Prospects

The number of feasible directions for exploring the possibilities and applications of classifier systems is almost daunting. Here we will mention only some of the broader paths.

Perhaps the most important thing that can be done at this point is an expansion of the theory. Classifier systems serve as a "testbed" for concepts applicable to a wide range of complex adaptive systems. In developing a mathematics to deal with the interaction of the genetic algorithm and classifier systems we perforce develop a mathematics for dealing with a much wider range of adaptive systems.

The process is reciprocal. For instance, in mathematical economics there are pieces of mathematics that deal with (1) hierarchical organization, (2) retained earnings (fitness) as a measure of past performance, (3) competition based on retained earnings, (4) distribution of earnings on the basis of local interactions of consumers and suppliers, (5) taxation as a control on efficiency, and (6) division of effort between production and research (exploitation versus exploration). Many of these fragments, mutatis mutandis, can be used to study the counterparts of these processes in classifier systems.

Similarly, in mathematical ecology there are pieces of mathematics dealing with (1) niche exploitation (models exploiting environmental regularities), (2) phylogenetic hierarchies, polymorphism and enforced diversity (competing subsystems), (3) functional convergence (similarities of subsystem organization enforced by environmental requirements on payoff attainment), (4) symbiosis, parasitism, and mimicry (couplings and interactions in a default hierarchy, such as an increased efficiency for extant generalists simply because related specialists exclude them from some regions in which they are inefficient), (5) food chains, predator-prey relations, and other energy transfers (apportionment of energy or payoff amongst component subsystems), (6) recombination of multifunctional coadapted sets of genes (recombination of building blocks), (7) assortative mating (biased recombination), (8) phenotypic markers affecting interspecies and intraspecies interactions (coupling), (9) "founder" effects (generalists giving rise to specialists), and (10) other detailed commonalities such as tracking versus averaging over environmental changes (compensation for environmental variability), allelochemicals (cross-inhibition), linkage (association and encoding of features), and still others. Once again, though mathematical ecology is a younger science than mathematical economics, there is much in the mathematics already developed that is relevant to the study of classifier systems and other nonlinear systems far from equilibrium.

In addition to attempting to adapt and extend these fragments, there are at least two broader mathematical tasks that can be undertaken. One is an attempt to produce a general characterization of systems that exhibit *implicit parallelism*. Up to now all such attempts have led to sets of algorithms which are easily recast as genetic algorithms—in effect, we still only know of one example of an algorithm that exhibits implicit parallelism. The second task involves developing a mathematical formulation of the process whereby a system can develop a useful internal model of an environment exhibiting perpetual novelty. In our (preliminary) experiments to date these models typically exhibit a (tangled) hierarchical structure with associative couplings. Such structures have been characterized mathematically as quasi-homomorphisms (see [33]). The perpetual novelty of the environment can be characterized by a Markov process in which each state has a recurrence time that is large relative to any feasible observation time. Considerable progress has been made along these lines (see [32]), but much remains to be done. In particular, we need to construct an interlocking set of theorems based on:

(1) a stronger set of fixed point theorems that relates the strengths of classifiers under the bucket brigade to observed payoff statistics,

(2) a set of theorems that relates building blocks exploited by the "slow" dynamics of the genetic algorithm to the sampling rates for rules at different levels of the emerging default hierarchy (more general rules are tried more often), and

(3) a set of theorems (based on the previous two sets) that detail the way in which various kinds of environmental regularities are exploited by the genetic algorithm acting in terms of the strengths assigned by the bucket brigade.

In the realm of experiment, aside from interesting new applications, the design of experiments centered on the *emergence* of tags under triggered coupling offers intriguing possibilities. Tags serve as the glue of larger systems, providing both associative and temporal (model building) pointers (see Example 3.5). Under certain kinds of triggered coupling (see the previous section) the message sent by the precursor in the coupled pair can have a "hash-coded" section (say a prefix or suffix). The purpose of this hash-coded tag is to prevent accidental eavesdropping by other classifiers—a sufficient number of randomly generated bits in the tag will prevent accidental matches with other conditions (unless the conditions have a lot of # in the tag region). If the coupled pair proves useful to the system then it will have further offspring under the genetic algorithm, and these offspring often will be coupled to other rules in the system. Typically, the tag will be passed on to the offspring, serving as a common element in all the couplings; the tag will only persist if the resulting cluster of rules proves to be a useful "subroutine." In this case, the "subroutine" can be "called" by messages that incorporate the tag, because the conditions of the rules in the cluster are satisfied by such messages. In short, the tag that was initially determined at random now "names" the developing subroutine. It even has a *meaning* in terms of the actions it calls forth. Moreover, the tag is subject to the same kinds of recombination as other parts of the rules (it is, after all, a schema). As such it can serve as a building block for other tags. It is as if the system were inventing symbols for its internal use. Clearly, any simulation that provides for a test of these ideas will be an order of magnitude more sophisticated than anything we have tried to date. Runs involving hundreds of thousands of time steps will probably be required.

Another set of possibilities, far beyond anything we yet understand either theoretically or empirically, is fully directed rule generation. In the *broadcast language* that was the precursor of classifier systems, provision was made for the generation of rules by other rules. With minor changes to the definition of classifier systems, this possibility can be reintroduced. (Both messages and rules are strings. By enlarging the message alphabet, lengthening the message string, and introducing a special symbol that indicates whether a string is to be interpreted as a rule or a message, the task can be accomplished.) With this

provision the system can invent its own candidate operators and rules of inference. Survival of these meta- (operator-like) rules should then be made to depend on the net usefulness of the rules they generate (much as a schema takes its value from the average value of its carriers). It is probably a matter of a decade or two before we can do anything useful in this area.

Another interesting possibility rests on the fact that classifier systems are general-purpose systems. They can be programmed initially to implement whatever expert knowledge is available to the designer; learning then allows the system to expand, correct errors, and transfer information from one domain to another. It is important to provide ways of instructing such systems so that they can generate rules—tentative hypotheses—on the basis of advice. Little has been done in this direction. It is also particularly important that we understand how lookahead and virtual explorations can be incorporated without disturbing other activities of the system.

Our broadest hopes turn on reincarnating in machine learning the cycle of theory and experiment so fruitful in physics. The close control of initial conditions, parameters, and environment made possible by simulation should enable the design of critical tests of the unfolding theory. And the simulations should suggest new directions for the theory. We hope to gain an understanding, not just of classifier systems, but of the consequences of competition in a changing population wherein subsystems are defined by combinations of building blocks that interact in a nonlinear fashion. In this context, classifier systems serve as a well-defined, precisely controllable testbed for a general theory.

Appendix A. Genetic Algorithm Applications in Search and Logic

Cat.	Year	Investigators	Description
Biology			
B	1967	Rosenberg	Simulation of the evolution of single-celled organism populations.
B	1970	Weinberg	Outline of cell population simulation including meta-level GA.
B	1984	Perry	Investigation of niche theory and specification with GAs.
B	1985	Grosso	Simulation of diploid GA with explicit subpopulations and migration.
Computer science			
CS	1967	Bagley	GA-directed parameter search for evaluation function in hexapawn-like game.
CS	1983	Gerardy	Probabilistic automaton identification attempt via GA.
CS	1983	Gordon	Adaptive document description using GA.
CS	1984	Rendell	GA search for game evaluation function.

Cat.	Year	Investigators	Description
Engineering			
E	1981	Goldberg	Mass-spring-dashpot system identification with simple GA.
E	1982	Etter, Hicks, Cho	Recursive adaptive filter design using a simple GA.
E	1983	Goldberg	Steady state and transient optimization of gas pipeline using GA.
E	1985	Davis	Outline of job shop scheduling procedure using GA.
E	1985	Davis, Smith	VLSI circuit layout via GA.
E	1985	Fourman	VLSI layout compaction via GA.
E	1985	Goldberg, Kuo	On-off, steady state optimization of oil pump-pipeline system via GA.
E	1986	Goldberg, Samtani	Structural optimization (plane truss) via GA.
E	1986	Minga	Aircraft landing strut weight optimization via GA.
E	1987	Davis, Coombs	Communications network link size optimization using GA plus advanced operators.
E	1987	Davis, Ritter	Classroom scheduling via simulated annealing with meta-level GA.
Function optimization			
FO	1985	Ackley	Connectionist algorithm with GA-like properties.
FO	1985	Brady	Traveling salesman problem via genetic-like operators.
FO	1985	Davis	Bin-packing and graph-coloring problems via GA.
FO	1985	Grefenstette, Gopal, Rosmaita, van Gucht	Traveling salesman problem via knowledge-augmented genetic operators.
FO	1986	Goldberg, Smith	Blind knapsack problem via simple GA.
Genetic algorithm parameters			
GA	1971	Hollstien	2-D function optimization with mating and selection rules.
GA	1972	Bosworth, Foo, Zeigler	GA-like operators on simulated genes with sophisticated mutation.
GA	1972	Frantz	Investigation of positional nonlinearity and inversion.
GA	1973	Martin	Theoretical study of GA-like probabilistic algorithms.
GA	1975	DeJong	Base-line parametric study of simple GA in 5-function testbed.
GA	1976	Bethke	Brief theoretical investigation of possible parallel GA implementation.
GA	1977	Mercer	GA controlled by meta-level GA.
GA	1981	Bethke	Application of Walsh functions to schema average analysis.
GA	1981	Brindle	Investigation of selection and dominance in GAs.
GA	1981	Grefenstette	Brief theoretical investigation of possible parallel GA implementation.
GA	1983	Pettit, Swigger	Cursory investigation of GAs in nonstationary search problems.
GA	1983	Wetzel	Traveling salesman problem via GA.
GA	1984	Mauldin	Study of several heuristics to maintain diversity in simple GA.

Cat.	Year	Investigators	Description
GA	1985	Baker	Trial of ranking selection procedure on DeJong testbed.
GA	1985	Booker	Suggestion for partial match scores, sharing, and mating restrictions.
GA	1985	Goldberg, Lingle	Traveling salesman problem using partially matched cross-over and schema analysis.
GA	1985	Schaffer	Multi-objective optimization using GAs with subpopulations.
GA	1986	Goldberg	Maximization of marginal schema content by optimization of estimated population size.
GA	1986	Grefenstette	GA controlled by meta-level GA.
GA	1986	Grefenstette, Fitzpatrick	Test of simple genetic algorithm with noisy functions.
GA	1987	Goldberg	Analysis of minimal deceptive problem for simple GAs.
Image processing			
IP	1970	Cavicchio	Selection of detectors for pixel-based pattern recognition.
IP	1984	Fitzpatrick, Grefenstette, van Gucht	Image registration via GA to highlight selected properties.
IP	1985	Englander	Selection of detectors for known image classification.
IP	1985	Gillies	GA search for diagnostic image feature subroutines in Cytocomputer.
Physical sciences			
PS	1985	Shaefer	Nonlinear equation solving with GA for fitting molecular potential surfaces.
Social sciences			
SS	1979	Reynolds	GA-guided adaptation in hunter gatherer/agricultural transition model.
SS	1981	Smith, DeJong	Calibration of population migration model using GA search.
SS	1985	Axelrod	Iterated prisoner's dilemma problem solution using GA.
SS	1985	Axelrod	Simulation of the evolution of behavioral norms with GA.

Appendix B. Genetic Algorithm Applications in Machine Learning

Cat.	Year	Investigators	Description
Business			
BU	1986	Frey	Architectural classification using CS.
BU	1986	Thompson, Thompson	GA search for rule sets to predict company profitability.

Cat.	Year	Investigators	Description
Computer science			
CS	1980	Smith	Draw poker bet decisions learned by pure GA (LS-1).
CS	1985	Cramer	GA learning of multiplication task using assembler-like instruction set.
CS	1985	Forrest	Interpreter to convert KL-ONE networks to CSs.
CS	1986	Riolo	General-purpose C-package for classifier system study.
CS	1986	Riolo	Letter sequence prediction task via CS.
CS	1986	Robertson	LISP version of letter sequence prediction task implemented on Connection Machine
CS	1986	Zeigler	GA searches for rule sets in symbolic rule-based system.
CS	1986	Zhou	GA builds finite automata from I/O examples.
Engineering			
E	1983	Goldberg	Pole-balancing task and gas pipeline control tasks learned by CS.
E	1984	Schaffer	LS-2 (see Smith) learns parity and signal problems.
E	1985	Kuchinski	GA search for battle management system rules.
E	1986	Liepins, Hilliard	Simple scheduling problem learned via CS.
E	1986	Wilson	Boolean multiplexer task learned via CS.
Psychology and social sciences			
SS	1978	Holland, Reitman	CS-1 learns to transfer information between maze-running tasks.
SS	1982	Booker	Animal-like automaton with CS "brain" learns in simple 2-D environment.
SS	1983	Wilson	Video eye learns to focus when driven by CS.
SS	1985	Axelrod	GA searches for rule-based strategies in iterated prisoner's dilemma.
SS	1985	Wilson	ANIMAT automaton with CS "brain" learns to acquire obstacle-hidden objects in 2-D environment.
SS	1986	Schrodt	Prediction of international events using CS.
SS	1986	Haslev (Skanland)	Past tense for Norwegian verb forms learned by CS.

REFERENCES

1. Anderson, J.R., *The Architecture of Cognition* (Harvard University Press, Cambridge, MA, 1983).
2. Axelrod, R., The evolution of strategies in the iterated prisoner's dilemma, in: L. Davis (Ed.), *Genetic Algorithms and Simulated Annealing* (Pitman, London, 1987).
3. Bagley, J.D., The behavior of adaptive systems which employ genetic and correlation algorithms, Ph.D. Dissertation, University Microfilms No. 68-7556, University of Michigan, Ann Arbor, MI (1967).
4. Bethke, A.D., Genetic algorithms as function optimizers, Ph.D. Dissertation, University Microfilms No. 8106101, University of Michigan, Ann Arbor, MI (1981).
5. Bledsoe, W.W. and Browning, I., Pattern recognition and reading by machine, in: *Proceedings Eastern Joint Computer Conference* (1959) 225–232.
6. Booker, L.B., Intelligent behavior as an adaptation to the task environment, Ph.D. Dissertation, University Microfilms No. 8214966, University of Michigan, Ann Arbor, MI (1982).

7. Cavicchio, D.J., Adaptive search using simulated evolution, Ph.D. Dissertation, University of Michigan, Ann Arbor, MI (1970).
8. Davis, L.D., Applying adaptive algorithms to epistatic domains, in: *Proceedings IJCAI-85*, Los Angeles, CA (1985) 162–164.
9. Davis, L.D., Job shop scheduling with genetic algorithms, in: J.J. Grefenstette (Ed.), *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (Carnegie-Mellon University Press, Pittsburgh, PA, 1985) 136–140.
10. Davis, L.D., *Genetic Algorithms and Simulated Annealing* (Morgan Kaufmann, Los Altos, CA, 1987).
11. DeJong, K.A., An analysis of the behavior of a class of genetic adaptive systems, Ph.D. Dissertation, University Microfilms No. 76-9381, University of Michigan, Ann Arbor, MI (1975).
12. Erman, L.D., Hayes-Roth, F., Lesser, V.R. and Reddy, D.R., The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty, *Comput. Surv.* **12** (1980) 213–253.
13. Fahlman, S., *NETL: A System for Representing and Using Real World Knowledge* (MIT Press, Cambridge, MA, 1979).
14. Farmer, J.D., Packard, N.H. and Perelson, A.S., The immune system and artificial intelligence, in: J.J. Grefenstette (Ed.), *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (Carnegie-Mellon University Press, Pittsburgh, PA, 1985) supplement; revised: *Phys. D* **22** (1986) 187–204.
15. Forrest, S., A study of parallelism in the classifier system and its application to classification in KL-ONE semantic networks, Ph.D. Dissertation, University of Michigan, Ann Arbor, MI (1985).
16. Fourman, M.P., Compaction of symbolic layout using genetic algorithms, in: J.J. Grefenstette (Ed.), *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (Carnegie-Mellon University Press, Pittsburgh, PA, 1985) 141–153.
17. Frantz, D.R., Non-linearities in genetic adaptive search, Ph.D. Dissertation, University Microfilms No. 73-11116, University of Michigan, Ann Arbor, MI (1973).
18. Goldberg, D.E., Computer-aided gas pipeline operation using genetic algorithms and rule learning, Ph.D. Dissertation, University Microfilms No. 8402282, University of Michigan, Ann Arbor, MI (1983).
19. Goldberg, D.E. and Lingle, R., Alleles, loci, and the travelling salesman problem, in: J.J. Grefenstette (Ed.), *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (Carnegie-Mellon University Press, Pittsburgh, PA, 1985) 154–159.
20. Grefenstette, J.J., *Genetic Algorithms and Their Applications* (Erlbaum, Hillsdale, NJ, 1987).
21. Grefenstette, J.J., Gopal, R., Rosmaita, B.J. and van Gucht, D., Genetic algorithms for the traveling salesman problem, in: J.J. Grefenstette (Ed.), *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (Carnegie-Mellon University Press, Pittsburgh, PA, 1985) 160–168.
22. Hayes-Roth, F. and McDermott, J., An interference matching technique for inducing abstractions, *Commun. ACM* **21** (1978) 401–410.
23. Hinton, G.E. and Anderson, J.A., *Parallel Models of Associative Memory* (Erlbaum, Hillsdale, NJ, 1981).
24. Hinton, G.E., McClelland, J.L. and Rumelhart, D.E., Distributed representations, in: D.E. Rumelhart and J.L. McClelland (Eds.), *Parallel Distributed Processing*, I: *Foundations* (MIT Press, Cambridge, MA, 1986).
25. Holland, J.H., Outline for a logical theory of adaptive systems, *J. ACM* **3** (1962) 297–314.
26. Holland, J.H., Processing and processors for schemata, in: E.L. Jacks (Ed.), *Associative Information Processing* (American Elsevier, New York, 1971) 127–146.
27. Holland, J.H., Genetic algorithms and the optimal allocation of trials, *SIAM J. Comput.* **2** (1973) 88–105.
28. Holland, J.H., *Adaptation in Natural and Artificial Systems* (University of Michigan Press, Ann Arbor, MI, 1975).

29. Holland, J.H., Adaptation, in: R. Rosen and F.M. Snell (Eds.), *Progress in Theoretical Biology* IV (Academic Press, New York, 1976) 263–293.
30. Holland, J.H., Adaptive algorithms for discovering and using general patterns in growing knowledge-bases, *Int. J. Policy Anal. Inf. Syst.* **4** (1980) 245–268.
31. Holland, J.H., Escaping brittleness: The possibilities of general purpose learning algorithms applied to parallel rule-based systems, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* **2** (Morgan Kaufmann, Los Altos, CA, 1986) 593–623.
32. Holland, J.H., A mathematical framework for studying learning in classifier systems, *Phys. D* **22** (1986) 307–317.
33. Holland, J.H., Holyoak, K.J., Nisbett, R.E. and Thagard, P.R., *Induction: Processes of Inference, Learning, and Discovery* (MIT Press, Cambridge, MA 1986).
34. Holland, J.H. and Reitman, J.S., Cognitive systems based on adaptive algorithms, in: D.A. Waterman and F. Hayes-Roth (Eds.), *Pattern-Directed Inference Systems* (Academic Press, New York, 1978) 313–329.
35. Hollstien, R.B., Artificial genetic adaptation in computer control systems, Ph.D. Dissertation, University Microfilms No. 71-23773, University of Michigan, Ann Arbor, MI (1971).
36. Hopfield, J.J., Neural networks and physical systems with emergent collective computational abilities, *Proc. Nat. Acad. Sci. USA* **79** (1982) 2554–2558.
37. Jain, R., Dynamic scene analysis using pixel-based processes, *IEEE Computer* **14** (1981) 12–18.
38. Laird, J.E., Newell, A. and Rosenbloom, P.S., SOAR: An architecture for general intelligence, *Artificial Intelligence* **33** (1987) 1–64.
39. Mitchell, T.M., Version spaces: A candidate elimination approach to rule learning, in: *Proceedings IJCAI-77*, Cambridge, MA (1977).
40. Newell, A. and Simon, H.A., *Human Problem Solving* (Prentice-Hall, Englewood Cliffs, NJ, 1972).
41. Rosenberg, R.S., Simulation of genetic populations with biochemical properties. Ph.D. Dissertation, University Microfilms No. 67-17836, University of Michigan, Ann Arbor, MI (1967).
42. Samuel, A.L., Some studies in machine learning using the game of checkers, *IBM J. Res. Dev.* **3** (1959) 210–229.
43. Schleidt, W.M., Die historische Entwicklung der Begriffe "Angeborenes Auslosendes Schema" und "Angeborener Auslosmechanismus", *Z. Tierpsychol.* **21** (1962) 235–256.
44. Schrodt, P.A., Predicting international events, *Byte* **11** (12) (1986) 177–192.
45. Simon, H.A., *The Sciences of the Artificial* (MIT Press, Cambridge, MA, 1969).
46. Smith, S.F., A learning system based on genetic adaptive algorithms, Ph.D. Dissertation, University of Pittsburg, Pittsburgh, PA (1980).
47. Sutton, R.S., Learning to predict by the methods of temporal difference, Tech. Rept. TR87-509.1, GTE, Waltham, MA (1987).
48. Sutton, R.S. and Barto, A.G., Toward a modern theory of adaptive networks: Expectation and prediction, *Psychol. Rev.* **88** (1981) 135–170.
49. Valiant, L.G., A theory of the learnable, *Commun. ACM* **27** (1984) 1134–1142.
50. Waterman, D.A., Generalization learning techniques for automating the learning of heuristics, *Artificial Intelligence* **1** (1970) 121–170.
51. Wilson, S.W., On the retinal-cortical mapping, *Int. J. Man-Mach. Stud.* **18** (1983) 361–389.
52. Wilson, S.W., Knowledge growth in an artificial animal, in: J.J. Grefenstette (Ed.), *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (Carnegie-Mellon University Press, Pittsburgh, PA, 1985) 16–23.
53. Winston, P.H., Learning structural descriptions from examples, in: P.H. Winston (Ed.), *The Psychology of Computer Vision* (McGraw-Hill, New York, 1975).