Holland Classifier Systems

Andreas Geyer-Schulz
Department of Applied Computer Science
Institute of Information Processing and Information Economics
Vienna University of Economics and Business Administration
A-1090 Vienna, Augasse 2-6, Austria
e-mail: geyers@wu-wien.ac.at

Abstract

A Holland classifier system is an adaptive, general purpose machine learning system which is designed to operate in noisy environments with infrequent and often incomplete feedback. Examples of such environments are financial markets, stock management systems, or chemical processes. In financial markets, a Holland classifier system would develop trading strategies, in a stock management system order heuristics, and in a chemical plant it would perform process control. In this paper we describe a Holland classifier system and present the implementation of its components, namely the production system, the bucket brigade algorithm, the genetic algorithm, and the cover detector, cover effector and triggered chaining operator. Finally, we illustrate the working of a Holland classifier system by learning to find a path with a high payoff in a simple finite state world.

Keywords Machine learning, classifier system, genetic algorithm, bucket brigade, triggered operations

1 What is a Classifier System?

Classifiers are simply if-then rules. The name *classifier* stems from the capability of rules to classify messages into arbitrary message sets [Holland, 1986a, p. 601]. However, this is only one facet of rules. In classifier systems rules or productions have the same role as instructions in ordinary programs. Such production systems are computationally complete (for a proof see, for example, chapter 12 in Minsky's volume [Minsky, 1971]) and therefore as powerful as any other Turing-equivalent programming language.

A *classifier system* is a machine learning system which learns rules in order to guide its performance in an arbitrary environment [Goldberg, 1989]. For example, consider the application of a classifier system to a stock management problem. In this setting, the decision-making task is straightforward: the classifier system must seek to minimize total costs (inventory costs and backlog costs) by managing its inventory appropriately in the face of uncertain demand. Experienced inventory managers use a set of simple rules or a few simple computations, the order heuristic, for this task. Typically, the classifier system learns rules which describe its order heuristic.

What distinguishes classifier systems from other learning mechanisms is that they gracefully adapt their order heuristic to changes in demand. They optimally balance the conflicting goals of trying new order rules in order to reduce uncertainty. Thus, they improve the order heuristic and exploit new information to increase average performance according to the information gain. In financial markets, a Holland classifier system would develop trading strategies, in a stock management system order heuristics, and in a chemical plant it would perform process control.

Many real world processes of this kind can be studied with the help of finite state Markov processes [Howard, 1971]. Therefore, we have implemented a task environment for learning finite state Markov processes with a Holland classifier system.

Its main components are a production system and one or several learning algorithms. We distinguish two fundamentally different families of classifier systems which differ in the level of what they learn, rule bases or rules: the first learns rule bases, the second learns rules.

The family of classifier systems which concentrate on rule bases needs only a genetic algorithm as learning component. Each individual represents a rule base which is evaluated in a (simulated) environment. The genetic algorithm solves the rule base discovery problem by generating a new set of rule bases for the next generation. This approach has been nick-named the Pitt approach after Pittsburgh, the city where the first classifier system of this type, S. F. Smith's poker player, was written [Goldberg, 1989].

The first classifier system ever written, Holland and Reitman's CS-1 system, is a representative of the second, the rule learning family [Holland and Reitman, 1978]. Members of this family have two learning problems to solve. The first learning problem is the apportionment of credit problem which is the problem of reinforcing or adapting weights of rules already existing in the classifier system, when payoff occurs infrequently. A typical example of such a situation is a game of chess, where the utility of a move or a short sequence of moves has to be assessed, even if such a move occurs very early in the game and payoff occurs only at the end of the game. The second learning problem is the problem of discovering new, potentially useful rules, when the existing rules prove inadequate. We call this family the Michigan family of classifier systems, because the first classifier systems of this type were written at the University of Michigan [Koza, 1992].

2 The Components of a Holland Classifier System

In this section we describe a particular version of a Michigan type classifier system, namely a Holland classifier system which is characterized by a very simple pattern language, parallel rule-firing and message based internal communication. We show the main system components and their interrelationship in Figure 1.

At the top level, the classifier system communicates with the environment. The classifier system effects actions in the environment (game-moves, moves of a robot arm, buying or selling shares or options, setting a price, etc.) and detects information on the state of the environment. Moreover, an action or sequence of actions may lead to payoff received by the classifier system.

The classifier system consists of the production system and two learning components, namely an apportionment of credit system and a rule discovery system. In a Holland classifier system, the apportionment of credit algorithm for updating rule weights is a bucket brigade algorithm. The rule discovery algorithm is a genetic algorithm.

The bucket brigade algorithm modifies the weight of rules (called the *strength* with the payoff from the environment, with payments from message consuming rules and with payments to message producing rules. However, for the bucket brigade to work properly, all

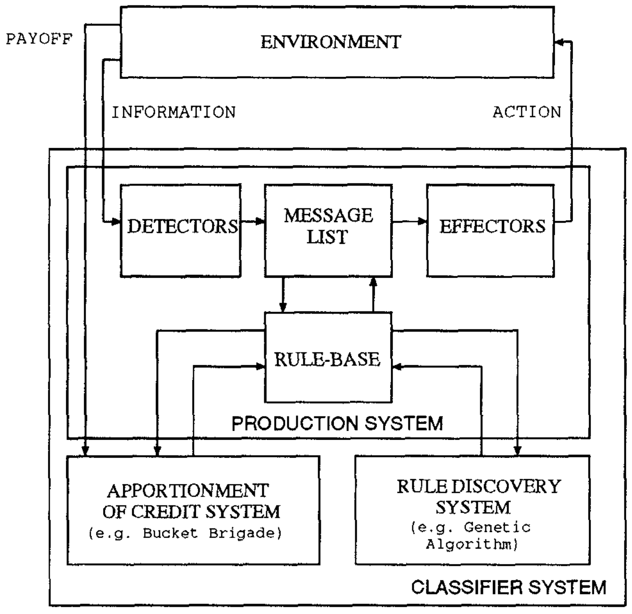

Figure 1: The Architecture of a Michigan Type Classifier System

useful rules must be present in the rule base.

The genetic algorithm sees the rule base as a population of classifiers, whose fitness is the rule-strength obtained under the bucket brigade algorithm. The genetic algorithm is periodically invoked by the production system, and it generates a new population of rules according to rule-strength. However, for the genetic algorithm to work properly, the strengths of the rules generated by the bucket brigade algorithm must reflect the "true" fitness of the rules.

3 The Production System

A Holland classifier system processes internally a stream of binary messages of fixed length. The message stream is filtered through the condition part of a classifier. The condition part is specified as a pattern string with the same length as a message. Matching messages are then rewritten by merging them with the action part of the classifier. The action part is again specified as a pattern string with the same length as a message.

More formally, let the *message* m be a k-symbol string from $\{0,1\}^k$, the *condition* c a k-symbol string from $\{0,1,*,\}^k$ and the *action* a a k-symbol string from $\{0,1,*,\}^k$. The patterns from $\{0,1,*,\}^k$ represent subsets of messages with $*$ representing a wild card symbol. For example, the pattern 100$*$ is matched by the messages 1001 and 1000.

Definition 3.1 *A message* matches *a condition, if for all* $j \in 1, \cdots, k$,

1. either $c[j] = 1$ *implies* $m[j] = 1$,
2. or $c[j] = 0$ *implies* $m[j] = 0$,
3. or $c[j] = *$ *implies* $m[j]$ *either* 1 *or* 0. *(Match.)*

Definition 3.1 is implemented by the function MATCH which is shown below.

```
    ∇Z←MSG MATCH CONDITION
[1]  Z←∧/(Z/MSG)=((Z←~'*'=
     CONDITION)/CONDITION)
    ∇
```

A condition has a one symbol prefix p from $\{0, 1, *\}$ which has the following meaning: 0 or 1 as prefix denote $NOTc$, $*$ as prefix denotes c. $NOTc$ means that the condition c is satisfied, if no message in the message list matches c.

Definition 3.2 *The* merger *of an action a with message m to produce an outgoing message m^* is defined as follows: For all $j \in 1, \cdots, k$,*

1. $m^*[j] = a[j]$, *if* $a[j] = 0$ *or* $a[j] = 1$,
2. $m^*[j] = m[j]$, *if* $a[j] = *$. *(Merge.)*

A r-condition classifier is then defined as the following string with ',' denoting catenation:

$$(p, c)^r, a$$

In our implementation we use two condition classifiers.

A classifier is satisfied, if all its conditions are satisfied by some messages on the current message list.

The following convention for the generation of an outgoing message is agreed upon: an outgoing message is generated from the message which satisfies the *first* condition of the classifier and from the action part of the classifier [Holland, 1986a]. However, this entails, that the first condition must not be prefixed by NOT.

Our implementation deviates from this convention with respect to two details. First, all messages matching a satisfied classifier produce output messages. Second, if a classifier contains only conditions prefixed with NOT, no message is generated.

With this machinery in place, a classifier system has the following components: the list of classifiers CL (the rule base), the message list MSG_L (the blackboard), the input interface DETECT (the detectors), and an output interface EFFECT (the effectors). The basic execution cycle of a classifier system has the following steps [Holland, 1986a] and [Holland, 1986b]:

1. Post all detector messages on the message list. (Function DETECT.)
2. "Compare all messages with all conditions and record all matches." [Holland, 1986a, p. 604] (Function RECORD_MATCH.)
3. Merge each matched message with the action part of the satisfied classifier as defined above. Replace the old message list with the new message list generated in this way. (Function GENERATE_NEW_MESSAGE.)
4. Process the message list through the output interface in order to produce messages in the environment. (Function EFFECT.)

The first step in the execution cycle requires the evaluation of the detector function DETECT. What is the task of DETECT? DETECT should read the CURRENT_STATE of the task environment and encode it as binary message. In our example shown in section 9 the CURRENT_STATE may take a value in $1, \cdots, 8$. Encoding is simple enough. We convert the state indices to binary numbers with K_BITS length which serve as detector messages. Note, however, that in line 1 we already prepare the production system for the bucket brigade algorithm by appending three zeros to each input message.

```
    ∇Z←DETECT
[1]  Z←(1 0⍉(K_BITS⍴2)⊤CURRENT_
     STATE),(0 0 0)
    ∇
```

The match step of the execution cycle requires matching all messages in the message list with all conditions in the rule base. This step is implemented by the function RECORD_MATCH with the help of the functions MATCH and GET_MSG. RECORD_MATCH takes the classifier list CL as right and the message list MSG_L as left argument and returns the match table whose first column contains the indices of messages in the message list and whose second column contains the indices of classifiers in the classifier list. The length of the information part of a message is specified by the global variable K_BITS.

```
    ∇Z←MSG_LIST RECORD_MATCH CL;N_
     LIST;C_LIST;M_TABLE;M;CL_NUMBER
[1]  C_LIST←⊂[2]((2×⍴CL),K_BITS)⍴,(⊃
     [2]CL)[;(1+⍳K_BITS),(2+K_BITS+⍳K_
     BITS)]
[2]  N_LIST←⊂[2]((2×⍴CL),1)⍴,(⊃[2]CL)[
     ;(1),(2+K_BITS)]
[3]  M_TABLE←(¯3↓¨MSG_LIST)∘.MATCH
     C_LIST
[4]  CL_NUMBER←(∧/((⍴CL),2)⍴M←('*'
     ≠∊N_LIST)≠∨/[1]M_TABLE)/⍳⍴CL
[5]  Z←((⌊0.5×⍴Z),2)⍴Z←∊GET_MSG¨CL_
     NUMBER
    ∇
```

The task of RECORD_MATCH can be divided in five smaller subtasks each of which is encoded on one line of RECORD_MATCH:

1. The first subtask is to extract all conditions from the classifier list CL. This can be achieved by a little bit of indexing and reshaping. As a result C_LIST contains the list of all conditions in CL.
2. Next we have to extract the list of all prefixes from the classifier list CL. Again, indexing and reshaping. The result N_LIST contains a list of all prefixes in CL.
3. We perform the match operation. All messages are matched with all conditions by the outer-product MATCH operation, where the function MATCH is defined in definition 3.1.
4. Our next task is to determine the indices of all satisfied conditions. We start by determining all matched conditions. In other words, there exists at least a 1 in the column of the condition in M_TABLE. Then we take the prefixes of the condition into account. If a condition is prefixed by NOT, at least a 1 in the column of M_TABLE indicates that this condition is not satisfied. Finally, a classifier is satisfied, if both of its conditions are satisfied. Now we have constructed the boolean mask with which we select the indices of the satisfied classifiers from the index vector of CL.
5. With the help of GET_MSG, we find for all satisfied classifiers all messages matching them and we record the the matches in the match table which is returned.

How does GET_MSG do its job? GET_MSG takes the index of a satisfied classifier as right argument and it selects all message indices via lookup in M_TABLE (line 1). In line 2, we test whether we have found message indices and, in line 3 we construct the message index/classifier index pairs which are returned as result in list form.

```
     ∇Z←GET_MSG CL_NUM
[1]    Z←(∨/M_TABLE[;(2×CL_NUM)-(0 1)]
       )/⍳⍴MSG_LIST
[2]    →(0=⍴Z)/0
[3]    Z←,Z,[1.5]CL_NUM
     ∇
```

The third step of the basic execution cycle is the merge phase. Matched messages are processed through the action part of the classifier and merged with the action pattern as defined in definition 3.2. The merge phase is implemented by the function GEN_NEW_MSG which does a little bit more than necessary for a pure classifier system:

1. It retrieves the message and the classifier from message and classifier list with the message/classifier index pair which is its argument.
2. It calls the function MERGE with message and action part of the classifier as argument.

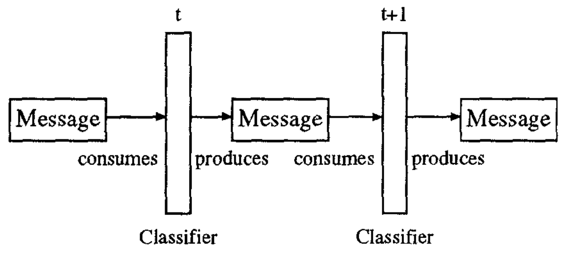

Figure 2: Classifiers as Consumers and Producers of Messages

3. The last three positions of the message are reserved for extensions. The first two of these positions are used by GEN_NEW_MSG for book-keeping. The first position is used to record the current classifier index which identifies the consumer of the message. The second position is used to record the index of the classifier which has produced this message in the previous cycle. Of course, this is the incoming message's producer, see Figure 2. In the next section this book-keeping will be the basis for the clearing of all transactions in the market.

```
     ∇Z←GEN_NEW_MSG ARG;M;MSG_N;CL_
       N
[1]    (MSG_N CL_N)←ARG
[2]    Z←((¯3↓M)OUT_MSG(-K_BITS)↑CL_
       N⊃CL),CL_N,(1↑¯3↑M←MSG_N⊃MSG_
       L),0
     ∇
```

In the last phase of the basic execution cycle the messages on the message list are processed through the output interface which is implemented by the function EFFECT.

```
     ∇Z←EFFECT MSG_LIST;MSG;E_SET;M
[1]    Z←MSG_LIST
[2]    PAYOFF←0
[3]    →(0=⍴E_SET←(((⊂'10')≡¨M)∨((
       ⊂'01')≡¨M←2↑¨MSG_LIST))/MSG_
       LIST)/0
[4]    MSG←∊1 COMPETE E_SET
[5]    ⍝ MSG←(?⍴E_SET)⊃ E_SET
[6]    Z←(MSG_LIST~E_SET),⊂MSG
[7]    MSG←1+2⊥¯1↑MSG←⍎,(2↓¯3↓MSG),[
       1.5]' '
[8]    →(~MSG∊1 2)/0
[9]    NEXT_STATE_FSW MSG
     ∇
```

In line 1 EFFECT assigns the message list as return value, in line 2 the global variable PAYOFF is set to zero, in line 3 effector messages are extracted from the message list. In our application effector messages start with 01 or 10. If the list of effector messages is empty, we have finished and no signal is sent to the environment.

In line 4 we choose the best effector message with the help of the function COMPETE described in the next section. For a pure classifier system another *conflict resolution* scheme like choosing one of the candidate messages at random (shown in the comment line 5) is equally suitable.

In line 6 we exclude the conflicting effector messages from the message list and assign the revised message list as return value. In line 7 we decode the effector message, check for feasibility of the action (line 8) and in line 9 we send the decoded effector message to our task environment, where the next state is computed by the function NEXT_STATE_FSW.

In summary, the production system of a Holland classifier system allows any number of classifiers to be active at the same time, because the only result of a firing classifier is to post a message on the global message list. This parallel firing of rules avoids the conflict resolution problems of production systems which fire one rule per execution cycle.

4 The Bucket Brigade

... and he is in this, as in many other cases, led by an invisible hand to promote an end which was no part of his intention.
Adam Smith [Smith, 1776, p. 456]

In a Holland classifier system the bucket brigade algorithm is the method of choice to tackle the apportionment of credit problem. Other approaches like profit sharing are analyzed elsewhere [Grefenstette, 1988]. The apportionment of credit problem is the problem of deciding which of the rules active at time t have been necessary and sufficient to reach a goal at time $t + n$, when several rules are active at every time step. In a game it is the problem of determining which of the early moves have been responsible for success. The apportionment of credit problem is aggravated:

1. When in a complex situation payoff is received only occasionally.
2. When the number of environmental states is so large that the system never observes the same state sequence more than once.

The bucket brigade algorithm operates by modifying the strength associated with each classifier. In our implementation, strength is stored in a real-valued vector U. The strength of a rule influences two important aspects of a classifier system:

1. The strength of a classifier influences which classifiers may become active and thus the short term behavior of the system.
2. The strength of a classifier serves as fitness function of the genetic algorithm and influences thus the long term behavior of the system.

Therefore, it is of high importance for the efficiency of the algorithm that strength is allocated so as to reflect the "true" fitness of rules.

The bucket brigade algorithm integrates an economic system into the classifier production rule interpreter of the previous section. The principle idea is that the allocation of strength problem can be efficiently solved by market mechanisms or bargaining procedures. Market mechanisms are explained in the early literature [Arrow and Hahn, 1971], whereas more recent literature illustrates more abstract versions of the same mechanisms [Aubin, 1979, Aliprantis *et al.*, 1990]. In the economic system created by the bucket brigade algorithm, classifiers and the environment are treated as economic agents which produce and consume a commodity, namely messages. The strength of a rule can be regarded as the agent's capital or wealth. The market mechanism is an auction in which agents compete for the right to place their message on a message list which is restricted in size.

The auction process cycles through the following steps:

1. **(Bidding.)** Agents make a bid for the right to put messages onto the message list.
2. **(Competition.)** A competition is run to determine the winning agents. The probability that an agent will win is proportional to its relative bid.
3. **(Clearing.)** Transactions are cleared:
   (a) Agents may receive payment from the environment, if they are among the winning agents.
   (b) They must pay their bids to their suppliers, if they are among the winning agents.
   (c) They receive payments from their consumers which are among the winning agents.

In detail, the *bid* $b_{i,t}$ of a classifier c_i is a function of its current strength $u_{i,t}$, a constant r_{const} which determines the percentage of current strength the classifier is prepared to lose in one auction and the specificity s_i which is the (constant) proportion of non-wild card symbols in the condition part of the classifier. The bid $b_{i,t}$ of a classifier c_i is then calculated as follows [Holland, 1986a]:

$$b_{i,t} = (r_{const} \cdot s_i) \cdot u_{i,t} \qquad (1)$$

with $0 < r_{const} \leq 1$, $0 \leq s_i \leq 1$. r_{const} is comparable to the learning rate in connectionist algorithms and should be quite small.

The *specificity* of a classifier c_i is defined as the number of 0s or 1s in the condition parts of c_i divided by the length of the condition parts of c_i. The specificity

s_i is constant throughout the life-time of the classifier c_i. The specificity categorizes the classifiers in a hierarchy of professionals with differing degree of specialization. The ultimate generalist which matches every message has a specificity of 0, the ultimate specialist which matches only one message has a specificity of 1. The specificity s_i ensures that specialists tend to be preferred over generalists, because specialists offer higher bids. This supports the emergence of default hierarchies, because the specialists protect the generalists, if they apply.

We have implemented formula 1 in the function BID whose first two lines serve for the calculation of the specificity. In line 3 of BID we replace the last value in the message with the value of the bid just computed.

```
    ∇Z←BID MSG;SPECIFICITY;MASK;CL_
      NUM
[1]   MASK←0,(K_BITSρ1),0,(K_BITSρ1),(
      K_BITSρ0)
[2]   SPECIFICITY←(+/~'⋆'=MASK/(CL_
      NUM←1↑¯3↑MSG)⊃CL)÷(2×K_BITS)
[3]   Z←(¯1↓MSG),(BID_CONST×U[CL_
      NUM]×SPECIFICITY)
    ∇
```

Satisfied classifiers compete for places on the message list. The probability that a satisfied classifier c_i will win is [Riolo, 1987a]:

$$P(c_i \text{ wins}) = \frac{\beta_{i,t}}{\sum_{j=1}^{n} \beta_{j,t}}, \tag{2}$$

where $\beta_{i,t}$ is the *effective bid* of the classifier c_i. The effective bid $\beta_{i,t}$ is calculated from the bid $b_{i,t}$ as follows:

$$\beta_{i,t} = b_{i,t}^{b_{pow}} \tag{3}$$

The parameter b_{pow} is in our implementation denoted by the global variable BID_POWER. Increasing b_{pow} to $2, 3, \ldots$ increases the probability that the classifier with the highest bids will win the competition. This competition algorithm is implemented in the function COMPETE:

```
    ∇Z←LIST_SIZE COMPETE LIST;P
[1]   Z←LIST
[2]   →(LIST_SIZE≥ρLIST)/0
[3]   P←+\(÷+/P)×P←((∊¯1↑¨LIST)+((ρ
      LIST)?ρLIST)÷100000000)⋆BID_POWER
[4]   Z←LIST[∊((?LIST_SIZEρ(¯1+2⋆31))÷2⋆
      31)IN¨⊂P]
    ∇
```

The function COMPETE takes the list of competing messages as right argument and the maximum size of the message list as left argument. It returns a message list whose length is at most of maximum size. Lines 1 and 2 cover the case when no competition is needed because of a lack of competitors. In line 3 the cumulative probability distribution is calculated and in line 4 the winners are drawn. Note, however, that we use the same procedure for conflict resolution in the output interface, with a maximum list size of 1.

Finally, all transactions are cleared. In our implementation the update of strength is computed in an economy with a tax system. We tax the income of a classifier to suppress free-riders, and we tax wealth, in order to eliminate rules which are never invoked.

Let us start with our explanation of the transaction clearing process by repeating what the last three positions of a message contain:

1. The last position, position $k+3$ of a k-symbol message string, contains the bid associated with this message.
2. Position $k+2$ contains the index of the classifier which produced the father of the message (see Figure 2).
3. Position $k+1$ contains the index of the classifier who consumed the father of the message and who offered the bid.

We denote the set of winning classifiers by *win*, and a classifier from this set by c_w. We denote the set of producers by *prod*, and a classifier from this set by c_p. We denote a classifier which is neither in the winning nor in the producing set by c_n and we designate payments from the environment by $u_{e,t}$. Our tax system has two taxes, with an income tax t_I on all payments from other classifiers, and a property tax t_P for all classifiers which are neither winners or producers. The update of strength is then calculated as follows:

$$u_{w,t+1} = u_{w,t} + \frac{u_{e,t}}{card(win)} - b_{w,t} \tag{4}$$

$$u_{p,t+1} = u_{p,t} + b_{p,t} \cdot (1 - t_I) \tag{5}$$

$$u_{n,t+1} = u_{n,t} \cdot (1 - t_P) \tag{6}$$

This update rules are implemented in the function UPDATE_U which is shown below.

```
    ∇Z←S UPDATE_U MSG_LIST;WIN;SEND;
      BIDS;M;UWIN;USEND;WBID;SBID;GAIN
[1]   Z←⊃[2]MSG_LIST
[2]   UWIN←UNIQUE WIN←Z[;K_BITS+1]
[3]   USEND←UNIQUE SEND←Z[;K_BITS+2]
[4]   WBID←(UWIN∘.=WIN)+.×BIDS←Z[;
      K_BITS+3]
[5]   SBID←(USEND∘.=SEND)+.×BIDS
[6]   Z←S
[7]   Z[M/UWIN]←S[M/UWIN]-(M←~0=
      UWIN)/WBID
[8]   Z[M/USEND]←Z[M/USEND]+(1-I_
      TAX)×(M←~0=USEND)/SBID
```

```
[9]    GAIN←(UWIN∘.=WIN)+.×GAIN←(⍴
       WIN)⍴(PAYOFF÷⍴WIN)
[10]   Z[M/UWIN]←Z[M/UWIN]+(M←~0=
       UWIN)/GAIN
[11]   M←(⍳⍴Z)~(UNIQUE(UWIN,USEND)~0)
[12]   Z[M]←(1-V_TAX)×Z[M]
[13]   CUM_PAYOFF←CUM_PAYOFF,PAYOFF
     ∇
```

In lines 1 to 5 the index sets for winners and producers and the bids they have to pay and the payments they should receive are calculated. In line 7 the winners pay their bids. In line 8 the producers receive their payments with the income tax already deducted. In lines 9 and 10 the payoff from the environment is distributed to the winners. In lines 11 and 12 the classifiers which are neither winners nor producers pay their property tax and in line 13 we record the payoff in the vector CUM_PAYOFF.

The features of the bucket brigade algorithm which make it attractive are [Riolo, 1987a]:

1. It uses only local information. We need to know only the classifiers which are active on the message list (the winners) and the classifiers which directly activated them (the producers). There is no need for additional book-keeping or for high-level critics which analyze sequences of actions to assign credit properly.
2. It works in parallel updating the strength of rule sets at the same time.
3. It changes the strength of classifiers gradually. This supports the tendency of the system to learn gracefully without drastic changes in performance due to a single outlier.

5 The Genetic Algorithm

The genetic algorithm is the rule discovery component of a Holland classifier system. It treats the classifier list as string population with the classifier strengths as fitness function. The integration of the genetic algorithm into our system is straightforward, the genetic algorithm is invoked every k cycles of the interpreter.

GEN_COND initializes classifiers with fixed length strings from the alphabet $0, 1, *$. The initial distribution of symbols is defined by P_GSYM_0 and P_GSYM_1.

```
     ∇Z←GEN_COND K;A;B
[1]    Z←((100-(A+B))⍴'*'),((B←⌊P_
       GSYM_1×100)⍴'1'),((A←⌊P_GSYM_
       0×100)⍴'0')
[2]    Z←Z[?K⍴100]
     ∇
```

The function GENETIC which uses a simple genetic algorithm to compute the next generation has the following steps:

1. In line 2 it selects a part of the population at random. The size of this part is controlled by GEN_SHARE. The function NEXT_STATE takes this subpopulation and its rule strength vector as fitness as arguments and generates by application of genetic operators a new generation of classifiers.
2. GENETIC tests for duplicates in line 3. For each duplicate a new random classifier is generated by GEN_COND and the number of duplicates is reported in CREATED.
3. In line 4 we replace the worst members in the population with the subpopulation generated by the genetic allgorithm and we add the random classifiers generated in the previous line. The strength of the new classifiers is initialized with the mean rule strength in the population.
4. In line 5 we delete all duplicates. Finally, in line 6 we add an empty message list as third element of the function's result.

```
     ∇Z←GENETIC ARG;P0;P1;P2;P;S0;S;CL;
       U;S1
[1]    (CL U)←ARG
[2]    P2←NEXT_STATE(CL[S])(U[S←(S1←⌊
       POP_SIZE×GEN_SHARE)?⍴U])1 1
[3]    Z←GEN_COND¨(CREATED←0⌈POP_
       SIZE-⍴UNIQUE((S1↑CL[⍒U]),P2))⍴(
       2+K_BITS×3)
[4]    Z←((S1↑CL[P1]),P2,Z)(((S1←POP_
       SIZE-S1)↑U[P1←⍒U]),((⍴P2)+⍴Z)⍴(
       +/U[S])÷⍴S)
[5]    P1←(1⊃Z)[S←POP_SIZE↑UNIQUE_I 1⊃
       Z]
[6]    Z←P1((2⊃Z)[S])(0⍴0)
     ∇
```

The function UNIQUE_I returns the indices of the unique elements in the right argument vector, UNIQUE is defined as usual.

```
     ∇Z←UNIQUE_I S
[1]    Z←((S⍳S)=⍳⍴S)/⍳⍴S
     ∇
```

```
     ∇Z←UNIQUE S
[1]    Z←((S⍳S)=⍳⍴S)/S
     ∇
```

For a simple genetic algorithm NEXT_STATE breeds the new generation by combining three genetic operators, namely replication, crossover and mutation. In the first phase strings are replicated (copied) by the replication function REPLICATE according to the outcome of a Darwinian selection process. Only the best survive. In this process the probability of survival of a string usually is proportional to its relative fitness. In the next phase pairs of strings are mated by the crossover function CROSS with probability P_CROSS. In the last phase single bit mutation may be applied with probability P_MUT to a string in the population. Single bit mutation is implemented by the function MUT.

```
    ∇Z←NEXT_STATE ARG;POP;FIT;I;K
[1]   (POP FIT I K)←ARG
[2]   Z←P_MUT MUT P_CROSS CROSS(FIT
      REPLICATE POP)
    ∇
```

In REPLICATE we draw the survivors of the next generation by indexing the population vector POP. For the calculation of the indices of the survivors we need two pieces of information, a vector of S random numbers, independently drawn from $[1, 2^{31} - 1]$ with equal probability, and the cumulative distribution function of the probability of survival mapped from $[0, 1]$ to $[0, 2^{31}]$. We get the first by drawing S random numbers from the integers in 1 to $2^{31} - 1$ with replacement with equal probability in line 1 of REPLICATE. The second piece, slots with size proportional to fitness, is computed in line 2 of REPLICATE.

```
    ∇Z←FIT REPLICATE POP;N;F
[1]   N←?(⍴FIT)⍴(¯1+2*31)
[2]   F←(+\FIT÷+/FIT←0.0001+FIT+(¯1×
      ⌊/FIT))×2*31
[3]   Z←POP[∊N IN¨⊂F]
    ∇
```

The function IN called in line 3 of REPLICATE calculates the slot number of the random number N from the vector of slot boundaries F:

```
    ∇Z←N IN F
[1]   Z←1↑(N≤F)/⍳⍴F
    ∇
```

On the string level the one-bit mutation function MUTATE randomly selects a bit of the string and replaces it with either a 0, a 1, or a $*$. This is controlled with the variables P_MSYM_0 and P_MSYM_1 which are set by the function INIT_HCS which is described in section 8.

```
    ∇Z←MUTATE S;A;B
[1]   Z←((100-(A+B))⍴'*'),((B←⌊P_
      MSYM_1×100)⍴'1'),((A←⌊P_MSYM_
      0×100)⍴'0')
[2]   S[?⍴S]←Z[?100]
[3]   Z←S
    ∇
```

In a genetic algorithm, *rolling the dice* is the standard procedure to decide if a string should be modified by a genetic operator or not. However, in general we need the functionality to randomize the execution of an arbitrary function. This means, the function is either executed with a certain probability or the arguments are passed without change. This is implemented by the operator DICE which maps functions to randomized functions.

```
    ∇Z←R(F DICE G)L
[1]   Z←L
[2]   →((?¯1+2*31)>F×2*31)/0
[3]   →(0=⎕NC 'R')/MON
[4]   Z←R G L
[5]   →0
[6]   MON:Z←G L
    ∇
```

The function MUT rolls the dice for each string of the population POP with probability PROB.

```
    ∇Z←PROB MUT POP
[1]   Z←(PROB DICE MUTATE)¨POP
    ∇
```

The function CROSS rolls the dice for each string of the population POP with probability PROB and randomizes the application of the crossover operator.

```
    ∇Z←PROB CROSS POP
[1]   Z←POP[(⍴POP)?⍴POP](PROB DICE
      CROSSOVER)¨POP
    ∇
```

Now, let us turn to the work-horse of a genetic algorithm, the crossover operator. The crossover operator creates a new string by recombining two parent strings in the following way. All bits up to a randomly selected position are from the first parent, the rest is from the second parent. In line 1 of CROSSOVER we select the position and create a Boolean mask, in line 2 we catenate the two selections we have made with the mask from the first and the second string.

```
    ∇Z←S1 CROSSOVER S2
[1]   Z←(⍳⍴S1)≤?⍴S1
[2]   Z←(Z/S1),(~Z)/S2
    ∇
```

6 Cover Detector and Cover Effector Operator

When starting with random patterns as classifiers, in practice the problem that the classifiers do not match any detector message or do not activate any effector message often occurs. In this situation the bucket brigade algorithm does not receive any payoff and the rule strengths will not reveal any useful information to the genetic algorithm. To solve this problem, Robertson and Riolo implement two additional operators [Robertson and Riolo, 1988].

The cover detector operator CDO is triggered whenever a detector message is not matched by any classifier. It constructs a classifier which matches with a random action part in line 1 and replaces the classifier with the lowest strength in the system with the new classifier in lines 3, 4, and 5.

```
    ∇Z←CDO A;CL;U;MSG_L;D_MSG;J
[1]   (CL U MSG_L)←A
[2]   A←'*',(K_BITS↑,⊃¯1↑MSG_L),((1+
      K_BITS)⍴'*'),(GEN_COND K_BITS)
[3]   J←(?⍴J)⊃J←(U=⌊/U)/⍳⍴U
[4]   CL[J]←⊂A
[5]   U[J]←(+/U)÷⍴U
[6]   Z←(CL U(¯1↓MSG_L))
    ∇
```

In line 2 the cover effector operator CEO is triggered whenever no effector is activated. It chooses one of the classifiers which match a message in line 3 and constructs new classifier which matches and activates a random effector action. Again, the classifier with the lowest strength in the system is replaced by the new classifier in lines 4, 5, and 6.

```
    ∇Z←CEO MSG_LIST;M;J;K;MSG
[1]   Z←MSG_LIST
[2]   →(0≠⍴(((⊂'10')≡¨M)∨((⊂'01')≡¨
      M←2↑¨MSG_LIST))/MSG_LIST)/0
[3]   K←(K_BITS+1)⊃(MSG←∊1 COMPETE
      MSG_LIST)
[4]   J←(?⍴J)⊃J←(U=⌊/U)/⍳⍴U
[5]   CL[J]←⊂((2+2×K_BITS)↑K⊃CL),M←
      '10',('01'[?(K_BITS−2)⍴2])
[6]   U[J]←(+/U)÷⍴U
[7]   Z←Z,⊂(M,J,MSG[K_BITS+1+⍳2])
    ∇
```

7 Triggered Chaining Operator

We say two classifiers are *coupled*, if a message produced by the first classifier satisfies a condition of the second classifier. Such chains of coupled classifiers are necessary to implement arbitrary computations or short term memory. To improve the learning capability of a classifier system, Holland, Holyoak, Nisbett, and Thagard suggest the triggered chaining operator TCO for generating coupled classifiers [Holland *et al.*, 1986].

The triggered chaining operator TCO is invoked, whenever payoff is received from the environment in line 2. From a succesful classifier and a classifier active in the previous step, it constructs two new classifiers which are coupled and it replaces the two weakest classifiers.

```
     ∇Z←TCO MSG_LIST;MSG;M;W;V
[1]    Z←MSG_LIST
[2]    →(PAYOFF=0)/0
[3]    MSG←'00','01'[?(K_BITS−2)⍴2]
[4]    V←(K_BITS+1)⊃M←∊1 COMPETE
       MSG_LIST
[5]    →(0=⍴W←(~(W=M[K_BITS+2])∨(
       W=0))/W←(⊃[2]MSG_LIST)[;K_BITS+
       2])/0
[6]    W←(?⍴W)⊃W
[7]    W←((2+2×K_BITS)↑W⊃CL),MSG
[8]    V←((1+K_BITS)↑V⊃CL),'*',MSG,((
       2+2×K_BITS)↓V⊃CL)
[9]    M←2↑⍋U
[10]   CL[M]←V W
[11]   U[M]←2⍴(+/U)÷⍴U
     ∇
```

8 The Interpreter

Finally we implement the interpreter HCS which combines the classifier system, the bucket brigade, the genetic algorithm, cover detector, cover effector and the triggered chaining operator. Clearly, most of the code is no surprise.

```
     ∇HCS I;M
[1]    INIT_FSW
[2]    ⎕RL←RL←(24 60 60⊥3↓¯1↓⎕TS)
[3]    INIT_HCS
[4]    LOOP:→(0∊⍴M←(MSG_L←MSG_L,
       ⊂DETECT)RECORD_MATCH CL)/
       DCOVER
[5]    U←U UPDATE_U MSG_L←(P_TCO DICE
       TCO)EFFECT CEO MAX_L COMPETE
       BID¨GEN_NEW_MSG¨⊂[2]M
[6]    →(0≥I←I−1)/0
[7]    →(0≠B_STEPS|I)/LOOP
[8]    DISCOVER:(CL U MSG_L)←GENETIC(
       CL U)
[9]    →LOOP
[10]   DCOVER:(CL U MSG_L)←CDO(CL U
       MSG_L)
[11]   →LOOP
     ∇
```

The new code integrates the following components:

1. The initialization of the finite state world in line 1.

2. The seeding of the random number generator in line 2.
3. The initialization of the classifier data structures and of the system's control parameters in line 3.
4. The basic execution cycle of the bucket brigade interpreter with cover effector and triggered chaining operator in lines 4 and 5.
5. The termination condition in line 6.
6. The invocation condition of the genetic algorithm in line 7.
7. The genetic algorithm in line 8 and the jump back to the basic execution cycle in line 9.
8. The cover detector operator with the jump back to the basic execution cycle in lines 10 and 11.

However, before we can start the system we have to initialize the classifier's system parameters, the classifier list, the message list, and the initial capital endowment of the classifiers.

```
    ∇INIT_HCS
[1]   (MAX_L POP_SIZE K_BITS)←4 50 12
[2]   (BID_CONST BID_POWER I_TAX V_TAX
      BUDGET)←0.1 2 0.05 0.01 100
[3]   (P_GSYM_0 P_GSYM_1)←0.4 0.4
[4]   (P_MSYM_0 P_MSYM_1)←0.4 0.4
[5]   (B_STEPS GEN_SHARE CREATED P_
      MUT P_CROSS)←200 0.2 POP_SIZE 0.5
      1
[6]   P_TCO←0
[7]   CL←GEN_COND¨POP_SIZEρ(2+K_
      BITS×3)
[8]   U←POP_SIZEρBUDGET
[9]   MSG_L←0ρ0
    ∇
```

The function INIT_HCS initializes the global data structures of the classifier system as follows:

1. In line 1 we create the parameters of the production system, namely, MAX_L, the maximum size of the message list, POP_SIZE, the number of rules in the classifier list, K_BITS, the length of a message, a condition and an action.
2. In line 2 the parameters of the bucket brigade algorithm are initialized, namely BID_CONST, the proportion of capital risked in one bid, BID_POWER, the power to which a bid is raised in the calculation of the effective bid, and last but not least two tax rates, I_TAX, the income tax rate, V_TAX, the property tax rate, and the initial BUDGET of a rule.
3. In lines 3 and 4 the initial symbol distribution of the classifiers and the genetic drift of the mutation operator are specified.
4. In line 5 the parameters of the genetic algorithm are initialized. They are B_STEPS, the number of execution cycles between two invocations of the genetic algorithm, GEN_SHARE, the proportion of the population replaced by offsprings in an invocation of the genetic algorithm, CREATED, a variable which reports the number of rules which have been generated at random in the last invocation of the genetic algorithm, and P_MUT and P_CROSS, the probabilities of invoking the mutation and crossover operator.
5. The probability of applying the triggered chaining operator P_TCO is specified in line 6.
6. In line 7 a new random classifier list is generated by the function GEN_COND.
7. The strength vector U is set to the initial capital endowment of the rules, 100 in our experiment (line 8).
8. We initialize the message list MSG_L as empty vector (line 9).

9 The Task Environment

In the experiment illustrating the working of a Holland classifier system, we have implemented a simple, but useful, task environment which allows the simulation of finite state Markov processes. Roughly speaking, a Markov process is a process with the property that the probability of any particular future behavior of the process, when its present state is known exactly, does not depend on its past behavior. For example, consider a mouse moving through the maze shown in Figure 3. Assume that the mouse moves from room to room by choosing at random one of the doors available with equal probability and that the mouse changes rooms at specified times. From these assumptions it is obvious that the probability that the mouse visits a certain room at time n depends only on his location at time $n-1$ and not on his location on earlier times.

Since finite state Markov processes serve as models of real processes over a variety of fields, from brand-choice models in marketing to inventory control, load balancing in distributed computer systems, and queuing theory, success in learning the abstract model indicates applicability in all domains where finite state Markov

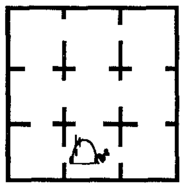

Figure 3: A Mouse in a Maze

processes are useful models of the real world. Therefore, learning of Markov processes seems to be quite an interesting task environment. Moreover, an elegant theory has been developed to explain Markov processes [Howard, 1971]. As a final argument, the task domain has been used in a series of experiments on the performance of the bucket brigade algorithm [Riolo, 1987a, Riolo, 1987b, Riolo, 1989a, Riolo, 1989b].

In principle, we model the environment as a finite state Markov process in which a payoff is associated with some states. The classifier system's detector (the input interface) reads the current state of the Markov process and delivers a message with the current state to the classifier system. The classifier system's output interface (the effector) can choose from a set of predefined probability transition matrices, so that the classifier system can control (at least in part) the path through the finite state world. By visiting states with non-zero payoffs the system accumulates rewards. The task of the classifier system is to learn to emit a sequence of actions, so that the rewards are maximized.

Formally, a finite state world is defined by the specification of:

1. A set of n states S_i, $i = 1, \cdots, n$, each with a payoff $U(S_i) \in R$, and one state designated as start state.
2. A set of probability transition matrices $P(r)$, with $r = 1, \cdots, m$. Each element $p_{i,j}(r)$ in $P(r)$ denotes the probability of going to state S_i, provided the process is in state S_j and the classifier action has been r. Note that in most standard sources (e.g. [Howard, 1971]) the first index denotes the start state and the second index the end state of a transition.

In our experiment we have used the eight state world depicted in Figure 4. S_1 is the start-state, and arrows labelled with the classifier actions 1 or 2 designate system transitions with a probability of 1 (deterministic), if the appropriate action is chosen by the classifier system. Arrows labelled with a 1 and a 2 indicate that

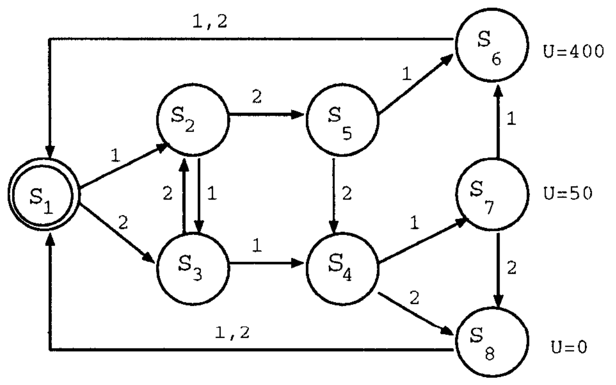

Figure 4: The Finite State World

a system transition occurs, if either action 1 or action 2 occurs. The specification of this finite state world is implemented by the function INIT_FSW shown below.

```
     ∇INIT_FSW
[1]   CURRENT_STATE←START_STATE←1
[2]   U_STATES←0 0 0 0 0 400 50 0
[3]   PAYOFF←0
[4]   P_TRANS←2 8 8ρ0
[5]   P_TRANS[1;1;]←0 0 0 0 0 1 0 1
[6]   P_TRANS[1;2;]←1 0 0 0 0 0 0 0
[7]   P_TRANS[1;3;]←0 1 0 0 0 0 0 0
[8]   P_TRANS[1;4;]←0 0 1 0 0 0 0 0
[9]   P_TRANS[1;5;]←0 0 0 0 0 0 0 0
[10]  P_TRANS[1;6;]←0 0 0 0 1 0 1 0
[11]  P_TRANS[1;7;]←0 0 0 1 0 0 0 0
[12]  P_TRANS[1;8;]←0 0 0 0 0 0 0 0
[13]  P_TRANS[2;1;]←0 0 0 0 0 1 0 1
[14]  P_TRANS[2;2;]←0 0 1 0 0 0 0 0
[15]  P_TRANS[2;3;]←1 0 0 0 0 0 0 0
[16]  P_TRANS[2;4;]←0 0 0 0 1 0 0 0
[17]  P_TRANS[2;5;]←0 1 0 0 0 0 0 0
[18]  P_TRANS[2;6;]←0 0 0 0 0 0 0 0
[19]  P_TRANS[2;7;]←0 0 0 0 0 0 0 0
[20]  P_TRANS[2;8;]←0 0 0 1 0 0 1 0
[21]  (CUM_PAYOFF ROUTE)←(1ρ0)(1ρ1)
     ∇
```

The function INIT_FSW initializes a set of global variables as follows:

1. The CURRENT_STATE and the START_STATE are set to 1. The variable CURRENT_STATE is read by the classifier system's input interface and translated into a message.
2. Since the payoff of the start-state of our example is 0, the variable PAYOFF is initialized with 0. The variable PAYOFF shows the payoff associated with the current state and is read by the bucket brigade algorithm.
3. U_STATES is the vector of the payoff of the states.
4. P_TRANS is in our experiment a 2 times 8 times 8 array. The subarray P_TRANS[1;;] is the probability transition matrix selected by action 1, P_TRANS[2;;] is selected by action 2.
5. CUM_PAYOFF and ROUTE record payments and visited nodes, respectively.

For our specification to work, we need a function which calculates the next state in our finite state world.

```
    ∇NEXT_STATE_FSW R;CPROB
[1]   CPROB←+\P_TRANS[R;;CURRENT_
      STATE]
[2]   CURRENT_STATE←(((?¯1+2*31)÷2*
      31)IN CPROB)⊃ιρU_STATES
[3]   PAYOFF←U_STATES[CURRENT_STATE]
    ∇
```

The function NEXT_STATE_FSW takes the action of the classifier system as its right argument. The actions act as indices into the array P_TRANS, so that a probability transition matrix can be chosen. The function NEXT_STATE_FSW has the following tasks:

1. It selects the probability distribution depending on the action R and on the CURRENT_STATE and calculates a cumulative probability distribution CPROB (line 1).
2. It chooses the next setting of CURRENT_STATE according to the cumulative probability distribution CPROB (line 2).
3. It sets the variable PAYOFF to the payoff associated with the CURRENT_STATE and reports the results.

In one run of 10,000 execution cycles with the parameters of the classifier system as shown in function INIT_HCS, the system reached an average payoff of 79.67 units for the finite state world shown in Figure 4. The average payoff in the last 500 execution cycles was 89.2. Compared to a recent version [Geyer-Schulz, 1995], this result constitutes a 15 percent performance increase. The improvement is due to the addition of the cover detector and cover effector operators. The best average payoff for this example is 100 units per execution cycle. The development of the average payoff per transaction over the run is shown in Figure 5. An analysis of the routes which have been learned indicate

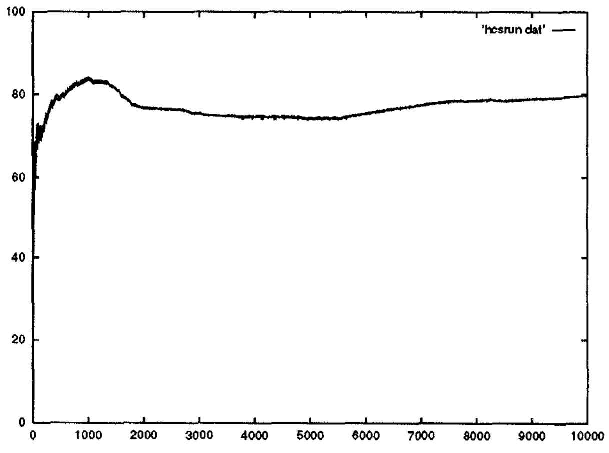

Figure 5: The average payoff per transaction

that the classifier system moved most of the time along the second and third best route of the world (1,044 and 698 times, respectively), whereas the optimal route 1, 2, 5, 6, 1 has been traversed only 7 times. However, no attempt to find optimal parameters for the classifier system has been made in this experiment.

10 Summary

Questions abound.
J. H. Holland [Holland, 1986a, p. 622]

In this paper a *full* reconstruction of Holland classifier systems in APL has been presented. It is based on chapters 7.2 and 9.2 in a recent publication [Geyer-Schulz, 1995]. The system can easily be tailored to other machine-learning tasks, e.g. learning order heuristics in MIT's beer game [Sterman, 1989] or learning price, capacity and marketing decisions when introducing innovative products in booming markets [Paich and Sterman, 1993]. Since the system can run on a PC, it offers a low cost chance to teach the inner workings of one of the most promising machine learning approaches and to design machine learning experiments in new application domains. Compared to simple genetic algorithms, however, classifier systems are difficult to parametrize and the interrelationship of their components and the influence on learning performance are still poorly understood.

References

[Aliprantis *et al.*, 1990] Charalambos D. Aliprantis, Donald J. Brown, and Owen Burkinshaw. *Existence and Optimality of Competitive Equilibria.* Springer, Berlin, 1990.

[Arrow and Hahn, 1971] Kenneth J. Arrow and F. H. Hahn. *General Competitive Analysis*, volume 6 of *Mathematical Economics Texts.* Holden-Day, Inc., San Francisco, 1971.

[Aubin, 1979] Jean Pierre Aubin. *Mathematical Methods of Game and Economic Theory*, volume 7 of *Studies in Mathematics and its Applications.* North-Holland, Amsterdam, 1979.

[Farmer *et al.*, 1986] Doyne Farmer, Alan Lapedes, Norman Packard, and Burton Wendroff, editors. *Evolution, Games and Learning: Models for Adaptation in Machines and Nature.* North-Holland, Amsterdam, 1986. Reprinted from Physica D, Vol. 22D(1986), Nos. 1-3.

[Geyer-Schulz, 1995] Andreas Geyer-Schulz. *Fuzzy Rule-Based Expert Systems and Genetic Machine*

Learning, volume 3 of *Studies in Fuzziness*. Physica-Verlag, Heidelberg, 1995. 431p.

[Goldberg, 1989] David E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[Grefenstette, 1987] John J. Grefenstette, editor. *Genetic Algorithms and their Applications.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1987.

[Grefenstette, 1988] J. J. Grefenstette. Credit assignment in rule discovery systems based on genetic algorithms. *Machine Learning*, 3(2/3):225–246, 1988.

[Holland and Reitman, 1978] John H. Holland and Judith S. Reitman. Cognitive systems based on adaptive algorithms. In Waterman and Hayes-Roth [1978], pages 313–329.

[Holland *et al.*, 1986] John H. Holland, Keith J. Holyoak, Richard E. Nisbett, and Paul R. Thagard. *Induction: Processes of Inference, Learning, and Discovery.* Computational Models of Cognition and Perception. The MIT Press, Cambridge, Massachusetts, 1986.

[Holland, 1986a] John H. Holland. Escaping brittleness: The possibilities of general–purpose learning algorithms applied to parallel rule-based systems. In Michalski et al. [1986], pages 593–623.

[Holland, 1986b] John H. Holland. A mathematical framework for studying learning in a classifier system. In Farmer et al. [1986], pages 307–317. Reprinted from Physica D, Vol. 22D(1986), Nos. 1–3.

[Howard, 1971] Ronald A. Howard. *Dynamic Probabilistic Systems. Volume I: Markov Models*, volume 1 of *Series in Decision and Control.* John Wiley & Sons, New York, 1971.

[Koza, 1992] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, Cambridge, Massachusetts, 1992.

[Michalski *et al.*, 1986] Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors. *Machine Learning*, volume II. Morgan Kaufmann, Los Altos, 1986.

[Minsky, 1971] Marvin L. Minsky. *Berechnung: Endliche und unendliche Maschinen.* Verlag Berliner Union GmbH, Stuttgart, 1971.

[Paich and Sterman, 1993] Mark Paich and John D. Sterman. Boom, bust, and failures to learn in experimental markets. *Management Science*, 39(12):1439–1458, December 1993.

[Riolo, 1987a] Rick L. Riolo. Bucket brigade performance: I. Long sequences of classifiers. In Grefenstette [1987], pages 184–195.

[Riolo, 1987b] Rick L. Riolo. Bucket brigade performance: II. Default hierarchies. In Grefenstette [1987], pages 196–201.

[Riolo, 1989a] Rick L. Riolo. The emergence of coupled sequences of classifiers. In Schaffer [1989], pages 256–264.

[Riolo, 1989b] Rick L. Riolo. The emergence of default hierarchies in learning classifier systems. In Schaffer [1989], pages 322–327.

[Robertson and Riolo, 1988] G. G. Robertson and R. L. Riolo. A tale of two classifier systems. *Machine Learning*, 3(2/3):139–160, 1988.

[Schaffer, 1989] J. David Schaffer, editor. *Proceedings of the Third International Conference on Genetic Algorithms.* Morgan Kaufmann Publishers, Inc, San Mateo, California, 1989.

[Smith, 1776] Adam Smith. *An Inquiry into the Nature and Causes of the Wealth of Nations.* W. Strahan and T. Cadell, London, 1st edition, 1776. Quoted from the LibertyClassics 1981 edition of Liberty Press, Indianapolis which is a reprint of the Glasgow edition of the works and correspondence of Adam Smith, volume 2, edited by R. H. Campbell and A. S. Skinner, Oxford, Clarendon Press, 1979.

[Sterman, 1989] John D. Sterman. Modeling managerial behavior: Misperceptions of feedback in a dynamic decision making experiment. *Management Science*, 35(3):321–339, March 1989.

[Waterman and Hayes-Roth, 1978] Donald A. Waterman and Frederick Hayes-Roth, editors. *Pattern-Directed Inference Systems.* Academic Press, Orlando, 1978.