Architecture Design and Simulation for Distributed Learning Classifier Systems

by

Douglas G. Gaff

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

APPROVED:

John S. Bay, Chair

Charles E. Nunnally

Hugh F. VanLandingham

May 1995

Blacksburg, Virginia

Keywords: Learning Classifier Systems, Distributed Artificial Intelligence, Robotics, Network Communications, The Animat Problem

ARCHITECTURE DESIGN AND SIMULATION FOR DISTRIBUTED LEARNING CLASSIFIER SYSTEMS

by

Douglas G. Gaff

John S. Bay, Chairman

Bradley Department of Electrical Engineering

(ABSTRACT)

In this thesis, we introduce the Distributed Learning Classifier System (DLCS) as a novel extension of J. H. Holland's standard learning classifier system. While the standard LCS offers effective real-time control and learning, one of its limitations is that it does not provide a mechanism for allowing communication between LCS agents in a multiple-agent scenario. Often multiple-agents are used to solve large tasks collectively by subdividing the task into smaller parts. Multiple agents can also be used to solve a task in parallel so that a solution can be arrived at more rapidly. With the DLCS, we introduce mechanisms that satisfy both of these cases, while still providing compatible operation with the LCS.

We introduce three types of messages that can be passed between DLCS agents. The first, the classifier message, allows agents to share learned information with one another, thereby helping agents benefit from each other's successes. The second, the action message, allows agents to "talk" to one another. The third, the bucket brigade algorithm payoff message, extends the chain rewarding payoff scheme of the standard LCS to multiple DLCS agents.

Finally, we present some simulation results for both the standard LCS and the DLCS. Our LCS simulations examine some of the important aspects of learning classifier system operation, as well as illustrate some of the shortcomings. The DCLS simulations justify the distributed architecture and suggest future directions for achieving learning among multiple agents.

Acknowledgment

First of all, I would like to thank Dr. John Bay for being my advisor throughout my introduction to the world of artificial intelligence. His unique sense of humor has been an enjoyable complement to my own, and I have greatly enjoyed working with him. Second, I would like to thank Dr. Charles Nunnally for being on my committee and for being a mentor and friend throughout my undergraduate and graduate years here at Virginia Tech. His support and guidance have been appreciated more than I can possibly verbalize here. I would also like to thank Dr. Hugh VanLandingham for being a member of my committee. While Dr. V. and I are new acquaintances, his support of my research is very much appreciated. Finally, I would like to thank Dr. Warren Stutzman, an ex-officio committee member. Dr. Stutzman has also been a mentor and friend throughout my time here at Virginia Tech. I very much appreciate the opportunities he has presented me throughout my educational career.

Next, I'd like to thank my parents, Dale and Barbara, for their support of my educational endeavors. I could not have accomplished what I have without their love and guidance. Finally, to my fellow armyant brats, thanks for being such cool office-mates and for your patience with my monopolizing of the PC. And last, but not least, my satcom buddies. Thanks for the many, many memorable experiences. I think we're about due for another power lunch.

This thesis was supported in part by the Naval Research Laboratory under grant no. N00014-93-1-G022 and in part by the Office of Naval Research under grant no. N00014-94-1-0676

Table of Contents

List of Figures

List of Tables

1. Introduction

In Elaine Rich and Kevin Knight's textbook, *Artificial Intelligence*, they loosely define Artificial Intelligence (AI) as "the study of how to make computers do things which, at the moment, people do better" [18]. While somewhat simplistic, this definition does provide insight into the goal behind AI research: give a machine (such as a computer) a task that requires intelligence and devise a method that allows the machine to successfully complete the task. Artificial intelligence tasks run the gamut from robot control to speech recognition to financial analysis. Some AI task domains are purely theoretical, while others find application in day-to-day life. However, the common thread that links these seemingly diverse applications is the "intelligent machine."

One of the key characteristics of the intelligent machine is that in its ideal state, it functions *autonomously*—without external control [8]. However, autonomous behavior in no way implies a lack of interaction with the environment or other intelligent machines. In fact, one of the important traits of an autonomous machine is its ability to interpret its environment and to share its knowledge with others. We often describe such a machine as an autonomous *agent* [18]. The agent can function as an individual, or it can interact collectively with other agents in a group.

We can carry the autonomous agent concept one step further and introduce *machine learning*. Unless the autonomous agent is to perform the same predefined task repeatedly, we clearly want to instill in the agent the ability to learn from mistakes and from environmental cues. The goal is that an agent might learn to perform a task better or maybe even perform a completely new task. Only with the ability to learn can we characterize a machine as "intelligent."

Michalski outlines three major research directions in the field of machine learning [16]. The first direction involves neural modeling, where neuron-like networks provide a mapping from input to output. The neural net has its origin in biology, where

physiologists are attempting to model the thought processes of the human brain [15]. On the engineering side, the neural network finds application in massive computation, where multiple outputs can be explored in parallel for a particular input. Although there are several implementations of the neural net, they share a commonality in their network topology of interconnected nodes. In general, neural nets must be initially trained, and research is focusing on methods for adaptation and learning [14].

The second major direction in machine learning research involves "symbolic concept acquisition." The purpose of symbolic concept acquisition is to define or classify objects. The process is initiated with a known instance or part of the concept to be learned. The process iterates by examining other objects to determine if they are part of the concept. If they are, the concept definition is expanded to include them. The goal of the process is to come up with a detailed definition of the object. For example, if the concept definition to be learned is "house," the process might start with "brick" and add definitions for support of the bricks, creation of a roof, *etc.* This approach is fairly effective in learning, but it relies heavily on a "teacher" to label and compare objects [18].

The third major direction involves a domain-specific type of learning in which the machine has a large amount of prior knowledge about the task or the environment in which the task is to be performed. By having a large knowledge base, the machine can employ more intricate reasoning and planning methods. While one could argue that this technique most closely models the way humans approach a particular action (such as opening a can of soda, an action that the human usually already knows how to do), such a technique can be extremely complicated to implement and can require large amounts of machine resources.

In an attempt to offer a compromise between these three learning paradigms, John Holland has proposed the *learning classifier system (LCS)* [4,10,11]. The LCS is a rule-based, message-passing, machine learning paradigm designed to process environmental stimuli, much like the input-to-output mapping provided by a neural network. In addition to neural-like mapping, the LCS provides learning through genetic and evolutionary

adaptation to changing task environments. Therefore, the learning mechanisms are built directly into the paradigm. The LCS excels above symbolic concept acquisition because the learning mechanism employed does not require a teacher-like environment that labels concepts. Adaptation and learning are based on simple, incremental feedback with little knowledge of the overall task at hand. In fact, LCS "concepts" are embedded within the system's rule base and are usually not explicitly defined by the designer. Finally, because the LCS requires few machine resources, it excels above a domain-specific machine learning approach, while still providing a mechanism for gaining limited environment knowledge. In all cases, the LCS provides a fast, real-time approach to machine learning.

One of the shortcomings of the LCS is that it does not provide a mechanism for use in multiple-agent scenarios. For example, we may want to employ several LCS agents to solve a large task or to work in parallel on the same task to expedite a solution. For such situations, we would like the agents to be able to interact with or "talk" to one another. In order to meet these requirements, we present a novel extension to the LCS we call the distributed learning classifier system (DLCS). The DLCS is a framework for distributing multiple learning classifier systems over a network-like interface. The DLCS provides various types of inter-agent communication that are completely compatible with the communication and learning mechanisms built into the standard LCS. We also provide a DLCS software code base that can be used to apply either distributed or standard classifier systems to virtually any type of task. The code is written in C++ to take advantage of the modularity inherent in the classifier system.

This thesis begins with an in-depth discussion of the learning classifier system. We present the LCS is such detail for two reasons. First, no single paper describes all of the facets of the LCS, and we wish to provide a reference for future research in the area. Second, the DLCS architecture is built around many of the same concepts that drive the standard learning classifier system, and we wish to emphasize these concepts. After describing the operation of the LCS, we present a few simulation results emphasizing some of the important features. We then describe the architecture for the DLCS and

provide a few simulations illustrating its operation. The thesis concludes with suggestions for future research and an explanation of how to use the software code.

2. The Learning Classifier System

2.1 Overview

As previously stated, the learning classifier system, as outlined by Holland, *etal.* in [4,10,11], is a rule-based, message-passing, machine-learning paradigm. The operation of the LCS is centered around a list of rules or *classifiers*. These rules are essentially a set of "if-then" statements, where the "if" part of a rule is called a *condition*, and the "then" part is called an *action*. As with if-then statements, a rule can have multiple conditions, which must be conjunctive before the action is taken. We call these conditions and actions "*words*," with each word being composed of a set of bits or *alleles*. The term "allele" derives from genetics, where an allele is part of a gene. A gene encodes a trait in an organism, and an allele is a specific part of that trait [13]. In the same manner, we can view a classifier as a gene of the LCS, and an allele as a particular part of that classifier. To keep the implementation of the classifier system simple, each allele is taken from the set {0, 1, #}. Using these three symbols, words take on the appearance of binary strings. The "#" symbol is used in condition words as a binary "don't-care." If # is used in an action word, it becomes a "pass-through" operator. The # symbol will be explained in more detail when we further discuss the classifier list in Section 2.3.

In addition to containing one or more conditions and an action, a classifier also has a *strength* associated with it. The strength of a classifier is a measure of that classifier's *fitness* in the LCS. By fitness, we mean the usefulness of the classifier to the system as a whole. Sometimes rules in an LCS may not be all that beneficial in solving the task at hand or may even be completely wrong. We introduce *credit assignment* below as a method of adjusting the strengths of rules as they become more or less useful.

The LCS operates by circulating *messages* in a message loop, shown by the solid arrows in Figure 2.1. Messages are similar to words in that they have the same number of alleles, but messages do not include the # symbol. They are simply strings of 1's and 0's. The LCS iterates by executing a series of sequential steps which we will hereafter refer to as the *execution cycle*. One iteration of the execution cycle is called a *clock tick* and is shown in Figure 2.2. Therefore, when we speak of "time" in the LCS, we are speaking of clock ticks, *e.g.* $t=2$ refers to the second clock tick in an LCS simulation.

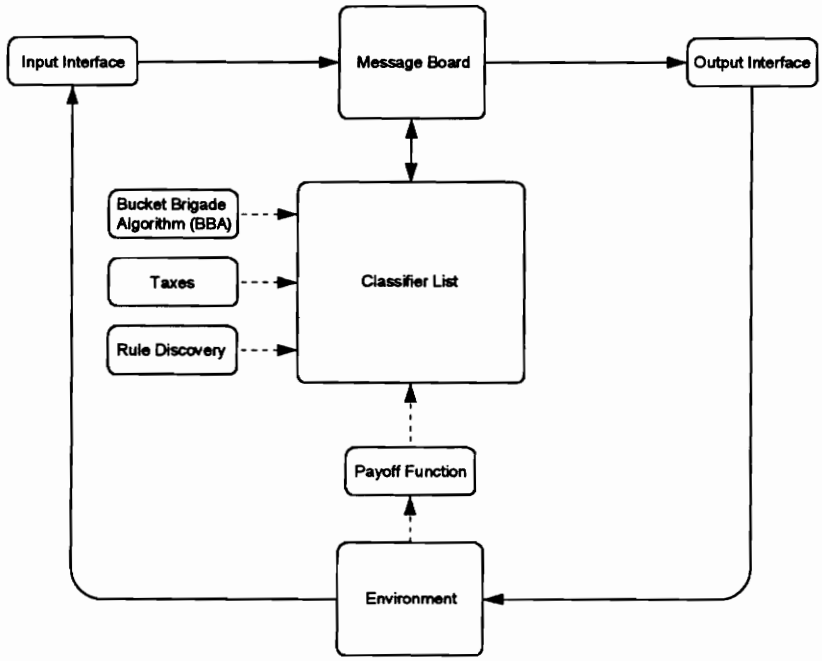


Figure 2.1. The learning classifier system.

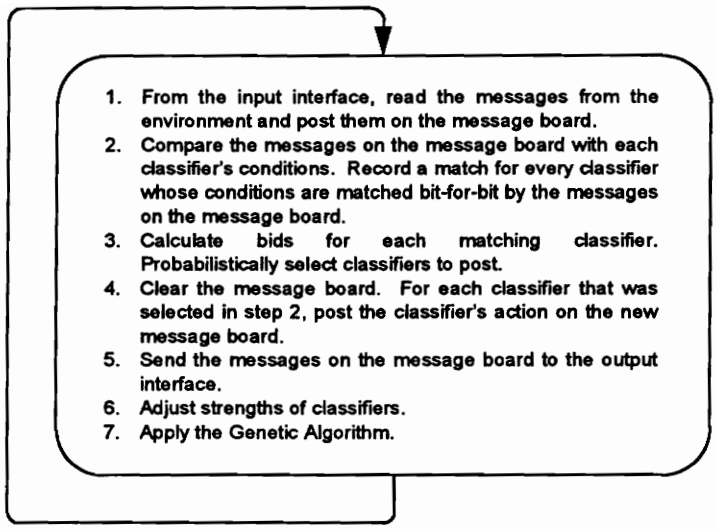


Figure 2.2. Execution cycle of a classifier system.

Step 1 of the execution cycle is to read messages from the *input interface* and post them to the *message board.* The input interface is a translator which takes sensory information from the *environment* and converts it to classifier system messages. The environment is simply the task to which the LCS is being applied. These messages can be viewed as a representation of the state of the environment. The message board is simply a bulletin board on which all messages to and from the environment are placed.

Clock tick step 2 performs a binary comparison between the environment messages on the message board and the conditions in the classifier list. The LCS creates a set of *eligible* classifier actions for those classifiers that have all of their conditions *matched* by some or all of the messages on the message board. A match is a one-to-one correspondence of condition word alleles to message alleles, except for those condition alleles that are don't-cares. Therefore, we can describe step 2 as performing the following check: if the state of the environment is approximately x, then action y is eligible to be taken. Often there are more eligible actions than can be realistically chosen, so the LCS must select a subset of eligible actions to take. The size of the message board is used as the upper limit for the number of actions in this subset. Therefore, in step 3, the LCS

computes a *bid* for each eligible classifier action based on the classifier's strength and then probabilistically selects actions from this eligible set. Those actions with larger bids have a higher probability of being selected. Note that the size of the message board is chosen based on the task being explored.

In step 4, the environment messages are cleared from the message board, and the probabilistically-selected actions are posted to the empty message board. Then in step 5, these action messages are sent to the *output interface*, which translates the messages into some sort of environment action. Therefore, the output interface performs the exact opposite role of the input interface, although the precise meaning of input and output messages will be different. The input interface reads sensory information or *detectors* from the environment, whereas the output interface sends action information or *effectors* to the environment.

We should point out that the input and output interfaces are solely responsible for the format and interpretation of messages in the LCS. Sometimes these interfaces are designed in such a way that messages or parts of messages are designated for use in the environment or for internal communication use. Specifically, a message might contain a tag which labels it as an *environment message* or as an *internal message*. Environment messages are translated into effector information, whereas internal messages are left on the message board to be used on the next clock tick. Alternatively, the interfaces may divide a message into two parts or *substrings*, one for effector information and one for internal communication. Then, on step 5, the output interface just strips off the *environment substring* and leaves all of the messages on the message board. On the next clock tick, the input interface puts detector information where the effector information was in the environment substring of each message on the message board. These messages still contain the *internal substrings* from the previous clock tick, so in effect, the LCS is given information about the previous time step via the internal substrings and information about the current time step via the detector information. Regardless of the message type used, it is important to keep in mind that the input and output interfaces are the *only* parts of the

LCS that understand the format of messages. All other parts of the LCS treat messages as strings of alleles to be processed.

The message formats described in the previous paragraph will make more sense when we present an example in Chapter 3, but it is necessary to mention them here because the byproduct of internal information passing is a facet of the LCS known as rule *chaining*. We say that a chain has formed between classifiers if the action message of a classifier that posted on the previous time step matches one or more of the conditions of a selected classifier on the current time step. For such a chain to form, an action word tagged as an *internal message* or an action word containing an *internal substring* must be posted on the previous time step.

Step 6 provides one of the two machine learning components of the LCS execution cycle. This step is responsible for *credit assignment*, or classifier strength adjustment. Credit assignment has three components, the *environment payoff function*, the *bucket brigade algorithm*, and the *rule taxes*. The purpose of the environment payoff function is to allow the environment to evaluate the effectiveness of the chosen actions. Since one of the requirements of the LCS is that it have a very limited amount of domain knowledge, the payoff function determines the action merit based on a very limited understanding of the overall task. The function then rewards actions that are useful and penalizes actions that are not. The environment payoff function is described in Section 2.4.1.

The bucket brigade algorithm (BBA), on the other hand, has *no* understanding of the overall task. The purpose of the BBA is solely to encourage chains of rules to form. An LCS without the bucket brigade algorithm is a purely reactive control structure, responding immediately to environment stimuli. With the BBA, however, the LCS can "plan" actions by forming rule chains. This concept of planning will be discussed with the formal definition of the BBA in Section 2.4.2.

Also included in credit assignment are rule *taxes*, which apply various types of tax-like strength adjustment to the rules in the classifier list. These taxes aid in the dynamics

of population control, instead of directly encouraging rule longevity or chain formation. Section 2.4.3 outlines the purpose of rule taxes.

The second machine learning component of the LCS is *rule discovery*. Step 7 is responsible for the primary rule discovery component, the *genetic algorithm (GA)*. The genetic algorithm is composed of a series of genetic-like operators which evolve the gene-pool (classifier list). This evolution allows the LCS to explore new rule variations for environment situations. In addition to the genetic algorithm, the LCS employs *rule discovery operators* which are responsible for LCS adaptation. These operators create new rules when environment situations arise for which there are no rules present in the classifier list. The LCS therefore adapts to a new situation. The GA and rule discovery operators are described in Section 2.5.

Figure 2.1 shows that all the learning components described above interact directly with the classifier list. Dashed lines are used in the figure to indicate such interaction, as opposed to the message flow shown by the solid lines.

We will spend the remainder of this chapter describing in greater detail the components of the learning classifier system. We present these components in their order of appearance in the execution cycle. We then provide a short synopsis of past and present research involving the learning classifier system. Finally, we briefly describe the C++ software written to implement the LCS.

2.2 The Message Board

As illustrated in Figure 2.3, the message board is essentially a bulletin board holding a list of q message objects, where q is chosen based on the task to which the LCS is applied.

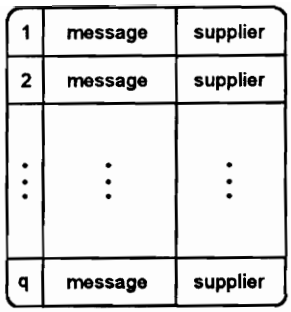


Figure 2.3. A typical message board.

Since these objects vary with time (clock tics), we describe the message board M as:

$$M(t) = \{M_1(t), \ldots, M_q(t)\} \tag{2.1}$$

Each message object has two parts, a message m of length l alleles and a supplier number L:

$$M_j(t) = \{m_j(t), L_j(t)\} \tag{2.2}$$

The supplier number L identifies the *number* of the classifier that posted the corresponding message on the message board. (Classifiers in the classifier list are assigned numbers from 1 to n). L is used by the bucket brigade algorithm in determining what chains have formed (see Section 2.4.2). Messages are posted to the message board on steps 1 and 4 of the execution cycle. Note that in step 1, messages corresponding to effector information from the environment interface are given a supplier number of zero because they are not associated with any particular classifier. Internal messages from the previous time step and all messages posted on step 4 will have a non-zero supplier number.

2.3 The Classifier List

The classifier list, shown in Figure 2.4, is composed of a set of rules containing one or more condition words, one action word, and a strength. We describe the classifier list C as a time-varying set of classifiers:

$$C(t) = \{C_1(t), \ldots, C_n(t)\} \tag{2.3}$$

where n is the number of classifiers in the list. An individual classifier i may be described as a time-varying set containing k condition words c, one action word a, and a scalar strength $S \in \Re$:

$$C_i(t) = \{c_{i1}(t), \ldots, c_{ik}(t), a_i(t), S_i(t)\} \tag{2.4}$$

All words in the classifier are of length l alleles. However, the # allele takes on a different meaning for condition and action words. In a condition word, the # allele represents the standard don't-care state used in binary representation. For example, the words 01101011 and 0##01011 are considered equivalent because of the don't-care allele in positions 2 and 3 of the second word.

In an action word, the # allele represents a "pass-through" operator. Pass-through simply means that should an action message be chosen to post, all pass-through alleles are replaced with the corresponding allele from the message that matches the condition part of the classifier. However, Holland's outline of the pass-through allele is somewhat ambiguous for classifiers that have more than one condition and thus require a matching message for each condition. In this case, one of the matching messages must be chosen before the action's pass-through alleles may be replaced. Pass-through will not be used for any of the LCS applications presented in this thesis.

1	condition 1	•••	condition k	action	strength
2	condition 1	•••	condition k	action	strength
3	condition 1	•••	condition k	action	strength
⋮	⋮	⋮	⋮	⋮	⋮
n	condition 1	•••	condition k	action	strength

Figure 2.4. A typical classifier list.

The strength $S(t)$ is a figure of merit for the corresponding classifier. As previously stated, strength is adjusted based on the fitness of a rule to a particular environment at a particular time. Since the state of the environment can change with time, the strengths of rules are also time-varying. Also, strengths are not allowed to go above a saturation value S_{max}. Strength adjustment is discussed when we outline credit assignment in Section 2.4.

The classifier list is used in clock tick steps 2, 3, and 4. In step 2, environment messages on the message board are compared to the conditions in the classifier list to determine which classifiers are eligible to post their actions on step 4. From this comparison, a set of eligible classifiers is formed from those classifiers having *all* of their conditions matched by some or all of the messages on the message board. We describe this time-varying set of eligible classifiers as

$$E(t) = \left\{ i : c_{ij}(t) \approx m_p(t) \text{ for all } j = 1,\ldots,k \text{ and } p \in \{1,\ldots,q\} \right\} \qquad (2.5)$$

We use $\approx$ to denote a match between a message and a condition word since the condition words may include #'s (don't-cares). Note that E contains the *numbers* of the eligible classifiers, *i.e.* the first column in Figure 2.4.

Because more classifiers may be eligible to post than there are message board slots ($E^{\#} > q$), the LCS must select at most q classifiers to post. To make a probabilistic selection from the set E, the LCS computes a *bid* for each eligible classifier. Then, the higher the bid, the higher the probability of being selected. Bids are calculated using the following formula:

$$B_i(t) = bP_i(t)S_i(t-1)(1+U_i(t)) \qquad (2.6)$$

where b is a constant much less than one, $P(t)$ is the *specificity* of the rule, $S(t\text{-}1)$ is the *strength* the rule had on the previous clock tick, and $U(t)$ is the *support*. We use $S(t\text{-}1)$ because the strength of the rule will be changed during step 6 of the execution cycle. The probability that a classifier will be selected to post is given by:

$$p_i(t) = \frac{B_i(t)}{\sum_{j \in E(t)} B_j(t)} \qquad (2.7)$$

Note that bids are not allowed to be larger than a saturation value B_{max}.

Specificity, as its name implies, describes in percentage form how specific a rule is. Rules with few #'s in their condition words are more specific than rules with many #'s. Specificity is given by the following formula:

$$P_i(t) = \frac{\text{Total number of non-\#'s (don't cares) in the conditions}}{l \cdot k} \qquad (2.8)$$

Note that the numerator of equation 2.8 is the *total* number of non-don't-cares. In other words, non-# are counted from *all* condition words. Recall that l is the length of a word and k is the number of condition words in a classifier. The purpose of specificity is to give

more specific rules a better chance of being selected. Highly-specific rules are better "tuned" to a particular facet of the task, whereas less specific rules are often just defaults.

The support quantity in equation 2.6 is based on the bids of those classifiers which, on the previous time step, posted the currently-matched messages. Support is part of the bucket brigade algorithm and is designed to encourage rule chaining by multiplicatively increasing a classifier's bid. We will describe support as part of the BBA in Section 2.4.2.

We designate the q selected classifiers from step 3 as a set $\hat{C}(t)$, where $\hat{C}(t) \subset E(t)$. The classifiers in $\hat{C}(t)$ are posted to a clean message board in step 4. For each message board slot filled, *L(t)* is assigned to the number of the classifier that posts the message.

2.4 Credit Assignment

Credit assignment is one of the factors that makes the classifier system a *learning* system. Essentially credit assignment is responsible for adjusting the strengths of the rules in the classifier list as these rules become more or less useful in the present environment situation. As previously stated, there are three forms of credit assignment used: the environment payoff function, the bucket brigade algorithm (BBA), and the rule taxes. The environment payoff function is strength adjustment based on feedback from the environment. The BBA is an internal payoff scheme that rewards *suppliers*, those classifiers that cause other classifiers to get selected via internal message passing. Rule taxes aid in the dynamics of population control by applying various tax-like strength adjustments to classifier list rules. All three credit assignment schemes are discussed below.

2.4.1 The Environment Payoff Function

The environment payoff function uses feedback from the environment to evaluate the fitness of a classifier. The amount of payoff $R_i(t)$ depends entirely on the task to which the LCS is being applied and takes on the general form:

$$R_i(t) = f\left(S_i(t-1), \text{environment}\right) \tag{2.9}$$

As an example of the usefulness of the environment payoff function, we use the animat problem. In the animat problem, an agent must maneuver around obstacles and reach a goal. The agent is assumed to have the capability to sense obstacles and goals within a certain range of itself (a very realistic assumption). Therefore, a *good* animat rule might be one which states "if a goal is to the left, move left." If this rule is selected to post, the environment payoff function would sense that the agent has moved closer to the goal and would reward this rule by increasing its strength some constant amount. By increasing the rule strength, the rule's bid will increase. The rule will then have a better chance of being selected in the future, should the "goal left" environment state arise again.

An important point to make here is that the environment payoff function can only decide the value of a rule at the current instant of time. The payoff function does not, in general, have an overall knowledge of the task at hand, *i.e.* the function is not omniscient. For example, from the environment payoff function's point of view, a *bad* animat rule might be "if a goal is to the left, move right." Such a rule would move the agent farther from the goal. Therefore, if this rule is selected, the payoff function would likely penalize the rule by subtracting a fixed amount from its strength. However, if the animat's environment also contains an obstacle to the left that the agent did not sense, moving right might be considered a good action from the standpoint of an outside observer. The observer may note that such an action is the only way to get around the obstacle. Perhaps this action would be the start of a chain of actions that maneuver the agent around an

obstacle. However, the environment payoff function, with its very limited knowledge of the world, would not be able to reason or directly reward such a chain.

2.4.2 The Bucket Brigade Algorithm

The bucket brigade algorithm is designed to compensate for the environment's lack of ability to reason or reward chains of actions. As previously mentioned, the input and output interfaces may be set up to distinguish between environment and internal messages or substrings. In such a case, a chain forms when internal information from the previous time step contributes to the selection of a classifier on the current time step. We say that the rule containing the internal message or substring from the previous time step is a *supplier* of the rule from the current time step. Chaining is considered a good phenomenon because it can provide a kind of memory in the classifier system, thereby enabling an LCS to perform tasks that require multiple stages. For example, in the animat scenario described in the previous section, the "goal left, move right" rule might work as part of a chain that allows the agent to maneuver around the obstacle which is also to the left. In this way, the BBA helps support a chain that the environment payoff function has no way of recognizing.

In order to describe the two facets of the BBA, we define the *support set* $\tilde{C}_i$ as the set of all classifiers whose posted actions have supplied an eligible classifier C_i:

$$\tilde{C}_i(t) = \left\{ L_j(t) : i \in E \text{ and } c_{ip} \approx m_j \text{ for some } p \in \{1,\ldots,k\} \right\} \qquad (2.10)$$

These suppliers are obtained from the supplier field L_i on the message board. A classifier with three conditions will have three suppliers, although these suppliers may not be unique.

The first facet of the BBA involves the *support* value $U(t)$ used in the bid calculation (see Equation 2.6). The support quantity is designed to encourage chaining. Support increases the bid for an eligible classifier whose predecessors (suppliers) also had large bids, thereby increasing its likelihood of being selected to post. The stronger the suppliers, the higher the bid of the eligible classifier. Therefore, strong chains have the potential to get stronger. Support is calculated as follows:

$$U_i(t) = \sum_{j \in \tilde{C}_i(t)} B_j(t-1) \qquad (2.11)$$

In other words, support is just the sum of the suppliers' bids from the previous time step. Note that support multiplicatively increases the bid, so even a small amount of support can have a tremendous effect on the bid. We follow the convention in [23] for support in the bid equation, where $1 + U_i(t)$ is the support multiplication factor. Holland omits the one in his bid equation, but such an omission can cause the overall bid to be reduced if $U_i(t)$ is less than one. For LCS applications where the BBA is not used, $U_i(t)$ is set to zero.

The second facet of the bucket brigade algorithm is responsible for paying the suppliers of the classifier actions that posted on clock tick step 4. The payoff algorithm is as follows:

given $C_i(t)$

if $i \in \hat{C}(t)$

$$S_i(t) = S_i(t-1) - B_i(t) \qquad (2.12)$$

for all $j \in \tilde{C}_i(t)$

$$S_j(t) = S_j(t) + \frac{B_i(t)}{k}$$

The algorithm is repeated for all $i \in \hat{C}(t)$ (for all classifiers that posted). In the first part of this payoff algorithm, the winning classifier pays the amount that it bid for its right to post. In the second part, the paid bid is divided among all supporting classifiers (the suppliers). For example, say a classifier with four conditions, $C_3(t)$, is selected to post. First, the strength of classifier 3 would be reduced by the bid calculated in clock tick step 3. Then, if its suppliers are $\tilde{C}_i(t) = \{ 10, 1, 10, 4 \}$, classifiers 1 and 4 would get $\frac{B_3(t)}{4}$ added to their strengths and classifier 10 would get $\frac{B_3(t)}{4} + \frac{B_3(t)}{4} = \frac{B_3(t)}{2}$ added to its strength, since it occurs twice in the support set. Note that $\tilde{C}_i(t)$ can also contain zero if an environment effector message "supported" the selection of $C_i(t)$. In such a case, the payoff algorithm simply discards the portion of the bid that would have been paid to supplier zero, since the environment effector message does not have a "strength" associated with it.

The bucket brigade algorithm has two requirements that must be satisfied in order for it to reinforce a chain. First, the chain must be executed repeatedly, because each rule in the chain only pays its most recent suppliers. Those rules must then pay their suppliers on the next execution of the chain, and so on. With enough repetitions, the payoff will "trickle down" to the early stage-setting classifiers. Second, in order for a chain to grow in strength, the last rule in the chain must experience a net strength increase when it is executed. This second requirement must occur for two reasons. First, the last rule in the chain will always have to pay its bid to its suppliers, thereby decreasing its strength. If the rule has no compensation for this payment, its strength will get repeatedly reduced until it goes to zero. Second, a net strength increase in the last rule in the chain translates to a higher bid $B_i(t)$ for this rule, which translates to a higher payoff for its suppliers. We detail how such a strength increase can occur in the next paragraph.

As discussed in Section 2.4.1, the environment payoff function is used to evaluate environment actions. This function rewards and penalizes the classifier that posted the

environment message based on the message's useful to solving the task. Depending on the message format used for internal rules, some or all of the rules in a chain will also have their strengths adjusted by the environment payoff function. If internal messages are used—action words tagged specifically for internal communication—then the output interface ignores these messages on clock tick step 5, and hence, the environment payoff function does not adjust the strength of the corresponding classifiers. However, in order for an internal message chain to prove useful to solving the task at hand, the chain will usually end with a classifier whose action is tagged for environment use. This action will subsequently be evaluated by the environment payoff function. If the rule experiences a net strength increase, the trickle down process will begin for the chain.

If action messages are divided into internal and environment substrings, then each action that gets posted will be evaluated by the environment payoff function, as well as being subject to the bucket brigade algorithm. In this case, the strength $S_i(t)$ of a posting classifier becomes a function of the bid it pays $B_i(t)$, as well as the payoff from the environment $R_i(t)$:

$$S_i(t) = S_i(t-1) + R_i(t) - B_i(t) \qquad (2.13)$$

This equation also holds for the last message in an internal message chain, since that message is subject to the environment payoff function as well. In either case, however, we must be careful in the definition of the payoff function so that the last rule in a chain can in fact experience a net strength gain if it produces a good action. The substring case adds yet another dimension of complexity, since *all* rules in the chain get evaluated. Often stage-setting classifiers are not directly beneficial to the environment, yet they are needed to produce useful actions at the end of the chain. In Section 3.4.3, we present an example of this situation. Ultimately, we would like to ensure that stage-setting classifiers do not get so heavily punished by the environment that the chain they are forming gets killed. In

light of these two requirements, we present two conditions that must hold for a chain of this nature to be successful:

$$\max\left(R_{i,reward}\right) > B_{\max} \tag{2.14}$$

$$B_{\max} > \left|\max\left(R_{i,penalty}\right)\right| \tag{2.15}$$

where $\max(R_{i,reward})$ is the maximum possible reward given by the environment payoff function, $\max(R_{i,penalty})$ is maximum penalty, and B_{max} is the bid saturation value, as previous discussed. We require B_{max} to be smaller than the maximum possible reward so that when the last rule in the chain pays its bid, it still has a chance of experiencing a strength increase, even if it currently has a strength of zero. Of course the equation implies that this rule must get the largest possible environment reward and the bids must be in saturation, but we are free to set B_{max} and the payoff function as necessary to loosen these constraints. The only other requirement is that B_{max} be greater than the worst possible penalty so that state-setting classifiers that get punished by the environment can still receive a strength increase when they get paid for supplying the next rule in the chain. In other words, the bid payoff compensates for the environment punishment. Note that the absolute value is used in the equation because penalties are negative.

While the aforementioned requirements are necessary, they are not sufficient. If the bids of the stage-setting classifiers do not reach saturation, then the bids may still not be able to compensate for environment punishment. Also, if the saturation value is set too low, then the rule-competition aspect of action selection is lost, because all rules have the same bid. Finally, chains are affected by length. The longer the chain, the more likely the bids will reach saturation for later elements in the chain, but the slower the strengths filter back to the stage setters. Unfortunately, the dynamics of the bucket brigade algorithm are not well understood. Mathematical models are difficult to derive because environment payoffs can differ for each element in the chain, and chains can be connected in rather

complex configurations when multiple conditions are used. Therefore, we have provided the previous discussion as the groundwork for future examination of the BBA.

We should point out a couple of characteristics of the BBA before concluding this section. As previously stated, the BBA introduces a certain amount of planning ability in the LCS. Without chaining, the LCS immediately responds to an environment state, thereby making the LCS more of a reactive controller. With the BBA, the LCS has the ability to delay effector actions, sequence a series of actions, or store information inside the classifier list. However, this stored "knowledge" takes the form of condition and action words, and an observer will not likely be able to translate it.

Finally, just because a chain forms does not mean that the chain is of any value. The BBA always encourages initial chain formation through the support quantity in the bid equation. Ultimately, one assumes that the end result of a chain is an action which the environment payoff function will either reward or punish. This reward or punishment will then propagate back to the stage setting rules in the chain. Again, the degree to which this assumption holds must be investigated further before any concrete mathematical conclusions can be drawn.

2.4.3 Taxes

Taxes are included under "credit assignment" because they also adjust the strengths of LCS rules. However, each of the taxes described below helps control classifier list dynamics, rather than directly encouraging rule longevity or chain formation. Taxes are applied to strengths after the payoff function and BBA payoff in clock tick step 6. There are three types of taxes: *head tax*, *bid tax*, and *producer tax*. The head tax helps reduce the strength of unused or weak classifiers so that they will have a better chance of being selected for replacement by the rule discovery component of the LCS. The head tax is a low, fixed-rate tax applied on every time step to each classifier.

The bid tax is also a low, fixed-rate tax, except that this tax is applied only to those classifiers that were eligible to bid on clock tick step 3. Rules that are extremely general—rules with low specificities—tend to get selected to post quite frequently because of the large number of # (don't-cares) in their condition words. The bid tax helps reduce these over-general rules by reducing the strengths of rules that bid too often.

The producer tax, a progressive tax, increases with the number of times a rule gets to post. While an exact method for applying this tax is not explicitly defined in the literature, we suggest the following equation:

$$producer\ tax = \frac{\text{number of postings in } n}{n} \qquad (2.16)$$

where n is the internal over which the tax is applied. For example, if n = 10, and the rule is posting for the fifth time in the interval, its tax is 50%. Like the bid tax, the producer tax helps reduce over-generalization. The producer task also helps reduce the formation of chains for the sake of chaining. In other words, it counteracts the BBA's indiscriminate encouragement of chain formation.

2.5 Rule Discovery

Rule discovery is the second factor that gives the classifier system the ability to learn. Many classifier system applications must operate in real-time with rapidly changing environmental conditions. In order for an LCS to adapt, rules that have little or no use in the current environment state must be replaced with better rules. In addition, sometimes a rule will not exist for the current environment state, and one must be created. Rule discovery helps meet these requirements. Figure 2.1 shows that rule discovery interacts directly with the classifier list.

Rule discovery is accomplished through the *genetic algorithm (GA)* and the *rule discovery operators*. The genetic algorithm is composed of three genetic operators which modify and reproduce the "gene pool" of classifiers, hence their name. On the other hand, the rule discovery operators perform creation based on need rather than creation based on random genetics. This distinction will become more apparent when the GA and the rule discovery operators are discussed in detail. The first two genetic operators, *reproduction* and *mutation*, are outlined by Holland in [11] and Wilson in [24]. The third, *elitism*, is discussed by Porter and Passino in [17]. Rule creation is accomplished by three different operators, the *Cover Detector Operator (CDO)*, the *Cover Effector Operator (CEO)*, and the *Triggered Chaining Operator (TCO)*, all of which are discussed by Robertson and Riolo in [19].

2.5.1 The Genetic Algorithm

The genetic algorithm (GA) draws its analogy from biology, where genes contain the traits for an organism. In a classifier system, the genes are the rules in the classifier list. By manipulating and mutating the classifiers' alleles, new rules are created. While the biological analogies to genetic algorithms will become more evident as the individual genetic operators are presented, one should use these analogies only as a tool for understanding. The GA's evolutionary process in a classifier system occurs at a much faster rate than in organisms. Also, in order to control the frequency of genetic changes to the classifier list, the GA is usually only performed at regular time intervals. This time interval, T_{GA}, is chosen based on how quickly one wants the "gene pool" to change. The GA, when performed, occurs on step 7 of the execution cycle.

For the sake of completeness, we mention here that the use of the GA and the bucket brigade algorithm to create and encourage chain formation has been named the "Michigan" approach by DeJong [6]. DeJong also sites another method used with rule-

based machine learning systems he calls the "Pittsburgh" approach[1]. In this approach, the genetic algorithm operates on entire genes (entire rules) instead of single alleles (parts of a rule). DeJong contends that the Michigan approach is more realistic for real-time system iteration, where extreme changes in operation are not desirable. He states that the Pittsburgh approach may be more appropriate for "off-line" iteration, where exploration of possible environmental outcomes is more realistic. However, we suggest that further research be done before such a conclusion can be made specifically for classifier systems.

2.5.1.1 Reproduction

Reproduction is the first of the three genetic operators to be discussed. Holland limits reproduction to the sexual type, where two rules reproduce to create two offspring via an operation called *crossover* [4]. In crossover, two parent classifiers of high strength are probabilistically chosen. Then, a "crossover point" is randomly selected somewhere within the rules. The crossover point is the location where the "strands" of alleles will be interchanged between the parents. The parent rules are copied, and then the strand exchange is performed, creating two new children. In Holland's implementation, these two children replace two probabilistically-chosen weak rules. In other implementations, such as Wilson's [24], one of the two children is chosen randomly, and only one weak rule is replaced. Figure 2.5 illustrates the crossover process.

The theory behind crossover is that strong parents will produce strong offspring. In order to give the offspring a "starting chance," the parents contribute some of their strength to the offspring. For example, the parents might each contribute one third of their strengths to the child if only one child is entered into the classifier list. If two children are entered into the list, each parent might contribute one sixth of their strength to each child. Note that for implementations that use environment and internal substrings,

[1] The names derive from Holland, the father of the classifier system, who is from the University of Michigan, and S.F. Smith from the University of Pittsburgh.

crossover might be more effective if the crossover point is chose at the substring boundary, so that this genetic operator has the effect of trying different combinations of environment information and internal communication information.

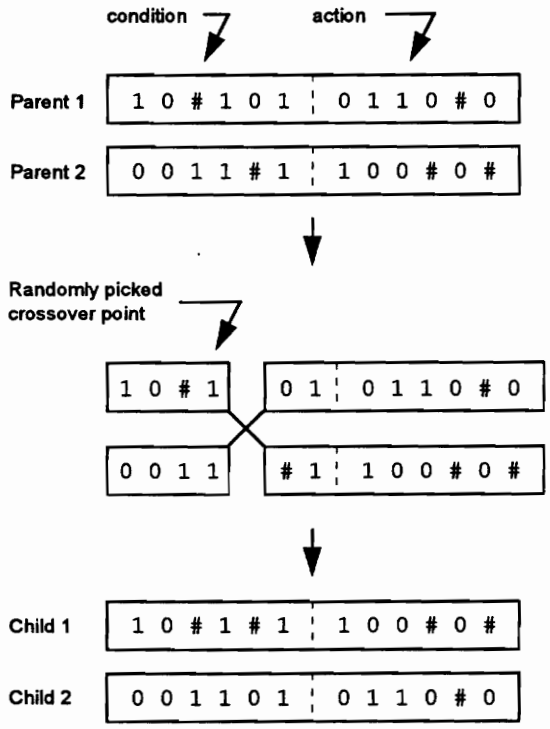


Figure 2.5. Steps in performing crossover.

Wilson extends the concept of reproduction to the asexual type, where a rule is simply duplicated. Asexual reproduction picks a high-strength parent, copies it, and replaces a weaker rule with it. The strength of the parent is then split with its single offspring [24].

A standard implementation of the reproduction operator is to perform reproduction once per genetic interval. A probability is specified for the type of reproduction. For example, one might set the probability for sexual reproduction to be 50%. Then, when the GA is performed, sexual and asexual reproduction will have the

same chance of occurrence. A higher probability of sexual reproduction will cause the LCS to explore new combinations of rules. A higher probability of asexual reproduction will cause the LCS to reinforce stronger rules by duplicating them in the classifier list.

2.5.1.2 Mutation

Mutation is the second genetic operator used in learning classifier systems. Mutation evolves the classifier "gene pool" by mutating existing rules. When a mutation is performed on an allele, the allele changes to a new state from the set {0, 1, #}. In other words, a *1* allele has a 50% chance of becoming a *0* and a 50% chance of becoming a # if it is selected to be mutated. A classifier is considered "mutated" if any of its alleles are mutated. However, a mutated classifier keeps its strength, since the mutated rule is just an evolved version of the original rule. Should this rule turn out not to be useful, its strength will be decreased by the environment payoff function and by taxes until it is eventually replaced. A standard implementation of the mutation operator is to specify a probability of mutation. Then, on every genetic interval, *each* allele of *each* classifier is subject to mutation.

2.5.1.3 Elitism

As previously stated, elitism is not a genetic operator in and of itself, but it is used in conjunction with the reproduction and mutation operators to provide *selective* evolution. Porter and Passino introduce elitism as a way of protecting high-strength classifiers or "elite rules" [17]. When good rules are discovered, they are protected from being genetically altered by mutation. Rules are deemed "elite" if their strengths are above a certain threshold. However, even though a rule is protected from mutation, it is not protected from having its strength adjusted by the environment payoff function.

Therefore, a rule may gain elite status only while it is useful to the current environment state. Once it is no longer useful and its strength decreases, it is once again subject to reproduction and mutation.

2.5.2 Rule Discovery Operators

The purpose of the rule discovery operators is to create rules on a need-basis, rather than wait for genetics to randomly create a rule that fits a particular situation. The rule discovery operators are analogous to an organism trying something new in a new environment situation. The conditions for trying a new rule vary with each of the three operators to be discussed.

2.5.2.1 Cover Detector Operator

The cover detector operator (CDO) creates a new classifier on clock tick step 2 for environment messages that have no match in the classifier list. The rule created by the CDO has the unmatched message(s) as its condition(s), and uses a randomly-generated action. In other words, the CDO tries something new for a detector situation which currently has no defined action. Note that the new rule created replaces a probabilistically-chosen weak rule, and the new rule is given the default starting strength used for new classifier list rules. This distinction is important because the CDO-created rule is not an offspring nor a mutation of any other rule in the classifier list. It is a new rule and therefore must not be biased with the strength of another classifier. Since the rule has been created for the current detector state, it is guaranteed to get selected on the current time step. If the rule proves useful, it will be rewarded by the environment payoff function. If not, the rule will eventually be replaced.

2.5.2.2 Cover Effector Operator

The cover effector operator (CEO) creates an environment-based rule when none have been selected from the classifier list. This need arises when all messages posted on step 4 are tagged for "internal use." In such a situation, the output interface will have no effector message to translate and thus no effector action to take. The CEO will take one of the rules that has been selected to post, copy it, and replace the action word with a randomly-generated action tagged for environment use. Thus, the CEO guarantees that the environment will always have an action to process. As with the CDO, the rule created by the CEO replaces a probabilistically-chosen weak rule, and the new rule is given the default starting strength. Note that CEO is performed only when necessary on step 5 of the execution cycle. Also, the particular task to which the LCS is being applied may not require an action on every clock tick. For such a case, the cover effector operator is disabled.

2.5.2.3 Triggered Chaining Operator

As with the bucket brigade algorithm, the triggered chaining operator (TCO) is designed to encourage rule chaining. The TCO creates a chain between two classifiers, $C_1(t\text{-}1)$ and $C_2(t)$, which were posted on successive clock tics purely by accident. The TCO is activated only when two conditions are met:

1. Classifier C_2's net strength increases.
2. Classifier C_1 posted on the time step before C_2, and C_1 is not already chained to C_2.

A net strength increase occurs when C_2's strength still increases from the previous time step even after it has been adjusted by the BBA payoff function and by the environment payoff function. If these two conditions are met, the TCO creates two new rules, C_1* and

C_2* which have been chained together as shown in Figure 2.6. The message *mmmmmm* is randomly created and has no predefined meaning, except that it is tagged as an internal message.

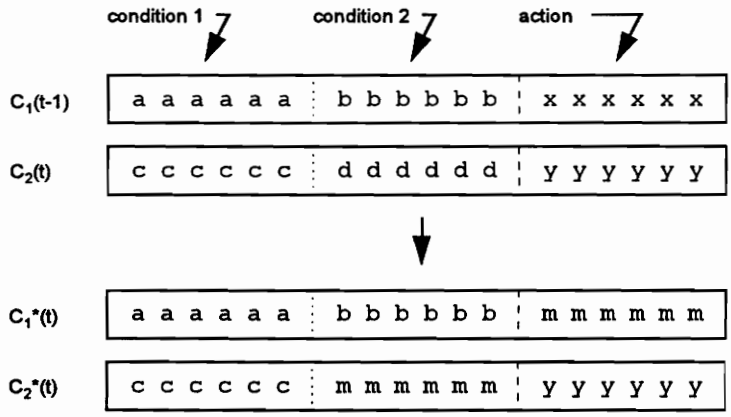


Figure 2.6: The triggered chaining operator.

As with the other rule discovery operators, the two new rules created by the TCO replace two probabilistically-chosen weak rules. Again, the strength assigned to these two new rules is the default starting strength. Note that the TCO is performed (if necessary) along with the genetic algorithm on step 7 of the execution cycle.

2.6 Programming vs. Learning

We present one final topic related to the LCS before discussing the current research in the area: the concept of classifier system *programming*. With all classifier systems, one must choose a task to be solved, decide on a message format, and design the input and output interface around this task and message format. However, at this point there are two directions the researcher might take. The first is to program the classifier system with a set of rules that perform a task in the environment. In Section 3.4, we

present such an example. A program, when entered correctly, will produce precise results each time, just as with any programming language. The second direction is to allow the classifier system to learn correct behavior under the influence of credit assignment and rule discovery. The hope is that with enough iterations and the correct combination of settings, rules that solve the environment task will emerge.

Belew and Forrest address this issue in [1]. They state that the flexible nature of the LCS lends itself to both programming and learning. In other words, one might "seed" the classifier list with a set of good rules and allow the LCS to iterate on those rules. In this way, the LCS is free to modify its original program. Such a technique can save computation time if the seed rules are useful, and can often produce surprising results. The LCS may need only a small subset of the seed rules entered into the classifier list by the designer, or the LCS may find some combination of the seed rules through genetics that prove more useful. With such flexibility, the LCS offers a wide range of possible application. Also, they contend that the mere definition of a message format is, in itself, the start of an LCS program, and we tend to agree.

One interesting side effect of this duality in classifier systems is something Belew and Forrest call "subsymbolic" knowledge representation. Since information is encoded in an LCS using strings of alleles or bits, it is possible for the LCS to evolve rules into strings of alleles that are not directly translatable. This potential outcome is especially true for internal message communication, where the message format is often not explicitly defined. For example, one might designate a message or substring as "internal," but not actually define what each of the alleles mean. Upon iteration, several sets of useful chains may emerge, but it is unlikely that the designer will be able to directly translate the internal message or substring. On the other hand, messages that go to the environment generally require a specific format that can be translated by the input and output interfaces. Ultimately, learning classifier systems will likely prove most useful when a compromise between learning and programming is used.

2.7 Current LCS Research

Before proceeding directly to the animat problem, we briefly describe some of the past and present work being done in learning classifier systems. Booker, Goldberg, and Holland introduce a simple classifier system in which an organism or robot moves through an environment and identifies "targets" and "dangers" [4]. They also introduce compound relationships and simple classifier system memory via classifiers with multiple conditions. Finally, they illustrate the formation of networks of rules. However, their discussion is more for illustrative purposes than for actual implementation, since they present no simulations or hardware implementation results.

Robertson and Riolo apply a learning classifier system to the task of letter sequence prediction [19]. In this scenario, the LCS is presented with a sequence of four letters, from which it must guess the next letter in the sequence. The LCS is repeatedly exposed to the sequence of letters, four letters at a time, until it learns the sequence correctly. The simulation culminates with the classifier system learning the entire alphabet. Their results indicate that population size affects the success of the LCS. They also explore the genetic algorithm and the rule discovery operators, but do not discover a precise relationship between the two.

Carse explores a modified classifier system he calls the Delayed Action Classifier System (DACS) [5]. The purpose of the DACS is to replace long chains with single classifiers that simply *wait* before posting their action. He contends that the maintenance of long chains is difficult because early stage-setting classifiers tend to get replaced by stronger classifiers at the end of the chain. The DACS maintains two message lists. The first is the traditional message board as previously outlined, and the second contains those actions *waiting* to post after their delay time has expired. The test scenario he uses simply looks for a predefined "correct" action message. Carse's results show that these delayed classifiers approach an asymptotic maximum strength much faster than traditional classifiers, thus implying better convergence. He also shows that in the presence of both

delayed-action classifiers and regular LCS chains, the delayed-action classifiers tend to interfere with the payoff of the regular chains.

Shu and Schaeffer explore another variant of the traditional LCS, the variable classifier system (VCS) [21]. The major difference between these two is that the VCS provides a method for storing quantities in the classifiers. For example, the condition and action words are broken into regions, each of which can contain either a constant from the traditional set of LCS symbols {0, 1, #}, or a variable whose meaning is explicitly defined by the environment. The variable fields in the condition word of the classifiers are used to store values from messages being compared to the classifier list. In other words, the variable fields provide parameter locations for the rule being compared. Shu and Schaeffer contend that this expansion to a traditional classifier list will allow the classifier system to solve more quantitative tasks.

Zhou and Grefenstette offer an expansion to the traditional classifier system by introducing a method in which an LCS can store learned abilities in long term memory [25]. They call their classifier system a "Classifier System with Memory" or CSM. Essentially, all "successful" rules are placed into long term memory (LTM). Once a rule is place into LTM, it is no longer subject to traditional LCS evaluation and replacement. This approach is somewhat similar to rule elitism as previously outlined, except that even with elitism is it possible for a rule to be replaced if it is no longer useful in the current environment setting. The CSM makes the assumption that if a rule is successful in one particular instance, it will be useful in the future if a similar instance arises again. Their test environment uses a simple robot that must navigate through a maze. The robot can sense its position, direction, and the presence and direction of obstacles in its path. Finally, the actions are evaluated by the environment payoff function and reinforced appropriately. They show that with the LTM, the robot completes "tasks" in the maze more quickly as it encounters situations it has previously seen.

Zhou and Grefenstette make some assumptions about the robot which may not be practical in an actual realization of the maze problem. First, they assume that the robot

can sense its absolute position in the maze. While such a condition is technologically feasible, it forces the robot to exist in a predetermined environment, thereby making it difficult to adapt to new environments. In addition, they assume that if the robot encounters a situation it has seen before, the solution will always be the same. For the animat problem, this assumption may not always be the case. Just because a chain was successful once does not mean it will be successful again. Zhou and Grefenstette assume that a rule or chain that has been successful once will remain useful throughout the life of the problem.

Tang also looks at the application of an LCS to the maze problem [23]. However, navigation though the maze is only rewarded if the robot reaches the goal. Also, he uses no genetic algorithm. Essentially his LCS performs an exhaustive search through the set of possible solutions in his simple maze, accelerated only by limited learning with the LCS. His simulations demonstrate the formation and use of chains more than the application of an LCS to the maze scenario.

Wilson takes a standard LCS and applies it to the learning of a k=2 multiplexer [24]. A k=2 multiplexer has two address bits which access 2^2 data bits. For example, an address of 00 would retrieve the bit in data position zero. The task of the LCS is to derive correct address and data output combinations. While his implementation of the LCS is close to the original Holland definition, he combines the rule discovery operators into one operation he calls the *creation* operator. This operator most closely resembles the CDO, and because of the relatively simplicity of his environment, the other rule discovery operators are not needed. His simulations show that the LCS is very effective in producing the correct data output for a given address. He also shows that crossover helps increase the convergence rate. Wilson concludes that his multiplexer problem has the same constraints as the animat problem, since the LCS must learn disjunctive concepts under knowledge-limited credit assignment by the environment payoff function. While these two constraints are also present in the animat problem, as will be shown in the next chapter, the multiplexer problem is definitely a simplification of the animat problem.

Finally, Dorigo and Schnepf use traditional learning classifier systems to explore robot control by presenting an environment in which a robot avoids hot objects and learns to follow light [7]. However, they base their robot model on a biological approach called the *Tinbergen model*, in which behavior is divided into a hierarchy of *instinct centers*. An instinct center is responsible for a particular part of a behavior in an animal. To simulate instinct centers, they use several classifier systems running in parallel, with each LCS learning a different part of a behavior. The classifier systems are then connected in a hierarchical fashion to build larger, more complex actions. Dorigo and Schnepf first simulate their environment by having the robot follow a moving light source. Then, the addition of a hot object is added. The robot successfully avoids the hot object while still tracking the light. In addition, the robot tends to take the shortest path to the light rather than tracking the light path if the light is moving quickly. Finally, they add "food" to the environment. The robot learns to feed, but at a much slower rate than the hot object avoidance and light tracking. Dorigo and Schnepf's work is very interesting in that they have a chosen rather difficult LCS task with an intricate biological model. Also, their works appears to be closest to the distributed learning classifier system concept in that they use multiple LCS's. However, in their model, these classifier systems do not explicitly communicate with one another. As will be shown, our DLCS architecture provides defined methods of communication for DLCS agents.

2.8 The *classifierSystem* C++ class

In order to facilitate application testing of the learning classifier system presented in this chapter, we have designed a C++ class called *classifierSystem* which allows the user to connect virtually any type of environment task to the standard LCS. A C++ *class* is a programming construct that allows a developer to define an object and encapsulate the functionality for that object. In other words, all functions, parameters, *etc.*, for an object

are included within the class. For the *classifierSystem* class, all necessary methods for executing an LCS clock tick are included, and the user simply designs the environment task to be tested and programs the input and output interface functions stubs contained in the class. Since a learning classifier system has many parameters which must be optimized for specific environments, the user can set all class parameters dynamically, thereby facilitating multiple executions of an LCS simulation with different parameters. The *classifierSystem* class also contains the necessary methods for distributing multiple DLCS agents. Objects of the class can be repeatedly instantiated, and all class parameters will remain the same for each instantiation. This ability allows the user to create multiple, identical copies of DLCS agents. The *classifierSystem* class is used for all simulations in this thesis and is presented in the Appendix.

We have chosen to use the C++ language for a variety of reasons. First and foremost, the structure of the learning classifier system itself is very object oriented. Objects such as the classifier list, message board, and environment communicate by passing messages to each other. Second, many applications of the LCS need more than one classifier system, as will be illustrated in the DLCS discussion in Chapter 4. Since the learning classifier system itself is a massively parallel paradigm, multiple-processor machines can take advantage of the object oriented programming (OOP) by assigning an LCS object to each processor in the machine. For an application like the armyant scenario, the only communication between objects is via a network, which is perfectly suited for parallel machine simulation. If each LCS in the group is put through 5000 simulations, clearly a simulation time advantage can be gained. Also, in a hardware realization, LCS objects represent separate, physical devices, and the *classifierSystem* object can be easily cross-compiled for use in an embedded system.

Finally, we use C++ because the C language is a subset of C++, and C is likely the most popular computer language in use at this time. Researchers familiar with C will be able to use the classifierSystem object with minimal learning. In addition, this

implementation will remain useful as C++ replaces C as the language of choice for programmers.

3. The LCS and the Animat Problem

3.1 Introduction

Now that we have thoroughly introduced the learning classifier system, we begin discussion of a test case which will illustrate some of the characteristics of the LCS: the animat problem. An animat, or "artificial animal," is an extremely simple autonomous robot modeled after an animal [24]. The "animat problem" describes the autonomous robot's search for a particular goal in an obstacle-filled environment. This chapter will first characterize the animat and describe in detail the animat problem. We then present the application of a learning classifier system to the animat problem and give some simulation results.

3.2 The Animat Problem

Wilson draws heavily on the similarities between an autonomous robot and an animal in describing the animat problem [24]. He sums up these similarities in the following:

> To survive in its environment, an animal must possess associations between environmental signals and actions that will lead to satisfaction of its needs. The animal is born with some associations, but the rest must be learned through experience. A similar situation might be said to hold for the autonomous robot.

Wilson states three commonalties shared between the animat and the animal. The first involves the animal's knowledge of its world. An animal is born with a small number of

"instinctive" behaviors, but learns the rest from its experiences in its environment. The animal must learn combinations of behaviors that will result in the satisfaction of its needs. Likewise, an autonomous robot will likely have to adapt to a rapidly changing environment by learning associations "on the fly." In addition, the concepts an animal and animat might discover will likely be *disjunctive*. In other words, the several reasons for an animal moving right will usually have little in common with one another. For example, an animal might move right because it smells food, or it might move right in order to avoid another animal.

The second comparison drawn between the animal and the animat involves memory and reason. Wilson assumes that an animal has only a limited amount of memory at its disposal. As the animal performs day to day tasks, it most likely will not store a *large collection* of previous experiences upon which it can draw should those situations arise again. At best, the animal may simply store a series of actions which led to the satisfaction of a particular need. By comparison, a simple autonomous robot with limited sensing capabilities will also only be able to store chains of actions that led to a particular reward in the past. Then, when the immediate environment changes, the animal and animat must learn to adapt to the new situation, often forgoing the knowledge gained from a previous experience, since this knowledge may no longer be useful. Wilson labels this assumption *incremental learning*, since animals will learn to succeed in a new situation with essentially no direct memory of the original events. While at first this assumption seems rather inaccurate, Wilson appears to be saying that an animal does not, in general, reason out a particular behavior before performing it, nor does it retain the many different experiences it underwent to achieve the satisfaction of its needs. The animal simply acts and then decides if the action (or series of actions) meets its needs. A human, on the other hand, would likely reason out, either consciously or subconsciously, the steps required to achieve a goal before proceeding. However, it is not the intent of this discussion to philosophize about the reasoning abilities of animals, and the incremental learning assumption should therefore be viewed as simply a limitation of an autonomous robot.

Note that the LCS is capable of limited planning, as discussed in Section 2.4.2. We can therefore assume that an animat controlled by an LCS will have the same sort of limited planning ability.

The third similarity drawn between an animal and an animat is that an animal usually does not have the ability to discern whether or not an action was "right" or "wrong." The only thing an animal can tell is whether or not an action provided some degree of satisfaction, even if that satisfaction is temporary. Such satisfaction of needs is called "payoff." Similarly, an animat cannot tell if a particular sequence of actions is going to move it closer to its goal; it simply tries one action after another and checks to see if that incremental move was successful. Those moves that are successful get rewarded.

With these similarities presented, Wilson sums up the animat problem as "incremental learning of multiple disjunctive concepts under payoff." The environment used to model the animat and interface the animat problem to a learning classifier system is outlined in the next section.

3.3 Applying an LCS to the Animat Problem

At this point, we are ready to introduce a simulation environment for the animat problem. Part of the motivation for this application is to illustrate some of the characteristics of the LCS. However, we hope to show that the LCS may also be used as the "brain" of an actual robot. Before such an implementation can be realized, however, the parameters of an LCS must be chosen to provide optimal robot behavior in the animat problem. The following two sections outline the single-agent animat environment.

3.3.1 Autonomous Robot Specifications

The animat presented in Section 3.2 will be realized by a *treaded* robot subject to the *nonholonomic constraint*. This constraint states that the robot's velocity is limited by its position, thereby requiring that the robot travel only in the direction in which it is pointed. Such a robot is illustrated in Figure 3.1.

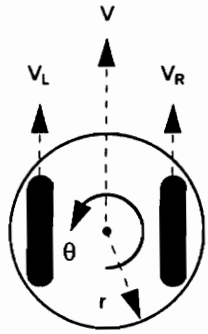


Figure 3.1. Animat kinematic model

Robot motion will be described with standard kinematic equations:

$$V = \frac{1}{2}(V_R + V_L) \tag{3.1}$$

$$\dot{\theta} = \frac{1}{2r}(V_R - V_L) \tag{3.2}$$

where V_R and V_L are the right and left robot tread velocities, respectively. V is the forward velocity of the robot. $\dot{\theta}$ is the angular velocity of rotation, and r is the radius of the robot.

The nonholonomic constraint dictates that the robot velocity be in the direction of θ, the angle in which the robot is pointed. To discretize these equations we define a time interval T where the velocities remain approximately constant:

$$V = \frac{D}{T} = \frac{P(t) - P(t-1)}{T} \tag{3.3}$$

$$\dot{\theta} = \frac{\theta(t) - \theta(t-1)}{T} \tag{3.4}$$

Where $P(t)$ is position at time t and $\theta(t)$ is direction at time t. Setting equation 3.1 equal to 3.3 and equation 3.2 equal to 3.4, we get the following:

$$P(t) \cong P(t-1) + \frac{T}{2}\left(V_R(t-1) + V_L(t-1)\right) \tag{3.5}$$

$$\theta(t) \cong \theta(t-1) + \frac{T}{2r}\left(V_R(t-1) - V_L(t-1)\right) \tag{3.6}$$

Note that these equations are exact only if the velocity remains constant throughout the interval T.

In addition to having a treaded movement, the robot will be able to sense goals and obstacles within a certain sensor range, as well as determine relative directions and distances to them. Sensing will be accomplished using arrays of infrared and ultrasonic sensors. The robot will have left and right sensors for both goals and obstacles, and the angular detection region of these sensors will overlap slightly in the middle to indicate a goal or an obstacle directly in front of the robot. Also, the distance range of the goal sensors will be much larger than that of the obstacle sensors. Figure 3.2 illustrates the animat sensor model.

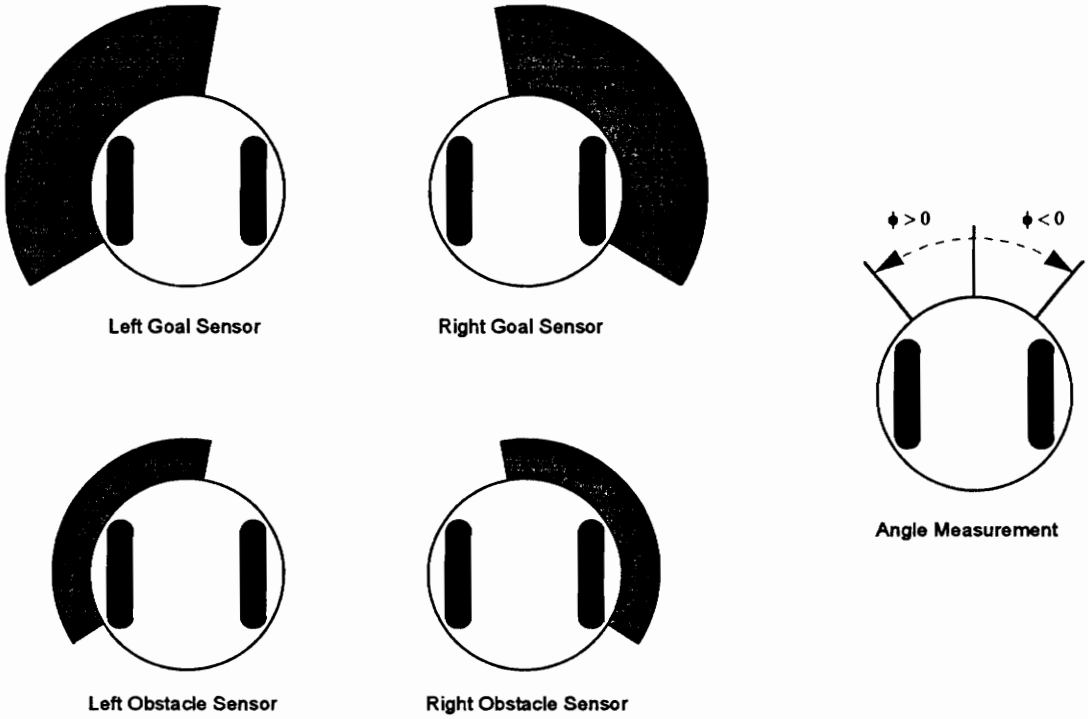


Figure 3.2. The Animat Sensor Model

Using this model, the sensor equations are

$$GL = d_g < R_g \quad \cap \quad \left(-10^\circ < \phi_g \quad \cap \quad \phi_g < R_\phi\right) \tag{3.7}$$

$$GR = d_g < R_g \quad \cap \quad \left(10^\circ > \phi_g \quad \cap \quad \phi_g > -R_\phi\right) \tag{3.8}$$

$$OL = d_o < R_o \quad \cap \quad \left(-30^\circ < \phi_o \quad \cap \quad \phi_o < R_\phi\right) \tag{3.9}$$

$$OR = d_o < R_o \quad \cap \quad \left(30^\circ > \phi_o \quad \cap \quad \phi_o > -R_\phi\right) \tag{3.10}$$

Where *GL*, *GR*, *OL*, and *OR* are flags which indicate *goal left*, *goal right*, *obstacle left*, and *obstacle right*, respectively. d_g and d_o are distances to the goal and obstacle from the robot, respectively. R_g and R_o are the distance ranges for the goal and obstacle sensors,

respectively. ϕ_g and ϕ_o are the angles to the goal and obstacle, respectively. Finally, R_ϕ is the angular range of each sensor. Note that the goal and obstacle angles are relative to the direction of the robot, as shown in Figure 3.2. Also, the obstacle sensors are given more of an overlap in the front than the goal sensors (30° as opposed to 10°) because the distance range on the obstacle sensors is much smaller than the goal sensors.

3.3.2 LCS Environment Specifications

We now focus our discussion on the environment to be used with the LCS in simulating the animat problem. We will first describe the physical animat environment and discuss the input and output interfaces for the LCS. Next we will discuss the environmental payoff function. Finally, we will conclude with typical LCS simulation settings.

3.3.2.1 The Playing Field and I/O Interfaces

The first facet of animat simulation is to devise a challenging scenario for the robot. This scenario must provide both obstacles for the robot to maneuver around and a goal to be reached. Figure 3.3 shows the typical "playing field" to be used in the animat simulations. The field is infinite in length and width and contains a goal and two long obstacles near the field's origin. While all units are dimensionless, it is helpful to think of all distances and lengths in centimeters. The robot starts in the left-hand corner of the field at (-400,-450) and attempts to reach the goal at (300,300).

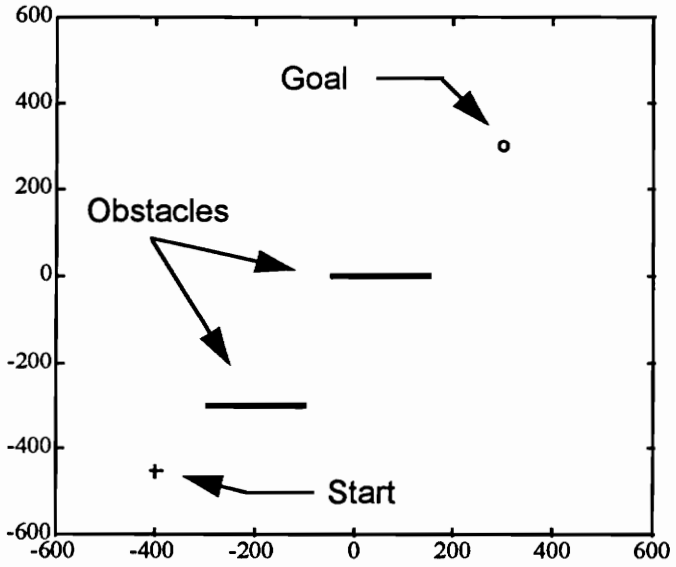


Figure 3.3. The animat playing field.

To interface this environment with the learning classifier system, the input and output interfaces must be defined. As previously stated, we can define information for internal communication as internal messages or as internal substrings. We have chosen the internal substring approach here because in the instances where we use internal communication, we want the LCS to chain together a series of actions that the animat should take. This requirement will become more evident when we discuss the concave obstacle problem in Section 3.4.3. With our choice of internal message substrings, the format for a condition word is:

$$c = GL\ GR\ OL\ OR\ I1\ I2\ I3\ I4\ I5 \qquad (3.11)$$

The first four alleles are the sensor flags from the input interface. The last five alleles comprise the internal communication substring. Each classifier will have one condition word. Using five alleles for the internal substring means that the longest possible chain

that might form would be 32 classifiers long, since $2^5 = 32$. Such a chain would have a unique link from each rule to the next. We do not, however, give any predefined meanings for the internal substring. An LCS action word has the form:

```
a =  X  X LW RW I1 I2 I3 I4 I5                    (3.12)
```

As with the condition word, the environment substring is contained in the first four alleles and the internal substring is contained in the last five. The first two alleles are not used in the environment substring, since the robot has only two actuator states, left wheel on and right wheel on. For example, LW=1 and RW=0 would turn on the left tread and turn off the right tread, thereby causing the robot to move to the right. Note that the animat is assumed to have constant tread velocity.

The input interface calculates the distances d_g and d_o and the angles ϕ_g and ϕ_o and determines if there are any goals or obstacles in the robot's field of view. Flags are set if objects are found to the left, right, or in front. The output interface takes the decision made by the LCS and translates it into tread control. The environment then uses equations 3.5 and 3.6 to calculate the new location and orientation of the robot, and the robot is rewarded or punished based on the following payoff function.

3.3.2.2 The Environmental Payoff Function

Classifier strength adjustment is performed on step 6 of the iteration cycle. The environmental payoff function tests four conditions to determine the rewards and penalties: goal distance, obstacle distance, goal angle, and obstacle crash. Rewards are given if the robot moves closer to the goal, points more directly at the goal, or moves

farther from obstacles. Penalties are given for the converse of the previous and for crashing into an obstacle. The payoff function is therefore:

$$\begin{aligned} payoff = {} & f_g W_g - (1 - f_g) P_g \quad + \quad f_\phi W_\phi - (1 - f_\phi) P_\phi \quad + \\ & f_o W_o - (1 - f_o) P_o \quad - \quad f_c P_c \end{aligned} \tag{3.13}$$

where f_g, f_ϕ, f_o, and f_c are flags indicating *closer to the goal*, *pointing more at goal*, *farther from obstacles*, and *obstacle crash*, respectively. W_g, W_ϕ, and W_o are reward amounts, and P_g, P_ϕ, P_o, and P_c are penalty amounts. In our simulations, P_g, P_ϕ, P_o are set to half of their corresponding reward values. Notice that crashing is always penalized.

This payoff is applied *only* to the classifier that posted the action. Wilson suggests a slightly different approach to payoff [24]. In his approach, payoff is distributed to all classifiers that were eligible to post and had the same action as the selected action. In Section 3.4, we will explore why our approach is better. Wilson also suggests that a fraction of the payoff be subtracted from those classifiers that were eligible to post but did not have the same action as the chosen action. However, just because an eligible rule did not have the same action as the one chosen does not mean that rule is necessarily bad. Again, our payoff function only pays the one classifier that posted the action.

3.3.2.3 LCS Settings

While individual LCS settings will be discussed when simulations are presented, some common parameter values exist between all simulations. The robot parameters, in particular, have been set to standard values:

$$
\begin{aligned}
R_g &= 2000 \text{ cm} \\
R_o &= 50 \text{ cm} \\
R_\phi &= 90° \\
T &= 15 \\
r &= 7.5 \text{ cm}
\end{aligned}
$$

Therefore, the diameter of the robot is 15 centimeters (about half a foot). Note that as long as the robot stays within the dimensions of the playing field shown in Figure 3.3, it will be able to see the goal when the goal is within the sensor's angular range.

Also, thirty-two randomly initialized classifiers are used upon simulation start. We will use internal communication only with simulations that employ the BBA. For simulations in which the BBA is disabled, the internal substring is simply set to all zeros. In either case, only one message board slot is needed since we only require one action for the actuators and the internal communication is built-in to the message format. Also we present a slight modification of the CEO since actions posted on the message board will always have a environment component. In our CEO, when the strength of the classifier that posted the rule is zero, CEO will create a new rule to be used by the environment. Obviously, this operation assumes that a zero-strength rule is not useful. Note that for cases where internal communication is used, CEO is disabled, since we do not what CEO destroying stage-setting classifiers.

3.4 Simulation Results

Having discussed the animat problem, we now turn our attention to simulation of the animat using the LCS. Our goal is to illustrate characteristics of both the animat and the LCS. We first present the "optimal solution" to the animat problem as a base case for comparison among simulations. Our first simulation addresses the payoff quantities of the environment payoff function and how they affect the number of clock tics required for the animat to reach the goal. The second simulation addresses the bucket brigade algorithm and chaining. For this simulation, we change the layout of the playing field to include a "concave obstacle." This obstacle presents a unique challenge for the animat, as will be explained. The third simulation compares our style of environment function payoff with Wilson's [24].

We should point out that the LCS code used in the simulations to follow performs all "random" operations using the standard C pseudo-random number generator, which must be initialized with a seed. Each seed produces a unique and fixed sequence of pseudo-random numbers. This sequence is "pseudo-random" because the random number generator will give the exact same sequence for the same seed. Therefore, in order to get more statistical results, most of the simulations have been run for several different generator seeds and averaged.

3.4.1 The Optimal Path Solution

As previously stated, it is possible to "program" the learning classifier system for a certain behavior. For the animat problem, we program the classifier list with a standard set of rules that enable the animat reach its destination, given the playing field in Figure 3.3. By programming such rules, we bypass the learning process and give the animat a series of logical instructions that enable it to get from start to destination while

maneuvering around obstacles. Table 3-1 lists the standard animat rules. With these rules, the animat follows an "optimal path" as shown in Figure 3.4. This optimal solution takes 97 clock ticks to complete. Note that we will use number of clock ticks as one figure of merit when rating simulation results. In the interest of computing efficiency, we assign a maximum number of clock ticks to a simulation, so that the simulation will not run indefinitely. As a second figure of merit, we define *percent success* as the percentage of simulations in a set that reach the goal before the maximum number of clock ticks.

Table 3-1. Standard animat rules.

Condition	Action	Meaning
0000	0011	Nothing visible, Move forward
0001	0001	Obstacle right, Move left
0010	0010	Obstacle left, Move right
0011	0010	Obstacle ahead, Move right
0100	0010	Goal right, Move right
0101	0011	Goal right, Obstacle right, Move forward
0110	0010	Goal right, Obstacle left, Move right
0111	0010	Goal right, Obstacle ahead, Move right
1000	0001	Goal left, Move left
1001	0001	Goal left, Obstacle right, Move left
1010	0011	Goal left, Obstacle left, Move straight
1011	0001	Goal left, Obstacle ahead, Move left
1100	0011	Goal ahead, Move forward
1101	0011	Goal ahead, Obstacle right, Move forward
1110	0011	Goal ahead, Obstacle left, Move forward
1111	0010	Goal ahead, Obstacle ahead, Move right

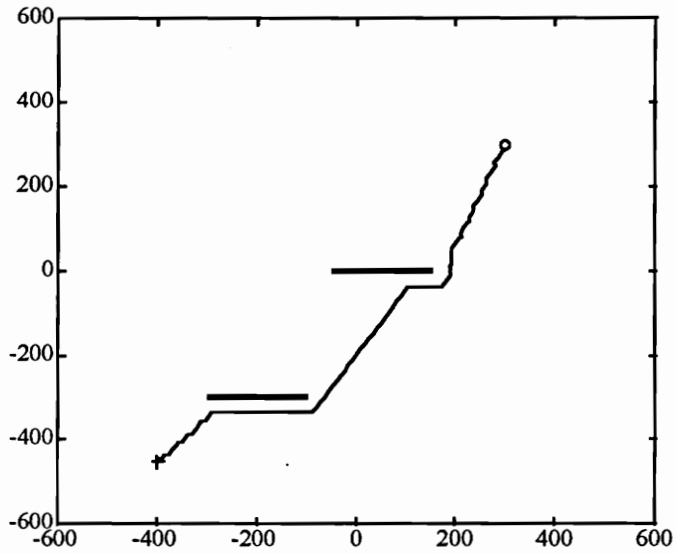


Figure 3.4. Optimal animat path.

While number of clock ticks does provide us with some insight into the rate of convergence of a simulation, we also need a method to characterize the path taken by the animat. To establish a figure of merit for the animat path, we introduce the curve shown in Figure 3.5 which plots distance to the goal vs. time (clock ticks). The animat starts at approximately 1026 units from the goal. As can be seen from the figure, the distance to the goal decrease steadily as time progresses. The curve would be linear if there were no obstacles in the animat's path. As a figure of merit for the path, we integrate the distance-time curve and designate this quantity as *path area*. For the optimal animat path, the path area is 50556.3. All path areas specified in the simulations will be normalized by this optimal path area, *e.g.* a value of 2.5 means the simulation's path area was 2½ times the optimal area. Also, statistics will only be calculated for those simulations that are successful. In other words, when a mean number of clock ticks is given for a set of simulations, that mean is calculated over those simulations that converge before the maximum number of clock ticks.

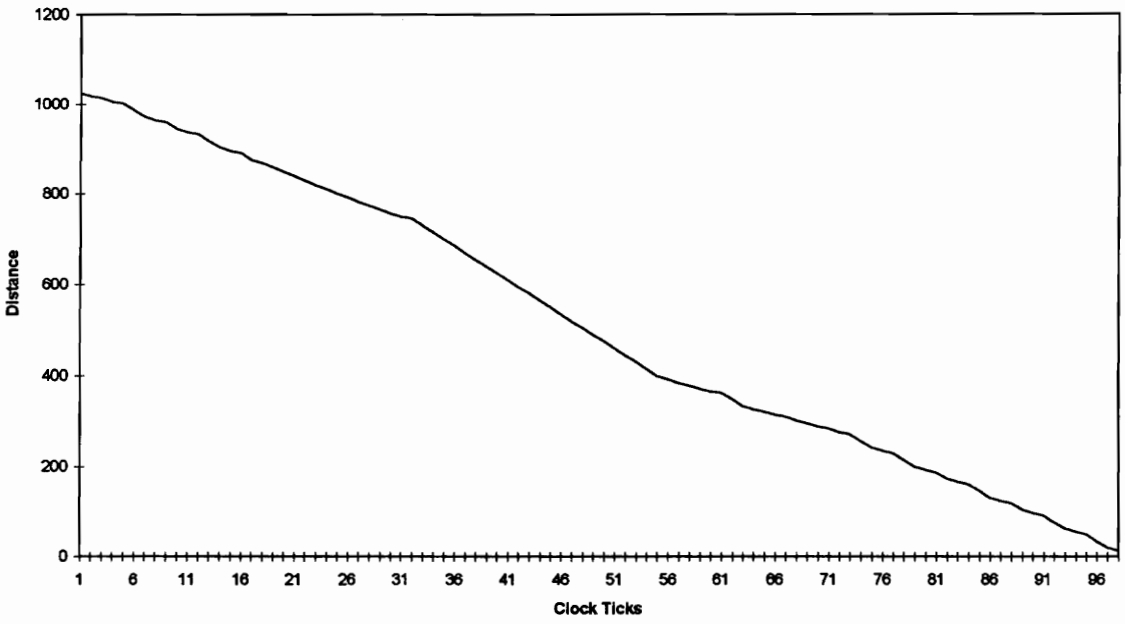


Figure 3.5. Distance vs. time curve for optimal animat path.

The simulations presented in the following sections and the next chapter differ from the optimal solution in that the classifier list is *not* programmed with the rules above. Instead, the classifier list is filled with random rules, thereby requiring that the LCS *learn* the correct behavior rather than be *programmed* with it. We simply use the optimal path as a basis for comparison.

3.4.2 Environment Payoff Parameter Variation

For our first simulation, we will examine the environment payoff function. As given in equation 3.13, the environment payoff function is influenced by seven payoff quantities, W_g, W_ϕ, W_o, P_g, P_ϕ, P_o, and P_c. These are simply the reward and penalty quantities for distance to goal, direction to goal, distance from obstacles, and obstacle crashes. In all of the simulations presented here, we set the obstacle reward to zero, since

an animat will often have to move closer to obstacles in order to reach a goal. Also, the distance and direction penalties are set to one half of their corresponding reward values. Therefore, environment payoff is a function of only three independent parameters: the distance to goal reward W_g, the direction to goal reward W_ϕ, and the crash penalty P_c.

Since the purpose of the animat problem is to reach the goal in the minimum number of clock ticks, we would like to find an optimal combination of payoff values that provides this minimum. For the simulation presented in this section, we have performed a parameter sweep over the three independent variables of the payoff function. This parameter sweep can be visualized as a four-dimensional data set, where the first three dimensions are the independent variables, and the fourth dimension is the number of iterations N for that particular combination of rewards and penalties:

$$N = f\left(W_g, W_\phi, P_c\right) \tag{3.14}$$

Graphically, we can visualize this data set as a volume of varying density, where each dimension of the volume is a payoff quantity, and the densities are the number of iterations for a particular combinations of payoffs. For the simulation, we vary W_g, W_ϕ, and P_c from 0 to 10 in steps of 1. We therefore get a data set consisting of $11^3 = 1331$ points. For each point in this set, we run ten simulations, each with a different random seed, and average the results. Finally, we have disabled genetics in order to reduce the number of extraneous parameters that might influence the solution. Instead, we rely solely on the CDO and CEO rule discovery operators for all new rule creation.[2]

Figure 3.6 shows the results of the simulation. We use color to represent the number of iterations, where the colors toward the purple end of the spectrum represent the

[2] As an interesting aside, if each simulation takes on average 1000 clock ticks to finish, and we run 13310 simulations, then there are 1.33×10^7 clock ticks total. On a Sparc 10 workstation, our code gets around 70 iterations per second running "un-niced." This corresponds to about 52.8 hours of continous simulation. When the process is niced, this simulation time doubles. Clearly, high-resolution data sets require significant processing ability.

smallest number of iterations. We set the maximum number of iterations to 2000, but choose a color range up to 1000 to improve the resolution of the lower end of the scale. The figure shows an iso-surface for parameter combinations which result in $N \leq 350$. The minimum number of clock ticks for this data set is approximately 188, and this minimum occurs when $W_g = 1.0$, $W_\phi = 1.0$, and $P_c = 3.0$. However, as can be seen from the figure, there are several good parameter combinations, most of which have larger goal and direction rewards and a small crash penalty. This result implies that the crash penalty is not as useful in the particular obstacle environment used for these simulations. However, in an environment cluttered with obstacles, a crash penalty might prove to be more useful. For comparison, we have also included a worst-case iso-surface for parameter combinations which give $N \geq 500$ in Figure 3.7.

The overall statistics for this data set are given in Table 3-2. We see a modest success rate of about 60%, with an average number of iterations about four times optimal. Clearly some of the parameter combinations are not useful. Note that unless otherwise noted, we will use a goal reward of 2.0 and a direction reward and crash penalty of 5.0 for all subsequent simulations. These values provide a modest average solution of about 480 iterations per second.

Table 3-2. Parameter variation statistics

Success Rate	59.6%
Number of Clock Ticks	$\mu = 405.13$ $\sigma = 296.66$
Path Area	$\mu = 4.47$ $\sigma = 3.88$

Finally, we should point out that the minima shown in the iso-surface of Figure 3.6 are only local minima. Several other LCS parameters were held constant in order to generate this plot, *e.g.* classifier list length, message board length, bid constant, *etc.* If we were to change these parameters or enable the genetic algorithm, the minima could possibly shift.

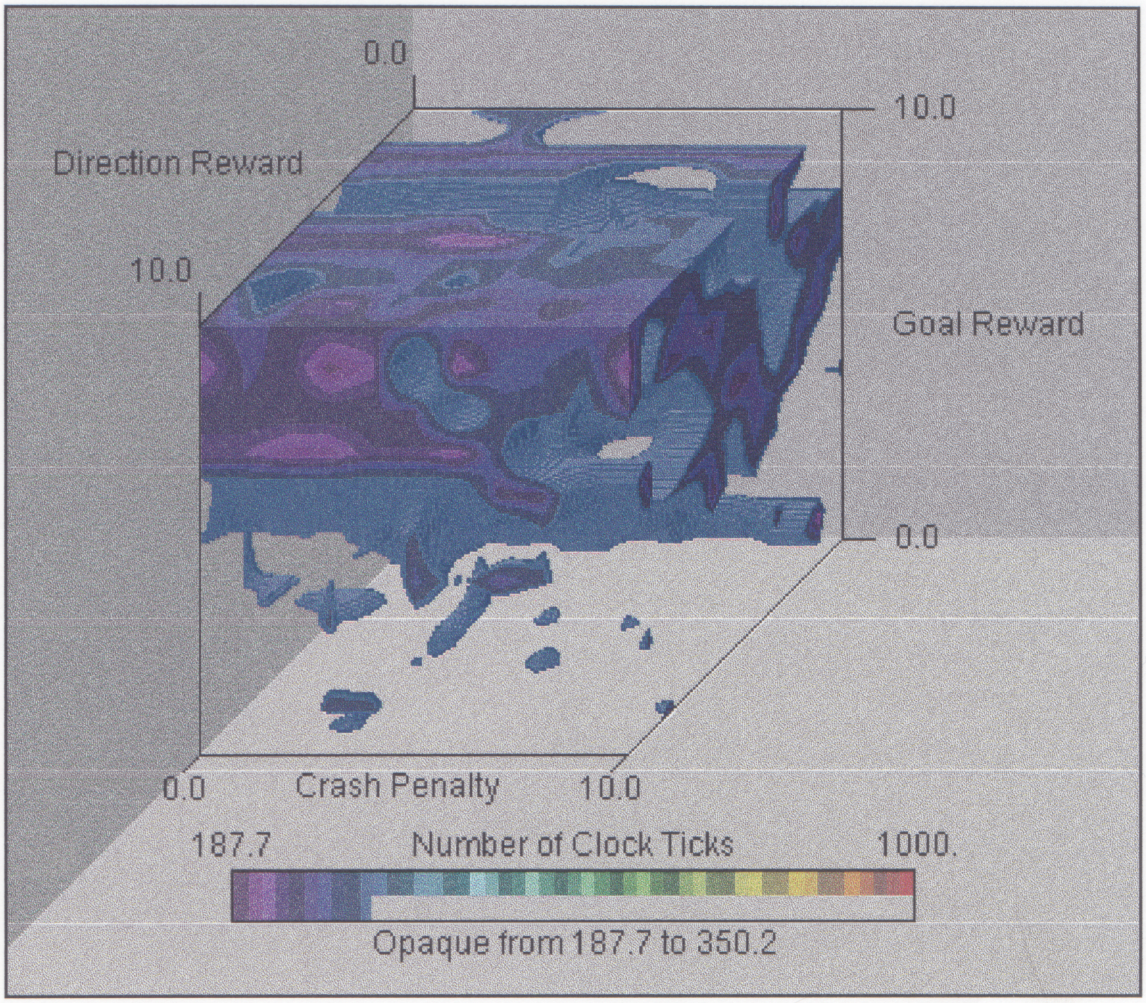


Figure 3.6. Iso-surface for good parameter combinations.

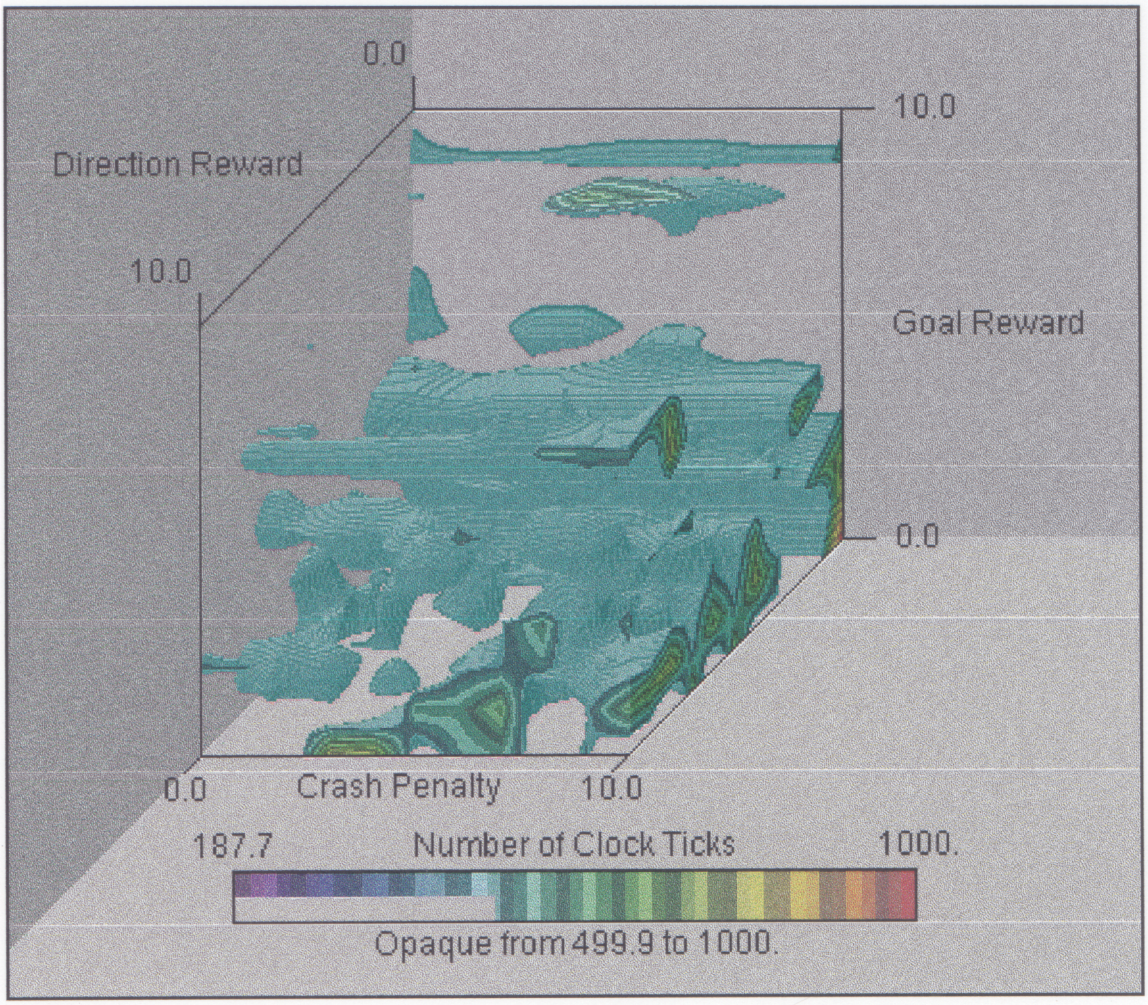


Figure 3.7. Iso-surface for poor parameter combinations.

3.4.3 The Bucket Brigade Algorithm and the Concave Obstacle

Our second simulation addresses one of the more obscure parts of the learning classifier system, the bucket brigade algorithm. As previously stated, the BBA is designed to encourage the formation of rule chains. It accomplishes this end by providing a method for the strengths of later rules in the chain to propagate back to the early stage-setting rules. In addition, bids grow in the chain as the chains grow. The three major requirements for a chain to be successful under BBA encouragement are: 1) the last rule in the chain must have a net strength increase on each iteration of the chain, 2) the chain must be executed repeatedly, and 3) the saturation level for rule bids must be set to a value between the largest possible punishment and the largest possible reward (see Section 2.4.2). Because of these stringent requirements, most research involving the BBA has failed to provide cases where a chain naturally evolves from an initial set of random classifiers.

The purpose of this simulation is to present a case of the animat problem where a chain would be required for success. We introduce the *concave obstacle problem* for this purpose. As shown in Figure 3.8, a single, concave obstacle is put into the playing field, with the animat on the concave side and the goal behind the obstacle. Since the environment payoff function only encourages rules that move closer to the goal and point more directly at the goal, the natural inclination of the animat will be to move into the obstacle. Since the obstacle is concave, the animat cannot easily escape it by simply moving left or right. The animat must instead move in the opposite direction of the goal in order to get around the obstacle. As will be shown, it is possible for *unchained* random rules to evolve into such a configuration, but since some of these rules have the robot moving away from the goal, they are repeatedly punished until their strengths go to zero and they are replaced. On the other hand, a *chain* of rules executed sequentially can enable the animat to repeatedly maneuver itself from behind the obstacle.

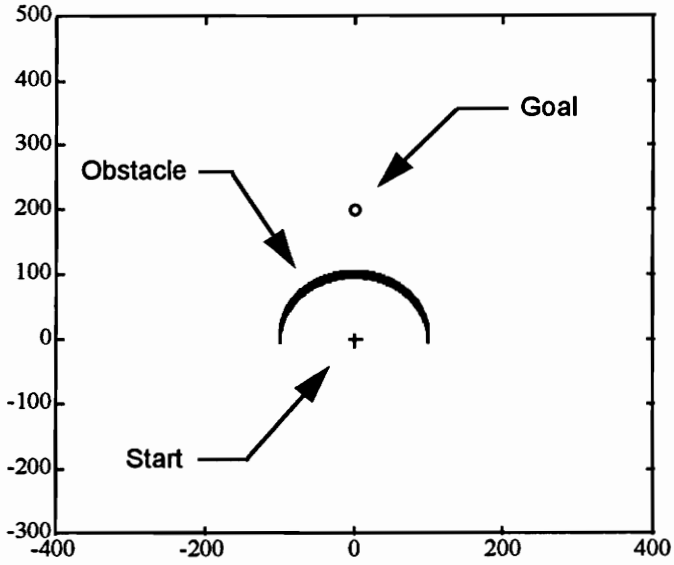


Figure 3.8. The concave obstacle playing field

To illustrate this situation, we present two simulations. The first involves the use of rules for which no BBA is used. The rules rely solely on environment payoff for reward. The second uses rules that are chained together in the classifier list. These rules get reinforced by the BBA and by the environment. In each simulation, the rule base is executed repeatedly to illustrate the effect of time on the original rules. Table 3-3 shows the rules to be used for each case. For the BBA case, the rules are listed in their order of execution in the chain from top to bottom. Note that we have used environment and internal substrings for these rules. We separate these substrings with a comma to facilitate interpretation.

The non-BBA rules evolved from repeated simulation over a random rule set during some of our initial BBA research. They have been given the maximum strength allowed (100.0) so that they have the best chance of survival. The BBA rules have been programmed to execute a successful chain for the purpose of illustrating the BBA and have been given a nominal strength of 10.0. Note that these rules could be reduced by

standard truth table techniques, but we keep them linear to make them easier to follow. We use the standard reward and penalty values for the non-BBA case, and the bid cap is also set at 100.0. In the BBA simulation, we meet the BBA requirements by settings the bid cap to 30.0 and the goal and direction rewards to 20.0. (We can ignore the crash penalty because the programmed chain does not move the robot into an obstacle.) With these settings, the maximum possible punishment is 0.5(20.0 + 20.0) = 20.0, and the maximum possible reward is 40.0. The bid cap falls exactly between these two extremes, as suggested by our rule of thumb for setting the BBA parameters.

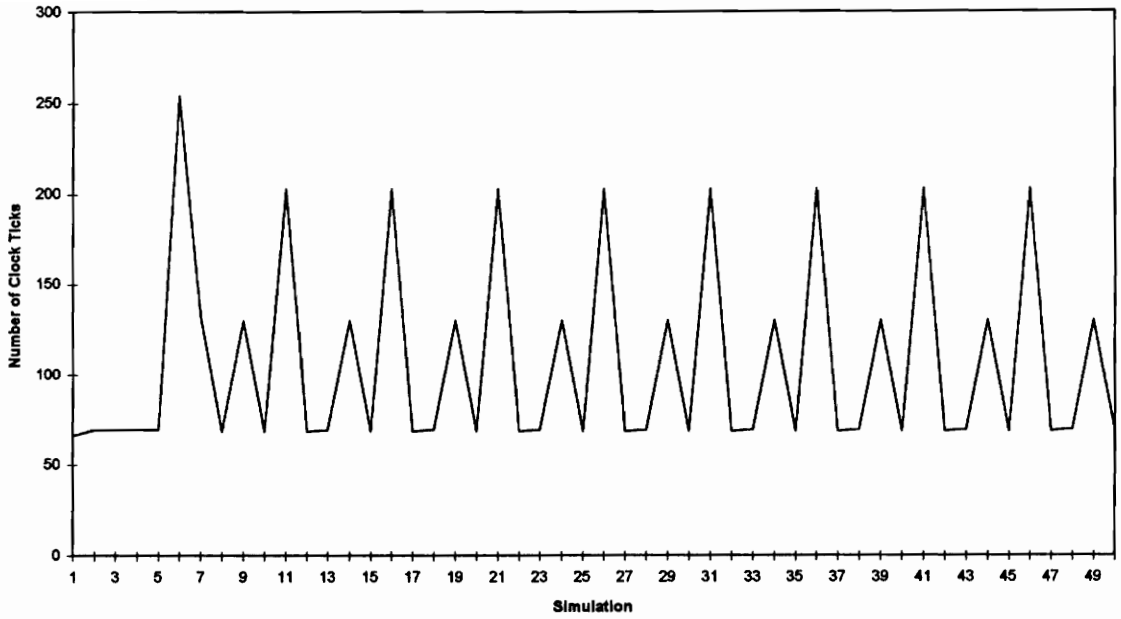
Table 3-3. Rule sets for BBA and non-BBA cases

BBA		
Condition	**Action**	**Strength**
1000,00000	0010,00001	10.0
0000,00001	0011,00010	10.0
0000,00010	0011,00011	10.0
0000,00011	0011,00100	10.0
0000,00100	0011,00101	10.0
0011,00101	0010,00110	10.0
0010,00110	0011,00111	10.0
0010,00111	0011,01000	10.0
0000,01000	0011,01001	10.0
0000,01001	0001,01010	10.0
0010,01010	0011,01011	10.0
0000,01011	0011,,01100	10.0
0000,01100	0011,01101	10.0
0000,01101	0011,01110	10.0
0000,01110	0001,01111	10.0
1000,01111	0011,10000	10.0
1000,10000	0011,10001	10.0
1000,10001	0011,10010	10.0
1000,10010	0011,10011	10.0
1000,10011	0011,10100	10.0
1000,10100	0011,10101	10.0
1000,10101	0001,10110	10.0
1000,10110	0001,00000	10.0

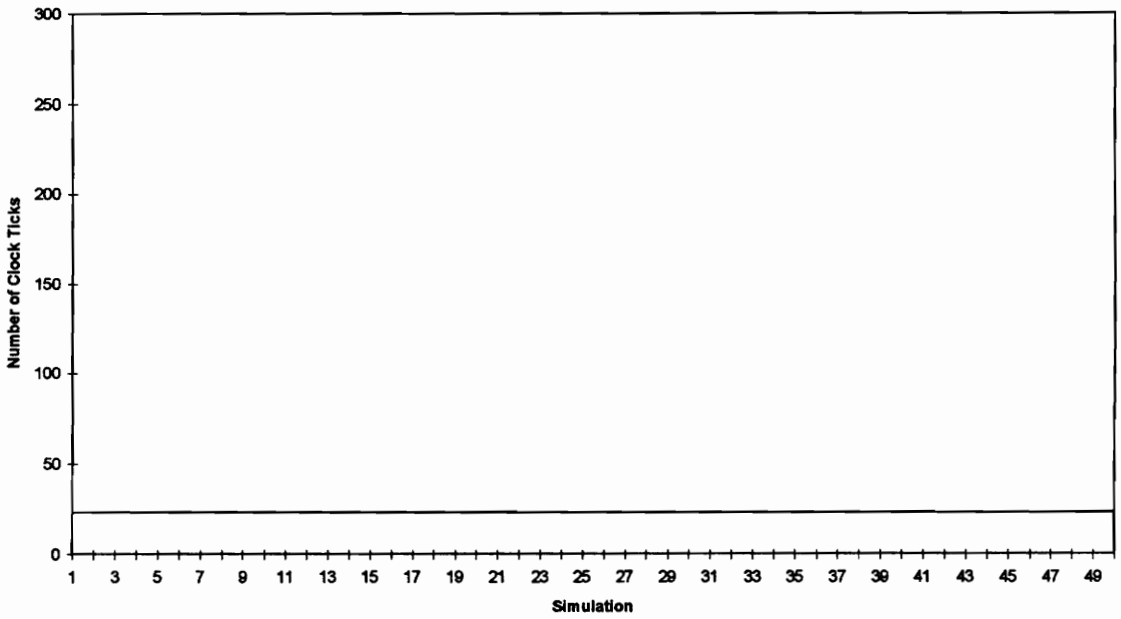
No BBA		
Condition	**Action**	**Strength**
1#00	0001	100.0
01#0	0010	100.0
##00	0010	100.0
###1	0001	100.0
1#00	0001	100.0
1010	0001	100.0

Figure 3.9 shows the number of clock ticks for each of fifty consecutive simulations of each original set of rules. Figure 3.10 shows sample animat paths taken. As expected, the non-BBA case maintains its original rules only for a few simulations and then loses part of those rules. At this point, the number of clock ticks drastically increases because the LCS has to re-learn the rules. This cycle continuously repeats itself. The number of clock ticks remains constant in the BBA case, however, since the BBA reinforces the chain, and the rules are therefore maintained.

The final rules for the BBA case are given in Table 3-4. Those rules which are rewarded by the environment have saturated at the maximum strength of 100.0. Those rules punished by the environment but reinforced by the BBA saturate at the bid cap of 30.0. For the non-BBA case, Table 3-4 shows the rules at the end of simulation 5—the simulation immediately preceding the point where the rule base is "corrupted" by the rule discovery operators. The third rule only has a strength of 9.0 and quickly gets replaced on the next simulation.



(a)



(b)

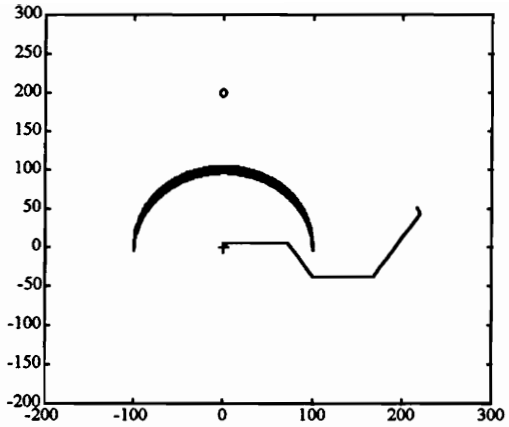
Figure 3.9. BBA comparison results. a) Non-BBA case. b) BBA case.

Table 3-4. Final rules for BBA case, intermediate rules for non-BBA case.

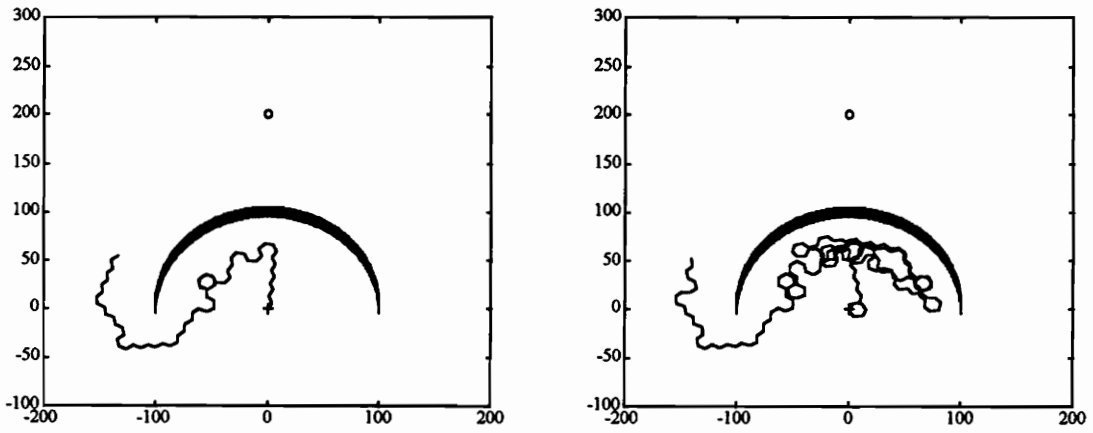
BBA		
Condition	**Action**	**Strength**
1000,00000	0010,00001	100.0
0000,00001	0011,00010	30.0
0000,00010	0011,00011	30.0
0000,00011	0011,00100	30.0
0000,00100	0011,00101	30.0
0011,00101	0010,00110	30.0
0010,00110	0011,00111	30.0
0010,00111	0011,01000	30.0
0000,01000	0011,01001	30.0
0000,01001	0001,01010	100.0
0010,01010	0011,01011	30.0
0000,01011	0011,,01100	30.0
0000,01100	0011,01101	30.0
0000,01101	0011,01110	30.0
0000,01110	0001,01111	100.0
1000,01111	0011,10000	100.0
1000,10000	0011,10001	100.0
1000,10001	0011,10010	100.0
1000,10010	0011,10011	100.0
1000,10011	0011,10100	100.0
1000,10100	0011,10101	100.0
1000,10101	0001,10110	100.0
1000,10110	0001,00000	100.0

No BBA		
Condition	**Action**	**Strength**
1#00	0001	100.0
01#0	0010	100.0
##00	0010	9.0
###1	0001	99.0
1#00	0001	100.0
1010	0001	100.0

Finally, we present the actual simulation paths the agents take. Figure 3.10 (a) shows the animat path for the BBA rule base. The plots in (b) show the animat paths for the non-BBA rule base before and after part of the rule base gets modified in simulation 11, respectively. As can be seen, the non-BBA path is not very direct after the rule base is modified.



(a)



(b)

Figure 3.10. a) Animat path for the BBA rule base. b) Non-BBA animat path at start of simulation and after altered rule base (simulation 11).

3.4.4 The Wilson Payoff Function

In Section 3.3.2.2, we briefly discussed the difference between the environment payoff scheme used in this thesis and Wilson's environment payoff scheme in [24]. Our payoff mechanism determines the payoff quantity as given by equation 3.13 and then pays the amount only to the rule that posted the current environment action. Wilson suggests a distributed payoff approach in which the payoff quantity be given to all classifiers in the eligible set $E(t)$ that had the same action as the action selected. Wilson also indicates that a fraction of this payoff quantity be deducted from those classifiers in $E(t)$ that have a different action than the one selected. Thus, when the payoff is positive—the action taken is rewarded by the environment—this payoff scheme has the effect of rewarding all classifiers that would have resulted in the same "good" action. Conversely, those classifiers that would have taken a different action under the same environment conditions are punished. When the payoff is negative, the exact opposite occurs. Those rules taking a "bad" action are punished, and those rules taking a different action (potentially good or bad) are rewarded.

We chose not to use Wilson's payoff scheme for two reasons. First, Wilson makes the assumption that all rules resulting in the same action are valuable in the current environment context. For example, say on a particular clock tick, the rules 1100 | 0011 and 11## | 0011 are eligible for posting. These rules both suggest moving forward if the goal is ahead of the animat; however, the second rule also allows for the possibility that an obstacle might be in the path. Under the Wilson payoff scheme, both of these rules would be rewarded, assuming the animat did not crash into an obstacle. However, if the second rule is selected at a later time and there are obstacles in the path, this rule would likely result in a crash. Our second reason for not using this payoff scheme is that it makes the assumption that those rules which would have taken a different action than the one chosen are bad if the chosen action is good, or good if the chosen action is bad. As we shall see from the simulation results, this assumption is not always valid. We felt that the

environment should subject one rule at a time to the payoff function, thereby allowing more accurate rule evaluation.

In order to test our hypothesis, we performed a comparison between the two payoff schemes. Since the Wilson payoff scheme is somewhat non-standard, we had to make a couple of assumptions in order to provide an accurate comparison. First, Wilson designates that a quantity *e* be deducted from the strengths of all eligible classifiers that had the same action as the one chosen. This strength adjustment appears to be similar to the bid tax, except that only those bidding classifiers with the same action as the posted action pay the tax. We decided to exclude this quantity from the comparison, since we are not using bid taxes with the our payoff function. Second, Wilson's simulations involve a different environment scenario than our animat problem (although he suggests that his payoff scheme can be used with the animat problem as well). Therefore, his payoff quantities are not directly applicable to the animat problem. To provide the most unbiased comparison, we use our environment payoff function to calculate the payoff quantity. In our payoff scenario, this payoff quantity is applied only to the rule that posted the action. In the Wilson payoff scenario, this value is applied to all eligible rules having the same action as the posted action. In addition, the payoff quantity is deducted from those eligible rules having different actions than the posted action. We therefore simplify the comparison to address only single and distributed payoff. Also, we have once again disabled genetics to reduce the number of degrees of freedom in the problem.

The results of the simulation are given in Table 3-5. The standard animat playing field is used, and four hundred simulations are performed for each payoff style, with each simulation having a different random seed. From the table, we can conclude that each payoff scheme has its merits. The Wilson scheme is more successful on the average, however, the standard payoff scheme has a slightly better mean number of clock ticks, and a much better standard deviation on the number of clock ticks. The path area results are very significant. The standard payoff scheme is over one half of the optimal path area better than the Wilson scheme. In examining some of the paths in the Wilson simulation,

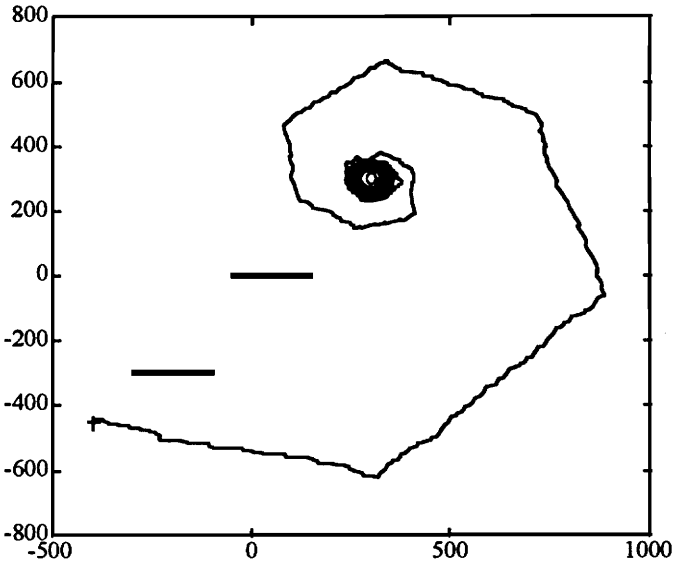
this difference becomes evident. Figure 3.11 illustrates a "bad path" and the corresponding distance plot. As can be seen, the animat finds its way to the goal fairly effectively, but continues to circle the goal repeatedly without actually reaching it. The reason the animat encircles the goal is because the Wilson payoff scheme reinforces identical actions for a particular environment sensor condition and punishes all other actions, thereby causing one action to dominate in terms of strength for a particular sensor condition. In the case shown in the figure, two rules 1### | 0010 and 1### | 0011 exist for the goal left condition. Once the animat has reached a point close to the goal, these rules are continuously selected, but since neither moves the robot closer to the goal or points the robot more directly at the goal, the selected action gets punished while the unselected action gets rewarded. Therefore, the rule cycle is cyclical and the robot reaches an equilibrium around the goal, without ever actually reaching the goal. This equilibrium situation is not present in our payoff scheme.

Table 3-5. Wilson payoff comparison results.

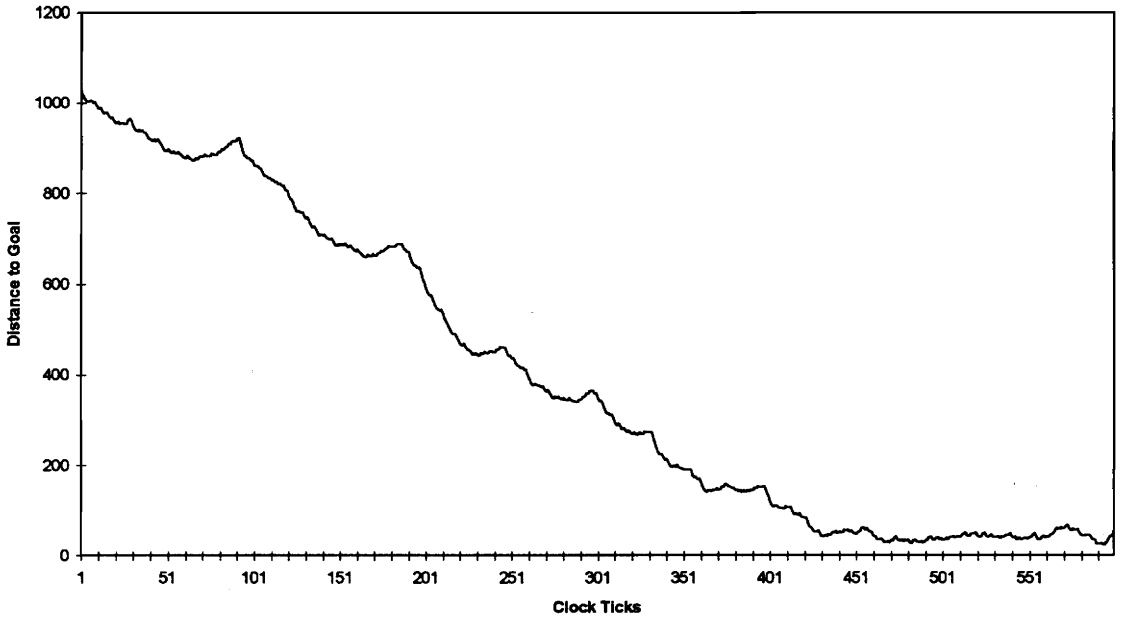
	Wilson	Standard
Success Rate	62.8%	52.0%
Number of Clock Ticks	$\mu = 567.53$ $\sigma = 388.56$	$\mu = 552.40$ $\sigma = 347.76$
Path Area	$\mu = 6.76$ $\sigma = 5.06$	$\mu = 6.16$ $\sigma = 3.84$

To be fair, we should point out that not all Wilson simulations are plagued by the problem shown in the figure. In fact, both the Wilson and the standard payoff schemes have extremely good cases, as shown in Figure 3.12 and Figure 3.13. We should also point out that the major reason for the low success rate of both payoff schemes is the fact that the genetic algorithm is disabled. It is possible for the animat to get caught in a loop where it simply circles endlessly, *i.e.* one actuator turns on and stays on. (This looping problem is not the same as reaching equilibrium around the goal.) These loops get

repeatedly rewarded and punished by the environment payoff function (by getting closer the goal, but pointing farther away), so the net strength of the rules never gets reduced to the point where CEO can replace them. Normally, genetics handles this situation by periodically mutating the rules. However, we removed genetics because of the large number of degrees of freedom it introduces into the LCS, so we must tolerate the low success rate.

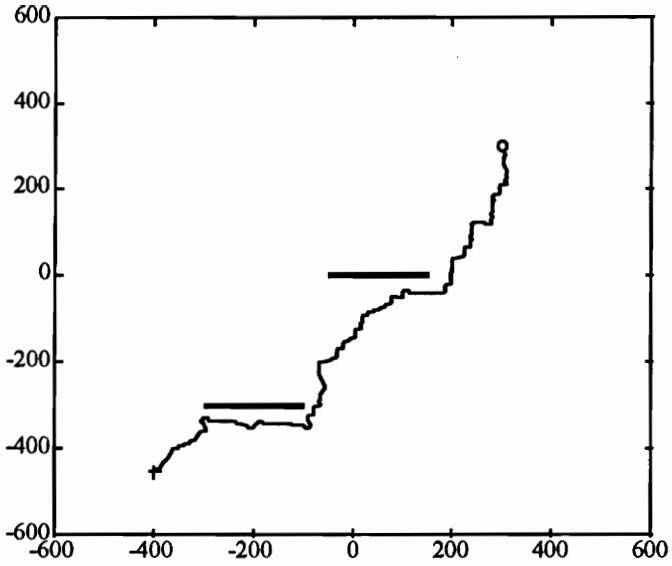


(a)

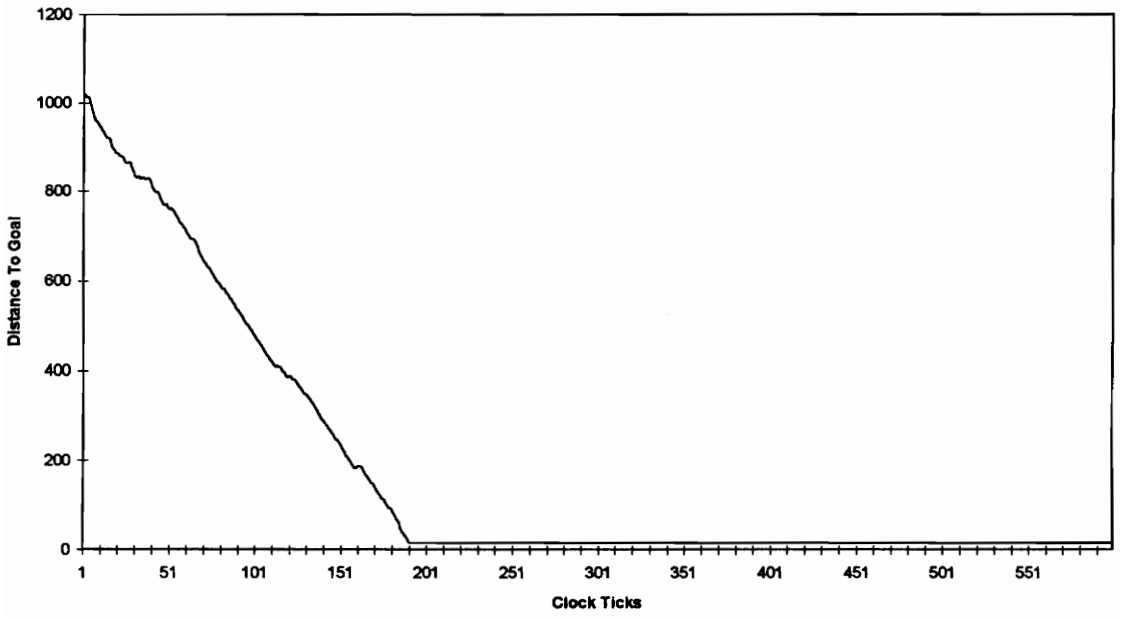


(b)

Figure 3.11. a) Wilson simulation with a bad path. b) Distance vs. time graph.

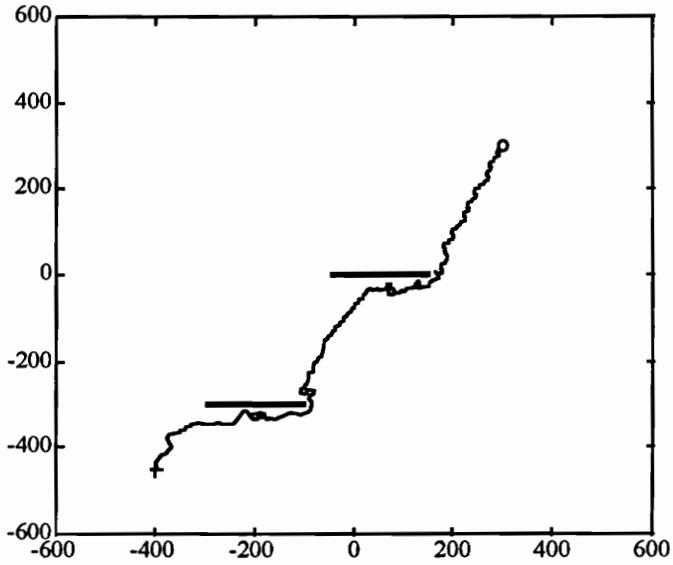


(a)

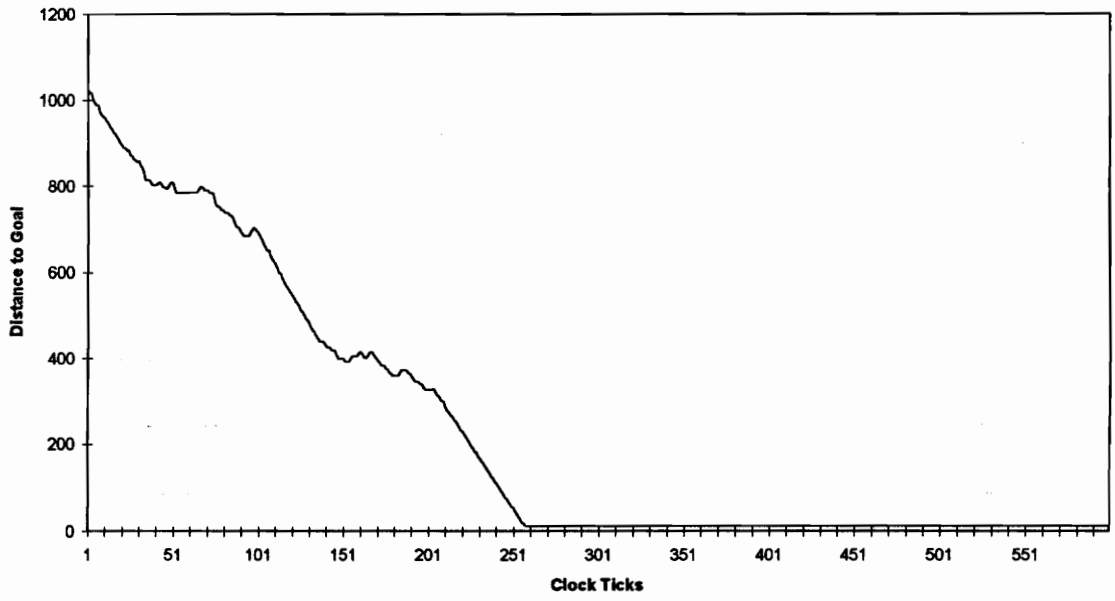


(b)

Figure 3.12. a) Wilson payoff animat path. b) Distance-Time plot.



(a)



(b)

Figure 3.13. a) Standard payoff animat path. b) Distance-Time plot.

4. The Distributed Learning Classifier System

4.1 Overview

Having discussed the standard learning classifier system in great detail, we now turn our attention to an extension of the LCS new to the field, the *Distributed Learning Classifier System* (*DLCS*). We have chosen to design the DLCS in order to provide a framework to enable multiple classifier system agents to work collectively to solve tasks. Collective task solution is the primary focus of Distributed Artificial Intelligence (DAI) research [12], and much work has been done in the area [3]. However, the learning classifier system itself has not previously been extended to the realm of DAI. The architecture we describe in the next few sections is a novel approach to the collective task solution problem which takes advantage of the classifier system's intrinsic message-passing and reward system.

We should point out before introducing the architecture that our design goal is to create a group of distributed, autonomous agents that have no central control and are relatively "anonymous." No central control means that the agents are more resilient to failure, *i.e.* if one agent "goes down" for a period, the other agents can compensate for it. We do not wish to explicitly select a leader from a group of agents, but instead allow the agents themselves to form leader-follower groups as necessary. Agents should be individualistic in that they operate autonomously, but should also be anonymous in that one agent cannot differentiate between other agents in the group.

We derive these design goal from Souza and Talukdar's discussion of "super-agents" in [22]. A super-agent is a collection of individual agents with limited tasks that work together to solve larger tasks. Souza and Talukdar differentiate between hierarchical and non-hierarchical organizations of agents. A hierarchical organization

consists of several layers, with each layer controlling the one below it. The task to be solved is sub-divided at each layer of the hierarchy until the smaller tasks are simple enough for a single agent to handle. This approach requires a supervisor or central controller to handle task division and can be fragile, since poor supervisor decisions propagate throughout the hierarchy. Souza and Talukdar describe a particular type of non-hierarchy called a *hetrarchy*—a single-layer organization of autonomous agents with no leader. They liken this organization to the type found in social insects such as army ants and flocks of birds. They maintain that the hetrarchy approach is better because it eliminates the fragility and overhead associated with the required central controller of the hierarchical organization. Therefore, the DLCS architecture is designed with the hetrarchy approach in mind to create a collection of autonomous agents. With the hetrarchy, we can ensure that agents will adapt more readily to changing environment scenarios and requirements.

We begin this chapter with a discussion of the architecture design of the distributed learning classifier system. We then present some simulation results illustrating the use of the DLCS in a multiple-agent scenario.

4.1.1 The Open Systems Interconnection (OSI) Protocol Model

In order to justify the architecture of the DLCS, we present a brief discussion of network theory taken from [2]. To encourage compatibility between various networks, the International Standards Organization has developed a network protocol model known as *Open Systems Interconnection* (*OSI*). This model divides the tasks of a network protocol into a seven-layer architecture as shown in Figure 4.1. The top layers correspond to the software facet of a protocol, while the bottom layers correspond to hardware. The model describes the flow of a message to be transmitted to or received from a network.

As a descriptive example, we will describe the process an electronic mail message would undergo according to this model.

When a user types and sends an e-mail message, the program (*application*) layer sends this message to the *presentation* layer, where the message is converted into a common data format for the network. Machines can have different character formats, such as ASCII or EBCDIC, and data must be converted to a standard network format so different machines will know how to interpret the message. The presentation layer also encrypts or decrypts data if necessary. Next, the message goes to the *session* layer, which handles connection establishment, access rights, and charges for network use. The *transport* layer then takes the message, partitions it into packets, and multiplexes it with any other messages to be sent. The *network* layer provides routing and flow control for these packets. The *data link* layer provides breaks the packets into bits and controls reliable transmission of these bits. The *physical* layer is a combination of a modem and a communications channel such as a coaxial or fiber optic cable. As shown in the figure, network *hosts* (the end-connections of the network) have all seven OSI layers, whereas network *switches* only need the bottom three, since they just perform routing and queuing.

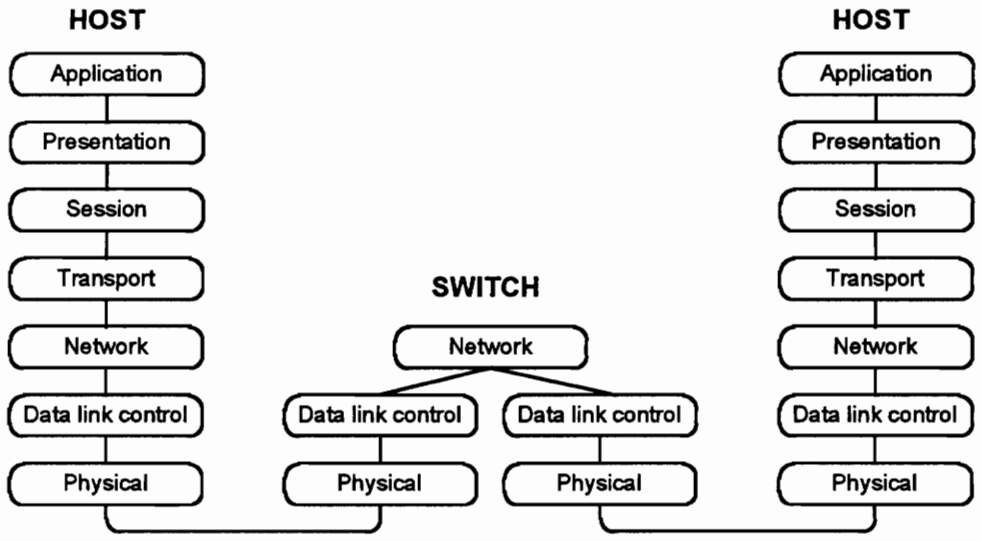


Figure 4.1. The OSI protocol model.

Most network protocols, such as TCP/IP, DECNET, and ARPANET, follow the OSI model, although the boundaries between the seven layers are often rather nebulous. The key feature of the model, however, is that it dictates from a network software development point of view, how access to the network can be achieved. A programmer developing distributed software need not be concerned with how the software's network messages are going to be sent from source to destination. He or she simply passes these messages to the presentation layer from the application, and the network protocol handles the rest. The common network protocol used on the Internet is TCP/IP (Transport Control Protocol / Internet Protocol). An application uses a set of functions known as the *socket* interface to interface TCP/IP. With sockets, all an application must do is specify a destination for the message and the socket interface handles the remainder of the transmission. Of course, the programmer must choose the type of delivery service desired and may wish to verify message delivery, but socket and TCP/IP handle actual message packetizing and routing. Reception works in the same manner; socket functions are called to get received messages.

4.1.2 The Network Interface

In keeping with the OSI model, we have designed the DLCS to take advantage of interfaces like socket. The DLCS extends the standard learning classifier system with the addition of a *network interface*, as shown in Figure 4.2. As will be discussed later, there are three types of messages that can be sent by a DLCS agent: action messages, classifier messages, and BBA payoff messages. Action messages move from the network interface to and from the message board. Classifier messages move from the network interface to and from the classifier list. BBA payoff messages involve strength adjustment based on the bucket brigade algorithm. Since the BBA is an internal credit assignment algorithm,

BBA message passing is indicated in the figure by a dashed line between the network interface and the classifier list.

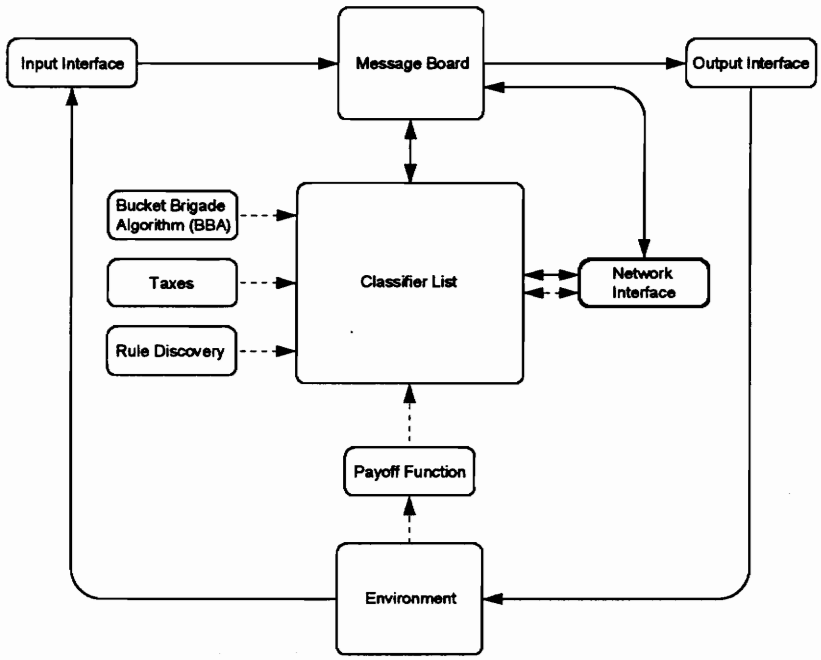


Figure 4.2. The Distributed Learning Classifier System.

The network interface accesses socket (or equivalent) functions for transmission and reception of the DLCS-specific messages listed above. Since almost all modern networks are asynchronous and subject to delay, the network interface includes separate *transmit* and *receive queues* which buffer outgoing and incoming DLCS messages, respectively. As will be discussed, agents do not know ahead of time how many messages they might receive during a clock tick, so it is not possible for an agent to wait for "all messages" to arrive before proceeding. The only exposure DLCS agents have to the network is through the network interface queues. The queues interface with a parent process whose sole responsibility is to transmit messages in the transmit queue and place newly-received messages in the receive queue. In some respects, the network interface is

similar to the environment interface in that the network interface is the liaison between the network and the DLCS, instead of the environment and the DLCS.

4.2 Agent Identification

In TCP/IP, destinations are given a unique 32-bit integer called an IP address. Usually a symbolic name, like *armyant.ee.vt.edu* or *whitehouse.gov*, is associated with this 32-bit number so that the IP address is easier to remember. The socket interface uses a name server to translate these symbolic names into their corresponding numbers. In general, a single machine is given a single IP address, and one refers to this machine as a *host*.

In keeping with this widely-used standard, we give each DLCS agent a unique 32-bit address. One can then visualize DLCS agents as hosts connected to a network, with each agent having its own identification. This identification is necessary for network organization, but in keeping with our desire to create a set of distributed, autonomous agents, we usually make agents "anonymous" by broadcasting DLCS messages. A broadcast goes to every host on the network, instead of just to a specific address. We will discuss this concept further in the next section, but the key point here is that the DLCS agents do not have any central controller. They operate as independent network hosts, broadcasting messages to one another as necessary. By contrast, most DAI architectures have some form of centralized control that manages agents. Lack of central control means that agents are free to organize themselves as they see fit, even changing group size if necessary. Also, without the dependency on a central controller, a network of DLCS agents is more resilient to system crashes and network delay.

4.3 Network Message Types

Before detailing how network message passing fits into the classifier system execution cycle, we must briefly describe the types of message employed by the DLCS. There are three types of messages that can be passed between DLCS agents: action messages, classifier messages, and BBA payoff messages. We describe each type below.

4.3.1 Action Messages

Action messages, as discussed in Chapter 2, are those instructions selected from the classifier list to be sent to the environment interface. Action messages are a response to a particular environment input. With the DLCS, we allow an agent to transmit a subset of these selected actions over the network and receive such transmissions from other agents, thereby allowing agents to "talk" to one another. We use the term "subset" because, in general, only actions with larger bids are transmitted and received. We discuss the semantics of action message transmission in Section 4.4, but the implication of sending and receiving only large-bid actions is that only "good" pieces of information are communicated between agents. Note that action messages are broadcast to all agents on the network.

The purpose of action message passing is to provide a means for one agent to notify the others of what it is doing. In multiple-agent scenarios where agents must coordinate their actions, such message passing is critical. Take for example, the complicated task of moving a large object. Even for humans, communication is necessary to accomplish the job, *e.g.* "You take that end of the couch, and I'll take this end." If several agents are trying to lift an object, each must communicate its progress so that coordination can be achieved. As a biological analogy, army ant insect species *Eciton* and *Dorylus* use pheromones and physical contact on their predatory swarm raids as methods

of communication [9, 20]. In like manner, DLCS agents communicate over the network with action messages.

We describe an action message a_M as:

$$a_M = \{a, L, I\} \tag{4.1}$$

where a is the action word, L is the supplier (the classifier from which the action originated), I is the ID of the agent transmitting the message, and the 'M' subscript denotes an action message instead of just an action word. The supplier and ID are needed for BBA payoff messages, as will be discussed in Section 4.3.3.

Since action messages contain both a supplier and an agent ID, it is necessary to make an addition to the message board to include an ID field for each message, as shown in Figure 4.3. Now messages on the message board are associated with both a posting classifier and the agent from which the classifier originated.

1	message	supplier	agent ID
2	message	supplier	agent ID
⋮	⋮	⋮	⋮
q	message	supplier	agent ID

Figure 4.3. The DLCS message board.

4.3.2 Classifier Messages

Classifiers in the LCS are the rules upon which decisions are based. They are the "brain" of the classifier system. Classifier messages are the second message type

communicated between DLCS agents. The purpose of sharing rules between agents is based on the assumption that agents are solving a common task and can benefit from sharing useful rules. Since agents tend to learn behaviors at different rates, one agent will likely learn part of a solution to the environment task before its siblings. By sharing this learned information with the other agents, all agents benefit from each others' successes and the group, on the whole, tends to learn more rapidly. The primary constraint put on classifier passing, as will be discussed in Section 4.4, is that only "good" rules are sent, *i.e.* rules with large strengths. Clearly this constraint helps ensure that agents only share the classifiers that have proven useful to them individually.

A classifier message is functionally equivalent to the definition given in Equation 2.4. The message contains the one or more condition words, an action word, and the strength of the classifier. Receiving agents assume that a received rule with a high strength is a useful rule. Should such a rule turn out to be inappropriate for a particular agent, the environment payoff function will penalize the rule sufficiently until it is replaced. If the rule is irrelevant to the agent's situation, it will never become eligible.

One can view classifier message passing as another facet of rule discovery. In the standard LCS, rule discovery is accomplished by the genetic algorithm and the rule discovery operators (CDO, CEO, and TCO). However, in the DLCS, we add classifier passing, which introduces new rules into an agent's classifier list from another agent. Also, since the sending agent has already tested the rule to some extent, rules received via classifier-passing are more likely to be useful than those created by genetics or the discovery operators.

We should indicate at this point that it is not necessary to use both action message passing and classifier message passing simultaneously in a multiple-agent scenario. For example, if individual agents are assigned to smaller parts of a larger task, and these parts are unrelated, agents may not find each other's evolved rules very useful. On the other hand, if multiple-agents are assigned to solve the same identical task simultaneously, there may be little value in their communicating anything but learned rules. Other situations

may dictate a combination of task division and cooperation, thereby requiring both action and classifier passing. As will be evident in Section 4.4, the DLCS has been designed so that these two message passing schemes are independent of one another in operation, and one has much flexibility in applying the DLCS architecture to a machine learning task.

4.3.3 BBA Payoff Messages

The final type of message that can be sent between agents is a bucket brigade algorithm payoff message. The bucket brigade algorithm is designed to encourage chains of rules to form by paying the suppliers of the classifier actions selected to post. Since actions can also be sent over the network under the DLCS paradigm, it is necessary to pay suppliers across the network as well. The agent ID and supplier sent with an action message are used by the receiving agent to determine to which agent and supplier the payoff message should be sent. We describe a BBA payoff message BBA_M as:

$$BBA_M = \{I, L, R\} \tag{4.2}$$

where I is the ID of the agent transmitting the action to be paid, L is the supplier (the classifier from which the action originated), R is the payoff amount.

Unlike action and classifier passing, BBA payoff messages are sent directly to the agent for which the payoff is intended, rather than being broadcast. This exception does not violate our goal to have anonymous agents because BBA payoff is a process internal to the architecture of the classifier system. In other words, agents do not "know" they are communicating with one another when they send a BBA payoff message. On the other hand, if an agent were to explicitly send an action or classifier message directly to another agent, the transmitting agent would have to consciously select the receiving. Such a decision violates the desire for anonymity and uniformity among agents. We could just as

easily broadcast the BBA payoff and let each receiving agent decide if the message is intended for it. We are simply reducing network traffic by transmitting to the intended agent.

4.4 DLCS Execution Cycle

In the following figure, we present the execution cycle for the distributed learning classifier system. The execution cycle is based on the standard LCS cycle as shown in Figure 2.2, with items in italics representing DLCS additions.

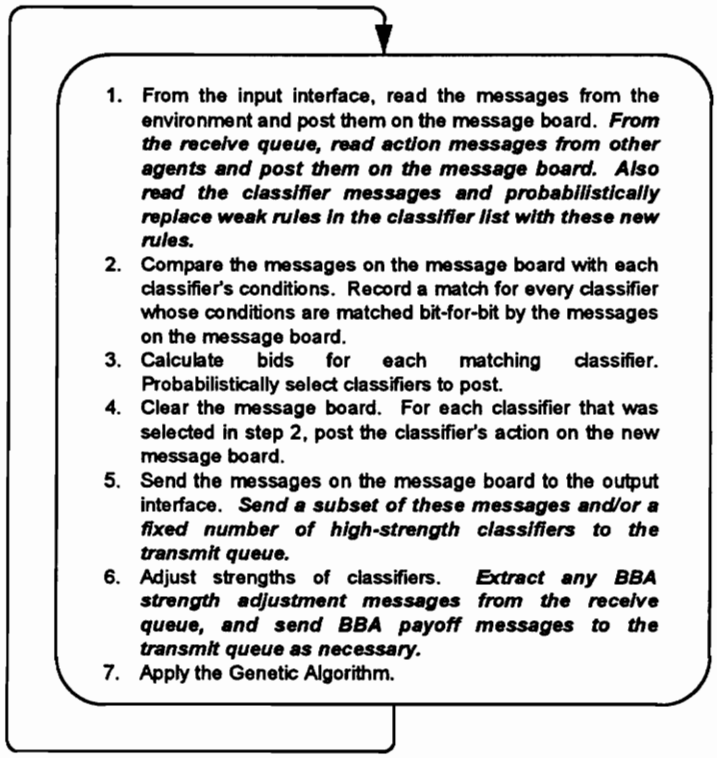


Figure 4.4. The DLCS execution cycle.

We will begin our discussion of this execution cycle at the end rather than the beginning so that we may describe network message transmission before message reception.

Action and classifier messages are sent to the transmit queue on step 5 of the cycle. For both cases, several DLCS parameters govern transmission. In the case of action message transmission, we state in the figure that a "subset" of the selected actions should be sent. This subset is defined by two quantities, the transmit bid threshold, B_{TX}, and the maximum number of actions to transmit, $N_{TX,action}$. The transmit bid threshold defines the minimum bid value required before an action is eligible to be sent over the network. Since an action's bid value is directly proportional to strength of the classifier that posted the action, the bid threshold has the effect of ensuring only those actions that derive from "good" rules can be sent. We specify B_{TX} as a percentage of the maximum bid on the message board at the current time, rather than as an absolute quantity, such that the minimum bid required for transmission $B_{TX,min}$ is given by:

$$B_{TX,\min}(t) = B_{TX} \cdot \max\left(B_j(t) : j = 1,q\right) \qquad (4.3)$$

where the $B_j(t)$'s are the bids of the q actions on the message board on step 5. Clearly the minimum bid requirement is time-varying since the maximum bid of the actions on the message board will also change with time. We use a percentage rather than an absolute quantity because it is difficult to quantify in absolute terms what a "good" bid is during DLCS execution. The bid equation is based on a number of quantities, most of which are tied to the credit assignment functions. Since we cannot predict ahead of time what the strength of a useful rule might be, we also cannot predict the bid of a useful action. Also, bids will have a tendency to be small at the start of a DLCS execution since the rules have yet to be influenced by the credit assignment functions, and we still want to allow action message passing during that time.

The $N_{TX,action}$ parameter provides a way to control the amount of network traffic by imposing an upper limit on the number of actions that can be sent, in case all of the actions have large bids. If $N_{TX,action}$ is smaller than the number of actions eligible to be transmitted,

then the actions with the largest bids are sent. This parameter cannot be any larger than the message board size.

In the case of classifier message transmission on step 5, two similar parameters exist, the transmit strength threshold, S_{TX}, and the maximum number of classifiers to transmit, $N_{TX,classifier}$. The transmit strength threshold functions like the transmit bid threshold in dictating a minimum strength required before a classifier can be transmitted and thus ensuring that only "good" rules are transmitted over the network. As with the bid threshold, the strength threshold is expressed as a percentage of the maximum strength in the classifier list, and the minimum strength required for transmission $S_{TX,min}$ is given by:

$$S_{TX,\min}(t) = S_{TX} \cdot \max\left(S_i(t) : i = 1,n\right) \qquad (4.4)$$

where the $S_i(t)$'s are the strengths of the n actions in the classifier list on step 5. Again these strengths are time-varying, so the minimum strength requirement will also be time-varying. $N_{TX,classifier}$ helps control the amount of network traffic even more so than $N_{TX,action}$ because most classifier lists tend to be rather large, and there are often several rules eligible for transmission.

As previously stated, action and classifier messages are *broadcast* over the network to all other agents. As will be shown when we discuss message reception, each agent has the opportunity to discard received messages. In addition, transmission of these two message types is globally paced by two time intervals: an action transmission interval, T_{action} and a classifier transmission interval $T_{classifier}$. These intervals determine how many clock tics occur between network transmissions. Larger intervals provide less inter-agent communication and therefore result in less coupling between agents. Smaller intervals provide more coupling.

Reception of action and classifier messages occurs on step 1 of the execution cycle. Since each agent broadcasts its action and classifier messages to all other agents in the network, there will usually be more messages in the receive queue than were

transmitted. Agents must therefore have a method of filtering out the best messages from all those received. As with transmission, action and classifier message reception are handled by separate parameters.

Action message reception is governed by two parameters that are effectively the converse of the transmission variables, the receive bid threshold, B_{RX}, and the maximum number of actions to receive, $N_{RX,action}$. These variables are separate from their corresponding transmit variables because we may wish to put more stringent requirements on reception than transmission, since reception will have a direct impact on an agent's behavior. The effects of message transmission are rather indirect, and an agent can therefore be more liberal in its sharing of information. On the other hand, reception of action messages from other agents will have direct impact on what rules are chosen from the classifier list, and an agent must therefore put a higher premium on the usefulness of received information. Therefore, one generally uses $B_{RX} > B_{TX}$ and $N_{RX,action} < N_{TX,action}$. As with transmission, B_{RX} is specified as a percentage of the maximum bid on the message board during step 1. The only messages posted on the message board at this time are messages from the environment and internal messages from the previous clock tick. Since environment messages have no bid, received actions messages are only "competing" with the bids of the internal messages. The B_{RX} quantity therefore ensures that received messages were generated from rules with similar strength values as those rules that posted the internal messages on the previous time step. The $N_{RX,action}$ parameter is usually set to the number of slots remaining on the message board beyond the environment and internal message postings. Alternatively, one could have the network messages overwrite the weaker internal messages. Note that those action messages from the receive queue which are not posted to the message board are discarded on step 1.

Classifier message reception is also controlled by two variables, the receive strength threshold, S_{RX}, and the maximum number of classifiers to receive, $N_{RX,classifier}$. These variables perform the same function as in action message reception. Again, we want the reception requirements to be more stringent because each rule that an agent

accepts will replace a weaker rule in the classifier list. The more network-based rules accepted, the more agent rules replaced. While rule replacement is not a necessarily a detrimental occurrence, we want to ensure that only "bad" rules are replaced with "good" rules. The DLCS ensures that the received rules are good by using the minimum strength requirement obtained from S_{RX}. A received classifier must meet this minimum strength requirement to be eligible for use. Note that if the rule is used in the agent's classifier list, the strength is also used. We limit the amount of bad rule replacement with $N_{RX,classifier}$. As with receive action messages, unused classifier messages are discarded. Note that there are no receive intervals for action or classifier reception since the frequency of reception is completely dependent on the frequency of transmission.

Finally on step 6, BBA payoff messages are processed. Any BBA payoff messages in the receive queue are removed and applied to the corresponding classifiers. For those classifier actions that were selected to post and have suppliers from other agents in the network, BBA payoff message are sent to the transmit queue. There are no restrictions on BBA payoff message transmission and reception, since this message passing is completely governed by the bucket brigade algorithm. As previously stated, BBA payoff messages are sent directly to the appropriate agent.

Before concluding this explanation of the DLCS, we take a moment to discuss strength and bid comparison. In classifier message passing, the strengths of received classifiers are compared with the strengths of the classifiers on the message list. In action message passing, the bids of the received actions are compared with the bids of the actions currently on the message board. We have assumed here that each DLCS agent has the same environment payoff function, and hence, strengths and bids are directly comparable. This assumption no longer holds if agents are used to perform completely different tasks, which is certainly one of the possible applications of the DLCS. In such a case, all strengths must be normalized before any message transmissions occur, so that relative comparisons can be made between rules and actions. Note that if strengths are normalized, the bids calculated from them will also be comparable. Of course, if agents

are given completely unrelated tasks that require a different payoff function, they implicitly lose their anonymity. Such a situation violates our desire to have identical, autonomous agents with no predefined leader. We would prefer to let agents subdivide the task themselves with each agent still influenced by the same environment payoff function We do not want to exclude the possibility of such an application, however, since the DLCS architecture will certainly support it. As a final note, agents that are assigned different tasks are not likely to benefit from classifier passing, since the context in which rules are used would vary from agent to agent.

4.5 Simulation Results

In order to illustrate the distributed learning classifier system, we now present two simulations, one involving classifier message passing and one involving action message passing. For these simulations, we have extended the single-agent animat problem to include multiple agents, with each agent having the task of maneuvering around obstacles and reaching a goal. This extension is a natural one because each agent is still subject to the limitations of the animat (see Section 3.2). In fact, in Souza and Talukdar's discussion on multiple agents groups, they refer to the very same agent limitations that Wilson does in his discussion of the single animat [22]. Each agent in a multiple-agent scenario still has a limited knowledge of the overall task to be solved and has limited learning and reasoning capabilities, just like the animat. Also, the multiple-agent animat problem models (in a very limited fashion) the biological word of social organisms. Organisms working together will have more success solving a task than organisms working alone.

Using the robot model described in Section 3.3.1, the major difference between the single and multiple agent animat problems is that each agent in the multiple-agent scenario treats the others as obstacles, since the limited sensing capabilities of the robot prohibit it from distinguishing between another robot and a fixed obstacle. The multiple-agent

animat problem is therefore more difficult than the single-agent animat problem because agents must now deal with moving obstacles, and hence, a more rapidly changing environment.

4.5.1 Classifier Passing Scenario

Our first simulation involves classifier passing. As previously stated, the purpose behind classifier passing is to allow agents performing the same task to benefit from sharing each other's rules. In order to illustrate this point, we provide two simulations which employ the standard animat playing field, except that three agents are present in the field instead of one, as shown in Figure 4.5. In our first simulation, agents do not communicate with one another but simply solve the animat problem individually, essentially operating with the standard LCS. In the second, agents employ classifier passing for communication using the DLCS framework. Our intention is to show that classifier passing is beneficial to the agents.

For both simulation cases, we use the standard environment payoff function values and again disable the genetic algorithm. One hundred simulations are run for each case, and we designate a simulation a success if all three agents reach the goal in under 2000 clock ticks. For the classifier passing parameters, we use $S_{TX} = S_{RX} = 0.8$, $N_{TX,classifier} = N_{RX,classifier} = 1$, and $T_{classifier} = 1$. These settings represent somewhat of an extreme case for DLCS operation, since each agent transmits a classifier over the network on every time step. This continuous transmission results in extremely frequent introduction of new rules into each agent's classifier list, thereby tightly coupling the three agents. As will be shown, this tight coupling can prove to be very helpful or very intrusive.

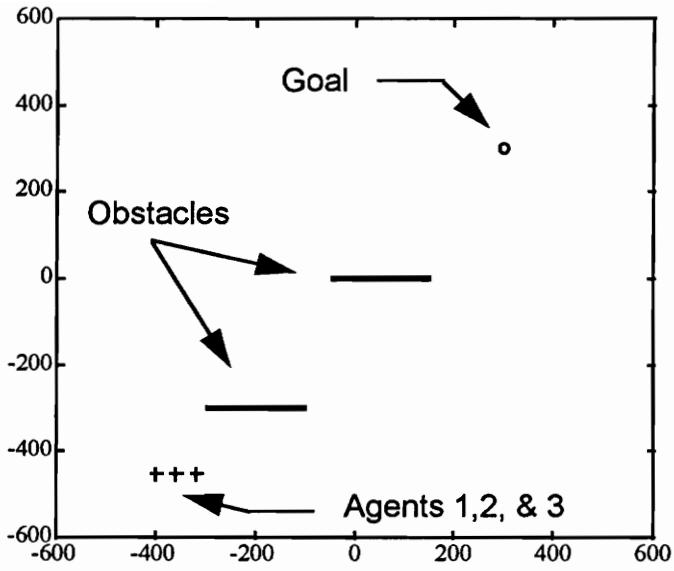


Figure 4.5. Multiple-agent animat playing field.

Table 4-1. Classifier passing simulation results.

	No Passing	Passing
Success Rate	16.0%	49.0%
Number of Clock Ticks	μ = 592.40 σ = 243.28	μ = 769.47 σ = 298.13
Path Area	μ = 6.91 σ = 3.42	μ = 8.13 σ = 3.33

The statistical results of the two simulations are given in Table 4-1, with sample agent paths for each given in Figure 4.6 and Figure 4.7. From the table, we see a much better success rate with classifier passing than without it. This results supports our conclusion that agents will be more successful in solving a task if they can share information. From the table, however, we also see that classifier passing has a larger mean number of clock ticks and path area. While the statistics for the no-passing case are somewhat inaccurate because of the extremely small percentage of success (statistics are

calculated for successful iterations only), the major reason for the higher number of clock ticks required by the classifier-passing case has to do with the frequency of classifier transmission. Since *each* DLCS agent transmits a classifier on *each* time step, the classifier list of each agent tends to converge to the first strong rule used in the agents, whether or not this rule will be useful throughout the remainder of the simulation. By convergence, we mean that every slot in the classifier list contains a copy of the convergent rule. Once convergence occurs, it is very difficult to replace this rule, since there are so many copies of it. To meet the demands of a situation not covered by the convergent rule, CDO periodically creates new rules. These rules are quickly replaced again by the convergent rule because this rule tends to have the highest strength in the classifier list and is therefore the rule that gets transmitted on each time step. In order for a rule to become convergent, it has to provide an action that almost always gets rewarded by the environment payoff function when it is selected. The rule does not, however, have to be extremely useful in a global sense, it merely has to get rewarded. This fact, combined with the need for CDO to compensate for situations the convergent rule does not handle, results in the increased number of clock ticks and path area statistics. However, as we shall demonstrate, DLCS agents using classifier passing take very similar paths, whether good or bad.

If the convergent rule is globally useful, as is the case in Figure 4.6, the agents take a relatively direct path to the goal. For this simulation, the convergent rule is #0## | 0001. This rule is useful in three ways. First, it allows an agent to move towards a goal on the agent's left, an action that will almost always get rewarded. Second, it allows an agent to move away from obstacles on the agent's right, an action that will avoid the punishment of a crash. Also, the agent does not have to have the goal in sight to move away from an obstacle. This benefit is important because agents must turn to get around the obstacles, and during this time, the goal tends not to be visible. Third, if the agent does not see anything, it continues to turn left until it does. This situation happens quite frequently when the agent is moving towards the goal in our playing field and then

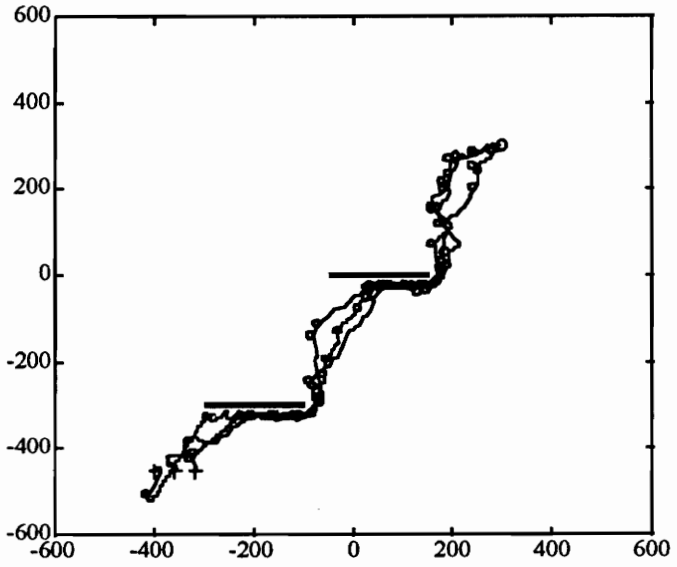
turns right. This third case has the effect of pointing the agent right back at the goal when such a situation occurs. On the other hand, this rule also presents the possibility that an agent will move towards an obstacle if the obstacle is on the right, *i.e.* the situation 0001 | 0001. This action can lead to a crash, provided the agent is right next to the obstacle. Overall, however, the convergent rule is useful throughout the simulation.

An example of mediocre rule is shown in Figure 4.7. In this case, the agents converge to the rule 10## | 0001. While this rule is very similar to the previous one and appears to be useful, there are two critical differences between the two. First, this rule requires that in order for an agent to move away from an obstacle, it must also see the goal. Second, this rule does not provide for the case when an agent cannot see anything. These two differences are enough to make the rule useful in only a limited number of circumstances, thereby increasing the number of clock tics required for the agents to reach the goal and making the agents' paths far from optimal. The distance-time plots in the figures further illustrate the difference in effectiveness of these two convergent rules.

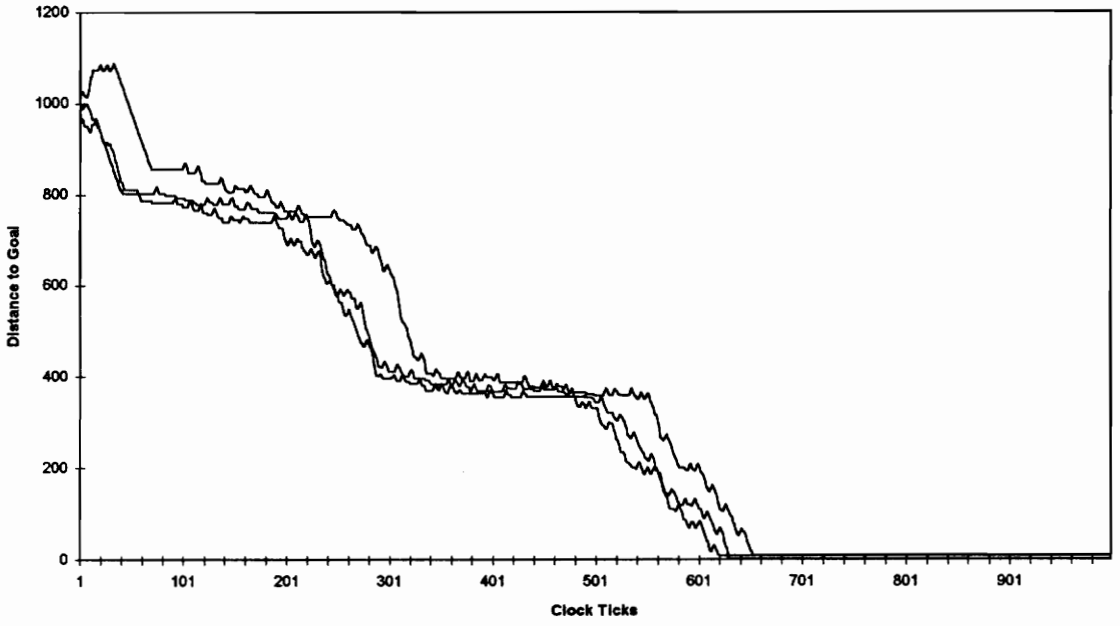
Clearly the convergent rule dictates the success of the agents. However, even if the agents take longer to reach the goal, they tend to take similar paths. This result is not surprising given the fact that the agents start in relatively the same area and begin sharing rules immediately. We should point out here that we have examined a case where the agents are very tightly coupled. Changing the classifier transmission interval would loosen the coupling between agents and likely modify their behavior. However, more research must be done to quantify the effects of agent coupling.

Figure 4.8 and Figure 4.9 show two solutions for the no-classifier-passing case. While the agents are successful in reaching the goal in Figure 4.8, they take radically different paths to get there. We expect such behavior, since there is no sharing of rules in this simulation. Figure 4.9 is even more striking in that agent 2 learns a very good solution on its own, but with classifier passing disabled, the agent cannot share this information. Agent 1 also reaches the goal, but with a less-than-optimal path, and agent 3

gets stuck in a loop and never comes close to the goal. These simulations provide a significant contrast to the classifier-passing case.

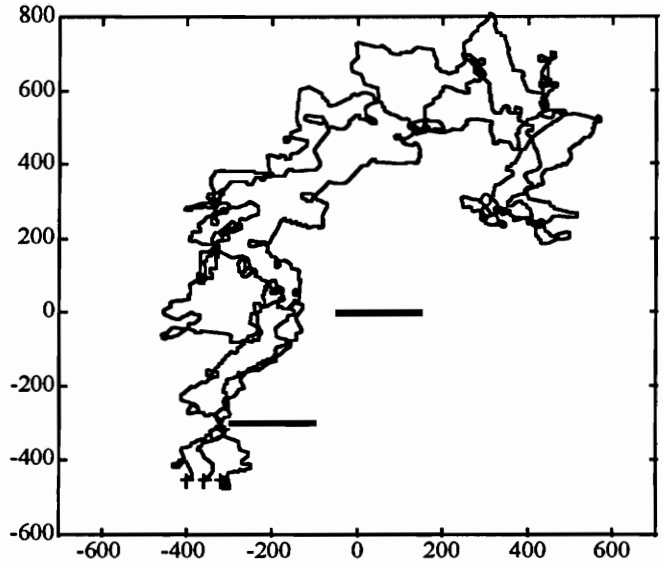


(a)

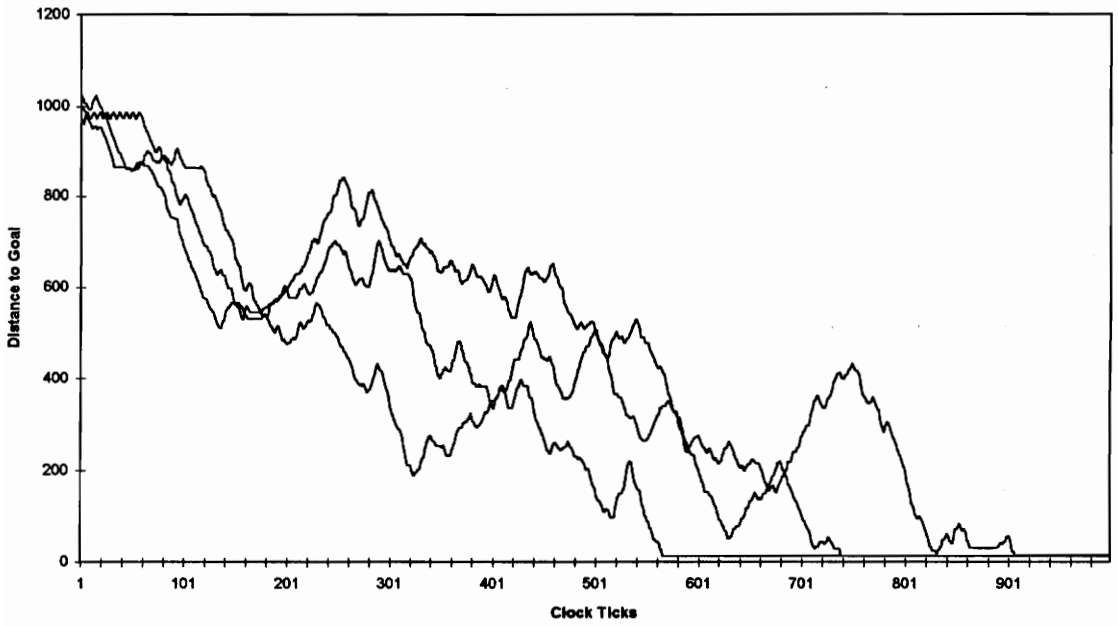


(b)

Figure 4.6. a) Agent paths for a good classifier passing solution. b) Distance-time curves.

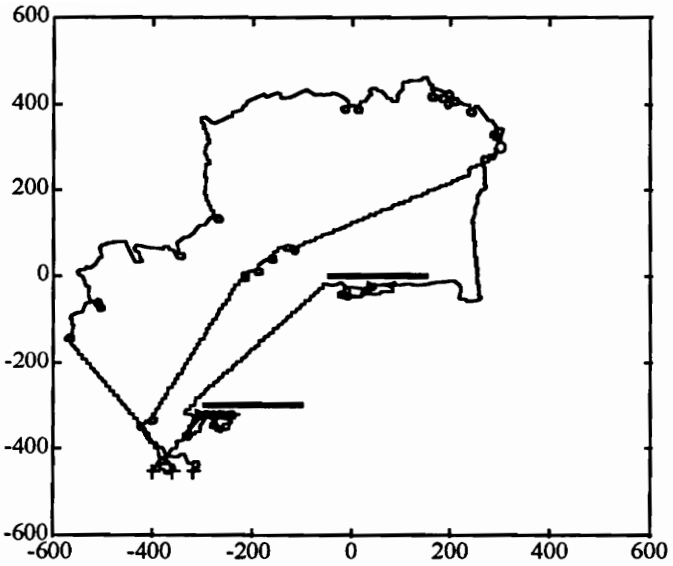


(a)

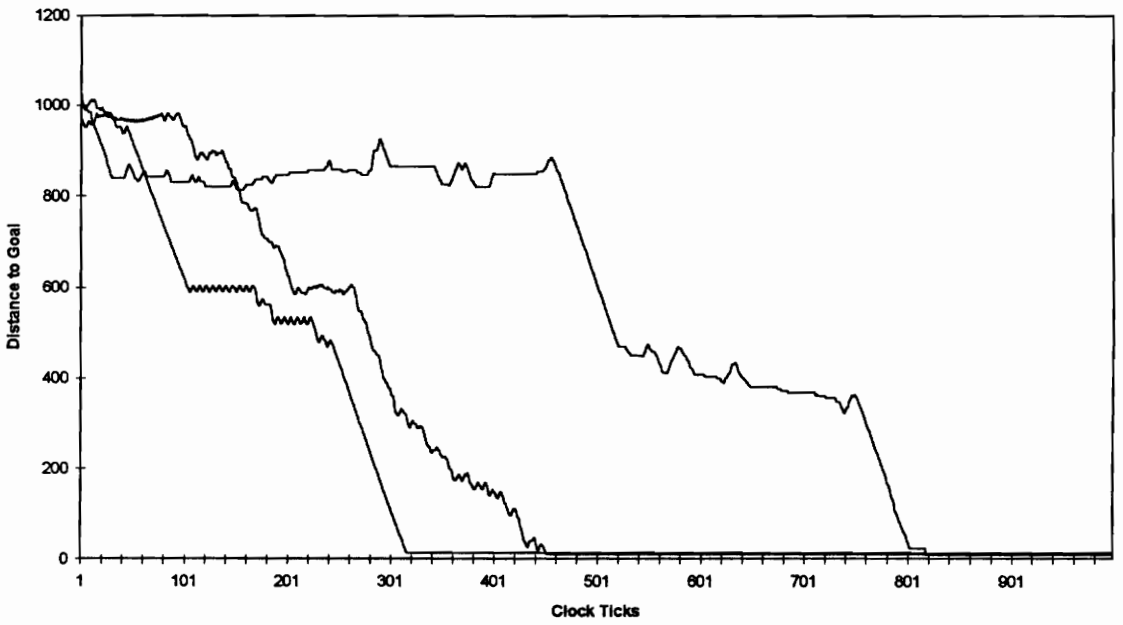


(b)

Figure 4.7. a) Agent paths for a poor classifier passing solution. B) Distance-time curves.

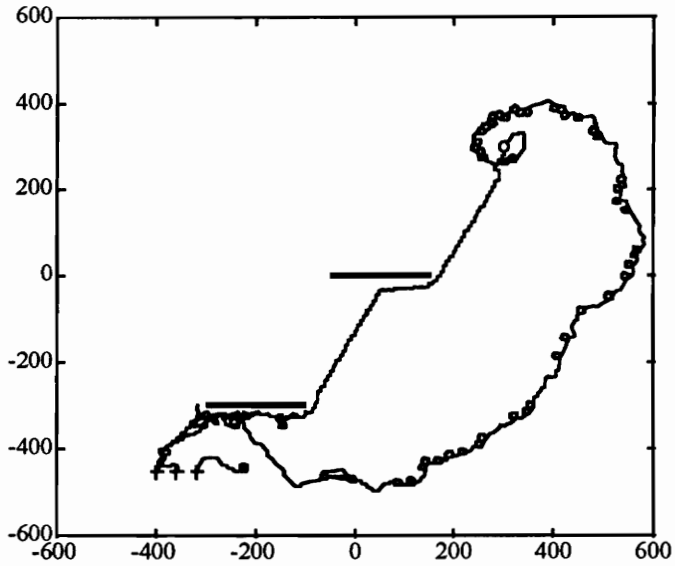


(a)

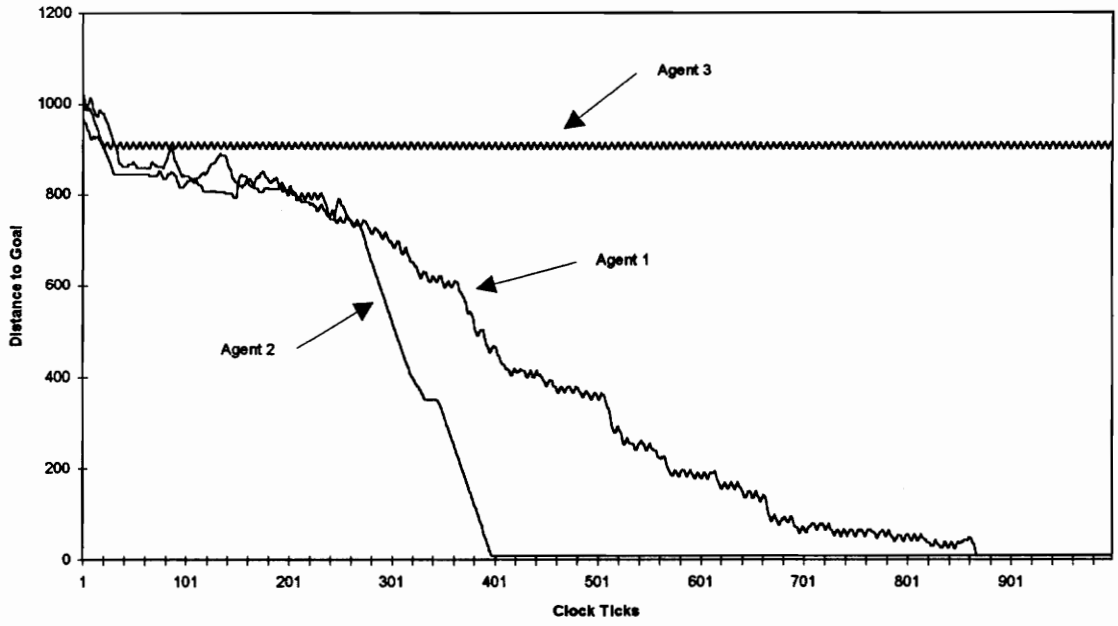


(b)

Figure 4.8. Successful no-classifier-passing simulation. a) Agent paths. b) Distance-time curves.



(a)



(b)

Figure 4.9. Unsuccessful no-classifier-passing simulation. a) Agent paths. b) Distance-time curves.

4.5.2 Action Passing Scenario

For our final simulation, we examine the action-passing feature of the DLCS. The purpose of action passing is to allow agents to "talk" to one another, thereby giving an agent the ability to tell the others what it is doing or when it has finished part of a task. Such communication is essential if agents are to coordinate their actions. Before presenting the simulation, we must review the input and output interfaces and message format.

As discussed in Section 3.3.2.1, an LCS message contains two parts, an environment string and an internal message string. On clock tick step 1, the input interface reads the status of the environment and copies the sensor information into the environment string of each message on the message board. When internal message passing is enabled, these message board messages are from the previous clock tick. However, if action passing is also enabled, the message board contains messages read from the receive queue on clock tick step 1, as well as internal messages. On clock tick step 1, we mask an environment substring over all messages on the message board, so messages compared to the classifier list will contain the state of the current agent's environment and either a message from the current agent's previous time step or a message from another agent. As discussed in Section 3.3.2.1, this message format is not mandatory. We could have separated messages into three categories, one for environment, one for bucket brigade, and one for action messages from other agents, with each message having a unique tag identifying its type. However, we wish to remain consistent with the single agent case, so we will not modify the current message format for this simulation.

The implications of this style of message, however, are that agents will coordinate actions with other agents based on the messages other agents sent to them *and* the current state of their environment. In this way, agents can be more selective in responding to another agent's message. Take, for example, the message chain presented in the BBA

simulation of Section 3.4.3. The chain forms during execution because the sensor information *and* the internal message information simultaneously match a condition word for each step of execution. If even one allele were changed in any of the rules, the chain could not completely link together because either the sensor information or the internal message did not exactly match the next rule in the chain. While at first this requirement appears restrictive, it actually ensures that a good chain will execute a very precise set of successful actions. This coordination is *required* if the animat is to exit the concave obstacle. By the same token, an agent must take into consideration its current environment state as well as the information presented to it by other agents when selecting an action.

Regardless of the style of message, however, the effect of action passing is to chain rules together between agents, hence the need for the BBA payoff message type in the DLCS architecture. Conceptually, this rule chaining between agents parallels chaining between internal rules, and the dynamics of the latter are not well understood. Therefore, in order to illustrate a potential application of action message passing, we will program a chain between agents for the simulation presented here rather than rely on the classifier system's learning process to generate one. The goal here is to demonstrate action passing and indicate a possible application in multi-agent coordination, not to discuss the learning characteristics of rule chaining.

The simulation we present involves coordination between two agents that must traverse a hallway one at a time. From a robotics point of view, such a scenario is not far from realistic, especially if there are several robots moving from one part of a building to another. Rather than have the robots move *en masse* down a corridor, we would like an orderly procession. Physical implications aside, however, the issue here is to establish coordinated behavior for this simple scenario. Therefore, we produce a third version of our animat playing field that includes a "hallway" created by two long, parallel obstacles, as shown in Figure 4.10. Two agents line up at one end of the hallway, as indicated by the +'s in the figure. At the other end is the goal, or simply, the agents' destination.

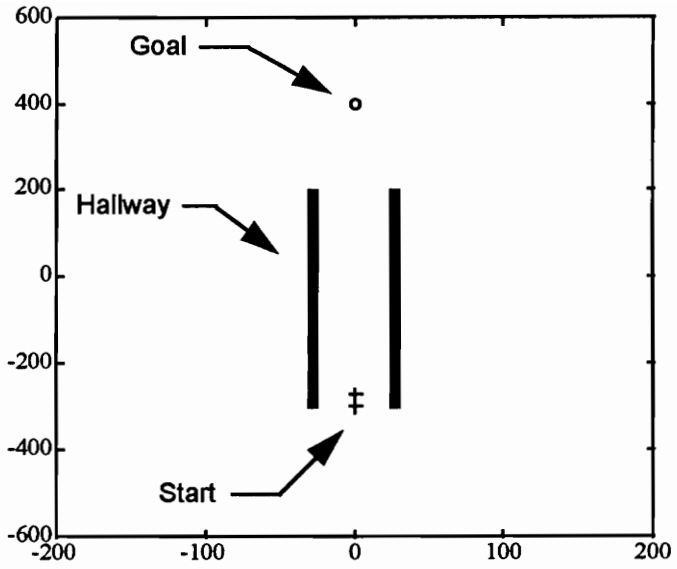


Figure 4.10. "Hallway" playing field for the action passing scenario.

The set of rules that implements the coordination we seek is given in Table 4-2. For this simulation, we enable both action passing, for inter-agent communication, and internal message passing, for intra-agent communication. We use action passing as the avenue for the first agent to tell the second agent when it has reached the end of the hallway. We use internal message passing as a way to store the "state" of the agent, specifically whether or not the agent is stopped, moving down the hallway, or outside of the hallway. In the table, we have separated the environment and internal parts of the condition and action words with a comma to make the rules easier to read.

In interpreting these rules, we note that if an agent's left and right obstacle sensors are both indicating an obstacle, the agent is in the hallway. This sensor state can actually be interpreted in two ways: as obstacles to the left and right or as an obstacle in front of the agent. Such ambiguity is a limitation of the simple robot model we are using, but can be easily remedied by adding a third obstacle sensor that only senses obstacles directly in

front of the agent. However, this limitation does not affect our simulation, because the "continue down hallway" rule looks for the goal to be straight ahead, in addition to checking the obstacle sensors. When the agent exits the hallway, it will no longer sense the obstacles and will only see the goal.

Table 4-2. Rules for solving the hallway problem. a) Agent 1. b) Agent 2.

(a)

Condition	Action	Meaning of rule
1111,0000	0011,1000	Start down hallway
1111,1000	0011,1000	Continue down hallway
1100,1000	0011,0100	End of hallway, tell second agent and continue forward
1100,0100	0011,0100	Out of hallway, continue forward
####,00#0	0000,0100	Ignore messages from second agent

(b)

Condition	Action	Meaning of rule
1111,0100	0011,0010	Start down hallway when first agent signals
1111,0010	0011,0010	Continue down hallway
1100,0010	0011,0010	End of hallway, continue forward
####,#000	0000,0000	Ignore some messages from the first agent

As the simulation executes, the first agent immediately begins moving down the hallway towards the goal, while the second agent remains motionless. When the first agent reaches the end of the hallway, it sends a unique message over the network, signalling the second agent to begin moving down the hall. Since this is the only network message required for this example of agent coordination, each agent also has a "filter" classifier that provides a rule match for all other network messages. The filter classifier is provided simply as a bypass for the cover detector operator, so that CDO does not create a rule for action messages from the network that the agents must ignore. Such rule creation could lead to corruption of the "program" we have created. Each agent also has

a "start down hallway" rule that begins the agent's movement down the hall, *i.e.* starts the internal message loop which uses the "continue down hallway" rule. For the first agent, this rule is triggered at the very beginning of the simulation. For the second agent, this rule is triggered when this agent receives the message from the first agent that it has reached the end of the hall.

We have set the action message passing parameters as follows: $B_{TX} = B_{RX} = 0.8$, $N_{TX,action} = N_{RX,action} = 1$, and $T_{action} = 1$. Clearly, if only one action message transmission is needed, we have once again chosen extreme parameter settings for the DLCS. However, we want to illustrate that the agents are capable of selectively ignoring spurious action messages from the network.

We present the results of the simulation in Figure 4.12 and Figure 4.12. As intended, both agents follow a straight path through the hallway. Part (a) of Figure 4.12 shows a "snapshot" of the simulation at clock tick 27. As can be seen, the first agent is moving down the hallway while the second agent remains motionless. Then, when the first agent exits the hallway, the second agent begins moving. Part (b) shows a snapshot of clock tick 40, a few ticks after the second agent starts down the hall. Figure 4.12 shows the distance-time curves for both agents. From this plot, we see that the second agent starts moving when the first agent gets 200 units from the goal—the point at which the first agent exits the hallway. Both agents converge on the goal.

While this simulation is somewhat simplistic, it does illustrate the potential coordination that can be achieved using action message passing in the DLCS. However, for action passing to be truly useful, rule chains between agents must form under the learning processes of the classifier system and should employ the use of the DLCS's BBA payoff message passing. However, more research will have to be done in internal message passing and BBA dynamics before true learning with action passing can be accomplished.

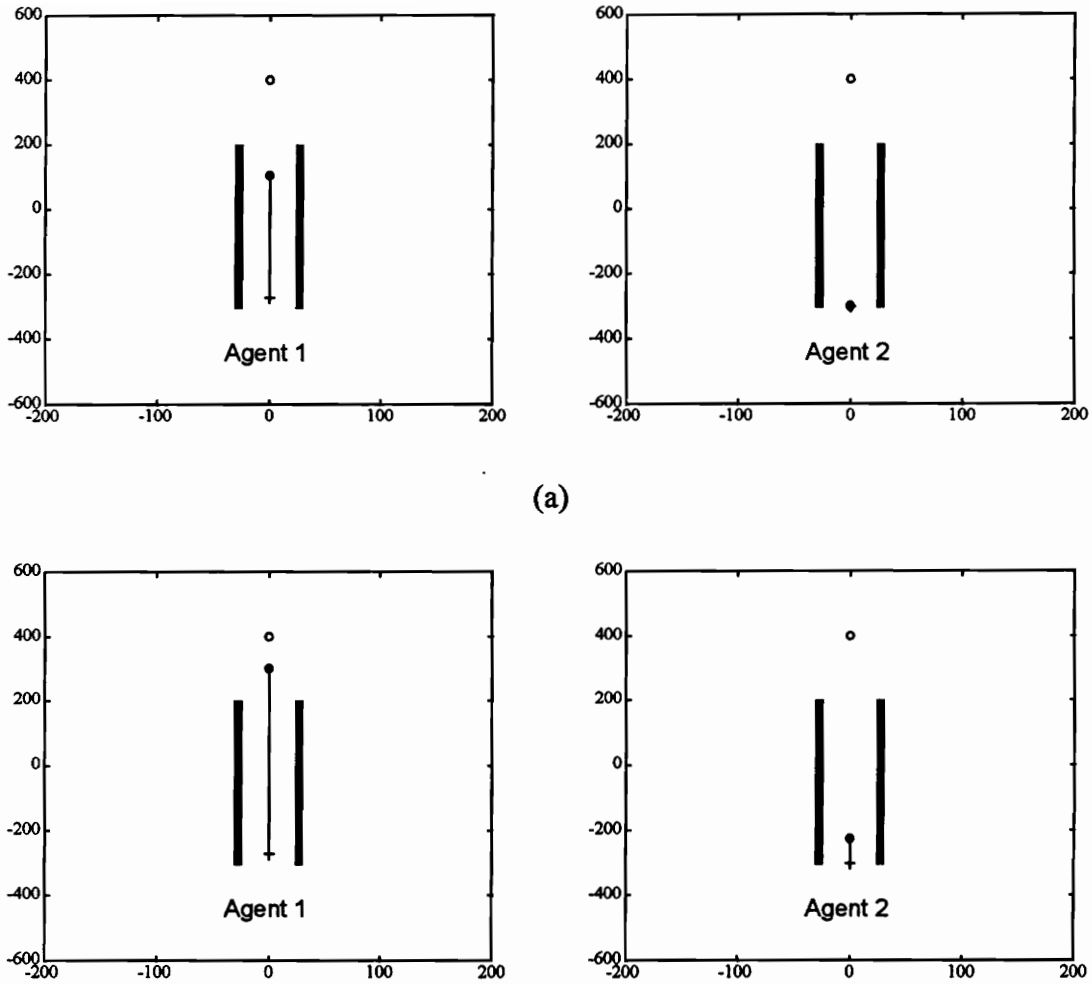


Figure 4.11. Results of hallway simulation. a) Paths at t=27. b) Paths at t = 40.

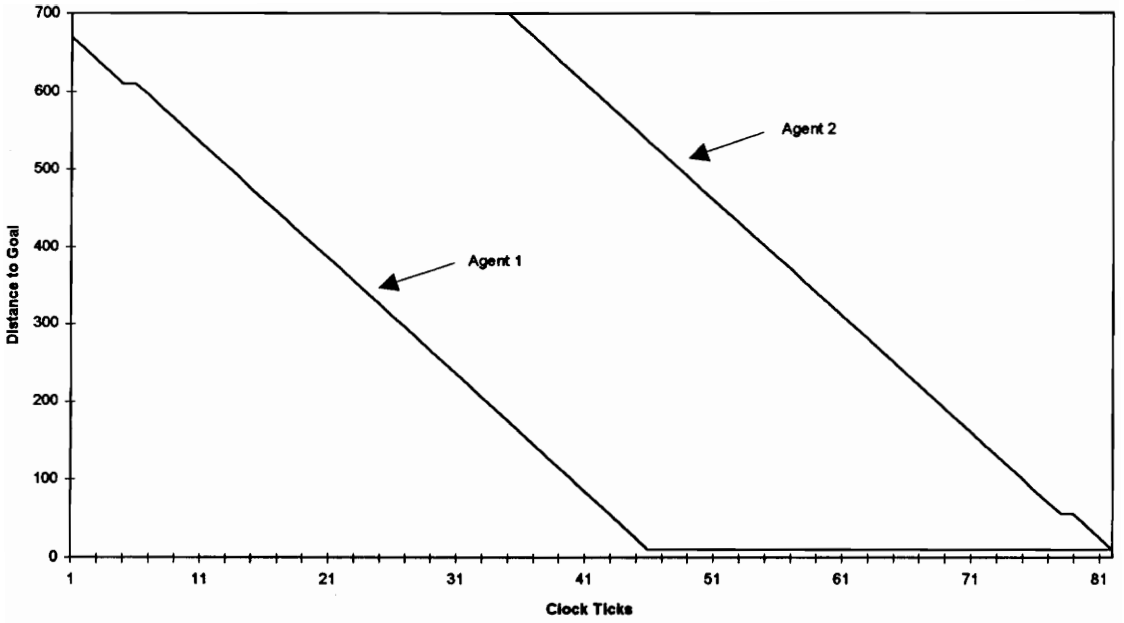


Figure 4.12. Distance-time curve for hallway simulation.

5. Conclusions and Suggestions for Further Research

In this thesis, we have presented the theoretical framework for a distributed form of the learning classifier system, as well as provided software code to implement this framework. The DLCS includes a network interface that allows agents to continue classifier system operation without being constrained by the network. The DLCS also provides three avenues of communication for agents. Through classifier message passing, agents can share learned knowledge with one another, thus facilitating group learning. Agents can also "talk" directly to one another using action message passing, thereby providing potential coordination and organization. Finally, with the use of BBA payoff message passing, agents are encouraged to interact with one another.

We have presented some simulation results for the LCS and DLCS as a means of conveying the benefits and challenges associated with the classifier system. To that end, we have illustrated environment payoff function optimization and bucket brigade algorithm operation. We have also compared our LCS results to similar research in the field. Finally, we have illustrated the potential of the DLCS for multiple-agent communication and coordination by providing examples of both classifier message passing and action message passing.

However, for every one research issue resolved, four new questions arise, and the classifier system is no exception. Probably the area most in need of further research is the bucket brigade algorithm. While we can devise programmed scenarios in which the BBA successfully reinforces a rule chain, chain emergence under learning conditions is something that has eluded us, as well as most other researchers. Before the BBA can become truly useful in a real-time learning context, further research needs to be done to either mathematically model its dynamics or to design an alternate chain encouragement mechanism based on a defined mathematical model. Since rule chaining takes the classifier

system beyond purely reactive behavior and into the realm of planning systems, such research is crucial. We feel that the treatment of the BBA in this thesis provides some groundwork in this direction.

The genetic algorithm is another area in need of further research. The GA is an essential part of the learning process in the classifier system, but is also plagued with much variability. We disabled the GA in our simulations because of this variability. Future research must focus on choosing genetic algorithm parameter settings that encourage evolution without destroying what has already been learned. Also, employing a "Pittsburgh approach" by using the GA to evolve substrings rather than individual alleles may be useful in some classifier system contexts, such as the animat problem presented in this thesis. Ultimately, a compromise between the rule discovery operators and the GA will be necessary to ensure useful rule adaptation and evolution.

Finally, we have only begun to scratch the surface of the potential applications and pitfalls associated with the DLCS. Issues such as network reliability, transmission delay, and agent coupling must be further explored. Also, by putting effort into understanding the BBA, we can come closer to effectively modeling action message passing, since this facet of the DLCS highly parallels the chaining that normally occurs internally in an LCS agent.

We see two potential futures of the distributed learning classifier system. In the near future, the DLCS can find application in simple, reactive control. Until rule chaining is better understood, this may be the fate of the classifier system in general, whether distributed or not. Reactive control is still a useful application, though, and initial tests on the DLCS have proven to be very effective in learning under such a context. The second future is, of course, planning and coordination between multiple agents using the DLCS architecture, and this future is the ultimate goal.

Appendix A

In this appendix, we present the software code listing for the classifierSystem class. There are seven *.cpp* and seven *.h* files. Since this software is very heavily documented, we will only briefly describe each below.

cfs.cpp

This module contains the *classifierSystem* class. All access to the LCS and DLCS should be through the functions included here. This class encapsulates the *messageBoard*, *classifierList*, *environmentInterface*, and *networkInterface* classes.

cfs.h

The header file for the *classifierSystem* class. All other header files are included by this one, so one need only include this file. Also, all classifier system parameter defaults are defined here.

msgboard.cpp

This module contains the *messageBoard* class.

msgboard.h

The header file for the *messageBoard* class.

clsslist.cpp

This module contains the *classifierList* class. This class contains not only the list of rules, but also the genetic algorithm, rule discovery operators, probabilistic rule selection functions, and the bucket brigade algorithm payoff functions. In general, all functions that in some way modify or access the classifier list are contained here.

clsslist.h

The header file for the *classifierList* class.

ei.cpp

This module contains the *environmentInterface* class. This class contains function stubs for the input and output interfaces and for the environment payoff function. The user must program these functions stubs to interface to a desired environment. This class is mostly empty and is provided for expansion by the user.

ei.h

The header file for the *environmentInterface* class. Specifications for the function stubs are given here.

ni.cpp

This module contains the *networkInterface* class. This class encapsulates the transmit and receive queues and is responsible for maintaining network transactions.

ni.h

The header file for the *networkInterface* class.

message.cpp

This module contains the *message* class. All messages and words used in the classifier system are instantiations of this class.

message.h

The header file for the *message* class.

queue.cpp

This module contains the *queue* class. The transmit and receive queues in the network interface are instantiations of this class.

queue.h

The header file for the *queue* class.

Before giving the code listing, we highlight a few issues contained in the code documentation. First, the code has both LCS and DLCS versions. If the user is designing an application for a single LCS, he or she can reduce the executable size by commenting out the `#define NETWORK` statement in the cfs.h module. Note that the LCS saving and loading functions are *different* for these two compiled versions. Alternatively, the user can compile with the network, and simply disable it with the `networkEnabled` parameter in the *classifierSystem* class.

Second, all rules start out as strings of zeros. The user must initialize each classifier with the `setClassifier()` function. This function provides both random and custom initialization. Also, to provide maximum message format flexibility, we have included message *masking*. A message mask tells the classifier system if the user has reserved any bit positions for tags. This information ensures that operations like mutation and random rule initialization will not change an allele that can only be a 0 or 1 to a #. See the clsslist.cpp module for more details.

Third, the classifierList class has a `supplierSet` array that provides unique integers which identify the rule that supplied the current classifier. A negative number in this array indicates that the rule has no suppliers. Therefore, when posting messages that are strictly from the environment, the user should be sure to post a -1 for the supplier. See the clsslist.cpp and msgboard.cpp modules for more details.

Fourth, when using DLCS code, the agent's ID must be initialized with a number greater than zero. Zero is used by the code to indicate a network broadcast. The

`agentID` field in the *classifierSystem* class is an unsigned 32 bit integer so that IP addressing can be used if desired. Note also that the network interface is read *before* the input interface in step one of the DLCS execution cycle so that the user can modify or discard messages from the network if desired. Conversely, the message board is posted to the output interface *before* being posted to the network interface on step 5 so that the user can remove messages not intended for network transmission. See the cfs.cpp and ei.cpp modules for more details.

Fifth, the reproduction function is set up to only introduce one child into the classifier list after a reproduction operator. This code is easily modified should the user wish to introduce both children into the classifier list.

Sixth, when porting this code to a UNIX-based computer, there are a few items to keep in mind. First, UNIX and DOS have different formats for data storage, so any binary information saved on one platform with the LCS saving and loading functions will *not* be readable on the other platform. Second, at the time this document was written, C++ exception handling was not supported in UNIX C++ compilers. Since the Borland C++ defaults to throwing an `xalloc` exeception when a memory allocation error occurs, all allocations check for this exception. For UNIX code, the user should change all allocation checks to look for a zero pointer. Finally, the standard integer size on most UNIX workstations is 4 bytes long (a long integer) rather than 2 bytes, as in DOS. This fact affects the pseudo-random number generator because the maximum random number that can be generated is twice as large on UNIX machines. Therefore, the same random number seeds will *not* produce the same simulation results for the two different platforms because the unqiue sequence of numbers from the generator is different.

The code listing begins on the next page.

Module cfs.h

```
#ifndef CFS_H
#define CFS_H

//=============================================================================
// The classifier system class contains all the necessary classes and
// functions to implement a learning classifier system.  Every instantiation
// of the classifierSystem object is a separate LCS.  The classifier system
// has several common parameters which are shared over multiple
// instantiations.  These are the static memeber variables, whose defaults
// are listed below.  To use the classifierSystem object, a user must
// program the function stubs contained in the ei module (environment
// interface) as well as build a front-end to repeatedly call the
// classifierSystem::clockTick() function.  See ei.h for more details.
// Note that all code in the modules for this software contains code for both
// the standard LCS and the distributed LCS.  Any portion of code surrounded
// by the
//    #ifdef NETWORK
//      ...
//    #endif
// statements is code specifically for DLCS operation.  To completely remove
// this code from a compile, comment out the #define NETWORK line.
// You can also compile the code for network use and just disable the network
// by setting the 'networkEnabled' parameter to 0, as well.  I have provided
// the define in order to explicitly differentiate between DLCS and LCS code
// and to provide a way to reduce code size if you don't play on distributing
// the LCS.
//
// NOTE:  The disk-saved version for code compiled with the NETWORK define
//        is DIFFERENT than code compiled without it.  You can't load one
//        from the other.  (This isn't too hard to change if you really have
//        a problem with it.  I just didn't see a reason to do it.)
//=============================================================================

#define NETWORK

//-----------------------------------------------------------------------------
// Standard C libary includes.
//-----------------------------------------------------------------------------
#include<math.h>
#include<stdlib.h>
#include<stdio.h>
#include<string.h>

//-----------------------------------------------------------------------------
// Since the new and delete operators are used for all memory allocations,
// it is necessary to use try-catch exception handling to check for
// allocation problems.  The default operation of the new operator in
// Borland C++ 4.0 is to throw the 'xalloc' exception when a new operation
// fails.  Therefore, a catch(xalloc){} block is included after all new
// operations.
//-----------------------------------------------------------------------------
#include<except.h>

//-----------------------------------------------------------------------------
// Interface to the pseudo-random number generator.  This inlined function
// must be defined BEFORE including the classifier system components.
//-----------------------------------------------------------------------------
extern long numRandCalls;
inline int getRandom(int __num)
{ ++numRandCalls;  return (int)(((long)rand()*__num)/(RAND_MAX+1)); }

//-----------------------------------------------------------------------------
// Swap functions are used in sorting.
//-----------------------------------------------------------------------------
inline void swap(int & val1, int & val2)
```

```
{ int temp = val1;  val1 = val2;  val2 = temp; }
inline void swap(double & val1, double & val2)
{ double temp = val1;  val1 = val2;  val2 = temp; }
inline void swap(unsigned long & val1, unsigned long & val2)
{ unsigned long temp = val1;  val1 = val2;  val2 = temp; }

//-----------------------------------------------------------------------------
// Classifier system components.
//-----------------------------------------------------------------------------
#ifdef NETWORK
class networkInterface;
#endif

#include"message.h"
#include"msgboard.h"
#include"classlist.h"
#include"ei.h"

#ifdef NETWORK
  #include"queue.h"
  #include"ni.h"
#endif

//-----------------------------------------------------------------------------
// Default classifier system settings.  These settings are used upon
// initialization and when restoreDefaults() is called in the
// classifierSystem class.  Some of the parameters are not supported yet
// but should be initialized to the values shown to provide future
// compatibility.  If you have some favorite settings, put them here.
//-----------------------------------------------------------------------------

#define DEF_MSG_LEN          9
#define DEF_SEED             100
#define DEF_MSGBOARD_LEN     1
#define DEF_CLASSLIST_LEN    32
#define DEF_NUM_COND         1            // NOT USED, set to 1
#define DEF_BBA_ENABLED      0
#define DEF_BBA_STYLE        0            // NOT USED, set to 0
#define DEF_ELITISM_ENABLED  0            // NOT USED, set to 0
#define DEF_MAX_ACTIONS_TO_POST 1
#define DEF_CDO_ENABLED      1
#define DEF_CEO_ENABLED      1
#define DEF_TCO_ENABLED      0            // NOT USED, set to 0
#define DEF_BID_CONST        0.125
#define DEF_SEXUAL_PROB      0.7
#define DEF_MUTATION_PROB    0.1
#define DEF_START_STREN      0.0
#define DEF_HEAD_TAX         0.0
#define DEF_BID_TAX          0.0          // NOT USED, set to 0.0
#define DEF_PRODUCER_TAX     0.0          // NOT USED, set to 0.0
#define DEF_PROD_TAX_INTERVAL 1           // NOT USED, set to 1
#define DEF_ELITE_THRESH     0.0          // NOT USED, set to 0.0
#define DEF_STRENGTH_CAP     100.0
#define DEF_BID_CAP          100.0
#define DEF_GENETIC_INTERVAL 4000
#define DEF_MAX_ITERATIONS   4000

#ifdef NETWORK
  #define BROADCAST_ID               0
  #define DEF_NET_STATE              1          // 1 == on, 0 == off

  #define DEF_CLASS_PASSING_STATE    1
  #define DEF_CLASS_TX_INTERVAL      1
  #define DEF_MAX_CLASS_TO_TX        1
  #define DEF_MAX_CLASS_TO_RX        1
  #define DEF_TX_STREN_THRESH        0.8
  #define DEF_RX_STREN_THRESH        0.8

  #define DEF_ACTION_PASSING_STATE   0
  #define DEF_ACTION_TX_INTERVAL     1
```

```
  #define DEF_MAX_ACTIONS_TO_TX       1
  #define DEF_MAX_ACTIONS_TO_RX       1
  #define DEF_TX_BID_THRESH           0.8
  #define DEF_RX_BID_THRESH           0.8
#endif

class classifierSystem
{
   //--------------------------------------------------------------------------
   // Settings
   //--------------------------------------------------------------------------
   static int geneticInterval;     // The number of time steps between genetics
   static int maxCount;            // The maximum number of iteration allowed
   static unsigned seed;           // The pseudo-random number generator seed
#ifdef NETWORK
   static int networkEnabled;      // Network on flag
#endif

   //--------------------------------------------------------------------------
   // Classifier system components
   //--------------------------------------------------------------------------
   messageBoard msgBoard;          // Message board
   classifierList classList;       // Classifier List
   environmentInterface environ;   // Environment
#ifdef NETWORK
   networkInterface network;       // Network
#endif

   //--------------------------------------------------------------------------
   // Parameters
   //--------------------------------------------------------------------------
   int clockCount;                 // Iteration counter
#ifdef NETWORK
   unsigned long agentID;          // Network identification
#endif

 public:

   //--------------------------------------------------------------------------//
   // Initialization & reset functions
   //--------------------------------------------------------------------------//
#ifdef NETWORK
   classifierSystem() : agentID(0) { reset(1); }
#else
   classifierSystem() { reset(1); }
#endif
   static void restoreDefaults();
   void reset(int firstReset = 0);
   void softReset();
   void initClassifier(int num, char *condStr=0, char *msgStr=0, double stren=0.0)
   { classList.addClassifier(num,condStr,msgStr,stren); }

   //--------------------------------------------------------------------------//
   // Classifier system access functions
   //--------------------------------------------------------------------------//
   int clockTick();
   int readClock() { return clockCount; }
   static int isBBAon()
   { return classifierList::isBBAon(); }
   static int getSeed() { return seed; }
   int getMsgBoardCurrentSize() { return msgBoard.actualSize(); }
   void readStats(int *cross, int *sexual, int *mutations,
                  int *CDOs, int *CEOs)
   { classList.readStats(cross,sexual,mutations,CDOs,CEOs); }
#ifdef NETWORK
   void getMsgBoardPosting(int num, message & msg, int * supplier,
                           unsigned long *ID, double *ruleBid)
   { msgBoard.getMessage(num,msg,supplier,ID,ruleBid); }
   void getClassifier(int num, message & cond, message & act,
                      double *stren, double *spec, double *sup, double *ruleBid,
```

```
                        int *match, int *sel, int *supplier, unsigned long *ID,
                        int *elite, int *ruleNum)
    {
     classList.getClassifier(num,cond,act,stren,spec,sup,ruleBid,match,sel,
                             supplier,ID,elite,ruleNum);
    }
    static int isNetworkOn() { return networkEnabled; }
    void setAgentID(unsigned long newID)
    { agentID = newID;
      classList.setAgentID(newID);
      network.setAgentID(newID);
      environ.setAgentID(newID);
    }
    unsigned long getAgentID() { return agentID; }
    int removeFromTxQueue(queueList & qMsg)
    { return network.removeFromTxQueue(qMsg); }
    void addToRxQueue(const queueList & qMsg)
    { network.addToRxQueue(qMsg); }
#else
    void getMsgBoardPosting(int num, message & msg, int * supplier,
                            double *ruleBid)
    { msgBoard.getMessage(num,msg,supplier,ruleBid); }
    void getClassifier(int num, message & cond, message & act,
                       double *stren, double *spec, double *sup, double *ruleBid,
                       int *match, int *sel, int *supplier, int *elite,
                       int *ruleNum)
    {
     classList.getClassifier(num,cond,act,stren,spec,sup,ruleBid,match,sel,
                             supplier,elite,ruleNum);
    }
#endif

    //-------------------------------------------------------------------------//
    // Saving and loading functions
    //-------------------------------------------------------------------------//
    static void saveStatic(FILE *fptr);
    static void loadStatic(FILE *fptr);
    void save(FILE *fptr);
    void load(FILE *fptr);
    void saveRules(FILE *fptr) { classList.saveRules(fptr); }
    void loadRules(FILE *fptr) { classList.loadRules(fptr); }

    //-------------------------------------------------------------------------//
    // Static parameter access functions
    //-------------------------------------------------------------------------//
    static void setMasks(char *cMask, char *aMask)
    { classifierList::setMasks(cMask,aMask); }
#ifdef NETWORK
    static void getSettings(unsigned *sd, int *genInterval, int *max, int *netOn)
    { *sd = seed; *genInterval = geneticInterval; *max = maxCount;
      *netOn = networkEnabled; }
    static void setSettings(unsigned sd, int genInterval, int max, int netOn)
    { seed = sd; geneticInterval = genInterval; maxCount = max;
      networkEnabled = netOn; }
    static void getSettings(unsigned *sd, int *genInterval, int *max, int *netOn,
                            int *msgBoardLength,
                            int *classListLen, int *numCond, int *BBAon,
                            int *BBAtype, int *elitismOn, int *prodTaxInt,
                            int *CDOon, int *CEOon, int *TCOon, int *maxPost,
                            double *bConst, double *crossProb,
                            double *mutateProb, double *startStren,
                            double *headTaxRate, double *bidTaxRate,
                            double *prodTaxRate, double *eliteT,
                            double *strenCap, double *bCap,
                            int *actPass, int *classPass, int *classTxInt,
                            int *maxClassTx, int *maxClassRx, int *actTxInt,
                            int *maxActTx, int *maxActRx, double *TxStren,
                            double *RxStren, double *TxBid, double *RxBid)
    { *sd = seed; *genInterval = geneticInterval; *max = maxCount;
      *netOn = networkEnabled;
```

```
    messageBoard::getSettings(msgBoardLength);
    classifierList::getSettings(classListLen,numCond,BBAon,BBAtype,elitismOn,
                                prodTaxInt,CDOon,CEOon,TCOon,maxPost,bConst,
                                crossProb,mutateProb,startStren,headTaxRate,
                                bidTaxRate,prodTaxRate,eliteT,strenCap,bCap);
    networkInterface::getSettings(actPass,classPass,classTxInt,maxClassTx,
                          maxClassRx, actTxInt,maxActTx,maxActRx,TxStren,
                          RxStren, TxBid, RxBid);
  }

  static void setSettings(unsigned sd, int genInterval, int max, int netOn,
                          int msgBoardLength,
                          int classListLen, int numCond, int BBAon,
                          int BBAtype, int elitismOn, int prodTaxInt,
                          int CDOon, int CEOon, int TCOon, int maxPost,
                          double bConst, double crossProb,
                          double mutateProb, double startStren,
                          double headTaxRate, double bidTaxRate,
                          double prodTaxRate, double eliteT,
                          double strenCap, double bCap,
                          int actPass, int classPass, int classTxInt,
                          int maxClassTx, int maxClassRx, int actTxInt,
                          int maxActTx, int maxActRx, double TxStren,
                          double RxStren, double TxBid, double RxBid)
  { seed = sd; geneticInterval = genInterval;  maxCount = max;
    networkEnabled = netOn;
    messageBoard::setSettings(msgBoardLength);
    classifierList::setSettings(classListLen,numCond,BBAon,BBAtype,elitismOn,
                                prodTaxInt,CDOon,CEOon,TCOon,maxPost,bConst,
                                crossProb,mutateProb,startStren,headTaxRate,
                                bidTaxRate,prodTaxRate,eliteT,strenCap,bCap);
    networkInterface::setSettings(actPass,classPass,classTxInt,maxClassTx,
                          maxClassRx, actTxInt,maxActTx,maxActRx,TxStren,
                          RxStren, TxBid,RxBid);
  }
#else
  static void getSettings(unsigned *sd, int *genInterval, int *max)
  { *sd = seed; *genInterval = geneticInterval; *max = maxCount; }
  static void setSettings(unsigned sd, int genInterval, int max)
  { seed = sd; geneticInterval = genInterval; maxCount = max; }
  static void getSettings(unsigned *sd, int *genInterval, int *max,
                          int *msgBoardLength,
                          int *classListLen, int *numCond, int *BBAon,
                          int *BBAtype, int *elitismOn, int *prodTaxInt,
                          int *CDOon, int *CEOon, int *TCOon, int *maxPost,
                          double *bConst, double *crossProb,
                          double *mutateProb, double *startStren,
                          double *headTaxRate, double *bidTaxRate,
                          double *prodTaxRate, double *eliteT,
                          double *strenCap, double *bCap)
  { *sd = seed; *genInterval = geneticInterval; *max = maxCount;
    messageBoard::getSettings(msgBoardLength);
    classifierList::getSettings(classListLen,numCond,BBAon,BBAtype,elitismOn,
                                prodTaxInt,CDOon,CEOon,TCOon,maxPost,bConst,
                                crossProb,mutateProb,startStren,headTaxRate,
                                bidTaxRate,prodTaxRate,eliteT,strenCap,bCap);
  }

  static void setSettings(unsigned sd, int genInterval, int max,
                          int msgBoardLength,
                          int classListLen, int numCond, int BBAon,
                          int BBAtype, int elitismOn, int prodTaxInt,
                          int CDOon, int CEOon, int TCOon, int maxPost,
                          double bConst, double crossProb,
                          double mutateProb, double startStren,
                          double headTaxRate, double bidTaxRate,
                          double prodTaxRate, double eliteT,
                          double strenCap, double bCap)
  { seed = sd; geneticInterval = genInterval;  maxCount = max;
    messageBoard::setSettings(msgBoardLength);
```

```
    classifierList::setSettings(classListLen,numCond,BBAon,BBAtype,elitismOn,
                                prodTaxInt,CDOon,CEOon,TCOon,maxPost,bConst,
                                crossProb,mutateProb,startStren,headTaxRate,
                                bidTaxRate,prodTaxRate,eliteT,strenCap,bCap);
  }
#endif

  // Animat-specific functions
  const message & getAgentSensorMsg()
  { return environ.getSensorMsg(); }
  void getAgentStats(int *numCrashes)
  { environ.getAgentStats(numCrashes); }
};

#endif
```

Module cfs.cpp

```
#include"cfs.h"

// Random number generator count.  Used to keep track of the number of
// getRandom() calls so that the pseudo-random number generator can be
// restored to its previous state upon loading.  See
// classifierSystem::load().
long numRandCalls;

// Static class memeber initialization
unsigned classifierSystem::seed = DEF_SEED;
int classifierSystem::geneticInterval = DEF_GENETIC_INTERVAL;
int classifierSystem::maxCount = DEF_MAX_ITERATIONS;
#ifdef NETWORK
int classifierSystem::networkEnabled = DEF_NET_STATE;
#endif

//============================================================================
// FUNCTION:     reset()
//
// DESCRIPTION:  This function is called to reset the classifier system
//               object.  A reset is done whenever a classifierSystem object
//               is instantiated.  Subsequent resets can be done to start a
//               simulation over, or to set up a new classifier system object
//               when system parameters are changed.
//
//               NOTE:  This reset is the ONLY reset that needs to be called
//                      to reset the classifier system object.  This function
//                      handles all calls to member objects.
//
// I/O:           -->  firstReset  -  (optional) This flag should be set only
//                                    on the first reset of the classifier
//                                    system.
//============================================================================

void classifierSystem::reset(int firstReset)
{
 srand(seed);
 numRandCalls = 0;
 clockCount = 0;
 msgBoard.reset(firstReset);
 classList.reset(firstReset);
 environ.reset(firstReset);
#ifdef NETWORK
 network.reset();
#endif
}
```

```
//=============================================================================
// FUNCTION:     softReset()
//
// DESCRIPTION:  This function is called to reinitialize the system counters
//               without disturbing the state of the system.  This function is
//               used to start a new iteration using the current rules.  All
//               statistics, rule numbers, iteration counters, etc. are reset.
//               Note that the random number generator is reset, too, since
//               we assume that the state of the classifier system will be
//               sufficiently different that the same string of random numbers
//               will not affect the results.
//
// I/O:          none
//=============================================================================

void classifierSystem::softReset()
{
 srand(seed);
 numRandCalls = 0;
 clockCount = 0;

 msgBoard.softReset();
 classList.softReset();
 environ.softReset();
#ifdef NETWORK
 network.softReset();
#endif
}

//=============================================================================
// FUNCTION:     restoreDefaults()
//
// DESCRIPTION:  This function is called to restore all defaults in the
//               classifier system and its member objects.  Defaults are
//               defined in the include file for this module.  You MUST
//               call reset after calling this function.
//
//               NOTE:  This is the only restoreDefaults() function that
//                      needs to be called to restore the defaults of the
//                      classifier system objects.  All member object
//                      default functions are called from this function.
//
// I/O:          none
//=============================================================================

void classifierSystem::restoreDefaults()
{
  seed = DEF_SEED;
  geneticInterval = DEF_GENETIC_INTERVAL;
  maxCount = DEF_MAX_ITERATIONS;
#ifdef NETWORK
  networkEnabled = DEF_NET_STATE;
#endif

  message::restoreDefaults();
  messageBoard::restoreDefaults();
  classifierList::restoreDefaults();
  environmentInterface::restoreDefaults();
#ifdef NETWORK
  networkInterface::restoreDefaults();
#endif
}
```

```
//============================================================================
// FUNCTION:     clockTick()
//
// DESCRIPTION:  This function is called to perform one iteration of the
//               classifier system.  This function should be called only
//               after the classifier system has been correctly setup.  The
//               basic execution cycle is:
//
//               1. Read messages from the input interface and the environment
//                  interface and post them on the message board.  Messages
//                  are read from the network first so that the user can
//                  overwrite parts of the message if substrings have been
//                  defined.
//               2. Check matches with message board
//               3. Calculate bids of matching classifiers
//               4. Select and post message(s)
//               5. Send messages to environment via the output interface and
//                  to network interface
//               6. Perform BBA payoff, environment payoff, and taxes.
//               7. Perform genetics
//
//               Step 7 is actually done at the beginning of the execution
//               cycle because the clock counter is also incremented there.
//               This function is called until the iteration is "finished."
//               An iteration is finished when the environment says so or when
//               'maxCount' is exceeded.
//
// I/O:          <--           1 == done, 0 == not done.
//============================================================================

int classifierSystem::clockTick()
{
  int done;

  // Increment the clock
  ++clockCount;

  // Perform genetic operators on classifier list
  if ( !(clockCount % geneticInterval) )
    classList.genetics();

  // Clear message board if the BBA is off (no internal messages when BBA off)
  if (!isBBAon()) msgBoard.actualSize(0);

  // Read network
#ifdef NETWORK
  network.readNetwork(msgBoard,classList);
#endif

  // Read environment.
  environ.readInput(msgBoard,classList);

  // Get matches with message board.
  classList.getMatches(msgBoard);

  // Calculate bids of matching classifiers
  classList.calcBids();

  // Select and post message(s)
  classList.fillMsgBoard(msgBoard);

  // Send messages to environment.  Message board can be altered here.
  done = environ.postOutput(msgBoard,classList);

  // Send messages to the network
#ifdef NETWORK
  network.postNetwork(clockCount,msgBoard,classList);
#endif

  // Call BBA payoff function (skipped if BBA is not on)
```

```
#ifdef NETWORK
  classList.adjustStrengths(network);
#else
  classList.adjustStrengths();
#endif

  // Call environment payoff function
  environ.adjustStrengths(msgBoard,classList);

  // Apply taxes to rules
  classList.taxes();

  // Make sure strengths are between 0 and classifierList::strengthCap
  classList.checkStrengths();

  // Check for maximum interations
  if (clockCount >= maxCount) done = 1;

  return done;
}

//=========================================================================
// FUNCTION:     saveStatic()
//
// DESCRIPTION:  This function is called to save the static members of the
//               classifier system.  This function MUST be called before
//               save().  Note that all saveStatic() functions of classifier
//               system members are called here.  In other words, to save
//               a classifier system object, just call saveStatic() and save()
//               in this order and these functions will ensure that all
//               embedded class information is saved.
//
// I/O:           -->  fptr  -  Pointer to an open, binary file.
//=========================================================================

void classifierSystem::saveStatic(FILE *fptr)
{
  message::saveStatic(fptr);

  fwrite(&seed,sizeof(unsigned),1,fptr);
  fwrite(&geneticInterval,sizeof(int),1,fptr);
  fwrite(&maxCount,sizeof(int),1,fptr);
#ifdef NETWORK
  fwrite(&networkEnabled,sizeof(int),1,fptr);
#endif

  classifierList::saveStatic(fptr);
  messageBoard::saveStatic(fptr);
  environmentInterface::saveStatic(fptr);
#ifdef NETWORK
  networkInterface::saveStatic(fptr);
#endif
}
```

```
//=============================================================================
// FUNCTION:     loadStatic()
//
// DESCRIPTION:  This function is called to load the static members of the
//               classifier system.  This function MUST be called before
//               load().  Note that all loadStatic() functions are called
//               for memeber objects.  In other words, just call loadStatic()
//               and load() in this order to load a classifier system object.
//               You do NOT have to call reset().  If you do, the system will
//               be reset to the parameters of the object you loaded.  Normally
//               one would use object saving and loaded to save a simulation
//               in the middle and come back to it later.  You especially do
//               not want to call reset if you intend to continue the
//               simulation.
//
// I/O:           -->  fptr  -  Pointer to an open, binary file.
//=============================================================================

void classifierSystem::loadStatic(FILE *fptr)
{
  message::loadStatic(fptr);

  fread(&seed,sizeof(unsigned),1,fptr);
  fread(&geneticInterval,sizeof(int),1,fptr);
  fread(&maxCount,sizeof(int),1,fptr);
#ifdef NETWORK
  fread(&networkEnabled,sizeof(int),1,fptr);
#endif

  classifierList::loadStatic(fptr);
  messageBoard::loadStatic(fptr);
  environmentInterface::loadStatic(fptr);
#ifdef NETWORK
  networkInterface::loadStatic(fptr);
#endif
}

//=============================================================================
// FUNCTION:     save()
//
// DESCRIPTION:  This function saves the classifier system parameters.  Note
//               that numRandCalls is generated by the global function
//               getRandom().  This quantity must be saved so that the
//               pseudo-random number generator can be "brought up to speed"
//               when the classifier system data is loaded.  This allows
//               the user to save a simulation and return to it later where
//               he or she left off.  This function must be called AFTER
//               saveStatic().  Note that all save() functions are called
//               for member objects.
//
// I/O:           -->  fptr  -  Pointer to an open, binary file.
//=============================================================================

void classifierSystem::save(FILE *fptr)
{
#ifdef NETWORK
  int temp = 1;
  fwrite(&temp,sizeof(int),1,fptr);
  fwrite(&agentID,sizeof(unsigned long),1,fptr);
#else
  int temp = 0;
  fwrite(&temp,sizeof(int),1,fptr);
#endif

  fwrite(&numRandCalls,sizeof(long),1,fptr);
  fwrite(&clockCount,sizeof(int),1,fptr);

  classList.save(fptr);
  msgBoard.save(fptr);
  environ.save(fptr);
```

```
#ifdef NETWORK
  network.save(fptr);
#endif
}

//============================================================================
// FUNCTION:     load()
//
// DESCRIPTION:  This function loads the classifier system parameters.  The
//               pseudo-random number generator is "brought up to speed"
//               by calling rand() (standard C library function)
//               numRandCalls times.  This allows the user to resume a
//               simulation where he or she left off.  Note that all load
//               functions are called for memeber objects.
//
// I/O:           -->  fptr  -  Pointer to an open, binary file.
//============================================================================

void classifierSystem::load(FILE *fptr)
{
  int temp;

#ifdef NETWORK
  fread(&temp,sizeof(int),1,fptr);
  if (temp != 1) { error(3,""); return; }
  fread(&agentID,sizeof(unsigned long),1,fptr);
#else
  fread(&temp,sizeof(int),1,fptr);
  if (temp != 0) { error(3,""); return; }
  fwrite(&temp,sizeof(int),1,fptr);
#endif

  fread(&numRandCalls,sizeof(long),1,fptr);
  fread(&clockCount,sizeof(int),1,fptr);

  classList.load(fptr);
  msgBoard.load(fptr);
  environ.load(fptr);
#ifdef NETWORK
  network.load(fptr);
#endif

  // Bring random number generator up-to-date on random number calls
  srand(seed);
  for (long i=0; i<numRandCalls; i++)
    rand();
}
```

Module msgboard.h

```
#ifndef MSGBOARD_H
#define MSGBOARD_H

//============================================================================
//  The message board class provides the message board onto which messages
//  are posted and from which messages are read.  Two sizes are used in the
//  message board.  One is the maximum size, and the other is the actual
//  size.  The classifier system may be unable to completely fill the
//  message board on any time step, so the actual size is adjusted by
//  classifierList after it posts.  In addition to storing messages, the
//  message board also stores the number of the classifier which posted the
//  message (the supplier).  A negative supplier number should be used for
//  those messages posted strictly by the environment.  The bid is also
//  placed on the message board for use in the support calculation.  If
//  the network code is compiled, the agent ID is also posted.
//============================================================================

class messageBoard
{
   static int length;     // Maximum number of messages on the message board.
   int actualLength;      // Actual number of messages on the message board.

   message *msgList;      // Array of messages.
   int *supplierList;     // Number of the classifier who posted the
                          //   corresponding message.
#ifdef NETWORK
   unsigned long *supplierID;      // ID Agent who posted the message
#endif
   double *bid;           // The bid of the supplier

   //-------------------------------------------------------------------------//
   // Functions to allocate and free all arrays
   //-------------------------------------------------------------------------//
   void allocateAll();
   void deleteAll();

 public:

   //-------------------------------------------------------------------------//
   // Initialization & reset functions
   //-------------------------------------------------------------------------//
   messageBoard() {}
   ~messageBoard() { deleteAll(); }
   void reset(int firstReset=0);
   void softReset();
   static void restoreDefaults();

   //-------------------------------------------------------------------------//
   // Message board access functions
   //-------------------------------------------------------------------------//
   int actualSize(int newSize = -1);
   static int size() { return length; }
#ifdef NETWORK
   void postMessage(int num, message & msg, int supplier, unsigned long ID,
                    double ruleBid);
   void getMessage(int num, message & msg, int * supplier, unsigned long *ID,
                   double *ruleBid);
#else
   void postMessage(int num, message & msg, int supplier, double ruleBid);
   void getMessage(int num, message & msg, int * supplier, double *ruleBid);
#endif
   double getMaxBid();
```

```
    //-------------------------------------------------------------------------//
    // Saving and loading functions
    //-------------------------------------------------------------------------//
    static void saveStatic(FILE *fptr) { fwrite(&length,sizeof(int),1,fptr); }
    static void loadStatic(FILE *fptr) { fread(&length,sizeof(int),1,fptr); }
    void save(FILE *fptr);
    void load(FILE *fptr);

    //-------------------------------------------------------------------------//
    // Static parameter access functions
    //-------------------------------------------------------------------------//
    static void getSettings(int *len) { *len = length; }
    static void setSettings(int len)  { length = len; }
};

#endif
```

Module msgboard.cpp

```
#include"cfs.h"

// Static class memeber initialization
int messageBoard::length = DEF_MSGBOARD_LEN;

//============================================================================
// FUNCTION:     allocateAll()
//
// DESCRIPTION:  Allocates the message board arrays.
//
// I/O:          none
//============================================================================

void messageBoard::allocateAll()
{
  try
  {
  msgList = new message[length];
  supplierList = new int[length];
  bid = new double[length];
#ifdef NETWORK
  supplierID = new unsigned long[length];
#endif
  }
  catch(xalloc) { error(0,"messageBoard::allocateAll()"); }
}

//============================================================================
// FUNCTION:     deleteAll()
//
// DESCRIPTION:  Frees the message board arrays.
//
// I/O:          none
//============================================================================

void messageBoard::deleteAll()
{
  delete [] msgList;
  delete [] supplierList;
  delete [] bid;
#ifdef NETWORK
  delete [] supplierID;
#endif
}

//============================================================================
// FUNCTION:     reset()
//
// DESCRIPTION:  Frees and re-allocates arrays as necesssary.  Initializes
//               the suppliers to -2, indicating that message was not
//               posted by anyone, although this is not really necessary
//               since the actual length is set to 0.  I just like to
//               initialize things so there are no surprises.
//
// I/O:          none
//============================================================================

void messageBoard::reset(int firstReset)
{
  if (!firstReset)
    deleteAll();

  allocateAll();

  for (int i=0; i<length; i++)
```

```
  {
    supplierList[i] = -2;
#ifdef NETWORK
    supplierID[i] = 0;
#endif
    bid[i] = 0.0;
  }

  actualLength = 0;
}

//============================================================================
// FUNCTION:     softReset()
//
// DESCRIPTION:  Called by the classifier system softReset function to
//               reinitialize the system counters without disturbing the
//               state of the system.  This function is used to start a
//               new iteration using the current rules.  The actual length is
//               reset to zero.
//
// I/O:          none
//============================================================================

void messageBoard::softReset()
{
  actualLength = 0;
}

//============================================================================
// FUNCTION:     restoreDefaults()
//
// DESCRIPTION:  Restores static parameters to their default settings.  Reset
//               should be called after this function.
//
// I/O:          none
//============================================================================

void messageBoard::restoreDefaults()
{
  length = DEF_MSGBOARD_LEN;
}

//============================================================================
// FUNCTION:     postMessage()
//
// DESCRIPTION:  Used to post a message to the message board.  The location
//               to post and the index of the posting classifier must also
//               be supplied.  Note that a supplier of -2 means that no
//               classifier supported the message, and a supplier of -1 means
//               that the environment posted the message.  In these two cases,
//               bid should be zero.
//
// I/O:           --> num       -  The location to which the message will
//                                 be posted.
//                --> msg       -  The message to post.
//                --> supplier  -  The classifier that posted the message.
//                --> ID        -  (NETWORK CODE ONLY) the agent ID of the
//                                 agent that posted the message.
//                --> ruleBid   -  The bid of the classifier that posted
//                                 the message.
//============================================================================
#ifdef NETWORK
void messageBoard::postMessage(int num, message & msg, int supplier,
                               unsigned long ID, double ruleBid)
{
  if (num >= length) error(1,"messageBoard::postMessage()");
  msgList[num] = msg;
  supplierList[num] = supplier;
  supplierID[num] = ID;
  bid[num] = ruleBid;
```

```
}
#else
void messageBoard::postMessage(int num, message & msg, int supplier,
                               double ruleBid)
{
  if (num >= length) error(1,"messageBoard::postMessage()");
  msgList[num] = msg;
  supplierList[num] = supplier;
  bid[num] = ruleBid;
}
#endif

//============================================================================
// FUNCTION:     getMessage()
//
// DESCRIPTION:  The exact opposite of postMessge(). Used to read messages
//               one the message board.  Note:  messages are NOT deleted when
//               they are read.  This is just so the environment can read
//               the messages and pick out the ones it wants.  If the
//               environment doesn't use some of the messages, it should
//               simply repost those messages and set the new actual message
//               board size.
//
// I/O:           -->  num       -  The location of the message to be read
//                <--  msg       -  Reference to the message
//                <--  supplier  -  The supplier
//                <--  ID        -  (NETWORK CODE ONLY) the agent ID
//                <--  ruleBid   -  The bid
//============================================================================

#ifdef NETWORK
void messageBoard::getMessage(int num, message & msg, int * supplier,
                              unsigned long *ID, double *ruleBid)
{
  if (num >= length) error(1,"messageBoard::getMessage()");
  msg = msgList[num];
  *supplier = supplierList[num];
  *ID = supplierID[num];
  *ruleBid = bid[num];
}
#else
void messageBoard::getMessage(int num, message & msg, int * supplier,
                              double *ruleBid)
{
  if (num >= length) error(1,"messageBoard::getMessage()");
  msg = msgList[num];
  *supplier = supplierList[num];
  *ruleBid = bid[num];
}
#endif
//============================================================================
// FUNCTION:     actualSize()
//
// DESCRIPTION:  Returns the actual size of the message board.  i.e. the
//               number of messages posted.  With an argument, this function
//               will also set the size of the message board.
//
// I/O:           -->  newSize  -  (optional) The new number of messages on
//                                 the message board.
//                <--             The size of the message board (or new size
//                                 if one is set).
//============================================================================

int messageBoard::actualSize(int newSize)
{
  if (newSize != -1)
    if (newSize <= length) actualLength = newSize;
    else error(1,"messageBoard::actualSize()");

  return actualLength;
```

```
}

//=============================================================================
// FUNCTION:     save()
//
// DESCRIPTION:  Used to save the message board.  This function should be
//               called AFTER saveStatic().
//
// I/O:           -->  fptr  -  A pointer to an open, binary file.
//=============================================================================

void messageBoard::save(FILE *fptr)
{
  for (int i=0; i<length; i++)
    msgList[i].save(fptr);

  fwrite(&actualLength,sizeof(int),1,fptr);
  fwrite(supplierList,sizeof(int),length,fptr);
#ifdef NETWORK
  fwrite(supplierID,sizeof(unsigned long),length,fptr);
#endif
  fwrite(bid,sizeof(double),length,fptr);
}

//=============================================================================
// FUNCTION:     load()
//
// DESCRIPTION:  Used to load the message board.  This function should be
//               called AFTER saveStatic().  It is not necessary to call
//               reset after loading the message board.  This function
//               handles all deallocation and reallocation.
//
// I/O:           -->  fptr  -  A pointer to an open, binary file.
//=============================================================================

void messageBoard::load(FILE *fptr)
{
  deleteAll();
  allocateAll();

  for (int i=0; i<length; i++)
    msgList[i].load(fptr);

  fread(&actualLength,sizeof(int),1,fptr);
  fread(supplierList,sizeof(int),length,fptr);
#ifdef NETWORK
  fread(supplierID,sizeof(unsigned long),length,fptr);
#endif
  fread(bid,sizeof(double),length,fptr);
}

//=============================================================================
// FUNCTION:     getMaxBid()
//
// DESCRIPTION:  Gets the maximum bid of the messages posted on the message
//               board.  This function was written for network use, but can
//               be used by the environment interface if so desired.
//
// I/O:          none
//=============================================================================

double messageBoard::getMaxBid()
{
  double maxBid = 0.0;
  for (int i=0; i<actualLength; i++)
    if (bid[i] > maxBid) maxBid = bid[i];

  return maxBid;
}
```

Module clsslist.h

```
#ifndef CLSSLIST_H
#define CLSSLIST_H

//============================================================================
//  The classifierList class contains the classifier list and all related
//  parameters.  The list consists of an array of condition messages and
//  an array of action messages.  In addition, there are several other
//  parallel arrays which hold specificity, strength, support, bids, etc.
//  The matchFlag array is used to flag classifiers which match the
//  message board.  Therefore a 1 in a particular element of this array
//  means that this classifier matches the message board.  Similarly, the
//  selected array contains a 1 if the corresponding classifier has been
//  selected to post.
//
//  In order to provide maximum flexibility, condition and action masking
//  is included.  Masking allows the user to specificy which alleles in
//  the condition and action messages can have which characters ('0', '1',
//  or '#').  See setMasks() for more information on masking.  Note that
//  "pass-through" is not supported.  Pass-through is used when a '#'
//  appears in the action string.  In such a case, the corresponding allele
//  in the condition message is used in place of the '#' before posting.
//  In this object, a '#' in the action message means "don't care".
//
//  The BBA can be enabled or disabled by setting the BBAenabled flag.
//  When enabled, two additional tasks are performed.  First, support
//  is calculated for each classifier that bids.  Second, winning
//  classifiers pay their supporters.  Note that the 'maxActionsToPost'
//  variable tells the classifier list how many messages it can post on the
//  message board.  See the messageBoard class for more details.
//
//  The supplierSet array is used to hold the unique number of the supplier of
//  the classifier.  For example, if supplierSet[3] == 58, then then 3rd
//  classifier if the list has a supplier whose unique number is 58.  We use
//  unique numbers because the rule discovery operators often modify the
//  rule base during mid-execution, and a supplier that was in the classifier
//  list at the beginning of the clock tick may get killed by CDO.  The
//  payoff functions search the list for the unique rule number rather than
//  an index to ensure the correct rule is paid.
//
//  NOTE:  All probabilities have a resolution of 2 decimal places!!  In other
//         words, don't set a probability lower than 0.01.
//
//  Multiple conditions, bid taxes, producer taxes, and TCOs
//  are not supported at this time, however, I have included variables for
//  to aid in implementation of them.
//============================================================================

class classifierList
{
   //-------------------------------------------------------------------------//
   // Settings
   //-------------------------------------------------------------------------//
   static int length,             // Length of classifier list
              numConditions,      // NOT USED, set to 1.
              BBAenabled,         // Turns Bucket Brigade on or off
              BBAstyle,           // NOT USED, set to 0.
              elitismEnabled,     // Turns Elitism on or off
              producerTaxInterval,// NOT USED, set to 0.
              CDOenabled,         // Turns CDO on or off
              CEOenabled,         // Turns CEO on of off
              TCOenabled,         // NOT USED, set to 0.
              maxActionsToPost;   // Maximum number of actions to post
```

```
   static double bidConstant,     // Bid constant multiplier
                 sexualProb,      // Probability of sexual reproduction
                 mutationProb,    // Probability of mutation
                 startStrength,   // Initial strength of all classifiers
                 headTax,         // Head tax for classifiers
                 bidTax,          // NOT USED
                 producerTax,     // NOT USED
                 eliteThresh,     // Threshold for elite status (% of max stren)
                 strengthCap,     // Maximum allowable strength value
                 bidCap;          // Maximum allowable bid value

   static char *condMask,         // Mask for condition word
               *actionMask;       // Mask for action word

#ifdef NETWORK
   unsigned long agentIDcopy;     // The LCS ID from classifierSystem class
#endif

   //---------------------------------------------------------------------------//
   // The classifier list
   //---------------------------------------------------------------------------//
   message *conditionMsg;         // Condition array
   message *actionMsg;            // Action array
   int *matchFlag;                // Messages matching the message board at a
                                  //   particular time step.  (1 = match)
   int *selected;                 // Classifiers selected to post. (1 = selected)
   double *bid;                   // Bid array
   double *specificity;           // Specificity array
   double *strength;              // Strength array
   double *support;               // Support array
   int *number;                   // Unique classifier number
   int *supplierSet;              // The supplier of the current classifier
   int *tempSupplierSet;          // Temporary supplier set created when eligible
                                  //   classifiers are found.
   int *eliteSet;                 // Flag array that indicates which rules have
                                  //   elite status
#ifdef NETWORK
   unsigned long *supplierID;     // Agent ID for supplier
   unsigned long *tempSupplierID; // Used when eligible classifiers are found.
#endif

   //---------------------------------------------------------------------------//
   // Stats
   //---------------------------------------------------------------------------//
   int numCrossovers, numSexual,  // Number of sexual and asexual crossovers performed
       numMutations,              // Number of mutations performed
       numCDOs,                   // Number of CDOs performed
       numClassifiers,            // Unique classifier number counter
       numCEOs,                   // NOT USED
       numTCOs;                   // NOT USED

   //---------------------------------------------------------------------------//
   // Functions to allocate and free all arrays
   //---------------------------------------------------------------------------//
   void deleteAll();
   void allocateAll();

   //---------------------------------------------------------------------------//
   // Rule selection function
   //---------------------------------------------------------------------------//
   int roulette(double *bids, int *eligible, int maxIndex);

   //---------------------------------------------------------------------------//
   // Internal genetic & rule creation functions
   //---------------------------------------------------------------------------//
   void crossOver();
   void mutation();
   void CDO(const message & cond, int *eligible);
```

```
  int mutateTernary(char *ch)
  {
   if ( (double(getRandom(100))/100.0) < mutationProb )
   {
     ++numMutations;

     if (*ch == '#')
       *ch = (getRandom(100) > 50) ? '1' : '0';
     else if (*ch == '0')
       *ch = (getRandom(100) > 50) ? '1' : '#';
     else
       *ch = (getRandom(100) > 50) ? '0' : '#';

     return 1;
   }
   else
     return 0;
  }

  int mutateBinary(char *ch)
  {
   if ( (double(getRandom(100))/100.0) < mutationProb )
   {
     ++numMutations;
     *ch = (*ch == '0') ? '1' : '0';
     return 1;
   }
   else
     return 0;
  }

  char randomTernary()
  {
    int randNum = getRandom(100);

    if (randNum > 66) return '1';
    else if (randNum > 33) return '0';
    else return '#';
  }

  char randomBinary()
  {
    if (getRandom(100) > 50) return '1';
    else return '0';
  }

  void setClassifier(int num, const message & cond, const message & act,
                     double stren);

public:

  //--------------------------------------------------------------------------//
  // Initialization & reset functions
  //--------------------------------------------------------------------------//
  classifierList() { }
  ~classifierList()
  { deleteAll();
    delete [] condMask; delete [] actionMask;
    condMask = 0; actionMask = 0;
  }
  void reset(int firstReset = 0);
  void softReset();
  static void restoreDefaults();
  void initRule(int num, char *condStr, char *actionStr, double stren);

#ifdef NETWORK
  void setAgentID(unsigned long newID) { agentIDcopy = newID; }
#endif
```

```
   //------------------------------------------------------------------------//
   // Public rule creation functions
   //------------------------------------------------------------------------//
   int CEO(int num, int *eligible);
   int createRule(const message & cond, int *eligible);
   message createRandomMessage(char *mask);

   //------------------------------------------------------------------------//
   // Rule selection function
   //------------------------------------------------------------------------//
   int pickWeakRule(int *eligible);

   //------------------------------------------------------------------------//
   // Classifier list access functions
   //------------------------------------------------------------------------//
   static int isBBAon() { return BBAenabled; }
   static int isCEOon() { return CEOenabled; }
   static int size() { return length; }
   void readStats(int *cross, int *sexual, int *mutations,
                  int *CDOs, int *CEOs);
#ifdef NETWORK
   void getClassifier(int num, message & cond, message & act,
                      double *stren, double *spec, double *sup, double *ruleBid,
                      int *match, int *sel, int *supplier, unsigned long *ID,
                      int *elite, int *ruleNum);
#else
   void getClassifier(int num, message & cond, message & act,
                      double *stren, double *spec, double *sup, double *ruleBid,
                      int *match, int *sel, int *supplier, int *elite,
                      int *ruleNum);
#endif
   void getClassifier(int num, message & cond, message & act, double *stren);
   void addClassifier(int num, char *condStr=0, char *actionStr=0, double stren=0.0);
   void addClassifier(int num, const message & cond, const message & act, double stren);
   double getStrength(int num);
   void payoffRule(int ruleNum, double payoff);
   int whatIsIndex(int ruleNum);
   double getMaxStren();

   //------------------------------------------------------------------------//
   // Saving and loading functions
   //------------------------------------------------------------------------//
   static void loadStatic(FILE *fptr);
   static void saveStatic(FILE *fptr);
   void load(FILE *fptr);
   void save(FILE *fptr);
   void loadRules(FILE *fptr);
   void saveRules(FILE *fptr);

   //------------------------------------------------------------------------//
   // Static parameter access functions
   //------------------------------------------------------------------------//
   static void getSettings(int *len, int *numCond, int *BBAon, int *BBAtype,
                           int *elitismOn, int *prodTaxInt, int *CDOon,
                           int *CEOon, int *TCOon, int *maxPost,
                           double *bConst, double *crossProb,
                           double *mutateProb, double *startStren,
                           double *headTaxRate, double *bidTaxRate,
                           double *prodTaxRate, double *eliteT,
                           double *strenCap, double *bCap)
   { *len = length;  *numCond = numConditions;  *BBAon = BBAenabled;
     *BBAtype = BBAstyle;  *elitismOn = elitismEnabled;
     *prodTaxInt = producerTaxInterval;  *CDOon = CDOenabled;
     *CEOon = CEOenabled;  *TCOon = TCOenabled;  *maxPost = maxActionsToPost;
     *bConst = bidConstant;  *crossProb = sexualProb;
     *mutateProb = mutationProb;  *startStren = startStrength;
     *headTaxRate = headTax;  *bidTaxRate = bidTax;  *prodTaxRate = producerTax;
     *eliteT = eliteThresh;  *strenCap = strengthCap;  *bCap = bidCap;
   }
```

```
   static void setSettings(int len, int numCond, int BBAon, int BBAtype,
                           int elitismOn, int prodTaxInt, int CDOon,
                           int CEOon, int TCOon, int maxPost,
                           double bConst, double crossProb,
                           double mutateProb, double startStren,
                           double headTaxRate, double bidTaxRate,
                           double prodTaxRate, double eliteT,
                           double strenCap, double bCap)
   { length = len ;  numConditions = numCond;  BBAenabled = BBAon;
     BBAstyle = BBAtype;  elitismEnabled = elitismOn;
     producerTaxInterval = prodTaxInt;  CDOenabled = CDOon;
     CEOenabled = CEOon;  TCOenabled = TCOon;  maxActionsToPost = maxPost;
     bidConstant = bConst;  sexualProb = crossProb;
     mutationProb = mutateProb;  startStrength = startStren;
     headTax = headTaxRate;  bidTax = bidTaxRate;  producerTax = prodTaxRate;
     eliteThresh = eliteT;  strengthCap = strenCap;  bidCap = bCap;
   }
   static void setMasks(char *cMask, char *aMask);

   //-------------------------------------------------------------------------//
   // LCS execution cycle functions
   //-------------------------------------------------------------------------//
   void genetics() { crossOver();  mutation(); }
   void getMatches(messageBoard & msgBoard);
   void calcBids();
#ifdef NETWORK
   void adjustStrengths(networkInterface & network);
#else
   void adjustStrengths();
#endif
   void fillMsgBoard(messageBoard & msgBoard);
   void checkStrengths();
   void taxes();
};

#endif
```

Module clsslist.cpp

```
#include"cfs.h"

// Static class memeber initialization
int classifierList::length = DEF_CLASSLIST_LEN;
int classifierList::numConditions = DEF_NUM_COND;
int classifierList::BBAenabled = DEF_BBA_ENABLED;
int classifierList::BBAstyle = DEF_BBA_STYLE;
int classifierList::elitismEnabled = DEF_ELITISM_ENABLED;
int classifierList::producerTaxInterval = DEF_PROD_TAX_INTERVAL;
int classifierList::maxActionsToPost = DEF_MAX_ACTIONS_TO_POST;
int classifierList::CDOenabled = DEF_CDO_ENABLED;
int classifierList::CEOenabled = DEF_CEO_ENABLED;
int classifierList::TCOenabled = DEF_TCO_ENABLED;

double classifierList::bidConstant = DEF_BID_CONST;
double classifierList::sexualProb = DEF_SEXUAL_PROB;
double classifierList::mutationProb = DEF_MUTATION_PROB;
double classifierList::startStrength = DEF_START_STREN;
double classifierList::headTax = DEF_HEAD_TAX;
double classifierList::bidTax = DEF_BID_TAX;
double classifierList::producerTax = DEF_PRODUCER_TAX;
double classifierList::eliteThresh = DEF_ELITE_THRESH;
double classifierList::strengthCap = DEF_STRENGTH_CAP;
double classifierList::bidCap = DEF_BID_CAP;
char * classifierList::condMask = 0;        // MUST be set to 0!
char * classifierList::actionMask = 0;      // MUST be set to 0!

//============================================================================
// FUNCTION:     deleteAll()
//
// DESCRIPTION:  Deletes all allocated arrays, except the condition and
//               action masks.  See "setMasks()" for informtion on these two
//               arrays.
//
// I/O:          none
//============================================================================

void classifierList::deleteAll()
{
  delete [] conditionMsg;
  delete [] actionMsg;
  delete [] matchFlag;
  delete [] selected;
  delete [] bid;
  delete [] specificity;
  delete [] strength;
  delete [] support;
  delete [] number;
  delete [] supplierSet;
  delete [] tempSupplierSet;
  delete [] eliteSet;
#ifdef NETWORK
  delete [] supplierID;
  delete [] tempSupplierID;
#endif
}
```

```
//==========================================================================
// FUNCTION:     allocateAll()
//
// DESCRIPTION:  Allocates all arrays, except for the condition and action
//               masks.  See "setMasks()" for information on these two
//               arrays.  Note that tempSupplierSet is a temporary array
//               used to store the suppliers of classifiers which are
//               elibigle to post at a given time step.  supplierSet is
//               updated only for those rules which are selected to post.
//               Therefore, to trace chains at any given timestep, you MUST
//               use supplierSet and NOT tempSupplierSet.  Also, the
//               supplierSet array holds an index not the unique classifier
//               number stored in 'number'.
//
// I/O:          none
//==========================================================================

void classifierList::allocateAll()
{
  try
  {
    conditionMsg = new message[length];
    actionMsg = new message[length];
    matchFlag = new int[length];
    selected = new int[length];
    bid = new double[length];
    specificity = new double[length];
    strength = new double[length];
    support = new double[length];
    number = new int[length];
    supplierSet = new int[length];
    tempSupplierSet = new int[length];
    eliteSet = new int[length];
#ifdef NETWORK
    supplierID = new unsigned long[length];
    tempSupplierID = new unsigned long[length];
#endif
  }
  catch(xalloc) { error(0,"classifierList::allocateAll()"); }
}

//==========================================================================
// FUNCTION:     reset()
//
// DESCRIPTION:  This function is called to allocate or re-allocate all
//               arrays in the classifier list.  Array values and statistics
//               variables are also initialized here.  Note that for
//               supplierSet, -2 indicates no support, and -1 indicates
//               environment support.  Support numbers >= 0 are reserved for
//               use by classifierList for keeping tracking of rule to rule
//               support.
//
// I/O:          --> firstReset - Set to 1 only on the first time this function
//                                is called.  (The default is 0).
//==========================================================================

void classifierList::reset(int firstReset)
{
  // Delete the old allocations if nec.
  if (!firstReset)
    deleteAll();

  // Allocate classifier arrays
  allocateAll();
```

```
  // Initialize classifier parameters
  for (int i=0; i<length; i++)
  {
    strength[i] = startStrength;
    bid[i] = 0.0;
    support[i] = 0.0;
    specificity[i] = 0.0;
    matchFlag[i] = 0;
    selected[i] = 0;
    number[i] = i;
    supplierSet[i] = -2;  // -2 means no support
    eliteSet[i] = 0;
#ifdef NETWORK
    supplierID[i] = 0;
    tempSupplierID[i] = 0;
#endif
  }

  // Reset masks to default of all X's

  int msgSize = conditionMsg->size();
  char *mask;
  try { mask = new char[msgSize+1]; }
  catch(xalloc) { error(0,"classifierList.reset()"); }

  for (i=0; i<msgSize; i++) mask[i] = 'X';  mask[i] = '\0';
  setMasks(mask,mask);

  delete [] mask;

  // Reset stat variables
  numCrossovers = numSexual = numMutations = numCDOs = 0;
  numCEOs = numTCOs = 0;
  numClassifiers = 0;  // changes as rules are added
}
//=============================================================================
// FUNCTION:     softReset
//
// DESCRIPTION:  Called by the classifier system softReset function to
//               reinitialize the system counters without disturbing the
//               state of the system.  This function is used to start a
//               new iteration using the current rules.
//
// I/O:          none
//=============================================================================

void classifierList::softReset()
{
  // Reset the unique classifier numbers
  for (int i=0; i<length; i++)
    number[i] = i;

  // Reset stat variables
  numCrossovers = numSexual = numMutations = numCDOs = 0;
  numCEOs = numTCOs = 0;
  numClassifiers = length;
}
```

```
//==========================================================================
// FUNCTION:      restoreDefaults()
//
// DESCRIPTION:   Restores the static memebers to their default settings (those
//                used on program startup).  Reset must be called after this
//                function.
//
// I/O:           none
//==========================================================================

void classifierList::restoreDefaults()
{
  BBAstyle = DEF_BBA_STYLE;
  length = DEF_CLASSLIST_LEN;
  numConditions = DEF_NUM_COND;
  BBAenabled = DEF_BBA_ENABLED;
  elitismEnabled = DEF_ELITISM_ENABLED;
  producerTaxInterval = DEF_PROD_TAX_INTERVAL;
  CDOenabled = DEF_CDO_ENABLED;
  CEOenabled = DEF_CEO_ENABLED;
  TCOenabled = DEF_TCO_ENABLED;
  maxActionsToPost = DEF_MAX_ACTIONS_TO_POST;
  bidConstant = DEF_BID_CONST;
  sexualProb = DEF_SEXUAL_PROB;
  mutationProb = DEF_MUTATION_PROB;
  startStrength = DEF_START_STREN;
  headTax = DEF_HEAD_TAX;
  bidTax = DEF_BID_TAX;
  producerTax = DEF_PRODUCER_TAX;
  eliteThresh = DEF_ELITE_THRESH;
  strengthCap = DEF_STRENGTH_CAP;
  bidCap = DEF_BID_CAP;
}

//==========================================================================
// FUNCTION:      setMasks()
//
// DESCRIPTION:   Condition and action message masks are supplied to allow
//                the user to mask alleles in each message.  The user has
//                the option of setting an allele to a particular value or
//                controlling the types of alleles that can appear in a
//                particular message location.  The allowable masking
//                characters are:
//
//                0  -  The classifierList will always put a zero in this
//                      location.
//                1  -  A one will always appear in this location.
//                #  -  A don't care will always appear in this location.
//                X  -  A 0, 1, or # can appear in this location.
//                B  -  A 0 or 1 can appear in this location.
//
//                Therefore, a mask of "01XXXX" would force the first allele
//                to be 0 always, the second allele to be 1 always, and the
//                last four alleles to be 0, 1, or #.  NOTE:  The "pass-
//                through" function in the action string is NOT supported in
//                this software.  The environment interface must perform
//                pass-through, if so desired.
//
//                Also, the condition and action masks must be allocated and
//                freed separately because they are static class variables.
//                They were made static so that multiple instantiations of
//                the classifierList would have the same mask.  Therefore,
//                it is ESSENTIAL that the condMask and actionMask pointers
//                be set to zero at their point of definition.  This routine
//                checks for a zero pointer before deleting any previously
//                allocated space.
//
// I/O:           --> cMask  -  The condition message mask.
//                --> aMask  -  The action message mask.
//==========================================================================
```

```
void classifierList::setMasks(char *cMask, char *aMask)
{
  // Delete only if not the first call to setMasks()
  if (condMask != 0) delete [] condMask;
  if (actionMask != 0) delete [] actionMask;

  message dummy;
  int size = dummy.size();

  if (strlen(cMask) != size || strlen(aMask) != size) error(1,"setMasks()");

  try
  { condMask = new char[size+1];
    actionMask = new char[size+1];
  }
  catch(xalloc) { error(0,"setMasks()"); }

  strcpy(condMask,cMask);
  strcpy(actionMask,aMask);
}

//============================================================================
// FUNCTION:     setClassifier()
//
// DESCRIPTION:  Sets the rule whose index is num to have the given condition,
//               action, and strength.  All other parameters are reset.  This
//               function is called from the addClassifier functions and is
//               therefore private.
//
// I/O:          --> num    - The index of the classifier to change
//               --> cond   - The new condition word
//               --> action - The new action word
//               --> stren  - The new strength
//============================================================================

void classifierList::setClassifier(int num, const message & cond,
                                   const message & action, double stren)
{
  if (num > length) error(1,"classifierList::setClassifier1()");

  conditionMsg[num] = cond;
  actionMsg[num] = action;
  strength[num] = stren;

  bid[num] = support[num] = specificity[num] = 0.0;
  matchFlag[num] = selected[num] = eliteSet[num] = 0;
  supplierSet[num] = -2;  // -2 means no support
#ifdef NETWORK
  supplierID[num] = agentIDcopy;
#endif
}
```

```
//=============================================================================
// FUNCTION:    addClassifier()
//
// DESCRIPTION:
//    This function is used to add a new rule to the classifier list.  The
//    rule is placed in the 'num' slot (array index).  The new rule is given a
//    new unique number.  If the function is called with only a number, the
//    condition and action words are set to random values based on the masks.
//
// I/O:          --> num       - The index of the classifier to change
//               --> condStr   - (optional) The new condition word
//               --> actionStr - (optional) The new action word
//               --> stren     - (optional) The new strength
//=============================================================================

void classifierList::addClassifier(int num, char *condStr, char *actionStr,
                                   double stren)
{
  if (num > length) error(1,"classifierList::addClassifier2()");

  message cond,action;

  if (condStr == 0)
  {
    cond = createRandomMessage(condMask),
    action = createRandomMessage(actionMask);
    stren = startStrength;
  }
  else
  {
    cond = condStr;
    action = actionStr;
  }

  setClassifier(num,cond,action,stren);

  number[num] = numClassifiers++;
}

//=============================================================================
// FUNCTION:     addClassifier()
//
// DESCRIPTION:  Same as previous except that the condition and action words
//               are messages instead of strings.
//
// I/O:          --> num    - The index of the classifier to change
//               --> cond   - The new condition word
//               --> act    - The new action word
//               --> stren  - The new strength
//=============================================================================

void classifierList::addClassifier(int num, const message & cond,
                                   const message & act, double stren)
{
  if (num > length) error(1,"classifierList::addClassifier1()");
  setClassifier(num,cond,act,stren);
  number[num] = numClassifiers++;
}
```

```
//============================================================================
// FUNCTION:     createRandomMessage()
//
// DESCRIPTION:  Creates a randomly generated message with the given mask.
//               See setMasks() function for info on masking.  A mask size
//               that differs from the current message size will cause an
//               error.
//
// I/O:           --> mask - The mask string.
//               <--       - Returns the randomly generated message.
//============================================================================

message classifierList::createRandomMessage(char *mask)
{
  int size = message::size();
  message msg = mask;

  for (int i=0; i<size; i++)
  {
    if (msg[i] == 'X') msg[i] = randomTernary();
    else if (msg[i] == 'B') msg[i] = randomBinary();
  }

  return msg;
}

//============================================================================
// FUNCTION:     pickWeakRule
//
// DESCRIPTION:  Probabilistically picks a weak rule from the classifier list
//               based on the eligible array.  A one in the eligible array
//               indicates that a rule is eligible for selection.  If elitism
//               is on, all elite rules are removed from eligibility.
//               The strengths of the eligible rules are reversed to give the
//               weakest rules the highest probability of being selected.
//
// I/O:           --> eligible - array indicating which rules are eligible
//                               for selection.
//               <--           - returns the INDEX of the rule selected.
//============================================================================

int classifierList::pickWeakRule(int *eligible)
{
  double maxStren = 0.0, *tempStren;

  try { tempStren = new double[length]; }
  catch(xalloc) { error(0,"classifierList::pickWeakRule()"); }

  // Determine the maximum strength of the eligible classifiers
  for (int i=0; i<length; i++)
  {
    if (elitismEnabled && eliteSet[i]) eligible[i] = 0;
    if (eligible[i] && strength[i] > maxStren) maxStren = strength[i];
  }

  // Reverse the strengths so a weak rule will be picked by roulette()
  for (i=0; i<length; i++)  tempStren[i] = maxStren - strength[i];
  int loc = roulette(tempStren,eligible,length);

  delete [] tempStren;

  return loc;
}
```

```
//============================================================================
// FUNCTION:     roulette()
//
// DESCRIPTION:  This function probabilistically picks a value from the
//               bids array.  Larger values have a higher probability of
//               being selected.  All values MUST be >= 0.  The function
//               uses a cumulative sum of the normalized bids along with the
//               uniformly distributed pseudo-random number generator to pick
//               a value.  Values are only selected from those elements tagged
//               with a one in the parallel eligible array.  The index of the
//               selected value is returned.  Note that if all of the
//               values are zero, a value is randomly picked from those
//               tagged as eligible.  If no values are tagged eligible,
//               a value is picked from all the given values.
//
// I/O:          --> bids      -  The list of values from which a value is
//                                 to be selected.
//               --> eligible  -  A parallel array to bids which tags values
//                                 to be used in selection.  1 == use this
//                                 value.
//               --> maxIndex  -  The total number of elements in the
//                                 bids array.
//               <--              The INDEX of the selected value.
//============================================================================

int classifierList::roulette(double *bids, int *eligible, int maxIndex)
{
  double *eligibleBids;
  int *indexSet, choice;

  try
  { eligibleBids = new double[maxIndex];
    indexSet = new int[maxIndex];
  }
  catch(xalloc) { error(0,"classifierList::roulette()"); }

  // Copy eligible bids into new array and sum the eligible bids
  int num=0;
  double sum = 0.0;
  for (int i=0; i<maxIndex; i++)
    if (eligible[i])
    {
      eligibleBids[num] = bids[i];
      sum += bids[i];
      indexSet[num++] = i;
    }

  // Make selection
  if (sum != 0.0)     // Eligible bids with nonzero values
  {
    // Normalize all bids and form cumulative sum
    for (i=0; i<num; i++)
    {
      if (i==0) eligibleBids[i] /= sum;
      else eligibleBids[i] = eligibleBids[i] / sum + eligibleBids[i-1];
    }

    // Pick the first classifier whose normalized bid is greater than the
    // generated random number.

    double randNum = double(getRandom(100)) / 100.0;

    choice = 0;
    while(choice<num && eligibleBids[choice] <= randNum) choice++;

    choice = indexSet[choice];
  }
  else if (num > 0)   // Sum is zero, but some values are still eligible
    choice = indexSet[getRandom(num)];
  else                // No values are eligible
```

```
    choice = getRandom(maxIndex);

  delete [] eligibleBids;
  delete [] indexSet;

  return choice;
}

//=============================================================================
// FUNCTION:      crossOver()
//
// DESCRIPTION:   Crossover is a genetic operator which is designed to
//                reproduce rules.  Rules can be reproduced either sexually
//                or asexually.  Crossover is performed amoung classifiers
//                with the highest strengths.  Sexual crossover occurs with
//                probability 'sexualProb', and two new rules are created.
//                One of the "children" is picked (prob. 50%) and the child
//                gets 1/3 of the strength of each of its parents.  Asexual
//                crossover occurs with probability '1-sexualProb', and a new
//                rule is not created, just copied.  The strength is split
//                between the parent and the child.  The point of crossover
//                for sexual crossover is picked randomly and can be anywhere
//                in the condition or action string of the classifiers chosen
//                to be parents.  Also, once a parent is chosen, it cannot be
//                chosen again to be the second parent or to be the child.
//                i.e. it is removed from the eligibility set before
//                roulette() is called.
//
//                Note that crossover results in the creation of a new rule.
//                As such, this new rule does not inherit the support values
//                of its parents.  Instead, supplierSet is assigned to -2,
//                indicating the creation of a new rule.
//
// I/O:           none
//=============================================================================

void classifierList::crossOver()
{
  int *eligible;

  try { eligible = new int[length]; }
  catch(xalloc) { error(0,"classifierList::crossOver()"); }

  // Increment the crossover counter
  ++numCrossovers;

  // All rules are eligible for first selection
  for (int i=0; i<length; i++) eligible[i] = 1;

  // Determine type of crossover
  if ( (double(getRandom(100))/100.0) < sexualProb )  // Sexual Crossover
  {
    int parent1, parent2;
    message condition1, condition2, action1, action2;

    // Increment sexual crossover counter
    ++numSexual;

    // Pick Parents
    parent1 = roulette(strength,eligible,length);
    eligible[parent1] = 0;
    parent2 = roulette(strength,eligible,length);
    eligible[parent2] = 0;

    // Perform crossover

    condition1 = conditionMsg[parent1];
    condition2 = conditionMsg[parent2];
    action1 = actionMsg[parent1];
    action2 = actionMsg[parent2];
```

```
    int size = conditionMsg->size(),
        xOverPt = getRandom(2*size);

    if (xOverPt < size)  // crossover in condition part
      for (i=xOverPt; i<size; i++)
      {
        condition1[i] = conditionMsg[parent2][i];
        condition2[i] = conditionMsg[parent1][i];
      }
    else                    // crossover in action part
      for (i=xOverPt; i<(2*size); i++)
      {
        int k = i-size;
        action1[k] = actionMsg[parent2][k];
        action2[k] = actionMsg[parent1][k];
      }

    // Pick where the child will go
    int loc = pickWeakRule(eligible);

    // Copy the new condition and action into the selected classifier
    if (getRandom(100) > 50)
      addClassifier(loc, condition1, action1,
                    (strength[parent1]+strength[parent2]) / 3.0);
    else
      addClassifier(loc, condition2, action2,
                    (strength[parent1]+strength[parent2]) / 3.0);

    // Adjust strengths of parents
    strength[parent1] *= 2.0/3.0;
    strength[parent2] *= 2.0/3.0;
  }
  else                                                  // Asexual Crossover
  {
    int parent;

    // Pick Parent
    parent = roulette(strength,eligible,length);
    eligible[parent] = 0;

    // Pick where the child will go
    int loc = pickWeakRule(eligible);

    // Add the child to the classifier list
    addClassifier(loc, conditionMsg[parent], actionMsg[parent],
                  .5 * strength[parent]);

    // Adjust strength of parent
    strength[parent] *= .5;
  }

  delete [] eligible;
}
```

```
//============================================================================
// FUNCTION:     mutation()
//
// DESCRIPTION:  This function mutates the alleles (bits) of a classifier.
//               Mutation is performed with probablilty 'mutationProb'.  Bits
//               are mutated based on their masks.  An 'X' bit is mutated to
//               one of the three alleles {0,1,#}.  A 'B' bit is mutated to
//               either 0 or 1.  Note that a bit is mutated to a different
//               value than it already is.  i.e., the mutation functions
//               don't just randomly pick from the allele set, they create a
//               different allele.  When a mutated rule is created, it keeps
//               the same support and the same strength.  Mutation is viewed as
//               the genetic alternation of a rule, not the creation of a new
//               rule.  Mutation can be viewed as rule "evolution".
//               Note that if elitism in on and a rule has elite status, the
//               rule is not mutated.
//
// I/O:          none
//============================================================================

void classifierList::mutation()
{
  int size = conditionMsg->size();

  for (int i=0; i<length; i++)
  {
    if (!elitismEnabled || (elitismEnabled && !eliteSet[i]))
    {
      for (int j=0; j<size; j++)
      {
        if (condMask[j] == 'X') mutateTernary(&conditionMsg[i][j]);
        else if (condMask[j] == 'B') mutateBinary(&conditionMsg[i][j]);
      }
      for (j=0; j<size; j++)
      {
        if (actionMask[j] == 'B') mutateBinary(&actionMsg[i][j]);
        else if (actionMask[j] == 'X') mutateTernary(&actionMsg[i][j]);
      }
    }
  }
}

//============================================================================
// FUNCTION:     CDO()
//
// DESCRIPTION:  CDO creates a new classifier with the given condition.  This
//               function is used when there is a message on the message
//               board which has no match in the classifier list.
//
//               The eligibility array is provided so that if multiple CDOs
//               are performed in one time step, one CDO will not replace
//               another CDO.  This can happen if all of the other rules have
//               high strengths and the starting strength value is low.  This
//               function modifies the eligibility list by making the newly-
//               created rule ineligible.  Also, if elitism is enabled, those
//               rules which are elite cannot be replaced by CDO.
//
//               Also, the rule is given default starting values for a new
//               classifier, since the rule created is not the genetic
//               modification of another rule but an entirely new one.  See
//               createRule() for more details.
//
// I/O:           -->  cond      -  The condition to use in the new rule.
//               <-->  eligible  -  The list of eligible locations for the
//                                  new rule to go.
//============================================================================

void classifierList::CDO(const message & cond, int *eligible)
{
  ++numCDOs;
```

```
  // Remove elite rules from eligibility
  if (elitismEnabled)
    for (int i=0; i<length; i++)
      if (eliteSet[i]) eligible[i] = 0;

  createRule(cond,eligible);
}

//============================================================================
// FUNCTION:     CEO()
//
// DESCRIPTION:  The purpose of this function is to create a new rule with the
//               current rule's condition and a random action.  This function
//               is intended for use in the environment object when a
//               situation arises where there are no environment messages and
//               one (or more) needs to be created.  Note that this function
//               would be called AFTER messages have been posted, and
//               therefore, messages on the message board should be excluded
//               from eligibility. (i.e. tagged with a 0 in the eligibile
//               array.)  The newly-created classifier will be automatically
//               removed from the eligibility array in case multiple CEO's are
//               to be performed.  Conceptually, the CEO creates an additional
//               rule that would have been posted on the message board along
//               with the others.
//
//               Note that this routine is functionally identical to CDO,
//               except that a specific rule is specified to be used as a
//               "base" for the createRule function.  Therefore, the newly-
//               created rule is given the SAME supplierSet has the rule upon
//               which it is based.
//
//               As with CDO, the eligibility array is provided if multiple
//               CEO's must be performed.  Each newly created rule is
//               removed from the eligibility list (by the CDO function).
//               Also, those rules which have already been selected to post
//               must also not be eligible
//
//
// I/O:           -->  cond      -  The condition to use in the new rule.
//               <-->  eligible  -  The list of eligible locations for the
//                                  new rule to go.
//               <--             -  The location of the newly-created rule.
//============================================================================

int classifierList::CEO(int num, int *eligible)
{
  ++numCEOs;

  // Remove elite rules from eligibility
  if (elitismEnabled)
    for (int i=0; i<length; i++)
      if (eliteSet[i]) eligible[i] = 0;

  // Store supplier since the createRule function can replace rule num.
  int oldSupplier = supplierSet[num], newRule;
#ifdef NETWORK
  unsigned long oldID = supplierID[num];
#endif
  newRule = createRule(conditionMsg[num],eligible);

  supplierSet[newRule] = oldSupplier;
#ifdef NETWORK
  supplierID[newRule] = oldID;
#endif

  return newRule;
}

//============================================================================
```

```
  // Remove elite rules from eligibility
  if (elitismEnabled)
    for (int i=0; i<length; i++)
      if (eliteSet[i]) eligible[i] = 0;

  createRule(cond,eligible);
}

//============================================================================
// FUNCTION:     CEO()
//
// DESCRIPTION:  The purpose of this function is to create a new rule with the
//               current rule's condition and a random action.  This function
//               is intended for use in the environment object when a
//               situation arises where there are no environment messages and
//               one (or more) needs to be created.  Note that this function
//               would be called AFTER messages have been posted, and
//               therefore, messages on the message board should be excluded
//               from eligibility. (i.e. tagged with a 0 in the eligibile
//               array.)  The newly-created classifier will be automatically
//               removed from the eligibility array in case multiple CEO's are
//               to be performed.  Conceptually, the CEO creates an additional
//               rule that would have been posted on the message board along
//               with the others.
//
//               Note that this routine is functionally identical to CDO,
//               except that a specific rule is specified to be used as a
//               "base" for the createRule function.  Therefore, the newly-
//               created rule is given the SAME supplierSet has the rule upon
//               which it is based.
//
//               As with CDO, the eligibility array is provided if multiple
//               CEO's must be performed.  Each newly created rule is
//               removed from the eligibility list (by the CDO function).
//               Also, those rules which have already been selected to post
//               must also not be eligible
//
//
// I/O:           -->  cond      -  The condition to use in the new rule.
//               <-->  eligible  -  The list of eligible locations for the
//                                  new rule to go.
//               <--             -  The location of the newly-created rule.
//============================================================================

int classifierList::CEO(int num, int *eligible)
{
  ++numCEOs;

  // Remove elite rules from eligibility
  if (elitismEnabled)
    for (int i=0; i<length; i++)
      if (eliteSet[i]) eligible[i] = 0;

  // Store supplier since the createRule function can replace rule num.
  int oldSupplier = supplierSet[num], newRule;
#ifdef NETWORK
  unsigned long oldID = supplierID[num];
#endif
  newRule = createRule(conditionMsg[num],eligible);

  supplierSet[newRule] = oldSupplier;
#ifdef NETWORK
  supplierID[newRule] = oldID;
#endif

  return newRule;
}

//============================================================================
```

```
// FUNCTION:     createRule()
//
// DESCRIPTION:  The purpose of this function is to create a new rule in the
//               classifier list with the given condition.  This function is
//               called by the CDO and CEO functions. In order to determine
//               where this classifier will go, the roulette wheel algorithm
//               is used with all of the strengths reversed
//               (i.e. maxStrength - Strength).  This allows the weakest
//               classifier to have the highest chance of being selected.
//
//               The eligibility array is provided to allow selective
//               replacement, i.e. exclude some classifiers from roulette
//               selection.  Note that the newly created rule is automatically
//               made ineligible by setting its location in the eligible
//               array to 0.
//
//               Finally, new rules are not considered the result of
//               genetics, but rather as "learned" behavior.  That is, this
//               function creates a rule which provides an guaranteed action
//               for a particular condition.  Therefore, the new rule is given
//               the default starting strength and no support number.
//               Essentially, the rule it replaces is destroyed, not
//               genetically altered.
//
// I/O:           -->  cond      -  The condition to use in the new rule.
//               <-->  eligible  -  The list of eligible locations for the
//                                  new rule to go.
//               <--             -  The index where the new rule was placed.
//=============================================================================

int classifierList::createRule(const message & cond, int *eligible)
{
  // Create a random action
  message act = createRandomMessage(actionMask);

  // Pick where the child will go
  int loc = pickWeakRule(eligible);

  // Add the new classifier to the list at loc
  addClassifier(loc,cond,act,startStrength);

  return loc;
}

//=============================================================================
// FUNCTION:     readStats()
//
// DESCRIPTION:  Returns the statistics from the genetic and rule creation
//               operations.  Note that the number of asexual crossovers is
//               the total number of crossovers - the number of sexual
//               crossovers.  Also, the mutations count is a count of the
//               number of bits mutated, not the number of rules mutated.
//
// I/O:          <--  cross      -  The total number of crossovers performed.
//               <--  sexual     -  The number of sexual crossovers performed.
//               <--  mutations  -  The number of mutations performed.
//               <--  CDOs       -  The number of CDOs performed.
//               <--  CEOs       -  The number of CEOs performed.
//=============================================================================
```

```
void classifierList::readStats(int *cross, int *sexual, int *mutations,
                               int *CDOs, int *CEOs)
{
 *cross = numCrossovers;
 *sexual = numSexual;
 *mutations = numMutations;
 *CDOs = numCDOs;
 *CEOs = numCEOs;
}

//============================================================================
// FUNCTION:     loadStatic()
//
// DESCRIPTION:  Loads all static parameters in the classifierList object
//               from an already-opened file.  The load() function MUST
//               be called immediately AFTER this function.  Reset does NOT
//               need to be called.  load() will handle all necessary
//               deallocation and reallocation tasks.
//
// I/O:          -->  fptr  -  File pointer to an opened, binary file.
//============================================================================

void classifierList::loadStatic(FILE *fptr)
 {
  fread(&length,sizeof(int),1,fptr);
  fread(&numConditions,sizeof(int),1,fptr);
  fread(&BBAenabled,sizeof(int),1,fptr);
  fread(&BBAstyle,sizeof(int),1,fptr);
  fread(&elitismEnabled,sizeof(int),1,fptr);
  fread(&producerTaxInterval,sizeof(int),1,fptr);
  fread(&maxActionsToPost,sizeof(int),1,fptr);
  fread(&CDOenabled,sizeof(int),1,fptr);
  fread(&CEOenabled,sizeof(int),1,fptr);
  fread(&TCOenabled,sizeof(int),1,fptr);
  fread(&bidConstant,sizeof(double),1,fptr);
  fread(&sexualProb,sizeof(double),1,fptr);
  fread(&mutationProb,sizeof(double),1,fptr);
  fread(&startStrength,sizeof(double),1,fptr);
  fread(&headTax,sizeof(double),1,fptr);
  fread(&bidTax,sizeof(double),1,fptr);
  fread(&producerTax,sizeof(double),1,fptr);
  fread(&eliteThresh,sizeof(double),1,fptr);
  fread(&strengthCap,sizeof(double),1,fptr);
  fread(&bidCap,sizeof(double),1,fptr);

  message dummy;
  int msgSize = dummy.size();
  char *cMask, *aMask;

  try
  { cMask = new char[msgSize+1];
    aMask = new char[msgSize+1];
  }
  catch(xalloc) { error(0,"classifierList::loadStatic()"); }

  fread(cMask,sizeof(char),msgSize+1,fptr);
  fread(aMask,sizeof(char),msgSize+1,fptr);
  setMasks(cMask,aMask);

  delete [] cMask;
  delete [] aMask;
}
```

```
//============================================================================
// FUNCTION:     saveStatic()
//
// DESCRIPTION:  Saves all static parameters from the classifierList object.
//               This function MUST be called BEFORE save().
//
// I/O:          -->  fptr  -  File pointer to an opened, binary file.
//============================================================================

void classifierList::saveStatic(FILE *fptr)
{
  fwrite(&length,sizeof(int),1,fptr);
  fwrite(&numConditions,sizeof(int),1,fptr);
  fwrite(&BBAenabled,sizeof(int),1,fptr);
  fwrite(&BBAstyle,sizeof(int),1,fptr);
  fwrite(&elitismEnabled,sizeof(int),1,fptr);
  fwrite(&producerTaxInterval,sizeof(int),1,fptr);
  fwrite(&maxActionsToPost,sizeof(int),1,fptr);
  fwrite(&CDOenabled,sizeof(int),1,fptr);
  fwrite(&CEOenabled,sizeof(int),1,fptr);
  fwrite(&TCOenabled,sizeof(int),1,fptr);
  fwrite(&bidConstant,sizeof(double),1,fptr);
  fwrite(&sexualProb,sizeof(double),1,fptr);
  fwrite(&mutationProb,sizeof(double),1,fptr);
  fwrite(&startStrength,sizeof(double),1,fptr);
  fwrite(&headTax,sizeof(double),1,fptr);
  fwrite(&bidTax,sizeof(double),1,fptr);
  fwrite(&producerTax,sizeof(double),1,fptr);
  fwrite(&eliteThresh,sizeof(double),1,fptr);
  fwrite(&strengthCap,sizeof(double),1,fptr);
  fwrite(&bidCap,sizeof(double),1,fptr);

  message dummy;
  int msgSize = dummy.size();
  fwrite(condMask,sizeof(char),msgSize+1,fptr);
  fwrite(actionMask,sizeof(char),msgSize+1,fptr);
}

//============================================================================
// FUNCTION:     load()
//
// DESCRIPTION:  Loads the classifier list and all supporting variables.
//               This function MUST be called AFTER loadStatic() since
//               allocations and deallocations are based on the static class
//               variables.
//
// I/O:          -->  fptr  -  File pointer to an opened, binary file.
//============================================================================
void classifierList::load(FILE *fptr)
{
  // Reallocate
  deleteAll();
  allocateAll();

  // Load classifier list

  for (int i=0; i<length; i++)
  {
    conditionMsg[i].load(fptr);
    actionMsg[i].load(fptr);
  }

  fread(matchFlag,sizeof(int),length,fptr);
  fread(bid,sizeof(double),length,fptr);
  fread(specificity,sizeof(double),length,fptr);
  fread(strength,sizeof(double),length,fptr);
  fread(support,sizeof(double),length,fptr);
  fread(number,sizeof(int),length,fptr);
  fread(selected,sizeof(int),length,fptr);
  fread(supplierSet,sizeof(int),length,fptr);
```

```
  fread(tempSupplierSet,sizeof(int),length,fptr);
  fread(eliteSet,sizeof(int),length,fptr);
#ifdef NETWORK
  fread(&agentIDcopy,sizeof(unsigned long),1,fptr);
  fread(supplierID,sizeof(unsigned long),length,fptr);
  fread(tempSupplierID,sizeof(unsigned long),length,fptr);
#endif

  // Load stats
  fread(&numCrossovers,sizeof(int),1,fptr);
  fread(&numSexual,sizeof(int),1,fptr);
  fread(&numMutations,sizeof(int),1,fptr);
  fread(&numCDOs,sizeof(int),1,fptr);
  fread(&numCEOs,sizeof(int),1,fptr);
  fread(&numTCOs,sizeof(int),1,fptr);
  fread(&numClassifiers,sizeof(int),1,fptr);
}

//============================================================================
// FUNCTION:     save()
//
// DESCRIPTION:  Saves the classifier list and all supporting variables.
//               This function MUST be called AFTER saveStatic().
//
// I/O:          -->  fptr  -  File pointer to an opened, binary file.
//============================================================================

void classifierList::save(FILE *fptr)
{
  // Save classifier list

  for (int i=0; i<length; i++)
  {
    conditionMsg[i].save(fptr);
    actionMsg[i].save(fptr);
  }

  fwrite(matchFlag,sizeof(int),length,fptr);
  fwrite(bid,sizeof(double),length,fptr);
  fwrite(specificity,sizeof(double),length,fptr);
  fwrite(strength,sizeof(double),length,fptr);
  fwrite(support,sizeof(double),length,fptr);
  fwrite(number,sizeof(int),length,fptr);
  fwrite(selected,sizeof(int),length,fptr);
  fwrite(supplierSet,sizeof(int),length,fptr);
  fwrite(tempSupplierSet,sizeof(int),length,fptr);
  fwrite(eliteSet,sizeof(int),length,fptr);
#ifdef NETWORK
  fwrite(&agentIDcopy,sizeof(unsigned long),1,fptr);
  fwrite(supplierID,sizeof(unsigned long),length,fptr);
  fwrite(tempSupplierID,sizeof(unsigned long),length,fptr);
#endif

  // Save stats
  fwrite(&numCrossovers,sizeof(int),1,fptr);
  fwrite(&numSexual,sizeof(int),1,fptr);
  fwrite(&numMutations,sizeof(int),1,fptr);
  fwrite(&numCDOs,sizeof(int),1,fptr);
  fwrite(&numCEOs,sizeof(int),1,fptr);
  fwrite(&numTCOs,sizeof(int),1,fptr);
  fwrite(&numClassifiers,sizeof(int),1,fptr);
}
```

```
//=============================================================================
// FUNCTION:     getMatches()
//
// DESCRIPTION:
//
//   This function checks for matches between the posted messages on the
//   message board and the classifier list conditions.  If the message board
//   has more than one message (i.e. the BBA is in effect), support for a
//   matching classifier is taken from the first matching message on the
//   board.  Note that the "actual msgBoard size" is used instead of the
//   maximum size since there may not be a full msgBoard of information from
//   the previous time step.  The actual msgBoard size will be at least 1
//   since the environment always gets to post.
//
//   e.g.,
//         # message  posting classifier         #  classifier    Support
//         1  1101          3                   0  1101 | 1110      3
//         2  1001          10
//         3  1101          6
//
//   In this case, classifiers 3 and 6 both posted messag 1101 on the last
//   time step.  Classifier 0 matches messages 1 and 3, but the support value
//   for classifier 0 is taken from message 1.
//
//   The reason tempSupplierSet is used instead of supplierSet is because the
//   supplierSet values are only changed for those messages who are selected to
//   post, and this selection happens in another function.  The support
//   quantity used in the bid calculation is simply the bid of the supplier.
//   This quantity increases the bid of the eligible classifier.  Note that
//   support is only filled when the BBA is enabled.
//
//   CDO is performed for all msgBoard messages without a matching classifier.
//   Once a CDO is performed, the newly-created rule is excluded from future
//   CDO calls.  This is necessary because it is possible for the CDO to
//   consistently replace the same rule, provided that rule is of a much lower
//   strength than all other rules.  While such a situation is rare, it can
//   lead to an infinite loop in the worst-case scenario.  Therefore, the
//   msgBoard-classList matching loop will need to be executed at most
//   msgBoardLength+1 times.  i.e. worst-case is that each rule has to be
//   created on a separate pass of the loop.  (Note:  CDO can be disabled.)
//
//   Note that while each message board message must have at least one match,
//   a single classifier can match more than one message on the message board.
//   This means that it is possible to get a set of matching (eligible)
//   classifiers that is smaller than the length of the message board.  The
//   only requirement is that each message board message have a match.
//
// I/O:          -->  msgBoard  -  Reference to the message board.
//=============================================================================

void classifierList::getMatches(messageBoard & msgBoard)
{
  int msgBoardLength = msgBoard.actualSize(),
      done = 0, numMsgMatched, loopCount = 0,
      *msgBoardMatch, *supplier, *eligibility;
  message *msg;
  double *ruleBid;
#ifdef NETWORK
  unsigned long *ID;
#endif

  try
  { msgBoardMatch = new int[msgBoardLength];
    supplier = new int[msgBoardLength];
    eligibility = new int[length];
    msg = new message[msgBoardLength];
    ruleBid = new double[msgBoardLength];
#ifdef NETWORK
    ID = new unsigned long[msgBoardLength];
#endif
```

```
  }
  catch(xalloc) { error(0,"classifierList::getMatches()"); }

  // Copy the message board
#ifdef NETWORK
  for (int j = 0; j < msgBoardLength; j++)
    msgBoard.getMessage(j,msg[j],&supplier[j],&ID[j],&ruleBid[j]);
#else
  for (int j = 0; j < msgBoardLength; j++)
    msgBoard.getMessage(j,msg[j],&supplier[j],&ruleBid[j]);
#endif

  // Make all classifiers eligible for CDO.  The CDO function will remove
  // rule eligibility as necessary.
  for (int i=0; i<length; i++) eligibility[i] = 1;

  while (!done)
  {
    ++loopCount;

    // Check loop count to see if this loop has been executed too many times,
    // if so, stop the program.
    if (loopCount > msgBoardLength+1) error(2,"classifierList::getMatches()");

    // Clear match flags for msgBoard messages
    for (j = 0; j < msgBoardLength; j++)
      msgBoardMatch[j] = 0;

    // Check for matches.  Assign support to the first match.  Each msgBoard
    // message must be checked against each classifier in case there are
    // duplicate messages on the msgBoard.  Each msgBoard message must have
    // at least one match.
    for (i = 0; i < length; i++)
    {
      tempSupplierSet[i] = -2;  // Init to no suppliers
#ifdef NETWORK
      tempSupplierID[i] = agentIDcopy;
#endif
      support[i] = 0.0;         // Init to zero support
      matchFlag[i] = 0;         // Init to no match

      for (j=0; j<msgBoardLength; j++)
      {
        if (conditionMsg[i] == msg[j])
        {
          if (!matchFlag[i])
          {
            tempSupplierSet[i] = supplier[j];          // Supplier's number
            if (BBAenabled) support[i] = ruleBid[j];   // Supplier's bid
#ifdef NETWORK
            tempSupplierID[i] = ID[j];                 // Supplier's network ID
#endif
            matchFlag[i] = 1;
          }
          msgBoardMatch[j] = 1;
        }
      }
    }

    // Determine how many message board messages were matched
    numMsgMatched = 0;
    for (j=0; j<msgBoardLength; j++)
      if (msgBoardMatch[j]) ++numMsgMatched;

    // Perform CDO for all messages not matched and check again if CDO is on
    if (numMsgMatched == msgBoardLength || !CDOenabled)
      done = 1;
    else
    {
      for (j = 0; j < msgBoardLength; j++)
```

```
        if (!msgBoardMatch[j])
          CDO(msg[j],eligibility);
      }
    }

    delete [] msgBoardMatch;
    delete [] msg;
    delete [] supplier;
    delete [] ruleBid;
    delete [] eligibility;
#ifdef NETWORK
    delete [] ID;
#endif
}

//============================================================================
// FUNCTION:      calcBids()
//
// DESCRIPTION:
//
//    Calculates the bids for each classifer.  Specificity is calculated using
//    the condition mask.  Only locations marked with an 'X' or '#' are
//    checked.  This ensures that conditions aren't improperly biased when the
//    environment setup dictates that some of the bits must be only 0 or 1.
//    NOTE:  If the condition mask has no X's or #'s in it, the specificity is
//    set to zero!
//
//    Support is determined by the getMatches function.  Each rule that posted
//    on the previous time step put its number and bid on the message board
//    along with the message.  The matching rule's Support is simply the bid
//    of the matching message board message.
//
// I/O:           none
//============================================================================

void classifierList::calcBids()
{
  int size = conditionMsg->size(), maskCount, specCount;

  for (int i=0; i<length; i++)
  {
    if (matchFlag[i])
    {
      // Calculate specificity

      specCount = maskCount = 0;

      for (int j=0; j<size; j++)
        if (condMask[j] == 'X' || condMask[j] == '#')
        {
          ++maskCount;
          if (conditionMsg[i][j] != '#' ) ++specCount;
        }

      if (maskCount > 0) specificity[i] = double(specCount) / double(maskCount);
      else specificity[i] = 0.0;

      // Calculate Bid and set to cap if necessary

      bid[i] = bidConstant * specificity[i] * strength[i] * (1 + support[i]);

      if (bid[i] > bidCap) bid[i] = bidCap;
    }
    else
      bid[i] = 0.0;
  }
}

//============================================================================
// FUNCTION:      fillMsgBoard()
```

```
//
// DESCRIPTION:
//
//  Once this function is reached, there is at least one match for each
//  message on the message board.  This does NOT mean that there are enough
//  eligible classifiers to fill the message board.  It means that there are
//  each message on the message board has a match, even if one classifier
//  matches more than one message.  This function attempts to pick
//  'maxActionsToPost' actions from the eligible classifiers.
//
//  A "select and remove" scheme is used to choose eligible messages.  This
//  means that once an action has been chosen with the roulette wheel, it is
//  removed from further roulette wheel selections.  Note that this changes
//  the probability of being selected because the cumulative sum changes with
//  each classifier removed from consideration.  However, a rule should only
//  be selected once, so this method is necessary.
//
//  Selected classifiers get their supplierSet array updated, since they have
//  new suppliers.  Unselected classifiers retain the previos supplierSet
//  vaule.  In this way, one can trace a chain from start to finish by
//  looking at the supplier for each classifier in the chain.
//
// I/O:          -->  msgBoard  -  Reference to the message board.
//=============================================================================

void classifierList::fillMsgBoard(messageBoard & msgBoard)
{
  int maxCount, numClassMatches, *eligible;

  try { eligible = new int[length]; }
  catch(xalloc) { error(0,"classifierList::fillMsgBoard"); }

  // Determine the number of eligible classifiers to post.  Reset the selected
  // flag array.  The eligible array tells the roulette routine which
  // classifiers to choose from.
  numClassMatches = 0;
  for (int i = 0; i<length; i++)
  {
    if (matchFlag[i]) ++numClassMatches;
    eligible[i] = matchFlag[i];
    selected[i] = 0;
  }

  // Set the maximum allowable number of postings
  if (numClassMatches < maxActionsToPost) maxCount = numClassMatches;
  else maxCount = maxActionsToPost;

  // Select maxCount messages and post
  for (int k=0; k<maxCount; k++)
  {
    int sel = roulette(bid,eligible,length);

    selected[sel] = 1;
    supplierSet[sel] = tempSupplierSet[sel];
#ifdef NETWORK
    supplierID[sel] = tempSupplierID[sel];
    msgBoard.postMessage(k,actionMsg[sel],number[sel],agentIDcopy,bid[sel]);
#else
    msgBoard.postMessage(k,actionMsg[sel],number[sel],bid[sel]);
#endif

    eligible[sel] = 0;
  }

  // Set the actual size of the message board based on the number of messages
  // posted.
  msgBoard.actualSize(maxCount);

  delete [] eligible;
}
```

```
//=============================================================================
// FUNCTION:     adjustStrengths()
//
// DESCRIPTION:
//
//  This function adjusts the strengths of the selected classifiers and the
//  support classifiers.  The selected classifiers, those given the right to
//  post, have their strength reduced by their bid.  The supplier classifiers
//  are those classifiers that posted the messages which fired the
//  selected classifiers.  Each selected classifier has one supplier
//  classifier corresponding to the message which fired the classifier.  e.g.
//
//  msgBoard  Posting ID            Selected Classifiers  Supplier
//   10010        5                   0   10010 | 10101       5
//   11011       10                   6   00101 | 10000       7
//   00101        7                  11   11011 | 11111      10
//
//  Support classifiers are paid the amount of the bid of the selected
//  classifier they support.  Note that the supplier number is obtained
//  from the number array, i.e. the supplier number is unique.
//
//  NETWORK:  If the network code has been compiled, BBA payoff messages are
//            also sent and received.  (The sendPayoff function only sends
//            messages if the network is turned on.)
//
//  This function is only performed if the BBA is enabled.
//
// I/O:          --> network  -  reference to the network interface if network
//                               code has been compiled.
//=============================================================================

#ifdef NETWORK
void classifierList::adjustStrengths(networkInterface & network)
#else
void classifierList::adjustStrengths()
#endif
{
  // Pay bids and supporters only if BBA is turned on
  if (!BBAenabled) return;

  for (int i=0; i<length; i++)
  {
    // Reward only the selected classifiers
    if (selected[i])
    {
      // Pay bid for getting the right to post
      strength[i] -= bid[i];

      // Pay the supplier if action has a supplier
      if (supplierSet[i] >= 0)
      {
#ifdef NETWORK
        if (supplierID[i] != agentIDcopy)
          network.sendPayoff(agentIDcopy,supplierID[i],supplierSet[i],bid[i]);
        else
#endif
        for (int j=0; j<length; j++)
          if (supplierSet[i] == number[j]) strength[j] += bid[i];
      }
    }
  }

#ifdef NETWORK
  network.readPayoff(*this);
#endif
}

//=============================================================================
// FUNCTION:     checkStrengths()
```

```
//
// DESCRIPTION:  Makes sure that all strengths are between 0.0 and
//               strengthCap.  Strengths < 0 cause problems with the
//               roulette wheel function.  Strengths are capped at a
//               saturation point to keep loops from causing overflow.
//               This function also determines which rules are elite if
//               elitism is enabled.
//
//               This function is called ONCE on each clock tick.
//
// I/O:          none
//=============================================================================

void classifierList::checkStrengths()
{
  for (int i=0; i<length; i++)
  {
    if (strength[i] < 0.0) strength[i] = 0.0;
    else if (strength[i] > strengthCap) strength[i] = strengthCap;
  }

  if (elitismEnabled)
  {
    double thresh = eliteThresh * getMaxStren();
    for (i=0; i<length; i++)
    {
      if (strength[i] >= thresh) eliteSet[i] = 1;
      else eliteSet[i] = 0;
    }
  }
}

//=============================================================================
// FUNCTION:     taxes()
//
// DESCRIPTION:  Applies a fixed amount of tax to the strengths of each
//               classifier.  This function performs the head tax only.
//               Bid tax and producer taxers have not been programmed into
//               this version.  If you want 'em, this is the place to put 'em.
//
// I/O:          none
//=============================================================================

void classifierList::taxes()
{
  double mult = 1 - headTax;

  for (int i=0; i<length; i++)
    strength[i] *= mult;
}
```

```
//=============================================================================
// FUNCTION:     loadRules()
//
// DESCRIPTION:  This is a retro-fit function used in testing LCS programming.
//               The read is not parsed in any fancy way, so you better be
//               sure you adhere to the file format.  No error checking is
//               performed.
//
//               The checkStrengths() function is called to make sure the
//               strengths are within range AND to set the elite rules
//               if elitism is enabled.
//
// I/O:          --> fptr  -  pointer to an open ASCII file set for reading.
//=============================================================================

void classifierList::loadRules(FILE *fptr)
{
  message temp;
  char text[81], *cond, *action;

  try
  { cond = new char[temp.size()+1];
    action = new char[temp.size()+1];
  }
  catch(xalloc) { error(0,"classifierList::loadRules()"); }

  for (int i=0; i<length; i++)
  {
    if(fgets(text,80,fptr))
    {

      sscanf(text,"%s %s %lf",cond,action,&strength[i]);
      conditionMsg[i] = cond;
      actionMsg[i] = action;
      bid[i] = support[i] = specificity[i] = 0.0;
      matchFlag[i] = selected[i] = eliteSet[i] = 0;
      number[i] = i;
      supplierSet[i] = -2;  // -2 means no support
    }
  }
  numClassifiers = length;

  delete [] cond;
  delete [] action;

  // Makes sure loaded strengths are within range and set elitism array if nec.
  checkStrengths();
}

//=============================================================================
// FUNCTION:     saveRules()
//
// DESCRIPTION:  The opposite of load rules.  Saves the rules in ASCII so you
//               can edit them.
//
//               Note that the maximum line width is 80 columns.  Make sure
//               that you don't overflow the string.
//
// I/O:          --> fptr  -  pointer to an open ASCII file set for writing.
//=============================================================================

void classifierList::saveRules(FILE *fptr)
{
  char text[81], *cond, *action;

  try
  { cond = new char[message::size()+1];
    action = new char[message::size()+1];
  }
  catch(xalloc) { error(0,"classifierList::saveRules()"); }
```

```
  for (int i=0; i<length; i++)
  {
    conditionMsg[i].msgToStr(cond);
    actionMsg[i].msgToStr(action);
    sprintf(text,"%s %s %lf\n",cond,action,strength[i]);
    fputs(text,fptr);
  }

  delete [] cond;
  delete [] action;
}

//============================================================================
// FUNCTION:     getClassifier()
//
// DESCRIPTION:  Gets all the information about the classifier one could have
//               any possible use for.
//
// I/O:           --> num      - the INDEX of the classifier to get
//                <-- cond     - the condition word
//                <-- act      - the action word
//                <-- stren    - the strength
//                <-- spec     - the specificity last calculated by calcBids()
//                <-- sup      - the support last calcaulted by getMatches()
//                <-- ruleBid  - the bid last calculated by calcBids()
//                <-- match    - the match flag last determined by getMatches()
//                <-- sel      - the select flag last determined by fillMsgBoard()
//                <-- supplier - the supplier (unique classifier number) last
//                               determined by getMatches()
//                <-- ID       - (NETWORK CODE ONLY) the agent ID of the supplier
//                <-- elite    - the elite flag determined by checkStrengths()
//                <-- ruleNum  - the unique number assigned to this rule
//============================================================================

#ifdef NETWORK
void classifierList::getClassifier(int num, message & cond, message & act,
                   double *stren, double *spec, double *sup, double *ruleBid,
                   int *match, int *sel, int *supplier, unsigned long *ID,
                   int *elite, int *ruleNum)
#else
void classifierList::getClassifier(int num, message & cond, message & act,
                   double *stren, double *spec, double *sup, double *ruleBid,
                   int *match, int *sel, int *supplier, int *elite,
                   int *ruleNum)
#endif
{
  if (num > length) error(1,"classifierList::getClassifier()");
  cond = conditionMsg[num];
  act = actionMsg[num];
  *stren = strength[num];
  *spec = specificity[num];
  *sup = support[num];
  *ruleBid = bid[num];
  *match = matchFlag[num];
  *sel = selected[num];
  *supplier = supplierSet[num];
  *elite = eliteSet[num];
  *ruleNum = number[num];
#ifdef NETWORK
  *ID = supplierID[num];
#endif
}

//============================================================================
// FUNCTION:     getClassifier()
//
// DESCRIPTION:  A shorter version of the previos that just gets the condition,
//               action, and strength of a classifier.
//
```

```
// I/O:           --> num      - the INDEX of the classifier to get
//                <-- cond     - the condition word
//                <-- act      - the action word
//                <-- stren    - the strength
//=============================================================================

void classifierList::getClassifier(int num, message & cond, message & act,
                                   double *stren)
{
  if (num > length) error(1,"classifierList::getClassifier() #2");
  cond = conditionMsg[num];
  act = actionMsg[num];
  *stren = strength[num];
}
//=============================================================================
// FUNCTION:     payoffRule()
//
// DESCRIPTION:  This function accepts a unique classifier number and a strength
//               adjustment quantity.  It is intended for use by the environment
//               payoff function to reward a classifier.  Note that it is
//               possible that the classifier does not exist anymore in the list
//               (because of genetics or rule discovery).
//
// I/O:          --> ruleNum - The unique classifier number of the rule to be
//                             paid.
//               <-- payoff  - The payoff quantity.
//=============================================================================

void classifierList::payoffRule(int ruleNum, double payoff)
{
  for (int i=0; i<length; i++)
    if (number[i] == ruleNum) strength[i] += payoff;
}

//=============================================================================
// FUNCTION:     whatIsIndex()
//
// DESCRIPTION:  This function converts a unique rule number to an index into
//               the classifier array so that you can use the getClassifier
//               function to get info about the classifier.  If the classifier
//               does not exist, a -1 is returned.
//
// I/O:          --> ruleNum - the unique number of the classifier of interest.
//=============================================================================

int classifierList::whatIsIndex(int ruleNum)
{
  int i=0;
  while (i < length && number[i] != ruleNum) i++;

  if (i==length) return -1;
  else return i;
}
```

```
//============================================================================
// FUNCTION:     getMaxStren()
//
// DESCRIPTION:  Gets the maximum strength of the classifier in the list.  The
//               function was written for network inteface use, but you may
//               have a use for it in the environment.
//
// I/O:          <--    The maximum strength in the classifier list.
//============================================================================

double classifierList::getMaxStren()
{
  double maxStren = 0.0;

  for (int i=0; i<length; i++)
    if (strength[i] > maxStren) maxStren = strength[i];

  return maxStren;
}

//============================================================================
// FUNCTION:     getStrength()
//
// DESCRIPTION:  Another classifier access function, except this one only
//               returns the strength of the given classifier.
//
// I/O:          --> num  -  The INDEX of the classifier of interest.
//============================================================================

double classifierList::getStrength(int num)
{
  if (num > length) error(1,"classifierList::getStrength()");
  return strength[num];
}
```

Module ei.h

```
#ifndef EI_H
#define EI_H

//========================================================================
//  The purpose of the environment interface is to provide user-definable
//  function stubs which allow the user to taylor the classifier system
//  object for a particular use.  The user can build as many functions and
//  variables as desired into this class, but the following functions
//  must exist, and they must perform certain tasks, as outlined below.
//
//  reset()
//
//    This function is used to reset the classifier system.  This function
//    is called whenever the classifierSystem::reset() function is called.
//    Specifically, it should handle memory allocation and deallocation
//    as do all other reset functions in this software.  It should also
//    handle parameter initialization.
//
//  softReset()
//
//    This function is used to reset the classifier system for a new run
//    without resetting the state of the classifier system.  (i.e. the
//    rules from the previous iteration are not reset).  This function is
//    called whenever classifierSystem::softReset() is called.  In a typical
//    environmental situation, this function should restore the environment to
//    its initial conditions.  Note that the user can call softReset whenever
//    he or she is trying to learn a set of rules by repeated iteration.  Also
//    note that NO ALLOCATION IS PERFORMED when classifierSystem::softReset()
//    is called, so no changes to the classifier system parameters are allowed
//    between iterations where softReset is used.
//
//  restoreDefaults()
//
//    This function is used to restore the environment parameters to their
//    default values.  For example, if the user has a dynamically-allocated
//    array of varying size, the function would be called to reset that
//    array length to its default size.  The user should NOT do any memory
//    allocation in this function.  All memory allocation should be done
//    in the reset() function.  This function is provided solely for the
//    purpose of returning parameters to their default values.  This function
//    is called whenever the classifierSystem::restoreDefaults() function
//    is called.  Then, when classifierSystem::reset() is called, the
//    environmentInteface::reset() function will also be called.  See the
//    next function description for more details.
//
//  saveStatic()
//
//     This function is called by classifierList::saveStatic() to save any
//     static member variables that your environment interface may have.
//     Static member variables are used to set parameters which apply to
//     multiple instantiations of the same object.  The classifier system
//     uses static parameters in this way, so that for LCS or DLCS
//     instantiations, the crucial system parameters do not vary
//     between instantiations.  In general, these are the kinds of parameters
//     that would be restored to their default values with the
//     restoreDefaults() function.
//
//  loadStatic()
//
//     Opposite of saveStatic().
//
//  save()
//
```

```
//      This function is called by classifierList::save() to save any non-
//      static member variables that your environment interface may have.
//      For example, variables containing information about a particular
//      iteration or environment setup might be saved here.  The classifier
//      system object uses saving and loading to restore the object to the
//      exact state when it was saved.  This allows the user to stop an
//      iteration, save it, and come back to it later.
//
//  load()
//
//      Opposite of save();
//
//  readInput()
//
//    This function is called during a clockTic cycle.  The purpose of the
//    function is to post one or more messages from the environment to the
//    message board.  Depending on your setup, this function will perform
//    different tasks.  If you are using tagged messages that differentiate
//    between environment and internal messages, you must be user not to
//    post environment message over the internal messages from the previous
//    time step.  In addition, you must give the environment messages that
//    you post a supplier number of -1, so that the BBA does not attemp to
//    pay off any suppliers for rules fired by enviornment messages.  However,
//    if you are using substrings, then each message will have an environment
//    part and an internal message part.  In this case, simply put the sensor
//    information into the environment substring of each of the messages on
//    the message board.  Of course you are free to set up any other type of
//    message format you can devise, just be sure that the input and output
//    interfaces work together to get internal messages on the message board
//    by the time you exit this function.  For DLCS operation, messages from
//    the network will also be on the message board at this time step (assuming
//    action passing is enabled), and you are free to treat them just like you
//    treat internal messages.  Remeber, messages must be posted on the
//    message board consecutively, starting at zero, and you must set the
//    actual message size of the message board when you are finished posting.
//
//  postOutput()
//
//    This function is called during a clockTic cycle.  The purpose of this
//    function is to post one or more messages to the environment.  The
//    classifier system calls this function after it has selected messages
//    to post from those eligible classifiers in the classifier list.  It is
//    possible to have more messages for the environment than allowed on the
//    message board at this time, since the classifier system does not
//    distinguish between types of messages when choosing messages to post.
//    Therefore, any extra environment messages should be resolved in this
//    function.  When action passing is enabled, any messages left on the
//    message board after exitting this function will be transmitted over the
//    network according to the transmission rules.  Therefore, if you have
//    different tags for internal, environment, and network messages, you
//    must remove those message you do not wish to have sent over the
//    environment.  Those internal messages would then be put back on the
//    message board by the readInput() function.  Again, if you are using
//    substrings, this is not an issue.  Also, the postOutput function
//    should return a flag indicating whether or not the classifier system
//    iteration should stop.  This flag is returned through the
//    classifierSystem::clockTic() function.
//
```

```
//  adjustStengths()
//
//    This is the environment payoff function.  This function is called
//    after postOutput() and is designed to provide environment feedback
//    for the environment messages posted.  What environment feedback
//    entails is adjusting the strength of the classifiers which posted
//    the environment messages.  The strength can be adjusted any way
//    the user sees fit, with the notion of increasing strength for
//    effective rules and decreasing strength for non-effective rules.  The
//    messageBoard::supplier() function can be used to identify which
//    classifier posted the corresponding message board message.  Note that
//    any strengths which end up less than zero are set to zero, and any
//    strengths greater than classifierList::strengthCap are set to
//    strengthCap by the classifier system.
//
//  Finally, a global error catching function called error() must be
//  supplied by the user.  This function is outlined in ei.cpp.
//==========================================================================


class environmentInterface
{
   // Agent ID
#ifdef NETWORK
   unsigned long agentIDcopy;
#endif

 public:
   //-------------------------------------------------------------------------
   // Reset and initialization functions
   //-------------------------------------------------------------------------
   environmentInterface() {}
   ~environmentInterface() {}

   void reset(int firstReset=0);
   void softReset();
   static void restoreDefaults();

   //-------------------------------------------------------------------------
   // Agent access functions
   //-------------------------------------------------------------------------
#ifdef NETWORK
   void setAgentID(unsigned long newID) { agentIDcopy = newID; }
#endif

   //-------------------------------------------------------------------------
   // Saving and loading functions
   //-------------------------------------------------------------------------
   static void saveStatic(FILE *fptr);
   static void loadStatic(FILE *fptr);
   void save(FILE *fptr);
   void load(FILE *fptr);

   //-------------------------------------------------------------------------
   // functions called on a clock tick
   //-------------------------------------------------------------------------
   void readInput(messageBoard & msgBoard, classifierList & classList);
   void adjustStrengths(messageBoard & msgBoard, classifierList & classList);
   int postOutput(messageBoard & msgBoard, classifierList & classList);
};

// Global error messsage function prototype
int error(int num, char *str = "");

#endif
```

Module ei.cpp

```
#define STRICT
#include<windows.h>
#include"task.h"
#include"ui.h"

//============================================================================
// FUNCTION:     error()
//
// DESCRIPTION:  This function is called by the classifier system whenever
//               there is an error in one of the functions.  The error
//               function can also be used to flag other errors in the
//               user-defined portion of the code.  However, the first ten
//               error numbers are reserved for classifier system errors.
//               They are as follows:
//
//                 0     -  Out of memory error.
//                 1     -  Array out of range error.
//                 2     -  Infinite loop error.
//                 3     -  Incompatible file type
//                 4-9   -  RESERVED for future use.
//
//               Errors 0 through 2 are fatal and the program should
//               terminate when it receives one of these errors.  Errors 1
//               and 2 represent a programming bug, most likely in the
//               environment interface.  Error 2 can also occur if there are
//               more message slots on the message board than there are
//               classifiers (see classiferList::getMatches()).  The
//               name of the function where the error occured is also
//               passed to this function to help facilitate debugging.  The
//               user must at least support these three errors, in addition
//               to any other error traps he or she wishes to add.  Also,
//               a return value is provided, although since errors 0 to 2
//               are fatal, the program must terminate and not return from
//               the error function.  Error 3 is called when the user tries to
//               load an no-network LCS from disk using network-compiled
//               code and vice-versa.  This error is non-fatal.
//
//               The user is, of course, welcome to use
//               these error traps in the environment code.
//
// I/O:           -->  num  -  The error number.  Errors 0 through 9 are
//                             RESERVED for classifier system use.
//                -->  str  -  A string passed by the routine that called the
//                             error function.  For all classifier system
//                             errors, this string contains the name of the
//                             function where the error occured.
//               <--           Return value is not defined for the classifier
//                             system.
//============================================================================

int error(int num,char *str)
{
}

//============================================================================
// Function Stubs
//============================================================================

void environmentInterface::reset(int firstReset)
{
}

void environmentInterface::softReset()
{
}
```

```
void environmentInterface::restoreDefaults()
{
}

void environmentInterface::readInput(messageBoard & msgBoard,
                                     classifierList & classList)
{
}

int environmentInterface::postOutput(messageBoard & msgBoard,
                                     classifierList & classList)
{
}

void environmentInterface::adjustStrengths(messageBoard & msgBoard,
                                   classifierList & classList)
{
}

void environmentInterface::saveStatic(FILE *fptr)
{
}

void environmentInterface::loadStatic(FILE *fptr)
{
}

void environmentInterface::save(FILE *fptr)
{
}

void environmentInterface::load(FILE *fptr)
{
}
```

Module ni.h

```
#ifdef NETWORK

//============================================================================
// The networkInterface class provides network message queueing for use in
// communication between mutliple DLCS agents  An LCS can pass three kinds
// of messages:
//
//   1.  Classifiers Messages
//   2.  Action Messages
//   3.  Payoff Messages
//
// Classifier message passing allows multiple agents to share "good" rules.
// High-strength rules are broadcast over the network to all LCS agents.
// Action message passing allows agents to "communicate" with one another.
// In other words, one agent can send an action message over the network which
// causes a rule in another agent to get selected.  In this way, chaining
// can occur between multiple agents.  Payoff messages are primarily used for
// BBA payoff across the network.  i.e., when a chain has formed across the
// network, and the end of the chain gets selected to post, the selected
// rule pays its supplier by sending a payoff message over the network.
//
// The network interface transmits and receives messages by posting to a
// transmit queue and reading from a receive queue.  In this way, the LCS
// does not have to stop and wait for messages (which it couldn't do anyway
// because it wouldn't know exactly how many messages it was waiting for).
// The LCS simply writes to and reads from the queues at specific points in
// the execution cycle.  The actual network is responsible for removing the
// messages from the transmit queue and transmitting them, and for putting all
// incoming messages into the receive queue.
//============================================================================

class networkInterface
{
    static int actionPassingEnabled,      // 1 == action passing enabled
               classifierPassingEnabled,  // 1 == classifier passing enabled
               classifierTxInterval,      // Frequency of classifier transmission
               maxClassifiersToTx,        // Maximum no. of classifiers to transmit
               maxClassifiersToRx,        // Maximum no. of classifiers to receive
               actionTxInterval,          // Frequency of action transmission
               maxActionsToTx,            // Maximum no. of actions to transmit
               maxActionsToRx;            // Maximum no. of actions to receive
    static double TxStrenThresh,          // Transmit strength threshold
                  RxStrenThresh,          // Receive strength threshold
                  TxBidThresh,            // Transmit bid threshold
                  RxBidThresh;            // Receive bid threshold

    queue Tx, Rx;                         // Transmit and Receive queues
    unsigned long agentIDcopy;            // ID copied from classifierSystem class

    void sortClassMsgs(classifierMessage *classMsg, int num);
    void sortActMsgs(actionMessage *actMsg, int num);

  public:

    //-------------------------------------------------------------------------//
    // Initialization and reset functions.
    //-------------------------------------------------------------------------//
    networkInterface() { }
    ~networkInterface() { }
    void reset();
    void softReset();
    static void restoreDefaults();
```

```
    //-------------------------------------------------------------------------//
    // Network interface access functions.
    //-------------------------------------------------------------------------//
    void setAgentID(unsigned long ID) { agentIDcopy = ID; }

    void postNetwork(int clockCount, messageBoard & msgBoard,
                     classifierList & classList);
    void readNetwork(messageBoard & msgBoard, classifierList & classList);
    void sendPayoff(unsigned long source, unsigned long dest, int supplier,
                    double payoff);
    void readPayoff(classifierList & classList);

    int removeFromTxQueue(queueList & qMsg)
    { return Tx.remove(qMsg); }
    void addToRxQueue(const queueList & qMsg)
    { Rx.add(qMsg); }
    static int isActionPassingOn() { return actionPassingEnabled; }
    static int isClassifierPassingOn() { return classifierPassingEnabled; }

    //-------------------------------------------------------------------------//
    // Static parameter access functions.
    //-------------------------------------------------------------------------//
    static void setSettings(int actPass, int classPass, int classTxInt,
                            int maxClassTx, int maxClassRx, int actTxInt,
                            int maxActTx, int maxActRx,
                            double TxStren, double RxStren,
                            double TxBid, double RxBid)
    {
      actionPassingEnabled = actPass; classifierPassingEnabled = classPass;
      classifierTxInterval = classTxInt; maxClassifiersToTx = maxClassTx;
      maxClassifiersToRx = maxClassRx; actionTxInterval = actTxInt;
      maxActionsToTx = maxActTx; maxActionsToRx = maxActRx;
      TxStrenThresh = TxStren; RxStrenThresh = RxStren;
      TxBidThresh = TxBid; RxBidThresh = RxBid;
    }

    static void getSettings(int *actPass, int *classPass, int *classTxInt,
                            int *maxClassTx, int *maxClassRx, int *actTxInt,
                            int *maxActTx, int *maxActRx,
                            double *TxStren, double *RxStren,
                            double *TxBid, double *RxBid)
    {
      *actPass = actionPassingEnabled; *classPass = classifierPassingEnabled;
      *classTxInt = classifierTxInterval; *maxClassTx = maxClassifiersToTx;
      *maxClassRx = maxClassifiersToRx; *actTxInt = actionTxInterval;
      *maxActTx = maxActionsToTx; *maxActRx = maxActionsToRx;
      *TxStren = TxStrenThresh; *RxStren = RxStrenThresh;
      *TxBid = TxBidThresh; *RxBid = RxBidThresh;
    }

    //-------------------------------------------------------------------------//
    // Saving and loading functions.
    //-------------------------------------------------------------------------//
    static void saveStatic(FILE *fptr);
    static void loadStatic(FILE *fptr);
    void save(FILE *fptr);
    void load(FILE *fptr);
};

#endif
```

Module ni.cpp

```
#include"cfs.h"

#ifdef NETWORK

// Static class memeber initialization
int networkInterface::actionPassingEnabled = DEF_ACTION_PASSING_STATE;
int networkInterface::classifierPassingEnabled = DEF_CLASS_PASSING_STATE;
int networkInterface::classifierTxInterval = DEF_CLASS_TX_INTERVAL;
int networkInterface::maxClassifiersToTx = DEF_MAX_CLASS_TO_TX;
int networkInterface::maxClassifiersToRx = DEF_MAX_CLASS_TO_RX;
int networkInterface::actionTxInterval = DEF_ACTION_TX_INTERVAL;
int networkInterface::maxActionsToTx = DEF_MAX_ACTIONS_TO_TX;
int networkInterface::maxActionsToRx = DEF_MAX_ACTIONS_TO_RX;
double networkInterface::TxStrenThresh = DEF_TX_STREN_THRESH;
double networkInterface::RxStrenThresh = DEF_RX_STREN_THRESH;
double networkInterface::TxBidThresh = DEF_TX_BID_THRESH;
double networkInterface::RxBidThresh = DEF_RX_BID_THRESH;

//============================================================================
// FUNCTION:      reset()
//
// DESCRIPTION:   This function is called when the DLCS is reset.  It simply
//                clears the queues.  Note that the 'firstReset' parameter
//                is not needed here because the queues get a reset(1) upon
//                instantiation.
//
// I/O:           none
//============================================================================

void networkInterface::reset()
{
  Tx.reset();
  Rx.reset();
}

//============================================================================
// FUNCTION:      softReset()
//
// DESCRIPTION:   Not used.  Provided for future compatibility.
//
// I/O:           none.
//============================================================================

void networkInterface::softReset()
{
}

//============================================================================
// FUNCTION:      restoreDefaults()
//
// DESCRIPTION:   Restores the DLCS network settings to their defaults.  Reset
//                must be called after this function.
//
// I/O:           none
//============================================================================

void networkInterface::restoreDefaults()
{
  actionPassingEnabled = DEF_ACTION_PASSING_STATE;
  classifierPassingEnabled = DEF_CLASS_PASSING_STATE;
  classifierTxInterval = DEF_CLASS_TX_INTERVAL;
  maxClassifiersToTx = DEF_MAX_CLASS_TO_TX;
  maxClassifiersToRx = DEF_MAX_CLASS_TO_RX;
  actionTxInterval = DEF_ACTION_TX_INTERVAL;
  maxActionsToTx = DEF_MAX_ACTIONS_TO_TX;
```

```
  maxActionsToRx = DEF_MAX_ACTIONS_TO_RX;
  TxStrenThresh = DEF_TX_STREN_THRESH;
  RxStrenThresh = DEF_RX_STREN_THRESH;
  TxBidThresh = DEF_TX_BID_THRESH;
  RxBidThresh = DEF_RX_BID_THRESH;
}

//============================================================================
// FUNCTION:      sortClassMsgs()
//
// DESCRIPTION:   This function sorts classifier message objects by from
//                highest to lowest strength, since the there will often be more
//                classifiers available than are allowed to be transmitted
//                or received.
//
// I/O:           --> classMsg  -  The array of classifier messages to be sorted.
//                --> num       -  The number of messages in the array.
//============================================================================

void networkInterface::sortClassMsgs(classifierMessage * classMsg, int num)
{
  for (int i=num-1; i>=0; i--)
    for (int j=0; j<i; j++)
      if (classMsg[j].strength < classMsg[j+1].strength)
      {
        swap(classMsg[j].condition,classMsg[j+1].condition);
        swap(classMsg[j].action,classMsg[j+1].action);
        swap(classMsg[j].strength,classMsg[j+1].strength);
      }
}

//============================================================================
// FUNCTION:      sortActMsgs()
//
// DESCRIPTION:   This function is called to sort actions messages from largest
//                to smallest strength, since there are often more eligible
//                action messages to be transmitted or received than are allowed.
//
// I/O:           --> actMsg  -  The array of action messages to be sorted.
//                --> num     -  The number of messages in the array.
//============================================================================

void networkInterface::sortActMsgs(actionMessage * actMsg, int num)
{
  for (int i=num-1; i>=0; i--)
    for (int j=0; j<i; j++)
      if (actMsg[j].bid < actMsg[j+1].bid)
      {
        swap(actMsg[j].action,actMsg[j+1].action);
        swap(actMsg[j].supplier,actMsg[j+1].supplier);
        swap(actMsg[j].bid,actMsg[j+1].bid);
        swap(actMsg[j].source,actMsg[j+1].source);
      }
}
```

```
//=============================================================================
// FUNCTION:     postNetwork
//
// DESCRIPTION:
//
//   This function is called after messages are posted to the environment.
//   It performs either classifier passing, action passing, or both depending
//   on the flags set.  Note that classifier and action messages are only
//   passed on the interval defined by 'classifierTxInterval' and
//   'actionTxInterval', respectfully.
//
//   Classifier passing involves the broadcasting of rules over the network.
//   The rule's condition, action, and strength are sent.  In order to decide
//   what rules will get passed, this function determines a minimum strength
//   value required for classifier transmission.  Then, an array of eligible
//   classifier messages is created.  The array is sorted from highest strength
//   to lowest strength, and at most 'maxClassifiersToTx' of the highest-strength
//   rules are put into the transmit queue.  Note that the strength threshold
//   is based on the largest strength in the classifer list when this function
//   is called.  The 'TxStrenThresh' parameter specifies the fraction of
//   maximum strength that the threshold be, e.g. if the maximum strength is
//   50 and TxStrenThresh = .80, the threshold will be 50 * .80 = 40.  Then,
//   all rules with strength higher than 40 will be eligible for transmission.
//
//   Action passing involves the broadcasting of posted actions over the
//   network.  In other words, the posted actions from the current clock tick
//   can be used to communicate with other LCS agents.  This function is called
//   after the postOutput function for the environment.  Instead of a strength
//   threshold, action passing uses a bid threshold to decide which rules are
//   eligible for transmission.  The threshold is calculated in the same manner
//   as above.  The steps taken are as follows.  First, the function uses the
//   bid threshold to decide what rules on the message board are eligible
//   for transmission.  These rules are copied into an array of eligible
//   action messages.  An action message consists of a message, supplier,
//   agent ID, and a bid.  The agent ID is simply the unique long integer
//   assigned to the current classifier.  (An IP address, for example).  The
//   eligible action messages are then sorted from highest to lowest bid,
//   and at most 'maxActionsToTx' are put into the transmit queue.  Note that
//   each message on the message board has a ID associated with it.  This is
//   the ID of the agent who posted the message.  However, for the messages
//   on the message board at this point in the execution cycle, all the ID's
//   will be the same as the current 'agentID'.  The 'agentID' parameter is
//   still passed to this function for use in classifier passing, in which the
//   message board is not accessed.  It is important to note however, that all
//   posted messages on the message board at this time are from the current
//   LCS.
//
// I/O:          --> clockCount  -  The current clock tick.
//               --> msgBoard    -  Reference to the messsage board.
//               --> classList   -  Reference to the classifier list.
//=============================================================================

void networkInterface::postNetwork(int clockCount,
                                   messageBoard & msgBoard,
                                   classifierList & classList)
{
  if (!classifierSystem::isNetworkOn()) return;

  // Classifier passing

  if (classifierPassingEnabled && (clockCount % classifierTxInterval) == 0)
  {
    int numClass = classList.size();
    double strenThresh = classList.getMaxStren() * TxStrenThresh,
           stren;
    message cond, act;
    classifierMessage *classToSend;

    try { classToSend = new classifierMessage[numClass]; }
    catch(xalloc) { error(0,"networkInterface::postNetwork()"); }
```

```
    // Copy classifiers that are eligible to be broadcast
    int j=0;
    for (int i=0; i<numClass; i++)
    {
      classList.getClassifier(i,cond,act,&stren);
      if (stren >= strenThresh)
        classToSend[j++].put(cond,act,stren);
    }

    // Sort the eligible classifiers from highest to lowest strength
    sortClassMsgs(classToSend,j);

    // Post min(maxClassifiersToTx, j) classifiers
    for (i=0; i<maxClassifiersToTx && i<j; i++)
      Tx.add(agentIDcopy,BROADCAST_ID,classToSend[i]);

    delete [] classToSend;
  }

  // Action passing

  if (actionPassingEnabled && (clockCount % actionTxInterval) == 0)
  {
    int msgBoardSize = msgBoard.actualSize(),
        supplier;
    message msg;
    double ruleBid,
           bidThresh = msgBoard.getMaxBid() * TxBidThresh;
    unsigned long ID; // ID of message on action board, will be same as agentID
    actionMessage *actToSend;

    try { actToSend = new actionMessage[msgBoardSize]; }
    catch(xalloc) { error(0,"networkInterface::postNetwork()"); }

    // Copy the action messages that are eligible to be broadcast
    int j=0;
    for (int i=0; i<msgBoardSize; i++)
    {
      msgBoard.getMessage(i,msg,&supplier,&ID,&ruleBid);
      if (ruleBid >= bidThresh)
        actToSend[j++].put(msg,supplier,ID,ruleBid);
    }

    // Sort the action messages from highest to lowest bid
    sortActMsgs(actToSend,j);

    // Post min(maxActionsToTx, j) action messages
    for (i=0; i<maxActionsToTx && i<j; i++)
      Tx.add(agentIDcopy,BROADCAST_ID,actToSend[i]);

    delete [] actToSend;
  }
}
```

```
//============================================================================
// FUNCTION:     readNetwork()
//
// DESCRIPTION:
//
//   This function is called before the readInput function from the environment.
//   The purpose of this function is to retrieve classifier messages, action
//   messages, or both from the receive queue.  The time interval in which
//   such messages will arrive is dictated by the transmission intervals used
//   in the postNetwork() function.  Note that in action passing, rules are
//   added to the END of the current message board so that they do not
//   replace internal rules already on the message board.  The environment is,
//   of course, free to manipulate the received actions as necessary, since
//   the read from the input interface occurs immediately after this function
//   is called.
//
//   Classifier passing works as follows.  All of the currently-enqueued
//   classifier messages are removed from the receive queue.  Those rules are
//   then sorted from highest to lowest strength.  As with rule transmission,
//   a strength threshold is calculated using the maximum strength in the
//   present classifier list (NOT the maximum strength of the received
//   classifiers).  This threshold keeps rules with strengths lower than the
//   majority of the rules in the classifier list from being added to the
//   list.  Up to 'maxClassifiersToRx' eligible rules are put into the classifier
//   list by probabilistically selecting weak rules in the list.  Note that
//   a "select and remove" scheme is used when selecting mulitple weak rules.
//   Therefore, if more than one classifier is to be added to the classifier
//   list, the "select and remove" scheme guarantees that each classifeir to
//   be added gets a unique spot in the message board.
//
//   Action passing works as follows.  First, all of the currently-enqueued
//   action messages are removed from the receive queue.  Those messages are
//   sorted from highest to lowest bid.  Again, a bid threshold is calculated
//   based on the maximum bid of the current message board (i.e. the
//   environment-posted messages).  Up to 'maxActionsToRx' messages with
//   bids higher than this threshold are posted to the message board.  Note
//   that if the message board is filled to capacity before 'maxActionsToRx'
//   is reached, no more messages will be posted.
//
// I/O:          --> msgBoard    -  Reference to the messsage board.
//               --> classList   -  Reference to the classifier list.
//============================================================================

void networkInterface::readNetwork(messageBoard & msgBoard,
                                   classifierList & classList)
{
  if (!classifierSystem::isNetworkOn()) return;

  // Classifier passing

  if (classifierPassingEnabled)
  {
    queueList temp;
    int numMsg = Rx.countType(temp.classMsgType),
        numClass = classList.size(),
        *eligible;
    double strenThresh = classList.getMaxStren() * RxStrenThresh;
    classifierMessage *classRecvd;

    if (numMsg > 0)
    {
      try
      { classRecvd = new classifierMessage[numMsg];
        eligible = new int[numClass];
      }
      catch(xalloc) { error(0,"networkInterface::readNetwork()"); }

      // The eligble array will be used to replace weak classifiers with those
      // from the receive queue.
      for (int i=0; i<numClass; i++) eligible[i] = 1;
```

```
      // Remove the classifiers from the queue
      for (i=0; i<numMsg; i++)
        Rx.remove(classRecvd[i]);

      // Sort the classifiers from largest to smallest strength
      sortClassMsgs(classRecvd,numMsg);

      // Put min(# of eligible classifiers, maxClassifiersToRx) in classifier list
      int j=0;
      for (i=0; i<numMsg && j<maxClassifiersToRx; i++)
        if (classRecvd[i].strength >= strenThresh)
        {
          j++;
          int loc = classList.pickWeakRule(eligible);
          classList.addClassifier(loc, classRecvd[i].condition,
                                  classRecvd[i].action, classRecvd[i].strength);
        }

      delete [] classRecvd;
      delete [] eligible;
    }
  }

  // Action passing

  if (actionPassingEnabled)
  {
    queueList temp;
    int numMsg = Rx.countType(temp.actMsgType),
        msgBoardIndex = msgBoard.actualSize(),
        maxMsgBoardSize = msgBoard.size();
    double bidThresh = msgBoard.getMaxBid() * RxBidThresh;
    actionMessage *actRecvd;

    if (numMsg > 0)
    {
      try { actRecvd = new actionMessage[numMsg]; }
      catch(xalloc) { error(0,"networkInterface::readNetwork()"); }

      // Remove the action messages from the queue
      for (int i=0; i<numMsg; i++)
        Rx.remove(actRecvd[i]);

      // Sort the action messages from highest to lowest bid
      sortActMsgs(actRecvd,numMsg);

      // Post min( # of message slots left on the message board, 'maxActionsToRx',
      // # of eligible actions) actions on the message board
      int j=0;
      for (i=0; i<numMsg && j<maxActionsToRx && msgBoardIndex < maxMsgBoardSize;
           i++)
        if (actRecvd[i].bid >= bidThresh)
        {
          msgBoard.postMessage(msgBoardIndex, actRecvd[i].action,
                               actRecvd[i].supplier, actRecvd[i].source,
                               actRecvd[i].bid);
          j++;  msgBoardIndex++;
        }

      // Set the new message board size
      msgBoard.actualSize(msgBoardIndex);

      delete [] actRecvd;
    }
  }
}
```

```
//============================================================================
// FUNCTION:     sendPayoff()
//
// DESCRIPTION:
//
//   This function is used to send a BBA payoff message to another LCS agent.
//   For example, if a rule from another agent is sent across the network
//   and triggers a rule in this LCS, this LCS can pay the "supplier"
//   across the network.  Payoff messages are given a specific destination,
//   rather than a broadcast, although this is not absolutely necessary.  I've
//   just done it for convenience.  The payoff message could instead by
//   broadcast, and each agent would check to see if the message belongs to
//   it.  I wanted to reduce network traffic.
//
// I/O:          --> source   - The sending agent ID.
//               --> dest     - The receiving agent ID.
//               --> supplier - The rule in the receiving agent to get paid.
//               --> payoff   - The amount to pay the rule.
//============================================================================

void networkInterface::sendPayoff(unsigned long source, unsigned long dest,
                                  int supplier, double payoff)
{
  if (!classifierSystem::isNetworkOn()) return;

  payoffMessage qMsg;

  qMsg.put(supplier,payoff);
  Tx.add(source,dest,qMsg);
}

//============================================================================
// FUNCTION:     readPayoff()
//
// DESCRIPTION:
//
//   This function checks for rule payoff messages in the receive queue and
//   adjusts the strengths of the affected rules accordingly.  Note that this
//   function assumes that the network has only delivered payoff messages to
//   this LCS if they are explicity for this LCS.  (Payoff messages are not
//   broadcast, they are sent to a specific destination.)  In other words,
//   this function doesn't check the destination ID.
//
// I/O:          --> classList  -  reference to the classifier list.
//============================================================================

void networkInterface::readPayoff(classifierList & classList)
{
  if (!classifierSystem::isNetworkOn()) return;

  payoffMessage qMsg;

  while (Rx.remove(qMsg))
    classList.payoffRule(qMsg.supplier,qMsg.payoff);
}

//============================================================================
// FUNCTION:     saveStatic()
//
// DESCRIPTION:  Used to save the static parameters for the network interface.
//               This function must be called before save().
//
// I/O:          --> fptr  -  Pointer to an open BINARY file set to write.
//============================================================================

void networkInterface::saveStatic(FILE *fptr)
{
  fwrite(&actionPassingEnabled,sizeof(int),1,fptr);
  fwrite(&classifierPassingEnabled,sizeof(int),1,fptr);
  fwrite(&classifierTxInterval,sizeof(int),1,fptr);
```

```
  fwrite(&maxClassifiersToTx,sizeof(int),1,fptr);
  fwrite(&maxClassifiersToRx,sizeof(int),1,fptr);
  fwrite(&actionTxInterval,sizeof(int),1,fptr);
  fwrite(&maxActionsToTx,sizeof(int),1,fptr);
  fwrite(&maxActionsToRx,sizeof(int),1,fptr);
  fwrite(&TxStrenThresh,sizeof(double),1,fptr);
  fwrite(&RxStrenThresh,sizeof(double),1,fptr);
  fwrite(&TxBidThresh,sizeof(double),1,fptr);
  fwrite(&RxBidThresh,sizeof(double),1,fptr);
}

//=============================================================================
// FUNCTION:     loadStatic()
//
// DESCRIPTION:  Used to load the static parameters for the network interface.
//               This function must be called before load().
//
// I/O:          --> fptr  -  Pointer to an open BINARY file set to read.
//=============================================================================

void networkInterface::loadStatic(FILE *fptr)
{
  fread(&actionPassingEnabled,sizeof(int),1,fptr);
  fread(&classifierPassingEnabled,sizeof(int),1,fptr);
  fread(&classifierTxInterval,sizeof(int),1,fptr);
  fread(&maxClassifiersToTx,sizeof(int),1,fptr);
  fread(&maxClassifiersToRx,sizeof(int),1,fptr);
  fread(&actionTxInterval,sizeof(int),1,fptr);
  fread(&maxActionsToTx,sizeof(int),1,fptr);
  fread(&maxActionsToRx,sizeof(int),1,fptr);
  fread(&TxStrenThresh,sizeof(double),1,fptr);
  fread(&RxStrenThresh,sizeof(double),1,fptr);
  fread(&TxBidThresh,sizeof(double),1,fptr);
  fread(&RxBidThresh,sizeof(double),1,fptr);
}

//=============================================================================
// FUNCTION:     save()
//
// DESCRIPTION:  Used to save the network interface.  This function must be
//               called after saveStatic();
//
// I/O:          --> fptr  -  Pointer to an open BINARY file set to write.
//=============================================================================

void networkInterface::save(FILE *fptr)
{
  fwrite(&agentIDcopy,sizeof(unsigned long),1,fptr);
  Tx.save(fptr);
  Rx.save(fptr);
}

//=============================================================================
// FUNCTION:     load()
//
// DESCRIPTION:  Used to load the network interface.  This function must be
//               called after loadStatic();
//
// I/O:          --> fptr  -  Pointer to an open BINARY file set to read.
//=============================================================================

void networkInterface::load(FILE *fptr)
{
  fread(&agentIDcopy,sizeof(unsigned long),1,fptr);
  Tx.load(fptr);
  Rx.load(fptr);
}

#endif
```

Module message.h

```
#ifndef MESSAGE_H
#define MESSAGE_H

//============================================================================
//  The message class is used as the basic message block for the entire
//  classifier system.  The classifierList class contains an array of
//  condition messages and an array of action messages.  The messageBoard
//  class contains an array of posted messages.  Note that all messages
//  have the same length, and therefore the 'length' parameter is static.
//  Also, we often refer to "words" instead of messages when we talk about
//  conditions and actions.  Conceptually there is a difference, but
//  syntactically, there isn't.
//============================================================================

class message
{
   static int length;  // Number of alleles (not incl. '\0').
   char *msg;          // Pointer to an allele string

 public:

   //-------------------------------------------------------------------------//
   // Initialization & reset functions and constructors
   //-------------------------------------------------------------------------//
   message();
   ~message();
   static void restoreDefaults();
   message(const char *str);
   void operator=(const char *str);
   message(message & msg2);
   void operator=(const message &msg2);

   //-------------------------------------------------------------------------//
   // Saving and loading functions
   //-------------------------------------------------------------------------//
   static void saveStatic(FILE *fptr) { fwrite(&length,sizeof(int),1,fptr); }
   static void loadStatic(FILE *fptr) { fread(&length,sizeof(int),1,fptr); }
   void save(FILE *fptr) { fwrite(msg,sizeof(char),length+1,fptr); }
   void load(FILE *fptr) { fread(msg,sizeof(char),length+1,fptr); }

   //-------------------------------------------------------------------------//
   // Message access functions and operators
   //-------------------------------------------------------------------------//
   static int size() { return length; }
   char & operator[](int i) const;
   int operator==(const message &msg2);
   int operator==(const char *str);
   int operator!=(const message &msg2);
   int operator!=(const char *str);

   void msgToStr(char *string);
};

inline void swap(message & msg1, message & msg2)
{ message temp = msg1;  msg1 = msg2;  msg2 = temp; }

#endif
```

Module message.cpp

```
#include"cfs.h"

// Static class memeber initialization
int message::length = DEF_MSG_LEN;

//============================================================================
// FUNCTION:     message()
//
// DESCRIPTION:  Message constructor.  Allocates space for the message and
//               initializes all alleles to '0';
//
// I/O:          none
//============================================================================

message::message()
{
  try { msg = new char[length+1]; }
  catch(xalloc) { error(0,"message::message()"); }
  for (int i=0; i<length; i++)
    msg[i] = '0';
  msg[length] = '\0';
}

//============================================================================
// FUNCTION:     ~message()
//
// DESCRIPTION:  Message destructor.  Frees message memory.
//
// I/O:          none
//============================================================================

message::~message()
{
  delete [] msg;
}

//============================================================================
// FUNCTION:     restoreDefaults()
//
// DESCRIPTION:  Restores the static parameters to their default values.
//
// I/O:          none
//============================================================================

void message::restoreDefaults()
{
  length = DEF_MSG_LEN;
}

//============================================================================
// FUNCTION:     operator[]()
//
// DESCRIPTION:  Overloaded array dereference operator.  Gives direct access
//               to the alleles of a message.  The function is const so that
//               it can be used to access both constant and non-constant
//               messages.
//
// I/O:           -->  i  -  The element to return.
//               <--      -  A reference to the char array storing the
//                           alleles.
//============================================================================

char & message::operator[](int i) const
{
 if (i >= length) error(1,"message::operator[]");
```

```
 return msg[i];
}

//============================================================================
// FUNCTION:     operator==()
//
// DESCRIPTION:  Overloaded equivalency operator.  Checks to see if two
//               messages are equivalent.  Equivalent is defined as
//                 0 == 0
//                 1 == 1
//                 # == 0, 1, or #
//
// I/O:           -->  msg2  -  Reference to second message to be compared.
//               <--         -  1 for equivalent, 0 for not equivalent.
//============================================================================

int message::operator==(const message &msg2)
{
 int equivalent = 1;

 for (int i=0; i<length; i++)
   if (msg[i] != '#' && msg2.msg[i] != '#' &&
       msg[i] != msg2.msg[i]) equivalent = 0;

 return equivalent;
}

//============================================================================
// FUNCTION:     operator!=()
//
// DESCRIPTION:  Overloaded non-equivalency operator.  Checks to see if two
//               messages are different.  See operator==().
//
// I/O:           -->  msg2  -  Reference to second message to be compared.
//               <--         -  1 for not equivalent, 0 for equivalent.
//============================================================================

int message::operator!=(const message &msg2)
{
 if (*this == msg2) return 0;
 else return 1;
}

//============================================================================
// FUNCTION:     operator==()
//
// DESCRIPTION:  Same as previous, except compares message to a string.
//
// I/O:           -->  str  -  Pointer to string to be compared.
//               <--        -  1 for equivalent, 0 for not equivalent.
//============================================================================

int message::operator==(const char *str)
{
 int equivalent = 1;

 if (strlen(str) != length) error(1,"message::operator==(char*)");

 for (int i=0; i<length; i++)
   if (msg[i] != '#' && str[i] != '#' &&
       msg[i] != str[i]) equivalent = 0;

 return equivalent;
}
```

```
//===========================================================================
// FUNCTION:     operator!=()
//
// DESCRIPTION:  Same as previous, except compares message to a string.
//
// I/O:           -->  str  -  Pointer to string to be compared.
//               <--       -  1 for not equivalent, 0 for equivalent.
//===========================================================================

int message::operator!=(const char *str)
{
 if (*this == str) return 0;
 else return 1;
}

//===========================================================================
// FUNCTION:     message() copy constructor with string assignment
//
// DESCRIPTION:  Copy constructor used to create and initialize messages in
//               the following manner:  message msg = "0##11";
//
// I/O:           -->  str  -  The string to be assigned to the message.
//===========================================================================

message::message(const char *str)
{
 if (strlen(str) != length) error(1,"message::message(char*)");

 try { msg = new char[length+1]; }
 catch(xalloc) { error(0,"message::message(char*)"); }
 msg[length] = '\0';  // Ensure that the string will be null-terminated.
 strcpy(msg,str);
}

//===========================================================================
// FUNCTION:     operator=()
//
// DESCRIPTION:  Assignment operator overload.  This function is used to
//               assign strings to a message which has already been defined:
//                 message msg;
//                 msg = "0##11";
//
// I/O:           -->  str  -  The string to be assigned to the message.
//===========================================================================

void message::operator=(const char *str)
{
 if (strlen(str) != length) error(1,"message::operator=(char*)");

 strcpy(msg,str);
}

//===========================================================================
// FUNCTION:     message() copy constructor with message assignment
//
// DESCRIPTION:  Copy constructor used to create and initalize a message
//               with another message:
//                   message msg1 = msg2;
//               with msg2 defined elsewhere.
//
// I/O:           -->  msg2  -  The message to be assigned to newly-created
//                              message.
//===========================================================================

message::message(message & msg2)
{
 try { msg = new char[length+1]; }
 catch(xalloc) { error(0,"message::message(&)"); }
 msg[length] = '\0';
 strcpy(msg,msg2.msg);
```

```
}

//============================================================================
// FUNCTION:     operatore=()
//
// DESCRIPTION:  Overloaded assignment operator with message initialzation.
//               This fucntion assigns a message to another message which
//               as already been created:
//                   message msg1;
//                   msg2 = msg1;
//               with msg2 defined elsewhere.
//
// I/O:           -->  msg2  -  The message to be assigned to given message.
//============================================================================

void message::operator=(const message &msg2)
{
 strcpy(msg,msg2.msg);
}

//============================================================================
// FUNCTION:     msgToStr()
//
// DESCRIPTION:  Converts the given message to a string.
//
// I/O:          <-- string  -  the string version of the message.
//============================================================================

void message::msgToStr(char *string)
{
 for (int i=0; i<length; i++)
   string[i] = msg[i];

 string[i] = '\0';
}
```

Module queue.h

```
#ifdef NETWORK
#ifndef QUEUE_H
#define QUEUE_H

//=============================================================================
// The queueList structure is used to build a linked list of various data types.
// The embedded union uses pointers to the objects, since some of the structures
// that can be queued are several bytes long and require special contruction.
// By using pointers, the compiler allocates enough memory for a pointer,
// instead of enough memory for the largest object inside the union.  Since
// this queue is used for network message queuing, the source and destination
// ID's are included for every queueList object.  The queue class is
// responsible for linking and unlinking queueList objects as well as filling
// in object information.  Queue objects are removed in a FIFO fashion,
// either in data type order or in generic queueList object order.
//
// Each message type contains a put() function which allows its elements to
// be set quickly.
//
// NOTE:  queue overflow is indicated by a return value of zero from the add()
//        functions in the queue class.
//=============================================================================

#define DEF_MAX_QUEUE_LENGTH    100

/*
   The action message type is used to send action messages over the network.
   Supplier and bid information are also sent, since these quantities must
   be posted on the message board.
*/

struct actionMessage
{
  message action;
  int supplier;
  unsigned long source;
  double bid;

  void put(const message & act, int sup, unsigned long src, double b)
  { action = act;  supplier = sup;  source = src;  bid = b; }
};

/*
   The classifier message type is used to send classifiers over the network.
   The condition, action, and strength are the only information sent.
*/

struct classifierMessage
{
  message condition;
  message action;
  double strength;

  void put(const message & cond, const message & act, double stren)
  { condition = cond; action = act; strength = stren; }
};
```

```
/*
   The payoff message type is used to send BBA payoff messages over the
   network.  The rule to be paid (supplier) is given, as well as the payoff
   amount.  The queueList object holds the agent's destination ID.
*/

struct payoffMessage
{
  int supplier;
  double payoff;

  void put(int sup, double pay)
  { supplier = sup; payoff = pay; }
};

/*
   The generic node type used in the linked list.  The node automatically
   deletes the data stored in it when the node itself is deleted.
*/

struct queueList
{
  enum { noType, actMsgType, classMsgType, payoffMsgType };

  int type;              // one of the types in the enum above

  unsigned long source;  // Sending agent ID
  unsigned long dest;    // Receiving agent ID

  union
  {
    actionMessage     *actMsg;
    classifierMessage *classMsg;
    payoffMessage     *payoffMsg;
  };

  queueList *next;

  queueList() { type = noType; }
  queueList(int objectType);
  ~queueList()
  { if (type == actMsgType) delete actMsg;
    else if (type == classMsgType) delete classMsg;
    else if (type == payoffMsgType) delete payoffMsg;
  }
};

/*
    Manages the linked list.
*/

class queue
{
     int count,              // The total number of items in the queue
         maxLength;          // The maximum number of items allowed
     queueList *top,         // Top of the linked list
               *current;     // Most recent entry into the list

     int addNode(int type);
     void removeNode(queueList *item);
     queueList * findFirst(int type);
     void allocateAll();
     void deleteAll();

   public:

     queue() { reset(1); }
     ~queue() { deleteAll(); }
     void reset(int firstReset = 0);
     int countType(int type);
```

```
    void save(FILE *fptr);
    void load(FILE *fptr);
    int add(const queueList & qMsg);
    int add(unsigned long src, unsigned long dst, const actionMessage & qMsg);
    int add(unsigned long src, unsigned long dst, const classifierMessage & qMsg);
    int add(unsigned long src, unsigned long dst, const payoffMessage & qMsg);
    int remove(queueList & qMsg);
    int remove(actionMessage & qMsg);
    int remove(classifierMessage & qMsg);
    int remove(payoffMessage & qMsg);
};

#endif // ifndef QUEUE_H
#endif // ifdef NETWORK
```

Module queue.cpp

```
#include"cfs.h"

#ifdef NETWORK

//============================================================================
// FUNCTION:     queueList()
//
// DESCRIPTION:  The constructor for a generic queue object.  Allocates space
//               based on the type passed to the constructor.
//
// I/O:          --> objectType  -  the type of queue object (see queueList
//                                  class)
//============================================================================

queueList::queueList(int objectType)
{
  type = objectType;
  if (type == actMsgType)
  {
    try { actMsg = new actionMessage; }
    catch(xalloc) { error(0,"queueList::queueList()"); }
  }
  else if (type == classMsgType)
  {
    try { classMsg = new classifierMessage; }
    catch(xalloc) { error(0,"queueList::queueList()"); }
  }
  else
  {
    try { payoffMsg = new payoffMessage; }
    catch(xalloc) { error(0,"queueList::queueList()"); }
  }
}

//============================================================================
// FUNCTION:     deleteAll()
//
// DESCRIPTION:  Deletes all the nodes in the queue, including the top.  Used
//               in the queue destructor and in loading a new queue.
//
// I/O:          none
//============================================================================

void queue::deleteAll()
{
  queueList *search = top->next,
            *next;

  // Delete the top node
  delete top;

  // Delete the sub nodes
  for (int i=0; i<count; i++)
  {
    next = search->next;
    delete search;
    search = next;
  }
}
```

```
//============================================================================
// FUNCTION:     allocateAll()
//
// DESCRIPTION:  Allocates a top node (of noType).  Reset the counter.  Used
//               in the constructor and in loading a new queue.
//
// I/O:          none
//============================================================================

void queue::allocateAll()
{
  try { top = current = new queueList; }
  catch(xalloc) { error(0,"queue::allocateAll()"); }

  count = 0;
}

//============================================================================
// FUNCTION:     reset()
//
// DESCRIPTION:  Clears the queue and prepares it for use.
//
// I/O:          --> firstReset  -  (optional) used by constructor to
//                                  indicating the first time the queue has
//                                  been instantiated.
//============================================================================

void queue::reset(int firstReset)
{
  if (!firstReset)
    deleteAll();

  maxLength = DEF_MAX_QUEUE_LENGTH;

  allocateAll();
}

//============================================================================
// FUNCTION:     countType()
//
// DESCRIPTION:  Counts the number of messages of a particular type in the
//               queue.
//
// I/O:          --> type  -  The type to count.
//               <--       -  The number of this type in the queue.
//============================================================================

int queue::countType(int type)
{
  queueList *search = top->next;
  int i=0, typeCount=0;

  while(i<count)
  {
    if (search->type == type) typeCount++;
    i++;
    search=search->next;
  }

  return typeCount;
}
```

```
//============================================================================
// FUNCTION:     findFirst()
//
// DESCRIPTION:  Returns a pointer to the first occurence of a particular
//               type.  Used for FIFO extraction by type.  Returns zero if
//               none of the given type are found.
//
// I/O:          --> type  -  The type to search for.
//               <--       -  A pointer to the first occurence of the type.
//============================================================================

queueList * queue::findFirst(int type)
{
  queueList *search = top->next;
  int i=0;

  while(i<count)
  {
    if (search->type == type) return search;
    i++;
    search=search->next;
  }

  return 0;
}

//============================================================================
// FUNCTION:     addNode()
//
// DESCRIPTION:  Used to add a node of the given type to the queue.  The
//               current pointer is set to this newly-added node.
//
// I/O:          --> type  -  The type of node to add.
//               <--       -  Returns one on success, zero on failure
//============================================================================

int queue::addNode(int type)
{
  if (count == maxLength) return 0;

  try { current = current->next = new queueList(type); }
  catch(xalloc) { error(0,"queue::addNode()"); }

  ++count;

  return 1;
}

//============================================================================
// FUNCTION:     removeNode()
//
// DESCRIPTION:  Private function called by the remove() functions.  This
//               function assumes that the item passed to it is a valid item.
//               This function unlinks and deletes the node.
//
// I/O:          --> item  -  pointer to the node to be removed
//============================================================================

void queue::removeNode(queueList *item)
{
  if (item == top) return;

  queueList *search = top->next,
            *prev = top;

  // Find the item to be deleted
  while(search != item) { prev = search;  search = search->next; }

  // Move the current pointer if item is the last in the list
  if (current == search) current = prev;
```

```
  // Hook previous and next objects together
  prev->next = search->next;

  // Decrement the item counter
  --count;

  // Delete the item
  delete item;
}

//============================================================================
// FUNCTION:     add()
//
// DESCRIPTION:  Accepts a reference to a queueList object, and copies that
//               object to the queue.  This function is used when adding
//               generic objects to the queue.  Usually these objects were
//               removed with the following remove() function.  Normally this
//               is only used to simulate network operation.  (See task.cpp)
//
// I/O:          --> qMsg  -  generic message object to be added.
//               <--       -  Returns one on success, zero on failure
//============================================================================

int queue::add(const queueList & qMsg)
{
  if (!addNode(qMsg.type)) return 0;

  current->source = qMsg.source;
  current->dest = qMsg.dest;

  if (qMsg.type == qMsg.actMsgType)
    current->actMsg->put(qMsg.actMsg->action, qMsg.actMsg->supplier,
                         qMsg.actMsg->source, qMsg.actMsg->bid);
  else if (qMsg.type == qMsg.classMsgType)
    current->classMsg->put(qMsg.classMsg->condition, qMsg.classMsg->action,
                           qMsg.classMsg->strength);
  else
    current->payoffMsg->put(qMsg.payoffMsg->supplier, qMsg.payoffMsg->payoff);

  return 1;
}

//============================================================================
// FUNCTION:     remove()
//
// DESCRIPTION:  Accepts a queueList reference which will be allocated to the
//               appropriate type based on the current object being removed.
//               The first object in the queue is copied into qMsg and then
//               deleted.  This function is used for network simulation.
//               (see task.cpp)
//
// I/O:          <-- qMsg  -  The queueList object in which to store the
//                            object removed from the queue.
//               <--       -  returns a one if an object was removed from the
//                            queue and a zero if not.
//============================================================================

int queue::remove(queueList & qMsg)
{
  if (current == top) return 0;

  // get the first item in the queue
  queueList *item = top->next;

  // delete the previous type allocation
  if (qMsg.type != qMsg.noType)
  {
    if (qMsg.type == qMsg.actMsgType) delete qMsg.actMsg;
    else if (qMsg.type == qMsg.classMsgType) delete qMsg.classMsg;
```

```
    else delete qMsg.payoffMsg;
  }

  // Set the new type.  Copy source and destination.
  qMsg.type = item->type;
  qMsg.source = item->source;
  qMsg.dest = item->dest;

  // Based on the new type, allocate space for the object, save the object
  // info.
  if (item->type == top->actMsgType)
  {
    try { qMsg.actMsg = new actionMessage; }
    catch(xalloc) { error(0,"queue::remove(queueList)"); }
    qMsg.actMsg->put(item->actMsg->action, item->actMsg->supplier,
                     item->actMsg->source, item->actMsg->bid);
  }
  else if (item->type == top->classMsgType)
  {
    try { qMsg.classMsg = new classifierMessage; }
    catch(xalloc) { error(0,"queue::remove(queueList)"); }
    qMsg.classMsg->put(item->classMsg->condition, item->classMsg->action,
                       item->classMsg->strength);
  }
  else
  {
    try { qMsg.payoffMsg = new payoffMessage; }
    catch(xalloc) { error(0,"queue::remove(queueList)"); }
    qMsg.payoffMsg->put(item->payoffMsg->supplier, item->payoffMsg->payoff);
  }

  // Delete the current node from the queue
  removeNode(item);

  return 1;
}

//=============================================================================
// FUNCTION:     add()
//
// DESCRIPTION:  Adds an action message to the queue.
//
// I/O:          --> src   -  The sending agent
//               --> dst   -  The receiving agent (usually 0 indicating a broadcast)
//               --> qMsg  -  The action message to add.
//               <--       -  Returns one on success, zero on failure
//=============================================================================

int queue::add(unsigned long src, unsigned long dst, const actionMessage & qMsg)
{
  if (!addNode(top->actMsgType)) return 0;

  current->source = src;
  current->dest = dst;
  current->actMsg->put(qMsg.action, qMsg.supplier, qMsg.source, qMsg.bid);

  return 1;
}

//=============================================================================
// FUNCTION:     remove()
//
// DESCRIPTION:  Removes an action message from the queue.
//
//
// I/O:          <-- qMsg  -  The queueList object in which to store the
//                            action message removed from the queue.
//               <--       -  returns a one if an object was removed from the
//                            queue and a zero if not.
//=============================================================================
```

```
int queue::remove(actionMessage & qMsg)
{
  queueList *list = findFirst(top->actMsgType);
  if (!list) return 0;

  qMsg.put(list->actMsg->action, list->actMsg->supplier, list->actMsg->source,
           list->actMsg->bid);

  removeNode(list);

  return 1;
}

//============================================================================
// FUNCTION:     add()
//
// DESCRIPTION:  Adds a classifier message to the queue.
//
// I/O:          --> src   -  The sending agent
//               --> dst   -  The receiving agent (usually 0 indicating a broadcast)
//               --> qMsg  -  The classifier message to add.
//               <--       -  Returns one on success, zero on failure
//============================================================================

int queue::add(unsigned long src, unsigned long dst, const classifierMessage & qMsg)
{
  if (!addNode(top->classMsgType)) return 0;

  current->source = src;
  current->dest = dst;
  current->classMsg->put(qMsg.condition, qMsg.action, qMsg.strength);

  return 1;
}

//============================================================================
// FUNCTION:     remove()
//
// DESCRIPTION:  Removes a classifier message from the queue.
//
// I/O:          <-- qMsg  -  The queueList object in which to store the
//                            classifier message removed from the queue.
//               <--       -  returns a one if an object was removed from the
//                            queue and a zero if not.
//============================================================================

int queue::remove(classifierMessage & qMsg)
{
  queueList *list = findFirst(top->classMsgType);
  if (!list) return 0;

  qMsg.put(list->classMsg->condition, list->classMsg->action,
           list->classMsg->strength);

  removeNode(list);

  return 1;
}

//============================================================================
// FUNCTION:     add()
//
// DESCRIPTION:  Adds a BBA payoff message to the queue.
//
// I/O:          --> src   -  The sending agent
//               --> dst   -  The receiving agent (usually 0 indicating a broadcast)
//               --> qMsg  -  The BBA payoff message to add.
//               <--       -  Returns one on success, zero on failure
//============================================================================
```

```
int queue::add(unsigned long src, unsigned long dst, const payoffMessage & qMsg)
{
  if (!addNode(top->payoffMsgType)) return 0;

  current->source = src;
  current->dest = dst;
  current->payoffMsg->put(qMsg.supplier, qMsg.payoff);

  return 1;
}

//=============================================================================
// FUNCTION:     remove()
//
// DESCRIPTION:  Removes a BBA payoff message from the queue.
//
// I/O:          <-- qMsg  -  The queueList object in which to store the
//                            BBA payoff message removed from the queue.
//               <--       -  returns a one if an object was removed from the
//                            queue and a zero if not.
//=============================================================================

int queue::remove(payoffMessage & qMsg)
{
  queueList *list = findFirst(top->payoffMsgType);
  if (!list) return 0;

  qMsg.put(list->payoffMsg->supplier, list->payoffMsg->payoff);

  removeNode(list);

  return 1;
}

//=============================================================================
// FUNCTION:     save()
//
// DESCRIPTION:  Saves all the elements in the queue.  This function is called
//               by the classifierSystem class when a save is initiated.
//
// I/O:          --> fptr  -  A pointer to an open BINARY file set for writing.
//=============================================================================

void queue::save(FILE *fptr)
{
  fwrite(&count,sizeof(int),1,fptr);
  fwrite(&maxLength,sizeof(int),1,fptr);

  queueList *search = top->next;

  for (int i=0; i<count; i++)
  {
    fwrite(&search->type,sizeof(int),1,fptr);
    fwrite(&search->source,sizeof(unsigned long),1,fptr);
    fwrite(&search->dest,sizeof(unsigned long),1,fptr);

    if (search->type == search->actMsgType)
    {
      search->actMsg->action.save(fptr);
      fwrite(&search->actMsg->supplier,sizeof(int),1,fptr);
      fwrite(&search->actMsg->source,sizeof(unsigned long),1,fptr);
      fwrite(&search->actMsg->bid,sizeof(double),1,fptr);
    }
    else if (search->type == search->classMsgType)
    {
      search->classMsg->condition.save(fptr);
      search->classMsg->action.save(fptr);
      fwrite(&search->classMsg->strength,sizeof(double),1,fptr);
    }
```

```
    else
    {
      fwrite(&search->payoffMsg->supplier,sizeof(int),1,fptr);
      fwrite(&search->payoffMsg->payoff,sizeof(double),1,fptr);
    }

    search=search->next;
  }

}

//=============================================================================
// FUNCTION:     load()
//
// DESCRIPTION:  Loads elements in a queue from disk.  This function is called
//               by the classifierSystem class when a load is initiated.
//
// I/O:          --> fptr  -  A pointer to an open BINARY file set for reading.
//=============================================================================

void queue::load(FILE *fptr)
{
  deleteAll();     // Deletes the current queue
  allocateAll();   // Starts a new queue in memory

  int tempCount;

  fread(&tempCount,sizeof(int),1,fptr);
  fread(&maxLength,sizeof(int),1,fptr);

  for (int i=0; i<tempCount; i++)
  {
    int type;
    unsigned long source, dest;

    fread(&type,sizeof(int),1,fptr);
    fread(&source,sizeof(unsigned long),1,fptr);
    fread(&dest,sizeof(unsigned long),1,fptr);

    if (type == top->actMsgType)
    {
      actionMessage actMsg;
      actMsg.action.load(fptr);
      fread(&actMsg.supplier,sizeof(int),1,fptr);
      fread(&actMsg.source,sizeof(unsigned long),1,fptr);
      fread(&actMsg.bid,sizeof(double),1,fptr);

      add(source,dest,actMsg);
    }
    else if (type == top->classMsgType)
    {
      classifierMessage classMsg;
      classMsg.condition.load(fptr);
      classMsg.action.load(fptr);
      fread(&classMsg.strength,sizeof(double),1,fptr);

      add(source,dest,classMsg);
    }
    else
    {
      payoffMessage payoffMsg;
      fread(&payoffMsg.supplier,sizeof(int),1,fptr);
      fread(&payoffMsg.payoff,sizeof(double),1,fptr);

      add(source,dest,payoffMsg);
    }
  }
}

#endif
```

References

[1] Belew, R.K., Forrest, S., "Learning and Programming in Classifier Systems" in *Machine Learning Vol. 3*, Boston: Kluwer Academic Publishers, 1988.

[2] Bertsekas, D.P., Gallager, R.G., *Data Networks, 2nd edition*, New Jersery: Prentice-Hall, Inc., 1992.

[3] Bond, A.H., Gasser, Les, *A Subject-Indexed Bibliography of Distributed Artificial Intelligence*, IEEE Transactions on Systems, Man, and Cybernetics, vol. 22, no. 6, November/December 1992.

[4] Booker, L.B., Goldberg, D.E., Holland, J.H., "Classifier Systems and Genetic Algorithms," *Artificial Intelligence*, vol. 40, nos. 1-3, September 1989, pp. 235-282.

[5] Carse, B., "Learning Anticipatory Behaviour Using a Delayed Action Classifier System," AISB Workshop on Evolutionary Computing, Leeds, England, April 1994.

[6] DeJong, K., "Learning with Genetic Algorithms: An Overview," in *Machine Learning: An Artificial Intelligence Approach Vol. 3*, California: Morgan Kaufmann Publishers, Inc., 1990.

[7] Dorigo, M., Schnepf, U., "Genetics-Based Machine Learning and Behavior-Based Robotics: A New Synthesis," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 23, no. 1, January/February 1993, pp. 141-153.

[8] Dougherty, E.R., Giardina, C.R., *Mathematical Methods for Artificial Intelligence and Autonomous Systems*, New Jersey: Prentice Hall, 1988.

[9] Franks, N.R., "Army Ants: A Collective Intelligence," *American Scientist*, vol. 77, March-April, 1989, pp. 139-145.

[10] Holland, J.H., "A Mathematical Framework for Studying Learning in Classifier Systems," *Physica*, 1986, pp. 307-317.

[11] Holland, J.H., *etal., Induction: Processes of Inference, Learning, and Discovery*, Cambridge: MIT Press, 1986.

[12] Huhns, M.N., *Distributed Artificial Intelligence*, California: Morgan Kaufmann, 1987.

[13] Levine, L.L., *Biology of the Gene, third edition*, St. Louis: The C.V. Mosby Company, 1980.

[14] Lippmann, R.P., "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine*, April 1987, pp. 4-22.

[15] Melsa, P.J.W., "Neural Networks: A Conceptual Overview," Tellabs Research Center Document 89-08, August 1989.

[16] Michalski, R.S., "Understanding the Nature of Learning: Issues and Research Directions," in *Machine Learning: An Artificial Intelligence Approach Vol. 2*, Boston: Kluwer Academic Publishers, 1987.

[17] Porter, L.L., II, Passino, K.M., "Genetic Model Reference Adaptive Control," *IEEE International Symposium on Intelligent Control*, 16-18 August 1994, Columbus, Ohio, pp. 219-224.

[18] Rich, E., Knight, K., *Artificial Intelligence, second edition*, New York: McGraw-Hill, 1991.

[19] Robertson, G.G., Riolo, R.L., "A Tale of Two Classifier Systems" in *Machine Learning Vol. 3*, Boston: Kluwer Academic Publishers, 1988.

[20] Schneirla, T.C., *Army Ants: A Study in Social Organization*, San Francisco: W.H. Freeman and Company, 1971.

[21] Shu, L., Schaeffer, J., "VCS: Variable Classifier Systems," *Proceedings of the 3rd International Conference on Genetic Algorithms*, June 1989, pp. 334-339.

[22] Souza, P., Talukdar, S., "Insects, Fish, and Computer-Based Super Agents," *The Robotics Institute 1992 Annual Research Review*, Pittsburgh, PA.

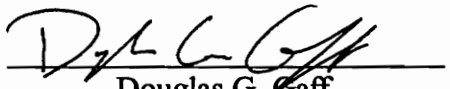
[23] Tang, A.Y., "Constructing Classifier Systems using TOLKIEN," Tutorial document from the Department of Computer Science, The Chinese University of Hong Kong, 10 April 1994.

[24] Wilson, S.W., "Classifier Systems and the Animat Problem" in *Machine Learning Vol. 2*, Boston: Kluwer Academic Publishers, 1987.

[25] Zhou, H.H., Grefenstette, J.J., "Learning by Analogy in Genetic Classifier Systems," *Proceedings of the 3rd International Conference on Genetic Algorithms*, June 1989, pp. 291-297.

Vita

Douglas Gene Gaff was born in Rolling Meadows, Illinois, a suburb of Chicago, on August 5, 1971. He expressed an interest in Electronics at a very young age, and decided to become an Electrical Engineer after reading about the occupation in the Career Center at Northside High School. He received his Bachelor of Science degree in Electrical Engineering in May of 1993 at Virginia Tech and went on to pursue a Master of Science degree in the same field. His extracurricular interests include mountain biking, photography, piano, camping, hiking, racquetball, golf, volleyball, and just about anything else that allows him to get away from a computer screen for a while.



Douglas G. Gaff