A DISTRIBUTED Q-LEARNING CLASSIFIER SYSTEM FOR TASK DECOMPOSITION IN REAL ROBOT LEARNING PROBLEMS

by

Kevin L. Chapman

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, Chairman

A. Lynn Abbott

Hugh F. VanLandingham

September, 1996

Blacksburg, Virginia

Keywords: Artificial Intelligence, Learning Classifier Systems, Q-learning, Mobile Robots, Task Decomposition

A DISTRIBUTED Q-LEARNING CLASSIFIER SYSTEM FOR TASK DECOMPOSITION IN REAL ROBOT LEARNING PROBLEMS

by

Kevin L. Chapman

John S. Bay, Chairman

Bradley Department of Electrical Engineering

(ABSTRACT)

A distributed reinforcement-learning system is designed and implemented on a mobile robot for the study of complex task decomposition in real robot learning environments. The Distributed Q-learning Classifier System (DQLCS) is evolved from the standard Learning Classifier System (LCS) proposed by J.H. Holland. Two of the limitations of the standard LCS are its monolithic nature and its complex apportionment of credit scheme, the bucket brigade algorithm (BBA). The DQLCS addresses both of these problems as well as the inherent difficulties faced by learning systems operating in real environments.

We introduce Q-learning as the apportionment of credit component of the DQLCS, and we develop a distributed learning architecture to facilitate complex task decomposition. Based upon dynamic programming, the Q-learning update equation is derived and its advantages over the complex BBA are discussed. The distributed architecture is implemented to provide for faster learning by allowing the system to effectively decrease the size of the problem space it must explore.

Holistic and monolithic shaping approaches are used to distribute reward among the learning modules of the DQLCS in a variety of real robot learning experiments. The results of these experiments support the DQLCS as a useful reinforcement learning paradigm and suggest future areas of study in distributed learning systems.

Acknowledgment

I would first like to express my thanks to my advisor, Dr. John Bay, for allowing me to work on this project. I would like to especially thank him for allowing (demanding) that my project involve both robotics and artificial intelligence. My understanding of and appreciation for the connection between AI theory and application in robotics has been immeasurably increased during the last year. I would like to thank Dr. Lynn Abbott for serving on my committee and for serving as my interim advisor during my first semester of graduate school. He has been very instrumental in sparking my interest in the fields of computer vision and artificial intelligence. I would also like to thank Dr. Hugh VanLandingham for being a member of my committee. Although I was never able to take a class taught by Dr. V, I appreciate his support of my research.

I owe a great deal of thanks to my friends in the Machine Intelligence Lab, Cem, Paul, John, and Mamun. Their input was appreciated more than they can know. Remember, Cem and Paul, graduate school is part of the journey, NOT the destination. Although its level of intelligence is debatable, I'd like to acknowledge Curly, the three-wheeled artificially intelligent MIL inhabitant with whom I've spent the past year.

Next, I'd like to thank my family for their unwavering support during the last six years. Thanks especially for not ragging me too much for the low frequency of my visits. I would like to dedicate this effort to my little niece, Carley Paige Chapman, for allowing all of us to look at the world with new eyes.

And finally, I'd like to thank all of my friends from South Carolina (wherever they may currently reside) for making the last six years the best of my life. I'm already looking forward to the next "La Fiesta Mas Grande", tailgating, biking the "YEE-HA Trail", and any other reason to find my way back home from DC. I have one question for them: Which way to Group?

This thesis was supported by the Naval Research Laboratory and Office of Naval Research under grant nos. N00014-93-1-G022 and N00014-94-1-0676.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

1. Introduction

Artificial Intelligence (AI) has historically been defined in many different ways. One popular definition comes from the Rich and Knight textbook entitled *Artificial Intelligence*. Here, AI is defined as "the study of how to make computers do things which, at the moment, people do better" [23]. This very broad definition is appropriate for a science whose paradigms shift as frequently as do those in AI. As the focus of artificial intelligence has continually changed, however, the field of *robotics* has maintained a vital role. In fact, the research dedicated to AI gave birth to the field of robotics. Given a robot that has the ability to sense its environment, artificial intelligence techniques are employed to solve the problems of how to represent domain knowledge (in particular, sensory data), how to locate and distinguish important details from the environment that impact the problem at hand, how to use domain knowledge for problem solving, and how to choose an action based on knowledge of the robot's current position in the environment [18]. One definition of robotics that demonstrates its importance as a tool of AI was given by Michael Brady: "Robotics is the intelligent connection of perception to action" [5].

Although ideas of intelligent robot servants and fears of humans evolving into robots dominated the early AI pop culture, most robots in use today fall far short of what we would like to call "intelligent". This shortcoming can be easily traced back to the types of problems early AI researchers attempted to solve. Early AI problems were abstract problems in perfect, contrived domains used to study one particular concept. Robotics, on the other hand, involves practical problems in an imperfect physical world [18]. Therefore, the problem solving approaches created by early AI researchers proved themselves to be almost exclusively non-portable to real world robot systems. Even in the area of reinforcement learning, much of the research still involves simulated robots. There are many time and cost advantages to using simulated robots for investigating robot

learning. But again, relying on simulated systems often leads to the development of algorithms that are not transferable to real robots. Mataric characterized the pitfalls of simulated robots in [6]. According to Mataric, the use of global information that could not be available to real robots often makes simulations "doomed to succeed." Also, the nondeterminism of the real world does not allow it to be represented correctly by sensor models.

Recently, a branch of AI called robot learning has become popular. Robot learning has been applied to the exact problem that early AI research avoided, namely the construction of a real, intelligent robot systems that *learn* specified tasks. The general theme in robot learning is that an intelligent machine is one that can sense its environment, learn how to cause change in its environment to achieve a goal, form plans to carry out tasks, and react to unpredicted external stimuli. If constructed, true learning systems would relieve their creators of the difficult (perhaps impossible) task of programming it with all of the domain knowledge it would need to perform its specified task. Again the notions of robot workers have arisen, but this time the approach is different. Using robot learning techniques, self-improving robots would learn how to integrate the disparate abilities of perception, planning, learning, and action using only a learning mechanism and feedback from the environment in which they would be operating [8].

The blanket area of robot learning, or machine learning, may be divided into the areas of *supervised* and *unsupervised learning*. In supervised learning, a training sequence is provided by a teacher, and the robot is passively guided through the task. A more interesting and challenging problem is for a robot to learn a task without the supervision of a teacher. In unsupervised learning, the robot, or *agent*, is given the ability to explore its environment in a trial-and-error fashion to collect data. From an evaluation of this data, the robot must learn a mapping from its input sensor values to its output effector actions. In essence, the robot learns about its own abilities. We may further characterize the type of learning to be discussed in this thesis as *reinforcement learning*,

learning that involves the use of feedback to reason about the quality of condition-action pairs.

We have selected one unsupervised reinforcement learning algorithm for study and implementation in this thesis, the *Learning Classifier System (LCS).* Developed by Holland [4,14,15], the LCS is a rule based, message passing machine learning paradigm that incorporates *planning* and *rule discovery* for intelligent problem solving in a dynamic environment. While the system implemented in this thesis resembles Holland's original LCS in structure, several additions and substitutions are included. One significant area of study is the *apportionment of credit* (AOC) mechanism in Holland's LCS, the *bucket brigade algorithm.* After discussing the limitations of this algorithm, we offer Watkins's *Q-learning* algorithm [29] as a replacement. To aid in the implementation of this system on a real robot, we also examine an enhancement of the original LCS, the *Distributed Learning Classifier System* (*DLCS*) [12]. The system that we actually implement is a union of the two. It has the configuration variability of the DLCS, but it uses Watkins's Q-Learning mechanism for apportionment of credit. This system is called the *Distributed Q-learning Classifier System* (*DQLCS*).

In this thesis, we examine the implementation of the DQLCS on a real mobile robot. We then use the robot to solve a classical robotics/AI problem. The problem we have selected for study is one in which the robot must learn to navigate from an initial position to a goal position using sensor inputs to determine the state of the environment (Figure 1.1). In our system, the goal position is marked by the location of an electronic beacon. From an initialized state, the objective of the learning system is to learn to coordinate its sensor inputs and action choices to find the best path to this goal position. We will refer to this task as a "docking" or a "goal seeking" task throughout this paper. The nature of this application will be discussed in greater detail in Chapter 6 when the performance of the DQLCS is discussed.

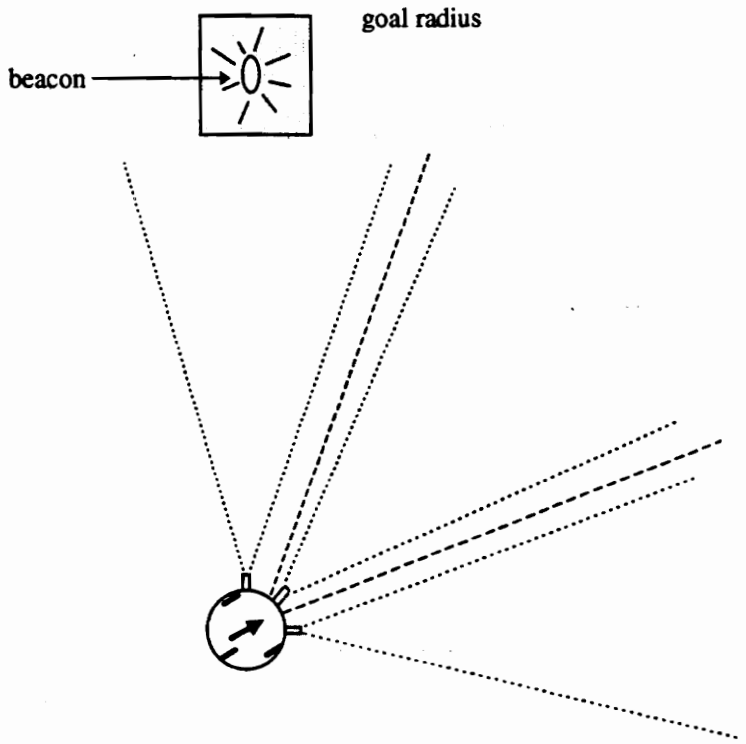


Figure 1.1. The goal seeking problem environment.

While the implementation of the DQLCS on a real robot is new, the concept of distributing control processes to enhance robot learning is well-known as *task decomposition*. Although the topics reinforcement learning and task decomposition have been researched, many questions still remain about various *shaping* techniques, the schemes for combining reinforcement learning and distributed control on the same system. From our experiments, we comment on the effectiveness of the DQLCS at decomposing and solving robot learning problems. We also compare and contrast the performances of two reinforcement distribution techniques, *holistic shaping* and *modular shaping,* in the DQLCS architecture. During the implementation of this hardware/software system, a number of topics that span the entire spectra of robotics and AI are covered. In doing so,

we obtain a better understanding of the difficult task of making that intelligent connection of perception to action to which Brady was referring [5].

The thesis begins with an analysis of the problem of learning in real robots. It is of great importance that we set the stage for the thesis as a hardware application of a software system; therefore, we wish first to document the factors influencing the performance of such a system and clearly establish the need for a learning system. This chapter also includes a discussion of *task decomposition*, the distribution of the learning process into parallel learning units. Task decomposition effectively decreases the size of the state space search involved in learning with the goal of increasing learning speed in real robots. Next, in Chapter 3, we introduce Holland's original LCS structure. While all of the components of the original LCS are covered, those components that are later replaced are covered in less detail than the others. In Chapter 4, we discuss in detail the problems involved with the bucket brigade algorithm AOC component of the LCS. Then we introduce Q-Learning as a viable alternative. We show that Q-Learning follows directly from the principle of *dynamic programming*. We then provide a detailed mathematical derivation of its update equations and a proof of its convergence to an optimal policy. We also introduce the concept of the Distributed Q-learning Classifier System (DQLCS), and we show how it lends itself to use in task decomposition problems. In Chapter 5, we discuss the implementation of the DQLCS on a real robot. This includes both hardware and software descriptions. We then show in Chapter 6 the performance of various configurations of the DQLCS robot system in the goal seeking problem, and we offer some reasoning for the results obtained. Finally, we present some avenues for further research in the area of robot learning.

2. The Learning Problem for Real Robots

2.1 Overview

Certain types problems lend themselves to solution by learning robots rather than robots that only have only *a priori*, or preprogrammed, knowledge of the environment. Problems involving "hard to program" knowledge are among these [9]. Instead of calculating necessary arm trajectories and effector positions, it is much more desirable to simply show the robot how to perform a task[1]. A second type of problem suited for learning robots is any problem where there is unknown environmental information. In such situations, learning robots may create a map of the environment through exploration. A very attractive class of problems that necessitates an adaptive robot are problems that involve changing environments. In dynamic environments, any *a priori* map of the world could soon become obsolete. Therefore, it would be essential that the robot be able to continually update its knowledge of the environment. This is an attractive problem for research because it closely matches the challenges facing humans as we live in an ever-changing world.

This chapter presents the internal and external (to the robot) factors that influence the effectiveness of robots at learning to solve problems.

2.2 Challenges Facing Real Learning Robots

As we mentioned in the Chapter 1, many classical AI methods were found to be inadequate at solving problems involving real robots. Even in the area of robot learning,

[1] Throughout this paper, the following terms will be used interchangeably: task, goal, and behavior.

these same problems must be considered during the design and implementation of a learning robot system.

Sensor noise: Many low-cost sensors, such as sonar, are somewhat unreliable. Unpredictable reflections of sound waves often cause objects to be missed or distances to be inaccurately measured. These errors then introduce inaccuracies in the state descriptions used by the learning system for action selection. Some sort of probabilistic averaging is required to lessen the effect of erroneous sensor data.

Nondeterministic Environment: Since the robot's model of its environment is incomplete, the same action may not always have similar effects. This makes the planning process more difficult because the robot now has to consider the possibility that the sequence of actions to be executed may lead to some state, not a goal state, where more planning will be required.

Reactivity: The robot must be able to react to unforeseen changes in the environment. This means that the robot must have some selection mechanism to decide a course of action given an environmental condition it has never seen before. Because the robot is operating in a real time, dynamic environment, such a mechanism must often react *quickly* to the environment changes.

Incrementality: Because the learning robot must obtain all of its domain knowledge through its interaction with its environment, it must be able to efficiently explore all of the various states of the environment. Incremental learning algorithms allow the robot to be better able to decide which environmental states it needs to explore next.

Limited training time: While learning trials often run into the thousands of time steps for simulated robots, this is totally impractical for real robots that face such basic concerns as battery power consumption. An efficient

learning algorithm for a real robot must converge to the desired solution quickly.

Groundedness: The learning algorithm must be able to use sensor data as its only knowledge of the environment. The problems posed for such a robot must also take the inexact nature of its environmental knowledge into account. For example, a learning navigating robot would almost assuredly perform poorly in a problem where it had to determine its exact coordinates on a map. In other words, learning robot problems must be posed with the accuracy of the robot's environmental interface in mind.

Generalization: While the knowledge obtained by a robot needs to be specific enough to solve the problem at hand, this knowledge should be applicable in other environments and other learning problems that have similar characteristics.

2.3 The Learning Problem

Now that we have looked at some of the difficulties facing learning robots, we can characterize the robot learning problem. Generally speaking, the robot learning problem is to determine a mapping from sensors to actions through the analysis of sensory inputs, action outputs, and environmental feedback [9]. These learning sets may be provided by a teacher (*supervised learning*), or more interestingly, they may be obtained through environmental exploration (*unsupervised learning*).

Unsupervised learning is the area of study in this thesis. In unsupervised learning, the robot must be able to determine when it is performing a task correctly or when it is achieving a goal. The task is more challenging than the task of supervised learning because the robot first solves the task by executing a trial-and-error approach to explore the state space of the problem. Then, once all of the environmental data are collected, the

robot evaluates the results of its actions and creates the sensor-action mapping discussed earlier.

2.3.1 Feedback

In both supervised and unsupervised robot learning, an essential element is *feedback*. Feedback is input used by the learning system to grade its own performance. There are several types of feedback that can be used in robot learning: *scalar feedback*, *control feedback*, and *analytic feedback*. Scalar feedback is a reward signal that tells the robot how well it is doing a task. Generally, rewards are positive values and punishments are negative. Control feedback consists of the action(s) the robot is supposed to perform, while analytic feedback consists of an explanation of why a given action sequence will or will not achieve the desired goal. Both of these latter methods require teacher supervision and a large amount of domain knowledge. The learning system implemented in this thesis receives scalar feedback only.

Systems that use scalar feedback can be further categorized based on the frequency at which rewards are received. Some systems receive reward only upon reaching a goal state. These must learn through trial and error exploration of the problem space and back-propagation of the received rewards [26]. Other systems receive feedback only when the robot fails at its task. For learning robots, more informative feedback comes upon the completion of intermediate tasks prior to reaching the goal. The most informative type of feedback occurs when the robot is given scalar feedback during every system time step. This feedback allows an evaluation of the quality of the last action taken in each time step. In the system implemented in this thesis, a combination of these last two scalar feedback approaches is studied. The robot interprets sensor feedback as reward or punishment

during every time step. There is also a specialized punishment feedback given if the robot commits a fatal error[2] at any point during a learning experiment.

2.3.2 Credit Assignment

Once the type of feedback system has been chosen, a method of evaluating this environmental feedback must be selected. Recognized as the crux of the robot learning problem, the *credit assignment problem* is the task of continually evaluating and updating the fitnesses of the rules in the robot's knowledge base. The credit assignment problem can be divided into two parts: *temporal credit assignment* and *structural credit assignment*. Temporal credit assignment refers to the task determining which action or set of actions were most responsible for the received reward. Structural credit assignment, on the other hand, involves the process of interpreting the distribution of rewards across the state space. In structural credit assignment, the question asked is the following: Given a *different* starting state and the *same* sequence of actions, what is the likelihood that the outcome will be similar? While not all feedback systems require the use of both types of credit assignment, the weak nature of the data in scalar feedback systems presents both problems. Scalar environmental feedback provides no indication of the amount of reward the robot will receive from being in a state even slightly different from the current state. Also, rewards for reaching goals are awarded infrequently in most problems. Therefore, the full temporal credit assignment problem must be solved for the distribution of the goal rewards. The credit assignment problem in the learning classifier system is discussed in great depth in Chapter 4, where Q-Learning is introduced as a replacement for the bucket brigade algorithm credit assignment scheme.

[2] A fatal error is any error that would require the activation of the emergency stop to avoid damage to the robot. A typical error of this sort is a collision with a wall.

2.4 Task Decomposition

To date, all research on reinforcement learning involving real robots has focused on fairly simple problems. Simple problems are appropriate to study the basic concepts of these learning systems, but their solution does not satisfy our expectations for the capabilities of "intelligent systems". The true test of the power of reinforcement learning techniques is then to apply them to larger and more complex problems. A very robust learning system would be able to scale to more complex problems and would contain knowledge that generalizes enough to allow the system to quickly adapt to different environments.

While it is easy to formulate reinforcement learning problems that involve complex or *multiple goal* behavior [30], it is less straightforward to efficiently find their solutions. When faced with one of these problems, the usual first approach is to apply one of the previously established monolithic reinforcement learning techniques to the problem directly. One must only consider that the problem's state space grows exponentially with the number of goals to realize that such an approach is doomed to poor performance. While it may learn the task, a monolithic system suffers from slow learning due to the large size of the state space created by multiple goal tasks. This explosion in state space size is called the *curse of dimensionality*. If we ask the question "What is a complex behavior?", one possible answer might be that it is a combination of simple behaviors or actions. *Task decomposition* is an approach to the problem of complex task learning in which the overall task is divided into its constituent pieces. After this decomposition, each individual task is given a control module whose objective is to learn only to achieve that task. Then, instead of learning over a single state space whose size is exponential in the number of tasks, the modular system learns over a linear number of constant sized state spaces. The capability of a system to then learn to coordinate these multiple behaviors, or *policies*, is sometimes called *dynamic policy merging* [30].

Under the assumption that a complex behavior consists of a group of simple behaviors, we may further classify that complex behavior based on the interaction of these simple behaviors. In some environments, a complex behavior may consist of a temporal sequence of simple behaviors. In other situations, the complex behavior might be the result of several simple behaviors acting in parallel. While the distinction may not appear to be significant, it greatly affects the decomposition approach used to learn the behavior.

2.4.1 Sequentially Complex Behaviors

One approach to learning multiple goal behavior is to divide the complex behavior into a sequence of simple behaviors. Each behavior in this arrangement is learned by a learning module that activates when its characteristic behavior is required. The operation of this system is equivalent to "exchanging" dedicated monolithic control modules in the robot whenever a new behavior is desired. It is arguable that, in itself, the robot's behavior is no more complex than the simple behavior that has control of the robot at that time. The difficulty in this system lies in the scheduling of these simple behaviors. While it may be possible to predefine some complex behaviors as a sequence of tasks, it is far more interesting to allow the system to learn the proper arrangement of these tasks based on the state of the environment.

There are several approaches to the task scheduling problem. One method is to teach a reinforcement learning algorithm, such as a neural network, to activate the necessary task module [17]. Another approach is to include task descriptors in the definition of the state space [25]. This essentially adds another dimension to the system's input conditions. This approach also uses a scheduler to learn how to group and arrange simple tasks to form complex ones. Still another approach to this scheduling problem is to employ a subsumption architecture to create a priority-based layering of task modules [19]. These approaches are all discussed in greater detail in the literature survey on task decomposition (Section 2.5).

All of the learning scheduling systems are very different and interesting applications of reinforcement learning methods. Unfortunately, they all suffer from the restrictions imposed by this approach to the construction of complex behaviors. This approach assumes that each simple task has an associated distinct goal that can be used in the apportionment of credit to the learning module while it is controlling the system. Most importantly, though, it assumes that there is no need for any of the low level behaviors to occur in parallel. In the modeling of very complex natural systems, this is a huge assumption that may cause insurmountable problems.

2.4.2 Parallel Complex Behaviors

Parallel complex behaviors present different problems than sequential behaviors. During each time step of a parallel behavior, each module updates the fitnesses of the rules in its knowledge base according to the module's reinforcement algorithm. In principle, it is assumed that a reward *vector* is generated at each time step, where each of the vector's component is associated with one subgoal. Thus, when the reward vector is generated, each reward component is routed directly its learning module for internal apportionment. The overall control of a modular system belongs to a separate module called the *arbiter*. The arbiter is responsible for creating the reward vectors from the environmental input and then distributing the appropriate values to the learning modules [30].

While this reward scheme is very appealing, it makes some critical assumptions about the nature of learning problem. It assumes that each simple learning behavior is easily distinguishable from all of the other behaviors. This system also assumes that there is a straightforward method for dividing the reward among separate modules. The truth is that this is not always the case. We will revisit this problem during the discussion of reward schemes for the distributed Q-learning classifier system in Section 4.2.3.4.

2.5 Current Learning Research

Although reinforcement learning is a fairly young research area, learning classifier systems, Q-learning, and task decomposition have been the subjects of study in a number of real and simulated learning robot applications. Before proceeding to detailed descriptions of the learning classifier system, Q-learning and their application to task decomposition techniques in real learning robot problems, we briefly describe some of the past and present research in these areas.

Ming Tan provides an early implementation of Watkins's Q-learning reinforcement algorithm in [28]. This work focuses on the reactivity and adaptivity of a Q-learning system as a *cost-sensitive* learning method. While Tan's work involves simulated robots only, it is demonstrated that Q-learning is a viable reinforcement technique in environments consisting of non-negligible delays (costs) introduced by the mechanical limitations on sensors. While these findings are encouraging for mobile robot applications, at least one of the assumptions behind these findings is not. A key assumption in the system is that all sensors are noiseless, a completely impractical assumption for real robots.

Another early implementation of Q-learning in a simulated robot learning problem is provided by Sutton in [26]. In Sutton's Dyna-Q learning system, Q-learning is used in combination with a learned world model to generate hypothetical experience and to achieve planning. Along with this augmented planning approach, Sutton's Q-learning system introduces the *exploration bonus* to encourage the system to explore the entire state space of the problem. Although Sutton notes the improved performance of the Dyna-Q system over some previous systems, he notes the limitations of such a monolithic system when faced with learning problems with larger state spaces. He also notes that the quality of the system's performance is dependent on its explicit knowledge of the state of the environment.

The application of Q-learning in simulated robot domains is revisited by Long-Ji Lin in [17]. In an effort to combat the problem of slow learning rates in Q-learning systems with large state spaces, he proposes a technique called *experience replay* and an approach in which unsupervised and supervised learning can be combined. According to Lin, experience replay allows for the reuse of sometimes rare experiences (rewards) to speed up learning. After introducing his additions to the normal Q-learning system, Lin examines task decomposition of complex behaviors into hierarchical learning modules. Because of the difficulty and time constraints of implementing such a system on a real robot, he opts to use a simulated system. In addition, he speculates that Q-learning's back-propagation of credit requirement is inefficient for such a problem compared to a connectionist implementation. Therefore, he implements his task decomposition scheme using neural networks to learn each elemental task. Although he offers an integrated learning system, the presence of these neural networks requires teaching sessions.

Whitehead, Karlsson, and Tenenberg formulate the problem of dynamic policy merging of decomposed tasks in [30]. In their simulations of a multiple-goal robot world, they decompose the complex goal into a set of learning modules that use Q-learning reinforcement techniques. Each of the sub-goals is given a time dependent *goal activation* function that encodes the goal's relative importance at any time t. An arbiter divides the reward for the robot's actions based on the activation values for each of the learning modules. From their simulations they compare the performance of this modular architecture to that of a monolithic architecture and a hybrid architecture that is part monolithic and part modular. They conclude that the modular architecture is a very robust system for robot learning problems. While they propose an interesting scheduling system for complex behaviors and offer a method for dividing reward among parallel modules, their model of the problem is too abstract to be applicable to real learning robots, and their division of rewards requires a significant amount of domain knowledge.

In Singh's work on the decomposition of complex tasks into sets of actions [25], he takes a slightly different approach. While he also implements his system using Q-

learning for reinforcement, he focuses on dividing up complex tasks into sequential, not parallel, sets of actions. Singh introduces the concept of *compositional Q-learning* as a means of constructing Q-value functions of composite tasks from the Q-values of their elemental tasks. The interesting property of Singh's system is its ability to learn to distinguish between elemental tasks that contribute to a complex goal and elemental tasks that do not. His system is limited, however, in the sense that it does not have the ability to form complex actions from parallel elemental actions. It is also uncertain how this decomposition scheme would work on a real robot where goals are ambiguously defined by sensor readings instead of simulated positional knowledge.

Whereas the two previously cited instances of the application of task decomposition were restricted to simulated systems, Mahadevan and Connell [19] offer an implementation of multiple goal learning on a real mobile robot. This learning system is like Singh's system in that it divides a problem into a sequential set of tasks that are learned using Q-learning modules. It differs from the previous work significantly, however, in the area of task scheduling. To achieve a complex behavior, the system uses a subsumption architecture to schedule its tasks. This system, whose task is to learn to push boxes (there are three different elemental behaviors: finding, pushing, and unwedging), shows great improvement over the performance of a monolithic system with the same complex task. One observation that we find important to our mobile robot application is the observation that the limitations of the ranging sensors contributed to relatively poor system performance.

Task decomposition in real robot problems is also being studied by Dorigo [11]. In his work, complex behaviors are decomposed into parallel learning modules in a hierarchical system. Each learning module in his decomposition scheme is an improved version of the original learning classifier system that used the bucket brigade algorithm AOC scheme. Among the improvements introduced by Dorigo is an automatically activating genetic algorithm that triggers when the bucket brigade reaches a steady state, a *mutespec* genetic operator whose task is to eliminate over general classifiers, and the

ability to dynamically change the rule base of the system to remove "bad" classifiers. In his experiments, Dorigo introduces and compares modular and holistic shaping, two methods for distributing reward among parallel learning modules. His work is discussed in greater detail in Section 4.2.3.4, where we compare shaping methods in the DQLCS.

Of particular interest in this thesis is the work of Bay and Stanhope in [2]. In their research, the DQLCS is proposed and implemented in a simulation of tactical behaviors of learning robots. While their work supports the viability of the DQLCS in a simulated environment, a physical implementation is not achieved. The distributed architectures they present are explained in greater detail in Section 4.2.2 and Section 4.2.3. In this thesis, we examine the performance a monolithic Q-learning classifier system and versions of the DQLCS as proposed by Bay and Stanhope [2] in real learning robot applications.

3. The Learning Classifier System

3.1 LCS Overview

As originally introduced by Holland, *et al.* in [4,14,15], the learning classifier system (LCS) is an adaptive, message-passing, rule based learning system. The structure of the LCS is shown in Figure 3.1. In Figure 3.1, boxes represent the various components of the LCS, while the arrows represent the flow on data through the system. The operation of the system is based upon the use of lists of evolved production rules or *classifiers*. Classifiers are basically "if-then" statements, where the "if" part is called the *condition*, and the "then" part is called the *action*. Each classifier is composed of sequences of bits called *alleles*. Found in genetics, the term "allele" is defined as one of a group genes that may occur at a given locus on a chromosome. The genetic makeup (sequences of alleles) of the chromosomes thereby defines traits of, or "classify," the organism to which the chromosomes belong. In general, alleles in the LCS are all members of the set {0, 1, #}[1]. An example classifier is shown in Figure 3.2. As a composition of sequences of the three alleles, each classifier defines a possible state of the machine's environment. As will be seen when the structure of the LCS is discussed in greater detail, the LCS is essentially a group of well-defined modules. The modularity of the system and the text string nature of its rules lend the system to straightforward implementation in any structured programming language.

In Figure 3.2, we may also see that external to the structure of the classifier is the classifier's associated individual *strength* or *fitness* value *F*. The value is of great importance to the learning system. The strength of a classifier is related to its current usefulness to the system as compared to all of the other classifiers in the system. The goal

[1] The "#" symbol acts as a binary "don't care".

of the LCS is to adjust the strength of all of the classifiers over time until the classifiers most useful in achieving the desired goal are distinguishable from the rest. The means by which the strengths of the classifiers are updated is called *credit assignment*. The credit assignment algorithm of the Holland's LCS encourages the formation of sequences of actions, or *chains*. Just as in the general learning systems described in Section 2.3.3, credit assignment is recognized as the key to the success of the LCS. Credit assignment is discussed in great detail in Section 3.6 and Section 4.1.

The operation of the learning classifier system involves a number of steps that, when grouped sequentially, are called the *execution cycle*. Figure 3.3 shows the LCS execution cycle at a high level. The execution cycle is discussed in intricate detail in Section 3.8. The LCS is a discrete-time system, e.g. a system whose execution consists of regular *time steps* or *clock ticks*. In the LCS, a clock tick is the length of one execution cycle, and the execution cycle is repeated indefinitely or until the predefined goal state is achieved.

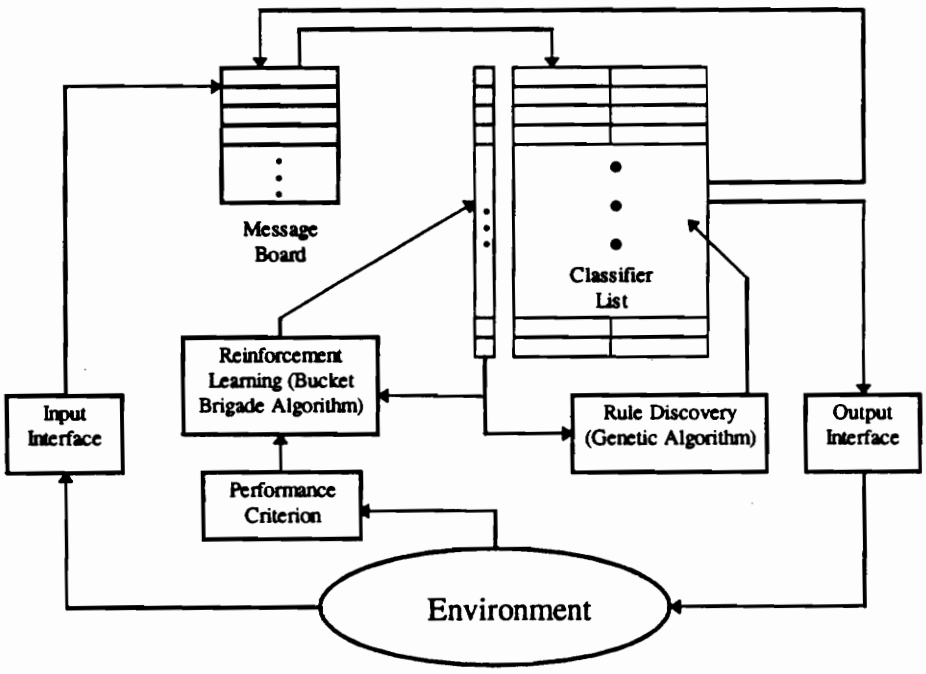


Figure 3.1. The Learning Classifier System (LCS) structure.

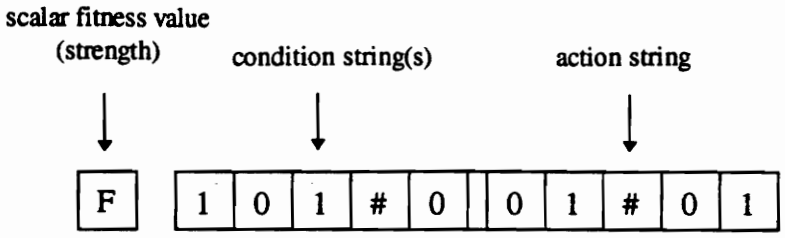


Figure 3.2. Classifier structure: condition(s), action, and strength.

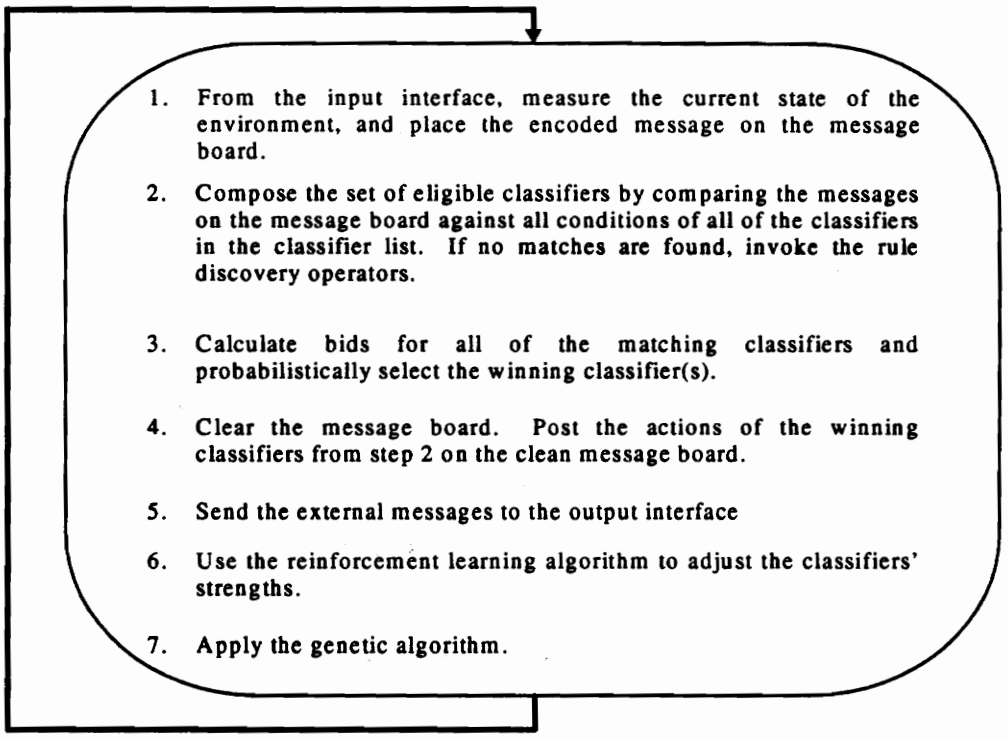


Figure 3.3. The LCS execution cycle.

3.2 The Input (Detector) Interface

The input interface is one of the two parts of the LCS used for interaction with the environment of the LCS application. Although some features of the LCS have properties dependent on the nature of the application, the content of input and output interfaces

(Section 3.3) are **totally** dependent on the application. In the typical application, the input interface receives as input sensor values of possibly varying sizes from one or more environment sensors. It then encodes these values into a message and posts the message on the message board.

3.3 The Output (Effector) Interface

The output interface is similar to the input interface in that its structure is absolutely dependent on the machine being used in the learning problem. The output interface provides the means for applying the action(s) of the winning classifier(s). The action part of a classifier is fed as input into the output interface. The effector interface maps (decodes) this bit string to one or more commands, then it sends the command(s) to the machine for execution.

3.4 The Message Board

The *message board* is essentially a bulletin board upon which messages describing the current state of the system are "posted" by rules that have been activated or by the input interface. The required size of the message board is dependent upon the nature of the application. If we determine that the message board may have at most q messages, we may describe a full message board M at clock tick t as:

$$M(t) = \{M_1(t), \ldots, M_q(t)\} \tag{3.1}$$

Corresponding to each message is a supplier number L that indicates the *number*[2] of the classifier that posted the message.

3.5 The Classifier List

The classifier list is the rule base for the LCS. As with the message board, the length of the classifier list may vary widely from one application to another. Depending on the nature of the application, each classifier may have multiple conditions that must all be met before the its action becomes eligible to be selected. The action part of the classifier is an *internal* or *external* message. Internal messages are used to activate other rules while external rules are encoded actuator commands for the machine being controlled by the system. If the LCS has both internal and external messages, special bits called *tags* are used to identify the destination of the action part of the classifier. For the case of classifiers of length l consisting of elements from the set {0, 1, #}, there are obviously 3^l possible distinct classifiers. For some applications, this number may be manageably small. For other problems, maintaining a list of *all* classifiers is unrealistic given memory availability on the host computer and execution speed requirements associated with the problem. Therefore, the classifier list is often limited to length n:

$$C(t) = \{C_1(t), \ldots, C_n(t)\} \tag{3.2}$$

The "#" symbol serves a special purpose in the operation of the system. A "#" in the condition substring of a classifier is equivalent to a "don't care" in binary representation. This means that it will successfully match either a "1" or a "0" when compared to an input condition string. For example, the two classifiers 0100111101 and 010##11101 are successful matches due to the presence of the don't care allele in the

[2] The position of the classifier in the classifier list. Messages that come from the detector interface have a supplier number of zero.

fourth and fifth bits of the second word. The occurrence of the "#" symbol in an action substring carries a different meaning. Here it acts as a "pass-through" operator. This means that if an action string containing one or more pass-through operators is selected, the positions containing the pass-through operators are filled with the corresponding symbols from the message that matched the condition string of the classifier.

Paired with each classifier in the classifier list is a strength value. The strength of a classifier $S(t)$, as mentioned above, is a measure of the its usefulness to the LCS compared to all of the other classifiers in the classifier list given the current environment. Credit assignment (Section 3.6) is used to adjust the strengths of the classifiers based on the positive or negative effects on the state of the system resulting from their use.

Elements of the classifier list are accessed during steps 2, 3, and 4 of the LCS execution cycle.

3.6 Credit Assignment

One of the necessary ingredients of a learning system is a means to evaluate its rules. The system must determine the rules responsible for its successes. In general, the rules in a system are of varying usefulness. Some are even incorrect. The purpose of the *credit assignment* or *apportionment of credit (AOC)* component of the LCS is to adjust the strengths of the classifiers according to their usefulness to the system. This is recognized as a very difficult task because most of the rules simply "set the stage" for success or failure at some future time step. The problem is often compounded by a changing environment and the limited sampling of the environment. The LCS AOC algorithm has three components: *rule taxes*, the *bucket brigade algorithm* (BBA), and the *environmental payoff function*.

3.6.1 Rule Taxes

Rule taxes are penalties applied in various ways to the rules of the LCS. Their purpose is population control. Three different taxes may be incurred: a *head tax*, a *bid tax*, and a *producer tax*. Head taxes are incurred by every rule during every execution cycle. Rules that are non-productive do not receive reinforcement from the system; therefore, they are gradually weakened by head taxes until their strength becomes low enough that they susceptible to replacement by the rule discovery component of the system (Section 3.7). Bid taxes are applied only to rules that are eligible to bid during step 3 of the execution cycle. Very general rules are often allowed to bid because they match a broad range of messages. They seldom win the bidding selection, however, due to their low specificity. Therefore, they are eventually killed off by bid taxes. The producer tax is a loosely defined incremental tax that is applied to the winners of the bidding process. The tax on a given classifier increases each time it wins the bid process with the goal of deterring useless chaining of rules.

3.6.2 The Bucket Brigade Algorithm

The Bucket-Brigade Algorithm is based upon an economic model in which rules bid for the right to have their actions posted on the message board. The algorithm treats the rule as a "middleman" in a complex economy. The rule only interacts with its *suppliers* and its *consumers*. A rule's suppliers are the rules that post messages satisfying its conditions, while its consumers are the rules whose conditions are matched when it is allowed to post its action message. Whenever a rule is wins the bidding competition and posts its action, it must pay out part of a fraction of its strength to its suppliers. Since the rule's action is now posted, it may directly or indirectly activate more rules during the next clock tick. In the rule's message is internal, one or more classifiers may have conditions matching that message during the next clock tick. If it is external, it may be beneficial in

changing the state of the environment. Then, more rules may be matched by the new environment condition string that is supplied by the input interface during the next clock tick, In this way the rule is a supplier to its consumers by giving them the right to enter the bidding competition during the next execution cycle. In return, the rule's consumers will pay it for the right to enter the bidding competition, just as the rule pays its suppliers in the current execution cycle. If the rule pays less to its suppliers in the current clock tick than it receives from its consumers in the next, it turns a profit, that is, its strength value undergoes a net increase.

Generally a rule is profitable only if its consumers also enjoy profitable transaction with their consumers. Eventually, one of the consumers of a rule will reach the system goal. The rule directly responsible for the reaching of a goal state receive a special bonus determined by the *environment payoff function.* But if this rule lies at the end of a sequence of rules that are always executed to move the system to its goal state, the reward is propagated backwards through the rule chain through the transactions discussed above. Therefore, all of the "stage setting" rules are eventually rewarded for their part in achieving the goal. This is where the theoretical performance and actual performance of the BBA begin to differ. Robertson and Riolo [24] found that in general, good classifiers that are only "stage setting" classifiers in rule chains have less strength than good classifiers that drive effectors and receive environmental payoff directly. Because the genetic algorithm component of the LCS replaces rules with low strength, these good stage setting rules are often eliminated.

If, on the other hand, a rule is coupled into a chain of rules that leads to no goal, the final rule in that sequence continually loses strength. As with the goal payoff, this net loss is also back-propagated through the rule chain. This causes the rule sequence to deteriorate over time. Eventually, the strength of some rule in the chain becomes low enough that it loses the bidding process. Then some new competing rule gains the chance to reform the rule sequence. While the chaining of classifiers seems logical theoretically,

research has failed to produce this emergence of rule chains from initially random sets of rules [12].

An interesting restriction is placed on the LCS's classifiers as a result of the coupling effect: the condition and action parts **must** be the same size. The classifier may have multiple conditions, but they also must all be the same size as the action string. As we will show later in the discussion of an alternative apportionment of credit algorithm (Section 4.1.4), this restriction is significant to the effectiveness of the LCS.

3.7 Rule Discovery

The second component of a learning system is *rule discovery*. Rule discovery provides the LCS the ability to adapt to rapid environmental changes. This is essential in real-time situations where the state space of the system is too large for every possible rule to be kept in the classifier list. The learning system must be able to generate a new rule "on the fly" to match the current environment state if no other rule in the rule list does so. Because classifier list space is often limited, the system must also adapt to its current environment by discarding useless rules and replacing them with better rules. Three algorithms are used to meet the rule discovery requirement of a learning system: *rule discovery operators*, *genetic algorithms*, and *triggered operators*.

Note that the requirement of a rule discovery component in the LCS hinges on the classifier list size limitation imposed by the programmer. If the state space of the selected problem is not very large, it is feasible to maintain a classifier list containing *all* of the possible rules for the system. Because of the limited environmental sampling capabilities of the robot used in this thesis and the nature of the problems chosen for exploration, all of the necessary rules are maintained in the classifier list and the rule discovery component of the LCS is not needed for the problems chosen. We choose to discuss the rule discovery mechanisms briefly because they are all parts of the original LCS structure and are

important in most classifier systems. All three mechanisms are described in greater detail in [24] and [4,14,15].

3.7.1 Rule Discovery Operators

Rule discovery operators are used in step 2 of the LCS execution cycle if no rule from the classifier list matches any of the messages on the message board. There are two types of rule discovery operators: the *cover detector operator* (CDO), and the *cover effector operator* (CEO). The CDO is executed when no matching classifiers for a message can be found. In this case, a new classifier that has a condition part that matches the message and random action bits is created. If there are matching classifiers, but none have action parts that cause one of the machine's effectors to be activated (if all of the action parts are tagged as internal messages), the CEO is executed. This operator creates a new classifier with the same condition as a message, an external action tag, and a random action string.

3.7.2 Genetic Algorithms

Genetic algorithms are used during step 7 of the execution cycle of the LCS to create new rules for the classifier list. As previously mentioned, the length of the classifier list is limited to a finite number of classifiers; therefore, adding new classifiers often requires the system to "kill off" old ones. In the spirit of evolution, classifiers chosen to be removed from the classifier list are weak rules, e.g., they have a low strength value. Because strength measures the usefulness of a classifier to the LCS, it is logical to target "useless" classifiers for replacement. New rules created by the genetic algorithm are each the "offspring" of two "parent" rules that have proven their usefulness to the LCS and therefore have high strength values. Randomness is introduced to the "offspring" classifier through *genetic operators* such as *mutation* and *crossover*. The main operator is

crossover, which exchanges a randomly selected substring between the parent pairs. For a full discussion of genetic operators, see [31].

The execution cycle of the genetic algorithm fairly simple, although its effects on the nature of the learning system are subtle:

Step 1: Select pairs of classifiers according to strength from the classifier list—the higher the strength of a classifier, the more likely it is to be chosen.

Step 2: Apply genetic operators to the selected pairs, creating "offspring" classifiers.

Step 3: Replace the weakest classifiers with the newly constructed classifiers.

3.7.3 Triggered Operators

The triggered chaining operator (TCO) is used to enforce chain forming among classifiers. TCO is activated during the credit assignment step, step 4, under two conditions. First, a classifier C_2 must have been rewarded with a profit from the BBA. Second, there must have been a classifier C_1 active prior the activation of C_2 that was not already coupled to it. When activated, TCO generates two new classifiers that are coupled.

3.8 The LCS Execution Cycle

Now that all of the individual components of the LCS have been defined, we can observe their interaction in the LCS execution cycle.

During the first step of the execution cycle, the input message, or condition, is read from the input interface and posted on the message board. The input interface receives a

current sample of the machines sensory data as input. It then converts the sensory data into the condition string format as described in Section 3.2 and shown in Figure 3.2. The message board consists of a list of all internal and external messages generated during the current execution cycle. During any clock tick, the contents of the message board may be interpreted as the current state of the system.

In step 2, the messages on the message board are compared to the condition strings of all of the classifiers in the classifier list. If **all** of the condition parts of a given classifier are matches by one or more messages, the classifier is become eligible to bid for the right to post its action message on the message board. The set of eligible classifiers is called $E(t)$. Two special cases must be considered during step 2: 1) there may be more classifiers eligible to post actions on the message board than there are positions on the message board; and 2) there may be no eligible classifiers to post actions on the message board. If there are too many eligible classifiers, a subset of them must be selected. The designated size of the message board is the upper limit on the number of classifiers eligible to post. The rest of the execution cycle is dependent an eligible classifier posting its action part on the message board. Therefore, if the classifier list contains no eligible classifier for the current message board, one must be constructed by the cover detector and cover effector operators (Section 3.7).

Because there may often be more eligible classifiers than there are open positions on the message board, the LCS must select the classifier(s) that will be allowed to post their actions. The bidding process is used to make a probabilistic selection of winning classifiers in step 3. Bids are computed for each of the eligible classifiers using the following formula:

$$B_i(t) = bP_i(t)S_i(t-1)\big(1+U_i(t)\big), \tag{3.3}$$

where b is a constant much less than one, $P(t)$ is *specificity* of the rule, $S_i(t\text{-}1)$ is the strength of the classifier during the previous clock tick[3], and $U(t)$ is the support.

Specificity reflects the amount of information required by the rule. The specificity value for a classifier is calculated as the following:

$$P(t) = \frac{\text{Total number of specifying bits (non-\#s) in the condition}}{l \cdot k} \tag{3.4}$$

Recall that l is the length of the condition part of the classifier, and k is the number of condition words in a classifier. It can easily be seen that the specificity of a classifier will always be between zero and one, with rules containing few don't-cares having a higher specificity than those with many don't-cares. The use of specificity in the bidding process encourages the selection of rules that are best tuned (high specificity) to the condition being matched.

The support parameter in (3.3) is included to encourage chaining among the classifiers. Support is computed as simply the sum of all of the bids of the winning classifiers from the previous clock tick.

According to the probabilistic selection process, the classifier with the highest bid has the highest probability of being chosen. There are several different rule selection algorithms in use. An example is the *roulette-wheel*. Given the set of eligible classifiers $E(t)$ and the set of their computed bids $B(t)$, the probability of eligible rule i winning the bidding process is

$$p(i) = \frac{B_i(t)}{\sum_{j \in E(t)} B_j(t)} \tag{3.5}$$

[3] We use $S_i(t\text{-}1)$ instead of $S_i(t)$ because the strength of the classifier will be updates during the apportionment of credit stage (step 6) of the current execution cycle.

There is an important reason for not simply choosing the strongest rules during the selection process. By purposely selecting poor rules or even by selecting randomly, the system encourages a thorough inspection of the entire state space of the problem. This is especially necessary in a rapidly changing environment. In this case a rule that may have been proven totally useless could become a very good rule. If the system only chooses rules that were proven good during earlier clock ticks, this rule might not ever be selected.

After the selection process of step 3, the q winning classifiers, designated $\hat{C}(t)$, are allowed to post their actions to the clean message board in step 4. Then, in step 5, all of the messages are activated. The nature of the action part determines the course of its execution. The action may be tagged as either an internal message or an environmental message. If the message is tagged "internal," it is left on the message board for the next execution cycle, where it will be compared to classifiers in the classifier list during step 2. If the message is tagged "environmental," it is removed from the message board and passed to the output interface. The output interface then translates the bit string into one or more commands that are executed by the machine's effectors. The positions on the message board vacated by the action substrings in the current clock tick are replaced by effector condition strings in the next. Remember, however, that in the next time step the message board will still contain any internal message substrings from the current clock tick. Therefore, during any given time step, the LCS may contain information about both the current and previous time steps. This is the message-passing facility of the LCS.

The first of the learning components of the LCS is encountered in step 6 of the execution cycle. In this step the strengths of the classifiers are updated through the credit assignment mechanism. Credit assignment consists of three components: the environmental payoff function, the bucket brigade algorithm, and rule taxes. The environmental payoff function evaluates the results of the chosen action and computes the classifier's effectiveness based upon this evaluation. This evaluation is based upon comparing the sensory information after the executed action to the predefined goal information. Considering this evaluation, a reward is added to the strengths of the

classifiers with matching conditions and useful actions, while a penalty is deducted from those with matching conditions and poor actions. The second part of the credit assignment, the Bucket Brigade Algorithm (BBA), encourages planning by rewarding chains of classifiers. Rule taxes, on the other hand, deduct from the strengths of classifiers. These act as population control measures, since the continual weakening of unused and thus unrewarded classifiers eventually drives their strengths to zero.

Low strength classifiers are susceptible to being replaced by new classifiers during step 7, the final step in the LCS execution cycle. This step contains the second half of the learning process: rule discovery. In the LCS, rule discovery occurs through the use of the genetic algorithms (GA) (Section 3.7) [4, 24]. Genetic algorithms generate new classifiers using combinations of rules from the current gene-pool, the classifier list. This means of adaptation helps the LCS adjust to new environmental conditions that are not matched by any rules in the current rule base.

4. Evolution of the Learning Classifier System

This chapter describes the "evolution" of the Learning Classifier System (LCS) into the Distributed Q-learning Classifier System (DQLCS). This evolution is a two part process. In the first stage, the apportionment of credit scheme from the LCS is replaced with the Q-learning algorithm. The reasoning behind this switch and a detailed derivation of the Q-learning update equations are provided. To further detail the effect of adding Q-learning to the LCS, a description of all of the structural and operational differences between the LCS and the QLCS is provided in Section 4.1.4. The second part of the evolution of the LCS involves the introduction of *distribution* to the QLCS. In Section 4.2, the DQLCS is presented as means of decreasing the state space of a learning problem.

4.1 Apportionment of Credit Schemes

The essential difference between the LCS and the QLCS is the choice of apportionment of credit schemes. As described in Section 3.4, apportionment of credit is the problem of distributing received rewards across the set of rules most responsible for the robot achieving the goals for which the rewards were given. In Holland's original LCS, the AOC algorithm is the Bucket Brigade Algorithm (BBA). On the other hand, the QLCS uses the Q-learning machine learning algorithm as its apportionment of credit component.

4.1.1 The Bucket Brigade Algorithm Revisited

The apportionment of credit component in the LCS architecture as originally proposed by Holland is the bucket brigade algorithm (BBA). In theory, the BBA solves

the temporal credit assignment problem by encouraging the formation of rule chains. Experimental results have revealed, however, that the BBA is not effective in the iterative learning problems for which it was designed. Research has shown that the BBA in its original form could sustain pre-existing rule chains, but it is not strong enough to successfully encourage rule chaining from a rule list consisting of initially random strings [3, 12, 24]. The backwards propagation of rewards from the rules receiving the rewards to their supporters was observed to take many time steps. This slow learning process results in the need for a large number of initially long-running experiments before the first rules in the chain receive any reward. Because of the time requirements imposed by learning robot problems, we decided that the BBA is not sufficient for use in our system and that some other apportionment of credit algorithm is necessary.

4.1.2 Q-learning

In the Q-learning classifier system (QLCS) [2] the BBA is replaced by Q-learning [29]. Q-Learning is a reinforcement learning algorithm that been successfully used in a variety of simulated and real robot learning problems [17, 19, 24, 25, 28] but has not until recently seen use in classifier systems. In Q-learning systems, each state-action rule has an associated *Q-value*. The Q-value is simply an estimate of the minimum cost-to-go associated with taking the rule's action given when the state of the environment matches the rule's condition. After a rule is executed, the environmental feedback is used to update the rule's Q-value. Ultimately, the system reaches a *optimum policy*, a path of least total cost. Similar to dynamic programming, the Q-value for a state-action rule is the sum of two quantities: the cost of taking the rule's action in the current state and the minimum value of all future costs starting with the state reached through the execution of the action (minimum cost-to-go).

Q-learning originated as a recursive algorithm for solving decision problems in Markov processes. Markov processes contain a finite number of states, with each state

having a finite number of actions that can be taken. The probability of making a transition from one state to another is a function of the current state and not on any past history of the system. To derive the Q-learning algorithm, we will first introduce notation to describe decision making in Markov decision processes.

We will call the set of n finite states of in a Markov decision process $S = \{s_i\}$, i=1, 2, ... , n_s. At each state, there is a set of n_{as} possible actions $A_s = \{a_{sj}\}$, j = 1, 2, ..., n_{as}. Also, the probability of making a transition from state s_i to state s_j given the action a is $pr(s_i \rightarrow s_j) = p_{ij}(a)$. When an action $a(t) \in A_s$ is taken from state $s(t) \in S$ at time step t, the *new state function* $S(t + 1) = \mu(s(t), a(t))$ determines the resulting state. Each state-action rule has an associated value, $c_i(a)$, that represents the instantaneous cost or penalty incurred by taking action a in state s_i. This value is assumed to be either always positive or always negative. Because this cost is often a random variable, we must use the expected value operator: $E[c_i(a)] = \bar{c}_i(a)$.

We can now establish the value function $V_\mu(i)$, the expected sum of all future discounted costs provided that the system starts at state s_i and follows the state function μ:

$$V_\mu(i) = \lim_{n_s \to \infty} E\left[\sum_{t=0}^{n_s-1} \gamma^t c_{s_t}\left(\mu(s_t)\right) \middle| s_0 = s_i\right] \tag{4.1}$$

The summation includes all future states of the system, where s_t is the state of the system at time t and γ is the *discount factor*, $\gamma \in [0,1]$. By applying the discount factor, more immediate future costs are emphasized over distant future costs. The inclusion of this discount factor and the previously mentioned restriction on the sign of the cost function facilitate the convergence of the summation [29].

From (4.1), the system goal can be stated as follows: Find an optimal decision policy that minimizes the future expected cost:

$$V^*(s_i) = V_{\mu^*} = \min_{\mu} V_{\mu}(s_i), \tag{4.2}$$

where V^* is the *optimal value function* and μ^* is the corresponding *optimal policy.*

Bellman's principle of optimality [16] is used to find the decision policy sought by (4.2). Bellman's equation can be stated as the following:

$$V^*(s_i) = \min_{a \in A(i)} \left\{ \bar{c}_i(a) + \gamma \sum_{s_j \in S} p_{ij}(a) V^*(s_j) \right\}. \tag{4.3}$$

Basically, this is equation says that the minimum total cost from the current state to the goal is found by summing the minimum of the expected instantaneous costs for actions from the current state and the minimum cost of going to the goal from the resulting next state. Therefore, we see that by carrying the minimum cost function backward from the goal state one step at a time, we can find the minimum cost at *any* state by calculating one decision at a time.

While this principle works in theory, it is not applicable to most problems since the minimum cost-to-go from the next state to the goal state is usually not known. If we reformulate (4.3) as a recurrence relation, however, the costs-to-go may be estimated over repeated trials. Value iteration, a recursive estimation equation of the form

$$V^{(k+1)}(s_i) = \min_{a \in A(i)} \left\{ \bar{c}_i(a) + \gamma \sum_{s_j \in S} p_{ij}(a) V^k(s_j) \right\}, \tag{4.4}$$

has been shown to converge to the optimal value policy $V^*(s_i)$ for a given initial estimate $V^0(s_i)$. That is, if at the k^{th} iteration $V^*(s_i)$ is estimated as $V^k(s_i)$, then $V^{(k+1)}(s_i) \rightarrow V^*(s_i)$ as $k \rightarrow \infty$.

When Watkins [29] introduced Q-learning in 1989, he reformulated Bellman's equation as by adding the Q-value notation shown below:

$$Q^*(s_i,a) = \bar{c}_i(a) + \gamma \sum_{s_j \in S} p_{ij}(a) V^*(s_j) . \tag{4.5}$$

In this equation, $Q^*(s_i,a)$ is the Q-value associated with taking action a from state s_i. Equation (4.6) shows the simplification of Bellman's equation (4.3) resulting from the substitution of Watkins's Q-value notation:

$$V^*(s_i) = \min_{a \in A(i)} Q^*(s_i,a). \tag{4.6}$$

If we apply the value iteration technique to this simplified form of Bellman's equation, we reach the following results:

$$V^k(s_i) = \min_{a \in A(i)} Q^k(s_i,a) \tag{4.7}$$

and

$$V^{k+1}(s_i) = \min_{a \in A(i)} Q^{k+1}(s_i,a). \tag{4.8}$$

Now the recurrence relation (4.4) becomes

$$\min_{a \in A_{(i)}} Q^{(k+1)}(s_i,a) \leftarrow \min_{a \in A_{(i)}} \left\{ \bar{c}_i(a) + \gamma \sum_{s_j \in S} p_{ij}(a) \min_{a \in A_{(j)}} Q^k(s_j,a) \right\}. \tag{4.9}$$

But we've already noted that this converges to an optimal solution. Therefore, we can rewrite (4.9) as

$$Q^{(k+1)}(s_i,a) \leftarrow \bar{c}_i(a) + \gamma \sum_{s_j \in S} p_{ij}(a) \min_{a \in A(j)} Q^k(s_j,a). \tag{4.10}$$

The recurrence relation of (4.10) provides an estimate for the Q-value of the state-action pair (s_i, a) in terms of an expected instantaneous cost and a weighted sum of minimum costs-to-go for the state action pairs (s_j, a). But there are two problems in this equation: 1) we do not know the expected value of all instantaneous costs $\bar{c}_i(a)$ that could be incurred; and 2), the state transition probabilities, $p_{ij}(a)$, are unknown. To solve these problems, we approximate the unknown values as shown below:

$$\bar{c}_i(a) \approx c_i(a), \tag{4.11}$$

and

$$\sum_{s_j \in S} p_{ij}(a) V(s_j) \approx V(s_j). \tag{4.12}$$

Now, according to (4.11), we see that the estimate of the expected instantaneous penalty is just the single sample value of the actually incurred instantaneous penalty. We also see that the expected minimum cost-to-go is estimated as the minimum of the estimated costs-to-go at the next state.

We can now write the estimate of $Q(s_i,a)$ at the $(k+1)^{st}$ by substituting the estimations of (4.11) and (4.12) into (4.10):

$$Q^{(k+1)}(s_i,a) = c_i(a) + \gamma V^k(s_j). \tag{4.13}$$

We may now finally establish the Q-learning update equation. Let $Q_t(s_t,a_t)$ be the current estimated minimum cost for executing action a in state s at time t. After taking this action, we update our estimate of $Q_t(s_t,a_t)$ using (4.13). The old and new estimates are combined in the relaxed *Q-learning update equation* as expressed below:

$$Q_{t+1}(s_t,a_t) \leftarrow [1-\alpha(s_t,a_t)]Q_t(s_t,a_t)+\alpha(s_t,a_t)[c_{s_t}(a_t)+\gamma V_t(s_{t+1})]. \quad (4.14)$$

In (4.14), $\alpha(s_t, a_t)$ is a learning rate between 0 and 1. The value chosen for $\alpha(s_t,a_t)$ has significant implications. If, for instance, $\alpha = 1$, the old estimate, $Q_t(s_t,a_t)$, is completely replaced by the new estimate, $[c_{s_t}(a_t)+\gamma V_t(s_{t+1})]$. This means that the learner is throwing away all of its past history every time it revisits a state. A learning rate of close to one might be chosen in a volatile environment, as past knowledge is soon obsolete. On the other hand, if $\alpha = 0$, the new estimate is ignored and there is no change in the Q-value estimate. This is the equivalent of discarding all new knowledge of the environment and simply relying on the previous history. The learning rate thus provides a means to weight the effects of the Q-value's past estimation history and the new measurement. Also note that although α is treated as a constant throughout the remainder of this thesis, it may be time varying.

4.1.3 Q-Learning in the Learning Classifier System

Now that the theoretical basis for Q-learning has been established as a temporal credit assignment method, an algorithm must be developed for applying it in the LCS architecture in place of the bucket brigade algorithm. For this new AOC scheme, the Q-values associated with the state-action rules become the strength values upon which rule selection is based during the execution cycle. After replacing the BBA with Q-learning and rule strengths with Q-values, the QLCS execution cycle is as follows.

From an initialized set of Q-values, Q_0:

1. Observe the current state s of the system.
2. Compile a list of all eligible classifiers $E(s, t)$ – a classifier is eligible if its condition part matches the current world state s. If there are no matches, invoke the rule discovery operator.
3. Select a winning classifier from the list of eligible classifiers using a stochastic selection method.
4. Pass the action part a of the winning classifier to the output interface to be executed.
5. Advance the system clock: $t = t + 1$.
6. Receive an immediate cost $c(s,a)$ (based on evaluation of environmental feedback) for executing action a in state s at time t.
7. Examine the new message board $M(t)$.
8. Compute the new eligibility set $E(t)$ given the new environment state. Again, invoke the discovery operator if necessary.
9. Rank all of the classifiers in $E(t)$ based on their Q-values.
10. Update the Q-value of the classifier chosen during the previous clock tick.

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(c + \gamma V(y))$$

where

$$V(y) = \min_{b \in A_y} Q(y,b)$$

α = learning factor, $0 \leq \alpha \leq 1$

γ = discount factor, $0 \leq \gamma \leq 1$

11. Make a probabilistic selection of the classifier with the maximum (or minimum) Q-value.

12. Goto 4.

4.1.4 Comparison of Q-Learning and the Bucket Brigade Algorithm

As previously stated in Section 2.3.2, scalar feedback problems require solution of the temporal credit assignment problem. This problem can be solved in two ways: the reward may be appropriated to all of the state-action rules after it is received, or an expected value of future reward may be calculated and maintained incrementally [20]. This second approach is characteristic of a class of reinforcement algorithms called *temporal difference* (*TD*) methods. Temporal difference methods assign credit locally based on the difference between temporally successive predictions. Both the bucket brigade algorithm and Q-learning are instances of temporal difference methods. While they are alike in their approach to the AOC problem, they have widely varying requirements on the structure of the learning system. Since Q-learning has been selected to replace the bucket brigade algorithm in the LCS studied in this thesis, it is necessary to note the structural differences between the new QLCS and Holland's original LCS.

4.1.4.1 Classifiers

The largest differences between the BBA and Q-learning algorithms reside in the classifiers used by the systems. In the original LCS using the BBA, classifiers are fairly complex, highly structured rules. In Q-learning the rules have far less restrictions.

Recall that in the BBA, chaining of rules was encouraged through the use of internal message passing. An action message tagged as an internal message is left on the message board until the next time step, at which time it has the opportunity to match another rule. Q-learning has no provision for internal message passing. This essentially prevents the explicit formation of rule chains, since one classifier cannot post another classifier's condition part on the message board. The QLCS combats this by forming implicit rule chains. In each state of the environment, the system evolves "good"

classifiers that incur the least cost by moving the environment closer into a state that is closer to the goal state.

Recall also that the internal message passing capabilities of the LCS necessitates rules in which each condition part is the same length as the action part. The lack of internal message passing in Q-learning also removes this restriction. In Q-learning systems, the lengths of condition and action parts are in no way necessarily related. This is very important in the application of the QLCS to a real robot system. In the QLCS, the sizes of the condition strings is directly related to the number and resolution of the sensors used to sample the environment. The action string sizes are similarly related to the number of possible actions that may be taken by the robot. For example, a typical application of the QLCS may result in classifiers with five condition bits and three action bits, an impossible combination for the original LCS. A more detailed discussion of the mappings between sensors and condition string sizes and between action sets and action string sizes will be included as we describe the various QLCS configurations implemented in this thesis.

In theory, the Q-leaning algorithm has no provisions for the inclusion of the "don't care" symbols in classifier strings or for the specificity values they create for the classifier bidding process of the LCS. This implies that a classifier list must contain **every** possible state-action pair. In reality, the classifier list maintains a finite set of the "best" rules and a rule discovery algorithm such as the cover detector operator or the genetic algorithm (Section 3.5) is used for applications of the QLCS where the state space is too large for all of its classifiers to be maintained.

According to the derivation in Section 4.1.2, the convergence of Q-learning to an optimal policy can only be guaranteed after each state of the system is visited an infinite number of times. This property is, of course, totally unacceptable for learning robots. This property is inherited from dynamic programming, where it was labeled *the curse of dimensionality*. This problem is exactly described in Section 2.2. For systems with relatively small state spaces, however, convergence to a nearly optimal policy is often fast

enough for practical use. With this in mind, the obvious solution to the dimensionality problem is to limit the size of the problem space. In the case of a multi-sensor robot system, this would translate to using only low resolution sensors and a very small set of actions. Obviously, there is a point of diminishing returns, e.g., a point when the system will perform poorly not because of the slow convergence of the learning algorithm but instead because of the limited capabilities of the sensors. In Section 4.2, we introduce an addition to the QLCS structure that allows a large input set but avoids the curse of dimensionality, namely the *distributed* QLCS.

4.1.4.2 Message Board

Whereas the structure of the LCS incorporates a large message board for several internal and external messages, Q-learning provides for a message board that only contains one message during a given clock tick. Again, since Q-learning does not have all of the internal message passing of the BBA, it does not require all of the message board space needed by the BBA. But not all of the messages in the original LCS were internal. In fact, with the intervention of the CDO and CEO rule discovery operators (Section 3.5), it is guaranteed that at least one, if not more, actions is external during every clock tick. So does the QLCS suffer from not being able to post more than one external action during a clock tick? While there is no definite proof, simulation studies have indicated that there is little advantage to be gained from the large message boards of the original LCS [2].

4.1.4.3 Rewards vs. Penalties

The final aesthetic difference between the LCS and the QLCS relates back to environmental payoff function. During a given clock tick, the bucket brigade algorithm returns either positive or negative payoff based on the response from the environment. Alternatively, to show that the total accumulated cost converges in Q-learning, the payoff

must be either always positive or always negative. This restriction prevents any indefinite oscillation of the Q-values around the optimum policy values that may happen if rewards are allowed to be positive and negative. Since the payoff now has a sign restriction, a new way of determining "goodness" or "badness" of a classifier must be adopted. A simple solution to this problem is to simply reward "bad" rules less than "good" rules, then use Q-learning maximize total reward. An equivalent approach is to penalize "bad" classifiers more than "good" ones, and then to minimize total punishment through Q-Learning.

4.2 The Distributed Q-learning Classifier System (DQLCS)

Having given a thorough introduction to standard learning classifier system and the Q-learning classifier system, we are now ready to discuss the extension of these systems into the Distributed Q-learning Classifier System (DQLCS). Various distributed versions of the original Learning Classifier System (DLCS) [2, 11, 12] have been studied previously. Introduced in 1995, Doug Gaff's Distributed LCS (DLCS) architecture [12] is a "flat" architecture (Figure 4.1) designed to be utilized by a group of robotic agents trying to solve the same problem. The agents essentially share all of their rules, providing for a quicker search of the state space of the problem. While this "external" approach to the distribution of learning is very useful in many situations, it is not easily applicable to the problems we wish to examine here. In many problems that require the use of single real robots, the state space representation of the environment is too large for a monolithic learning system to explore in a feasible time period. This is our problem. A distributed architecture more suited to this type of problem was introduced by Dorigo [11]. His architecture allows for the distribution of *internal* LCSs for the study of learning problems involving real robots. Variations of both of these architectures were presented by Bay and Stanhope [2]. These architectures were referred to as the *Peer* architecture and the *Division of Labor* (*DOL*) architecture. The Peer architecture is very similar to the architecture presented by Gaff, except that it is modified for application on a single robotic

agent instead of a group of robots. In this architecture, all of the classifier systems are fundamentally equal. They gather information separately and compete to solve a single problem. The second architecture is called the DOL architecture because of its organizational properties. In the DOL learning system, separate modules focus on solving independent parts of a learning problem.

In this section, we examine the proposed distributed LCS architectures. For our implementation, we use only the DOL architecture; therefore, the other two architectures are covered in less detail. After presenting the architectures, we relate the DOL architecture to the concept of task decomposition discussed in Section 2.4. Our selection of the DOL architecture over the Peer architecture for implementation is justified later in Section 4.2.3 and in Chapter 5.

4.2.1 Gaff's DLCS Architecture

As previously mentioned and as shown in Figure 4.1, Gaff's DLCS architecture consists of a single layer of LCSs that interface with the environment independently. architecture was developed to enable multiple agents to work collectively to solve tasks and to make the system resilient to failure. In essence, this approach simply speeds up the learning process of the original LCS by allowing multiple agents to learn simultaneously and then share their knowledge. The architecture lacks any central control. The only interconnection between agents is a communications bus that allows for the exchange of classifiers. The independence of each agent makes the system very robust. If one agent "breaks down" for a any length of time, the other agents may continue without any major handicap. The only result is that, although the remaining agents learn the solution to the task, the learning process of the group as a whole is slowed. The original implementation of the architecture used Holland's bucket brigade as the apportionment of credit scheme. A later attempt to add Q-learning to this architecture failed to produce significant improvements in performance.

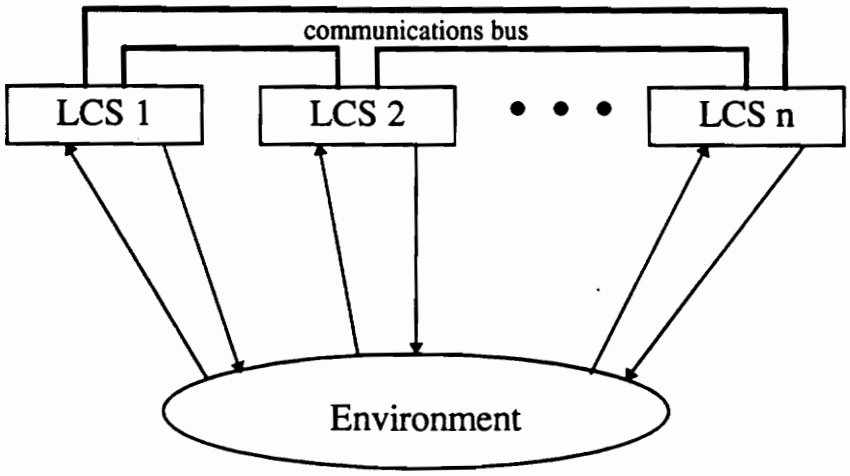


Figure 4.1. Gaff's Distributed Learning Classifier System architecture.

4.2.2 The Peer Architecture

The Peer architecture (Figure 4.2) is also essentially a "flat" architecture. The main difference between it and Gaff's architecture stems from the nature of the application for which it was designed. Whereas Gaff's architecture was designed for implementation on robot groups only, the Peer system was designed for use internally on a single mobile robotic agent or on a group of robots. For a single robot, only one set of sensor values may be taken during a clock tick. Also, only one action may be selected for execution during a clock tick. To this end, the Peer system consists of a layer of LCSs that are all assigned the same task. Each independent LCS has the general architecture and execution cycle covered earlier. The only new element in this system is the *mediator*. The mediator acts as an interface between the environment and the LCSs. All communications to and from the environment pass through the mediator. In essence, the mediator acts as a combination of the input and output interfaces for the system as a whole.

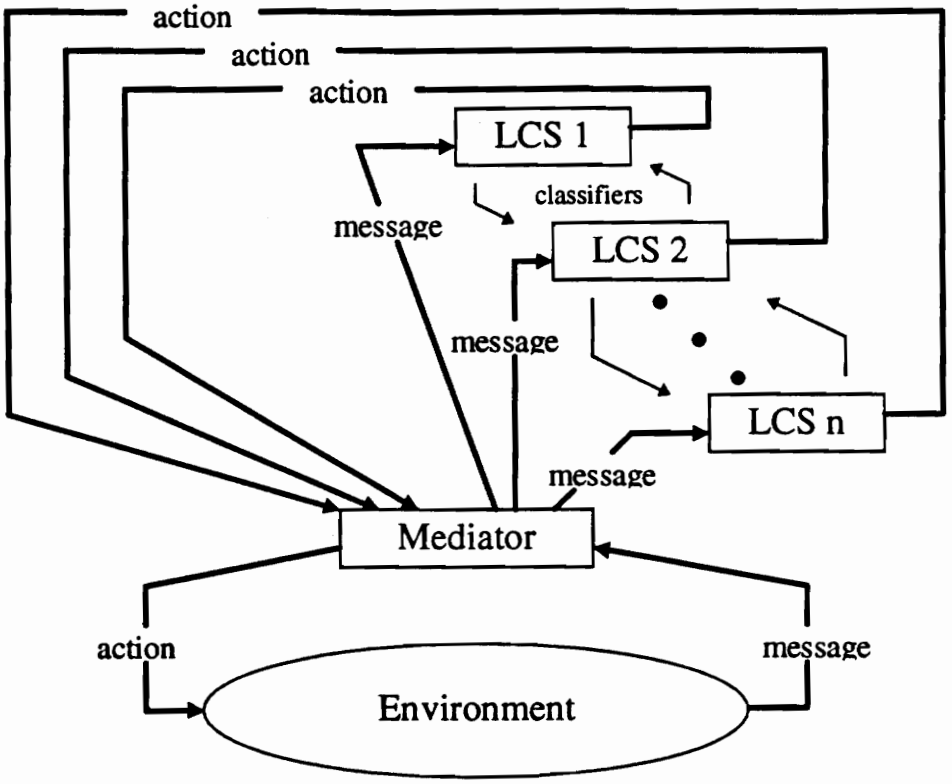


Figure 4.2. Sample *Peer* DLCS architecture.

During each execution cycle, the mediator distributes the same environmental information to all of the LCSs. Using their own rule bases and the rule selection process covered earlier, each LCS then selects an action to be taken and forwards that action to the mediator. Based on a stochastic selection method, the mediator picks the winner from the classifiers nominated by the LCSs and sent to the environment for execution. Reward for the consequences of that action is returned to the mediator, and the mediator forwards this reward to the LCS from which the winning action was sent. After receiving the reward from the mediator, the LCS that posted the ultimate winning classifier updates the classifier's strength based on its apportionment of credit algorithm.

Although this architecture differs somewhat from Gaff's architecture, it suffers from the same fundamental problems that plague Gaff's system when used on a real robot. Namely, the size of the state space is too large for this architecture to search in a

reasonable amount of time. A desirable architecture for real robot learning problems is one in which the size of the state space is decreased, not one where more learning systems are thrown at the same problem.

4.2.3 The DOL Architecture

The DOL architecture (Figure 4.3) is designed with the specific purpose of breaking down a learning problem into a set of smaller problems. This purpose closely matches the idea of task decomposition presented in Chapter 2. This task decomposition nature of the DOL architecture is our reason for choosing it over the other distributed architectures presented above. Referred to by Dorigo as *high level parallelism* [11], the DOL architecture contains a layer of *thinker* LCSs (Section 4.2.3.2) with each focusing on learning a specific behavior. Above this layer of thinker LCSs resides a *combiner* LCS. The job of the combiner is dependent on the application, but in general it groups the decisions of the thinker LCSs and transmits the final action command to the effectors for execution. The combiner LCS is covered in greater detail in Section 4.2.3.3. There is again a *mediator* present in this architecture. The mediator's job in this configuration is somewhat different than in the Peer architecture, as we discuss in section 4.2.3.1.

Note that according to the complexity of the application, the DOL architecture may be expanded into several layers of intermediate combiners before the ultimate combiner that is responsible for transmitting the actual output action. These intermediate layers are somewhat analogous to the hidden layers in a neural network. While this appears to be a viable solution to the computational complexity problem, the introduction of these intermediate combiners does raise some concerns. These are magnifications of the apportionment of credit problems associated with the simple two level DOL architecture. These problems are discussed in section 4.2.3.4. For the applications studied in this thesis, we only implement variations of the "basic" two layer architecture shown in Figure 4.3.

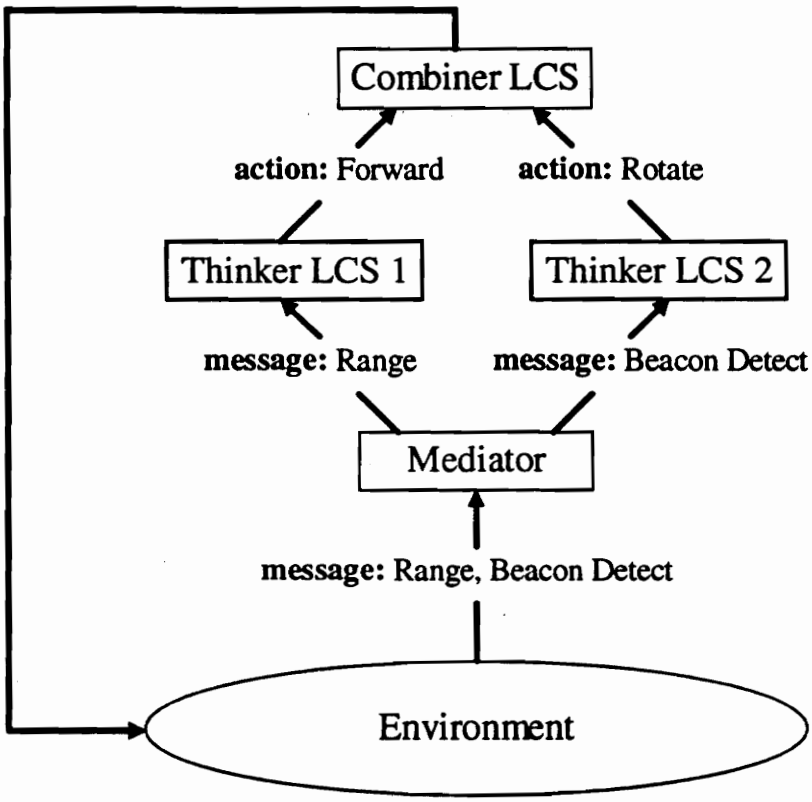


Figure 4.3. Sample *DOL* DLCS architecture.

4.2.3.1 The Mediator

Recall that the first role of the mediator in the Peer architecture is to retrieve the condition message from the environment and then to distribute it to all of the Peer LCSs. The second role of the Peer architecture's mediator is to select the winning action from those nominated by the individual LCSs and then to forward that action to the effectors for execution. In the DOL architecture, the mediator handles only input. As we show in Section 4.2.3.3, the combiner LCS controls the transmission of the winning action to the effectors. The mediator of the DOL architecture is responsible for retrieving the condition from the environment. After retrieving this message, the mediator then splits it into

smaller pieces and forwards the pieces to the thinker LCSs. The mediator is also responsible for distributing the received reward for the system's actions. The manner in which this reward is distributed is the subject of discussion in section 4.2.3.4. In these two capacities, the mediator plays the role of the arbiter of the task decomposition process as discussed in Section 2.4.

The decomposition of the input messages is the most important property of the DOL architecture. It greatly reduces the size of the state space search that must be completed in a learning application. For example, consider a system whose input is a condition string of length 10 bits from the set {0, 1, #} and whose output is a 4-bit action string form the set {0, 1} for control of its effectors. In the classic monolithic LCS approach, as well as in Gaff's and the Peer distributed architectures, each classifier must consist of 14-bits. The cardinality of the state space of the problem is then $3^{10}*2^4$ or 944,784. Next consider an equivalent distributed system consisting of two thinker LCSs, one with a 4-bit input and 2-bit output, and the other with a 6-bit input and 2-bit output. By working in parallel on separate parts of the problem, these two thinkers decrease the size of the state space seen by the thinkers to $3^4*2^2 + 3^6*2^2$, or 3,240 rules. While in actuality even 3,240 possible classifiers is probably too much for a real robot application, this example does characterize the huge reduction in state space seen by the use of a distributed system such as the DOL architecture.

There are obviously many different divisions of the input condition that can occur in the mediator. It appears that there is no way to automate this decomposition; therefore, it must be left to the creator of the system. In this requirement, the mediator becomes another part of the LCS that is specific to the application.

4.2.3.2 The Thinker LCSs

The layer of thinker LCSs consists of two or more LCSs, each focusing on separate parts of the environmental input. Throughout the learning process, each thinker

LCS becomes an "expert" on some aspect of the larger problem that is given to the system. Even though these systems are called thinkers, they require surprisingly little knowledge about the overall task or even their part of that task. Because they do not directly interface with the environment, the thinkers are simply number crunching modules that have no real understanding of the meaning of the rules in their knowledge bases. The overall knowledge lies in the combiner LCS that coordinates the action messages of these thinkers. As with the mediator, the specific configuration of the thinker LCSs (number of input and output bits) must be established as part of the system's configuration process. The thinkers follow the basic LCS execution cycle with only minor differences. Whereas the standard LCS interacts with the environment, the thinker receives input and reinforcement from the mediator and sends its output to the combiner.

4.2.3.3 The Combiner LCS

After the thinker LCSs have selected their individual actions, they pass the action messages to the combiner. Because the combiner is an LCS, it uses the input actions from the thinker LCSs as its environmental condition. It then selects the final action and outputs the action to the robot's effectors. This is much different from the mediator's output role in the Peer system. In the Peer architecture, the mediator simply selects one of the suggested actions that are input by the layer of LCSs, each of which have complete environmental information. The DOL combiner chooses an action based on suggestions from LCSs that each have only a limited knowledge of the environmental state. As the system learns, the thinkers and the combiner should evolve their own "language" to allow the combiner to correctly interpret each thinker's suggestion.

4.2.3.4 Apportionment of Credit in the DOL Architecture

Recall that the apportionment of credit problem is recognized as the largest problem in reinforcement learning. The AOC problem arises in two places in distributed architectures. There is the normal LCS problem of internally apportioning credit, as received by the LCS from the environment, to the appropriate classifier(s). In addition, distributed systems have a similar problem at a higher level. This is the problem of deciding how much of the overall reward should be distributed to each LCS in the system. Since the internal apportionment of credit problem was described in detail in Section 4.1.4, we will only examine the high-level AOC problem here.

In Gaff's distributed architecture, each independent LCS receives reinforcement directly from the environment; therefore, there is no high-level AOC problem. In the Peer system, apportionment of credit is relatively straightforward: simply provide reward to the classifier that won the bidding processes at the LCS and mediator levels or use state generalization to share reward with other similar classifiers. In the DOL architecture, AOC is not as simple. The classifier that wins the combiner's bidding process is an interpretation of multiple actions suggested by the thinker LCSs. Simply assigning all of the credit to the combiner's classifier would prevent the thinkers from learning; therefore, some means of distributing credit to the thinker LCSs must be implemented. The problem of distributing reinforcement across modules in a hierarchical system is referred to as *shaping* by Dorigo [11]. There are two forms of shaping presented in Dorigo's work: *holistic shaping* and *modular shaping*. In our application of the DQLCS in this thesis, we examine the performance of both forms of shaping in the DQLCS as implemented in real robot learning problems.

Holistic shaping treats the entire learning system as one black box. The decisions of individual LCSs are not used in determining the reinforcement. When the system receives a reinforcement for its action on the environment, that reinforcement is given to *all* LCSs in the learning system. This reward scheme introduces some ambiguity problems

to the system. At times, the correct reinforcement will not be given to the thinker LCSs. For example, the combiner may stochastically select the "correct" action even though the inputs from the thinker LCSs are incorrect. This results in bad thinker classifiers being rewarded. Similarly, good thinker classifiers may go unrewarded if the combiner chooses the wrong action when they are supplied as inputs. Regardless, this reinforcement scheme is attractive because it is not very complex, since no internal interpretation of the reward is required. Using this reward scheme, the mediator is able to assign rewards without any knowledge of the goals of the thinker LCSs.

Modular shaping requires an entirely different training strategy for the system. In modular shaping, the thinker LCSs are first trained independently of each other. After they have obtained a suitable level of performance, their learning is "turned off". That is, they are no longer learning, only performing. Essentially, they become reactive systems whose outputs are used by a learning system, the combiner LCS. The combiner learns to coordinate the action messages of the thinker reactive systems to achieve the complex goal in a separate learning exercise. While this system has the benefit that it does not suffer from the ambiguity problems of the holistic reinforcement scheme, for a system with a combiner QLCS and `n` thinker QLCSs, it requires `n+1` different training sessions to learn the task. More importantly, this system operates under the assumption that each thinker LCS *can* be given a task that is capable of being learned independently of the other thinkers. While this may be the case in some simple robot learning applications, one may speculate that not all complex behaviors can be divided into disjoint constituent behaviors.

5. Implementation of the DQLCS on a Real Robot

As previously mentioned in our review of research in the area of robot learning, most of the research to date has involved simulated robots and simulated environments as test-beds for learning systems. This trend is a direct result of the inherent problems that must be faced when implementing a real robot system (Section 2.2). In this thesis, the distributed Q-learning classifier system (DQLCS) is implemented on a real mobile robot for the study of our "goal seeking" problem. To achieve a better understanding of the challenges and the learning limitations introduced by the configuration of the system, we examine the system's hardware and software components here. Our examination of this implementation focuses on three distinct areas: the robot, the environment, and the DQLCS.

5.1 The Robot

The base vehicle used in this research is Curly, a Real World Interfaces [22] B12 platform. The B12 is a three-wheeled base with an on-board microcontroller. The robot's computer architecture is covered in more detail in Section 5.2.2. The robot's drive train is a *synchrodrive* system – the rotations and translations of all three wheels are synchronized. This configuration allows the robot to rotate within its own footprint and is a popular system for mobile robot applications. Above the drive train of the vehicle sits a platform that is attached to the wheel mounts. This point of attachment is important because it allows the platform to be always oriented in the direction of positive translation of the robot. This platform provides a location for any additional hardware that is added to the B12. A more detailed description of the robot base used in this experiment can be found in [22].

In previous research using this robot, an additional microcontroller external to the robot base control computer was added to the system. Also, three low-level obstacle sensors were installed. While the original base structure and these additions provide a very nice platform, it still has minimal external sensing and severely limited communications capabilities. To use this platform in a classifier system application, many additions are necessary. Figure 5.1 shows the completed vehicle that we used in the research presented in this thesis. We document the additions that were made to Curly in this section.

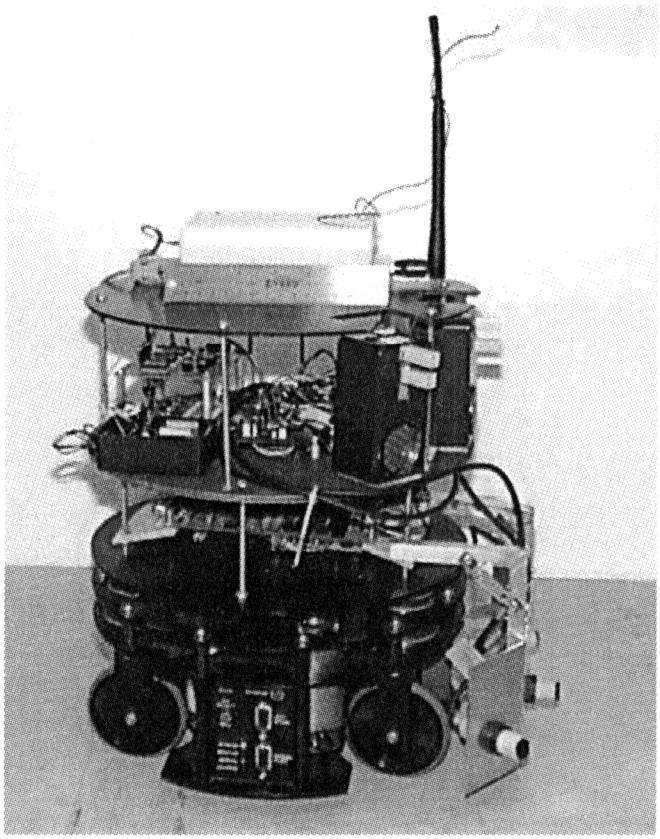


Figure 5.1. Curly, a mobile robot used to study robot learning.

5.1.1 Computer Architecture

As mentioned above, the robot's computer architecture consists of two microcontrollers. We call the microcontroller located inside the base of the robot the *base computer*. It contains the robot's operating system, an interpreter for serially input commands from the robot's control language. Unfortunately, this controller is dedicated to command interpretation and base system control. It has no allowances for development of on-board programs. This limitation led to the addition of a second on-board microcontroller. This new computer, the *command computer*, has two purposes: allow for storage and execution of user programs, and control any additional devices that may be added to the robot. The serial port of this computer can be attached to the serial port of the base computer. This configuration allows for the transfer of robot language commands to the base computer and the transfer of the robot's internal status reports to the command computer. Alternatively, the command computer's serial port may be connected to an off-board device, such as a personal computer. In the original hardware configuration, the access to the command computer's serial port is controlled by a hardware switch.

In our application, the three computer architecture shown in Figure 5.2 is used. In addition to the two on-board computers, an off-board personal computer is used. A program implementation of the Q-learning classifier system resides on this off-board computer that we will call the *host computer*. Because it is desirable to allow the robot a free range of motions, a wireless link between the host computer and the robot is preferred. A pair of radio modems is used to create this wireless link. In our system, the command computer is responsible both for obtaining and transmitting sensory data to the host computer and for transmitting drive commands to the base computer. Because this requires two different computers to have sequential access to the command computer's one serial port, the static serial switch originally on the robot was replaced with a logic

switching circuit that allows the command computer to select the computer with which it is communicating.

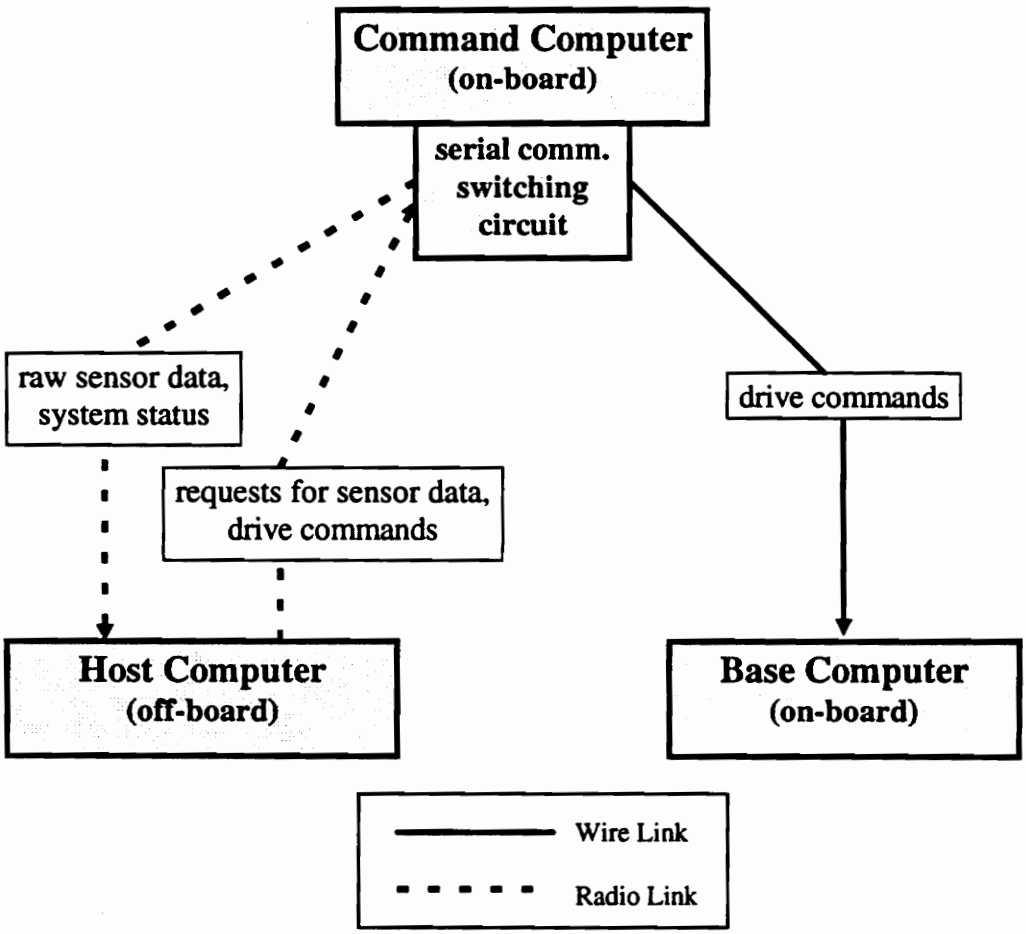


Figure 5.2. Robot computer architecture and communication flow diagram.

5.1.2 Sensing Capabilities

A learning robot must be equipped with sensors in order to sample the state of its environment. The learning robot used in this application was outfitted with three sensor groups, each consisting of three sensors, and an emergency stop. Each sensor group consists of two goal beacon detectors and one ranging sensor. For this application, the

groups of sensors are positioned across the front of the robot in approximately the same orientation as the pre-existing obstacle sensors. However, they may be easily repositioned as necessary in future exercises. Figure 5.3 and Figure 5.4 show the sensors as they appear on the robot.

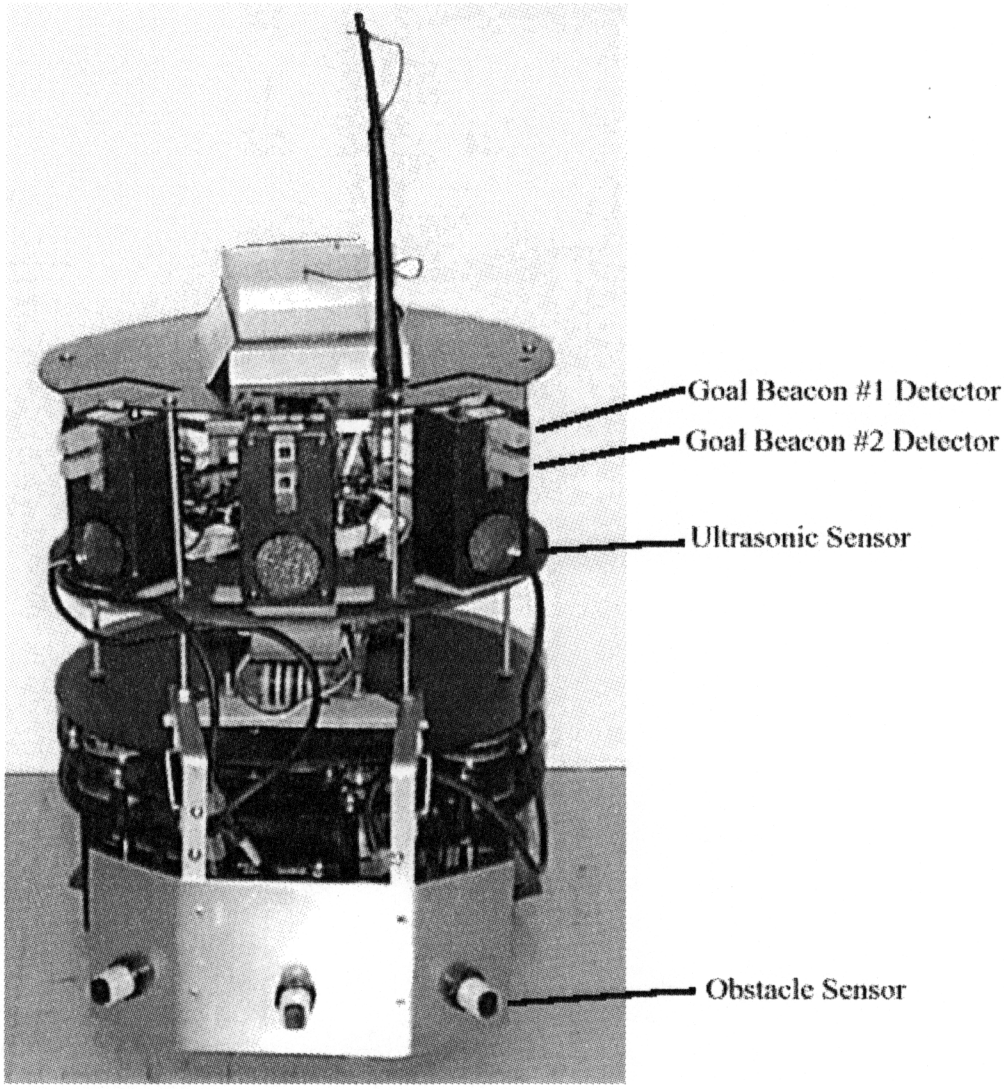


Figure 5.3. Front view of Curly's sensor groups.

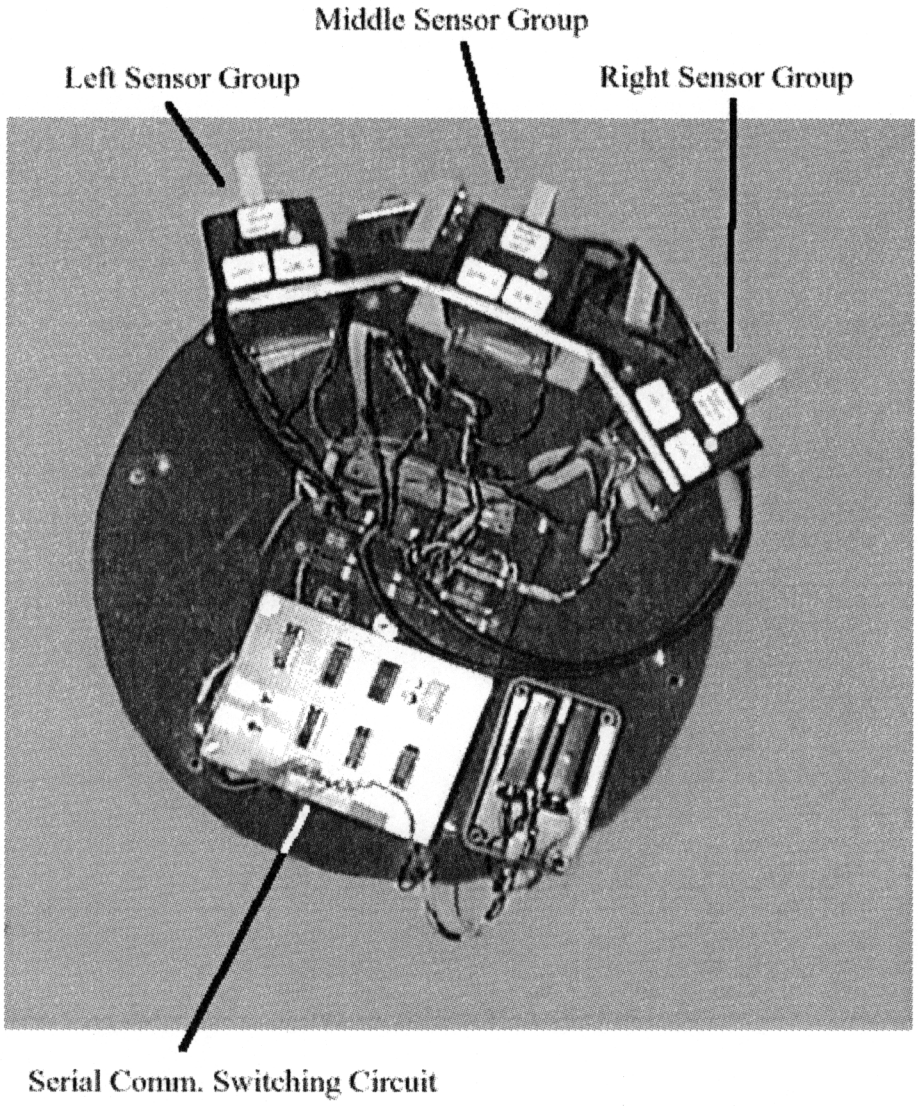


Figure 5.4. Top view of Curly's sensor groups.

Ultrasonic sensors are used to determine the distance to the nearest object in the field of view of the sensor. An input initialization causes the sensor to emit an ultrasonic wave. It then waits for an echo. When the echo is received, the logic signal output of the sensor is driven high. The computer uses the elapsed system time between the initialization and the echo as a time of flight to calculate the distance to the nearest object. While ultrasonic sensors provide a low cost means of obtaining range data that is

extremely important in most robot applications, they are widely recognized for their erratic performance and their limitations. Because of a settling time required by the sensing foil on the sensor, they cannot be used to accurately measure distances less than approximately one foot. Also, they sometimes "miss" objects in their field of view because of the surface orientations of the objects. The detection characteristics for ultrasonic sensors are shown in Figure 5.5. We call the ~30° middle lobe the *field of view* of the sensor. In future diagrams, this lobe is represented by a simple cone of detection.

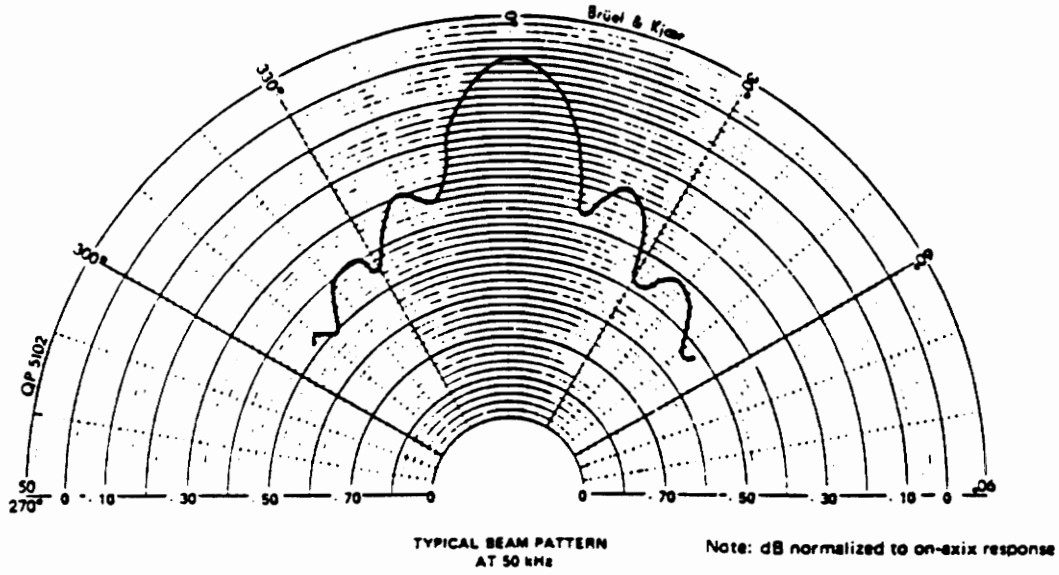


Figure 5.5. Typical beam pattern of an ultrasonic sensor (from [21]).

Each sensor group also contains two goal beacon detection circuits. Each of these sensors is capable of detecting an infrared signal of a designated frequency. Each of these sensors is "tuned" to the output frequency of one of the goal beacons (Section 5.2). If the goal beacons emit significantly different frequencies, they are essentially invisible to all of the beacon detectors except those tuned to that frequency. These sensors return a logic one (+5 V) if a beacon is detected and a logic zero (0 V) if not. We also represent the field of view of these sensors by cones. Initial results indicated that a relatively large field

of view was necessary for the robot to be able to locate goal beacons. By removing the plastic "blinders" around these sensors, the field of view was increased to ~45°. Figure 5.6 shows the approximate fields of view of the ultrasonic and beacon sensors on Curly. Note that the depths of the fields of view shown in this diagram are not to scale. In our system, the depth of view of the ultrasonic sensors is roughly 30 feet, while the depth of view of the beacon sensors is approximately 20-25 feet. In our goal seeking experiment, the robot is never too far away from away from a goal to see it (if oriented towards it).

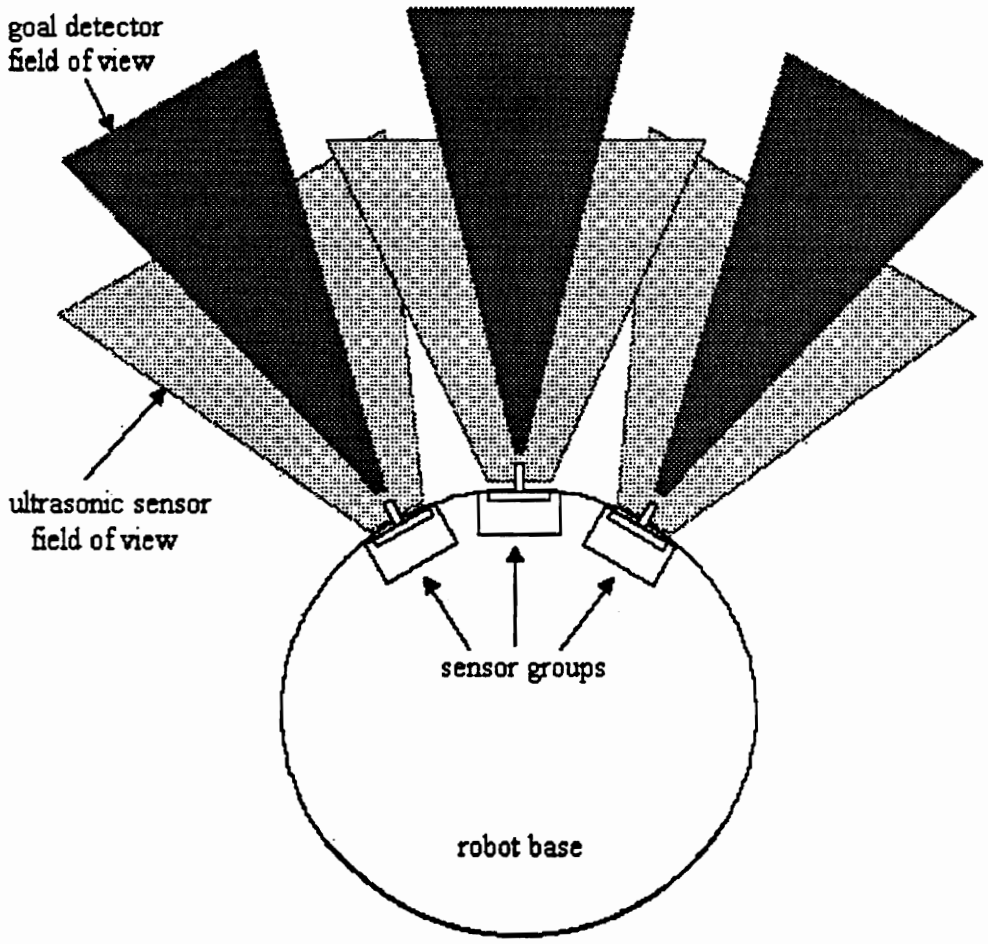


Figure 5.6. Top view sensor diagram of Curly.

Whereas the sensors added during our research are grouped together in boxes on top of the microcontroller platform, the pre-existing obstacle sensors are positioned very close to the ground on a bumper. They each have ranges of four to six inches and return logic values of zero or one. Their position and range allow them to detect the presence of obstacles that cannot be detected by the ultrasonic sensors. This configuration allows for some interesting problem environments to be explored. Using this sensor configuration, the robot may detect the presence of an obstacle directly in front of it, while simultaneously looking over the obstacle and determining the range to a goal beacon (Figure 5.7)

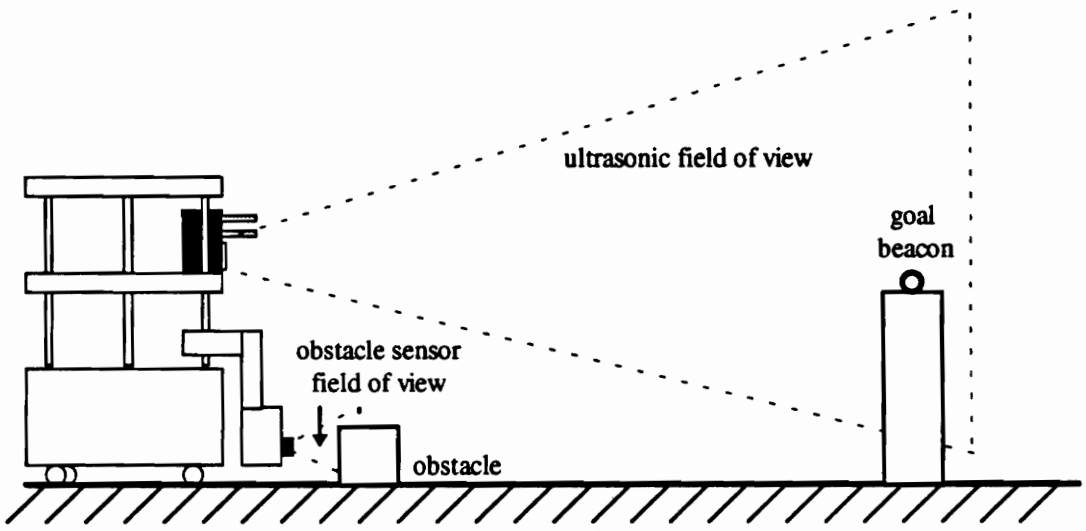


Figure 5.7. Possible application of multiple distance sensing.

5.1.3 Sensor Data Acquisition

The command computer is responsible for collecting, packaging, and transmitting all sensor data. An assembly language program running on the command computer carries out these tasks. After receiving a request from the host computer for sensor data, the

command computer "polls" all of the robot's sensors. For the obstacle and goal beacon detectors, this process requires the simple read of an input port. For the ultrasonic sensors, however, more work is required. The command computer must first initialize the ultrasonic sensor by transmitting a high signal to the sensor's control circuit. The program computes a time of flight distance measurement from the number of CPU cycles that pass between the initialization and return of the ultrasonic signal.

After collection from all of the sensors in a sensor group, the raw sensor data consists of a four-digit range value, two single-bit goal detector values, and a one-bit obstacle detector value. Recall that radio modems are used for communication between the command computer and the host computer. This necessary means of data transfer introduces a significant bottle-neck in the system. Therefore, communications across the radio modems must be minimized. Instead of transmitting all of the sensor data separately, each sensor group's data are combined for transmission. Depending upon the user's choice of range resolution, the command computer scales down the range data until it can be represented by 2, 3, or 4 bits. Whatever the choice, the range data occupies the left nibble of the sensor group information byte (Figure 5.8). Using a series of binary operations, the four sensor values are combined to form a single byte value for transmission purposes. The bit-wise composition of this sensor value byte is as follows:

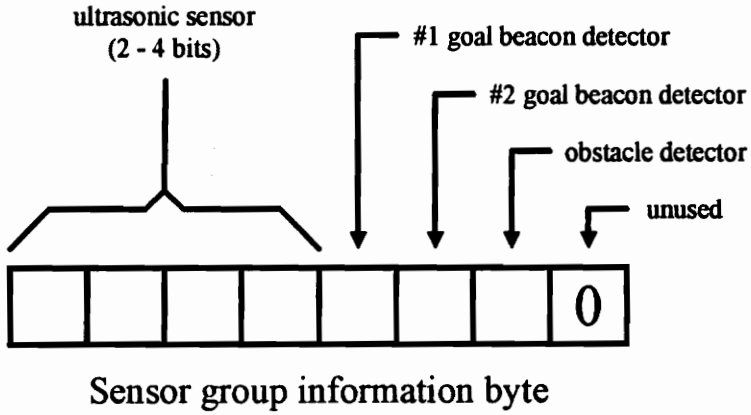


Figure 5.8. Packaged sensor data for one sensor group.

After repeating the sensing and packaging operations for all three sensor groups, the three bytes of sensor data must be transmitted to the host computer. Each byte is transmitted as a pair of ASCII characters, each from the range `0-9` or the range `A-F`, through writes to the command computer's serial port. As part of the handshaking between the command computer and the host computer, the host computer transmits an acknowledgment character each time it receives an ASCII character from the command computer.

5.1.4 E-Stop

Because of the chance that robot might damage itself, the it was fitted with a remote emergency stop. In the learning exercises, no restrictions are placed on the possible actions of the robot. Therefore, it is quite possible that the learning system would choose an action that causes the robot to crash into a wall. That is, after all, part of the learning process. To avoid damaging the robot, the emergency stop may be activated when the robot is about to collide with an obstacle. When activated, a high priority interrupt occurs in the command computer's program. The servicing of this interrupt includes the transmission of the "kill" command to the base computer to stop the translation and rotation of the robot. A special character is also returned to the host computer in place of the next set of sensor data. Upon recognizing this character, the classifier system program terminates.

While the emergency stop was implemented solely as a safety measure, it is possible to use the information it provides in the learning classifier system. From a software standpoint, the emergency stop may be treated as another sensor that detects a highly detrimental environmental condition. As with the other sensors, a reward or punishment may be associated with this sensor to reinforce the robot's action that necessitated its activation. In the goal seeking experiment, the emergency stop does not

take on this role. The goal seeking environment (Figure 1.1) is essentially boundaryless; therefore, no undesirable collisions are possible. We associate a very high cost with an action that results in the activation of the emergency stop.

5.1.5 Robot Control

A very small subset of the B12's command language set is used in this system. For general purpose robot control, only translation and rotation commands are necessary. In the B12 command language, each translation and rotation is achieved by issuing two separate commands. The first command dictates the direction of the motion – forward or reverse for translations and clockwise or counter-clockwise for rotations. The second command then sets the velocity of the motion. The "kill" command `KI` is only used to stop the robot when the system's goal is reached or if the emergency stop is activated. Table 5.1 shows the subset of the B12 command language used in this application.

Table 5.1. B12 command subset.

Description	Robot command
Translate positive	`T+`
Translate negative	`T-`
Set translation velocity	`TV` <4-digit hexadecimal value>
Rotate clockwise	`R+`
Rotate counter-clockwise	`R-`
Set rotation velocity	`RV` <4 digit hexadecimal value>
Kill rotation and translation motors	`KI`

During the operation of the system, all command strings are formulated by the host computer and transmitted to the command computer. Special delimiters are used to denote the beginning and end robot command transmissions. As the command computer receives the drive command, each character is written into a buffer. Then, after the "end

of transmission" symbol is received from the host computer, the command computer triggers the serial port switch and re-establishes communications with the base computer. The command computer then transmits the contents of the command buffer across the serial link to the base computer. The base computer then responds by activating the steering and/or drive motors as needed.

5.1.6 Command Computer Execution Cycle

We have already discussed the execution cycle of the LCS and the DQLCS in previous sections. During these discussions, we indicated that the learning system interfaces with the environment twice, once to receive sensor data (input interface) and once to output actions (output interface). This description does not cover the complex set of actions that must take place inside the robot for these two interfaces to be of any use. As was discovered during the construction of this learning robot system, the command computer's execution cycle is a very delicate, complex, timed procedure that begins when the host computer requests sensor data and ends when an action is taken. During the execution cycle, the command computer must communicate with the base computer and the host computer several times. As previously stated, the command computer runs an assembly language program that controls all of its actions. The execution cycle of this software system is as follows.

From the point of view of the command computer:

1. Establish communication with the host computer.
2. Receive request for sensor data from the host computer.
3. If emergency stop has been activated, transmit E-stop character to host computer. Otherwise, read sensors, package sensor data, and transmit data to host computer.

At this point, there are two possible courses of action that may take place. The host computer receives either sensor data or the emergency stop indicator. If it receives the E-stop indicator, then it terminates execution, and the trial ends. Otherwise it executes the LCS execution cycle and outputs directional drive command. Assuming that a drive command is produced, the rest of the command computer execution cycle occurs.

4. Receive drive command (**T+**, **T-**, **R+**, or **R-**) into input buffer.
5. Toggle communication switch to establish communication with base computer.
6. Transmit buffered directional drive command to robot base.

At this point the base computer actuates the drive motors according to the drive command.

7. Toggle communication switch to re-establishes communication with host computer.

The host computer now transmits the velocity command (**TV** <param> or **RV** <param>) to the command computer.

8. Receive velocity command into input buffer
9. Toggle communication switch to establish communication with base computer.
10. Transmit velocity command to base computer.

The base computer now sets the appropriate motor velocity at new velocity value.

11. Goto Step 1.

We should note that the execution cycle shown above assumes that the classifier system output is either a translation command or a rotation command, and not both. In actuality, the output of the LCS may contain both a translation and a rotation. In this case steps 4-10 are executed twice before the execution cycle ends.

5.2 The Environment

Curly is designed to operate in a variety of environments. It is easy to test its learning capacity in a variety of navigation, avoidance, and searching problems using static environments constructed of obstacles and beacons. While the technical nature and implementation of an obstacle is rather trivial, we will further detail the nature and use of beacons.

Shown in Figure 5.9, the goal beacon is a circuit containing an array of infrared light emitting diodes. This diode array is powered by a source voltage signal whose frequency may be adjusted. As mentioned in Section 5.1.2, in a typical application one goal detector in each sensor group on the robot is "tuned" to the beacon. That is, the robot's sensor only detects input infrared signals of the frequency emitted by the beacon. Then, through the use of beacons with different emission frequencies, the robot may distinguish between the them by detecting them with the two different sets of goal detectors. A full use of this capability would be an environment with two distinguishable beacons. For a multiple goal environment where it is not necessary to distinguish the goals, multiple beacons of the same frequency may be used. The meaning of these beacons is determined by the creator of the environment. They may represent goals to be achieve or obstacles to avoid. As we will see during the discussion of the classifier system implementation, the nature of these beacons is determined by the nature of the reinforcement given to the robot for seeing them.

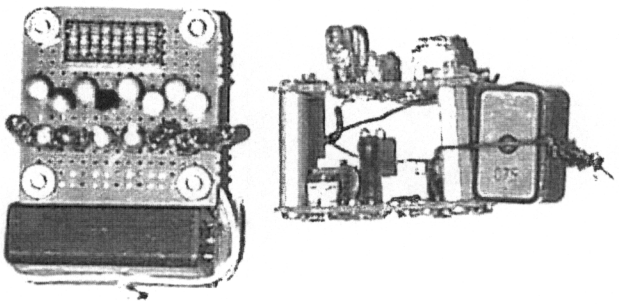


Figure 5.9. Infrared beacons used to mark environment goals.

5.3 The DQLCS

In the previous descriptions of learning classifier systems and the DQLCS, we indicated that the contents of several components are dependent on the nature of the application of the learning system. In this section, we provide descriptions of the components of our mobile robot implementation of the DQLCS.

To allow for the study of multiple classifier system configurations, the application requires the user to configure each new learning experiment. The user may construct a monolithic classifier system with any combination of the robot's sensors as input and any combination of robot actions as output. Alternatively, the user may choose to build a two-level DOL distributed system. In this case, the user may again choose any combination of sensors as input for the thinker classifier systems and any action string size as output. For the hierarchical system, the user must also configure the combiner QLCS. The input size is predetermined by the action sizes of the thinker QLCSs, but the user must choose the robot actions to be used as output. We describe the configuration process in greater detail during the discussion of the individual components of the DQLCS.

5.3.1 Classifier Structure

Classifier conditions in combiner QLCSs have very different "meanings" from classifier conditions in thinker QLCSs and in monolithic QLCSs. The size of each classifier condition in a monolithic QLCS or for a thinker QLCS in a distributed system is determined by the choice of the system's input sensors. Each sensor has an associated number of bits. The order of appearance of the sensor data in the condition part of the classifier is determined simply by the order in which the sensors are chosen during the setup phase of the experiment. The order is not important, however, since it in no way effects the operation of the system. In a combiner QLCS of a distributed system, the condition part of each classifier is a concatenation of the output action strings of the individual thinker QLCSs. It is not possible to give any "real world" meaning to these condition strings or the action strings from which they are composed. In fact, one popular interpretation is that these strings are words in a complex language that is developed between the combiner and the thinker QLCSs.

As with the condition strings, the structure of action strings varies according to the role of the QLCS. For QLCSs that interface with the robot (combiner and monolithic QLCSs), the action string is an index into a lookup table of robot actions. Depending upon the application, the action string may represent one or two robot actions – rotation and/or translation. The size of these action strings is determined by the size of the lookup tables of robot actions. If the action string belongs to a classifier in a thinker QLCS, the size is determined by the user. There is no formula for determining the "correct" action string size for thinker QLCSs in a distributed learning system.

5.3.2 Input Interface

The input interface of a combiner QLCS forms messages by simply concatenating all of the output messages of the thinker QLCSs in the system. Again, the order in which

the thinkers' actions messages appear in the condition of the combiner QLCS has no impact on the performance of the DQLCS.

Recall that all of the robot's sensor data is transmitted to the host computer during each clock cycle of the system as three packaged bytes of data (Figure 5.8). The role of each thinker QLCS input interface is to construct messages from this raw data. During the setup phase of an experiment, a software structure is filled with the QLCS's configuration as specified by user input[1]. Then, during each clock tick, each QLCS input interface receives the same set of raw data from the mediator of the system. After the raw data arrives at each input interface, the QLCS's input message is formed by referencing its configuration structure and extracting the appropriate bits from the sensor data bytes. Figure 5.10 shows the construction of a classifier system input message from the sensor data for a classifier system that uses an ultrasonic sensor and a "Goal #2" sensor as input.

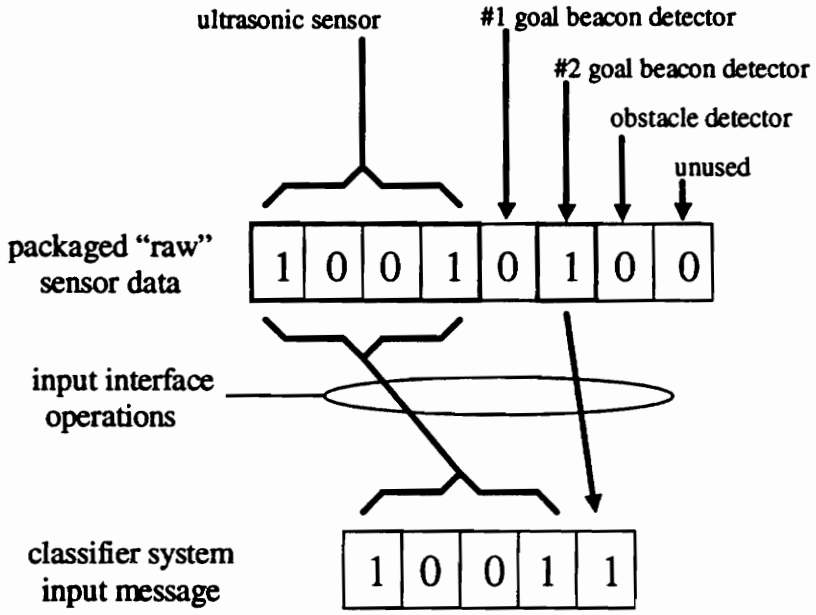


Figure 5.10. Example of input interface encoding of input sensor data.

[1] It is possible to have sensor overlap between separate classifier systems in a hierarchy. No restrictions are placed on the sensor data used in a QLCS; therefore, multiple QLCSs may share the same sensor data. Although this capability exists, it was not used in the learning experiments performed in this thesis.

5.3.3 Output Interface

Whereas the input interface acts as an encoder to format the environmental condition messages as a binary string, the output interface acts as a decoder. For thinker QLCSs in distributed system, the output interface's task is rather trivial. It simply forwards the exact action string from the winning classifier to the input interface of the combiner classifier system.

For the output interface of a monolithic system or a combiner, the task is much more complex. Here, the binary action string from the winning classifier is received and mapped to one or more robot commands. The number of bits in the action part of the classifier reflects the number of possible robot commands for the system. The translation and rotation velocities are kept in separate lookup tables, and the appropriate substrings of the action strings are converted to integers and used as indices into these lookup tables. Figure 5.11 shows the mapping process for a classifier system that uses eight translation velocities and four rotation velocities.

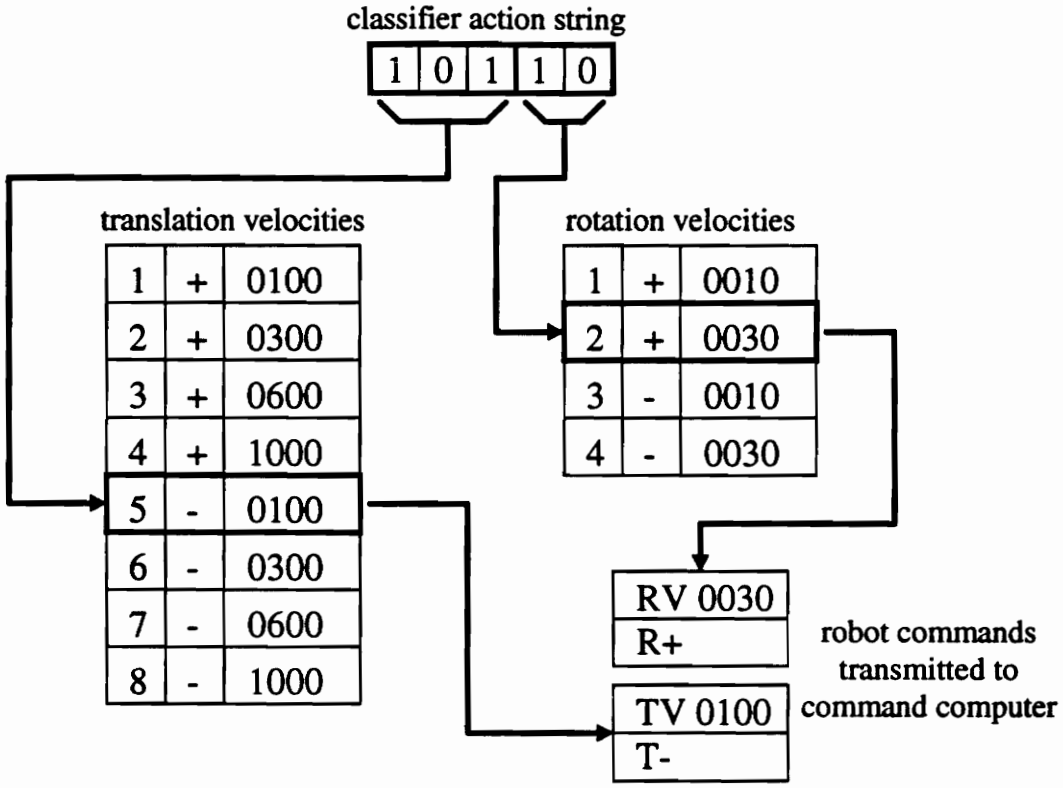


Figure 5.11. Output interface processes for an example application.

5.3.4 Rule Selection

In learning systems, it is important to sometimes choose rules that do not appear to be the best rules for the environment state. A rule's strength is only an *estimate* of the minimum cost that will be incurred through its use. Because credit assignment is calculated from the state of the environment as determined by the robot's sensor values, any sensor noise may cause an erroneous reinforcement to be given to a rule. Also, rules that were once poor rules may become good rules through changes in the environment. Finally, it is important to test unproven rules to determine if they lead to a better solution than any currently strong rules. Learning systems combat this "shortcut problem" by using probabilistic rule selection algorithms. There are a variety of popular selection

mechanisms, such as the roulette-wheel (Section 3.8). In our system we use a slightly different approach, although we do not make any claims about the comparative qualities of such algorithms. The DQLCS rule selection procedure is simple: select the rule with the lowest Q-value 90% of the time, and randomly select an eligible rule the remaining 10% of the time.

5.3.5 Rule Execution

After a rule is selected, the action message is transformed into a set of robot commands by the effector interface as shown in Figure 5.11. The execution of these commands involves a routing procedure in which the rule is transmitted from the host computer through the command computer to the base computer. The specifics of this procedure are discussed in Section 5.2.6.

Preliminary results revealed the emergence of an undesirable behavior in learning problems that involved rotation only. While a rotation-only task has limited use for *mobile* robot systems, the behavior is one that results from sensing limitations and not robot action limitations. It was observed that if the robot is in a position in which no single move could change the state of the system, the system soon learns that all rules are equally bad. The specific example is the problem in which the robot learns to orient itself with a goal beacon using its goal detection sensors as inputs and rotations as actions. Because there are three bits of sensor input data, there are 2^3 or eight possible environment states. We see that the locations of the sensors across the front of the robot results in environmental states that are not all equally sized, as shown in Figure 5.12. The numbers in this figure represent the sensor values that would be input into the system if the goal beacon was located in the corresponding region.

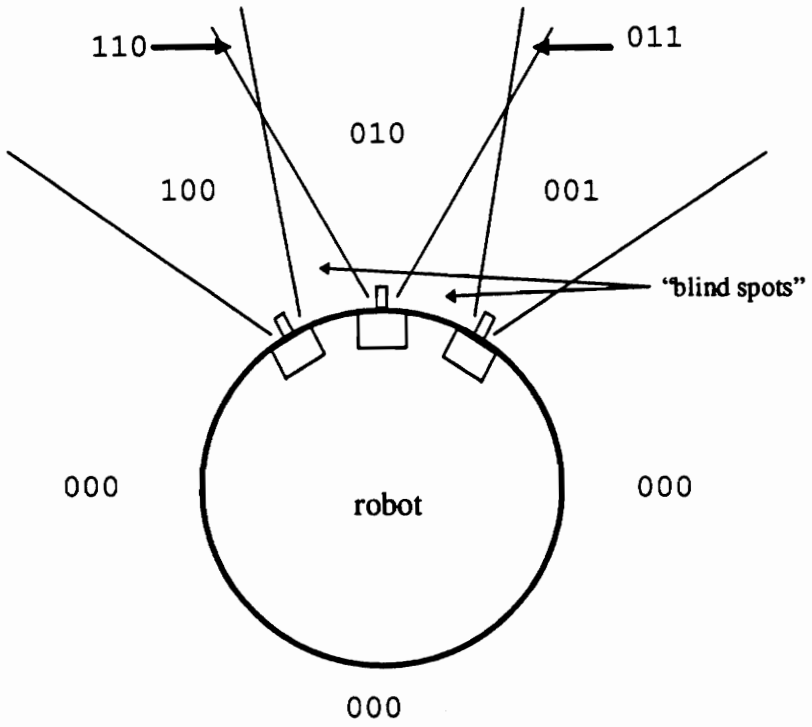


Figure 5.12. Sensor value regions for goal detection sensors.

If the robot is facing away from the goal beacon (sensor value 000), it is possible that no single rotation will change the state of the environment. If a chosen action does not change the state, it receives a penalty and has a lower probability of being chosen during the next time step. In the next time step, another action that is also incapable of single-handedly changing the state of the system may be chosen. From a sensor standpoint, these actions are then equally useless to the system. The result is a process in which the robot repeated cycles through all of the possible actions in that state. If there are equivalent clockwise and counterclockwise rotation commands, the net change in position from executing all of them is zero. The occasional random action may break the cycle, but it may not if the state is too large. This behavior was observed during early experiments with Curly. There is an equivalent translation problem. If the robot's task is to translate to a goal using a sensor to determine the goal's distance, there is region that

corresponds to the 000 region shown in Figure 5.12. The region labeled as the highest distance value encompasses every distance from just larger than the second-highest distance value to an infinite distance. It is easily possible that a single translation will not cause a change in the state of the environment. We should note that the translation case is less likely to happen and is reasonably easy to avoid in a real robot system since the robot's environment is usually finitely large. The only mechanical way to avoid the rotation problem is to add more sensors to the robot, a task which may not be possible considering the available computing resources on the robot. A second approach would be to restrict the rotations of the robot to one direction only. This solution is equally unreasonable, since it would sometimes require the robot to rotate almost 360° to change it's overall position even slightly.

This problem is not a new problem for learning systems. In [7], the idea of "learning the persistence of actions" is explored. In their research, Cobb and Grefenstette present a system in which the agent learns not only the control action to apply for a given conditions, but also the length of time to apply the action. While the inclusion of action duration in the learning process is interesting, we selected a simpler method for overcoming the problem in our system. During each clock cycle the system, if the state of the system is the same as during the last clock cycle, an error value is incremented. Then, the duration of the present clock cycle is extended by a factor of this error value. If the state of the environment changes from one clock cycle to the next, this error is reset to zero. This process in no way encourages the system to learn the best action to take when in the 000 state, but it does provide a means for eventually making a transition to another state.

5.3.6 Credit Assignment

The choice of Q-learning over the bucket brigade algorithm for the apportionment of credit component of our distributed system has been well documented earlier in the

thesis. For implementation purposes, however, some specifics must be covered. As mentioned in Section 5.3.4, it is often useful to choose rules that do not appear to be good rules for the current state of the environment. In the QLCS, as in the LCS, a classifier's usefulness to the system is determined by it's strength. The role of the credit apportionment algorithm is to tune these strengths over time. There are two divergent opinions about the "meaning" these Q-values should have. Recall that our convergence proof for the Q-learning algorithm requires that all reinforcement is of the same sign, that is, the system must receive all rewards or all punishments. Some assert that the Q-values should reflect the maximum reward that can be achieved by executing the given action in the given environment. The optimum path in these learning systems is one that receives the maximum amount of reward. In our implementation, we choose an alternative representation. The Q-values of rules in the DQLCS represent the cost incurred by taking the given action in the given state. All reinforcements from the environment are in the form of penalties, and it is the goal of the system to find a path that minimizes the total penalty for going from the starting position to the goal.

The differences between the "maximization of rewards" and "minimization of penalties" approaches may seem insignificant at first. We assert, however, that the latter approach, combined with our system initialization, works best in our classifier system framework. Using our cost minimization approach, we initialize the system by assigning an initial Q-value of zero to each classifier. In this sense, every classifier is a good choice initially, and it is left to the learning algorithm to determine which are actually bad. While a classifier's Q-value may decrease through the application of the update equation (4.14), it never falls below zero. We can see then that a Q-value of zero is the strength of the best possible classifier. A nice effect of this approach is that it directly encourages the total exploration of the state space. Unless the a rule's execution causes the system to reach the goal state, some penalty is incurred. From an initial state where the rules Q-value is zero, the update equation uses the incurred cost and causes the Q-value to become some positive value. Therefore, the next time the environment is in this same state, the

previously chosen classifier is relatively poor compared to the remaining classifiers whose Q-values are all still zero. The system then probabilistically selects one of the other rules for execution. This process iterates every time the environment reaches this state, and soon all of the rules associated with the state are tested. Notice that this is not as straightforward in a maximization of rewards approach. In the maximization strategy the best classifiers may have arbitrarily high strength values. Since the exact Q-value of a good rule is not known, it becomes more difficult to encourage state exploration through the initialization process. All of the classifier's may be initialized with some high value, but this may not be as high as the best Q-value. If the update causes a classifiers strength to go up, it then becomes more likely to be chosen during the next visit to that environment state. The remaining eligible rules go untested in this case.

6. Robot Learning Experiments

We wish to observe the performance of various configurations of the Distributed Q-learning Classifier System (DQLCS) when applied to typical real robot learning problems. The problem environment selected for the learning system is shown in Figure 6.1. The problem is a very common problem in robotics: find the best path through an environment to a goal position. A analogous real-world problem is a docking manuever. While this problem may be rather basic, it does allow for the application of several components of our learning system. The solution may require the robot to execute both translations and rotations.

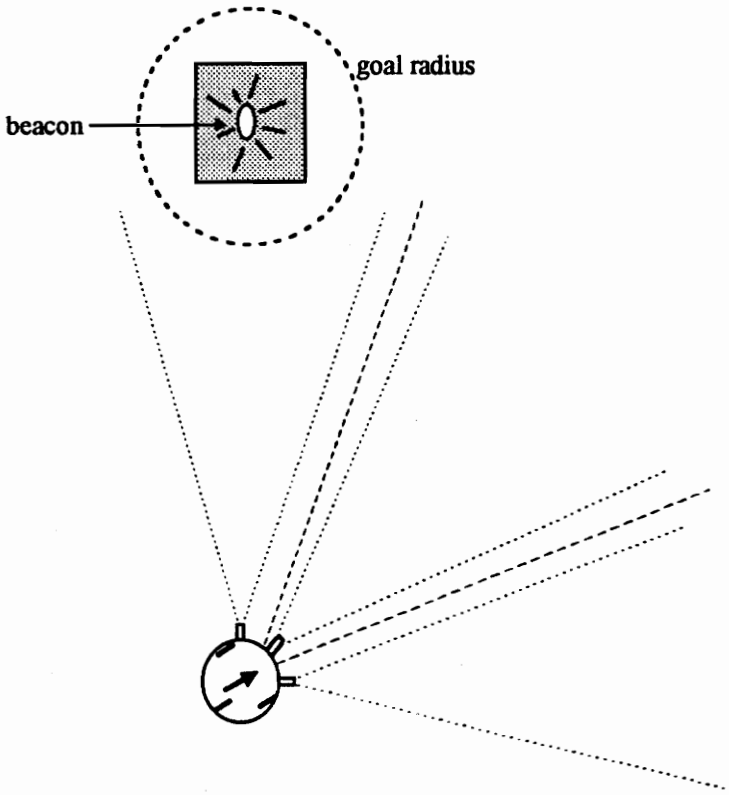


Figure 6.1. The goal seeking problem environment.

For the application of our robot system, the problem statement had to be refined slightly. Because the real robot has no way of determining its exact absolute position in the environment, it cannot know when it is in the goal position unless that position corresponds to some known sensor value combination. In each experiment, the system's goal is a desired sensor value specified by the user during the configuration of the QLCS/DQLCS. In our application, we used the three "Goal #2" goal beacon detectors and the middle ultrasonic sensor as input sensors. For the problem studied, the ultrasonic sensor was used to return 3-bit (eight possible values) data describing the goal beacon distance. For these sensors, the "docked", or goal, position was then defined as any position where the robot was "seeing" the goal beacon with *at least* the middle goal detector and the range from the ultrasonic sensor is one unit of measure. Note that these goal requirements actually specify a circle of radius 1-unit around the goal beacon (Figure 6.1). Because this is a reinforcement learning system, penalty values must be chosen to provide feedback from the environment to enable the learning system to compute the usefulness of its rules. Table 6.1 provides a breakdown of sensors and their associated penalties. When selecting these values, only a few qualifications were used. One was that all penalties be in the same order of magnitude. We did not want one penalty to totally dominate the others. We also wanted to penalize the robot for not seeing the goal with the middle detector more than for not seeing the goal with the right and left detectors. The coefficient of ¼ was arbitrarily chosen as this factor.

Table 6.1. QLCS penalty values for experiments

Sensor	Associated Penalty
Middle ultrasonic	10*(3-bit range value)
Left Goal #2	0.25*50 = 12.5
Middle Goal #2	50
Right Goal #2	0.25*50 = 12.5

Because the selected problem is one that requires the use of several sensor inputs and two action outputs, it is a good candidate for solution using the task decomposition capabilities of the DQLCS. Therefore, in our study of this problem, we tested several configurations of our learning system. First, we observed the performance of a monolithic QLCS. Next, we distributed the problem across a set of thinker QLCSs and a combiner QLCS in a hierarchical system. As mentioned in Section 6.2, there are many approaches to the distribution of a learning problem across a group of modules. For the problem studied here, we decomposed the problem goal into two subgoals: translation and rotation. The goal of the translation module was to place the robot at the desired range from the goal beacon, while the goal of the rotation module was to simultaneously place the robot in the correct orientation. Using the distributed approach, a learning module was dedicated to each subtask. The overall goal of the system was said to have been realized when both of these subgoals were met during the same time step. In all of the experiments we conducted, genetics were not used as part of the QLCS execution. All of the possible rules were maintained from beginning to end. In an experiment the classifier's Q-values were saved at the end of each trial and used at the beginning of the next trial.

6.1 The Monolithic Approach

The docking problem was first studied using a monolithic Q-learning classifier system. The inputs from the three 1-bit "Goal #2" beacon detectors and the 3-bit middle ultrasonic sensor were concatenated to form the condition string for this system. Because four translations and four rotations were possible for the system, the output action string for each classifier was four bits: a 2-bit rotation substring and a 2-bit translation substring. The state space of the problem was then covered by 2^{10} or 1024 classifiers. For this and the distributed learning experiments, the initial state of the environment was one where the robot could "see" the goal with its left "Goal Beacon #2" detector as shown in Figure 2.1. Also note that for this experiment, and for all other experiments, an upper limit of 50 time

steps was used. Regardless of the position of the robot, the system terminated the trial after the 50^{th} clock cycle.

Figure 6.2 shows the monolithic QLCS's performance during this experiment. As seen on the graph, the performance of the system is measured by the speed at which it was able to reach the goal position. This system was observed for 100 trials, where the Q-values at the end of each trial were saved and used as the initial Q-values for the next trial. For most of the experiment, it is difficult to find any system improvement. Within the final 15 trials, however, the system appears to be settling to some standard performance. While we believe that further experimentation would have yielded a more complete performance curve and eventually very good system performance, it would have also required much more time. The purpose of this experiment was not to show that the problem could be solved. The results from this experiment are presented simply to provide a "measuring stick" for understanding the significance of the results obtained from the two distributed approaches to the same problem.

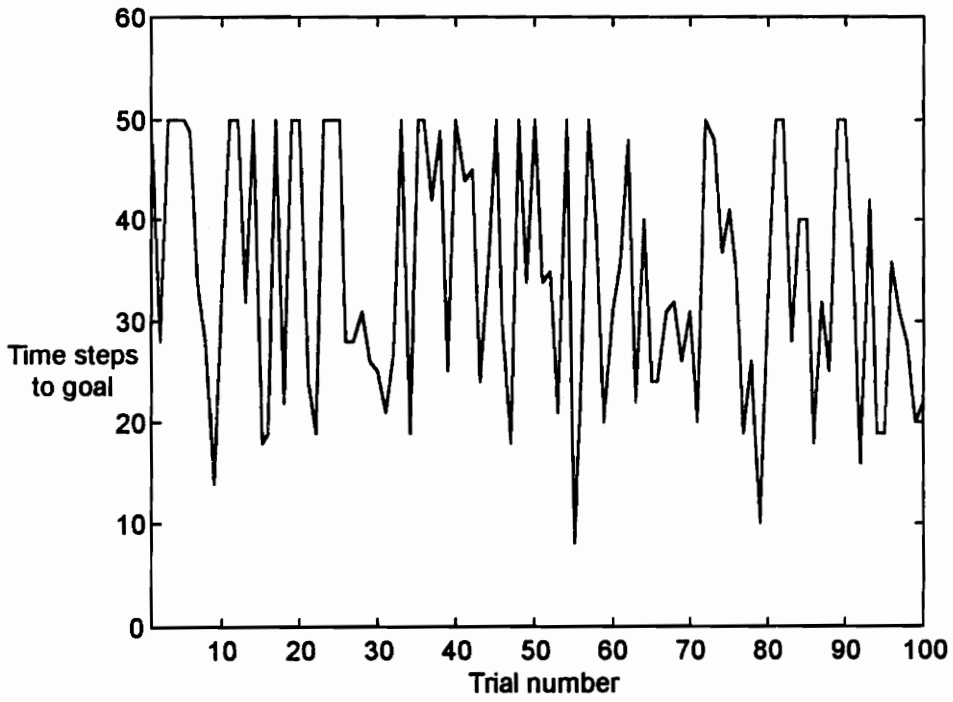


Figure 6.2. Learning curve for monolithic QLCS.

6.2 The Distributed Approach

As stated previously, we chose to study this problem using the Division of Labor (DOL) architecture for our DQLCS. Using this modular architecture, we hoped to decompose this task into two easily solvable sub-tasks: locating the goal and approaching the goal. Inside this distributed framework, we chose to observe the performances of two reinforcement schemes. These two reinforcement methods are called *holistic shaping* and *modular shaping*. After the task decomposition is complete, these shaping methods would then determine how to merge these two behaviors into one cohesive complex behavior.

For these experiments, the initial system environment was the same as in the monolithic QLCS experiment covered in Section 6.1 and shown in Figure 6.1. The classifier system arrangement was quite different, however. For both of the distributed system experiments, the classifier system configuration was as described in Table 6.2.

Table 6.2. DQLCS classifier configuration.

LCS	Input	Output	Classifiers
Thinker #1	3-bit range data from middle ultrasonic sensor	2-bit string	32
Thinker #2	(3) 1-bit goal beacon detectors	2-bit string	32
Combiner	2-bit output from Thinker #1 2-bit output from Thinker #2	4-bit string	256

6.2.1 Holistic Shaping

Recall that in holistic shaping, all levels of the classifier system hierarchy begin with no knowledge and obtain knowledge simultaneously. In this system, each QLCS is rewarded equally during each time step for the performance of the system as a whole. In

this experiment, the system's performance was observed for 80 trials. Figure 6.3 shows the resulting learning curve. While at first glance it may be difficult to notice any trend in performance, further observation reveals that there was some marginal improvement shown by this system. During the first 20 trials of the experiment, the 50 time step "time-out" was invoked during most of the trials. Even in those trials where the robot did reach the goal, more than 25 time steps were always necessary for the goal state to be reached. Throughout the remainder of the experiment, the robot was able to find the goal more often and in less time.

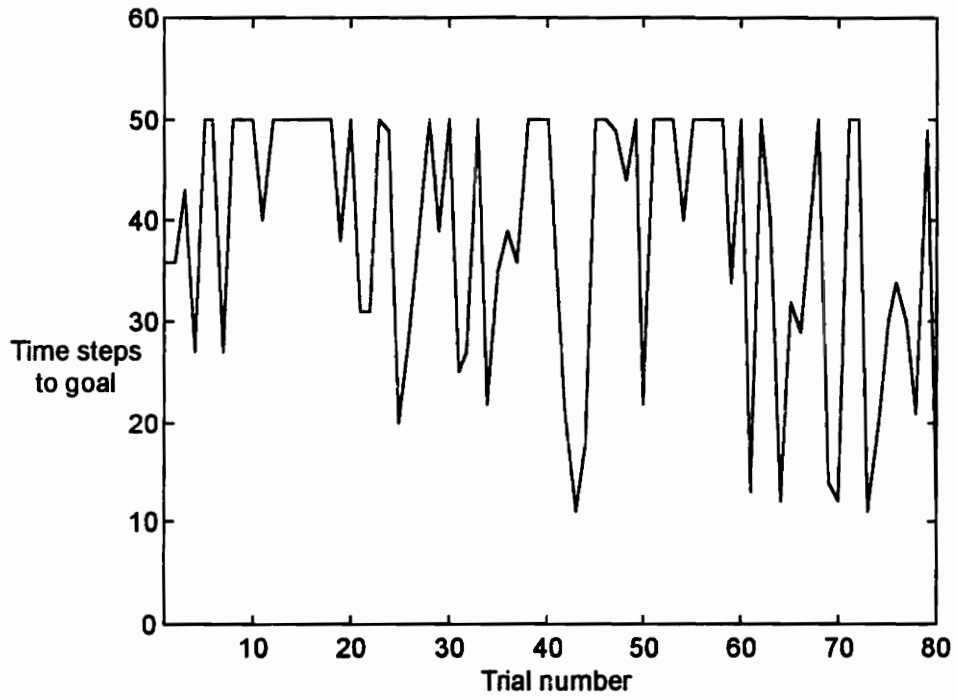


Figure 6.3. Learning curve for DOL architecture with holistic shaping.

6.2.2 Modular Shaping

In a system using modular shaping, the overall desired behavior must be divided into separate sub-behaviors for individual learning sessions. As previously mentioned, we chose to divide the overall "goal-seeking" or "docking" behavior into the sub-tasks of "approach goal" and "locate goal", where the "approach goal" behavior involves translation only and the "locate goal" behavior involves rotation only. After each of the two low-level behaviors were learned, the two separate modules were used in a third experiment in which the "seek goal" behavior was learned.

6.2.2.1 Translation: The "Approach Goal" Behavior

The goal of this experiment was to allow one thinker QLCS to learn the best means of approaching a goal as detected by the robot's middle ultrasonic sensor. The environment for this experiment is shown in Figure 6.4. In essence, the thinker was to learn the relationship between distance and translation. The QLCS used in the experiment received 3-bit ultrasonic ranges as input and produced a 2-bit output that was used to select from four possible translate commands. The goal distance for the experiment is 1 unit. During each time step, the cost of an action was determined by a constant multiple of the range to the goal as a result of that action. The learning curve for the system is shown in Figure 6.5.

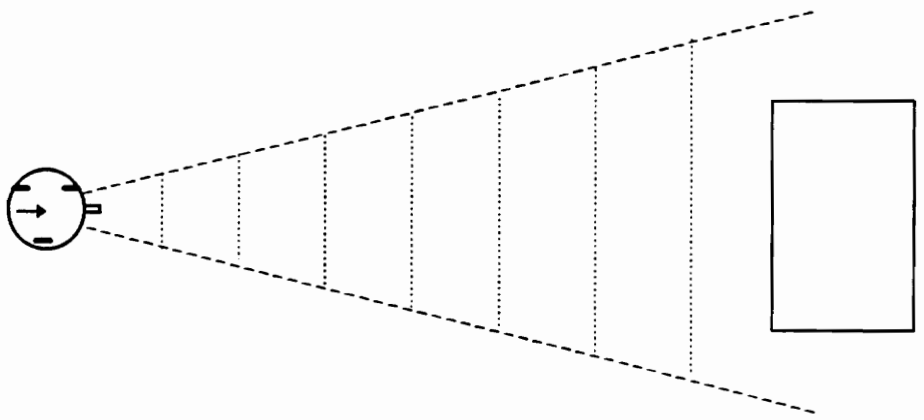


Figure 6.4. The "approach goal" environment.

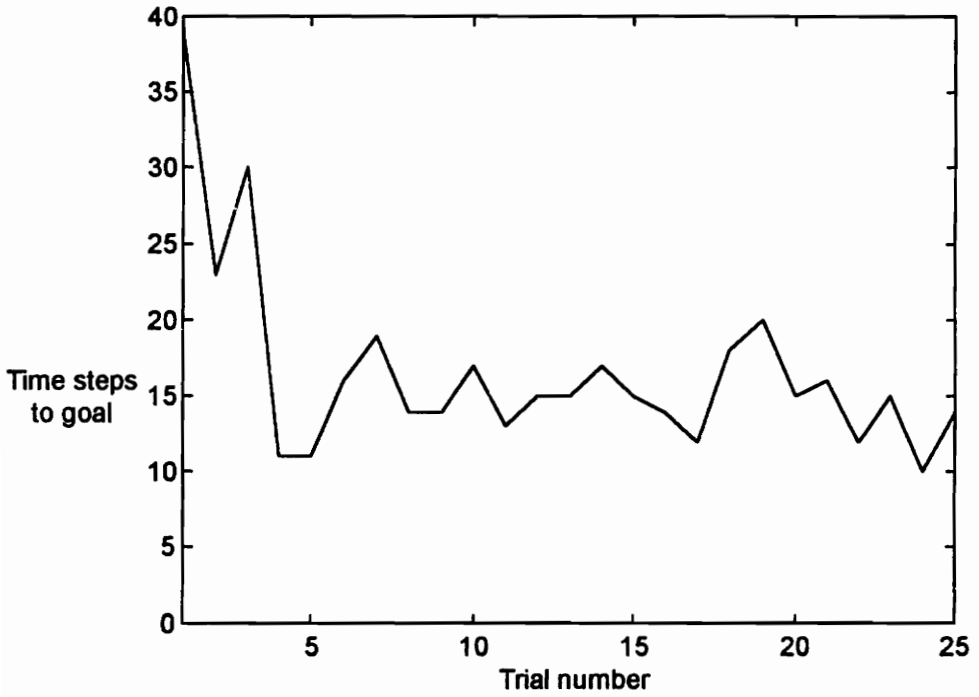


Figure 6.5. Learning curve for "approach goal" behavior.

This experiment is the only one in which the sensor data can be used to reconstruct a meaningful course mapping for individual trials; therefore, we provide some examples here in Figure 6.6. There is a noticable improvement in the speed at which the goal state is found between the Trial #1 and Trial #17. Note that during Trial #2, the robot negative translation caused a change from one system state (distance = 5) to another (distance = 6). The robot was actually backing up during several of the time steps of Trial 1 as well, but the negative results of these motions were not seen due to the discretization of the problem space.

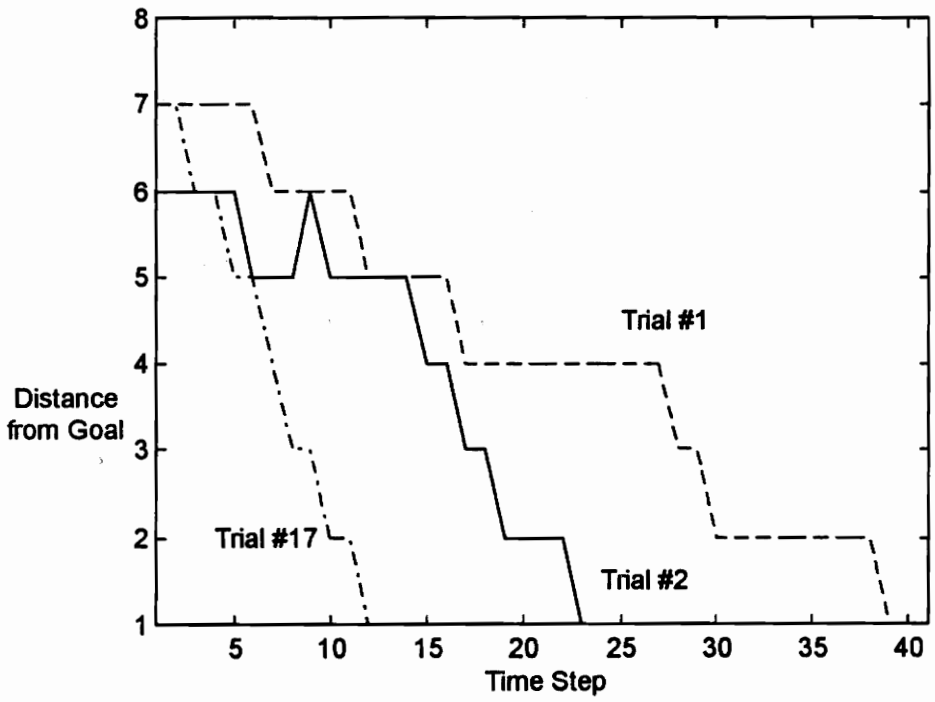


Figure 6.6. Paths from translation experiment.

6.2.2.2 Rotation: The "Locate Goal" Behavior

Similar to the translation behavior discussed in the previous section, the rotation, or "locate goal", behavior is quite simple when examined alone. To achieve the objective of seeing the goal beacon, the robot simply needs to learn to coordinate its goal beacon detectors with its rotation commands. For this experiment, the QLCS used a 3-bit input (one bit from each of the three "Goal #2" detectors) and generated a 2-bit action command that was used to select from among four rotation commands. The system goal was to see the goal beacon with at least the middle beacon detection sensor. Penalties were issued for not seeing the beacon with middle sensor and lesser penalties were issued for not seeing the beacon with the right and left goal detectors. To ensure that the system learned the behavior completely, three different initial environments were used as shown in Figure 6.7. For Trials 1-30, the starting position had the robot's left goal detector seeing the beacon. In Trials 31-50, the right beacon detector initially saw the beacon. In the remaining trials, the robot was placed such that the beacon was directly behind it in the rather large 000 "blindspot" discussed in Section 5.3.5.

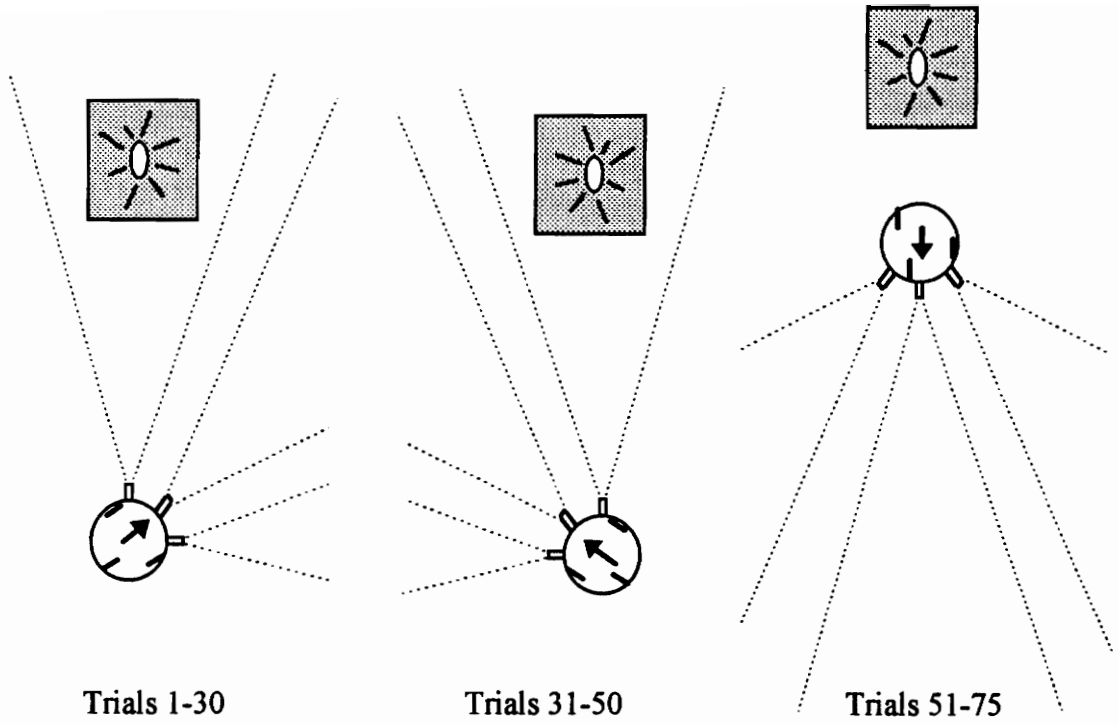


Figure 6.7. The "locate goal" environment.

The results of the experiment are shown in Figure 6.8. This learning curve is actually a combination of three separate learning curves, one for each new starting position. In each of the three starting positions, we see that the system quickly learns the appropriate course of action to quickly find reach the goal state.

Note that in each of the first two starting positions, the time required to reach the goal state settles to the same value (5 time steps). This is expected since these two starting positions were basically mirror opposites. In these two cases, the physical distance between the starting position and the goal position were the same. It is also noteworthy to discuss the third part of the learning curve shown in Figure 6.8 (Trials 51-70). From this new starting position, the robot learns that the best course of action requires approximately 10 time steps to reach the goal state. This higher settling value is expected since the initial system state is much farther from the goal state.

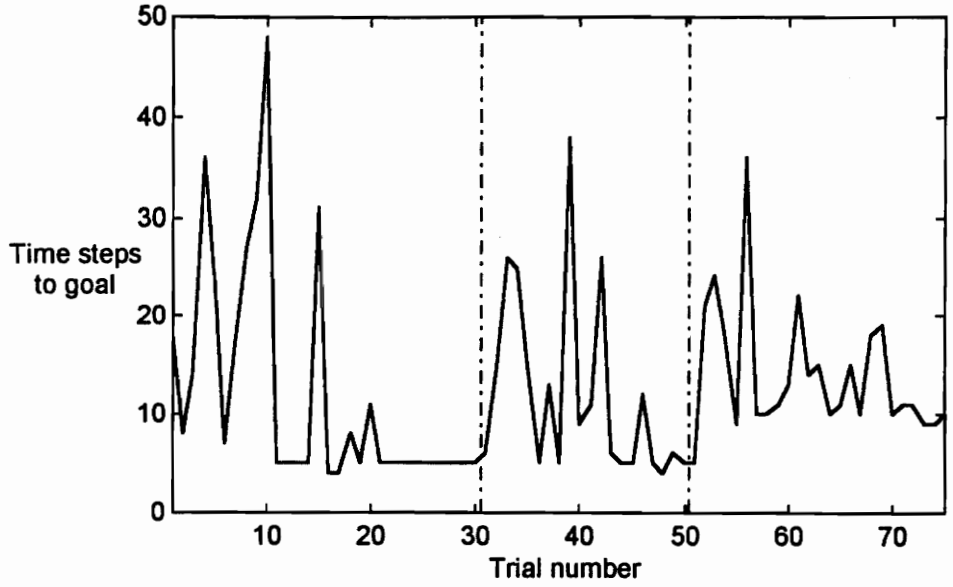


Figure 6.8. Learning curve for rotation behavior.

6.2.2.3 Translation and Rotation: The "Seek Goal" Behavior

After the "locate goal" and "approach goal" behaviors were learned, their learning mechanisms were "turned off". This essentially transformed them into reactive systems. These two QLCSs were then incorporated as the thinkers in the hierarchy described in Table 6.2. The only change that was made to the systems was in the routing of their outputs. Instead of mapping their actions strings to robot commands, the hierarchical system used the action strings as conditions for a combiner QLCS. Using the two, now static, QLCSs as thinkers, the untrained combiner was given the same task and initial environment (Figure 6.1) as the monolithic QLCS in Section 6.1 and the holosticly shaped DOL system in Section 6.2.3, namely to locate and approach the goal beacon.

Using modular shaping as its reinforcement paradigm, all reinforcements were applied to the Q-values of the combiner's classifiers only. Figure 6.9 shows the learning curve for the DOL architecture using modular shaping. The improvement of the system

from the first trial to the last is easily visible in this plot. As is shown by the learning curve, the system learned a path that required approximately 11 time steps after only 30 trials.

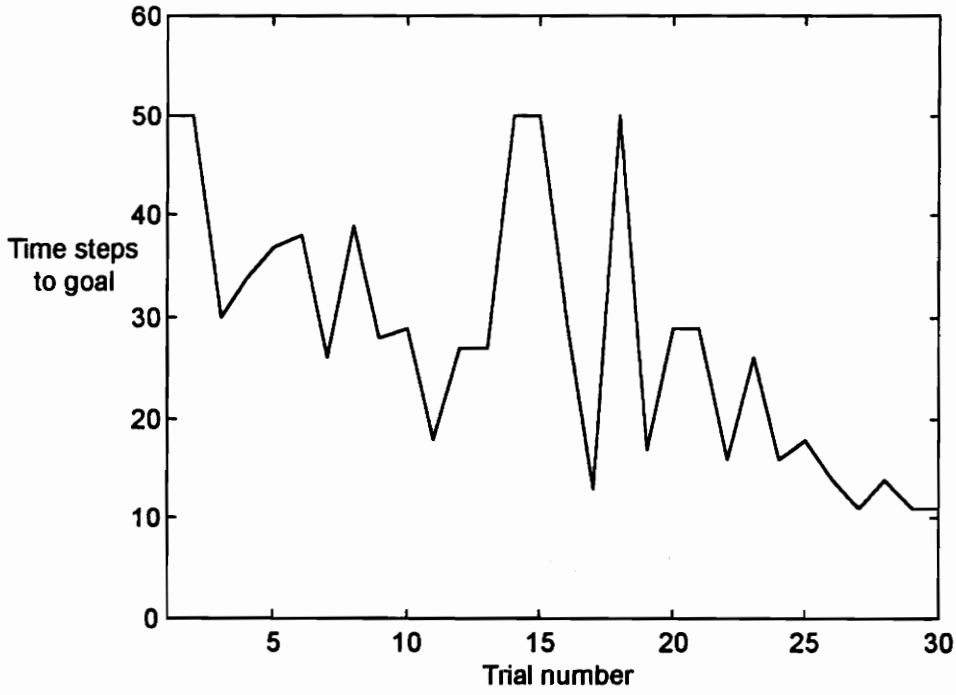


Figure 6.9. Learning Curve for goal seeking behavior using DOL architecture with modular shaping.

6.3 Interpreting Rules and Strengths

Even though the nature of the learning classifier system allows the robot to develop its own set of "good" rules without supervision, it is worthwhile to examine the contents of the knowledge base to determine the meanings of the "good" rules and to compare them to the "bad" rules. Because the rules are represented as binary strings, this is not a straightforward task. We must decode the rules to infer their meanings. Note that

this is an even more complicated task in the case of a hierarchical classifier system, since much of the encoding of real world meaning is evolved by the system, not the programmer. To this end, we examine the resulting rule bases from the learning experiments for the "approach goal" behavior (Section 6.2.2.1) and the "locate goal" behavior (Section 6.2.2.2).

6.3.1 Interpreting the "Approach Goal" Rule Base

A plot of the Q-values resulting from the "approach goal" experiment is shown in Figure 6.10. The rules that comprise the horizontal axis of this plot are explaned in Table 6.3. The information in this plot and table provides some very useful insight into the operation of Q-learning systems. The information in the table is grouped based on the sensor input. Recall the experimental setup:

Starting state:	6-7 units away from goal (Figure 6.4)
Goal state:	1 unit away from goal
Sensor Input:	Distance to goal (first 3 bits of each classifier)
Action Set:	Translations (forward fast, forward slow, stay still, reverse slow)

Because of the ultrasonic sensor's inability to accurately measure very short distances, the robot was physically unable to be 0 units away from the goal (condition 000). Therefore, we may disregard classifiers 1-4.

Classifiers 5-8 are the rules for the goal state. Because the trials were terminated upon reaching the goal state, no action was ever taken from this state. The zero Q-values for these classifiers are a result of them being unused by the system.

The remaining classifiers provide the most useful information about the system. Recall that in implementing our Q-leaning system, we chose a "minimization of costs"

approach rather than a "maximization of rewards" approach. This means that for a given input, the matching classifier with the *lowest* Q-value is believed to be the current best rule for the system. In Figure 6.10, the "troughs" in the plot correspond to the Q-values of these best classifiers. The interpretations of these best rules are highlighted in Table 6.3. As we might have predicted, for each input state the system learned that the best action was the "move forward fast" action.

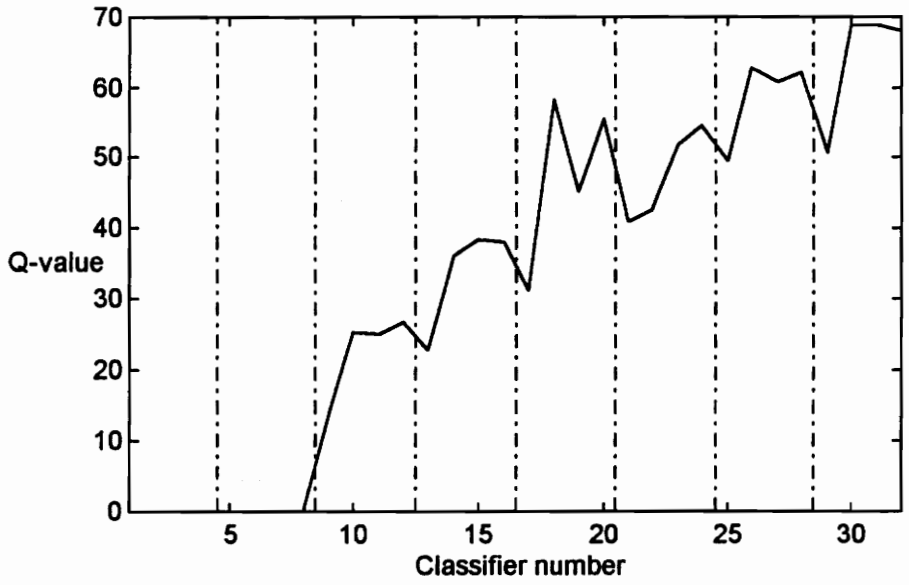


Figure 6.10. Q-value plot for the "approach goal" behavior.

Table 6.3. Rule interpretation for "approach goal" behavior.

Classifier #	Binary Rep.	Interpretation	Q-value
1	000 00	If goal is 0 units away, forward fast	0.00
2	000 01	If goal is 0 units away, forward slow	0.00
3	000 10	If goal is 0 units away, stay still	0.00
4	000 11	If goal is 0 units away, reverse slow	0.00
5	001 00	If goal is 1 unit away, forward fast	0.00
6	001 01	If goal is 1 unit away, forward slow	0.00
7	001 10	If goal is 1 unit away, stay still	0.00
8	001 11	If goal is 1 unit away, reverse slow	0.00
9	**010 00**	**If goal is 2 units away, forward fast**	**13.97**
10	010 01	If goal is 2 units away, forward slow	25.34
11	010 10	If goal is 2 units away, stay still	25.04
12	010 11	If goal is 2 units away, reverse slow	26.71
13	**011 00**	**If goal is 3 units away, forward fast**	**22.85**
14	011 01	If goal is 3 units away, forward slow	36.00
15	011 10	If goal is 3 units away, stay still	38.12
16	011 11	If goal is 3 units away, reverse slow	38.01
17	**100 00**	**If goal is 4 units away, forward fast**	**31.20**
18	100 01	If goal is 4 units away, forward slow	58.16
19	100 10	If goal is 4 units away, stay still	44.94
20	100 11	If goal is 4 units away, reverse slow	55.28
21	**101 00**	**If goal is 5 units away, forward fast**	**40.86**
22	101 01	If goal is 5 units away, forward slow	42.43
23	101 10	If goal is 5 units away, stay still	51.66
24	101 11	If goal is 5 units away, reverse slow	54.45
25	**110 00**	**If goal is 6 units away, forward fast**	**49.47**
26	110 01	If goal is 6 units away, forward slow	62.58
27	110 10	If goal is 6 units away, stay still	60.80
28	110 11	If goal is 6 units away, reverse slow	62.26
29	**111 00**	**If goal is 7 units away, forward fast**	**50.73**
30	111 01	If goal is 7 units away, forward slow	68.77
31	111 10	If goal is 7 units away, stay still	68.80
32	111 11	If goal is 7 units away, reverse slow	67.91

6.3.2 Interpreting the "Locate Goal" Rule Base

In the "locate goal" experiment, the objective was for the robot to learn to use clockwise (CW) and counter-clockwise (CCW) rotations to orient itself in the direction of the goal beacon. The experimetnal setup was as follows:

Starting state:	3 different orientations (Figure 6.7)
Goal state:	Seeing goal beacon with middle goal detector
Sensor Input:	Left, middle, right goal detectors (first 3 bits of each classifier)
Action Set:	Rotations (CW fast, CW slow, CCW fast, CCW slow)

The resulting Q-values for the rules in this experiment are plotted in Figure 6.11 and interpreted in Table 6.4. As in the "approach goal" experiment, some of the sensor inputs covered by these rules were not likely to ever be seen by the system. The inputs of `101` and `111` (rules 21-24 and 29-32, respectively) fall into this category. Both of these inputs require the robot to see the goal with the left and right simultaneously, an unlikely occurrence due to the physical positions of the sensors. Also, because an input of `010` signified that the goal had been reached, none of the rules with this condition (rules 9-12) were ever used.

Rules 1-4 represent the large "`000`" blindspot discussed in Section 5.3.5. Although the best rule according to Table 6.4 calls for a fast counter-clockwise rotation, that carries little merit. As discussed in Section 5.3.5, all of the rules in this case are considered to be equally bad.

Rules 13-16 and 25-28 cover the cases where the beacon is slightly off-center from the robot. As is shown in Figure 5.12, the physical area defined by these sensor values is small in comparison to the other sensor areas. Therefore, it is less likely that the robot received these sensor values as input often. In fact, for the input condition `011` (goal is

ahead and to the right) the rule with the lowest Q-value (rule 13) has a Q-value of zero, indicating that it was never tested by the system. For an input of `110` (goal is ahead and to the left), the best rule calls for a fast clock-wise rotation, a move that is counter-intuitive. Because this system learns by visiting each state multiple times, it is possible that these states were not explored enough in the experiment for the correct rules to emerge.

For the more common environment states of `001` and `100`, the system did learn the correct rules. If the goal beacon is seen by the right beacon detector (condition is `001`), the best action is a fast clockwise rotation (rule 5). If the goal beacon is seen only by the left detector (input condition of `100`), the best rule is to make a fast counter-clockwise rotation (rule19). These two conditions are basically mirror images of each other. The robot is required to rotate approximately the same amount in each case to reach the goal state. This similarity is even reflected in the Q-values for the various possible movements from these environmental states. On the Q-value plot (Figure 6.11), these two conditions are represented by the plateaus for classifiers 5-8 and 17-20. Both of these plateaus are located in the 150 Q-value area.

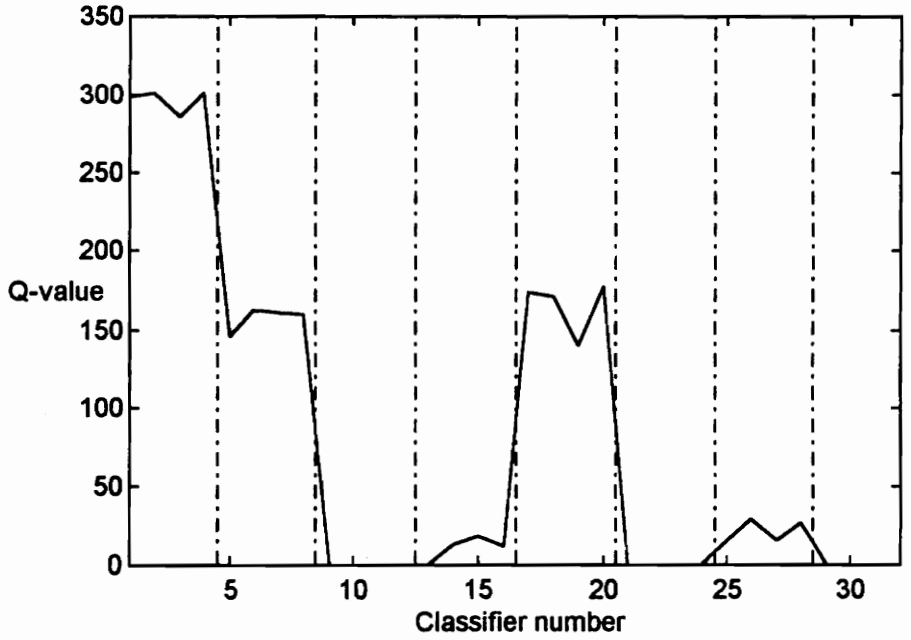


Figure 6.11. Q-value plot for the "approach goal" behavior.

Table 6.4. Rule interpretation for "locate goal" behavior.

Classifier #	Binary Rep.	Interpretation	Q-value
1	000 00	If goal is not seen, turn CW fast	299.42
2	000 01	If goal is , turn CW slow	301.50
3	**000 10**	**If goal is not seen, turn CCW fast**	**286.20**
4	000 11	If goal is not seen, turn CCW slow	301.63
5	**001 00**	**If goal is at right, turn CW fast**	**145.56**
6	001 01	If goal is at right, turn CW slow	162.36
7	001 10	If goal is at right, turn CCW fast	161.35
8	001 11	If goal is at right, turn CCW slow	159.17
9	010 00	If goal is straight ahead, turn CW fast	0.00
10	010 01	If goal is straight ahead, turn CW slow	0.00
11	010 10	If goal is straight ahead, turn CCW fast	0.00
12	010 11	If goal is straight ahead, turn CCW slow	0.00
13	**011 00**	**If goal is ahead and right, turn CW fast**	**0.00**
14	011 01	If goal is ahead and right, turn CW slow	12.39
15	011 10	If goal is ahead and right, turn CCW fast	17.54
16	011 11	If goal is ahead and right, turn CCW slow	11.43
17	100 00	If goal is at left, turn CW fast	173.50
18	100 01	If goal is at left, turn CW slow	171.68
19	**100 10**	**If goal is at left, turn CCW fast**	**138.73**
20	100 11	If goal is at left, turn CCW slow	177.95
21	101 00	If goal is right and left, turn CW fast	0.00
22	101 01	If goal is right and left, turn CW slow	0.00
23	101 10	If goal is right and left, turn CCW fast	0.00
24	101 11	If goal is right and left, turn CCW slow	0.00
25	**110 00**	**If goal is ahead and left, turn CW fast**	**15.05**
26	110 01	If goal is ahead and left, turn CW slow	29.63
27	110 10	If goal is ahead and left, turn CCW fast	15.96
28	110 11	If goal is ahead and left, turn CCW slow	27.21
29	111 00	If goal is not seen, turn CW fast	0.00
30	111 01	If goal is not seen, turn CW slow	0.00
31	111 10	If goal is not seen, turn CCW fast	0.00
32	111 11	If goal is not seen, turn CCW slow	0.00

6.4 Discussion of Results

From a comparison of the performances of the three different classifier systems on this sample problem, we can definitely see that the DQLCS with modular shaping was best suited *for this problem*. There are several characteristics of each system that qualify it as a good or bad choice for this sort of application, including knowledge base size and shaping technique. While it may not be possible to conclude that these characteristics determined the outcomes of these experiments, they do offer a good basis for speculation about each system's performance.

It appears that the shear size of the knowledge base in the monolithic system was its downfall. In the number of time steps allowed for this experiment, there simply were not enough visits to each state for an incremental learning algorithm to be successful. Since there were 16 possible action sequences (4 bits) for each state of the environment, it is quite possible that several actions were never tried.

From the results, it appears that the monolithic system did outperform the DOL system that used holistic shaping, however. Although this is somewhat surprising, it is not at illogical. Holistic shaping has a fundamental flaw. All learning robots with simple sensors suffer from environmental ambiguity. A straightforward example of this occurred often during these experiments. If the robot finds itself near a wall, but unable to see a goal, it has no other sensors to distinguish one section of wall from another. This environmental ambiguity is a well documented problem in robot learning termed *perceptual aliasing* by Whitehead and Ballard [1]. While this ambiguity occurs in all of the systems, holistic shaping has an internal ambiguity all its own. It appears that reinforcement is assigned to the thinker classifiers in a manner that is much too haphazard. The learning process is slow; therefore, overcoming misguided reinforcement often takes too long for practical real robot applications.

It is usually easier to explain why something fails than it is to explain how something else succeeds. In this case, however, there is a probable explanation for the

success of the modular shaping DOL system. That explanation is that it is exactly the polar opposite of the holistic shaping DOL system. Because each component learns separate from the rest, there is none of the ambiguity associated with holistic shaping. Unfortunately, these same characteristics that allow modular shaping to excel in this experiment doom it to failure in many others. The simple assertion that a complex behavior must be decomposed into several disjoint base behaviors is absurd in most natural systems. In fact, this system almost borders on the same problems that simulations have – too much domain knowledge. To use the modular shaping procedure, problem domains must be somewhat contrived.

7. Conclusions and Suggestion for Further Research

In this thesis, we have examined the problems of applying reinforcement learning systems to real robot learning problems. To facilitate our study of robot learning, we chose the learning classifier system (LCS). In deciding how to best implement the LCS on a real robot, we first examined some of the previously discovered difficulties with the original LCS. Previous research in the field indicated that the apportionment of credit component of the LCS, the bucket brigade algorithm, was too weak to converge to an optimal policy quickly and often required the infusion of domain knowledge to achieve desirable performance. As an alternative to the BBA, we presented the simpler, less restricting Q-learning algorithm for the AOC component of our learning system. While this new system, the QLCS, was an improvement, we realized that it was not powerful enough to overcome the curse of dimensionality and learn task solutions in a reasonable time period. To overcome this problem, we incorporated the principle of task decomposition into our system and created the distributed Q-learning classifier system with the goal of distributing the learning problem across small independent learning modules. To allow for the study of the DQLCS in real robot learning problems, a sensing system and computing architecture were designed and implemented on a mobile robot base.

After its hardware implementation, the performances of the QLCS and DQLCS were observed for a goal seeking learning problem. The results of these experiments lead us to two conclusions. First, the performances of the QLCSs show that Q-learning is an acceptable alternative to the BBA as the apportionment of credit component of a learning classifier system. The main conclusion in this thesis may be drawn from a comparison of the performances of the various Q-learning systems we have studied. The results from the DQLCS experiments indicate that the distributed architecture using modular shaping is

more time efficient than a distributed architecture using holistic shaping or a monolithic architecture at solving tasks that require complex behaviors. There are advantages and disadvantages to each of the three approaches to learning problem we have presented. When weighing one against the other, we must consider much more than the performance of each system on a given problem. We must also consider the limitations of each system. While the DQLCS with modular shaping far outperformed the DQLCS with holistic shaping, it also required the infusion of much more domain specific knowledge. This requirement limits its application to a considerably smaller class of problems than may be attempted by the DQLCS with holistic shaping.

At a higher level, much of the reward and blame for the performances of these systems can be placed on the learning classifier system itself. Like all reinforcement learning algorithms, it has several advantages and disadvantages. While its modularity makes it a very attractive base for studying distributed learning, its lack of a structural credit assignment component causes learning to proceed much too slowly. Without adding too much domain knowledge, the QLCS and DQLCS could both benefit from some means of assigning credit or punishment to classifiers that closely match the active classifier during each time step. Another fundamental problem of the classifier system, especially the distributed classifier system, is that there is almost a *curse of dimensionality* in its configurability alone. There are infinitely many combinations of cost functions and learning factors that can be used on a single problem. Unfortunately, it has been shown that the success or failure of these systems is highly dependent on the values of the learning factors [12, 20].

In the research presented in this thesis, the greatest unknown in the area of classifier systems was found in the area of distributed systems. At this point, it appears that a random choice of action string lengths for thinker QLCSs is as good as any. Since these strings are internally used by the system and never obtain any real-world meaning, it is nearly impossible to evaluate them. In the effort to decrease the problem size, it is always appealing to make these as small as possible. At the same time, however, we do

not want to make them so small as to limit their ability to communicate information to the combiner QLCS. It appears that classifier systems are useful in distributed learning problems.

If there is one conclusion we may draw from this study of distributed learning systems, it is that there is still much room for research. This is especially true in the area of shaping methods. In our work, we presented two shaping methods that should be at opposite ends of a spectrum of credit assignment methods. Unfortunately, these two methods are the only methods available. The holistic shaping method is far too general, while the modular method is far too specific. For a learning system to be capable of learning a variety of complex behaviors in a reasonable amount of time, new shaping methods that fall between the all or nothing approaches of holistic and modular will have to be developed.

Appendix

A. Circuit Schematics

Circuit schematics are provided for the goal beacon circuit, the beacon detector circuit, and the serial communications switching circuit.

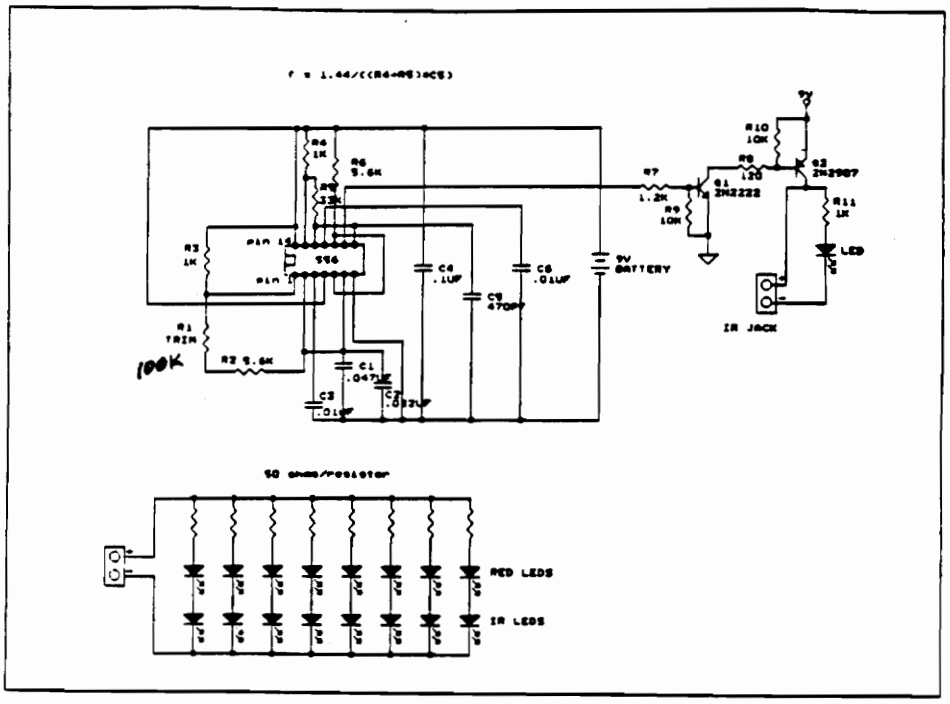


Figure A.1. Infrared goal beacon circuit schematic.

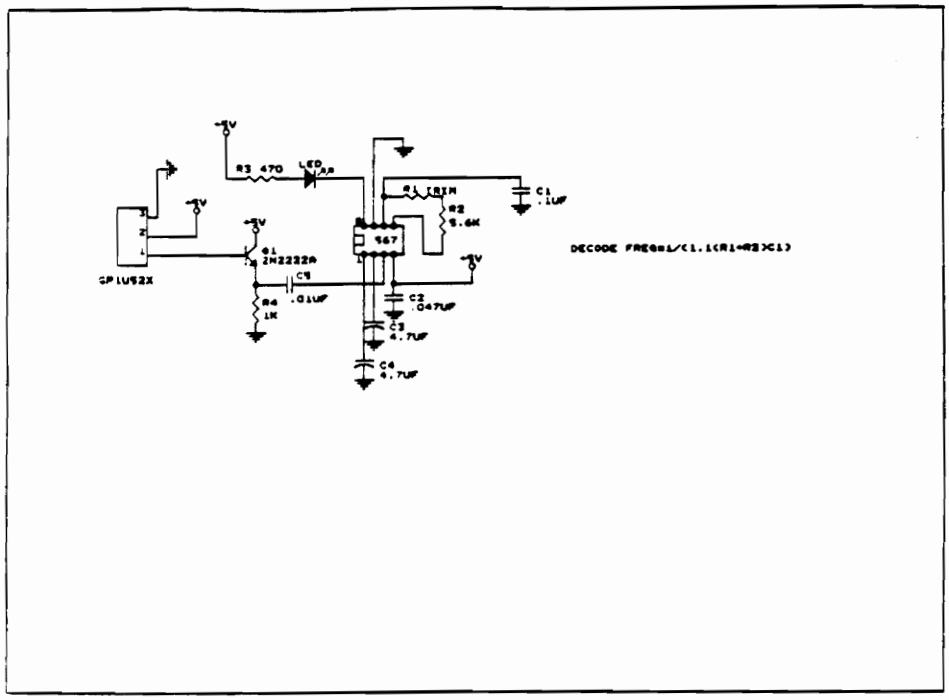


Figure A.2. Infrared beacon detector circuit schematic

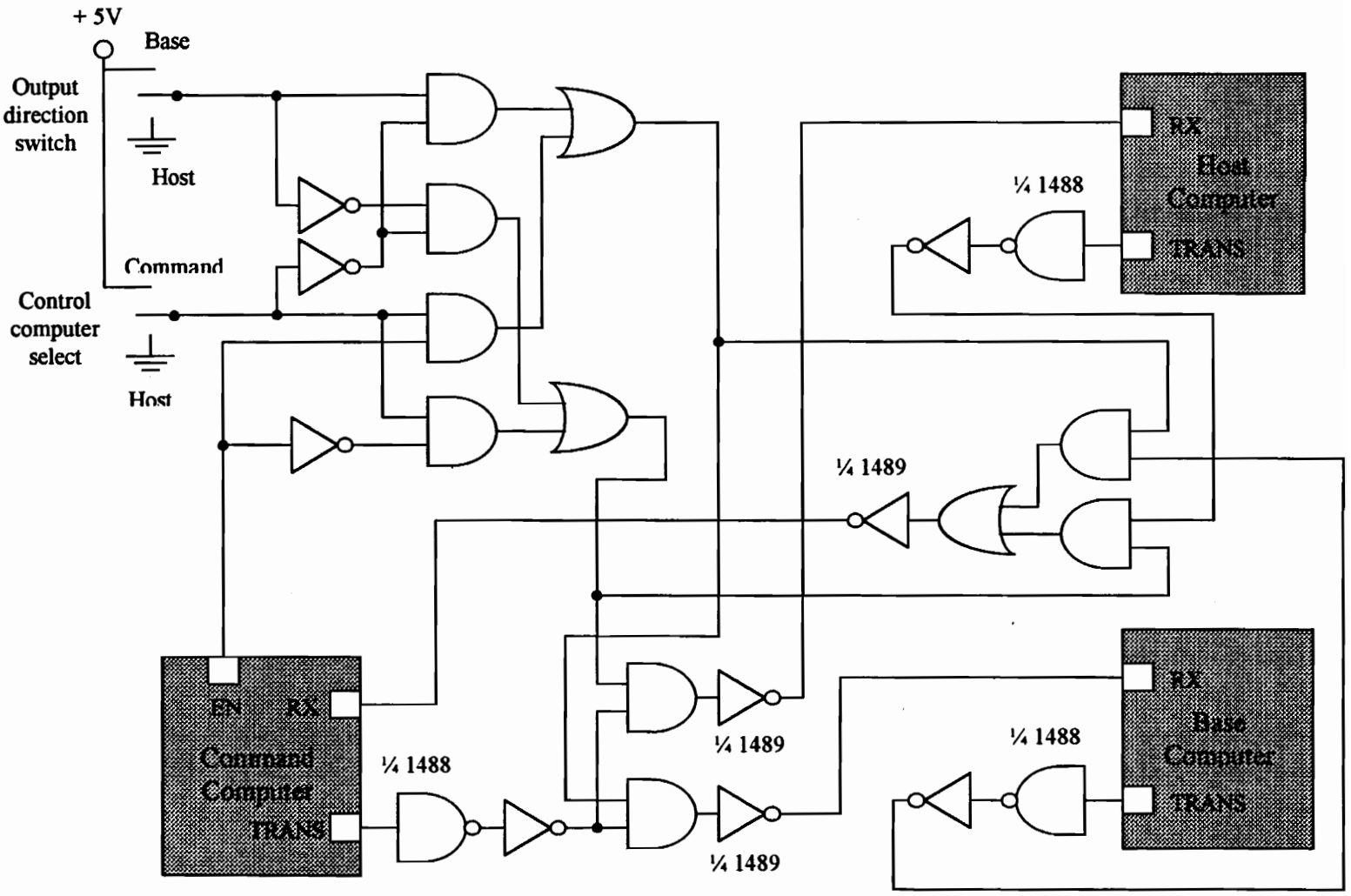


Figure A.3. Serial communications switching circuit.

References

[1] Ballard, D.H., Whitehead, S.D., "Active Perception and Reinforcement Learning," *Proceedings of the 7th International Conference on Machine Learning*, 1990.

[2] Bay, J.S., Stanhope, J.D., "Distributed Optimization of Tactical Actions by Mobile Intelligent Agents," *Journal of Robotic Systems, Special Issue on Mobile Robots*, 1996.

[3] Belew, R.K., Forrest, S., "Learning and Programming in Classifier Systems," *Machine Learning Volume 3*, pp. 193-223, Netherlands: Kluwer Academic publishers, 1988.

[4] Booker, L.B., Goldberg, D.E., Holland, J.H., "Classifier Systems and Genetic Algorithms," *Artificial Intelligence*, Vol. 40, no. 3, pp. 235-282, September 1989.

[5] Brady, M., "Artificial Intelligence and Robotics," *Artificial intelligence*, 1985.

[6] Brooks, R.A., Mataric, M.J., "Real Robots, Real Learning Problems," *Robot Learning*, Boston: Kluwer Academic Publishers, 1993.

[7] Cobb, H.G., Grefenstette, J.J., "Learning the Persistence of Actions in Reactive Control Rules", *Proceedings of the 8th International Conference on Machine Learning*, L. Birnbaum and G. Collins, eds, 1991.

[8] Connell, J.H., Mahadevan, S., eds., *Robot Learning*, Boston: Kluwer Academic Publishers, 1993, preface.

[9] Connell, J.H., Mahadevan, S., "Introduction to Robot Learning," *Robot Learning*, Boston: Kluwer Academic Publishers, 1993.

[10] Connell, J.H., Mahadevan, S., "Rapid Task Learning for Real Robots," *Robot Learning*, Boston: Kluwer Academic Publishers, 1993.

[11] Dorigo, M., "ALECSYS and the AutonoMouse: Learning to Control a Real Robot by Distributed Classifier Systems," Machine Learning, vol. 19, no. 3, pp. 209-240, 1995.

[12] Gaff, D.G., *Architecture Design and Simulation for Distributed Learning Classifier Systems*, MS Thesis, Bradley Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, May 1995.

[13] Garvey, T.D., Lowrance, J.D., Fischler, M.A., "An Inference Technique for integrating Knowledge from Disparate Sources," *Proceedings of the 7th International Conference on Artificial Intelligence*, pp. 319-325, 1981.

[14] Holland, J.H., "A Mathematical Framework for Studying Learning in Classifier Systems," *Physica*, 1986, pp. 307-317.

[15] Holland, J.H., *etal., Induction: Processes of Inference, Learning, and Discovery*, Cambridge: MIT Press, 1986.

[16] Larson, R.E., Casti, J.L., *Principles of Dynamic Programming, Part 1*, New York: Marcel Dekker, Inc., 1978, pp. 54-61.

[17] Lin, L, "Programming Robots Using Reinforcement Learning and Teaching," *Proceedings of the 9th American Association for Artificial Intelligence National Conference on Artificial Intelligence*, July 1991.

[18] McKerrow, P.J., *Introduction to Robotics*, New York: Addison-Wesley Publishing Company, 1991, pp. 10-17.

[19] Mahadevan, S., Connell, J., "Automatic Programming of Behavior-based Robots Using Reinforcement Learning," *Proceedings of the 9th American Association for Artificial Intelligence National Conference on Artificial Intelligence*, July 1991.

[20] Mataric, M.J. "A Comparative Analysis of Reinforcement Learning Methods," A.I. Memo No. 1322, Massachusetts Institute of Technology Artificial Intelligence Laboratory, October 1991.

[21] Polaroid Corporation, *Ultrasonic Ranging Systems*, 1984.

[22] Real World Interfaces, Inc., "B12 Mobile Robot Base Guide to Operations, Version 2.1," Dublin, Ohio: Real World Interface, Inc.

[23] Rich, E., Knight, K., *Artificial Intelligence*, New York: McGraw-Hill, Inc., 1991.

[24] Robertson, G.G., Riolo, R.L., "A Tale of Two Classifier Systems," *Machine Learning Volume*, Boston: Kluwer Academic Publishers, 1988.

[25] Singh, S.P. "Transfer of learning Across Compositions of Sequential Tasks," *Proceedings of the 8th International Workshop on Machine Learning*, L. Birnbaum and G. Collins, eds, 1991.

[26] Sutton, R.S., "Integrate Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming," *Proceedings of the 7th International Conference on Machine Learning*, R. Mooney and B. Porter, eds, 1990.

[27] Sutton, R.S., "Planning by Incremental Dynamic Programming," *Proceedings of the 8th International Workshop on Machine Learning*, L. Birnbaum and G. Collins, eds, 1991.

[28] Tan, M., "Learning a Cost Sensitive internal Representation for Reinforcement Learning," *Proceedings of the 8th International Workshop on Machine Learning*, L. Birnbaum and G. Collins, eds, 1991.

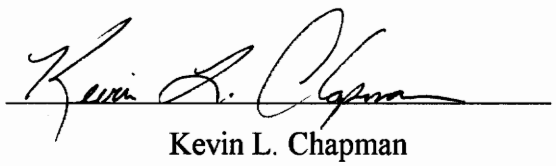
[29] Watkins, C.J..C.H, Dayan, P., "Q-Learning," *Machine Learning*, vol. 8, 1992, pp. 55-68.

[30] Whitehead, S., Karlsson, J., Tenenberg, J., "Learning Multiple Goal Behavior Via Task Decomposition and Dynamic Policy Merging," *Robot Learning*, Boston: Kluwer Academic Publishers, 1993.

[31] Wilson, S.W., "Classifier Systems and the Animat Problem" in *Machine Learning Volume*, Boston: Kluwer Academic Publishers, 1988.

VITA

Kevin Lynn Chapman was born in the rural community of Wallace, South Carolina, on March 28, 1972. At an early age, Kevin's deep interest in mechanical systems and problem solving surfaced as a simple desire to disassemble and study everything in his path. While finishing high school at the South Carolina Governor's School for Science and Mathematics, a computer science course created an equally strong interest in computer programming. Following high school, Kevin began study at the University of South Carolina in Columbia, South Carolina. While obtaining a Bachelor of Science degree in computer engineering from USC, he discovered an application of his childhood interests through his participation in a variety of robotics and artificial intelligence projects. Kevin continued working in these areas throughout his pursuit of a Master of Science degree in electrical engineering at Virginia Polytechnic Institute and State University in Blacksburg, Virginia. Kevin is currently a member of the Intelligent Decision Support Systems group at Raytheon E-Systems Corporation in Falls Church, Virginia. Whenever he's not exploring the wonderful world of artificial intelligence at work or at school, Kevin enjoys mountain biking, running, back packing, and snow skiing.



Kevin L. Chapman