LARGE-SCALE DYNAMIC OPTIMIZATION USING TEAMS OF REINFORCEMENT LEARNING AGENTS

A Dissertation Presented

by

ROBERT HARRY CRITES

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1996

Computer Science

LARGE-SCALE DYNAMIC OPTIMIZATION USING TEAMS OF REINFORCEMENT LEARNING AGENTS

A Dissertation Presented

by

ROBERT HARRY CRITES

Approved as to style and content by:

Andrew G. Barto, Chair

Victor R. Lesser, Member

Roderic A. Grupen, Member

Christos G. Cassandras, Member

David W. Stemple, Department Chair
Computer Science

ACKNOWLEDGEMENTS

So many people have contributed to my growth during my work on this thesis that I am sure I will not remember to name them all.

My advisor Andy Barto has been a boundless resource, giving me much needed freedom while also providing superb guidance about interesting research areas. I have benefitted tremendously from his wide ranging expertise and good judgement. His striving for excellence has inspired me and shown me the type of researcher I would like to be. My committee members, Victor Lesser, Rod Grupen, and Christos Cassandras, have also played a crucial role in shaping this dissertation by asking me questions that helped to clarify some important issues. I have also learned a great deal from discussions with past and present members of the adaptive networks group at UMASS, including Steve Bradtke, Mike Duff, Andrew Fagg, Vijay Gullapalli, Sergio Guzman-Lara, John McNulty, Satinder Singh, Rich Sutton, and Richard Yee. Christos Cassandras kindly provided the elevator simulator developed by his group. Asif Gandhi helped me to get the simulator running and answered many questions that enabled me to perform fair comparisons with a wide variety of algorithms. John McNulty helped write some of the simulation routines, and Dave Pepyne also contributed to my understanding of the elevator dispatching problem.

My thesis support group, Jeff Clouse, Carla Brodley, Joe McCarthy, Dorothy Mammen, and Malini Bhandaru provided pep talks during some of the most difficult times. Many friends from Agape Community Church, UMASS Graduate Christian Fellowship, and Cedar Hill Mennonite Church also eased my burdens on many occasions. I think particularly of Holly Shriver, Liz Spencer, Mary DiMarco, Anand,

Amalia, and Gita Gnanadesikan, and especially Jose Antonio Medina-Peralta and Corinna Jaudes for their wonderful hospitality to me on my visits from Pennsylvania.

My family has also been a great blessing and source of joy to me over the years. My deepest thanks to them for always being there for me. Finally, thanks to my wife Dana for her patience and encouragement. Now we can be a "paradox", and hopefully have some "doc–lings", too.

ABSTRACT

LARGE-SCALE DYNAMIC OPTIMIZATION USING TEAMS OF REINFORCEMENT LEARNING AGENTS

SEPTEMBER 1996

ROBERT HARRY CRITES

A.B., PRINCETON UNIVERSITY

M.S., JOHNS HOPKINS UNIVERSITY

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Andrew G. Barto

Recent algorithmic and theoretical advances in reinforcement learning (RL) are attracting widespread interest. RL algorithms have appeared that approximate dynamic programming (DP) on an incremental basis. Unlike traditional DP algorithms, these algorithms do not require knowledge of the state transition probabilities or reward structure of a system. This allows them to be trained using real or simulated experiences, focusing their computations on the areas of state space that are actually visited during control, making them computationally tractable on very large problems. RL algorithms can be used as components of multi-agent algorithms. If each member of a team of agents employs one of these algorithms, a new *collective* learning algorithm emerges for the team as a whole. In this dissertation we demonstrate that such collective RL algorithms can be powerful heuristic methods for addressing large–scale control problems.

Elevator group control serves as our primary testbed. The elevator domain poses a combination of challenges not seen in most RL research to date. Elevator systems operate in continuous state spaces and in continuous time as discrete event dynamic systems. Their states are not fully observable and they are non-stationary due to changing passenger arrival rates. As a way of streamlining the search through policy space, we use a team of RL agents, each of which is responsible for controlling one elevator car. The team receives a global reinforcement signal which appears noisy to each agent due to the effects of the actions of the other agents, the random nature of the arrivals and the incomplete observation of the state. In spite of these complications, we show results that in simulation surpass the best of the heuristic elevator control algorithms of which we are aware. These results demonstrate the power of RL on a very large scale stochastic dynamic optimization problem of practical utility.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Multiple Agents

An agent is an entity that has the power to act [36]. An agent is rarely found in isolation. Its environment usually contains other agents, which is one reason why the world is such a dynamic place. Multi-agent systems pervade human experience on many different levels. On a macroscopic level, we function as agents in a variety of social and economic systems. Taking a more microscopic view, living organisms that we normally think of as single agents can also be viewed as multi-agent systems. The human nervous system is an extremely complex multi-agent system. Each neuron in the brain is a decision maker acting on its own information, and having some effect on the performance of the system as a whole. None of them has access to complete information, and they must adapt to each other and to the external environment in order to properly coordinate their actions.

Multi-agent systems are important to study for a number of reasons, both in terms of the analysis of existing systems, and in terms of the more effective synthesis of new systems in such fields as distributed computation, decentralized control, organization design, distributed artificial intelligence, and artificial neural networks. Multi-agent systems are often required because of spatial or geographic distribution, or in situations where centralized information is not available or is not practical. But even when a distributed approach is not required, multiple agents may still provide an excellent way of scaling up to approximate solutions for very large problems by streamlining the search through the space of possible policies. Dawkins [38] argues that decisions

based on local information are most efficient if made locally. Chandrasekaran [27] notes that for "complex information processing systems involving very large numbers of sensors and effectors, a central processor will require very large bandwidths for responding to sensors or activating effectors." As he says, it is difficult to imagine "an army whose commanding general alone is authorized to make all the field decisions." Tsetlin [113] makes the same observation: "If one assumes that all control proceeds from the top down to a specific address, then the system becomes very complex.... but if the conditions of a game are given, then the automata find the required actions by themselves. In this case they do not need individual commands." Additional benefits of decentralization include the possibility of increased speed through parallelism, often at lower cost than in comparable centralized systems. Multi-agent systems tend to be easier to extend, since new agents can often be added with mostly local changes. Multiple agents may also provide increased reliability and fault tolerance.

This dissertation focuses on *teams* of agents that share *identical* objectives corresponding directly to the goals of the system as a whole. Though it may be possible to synthesize a system whose goals can be achieved by agents with *conflicting* objectives, such a design task would be more complicated and is less well understood.

Multi-agent systems are often categorized by the sophistication of their constituent agents. The distributed artificial intelligence (DAI) research community [58, 59, 54] has generally focused on building systems that employ very sophisticated local decision makers. They assume that each decision maker is an intelligent problem solver in its own right, and that coordination among the problem solvers is based on a large amount of high-level knowledge, including information about the plans of the other agents, as well as the goals of the entire system. There also tends to be less of an emphasis on learning in such systems, though this is starting to change [122]. Artificial life and behavior-based AI researchers have generally taken the opposite approach,

combining large numbers of relatively unsophisticated agents in a bottom-up manner, and seeing what emerges when they are put together into a group [72, 73].

The elevator control application developed in this dissertation employs relatively unsophisticated, homogeneous agents that do not communicate. However, these characteristics should not be interpreted as restrictions. The reinforcement learning (RL) techniques described in this dissertation are applicable to a wide variety of agents, including more sophisticated agents, heterogeneous agents, and agents that communicate.

1.2 Reinforcement Learning

Interest in developing capable learning systems has increased recently within the AI research community. Learning enables systems to be more flexible and robust, and it makes them better able to handle uncertainty and changing circumstances. This is especially important in multi-agent systems, where learning has often been neglected. The *designers* of multi-agent systems have generally taken on the extremely difficult task of trying to anticipate all possible contingencies and interactions among the agents ahead of time. Much the same could be said concerning the field of decentralized control, where policies for the control stations are developed from a global vantage point, and learning does not play a role. Even though *executing* the policies depends only on the information available at each control station, the policies are *designed* in a centralized way, with access to a complete description of the problem. Research has focused on what constitutes an optimal policy under a given information pattern but not on how such policies might be *learned* under the same constraints.

Both symbolic and connectionist learning researchers have focused primarily on supervised learning, where a "teacher" provides the learning system with a set of training examples in the form of input-output pairs. Supervised learning techniques are useful in a wide variety of problems involving pattern classification and function

approximation. However, there are many situations in which training examples are costly or even impossible to obtain. RL is applicable in these more difficult situations, where the only help available is a "critic" that provides a scalar evaluation of the output that was selected, rather than specifying the best output or a direction of how to change the output. In RL, one faces all the difficulties of supervised learning combined with the additional difficulty of exploration, that is, determining the best output for any given input. RL applies naturally to the case of an autonomous agent, which receives sensations as inputs, and takes actions that affect its environment in order to achieve its own goals. This framework is appealing from a biological point of view, since an animal has certain built-in preferences (such as pleasure or pain), but does not always have a teacher to tell it exactly what action it should take in every situation.

RL is based on the idea that the tendency to produce an action should be strengthened (*reinforced*) if it produces favorable results, and weakened if it produces unfavorable results. RL tasks can be divided naturally into two types. In non-sequential tasks, the agent must learn a mapping from situations to actions that maximizes the expected immediate payoff. In sequential tasks, the agent must learn a mapping from situations to actions that maximizes the expected long-term payoffs. Sequential tasks are more difficult because the actions selected by the agent may influence its future situations and thus its future payoffs. In this case, the agent interacts with its environment over an extended period of time, and it needs to evaluate its actions on the basis of their long-term consequences.

From the perspective of control theory, RL techniques are ways of finding approximate solutions to stochastic optimal control problems. The agent is the controller, and the environment is the system to be controlled. The objective is to maximize some performance measure over time. Given a model of the state transition probabilities and reward structure of the environment, these problems can be solved in

principle using dynamic programming (DP) algorithms. However, even though DP only requires time that is polynomial in the number of states, in many problems of interest, there are so many states that the amount of time required for a solution is infeasible. Some recent RL algorithms have been designed to perform DP in an incremental manner. Unlike traditional DP, these algorithms do not require knowledge of the state transition probabilities and reward structure of the environment and can be used to improve performance *on-line* while interacting with the environment. This on-line learning focuses computation on the areas of state space that are actually visited during control. Thus, these algorithms are a computationally tractable way of approximating DP on very large problems.

The same focusing phenomenon can also be achieved with *simulated* online training. One can often construct a simulation model without ever explicitly determining the state transition probabilities for an environment [13, 34]. (For an example of such a simulation model, see section 4.3.) There are several advantages to this use of a simulation model if it is sufficiently accurate. It is possible to generate huge amounts of simulated experience very quickly, potentially speeding up the training process by many orders of magnitude over what would be possible using actual experience. In addition, one need not be concerned about the performance level of a simulated system during training. A successful example of simulated online training is found in Tesauro's TD-Gammon system [108, 109, 110], which used RL techniques to learn to play strong master–level backgammon.

RL algorithms can be used as components of multi-agent algorithms. If the members of a group of agents each employs one of these algorithms, a new *collective* algorithm emerges for the group as a whole. This type of collective algorithm allows control policies to be learned in a decentralized way. Even in situations where centralized information is available, it may be advantageous to develop control policies in a decentralized way in order to simplify the search through policy space.

To demonstrate the power of multi-agent RL, we focus on the problem of elevator group supervisory control. The elevator domain poses a combination of challenges not seen in most RL research to date. Elevator systems operate in continuous state spaces and in continuous time as discrete event dynamic systems. Their states are not fully observable and they are non-stationary due to changing passenger arrival rates. We use a team of RL agents, each of which is responsible for controlling one elevator car. The team receives a global reinforcement signal which appears noisy to each agent due to the effects of the actions of the other agents, the random nature of the arrivals and the incomplete observation of the state. In spite of these complications, we show results that in simulation surpass the best of the heuristic elevator control algorithms of which we are aware. These results demonstrate the power of multi-agent RL on a very large scale stochastic dynamic optimization problem of practical utility.

1.3 Summary of Contributions

The field of reinforcement learning is currently facing the challenge of moving from theory to real-world applications. This dissertation describes the application of RL to the difficult problem of elevator dispatching. This is one of the first successful large–scale applications of RL, and it forms an existence proof that such applications are feasible. It involves several extensions over the usual RL framework, including the control of a continuous-time discrete-event system, the use of multiple agents, and the need for asynchronous decision-making. A number of practical issues are addressed, and information is presented that should be of use in future applications of RL. Existing discrete event RL algorithms are extended to accomodate continuously changing instantaneous costs. The performance of parallel and decentralized RL algorithms and the effects of varying levels of incomplete state information are also analyzed. By displaying successful performance in learning elevator dispatching, this

work demonstrates that collective RL algorithms are powerful heuristic methods for addressing complex multi-agent control problems.

1.4 Organization of the Dissertation

Chapter 2 discusses a variety of perspectives on multi-agent systems, including some definitions from game theory and viewpoints from AI and control theory. Chapter 3 provides an introduction to RL and reviews its previous applications to multi-agent systems. The problem of elevator group control is discussed in chapter 4, including a description of the particular testbed used in this dissertation. A collective discrete-event RL algorithm is introduced in chapter 5, with parallel and distributed architectures and two methods for calculating reinforcements. Chapter 6 summarizes and analyzes the results. Areas for future work are outlined in chapter 7. Finally, chapter 8 draws some conclusions.

CHAPTER 2

PERSPECTIVES ON MULTIPLE AGENTS

A variety of disciplines have contributed to the study of multi-agent systems. Researchers in distributed artificial intelligence (DAI) have focused on building intelligent agents and exploring how such agents can achieve coordinated behavior. Coordination in DAI systems is usually achieved by having each agent perform sophisticated reasoning based on a large amount of knowledge, including information about the plans and goals of the other agents. One drawback to such an approach is the extraordinary complexity of designing such agents.

The artificial life community has taken the opposite approach, combining large numbers of relatively unsophisticated agents in a bottom-up manner and seeing what emerges when they are put together into a group. This amounts to a sort of iterative procedure: designing a set of agents, observing their group behavior, and repeatedly adjusting the design and noting its effect on group behavior. Some researchers have taken a behavior-based approach to the bottom-up design of multi-agent systems, equipping agents with simple behaviors such as *following* or *avoidance*. However, there is little evidence as yet that such bottom-up designs really provide a simple way of achieving complex specific goals.

Control theorists take a more rigorous, mathematical approach to the design of multi-agent systems. Strategies for the agents are developed top-down from a global vantage point, and learning does not play a role. However, all but the simplest decentralized control problems are intractable using classical control techniques.

The most promising approach may be to build a system with agents that can learn. With adaptive agents, the system designer need not anticipate and solve all possible contingencies during the design phase. RL techniques attempt to combine the principled nature of control theory with the satisficing, heuristic nature of AI.

This chapter begins with some definitions from game theory that provide a useful framework for dealing with multiple agents. Following that is a discussion of a number of multi-agent perspectives from AI and control theory and their relationship to the RL perspective.

2.1 Game Theory

A game can be thought of as an interaction between multiple decision-makers. (We can consider *nature* or *chance* to be one of the decision-makers). Each decision point in a game is called a *move*. It is possible to describe a game by constructing a game tree, where each move is a node of the tree, each possible decision is a branch, and each possible outcome of the game is a leaf of the tree. A game described in this way is said to be in *extensive* form. We can also define a *strategy* as prescribing an action for every possible situation (this corresponds to a policy in control theory). Then it becomes possible to reduce games in extensive form to *strategic* or *normal* form, where each player has only one move, namely, the choice of a strategy. The outcome or payoff for each player depends on the strategy choices of all the players. A game matrix is often used to describe the payoff structure of a game in strategic form. The matrix has as many dimensions as there are players, and its payoff entries are indexed by the strategy choices of the players.

Strategy A *dominates* strategy B when it leads to at least as good an outcome as strategy B regardless of the strategies chosen by the other players. A combination of strategy choices is called an *equilibrium point* when none of the players can benefit by unilaterally changing its strategy.

Games are often categorized on the basis of their performance criteria, what information is available to the players, and whether pre-game cooperation is allowed.

2.1.1 Performance Criteria

In game theory, a distinction is made between zero-sum and non-zero-sum games. *Zero-sum* games are games of pure competition where the payoffs in every possible outcome sum to zero, i.e., where an increase in the payoff of one player means a corresponding decrease in the payoffs of the other players. *Non-zero-sum* games are not strictly competitive and so there may be some outcomes where the payoffs are better or worse for all the players. *Team problems* are a sub-category of non-zero-sum games where the players always receive identical payoffs.

In team problems, there is never any discrepancy between the preferences of the individuals and those of the team as a whole. What is good for one player is always good for the entire team. However, in non-zero-sum games, there is not always even a clear definition of what is best for the players as a whole. Even in cases where such a definition is available, it may be difficult for the players to achieve since they are working for their own interests. For example, consider the Prisoner's Dilemma, where there are two players, and each must select one of two actions: "cooperate" or "defect". No communication is allowed. The payoffs are as follows: 3 points for both players if they both cooperate, 1 point for both players if they both defect, and if one cooperates and one defects, the defector gets 5 points and the cooperator gets 0 points. These can be shown using the payoff matrix in table 2.1.

In team problems, equilibrium points are locally optimal, that is, they provide the best payoff within their row and column. But in non-zero-sum games in general, the situation is more perverse: equilibrium points may actually be quite bad for all the players. There is only one equilibrium point in the Prisoner's Dilemma, and it is the

Table 2.1 *Payoff matrix for the Prisoner's Dilemma.*

		Player 2	
		cooperate	defect
Player 1	cooperate	$(3,3)$	$(0,5)$
	defect	$(5,0)$	$(1,1)$

strategy combination which is the *worst* for the players as a whole, namely, where both defect.

2.1.2 Information Availability

Harsanyi [46] makes a distinction between imperfect and incomplete information. *Imperfect* information refers to a lack of knowledge the players have about any previous moves (including their own, those of the other players, or any chance moves). An example of this is a card game where the deck is shuffled and placed face down in a pile. Here, the shuffling of the deck is the first move. It is a chance move with 52! possible branches. Since the deck is face down, the players have only imperfect information about that first move. Imperfect information thus corresponds to partial state observation in control problems. *Incomplete* information, on the other hand, refers to a lack of information about the *rules* of the game. The rules include the complete extensive or strategic form of the game, including all strategies available to all players, all the payoff functions, the probability distributions for all chance moves, what information is available to each player, and so on. Games with incomplete information thus correspond to adaptive control problems. The incomplete information problem is much more difficult than the imperfect information problem, and unfortunately, it has received much less attention.

2.1.3 Pre-Game Cooperation

If preplay communication and binding agreements are allowed, we say that a game is *cooperative*. Traditional control theory approaches decentralized control from a cooperative point of view. As Ho puts it [47], "we permit any kind of communication and agreement among the decision makers *beforehand*." This dissertation, however, focuses on the non-cooperative case, where cooperation must be learned.

In strictly competitive (zero-sum) games, it is impossible for the players to receive mutual benefit from cooperation. Once again, the situation is more complex for general non-zero-sum games. It is possible for all players to benefit from cooperation. However, there are also cases where pre-game communication can involve threats. Luce & Raiffa [67] give an interesting example of this:

Table 2.2 *An example involving threats.*

		Player 2	
		Strategy 1	Strategy 2
Player 1	Strategy 1	$(1, 2)$	$(3, 1)$
	Strategy 2	$(0, -200)$	$(2, -300)$

In the non-cooperative game, the first strategy is dominant for both players, and so the payoff pair (1,2) is the unique equilibrium point. However, in the cooperative version, the first player can demand that the outcome be (3,1) by threatening to use his second strategy. Even if this explicit cooperation is not allowed, an implicit form of cooperation may develop through repetition of a non-cooperative game. In the example above, the first player would use his second strategy until the second player "learns the score". In this context, however, the idea of using threats assumes that the players have high-level cognitive skills and complete information about the game.

2.1.4 Repeated Games

Repeated games (RGs) come from a perspective more akin to learning. The repeated play of non-cooperative games allows the implicit development of cooperation, and the repeated play of games of incomplete information allows the gathering of information about some of the rules of the game (such as one's own payoff structure). The literature on RGs has been surveyed by Sabourian [93] and Aumann [3]. Much of the work on RGs addresses competitive games, and is thus not directly applicable to team problems.

An RG can be defined as a one-shot game repeated many times. If a one-shot game is repeated an infinite number of times it is called a *supergame*. RGs can possess a large number of equilibria. The Folk Theorem of RGs [93] states that any norm of behavior that guarantees payoffs to players more than their security levels (the minimum payoffs within their own control) can be supported as an equilibrium of a supergame. This can be proved by constructing strategies that punish any player that deviates from the norm.

Axelrod [4] uses the repeated Prisoner's Dilemma to study how cooperation can develop implicitly. He shows that in that game, there is no best strategy independent of the strategy used by the other player. He ran several round robin tournaments to determine what types of strategies would do well. The winner was a program called "Tit for Tat", which cooperates on the first move, and after that does whatever the other player did on the previous move. It is quite good at encouraging cooperation, for several reasons: it is a *nice* rule that is never the first to defect; it will retaliate if provoked; and it is forgiving. The most successful strategies had these properties. Axelrod then conducted an *evolutionary* tournament, where the number of copies or offspring of each strategy was proportional to its last tournament score. He found that strategies that were successful with a variety of others would proliferate at first, but

later as the unsuccessful strategies disappeared, success required good performance with other successful ones.

Although game theory provides a useful framework for dealing with multi-agent systems, it generally assumes that the players are already intelligent. It says little about how that intelligence might be created.

2.2 Artificial Intelligence

One thing AI researchers have learned over the past 40 years is that creating truly intelligent agents is extremely difficult. AI systems are often quite brittle, in the sense that they break down badly when faced with problems outside of their often narrow areas of expertise. The traditional answer in AI has been to add more knowledge into the system, including commonsense knowledge [44], but there is increasing skepticism about the feasibility of this approach. DAI researchers have proposed another answer: build a system consisting of smaller, more manageable components that can communicate and cooperate [55], possibly representing a diverse collection of capabilities and expertise [41]. However, one may ask whether even these components are truly manageable, since they are regarded in most DAI research as sophisticated, intelligent agents, themselves. And they must be intelligent if they are to coordinate with each other by reasoning about how their own plans and goals relate to the plans and goals of other agents.

In some sense, this is the old debate about strong versus weak methods. Weak methods are general-purpose problem-solving strategies, while strong methods utilize domain knowledge. Strong methods are indeed more powerful than weak methods, but they require the proper knowledge to be supplied to the system. It is this requirement that has been so hard to achieve. It is extremely difficult and time-consuming for humans to try to build knowledgeable, intelligent systems.

2.2.1 Distributed Artificial Intelligence

DAI research can be divided into two general categories: distributed problem solving, and multi-agent systems. Distributed problem solving considers how to design groups of agents that can cooperate to solve problems. The agents often differ in their areas of expertise, or in their access to sensory information. Research in multi-agent systems deals with interactions between self-motivated autonomous agents with conflicting objectives.

One of the key issues in DAI research has been finding efficient mechanisms for coordination among multiple intelligent agents. For example, in partial global planning (PGP) [40], after determining their own goals and plans, agents exchange information about how their plans interact and modify their plans in order to better coordinate their activities. They iteratively exchange tentative partial solutions in order to construct global solutions.

Coordination may also be achieved by means of negotiation. For example, the contract net protocol [37] assigns tasks to agents on the basis of a bidding mechanism. Agents respond to task announcements with bids indicating how well they believe they can perform the task. The Clarke tax [42] is a technique for ensuring that agents reveal their true preferences. Each agent pays a tax corresponding to the portion of its bid that makes a difference to the outcome.

Recently, there has been a great deal of interest in incorporating learning into DAI systems [122]. Weiss has provided a useful bibliography [121].

Sugawara & Lesser [103] have developed a system of knowledge-based local area network diagnosis agents that learn situation-specific coordination strategies. The agents start by acting based only on their local views. Whenever performance suffers, they replay traces of their inferences and make changes. Learning is done using heuristic rules for recognizing costly incoherent behavior, identifying the decisions that led to the behavior, and modifying those decisions in the future.

Nagendra Prasad & Lesser [78] describe a system that learns situation-specific coordination strategies. During each problem solving instance, the agents communicate their local views to form a common global view of the situation, agree upon a coordination strategy, and measure its performance. Each of five possible coordination strategies are run on each problem instance during training, and the performance of each strategy is noted. After the learning phase, the agents again communicate to form a global picture of the situation, and use a nearest neighbor technique to select the most appropriate coordination strategy.

Nagendra Prasad et al [79] present a multi-agent parametric design system called L-TEAM where a set of heterogeneous agents learn appropriate organizational roles. The system is applied to steam condenser design. Agents can choose to initiate a new design, or extend or critique an existing partial design. During learning, roles are chosen probabilistically. At the end of each problem solving episode, a credit assignment process determines the rating for each role selection. After learning, each agent chooses its highest rated role for each situation.

In an effort to combine distributed learning and organizational design, Weiss [120] considers a blocks world with heterogeneous agents, where each agent is able to perform only a limited number of operations. For example, one agent may only be able to put block F on block A or block G on block B. The agents must transform a starting configuration into a desired configuration. He presents an algorithm called DFG (Dissolution and Formation of Groups), that combines the agents into groups that compete to execute their actions. A credit-assignment process evaluates the relevance of the groups, dissolving some and allowing others to form.

2.2.2 Artificial Life and Behavior-Based Approaches

Some work in the DAI, artificial life, and behavior-based robotics communities is now focusing on a more bottom-up design methodology. The approach is to build less sophisticated agents, and see what emerges when they are put together into a group.

Shoham & Tennenholtz [99, 100] investigate the social behavior that can emerge from agents with simple learning rules. They focus on two simple n-k-g iterative games, where n agents meet k at a time (randomly) to play game g. The games are the convention game where the social goal is for the agents to agree on a single bit, and the Prisoner's Dilemma, where the social goal is cooperation. The "Cumulative Best Response" algorithm they present works fairly well in the convention game, but rather poorly in the Prisoner's Dilemma. It is a relatively greedy algorithm that could not be expected to perform well in more general situations such as bandit problems [17], and they do not compare it with existing RL algorithms.

In behavior-based robotics, the conditions under which behaviors should be activated are generally fixed at design-time by hand. However, when the behaviors and the environment are complex, it becomes difficult for the designer to specify a strategy for all contingencies. Maes & Brooks [68] present a distributed learning algorithm that learns to coordinate the behaviors necessary for six-legged walking in an insect-like robot. Each of the robot's six built-in swing-leg-forward behaviors can be considered as an agent. Each agent observes the six perceptual bits that specify which legs are touching the ground. The action set for agent i consists of turning the behavior on or off for leg i. Each behavior must learn the conditions under which it should become active. They all learn in parallel, receiving immediate global reinforcement signals. Positive feedback comes from forward motion, and negative feedback comes when sensors on the belly of the insect detect that it has fallen. Each behavior's learning algorithm tests the condition bits one at a time for correlation with positive

or negative reinforcement. The insect eventually learns a tripod gait, where it lifts three legs at a time.

Humphrys [50] presents a behavior-based algorithm where the behaviors of a robot are considered to be agents competing for control of the robot. The agents each perform Q-learning [119] with their own separate reward functions. Each agent suggests an action with some weight based on its Q-values and the robot executes the action with the highest weight. Rather than specify reward structures based on external goals for the system, reward structures are chosen randomly to see what kind of system emerges. In other experiments, reward structures are evolved using genetic algorithms with some fitness measure for the system. Unfortunately, Humphrys does not compare the fitness of his evolved agents with the fitness that could be achieved by using this fitness measure as the direct basis for a reward structure for a Q-learning robot.

Mataric [72] studies the collective behavior of groups of behavior-based robots. For example, *flocking* behavior can be obtained by having each robot sum the outputs of its individual *avoidance*, *aggregation*, and *wandering* behaviors. *Foraging* behavior can be obtained from a more complex combination of behaviors that are sensitive to sensory conditions. Mataric has also adapted some reinforcement learning techniques to enable learning in these systems [73]. Behaviors take the place of actions, states are clustered into *conditions*, and shaping speeds the learning process. These changes amount mostly to incorporating some domain knowledge in order to speed up learning.

Much of the work in artificial life, behavior-based AI, and other bottom-up approaches has grown out of a frustration with the difficulty of designing complex systems from the top down. However, bottom-up techniques also seem to require a very high level of human intervention and experimentation, unless there is no goal except to "see what happens." For a more principled, but computationally intensive viewpoint, we next consider the control theory perspective.

2.3 Control Theory

Control theorists take a rigorous, mathematical approach to the design of multi-agent systems. In this section, we introduce the control-theoretic framework, and discuss relevant solution techniques based on dynamic programming.

2.3.1 Stochastic Optimal Control

Most controllers periodically observe a system that is to be controlled (called a *plant*), and take actions to influence the state of that system. In a *regulation* problem, the objective is to keep the output of the plant close to some constant target. In a *tracking* problem, the objective is for the output of the plant to follow some specified trajectory. In the most general case, called an *optimal control* problem, the objective is to minimize an arbitrary cost function, which may depend on both the state trajectory of the plant and on the actions generated by the controller. More formally, one can consider the plant to be a discrete-time stochastic dynamic system, observed from M observation posts, and influenced by actions from K control stations, as follows:

$$\begin{aligned} x_{t+1} &= f_t(x_t, u_t, v_t) \\ y_t &= g_t(x_t, w_t) \\ cost &= \sum_t h_t(x_t, u_t) \end{aligned}$$

where

- x_t is the state vector at time t,
- u_t is the K-dimensional vector of actions $[u_t^1, \ldots, u_t^K]$,
- y_t is the M-dimensional vector of observations $[y_t^1, \ldots, y_t^M]$, and
- v_t and w_t are random disturbances.

2.3.2 Information Patterns and Decentralized Control

One must also specify the possible ways in which the actions u_t^i can be generated. The *information pattern* I defines what data is available to the *control law* (or *policy*) π:

$$u_t^i = \pi^i(I_t^i),\ i = 1, \ldots, K.$$

Here, the policy π of the controller is a K-dimensional vector whose components are the policies π^i of the control stations. It is thus reasonable to view the control stations as being controllers in their own right. Each component I_t^i of the information pattern defines what data is available to control station i at time t. Each control station will have access to some (possibly different) subset of the current and past observations from the M observation posts and the past actions from the K control stations. The maximum possible amount of data available to each control station consists of all previous observations and actions, i.e., for each i, $1 \leq i \leq K$:

$$I_t^i = \{y_0, \ldots, y_t, u_0, \ldots, u_{t-1}\}$$

An example of a more restrictive information pattern is one where the control stations do not have access to each other's actions, i.e., for each i, $1 \leq i \leq K$:

$$I_t^i = \{y_0, \ldots, y_t, u_0^i, \ldots, u_{t-1}^i\}$$

A control station has *perfect recall* [126] if it never forgets any information. In a *classical* information pattern, all the control stations receive the same data and have perfect recall. In a *strictly classical* pattern, there is a single control station ($K = 1$) with perfect recall. The classical and strictly classical patterns are equivalent if the K stations are allowed to cooperate ahead of time in determining their policies. When the information pattern is not classical, one is faced with a *decentralized* control problem.

2.3.3 Markov Decision Problems

We first consider the simplest case, the single-agent case, with a strictly classical information pattern and fully observable state. Let us assume that the agent is interacting with a discrete-time, finite state environment. One can formalize this interaction as a Markov decision problem (MDP). At each time t, the agent observes the state $x_t \in X$ of the environment, selects an action $u_t \in U$, and receives a real-valued reward $r_t = h(x_t, u_t)$. The environment then enters a new state x_{t+1} according to some probability distribution based only on x_t and u_t. The agent's objective is to find a policy $\pi : X \rightarrow U$ that will select actions that maximize the return from any starting state, where the return is defined as the sum of the expected rewards over some finite or infinite time horizon. In infinite horizon problems, it is common to multiply the future rewards by powers of a discount factor $0 < \gamma < 1$ to keep the sum finite. In that case, the agent attempts to maximize $\sum_{t=0}^{\infty} \gamma^t r_t$. The use of a discount factor also puts a greater value on immediate rewards than on those in the more distant future. In cases where the objective is to minimize costs rather than maximize rewards, one usually refers to the *cost–to–go* rather than the return.

The expected return from any state is also called the *value* of that state. Of course, the value of a state depends on what policy the agent is following. Given any policy π, there is a corresponding value function V_π that maps states to values. An optimal policy is one that maximizes the expected return from any starting state. There may be more than one optimal policy, but all optimal policies will share the same (optimal) value function.

Dynamic programming (DP) can be used to find an optimal policy if the reward function and state transition probabilities are known *a priori*. In this case, we say that the agent has a *probability model* of its environment [13]. Although uncertainties exist, they are already well understood.

2.3.4 Dynamic Programming

A DP technique called *value iteration* can be used to calculate the optimal value function. Starting with an initial estimate $\hat{V}^*$, repeatedly update this estimate by performing "backup" operations as follows:

$$\hat{V}^*(x) = \max_u [h(x,u) + \gamma \sum_{z \in X} \hat{V}^*(z) Prob(x_{t+1} = z | x_t = x, u_t = u)].$$

Traditionally, these backup operations are ordered to form repeated sweeps through the state set. (For alternatives to this, see [11, 20]). Regardless of the initial estimate $\hat{V}^*$, this process converges to the optimal value function V^*. Once the optimal value function V^* has been approximated closely enough, it can be used to easily find an optimal policy. A *greedy* policy with respect to a value function V selects actions that maximize $r_t + \gamma \sum_{z \in X} V(z) Prob(x_{t+1} = z | x_t, u_t)$. If the optimal value function is known, then a greedy policy with respect to that value function will be optimal.

A similar approach may be used to find the value function for any policy. Starting again with an arbitrary estimate of the value function, repeatedly perform the following backup operations:

$$\hat{V}_\pi(x) = h(x, \pi(x)) + \gamma \sum_{y \in X} \hat{V}_\pi(z) Prob(x_{t+1} = z | x_t = x, u_t = \pi(x_t)).$$

As was the case above, $\hat{V}_\pi$ converges to V_π.

Another DP technique called *policy iteration* can be used to find an optimal policy as follows: Starting from an arbitrary initial policy π_0, compute its value function V_{π_0} according to the method described above. Then create a new policy π_1 that is greedy with respect to V_{π_0}. Then repeat the process to find better and better policies until π_{i+1} is the same as π_i, in which case they are both optimal. This is guaranteed to occur within a finite number of iterations.

MDPs can be solved efficiently by value iteration and policy iteration in time that is polynomial in the size of the state space. However, in many problems of interest, the

state space is so large that even polynomial time is completely infeasible. Although deterministic versions of an MDP can be solved very fast in parallel, Papadimitriou & Tsitsiklis [83] show that in general MDPs are P-complete, and thus most likely cannot be solved significantly faster with highly parallel algorithms.

2.3.5 Non-Classical Information Patterns

Most of control theory has dealt with the classical information pattern. Under the classical pattern, a wide variety of powerful results have been found for controlling linear systems with quadratic cost functions (called LQ problems). Unfortunately, decentralized control is much more difficult than centralized control. For example, in 1968, Witsenhausen [125] showed that under a non-classical information pattern, an optimal controller for an LQ problem is no longer necessarily linear in the state of the system.

Some work has been done on the decentralized control problem, but policies for the control stations are developed from a global vantage point with complete knowledge of the problem, and learning does not play a role. Even though *executing* the policies depends only on the information available at each control station, the policies are *designed* in a centralized way, with access to complete knowledge about the problem. Research has focused on what constitutes an optimal policy under a given information pattern, but not on how such policies might be learned within the constraints of that same information pattern.

Decentralized control problems are thus treated as centralized problems with incomplete state information. The standard approach in problems with incomplete state information is to reduce them to problems with perfect state information, where the new state consists of all the information that *is* available [2]. The problem of delayed sharing patterns, where the controllers share their own private information after k steps of delay, is one example in decentralized control where this technique has been

used. Witsenhausen [126] conjectured in 1971 that the optimal control strategy in such cases could be made to depend on the common information only through the conditional distribution of the state given the common information. Varaiya & Walrand [118] gave a counterexample to this in 1978 for delays of more than one step. The reason is that the common information might contain not only information about the state, but also about the actions selected between time t-k and time t. However, they proved the conjecture to be true for one step delays. Hsu & Marcus [49] used this result in 1982 to reduce the one step delay problem to a centralized problem. The common information becomes the more complex state in the equivalent centralized problem. Classical methods could then be used to solve for the optimal policy. Aicardi et al [1] extended this result in 1987 to k step delay problems by finding a sufficient statistic that captured all the relevant information from the common information. Sufficient statistics are used to respond to the problem that the dimension of the state (the information vector) grows as new measurements come in. One wants to reduce this information to the data actually needed for control purposes. Sufficient statistics are ideally of a smaller dimension, and yet summarize all the information that is relevant to control [18, 19]. In the case of Aicardi et al, however, even though the quantity now serving as the state may be finite and constant dimensional, it will generally be far too massive to be of any practical value, since it will include not only variables, but also information on entire mappings.

It is also important to mention that methods such as sufficient statistics that are used in dealing with incomplete state information tend to assume that though there are uncertainties, they are already well understood (e.g. all the probabilities are known ahead of time). In *adaptive control*, accurate knowledge about the uncertainties is not available ahead of time, but must be learned on-line during control itself.

Even though one calls solutions to the equivalent perfect state information problems *optimal*, they are only optimal *given the restricted information*. A better policy might be available if the original problem had perfect information. So the motivation for decentralized control is not that it is more powerful than centralized control. The motivation is the *necessity* of decentralized control in large scale systems where centralized information is not available or is not practical.

In operations research, control problems with incomplete state information are refered to as partially observable Markovian decision problems (POMDP's). For an overview, see [65, 66, 76]. As is the case in the control literature, a non-adaptive viewpoint is usually taken, where the set of underlying states, transition probabilities, and observation probabilities are assumed to be known *a priori*. Given this knowledge, the useful information in the history of observations and actions can be summarized in a vector containing the current probability distribution over the state set. Then the POMDP can be reformulated as an equivalent MDP whose state space is the set of all probability distributions over the original state space. The trouble is that this new state space is uncountably infinite. All feasible numerical methods thus attempt to restrict the number of distributions actually considered. Still, the number of points in the new state space that must be considered generally grows exponentially in the length of the observation history, and so all but the smallest of these problems are intractable.

Papadimitriou & Tsitsiklis [83] show that finite-horizon POMDPs are PSPACE-complete, and that "most likely, it is not possible to have an efficient on-line implementation (involving polynomial time on-line computations and memory) of an optimal policy, even if an arbitrary amount of precomputation is allowed." Finally, they do not even consider infinite horizon POMDPs because they "do not seem to be exactly solvable by finite algorithms." The *adaptive* version of a POMDP where the

state set, transition probabilities, and observation probabilities are not known, can only be more difficult.

2.4 Reinforcement Learning

Researchers in distributed artificial intelligence and control theory have generally focused on top-down approaches to building distributed systems, creating them from a global vantage point. Even though the agents in such systems are restricted to their own local point of view, they are designed in a centralized way, with access to complete knowledge about the problem. One drawback to this top-down approach is the extraordinary complexity of designing such agents, since it is extremely difficult to anticipate all possible interactions and contingencies ahead of time in complex systems.

Artificial life and behavior-based AI researchers have generally taken the opposite approach, combining large numbers of relatively unsophisticated agents in a bottom-up manner and seeing what emerges when they are put together into a group. However, there is little evidence as yet that such bottom-up designs really provide a simple way of achieving complex specific goals.

Multi-agent RL combines the advantages of both approaches. It achieves the simplicity of a bottom-up approach by allowing the use of relatively unsophisticated agents that learn on the basis of their own experiences. At the same time, RL agents adapt to a top-down global reinforcement signal, which guides their behavior toward the achievement of complex specific goals. As a result, very robust systems for complex problems can be created with a minimum of human effort [35].

RL also combines the heuristic, satisficing nature of AI with the principled nature of control theory, by approximating DP in an incremental manner. RL algorithms can be trained using actual or simulated experiences, allowing them to focus computation on the areas of state space that are actually visited during control, making them

computationally tractable on very large problems. If each of the members of a team of agents employs an RL algorithm, a new *collective* algorithm emerges for the group as a whole. This type of collective algorithm allows control policies to be learned in a decentralized way. This dissertation shows that even though RL agents in a team face added stochasticity and non-stationarity due to the changing stochastic policies of the other agents on the team, they display an exceptional ability to cooperate with one another in maximizing their rewards. The next chapter reviews RL and its application to multi-agent systems in more detail.

CHAPTER 3

REINFORCEMENT LEARNING

This chapter provides an overview of several reinforcement learning (RL) algorithms and reviews how they have been applied to learning in multi-agent systems. RL tasks can be divided naturally into two types. In non-sequential tasks, the objective is to learn a mapping from situations to actions that maximizes the expected immediate payoff. In sequential tasks, the objective is to maximize the expected long-term payoffs. Sequential tasks are more difficult because the actions selected may influence the future situations and thus also the future payoffs.

3.1 Non-Sequential Reinforcement Learning

The problem an agent faces in learning to select the best action for a given situation has been studied extensively in the field of Learning Automata. Narendra & Thathachar [80] give an excellent introduction to this field.

3.1.1 Learning Automata

A *variable-structure* automaton is a quadruple (α, β, p, T) where

- α is the output or action set of the automaton,
- β is the set of possible reinforcement signals it can receive,
- p(t) is the vector of its action probabilities at step t, and
- T is its learning scheme such that $p(t+1) = T(p(t), \alpha(t), \beta(t))$.

An automaton is connected to its environment in a feedback loop. On each time step, the automaton selects an action and receives a reinforcement feedback from its environment according to some fixed probability distribution for each action. The simplest environments give binary reinforcement signals corresponding to rewards and penalties.

The behavior of an automaton in its environment can be analyzed as a continuous-state, discrete-time Markov process. The state space of the Markov process corresponds to the space of possible action probability vectors of the automaton.

The learning schemes used by automata work by increasing the probability of actions that receive rewards and decreasing the probability of actions that receive penalties. If action α_i is chosen at time t, then if a reward is received,

$$p_i(t+1) = p_i(t) + \sum_{j \neq i} g_j(p(t))$$

and

$$p_j(t+1) = p_j(t) - g_j(p(t)),$$

while if a penalty is received,

$$p_i(t+1) = p_i(t) - \sum_{j \neq i} h_j(p(t))$$

and

$$p_j(t+1) = p_j(t) + h_j(p(t)).$$

To describe a learning scheme in this framework, one only needs to specify the functions g_j and h_j.

L_{R-P} and L_{R-I} are two common learning schemes. The linear reward-penalty (L_{R-P}) scheme makes the same magnitude probability adjustments in response to both rewards and penalties. The linear reward-inaction (L_{R-I}) scheme only adjusts the probabilities after receiving rewards. The probabilities are not changed following penalties. L_{R-P} and L_{R-I} have very different convergence properties. L_{R-P} has

no absorbing states and its action probabilities converge in distribution to a random vector which does not depend on the initial state p(0). L_{R-I} converges to an absorbing state with probability one. The initial state p(0) and the penalty probabilities both affect the probabilities of converging to each of the absorbing states. By choosing a g that makes slow enough changes, the probability of converging to the optimal action can be made arbitrarily close to one.

The $L_{R-\epsilon P}$ algorithm makes much smaller changes for penalties than for rewards. The amount of asymmetry in the changes is specified by a parameter $\lambda \in (0, 1)$. (In the L_{R-P} scheme, $\lambda = 1.0$ since the magnitude of changes for penalties and rewards are the same. In the L_{R-I} scheme, $\lambda = 0.0$ since there are no changes for penalties.) $L_{R-\epsilon P}$ combines some of the best properties of L_{R-P} and L_{R-I}. Like L_{R-I}, it will tend toward a deterministic policy (taking the best action with almost probability one). However, like L_{R-P}, it is ergodic, so it is more robust than L_{R-I} at avoiding the wrong absorbing states. It might be particularly useful in nonstationary environments, such as those containing other adaptive agents. The performance of $L_{R-\epsilon P}$ may also benefit by annealing λ over the course of learning.

3.1.2 Games of Automata

Chapter 8 of [80] summarizes much of the work on games of automata. Let G be an N-person game in strategic form. The automata participate in the repeated game G^{∞}, where the stage-game G is repeated infinitely often. The vector of actions $\alpha(t) = [\alpha^1(t), \alpha^2(t), ..., \alpha^N(t)]$ chosen by the automata at stage t correspond to their strategy choices in that stage-game. The outcome of stage t is $\beta(t) = [\beta^1(t), \beta^2(t), ..., \beta^N(t)]$. Each automaton receives feedback $\beta^j(t)$ about the outcome which is used to update its action probabilities for the next stage. The automata are faced with an extreme case of incomplete information. They are not even aware that they are participating

in a game. In spite of this handicap, their asymptotic behavior is quite impressive, as shown by a number of analyses.

- Two-Person Zero-Sum Games: Lakshmivarahan & Narendra [56, 57] show that if a two-person zero-sum game has a unique pure (i.e. deterministic) strategy equilibrium, two automata using the L_{R-I} scheme will converge to that solution. Further, two automata using the $L_{R-\epsilon P}$ scheme will converge to an equilibrium point of the game whether or not it has a pure strategy equilibrium.

- Team Problems: The results of Narendra & Wheeler and Thathachar & Ramakrishnan are also summarized in [80]. In team problems, an arbitrary number of automata using the L_{R-I} scheme, each of which has an arbitrary number of actions, will converge to an equilibrium point which is a *local* maximum (an element of the game matrix that is the maximum of both its row and its column).

- Non-Zero-Sum Games: As was the case with team problems, in non-zero-sum games, automata using the L_{R-I} scheme will converge to an equilibrium point. The problem is that equilibria in non-zero-sum games often provide poor payoffs for all players. A good example of this is the Prisoner's Dilemma, where the only equilibrium point produces the lowest total payoff.

3.1.3 Associative Learning Automata

It is useful to generalize learning automata to the case of multiple contexts. *Associative* learning automata receive the context as an additional input, and must learn a mapping from contexts to actions that maximizes the expected reinforcement. Such a task is known as an associative reinforcement learning task [9] because an association must be made between contexts and actions. One way to address this would be to have a group of learning automata, and assign one to each possible context. Such a

lookup-table approach is flexible in terms of the mappings it can do, but it would not allow any generalization across different contexts.

Barto & Anandan [10] introduced a learning algorithm for associative learning automata that does allow generalization. They called it the *associative reward-penalty*, or A_{R-P} algorithm. An A_{R-P} unit operates in discrete time. At step t, it receives the context through inputs $y_1(t)$ to $y_n(t)$. The unit calculates its output $u(t)$ as follows:

$$u(t) = \begin{cases} +1 & if \sum_{i=1}^{n} w_i(t) y_i(t) > \eta(t) \\ -1 & otherwise \end{cases}$$

where the w_i's are parameters (or weights) adjusted by the learning rule, and $\{\eta(t), t \geq 1\}$ is a sequence of independent, identically distributed random variables. (The distribution is usually assumed to be normal with zero mean.) A_{R-P} units are like linear threshold units but with a random threshold. Here, the weighted sum determines the output *probabilities*. If the sum is zero, then both output values are equally likely, but as the sum increases, the likelihood of output $+1$ also increases.

Based on the output $u(t)$, the unit receives a reinforcement $r(t)$ of either $+1$ (indicating a "reward") or -1 (indicating a "penalty"). We assume that the reinforcements are probabilistic, just as they were for non-associative learning automata. The A_{R-P} unit then updates its weights as follows:

$$\Delta w_i = \begin{cases} \rho[r(t)u(t) - E\{u(t)\}]y_i(t) & \text{if } r(t) = +1 \\ \lambda\rho[r(t)u(t) - E\{u(t)\}]y_i(t) & \text{if } r(t) = -1 \end{cases}$$

where $\rho > 0$, $0 \leq \lambda \leq 1$, and $E\{u(t)\}$ is the expected value of the output given the inputs and the weights. The noisy threshold enables the unit to explore. If the results of the exploration produce a reward, the weights are changed to make that output more likely when faced with the same context in the future. Likewise, if a penalty is received, that output is made less likely. The value of λ determines the relative magnitude of the changes made for rewards and penalties. If $\lambda = 0$, we have the A_{R-I} algorithm, where the weights are only updated after receiving rewards. If λ is

a small positive number, we have the $A_{R-\epsilon P}$ algorithm, where the changes are much larger for rewards than for penalties.

Associative learning automata have several applications from a multi-agent point of view. They can be applied to team decision problems, where each automaton has incomplete information about some underlying uncertainty and their actions must be coordinated in order to maximize their payoffs. They can also be connected into networks allowing the actions of one automaton to form part of the context input for other automata, thus allowing a form of explicit communication. The next two sections discuss these configurations.

3.1.4 Teams of Associative Learning Automata

In static team decision problems [114, 47, 71], each agent i has a finite set of possible observations Y_i and possible actions U_i. Given a joint probability distribution on $Y_1 \times \ldots \times Y_N$ and a payoff function $r : Y_1 \times \ldots \times Y_N \times U_1 \times \ldots \times U_N \rightarrow R$, the problem is to find policies $\pi_i : Y_i \rightarrow U_i$, i=1,...,N, that maximize expected payoff

$$J(\pi_1, \ldots, \pi_N) = \sum_{y_1} \ldots \sum_{y_N} r(y_1, \ldots, y_N, \pi_1(y_1), \ldots, \pi_N(y_N)) pr(y_1, \ldots, y_N).$$

Tsitsiklis & Athans [114] have shown that this problem is NP hard, even in the case where there are only two agents, each with only two possible actions, and where the payoff function r only takes on the values 0 and 1. This analysis assumes that the agents are given the complete problem description and are allowed to determine their joint strategies before they must act, as would be the case in a cooperative game with complete information. Team decision problems can only be *more* difficult when no member of the team receives the complete problem description, and cooperation must be learned implicitly through repeated trials. Associative learning automata are not even aware that there *are* other agents.

Though it is clear that these problems are very difficult, this is a worst-case result. Approximate methods and decentralized learning methods may work well on

many problem instances encountered in practice, and more restricted classes of team problems with additional structure may be easier to solve.

3.1.5 Networks of Associative Learning Automata

Associative learning automata can also be connected into networks allowing the actions of one automaton to form part of the context input for other automata, thus allowing a form of explicit communication. Barto [7] studied the learning behavior of networks of A_{R-P} units and showed empirical results on their ability to learn non-linear tasks, including an exclusive-or problem and a multiplexor problem. Williams [124] proved that for a class of learning algorithms that includes networks of A_{R-I} units, their expected update direction in weight space follows the gradient of the expected reinforcement as a function of the weights. The long-term behavior of these networks was analyzed by Phansalkar & Thathatchar [88] using weak convergence techniques. Networks of associative learning automata perform a statistical analogue to the type of gradient descent carried out by error backpropagation [91]. Barto & Jordan [12] compared the backpropagation algorithm with a network of modified A_{R-P} units that handle real-valued reinforcement and use a *batching* method to increase learning efficiency. Backpropagation is faster since it can compute the gradient directly, while the A_{R-P} units must estimate it in the presence of noise. However, networks of learning automata have a biological plausibility that backpropagation seems to lack [32], and they are applicable to problems more difficult than supervised learning. The two approaches can be effectively combined by using a backpropagation network with associative learning automata as *output* units. Gullapalli [45] applied this architecture successfully to a difficult robotics problem. He used SRV (Stochastic Real-Valued) units, a generalization of the A_{R-P} unit that produces real-valued outputs.

3.2 Sequential Reinforcement Learning

Sequential decision problems differ from single-stage problems in that the agents seek to maximize the rewards received over some period of time. This generally requires foresight and planning on the part of the agents, since it may sometimes be necessary to take smaller immediate rewards in order to receive better rewards in the long run. Sequential problems benefit from the notion of *state*, that is, a summary of all past and present conditions that have a bearing on predicting the future. If a long-term value can be associated with each state, then a greedy policy with respect to these values will be optimal. An even more basic problem exists when the state is not fully observable. This is a particularly difficult stumbling block because, in general, the probability distribution of the underlying state depends on the *entire past history* of observations and actions. This section discusses sequential reinforcement learning algorithms and their application to multi-agent systems.

3.2.1 RL Techniques Based on Dynamic Programming

The dynamic programming (DP) algorithms discussed in section 2.3.4 are very efficient methods of searching for optimal policies, requiring only time polynomial in the number of states. However, many important problems contain a very large number of states. DP is computationally infeasible for such problems, since it requires multiple complete sweeps of the entire state set. In addition, it requires a probability model representing the state transition probabilities and reward structure of the environment, which may be difficult to obtain.

A number of sequential RL algorithms have been developed that approximate DP on an incremental basis [11]. Unlike traditional DP algorithms, these algorithms can perform with or without a model of the environment, and they can be used online as well as offline. By using actual or simulated control experiences to determine which states should be updated, they seek to focus computational resources on the areas

of state space that are most relevant to optimal control. By using compact function approximators such as artificial neural networks, they can represent value functions that are too massive to store as lookup tables. This allows generalization that can speed up learning, but that also removes most theoretical guarantees of convergence [21, 22, 115].

When a probability model of the environment is not available, two basic approaches are generally used. *Indirect* approaches first estimate an explicit probability model of the environment, and then use model-based techniques. *Direct* approaches do not learn an explicit probability model of the environment. Environments in multi-agent RL are generally nonstationary, since they contain other learning agents. Although this poses a problem for both approaches, indirect methods may suffer the most from nonstationarity since any adjustments must filter through both the model and the computations based on the model. Therefore, one might expect to achieve better results in multi-agent RL using direct approaches.

Backup operations (section 2.3.4) can be based on *sampling* the environment when a probability model is not available. Instead of explicitly considering the probabilities of all possible next states, backups can be based only on the next state that actually occurred. For good results, enough samples must be taken to accurately reflect the underlying probabilities in the environment and the updates should be made gradually. In stationary environments, the magnitudes of the updates should be reduced over time. This can be achieved by making the size of the updates proportional to a learning rate parameter, which can be decreased according to some schedule.

In many cases it is possible to generate sample realizations of a process without an explicit probability model. One can often construct a simulation model of an environment using a high-level structural view of the environment without ever explicitly determining the state transition probabilities [13, 34]. (For an example of

such a simulation model, see section 4.3.) Sample backups can then be performed along trajectories through state space generated by the simulation model.

If an indirect approach is used, some backups may be performed using the environment, and some using the partially completed probability model. This interleaves the process of building and using the model. As the model improves, it naturally becomes more useful in performing backups. Sutton has discussed this approach and calls it the "Dyna" architecture [105, 84]. Moore & Atkeson's "Prioritized Sweeping" algorithm [77] builds a forward and backward model incrementally based on observations of the environment. After each real-world transition, the model is updated and some number of backups are performed. A priority queue is used to keep track of the areas of state space where backups would be most useful. If an observation is surprising, then that state's predecessors are promoted to the top of the queue.

Watkins developed a direct approach called "Q-learning" [119]. Instead of learning the value of states, this algorithm learns the value of state-action pairs. $Q(x, a)$ is defined to be the expected discounted return for performing action a in state x, and performing optimally thereafter. Once the Q function has been learned, the optimal action in any state x is the one with the highest value $Q(x, a)$. Learning is achieved by updating an estimate of the Q function using repeated backups of the form:

$$\Delta\hat{Q}(x_t, a_t) = \alpha[r_t + \gamma \max_b \hat{Q}(x_{t+1}, b) - \hat{Q}(x_t, a_t)],$$

where α is a learning rate parameter. Watkins has shown that given a lookup table representation, $\hat{Q}$ converges to Q if all actions continue to be tried from all states and the learning rate is decreased appropriately over time. In practice, it is useful to balance the need for exploration with the need to select the best actions. There are several methods for doing this, including the selection of actions according to a Boltzmann distribution or the methods discussed by Sato et al [97].

Sarsa [92] is a modification of the Q-learning rule based more closely on temporal difference learning [104]. It uses the Q-value for the action actually selected at time $t+1$ rather than the greedy max action in its update rule:

$$\Delta\hat{Q}(x_t, a_t) = \alpha[r_t + \gamma\ \hat{Q}(x_{t+1}, a_{t+1}) - \hat{Q}(x_t, a_t)],$$

Sarsa attempts to estimate the state-action values for the current policy rather than directly estimating the state-action values for an optimal policy. Rummery & Niranjan [92] argue that if the amount of exploration is reduced as training proceeds, the greedy action will eventually be chosen at each step, and the Q-function will converge to the optimal values.

3.2.2 Policy-Based RL Techniques

Most of the sequential RL algorithms discussed above attempt to learn the values of states or actions explicitly, and then derive a policy based on those values. This stands in contrast to the learning automata discussed in section 3.1.1, which are *policy-based* algorithms that do not attempt to learn action values, but work directly with action probabilities. Of course, it is possible to use value-based RL algorithms in non-sequential problems, for example, Q-learning with $\gamma = 0$. Likewise, there are policy-based sequential RL algorithms, such as actor/critic algorithms [14, 15, 105], where the actor implements a stochastic policy that maps states to action probability vectors, and the critic attempts to estimate the value of each state under the current policy in order to provide more useful feedback to the actor. For further insight into the relationship between value-based and policy-based RL algorithms, see Crites & Barto [33].

3.2.3 Sequential Multi-Agent Reinforcement Learning

Starting in approximately 1993, a number of researchers began to investigate applying sequential RL algorithms in multi-agent contexts. Although much of the work

has been in simplistic domains such as grid worlds, several interesting applications have appeared that have pointed to the promise of sequential multi-agent RL.

Markey [69] applies parallel Q-learning to the problem of controlling a vocal tract model with 10 degrees of freedom. He discusses two architectures equivalent to the distributed and parallel architectures described in section 5.4. Each agent controls one degree of freedom in the action space, and distinguishes Q-values based only on its own action selections. The agents select their actions synchronously in discrete time based on the state of the vocal tract. Their goal is to learn a sequence of actions corresponding to a particular articulatory trajectory. The parallel Q-learning architecture is able to handle a problem with 55 million possible action combinations that would be unthinkable using a centralized architecture. Tham & Prager [111] briefly mention the same idea of having a separate Q-network independently control each actuator.

Bradtke [23] describes some initial experiments using RL for the decentralized control of a flexible beam. The task is to efficiently damp out disturbances of a beam by applying forces at discrete locations and times. He uses 10 independent adaptive controllers distributed along the beam. Each controller senses the position and velocity at its location on the beam, and each applies a force to the beam. There is no communication between the controllers. He starts with a group of stabilizing controllers, each of which then employs a policy iteration algorithm based on Q-learning. Each controller attempts to minimize its own local costs and observes only its own local portion of the state information. By collapsing the states into a much smaller group of equivalence classes and considering the actions independently, the size of the problem is significantly reduced, resulting in a 99.83 percent decrease in computation, while the policy derived is still within 8 percent of optimal. As Bradtke notes, this result depends on the structure inherent in the problem. That is, changes

in a component of the state vector depend primarily on the influence of neighboring components.

Dayan & Hinton [39] propose a managerial hierarchy they call *Feudal RL*. In their scheme, higher-level managers set tasks for lower level managers, and reward them as they see fit. Since the rewards may be different at different levels of the hierarchy, this is not strictly a team. Furthermore, only a single action selected at the lowest level actually affects the environment, so in some sense, this is a hierarchical architecture for a single agent. The motivation for the architecture is to allow learning to occur at multiple levels of resolution. One key question is whether the organizational structure must be built-in ahead of time or whether it could be learned.

Tan [107] reports on some simple hunter-prey experiments with multi-agent RL. His focus is on the sharing of sensory information, policies, and experience among the agents. Some of the hunters receive sensory information from each other or from scouts. Others use and update a single shared policy or exchange their policies or experiences, speeding up the learning process.

Sen & Sekaran [98] apply distributed Q-learning to a simple block pushing problem. Each agent independently selects the direction and magnitude of the force it will exert on a block. The resultant force is determined by vector addition. The global reinforcement signal is determined by the deviation from the desired path of the block. It is not clear why Q-learning was used, since this is a non-sequential RL problem where the goal is to maximize the immediate reinforcement at each step.

Littman & Boyan [63] describe an unusual distributed reinforcement learning algorithm for packet routing based on the asynchronous Bellman-Ford algorithm. Their scheme uses a single Q-function, where each state entry in the Q-function is assigned to a node in the network which is responsible for storing and updating the value of that entry. This differs from most other work on distributed RL, where an entire Q-function, not just a single entry, must be stored at each node. From the point of

view of each node using their algorithm, the notion of "state" is degenerate since each node deals with only one state. Therefore, the usefulness of this idea seems limited to a small group of spatial applications.

In addition to the multi-agent RL research concerned with team problems, a significant amount of work has focused on zero-sum games, where a single agent learns to play against an opponent. One of the earliest examples of this is Samuel's checker-playing program [95]. A more recent example is Tesauro's TD-Gammon program [108, 109, 110], which has learned to play strong Master level backgammon. These types of programs are often trained using self-play, and they can generally be viewed as single agents. Littman [64, 65] provides a detailed discussion of RL applied to zero-sum games, both in the case where the agents alternate their actions and where they take them simultaneously.

Very little work has been done on multi-agent RL in more general non-zero-sum games. Sandholm & Crites [96] study the behavior of multi-agent RL in the context of the iterated prisoner's dilemma (see section 2.1). They show that Q-learning agents are able to learn the optimal strategy against the fixed opponent Tit-for-Tat. In addition, they investigate the behavior that results when two Q-learning agents face each other.

3.2.4 RL Approaches to Hidden State

We end this chapter with a review of RL approaches to the problem of incomplete state observation. Hidden state is common in multi-agent contexts, since each agent may only perceive its own local portion of the state. The hidden state problem has recently received a great deal of attention from AI researchers who are attempting to build *situated* or *embedded* agents, that is, agents that must perceive and act in the world. These agents are driven by their perceptions, and in most cases they are not able to perceive the complete state of their environments. *Perceptual aliasing*

[123] occurs when a perception does not uniquely identify a state of the world. As McCallum points out [74], perceptual aliasing can be a blessing or a curse. It is a blessing when it confounds states where the same action is best, since it reduces the size of the policy space to be searched. However, it is a curse when it confounds states where different actions are best.

Jaakkola et al [52] present an algorithm that does not attempt to estimate the underlying state, but rather seeks to find a stochastic policy that can work well in spite of perceptual aliasing. The algorithm iterates Monte Carlo policy evaluations with policy improvement steps. However, given many aliased states, the resulting stochastic policy may perform quite poorly.

There are two main approaches that attempt to identify hidden states. The first could be called the sensor-based approach. The idea is to use additional perceptual actions to disambiguate the state. Examples of this include work by Whitehead & Ballard [123] and Tan [106]. One disadvantage of the algorithms they present is the assumption of a deterministic environment. A limitation of sensor-based approaches in general is that they cannot handle what Chrisman calls *incomplete perception* [30]. This refers to situations where no perceptual actions can be taken to disambiguate world states. In these situations, the only chance at being able to distinguish certain states is by remembering past perceptions and actions. This is called the memory-based approach.

An example of the memory-based approach is the window-Q architecture of Lin & Mitchell [62]. It is an attempt to extend the Q-learning algorithm to non-Markovian domains. For input, it uses not only the current observation, but also a sliding window over the past N observations and actions. Unfortunately, it is seldom possible to know the necessary window size in advance, and if N is chosen too small, the problem will still be non-Markovian. In addition, the size of the training problem grows

exponentially in the size of the window. Still, this may be a useful method in cases where it is known that only a small memory depth is needed.

Lin & Mitchell [62] also describe two additional architectures that attempt to distill relevant features out of the past observations and actions (much like sufficient statistics in control theory). Both architectures use recurrent neural networks trained with gradient descent. The networks are supposed to learn to recognize and store the features in their recurrent connections. In the recurrent-model architecture one network is trained to predict the next observation and payoff. A second network then attempts to learn the Q function using the internal representation formed by the first network. In the recurrent-Q architecture there is only one network, and it tries to minimize the error between temporally successive utility predictions. It makes sense to only try to learn the features necessary to correctly predict rewards, rather than learning a complete model of the environment. However, it is far from clear that the recurrent-Q architecture should work in principle, since unlike the recurrent-model architecture, the training information is based partly on the network's own (possibly incorrect) predictions. Another problem with both recurrent architectures is that it is again difficult to know what the size of the networks should be ahead of time.

Ring [90] describes a non-recurrent network architecture for learning sequential tasks. The architecture incrementally adds "high level" units as training progresses. A unit is added whenever a weight is being pulled strongly in both directions. The unit is created to determine the contexts in which the weight is pulled in each direction. It learns to utilize the context from the previous time-step to dynamically adjust the weight at the current time step. The architecture is faster to train on supervised tasks than recurrent networks. However, it suffers from several difficulties. For long time delays, a large string of units must be created, and the network does not then generalize to other time delays. There is also a problem with inherently stochastic environments, as the architecture may try to keep adding more and more units.

There are several new methods that deal with hidden state by constructing hidden Markov models (HMMs). In the *Perceptual Distinctions Approach* [29] developed by Chrisman, the agent learns to make predictions about the environment by estimating the underlying state of the environment using an HMM. The HMM estimates the state based on not only the current perceptions, but also on estimates of the previous state. The influence of the previous state estimate implicitly incorporates a memory of past perceptions and actions. The main idea behind this approach is that if the HMM is able to make accurate predictions, then its state representation should also be sufficient to restore the Markov property.

Rabiner & Juang [89] provide an excellent overview of HMMs. A hidden Markov model consists of:

- a set of states $X = \{x_1, \ldots, x_N\}$,
- a set of observations $Y = \{y_1, \ldots, y_M\}$,
- state transition probabilities $a_{ij} = Pr[x_j(t+1)|x_i(t)]$,
- and observation probabilities $b_{ij} = Pr[y_j(t)|x_i(t)]$.

There are algorithms for adjusting the parameters of an HMM to better account for some sequence of observations (allowing the model to be *trained*), and for estimating the state occupation probabilities after some sequence of observations (allowing the model to be *used*). One difficulty with this approach is that the number of underlying states needed in the HMM is not known a priori. Chrisman [29] and McCallum [74] both add states incrementally by splitting existing ones.

State-splitting was also used by Chapman & Kaelbling [28] in a reinforcement learning context. They built an agent to play a videogame called *Amazon*. The visual system of their agent had 100 input bits. With that large of an input space, each input was so rare that the agent needed some kind of input generalization.

Their solution was to have the agent start with a single equivalence class, and make incremental refinements based on statistical tests. In order for the scheme to work, the bits had to be relevant in isolation.

In Chrisman's approach, states are split incrementally based on statistical tests about environmental predictions. McCallum [74] developed an approach similar to Chrisman's that he calls *Utile Distinction Memory* (UDM). The main difference is that states are split in order to increase the ability to predict rewards rather than perceptions. In a stochastic environment, there will be a large amount of variance in the rewards. So splitting states simply on that basis would not discriminate which splits would be valuable. Instead, states are split when reward predictions are significantly different depending on the incoming transition that was traversed. UDM solves problems in which a conjunction of perceptions in sequence predicts reward, but only if they each predict reward on their own. The relevance of pieces of information must be detectable in isolation. This limits the feasibility of UDM in environments where there is extended concealment of crucial features.

McCallum [75] has developed one of the most promising approaches to RL with hidden state, called *instance-based state identification*, which applies memory-based learning techniques such as nearest-neighbor to perception-action-reward sequences. He reports orders of magnitude faster learning than previous memory-based hidden state techniques.

In spite of the recent progress that has been made in dealing with hidden state, it is still an extremely difficult problem that no current method is able to address in a computationally tractable manner. However, section 6.7 describes encouraging experiments from a real-world domain with hidden state information, where an RL algorithm that did not specifically address the issue of hidden state was robust enough to degrade gracefully as the level of hidden state was increased. In the next chapter, we introduce this domain in detail.

CHAPTER 4

ELEVATOR GROUP CONTROL

This chapter introduces the problem of elevator group control, which is our primary testbed for multi-agent reinforcement learning. It is a familiar problem to anyone who has ever used an elevator system, but in spite of its conceptual simplicity, it poses significant difficulties. Elevator systems operate in high–dimensional continuous state spaces and in continuous time as discrete event dynamic systems. Their states are not fully observable and they are non-stationary due to changing passenger arrival rates. An optimal policy for elevator group control is not known, so we use existing control algorithms as a standard for comparison. The elevator domain provides an opportunity to compare parallel and distributed control architectures where each controller controls one elevator car, and to monitor the amount of degradation that occurs as the controllers face increasing levels of incomplete state information.

A schematic diagram of an elevator system [61] is presented in figure 4.1. The elevators cars are represented as filled boxes in the diagram. '+' represents a *hall call* or someone wanting to *enter* a car. '−' represents a *car call* or someone wanting to *leave* a car. The left side of a shaft represents upward moving cars and calls. The right side of a shaft represents downward moving cars and calls. Cars therefore move in a clockwise direction around the shafts.

The first section of this chapter considers the nature of different passenger arrival patterns, and their implications. The next section reviews a variety of elevator control strategies from the literature. The last section describes the particular simulated elevator system that will be the focus in the remainder of this dissertation.

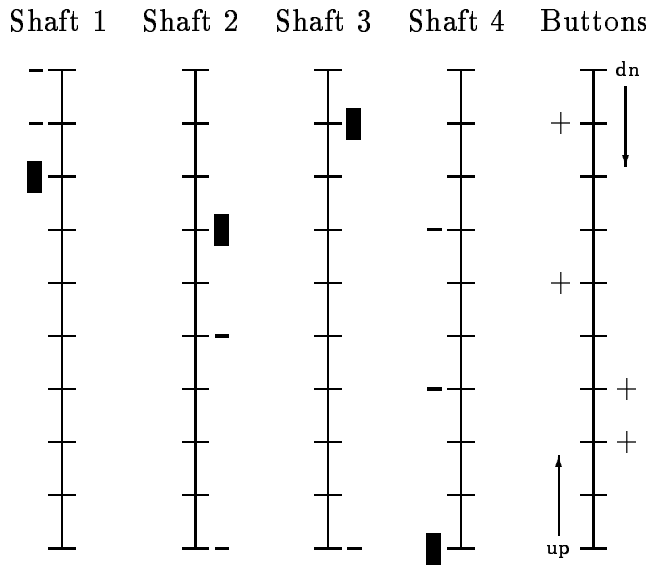

Figure 4.1 *Elevator system schematic diagram.*

4.1 Passenger Arrival Patterns

Elevator systems are driven by passenger arrivals. Arrival patterns vary during the course of the day. In a typical office building, the morning rush hour brings a peak level of up traffic, while a peak in down traffic occurs during the afternoon. Other parts of the day have their own characteristic patterns. Different arrival patterns have very different effects, and each pattern requires its own analysis. Up-peak and down-peak elevator traffic are not simply equivalent patterns in opposite directions, as one might initially guess. Down-peak traffic has many arrival floors and a single destination, while up-peak traffic has a single arrival floor and many destinations. This distinction has significant implications. For example, in light up traffic, the average passenger waiting times can be kept very low by keeping idle cars at the lobby where they will be immediately available for arriving passengers. In light down traffic, waiting times will be longer since it is not possible to keep an idle car at every upper floor in the building, and therefore additional waiting time will be incurred

while cars move to service hall calls. The situation is reversed in heavy traffic. In heavy up traffic, each car may fill up at the lobby with passengers desiring to stop at many different upper floors. The large number of stops will cause significantly longer round-trip times than in heavy down traffic, where each car may fill up after only a few stops at upper floors. For this reason, down-peak handling capacity is much greater than up-peak capacity. Siikonen [101] illustrates these differences in an excellent graph obtained through extensive simulations.

Since up-peak handling capacity is a limiting factor, elevator systems are designed by predicting the heaviest likely up-peak demand in a building, and then determining a configuration that can accomodate that demand. If up-peak capacity is sufficient, then down-peak generally will be also. Up-peak traffic is the easiest type to analyze, since all passengers enter cars at the lobby, their destination floors are serviced in ascending order, and empty cars then return to the lobby. The standard capacity calculations [102, 101] assume that each car leaves the lobby with M passengers (80 to 100 percent of its capacity) and that the average passenger's likelihood of selecting each destination floor is known. Then probability theory is used to determine the average number of stops needed on each round trip. From this one can estimate the average round trip time τ. The five minute handling capacity C equals $\frac{300ML}{\tau}$, where L is the number of cars. The *interval* $I = \frac{\tau}{L}$ represents the average amount of time between car arrivals to the lobby. Assuming that the cars are evenly spaced, the average waiting time is one half the interval. In reality, the average wait is somewhat longer.

Dividing a building into zones significantly increases the up-peak capacity, but at the expense of much longer intervals, and thus longer average wait times. Strakosch [102] provides an example of this in a 12 floor building with 6 cars. By dividing the building into an upper and a lower zone, and assigning 3 cars to each zone, the number of stops needed and thus the round trip times can be reduced. However, the

interval increases since only 3 cars service each zone. Overall, a 41 percent increase in capacity was achieved at the price of a 42 percent increase in the interval. Thus, when zoning is used in hi-rise buildings, enough cars must be planned to keep the interval times at an acceptable level.

A technique known as *channeling* uses additional hardware to improve up-peak service. Displays placed in the lobby indicate the floors each car will be servicing. This reduces the wait times and the travel times, particularly to the highest floors, by reducing the number of stops each car needs to make. Mitsubishi's latest elevator system [116] also adds destination buttons in the lobby. When a passenger presses a destination button, the controller immediately allocates the call to an appropriate car, activates its hall lantern and chime to show the passenger where to go, and updates the service area indicators. The Schindler Elevator Company recently developed a system that has destination buttons at all of the hall landings [48]. There are no buttons inside the cars, and that helps to offset the cost of the additional keypads at the landings. With this additional hardware, the controller has more information about passenger destinations that it can use while making its decisions, allowing it to reduce the average wait and travel times. However, the number of states that it can distinguish is vast, making the search for an optimal solution even more complex. One drawback of this hardware is that a passenger can misuse it by pressing more than one button at a time [60]. With an appropriate state representation, it should be possible in principle to control such a system by applying the same reinforcement learning techniques we describe in this dissertation. For our present purposes we will restrict our attention to the more familiar elevator systems with up and down hall call buttons.

The only control decisions in pure up-peak traffic are to determine when to open and close the elevator doors at the lobby. These decisions affect how many passengers will board an elevator at the lobby. Once the doors have closed, there is really no

choice about the next actions: the car calls registered by the passengers must be serviced in ascending order and the empty car must then return to the lobby. Pepyne & Cassandras [87] show that the optimal policy for handling pure up-peak traffic is a threshold-based policy that closes the doors after an optimal number of passengers have entered the car. The optimal threshold depends upon the traffic intensity, and may also be affected by the number of car calls already registered and by the state of the other cars. Another commonly used up-peak strategy involves selecting a *dwell time*, which is the amount of time a car will spend waiting for the next passenger to arrive [16]. It is reinitialized every time a passenger enters the car. Finding an optimal threshold or dwell time for a given traffic intensity would involve minimizing a function (the expected wait time) of one variable (the threshold or dwell time). Evaluations of the function could be performed using Monte Carlo simulations, which would not be at all computationally prohibitive. Pepyne & Cassandras [86] suggest an even more efficient method called *concurrent estimation* which is based on the theory of perturbation analysis and involves using a single sequence of events to drive a set of simulators each running under a different value of the control variable.

Reinforcement learning could be applied to this problem in two different ways. The simpler approach would be to treat it as a stochastic function optimization problem as described above, and apply non-sequential RL algorithms with the reinforcement feedback being determined by simulation. The more complicated approach would be to treat it as a sequential decision problem. This would allow more of the state variables to be considered in the decision making process. The decision about whether to close the doors could potentially be made at each instant, and so a continuous time algorithm such as advantage updating [5] could be used.

Of course, up-peak traffic is seldom completely pure. Some method must be used to deal with any down hall calls. Making down stops increases the round trip time but allows passengers to be carried in both directions. Having a basement or multiple

entrance floors also adds significantly to the round trip times. For this reason, it is often advisable to have an escalator provide service between entrance floors.

Two way traffic comes in two varieties. In two way *lobby* traffic, up-moving passengers arrive at the lobby and down-moving passengers depart at the lobby. Compared with pure up traffic, the round trip times will be longer, but more passengers will be served. In two way *interfloor* traffic, most passengers travel between floors other than the lobby. Interfloor traffic is more complex than lobby traffic in that it requires almost twice as many stops per passenger, further lengthening the round trip times.

Two way and down-peak traffic patterns require many more decisions than does pure up traffic. After leaving the lobby, a car must decide how high to travel in the building before turning, and at what floors to make additional pickups. Because more decisions are required in a wider variety of contexts, more control strategies are also possible in two way and down-peak traffic situations. For this reason, a down-peak traffic pattern is used as a testbed for this dissertation. Before describing the testbed in detail, we review various elevator control strategies from the literature.

4.2 Elevator Control Strategies

The oldest relay-based automatic controllers used the principle of *collective control* [102, 101], where cars always stop at the nearest call in their running direction. One drawback of this scheme is that there is no means to avoid the phenomenon called *bunching*, where several cars arrive at a floor at about the same time, making the interval, and thus the average waiting time, much longer. Advances in electronics, including the advent of microprocessors, made possible more sophisticated control policies.

It is useful to distinguish between *greedy* and *non-greedy* algorithms. Greedy algorithms perform immediate call assignment, that is, they assign hall calls to cars when they are first registered, and never reconsider those assignments. Non-greedy algo-

rithms postpone their assignments or reconsider them in light of updated information they may receive about additional hall calls or passenger destinations. Greedy algorithms give up some measure of performance due to their lack of flexibility, but also require less computation time. In western countries, an arriving car generally signals waiting passengers when it begins to decelerate [101], allowing the use of a non-greedy algorithm. The custom in Japan is to signal the car assignment immediately upon call registration. This type of signalling requires the use of a greedy algorithm.

The approaches to elevator control discussed in the literature generally fit into the following categories, often more than one category. Unfortunately the descriptions of the proprietary algorithms are often rather vague, since they are written for marketing purposes, and are specifically not intended to be of benefit to competitors. For this reason, it is difficult to ascertain the relative performance levels of many of these algorithms, and there is no accepted definition of the current state of the art [81].

4.2.1 Zoning Approaches

The Otis Elevator Company has used zoning as a starting point in dealing with various traffic patterns [102]. Each car is assigned a zone of the building. It answers hall calls within its zone, and parks there when it is idle. The goal of the zoning approach is to keep the cars reasonably well separated and thus keep the interval down. The drawback of this approach seems to be a significant loss of flexibility. Several papers include zoning in a comparison with other strategies [6, 16]. Sakai & Kurosawa [94] of Hitachi describe a concept called *area control* that is related to zoning. If possible, it assigns a hall call to a car that already must stop at that floor due to a car call. Otherwise, a car within an area α of the hall call is assigned if possible. The area α is a control parameter that affects both the average wait time and the power consumption. A learning function unit collects time-varying traffic data and learns the traffic flow patterns of the building including annual, weekly, and

daily patterns. An artificial intelligence unit simulates the system with various values of α for each traffic flow pattern and selects appropriate values.

4.2.2 Search-Based Approaches

Another control strategy is to search through the space of all possible car assignments, selecting the one that optimizes some criterion such as the average waiting time. Tobita et al [112] of Hitachi describe a system where car assignment occurs when a hall button is pressed. They indicate that their main contribution was to attempt to reduce not only the wait time, but also travel time and crowding, and to let the building manager decide the priorities. They assign the car that minimizes a weighted sum of predicted wait time, travel time, and riding ratio. A fuzzy rule-based system is used to pick the coefficients and estimating functions. Simulations are used to verify their effectiveness. This scheme seems reminiscent of the Relative System Response (RSR) algorithm developed by Otis and patented in 1982. RSR evaluates the amount of time each car would take to serve a new hall call given the car's current state and current hall call assignments. It generally selects the car that can respond most quickly. Pang [82] advances an extremely simplistic blackboard system with only five rules that pursues essentially the same greedy strategy, assigning calls to the cars that can respond most quickly. Because of their greedy commitment to assignments, it seems doubtful that these algorithms would work as well as algorithms that consider re-shuffling their assignments.

Receding horizon controllers are examples of such non-greedy search-based approaches. After every event, they perform an expensive search for the best assignment of hall calls assuming no new passenger arrivals. Closed-loop control is achieved by re-calculating a new open-loop plan after every event. The weaknesses of this approach are its computational demands, and its lack of consideration of future arrivals. Examples of receding horizon controllers are Finite Intervisit Minimization (FIM) and

Empty the System Algorithm (ESA) [6]. FIM attempts to minimize squared waiting times and ESA attempts to minimize the length of the current busy period, the same objective as Levy et al [60].

4.2.3 Rule-Based Approaches

In some sense, all control policies could be considered rule-based: IF situation THEN action. However, here we are more narrowly considering the type of production systems commonly used in Artificial Intelligence. Ujihara & Tsuji [117] of Mitsubishi describe the AI-2100 system. It uses expert-system and fuzzy-logic technologies. They claim that experts in group-supervisory control have the experience and knowledge necessary to shorten waiting times under various traffic conditions, but admit that expert knowledge is fragmentary, hard to organize, and difficult to incorporate. They created a rule base by comparing the decisions made by a conventional algorithm with decisions determined by simulated annealing. The discrepancies were then analyzed by the experts, whose knowledge about solving such problems was used to create fuzzy control rules. The fuzziness lies in the IF part of the rules. Ujihara & Amano [116] describe the latest changes to the AI-2100 system. A previous version used a fixed evaluation formula based on the current car positions and call locations. A more recent version considers future car positions and probable future hall calls. For example, one rule is IF (there is a hall call registered on an upper floor) AND (there are a large number of cars ascending towards the upper floors) THEN (assign one of the ascending cars on the basis of estimated time of arrival). Note that this is an immediate call allocation algorithm, and the consequent of this particular rule about assigning cars on the basis of estimated time of arrival bears some similarity to the greedy search-based algorithms described above. One clever option on their system is a graphical display of the estimated waiting time as an hourglass with sand trickling down, where the amount of sand left indicates the remaining waiting time.

4.2.4 Other Heuristic Approaches

The Longest Queue First (LQF) algorithm assigns upward moving cars to the longest waiting queue, and the Highest Unanswered Floor First (HUFF) algorithm assigns upward moving cars to the highest queue with people waiting [6]. Both of these algorithms are designed specifically for down-peak traffic. They assign downward moving cars to any unassigned hall calls they encounter. The Dynamic Load Balancing (DLB) algorithm attempts to keep the cars evenly spaced by assigning contiguous non-overlapping sectors to each car in a way that balances their loads [61]. DLB is a non-greedy algorithm because it reassigns sectors after every event.

Pepyne [85] describes a Q-learning elevator dispatching algorithm that uses lookup tables rather than neural networks, and has a drastically reduced state representation containing less than twelve thousand states. His work contains the clever idea of using three primitive actions, stop, continue, and turn, which keeps the action set small and allows a decentralized approach to the problem. Another interesting idea in his work is for all the cars to share one set of Q-values, which we refer to as a parallel architecture. However, his cost function is flawed. It does not properly reflect the quantity he is trying to minimize: the average waiting time. It makes turning appear artificially attractive, and stopping artificially unattractive. As a result of this, and possibly other problems such as aliasing, his results are much worse than random. After this, he takes a different approach, purposely giving rewards in an ad hoc, heuristic way, rather than trying to base them directly on the quantity to be optimized. Here, the most important learning is done by the human who tries to find a set of heuristic rewards that will cause good performance. The trouble with using heuristics for rewards is that their quality places a limit on the eventual performance of the system. It also raises the question of why Q-learning should be used at all, when the heuristics could be implemented directly.

4.2.5 Adaptive and Learning Approaches

Imasaki et al [51] of Toshiba use a fuzzy neural network to predict passenger waiting time distributions for various sets of control parameters. Their system adjusts the parameters by evaluating alternative candidate parameters with the neural network. They do not explain what control algorithm is actually used, what its parameters are, or how the network is trained.

Hitachi researchers [43, 112] use a greedy RSR-like control algorithm that combines multiple objectives such as wait time, travel time, crowding, and power consumption. The weighting of these objectives is accomplished using parameters that are tuned online. A module called the learning function unit collects traffic statistics and attempts to classify the current traffic pattern. The tuning function unit generates parameter sets for the current traffic pattern and tests them using a built-in simulator. The best parameters are then used to control the system. Searching the entire parameter space would be prohibitively expensive, so heuristics are used about which parameter sets to test.

Levy et al [60] use dynamic programming (DP) offline to minimize the expected time needed for completion of the current busy period. The idea of minimizing the busy period also appears in the receding horizon controller ESA (Empty the System Algorithm) [6]. $K(b, d)$ is the time needed for completion of the busy period given state b and decision d, and all subsequent decisions optimal. This definition, with its state-action pair, seems almost prophetic of the Q-learning algorithm developed 12 years later by Watkins [119]. No discount factor is used, since it is assumed that the values will all be finite. The major difference between this and Q-learning is that it must be performed offline since it uses a model of the transition probabilities of the system and performs sweeps of the state space. The trouble with using DP to calculate an optimal policy is that the state space is very large, requiring drastic simplification. Levy et al use several methods to keep the size of the state space

manageable: they consider a building with only 2 cars and 8 floors, where the number of buttons that can be on simultaneously is restricted, the state of the buttons are restricted to binary values (i.e., elapsed times are discarded), and the cars have unlimited capacity. Zoning is also mentioned as a way to reduce the number of states in higher buildings. Construction of the transition probability matrix is the principle part of the procedure, and it assumes that the intensity of Poisson arrivals at each floor is known. Value iteration or policy iteration is then performed to obtain the solution. They claim that policy iteration is likely to converge much faster than value iteration. They also outline a way of positioning the cars after the busy period has ended, utilizing the busy period solution already obtained. They note that a new action cannot be assigned to a car until it has completed its last action. In other words, it cannot change its mind based on new information during its trip. One fascinating part of their work is their effort to distill down the policy in order to understand what it is doing. For a given positioning of the cars, they attempt to express as simply as possible the button configurations for which the policy prescribes action d. In spite of their efforts at simplification, the policy they present in tabular form is still very difficult to understand, and it seems doubtful that their minimization procedure could be effective if the number of cars were increased from 2 to 4, or if the elapsed times of the hall buttons were taken into account.

Markon et al [70] have devised a system that trains a neural network to perform immediate call allocation. There are three phases of training. In phase one, while the system is being controlled by an existing controller (the FLEX-8820 Fuzzy/AI Group Control System of Fujitec), supervised learning is used to train the network to predict the hall call service times. This first phase of training is used to learn an appropriate internal representation, i.e., weights from the input layer to the hidden layer of the network. At the end of the first phase of training, those weights are fixed. In phase two, the output layer of the network is retrained to emulate the existing controller.

In phase three, single weights in the output layer of the network are perturbed, and the resulting performance is measured on a traffic sample. The weights are then modified in the direction of improved performance. This can be viewed as a form of non-sequential reinforcement learning. The single-stage reward is determined by measuring the system's performance on a traffic sample.

Their input representation uses 25 units for each car, and their output representation uses one unit for each car. Hall calls are allocated to the car corresponding to the output unit with the highest activation. They also describe a very clever way of incorporating the permutational symmetry of the problem into the architecture of their network. As they say, "If the states of two cars are interchanged, the outputs should also be interchanged." This is done by having as many sets of hidden units as there are cars, and then explicitly linking together the appropriate weights.

Their system was tested in a simulation with 6 cars and 15 floors. In a "typical building", trained on 900 passengers per hour, there was a very small improvement of around 1 second in the average wait time over the existing controller. In a more "untypical" building with uniformly distributed origin and destination floors and 1500 passengers per hour, the improvement in average wait time was almost 4 seconds.

One advantage of this system is that it can maintain an adequate service level from the beginning since it starts with a pre-existing controller. On the other hand, it is not clear whether this also may trap the controller in a suboptimal region of policy space. It would be very interesting to use this centralized, immediate call allocation network architecture as part of a *sequential* reinforcement learning algorithm.

4.3 The Elevator Testbed

The particular elevator system we examine in this dissertation is a simulated 10-story building with 4 elevator cars. The simulator was written by Lewis [61]. Passenger arrivals at each floor are assumed to be Poisson, with arrival rates that

vary during the course of the day. Our simulations use a traffic profile [6] which dictates arrival rates for every 5-minute interval during a typical afternoon down-peak rush hour. Table 4.1 shows the mean number of passengers arriving at *each* of floors 2 through 10 during each 5-minute interval who are headed for the lobby. In addition, there is inter-floor traffic which varies from 0% to 10% of the traffic to the lobby.

Table 4.1 *The down–peak traffic profile.*

Time	00	05	10	15	20	25	30	35	40	45	50	55
Rate	1	2	4	4	18	12	8	7	18	5	3	2

4.3.1 System Dynamics

The system dynamics are approximated by the following parameters:

- Floor time (the time to move one floor at maximum speed): 1.45 secs.
- Stop time (the time needed to decelerate, open and close the doors, and accelerate again): 7.19 secs.
- Turn time (the time needed for a stopped car to change direction): 1 sec.
- Load time (the time for one passenger to enter or exit a car): random variable from a $20th$ order truncated Erlang distribution with a range from 0.6 to 6.0 secs and a mean of 1 sec.
- Car capacity: 20 passengers.

The simulator is quite detailed, and is certainly realistic enough for our purposes. However, a few minor deviations from reality should be noted. In the simulator, a car can accelerate to full speed or decelerate from full speed in a distance of only

one half of a floor, while the distances would be somewhat longer in a real system. Thus, the simulated acceleration and deceleration times are always the same, but in a real system, they will vary depending on the speed of the elevator. For example, an express car descending from the tenth floor at top speed will take longer to decelerate at the first floor than a car that is descending from the second floor. The simulator also allows the cars to commit to stopping at a floor when they are only one half of a floor away. Though this is not realistic for cars moving at top speed, the concept of making decisions regarding the next floor where the car could commit to stopping is valid.

Although the elevator cars in this system are homogeneous, the learning techniques proposed in this dissertation can also be used in more general situations, e.g., where there are several express cars or cars that only service some subset of the floors.

4.3.2 State Space

The state space is continuous because it includes the elapsed times since any hall calls were registered, which are real–valued. Even if these real values are approximated as binary values, the size of the state space is still immense. Its components include 2^{18} possible combinations of the 18 hall call buttons (up and down buttons at each landing except the top and bottom), 2^{40} possible combinations of the 40 car buttons, and 18^4 possible combinations of the positions and directions of the cars (rounding off to the nearest floor). Other parts of the state are not fully observable, for example, the exact number of passengers waiting at each floor, their exact arrival times, and their desired destinations. Ignoring everything except the configuration of the hall and car call buttons and the approximate position and direction of the cars, we obtain an extremely conservative estimate of the size of a discrete approximation to the continuous state space:

$$2^{18} \cdot 2^{40} \cdot 18^4 \approx 10^{22} \text{ states.}$$

4.3.3 Control Actions

Each car has a small set of primitive actions. If it is stopped at a floor, it must either "move up" or "move down". If it is in motion between floors, it must either "stop at the next floor" or "continue past the next floor". Due to passenger expectations, there are two constraints on these actions: a car cannot pass a floor if a passenger wants to get off there and cannot turn until it has serviced all the car buttons in its present direction. We also added three additional heuristic constraints in an attempt to build in some primitive prior knowledge: a car cannot stop at a floor unless someone wants to get on or off there, it cannot stop to pick up passengers at a floor if another car is already stopped there, and given a choice between moving up and down, it should prefer to move up (since the down-peak traffic tends to push the cars toward the bottom of the building). Because of this last constraint, the only real choices left to each car are the stop and continue actions. The actions of the elevator cars are executed asynchronously since they may take different amounts of time to complete.

4.3.4 Performance Criteria

The performance objectives of an elevator system can be defined in many ways. One possible objective is to minimize the average *wait* time, which is the time between the arrival of a passenger and his entry into a car. Another possible objective is to minimize the average *system* time, which is the sum of the wait time and the travel time. A third possible objective is to minimize the percentage of passengers that wait longer than some dissatisfaction threshold (usually 60 seconds). Another common objective is to minimize the average *squared* wait time. We chose this latter performance objective since it tends to keep the wait times low while also encouraging fair service. For example, wait times of 2 and 8 seconds have the same average (5

seconds) as wait times of 4 and 6 seconds. But the average *squared* wait times are different (34 for 2 and 8 versus 26 for 4 and 6).

CHAPTER 5

THE ALGORITHM AND NETWORK ARCHITECTURE

This chapter describes the multi-agent reinforcement learning algorithm that we have applied to elevator group control. In our scheme, each agent is responsible for controlling one elevator car. Each agent uses a modification of Q-learning for discrete-event systems. Together, they employ a collective form of reinforcement learning. We begin the chapter by describing the modifications needed to extend Q-learning into a discrete-event framework, and derive a method for determining appropriate reinforcement signals in the face of uncertainty about exact passenger arrival times. Then we describe the algorithm, the feedforward networks used to store the Q-values, and the distinction between parallel and distributed versions of the algorithm.

5.1 Discrete-Event Reinforcement Learning

Elevator systems can be modeled as *discrete event* systems [26], where significant events (such as passenger arrivals) occur at discrete times, but the amount of time between events is a real-valued variable. In such systems, the constant discount factor γ used in most discrete-time reinforcement learning algorithms is inadequate. This problem can be approached using a variable discount factor that depends on the amount of time between events [24]. In this case, the cost–to–go is defined as an integral rather than as an infinite sum, as follows:

$$\sum_{t=0}^{\infty} \gamma^t c_t \quad \text{becomes} \quad \int_0^{\infty} e^{-\beta\tau} c_\tau d\tau,$$

where c_t is the immediate cost at discrete time t, c_τ is the instantaneous cost at continuous time τ (the sum of the squared wait times of all currently waiting passengers),

and β controls the rate of exponential decay. $\beta = 0.01$ in the experiments described in this dissertation. Since the wait times are measured in seconds, we scale down the instantaneous costs c_τ by a factor of 10^6 to keep the cost–to–go values from becoming exceedingly large.

Because elevator system events occur randomly in continuous time, the branching factor is effectively infinite, which complicates the use of algorithms that require explicit lookahead. Therefore, we employ a discrete event version of the Q-learning algorithm since it considers only events encountered in actual system trajectories and does not require a model of the state transition probabilities. The Q-learning update rule [119] takes on the following discrete event form:

$$\Delta\hat{Q}(x,a) = \alpha \cdot [\int_{t_x}^{t_y} e^{-\beta(\tau - t_x)} c_\tau d\tau + e^{-\beta(t_y - t_x)} \min_b \hat{Q}(y,b) - \hat{Q}(x,a)],$$

where action a is taken from state x at time t_x, the next decision is required from state y at time t_y, α is the learning rate parameter, and c_τ and β are defined as above. $e^{-\beta(t_y - t_x)}$ acts as a variable discount factor that depends on the amount of time between events.

Bradtke and Duff [24] consider the case where c_τ is constant between events. We extend their formulation to the case where c_τ is quadratic, since the goal is to minimize squared wait times. The integral in the Q-learning update rule then takes the form:

$$\int_{t_x}^{t_y} \sum_p e^{-\beta(\tau - t_x)} (\tau - t_x + w_p)^2 d\tau,$$

where w_p is the amount of time each passenger p waiting at time t_y has already waited at time t_x. (Special care is needed to handle any passengers that begin or end waiting between t_x and t_y. See section 5.2.1.)

This integral can be solved by parts to yield:

$$\sum_p e^{-\beta w_p} [\frac{2}{\beta^3} + \frac{2w_p}{\beta^2} + \frac{w_p^2}{\beta}] - e^{-\beta(w_p + t_y - t_x)} [\frac{2}{\beta^3} + \frac{2(w_p + t_y - t_x)}{\beta^2} + \frac{(w_p + t_y - t_x)^2}{\beta}].$$

A difficulty arises in using this formula since it requires knowledge of the waiting times of all waiting passengers. However, only the waiting times of passengers who

press hall call buttons will be known in a real elevator system. The number of subsequent passengers to arrive and their exact waiting times will not be available. We examine two ways of dealing with this problem, which we call *omniscient* and *online* reinforcement schemes.

The simulator has access to the waiting times of all passengers. It could use this information to produce the necessary reinforcement signals. We call these *omniscient* reinforcements, since they require information that is not available in a real system. Note that it is not the controller that receives this extra information, however, but rather the *critic* that is evaluating the controller. For this reason, even if omniscient reinforcements are used during the design phase of an elevator controller on a simulated system, the resulting trained controller can be installed in a real system without requiring any extra knowledge.

The other possibility is to train using only information that would be available to a real system online. Such *online* reinforcements assume only that the waiting time of the first passenger in each queue is known (which is the elapsed button time). If the Poisson arrival rate λ for each queue is known or can be estimated, the Gamma distribution can be used to estimate the arrival times of subsequent passengers. The time until the n^{th} subsequent arrival follows the Gamma distribution $\Gamma(n, \frac{1}{\lambda})$. For each queue, subsequent arrivals will generate the following expected costs during the first b seconds after the hall button has been pressed:

$$\sum_{n=1}^{\infty} \int_0^b (\text{prob } n^{th} \text{ arrival occurs at time } \tau) \cdot (\text{cost given arrival at time } \tau)\, d\tau$$

$$= \sum_{n=1}^{\infty} \int_0^b \frac{\lambda^n \tau^{n-1} e^{-\lambda\tau}}{(n-1)!} \int_0^{b-\tau} w^2 e^{-\beta(w+\tau)} dw\, d\tau \;=\; \int_0^b \int_0^{b-\tau} \lambda w^2 e^{-\beta(w+\tau)} dw\, d\tau.$$

This integral can also be solved by parts to yield expected costs. A general solution is provided in section 5.2.2. As described in section 6.4, using online reinforcements produces results that are almost as good as those obtained with omniscient reinforcements.

5.2 Collective Discrete-Event Q-Learning

Elevator system events can be divided into two types. Events of the first type are important in the calculation of waiting times and therefore also reinforcements. These include passenger arrivals and transfers in and out of cars in the omniscient case, or hall button events in the online case. The second type are car arrival events, which are potential decision points for the RL agents controlling each car. A car that is in motion between floors generates a car arrival event when it reaches the point where it must decide whether to stop at the next floor or continue past the next floor. In some cases, cars are constrained to take a particular action, for example, stopping at the next floor if a passenger wants to get off there. An agent faces a decision point only when it has an unconstrained choice of actions.

5.2.1 Calculating Omniscient Reinforcements

Omniscient reinforcements are updated incrementally after every passenger arrival event (when a passenger arrives at a queue), passenger transfer event (when a passenger gets on or off of a car), and car arrival event (when a control decision is made). These incremental updates are a natural way of dealing with the discontinuities in reinforcement that arise when passengers begin or end waiting between a car's decisions, e.g., when another car picks up waiting passengers. The amount of reinforcement between events is the same for all the cars since they share the same objective function, but the amount of reinforcement each car receives between its *decisions* is different since the cars make their decisions asynchronously. Therefore, each car i has an associated storage location, $R[i]$, where the total discounted reinforcement it has received since its last decision (at time $d[i]$) is accumulated.

At the time of each event, the following computations are performed: Let t_0 be the time of the last event and t_1 be the time of the current event. For each passenger

p that has been waiting between t_0 and t_1, let $w_0(p)$ and $w_1(p)$ be the total time that passenger p has waited at t_0 and t_1 respectively. Then for each car i,

$$\Delta R[i] = \sum_p e^{-\beta(t_0-d[i])}(\frac{2}{\beta^3}+\frac{2w_0(p)}{\beta^2}+\frac{w_0^2(p)}{\beta}) - e^{-\beta(t_1-d[i])}(\frac{2}{\beta^3}+\frac{2w_1(p)}{\beta^2}+\frac{w_1^2(p)}{\beta}).$$

5.2.2 Calculating Online Reinforcements

Online reinforcements are updated incrementally after every hall button event (signaling the arrival of the first waiting passenger at a queue or the arrival of a car to pick up any waiting passengers at a queue) and car arrival event (when a control decision is made). We assume that online reinforcements caused by passengers waiting at a queue end immediately when a car arrives to service the queue, since it is not possible to know exactly when each passenger boards a car. The Poisson arrival rate λ for each queue is estimated as the reciprocal of the last inter-button time for that queue, i.e., the amount of time from the last service until the button was pushed again. However, a ceiling of $\hat{\lambda} \leq 0.04$ passengers per second is placed on the estimated arrival rates to prevent any very small inter-button times from creating huge penalties that might destabilize the cost–to–go estimates.

At the time of each event, the following computations are performed: Let t_0 be the time of the last event and t_1 be the time of the current event. For each hall call button b that was active between t_0 and t_1, let $w_0(b)$ and $w_1(b)$ be the elapsed time of button b at t_0 and t_1 respectively. Then for each car i,

$$\begin{aligned}\Delta R[i] = {} & e^{-\beta(t_0-d[i])}\sum_b\{\frac{2\hat{\lambda}_b(1-e^{-\beta(t_1-t_0)})}{\beta^4}+ \\ & (\frac{2}{\beta^3}+\frac{2w_0(b)}{\beta^2}+\frac{w_0^2(b)}{\beta}) - e^{-\beta(t_1-t_0)}(\frac{2}{\beta^3}+\frac{2w_1(b)}{\beta^2}+\frac{w_1^2(b)}{\beta})+ \\ & \hat{\lambda}_b[(\frac{2w_0(b)}{\beta^3}+\frac{w_0^2(b)}{\beta^2}+\frac{w_0^3(b)}{3\beta}) - e^{-\beta(t_1-t_0)}(\frac{2w_1(b)}{\beta^3}+\frac{w_1^2(b)}{\beta^2}+\frac{w_1^3(b)}{3\beta})]\}.\end{aligned}$$

5.2.3 Making Decisions and Updating Q-Values

A car that is in motion between floors generates a car arrival event when it reaches the point where it must decide whether to stop at the next floor or continue past the next floor. In some cases, cars are constrained to take a particular action, for example, stopping at the next floor if a passenger wants to get off there. An agent faces a decision point only when it has an unconstrained choice of actions. The algorithm used by each agent for making decisions and updating its Q-value estimates is as follows:

1. At time t_x, observing state x, car i arrives at a decision point. It selects an action a using the Boltzmann distribution over its Q-value estimates:

$$Pr(stop) = \frac{e^{Q(x,cont)/T}}{e^{Q(x,stop)/T} + e^{Q(x,cont)/T}},$$

   where T is a positive "temperature" parameter that is "annealed" or decreased during training. The value of T controls the amount of randomness in the selection of actions. At the beginning of training, when the Q-value estimates are very inaccurate, high values of T are used, which give nearly equal probabilities to each action. Later in training, when the Q-value estimates are more accurate, lower values of T are used, which give higher probabilities to actions that are thought to be superior, while still allowing some exploration to gather more information about the other actions. As discussed in section 6.3, choosing a slow enough annealing schedule is particularly important in multi-agent settings.

2. Let the next decision point for car i be at time t_y in state y. After *all* cars (including car i) have updated their $R[\cdot]$ values as described in the last two sections, car i adjusts its estimate of $Q(x,a)$ toward the following target value:

$$R[i] + e^{-\beta(t_y - t_x)} \min_{\{stop,cont\}} \hat{Q}(y,\cdot).$$

Car i then resets its reinforcement accumulator $R[i]$ to zero.

3. Let $x \leftarrow y$ and $t_x \leftarrow t_y$. Go to step 1.

5.3 The Networks Used to Store the Q-Values

Using lookup tables to store the Q-values was ruled out for such a large system. Instead, we used feedforward neural networks trained with the error backpropagation algorithm [91]. The networks receive some of the state information as input, and produce Q-value estimates as output. The Q-value estimates can be written as $\hat{Q}(x, a, \phi)$, where ϕ is a vector of the parameters or weights of the networks. The exact weight update equation is:

$$\Delta\phi = \alpha[R[i] + e^{-\beta(t_y - t_x)} \min_{\{stop,cont\}} \hat{Q}(y, \cdot, \phi) - \hat{Q}(x, a, \phi)] \bigtriangledown_\phi \hat{Q}(x, a, \phi),$$

where α is a positive learning rate or stepsize parameter, and the gradient $\bigtriangledown_\phi \hat{Q}(x, a, \phi)$ is the vector of partial derivatives of $\hat{Q}(x, a, \phi)$ with respect to each component of ϕ.

At the start of training, the weights of each network are initialized to be uniform random numbers between -1 and $+1$. Some experiments in this dissertation use separate single-output networks for each action–value estimate, while others use one network with multiple output units, one for each action. Our basic network architecture for pure down traffic uses 47 input units, 20 hidden sigmoid units, and 1 or 2 linear output units. The input units are as follows:

- 18 units: Two units encode information about each of the nine down hall buttons. A real–valued unit encodes the elapsed time if the button has been pushed and a binary unit is on if the button has not been pushed.

- 16 units: Each of these units represents a possible location and direction for the car whose decision is required. Exactly one of these units will be on at any

given time. Note that each car has a different egocentric view of the state of the system.

- 10 units: These units each represent one of the 10 floors where the other cars may be located. Each car has a "footprint" that depends on its direction and speed. For example, a stopped car causes activation only on the unit corresponding to its current floor, but a moving car causes activation on several units corresponding to the floors it is approaching, with the highest activations on the closest floors. No information is provided about *which one* of the other cars is at a particular location.

- 1 unit: This unit is on if the car whose decision is required is at the highest floor with a waiting passenger.

- 1 unit: This unit is on if the car whose decision is required is at the floor with the passenger that has been waiting for the longest amount of time.

- 1 unit: The bias unit is always on.

In the next chapter, we introduce other representations, including some with more restricted state information.

5.4 Parallel and Distributed Implementations

Each elevator car is controlled by a separate Q-learning agent. We experimented with both parallel and decentralized implementations. In parallel implementations, the agents use a central set of shared networks, allowing them to learn from each other's experiences, but forcing them to learn identical policies. In totally decentralized implementations, the agents have their own networks, allowing them to specialize their control policies. In either case, none of the agents has explicit access to the actions of the other agents. Cooperation has to be learned indirectly via the global

reinforcement signal. Each agent faces added stochasticity and non-stationarity because its environment contains other learning agents.

CHAPTER 6

RESULTS AND DISCUSSION

6.1 Basic Results Versus Other Algorithms

Since an optimal policy for the elevator group control problem is unknown, we measured the performance of our algorithm against other heuristic algorithms, including the best of which we were aware. The algorithms were: SECTOR, a sector–based algorithm similar to what is used in many actual elevator systems; DLB, Dynamic Load Balancing, attempts to equalize the load of all cars; HUFF, Highest Unanswered Floor First, gives priority to the highest floor with people waiting; LQF, Longest Queue First, gives priority to the queue with the person who has been waiting for the longest amount of time; FIM, Finite Intervisit Minimization, a receding horizon controller that searches the space of admissible car assignments to minimize a load function; ESA, Empty the System Algorithm, a receding horizon controller that searches for the fastest way to "empty the system" assuming no new passenger arrivals. FIM is very computationally intensive, and would be difficult to implement in real time in its present form. ESA uses queue length information that would not be available in a real elevator system. ESA/nq is a version of ESA that uses arrival rate information to estimate the queue lengths. For more details, see [6]. RLp and RLd denote the RL controllers, parallel and decentralized. The RL controllers were each trained on 60,000 hours of simulated elevator time, which took four days on a 100 MIPS workstation. The results for all the algorithms were averaged over 30 hours of simulated elevator time to ensure their statistical significance. The average waiting times listed below for the trained RL algorithms are correct to within ± 0.13 at a 95%

confidence level, the average squared waiting times are correct to within ± 5.3, and the average system times are correct to within ± 0.27. Table 6.1 shows the results for the traffic profile with down traffic only.

Table 6.1 *Results for down–peak profile with down traffic only.*

Algorithm	AvgWait	SquaredWait	SystemTime	Percent>60 secs
SECTOR	21.4	674	47.7	1.12
DLB	19.4	658	53.2	2.74
BASIC HUFF	19.9	580	47.2	0.76
LQF	19.1	534	46.6	0.89
HUFF	16.8	396	48.6	0.16
FIM	16.0	359	47.9	0.11
ESA/nq	15.8	358	47.7	0.12
ESA	15.1	338	47.1	0.25
RLp	14.8	320	41.8	0.09
RLd	14.7	313	41.7	0.07

Table 6.2 shows the results for the down–peak traffic profile with up and down traffic, including an average of 2 up passengers per minute at the lobby. The algorithm was trained on down–only traffic, yet it generalizes well when up traffic is added and upward moving cars are forced to stop for any upward hall calls.

Table 6.2 *Results for down–peak profile with up and down traffic.*

Algorithm	AvgWait	SquaredWait	SystemTime	Percent>60 secs
SECTOR	27.3	1252	54.8	9.24
DLB	21.7	826	54.4	4.74
BASIC HUFF	22.0	756	51.1	3.46
LQF	21.9	732	50.7	2.87
HUFF	19.6	608	50.5	1.99
ESA	18.0	524	50.0	1.56
FIM	17.9	476	48.9	0.50
RLp	16.9	476	42.7	1.53
RLd	16.9	468	42.7	1.40

Table 6.3 shows the results for the down–peak traffic profile with up and down traffic, including an average of 4 up passengers per minute at the lobby. This time there is twice as much up traffic, and the RL agents generalize extremely well to this new situation.

Table 6.3 *Results for down–peak profile with twice as much up traffic.*

Algorithm	AvgWait	SquaredWait	SystemTime	Percent>60 secs
SECTOR	30.3	1643	59.5	13.50
HUFF	22.8	884	55.3	5.10
DLB	22.6	880	55.8	5.18
LQF	23.5	877	53.5	4.92
BASIC HUFF	23.2	875	54.7	4.94
FIM	20.8	685	53.4	3.10
ESA	20.1	667	52.3	3.12
RLd	18.8	593	45.4	2.40
RLp	18.6	585	45.7	2.49

One can see that both the RL systems achieved very good performance, most notably as measured by system time (the sum of the wait and travel time), a measure that was not directly being minimized. Surprisingly, the decentralized RL system was able to achieve as good a level of performance as the parallel RL system.

6.2 Analysis of Decentralized Results

In view of the outstanding success of the decentralized RL algorithm, several questions suggest themselves: How similar are the policies that the agents have learned to one another and to the policy learned by the parallel algorithmΓ Can the results be improved by using a voting schemeΓ What happens if one agent's policy is used to control all of the carsΓ This section addresses all of these questions.

First the simulator was modified to poll each of the four Q-networks and the parallel Q-network on every decision by every car, and compare their action selections.

During one hour of simulated elevator time, there were a total of 573 decisions required. The four networks were unanimous on 505 decisions (88 percent), they split 3 to 1 on 47 decisions (8 percent), and they split evenly on 21 decisions (4 percent). The parallel network agreed with 493 of the 505 unanimous decisions (98 percent). For some reason, the parallel network tended to favor the STOP action more than the decentralized networks, though that apparently had little impact on the overall performance. The complete results are listed in the following tables:

Table 6.4 *Amount of agreement between decentralized agents.*

Agents Saying STOP	Agents Saying CONTINUE	Number of Instances	Percent
4	0	389	68
3	1	29	5
2	2	21	4
1	3	18	3
0	4	116	20
		573	100

Table 6.5 *Amount of agreement between decentralized and parallel agents.*

Agents Saying STOP	Agents Saying CONTINUE	Number of Instances	Parallel Says STOP	Parallel Says CONT
4	0	389	386	3
3	1	29	27	2
2	2	21	17	4
1	3	18	11	7
0	4	116	9	107

While these results show considerable agreement, there are a minority of situations where the agents disagree. In the next experiment the agents vote on which actions should be selected for all of the cars. In the cases where the agents are evenly split, we examine three ways of resolving the ties: in favor of STOP (RLs), in favor of

CONTINUE (RLc), or randomly (RLr). The following table shows the results of this voting scheme compared to the original decentralized algorithm (RLd). The results are averaged over 30 hours of simulated elevator time on pure down traffic.

Table 6.6 *Comparison with several voting schemes.*

Algorithm	AvgWait	SquaredWait	SystemTime	Percent>60 secs
RLc	15.0	325	41.7	0.09
RLs	14.9	322	41.7	0.10
RLr	14.8	314	41.7	0.12
RLd	14.7	313	41.7	0.07

These results show no significant improvement from voting. In the situations where the agents were evenly split, breaking the ties randomly produced results that were almost identical to those of the original decentralized algorithm. This seems to imply that the agents generally agree on the most important decisions, and disagree only on decisions of little consequence where the action values are very similar.

In the next experiment the agent for a single car selects actions for all the cars. RL1 uses the agent for car 1 to control all the cars, RL2 uses the agent for car 2, and so on. The following table compares these controllers to the original decentralized algorithm (RLd). The results are averaged over 30 hours of simulated elevator time on pure down traffic.

Table 6.7 *Letting a single agent control all four cars.*

Algorithm	AvgWait	SquaredWait	SystemTime	Percent>60 secs
RL1	14.7	315	41.6	0.15
RL2	15.0	324	41.9	0.10
RL3	15.0	333	41.9	0.26
RL4	15.0	324	41.8	0.15
RLd	14.7	313	41.7	0.07

While agent 1 outperformed the other agents, all of the agents performed well relative to the non–RL controllers discussed above. In summary, it appears that all the decentralized and parallel agents learned very similar policies. The similarity of the learned policies may have been caused in part by the symmetry of the elevator system and the input representation we selected, which did not distinguish among the cars. For future work, it would be interesting to see whether agents with input representations that *did* distinguish among the cars would still arrive at similar policies.

6.3 Annealing Schedules

One of the most important factors in the performance of the algorithms is the annealing schedule used to control the amount of exploration performed by each agent. The slower the annealing process, the better the final result. This is illustrated in table 6.8 and figure 6.1, which show the results of one training run with each of a number of annealing rates. The temperature T was annealed according to the schedule: $T = 2.0 * (Factor)^h$, where h represents the hours of training completed. Once again, the results were measured over 30 hours of simulated elevator time. Even though they are somewhat noisy due to not being averaged over multiple training runs, the trend is still quite clear.

Each of the schedules that we tested shared the same starting and ending temperatures. Although the annealing process can be ended at any time with the current Q-value estimates being used to determine a control policy, if the amount of time available for training is known in advance, one should select an annealing schedule that covers a full range of temperatures.

While gradual annealing is important in single–agent RL, it is even more important in multi–agent RL. The tradeoff between exploration and exploitation for an agent

Table 6.8 *The effect of varying the annealing rate.*

Factor	Hours	AvgWait	SquaredWait	SystemTime	Pct>60 secs
0.992	950	19.3	581	44.7	1.69
0.996	1875	17.6	487	43.2	1.17
0.998	3750	15.8	376	42.0	0.28
0.999	7500	15.3	353	41.8	0.30
0.9995	15000	15.7	361	42.4	0.17
0.99975	30000	15.1	335	41.9	0.12
0.999875	60000	14.7	313	41.7	0.07

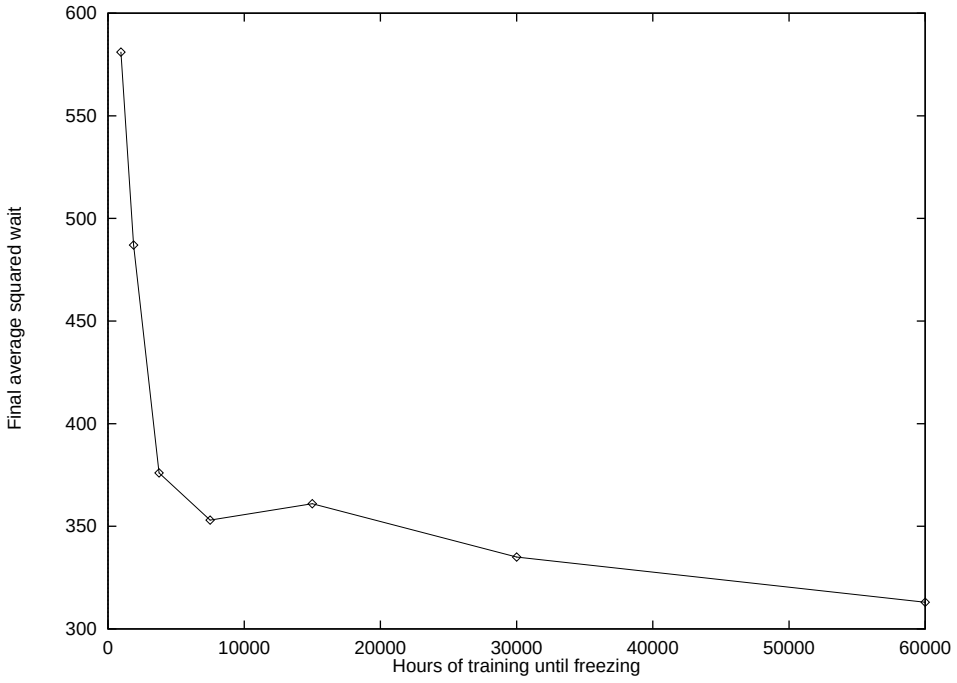

Figure 6.1 *The effect of varying the annealing rate.*

now must also be balanced with the need for other agents to learn in a stationary environment and while that agent is doing its best. At the beginning of the learning process, the agents are all extremely inept. With gradual annealing they are all able to raise their performance levels in parallel. Tesauro [108, 109, 110] notes a slightly different but related phenomenon in the context of zero–sum games, where training with self-play allows an agent to learn with a well–matched opponent during each stage of its development.

6.4 Omniscient Versus Online Reinforcements

This section examines the relative performance of the omniscient and online reinforcements described in section 5.1, given the same network structure and temperature and learning rate schedule. As shown in table 6.9, omniscient reinforcements led to slightly better performance than online reinforcements. This should be of little concern regarding the application of RL to a real elevator system, since one would want to perform the initial training in simulation in any case, not only because of the huge amount of experience needed, but also because performance would be poor during the early stages of training. In a real elevator system, the initial training would be performed using a simulator, and the networks would be fine-tuned on the real system.

Table 6.9 *Omniscient versus online reinforcements.*

Reinforcements	AvgWait	SquaredWait	SystemTime	Pct>60 secs
Omniscient	15.2	332	42.1	0.07
Online	15.3	342	41.6	0.16

6.5 Unbalanced Floor Populations

This section examines the case where the traffic intensity varies from floor to floor. The traffic profile from table 6.2 was used, except that floors 2,4,6,8,10 had 50 percent more traffic and floors 3,5,7,9 had 50 percent less traffic. Two RL controllers were tested. RLg used the input representation described in section 5.3 and was trained with the unbalanced traffic profile minus any up traffic. During testing, RLg was thus forced to generalize to the case with up traffic. RLu had 9 additional real-valued input units representing the elapsed times for the 9 up hall call buttons and was trained with the unbalanced traffic including up traffic. The results are shown in table 6.10. Including the up–button elapsed times and up traffic during training did improve the final performance. Therefore it may also be possible to improve on the results reported in section 6.1 by including this same information. One of the advantages of RL is its ability to adapt to any type of building configuration. However, it appears that equal arrival rates at all floors may be at least as difficult a configuration to control as unequal arrival rates.

Table 6.10 *Results with unbalanced floor populations.*

Algorithm	AvgWait	SquaredWait	SystemTime	Pct>60 secs
SECTOR	28.2	1343	55.8	10.83
DLB	21.7	839	54.2	4.83
BASIC HUFF	22.0	771	51.8	3.72
LQF	21.5	712	50.6	2.73
HUFF	20.3	671	50.8	2.80
ESA	18.6	564	50.3	2.32
FIM	17.9	494	49.6	1.40
RLg	17.6	501	43.4	1.47
RLu	17.2	478	43.0	1.38

6.6 Very Heavy Traffic

All of the experiments described to this point have dealt with relatively moderate traffic loads. The down-peak traffic profile we used contains several bursts of heavy traffic, but these only last for 5 or 10 minutes at a time. In this section, we examine what the algorithm does in sustained heavy traffic that approaches the capacity limit of the system. As discussed in section 4.1, the down-peak capacity of an elevator system is greater than its up-peak capacity. Therefore, the 100% traffic intensity level of a system is based on its up-peak capacity. For the elevator testbed in this dissertation, 100% intensity is roughly 2000 passengers per hour. In terms of pure down traffic, this amounts to an average of 18 passenger arrivals during each 5-minute interval at each of floors 2 through 10.

As discussed in section 6.10, heavier traffic tends to bring on instability in the weights of the neural networks. This can be countered by lowering the learning rate, and also by using a process of shaping, that is, gradually increasing the intensity of the traffic during training. Figure 6.2 compares the performance of the SECTOR and BASIC HUFF controllers with two parallel RL controllers. RL100 was trained on a 100% traffic intensity level, while RL125 was trained on intensity that increased from 50% to 125% over the course of training. Attempts to train on intensities greater than 125% consistently caused instabilities.

SECTOR was the most robust in very heavy traffic. By training on heavier traffic, RL125 was able to perform better on heavier traffic, but it was not as good as RL100 at 100% intensity. There seems to be a shift in the type of strategy needed at very high intensity levels, while a single strategy seems to work well over a variety of moderate intensity levels. This may be one reason why the nonstationarity of the moderately intense down-peak profile was not a major problem to the RL algorithms.

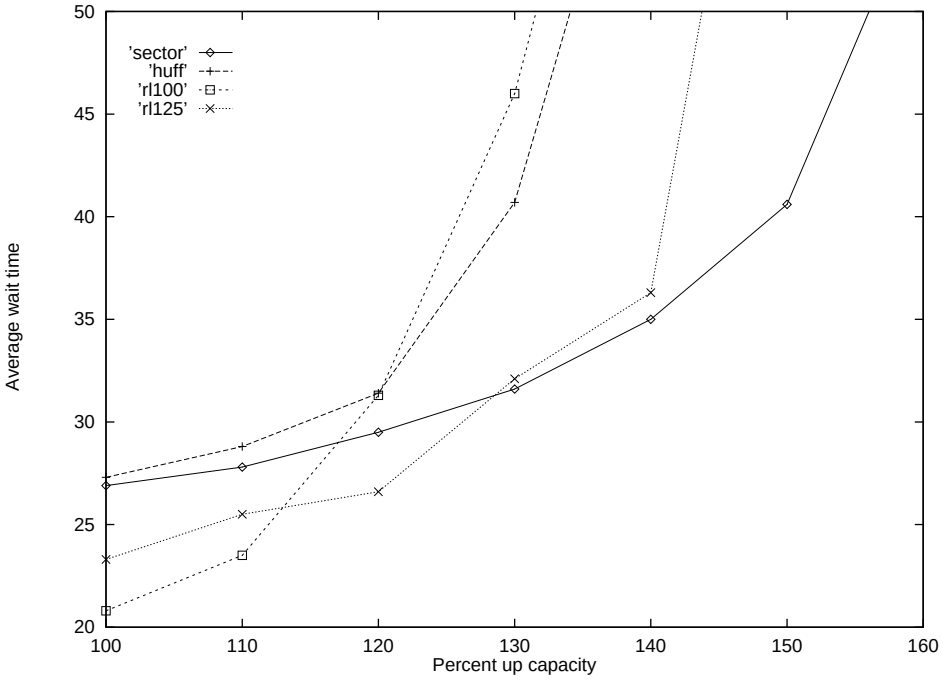

Figure 6.2 *Comparison of wait times in very heavy pure down traffic.*

If an RL controller was to be implemented on a real system, it might make sense to train different networks for different traffic intensities and then switch between them. The magnitude of the penalties being incurred by each network could be used to help determine the appropriate times to switch to other networks. In cases of extremely intense traffic, control could be passed to a sector–based algorithm.

6.7 Levels of Incomplete State Information

If parallel or decentralized RL were to be implemented in a real elevator system, there would be no problem providing whatever state information was available to all of the agents. However, in a truly decentralized control situation, this might not be possible. This section looks at how performance degrades as the agents receive less state information.

In these experiments, the amount of information available to the agents was varied along two dimensions: information about the hall call buttons, and information about the location, direction, and status of the other cars.

The input representations for the hall call buttons were: REAL, consisting of 18 input units, where two units encode information about each of the nine down hall buttons. A real-valued unit encodes the elapsed time if the button has been pushed and a binary unit is on if the button has not been pushed; BINARY, consisting of 9 binary input units corresponding to the nine down hall buttons; QUANTITY, consisting of two input units measuring the number of hall calls above and below the current decision-making car, and NONE, with no input units conveying information about the hall buttons.

The input representations for the configuration of the other cars were: FOOTPRINTS, consisting of 10 input units, where each unit represents one of the 10 floors where the other cars may be located. Each car has a "footprint" that depends on its direction and speed. For example, a stopped car causes activation only on the unit corresponding to its current floor, but a moving car causes activation on several units corresponding to the floors it is approaching, with the highest activations on the closest floors. Activations caused by the various cars are additive; QUANTITY, consisting of 4 input units that represent the number of upward and downward moving cars above and below the decision-making car; and NONE, consisting of no input units conveying information about the hall buttons.

All of the networks also possessed a bias unit that was always activated, 20 hidden units, and 2 output units (for the STOP and CONTINUE actions). All used the decentralized RL algorithm, trained for 12000 hours of simulated elevator time using the down–peak profile and omniscient reinforcements. The temperature T was annealed according to the schedule: $T = 2.0 * (.9995)^h$, where h is the hours

of training. The learning rate parameter was decreased according to the schedule: $LR = 0.01 * (.99975)^h$.

The results shown in table 6.11 are measured in terms of the average squared passenger waiting times over 30 hours of simulated elevator time. They should be considered to be fairly noisy because they were not averaged over multiple training runs. Nevertheless, they show some interesting trends.

Table 6.11 *Average squared wait times with various levels of incomplete state information.*

		Location of Other Cars		
		Footprints	Quantity	None
Hall Buttons	Real	370	428	474
	Binary	471	409	553
	Quantity	449	390	530
	None	1161	778	827

Clearly, information about the hall calls was more important than information about the configuration of the other cars. In fact, performance was still remarkably good even without any information about the other cars. (Technically speaking, some information was always available about the other cars because of the constraint that prevents a car from stopping to pick up passengers at a floor where another car has already stopped. No doubt this constraint helped performance considerably.)

When the hall call information was completely missing, the network weights had an increased tendency to become unstable or grow without bound and so the learning rate parameter had to be lowered in some cases. For a further discussion of network instability, see section 6.10.

The way that information was presented was important. For example, being supplied with the number of hall calls above and below the decision-making car was more useful to the networks than the potentially more informative binary button in-

formation. It also appears that information along one dimension is helpful in utilizing information along the other dimension. For example, the FOOTPRINTS representation made performance much worse than no car information in the absence of any hall call information. The only time FOOTPRINTS outperformed the other representations was with the maximum amount of hall call information.

Overall, the performance was quite good except in the complete absence of hall call information (which is a significant handicap indeed), and it could be improved further by slower annealing. It seems reasonable to say that the algorithm degrades gracefully in the presence of incomplete state information in this problem.

In a final experiment, two binary features were added to the REAL/FOOTPRINTS input representation. They were activated when the decision-making car was at the highest floor with a waiting passenger, and the floor with the longest waiting passenger, respectively. With the addition of these features, the average squared wait time decreased from 370 to 359, so they appear to have some value.

6.8 Separate Action Networks

The results presented in section 6.1 were obtained using networks with two output units, one for the *stop* action and one for the *continue* action. It is also possible to implement the parallel and distributed architectures using twice as many single–output networks, with a separate network for each action. RLds and RLps refer to the distributed and parallel RL architectures with separate action networks. Tables 6.12, 6.13, and 6.14 show a comparison of the performance of the combined versus separate action networks, trained under the same conditions. In spite of experiments [25] that seem to suggest that combined networks should be superior, in this particular problem that does not appear to be the case.

Table 6.12 *Combined versus separate action network results for down–peak profile with down traffic only.*

Algorithm	AvgWait	SquaredWait	SystemTime	Percent>60 secs
RLp	14.8	320	41.8	0.09
RLd	14.7	313	41.7	0.07
RLds	14.9	326	41.8	0.07
RLps	14.4	296	42.1	0.05

Table 6.13 *Combined versus separate action network results for down–peak profile with up and down traffic.*

Algorithm	AvgWait	SquaredWait	SystemTime	Percent>60 secs
RLp	16.9	476	42.7	1.53
RLd	16.9	468	42.7	1.40
RLds	16.9	465	42.5	1.29
RLps	16.8	454	42.8	1.20

Table 6.14 *Combined versus separate action network results for down–peak profile with twice as much up traffic.*

Algorithm	AvgWait	SquaredWait	SystemTime	Percent>60 secs
RLd	18.8	593	45.4	2.40
RLp	18.6	585	45.7	2.49
RLds	19.0	614	45.8	2.71
RLps	18.7	583	45.6	2.29

6.9 Practical Issues

One of the biggest difficulties in applying RL to the elevator control problem was finding the correct temperature and learning rate parameters. It was very helpful to start with a scaled down version consisting of 1 car and 4 floors and a lookup table for the Q-values. This made it easier to determine rough values for the temperature and learning rate schedules.

The addition of the constraints described in section 4.3.3 also helped to improve performance. The importance of focusing the experience of the learner into the most appropriate areas of the state space cannot be overstressed. Training with trajectories of the system is an important start, but adding reasonable constraints also helps. Further evidence supporting the importance of focusing is that given a choice between training on heavier or lighter traffic than one expects to face during testing, it is better to train on the heavier traffic. This type of training gives the system more experience with states where the queue lengths are long and thus where making the correct decision is crucial.

6.10 Instability

The weights of the neural networks can become unstable, their magnitude increasing without bound. Two particular situations seem to lead to instability. The first occurs when the learning algorithm makes updates that are too large. This can happen when the learning rate is too large, or when the network inputs are too large (which can happen in very heavy traffic situations), or both. The second occurs when the network weights have just been initialized to random values, producing excessively inconsistent Q-values. For example, while a learning rate of 10^{-2} is suitable for training a random initial network on moderate traffic (700 passengers/hour), it very consistently brings on instability in heavy traffic (1900 passengers/hour). However,

a learning rate of 10^{-3} keeps the network stable even in heavy traffic. If we train the network this way for several hundred hours of elevator time, leading to weights that represent a more consistent set of Q-values, then the learning rate can be safely raised back up to 10^{-2} without causing instability.

6.11 Linear Networks

One may ask whether nonlinear function approximators such as feedforward sigmoidal networks are necessary for good performance in this elevator control problem. A test was run using a linear network trained with the delta rule. The linear network had a much greater tendency to be unstable. In order to keep the weights from blowing up, the learning rate had to be lowered by several orders of magnitude, from 10^{-3} to 10^{-6}. After some initial improvement, the linear network was unable to further reduce the average TD error, resulting in extremely poor performance. This failure of linear networks lends support to the contention that elevator control is a difficult problem.

6.12 Summary

As detailed in this chapter, both the parallel and distributed multi-agent RL architectures were able to outperform all of the elevator algorithms they were tested against. The two architectures learned very similar policies. Gradual annealing appeared to be a crucial factor in their success. Training was accomplished effectively using both omniscient and online reinforcements. The algorithms were robust, easily generalizing to new situations such as added up traffic. They were also able to tailor themselves to a variety of situations, such as buildings with unbalanced floor populations or very intense traffic. Finally, they degraded gracefully in the face of increasing levels of incomplete state information. Although the networks became unstable under certain circumstances, techniques were discussed that prevented the

instabilities in practice. Taken together, these results demonstrate that multi–agent RL algorithms are very powerful techniques for addressing very large scale stochastic dynamic optimization problems.

CHAPTER 7

FUTURE WORK

There are many areas of research in both elevator group control and general multi-agent RL that deserve further investigation. Implementing an RL controller in a real elevator system would require training on several other traffic profiles, including up-peak and inter-floor traffic patterns. Additional actions would be needed in order to handle these traffic patterns. For example, in up-peak traffic it would be useful to have actions to specifically open and close the doors or to control the dwell time at the lobby. In inter-floor traffic, unconstrained "up" and "down" actions would be needed for the sake of flexibility. The cars should also have the ability to "park" at various floors during periods of light traffic.

It would also be useful to be able to interpret or summarize the policies the RL controllers have learned, e.g., [31]. There may or may not be a simple explanation of the strategies they have learned, but this is difficult to ascertain because the state space has such a high dimension. However, if the policies could be expressed in terms of simple rules, they would probably inspire more confidence among elevator designers than the numerical weights of artificial neural networks.

Exploring other state representations may be another way of improving performance. In particular, the addition of hand-crafted features may be able to significantly boost performance. Other state information could also be added, for example, the configuration of the car buttons. It is not clear whether the benefits of this additional information would outweigh the added complexity it would engender. Another

interesting question is whether RL agents with input representations that distinguish among the other cars would still arrive at similar policies.

We were able to obtain good elevator performance in spite of ignoring the non–stationarity of the passenger arrival patterns. However, there are several ways one might explicitly address this non–stationarity. One could train different networks for different traffic profiles, and use a gating architecture [53] to select the appropriate network. Another possibility would be to provide the networks with additional information about the traffic context or the time of day.

The custom in Japanese elevator systems is to signal the car assignment as soon as a hall call has been registered. This requirement could be satisfied by taking the architecture for immediate call assignments proposed by Markon et al [70] (see section 4.2.5), and training it with a sequential RL algorithm. The outputs of the network could be trained to learn Q-values. It would be fascinating to compare the performance of such a centralized immediate call assignment architecture with the architecture we have described.

There are also a number of more general RL issues that need to be addressed. It would be useful to find criteria for determining which application areas are especially well suited to RL methods. In addition, a deeper understanding is needed of RL with compact function approximators in order to find ways of minimizing problems with instability. Another area of interest is how to incorporate prior knowledge in RL, for example, by bootstrapping off of existing controllers.

In terms of multi-agent RL, it would be interesting to try something other than a uniform annealing schedule for all the agents. For example, a coordinated exploration strategy or round-robin type of annealing might be a way of reducing the noise generated by the other agents. However, such a coordinated exploration strategy may have a greater tendency to become stuck in sub-optimal policies.

Theoretical results for sequential multi-agent RL are also needed to supplement the results for non-sequential multi-agent RL described in section 3.1.2. Another area that needs further study is RL architectures where reinforcements are tailored to individual agents, possibly by using a hierarchy or some other advanced organizational structure. Such local reinforcement architectures have the potential to greatly increase the speed of learning, but they will require much more knowledge on the part of whatever is producing the reinforcement signals [8]. Finally, it is important to find effective methods of allowing the possibility of explicit communication among the agents.

CHAPTER 8

CONCLUSIONS

Multi-agent control systems are often required because of spatial or geographic distribution, or in situations where centralized information is not available or is not practical. But even when a distributed approach is not required, multiple agents may still provide an excellent way of scaling up to approximate solutions for very large problems by streamlining the search through the space of possible policies.

Researchers in distributed artificial intelligence and control theory have generally focused on top-down approaches to building distributed systems, creating them from a global vantage point. Even though the agents in such systems are restricted to their own local point of view, they are designed in a centralized way, with access to complete knowledge about the problem. One drawback to this top-down approach is the extraordinary complexity of designing such agents, since it is extremely difficult to anticipate all possible interactions and contingencies ahead of time in complex systems.

Artificial life and behavior-based AI researchers have generally taken the opposite approach, combining large numbers of relatively unsophisticated agents in a bottom-up manner and seeing what emerges when they are put together into a group. However, there is little evidence as yet that such bottom-up designs really provide a simple way of achieving complex specific goals.

Multi-agent RL combines the advantages of both approaches. It achieves the simplicity of a bottom-up approach by allowing the use of relatively unsophisticated agents that learn on the basis of their own experiences. At the same time, RL

agents adapt to a top-down global reinforcement signal, which guides their behavior toward the achievement of complex specific goals. As a result, very robust systems for complex problems can be created with a minimum of human effort.

RL also combines the heuristic, satisficing nature of AI with the principled nature of control theory, by approximating DP in an incremental manner. RL algorithms can be trained using actual or simulated experiences, allowing them to focus computation on the areas of state space that are actually visited during control, making them computationally tractable on very large problems. If each of the members of a team of agents employs an RL algorithm, a new *collective* algorithm emerges for the group as a whole. This type of collective algorithm allows control policies to be learned in a decentralized way. Even though RL agents in a team face added stochasticity and non-stationarity due to the changing stochastic policies of the other agents on the team, they display an exceptional ability to cooperate with one another in maximizing their rewards.

In order to demonstrate the power of multi-agent RL, we focused on the problem of elevator group supervisory control. The elevator domain poses a combination of challenges not seen in most RL research to date. Elevator systems operate in continuous state spaces and in continuous time as discrete event dynamic systems. Their state is not fully observable and they are non-stationary due to changing passenger arrival rates. We used a team of RL agents, each of which was responsible for controlling one elevator car. The team received a global reinforcement signal which appeared noisy to each agent due to the effects of the actions of the other agents, the random nature of the arrivals, and the incomplete observation of the state. In spite of these complications, we obtained results that in simulation surpass the best of the heuristic elevator control algorithms of which we are aware. Performance was also very robust in the face of increased levels of incomplete state information. These results

demonstrate the power of multi-agent RL on a very large scale stochastic dynamic optimization problem of practical utility.

BIBLIOGRAPHY

[1] Aicardi, M., Davoli, F., and Minciardi, R. Decentralized optimal control of Markov chains with a common past information set. *IEEE Transactions on Automatic Control*, 32:1028–1031, 1987.

[2] Astrom, K. J. Optimal control of Markov processes with incomplete state information. *Journal of Mathematical Analysis and Applications*, 10:174–205, 1965.

[3] Aumann, R. J. Survey of repeated games. In *Essays in Game Theory and Mathematical Economics*. Bibliographisches Institut, Mannheim, Germany, 1981.

[4] Axelrod, R. M. *The Evolution of Cooperation*. Basic Books, New York, 1984.

[5] Baird, L. C. Advantage updating. Technical report, Wright-Patterson Air Force Base, 1993.

[6] Bao, G., Cassandras, C. G., Djaferis, T. E., Gandhi, A. D., and Looze, D. P. Elevator dispatchers for down peak traffic. ECE Department Technical Report, University of Massachusetts, 1994.

[7] Barto, A. G. Learning by statistical cooperation of self-interested neuron-like adaptive elements. *Human Neurobiology*, 4:229–256, 1985.

[8] Barto, A. G. From chemotaxis to cooperativity: Abstract exercises in neuronal learning strategies. In Durbin, R., Miall, C., and Mitchison, G., editors, *The Computing Neuron*. Addison-Wesley, Wokingham, England, 1989.

[9] Barto, A. G. Some learning tasks from a control perspective. COINS Technical Report 90-122, University of Massachusetts, 1990.

[10] Barto, A. G. and Anandan, P. Pattern recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15:360–375, 1985.

[11] Barto, A. G., Bradtke, S. J., and Singh, S. P. Learning to act using real-time dynamic programming. CMPSCI Technical Report 93-02, University of Massachusetts, 1993.

[12] Barto, A. G. and Jordan, M. I. Gradient following without back-propagation in layered networks. In *Proceedings of the IEEE First Annual Conference on Neural Networks*, 1987.

[13] Barto, A. G. and Sutton, R. S. *Learning By Interaction: An Introduction to Modern Reinforcement Learning.* Forthcoming.

[14] Barto, A. G., Sutton, R. S., and Anderson, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846, 1983.

[15] Barto, A. G., Sutton, R. S., and Watkins, C. J. C. H. Learning and sequential decision making. COINS Technical Report 89-95, University of Massachusetts, 1989.

[16] Benmakhlouf, S. M. and Khator, S. K. Smart lifts: Control design and performance evaluation. *Computers and Industrial Engineering*, 25:175–178, 1993.

[17] Berry, D. A. and Fristedt, B. *Bandit Problems.* Chapman and Hall, London, 1985.

[18] Bertsekas, D. P. *Dynamic Programming and Stochastic Control.* Academic Press, New York, 1976.

[19] Bertsekas, D. P. *Dynamic Programming: Deterministic and Stochastic Models.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

[20] Bertsekas, D. P. and Tsitsiklis, J. N. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[21] Bertsekas, D. P. and Tsitsiklis, J. N. *Neuro–Dynamic Programming.* Athena Scientific Press, Belmont, MA, 1996.

[22] Boyan, J. A., Moore, A. W., and Sutton, R. S. Proceedings of the workshop on value function approximation, Machine Learning Conference 1995. Technical Report CMU-CS-95-206, Carnegie Mellon University, 1995.

[23] Bradtke, S. J. Distributed adaptive optimal control of flexible structures, 1993. Unpublished manuscript.

[24] Bradtke, S. J. and Duff, M. O. Reinforcement learning methods for continuous-time Markov decision problems. In Tesauro, G., Touretzky, D., and Leen, T., editors, *Advances in Neural Information Processing Systems 7.* MIT Press, Cambridge, MA, 1995.

[25] Caruana, R. Learning many related tasks at the same time with backpropagation. In Tesauro, G., Touretzky, D., and Leen, T., editors, *Advances in Neural Information Processing Systems 7.* MIT Press, Cambridge, MA, 1995.

[26] Cassandras, C. G. *Discrete Event Systems: Modeling and Performance Analysis.* Aksen Associates, Homewood, IL, 1993.

[27] Chandrasekaran, B. Natural and social system metaphors for distributed problem solving: Introduction to the issue. *IEEE Transactions on Systems, Man, and Cybernetics*, 11:1–5, 1981.

[28] Chapman, D. and Kaelbling, L. P. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of IJCAI*, 1991.

[29] Chrisman, L. Planning for closed-loop execution using partially observable Markovian decision processes. In *AAAI Spring Symposium Series: Control of Selective Sensing*, 1992.

[30] Chrisman, L., Caruana, R., and Carriker, W. Intelligent agent design issues: Internal agent state and incomplete perception. In *AAAI Fall Symposium Series: Sensory Aspects of Robotic Intelligence*, 1991.

[31] Craven, M. W. and Shavlik, J. W. Learning symbolic rules using artificial neural networks. In *Proceedings of the Tenth International Conference on Machine Learning*, 1993.

[32] Crick, F. The recent excitement about neural networks. *Nature*, 337:129–132, 1989.

[33] Crites, R. H. and Barto, A. G. An actor/critic algorithm that is equivalent to Q-learning. In Tesauro, G., Touretzky, D., and Leen, T., editors, *Advances in Neural Information Processing Systems 7*. MIT Press, Cambridge, MA, 1995.

[34] Crites, R. H. and Barto, A. G. Forming control policies from simulation models using reinforcement learning. In *Proceedings of the Ninth Yale Workshop on Adaptive and Learning Systems*. 1996.

[35] Crites, R. H. and Barto, A. G. Improving elevator performance using reinforcement learning. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems 8*. MIT Press, Cambridge, MA, 1996.

[36] Davies, P., editor. *The American Heritage Dictionary of the English Language*. Dell Publishing, 1980.

[37] Davis, R. and Smith, R. G. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, 1983.

[38] Dawkins, R. Hierarchical organisation: A candidate principle for ethology. In Bateson, P. and Hinde, R., editors, *Growing Points in Ethology*. Cambridge University Press, 1976.

[39] Dayan, P. and Hinton, G. E. Feudal reinforcement learning. In Hanson, S. J., Cowan, J. D., and Giles, C. L., editors, *Advances in Neural Information Processing Systems 5*. Morgan Kaufmann, San Mateo, CA, 1993.

[40] Durfee, E. H. *Coordination of Distributed Problem Solvers.* Kluwer Academic Publishers, 1988.

[41] Durfee, E. H. The distributed artificial intelligence melting pot. *IEEE Transactions on Systems, Man, and Cybernetics*, 21:1301–1306, 1991.

[42] Ephrati, E. and Rosenschein, J. S. The Clarke tax as a consensus mechanism among automated agents. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 173–178, 1991.

[43] Fujino, A., Tobita, T., and Yoneda, K. An on-line tuning method for multi-objective control of elevator group. In *Proceedings of the International Conference on Industrial Electronics, Control, Instrumentation, and Automation*, pages 795–800, 1992.

[44] Guha, R. V. and Lenat, D. B. CYC: A midterm report. *AI Magazine*, 11:32–59, 1990.

[45] Gullapalli, V. *Reinforcement Learning and Its Application to Control.* PhD thesis, University of Massachusetts, 1992. COINS Technical Report 92-10.

[46] Harsanyi, J. C. Games with incomplete information played by Bayesian players, part 1. *Management Science*, 14:159–182, 1967.

[47] Ho, Y.-C. Team decision theory and information structures. *Proceedings of the IEEE*, 68:644–654, 1980.

[48] Holusha, J. Want the 99th floorΓ Getting there may be half the fun. The New York Times, Sunday, January 22, 1995.

[49] Hsu, K. and Marcus, S. Decentralized control of finite state Markov processes. *IEEE Transactions on Automatic Control*, 27:426–431, 1982.

[50] Humphrys, M. W-learning: Competition among selfish Q-learners. Computer Laboratory Technical Report 362, University of Cambridge, 1995.

[51] Imasaki, N., Kiji, J., and Endo, T. A fuzzy neural network and its application to elevator group control. In Terano, T., Sugeno, M., Mukaidono, M., and Shigemasu, K., editors, *Fuzzy Engineering Toward Human Friendly Systems.* IOS Press, Amsterdam, 1992.

[52] Jaakkola, T., Singh, S. P., and Jordan, M. I. Reinforcement learning algorithm for partially observable Markov decision problems. In Tesauro, G., Touretzky, D., and Leen, T., editors, *Advances in Neural Information Processing Systems 7.* MIT Press, Cambridge, MA, 1995.

[53] Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. Adaptive mixtures of local experts. *Neural Computation*, 3:79–87, 1991.

[54] Jagannathan, V. and Dodhiawala, R. Distributed artificial intelligence: An annotated bibliography. *ACM SIGART Newsletter*, 95:44–56, 1986.

[55] Jennings, N. R. Controlling cooperative problem solving in industrial multiagent systems using joint intentions. *Artificial Intelligence*, 75:195–240, 1995.

[56] Lakshmivarahan, S. and Narendra, K. S. Learning algorithms for two-person zero-sum stochastic games with incomplete information. *Mathematics of Operations Research*, 6:379–386, 1981.

[57] Lakshmivarahan, S. and Narendra, K. S. Learning algorithms for two-person zero-sum stochastic games with incomplete information: A unified approach. *SIAM Journal on Control and Optimization*, 20:541–552, 1982.

[58] Lesser, V., editor. *Proceedings of the First International Conference on Multi-Agent Systems*. AAAI Press, Menlo Park, CA, 1995.

[59] Lesser, V. and Corkill, D. Distributed problem solving. In Shapiro, S. and Eckroth, D., editors, *Encyclopedia of Artificial Intelligence*. Wiley, New York, 1987.

[60] Levy, D., Yadin, M., and Alexandrovitz, A. Optimal control of elevators. *International Journal of Systems Science*, 8:301–320, 1977.

[61] Lewis, J. *A Dynamic Load Balancing Approach to the Control of Multiserver Polling Systems with Applications to Elevator System Dispatching*. Ece department, University of Massachusetts, September 1991.

[62] Lin, L.-J. and Mitchell, T. M. Memory approaches to reinforcement learning in non-Markovian domains. Technical Report CMU-CS-92-138, Carnegie Mellon University, 1992.

[63] Littman, M. and Boyan, J. A distributed reinforcement learning scheme for network routing. Technical Report CMU-CS-93-165, Carnegie Mellon University, 1993.

[64] Littman, M. L. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning*. Morgan Kaufmann, 1994.

[65] Littman, M. L. *Algorithms for Sequential Decision Making*. PhD thesis, Brown University, March 1996.

[66] Lovejoy, W. S. A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, 28:47–66, 1991.

[67] Luce, R. D. and Raiffa, H. *Games and Decisions*. Wiley, New York, 1957.

[68] Maes, P. and Brooks, R. A. Learning to coordinate behaviors. In *AAAI-90 Proceedings*, 1990.

[69] Markey, K. L. Efficient learning of multiple degree-of-freedom control problems with quasi-independent Q-agents. In Mozer, M. C., Smolensky, P., Touretzky, D. S., Elman, J. L., and Weigend, A. S., editors, *Proceedings of the 1993 Connectionist Models Summer School*. Erlbaum Associates, Hillsdale, NJ, 1994.

[70] Markon, S., Kita, H., and Nishikawa, Y. Adaptive optimal elevator group control by use of neural networks. *Transactions of the Institute of Systems, Control, and Information Engineers*, 7:487–497, 1994.

[71] Marschak, J. and Radner, R. *Economic Theory of Teams*. Yale University Press, New Haven, 1972.

[72] Mataric, M. J. Issues and approaches in the design of collective autonomous agents. *Robotics and Autonomous Systems*, 16:321–331, 1995.

[73] Mataric, M. J. Learning in multi-robot systems. In *Proceedings of the IJCAI-95 Workshop on Adaptation and Learning in Multiagent Systems*, 1995.

[74] McCallum, R. A. Overcoming incomplete perception with utile distinction memory. In *Proceedings of the Tenth International Conference on Machine Learning*, 1993.

[75] McCallum, R. A. Instance-based state identification for reinforcement learning. In Tesauro, G., Touretzky, D., and Leen, T., editors, *Advances in Neural Information Processing Systems 7*. MIT Press, Cambridge, MA, 1995.

[76] Monahan, G. A survey of partially observable Markov decision processes. *Management Science*, 28:1–16, 1982.

[77] Moore, A. W. and Atkeson, C. G. Memory-based reinforcement learning: Efficient computation with prioritized sweeping. In Hanson, S. J., Cowan, J. D., and Giles, C. L., editors, *Advances in Neural Information Processing Systems 5*. Morgan Kaufmann, San Mateo, CA, 1993.

[78] Nagendra Prasad, M. V. and Lesser, V. R. Learning situation-specific coordination in generalized partial global planning. In *AAAI Spring Symposium on Adaptation, Co-Evolution, and Learning in Multiagent Systems*, 1996.

[79] Nagendra Prasad, M. V., Lesser, V. R., and Lander, S. E. Learning experiments in a heterogeneous multi-agent system. In *Proceedings of the IJCAI-95 Workshop on Adaptation and Learning in Multiagent Systems*, 1995.

[80] Narendra, K. S. and Thathachar, M. A. L. *Learning Automata: An Introduction*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[81] Ovaska, S. J. Electronics and information technology in high-range elevator systems. *Mechatronics*, 2:89–99, 1992.

[82] Pang, G. K. H. Elevator scheduling system using blackboard architecture. *IEE Proceedings-D*, 138:337–346, 1991.

[83] Papadimitriou, C. H. and Tsitsiklis, J. N. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12:441–450, 1987.

[84] Peng, J. and Williams, R. J. Efficient search control in Dyna. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*. MIT Press, 1993.

[85] Pepyne, D. L. Application of Q-learning to elevator dispatching. Ece department, University of Massachusetts, February 1995.

[86] Pepyne, D. L. and Cassandras, C. G. Concurrent estimation for on-line adaptive control of elevator systems during uppeak traffic. In Preparation.

[87] Pepyne, D. L. and Cassandras, C. G. Optimal dispatching control for elevator systems during uppeak traffic. Submitted to CDC-96.

[88] Phansalkar, V. V. and Thathachar, M. A. L. Local and global optimization algorithms for generalized learning automata. *Neural Computation*, 7:950–973, 1995.

[89] Rabiner, L. R. and Juang, B. H. An introduction to hidden Markov models. *IEEE ASSP Magazine*, 3:4–16, 1986.

[90] Ring, M. Learning sequential tasks by incrementally adding higher orders. In Hanson, S. J., Cowan, J. D., and Giles, C. L., editors, *Advances in Neural Information Processing Systems 5*. Morgan Kaufmann, San Mateo, CA, 1993.

[91] Rumelhart, D. E., McClelland, J. L., and the PDP Research Group. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, Cambridge, MA, 1986.

[92] Rummery, G. A. and Niranjan, M. On-line Q-learning using connectionist systems. CUED/F-INFENG/TR 166, Cambridge University Engineering Department, 1994.

[93] Sabourian, H. Repeated games: A survey. In Hahn, F., editor, *The Economics of Missing Markets, Information, and Games*. Clarendon Press, Oxford, 1989.

[94] Sakai, Y. and Kurosawa, K. Development of elevator supervisory group control system with artificial intelligence. *Hitachi Review*, 33:25–30, 1984.

[95] Samuel, A. L. Some studies in machine learning using the game of checkers. In Feigenbaum, E. and Feldman, J., editors, *Computers and Thought*. McGraw-Hill, New York, 1963.

[96] Sandholm, T. W. and Crites, R. H. Multiagent reinforcement learning in the iterated prisoner's dilemma. *Biosystems*, 37:147–166, 1996.

[97] Sato, M., Abe, K., and Takeda, H. Learning control of finite Markov chains with explicit tradeoff between estimation and control. *IEEE Transactions on Systems, Man, and Cybernetics*, 18:677–684, 1988.

[98] Sen, S., Sekaran, M., and Hale, J. Learning to coordinate without sharing information. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 426–431, 1994.

[99] Shoham, Y. and Tennenholtz, M. Emergent conventions in multi-agent systems: Initial experimental results and observations. In *Proceedings of KR-92.*

[100] Shoham, Y. and Tennenholtz, M. Co-learning and the evolution of coordinated multi-agent activity. 1993.

[101] Siikonen, M.-L. Elevator traffic simulation. *Simulation*, 61:257–267, 1993.

[102] Strakosch, G. R. *Vertical Transportation: Elevators and Escalators.* Wiley and Sons, New York, 1983.

[103] Sugawara, T. and Lesser, V. Learning coordination plans in distributed problem-solving environments. In *Twelfth International Workshop on Distributed Artificial Intelligence*, 1993.

[104] Sutton, R. S. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[105] Sutton, R. S. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, 1990.

[106] Tan, M. Learning a cost-sensitive internal representation for reinforcement learning. In *Proceedings of the Eighth International Workshop on Machine Learning (ML91)*, 1991.

[107] Tan, M. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the Tenth International Conference on Machine Learning*, 1993.

[108] Tesauro, G. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

[109] Tesauro, G. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

[110] Tesauro, G. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38:58–68, 1995.

[111] Tham, C. and Prager, R. A modular Q-learning architecture for manipulator task decomposition. In *Proceedings of the Eleventh International Conference on Machine Learning.* Morgan Kaufmann, 1994.

[112] Tobita, T., Fujino, A., Inaba, H., Yoneda, K., and Ueshima, T. An elevator characterized group supervisory control system. In *Proceedings of IECON*, pages 1972–1976, 1991.

[113] Tsetlin, M. L. *Automaton Theory and Modeling of Biological Systems*. Academic Press, New York, NY, 1973.

[114] Tsitsiklis, J. N. and Athans, M. On the complexity of decentralized decision making and detection problems. *IEEE Transactions on Automatic Control*, 30:440–446, 1985.

[115] Tsitsiklis, J. N. and Van Roy, B. An analysis of temporal–difference learning with function approximation. Technical Report LIDS-P-2322, MIT, 1996.

[116] Ujihara, H. and Amano, M. The latest elevator group-control system. *Mitsubishi Electric Advance*, 67:10–12, 1994.

[117] Ujihara, H. and Tsuji, S. The revolutionary AI-2100 elevator-group control system and the new intelligent option series. *Mitsubishi Electric Advance*, 45:5–8, 1988.

[118] Varaiya, P. and Walrand, J. On delayed sharing patterns. *IEEE Transactions on Automatic Control*, 23:443–445, 1978.

[119] Watkins, C. J. C. H. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.

[120] Weiss, G. Some studies in distributed machine learning and organizational design. FKI 189-94, Institut fur Informatik, Technische Universitat Muenchen, 1994.

[121] Weiss, G. Adaptation and learning in multi-agent systems: Some remarks and a bibliography. In *Adaptation and Learning in Multi-Agent Systems*, Berlin, 1996. Springer Verlag. Lecture Notes in Artificial Intelligence, Volume 1042.

[122] Weiss, G. and Sen, S. *Adaptation and Learning in Multi-Agent Systems*. Springer Verlag, Berlin, 1996. Lecture Notes in Artificial Intelligence, Volume 1042.

[123] Whitehead, S. D. and Ballard, D. H. Active perception and reinforcement learning. *Neural Computation*, 2:409–419, 1990.

[124] Williams, R. J. Toward a theory of reinforcement-learning connectionist systems. Technical Report NU-CCS-88-3, Northeastern University, 1988.

[125] Witsenhausen, H. S. A counterexample in stochastic optimum control. *SIAM Journal of Control*, 6:138–147, 1968.

[126] Witsenhausen, H. S. Separation of estimation and control for discrete time systems. *Proceedings of the IEEE*, 59:1557–1566, 1971.