Machine Learning, 33, 235–262 (1998)
© 1998 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Elevator Group Control Using Multiple Reinforcement Learning Agents

ROBERT H. CRITES rcrites@unica-usa.com
Unica Technologies, Inc., 55 Old Bedford Road, Lincoln, MA 01773

ANDREW G. BARTO barto@cs.umass.edu
University of Massachusetts, Department of Computer Science, Amherst, MA 01003

Editors: Michael Huhns and Gerhard Weiss

Abstract. Recent algorithmic and theoretical advances in reinforcement learning (RL) have attracted widespread interest. RL algorithms have appeared that approximate dynamic programming on an incremental basis. They can be trained on the basis of real or simulated experiences, focusing their computation on areas of state space that are actually visited during control, making them computationally tractable on very large problems. 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 paper we demonstrate that such collective RL algorithms can be powerful heuristic methods for addressing large-scale control problems.

Elevator group control serves as our testbed. It is a difficult domain posing a combination of challenges not seen in most multi-agent learning research to date. We use a team of RL agents, each of which is responsible for controlling one elevator car. The team receives a global reward 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.

Keywords: Reinforcement learning, multiple agents, teams, elevator group control, discrete event dynamic systems

1. Introduction

Interest in developing capable learning systems is increasing within the multi-agent and AI research communities (e.g., Weiss & Sen, 1996). 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 the *designers* of such systems have often faced 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.

Reinforcement learning (RL) (Sutton & Barto, 1998; Bertsekas & Tsitsiklis, 1996) applies naturally to the case of autonomous agents, which receive sensations as inputs and take actions that affect their environments in order to achieve their own goals. 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. This framework is appealing from a biological point of view, since an animal has built-in preferences but does not always have a teacher to tell it exactly what action it should take in every situation.

If the members of a group of agents each employ an RL algorithm, the resulting 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. Although it may be possible to synthesize a system whose goals can be achieved by agents with *conflicting* objectives, this paper focuses on *teams* of agents that share *identical* objectives corresponding directly to the goals of the system as a whole.

To demonstrate the power of multi-agent RL, we focus on the difficult problem of elevator group supervisory control. 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. We use a team of RL agents, each of which is responsible for controlling one elevator car. Each agent uses artificial neural networks to store its action-value estimates. We compare a *parallel* architecture, in which the agents share the same networks, with a *decentralized* architecture, in which the agents have their own independent networks. In either case, the team receives a global reward signal that is noisy from the perspective of each agent due in part to the effects of the actions of the other agents. Despite these difficulties, our system outperforms all of the heuristic elevator control algorithms known to us. We also analyze the policies learned by the agents, and show that learning is relatively robust even in the face of increasingly incomplete state information. These results suggest that approaches to decentralized control using multi-agent RL have considerable promise.

In the following sections, we give some additional background on RL, introduce the elevator domain, describe in more detail the multi-agent RL algorithm and network architecture we used, present and discuss our results, and finally draw some conclusions. For further details on all these topics, see Crites (1996).

2. Reinforcement Learning

Machine 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 algorithms are useful in a wide variety of problems involving pattern classification and function approximation. However, there are many problems in which training examples are costly or even impossible to obtain. RL can be useful in these more difficult problems, where training information comes from a "critic" providing a scalar evaluation of the output that was selected rather than specifying the best output or a direction to change the output. In RL, one faces all the difficulties of supervised learning combined with the additional difficulty of exploration, that is, of determining the best output for any given input.

It is useful to distinguish between two types of RL tasks. In non-sequential tasks, agents must learn mappings from situations to actions that maximize the expected immediate reward. In sequential tasks, agents must learn mappings from situations to actions that maximize the expected long-term rewards. Sequential tasks are generally more difficult because an agent's actions can influence the situations it must face in the future and thus the rewards that are available in the future. In these tasks, agents interact with their environments over extended periods of time, and they need to evaluate their decisions on the basis of their long-term consequences.

From the perspective of control theory, RL algorithms are ways of finding approximate solutions to stochastic optimal control problems. The agent is a controller, and the environment is a system to be controlled. The objective is to maximize some performance measure over time. Given knowledge of the state transition probabilities and reward structure of the environment, these problems can be solved in principle using dynamic programming (DP) algorithms. However, although DP requires time polynomial in the number of states, in many problems of interest there are so many states that the amount of time required makes it infeasible to obtain an exact solution.

Some recent RL algorithms have been designed to perform DP in an incremental manner. Unlike traditional DP, these algorithms do not require *a priori* knowledge of the state transition probabilities and reward structure of the environment and can be used to improve performance on-line while the agent and environment interact. This on-line learning focuses computation onto the areas of state space that are actually visited during control. Thus, when combined with suitable function approximation methods, RL algorithms can provide computationally tractable ways to approximate the solutions of very large-scale stochastic optimal control problems (Barto et al., 1995; Sutton & Barto, 1998; Bertsekas & Tsitsiklis, 1996).

The same focusing phenomenon can also be achieved with *simulated* on-line learning. One can often construct a simulation model without explicitly obtaining the state transition probabilities and reward structure of the environment (Sutton & Barto, 1998; Crites & Barto, 1996). (For an example of such a simulation model, see Section 3.3.) 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 accelerating the learning 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 learning. A successful example of simulated on-line RL is found in Tesauro's TD-Gammon system (1992, 1994, 1995), which learned to play strong master-level backgammon.

Research on multi-agent RL dates back at least to the work of the Russian mathematician Tsetlin (1973) and others working in the field of learning automata (reviewed by Narendra & Thathachar, 1989). A number of theoretical results have been obtained for non-sequential RL tasks. Certain types of learning automata converge to an equilibrium point in zero-sum and non-zero-sum repeated games. For teams, an equilibrium point is a local maximum (an element of the game matrix that is the maximum of both its row and its column). However, in more general non-zero-sum games, equilibrium points often provide poor rewards for all players. A good example of this is the Prisoners' Dilemma, where the only equilibrium point produces the lowest total reward (Axelrod, 1984).

Starting in approximately 1993, a number of researchers began to investigate sequential RL algorithms in multi-agent contexts. Although much of the work has been illustrated only in very simple abstract problems, several interesting applications have appeared that have pointed to the promise of sequential multi-agent RL. Bradtke (1993) described 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 used 10 independent adaptive controllers distributed along the beam. Each controller attempted to minimize its own local costs and observed only its own local portion of the state information. Dayan & Hinton (1993) proposed a managerial hierarchy they called *Feudal RL*. In this 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 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. Tan (1993) reported on some simple hunter-prey experiments with multi-agent RL. His focus was on the sharing of sensory information, policies, and experience among the agents. Shoham & Tennenholtz (1993) investigated the social behavior that can emerge from agents with simple learning rules. They focused on two simple n-k-g iterative games, where n agents meet k at a time (randomly) to play game g. Littman & Boyan (1993) described a distributed RL 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. Markey (1994) applied parallel Q-learning to the problem of controlling a vocal tract model with 10 degrees of freedom. He discussed two architectures equivalent to the distributed and parallel architectures we describe in Section 4.4. Each agent controlled one degree of freedom in the action space and distinguished Q-values based only on its own action selections.

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 (1963) checkers-playing program. A more recent example is Tesauro's TD-Gammon program (1992, 1994, 1995). These types of programs often learn using self-play, and they can generally be viewed as single agents. Littman (1994, 1996) provided 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. Little work has been done on multi-agent RL in more general non-zero-sum games. Sandholm & Crites (1996) studied the behavior of multi-agent RL in the context of the iterated prisoners' dilemma. They showed that Q-learning agents are able to learn the optimal strategy against the fixed opponent Tit-for-Tat. They also investigated the behavior that results when two Q-learning agents face each other.

3. Elevator Group Control

This section introduces the problem of elevator group control, which serves as our testbed for multi-agent RL. 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. An optimal policy for elevator group control is not known, so we use existing control algorithms as standards for comparison. The elevator domain provides an opportunity to compare parallel and distributed control architectures in which each agent controls one elevator car, and to monitor the amount of degradation that occurs as the agents face decreasing levels of state information.

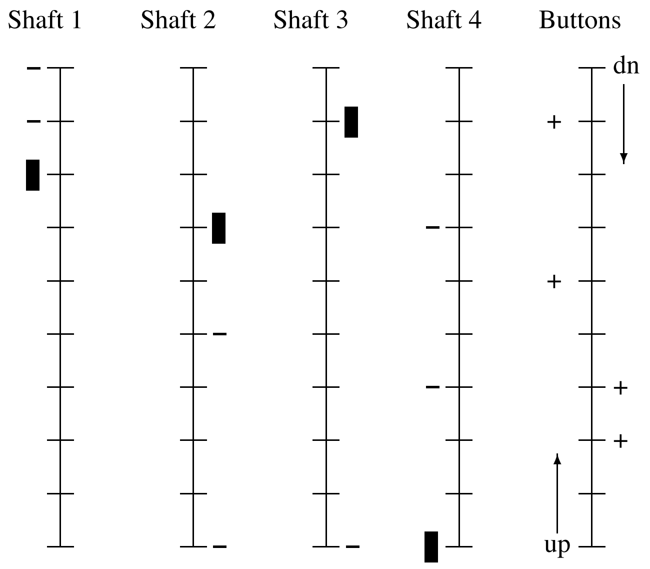

Figure 1. Elevator system schematic diagram.

Figure 1 is a schematic diagram of an elevator system (Lewis, 1991). The elevator 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.

Section 3.1 considers the nature of different passenger arrival patterns and their implications. Section 3.2 reviews a variety of elevator control strategies from the literature. Section 3.3 describes the particular simulated elevator system that will be the focus in the remainder of this paper.

3.1. 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 (1993) 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 capacity generally will be as well. 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 (Strakosch, 1983; Siikonen, 1993) 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 *interval* $I = \frac{\tau}{L}$ represents the average amount of time between car arrivals to the lobby, where L is the number of cars. Assuming that the cars are evenly spaced, the average waiting time is one half the interval. In reality, the average wait is somewhat longer.

The only control decisions in pure up 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 (1996) show that the optimal policy for handling pure up traffic is a threshold-based policy that closes the doors after an optimal number of passengers have entered the car. The optimal threshold depends on the traffic intensity, and it may also be affected by the number of car calls already registered and by the state of the other cars. Of course, up traffic is seldom completely pure. Some method must be used for assigning any down hall calls.

More general two-way traffic comes in two varieties. In two-way *lobby* traffic, up-moving passengers arrive at the lobby and down-moving passengers depart from 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 was chosen as a testbed for our research. Before describing the testbed in detail, we review various elevator control strategies from the literature.

3.2. Elevator Control Strategies

The oldest relay-based automatic controllers used the principle of *collective control* (Strakosch, 1983; Siikonen, 1993), 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.

The approaches to elevator control discussed in the literature generally fall into the following categories, often falling into 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 (Ovaska, 1992).

3.2.1. Zoning Approaches In a zoning approach (Strakosch, 1983), 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. While this approach is quite robust in heavy traffic, it gives up a significant amount of flexibility. Sakai & Kurosawa (1984) described 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.

3.2.2. Search-Based Approaches Another control strategy is to search through the space of possible car assignments, selecting the one that optimizes some criterion such as the average waiting time. *Greedy* search strategies perform immediate call assignment, that is, they assign hall calls to cars when they are first registered and never reconsider those assignments. Non-greedy algorithms 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 (Siikonen, 1993), 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 signaling requires the use of a greedy algorithm.

Tobita et al. (1991) described a system in which car assignment occurs when a hall button is pressed. The car assigned is the one that minimizes a weighted sum of predicted wait time, travel time, and number of riders. A fuzzy rule-based system picks the coefficients and estimating functions. Simulations are used to verify their effectiveness.

Receding horizon controllers are examples of 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) (Bao et al., 1994). FIM attempts to minimize squared waiting times, and ESA attempts to minimize the length of the current busy period.

3.2.3. 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 (1988) described a system that uses expert-system and fuzzy-logic technologies. Claiming that experts in group-supervisory control have the experience and knowledge necessary to shorten waiting times under various traffic conditions, they admitted 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 (1994) described the latest changes to this 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.

3.2.4. 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 (Bao et al., 1994). 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 (Lewis, 1991). DLB is a non-greedy algorithm because it reassigns sectors after every event.

3.2.5. Adaptive and Learning Approaches Imasaki et al. (1991) described a system that uses 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 did not explain what control algorithm is actually used, what its parameters are, or how the network is trained.

A system described by Tobita et al. (1991) and Fujino et al. (1992) uses a greedy 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 on-line. 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. (1977) described a system that uses dynamic programming (DP) off-line to minimize the expected time needed for completion of the current busy period. No discount factor is used, since it is assumed that the values are finite. The trouble with using DP in this way is that the state space is very large, requiring drastic simplification. Levy et al. used several methods to keep the size of the state space manageable: they considered a building with only 2 cars and 8 floors, where the number of buttons that could be on simultaneously was restricted, the states of the buttons were restricted to binary values (i.e., elapsed times were discarded), and the cars had unlimited capacity. 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.

Markon et al. (1994) 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 RL. The single-stage reward is determined by measuring the system's performance on a traffic sample. The input representation uses 25 units for each car, and the output representation uses one unit for each car. Hall calls are allocated to the car corresponding to the output unit with the highest activation. Markon et al. also described 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.

This 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 approximately one second in the average wait time over the existing controller. In a more "atypical" 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* RL algorithm.

3.3. The Elevator Testbed

The particular elevator system we study in this paper is a simulated 10-story building with 4 elevator cars. The simulator was written by Lewis (1991). Passenger arrivals at each floor are assumed to be Poisson, with arrival rates that vary during the course of a day. Our simulations use a traffic profile (Bao et al., 1994) that specifies arrival rates for every 5-minute interval during a typical afternoon down-peak rush hour. Table 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 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

3.3.1. 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. Further, the simulated acceleration and deceleration times are always the same, whereas in a real system they 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. Although this is not realistic for cars moving at top speed, the concept of making stopping decisions at suitable positions is valid.

Although the elevator cars in this system are homogeneous, the learning techniques described in this paper 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.

3.3.2. 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:

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

3.3.3. 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 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.

3.3.4. 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 a passenger's arrival and 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 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).

4. The Algorithm and Network Architecture

This section describes the multi-agent RL algorithm that we applied to elevator group control. In our scheme, each agent is responsible for controlling one elevator car. The reward structure of each agent's environment is defined in terms of the waiting times of the passengers, and it is more natural to refer to costs rather than rewards because we wish to minimize the average wait time. Each agent uses a modification of Q-learning (Watkins, 1989) for discrete-event systems (Bradtke & Duff, 1995). Together, they employ a collective form of RL. We begin by describing the modifications needed to extend Q-learning to a discrete-event framework, and then derive a method for determining appropriate reward 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.

4.1. Discrete-Event Reinforcement Learning

Elevator systems can be modeled as *discrete event* systems (Cassandras, 1993), 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 RL algorithms is inadequate. This problem can be approached using a variable discount factor that depends on the amount of time between events. 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. In the experiments described in this paper $\beta = 0.01$. 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 explicit knowledge of the state transition probabilities. Bradtke & Duff (1995) extended the Q-learning update rule of Watkins (1989) to 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 step-size parameter, and c_τ and β are defined as above. The quantity $e^{-\beta(t_y-t_x)}$ acts as a variable discount factor that depends on the amount of time between events.

Bradtke & Duff (1995) considered the case in which c_τ is constant between events. We extend their formulation to the case in which 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 4.2.1.)

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 are known in a real elevator system. The number of subsequent passengers to arrive and their exact waiting times are not available. We examine two ways of dealing with this problem, which we call *omniscient* and *on-line* reinforcement schemes.

Since the simulator has access to the waiting times of all passengers, it could use this information to produce the necessary reinforcement signals. We call this the *omniscient* reinforcement scheme, since it requires 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 the omniscient reinforcement scheme is 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 let the system learn using only information that would be available to a real system on-line. Such an *on-line* reinforcement scheme assumes 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 4.2.2. As described in Section 5.4, using the on-line reinforcement scheme produces results that are almost as good as those obtained with the omniscient reinforcement scheme.

4.2. Collective Discrete-Event Q-Learning

There are two main types of elevator system events. Events of the first type are important in calculating waiting times, and therefore are important in calculating the reinforcement signal used by the RL algorithm. These include passenger arrivals and transfers in and out of cars in the omniscient case, or hall button events in the on-line case. The second type are car arrival events, which are potential decision points for the RL agents controlling each car. A car in motion between floors generates a car arrival event when it reaches the point at which it must decide whether to stop at, 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.

4.2.1. Calculating Omniscient Reinforcements In the omniscient reinforcement scheme, accumulated cost is 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 cost 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 cost accumulated between events is the same for all the cars since they share the same objective function, but the amount of cost each car accumulates 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 cost it has incurred 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,

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

4.2.2. Calculating On-line Reinforcements In the on-line reinforcement scheme, accumulated cost is 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 a passenger ceases to wait when a car arrives to service the passenger's 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,

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

4.2.3. Making Decisions and Updating Q-Values A car moving between floors generates a car arrival event when it reaches the point at which it must decide whether to stop at the next floor or to 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" (decreased) during learning. The value of T controls the amount of randomness in the selection of actions. At the beginning of learning, when the Q-value estimates are very inaccurate, high values of T are used, which give nearly equal probabilities to each action. Later in learning, 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 5.3, 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 above, 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 cost accumulator $R[i]$ to zero.

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

4.3. The Networks Used to Store the Q-Values

Using lookup tables to store the Q-values is ruled out for such a large system. Instead, we used feedforward neural networks and the error backpropagation algorithm (Rumelhart et al., 1986). 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 $\hat{Q}(x, a, \theta)$, where θ is a vector of the parameters or weights of the networks. The weight update equation is:

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

where α is a positive step-size parameter, and the gradient $\bigtriangledown_\theta \hat{Q}(x, a, \theta)$ is the vector of partial derivatives of $\hat{Q}(x, a, \theta)$ with respect to each component of θ.

At the start of learning, the weights of each network are initialized to be uniform random numbers between -1 and $+1$. Some experiments in this paper 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 ($= 1$) 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 is 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 of the unit corresponding to its current floor, but a moving car causes activation of several units corresponding to the floors it is approaching, with the unit representing the closest floor showing the highest activation. Activations caused by the various cars are additive. 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 Section 5, we introduce other representations, including some with more restricted state information.

4.4. Parallel and Decentralized 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 is given 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.

5. Results and Analysis

5.1. Basic Results Versus Other Algorithms

Since an optimal policy for the elevator group control problem is unknown, we compared our algorithm with 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 Bao et al. (1994). RLp and RLd denote the RL controllers, parallel and decentralized. The RL controllers each learned over 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 2 shows the results for the traffic profile with down traffic only. Table 3 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 learned during down-only traffic, yet it generalized well when up traffic was added and upward moving cars were forced to stop for any upward hall calls. Table 4 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 was twice as much up traffic, and the RL agents generalized extremely well to this new situation.

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.

Table 2. 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 3. 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

5.2. *Analysis of Decentralized Results*

The success of the decentralized RL algorithm suggests several questions: How similar are the learned policies of the agents to one another and to the policy learned by the parallel algorithm? Can the results be improved by using a voting scheme? What happens if one agent's policy is used to control all of the cars? This section addresses these questions.

Table 4. 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

The simulator was modified to compare the action selections of the four decentralized Q-network agents as well as the parallel Q-network on every decision by every car. During one hour of simulated elevator time, there were a total of 573 decisions required. The four agents 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, although this apparently had little impact on overall performance. The complete results are listed in Table 5.

Table 5. Amount of agreement between decentralized 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 were some situations in which the agents disagreed. In the next experiment the agents voted on which actions should be selected for all of the cars. We examined three ways of resolving voting ties: in favor of STOP (RLs), in favor of CONTINUE (RLc), or randomly (RLr). Table 6 shows the results of this voting scheme compared to the original decentralized algorithm (RLd). The results were averaged over 30 hours of simulated elevator time on pure down traffic.

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 suggests that the agents generally agree on the most important decisions, and disagree only on decisions of little consequence, i.e., where the Q-values are very similar for the different action choices.

In the next experiment the agent for a single car selected actions for all the cars. RL1 used the agent for car 1 to control all the cars, RL2 used the agent for car 2, and so on. Table 7

Table 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

compares these controllers to the original decentralized algorithm (RLd). The results were averaged over 30 hours of simulated elevator time on pure down traffic.

Table 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.

5.3. 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 8 and Figure 2, which show the results of one learning 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 denotes the number of simulated hours of learning completed. Once again, the results were measured over 30 hours of simulated elevator time. Although they are somewhat noisy due to not being averaged over multiple learning runs, the trend is still quite clear.

Each of the schedules that we tested shared the same starting and ending temperatures. Although annealing can be ended at any time, if the amount of time available for learning 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 need for an agent to explore now must be balanced with the need

Table 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

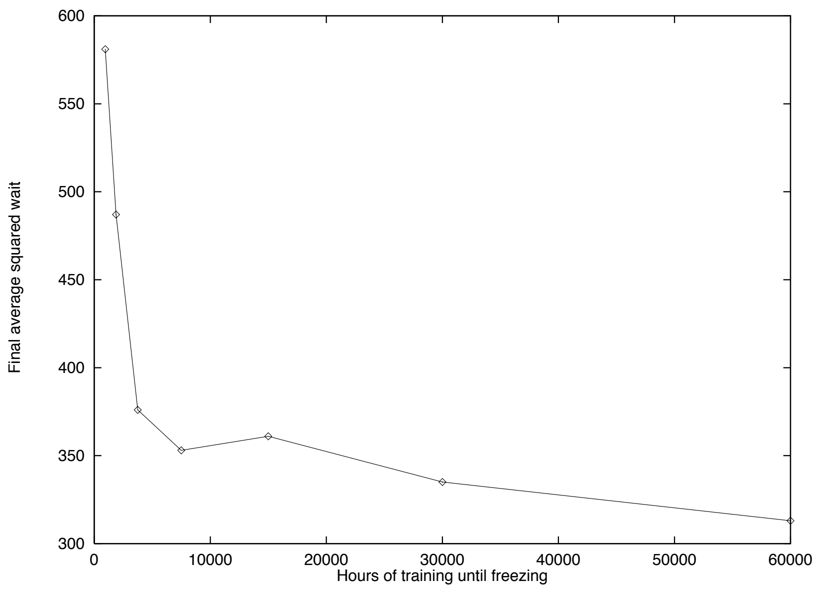

Figure 2. The effect of varying the annealing rate.

for the other agents to learn in a stationary environment. 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 (1992, 1994, 1995) notes a slightly different but related phenomenon in the context of zero-sum games, where learning with self-play allows an agent to learn with a well-matched opponent during each stage of its development.

5.4. Omniscient Versus On-line Reinforcement Schemes

This section examines the relative performance of the omniscient and on-line reinforcement schemes described in Section 4.1, given the same network structure, step-size parameter, and annealing schedule. As shown in Table 9, the omniscient reinforcement scheme led to slightly better performance than did the on-line reinforcement scheme. This should be of little concern regarding the application of RL to a real elevator system since one would want to let the initial learning take place during 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 learning. For application to a real elevator system, initial learning could occur from simulated experience, and the networks could be fine-tuned using actual experience.

Table 9. Omniscient versus on-line reinforcements.

	AvgWait	SquaredWait	SystemTime	Pct>60 secs
Omniscient	15.2	332	42.1	0.07
On-line	15.3	342	41.6	0.16

5.5. 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. Here we examine how performance degrades as the amount of state information received by the agents decreases. We varied the amount of information available to the agents along two dimensions: information about the hall call buttons; and information about the location, direction, and status of the other cars.

We experimented with four input representations for the hall call buttons: 1) REAL, consisting of 18 input units, where two units encode information about each of the nine down hall buttons (the representation described in Section 4.3 used to obtain the results reported above); 2) BINARY, consisting of 9 binary input units corresponding to the nine down hall buttons; 3) QUANTITY, consisting of two input units measuring the number of hall calls above and below the current decision-making car, and 4) NONE, with no input units conveying information about the hall buttons.

We experimented with three input representations for the configuration of the other cars: 1) FOOTPRINTS, consisting of 10 input units, where each unit represents one of the 10 floors where the other cars may be located (the representation described in Section 4.3 used to obtain the results reported above); 2) QUANTITY, consisting of 4 input units that represent the number of upward and downward moving cars above and below the decision-making car; and 3) NONE, consisting of no input units conveying information about the configuration of the other cars.

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, learned over 12000 hours of simulated elevator time using the down-peak profile and omniscient reinforcement scheme. The temperature T was annealed according to the schedule $T = 2.0 * (.9995)^h$, where h denotes the number of simulated hours of learning. The step-size parameter was decreased according to the schedule $\alpha = 0.01 * (.99975)^h$.

The results shown in Table 10 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 learning runs. Nevertheless, they show some interesting trends.

Table 10. Average squared wait times with various levels of incomplete state information.

Hall Buttons	Location of Other Cars		
	Footprints	Quantity	None
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. This constraint probably 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 step-size parameter had to be lowered in some cases. For a further discussion of network instability, see Section 5.7.

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 information. 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.

5.6. Practical Issues

One of the greatest difficulties in applying RL to the elevator control problem was finding the correct temperature and step-size parameters. It was very helpful to start with a scaleddown version consisting of just 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 step-size schedules.

The importance of focusing the experience of the learner onto the most appropriate areas of the state space cannot be overstressed (Barto et al., 1995). Learning along simulated trajectories is an important start, but adding reasonable constraints such as those described in Section 3.3.3 also helps. Further evidence supporting the importance of focusing is that given a choice between learning on heavier or lighter traffic than one expects to face during testing, it is better to learn on the heavier traffic. This gives the system more experience with long queue lengths, where making the correct decisions is crucial.

5.7. *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 step-size parameter 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 step-size parameter of 10^{-2} is suitable for learning by a random initial network on moderate traffic (700 passengers/hour), it very consistently brings on instability in heavy traffic (1900 passengers/hour). However, a step-size of 10^{-3} keeps the network stable even in heavy traffic. If the network learns using this smaller step-size parameter for several hundred simulated hours of elevator time, leading to weights that represent a more consistent set of Q-values, then the parameter can be safely raised back up to 10^{-2} without causing instability.

5.8. *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 that learned using the Least Mean Square algorithm (Widrow & Stearns, 1985). The linear network had a much greater tendency to be unstable. In order to keep the weights from growing without bound, the step-size parameter 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 error in the updates of the Q-values, resulting in extremely poor performance. This failure of linear networks lends support to the contention that elevator control is a difficult problem.

6. Discussion

The policies learned by both the parallel and decentralized multi-agent RL architectures were able to outperform all of the elevator algorithms with which we compared them. Gradual annealing appeared to be a crucial factor enabling the RL architectures to obtain these successful policies, which were very similar for the two architectures. Learning was effective using both omniscient and on-line reinforcement schemes. The algorithms were robust, easily generalizing to new situations such as added up traffic. Finally, they degraded gracefully in the face of decreasing levels of 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 can be very powerful techniques for addressing very large-scale stochastic dynamic optimization problems.

A crucial ingredient in the success of multi-agent RL is the careful control of the amount of exploration performed by each agent. Exploration in this context means trying an action believed to be suboptimal in order to gather additional information about its potential value. At the beginning of the learning process, each RL agent chooses actions randomly, without any knowledge of their relative values, and thus all the agents are extremely inept. However, in spite of the noise in the reinforcement signal caused by the actions of the other agents, some actions will begin to appear to be better than others. By gradually annealing (or lowering) the amount of exploration performed by the agents, these better actions will be taken with greater frequency. This gradually changes the environment for each of the agents, and as they continue to explore, they all raise their performance levels in parallel. Although 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 learning to maximize their rewards.

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 learning 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 be interesting to try something other than a uniform annealing schedule for 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 needed to supplement the results for non-sequential multi-agent RL described in Section 2. Another area that needs further study is RL architectures where reinforcement is 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 (Barto, 1989). Finally, it is important to find effective methods for allowing the possibility of explicit communication among the agents.

7. 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.

Multi-agent RL combines some advantages of bottom-up and top-down approaches to the design of multi-agent systems. 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, systems for complex problems can be created with a minimum of human effort.

RL algorithms can learn from actual or simulated experience, allowing them to focus computational effort onto regions of state space that are likely to be visited during actual control. This makes it feasible to apply them to very large-scale 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 can learn to cooperate with one another in attempting to maximize their common rewards.

In order to demonstrate the power of multi-agent RL, we focused on the difficult problem of elevator group supervisory control. We used a team of RL agents, each of which was responsible for controlling one elevator car. Results obtained in simulation surpassed the best of the heuristic elevator control algorithms of which we are aware. Performance was also robust in the face of decreased levels of state information.

Acknowledgments

We thank John McNulty, Christos Cassandras, Asif Gandhi, Dave Pepyne, Kevin Markey, Victor Lesser, Rod Grupen, Rich Sutton, Steve Bradtke, and members of the Adaptive Networks Laboratory for assistance with the simulator and for many helpful discussions. This research was supported by the Air Force Office of Scientific Research under grant F49620–93–1–0269 to A. Barto and the National Science Foundation under grant ECS–9214866 to A. Barto.

References

Axelrod, R.M. (1984). *The Evolution of Cooperation*. New York, NY: Basic Books.

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

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

Barto, A.G., Bradtke, S.J., & Singh, S.P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72, 81–138.

Bertsekas, D.P. & Tsitsiklis, J.N. (1996). *Neuro-Dynamic Programming*. Belmont, MA: Athena Scientific Press.

Bradtke, S.J. (1993). Distributed adaptive optimal control of flexible structures. Unpublished manuscript.

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

Cassandras, C.G. (1993). *Discrete Event Systems: Modeling and Performance Analysis*. Homewood, IL: Aksen Associates.

Crites, R.H. (1996). *Large-Scale Dynamic Optimization Using Teams of Reinforcement Learning Agents*. PhD thesis, University of Massachusetts.

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

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

Dayan, P. & Hinton, G.E. (1993). Feudal reinforcement learning. In S. J. Hanson, J. D. Cowan, and C. L. Giles, (Eds.), *Advances in Neural Information Processing Systems 5*. San Mateo, CA: Morgan Kaufmann.

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

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

Levy, D., Yadin, M., & Alexandrovitz, A. (1977). Optimal control of elevators. *International Journal of Systems Science*, 8, 301–320.

Lewis, J. (1991). *A Dynamic Load Balancing Approach to the Control of Multiserver Polling Systems with Applications to Elevator System Dispatching*. PhD thesis, ECE department, University of Massachusetts.

Littman, M. & Boyan, J. (1993). A distributed reinforcement learning scheme for network routing. Technical Report CMU-CS-93-165, Carnegie Mellon University.

Littman, M.L. (1994). Markov games as a framework for multi-agent reinforcement learning. *Proceedings of the Eleventh International Conference on Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Littman, M.L. (1996). *Algorithms for Sequential Decision Making*. PhD thesis, Brown University.

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

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

Narendra, K.S. & Thathachar, M.A.L. (1989). *Learning Automata: An Introduction*. Englewood Cliffs, NJ: Prentice-Hall.

Ovaska, S.J. (1992). Electronics and information technology in high-range elevator systems. *Mechatronics*, 2, 89–99.

Pepyne, D.L. & Cassandras, C.G. (1997). Optimal dispatching control for elevator systems during uppeak traffic. *IEEE Transactions on Control Systems Technology*, 5, 629–643.

Rumelhart, D.E., McClelland, J.L., & the PDP Research Group. (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge, MA: MIT Press.

Sakai, Y. & Kurosawa, K. (1984). Development of elevator supervisory group control system with artificial intelligence. *Hitachi Review*, 33, 25–30.

Samuel, A.L. (1963). Some studies in machine learning using the game of checkers. In E. Feigenbaum and J. Feldman, (Eds.), *Computers and Thought*. New York, NY: McGraw-Hill.

Sandholm, T.W. & Crites, R.H. (1996). Multiagent reinforcement learning in the iterated prisoner's dilemma. *Biosystems*, 37, 147–166.

Shoham, Y. & Tennenholtz, M. (1993). Co-learning and the evolution of coordinated multi-agent activity.

Siikonen, M.L. (1993). Elevator traffic simulation. *Simulation*, 61, 257–267.

Strakosch, G.R. (1983). *Vertical Transportation: Elevators and Escalators*. New York, NY: Wiley and Sons.

Sutton, R.S. & Barto, A.G. (1998). *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press.

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

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8, 257–277.

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

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38, 58–68.

Tobita, T., Fujino, A., Inaba, H., Yoneda, K., & Ueshima, T. (1991). An elevator characterized group supervisory control system. *Proceedings of IECON*, (pp. 1972–1976).

Tsetlin, M.L. (1973). *Automaton Theory and Modeling of Biological Systems*. New York, NY: Academic Press.

Ujihara, H. & Amano, M. (1994). The latest elevator group-control system. *Mitsubishi Electric Advance*, 67, 10–12.

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

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis, Cambridge University.

Weiss, G. & Sen, S. (1996). *Adaptation and Learning in Multi-Agent Systems*. Lecture Notes in Artificial Intelligence, Volume 1042. Berlin: Springer Verlag.

Widrow, B. & Stearns, S.D. (1985). *Adaptive Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall.

Received January 31, 1997
Accepted September 18, 1997
Final Manuscript January 31, 1997