Course Notes

Reinforcement Learning III: Applications

Richard S. Sutton and Andrew G. Barto

These notes describe the TD-Gammon application. An attached paper describes a small collection other control applications.

1 TD-Gammon: An Extended Example

One of the most impressive applications of temporal-difference learning to date has been that by Gerry Tesauro of IBM to learning to play the game of backgammon (Tesauro, 1992, 1994). Teasuro's program, *TD-Gammon*, required little backgammon knowledge and yet played extremely well, near that of the world's strongest grandmasters. The learning algorithm in TD-Gammon was a straightforward combination of TD prediction and a nonlinear function approximator in the form of a multi-layer connectionist network trained by backpropagation. In this section we briefly describe this work as an extended, non-trivial example of the application of temporal-difference learning.

Backgammon is a major game in the sense that it is played throughout the world, with numerous tounaments and regular world championship matches. It is in part a game of chance, and it is a popular vehicle for waging significant sums of money. There are probably more professional backgammon players than there are professional chess players. The game itself is played with 15 white and 15 black pieces on a board of 24 locations, called *points*. Figure 1 shows a typical position early in the game, seen from the perspective of the white player.
White has just rolled the dice and obtained a 5 and a 2. This means that he can move one of his pieces 5 steps and one (possibly the same piece) 2 steps. For example, he could move two pieces from the 12 point, one to the 17 point, and one to the 14 point. White's objective is to advance all of his pieces into the last quadrant (points 19-24) and then off the board. The first player to remove all his pieces wins. One complication is that the pieces interact as they pass each

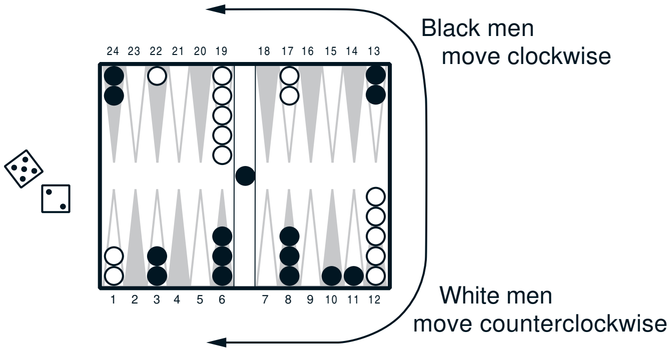

Figure 1: A Backgammon Position

other going in different directions. For example, if it were black's move in Figure 3.8, he could use the dice roll of 2 to move a piece from the 24 point to the 22 point, "hitting" the white piece there. Pieces that have been hit are placed on the "bar" in the middle of the board (where we already see one previously hit black piece), from whence they re-enter the race from the start. However, if there are two pieces on a point, then the opponent cannot move to that point; the pieces are protected from being hit. Thus, white cannot move either of his pieces on the 1 point at all because their possible result squares are occupied by groups of black pieces. The formation of contiguous blocks of occupied points to block the opponents is one of the elementary strategies of the game.

The game of backgammon involves several further complications, but the above description gives the basic idea. With 30 pieces and 24 possible locations (25, counting the bar) it should be clear that there are an enormous number of possible backgammon positions, far more than the number of memory elements one could have in any physically realizable computer. There are also a large number of moves possible from each position. For a typical dice roll there might be 20 different ways of playing. In considering future moves, such as the response of the opponent, one must consider the possible dice rolls as well. The result is an effective branching factor of about 400. That is far too large to permit effective use of the conventional search algorithms that have proved so effective in games like chess and checkers.

On the other hand, the game is a good match to the capabilities of temporal-difference learning methods. Although stochastic, there is available at each time a complete description of the game's state. The game evolves over a sequence of moves and positions until finally ending in a win for one player or the other, ending the game. The outcome can be interpretted as a final reward to be predicted. On the other hand, there is no chance that any of the theoretical results we have described so far will apply to this task. This is so not only because the state space is so large we will not be able to use table lookup or a linearly independent

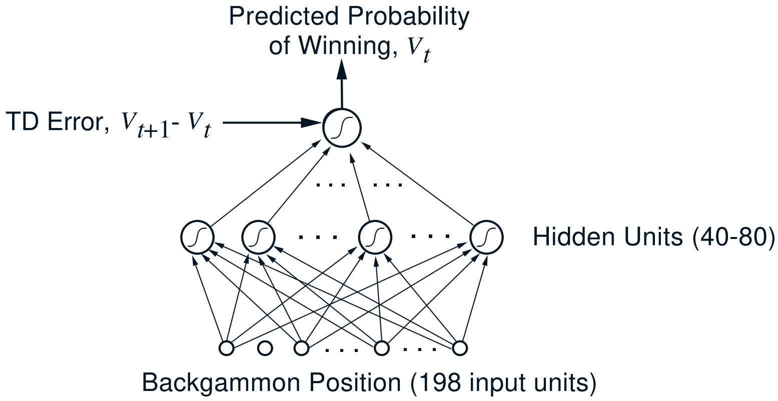

Figure 2: The connectionist network used in TD-Gammon

representation, but also because of the uncertainties and time variations of the opponent.

Tesauro applied a nonlinear form of TD prediction to this problem. The learned value function $V_t(s)$ was meant to return an estimate of the probability of winning starting from state (board position) s. The learned value function was implemented by a multi-layer connectionist network, as shown in Figure 2.[1] The input to the network was a representation of a backgammon position, and the output was an estimate of the value of the position. The value of a won position was defined as 1, and of a lost position as 0. Thus, just as in the random-walk example, the value of a position was the probability of winning from it.

Backgammon positions were represented to the network via 198 input units. The representation used in the first version, TD-Gammon 0.0, was a relatively raw representation of the board position and involved little backgammon knowledge. However, it did involve substantial knowledge of how connectionist networks work and how information is best presented to them. It is instructive to note the exact representation Tesauro chose. For each point on the backgammon board, 4 units indicated the number of white pieces on the point. If there were no white pieces, then all 4 units took on the value zero. If there was one piece, then the first unit took on the value 1. If there were two pieces, then both the first and the second unit were 1. If there were three or more pieces on the point, then all of the first three units were 1. If there were more than three pieces, the fourth unit also came on, to a degree indicating the number of additional pieces beyond three. If n was the total number of pieces and it was greater than three, then the fourth unit took on the value $\frac{n-3}{2}$. With four units for white and four for black, at each of the 24 points, that made a total of 192 units. Two additional units encoded the number of white and black pieces on the bar (these took on the

[1]The network used in TD-Gammon actually had two additional units in its final layer, to estimate the probability of each player winning in a special way called a "gammon" or "backgammon". See exercise 6.5.

value $n/2$, where n is the number of pieces) and two more encoded the number of black and white pieces already successfully removed from the board (these took on the value $n/15$, where n is the number of pieces already borne off). Finally, two units indicated in a binary fashion whether it was white's or black's turn to move. The general logic behind these choices should be clear. Basically it is an attempt to straightforwardly represent the position. Little attempt is made to minimize the absolute number of units. Instead one unit is provided for each conceptually distinct possibility that seems likely to be relevant. And the units are scaled into roughly the same range, in this case into the 0–1 range.

TD-Gammon used a classic multi-layer connectionist network. Corresponding to each connection from input unit to hidden unit was a real valued weight. The signals from the input units were each multiplied by their correposnding weight, and summed at the hidden unit. The output $h(j)$ of hidden unit j was a nonlinear "sigmoid" function of the weighted sum:

$$h(j) = \sigma(\sum_i w(i,j)\phi(i)) = \frac{1}{1 + exp(-\sum_i w(i,j)\phi(i)} \tag{1}$$

where $\phi(i)$ is the ith input signal and $w(i,j)$ is its connection to the jth hidden unit. The computation from hidden units to the output unit is entirely analogous. Each hidden-output connection has a separate weight. The output unit forms the weighted sum and then passes it through the same sigmoid nonlinearity. The output of the sigmoid is always between 0 and 1, and has a natural interpretation as a probability based on a summation of evidence.

The weights of the network were adjusted according to a version of the TD prediction method and the classic backpropagation learning algorithm (Rumelhart, Hinton, & Williams, 1986). The general gradient-descent form of the TD prediction algorithm is

$$\Delta w_t = \alpha\Big[r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)\Big]\nabla_{w_t} V_t(s_t) \tag{2}$$

where w_t is the vector of all modifiable parameters (the weights of the network). This gradient can be computed efficiently according to the backpropagation procedure. For the backpropagation application, $\gamma = 1$ and the reward r_t is zero on all time steps except those arriving in the terminal winning position. Thus, if we consider the winning terminal positions to have fixed value 1 and the losing terminal positions to have fixed value 0, then the TD-error portion of the learning rule is just $V_t(s_{t+1}) - V_t(s_t)$, as suggested in Figure 2.

In order to apply the learning rule we need a source of backgammon games. Tesauro cleverly obtained an unending sequence of games by playing his learning backgammon player against itself. To choose its moves, TD-Gammon considers each of the 20 or so ways it could play its dice role and the corresponding positions that would result. For each resulting position, the the connectionist network was applied to estimate its value. The move was then selected that lead to the position with the highest estimated value. Continuing in this way, with TD-Gammon making the moves for both sides, it was possible to easily generate large numbers of backgammon games. Each game was treated as a trial, with the sequence of positions acting as the states, $s_1, s_2, \ldots$. Tesauro applied the nonlinear TD rule (2) fully incrementally, after each individual move.

The weights of the network were set initially to small random values. The initial evaluations were thus entirely arbitrary. Since the moves were selected on the basis of these evaluations, the initial moves were inevitably poor, and the initial games often lasted hundreds or thousands of moves before one side or the other won, almost by accident. After a few dozen games however, performance improved rapidly.

After playing about 300,000 games against itself, TD-Gammon 0.0 as described above learned to play approximately as well as the best previous backgammon computer programs. This is striking because the previous champion computer programs had all used extensive backgammon knowledge. For example, the reigning champion computer program at the time was arguably *Neurogammon*, another program written by Gerry Tesauro that used connectionist networks but not TD learning. Neurogammon's network was trained based on a large training corpus of exemplary moves provided by backgammon experts, and, in addition, started with a set of features specially crafted for backgammon. Neurogammon was a highly tuned, highly effective backgammon program that decisively won the World Backgammon Olympiad in 1989. TD-Gammon 0.0, on the other hand, was constructed with essentially zero backgammon knowledge. That it was able to do as well as Neurogammon and all other approaches is striking testamony to the potential of such self-play learning procedures.

The tournament success of TD-Gammon 0.0 with zero backgammon knowledge, suggested an obvious modification: add the specialized backgammon features but keep the self-play TD learning method. This produced TD-Gammon 1.0. TD-Gammon 1.0 was clearly substantially better than all previous backgammon programs and only found serious competition among the human experts. The most recent versions of the program, TD-Gammon 2.0 (40 hidden units) and TD-Gammon 2.1 (80 hidden units), were also augmented with a selective 2-ply search procedure. To select their moves, these programs looked ahead not just to the positions that would immediately result, but also to the opponent's possible dice rolls and moves. Assuming the opponent always took the move that appeared immediately best for him, the expected value of each candidate move was computed and the best selected. To save computer time, a 2-ply search was only conducted for candidate moves that were ranked highly after the 1-ply search, about 4 or 5 moves on average. 2-ply search affected only the moves selected; the learning process proceeded exactly as before.

Tesauro was able to play his programs in a significant number of games against world-class human players. A summary of the results in given in Table 1. Based on these results and analyses by backgammon grandmasters (Robertie, 1992; Wolfy, 1994) TD-Gammon 3.0 appears to be at or near the playing strength of the best human players in the world. It may already be the world champion. These programs have also already changed the way the best human players play the game. For example, TD-Gammon learned to play certain opening positions differently than was the convention among the best human players. Now, based on TD-Gammon's success and further analysis, the best people play those positions as TD-Gammon does (Tesauro, 1995).

Program	**Hidden Units**	**Training Games**	**Opponents**	**Results**
TD-Gam 0.0	40	300,000	Other Programs	Tied for Best
TD-Gam 1.0	80	300,000	Robertie, Magriel, ...	-13 pts /51 games
TD-Gam 2.0	40	800,000	Var. Grandmasters	-7 pts /38 games
TD-Gam 2.1	80	1,500,000	Robertie	-1 pts /40 games
TD-Gam 3.0	80	1,500,000	Kazaross	+6 pts /20 games

Table 1: Summary of TD-Gammon Results

2 Elevator Dispatching

Waiting for an elevator is a situation with which we are all familiar. We press a button and then wait for an elevator to arrive travelling in the right direction. We may have to wait a long time if there are too many passengers or not enough elevators. Just how long we wait depends on the algorithm the elevators use to decide where to go. For example, if passengers on several floors have requested pickups, which should be served first? If there are no pickup requests, how should the elevators distribute themselves to await the next one? Elevator dispatching is a good example of a stochastic optimal control problem of economic importance tyhat is too large to be solved by classical techniques such as dynamic programming.

To get a sense of the potential scale of the elevator dispatching problem, consider the four-elevator, ten-floor system shown Figure 3 and studied by Crites and Barto in 1995. Along the righthand side we have pickup requests and an indication of how long each has been waiting. Each elevator has a position, direction, and speed, plus a set of buttons of its own to indicate where passengers want to get off. Roughly quantizing the continuous variables, Crites and Barto estimated the system as having over 10^{22} states. This large state space alone rules out classical dynamic programming methods such as value iteration. Even if one state could be backed up every microsecond it would still require over 1000 years to complete just one sweep through the state space.

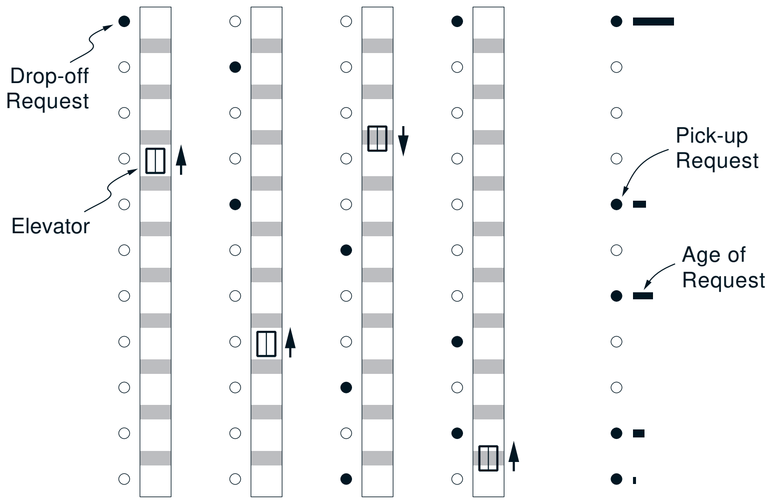

Figure 3: Four Elevators in a Ten-Story Building.

In practice, modern elevator dispatching algorithms are designed heuristically and their performance evaluated on simulated buildings. The simulators are quite sophisticated and detailed. The physics of each elevator car is modeled in continuous time with continuous state variables. Passenger arrivals are modeled as discrete, stochastic events, with poisson arrival rates varying frequently over the course of a simulated day. Not surprisingly, the times of greatest traffic and greatest challenge to the dispatching algorithm are the morning and evening rush hours. Dispatching algorithms are generally designed primarily with these difficult periods in mind.

Figure 4 shows the performance of several different dispatching algorithms during a simulated evening rush hour. Performance here is measured in several different ways, but all with respect to an average passenger entering the system. The average *wait* time is how long the passenger waits before getting on an elevator, whereas the average *system* time is how long she waits before being dropped off at the destination floor. Also shown is the percentage of passengers whose wait time exceeds 60 seconds. Perhaps the most important abjective is the fourth one shown, the average *squared* wait time. This objective is very commonly used because it tends to keep the wait times low while also encouraging fairness in serving all the passengers.

The dispatchers for whom results are shown in Figure 4 include methods like those commonly used in the industry, a variety of heuristic methods, sophisticated research algorithms based on approximating optimal methods (Bao et al., 1994), and two dispatchers constructed through reinforcement learning. By all measures, the performance of the RL dispatchers compares very favorably with that of the other methods. To our knowledge these RL dispatchers are the best feasible

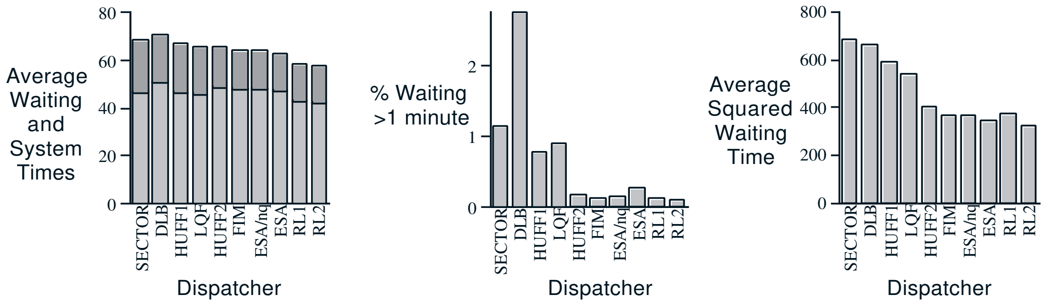

Figure 4: Comparison of Elevator Dispatchers on Simulated Building Shown in Figure 3. The "SECTOR" dispatcher is an industry standard method. The last two dispatchers were constructed by reinforcement learning methods.

solutions to this problem.

In order to obtain these results, Crites and Barto used Q-learning and back-propagation augmented in several ways to take advantage of special features of this problem. The most important of these concerned how they formulated the decisions. First, each elevator made its decisions independently of the others. Second, constraints were placed upon its decisions. An elevator carrying passengers could not pass by a floor where some of its passengers wanted to get off, nor could it reverse direction until all of its passengers in its current direction had reached their floor. In addition, a car was not allowed to stop at a floor unless someone wanted to get on or off there, and it could not stop to pick up passengers at a floor if another elevator was already stopped there. Finally, given a choice between moving up or down, the elevator was constrained always to move up (otherwise evening rush hour traffic tends to push all the elevators down to the bottom of the building). These last three constraints were explicitly provided in order to provide some prior knowledge and make the problem easier for the learning system. The net result of all these constraints was that each elevator had to make very few and very simple decisions. The only decision that had to be made was whether or not to stop at a floor that was being approached and where passengers were waiting to be picked up. At all other times, there really were no choices to be made.

The fact that each elevator had to make choices only at infrequent times permitted the second major simplification of the problem. As far as the learning agent was concerned, the system was considered to make discrete jumps from one time at which it had to make a decision to the next. When a continuous-time decision problem is treated as a discrete-time system in this way it is known as a *semi-Markov* decision process. To a large extent, such processes can be treated just like any other Markov decision process by taking the reward on each discrete transition simply as the integral of the reward over the continuous-time interval making up the transition. The situation is just a little more complex in the discounted, infinite-horizon case studied by Crites and Barto. For example, the notion of return generalizes naturally from a discounted sum of future rewards to a discounted *integral* of future rewards:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \qquad \text{becomes} \qquad R_t = \int_0^{\infty} e^{-\beta\tau} r_{t+\tau}\, d\tau$$

where r_t is the usual immediate reward in discrete time and r_τ is the reward at continuous time τ. In the elevator problem the reward was the negative of the sum of the squared waiting times of all waiting passengers. The parameter $\beta > 0$ plays a role similar to that of the discount-rate parameter γ.

The basic idea of the Q-learning algorithm for semi-Markov decision problems can now be explained. Suppose the system is in state s and takes action a at time t_1, and then the next decision is required at time t_2 in state s'. After this discrete-event transition, the Q-learning update rule for the table-lookup case would be:

$$\Delta Q(s,a) = \alpha \left[\int_{t_1}^{t_2} e^{-\beta(\tau - t_1)} r_\tau \, d\tau + e^{-\beta(t_2 - t_1)} \min_{a'} Q(s', a') - Q(s,a) \right]$$

Note how $e^{-\beta(t_2-t_1)}$ acts as a variable discount factor that depends on the amount of time between events.

One complication is that the reward as defined—the negative sum of the squared waiting times—is not something that would normally be known while an actual elevator was running. This is because in practice one does not know how many people are waiting at a floor, only how long the button requesting a pickup on that floor has been pressed. In a simulator one can know and use this information, but another approach, using only information that one would know in a real elevator, ended up proving more effective. This is simply to use knowledge of how long the button has been pushed and of the arrival rate to compute an *expected* summed squared waiting time for each floor. This approach requires some knowledeg of the arrival rates, but has the advantage that it is less noisy than the actual waiting times during a particular experience. The results presented in Figure 4 are based on using this approach.

Crites and Barto represented the Q-function as a connectionist network trained by gradient descent, i.e., by error backpropagation as described in Chapter 8. The results presented above were obtained using networks with 47 input units, 20 hidden units, and 2 output units, one for each action. The encoding of the state in the input units was one of the most important factors in the success of their system. The 47 input units were as follows:

- 18 - The elapsed times for the pickup-request buttons (2-10 down) encoded using five units each.

- 16 - The location and direction (2-9 up/down) of the elevator whose decision is required.
- 10 - The location of the other elevators superimposed over the 10 floors. Each elevator had a "footprint" that depended on its direction and speed. For example, a stopped elevator caused activation only on the unit corresponding to its current floor, but a moving elevator caused activation on several units corresponding to the floors it was approaching, with the highest activations on the closest floors.
- 1 - This unit was on if the elevator whose decision is required was at the highest floor with a passenger waiting.
- 1 - This unit was on if the elevator whose decision is required was at the floor with the passenger that had been waiting for the longest amount of time.
- 1 - Bias unit was always on.

Crites and Barto experimented with two architectures. In one, each elevator was given its own Q-function and its own network. In the other, there was only one network and one Q-function, with the experiences of all four elevators contributing to learning in the one network. In both cases, each elevator made its decisions completely independently of the other elevators, but shared a single reward signal with them. This introduces additional stochasticity as far as each elevator is concerned because its reward depends in part on the actions of the other elevators, which it cannot control. The architecture in which each elevator has its own Q-function enables different elevators to learn different things. For example, one elevator might learn to dwell in the upper floors and another in the lower floors. On the other hand, the architecture with a common Q-function may be able to learn up to four times faster because it learns simultaneously from the experiences of all elevators. Training time is an issue here, even though the system was trained in simulation. To obtain the results in Figure 4, the RL systems trained for about four days of computer time on a 100 mips processor (corresponding to about 60,000 hours of simulated time). On the other hand, although a considerable amount of computation, this is neglible compared to what would be required by any conventional dynamic programming algorithm.

The networks were trained in the usual way, by simulating a great many evening rush hours while making dispatching decisions using the developing, learned Q-functions. Crites and Barto used a Boltzmann distribution to pick actions, reducing the "temperature" gradually over training. A temperature of zero was used during the tests whose results are given above. Additional details of their procedure can be found in (**?**).

3 Job–Shop Scheduling

Zhang and Dietterich's application of RL to job–shop scheduling (?; ?) differs from the applications just described in that job–shop scheduling is usually formulated as a deterministic combinatorial optimization problem, not as an MDP. However, Zhang and Dietterich showed how RL can be used to improve a combinatorial optimization algorithm enabling it to find better schedules faster than competing methods. They addressed the NASA space–shuttle payload processing problem, which requires scheduling the tasks required for installation and testing of shuttle cargo bay payloads. Some of the tasks must be performed before launch; others are performed after the shuttle has landed. Each task has a duration and a list of resource requirements. A feasible schedule is one which specifies the start time of each task and the resource pool used to satisfy its resource requirements. The objective is to find a feasible schedule that minimizes the total schedule duration. The best non–learning method for this task is an iterative repair method that uses heuristics and simulated annealing (?). This method conducts a state–space search in the space of schedules starting with a suboptimal critical–path schedule formed by igoring resource constraints. Two repair operators are used to improve the schedule, and simulated annealing is used to minimize a cost function.

Zhang and Dietterich's RL system is based on formulating this search as an MDP whose states are schedules and whose actions are the repair operators. Constraint violations produce penalties, and a reward is produced inversely related to schedule length. To learn an evaluation function, they used Sutton's TD(λ) algorithm and the error backpropagation algorithm to train a multi–layer neural network, as in TD–Gammon. The input to the network was a hand–crafted feature vector representing the current schedule. Also as in TD–Gammon, the system selected actions via a one–step lookahead search. 100 scheduling problems were created for training and testing the system. One can think of finding a schedule for one of these problems as analogous to a simulated game of backgammon. The system uses this experience to improve its evaluation function: it learns the skill of efficiently finding short, high–quality schedules for this problem domain. After training, the resulting knowledge, which is in the form of a closed–loop policy for directing the search in schedule space, can be used on other problems in the same domain. Zhang and Dietterich's results show that the use of this policy to direct search clearly outperforms Zweben's algorithm in terms of the quality of the resulting schedules and the number of search steps needed to construct them. While these results must be regarded as preliminary in estabilishing the utility of RL for job–shop scheduling, they suggest that RL can be useful for producing efficient search policies for difficult combinatorial optimization problems.

An important aspect of this application of RL is that the system learns not just to perform well in creating *one* schedule (a skill that would not be useful), but in creating a *class* of schedules. This means that the MDP underlying the RL algorithm's operation is actually more abstract than indicated above. Each state of this MDP represents a class of schedules that share certain properties, as determined by the feature–vector representation chosen. This MDP's state transitions therefore represent an aggregate of the transitions among the particular schedules in these classes. The success of the system is therefore critically dependent on the choice of features used to represent schedules, and these features have to be selected in a domain–dependent way, or developed autonomously by the system itself. Zhang and Dietterich begin to address this problem in ref. (**?**), but overall we currently have little understanding of what makes it possible for RL to succeed in this kind of application. It is clear, however, that RL is well suited for applications like this because it can form good approximate solutions to MDPs with very large state sets and it does not require an explicit probabilistic model of the MDP.

4 Channel Allocation for Mobile Cellular Radio

5 Other Applications

Manufacturing, Robotics (Mahadevan and Connel, Maes, Long-ji Lin, Miller), Routing, Scheduling, Kelly's use of learning automata, DAX prediction, ATM scheduling, flight control, gas-pipeline via BB, other games, operating systems, active vision (Chapman, Whitehead, Peng), Bioreactor, ship control.