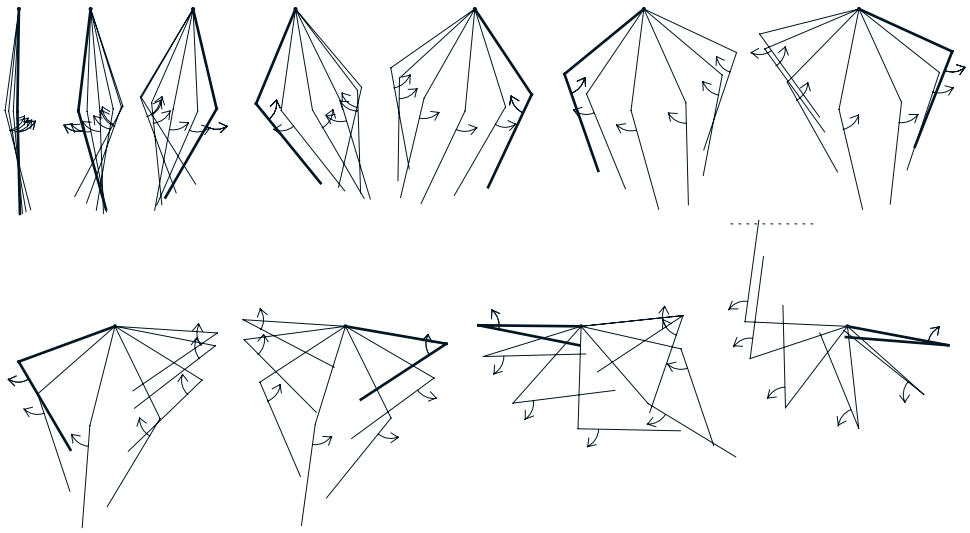

Figure 14.7: A typical learned behavior of the acrobot. Each group is a series of consecutive positions, the thicker line being the first. The arrow indicates the torque applied at the second joint.

14.4 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 traveling 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 dispatching strategy 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 request? Elevator dispatching is a good example of a stochastic optimal control problem of economic importance that is too large to solve by classical techniques such as dynamic programming.

Crites and Barto (1996; Crites, 1996) studied the application of reinforcement learning techniques to the four-elevator, ten-floor system shown in Figure 14.8. Along the right-hand side are 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 to indicate where passengers want to get off. Roughly quantizing the continuous variables, Crites and Barto estimated that the system has over 10^{22} states. This large state set 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.

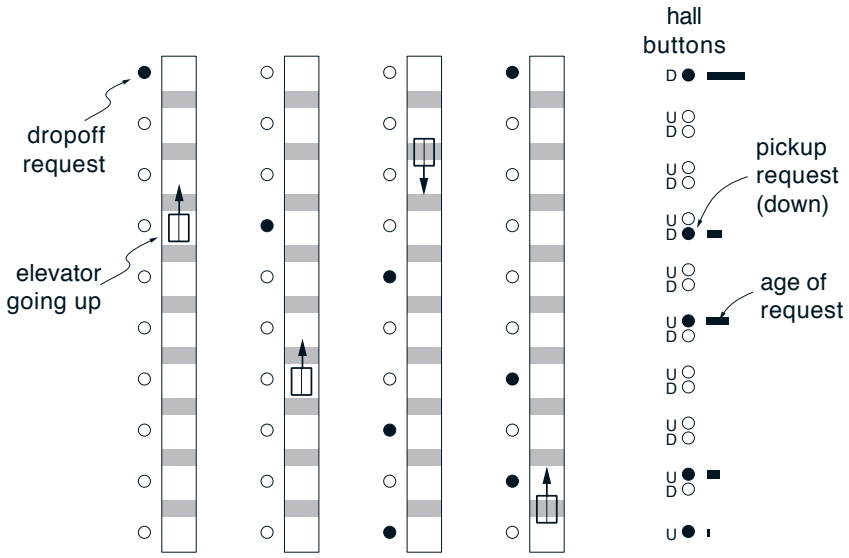

Figure 14.8: Four elevators in a ten-story building.

In practice, modern elevator dispatchers are designed heuristically and 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 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. Dispatchers are generally designed primarily for these difficult periods.

The performance of elevator dispatchers is measured in several different ways, all with respect to an average passenger entering the system. The average *waiting time* is how long the passenger waits before getting on an elevator, and the average *system time* is how long the passenger waits before being dropped off at the destination floor. Another frequently encountered statistic is the percentage of passengers whose waiting time exceeds 60 seconds. The objective that Crites and Barto focused on is the average *squared waiting time*. This objective is commonly used because it tends to keep the waiting times low while also encouraging fairness in serving all the passengers.

Crites and Barto applied a version of one-step Q-learning augmented in several ways to take advantage of special features of the problem. The most important of these concerned the formulation of the actions. First, each elevator made its own decisions independently of the others. Second, a number of constraints were placed on the decisions. An elevator carrying passengers could not pass by a floor if any of its passengers wanted to get off there, nor

could it reverse direction until all of its passengers wanting to go in its current direction had reached their floors. 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 would tend to push all the elevators down to the lobby). These last three constraints were explicitly included to provide some prior knowledge and make the problem easier. The net result of all these constraints was that each elevator had to make few and simple decisions. The only decision that had to be made was whether or not to stop at a floor that was being approached and that had passengers waiting to be picked up. At all other times, no choices needed to be made.

That each elevator made choices only infrequently permitted a second simplification of the problem. As far as the learning agent was concerned, the system made 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 as the integral of the reward over the corresponding continuous-time interval. The notion of return generalizes naturally from a discounted sum of future rewards to a discounted *integral* of future rewards:

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

where R_t on the left is the usual immediate reward in discrete time and $R_{t+\tau}$ on the right is the instantaneous reward at continuous time $t + \tau$. In the elevator problem the continuous-time reward is 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 $\gamma \in [0, 1)$.

The basic idea of the extension of Q-learning to 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 semi-Markov Q-learning backup for a tabular action-value function, Q, would be:

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left[\int_{t_1}^{t_2} e^{-\beta(\tau - t_1)} R_\tau\, d\tau + e^{-\beta(t_2 - t_1)} \min_a Q(S', a) - Q(S, A) \right].$$

Note how $e^{-\beta(t_2 - t_1)}$ acts as a variable discount factor that depends on the amount of time between events. This method is due to Bradtke and Duff (1995).

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 a real elevator system one does not know how many people are waiting at a floor, only how long it has been since the button requesting a pickup on that floor was pressed. Of course this information is known in a simulator, and Crites and Barto used it to obtain their best results. They also experimented with another technique that used only information that would be known in an on-line learning situation with a real set of elevators. In this case one can use how long since each button has been pushed together with an estimate of the arrival rate to compute an *expected* summed squared waiting time for each floor. Using this in the reward measure proved nearly as effective as using the actual summed squared waiting time.

For function approximation, a nonlinear neural network trained by back-propagation was used to represent the action-value function. Crites and Barto experimented with a wide variety of ways of representing states to the network. After much exploration, their best results were obtained using networks with 47 input units, 20 hidden units, and two output units, one for each action. The way the state was encoded by the input units was found to be critical to the effectiveness of the learning. The 47 input units were as follows:

- 18 units: Two units encoded information about each of the nine hall buttons for down pickup requests. A real-valued unit encoded the elapsed time if the button had been pushed, and a binary unit was on if the button had not been pushed.
- 16 units: A unit for each possible location and direction for the car whose decision was required. Exactly one of these units was on at any given time.
- 10 units: 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. No information was provided about which one of the other cars was at a particular location.
- 1 unit: This unit was on if the elevator whose decision was required was at the highest floor with a passenger waiting.
- 1 unit: This unit was on if the elevator whose decision was required was at the floor with the passenger who had been waiting for the longest amount of time.

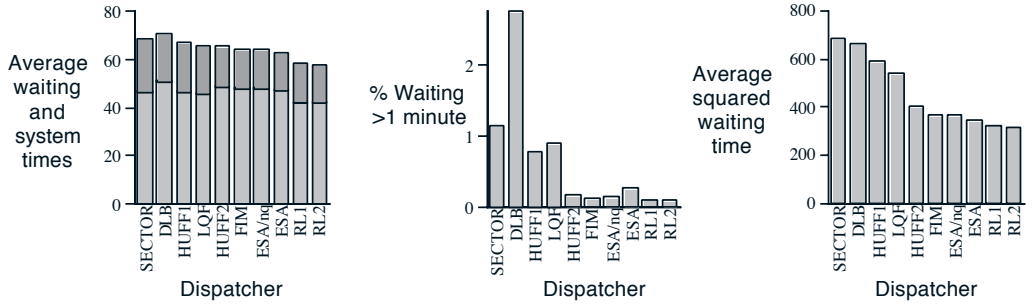

Figure 14.9: Comparison of elevator dispatchers. The SECTOR dispatcher is similar to what is used in many actual elevator systems. The RL1 and RL2 dispatchers were constructed through reinforcement learning.

- 1 unit: Bias unit was always on.

Two architectures were used. In RL1, each elevator was given its own action-value function and its own neural network. In RL2, there was only one network and one action-value function, with the experiences of all four elevators contributing to learning in the one network. In both cases, each elevator made its decisions independently of the other elevators, but shared a single reward signal with them. This introduced additional stochasticity as far as each elevator was concerned because its reward depended in part on the actions of the other elevators, which it could not control. In the architecture in which each elevator had its own action-value function, it was possible for different elevators to learn different specialized strategies (although in fact they tended to learn the same strategy). On the other hand, the architecture with a common action-value function could learn faster because it learned simultaneously from the experiences of all elevators. Training time was an issue here, even though the system was trained in simulation. The reinforcement learning methods were trained for about four days of computer time on a 100 mips processor (corresponding to about 60,000 hours of simulated time). While this is a considerable amount of computation, it is negligible compared with what would be required by any conventional dynamic programming algorithm.

The networks were trained by simulating a great many evening rush hours while making dispatching decisions using the developing, learned action-value functions. Crites and Barto used the Gibbs softmax procedure to select actions as described in Section 2.3, reducing the "temperature" gradually over training. A temperature of zero was used during test runs on which the performance of the learned dispatchers was assessed.

Figure 14.9 shows the performance of several dispatchers during a simulated evening rush hour, what researchers call *down-peak* traffic. The dispatchers

include methods similar to those commonly used in the industry, a variety of heuristic methods, sophisticated research algorithms that repeatedly run complex optimization algorithms on-line (Bao et al., 1994), and dispatchers learned by using the two reinforcement learning architectures. By all of the performance measures, the reinforcement learning dispatchers compare favorably with the others. Although the optimal policy for this problem is unknown, and the state of the art is difficult to pin down because details of commercial dispatching strategies are proprietary, these learned dispatchers appeared to perform very well.

14.5 Dynamic Channel Allocation

An important problem in the operation of a cellular telephone system is how to efficiently use the available bandwidth to provide good service to as many customers as possible. This problem is becoming critical with the rapid growth in the use of cellular telephones. Here we describe a study due to Singh and Bertsekas (1997) in which they applied reinforcement learning to this problem.

Mobile telephone systems take advantage of the fact that a communication channel—a band of frequencies—can be used simultaneously by many callers if these callers are spaced physically far enough apart that their calls do not interfere with each another. The minimum distance at which there is no interference is called the *channel reuse constraint*. In a cellular telephone system, the service area is divided into a number of regions called cells. In each cell is a base station that handles all the calls made within the cell. The total available bandwidth is divided permanently into a number of channels. Channels must then be allocated to cells and to calls made within cells without violating the channel reuse constraint. There are a great many ways to do this, some of which are better than others in terms of how reliably they make channels available to new calls, or to calls that are "handed off" from one cell to another as the caller crosses a cell boundary. If no channel is available for a new or a handed-off call, the call is lost, or *blocked*. Singh and Bertsekas considered the problem of allocating channels so that the number of blocked calls is minimized.

A simple example provides some intuition about the nature of the problem. Imagine a situation with three cells sharing two channels. The three cells are arranged in a line where no two adjacent cells can use the same channel without violating the channel reuse constraint. If the left cell is serving a call on channel 1 while the right cell is serving another call on channel 2, as in the left diagram below, then any new call arriving in the middle cell must be blocked.

Obviously, it would be better for both the left and the right cells to use channel 1 for their calls. Then a new call in the middle cell could be assigned channel 2, as in the right diagram, without violating the channel reuse constraint. Such interactions and possible optimizations are typical of the channel assignment problem. In larger and more realistic cases with many cells, channels, and calls, and uncertainty about when and where new calls will arrive or existing calls will have to be handed off, the problem of allocating channels to minimize blocking can become extremely complex.

The simplest approach is to permanently assign channels to cells in such a way that the channel reuse constraint can never be violated even if all channels of all cells are used simultaneously. This is called a *fixed assignment* method. In a *dynamic assignment* method, in contrast, all channels are potentially available to all cells and are assigned to cells dynamically as calls arrive. If this is done right, it can take advantage of temporary changes in the spatial and temporal distribution of calls in order to serve more users. For example, when calls are concentrated in a few cells, these cells can be assigned more channels without increasing the blocking rate in the lightly used cells.

The channel assignment problem can be formulated as a semi-Markov decision process much as the elevator dispatching problem was in the previous section. A state in the semi-MDP formulation has two components. The first is the configuration of the entire cellular system that gives for each cell the usage state (occupied or unoccupied) of each channel for that cell. A typical cellular system with 49 cells and 70 channels has a staggering 70^{49} configurations, ruling out the use of conventional dynamic programming methods. The other state component is an indicator of what kind of event caused a state transition: arrival, departure, or handoff. This state component determines what kinds of actions are possible. When a call arrives, the possible actions are to assign it a free channel or to block it if no channels are available. When a call departs, that is, when a caller hangs up, the system is allowed to reassign the channels in use in that cell in an attempt to create a better configuration. At time t the immediate reward, R_t, is the number of calls taking place at that time, and the return is

$$G_t = \int_0^\infty e^{-\beta\tau} R_{t+\tau}\, d\tau,$$

where $\beta > 0$ plays a role similar to that of the discount-rate parameter γ. Maximizing the expectation of this return is the same as minimizing the expected (discounted) number of calls blocked over an infinite horizon.

This is another problem greatly simplified if treated in terms of afterstates (Section 6.6). For each state and action, the immediate result is a new configuration, an afterstate. A value function is learned over just these configurations. To select among the possible actions, the resulting configuration was determined and evaluated. The action was then selected that would lead to the configuration of highest estimated value. For example, when a new call arrived at a cell, it could be assigned to any of the free channels, if there were any; otherwise, it had to be blocked. The new configuration that would result from each assignment was easy to compute because it was always a simple deterministic consequence of the assignment. When a call terminated, the newly released channel became available for reassigning to any of the ongoing calls. In this case, the actions of reassigning each ongoing call in the cell to the newly released channel were considered. An action was then selected leading to the configuration with the highest estimated value.

Linear function approximation was used for the value function: the estimated value of a configuration was a weighted sum of features. Configurations were represented by two sets of features: an availability feature for each cell and a packing feature for each cell–channel pair. For any configuration, the availability feature for a cell gave the number of additional calls it could accept without conflict if the rest of the cells were frozen in the current configuration. For any given configuration, the packing feature for a cell–channel pair gave the number of times that channel was being used in that configuration within a four-cell radius of that cell. All of these features were normalized to lie between -1 and 1. A semi-Markov version of linear TD(0) was used to update the weights.

Singh and Bertsekas compared three channel allocation methods using a simulation of a 7×7 cellular array with 70 channels. The channel reuse constraint was that calls had to be 3 cells apart to be allowed to use the same channel. Calls arrived at cells randomly according to Poisson distributions possibly having different means for different cells, and call durations were determined randomly by an exponential distribution with a mean of three minutes. The methods compared were a fixed assignment method (FA), a dynamic allocation method called "borrowing with directional channel locking" (BDCL), and the reinforcement learning method (RL). BDCL (Zhang and Yum, 1989) was the best dynamic channel allocation method they found in the literature. It is a heuristic method that assigns channels to cells as in FA, but channels can be borrowed from neighboring cells when needed. It orders the channels in each cell and uses this ordering to determine which channels to borrow and how calls are dynamically reassigned channels within a cell.

Figure 14.10 shows the blocking probabilities of these methods for mean arrival rates of 150, 200, and 350 calls/hour as well as for a case in which

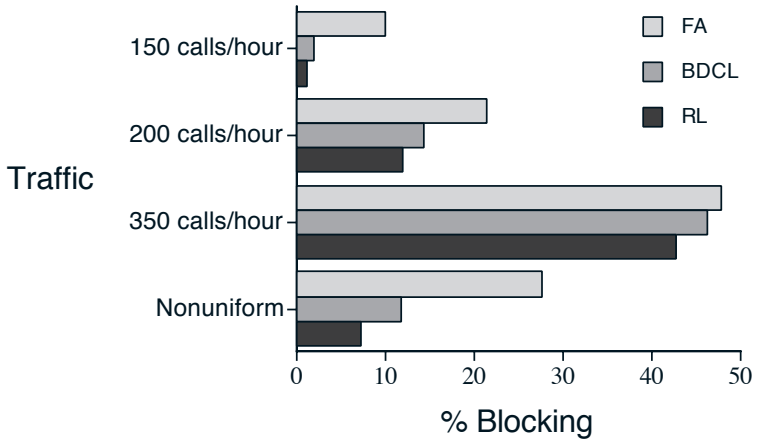

Figure 14.10: Performance of FA, BDCL, and RL channel allocation methods for different mean call arrival rates.

different cells had different mean arrival rates. The reinforcement learning method learned on-line. The data shown are for its asymptotic performance, but in fact learning was rapid. The RL method blocked calls less frequently than did the other methods for all arrival rates and soon after starting to learn. Note that the differences between the methods decreased as the call arrival rate increased. This is to be expected because as the system gets saturated with calls there are fewer opportunities for a dynamic allocation method to set up favorable usage patterns. In practice, however, it is the performance of the unsaturated system that is most important. For marketing reasons, cellular telephone systems are built with enough capacity that more than 10% blocking is rare.

Nie and Haykin (1996) also studied the application of reinforcement learning to dynamic channel allocation. They formulated the problem somewhat differently than Singh and Bertsekas did. Instead of trying to minimize the probability of blocking a call directly, their system tried to minimize a more indirect measure of system performance. Cost was assigned to patterns of channel use depending on the distances between calls using the same channels. Patterns in which channels were being used by multiple calls that were close to each other were favored over patterns in which channel-sharing calls were far apart. Nie and Haykin compared their system with a method called MAXAVAIL (Sivarajan, McEliece, and Ketchum, 1990), considered to be one of the best dynamic channel allocation methods. For each new call, it selects the channel that maximizes the total number of channels available in the entire system. Nie and Haykin showed that the blocking probability achieved by their reinforcement learning system was closely comparable to that of MAXAVAIL under a variety of conditions in a 49-cell, 70-channel simulation. A

key point, however, is that the allocation policy produced by reinforcement learning can be implemented on-line much more efficiently than MAXAVAIL, which requires so much on-line computation that it is not feasible for large systems.

The studies we described in this section are so recent that the many questions they raise have not yet been answered. We can see, though, that there can be different ways to apply reinforcement learning to the same real-world problem. In the near future, we expect to see many refinements of these applications, as well as many new applications of reinforcement learning to problems arising in communication systems.

14.6 Job-Shop Scheduling

Many jobs in industry and elsewhere require completing a collection of tasks while satisfying temporal and resource constraints. Temporal constraints say that some tasks have to be finished before others can be started; resource constraints say that two tasks requiring the same resource cannot be done simultaneously (e.g., the same machine cannot do two tasks at once). The objective is to create a schedule specifying when each task is to begin and what resources it will use that satisfies all the constraints while taking as little overall time as possible. This is the job-shop scheduling problem. In its general form, it is NP-complete, meaning that there is probably no efficient procedure for exactly finding shortest schedules for arbitrary instances of the problem. Job-shop scheduling is usually done using heuristic algorithms that take advantage of special properties of each specific instance.

Zhang and Dietterich (1995, 1996; Zhang, 1996) were motivated to apply reinforcement learning to job-shop scheduling because the design of domain-specific, heuristic algorithms can be expensive and time-consuming. Their goal was to show how reinforcement learning can be used to learn how to quickly find constraint-satisfying schedules of short duration in specific domains, thereby reducing the amount of hand engineering required. They addressed the NASA space shuttle payload processing problem (SSPP), which requires scheduling the tasks required for installation and testing of shuttle cargo bay payloads. An SSPP typically requires scheduling for two to six shuttle missions, each requiring between 34 and 164 tasks. An example of a task is MISSION-SEQUENCE-TEST, which has a duration of 7200 time units and requires the following resources: two quality control officers, two technicians, one ATE, one SPCDS, and one HITS. Some resources are divided into pools, and if a task needs more than one resource of a specific type, the resources must belong to the same pool, and the pool has to be the right one.

For example, if a task needs two quality control officers, they both have to be in the pool of quality control officers working on the same shift at the right site. It is not too hard to find a conflict-free schedule for a job, one that meets all the temporal and resource constraints, but the objective is to find a conflict-free schedule with the shortest possible total duration, which is much more difficult.

How can you do this using reinforcement learning? Job-shop scheduling is usually formulated as a search in the space of schedules, what is called a discrete, or combinatorial, optimization problem. A typical solution method would sequentially generate schedules, attempting to improve each over its predecessor in terms of constraint violations and duration (a hill-climbing, or local search, method). You could think of this as a nonassociative reinforcement learning problem of the type we discussed in Chapter 2 with a very large number of possible actions: all the possible schedules! But aside from the problem of having so many actions, any solution obtained this way would just be a *single* schedule for a *single* job instance. In contrast, what Zhang and Dietterich wanted their learning system to end up with was a *policy* that could quickly find good schedules for *any* SSPP. They wanted it to learn a skill for job-shop scheduling in this specific domain.

For clues about how to do this, they looked to an existing optimization approach to SSPP, in fact, the one actually in use by NASA at the time of their research: the iterative repair method developed by Zweben and Daun (1994). The starting point for the search is a *critical path schedule*, a schedule that meets the temporal constraints but ignores the resource constraints. This schedule can be constructed efficiently by scheduling each task prior to launch as late as the temporal constraints permit, and each task after landing as early as these constraints permit. Resource pools are assigned randomly. Two types of operators are used to modify schedules. They can be applied to any task that violates a resource constraint. A Reassign-Pool operator changes the pool assigned to one of the task's resources. This type of operator applies only if it can reassign a pool so that the resource requirement is satisfied. A Move operator moves a task to the first earlier or later time at which its resource needs can be satisfied and uses the critical path method to reschedule all of the task's temporal dependents.

At each step of the iterative repair search, one operator is applied to the current schedule, selected according to the following rules. The earliest task with a resource constraint violation is found, and a Reassign-Pool operator is applied to this task if possible. If more than one applies, that is, if several different pool reassignments are possible, one is selected at random. If no Reassign-Pool operator applies, then a Move operator is selected at random based on a heuristic that prefers short-distance moves of tasks having

few temporal dependents and whose resource requirements are close to the task's overallocation. After an operator is applied, the number of constraint violations of the resulting schedule is determined. A simulated annealing procedure is used decide whether to accept or reject this new schedule. If ΔV denotes the number of constraint violations removed by the repair, then the new schedule is accepted with probability $\exp(-\Delta V/T)$, where T is the current computational temperature that is gradually decreased throughout the search. If accepted, the new schedule becomes the current schedule for the next iteration; otherwise, the algorithm attempts to repair the old schedule again, which will usually produce different results due to the random decisions involved. Search stops when all constraints are satisfied. Short schedules are obtained by running the algorithm several times and selecting the shortest of the resulting conflict-free schedules.

Zhang and Dietterich treated entire schedules as states in the sense of reinforcement learning. The actions were the applicable REASSIGN-POOL and MOVE operators, typically numbering about 20. The problem was treated as episodic, each episode starting with the same critical path schedule that the iterative repair algorithm would start with and ending when a schedule was found that did not violate any constraint. The initial state—a critical path schedule—is denoted S_0. The rewards were designed to promote the quick construction of conflict-free schedules of short duration. The system received a small negative reward (-0.001) on each step that resulted in a schedule that still violated a constraint. This encouraged the agent to find conflict-free schedules quickly, that is, with a small number of repairs to S_0. Encouraging the system to find short schedules is more difficult because what it means for a schedule to be short depends on the specific SSPP instance. The shortest schedule for a difficult instance, one with a lot of tasks and constraints, will be longer than the shortest schedule for a simpler instance. Zhang and Dietterich devised a formula for a *resource dilation factor* (RDF), intended to be an instance-independent measure of a schedule's duration. To account for an instance's intrinsic difficulty, the formula makes use of a measure of the resource overallocation of S_0. Since longer schedules tend to produce larger RDFs, the negative of the RDF of the final conflict-free schedule was used as a reward at the end of each episode. With this reward function, if it takes N repairs starting from a schedule s to obtain a final conflict-free schedule, S_f, the return from s is $-RDF(S_f) - 0.001(N-1)$.

This reward function was designed to try to make a system learn to satisfy the two goals of finding conflict-free schedules of short duration and finding conflict-free schedules quickly. But the reinforcement learning system really has only one goal—maximizing expected return—so the particular reward values determine how a learning system will tend to trade off these two goals.

Setting the immediate reward to the small value of -0.001 means that the learning system will regard one repair, one step in the scheduling process, as being worth 0.001 units of RDF. So, for example, if from some schedule it is possible to produce a conflict-free schedule with one repair or with two, an optimal policy will take extra repair only if it promises a reduction in final RDF of more than 0.001.

Zhang and Dietterich used TD(λ) to learn the value function. Function approximation was by a multilayer neural network trained by backpropagating TD errors. Actions were selected by an ε-greedy policy, with ε decreasing during learning. One-step lookahead search was used to find the greedy action. Their knowledge of the problem made it easy to predict the schedules that would result from each repair operation. They experimented with a number of modifications to this basic procedure to improve its performance. One was to use the TD(λ) algorithm *backward* after each episode, with the eligibility trace extending to future rather than to past states. Their results suggested that this was more accurate and efficient than forward learning. In updating the weights of the network, they also sometimes performed multiple weight updates when the TD error was large. This is apparently equivalent to dynamically varying the step-size parameter in an error-dependent way during learning.

They also tried an *experience replay* technique due to Lin (1992). At any point in learning, the agent remembered the best episode up to that point. After every four episodes, it replayed this remembered episode, learning from it as if it were a new episode. At the start of training, they similarly allowed the system to learn from episodes generated by a good scheduler, and these could also be replayed later in learning. To make the lookahead search faster for large-scale problems, which typically had a branching factor of about 20, they used a variant they called *random sample greedy search* that estimated the greedy action by considering only random samples of actions, increasing the sample size until a preset confidence was reached that the greedy action of the sample was the true greedy action. Finally, having discovered that learning could be slowed considerably by excessive looping in the scheduling process, they made their system explicitly check for loops and alter action selections when a loop was detected. Although all of these techniques could improve the efficiency of learning, it is not clear how crucial all of them were for the success of the system.

Zhang and Dietterich experimented with two different network architectures. In the first version of their system, each schedule was represented using a set of 20 handcrafted features. To define these features, they studied small scheduling problems to find features that had some ability to predict RDF. For example, experience with small problems showed that only four of the resource pools tended to cause allocation problems. The mean and standard deviation

of each of these pools' unused portions over the entire schedule were computed, resulting in 10 real-valued features. Two other features were the RDF of the current schedule and the percentage of its duration during which it violated resource constraints. The network had 20 input units, one for each feature, a hidden layer of 40 sigmoidal units, and an output layer of 8 sigmoidal units. The output units coded the value of a schedule using a code in which, roughly, the location of the activity peak over the 8 units represented the value. Using the appropriate TD error, the network weights were updated using error backpropagation, with the multiple weight-update technique mentioned above.

The second version of the system (Zhang and Dietterich, 1996) used a more complicated time-delay neural network (TDNN) borrowed from the field of speech recognition (Lang, Waibel, and Hinton, 1990). This version divided each schedule into a sequence of blocks (maximal time intervals during which tasks and resource assignments did not change) and represented each block by a set of features similar to those used in the first program. It then scanned a set of "kernel" networks across the blocks to create a set of more abstract features. Since different schedules had different numbers of blocks, another layer averaged these abstract features over each third of the blocks. Then a final layer of 8 sigmoidal output units represented the schedule's value using the same code as in the first version of the system. In all, this network had 1123 adjustable weights.

A set of 100 artificial scheduling problems was constructed and divided into subsets used for training, determining when to stop training (a validation set), and final testing. During training they tested the system on the validation set after every 100 episodes and stopped training when performance on the validation set stopped changing, which generally took about 10,000 episodes. They trained networks with different values of λ (0.2 and 0.7), with three different training sets, and they saved both the final set of weights and the set of weights producing the best performance on the validation set. Counting each set of weights as a different network, this produced 12 networks, each of which corresponded to a different scheduling algorithm.

Figure 14.11 shows how the mean performance of the 12 TDNN networks (labeled G12TDN) compared with the performances of two versions of Zweben and Daun's iterative repair algorithm, one using the number of constraint violations as the function to be minimized by simulated annealing (IR-V) and the other using the RDF measure (IR-RDF). The figure also shows the performance of the first version of their system that did not use a TDNN (G12N). The mean RDF of the best schedule found by repeatedly running an algorithm is plotted against the total number of schedule repairs (using a log scale). These results show that the learning system produced scheduling algorithms that needed many fewer repairs to find conflict-free schedules of the

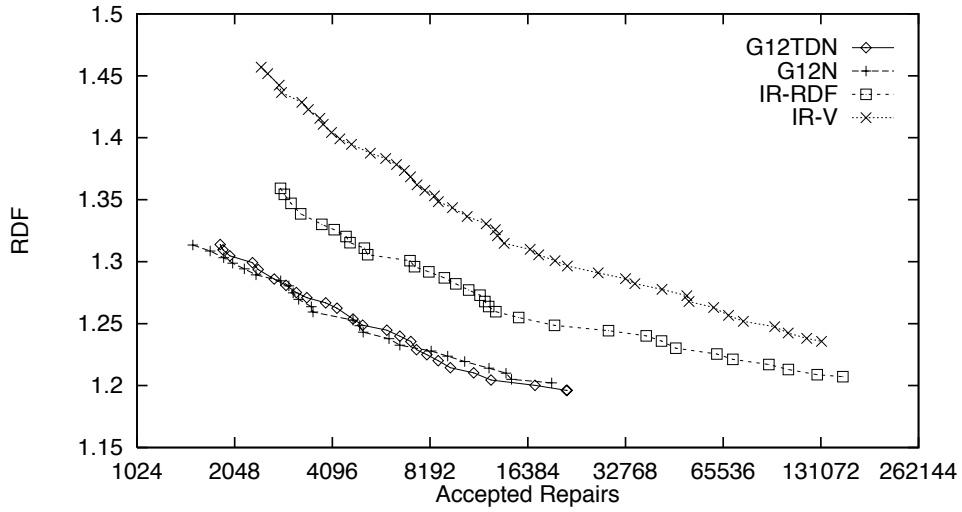

Figure 14.11: Comparison of accepted schedule repairs. Reprinted with permission from Zhang and Dietterich, 1996.

same quality as those found by the iterative repair algorithms. Figure 14.12 compares the computer time required by each scheduling algorithm to find schedules of various RDFs. According to this measure of performance, the best trade-off between computer time and schedule quality is produced by the non-TDNN algorithm (G12N). The TDNN algorithm (G12TDN) suffered due to the time it took to apply the kernel-scanning process, but Zhang and Dietterich point out that there are many ways to make it run faster.

These results do not unequivocally establish the utility of reinforcement learning for job-shop scheduling or for other difficult search problems. But they do suggest that it is possible to use reinforcement learning methods to learn how to improve the efficiency of search. Zhang and Dietterich's job-shop scheduling system is the first successful instance of which we are aware in which reinforcement learning was applied in *plan-space*, that is, in which states are complete plans (job-shop schedules in this case), and actions are plan modifications. This is a more abstract application of reinforcement learning than we are used to thinking about. Note that in this application the system learned not just to efficiently create *one* good schedule, a skill that would not be particularly useful; it learned how to quickly find good schedules for a class of related scheduling problems. It is clear that Zhang and Dietterich went through a lot of trial-and-error learning of their own in developing this example. But remember that this was a groundbreaking exploration of a new aspect of reinforcement learning. We expect that future applications of this kind and complexity will become more routine as experience accumulates.

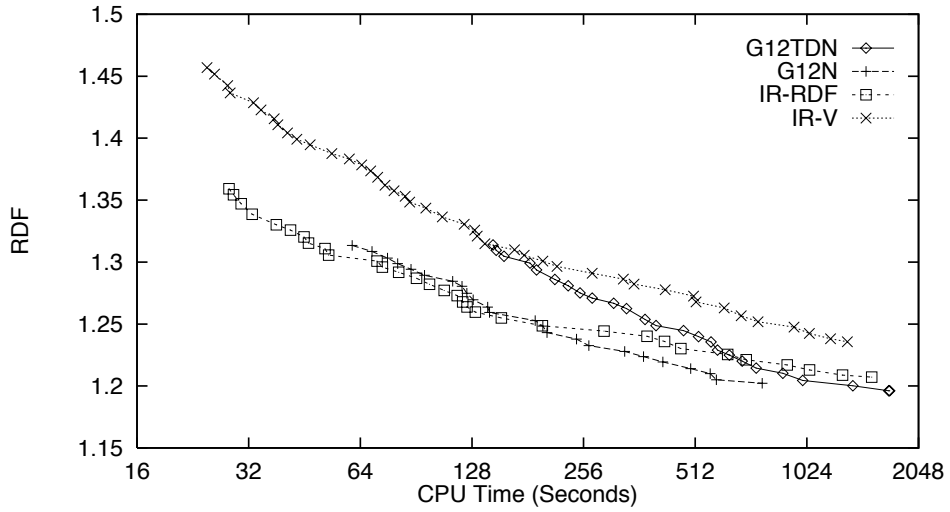

Figure 14.12: Comparison of CPU time. Reprinted with permission from Zhang and Dietterich, 1996.