Value Function Approximations and Job-Shop Scheduling

Wei Zhang and **Thomas G. Dietterich**
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202

May 30, 1995

Abstract

We report a successful application of TD(λ) with value function approximation to the task of job-shop scheduling. Our scheduling problems are based on the problem of scheduling payload processing steps for the NASA space shuttle program. The value function is approximated by a 2-layer feedforward network of sigmoid units. A one-step lookahead greedy algorithm using the learned evaluation function outperforms the best existing algorithm for this task, which is an iterative repair method incorporating simulated annealing. To understand the reasons for this performance improvement, this paper introduces several measurements of the learning process and discusses several hypotheses suggested by these measurements. We conclude that the use of value function approximation is not a source of difficulty for our method, and in fact, it may explain the success of the method independent of the use of value iteration. Additional experiments are required to discriminate among our hypotheses.

1 Introduction

In [Zhang and Dietterich, 1995], we present a reinforcement learning approach to job-shop scheduling. Our research focuses on the application domain of space shuttle payload processing for NASA. The goal is to schedule a set of tasks to satisfy a set of temporal and resource constraints while also seeking to minimize the total duration (makespan) of the schedule. In previous work on this task, Zweben and colleagues [Zweben *et al.*, 1994] developed an iterative repair-based scheduling procedure that combines a set of heuristics with a simulated annealing search procedure. A repair-based scheduler starts with a critical-path schedule and incrementally repairs constraint violations with the goal of finding a short conflict-free schedule. This repair-based scheduling problem can be viewed approximately as a Markov decision problem. Each repair has a cost, and the final, conflict-free, schedule receives a reward inversely proportional to its duration.

We applied the temporal difference algorithm $TD(\lambda)$ with value iteration to train a neural network to learn a heuristic evaluation function over states. This evaluation function is used by a one-step lookahead search procedure to find good solutions to new scheduling problems. The experiment has shown that the scheduler using the learned evaluation function can outperform the iterative repair scheduler—the best known scheduler for the domain—on realistic scheduling tasks.

Although reinforcement learning succeeds in finding high quality schedules quickly, it is not clear why it works for this domain. For this complex domain, we are not able to evaluate the learning performance in terms of the accuracy of prediction for the value function over states, since generalization is performed over multiple scheduling problems. Even for a single problem, there can be huge number of states. In this paper, we describe several measurements to evaluate learning performance as a step toward understanding and explaining the success of reinforcement learning in this domain. Based on our measurements, we introduce some hypotheses and discuss the evidence supporting and opposing these hypotheses.

In the remainder of the paper, we first describe the scheduling task and how this task can be formulated so that TD(λ) can be applied. We then describe our learning method, which involves several modifications to TD(λ). Following this, we present experiments on a set of simulated problems and discuss the learning behavior on both training and validation test problems. Our results indicate that reinforcement learning outperforms the iterative repair scheduler. Finally, we raise and discuss our hypotheses for why reinforcement learning with function approximation works for this domain.

2 The NASA Domain and TD(λ) for the Task

The NASA space shuttle payload processing (SSPP) domain requires scheduling the various tasks that must be performed to install and test the payloads that are placed in the cargo bay of the space shuttle. In job-shop scheduling terminology, each shuttle mission is a job. Each job consists of a partially-ordered set of tasks that must be performed. Each task has a duration and a list of resource requirements. For example, the task `MISSION-SEQUENCE-TEST` has a duration of 7200 and requires two quality-control officers, two technicians, one `ATE`, one `SPCDS`, and one `HITS`. There are 35 different types of resources. There may be many units of a resource available. For example, there are 8 quality control officers available and 25 technicians. However, these available resources may be split into resource pools, so that, for example, the 8 quality control officers might be subdivided into three pools of size 2, 2, and 4. If a task requires two quality control officers, they must both be drawn from the same pool. Resource pools model multiple work shifts and multiple physical locations. A complete schedule must specify the start time of each task and the resource pool by which each resource requirement of each task is satisfied.

A typical SSPP problem involves the simultaneous scheduling of between two and six shuttle missions; each mission involves between 32 and 164 tasks. Hence, the SSPP domain requires solving scheduling problems containing several hundred tasks. Most of these tasks must be performed prior to launch, but some also take place after the shuttle has landed. The former are referred to as "pre-tasks" and the latter as "post-tasks." Each shuttle mission has a fixed launch date, but no starting date or ending date. Hence, pre-tasks have deadlines but no ready times; post-tasks have ready times but no deadlines. A key goal of the scheduling system is to minimize the total duration of the schedule. This is much more challenging than simply finding a *feasible* schedule.

Zweben et al. 1994 developed the following iterative repair method for solving this scheduling problem. First, a critical path schedule is constructed by working backward and forward from the launch and landing dates. Each task prior to launch is scheduled as late as the temporal partial order will permit; each task after landing is scheduled as early as the temporal partial order will permit. Resource constraints are ignored; resource requests are randomly assigned to resource pools. This critical path schedule can be constructed very efficiently, and it provides the starting state for the scheduling problem space. In each state of this problem space, there are two possible operators that can be applied. The REASSIGN-POOL operator changes the pool assignment for one of the resource requirements of a task. It is only applied when the pool reassignment would allow the resource requirement to be successfully satisfied. The MOVE operator moves a task to a different time and then reschedules all of the temporal dependents of the task using the critical path method (leaving the resource pool assignments of the dependents unchanged). The MOVE operator is only applied to move a task to the first earlier or the first later time at which the violated resource requirement can be satisfied. The iterative repair method uses a combination of three heuristics to choose a task to repair. It prefers to move the task that (a) requires an amount of resource nearly equal to the amount that is over allocated, (b) has few temporal dependents, and (c) needs to be moved only a short distance to satisfy the resource request. The overall control structure of the algorithm applies simulated annealing to minimize the number of resource pool violations.

Reinforcement learning methods learn a *policy* for selecting actions in a problem space. The policy tells for each state which action is to be performed in that state. After an action a is chosen and applied in state s, the problem space shifts to state s' and the learning system receives reinforcement $R(s, a, s')$.

To view the scheduling problem as a reinforcement learning problem, we must describe the problem space and the reinforcement function R. We employ the same problem space as Zweben et al. The starting state s_0 is the critical path schedule as discussed above. We define the reinforcement function $R(s, a, s')$ to give a reinforcement of -0.001 for each schedule s' that still contains constraint violations. This assesses a small penalty for each scheduling action (REASSIGN-POOL or MOVE), and it is intended to encourage reinforcement learning to prefer actions that *quickly* find a good schedule. For any schedule s' that is free of violations, the reinforcement is the negative of the *resource dilation factor*, $-RDF(s', s_0)$. The RDF attempts to provide a scale-independent measure of the length of the schedule, and this final reinforcement is intended to encourage reinforcement learning to find short final schedules. Because the reinforcement function depends only on the resulting state, we will write it as $R(s')$.

The RDF is defined as follows. Let $capacity(i)$ be the (fixed) capacity of resource type i—that is, the combined capacity of all resource pools of resource type i. At each time t in the schedule, let $u(i, t)$ be the current utilization of resources of type i. If $u(i, t) > capacity(i)$, then the resource of type i is overallocated at time t (no matter how we assign resource requests to resource pools of this type). We define the *resource utilization index* $RUI(i, t)$ for resource type i at time t to be

$$RUI(i, t) = \max\left\{1, \frac{u(i, t)}{capacity(i)}\right\}.$$

If the resource is not over-allocated, $RUI(i,t)$ is 1; otherwise it is the fraction of overallocation.

The *total resource utilization index* ($TRUI$) for a schedule of length l is the sum of the resource utilization index taken over all n resources and all l times:

$$TRUI = \sum_{i=1}^{n} \sum_{t=1}^{l} RUI(i,t).$$

Given these definitions, the resource dilation factor is defined as

$$RDF(s, s_0) = \frac{TRUI(s)}{TRUI(s_0)}.$$

To understand the rationale behind this formula, first note that in the final schedule s, $TRUI(s)$ is just n times the length of the schedule. This is because in the final schedule, no resource is overallocated, so $RUI(i,t) = 1$. Hence, $TRUI(s) = l \times n$. We could have used the negative of this value as the reinforcement function, but reinforcement learning is easier if the reinforcement function is independent of the difficulty of the scheduling problem. A very difficult problem (e.g., with many jobs that have simultaneous deadlines) would require a very long schedule, whereas a simple problem would require a much shorter schedule. The total resource utilization index of the initial schedule, $TRUI(s_0)$, measures the amount of overallocation of resources in the initial state, and hence, provides a crude measure of the difficulty of the scheduling problem. Hence, we use this to normalize the final schedule length to produce the resource dilation factor.

Now that we have specified how to view repair-based scheduling as a reinforcement learning problem, we turn our attention to the learning algorithm. Suppose at a given point in the learning process we have developed policy π, which says that in state s the best action to select is $a = \pi(s)$. We can define an associated function f_π, called the *value function*, such that $f_\pi(s)$ tells the cumulative reward that we will receive if we follow policy π from state s onward. Formally, $f_\pi(s) = \sum_{j=0}^{N} R(s_{j+1})$, where N is the number of steps until a conflict-free schedule is found.

As in most reinforcement learning work, we will attempt to learn the value function of the optimal policy π^*, denoted $f^* = f_{\pi^*}$, rather than directly learning π^*. Once we have learned this optimal value function, we can transform it into the optimal policy via a simple one-step lookahead search: To choose the best action in state s, we compute the state $a(s)$ that would result from applying each possible action a to state s. For each such action, we compute the value of the resulting state, $f^*(a(s))$, and choose the action a that maximizes this value. Note that this approach requires that we know the effects of our operators—which is certainly true for repair-based scheduling operators.

To learn the value function, we can apply the method of temporal difference learning known as $TD(\lambda)$ developed by Sutton 1988. In $TD(\lambda)$, the value function is represented by a feed-forward neural network, $\hat{f}(s, \mathbf{W})$, where $\mathbf{W}$ is the vector of weights in the network. If the policy π were fixed, $TD(\lambda)$ could be applied to learn the value function f_{pi} as follows. Let $s_0, s_1, \ldots, s_N$ be a sequence of states visited by following policy π with associated reinforcements $R(s_1), \ldots, R(s_N)$. At step $j+1$, we can compute the temporal difference error at step j as

$$J_j = [\hat{f}(s_{j+1}, \mathbf{W}) + R(s_{j+1})] - \hat{f}(s_j, \mathbf{W}).$$

$TD(\lambda)$ then computes the smoothed gradient

$$e_j = \nabla_{\mathbf{W}} \hat{f}(s_j, \mathbf{W}) + \lambda e_{j-1}$$

and updates the weights of the network according to

$$\Delta \mathbf{W} = \alpha J_t e_j.$$

Here, λ is a smoothing parameter that combines previous gradients with the current gradient in e_j, and α is the learning rate.

The $TD(\lambda)$ algorithm was designed to learn the value function for a stationary Markov random process such as would result from following a *fixed* policy. In reinforcement learning, however, we want to apply it to learn the value function of the *optimal* policy starting with an initial, random policy. To do this, we employ a form of value iteration. $TD(\lambda)$ is applied online to the sequences of states and reinforcements that result from choosing actions according to the current estimated value function, $\hat{f}$. At each state s during learning, we conduct a one-step lookahead search using the current estimated value function $\hat{f}$ to evaluate the states resulting from applying each possible operator. We then select the action that maximizes the predicted value of the resulting state s'. After applying this action and receiving the reward, we update our estimate of $\hat{f}$ to reflect the difference between the value of $\hat{f}(s)$ and the more informed value $R(s') + \hat{f}(s')$. (We actually employ a slightly more complex procedure described below.) This means that the policy is continually changing during the learning process. Fortunately, $TD(\lambda)$ will still converge under these conditions [Sutton, 1988].

3 Methods

There are several modifications that we made to this algorithm based on preliminary experiments. First, for any reinforcement learning algorithm it is critical to perform some kind of exploration to discover new and better ways of getting from the start state to the goal. We employed the following simple exploration strategy. At each state, with probability β we choose a random action instead of the action recommended by the current value function and policy. Initially, β is set to 1. After each training sequence, β is decreased by an amount $\Delta\beta$ until it reaches a final value of 0.05. The value used for $\Delta\beta$ is 0.0004. A smaller $\Delta\beta$ value suggests a longer training.

Second, we do not perform weight updates in the neural network after each action. Instead, we remember the sequence of states visited along the path from the starting state to the final conflict-free schedule. Then we update the network starting with the final action and working backward to the start of the action sequence. Experimentally, this works better than simple online training, because the values being backed up are more up-to-date.

Third, we employ Lin's *experience replay* method [Lin, 1992]. During learning, the best sequence of moves from start to goal is remembered, and after every four training sequences, we update the network using this best training sequence. This improved learning and performance significantly.

Fourth, learning rate α is determined automatically according to the backpropagation error. We employ multiple updates for a training state-pair until the error is smaller than 0.05 or 100 updates is conducted. This also improved learning and performance significantly.

Next, to represent the value function, we trained feed-forward networks having 40 sigmoidal hidden units and 8 sigmoidal output units. The 8 output units encode the predicted RDF using the technique of overlapping gaussian ranges [Pomerleau, 1991] as follows. Each output unit represents one assigned RDF value, v_j ($j = 1, \ldots, 8$). For the artificial problems, these RDF values are $v_1 = 0.8, v_2 = 1.0, \ldots, v_8 = 2.2$. During training, the target output activation for each output unit is set to be $target_j = \varphi(RDF - v_j, 0.2)/\varphi(0, 0.2)$, where $\varphi(\mu, \sigma)$ is the standard normal probability density function with mean μ and standard deviation σ. During testing, the predicted RDF value is computed as $(\sum_j act_j \cdot v_j)/(\sum_j act_j)$, where act_j is the actual output activation for output unit j.

Finally, we do not use the actual states of the scheduling process as input to the neural network. The neural network can accept only a fixed vector of feature values describing each state (i.e., each current schedule). Schedules, on the other hand, are variable-length objects. Hence, it was necessary to define a set of useful features that extract important aspects of the current schedule that the neural network can use to predict the value of the state. We defined the following features:

Mean and standard deviation of the free pool capacity for bottleneck pools: Simple experiments showed that only the technician, logistics, electrical engineer, mechanical engineer, and quality control officer resource types became major bottleneck resources. For each bottleneck pool, the number of unallocated units (the free capacity) is measured over the whole schedule period and the mean and standard deviation of this quantity provide two features for each pool.

Mean and standard deviation of slacks: The slack time between a task and one of its temporal prerequisites is the difference between the end time of the prerequisite task and the scheduled start time of the task. We measure the minimum slack for each task (and all of its temporal prerequisites) and the average slack for each task. The mean and standard deviation of these two quantities taken over all tasks provide four features.

Modified RDF: We use a slightly modified version of the resource dilation factor of the current schedule. The numerator of the modified RDF is computed using the capacity and allocation of individual resource-pools rather than of resource types. We use two modified RDF features, the second feature only gives the RDF for final schedules (and has the value 0 for all other schedules). We also included a boolean feature stating whether a schedule is final.

Over-allocation index: This is the total number of units of over-allocated resources in the current schedule divided by the total number of units of over-allocated resources in the starting schedule.

Windows in violation: A window is defined to be a maximal period of time during which the set of currently scheduled tasks does not change. A schedule can be segmented into a sequence of windows. We compute the number of windows and the percentage of windows that contain a constraint violation. We also find the earliest window that contains a constraint violation and compute the percentage of the following 9 windows that have violations.

Percentage of windows in violation that can be resolved by pool reassignment: This is the fraction of those windows having constraint violations where the total amount of resources assigned is actually less than the total capacity, so that—if the resources were not subdivided into pools—the resource requirements could be met.

Percentage of time units in violation: This is measured over the whole schedule period.

First violated window index (normalized): Let w_0 be the index of the earliest window that has a violation. Let W be the total number of windows. Then this feature is $(W - w_0)/W$. As violations are repaired, this value decreases to zero. If no window has a violation, we set $w_0 = W$.

Critical path schedule resource utilization: This is useful to characterize different scheduling problems.

Each of these features was developed by studying small scheduling problems to find quantities that had some ability to predict RDF. However, we believe that these features can be improved substantially, and this is a goal of our ongoing research.

4 Experiment and Evaluation

4.1 Problem Set

We constructed an artificial problem set based on specifications for the NASA SSPP problem. The artificial problems were generated as follows. First, we generated a pool of 20 jobs. From these, we constructed scheduling problems by choosing random subsets of these jobs. This was intended to model the NASA setting where there are only a limited number of possible shuttle-cargo-bay configurations (i.e., jobs), but where each scheduling problem is a unique combination of such shuttle missions. More generally, this models a job shop where each new scheduling interval requires scheduling a unique mix of more-or-less standard jobs.

To generate a synthetic job, we choose the number of tasks randomly in the range 6 to 10 as pre-tasks and the number of tasks in the range 1 to 4 as post-tasks. Together with the launch task, the number of tasks in a job ranges from 8 to 15. The launch task is a milestone task with a fixed schedule time and a fixed work duration (set at 5). The work duration for other tasks is randomly generated in the range 6 to 15. A set of temporal constraints among these tasks is then randomly generated such that approximately 60% of all possible pairwise precedence constraints are asserted. Next, resource requirements are determined for each task. There are two types of resources. Each resource has two pools—one pool has a capacity of 6 units, and the other has a capacity of 8 units. Each task randomly chooses to require 1 or 2 types of resources. Resource amount requirements are then randomly assigned uniformly in the range from 1 to 6 units for each chosen resource type.

Once the pool of 20 jobs is generated in this way, 50 training problems and 50 test problems are constructed. To generate a problem, we first choose the number of jobs in the problem to be either 3 or 4 (with equal probability). The desired number of jobs is selected randomly with replacement from the 20-job pool. Each job is assigned a launch time with the times randomly separated by between 8 and 15 time units.

Twenty-two input features are computed to represent schedules for these problems: 8 pool capacity features for the 4 pools, 4 slack features, 2 modified RDF features (the second one only describing final schedules), 3 features describing windows in violation (number of violations, percentage of violations, percentage of violations in 10 windows), percentage of time units in violation, and percentage of violated windows in which the violation can be resolved by pool reassignment, first violated window index, critical path schedule resource utilization, and a boolean feature describing whether a schedule contains a violation.

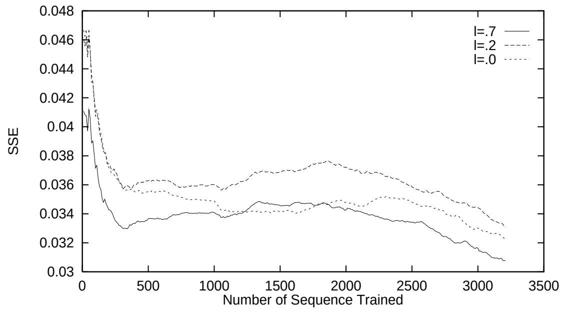

Figure 1: Training Error on 30 Training Problems

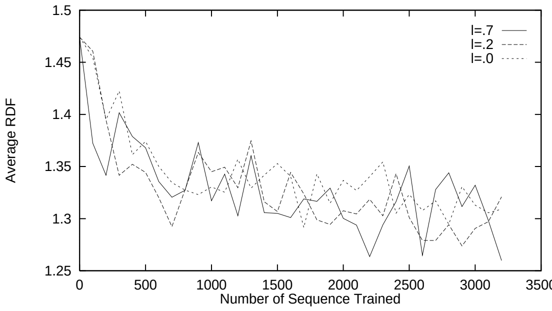

Figure 2: Average RDF over 20 Validation Problems

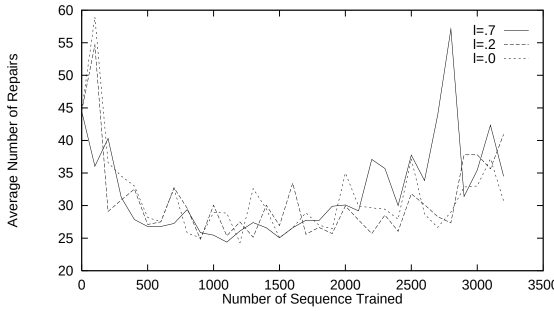

Figure 3: Average Number of Repairs over 20 Validation Problems

4.2 Training and Its Performance Evaluation

During training, 20 of the 50 training problems were held out as a validation set to test the learning behavior. The remaining 30 problems were repeatedly processed to train the value function networks. We use two training problem settings; each chooses different problems for training and validation.

We trained six different networks using all combinations of three λ values, 0.0, 0.2, and 0.7, and two training problem settings. The training set problems are processed in round-robin fashion. Each problem is solved using the β probabilistic exploration strategy to obtain a sequence of states and actions. The network is then updated (via backpropagation with $TD(\lambda)$) by processing the state sequence working backward from the final state. After every 100 passes through the training set, a validation test is conducted. Training is halted when the exploration probability β has reached to 0.05 for one hundred epochs or the measurements on the validation test suggest no need for further training (see below).

We employ five measurements to evaluate the training performance. First, we measure the average backpropagation error over a sequence during training. We only use the error in the first update among multiple weight updates for a state pair. Since training involves exploration of the search space, this error can be very noisy. Our measurement takes an average of the error over the past 1,000 sequences. Figure 1 shows the error for three values of λ and one training problem setting. The horizontal axis gives the number of training sequences processed. This shows that $\lambda = 0.7$ learns faster than the other two λ settings.

The other four measurements evaluate the performance on the validation test. In a val-

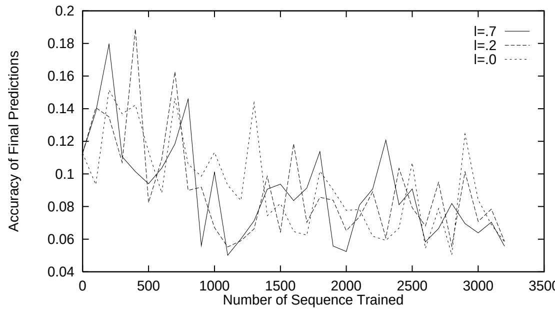

Figure 4: Accuracy of Final Predictions on 20 Validation Problems

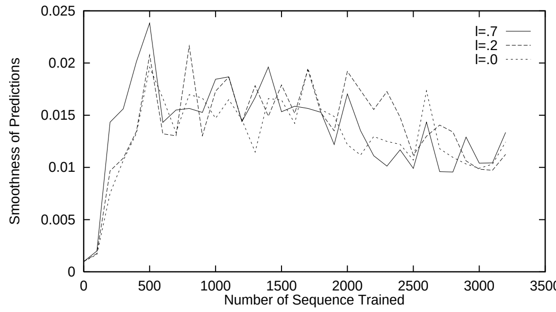

Figure 5: Smoothness of Predictions on 20 Validation Problems

idation test for a network, we employ a one-step lookahead search using the network as the evaluation function to solve the validation problems. First, we measure the average final schedule RDF over 20 validation problems. This evaluates the quality of final schedules that can be produced by the network. Figure 2 shows the average final schedule RDF is improving with training.

Second, we measure the average number of repair actions taken to produce a final schedule. Figure 3 plots the average number of repair actions for these networks to find solutions. This shows that there is significant reduction during the early stages of training, but with more training the networks start making more repairs.

Next, we measure the accuracy of predicted values of states along the search path produced by a learned network. An optimal path produced by the optimal value function has the property that for any state s in this path, the value of the next state s' must be equal to the value of state s plus its reinforcement $R(s)$. Therefore, we can use the prediction error $(\hat{f}(s) - R(s)) - \hat{f}(s')$ for all states s along the path produced by a network to measure how well the value function is learned. We noticed that the biggest error comes from the prediction for the value of the final state, so we separate this measure into two parts: one measures the error for final states and the other measures the error for non-final states. These two measurements are referred to as the accuracy of final predictions and the smoothness of predictions respectively. The smoothness of predictions is defined as the average prediction error for all states except the last one in a path. Figure 4 shows that the accuracy of final predictions improves with training. Figure 5 plots the smoothness of predictions over training. A network is very smooth in the beginning, because all the weights are very small and its predictions are nearly constant. So its smoothness is $|R(s)|$, which is 0.001. With training, the network begins to distinguish different states, and its smoothness error increases. After 500 epochs of training, the network begins to reduce this error.

4.3 Test Results

In optimization problems, there is a tradeoff between the amount of CPU time expended and the quality of the resulting solution. For the iterative repair (IR) scheduler using simulated annealing, this tradeoff can be explored simply by repeatedly restarting the annealing search and retaining the best schedule found.

For the TD scheduling approach (which we will refer to as TD), each learned evaluation

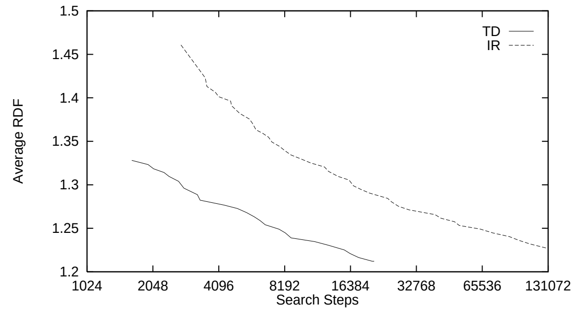

Figure 6: Performance of Comparison of TD to IR in number of search steps

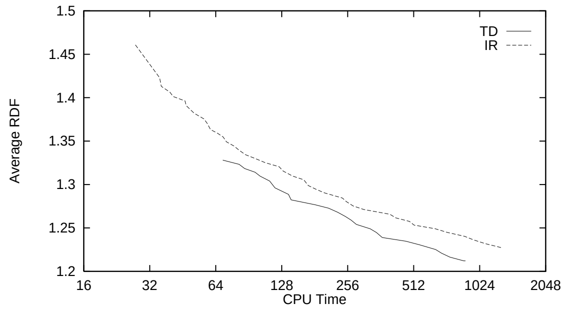

Figure 7: Performance of Comparison of TD to IR in CPU time

function is effectively deterministic, so rerunning it does not produce significantly different results (ties are broken randomly, but such ties do not occur very often or lead to significantly different results). Hence, to explore the tradeoff, we constructed multiple learned networks and computed the quality of the solution that is found by using the single best network, the quality of the best solution found by the 2 top-ranked networks, and so on. The multiple networks are trained using the various configurations described above, and ranked based on a combination of four validation set measurements prior to testing.

Figure 6 compares the performance of TD with IR. IR was run for 50 iterations (an iteration is one restart). We set the starting temperature for each iteration to 50. The temperature was reduced according to $T := 0.98T$ after each accepted repair to the schedule. TD was run for 12 iterations, where each iteration is a restart using a different trained network (in rank order).

The vertical axis is the average RDF of the best conflict-free schedule found so far. The horizontal axis is the log of the number of search steps conducted in scheduling. For TD, a search step is exactly the same as a schedule repair. But for IR, a search step may be either a repair that was made and accepted or a repair that was made and then not accepted by the simulated annealing algorithm. The average number of search steps for an IR iteration over 50 iterations for the 50 test problems is 64.6, and the average number of accepted steps is 52.1. By comparison, the average number of search steps for a TD iteration is only 35.1, which means that TD has captured knowledge for quickly finding schedules. This plot shows that TD can find good solutions with many fewer search steps than IR. TD finished its first iteration for the 50 problems with 1,640 search steps and produced a solution with an average RDF of 1.328, and after 12 iterations and 21,006 steps, it produced a solution with an averege RDF of 1.212. IR needs 2,751 steps to produce its first solution with much worse quality—1.461 average RDF—and ends its 50 iterations with 130,079 steps for a solution with an average RDF of 1.227.

This comparison tells us about the length of the solution paths found by the two algorithms, but it is not a fair comparison of CPU time, because TD must perform a one-step lookahead search during each step and because the cost of evaluating the learned networks is significant. With a branching factor ranging from 2 to 9 for the simulated problems, on average TD spends 4.2 times as much CPU time per step as IR. Figure 7 plots the performance comparison in log CPU time. We see that TD maintains a constant *factor* advantage over IR.

4.4 Discussion

The results clearly show that reinforcement learning is succeeding on this problem. Evidentally, this is another example of "lucky convergence" of value iteration under function approximation [Boyan and Moore, 1995]. However, the results do not show completely successful learning of the value function. For example, while the smoothness error (Figure 5) over the cross-validation set decreases during learning, it is still much larger than the "correct" value of 0.001. Similarly, the final prediction error (Figure 4) is also still substantial, despite the changes we made to the training procedure and to the input features to try to reduce it.

Are these problems an indication of pathologies caused by function approximation? Or are they a consequence of inadequate input features? Based on the training curves, we believe that function approximation is not to blame. Instead, we believe that our input features need more work. One piece of evidence supporting this is that when we added the two features that encode final RDF and signal the final state, our performance improved significantly, both for final state RDF prediction accuracy and also for smoothness. Hence, improvements in the final state accuracy were successfully propagated backward by TD(λ).

It is possible that the use of function approximation alone is responsible for the superiority of TD over IR even ignoring the value-iteration component of our approach. This could be true if the main reason for the superiority of TD in this problem is that it can smooth away local minima in the evaluation function. Briefly, the argument goes as follows. Iterative repair (IR) must use simulated annealing because its evaluation function (number of constraint violations) suffers from local minima that must be overcome by the controlled application of randomness. (We have confirmed this—pure greedy search using this evaluation function gives miserable results.) If these local minima could somehow be smoothed away, then a one-step lookahead greedy algorithm could obtain substantially the same final schedules more quickly (even without using the "true" value function). TD(λ) combined with backpropagation can be viewed as a smoothing operation applied to the network's input features. These input features include some features closely related to the number of constraint violations. So perhaps value iteration plays little role in the success of our TD scheduler.

Yet another explanation of the success of our TD scheduler is that its other input features—particularly current-state-RDF—provide a better evaluation function than the number of constraint violations and that the network is essentially just copying this input feature to its output. We have tested this in two ways. First, we ran a pure greedy algorithm using only current-state-RDF as the value function. This performs only slightly worse than our TD scheduler. Second, we compared the current-state-RDF to the RDF predicted by the TD networks. These are *very* different from each other. Current-state-RDF starts very low and increases rather erratically as the schedule is alternately lengthened and then improved. Predicted RDF generally starts too high and decreases fairly monotonically toward the final state RDF. From these experiments, we conclude both that our input features are good and that the networks are doing substantially more than just copying these input features to the output.

5 Summary and Conclusions

We have shown that a scheduling algorithm based on value iteration with TD(λ) out-performs the best existing algorithm on a set of realistic scheduling problems. The training process clearly improves performance as measured by five different criteria: backpropagation error signal, actual RDF of final schedule, number of repairs, smoothness of predictions, and accuracy of predicted final schedule RDF.

The use of function approximation to represent the value function does not appear to have caused problems in this application. Indeed, we have introduced an hypothesis that suggests that function approximation is critical for obtaining the observed performance improvements.

Much more work is required to understand and explain the success we have observed. A full understanding is critical to developing further improvements and to converting these initial results into a scheduling technology that can be widely applied in industry.

Acknowledgments

The authors thank Rich Sutton and Monte Zweben for several helpful discussions. The authors gratefully acknowledge the support of NASA grant NAG 2-630 from NASA Ames Research Center. Additional support was provided by NSF grants CDA-9216172 and IRI-9204129.

References

[Boyan and Moore, 1995] J. A. Boyan and A. W. Moore. Generalization in reinforcement learning: safely approximating the value function. In *Advances in Neural Information Processing Systems 6*, San Mateo, CA, 1995. Morgan Kaufmann.

[Lin, 1992] Longji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *machine learning*, 8:293–321, 1992.

[Pomerleau, 1991] D. A. Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3(1):88–97, 1991.

[Sutton, 1988] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, August 1988.

[Zhang and Dietterich, 1995] W. Zhang and T. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, 1995. (to appear).

[Zweben *et al.*, 1994] M. Zweben, B. Daun, and M. Deale. Scheduling and rescheduling with iterative repair. In M. Zweben and M. S. Fox, editors, *Intelligent Scheduling*, chapter 8, pages 241–255. Morgan Kaufmann, San Francisco, CA, 1994.