Journal of Artificial Intelligence Research 1 (2000) 1-1 Submitted 6/91; published 9/91

Solving Combinatorial Optimization Tasks by Reinforcement Learning: A General Methodology Applied to Resource-Constrained Scheduling

Wei Zhang ZHANGW@REDWOOD.RT.CS.BOEING.COM
Boeing Research & Technology, P.O. Box 3707, MS 7L-66, Seattle, WA 98124-2207 USA

Thomas G. Dietterich TGD@CS.ORST.EDU
Department of Computer Science, Oregon State University, Corvallis, OR 97331-3202 USA

Abstract

This paper introduces a methodology for solving combinatorial optimization problems through the application of reinforcement learning methods. The approach can be applied in cases where several similar instances of a combinatorial optimization problem must be solved. The key idea is to analyze a set of "training" problem instances and learn a search control policy for solving new problem instances. The search control policy has the twin goals of finding high-quality solutions and finding them quickly. Results of applying this methodology to a NASA scheduling problem show that the learned search control policy is much more effective than the best known non-learning search procedure—a method based on simulated annealing.

1. Introduction

Combinatorial optimization problems such as the traveling salesperson problem (TSP) and the resource-constrained scheduling problem (RCSP) are hard to solve optimally. All interesting formulations of these problems are NP-Hard (Garey & Johnson, 1979; Garey, Johnson, & Smith, 1976). High-quality solutions to instances of these problems can result in huge financial savings in business operations, particularly in the areas of supply chain management, airline flight crew scheduling, and scheduling for manufacturing and assembly.

For particular combinatorial optimization problems, such as the TSP, many years of research have produced several good general-purpose heuristics, such as the Lin-Kernighan heuristic (Lin & Kernighan, 1971). General purpose heuristics have also been discovered for combinatorial search problems such as boolean satisfiability (GSAT, Selman, Levesque, & Mitchell, 1992). For specific application problems, domain-specific heuristics can usually be discovered. For example, Zweben, Daun, and Deale (1994) describe a very good heuristic approach to resource-constrained scheduling problems.

The fact that clever researchers can devise good heuristics shows that commonly-occurring instances of these problems can be solved approximately without requiring infeasible amounts of search. Unfortunately, every new combinatorial optimization problem that arises can require substantial new research to discover these search heuristics.

This paper introduces a methodology for applying reinforcement learning methods to discover application-specific search control policies automatically. We adopt the familiar

©2000 AI Access Foundation and Morgan Kaufmann Publishers. All rights reserved.

state-space search perspective in which a combinatorial optimization problem is solved by starting in some initial state and applying operators to move to new states. The key issue from this perspective is to decide which operator to apply in each state. Our particular approach assumes that the starting state corresponds to an infeasible solution and that the operators progressively improve the candidate solution until it becomes a feasible solution. By making good operator choices, we hope to find a feasible solution that is optimal or near-optimal according to a given objective function. The heuristic for making these operator choices is called the control *policy*.

For a particular problem instance, thorough state space search will eventually discover a good solution. If we keep track of the states that were visited and the operators that were applied, we can also define a (partial) control policy that could find the same solution again, but more quickly—because unfruitful search steps would not be performed. However, learning a control policy that works only for a single problem instance is worthless. Once we have found a good solution to a particular problem instance, we can just remember the solution. We don't need to rediscover it at some future time.

Hence, there is no point in learning a control policy unless it can be applied to new problem instances. The main contribution of this paper is to show how a search control policy can be learned that can be applied to new instances of a combinatorial optimization problem. There are three key ideas. The first idea is to define a state space in which all problem instances can be represented and solved. The second idea is to formulate this general search control problem as a reinforcement learning problem with a single reward function that applies across all problem instances. The third idea is to solve this reinforcement learning problem by applying the $TD(\lambda)$ algorithm (with a neural network function approximator).

The rest of the paper is organized in three main parts. The first part describes the methodology. It reviews background information and definitions of combinatorial optimization, resource-constrained scheduling, and reinforcement learning. Then it shows how we formulate the scheduling problem so that it can be solved by reinforcement learning methods.

The second part decribes a series of three experimental studies that were carried out to evaluate and understand the new method and to compare it to the best previous method. The first study is based on a NASA scheduling problem. The second study performs analysis of synthetic scheduling problems to understand the behavior of the learning procedures. The third study performs further analysis of synthetic problems to understand when and why the method works.

Finally, the third part of the paper discusses the results of the experiments and identifies those properties that are important for the success of the methodology. We provide guidelines for applying the methodology to new problems.

2. Background and Methodology

2.1 Combinatorial Optimization and Resource-Constrained Scheduling

A combinatorial optimization problem consists of an *objective function* and a set of *problem instances.* Each problem instance is defined by a set of variables and a set of constraints among those variables. A *candidate solution* to a problem instance is an assignment of values to the variables. A *feasible solution* is a candidate solution that satisfies the set of constraints. An *optimal solution* is a feasible solution that minimizes value of the objective function.

For example, in resource-constrained scheduling, the objective function is to minimize the total length of the schedule (i.e., the difference between the time that the last task is completed and the time the first task was started). A problem instance consists of a set of tasks (or activities) $A = \{a_1, \ldots, a_N\}$ and a set of constraints. Each task a_n has a duration $dur(a_n)$, a set of prerequisite tasks $pre(a_n)$, and a resource requirement vector $\mathbf{r}_n$. The variables whose values are to be determined are the start times of the tasks, denoted $start(a_n)$.

There are K different kinds of resources. The global resource capacity vector $\mathbf{C}$ specifies the number of units of each resource that are available. For the k-th type of resource, $r_{n,k}$ gives the number of units of resource type k that are needed by task n, and C_k gives the number of units of resource type k that are available for all tasks to use. Resources are re-usable.

The constraints are as follows.

- Prerequisites. The start time of a task must come after each of its prerequisite tasks has finished:

$$\forall\, n, a \in pre(a_n) \;\; start(a_n) > start(a) + dur(a)$$

- Resources. The sum of all allocated resources of each type must not exceed the total amount of resources available. Define $active(t)$ to be the set of all tasks active at time t:

$$active(t) = \{a \mid start(a) \leq t < start(a) + dur(a)\}$$

  Then the resource constraints can be written as

$$\forall\, k, t \left(\sum_{a \in active(t)} r_{a,k} \right) \leq C_k$$

In the scheduling problems addressed in this paper, there are two further complications. First, for each type of resource, the total available resources C_k are divided into resource pools $C_{1,k}, \ldots, C_{J_k,k}$ such that $C_{j,k}$ is the number of units of resource type k available in resource pool j. For each type of resource, each task must draw all of its resources from a single pool. Hence, each task has an additional vector $\mathbf{L}_a$ whose element $L_{a,k}$ indicates which pool is used to satisfy the resource requirement for resources of type k. The resource constraint becomes

$$\forall\, k, t, j \left(\sum_{\substack{a \in active(t) \\ L_{a,k} = j}} r_{a,k} \right) \leq C_{j,k}$$

The second complication is that tasks can be structured hierarchically. An abstract task is a task with child subtasks. An abstract task does not have a fixed duration—its duration is the time from when the first of its children is scheduled to the time that the last of its children is completed. Abstract tasks can have prerequisite tasks and resource requirements. If one abstract task A_1 is the prerequisite of another abstract task A_2, then all of A_1's subtasks must complete before any of A_2's subtasks can begin. If an abstract task has resource requirements, then those resources are "in use" for the entire duration of the abstract task.

A final detail concerning our scheduling problems is that each problem instance has one or more "anchor" tasks whose start time is fixed. Each anchor task serves as a deadline for its prerequisite tasks and it serves as a "ready time" (i.e., a lower bound on the start time) for tasks for which it is a prerequisite. In space shuttle scheduling, for example, the anchor task is the launch of the shuttle. (Launch dates are fixed prior to scheduling pre-launch and post-launch activities.)

Combinatorial optimization problems can be solved by state space search. In this paper, we assume that this search is organized as a repair-based search in which each state s is a candidate solution (i.e., it corresponds to a completely defined schedule, with start times for every task). There is a set of operators $\{u_1, \ldots, u_U\}$. Each operator is a function that maps an input state into a result state. We make the further assumption that the start state is an infeasible solution and that every feasible solution can be reached from the start state by some sequence of operators applied to infeasible solutions. This means that we can conduct our search solely by applying operators to infeasible solutions; we can halt the search as soon as we reach a feasible solution. Figure 1 gives a schematic diagram of the structure of the state space. The reason for this restriction will become clear when we discuss the reinforcement learning formulation below.

In our approach to resource-constrained scheduling, we employ the following search space, which is based on the work of Zweben and colleagues (Zweben et al., 1994; Deale, Yvanovich, Schnitzius, Kautz, & et al, 1994). The starting state is a critical path schedule in which every task prior to launch is scheduled as late as possible subject to the tasks for which it is a prerequisite and every task after landing is scheduled as early as possible subject to its prerequisite tasks. Resource constraints are ignored in constructing the critical path schedule. All of our operators maintain the prerequisite constraints while attempting to satisfy the resource constraints. There are two general operators: ReassignPool and Move.

The ReassignPool(a, k) operator changes the pool assignment $L_{a,k}$ for resource requirement k of task a. It is only applied when it allows the resource requirement to be successful satisfied.

The Move(a, d) operator moves task a to a different start time. The argument d specifies the direction of the move. The task is moved to an earlier time if $d = -1$ and to a later time if $d = +1$. The Move(a, d) operator is implemented as follows. First, the *temporal dependents* in the specified direction are "unscheduled"—that is, their assigned resources are released. If task a is being moved earlier, then its temporal dependents are all of its prerequisites and their prerequisites (recursively). If a task a is being move later, then its temporal dependents are all tasks for which it is a prerequisite (and so on, recursively). Once the temporal dependents have released their resources, the schedule is scanned in the indicated direction until a time is found where all of the resource requirements of task a can be satisfied. The start time and pool assignments for task a are updated to reflect this. Then all of the temporal dependents are rescheduled using the critical path method (leaving the resource pool assignments of the dependents unchanged).

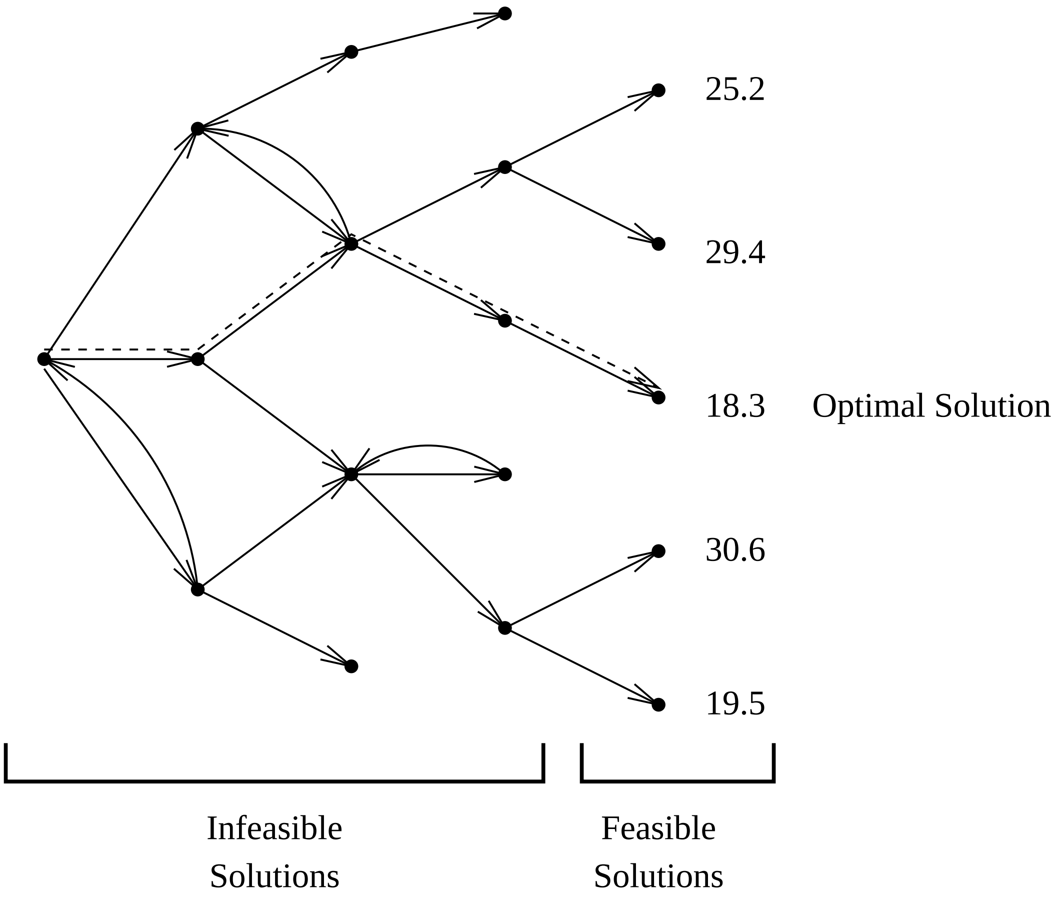

Figure 1: Assumed structure of the state space. Only the terminal states are feasible solutions. The optimal path is shown as a dashed line leading to the optimal solution.

In the starting state, almost every task may be involved in a resource constraint violation. Hence, the branching factor in this state space could be very large. To reduce the branching factor, we again adopt a method developed by Zweben, et al. We consider pool reassignments and moves only for those tasks involved in the earliest constraint violation in the schedule. Specifically, we scan the schedule from the start until we find a constraint violation. At the earliest violation, we consider each resource pool that is over-allocated and consider pool reassignments and moves for all tasks involved in the over-allocation.

This problem space is incomplete—there exist schedules that cannot be reached through any sequence of Move and ReassignPool operators. However, it is not easy to construct examples where this problem arises.

This problem space also contains cycles—that is, sequences of operators that when applied to state s take us back to the same state s. A simple way in which a cycle can arise occurs if there are two tasks a_1 and a_2, where a_1 is a prerequisite of a_2. Suppose that in state s, a_1 is involved in a constraint violation. This is repaired by a Move of a_1 to a later time. Of course a_2 is also moved as a result, because it is a temporal dependent of a_1. This may cause a_2 to be involved in a new constraint violation. If this violation is resolved by moving a_2 back to its original start time, then a_1 may be moved back to its original start time as well, and we have returned to state s.

Now that we have described the resource-constrained scheduling problem and our formulation of it as a state-space search, we turn our attention to defining a learning approach for solving this problem.

2.2 A Learning Approach: Reinforcement Learning and $TD(\lambda)$

To formulate a machine learning approach to combinatorial optimization problems, let us suppose that there is a generator of instances of a given combinatorial optimization problem. For example, in the NASA Space Shuttle program, each new shuttle mission creates a new problem instance—all of the tasks involved in preparing that mission for launch must be scheduled so that they do not conflict with each other or with other missions that are simultaneously being prepared for launch. Although each problem instance is unique, it is reasonable to suppose that there are many similarities between the problem instances, so that search control heuristics might be learned that could solve new problem instances efficiently and nearly optimally.

Abstractly, let us suppose there is an infinite sequence of combinatorial optimization problem instances, $I_1, I_2, \ldots$. In a non-learning approach, each instance is solved separately and independently as it arrives. But in a learning approach, previous problem instances are analyzed to identify a domain-specific search control policy that can then be applied to solve future instances. If new problem instances arrive relatively infrequently, there is plenty of time available to analyze previous problem instances. Hence our main concern is not how much time we spend analyzing previous problem instances, but rather how quickly and optimally we can solve new problem instances.

We will analyze these previous problem instances and learn a control policy by applying reinforcement learning methods. We briefly review reinforcement learning problems and solution methods.

A reinforcement learning problem is a state space search problem with three additions: A probability transition function $P(s'|s,u)$, an immediate reward function $R(s'|s,u)$ and a long-term reward criterion.

The probability transition function $P(s'|s,u)$ gives the probability of entering state s' given that operator u was applied to state s. For the problems we are discussing in this paper, all operators are deterministic, so we will ignore the probability transition function from now on.

The immediate reward is a real-valued quantity that is obtained when operator u is applied in state s to move to state s'.

The long term reward criterion defines the goal of reinforcement learning. In this paper, the long-term reward criterion will be the sum of the immediate rewards until a terminal state is entered:

$$\sum_{t=0}^{T} R(s_{t+1}|s_t,u_t),$$

where T is the total number of steps along a trajectory from the start state to a terminal state, u_t is the operator chosen at step t of the trajectory, and $s_{t+1}=u_t(s_t)$ is the resulting state at step t.

The goal of reinforcement learning algorithms is to discover an optimal policy for choosing operators. A *policy* π is a function that indicates, for each state s, which operator $u=\pi(s)$ should be chosen. An optimal policy, π^*, is a policy that maximizes the long-term reward criterion.

One approach to finding optimal policies is to first compute the optimal value function and then compute the policy from the value function (Barto, Bradtke, & Singh, 1995; Kaelbling, Littman, & Moore, 1996). The *value function* f^π for policy π is a function that tells for each state s what the long term reward will be of executing policy π starting in state s. It can be defined recursively as

$$f^\pi(s) = R(s'|s,\pi(s)) + f^\pi(s'), \tag{1}$$

where s' is the state that results from applying operator $u=\pi(s)$ to state s.

Given a fixed policy, the value function can be computed by solving the system of linear equations defined by Equation (1). This can be accomplished online via the $TD(\lambda)$ algorithm for temporal difference learning (Sutton, 1988). In $TD(\lambda)$, the value function is represented by some form of trainable function approximator, such as a neural network. We will use the notation $f(s,W)$ to denote the value computed by this function approximator in state s. The symbol W denotes a vector containing all of the tunable parameters in the function approximator. The basic step of $TD(\lambda)$ is the following:

$$W := W - \alpha\left(\sum_{i=0}^{\infty}\lambda^i \nabla_{W_{t-i}} f^\pi(s_{t-i},W_{t-i})\right)\Big(f^\pi(s_t,W) - [R(s_{t+1}|u,s_t) + f^\pi(s_{t+1},W)]\Big).$$

The formula involves two parameters: a learning rate α and an eligibility parameter λ ($\lambda \in [0,1]$). The last term on the right is the so-called *temporal difference* between the value predicted by the function approximator at time t and the *backed-up value* $[R(s_{t+1}|u,s_t) + f^\pi(s_{t+1},W)]$ computed at time $t+1$. The purpose of the $TD(\lambda)$ algorithm is to drive this temporal difference to zero. To understand the summation term, consider setting the λ parameter to zero. In this case, the term becomes $\nabla_{W_t} f^{\pi}(s_t, W_t)$, which is the gradient of the value function with respect to the current weight vector. Hence, this rule can be seen as taking a step in the direction of the negative gradient in order to reduce the observed temporal difference. If the temporal difference is driven to zero, then Equation (1) will be satisfied. When $\lambda > 0$, additional gradients (from previous steps $t - i$) contribute to the weight update. This helps propagate information backwards along the trajectory more quickly than when $\lambda = 0$.

For a fixed policy, $TD(\lambda)$ can find an approximation to the value function for states visited by that policy. However, our goal is to find the *optimal value function* (and the optimal policy). A promising approach to finding the optimal value function is to combine $TD(\lambda)$—whose goal is to learn the value function of the current policy—with one-step lookahead search—whose goal is to improve the current policy. This is accomplished by choosing, at each state s, the action that maximizes the backed-up value of the resulting state. In other words, we execute the following policy π_g:

$$\pi_g(s) = \operatorname*{argmax}_u \, [R(s'|s,u) + f(s', W)],$$

where $s' = u(s)$ is the state that results from applying u in s. After each action is chosen, the parameter vector W is updated via $TD(\lambda)$.

Unfortunately, there are many cases where this approach will get caught in a local optimum rather than finding the globally optimal policy. The difficulty is that in order to find the optimal policy, it is necessary to actively search for ways of improving the current policy.

One solution to this difficulty is to introduce random exploration. Let β be a parameter between 0 and 1. Define a new policy π_{rg} such that in state s, with probability β we choose an operator at random from among the possible operators in that state. With probability $1 - \beta$, we choose the best operator based on a one-step lookahead search:

$$\pi_{rg}(s) = \begin{cases} \text{randomly chosen } u & \text{with probability } \beta \\ \operatorname{argmax}_u \, [R(s'|s,u) + f(s', W)] & \text{with probability } 1 - \beta \end{cases} \tag{2}$$

Initially β is set to 1. β is slowly decreased to zero during learning. The hope is that when β reduces to zero, the function approximation $f(s, W)$ will be sufficiently accurate so π_{rg} converges to the optimal policy. The intuition is that the random exploratory moves will discover ways of improving the current policy (by discovering states with better values), and these will then be selected by the one-step greedy lookahead search.

This approach of applying $TD(\lambda)$ for control was very successful in the domain of backgammon, where Tesauro's TD-gammon has become the world's best backgammon program (Tesauro, 1992, 1995). Because of the random dice rolls in backgammon, Tesauro did not need to include any random exploration—TD-gammon always chooses the best operator based on a one-step lookahead search. One motivation for the current research was to determine whether Tesauro's success could be duplicated in an industrial scheduling application.

2.3 Applying $TD(\lambda)$ to Scheduling Problems

To apply $TD(\lambda)$ to resource-constrained scheduling, we must formulate the scheduling problem as a reinforcement learning problem. Furthermore, this formulation must work in such a way that we can discover our search control policy by "training" on a set of scheduling problem instances and then apply our learned control policy to new problem instances.

We begin by defining the immediate reward function $R(s'|s,u)$. There are three requirements that this reward function should satisfy. First, it should give higher rewards to shorter schedules, since our goal is to minimize the schedule length. Second, it should encourage the reinforcement learning system to find efficient search policies—policies that involve few steps. Third, it should be a normalized measure so that a reward of, for example, 10 units, should mean the same relative level of quality for different scheduling problems. Without this property, the function approximator will have difficulty learning a value function that can be transferred to new problem instances.

To satisfy these three requirements, we developed a normalized measure of schedule length called the Resource Dilation Factor (RDF) and defined an immediate reward function as follows:

$$R(s'|s,u) = \begin{cases} -0.001 & \text{if } s' \text{ is an infeasible solution} \\ -RDF(s') & \text{if } s' \text{ is a feasible solution} \end{cases} \tag{3}$$

The small negative reward (-0.001) is intended to impose a small cost for each operator that is applied. This encourages the algorithm to find short paths to a feasible solution. As we will see below, the RDF values lie predominantly in the range 1.1 to 1.4, so a difference of 0.01 between the value of two different operators can be significant. The number of steps from the start state to a terminal state can vary between 20 and 90. The small cost of 0.001 says that an improvement in the RDF of 0.01 is worth 10 extra search steps. This was the right tradeoff point in our judgement.

To define the RDF, our idea was to divide the final schedule length by some quantity related to the difficulty of the scheduling problem. One measure of the difficulty of the scheduling problem is the extent to which resources are over-allocated in the starting critical path schedule. We can capture this through the *total resource utilization index, $TRUI$*, and the *resource utilization index, RUI*.

Definition 1 *Let C_k be the (fixed) capacity of resource type k—that is, the combined capacity of all resource pools of resource type k:*

$$C_k = \sum_j C_{j,k}.$$

At each time t in the schedule, let $util(k,t)$ be the current utilization of resources of type k. If $util(k,t) > C_k$, then the resource of type k is over-allocated at time t (no matter how we assign resource requests to resource pools of this type). The resource utilization index *$RUI(k,t)$ for resource type k at time t is*

$$RUI(k,t) = \max\left\{1, \frac{util(k,t)}{C_k}\right\}. \tag{4}$$

If the resource is not over-allocated, $RUI(k,t)$ is 1; otherwise $RUI(k,t)$ is the ratio of over-allocation.

Definition 2 *The* total resource utilization index $TRUI(s)$ *for a schedule* s *of length* l *is the sum of the resource utilization index taken over all* K *resource types and all* l *times:*

$$TRUI(s) = \sum_{k=1}^{K} \sum_{t=1}^{l} RUI(k,t). \tag{5}$$

The total resource utilization index does not consider the resource pool assignments. It is determined only by the scheduled times of the tasks. This index is useful in producing a unique measure for a schedule no matter how resources are assigned. It is also a good measure to use in the starting state, because initial resource pool assignments are random.

After several moves and pool reassignments have been made, a more accurate measure of resource utilization of a schedule is needed—one that considers each individual resource pool. For this purpose, we defined the *pool utilization index*, PUI.

Definition 3 *Let* $C_{j,k}$ *be the (fixed) capacity of resource pool* j *of resource type* k*. At each time* t *in the schedule, let* $util(j,k,t)$ *be the current utilization of resources of type* k *from pool* j*. If* $util(j,k,t) > C_{j,k}$*, then resource type* k *in resource pool* j *is over-allocated at time* t*. The* pool utilization index $PUI(j,k,t)$ *for resource pool* j *of resource type* k *at time* t *is*

$$PUI(j,k,t) = \max\left\{1, \frac{util(j,k,t)}{C_{i,j}}\right\}. \tag{6}$$

If the resource is not over-allocated, $PUI(j,k,t)$ *is 1; otherwise it is the ratio of over-allocation.*

Definition 4 *The total pool utilization index* $TPUI(s)$ *for a schedule* s *of length* l *is the sum of the pool utilization index taken over all* K *resource types and all* l *times where for each type of resource the average pool utilization index over all pools is computed:*

$$TPUI(s) = \sum_{k=1}^{K} \left[\frac{1}{J_k} \sum_{j=1}^{J_k} \sum_{t=1}^{l} PUI(j,k,t)\right]. \tag{7}$$

Definition 5 *The resource dilation factor (RDF) for a schedule at state* s *is the ratio of total pool utilization index of* s *to total resource utilization index of its initial critical path schedule,* s_0*:*

$$RDF(s) = \frac{TPUI(s)}{TRUI(s_0)}. \tag{8}$$

Figure 2 provides a graphical interpretation of these definitions. It plots the utilization of the resources in a single pool as a function of time. Each rectangular block corresponds to the resource requirements of one task.

In the case where there is only one resource pool and one resource type, $PUI = RUI = TPUI = TRUI$, and the RDF can be visualized as the area enclosed by the heavy black line in the lower diagram divided by the area enclosed by the heavy black line in the upper diagram. The area in the lower diagram is $C_{j,k} \cdot l$, where l is the length of the final schedule. Hence, the RDF can be seen to be a normalized version of the final schedule length (normalized by the level of difficulty of the starting state).

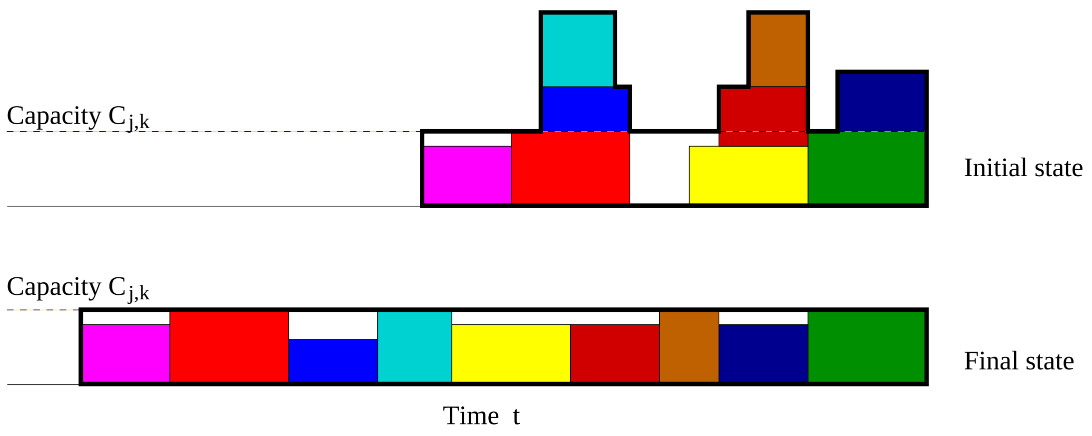

Figure 2: $util(j, k, t)$ for a single resource pool j and a single resource type k at time t. $PUI(j, k, t)$ is the total area enclosed by the heavy black line. The upper diagram shows the $PUI(j, k, t)$ for the starting state. Because of resource conflicts, the utilization $util(j, k, t)$ exceeds the capacity $C_{j,k}$ frequently. The lower diagram shows $PUI(j, k, t)$ for the final state. RUI and $TRUI$ are defined analogously, but summed across all pools of a given type k.

When there are multiple pools and multiple resource types, $TRUI$ is a better measure for the initial state, because initial resource assignments are random, so $PUI(s_0)$ and $TPUI(s_0)$ are artificially elevated. At intermediate states, $PUI(s)$ and $TPUI(s)$ are better measures, because they reflect the actual over-allocations within individual pools. In the final state, $TPUI = TRUI$, so the distinction is irrelevant.

The RDF of a final schedule normally is in the range from 0.8 to 2.0. The RDF can be less than 1 when a small number of MOVE operations permits several REASSIGNPOOL operations to construct a well-packed feasible solution.

Now that we have defined the reward function, let us consider how to define a state representation that can generalize across multiple problem instances.

If we were only going to solve a single problem instance using reinforcement learning, there would be an obvious representation for each state s: We could represent each state by the vector of values of the variables in the problem. For example, in scheduling, we could represent each state by the set of start times $start(a)$ and resource pool assignments $L_{a,k}$. For a single problem instance, there are a fixed number of such variables. This is good, because virtually all function approximation algorithms require that the input be a fixed length vector.

However, because we need to have a representation that works for all problem instances, we must find a different approach. The number of tasks varies from one problem instance to another. In the cases we are studying, the number of resource pools and their capacities do not vary across problem instances.

The approach that we have taken is to define a fixed set of summary statistics describing each state, and use these statistics as inputs to the function approximator $f(s, W)$. We employed the following features in our experiments.

- **Mean and standard deviation of the free pool capacity:** For each resource type and 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.

- **Mean and standard deviation of slacks:** The slack time between a task and one of its prerequisites is the difference between the end time of the prerequisite task and the scheduled start time of the task. We compute 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.

- **Current state RDF:** We found that the RDF of the current state is an important predictor for the final value. We use two RDF features. The first gives the RDF for the current state. The second feature gives 0 for non-terminal (infeasible) states and gives the RDF for terminal (feasible) states.

- **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 that contain a constraint violation n_v. We also compute the percentage of windows that contain a constraint violation n_v/n_{wds}, where n_{wds} is the total number of windows in a schedule.

- **Percentage of windows in violation that can be resolved by pool reassignment n_p/n_v:** 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:** Let ℓ_v be the amount of time in which the schedule contains a violation and ℓ be the total schedule length. This feature is ℓ_v/ℓ.

- **First violated window index:** Let I_{v1} be the index of the earliest window that has a violation. We use the feature $(n_{wds} - I_{v1})/n_{wds}$. If no window has a violation, we set $I_{v1} = n_{wds}$. As violations are repaired, this value decreases to zero. We also consider the 9 windows that immediately follow I_{v1} and compute the percentage of those windows that have violations, $n_{v10}/10$.

- **Features to discriminate instances:** These two features discriminate different scheduling problem instances. One is the resource utilization of the initial critical path schedule $Util_{CPS}$. The other is the average of job due-time separations in the problem instance t_{JDS}. This is the average of the separations between consecutive pairs of anchor tasks.

Each of these features was developed by studying small scheduling problems to find quantities that had some ability to predict final RDF. These features are used as inputs to a neural network whose goal is to predict the value function of the optimal policy for operator selection.

2.4 Summary of Methodology

This completes our description of the methodology. We can summarize it as follows. Given a sequence of previous problem instances $I_1, I_2, \ldots$, we apply the $TD(\lambda)$ algorithm with a mixture of random exploration and one-step greedy lookahead search to solve the problem instances repeatedly and learn a value function (represented as a neural network).

To solve a new problem instance, we employ one-step greedy lookahead search (using the learned value function to evaluate all potential next states) to choose which operator to apply in each state. If the value function has been learned properly, this will result in a very direct search to a nearly-optimal feasible solution.

3. Experimental Studies of the Method

We conducted three experimental studies to evaluate and understand the method.

3.1 Algorithms and Methods

In most of the experiments, we compared the performance of our learning approach with the best previous non-learning method, a procedure developed by Zweben et al. (1994). We begin by describing Zweben's method. Then we explain the details of how our $TD(\lambda)$ method was applied.

3.1.1 Zweben's iterative repair method

Table 1 shows the iterative repair algorithm developed by Zweben and colleagues for solving resource-constrained scheduling problems. This algorithm performs several iterations. In each iteration, it first constructs a critical path schedule. Then, it repeatedly finds and removes constraint violations until a feasible solution is found. If this feasible solution is the best one found so far, it is saved. After i_{max} iterations, the procedure terminates and returns the best solution found.

To repair a constraint violation, the algorithm identifies all tasks that participate in the violation. It then checks whether applying the ReassignPool operator to any one of these tasks could repair the over-allocated resource pool. If so, the operator is applied (if more than one ReassignPool can be applied, one is chosen randomly). If no pool reassignments can repair the constraint violation, the algorithm then considers the Move operator. For each task involved in the constraint violation, the algorithm determines how far the Move operator would move the task in each direction in order to repair the violation. Then it chooses one of the tasks at random based on a heuristic that 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. If both an earlier and a later move are possible, one is chosen at random. The state that results from applying the ReassignPool or Move operator is called s'.

After the new state s' is generated, the iterative repair algorithm must decide whether to accept this repair or consider a different repair. To make this decision, it computes a cost function of the form

$$cost(s) = \sum_{j,k,t} overallocation(j,k,t)$$

where

$$overallocation(j,k,t) = \begin{cases} 0 & \text{if } [util(j,k,t) - C_{j,k}] \leq 0 \\ util(j,k,t) - C_{j,k} & \text{otherwise} \end{cases}$$

This cost function computes the number of "unit-minutes" that resources are over-allocated (summed over all resource types and all pools).

If the cost of the new state s' is less than the cost of the old state s, then the repair is accepted. However, if the cost is greater than the cost of the old state, the repair may still be accepted with probability $exp(-\Delta cost/T)$, where $\Delta cost$ is the increase in the cost and T is the current temperature. Whenever a repair is accepted, the temperature is reduced slightly. (A more general cooling procedure was used in each experiment as explained below).

3.1.2 Training the $TD(\lambda)$ Method

We now describe the details of our $TD(\lambda)$ scheduling procedure. This subsection discusses the network architecture, output representation, and training procedure. The next subsection presents the algorithms for applying the learned value function to solve new problem instances.

To represent the value function, we employed a feedforward neural network architecture with 40 sigmoidal hidden units and 8 sigmoidal output units. The hidden units were fully connected to the input features, and the output units were fully connected to the hidden units. The standard sigmoid activation function

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

was used.

The 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_i $(i = 1, \ldots, 8)$. During training, the target output activation for each output unit is set to be

$$target_j = e^{-\frac{(v_j - RDF)^2}{0.08}}. \tag{9}$$

Table 1: The Iterative Repair Algorithm

```
procedure IterativeRepair
inputs:
    T_0          // starting temperature
    r_cool       // cooling rate
    I            // scheduling problem instance
    i_max        // maximum number of iterations

best := null;  // remember best schedule

for i from 1 to i_max do
    T := T_0;         // set current temperature
    s := CriticalPathSchedule(I);
    while s is not feasible do
        let V be the earliest constraint violation in s
        if a ReassignPool can repair V
            s' := result of ReassignPool
        else select Move probabilistically according to heuristics
            s' := result of Move

        if [cost(s') < cost(s)] or [rand() < exp(-[cost(s') - cost(s)]/T)]
            s := s'
            T := T * r_cool
    end while

    if length(s) < length(best)
        best := s
end for
return best
```

Table 2: Training procedure for $TD(\lambda)$ scheduling

---

procedure TDTrain($\lambda, \alpha, \Delta\beta$)
for each network m **do**
  initialize weights randomly
  $\beta := 1.0$
end for m

repeat
  for each instance I **do**
    for each network m **do**
      $((s_0, 0), (s_1, R_1), \ldots, (s_T, R_T))$:= TDScheduleWithExploration(I, m, β);
      for $t = T$ **downto** 1 **do**
        TDUpdate($m, \lambda, \alpha, s_{t-1}, s_t, R_t$)
      end for t
    end for m
    $\beta := \beta - \Delta\beta$
  end for I
end repeat

procedure TDUpdate($m, \lambda, \alpha, s, s', R$)
repeat
  // compute temporal difference error
  $error := f(s, W_m) - [R + f(s', W_m)]$
  // compute gradient (including eligibility)
  $\mathbf{E} := \nabla_{W_m} f(s, W_m) + \lambda \mathbf{E}$
  // update weights
  $W_m := W_m - \alpha \cdot error \cdot \mathbf{E}$
until $|error| < 0.05$ or 20 updates have been performed

This gives a gaussian-like pattern of target activations, centered on the target RDF value. The variance term, 0.08, was chosen based on preliminary experiments that tested several possible values.

During execution, the predicted RDF value is computed as

$$\widehat{RDF} = (\sum_i act_i \cdot v_i)/(\sum_i act_i), \tag{10}$$

where act_i is the output activation of output unit i. This treats each output activation as an unnormalized probability giving the probability that $RDF = v_i$. The predicted RDF value is the expected value according to this probability distribution.

The training procedure is shown in Table 2. In each experiment, we train between 8 and 12 networks using various combinations of the learning rate α, the exploration schedule

$\Delta\beta$, and the eligibility parameter λ. The exact values of these parameters are described before each experiment below.

Each network m solves each of the problem instances in round robin fashion. The procedure TDScheduleWithExploration(I, m, β) solves problem instance I using network m with an amount of exploration β. It returns the sequence of state-reward pairs $(s_0, 0), (s_1, R_1), \ldots, (s_T, R_t)$ which list each state that was visited and the reward that was received. This sequence is then processed *in reverse order* to update the network via the $TD(\lambda)$ rule by procedure TDUpdate. Each call to this procedure may perform up to 20 steps of gradient descent.

We perform the updates in reverse order, because preliminary experiments showed that this produced substantially better results (Zhang, 1996). We suspect that performing updates in reverse order would improve the performance of other $TD(\lambda)$ applications. Since the goal of temporal difference learning is to propagate information about later states backwards along the state trajectory, it makes sense to update the neural network in this order. Note that this means that the eligibility trace extends forward in time, rather than backward in time as in standard $TD(\lambda)$. This may seem counterintuitive, but it worked better experimentally, and we believe that the eligibility trace helps maintain the temporal coherence of the value function (and hence, of the learned policy).

Table 2 does not show three other aspects of our training procedure. First, for each problem instance I, we keep track of the best scheduling sequence found so far. After every four iterations of the main loop in TDTrain, we replay these "best" training sequences instead of calling TDScheduleWithExploration to construct a new sequence. This technique of experience replay was developed by Lin (1992, 1993) to ensure that the network does not forget good experiences from previous training episodes.

Second, to accelerate the learning process, for the first 20 iterations, we do not generate the scheduling sequences using the neural network or TDScheduleWithExploration. Instead, we implemented a simple greedy scheduling procedure that chooses the repair action resulting in the state with the smallest current RDF (breaking ties randomly). After 20 iterations, we switch to the neural network. Note that until the neural network discovers a better sequence, these initial training sequences will continue to be used by the experience replay procedure.

The third omission from Table 2 is the validation mechanism. Given a collection of problem instances for training, we subdivide them into a training set and a validation set. After every 50 iterations of training, we use each of the networks to solve each of the validation set instances and compute the average final RDF over the validation set for each network. For each network, we save the weight vector that gives the best score on the validation set. The validation set is also used to decide when to stop training. For each network m, training continues until the validation set RDF of that network is worse than its nine previous measured values.

When training terminates, we have final weight vectors for each of the trained networks and the best weight vectors (as measured on the validation set) for each of the trained networks. From these weight vectors, a subset of N are chosen for testing as follows. The $N/2$ networks with the best validation-set RDF values are identified, and their best weight vectors and final weight vectors are selected. These N weight vectors will be used to solve future problem instances. The value of N is specified below in each experimental study.

Table 3: The TD scheduling procedure

---

procedure TDSchedule(I, W)
stack // *a push-down stack, initially empty*
visits // *a hash table for remembering the number of times each state has been visited*
$s :=$ CriticalPathSchedule(I)
$visits(s) := 1$

while s is infeasible **do**
  if $visits(s) > 2$
    $s := pop(stack)$ // *Backtrack*
  else
    push s onto *stack*
    $V :=$ earliest violation in s
    $Op :=$ the set of all operators that could repair V
    apply all operators in Op to s to compute their resulting states
    if $visits(s) = 1$
      $s :=$ result state s' with highest backed-up value $R(s') + f(s', W)$
    else
      $s :=$ result state s' with second highest backed-up value $R(s') + f(s', W)$
  $visits(s) := visits(s) + 1$
end while
return s

3.1.3 Solving problem instances with the learned value function

Once these neural networks have been trained, they must be applied to solve new scheduling problem instances. Table 3 shows the TDSchedule procedure that constructs a schedule for a problem instance I using a trained neural network weight vector W. The basic procedure is the following. In each state s, the procedure finds the earliest constraint violation V and builds a list of all operators Op that could repair V. It then applies each of these operators to s to generate the possible resulting states and computes their estimated values according to the value function. The resulting state with the highest backed-up value is chosen as the next state.

Unfortunately, this simple procedure will not always work, because the state space may contain loops, as discussed above. To detect and avoid loops, this routine maintains a hash table, *visits*, and a pushdown stack, *stack*, for previously visited states. Each time a state is visited, its entry in *visits* is incremented and it is pushed onto the *stack*. If the state has been visited more than once, then we have detected a loop. The second time a state is visited, the second-best successor state is chosen (i.e., the state with the second-highest backed-up value). The third time a state is visited, we backtrack to the previous state (by popping it off the stack) and apply the second-best operator in that state, and so on recursively.

A drawback of using $TD(\lambda)$ for control problems is that at each state we must generate and evaluate all of the possible successor states. In search spaces with large branching factors, this becomes quite costly. An alternative is a procedure we call *random sample greedy search* (RSGS), which generates a random subset of the possible operators and evaluates their resulting states. The best of these operators is then chosen. The size of the random sample is determined incrementally. An initial sample of four operators is chosen. Based on the resulting computed values and a permitted amount of error ϵ and desired confidence $1-\delta$, we can compute the probability that the value of the best sampled operator is within ϵ of the best possible operator. We continue sampling possible operators until this probability exceeds $1-\delta$.

We now discuss how to compute this probability and to determine the sample size during the incremental sampling process. Suppose that $U = \{u_1, u_2, \ldots, u_k\}$ is the current sample of operators and $V = \{v_1, v_2 \ldots, v_k\}$ gives the corresponding values of these operators according to the value function. Let $\hat{v}_{max}$ be the largest value in V; we can regard it as an estimate of the true maximum value v_{max}. Our goal is to find an operator whose value is greater than or equal to $v_\epsilon = v_{max}(1-\epsilon)$ with probability $1-\delta$. To do this, we will first estimate the probability θ that a randomly chosen operator has a value greater than or equal to v_ϵ. Then we can determine the sample size necessary to ensure that at least one operator with value greater than or equal to v_ϵ has been generated.

To estimate θ, let us assume that the current observed operator sample has the same distribution of values as the complete operator sample. Under this assumption, we can estimate the value of the best possible operator within permitted error ϵ using the current sample, which is

$$\hat{v}_\epsilon = \hat{v}_{max}(1-\epsilon). \tag{11}$$

We then compute the fraction $\hat{\theta}$ of operators with values greater than or equal to $\hat{v}_\epsilon$:

$$\hat{\theta} = \frac{|V_\epsilon|}{|V|}, \tag{12}$$

where $V_\epsilon = \{v_i \in V | v_i \geq \hat{v}_\epsilon\}$ is the subset of V that contains elements greater than or equal to $\hat{v}_\epsilon$. Thus, the probability that the value of a single operator in the current operator sample is larger than or equal to $\hat{v}_\epsilon$ is $\hat{\theta}$. Let us assume that $\hat{\theta} = \theta$.

Given this estimate for θ, we can now ask what is the probability that in a sample of size m, *none* of the operators has value greater than v_ϵ. This is $(1-\theta)^m$. We want this quantity to be less than δ. Solving for m, we obtain

$$m \geq \left\lceil \frac{\ln \delta}{\ln(1-\theta)} \right\rceil. \tag{13}$$

We apply this formula as follows. We begin by generating a random sample of four possible operators. From this sample we compute initial estimates for θ and m. Then, as long as the estimated m is greater than the current sample size k, we repeatedly generate one new operator, evaluate it, and recompute estimates for θ and m. In the cases where we employed RSGS, we set $\epsilon = 0.1$ and $\delta = 0.1$.

It is reasonable to ask whether this procedure will work well, particularly starting with a sample of size 4. In our experiments, we found that in some cases, it ends up generating all possible successor states, and therefore does not provide any savings. However, in other cases, it can be very beneficial. Consider, for example, a situation in which there are 20 possible operators, 10 of them have a single value and the other 10 have another, single higher value. In this case, RSGS chooses one of the higher-valued operators after evaluating only a small fraction of the 20 possible operators. Situations like this are quite common in our scheduling problems.

In some cases, of course, RSGS stops too soon—without finding a high-value operator. The consequences of this are not as severe as one might fear. Given that we are only approximating the value function, it may well be that the operator with the best predicted value is not in fact the best operator, so even an exhaustive search would not necessarily produce the best result.

3.1.4 Comparing IR and TD scheduling as anytime algorithms

As with any optimization problem, there is a tradeoff between the amount of computer time expended searching and the quality of the solution. Exhaustive search can find optimal solutions, but the CPU time required is enormous.

The fairest way to compare two optimization algorithms is to compare the quality of the solutions that they find as a function of the amount of CPU time expended. This is most convenient for *anytime algorithms.* An anytime algorithm is an algorithm having two properties. First, it can be interrupted at any time, and it will provide a candidate solution. Second, the quality of this solution improves as the algorithm is given more CPU time to run.

The IR scheduling algorithm is an anytime algorithm—it keeps track of the best solution found so far, and this solution improves as IR continues to run.

The TD scheduling algorithm, on the other hand, is not an anytime algorithm. With the exception of random tie breaking and RSGS, it is deterministic. This means that given a weight vector and a problem instance, it will find nearly the same solution each time it is executed. To approximate an anytime algorithm, we can use multiple weight vectors. Specifically, we execute the TD scheduling algorithm once with each of the six weight vectors selected from the 16 generated during training. The weight vectors are considered in decreasing quality order, according to their performance on the problem instances in the validation set. We keep track of the best solution found by any of these 6 networks.

Given a set of test problem instances, we run both the IR and TD algorithms to generate one candidate solution for each problem instance. We record the total number of search steps expended and the mean RDF of the candidate solutions. We can plot this point on an anytime graph that shows the quality of the solutions as a function of time. Once we have this initial point, we then execute the IR and TD algorithms on each test instance again (with TD using the second chosen weight vector). Each time an algorithm finds an improved solution to one of the test instances, we can plot a new point whose x coordinate is the total number of search steps so far and whose y coordinate is the mean RDF of the best candidate solutions found so far for the test instances. In this way, we sweep out a curve of points showing how the mean RDF on the test set improves as a function of the number of search steps performed.

This completes the description of the IR and TD scheduling algorithms and our experimental methodology. Now we describe the three experimental studies that we performed.

3.2 Study 1

The purpose of this study was to compare the IR and TD algorithms on the NASA Space Shuttle Payload Processing (SSPP) problem. This is a resource-constrained scheduling problem that involves 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. The payloads are typically scientific experiments or satellites. They are assembled and then placed in various kinds of carriers. The carriers are then placed into the shuttle cargo bay. The specifications for the SSPP problem were provided to us by Todd Stock of NASA's Ames Research Center. The scheduling problem instances that we generated from these specifications are available in an online appendix to this paper.

3.2.1 The Data Set

We considered three types of carrier: long module (LM), mission peculiar equipment support structure (MPESS), and pallet and igloo (PI). The carriers used for any particular mission determine which tasks need to be performed for that mission. The LM requires 62 tasks, the MPESS, 32 tasks, and the PI, 82 tasks. A mission may involve up to two of these carriers, and hence, between 32 and 164 tasks. These tasks include several abstract tasks, structured hierarchically.

There are 35 different types of resources of which only five are major bottleneck resources: Electrical Engineer, Logistics, Mechanical Engineer, Quality Control Officer, and Technician.

Each instance of an SSPP involves the simultaneous scheduling of between two and six shuttle missions whose launch dates have been determined in advance. Each mission therefore has an anchor task, `FLY-MISSION`, whose start time (the launch date) is fixed.

Note that because there are only three kinds of carriers, there are only 9 different possible carrier configurations in a shuttle mission, and hence, only 9 possible missions in this data set. However, there are effectively a huge number of SSPP problem instances, because each instance can contain up to 6 missions, and the amount of time separating the launch dates can vary over the range between 30 to 90 days.

We randomly generated 20 training instances (each containing 2 to 4 missions) and 20 test instances (each containing 3 to 6 missions). Of the 20 training instances, 15 were presented to the $TD(\lambda)$ algorithm for training, and 5 were used as the validation set to determine when to stop training and to choose weight vectors for testing.

3.2.2 Algorithm specifications

For the iterative repair algorithm, the initial temperature was set to 200.0. After every 10 accepted repairs, it was decreased according to $T := 0.95T$.

For the neural networks trained with $TD(\lambda)$, twenty input features were provided. These included all of the features described in Section 2.3 except for the percentage of windows in violation that can be resolved by pool reassignment and the extra features for discriminating problem instances. This gave us 20 features: 10 features for pool capacity (describing the

5 bottleneck resource types), 4 slack features, the RDF of the current schedule, the over-allocation index, 2 features describing windows in violation, the percentage of time units in violation, and the index of the first violated window.

We trained eight different networks using all combinations of the following parameters: learning rate $\alpha = 0.1$ or 0.05, exploration schedule $\Delta\beta = 0.001$ or 0.0005, and eligibility parameter $\lambda = 0.2$ or 0.7. The weights of the neural network were initialized to small random values in the range $[-0.05, 0.05]$. The target RDF values for the 8 network output units were 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, and 1.6.

During training and testing, we employed random sample greedy search to choose operators. We did not use teaching sequences in this study.

When training terminates, six of the 16 available weight vectors are chosen using the validation set. These six weight vectors are used to solve future problem instances.

3.2.3 Results

Figure 3 shows the anytime performance of the IR and TD algorithms on the 20 SSPP test instances. The vertical axis plots the mean RDF of the 20 instances and the horizontal axis shows the CPU time on a logarithmic scale. Given the same amount of CPU time, the TD scheduler finds higher-quality schedules. Alternatively, if our goal is to find a schedule of a given quality, then the TD scheduler will find it less than half the time of the IR scheduler.

Because Figure 3 reports only the mean RDF, it hides considerable variation. Figure 4 provides some insight into this variation. Let us say that TD "wins" on a particular instance if the RDF of its best schedule computed so far is better than the RDF of the best IR schedule computed with the same amount of CPU time. The two algorithms will be said to "tie" if they find schedules with identical RDF values. Figure 4 plots the fraction of TD "wins" and TD "wins + ties" as a function of log CPU time. We see that at low CPU costs, TD wins on almost every problem instance. Eventually, as the available CPU time is increased, TD still wins or ties slightly more than 50% of the time. This is true even after 37,114 seconds, which is the point at which the TD algorithm had used all 6 of its learned neural networks. Of course, if IR is permitted to run long enough, it will eventually generate all possible operator sequences and hence find the optimal solution, but this is not of practical value.

The results in this study have been replicated using synthetic scheduling problems (Zhang & Dietterich, 1995). That study also showed that the learned TD scheduler worked very well on problem instances that were larger than the ones on which it had been trained.

From this first study, we conclude that the TD method shows considerable promise as an alternative to the existing IR method.

3.3 Study 2

The second study was aimed at understanding the behavior of $TD(\lambda)$ during training. The design of the experiment was to train several networks on a set of synthetic scheduling problems. During training, we computed several different measures of the networks and observed how these measures changed during training. We chose the following measures to observe. The first measure is computed over the training examples. The remaining four measures are computed over the validation examples.

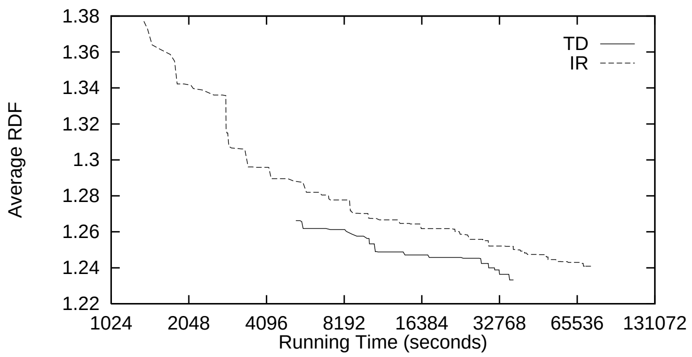

Figure 3: Mean RDF for the TD to IR scheduling algorithms as a function to CPU time for 20 SSPP Problems.

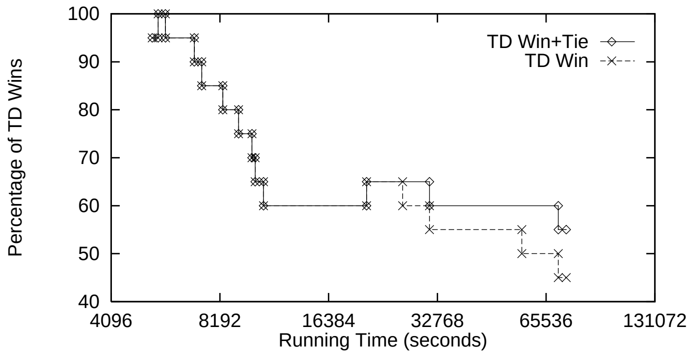

Figure 4: Performance Comparison of TD to IR on 20 SSPP Problems (% Wins).

1. **Training set average squared temporal difference error.** The temporal difference error is the difference between the predicted value of the current state and the backed-up value computed from the next state and reward:

$$TDError_t = \Big(f^\pi(s_t, W) - [R(s_{t+1}|u, s_t) + f^\pi(s_{t+1}, W)]\Big).$$

The squared TD error is the quantity that backpropagation is attempting to minimize. The average squared TD error is the average taken over all steps t along each training sequence of each training instance. Because training involves exploration of the search space, this can be very noisy, so we compute a moving average of this quantity over the last 1000 iterations.

2. **Validation set average RDF.** This is the same measure used in the previous study. It computes the average over the validation set instances of the final schedule RDF.

3. **Average number of repairs needed to solve each validation instance.**

4. **Validation set TDError (2 measures)**. As the TD scheduler solves each validation set instance, we can measure the temporal difference error of each state transition. We noticed that the largest TD errors were measured at the last state transition (from the last infeasible solution to the final feasible solution). Hence, we separated this TD error out to yield two measures:

`Is this the absolute value of the TD error or the squared TD Error? The caption on Figure 8 says MSE, but the discussion of Figure 9 suggests that it is the absolute value of the error.`

Average final step TD Error. This is the average over all validation set instances of the squared TD error of the final step in the solution sequence.

Average non-final-step TD Error. This is the average over all validation set instances of the squared TD error over all steps along the solution sequence except the final step.

We wanted to use a larger set of more difficult scheduling problem instances for this experiment, so we designed a synthetic data set called ART-2.

3.3.1 Data Set ART-2

The ART-2 data set was generated as follows. First, we generated a pool of 20 jobs. To generate a synthetic job, we chose the number of tasks randomly in the range 8 to 15. A set of temporal constraints among these tasks was then randomly generated such that approximately 60% of all possible pairwise precedence constraints were asserted. Note that there are no abstract tasks in this domain, so the problem is not hierarchical.

Next, resource requirements were determined for each task. There were two types of resources. Each resource has two pools—one pool with a capacity of 6 units, and the other with a capacity of 8 units. Resource requirements were randomly assigned to each task uniformly in the range from 0 to 6 units for each resource type.

Once the pool of 20 jobs was generated in this way, 30 training instances, 20 validation instances, and 50 test instances were constructed. To generate a problem instance, we first chose the number of jobs in the instance to be either 3 or 4 (with equal probability). The desired number of jobs was selected randomly with replacement from the 20-job pool. Each job was assigned a completion deadline with the deadlines randomly separated by between 4 and 7 time units.

3.3.2 Algorithm specifications

For the neural networks, the input features included all of the features defined in Section 2.3. This gave a total of 22 features: 8 pool capacity features for the 4 pools, 4 slack features, and features describing the RDF, percentage of windows and time units in violation, and percentage of violated windows in which the violation can be resolved by pool reassignment, the over-allocation index, the first violated window index, and the features to discriminate problem instances.

We employed the training procedure described earlier, including the procedure for generating teaching sequences using a greedy scheduler based on current-state RDF. Because the branching factor was lower in these instances than in the the SSPP task, we employed full greedy lookahead search rather than random sample greedy search.

The learning rate α was determined dynamically by setting it equal to one third of the initial temporal difference error in each state. The exploration reduction rate $\Delta\beta$ was set to 0.0005 and the eligibility parameter λ was set to 0.2. We trained four different networks using four training-set settings: the first 10 instances (denoted P10a), the second 10 instances (P10b), the last 10 instances (P10c), or all 30 instances (P30). We trained the networks for 5000 epochs (training sequences).

The weights in the network were initialized to random values in the range $[-0.05, 0.05]$. The target RDF values for the 8 network output units were 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0, and 2.2.

3.3.3 Results

Figure 5 shows the (smoothed) average TD Error error for the four training problem settings. The horizontal axis gives the number of training sequences processed. As one would expect, the larger the training set, the harder it is for backpropagation to reduce the TD Error. Two of networks trained on 10-instance training sets (P10b and P10c) converged more quickly than P30 or P10a.

Figure 6 plots the mean RDF over the 20 validation set instances as a function of the number of sequences trained. We can see that all four training problem settings show substantial improvement in the RDF, which demonstrates that the training procedure is learning and generalizing well. The curve for P30 is slightly lower than the curves for the networks trained on only 10 instances, which shows that there is some benefit from having a larger training set.

Figure 7 plots the average number of repair actions taken to produce a final schedule. This shows that the number of repairs drops from an initial value of more than 180 to values in the range from 30 to 50. This drop occurs within the first 2000 epochs of training. Beyond that point, there is very little improvement in this measure, although the RDF appears to continue to decrease.

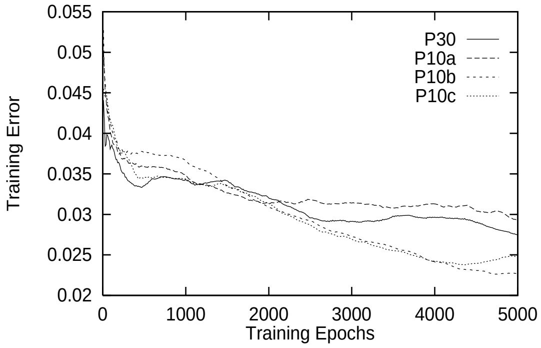

Figure 5: Squared TD Error on 30 Training Problems.

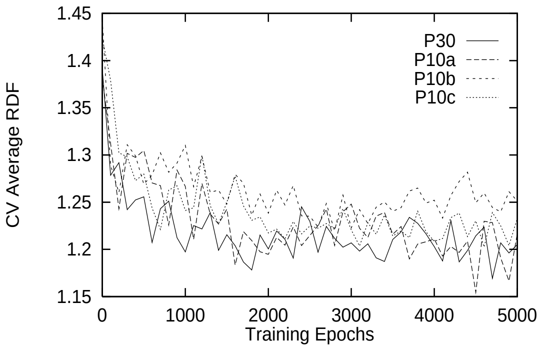

Figure 6: Average RDF over 20 Validation Problems.

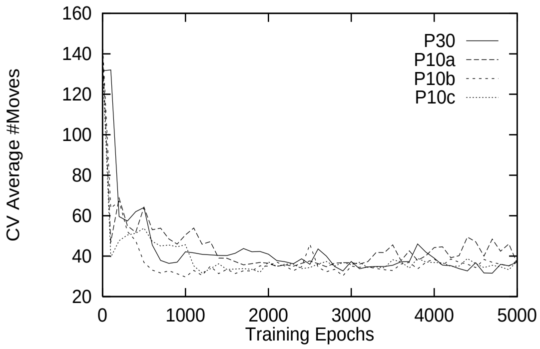

Figure 7: Average Number of Repairs over 20 Validation Problems.

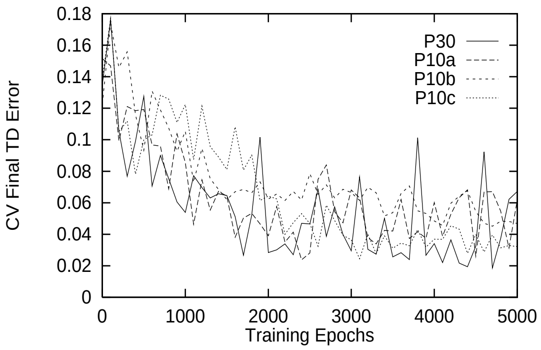

Figure 8: Average Squared Final Step TD Error on 20 Validation Problems.

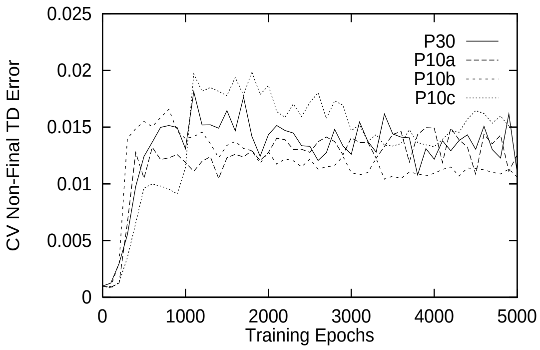

Figure 9: Average Squared Non-Final Step TD Error on 20 Validation Problems.

Figure 8 shows that the accuracy of final-step predictions improves with training. We should mention that to obtain this performance improvement, it was essential to include the special "final state RDF" feature described earlier. (This input feature is zero in all non-final states and gives the final state RDF in the final state). Figure 9 plots the non-final state TD errors. These start out very small, because all the weights in the neural networks are very small so the network predictions are nearly constant. Consequently, the TD error is $|R(s, a, s')|$, which is 0.001. With training, the network begins to distinguish different states, and the TD Errors increase rapidly. After 1000 epochs of training, the network begins to reduce this error slowly.

This study shows that the $TD(\lambda)$ procedure trains the networks fairly well. The number of repairs required for scheduling is reduced very rapidly, while the final state RDF requires a larger amount of training to be learned well. The TD error graphs show that the value function is not completely learned. Both the final-step TD error and the non-final-step TD error are quite far from zero. The final-step TD error is quite large.

3.4 Study 3

The results of these first two studies show that $TD(\lambda)$ can learn a very good scheduling policy. The first study showed that the learned policy can work better than the non-learning policy of the iterative repair algorithm. The second study showed that training reduces several different error measures. However, these experiments do not provide much insight into *why* TD is doing better. The purpose of our third experiment was to answer this question.

Based on the first two studies, and many other experiments, we developed the following three hypotheses that could explain why TD scheduling outperformed IR scheduling.

- **Hypothesis 1: Feature Copying**. The first hypothesis is that the input features we provided were so good that the neural network simply needed to copy one of them (or some simple combination of them) to the output. According to this hypothesis, $TD(\lambda)$ is not propagating information backward along solution paths, and the theory of reinforcement learning as a form of online dynamic programming (Barto et al., 1995) is irrelevant to the success of TD scheduling.

- **Hypothesis 2: Feature Smoothing**. The second hypothesis is that the network is smoothing the input features to remove local minima, but the network is not learning to predict the final RDF. Why would feature smoothing give good performance? The argument goes as follows. The reason iterative repair must employ simulated annealing is because its cost function ($cost(s)$, the 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 the $cost$ function gives very poor results.) If these local minima could 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 temporal smoothing operation applied to the network's input features. These input features include a number of features closely related to the number of constraint violations. So perhaps dynamic programming plays little role in the success of our TD scheduler.

- **Hypothesis 3: Dynamic Programming.** The third hypothesis is that the network is actually learning to predict final state RDF and that the view of $TD(\lambda)$ as a form of online dynamic programming is correct.

To test these experiments, we performed several small experiments on the ART-2 data set.

3.4.1 Algorithm specifications

For the iterative repair algorithm, the initial temperature was set to 100.0. After every 10 accepted repairs, it was decreased according to $T := 0.95T$.

The neural nets were trained in the same way as in Study 2. We trained on 30 instances and validated on 20 instances. Eight different networks were trained using all combinations of two λ settings (0.2 and 0.7) and four training-set settings: the first 10 instances, the second 10 instances, the last 10 instances, or all 30 instances. Each network was trained for 5000 passes through the training instances. The best 6 networks, based on validation set performance, were identified, and 2 weight vectors were saved from each network: the best weight vector (as determined by the validation set) and the final weight vector. This provided a total of 12 weight vectors for testing. We used the disjoint set of 50 test instances for final testing.

3.4.2 Results

The first thing we did was to examine the quality of the intermediate predictions made by the learned networks. Specifically, for each state along a repair path, we computed the current state RDF and the network output. Figure 10 shows an example of this analysis. Here we see that the current state RDF is not a very good predictor of final RDF, whereas the network output is much better. The jumpy behavior of the current state RDF can be explained as follows. Each jump corresponds to a Move operator that has lengthened the schedule. The curve then drops as the result of moves and pool reassignments that remove resource violations without increasing the length of the schedule. At some point, this no longer becomes possible, so another Move causes the current state RDF to increase again. In contrast, the network prediction is much more level, although it is over-predicting the final RDF (i.e., the current state RDF at position 46). We observe this in most cases: the predicted RDF is much flatter (and therefore a better estimator) than the current state RDF, but it typically has a systematic prediction error. It usually slightly overestimates the RDF for a good final solution and slightly underestimates the RDF for a bad solution. This error decreases smoothly toward the end of the sequence. This provides some evidence that the learned network is not merely copying its best features to the output. Hence, it is evidence against Hypothesis 1.

The second experiment that we performed was to compare a network trained with no final reward to a network trained in the standard way with a final reward. We trained a network on a set of 10 problem instances and tested the network on the same instances. Figures 11 and 12 compare the performance of these two networks. If either Hypothesis 1 or

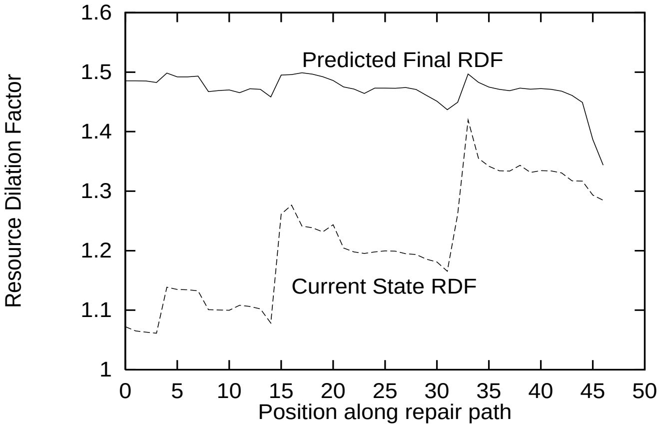

Figure 10: Current-state RDF and predicted final RDF (computed by the learned value function) as a function of the position along a TD scheduling path for a single problem instance.

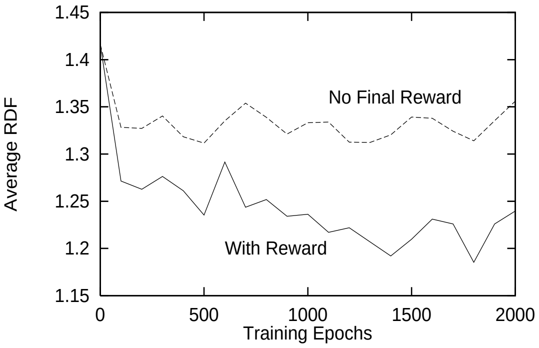

Figure 11: Average RDF on 10 training instances for TD scheduling when trained with and without final rewards.

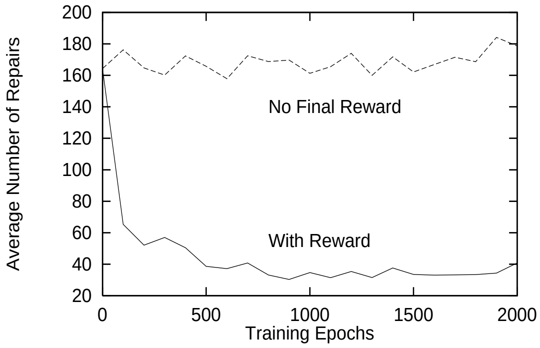

Figure 12: Average number of repairs on 10 training instances for TD scheduling when trained with and without final rewards.

Hypothesis 2 is correct, then there should be little or no difference between the two networks. Even without the final reward, the $TD(\lambda)$ procedure will still smooth the predictions along the path from the start state to the final state. However, the figures show that the average RDF is decreased only slightly (from 1.41 to 1.35) when trained without the final reward, whereas it decreases much more (from 1.41 to 1.25) when trained with the final reward. We see an even more dramatic effect if we look at the number of repairs. Without a final reward, the number of repairs is essentially unchanged at around 175, whereas with a final reward, the number of repairs drops to around 40.

This experiment provides strong evidence that Hypotheses 1 and 2 are false.

We performed one more experiment with the goal of determining whether the success of TD scheduling is primarily the result of making good decisions near the beginning of the scheduling sequence, near the end, or perhaps throughout the process. If TD scheduling is only making good decisions near the end of the scheduling sequence, then this would be evidence against Hypothesis 3. To the extent that TD scheduling is making good decisions earlier in the scheduling sequence, this is evidence in favor of Hypothesis 3, because it shows that information about the final RDF is being propagated backward to earlier points in the solution sequence.

The key idea of the experiment was to replace one quarter of the IR policy with a TD learned policy (and vice versa). Specifically, suppose that a particular scheduling instance requires n repairs to remove all violations. In this experiment, we construct four new scheduling policies. The first policy will apply the TD policy for $n/4$ repairs and then apply the IR scheduler for the remainder of the repairs. The second policy will apply the IR scheduler for $n/4$ repairs, then the TD scheduler for the next $n/4$ repairs, and finally the IR policy for the remainder of the repairs. Similarly, the third policy will use the TD scheduler for the third quarter of the repairs, and the final policy uses the TD scheduler for the last quarter of the repairs.

We also performed the opposite experiment in which we use the IR scheduler to make one fourth of the repairs and otherwise employ the TD scheduler to make repairs. Again, we predict that the damage inflicted by the IR scheduler will be essentially the same regardless of which quarter of the repairs it performs.

The interpretation of this experiment is based on the assumption that the effect of problem-solving decisions is cumulative. Both the TD and IR schedulers scan along the schedule making repairs from earliest to latest. These repairs are rarely clobbered by subsequent operators. Hence, if a scheduler finds a good way of "packing" tasks together early in the scheduling process, this good configuration of tasks will still be present in the final schedule and it will be reflected in the final RDF. Conversely, if a scheduler constructs a bad configuration of tasks early in the process, this bad configuration will also survive to the final schedule, and the final RDF will be bad.

According to this analysis, if we make good scheduling decisions in any quarter of the scheduling process, this should be detectable in the final RDF. So if the TD scheduler has learned to make good decisions during the first and second quarters of the scheduling process, we should be able to observe it.

Our TD scheduler used the best of the 12 schedules constructed by applying each of the 12 selected neural networks. Our IR scheduler used the best of 50 schedules constructed by performing 50 restarts of the IR procedure. In place of the *cost* function used in all previous experiments, we employed the current-state RDF to make the IR and TD procedures more similar. To estimate the expected number of repairs for a problem instance, we multiplied the number of tasks by 1.6 for the TD scheduler and by 2.0 for the IR scheduler (IR requires more repairs than TD). This number was divided by four to give the number of repairs that each scheduling procedure was permitted to make in the first three quarters. In the final quarter, the selected scheduling procedure runs until a feasible schedule is found.

Figure 13 plots that the average RDF of the solutions over all iterations and all test problems. The vertical error bars show 95% confidence intervals based on the t statistic. TD1Q, TD2Q, TD3Q, and TD4Q denote the four procedures where the TD policy is used

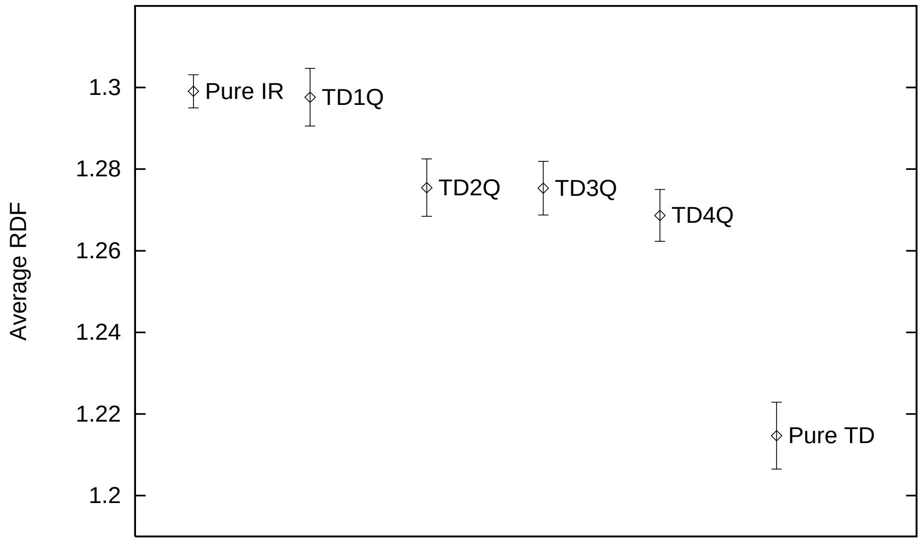

Figure 13: Effect of the TD policy inserted in different stages of the IR scheduling procedure.

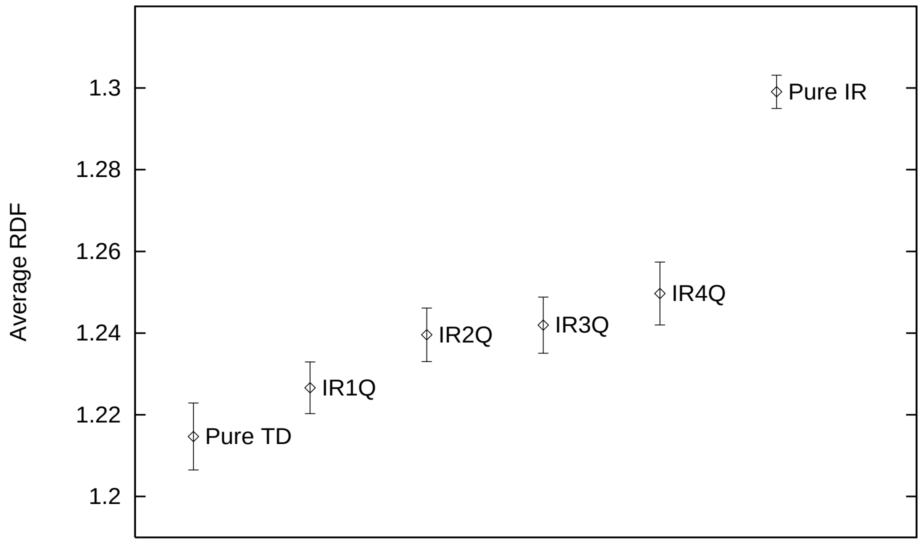

Figure 14: Effect of the IR policy inserted in different stages of the TD scheduling procedure.

in the first, second, third, and final quarters of the IR procedure (respectively). We see that TD2Q, TD3Q, and TD4Q all improve solution quality significantly. They clearly perform better than Pure IR but worse than Pure TD. This shows that TD learning has captured a good policy for decisions during these quarters, and therefore provides support for Hypothesis 3. However, the TD1Q procedure provides only a very slight improvement over Pure IR. Under the assumption of cumulative decision making, this suggests that $TD(\lambda)$ has not succeeded in learning a good policy for the first quarter.

Figure 14 shows the results of inserting the IR policy into the TD policy. This measures how badly the TD scheduler is affected by bad decisions introduced by the IR policy. The results are basically the mirror image of Figure 13. IR2Q, IR3Q, and IR4Q all introduce statistically identical damage into the TD policy, which shows that the quality of the TD decisions is better than the IR policy in each of these quarters. Even in IR1Q, there is weak evidence that IR1Q is worse than pure TD, although the confidence intervals overlap.

This experiment shows that the policy learned by $TD(\lambda)$ is clearly better than the IR policy in quarters 2, 3, and 4 of the solution sequence, and probably slightly better in the first quarter as well. This provides important evidence in favor of Hypothesis 3 and against Hypothesis 2.

Based on these experiments, we conclude that Hypothesis 3 is correct—the success of TD scheduling can be accounted for through the use of good input features and the application of $TD(\lambda)$ to train the neural network to estimate the value function.

4. Discussion

These experiments raise two important issues. First, why does the combination of $TD(\lambda)$ and random exploration succeed in the scheduling domain? Second, when can we expect this general methodology to work in other domains?

We have identified several properties that are important for the success of this methodology. They can be divided into three groups: (a) properties affecting the ability of $TD(\lambda)$ to learn a near-optimal policy for individual problem instances, (b) properties affecting the ability of $TD(\lambda)$ to generalize to new instances, and (c) properties affecting the efficiency of applying the learned policy. We discuss each of these in the following subsections.

4.1 Properties important for $TD(\lambda)$ learning

We believe three properties must hold in order for $TD(\lambda)$ plus random exploration to learn a good policy in a feasible amount of time. First, initial exploration in the domain must reach a terminal state in a reasonable amount of time. Second, at each step during learning, exploration must be able to discover improvements in the policy in reasonable amounts of time. Third, those improvements must be detectable, so that $TD(\lambda)$ can remember them. If these three properties are satisfied, then $TD(\lambda)$ (with exploration) will be able to discover an initial policy and incrementally make improvements to that policy until it becomes nearly optimal. We now elaborate on each of these points.

The first requirement, that initial exploration in the domain must quickly lead to a reward, is satisfied in any problem space that is closed and acyclic. Domains that contain many cycles or that are unbounded can cause randomized exploration methods to take exponential or even infinite expected time to reach a terminal state (Koenig & Simmons, 1996). Our scheduling problem space is closed and nearly acyclic: most repairs do not undo previous repairs. Furthermore, our cycle detection mechanism removes cycles when they do occur (which is around 4% of the time), so this makes the problem space completely acyclic. Backgammon is also approximately acyclic. Most moves advance pieces toward the end of the board (although hitting an opponent's piece sends it back to the start). Tesauro (1992) reports that random policies still reach the end of the game in around a thousand moves. Once an initial reward is attained, repeated trials with $TD(\lambda)$ can learn an initial policy that will reproduce the same trajectory.

The second requirement, that exploration can discover improvements to the policy in reasonable amounts of time, rules out domains in which good policies are very different from one another. In such domains, any significant improvement in a policy π requires changing the recommended operator $u = \pi(s)$ for many states s. Exploration is unlikely to discover such large, coordinated, policy changes. Exploration can easily discover improvements if every policy has many better policies that can be created by making a small number of changes. This is true for problem spaces that are composed of nearly-independent subproblems. Changes to any one subproblem can improve the overall policy without requiring coordinated changes in the other subproblems. For our repair-based scheduling problem space, it is reasonable to believe that the way tasks are scheduled in one part of the schedule is largely independent of the way tasks are scheduled in other parts of the schedule. The policy-switching experiments provide evidence for this by showing that bad decisions inserted in the scheduling process caused the same amount of damage regardless of where they were inserted (except possibly in the first quarter of the schedule), and good decisions inserted into a bad scheduling process produced benefits regardless of where they were inserted. It is also reasonable to believe that backgammon enjoys a similar property: the optimal decisions to make late in the game are largely independent of the optimal decisions decisions to make early in the game.

The third requirement, that improvements in the policy be detectable, is a statement about the relative size of the improvements in the policy compared to the amount of noise in the domain. Our problem space is deterministic, so the only source of noise is the random exploration process. If an improvement to the policy is made early in the scheduling sequence, random exploration later in the sequence may make it difficult to detect the improvement. There is a fundamental problem here: some degree of exploration is required to discover policy improvements, but exploration can also hide the effect of those improvements on the final RDF. The $TD(\lambda)$ methodology will only succeed if the size of the improvements is large enough that they can be detected over the noise introduced by further exploration.

These three requirements provide some guidance for attacking new combinatorial optimization problems. The first requirement suggests that it is important to formulate problem spaces so that they are acyclic. When cycles cannot be completely eliminated in the formulation of the problem space, they should be detected and removed during learning and problem solving. The other two requirements suggest that it is important to identify and preserve independence between subproblems within the domain. Formulations that introduce dependencies should be avoided.

4.2 Properties important for generalization

The three properties listed above are important even when learning to solve individual problem instances. There are three more properties that are important for generalizing across problem instances. First, all states of all problem instances must be represented using a common set of features. Second, the features and the rewards must be normalized. Third, the function approximation method must generalize "appropriately."

We have discussed the first property earlier in the paper. Without a common set of features, the value function learned while solving one set of instances cannot be applied to solve new instances.

The second property is important for the success of generalization. A reward of R points should mean the same relative degree of solution quality across problem instances. The input features should describe each problem instance in a way that makes it easy to predict the final reward. For example, the summary statistics that we used in scheduling (e.g., mean and standard deviation of the free capacity of the resource pools) have the same meaning regardless of the length of the schedule. The RDF was explicitly constructed to be normalized for the degree of difficulty of the problem instance.

The third property—that the function approximator generalize appropriately—is a property shared by all machine learning applications. It is also a property that is impossible to guarantee. Nonetheless, by testing function approximators across many domains, researchers can discover which function approximators appear to work well for reinforcement learning applications.

4.3 Properties important for efficient execution

There are two properties that are critical to the efficient execution of the learned policy in the $TD(\lambda)$ methodology. First, the branching factor in the problem space must be small. This is because each execution step requires constructing and evaluating the possible successor states (or at least a large sample of them, using random sample greedy search). When formulating a problem space for $TD(\lambda)$, it is therefore important to keep the branching factor as small as possible.

There is an alternative approach that can work efficiently with much larger branching factors: Q learning (Watkins & Dayan, 1992). In the Q learning method, the function approximator learns a value function $Q(s, u)$, that is a function of the state s and the operator u. To select the best operator to apply, we can then compute $\pi(s) = \text{argmax}_u Q(s, u)$ without generating any successor states. Unfortunately, this approach only works if the operators can be represented in a sensible way. In many domains, each operator has a well-defined identity and semantics that is largely independent of the state to which it is applied. For example, moving "north" in a grid world has the same basic effect (of moving one step north) regardless of the current state.

However, in the repair-based scheduling problem space, the effect of the ReassignPool and Move operators (and even the number of possible operators) depends intricately on the current state, so a Q learning approach is not possible.

The second property important for efficient execution is that the features should be very efficient to compute. We performed a study in which a time-delay neural network was used to learn good features for resource-constrained scheduling (Zhang & Dietterich, 1996) using an extension of the $TD(\lambda)$ method. We found that these learned features resulted in slightly better schedules than the features described in Study 1 in this paper. However, the computational cost of evaluating the time-delay neural networks to compute those features was so large that it negated the improvements in the schedules when evaluated as an anytime algorithm.

This concludes our discussion of the requirements for successful application of the $TD(\lambda)$ methodology. We now briefly summarize related work on resource-constrained scheduling and reinforcement learning.

5. Related Work

Many different methods in operations research (OR) and artificial intelligence (AI) have been applied to scheduling problems including constraint-based search (Fox, 1983; Crawford & Baker, 1994), simulated annealing (Zweben et al., 1994), tabu search (Hooker & Natraj, 1995), and genetic programming (Cleveland & Smith, 1989). Some of this work has resulted in systems for solving large-scale real-world problems. For example, the work on the ISIS (Fox, 1983), OPIS (Smith, 1994), and Micro-Boss (Sadeh, 1991) systems showed that large-scale manufacturing scheduling problems can be efficiently solved by using combinations of several methods (including constraint propagation and opportunistic search). Several companies, including i2 and Red Pepper Software, are now applying combinations of these technologies to solve large scale scheduling and supply-chain management problems.

However, none of these methods is based on a learning approach. We believe that the methods introduced in this paper will provide improvements in the speed and performance of scheduling methods on large-scale industrial problems.

Other important large applications of reinforcement learning include Tesauro's TD gammon (Tesauro, 1992), Crites and Barto's work on elevator control (Crites & Barto, 1996), and the work of Bertsekas and Singh (1997) on cellular telephone channel allocation. All of these studies showed that reinforcement learning algorithms combined with neural network function approximators can learn control policies in large search spaces.

6. Concluding Remarks

This paper has introduced a new approach to learning problem-specific search control heuristics for combinatorial optimization problems. The methodology is based on the observation that in practical settings, one often must solve many similar instances of the same optimization problem. This suggests a learning approach in which previous instances are analyzed (possibly at substantial computational cost) to discover a search control policy for solving new instances.

This paper demonstrated that this methodology has been successfully applied in a NASA scheduling problem. The learned policy performs better than a non-learning search approach based on simulated annealing.

We believe the methodology introduced in this paper can be applied to many other combinatorial optimization problems of industrial importance. It is usually straightforward to design a repair-based problem space for a combinatorial optimization problem. It is also fairly easy to come up with features that can characterize the degree of difficulty of a problem instance, and such features are very useful for learning value functions.

The two biggest challenges to applying this methodology are (a) normalizing the features and the immediate reward function so that they can generalize across problem instances and (b) training neural networks to be value function approximators.

The fact that our largest prediction errors occur near the final state suggests that we may have not fully succeeded in normalizing our reward function. Perhaps if the reward function for each instance were normalized using the length of the best known solution to each problem instance, this would reduce these late prediction errors. On the other hand, Tesauro observed similar problems with backgammon, where normalization was not an issue, so this problem may be inherent in learning value functions for Markov processes.

There is a wealth of lore concerning the best ways to train neural networks for supervised learning problems. However, our knowledge of methods for training them for value function approximation in reinforcement learning is still primitive. We suspect that neural networks may not be the best method for value function approximation. An important problem for future research is to develop function approximators that are better suited to reinforcement learning applications.

Another promising direction for future research is to develop algorithms that can exploit the unique properties of acyclic (or nearly-acyclic) problem spaces. Along these lines, Boyan and Moore (1996) have developed an algorithm called Rout that looks very promising. Testing Rout on our scheduling problem would be a very interesting future work.

7. Acknowledgements

The authors thank Monte Zweben and Todd Stock for introducing us to the space shuttle scheduling domain and for providing us with the domain specifications. The authors gratefully acknowledge the support of this research by the following funding agencies: NASA Ames Research Center (grant number NAG 2-630), the Office of Naval Research (grant number N00014-95-1-0557), and the National Science Foundation (grant number IRI-9204129).

References

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, *72*(1), 81–138.

Boyan, J. A., & Moore, A. W. (1996). Learning evaluation functions for large acyclic domains. In Saitta, L. (Ed.), *Machine Learning: Proceedings of the Thirteenth International Conference*, pp. 63–70. Morgan Kaufmann.

Cleveland, G. A., & Smith, S. F. (1989). Using genetic algorithms to schedule flow shop release. In *Proceedings of the the Third International Conference on Genetic Algorithms*, pp. 160–169. Morgan Kaufmann.

Crawford, J. M., & Baker, A. B. (1994). Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence.*

Crites, R. H., & Barto, A. G. (1996). Improving elevator performance using reinforcement learning. In *Advances in Neural Information Processing Systems 8* Cambridge, MA. MIT Press.

Deale, M., Yvanovich, M., Schnitzius, D., Kautz, D., & et al (1994). The space shuttle ground processing scheduling system. In Zweben, M., & Fox, M. S. (Eds.), *Intelligent Scheduling*, chap. 15, pp. 423–449. Morgan Kaufmann, San Francisco, CA.

Fox, M. S. (1983). *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* Ph.D. thesis, Carnegie Mellon University.

Garey, M. R., Johnson, D. S., & Smith, R. (1976). The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, *1*(2), 117–129.

Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of NP-completeness.* W. H. Freeman, New York.

Hooker, J. N., & Natraj, N. R. (1995). Solving a general routing and scheduling problem by chain decompostion and tabu search. *Transportation Science*, *29*(1), 30–44.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of AI Research*, *4*, 237–285.

Koenig, S., & Simmons, R. G. (1996). The effect of representation and knowledge on goal-directed exploration with reinforcement-learning algorithms. *Machine Learning*, *22*, 227–250.

Lin, L. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, *8*, 293–321.

Lin, L. (1993). *Reinforcement Learning for Robots using Neural Networks.* Ph.D. thesis, Carnegie Mellon University.

Lin, S., & Kernighan, B. W. (1971). An effective heuristic algorithm for the TSP. *Operations Research*, *21*, 498–516.

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

Sadeh, N. (1991). *Look-ahead Techniques for Micro-opportunistic Job Shop Scheduling.* Ph.D. thesis, Carnegie Mellon University.

Selman, B., Levesque, H., & Mitchell, D. (1992). A new method for solving hard satisfiability problems. In *Proceedings of AAAI-92*, pp. 440–446. AAAI/MIT Press.

Singh, S., & Bertsekas, D. (1997). Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems*, Vol. 10, pp. 974–980. MIT Press, Cambridge, MA.

Smith, S. F. (1994). OPIS: A metholology and architecture for reactive scheduling. In Zweben, M., & Fox, M. S. (Eds.), *Intelligent Scheduling*, chap. 2, pp. 29–66. Morgan Kaufmann, San Francisco, CA.

Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, *3*, 9–44.

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

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, *28*(3), 58–68.

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*(3/4), 279–292.

Zhang, W. (1996). *Reinforcement Learning for Job-Shop Scheduling.* Ph.D. thesis, Oregon State University.

Zhang, W., & Dietterich, T. (1995). A reinforcement learning approach to job-shop scheduling. In *IJCAI-95*, pp. 1114–1120.

Zhang, W., & Dietterich, T. (1996). High-performance job-shop scheduling with a time-delay TD(λ) network. In *Advances in Neural Information Processing Systems 8* Cambridge, MA. MIT Press.

Zweben, M., Daun, B., & Deale, M. (1994). Scheduling and rescheduling with iterative repair. In Zweben, M., & Fox, M. S. (Eds.), *Intelligent Scheduling*, chap. 8, pp. 241–255. Morgan Kaufmann, San Francisco, CA.