Reinforcement Learning for Job-Shop Scheduling

by

Wei Zhang

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

Presented April 16, 1996

Commencement June 1996

AN ABSTRACT OF THE DISSERTATION OF

Wei Zhang for the degree of Doctor of Philosophy in Computer Science
presented on April 16, 1996.
Title: Reinforcement Learning for Job-Shop Scheduling

Abstract approved: _____
Thomas G. Dietterich

This dissertation studies applying reinforcement learning algorithms to discover good domain-specific heuristics automatically for job-shop scheduling. It focuses on the NASA space shuttle payload processing problem. The problem involves scheduling a set of tasks to satisfy a set of temporal and resource constraints while also seeking to minimize the total length (makespan) of the schedule.

The approach described in the dissertation employs a repair-based scheduling problem space that starts with a critical-path schedule and incrementally repairs constraint violations with the goal of finding a short conflict-free schedule. The temporal difference (TD) learning algorithm $TD(\lambda)$ is applied to train a neural network to learn a heuristic evaluation function for choosing repair actions over schedules. This learned evaluation function is used by a one-step lookahead search procedure to find solutions to new scheduling problems.

Several important issues that affect the success and the efficiency of learning have been identified and deeply studied. These issues include schedule representation, network architectures, and learning strategies. A number of modifications to the $TD(\lambda)$ algorithm are developed to improve learning performance. Learning is investigated based on both hand-engineered features and raw features. For learning from raw features, a time-delay neural network architecture is developed to extract features from irregular-length schedules. The learning approach is evaluated on synthetic problems and on problems from a NASA space shuttle payload processing task. The evaluation function is learned on small problems and then applied to solve larger problems.

Both learning-based schedulers (using hand-engineered features and raw features respectively) perform better than the best existing algorithm for this task—Zweben's iterative repair method.

It is important to understand why TD learning works in this application. Several performance measures are employed to investigate learning behavior. We verified that TD learning works properly in capturing the evaluation function. It is concluded that TD learning along with a set of good features and a proper neural network is the key to this success. The success shows that reinforcement learning methods have the potential for quickly finding high-quality solutions to other combinatorial optimization problems.

ACKNOWLEDGEMENT

My first thanks go to Tom Dietterich. Tom has been providing me with a lot of ideas, criticism, and encouragement through the course of my Ph.D. study, and with the exciting NASA scheduling domain for carrying out my reinforcement learning research. My special thanks also go to Prasad Tadepalli. Prasad was a big help especially during the early stage of my dissertation research, when I encountered many problems and frustrations regarding the research approach. Thanks to my thesis committee, Tom Dietterich, Prasad Tadepalli, Bruce D'Ambrosio, and Curtis Cook, for insightful comments on a draft of this dissertation. Eric Chown and Bill Langford also gave helpful comments during the revisions of this dissertation.

Furthermore, I would like to express my gratitude to many colleagues at Oregon State University and other institutions for their indirect but important help, especially to Rich Sutton and Monte Zweben for helpful discussions and encouragement.

Finally, a lot of my thanks go to my wife and my parents for their love and support throughout all these years.

TABLE OF CONTENTS

		Page
1	Introduction	1
1.1	Job-Shop Scheduling Methodologies	3
1.1.1	OR Methods vs. AI Methods	3
1.1.2	Heuristic Methods vs. Stochastic + Heuristic Methods	7
1.1.3	Constructive Methods vs. Repair Methods	8
1.2	Speedup Learning and Its Application to Scheduling	9
1.2.1	Supervised Learning	10
1.2.2	Explanation-Based Learning	11
1.3	The Thesis: Reinforcement Learning for Scheduling	13
1.4	Organization	15
2	Reinforcement Learning & State-Space Search	17
2.1	Markov Decision Processes	19
2.2	Standard Methods	22
2.2.1	Value Iteration	23
2.2.2	Policy Iteration	24
2.2.3	Modified Policy Iteration	25
2.2.4	Adaptive State Aggregation	26
2.2.5	Linear Programming	29
2.3	Scale Up: Learning Partial Policies	30
2.3.1	Real-Time Dynamic Programming	32
2.3.2	Temporal Difference Learning	34
2.3.3	Sarsa and Q-learning	36
2.4	Scale Up: Generalization	37
2.5	Improving Learning Performance	40
2.5.1	Exploration	40
2.5.2	Teaching	41
2.5.3	Experience Replay	42

TABLE OF CONTENTS (CONTINUED)

Page

2.6 Learning to Search A State Space 43

2.6.1 Types of State-Space Search Learning Tasks 43
2.6.2 Designing Reinforcement Functions 45
2.6.3 The Looping Policy Problem 49
2.6.4 An Example 51

3 Domain: The Task, Scheduling, and Learning 55

3.1 Space Shuttle Payload Processing 55

3.2 A Repair-Based Scheduling Method 60

3.2.1 A Repair-Based Scheduling Problem Space 60
3.2.2 The Iterative Repair Algorithm 65

3.3 Defining Reinforcement Learning Problem 66

3.3.1 Resource Utilization and RDF 66
3.3.2 A Deterministic Decision Process for Scheduling 70

3.4 Challenges for the Learning Task 73

3.5 Learning Methods 74

3.5.1 Applying Value Function Approximation 75
3.5.2 Applying the TD(λ) algorithm 75
3.5.3 Exploration 77
3.5.4 Trajectory Backward Learning 77
3.5.5 Overtraining Big Errors with Multiple Updates 78
3.5.6 Best Experience Replay 78
3.5.7 Teaching 79
3.5.8 Policy Determination Methods 79

3.6 Backward TD(λ) 81

3.6.1 A Walk-Control Problem 84
3.6.2 Learning Forward 85
3.6.3 Learning Backward 88

TABLE OF CONTENTS (CONTINUED)

Page

4 Learning with Hand-Engineered Features 92

4.1 Main Results 92

4.1.1 Features 92
4.1.2 Problem Sets 94
4.1.3 Network Architecture and Training Procedure 96
4.1.4 Performance Results 97

4.2 Results of the Preliminary Study 101

4.2.1 The Approach 102
4.2.2 Significance of Features 104
4.2.3 Network Architectures 111

4.3 Evaluation of Learning Performance 114

4.4 Evidence for TD Learning 120

4.4.1 Learning Is Not Feature Copying 120
4.4.2 Training with No Final Rewards 121
4.4.3 Training with the Last Section of A Sequence 122
4.4.4 Learning Early Decisions 124

4.5 How to Learn? 129

4.5.1 Backward Learning and the λ Parameter 129
4.5.2 Other Strategies 133

4.6 Generality 135

4.6.1 Generality of Learning Performance 135
4.6.2 Generality of Learned Knowledge 136

4.7 Summary 137

5 Learning with A Time-Delay TD(λ) Network 140

5.1 Schedule Representation 140

5.2 Network Architecture 141

TABLE OF CONTENTS (CONTINUED)

Page

5.3 Experimental Results 144

5.4 Discussion and Summary 148

6 Conclusions 153

6.1 Contributions 155

6.2 Future Work 157

Bibliography 159

Appendix 169

Appendix Experimental Problem Sets 170

LIST OF FIGURES

Figure		Page
2.1.	The reinforcement learning paradigm.	18
2.2.	The value iteration algorithm.	24
2.3.	The policy iteration algorithm.	25
2.4.	The modified policy iteration algorithm.	26
2.5.	The adaptive state aggregation algorithm.	28
2.6.	The real-time dynamic programming algorithm.	33
2.7.	A Simple State-Space Search Problem.	51
3.1.	A simple scheduling problem.	63
3.2.	A critical path schedule of the 8-task job.	64
3.3.	The iterative repair algorithm.	67
3.4.	A walk control problem.	84
3.5.	Input representation of a walk-control problem with $n = 20$.	84
3.6.	The mean error for learning with different λ using the best α.	87
3.7.	The policy correctness for learning with different λ using the best α.	87
3.8.	The best α for different λ.	87
3.9.	The number of trials of update required to reach a set of goals for learning with different λ using the best α.	87
3.10.	Compare forward learning and backward learning in the mean error for learning with different λ using the best α.	89
3.11.	The number of trials of update required to reach a set of goals for backward learning with different λ using the best α.	89
3.12.	Compare forward learning and backward learning in the number of trials of update required to reach a set of goals using the best λ.	89

LIST OF FIGURES (CONTINUED)

Figure		Page
4.1.	Average RDF over 15 CV Problems.	98
4.2.	Average Number of Repairs over 15 CV Problems.	98
4.3.	Performance Comparison of TD to IR on 50 Small-scale Problems.	99
4.4.	Performance Comparison of TD to IR on 20 Medium-scale Problems.	99
4.5.	Performance Comparison of TD to IR on 20 SSPP Problems (RDF).	101
4.6.	Performance Comparison of TD to IR on 20 SSPP Problems (% Wins).	101
4.7.	Scatter plots for 22 features.	105
4.8.	Sequence plots for 22 features and the value.	106
4.9.	The Hinton Diagram.	110
4.10.	Overlapping Gaussian ranges with $\sigma = 1/10, 1/14$, and $1/18$.	112
4.11.	Training MSE.	113
4.12.	Correctness of branch decisions.	113
4.13.	Training Error on 30 Training Problems.	118
4.14.	Average RDF over 20 Validation Problems.	118
4.15.	Average Number of Repairs over 20 Validation Problems.	118
4.16.	MSE of Final Predictions on 20 Validation Problems.	118
4.17.	Smoothness of Predictions on 20 Validation Problems.	118
4.18.	Compare the current-state-RDF to the learned value function.	121
4.19.	Effect of training with no final rewards shown with the average RDF.	122
4.20.	Effect of training with no final rewards shown with the average number of repairs.	122

LIST OF FIGURES (CONTINUED)

Figure — Page

4.21. Effect of training with the last section shown with the average RDF. 123

4.22. Effect of training with the last section shown with the average number of repairs. 123

4.23. Effect of the TD policy inserted in different stages of the IR scheduling procedure. 126

4.24. Effect of the IR policy inserted in different stages of the TD scheduling procedure. 126

4.25. Histograms of the average RDF of the solutions produced by 10 different scheduling procedures. 128

4.26. The effect of λ and learning with actual outcomes. 130

4.27. Forward learning under $\lambda = 0.2, 0.9$ and 1. 132

5.1. Time-Delay Neural Network Architecture. 143

5.2. Mean RDF values. 146

5.3. Mean number of repairs. 146

5.4. Comparison of Accepted Schedule Repairs. 147

5.5. Comparison in CPU time. 147

5.6. Five kernel networks. 149

5.7. The utility problem. 150

LIST OF TABLES

Table		Page
2.1.	The reinforcement function.	47
2.2.	The optimal policy.	52
2.3.	A bad policy.	52
2.4.	A dynamic policy.	53
2.5.	Value function approximation with a linear function.	53
3.1.	Resource pools.	57
3.2.	Pallet-Igloo high-level tasks.	58
3.3.	Pallet-Igloo LEVEL-III-II tasks.	59
3.4.	The repair-based scheduling problem space.	62
3.5.	Five goals for learning to walk.	86
4.1.	Features.	103
4.2.	Significance of the features according the linear and regression tree model.	108

LIST OF APPENDIX TABLES

Table		Page
A.1.	ART-1 Problem Specification.	170
A.2.	ART-2 Problem Specification.	171
A.3.	SSPP Problem Specification.	172
A.4.	SSPP Training Problems.	173

Reinforcement Learning for Job-Shop Scheduling

Chapter 1
Introduction

The problem of planning and scheduling has captured the interest of many researchers from a number of research communities: management science, industrial engineering, operations research (OR), and artificial intelligence (AI). Planning determines how the system achieves different types of goals by selecting and sequencing activities to satisfy a set of logic domain constraints. Scheduling takes responsibility for day to day operations. It is the process of selecting alternative plans and assigning resources and times to the set of activities in the plan. These assignments must obey a set of constraints that reflect the temporal relationships between activities and the capacity limitations of a set of shared resources. The assignments also affect the optimality of a schedule with respect to criteria such as cost, tardiness, or throughput.

This defines the basic difference between planning and scheduling. However, a clear understanding and agreement on this difference does not exist. A planning problem may be concerned with resource allocation, while a scheduling problem may involve logic constraints. Fox and Zweben (1995) define a plan as a chain of activities to be performed, where each activity may specify its duration, start/end time, resource requirements, and some special restrictions. Planning is described as a cycle of three phases: plan creation, plan instantiation, and plan execution. *Plan creation* selects and sequences activities such that they achieve one more goals and satisfy a set of logic constraints. *Plan instantiation* then assigns resources and times for each activity of the plan. *Plan execution* dispatches activities and resources to agents that perform them, monitors their execution, and repairs plans within the bounds specified by the plan creation and instantiation phases. This definition roughly describes the whole process of planning and scheduling in the real-world industrial setting.

Planning and scheduling are a ubiquitous activity in everyday life. They exist with great commercial value in many domains such as manufacturing, distribu-

tion, transportation, construction, engineering maintenance, and spacecraft operations. Unfortunately, almost all of the nontrivial planning/scheduling tasks are at least NP-hard. Producing a useful plan or schedule can be very expensive and time-consuming. Under our current understanding of computational complexity theory, there is little hope of finding general-purpose solutions to these problems. However, in any particular application setting, there are usually domain-specific constraints and regularities that can be exploited to construct effective heuristic methods to produce useful plans and schedules quickly.

The research presented in this dissertation focuses on the application domain of space shuttle payload processing (SSPP) for NASA. The problem is to schedule the various tasks that must be performed to install and test the payloads that are placed in the cargo bay of the space shuttle. This study concerns a typical specification of the task, which is in the form of *job-shop scheduling*. Each shuttle mission is a job, which is tied to a fixed launch date. Each job consists of a partially-ordered set of tasks with resource requirements. The goal is to schedule the set of tasks to satisfy a set of temporal and resource constraints while also seeking to minimize the total duration (makespan) of the schedule.

The terminology of job-shop scheduling arose in the manufacturing industries. In the fundamental theory of scheduling [Baker, 1974, Coffman, Jr., 1976, Garey and Johnson, 1979, French, 1982], job-shop scheduling defines a set of scheduling problems in the following mathematical formulation. Given a set of *machines* that can process only one job at a time, and given a set of *jobs* that must be processed on each of these machines in a prescribed, job-dependent order, find a feasible schedule or a schedule that minimizes the total processing time. The processing time of a job on a machine may vary. Each job may define a ready time to be released (no earlier than the time) and a due time to be finished (no later than the time).

Real-world job-shop scheduling problems are usually more complicated. Now job-shop scheduling has been viewed as a general type of scheduling problem that involves scheduling a number of jobs, but may involve more types of constraints. The optimization criterion may also vary. Because of the mathematical similarity,

the terminology of job-shop scheduling is adopted in many other scheduling domains such as transportation scheduling and spacecraft scheduling.

Space shuttle operations scheduling is dynamic in nature. [Deale *et al.*, 1994] stated that

> *... Weather plays a major role in the transportation of the orbiter from California to KSC, and on a number of occasions, the orbiters have been damaged as a result of the weather as they cross the country. As tests are performed in the OPF and some of them fail, new tasks must be added to the schedule to diagnose and fix the problems. Damage to the orbiter can also introduce new tasks. These new tasks can involve small or large amount of work (hour to weeks) and can require extensive rescheduling.*

The dynamics of the scheduling problem addresses several special difficulties to the scheduling systems. It needs a scheduling system able to produce schedules quickly for real-time usage. It also requires that the system have a large collection of capabilities to handle the dynamics of the scheduling problem.

1.1 Job-Shop Scheduling Methodologies

The literature in job-shop scheduling is enormous. The scheduling methodologies can be basically categorized in terms of the following disciplines.

1.1.1 OR Methods vs. AI Methods

Starting in the 1950's operations researchers began advocating formal optimization approaches to scheduling. Operations research contributes to scheduling with a variety of nice mathematical optimization approaches including linear programming (LP), dynamic programming (DP), branch and bound methods, and cutting-plane methods. Most of these methods provided guaranteed solutions to achieving optimality

and can be applied to produce schedules in a reasonable amount of time when the size of a given scheduling problem is not too big.

Unfortunately, most real-world scheduling problems are NP-hard and large. For these problems, optimization methods fail to provide timely solutions. Therefore, we must abandon our desire to searching for optimality and turn our attention to find methods that can efficiently produce satisfactory (but not necessarily optimal) solutions. These methods do not provide a guarantee of optimality, and hence they are called *heuristic methods*. Many efforts have been made in developing effective heuristics in the OR framework. Accordingly, such OR methods have been more connected to AI methods.

AI began its attack on the scheduling problem in early 1980s [Fox, 1983], aiming at developing approaches capable of solving general and large-scale real-world scheduling problems. The AI methodology basically is knowledge-based search or heuristic search. The task of AI, in general, is to develop heuristic search paradigms for problem-solving and to discover and represent knowledge for efficient search. The following four heuristic search paradigms basically characterize the state of the art of the intelligent scheduling technologies that have been applied in the scheduling domain.

The first scheduling search paradigm is *constraint-directed search*. By this paradigm, a scheduling problem is formulated as a *constraint diagram*, which can be a *constraint satisfaction problem* (CSP) or a *constraint optimization problem* (COP). The constraint diagram is normally specified with a set of *variables*, a *domain* for each variable specifying the *values* to which the variable may be assigned, and a set of *constraints* on the variables. One major benefit of using this paradigm is due to its *constraint propagation* techniques. By performing constraint consistency check and propagation, these techniques may greatly narrow the possible assignments of values to each variable, therefore, reducing the search space. The methods that follow this paradigm included ISIS, OPIS, and MICRO-BOSS.

ISIS developed by Mark Fox and colleagues [Fox, 1983, Fox, 1994] first demonstrated AI technologies in scheduling. One of the most important contribution of this

work is that ISIS provided approaches to dealing with the full range of constraints and objectives encountered in the manufacturing domain. Unfortunately, the overall performance of ISIS is not very good due to its rigid search mechanism: it only paid attention to the order of the jobs in a problem, but without considering the utilization of bottleneck resources. OPIS [Smith *et al.*, 1986, Smith and Ow, 1986, Smith, 1994] is the successor to ISIS. OPIS introduced the *opportunistic* scheduling mechanism, which can identify bottlenecks and switch from job-order perspective to a resource perspective. MICRO-BOSS was a more powerful opportunistic scheduler. Rather than scheduling entire bottleneck resource (which is called the macro-opportunism) of OPIS, MICRO-BOSS took a microscopic view of opportunism by iteratively selecting and scheduling one activity at a time. The search was performed based on the evaluation of the "*textures*" of the constraints. These textures measure the properties of the constraint diagram such as contention and reliance of variables on values, providing useful heuristics for variable and value ordering.

The second paradigm is using simulated annealing [Kirkpatrick *et al.*, 1983, Van Laarhoven, 1988, Aarts, 1989]. Rather than regular hill climbing, simulated annealing allows some downhill moves with a certain probability; therefore, it comes up with a higher chance of finding the global optimal solution. As a computational procedure, simulated annealing mimics the physical process of annealing for producing a minimal-energy solid state. In an annealing process, physical substances such as metals are melted and then gradually cooled until some solid state is reached. This paradigm has been applied in many scheduling applications [Ogbu and Smith, 1990, Abramson, 1991, Van Laarhoven *et al.*, 1992, Zweben *et al.*, 1992, Mittenthal *et al.*, 1993]. One most prominent work was Zweben and colleague's Space Shuttle scheduling application [Zweben *et al.*, 1994]. The resulting scheduling system is able to provide an efficient and flexible facility for scheduling space shuttle ground operations. It is in regular use at the Kennedy Space Center.

The next noticeable search paradigm is tabu search [Glover, 1990]. Tabu search performs hill climbing except it makes use of a *tabu list* to force the search away from solutions selected for recent iterations to escape the local maxima. The tabu

list records both local and large-step search information. Tabu search uses tabu "restriction" and "aspiration" criteria to prevent search from cycling for a better local search. It also uses "intensification" and "diversification" mechanisms to determine when and how to jump search to another area. The work of applying tabu search to scheduling includes [Dell'Amico, 1993], [Icmeli and Erenguc, 1994], [Hübscher and Glover, 1994], and [Hooker and Natraj, 1995]. Several researchers have shown that tabu search has the potential to outperform simulated annealing for scheduling and optimization problems [Lourenco and Zwijnenburg, 1995, Woodruff, 1994].

Finally, there has been a large amount of research using genetic algorithms (GAs) [Goldberg, 1989]. Many of the projects showed the success of their applications, including [Biegal and Davern, 1990], [Syswerda, 1991], [Fang *et al.*, 1993], [Uckun *et al.*, Oct 1993], [Lee and Piramuthu, Oct 1994], and [Nordström and Tufekci, 1994]. GAs mimic the process of natural evolution in searching a solution space for optimal solutions. A GA combines the principle of "survival-and-the-fittest" with randomized information exchange. Solutions are improved over "generations" of the mating and offspring bearing process on the "population" (the set of current solutions) until the solutions are near optimal. Another power of GAs is that they can be applied to learn problem-solving knowledge [Montana and Davis, 1989, Goldberg, 1989]. While GAs are mostly applied as optimization methods for scheduling, there is some work in using GAs to learn scheduling knowledge. [Hilliard *et al.*, 1988] showed experimentally that optimal rules for simple scheduling problems could be discovered by GAs. [Davis, 1985] focused attention on the order-dependent nature of job-shop scheduling, and then [Cleveland and Smith, 1989] and [Whitley *et al.*, 1989] explored new GA operators to handle these order considerations. [Aytug *et al.*, 1994] presented an approach of using GAs to learn composite dispatching rules as scheduling knowledge based on some "base" dispatching rules such as FCFS (first come first serve), EDD (earliest due date), and MST (minimum slack time) rules.

GAs work well on problems that can be effectively encoded. Many authors have developed problem-specific representations of solutions for scheduling problems. However, this requires specialized representations for each problem variation and fails to

work for general problems. Furthermore, it is not yet clear whether and how much the GA mechanisms benefit the search process. Recently, many researchers raised questions about GA mechanisms and showed negative experimental results. [DeJong, 1992] claims that GAs are not a function optimizer and that typical GAs for optimization often use different and specially-customized mechanisms which are not suited for GAs used for adaptation in dynamic environments. [Lang, 1995] described "an experiment that cast doubt not only on the relative efficiency of genetic programming, but also on the validity of the story about the value of the high-fitness gene pool". [Baluja and Caruana, 1995] and [Baluja, 1995] found that GA's population-based search mechanism and the "crossover" operator (which have been the keys of GAs) are not particularly helpful. These mechanisms did not lead to an efficient search not only for a set of commonly used benchmark problems but also for a problem custom-designed to benefit both from using of a population and the crossover operator.

1.1.2 Heuristic Methods vs. Stochastic + Heuristic Methods

A heuristic method relies on a set of heuristics to perform search. However, heuristics are not able to guarantee finding the optimal solutions. Even worse, in many situations they may lead to a search with very long time and finally fail to produce even a feasible solution. Discovering effective heuristics in a complex domain is difficult and time-consuming. For this reason, an approach that combines stochastic search with heuristics has become popular.

Random operations can be introduced in a heuristic method with different degrees. Constraint-directed search and tabu search make little use of random operations. These methods compute more-informed heuristics, and accordingly, the computation involved in each search step is more expensive. In contrast, simulated annealing and GAs more rely on the randomness. These methods involve less-informed heuristics, and the heuristics are cheap to compute. Because heuristics for these methods are less reliable, it becomes necessary to introduce certain randomness during the search.

Certainly, for a problem where we have known effective heuristics or where we can easily find these heuristics, pure heuristic methods will be preferred. Otherwise, the combination of stochastic and heuristic methods would be more useful. Job-shop scheduling has been recognized to be very difficult. For many applications, even under extensive investigation, we would still have no luck finding effective heuristics. A system with too much randomness must have a poor guarantee on its performance. Therefore, a high performance scheduling system must rely on high quality heuristics.

1.1.3 Constructive Methods vs. Repair Methods

Scheduling methodologies can also be classified according to whether a schedule is constructively made or is made by iteratively repairing a complete schedule. The *constructive methods* incrementally extend a partial schedule until every task in the schedule has been scheduled. The *repair methods* iteratively modify a complete schedule to remove conflicts or to further optimize the solution.

Most optimization methods and constraint-directed search methods follow the constructive approach. These methods has been proved very successful for a variety of classes of scheduling problems. The constructive methods seem more straightforward for scheduling.

A repair method can also perform efficiently in making new schedules. Simulated annealing, tabu search and GAs are generally practiced under a complete schedule search space. Some evaluation on a complete schedule, such as the number of violations [Minton *et al.*, 1992] and the current schedule length, can be very useful for searching the problem space effectively. With such a good objective evaluation, methods like simulated annealing can often perform very well in quickly finding high-quality solutions.

The fundamental motivation for using the repair-based approach is to deal with the *rescheduling* problem. For most real-world scheduling tasks, only a very small percentage (less than 5%) of time spent scheduling is making new schedules while most of the time is spent rescheduling, which revises and maintains schedules in the

context of daily progress and changing resources. The constructive methods work inefficiently for rescheduling. They must remove a set of tasks from the schedule and then implement the scheduling procedure. Unfortunately, determining which task to unschedule is not straightforward. In this situation, the repair-based approach certainly is preferred.

The dynamics of the scheduling tasks, which requires frequent rescheduling, is the top concern in the NASA space shuttle scheduling domain. The work presented here focuses on the study of repair-based scheduling problem space.

1.2 Speedup Learning and Its Application to Scheduling

While there exist many real-world problems such as job-shop scheduling that are difficult to solve, a large number of researchers have been working on developing machine learning techniques to automatically discover problem-solving strategies. The field of *speedup learning* is concerned with the study of systems that attempt to improve the efficiency of a problem solver. Speedup learning meets the dynamic requirement of the scheduling world in many aspects. Learning is aimed at acquiring better scheduling strategies to find good schedules more quickly. While it is very expensive for humans to develop effective scheduling heuristics, by learning this process is done automatically. Furthermore, learning is an adaptive process. The learned heuristics can be improved and refined automatically in everyday use.

Speedup learning is studied in the AI problem-solving framework. In general, the AI methodology defines a problem-solving task as a *state-space search*. To perform state-space search, we first specify one or more states within the space where the search may start. We then specify a set of states that would be acceptable as solutions to the problem. We also specify a set of operators (or actions) to perform search. Applying an operator in a state will modify the configuration of the state to produce a new state. The state space contains all the states reachable by the search operations with the provided operators.

One general technique for speedup learning is to learn some form of the evaluation function as search control knowledge. An evaluation function is a real-valued function $f(s)$ that can be applied to a state, s, to estimate how close that state is to the desired goal. A number of search paradigms rely on an evaluation function as the heuristic to guide search. These search paradigms includes best-first search, mini-max game tree search, and various forms of greedy search and real-time search [Russell and Norvig, 1995, Rich and Knight, 1991].

Search control knowledge can be also learned in the form of *policies*. An *exact* policy explicitly tells what action should be chosen in a state; while an *generalized* policy can be in a form of *operator selection/rejection rules* [Laird *et al.*, 1986, Minton, 1990] or *operator strength functions* [Grefenstette, 1988]. Operator selection/rejection rules are "meta-level" rules that are evaluated to decide which "base-level" rule or operator should be applied next. An operator strength function evaluates each operator based on how successful that operator has been in the past.

Another general technique is to learn new "macro operators" by composing sequences of original operators [Mitchell *et al.*, 1986, DeJong and Mooney, 1986]. The learned macro operators are added to the set of operators considered by the problem solver, and they allow it to take "big steps" in the search space.

Speedup learning has mostly adopted the following two learning methodologies: supervised learning and explanation-based learning, which will be discussed in the following two sections.

1.2.1 Supervised Learning

Supervised learning has been well investigated with much progress in the past two decades. Supervised learning learns classification rules inductively from examples. During learning, a learning algorithm receives a set of training examples provided by a "teacher", each labeled as belonging to a particular class. The goal of the algorithm is to produce classification rules that can correctly assign future examples

to given classes. Popular supervised learning models include decision trees and neural networks.

Learning is an *inductive* process that produces general rules from specific examples. An "inductive leap" is considered reasonable in terms of certain *biases* of learning. For example, the bias for learning might specify that hypotheses are restricted to be decision trees. The bias might also say that the smallest tree that is consistent with all examples is preferred over all other consistent trees.

While supervised learning was designed to solve various kinds of prediction problems, it can also be applied to learn an evaluation function and meta-level rules as search control knowledge to improve the efficiency of a problem solver. The most impressive works include Samual's checker-playing program [Samuel, 1959] and the LEX system [Mitchell *et al.*, 1983]. Applying supervised learning to learn search control knowledge requires having knowledge about the correct solution paths. An analysis of the traces provides positive and negative instances for generating rules to control search.

Unfortunately, there is some difficulty in applying supervised learning techniques to the scheduling domain. Supervised learning requires a teacher to provide training examples to tell what are the right actions to take to modify a schedule. This is not realistic for a complex scheduling task. Ideally for scheduling, we need to have some mechanism that can incrementally generate training patterns for learning scheduling strategies.

1.2.2 Explanation-Based Learning

Explanation-based learning (EBL) [Mitchell *et al.*, 1986, DeJong and Mooney, 1986] is a knowledge-based analytical learning methodology. EBL has mostly been applied to the speedup learning problem. It has been developed to learn search control knowledge in the forms of meta-level rules as well as to learn new macro operators. EBL has been integrated into several problem-solving architectures including SOAR [Laird *et al.*, 1986] and PRODIGY [Minton, 1990]. When these architectures need

to choose an operator to apply to some state, they consult a (learned) set of control rules that recommend which operator to apply. If no control rules apply to the current state (or if the rules conflict), then the problem solver attempts to solve the problem by some form of search. An EBL training example is then obtained, and accordingly, a generalized description of the current state and a record of the operator sequence applied through the search path are kept and treated as a new decision rule.

Based on a given *domain theory* (that is a set of rules and facts to be used in explaining how a training example is transformed to satisfy a given goal) and an *operationality criterion* (that constrains the generalization process by specifying the representation form required in the learned description), EBL generates a description that is a generalization of the training example and forms a control rule specifying that any state satisfying this description should perform the obtained operator sequence.

Therefore, EBL provides a powerful speedup learning approach that can be applied to general problem-solving. It can operate with complex problem solver architectures and learn various forms of knowledge to improve the efficiency of a problem solver.

One shortcoming of EBL is that there is no performance guarantee on the learning process. EBL does not learn optimal or near-optimal problem-solving strategies [Dietterich and Flann, 1995]. Another problem with EBL is the *utility* problem [Minton, 1990]. Although learning provides a problem solver with better knowledge of search, the system may slow down dramatically as more complex control knowledge is learned, because of the cost of evaluating the control knowledge.

The EBL methodology has been applied in a scheduling domain. For example, [Zweben *et al.*, 1992] developed an EBL method named *plausible explanation-based learning* (PEBL) to overcome the utility problem and applied it to solving the space shuttle payload processing problem. In PEBL, the concept description is generalized by a set of examples rather than by a single example. Learning a control rule is delayed until there is enough evidence showing that the conjectured explanation is likely to be true. An experiment using PEBL to learn variable ordering heuristics showed that scheduling with PEBL is faster than no learning or learning with regular EBL.

However, to make EBL more useful for scheduling, further development is needed to improve its ability to approach optimality.

1.3 The Thesis: Reinforcement Learning for Scheduling

A growing number of researchers have been developing learning methods based on dynamic programming [Bellman, 1957, Bertsekas, 1987] for solving stochastic optimal control problems, leading to the coming of current *reinforcement learning* (RL) [Sutton, 1990, Sutton, 1991, Watkins, 1989, Watkins and Dayan, 1992, Barto *et al.*, 1995]. Briefly, reinforcement learning algorithms learn policies for optimal control problems through "trial-and-error". For a state space problem-solving task, a policy is the search control knowledge that tells what operator should be performed in each state. During learning, the learning system receives a reinforcement signal (or "reward") after each action. The goal of the learning system is to find a policy that maximizes the expected reinforcement over future actions.

The theory of convergence available for reinforcement learning algorithms [Watkins and Dayan, 1992, Dayan and Sejnowski, 1994, Jaakkola *et al.*, 1994, Sutton, 1988, Tsitsiklis, 1994, Watkins and Dayan, 1992] makes them attractive as a basis for building learning systems to solve a wide variety of control, planning, and search problems. Unfortunately, almost all of the convergence results assume lookup table representation for the value function. An important research problem is to make reinforcement learning algorithms work with more compact representations.

The research described in this dissertation is motivated and greatly encouraged by the success of the TD-gammon system developed by Tesauro [Tesauro, 1992]. Tesauro showed that the reinforcement learning algorithm TD(λ) could learn to play world-class backgammon. His TD-gammon program is now far better than any other computer program and competitive with the best human players. A question addressed in this research is whether the success in TD-gammon can be replicated in an industrial problem.

The goal of this research is to explore the possibility of applying reinforcement learning algorithms to discover good scheduling heuristics automatically. This work focuses on the NASA space shuttle payload processing scheduling task. To make reinforcement learning work for the task, the following key research problems must be addressed.

1. How do we define scheduling as state-space search?
   - What should be the starting schedule?
   - What should be the operators to be used for modifying a schedule?
2. How do we define the reinforcement learning problem for the scheduling task?
   - How do we define the reinforcement function?
3. How do we represent a schedule? We need an abstract state representation.
   - What are useful features for describing schedules abstractly?
   - Can we discover these features automatically?
4. What kind of function approximators should be used?
5. What learning algorithms should be used?
6. How do we perform scheduling using learned knowledge?
7. How do we evaluate learning performance?

In previous work for this task, Zweben and colleagues developed an iterative repair method that combines a set of heuristics with a simulated annealing search procedure. The resulting scheduling system provides an efficient and flexible facility for space shuttle payload processing. The challenge for the learning approach is to discover scheduling heuristics that can match or exceed the quality and efficiency of this iterative repair method.

1.4 Organization

Reinforcement learning is the problem faced by a learner that must learn to act through trial-and-error interactions with its environment. The next chapter first reviews the mathematical foundation of reinforcement learning—the Markov decision theory—and discusses standard dynamic programming methods to solve Markov decision problems. However, standard methods fail to find solutions when the state space is large. The chapter then discusses the reinforcement learning methods developed for solving large-scale problems. Two types of learning methodologies are discussed, which are learning partial policies and generalization. Following this, the chapter presents methods of formulating different types of state-space search problems as reinforcement learning problems. The author introduces the concept of a "dynamic policy" for decision problems to solve the infinite loop problem for state-space search.

Chapter 3 describes the space shuttle payload processing task in greater detail. It then describes a repair-based scheduling problem space and Zweben's iterative repair scheduling method. Following this, the chapter describes how the scheduling task is formulated as a reinforcement learning problem. It then shows how the temporal difference learning algorithm TD(λ) can be applied to train a neural network to learn an evaluation function over states. There are several modifications to the TD(λ) algorithm to make it work efficiently. Finally, the chapter presents a study of a key modification to TD(λ)—backward TD(λ), which shows that backward learning learns more accurately and more efficiently than standard forward TD(λ).

Chapter 4 presents a detailed study of learning scheduling knowledge using TD(λ) based on hand-engineered schedule features. Three different sets of scheduling problems are used as the testbed for training and testing. A set of features is drawn that basically describes the resource utilizations, schedule length, and slack times in a schedule. Based on this, TD(λ) (with modifications) is applied to learn the value function. This chapter shows that the scheduling system using the evaluation function learned by temporal difference learning can outperform the iterative repair scheduler on realistic scheduling tasks. Furthermore, the learned knowledge can be

applied to scheduling problems that are larger and more complex than the ones that were studied during training.

Following this, Chapter 4 presents a preliminary study of features using statistical and supervised learning methods. It then presents a study of understanding why reinforcement learning works for this domain. Learning performance is assessed by a number of measures. The chapter then presents an investigation of the TD learning method and shows the role of the different mechanisms in the learning process. Finally, it discusses the generality of learning performance and learned knowledge. These results show that reinforcement learning is the key to the success of acquiring good evaluation functions.

Chapter 5 presents a study of applying the time-delay neural network (TDNN) architecture to learn the value function from raw schedule features instead of the hand-engineered features used in Chapter 4. It first shows how to extend the TDNN architecture to apply it to irregular-length schedules. It then verifies that the TDNN combined with TD(λ) can match the performance of the previous hand-engineered system. It finally discusses some important practical issues that are particularly related to the TDNN-TD(λ) approach.

Finally, Chapter 6 gives a summary of the work presented in the previous chapters. It concludes the thesis with a discussion of important directions for future research.

Chapter 2
Reinforcement Learning & State-Space Search

In the standard reinforcement learning paradigm, an agent is connected to its environment via perception and action, as depicted in Figure 2.1. In each step of interaction, the agent senses the current state of its environment, s, and then selects an action, a, to change the state of the environment. This state transition would generate a reinforcement signal, r, that can be received by the agent. The task of the agent is to learn a policy for choosing actions in each state to receive the maximal long-run cumulative reinforcements. Reinforcement learning methods apply the trial-and-error methodology to explore the environment over time to come up with a desired policy.

A typical type of the environment is one that possesses the *Markov property*. In such an environment, what will happen in the future depends on the current state of the environment and only on it. Most reinforcement learning researchers have been focused on learning in this type of environment, coming up with a number of important reinforcement learning methods such as model-free method *Q-learning* [Watkins, 1989, Watkins and Dayan, 1992] and model-based methods Dyna [Sutton, 1990] and prioritized sweeping [Moore and Atkeson, 1993]. Recently, a number of researchers are also concerned about developing reinforcement learning methods for non-Markov domains, particularly for *partially observable* stochastic domains, yielding several important results [Cassandra *et al.*, 1994, Littman *et al.*, 1995a, Parr and Russell, 1995]. Learning for non-Markov problems is generally much more difficult.

The research presented in this dissertation is concerned with developing reinforcement learning methods for constructing scheduling policies to solve the job-shop scheduling problem. To do this, we focus on the study of another key research problem in reinforcement learning: learning to act in a very large state space. In this

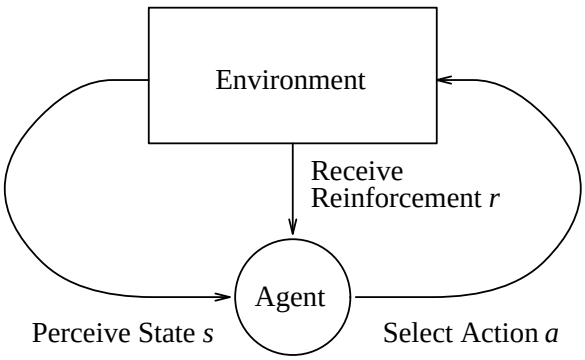


Figure 2.1. The reinforcement learning paradigm.

situation, the trial-and-error reinforcement learning methodology must be applied because complete exploration of the state space is intractable.

A nice characteristic about the scheduling problem space is that it follows the Markov property. We can determine how to repair a schedule completely based on the current schedule. Furthermore, we take advantage of the state-of-the-art of design of the scheduling problem space. As in most scheduling work, we employ deterministic operators to repair a schedule. In this situation, the state-transition under an action is exactly known. By this the effect of learned search control knowledge would be easier to predict and presumably better scheduling performance would be achieved.

The following sections of this chapter review the foundation of reinforcement learning concerned with the class of problems which have a model of state transitions, named Markov Decision Processes, and discuss the reinforcement learning methods for solving large-scale problems. In particular, we will present the methods of formulating the reinforcement learning problem for different types of state-space search problems. We will discuss the infinite loop problem in state-space search and introduce the notion of dynamic policy to solve this problem.

2.1 Markov Decision Processes

The Markov decision process (MDP) is a mathematical model to describe the decision problem for an agent in an environment with the Markov property. A Markov decision process is defined as a collection of the following objects:

- a finite set of states S,
- a finite set of actions $\mathcal{A}$, which usually is more specifically described as a set of *admissible actions* $A(s) \subseteq \mathcal{A}$ for each state $s \in S$,
- a reward function $R : S \times A \times S \to \Re$, which describes the immediate reinforcement (or immediate reward) of taking an action a in state $s \in S$ with resulting state $s' \in S$, and
- an action model $P : S \times A \times S \to [0, 1]$, which describes the state transition probabilities $P(s, a, s')$ (or written as $P_a(s, s')$) from current state s to next state s' when action a is taken. The model says that the probabilities of being in the next state is independent of states prior to the present state. Note that the sum of probabilities of moving to all possible next state s' for any state-action pair (s, a) must be 1:

$$\sum_{s' \in S} P_a(s, s') = 1. \tag{2.1}$$

If $P_a(s_i, s'_{i+1}) \in \{0, 1\}$, which says that the probability of moving to state s' is either zero or one, then action a is deterministic. If this is true for all state-action pairs, then the process is called a *deterministic decision process* (DDP).

A set of states $S' \subseteq S$ is said to be an *ergodic class* if any two states in S' can be reached from each other with some finite steps. A MDP is ergodic if the whole state space is an ergodic class. A state is called *absorbing* if it is a terminal state. Once a terminal state is entered, it is never left. An absorbing state constitutes a single-element ergodic class.

The solution of a Markov decision problem is an optimal *policy* for choosing actions at each state measured by an optimality criterion. The optimality criterion

describes how reinforcements are accumulated. It indicates the time horizon to be considered and whether the future reward should be discounted. The time horizon, described as discrete stages, can be finite or infinite, which defines a *finite-stage* problem or an *infinite-stage* problem respectively. Based on the optimality criterion, it is useful to define a *value function* $V^\pi(s)$ that gives the long-term expected total reinforcement of using policy π, starting in state s. The value of s with H-step horizon is

$$V^\pi(s) = E\left[\sum_{t=0}^{H} \gamma^t R(s_t, a_t, s_{t+1}) | s_0 = s\right], \tag{2.2}$$

here E is the notation of expectation, $a_t = a^\pi(s_t)$ is the action suggested by policy π in state s_t at stage t, $\gamma \in [0,1]$ is a *discount factor* that discounts the future reinforcement.

Alternatively, the value function can be defined recursively:

$$\begin{aligned} V^\pi(s) &= V_H^\pi(s) && (2.3)\\ &= \sum_{s' \in S} \left[P_a(s,s')R(s,a,s') + \gamma P_a(s,s')V_{H-1}^\pi(s')\right] && (2.4)\\ &= R^E(s,a) + \sum_{s' \in S} \gamma P_a(s,s')V_{H-1}^\pi(s') && (2.5)\\ V_0^\pi(s) &= 0 && (2.6) \end{aligned}$$

where $R^E(s,a)$ is the expected immediate reinforcement for taking action a at state s:

$$R^E(s,a) = \sum_{s' \in S} P_a(s,s')R(s,a,s'). \tag{2.7}$$

When $\gamma = 1$, the value function is *undiscounted*, otherwise, it is *discounted*. The decision to use discounting should be determined by the application. Some applications such as financial systems employ discounting to discount the future dollar values for inflation, while for many other applications discounting may not make sense.

For the infinite time horizon, the value of state s in the value function under policy π is

$$V^\pi(s) = \lim_{k \to \infty} V_k^\pi(s). \tag{2.8}$$

Some special concern must be addressed under the infinite time horizon: The future reinforcement must be discounted to guarantee that the cumulative future reward will not go to infinity. For applications where discounting does not apply, the value function can adopt the form of the *expected average future reinforcement* over time:

$$V^{\pi}(s) = E\left[\lim_{k\to\infty} \frac{\sum_{t=0}^{k} \gamma^t R(s_t, a_t, s_{t+1}) | s_0 = s}{k}\right]. \tag{2.9}$$

Solving MDPs with this type of value function is generally more complicated [Schwartz, 1993, Tadepalli and Ok, 1994].

The optimal value function, noted as V^*, is a function that maximizes the expected total future reward over all states. A policy that achieves this objective is an *optimal policy*, noted π^*. It should be noticed that there can be many optimal policies achieving the single optimal value function.

In general, a policy is described in the form of a sequence of decision rules over time: $\pi = (\delta_H, ...\delta_i, ...\delta_1)$, where δ_i is a decision rule for an i-step future. Each decision rule specifies an action for each state. A *stationary policy* is a policy in which a single decision rule is applied over time. The solution of an infinite-stage problem is a stationary policy. An important property of MDPs is captured by the following *turnpike theorem* [Denardo, 1982]: There exists a threshold n_0 such that an optimal policy for the infinite time horizon π^*_∞ is also an optimal decision rule for a more than n_0 step future.

The definition of the value function implies that the optimal value function can be computed from an optimal policy. On the other hand, the optimal policy can also be computed from an optimal optimal value function via a simple one-step greedy lookahead search. The resulting policy is called the *greedy policy* generated by the value function. Let us consider a discounted infinite MDP. At state s, for each possible action a, we compute its *"Q-value"* $Q(s,a)$, the expected total future reinforcement under action a:

$$Q(s,a) = R^E(s,a) + \gamma \sum_{s' \in S} P_a(s,s')V(s'). \tag{2.10}$$

Obviously, if $V(s') = V^*(s')$ for all s', then $Q(s,a)$ is also optimal (denoted $Q^*(s,a)$) because $Q(s,a)$ is computed based on the biggest value $V^*(s')$ of all possible next states s'. According to the *principle of optimality*, which says if the sub-sequences are optimal then the combination of the sub-sequences is also optimal, the action with the biggest Q-value $Q(s,a)$ is the best action at state s. Taking this action produces the largest expected total future reinforcement for s:

$$V^*(s) = \max_{a \in A(s)} Q^*(s,a). \tag{2.11}$$

Putting these two parts together, we have:

$$V^*(s) = \max_{a \in A(s)} \left[R^E(s,a) + \gamma \sum_{s' \in S} P_a(s,s') V^*(s') \right]. \tag{2.12}$$

Equation 2.12 is called the *optimality equation* or *Bellman equation* due to Bellman's contribution [Bellman, 1957]. This equation explicitly tells whether a value function is optimal. A value function is optimal if and only if the lefthand side of the equation is equal to the righthand side over all states.

For a finite MDP, however, the optimality equation is stage dependent:

$$V_H^*(s) \quad = \quad \max_{a \in A(s)} \left[R^E(s,a) + \gamma \sum_{s' \in S} P_a(s,s') V_{H-1}^*(s') \right]. \tag{2.13}$$

The decision rule for a given future step must be transformed from the corresponding optimal value function.

2.2 Standard Methods

Computational methods for solving MDPs and the complexity of these methods have been well presented in the literature [Howard, 1960, Puterman and Shin, 1978, Denardo, 1982, Ross, 1983, Bertsekas, 1987, Bertsekas and Tsitsiklis, 1989, Puterman, 1990, Puterman, 1994, Littman *et al.*, 1995b, Kaelbling *et al.*, 1996]. This section summaries the major approaches to solving MDPs and their characteristics. These approaches, particularly value iteration, are important for our concern because they provide the foundation of the current reinforcement learning methods. To describe

computational complexity of a method, for a MDP, we let N be the number of states, M be the maximum number of actions for a state over all states, K ($K \leq N$) be the maximum number of next states with non-zero probability for an action at a state, which reflects the dynamics of the process. Commonly, for a large MDP, $K \ll N$. We also let B be the maximum number of bits required to represent any component of the action model and the reward function.

2.2.1 Value Iteration

The first approach, *value iteration* (also called successive approximation), is essentially the dynamic programming method—the method resting on the principle of optimality. The value iteration algorithm, as shown in Figure 2.2, iteratively updates the value of each state by stage by assigning the righthand side value of the Bellman equation to the lefthand side until the stopping criterion is satisfied. The computation for each iteration is cheap: only requiring $O(KMN)$ time (arithmetic operations). The convergence of value function approximation has been theoretically proved, although it would require an infinite number of iterations for an infinite-stage problem. The convergence of an optimal policy (reaching the threshold n_0 of the turnpike theorem), however, only requires a finite number of iterations, and in practice, this convergence can be quite fast. Furthermore, it was proved that for fixed $\gamma < 1$, the value iteration algorithm takes time polynomial in M, N, and B [Tseng, 1990, Littman *et al.*, 1995b], which is *pseudo-polynomial* (because it relies on B). In the worst case the number of iterations grows polynomially in $1/(1-\gamma)$, so the convergence could be very slow when the discount factor is close to 1. For a finite MDP, the maximal number of iterations certainly will not exceed time horizon H.

In general, it is not obvious when to stop the algorithm. A useful stopping criterion is based on the *Bellman residual*—the maximal difference between the two sides of the Bellman equation. By this the performance of a greedy policy on the current value function is bounded. If the Bellman residual is less than ϵ, then the value of the greedy policy differs from the value of the optimal policy by no more than

Algorithm 1 (VI)

1. *Initialize* $V(s)$ *arbitrarily*
2. *Repeat until stopping criterion is satisfied*
   (a) *Perform Bellman update, for all* $s \in S$ *simultaneously set*
   $V(s) := \max_{a \in A(s)} \left[R^E(s,a) + \gamma \sum_{s' \in S} P_a(s,s') V(s') \right]$

Figure 2.2. The value iteration algorithm.

$2\epsilon\gamma/(1-\gamma)$ [Williams and Baird, 1993]. Under this stopping criterion, the resulting policy is called the ϵ-*optimal* policy.

There have been a number of variations to the standard value iteration algorithm for accelerating convergence. These variations include the Gauss-Seidel method, the Jacobi method, and the asynchronous method [Bertsekas, 1987, Puterman, 1994], which permits different value update orders and updating rules. Different variations win in different situations. Convergence of these variations have all been proved. In most situations, they converge more quickly than the standard methods.

2.2.2 Policy Iteration

Policy iteration gradually improves the current policy based on policy evaluation. The algorithm is presented in Figure 2.3. Policy iteration is good for small-scale infinite-stage problems. Unlike value iteration which would take infinite iterations (for infinite-stage problems) to converge to the optimal value function, policy iteration computes this function in a finite number of iterations. It was proved that an iteration of policy iteration dominates an iteration of value iteration by producing a better policy. However, each iteration of policy iteration is expensive, because it requires $O(N^3)$ time to solve a set of N linear equations. Policy iteration terminates in at most an exponential number of iterations because there are at most M^N distinct policies.

Algorithm 2 (PI)

1. *Choose an arbitrary policy* π
2. *Repeat until no further policy improvement*
   (a) *(Policy evaluation) Compute the value of* π *by solving* N *linear equations* $(\forall s \in S)$*:*
   $V(s) = R^E(s, \pi(s)) + \gamma \sum_{s' \in S} P_{\pi(s)}(s, s') V(s')$
   (b) *(Policy improvement) Obtain greedy policy, for* $s \in S$ *set*
   $\pi(s) := \arg\max_{a \in A(s)} \left[R^E(s, a) + \gamma \sum_{s' \in S} P_a(s, s') V(s') \right]$

Figure 2.3. The policy iteration algorithm.

The best complexity result of policy iteration is due to value iteration, thus, it is also a pseudo-polynomial algorithm under a fixed discount factor.

2.2.3 Modified Policy Iteration

The *Modified policy iteration* algorithm, shown in Figure 2.4, combines features of policy iteration and value iteration. Like value iteration, it begins with an arbitrary value function and relies on the stopping criterion to terminate the algorithm. Also, it performs the Bellman update once in each iteration. The computation for the Bellman update is conducted earlier in greedy policy computation in the same step (Step 2(a)), thus, it requires no additional work. Like policy iteration, the algorithm contains a policy improvement step and an evaluation step. However, the evaluation is not done precisely. Instead, it is carried out iteratively with a number of loops of value updates under the current policy. An "evaluation rule" determines how many loops are needed to evaluate the current policy for an iteration. The number of loops for value updating m could be fixed for all main iterations or chosen according to some prespecified schedule (there is some evidence suggesting that m should be increasing

Algorithm 3 (MPI) *Given an evaluation rule*

1. *Initialize* $V(s)$ *arbitrarily*
2. *Repeat until stopping criterion is satisfied*
   (a) *(Policy improvement) Obtain greedy policy and perform Bellman update, for* $s \in S$ *simultaneously set*
   $\pi(s) := \arg\max_{a \in A(s)} \left[R^E(s,a) + \gamma \sum_{s' \in S} P_a(s,s') V(s') \right]$
   $V(s) := \max_{a \in A(s)} \left[R^E(s,a) + \gamma \sum_{s' \in S} P_a(s,s') V(s') \right]$
   (b) *(Partial policy evaluation) Repeat for* m *iterations (*m *is determined by the evaluation rule)*
      i. *for* $s \in S$ *simultaneously set*
      $V(s) = R^E(s, \pi(s)) + \gamma \sum_{s' \in S} P_{\pi(s)}(s,s') V(s')$

Figure 2.4. The modified policy iteration algorithm.

in later iterations). [Puterman, 1994] claims that modified policy iteration always converges more quickly than value iteration, because it avoids maximization over the set of actions at each pass. This algorithm is also a pseudo-polynomial algorithm under a fixed discount factor.

As variations of the modified policy iteration algorithm, we can use other update orders such as Gauss-Seidel steps instead to evaluate the current policy.

2.2.4 Adaptive State Aggregation

The *adaptive state aggregation* algorithm [Bertsekas, 1987] differs from the modified policy iteration in that rather than using multiple value updates in the policy evaluation phase, it employs a state aggregation method to evaluate the current policy. Briefly, a state aggregation step works as follows: The state space S first is partitioned into a set of k disjoint subset $S_1, S_2, ..., S_k$ $(k \leq N)$, called *aggregate states*.

This comes up with a set of k linear equations with k unknowns. We then solve these k equations and finally perform value function update according to the solution of the equations.

We now consider how to do this in detail. Let V be the current value function and V^π be the value function that executes the current policy, which is the function that we want to compute in each policy evaluation step. Also, let P^π be the state-transition probability matrix under policy π (P^π is a $N \times N$ matrix). We postulate that over the state s of every aggregate state S_i the difference $V^\pi(s) - V(s)$ is constant. This leads us to generate a $N \times k$ transformation matrix M and a k-dimensional vector Y such that

$$V^\pi - V = MY, \tag{2.14}$$

where for each row of M, the ith column has value one at coordinates corresponding to states in S_i and all other entries have value zero.

We compute a value function V' with one-step value update under the current policy π, that is, performing the following update for all state $s \in S$:

$$V'(s) := R^E(s, \pi(s)) + \gamma \sum_{s' \in S} P_{\pi(s)}(s, s')V(s'). \tag{2.15}$$

We write this as equations

$$V' = R + \gamma P^\pi V, \tag{2.16}$$

where R denotes the vector of $R^E(s, \pi(s))$ values for all $s \in S$. Combining this with the following equations for V^π:

$$V^\pi = R + \gamma P^\pi V^\pi, \tag{2.17}$$

we have

$$(I - \gamma P^\pi)(V^\pi - V) = V' - V, \tag{2.18}$$

Then, let us multiply both sides with the transpose M' and use Equation 2.14, we have

$$M'(I - \gamma P^\pi)MY = M'(V' - V). \tag{2.19}$$

Algorithm 4 (ASA)

1. *Initialize* $V(s)$ *arbitrarily*
2. *Repeat until stopping criterion is satisfied*
   (a) *(Policy improvement) Obtain greedy policy and perform Bellman update, for* $s \in S$ *simultaneously set*
   $\pi(s) := \arg\max_{a \in A(s)} \left[R^E(s,a) + \gamma \sum_{s' \in S} P_a(s,s') V(s') \right]$
   $V(s) := \max_{a \in A(s)} \left[R^E(s,a) + \gamma \sum_{s' \in S} P_a(s,s') V(s') \right]$
   (b) *(Partial policy evaluation)*
      i. *Compute* V'*, for all* $s \in S$ *set*
      $V'(s) := R^E(s, \pi(s)) + \gamma \sum_{s' \in S} P_{\pi(s)}(s,s') V(s')$
      ii. *Delineate the aggregate states into* k *partitions (i.e, define* M*)*
      iii. *Solve for the vector* Y *in the equations*
      $M'(I - \gamma P^\pi) M Y = M'(V' - V)$
      iv. *Let* $Y' := MY$*, perform value update, for all* $s \in S$ *set*
      $V(s) := V'(s) + \gamma \sum_{s' \in S} P_{\pi(s)}(s,s') Y'(s')$

Figure 2.5. The adaptive state aggregation algorithm.

This set of equations can be solved for Y. Finally, using Equation 2.14, we obtain V^π.

We can conclude that if the difference $V^\pi(s) - V(s)$ is roughly constant over each aggregate state, then a good approximation for V^π will be obtained. The policy evaluation steps are described in Figure 2.5. In the last step of policy evaluation, rather than using Equation 2.14 to update the value function, we use

$$V(s) := V'(s) + \gamma \sum_{s' \in S} P_{\pi(s)}(s,s') Y'(s'). \tag{2.20}$$

This works better because it takes advantage of one more value update.

The key issue is how to identify the aggregate states in a way that the error between V^π and V is of similar magnitude on each aggregate state. One way to resolve this is to group states according to magnitude of the differences of values between V' and V. This partition strategy works well if the problem does not involve several ergodic classes. Otherwise, we should employ several value update steps as in modified policy iteration and then perform the state aggregation step. Furthermore, if there exists some particular structure in a MDP, we can delineate the aggregate states according to this structure and make the algorithms more efficient. Note that the aggregate states can change from one iteration to the next, this is the reason for characterizing this procedure as adaptive.

The adaptive state aggregation algorithm can be viewed as a method that bridges the gap between the efficiency of value iteration and the accuracy of policy iteration. In the one extreme that k is fixed to 1, the algorithm becomes value iteration. In the other extreme that k is fixed to N, it becomes policy iteration. This algorithm is also a pseudo-polynomial algorithm under a fixed discount factor.

2.2.5 Linear Programming

The problem of computing the optimal value function for an infinite-horizon MDP can be formulated as a linear program as follows [Littman *et al.*, 1995b].

Let V be a vector of real variables $(v_1, .., v_i, .., v_N)$ where i $(i = 1, 2, ..., N)$ corresponds to a state of a MDP and v_i corresponds to the value of state i. Find the value of all variables in V to maximize the sum

$$\sum_{i \in S} v_i$$

subject to the constraints

$$v_i \leq R^E(i, a) + \gamma \sum_{j \in S} P_a(i, j) v_j, \tag{2.21}$$

for all $i \in S$ and $a \in A(i)$.

Linear programming is a general technique. Its most popular algorithm is the simplex method, which runs very fast for most problems, although its worst case complexity is exponential to the size of a problem. [Denardo, 1982] pointed out that the simplex method performs exactly the same as policy iteration: it makes the same sequence of policy substitutions (pivots) as policy iteration does. Like policy iteration, linear programming is only good in solving small MDPs. Nonetheless, the most striking feature about linear programming for MDPs is that it comes up with so far the best computational complexity of MDPs. It proves that MDPs are solvable in polynomial time in N, M, and B, independent of the discount factor. This result is due to [Khachian, 1979] and [Karmarkar, 1984], who presented algorithms for solving rational linear programs in time polynomial in the number of variables and constraints as well as in the number of bits used to represent the coefficients.

Real-world applications involve large MDPs. When the problem size goes up to more than 10^6 states, unfortunately, all the above methods become impractical. One goal of reinforcement learning is to learn policies for large MDPs.

2.3 Scale Up: Learning Partial Policies

Machine learning operates under the assumption that there exist some regularities that can be exploited in most domain-specific problems. Learning is a biased activity. A supervised learning task commonly is biased to construct a simple (or general) classification rule under a given description language for a set of data. The learned classification rule actually makes an inductive leap from the original data. A good learning bias should reflect the regularities of the data space. If such a bias is employed, the learned inductive rule should predict well for new data.

Likewise, the reinforcement learning approach to large MDPs is concerned with constructing policies for problems that contain certain regularities. In many application problems, there can be some "simple" policies that work well enough for the application. While it is impossible to construct a complete optimal policy for large problems, it is important to develop methods that are able to discover such simple

policies. A simple policy can be a partial policy—a policy that only covers a subset of the total state-space. It can also be a generalized policy—a policy that generalizes over states such that similar states have similar values and similar best actions. A generalized policy is not recorded with a table, rather, it employs a function approximation method to map states into a policy. We will discuss generalized policies in the next section.

A partial policy makes sense for many applications. For example, suppose we want to drive from Corvallis to the Portland airport and we have a map describing all cities in Oregon and highways connecting these cities. Certainly, we do not need a policy that covers all cities such as Eugene because Eugene is south of Corvallis and Portland is north. Another example is about job-shop scheduling. In a search space that contains about 10^{30} possible schedules for a scheduling problem, we may find that some parts of state space may contain much more high-quality solutions than other parts. We may also find that some states are good starting states for the search. These regularities can help construct very effective search policies. Obviously, we do not need a policy to cover all states, because it is not necessary and often impossible to search the whole state space. Therefore, our goal becomes to construct a good search policy that can lead us to search in the right region and get a high-quality solution quickly. Furthermore, because of having knowledge about starting states, we can construct a policy that considers always starting search from a particular state. A partial policy is much cheaper to compute than a complete policy.

The partial-policy learning methodology can also be applied to generate a complete policy. For problems that have structures with weakly-coupled components or continuity structures such as geographical continuity (such structures often exist in real-world problems), we can decompose the whole problem into a set of subproblems. We can learn a set of partial policies where each policy works for a subproblem and then combine them to be a whole policy to solve the whole problem. This usually is much more efficient than learning a policy as a whole. Several researchers have reported important results about learning using a decomposition of a state space [Kaelbling, 1993, Dayan and Hinton, 1993, Dean and Lin, 1995]. Like state aggrega-

tion, problem-decomposition takes advantage of problem structure to group states. But it differs from state aggregation in that it pays attention to each individual sub-component and makes partial policies for them.

The key reinforcement learning methodology is fllearning. It actively explores a state space, by trial-and-error, to learn a policy. The learning process recognizes and explores the interesting parts of the state space and learns a policy only for the states in these parts of the space. Active learning is often performed along with a problem-solving process. By this, we simultaneously search solutions to a problem and gain search control knowledge. This learning paradigm is called *on-line* learning.

2.3.1 Real-Time Dynamic Programming

Real-time dynamic programming (RTDP) [Barto *et al.*, 1995] is an active on-line learning approach to compute the value function. In contrast to standard value iteration methods that systematically compute the value function, RTDP employs a controller to explore the state space. At any given time t, the controller is in one state, named s_t. The controller executes a greedy action a_t—an action which is greedy with respect to the current value function. It then incurs the immediate reward r and the environment switches to state s_{t+1}. By the time the next action a_{t+1} to be selected, the algorithm performs the Bellman update for the states in a scope S' that the controller can foresee by a type of lookahead search. In the simplest case, $S' = \{s_t\}$. S' might consist of the all states that can be visited from s_t in some fixed search depth.

Value update in RTDP actually follows a kind of asynchronous dynamic programming by which at any time, the Bellman update is performed on a subset of states. Therefore, RTDP converges as asynchronous dynamic programming if the real-time mechanism does not rule out the chance for the controller to visit all states. The algorithm certainly converges for an ergodic problem where every two states can visit each other in the long run. For general problems, we can employ multiple trials (with restart) to ensure that each state is visited infinitely often. A trial terminates

Algorithm 5 (RTDP)

1. *Set* $t = 0$ *and let the controller be in an arbitrary state* s_0*; Initialize* $V(s)$ *arbitrarily*
2. *Repeat until stopping criterion is satisfied*
   (a) *Perform a lookahead search from* s_t *to obtain a set of states* S'
   (b) *Perform Bellman update, for* $s \in S'$ *simultaneously set* $V(s) := \max_{a \in A(s)} \left[R^E(s,a) + \gamma \sum_{s' \in S} P_a(s,s') V(s') \right]$
   (c) *Let the controller perform a greedy action and move into state* s_{t+1}*, set* $t := t + 1$
   (d) *If the restart criterion is satisfied, then restart a trial, set* s_t *to an arbitrary state.*

Figure 2.6. The real-time dynamic programming algorithm.

when a restart criterion is satisfied. This criterion can be "terminate a trial with a fixed amount of time" or "terminate when an absorbing state is reached". The trial-based RTDP algorithm is shown in Figure 2.6. [Barto *et al.*, 1995] presented the convergence proof of this algorithm.

A comparable and closely related work to RTDP is learning real-time A^*, *LRTA** [Korf, 1990]. RTDP generalizes LRTA* to stochastic problems and includes the option of updating values of many states at a time. RTDP also relates to "universal planning" [Chapman, 1989, Ginsberg, 1989, Schoppers, 1987, Schoppers, 1989].

RTDP can be employed to learn a partial policy for a problem with a fixed starting state. In this situation, some exploration strategy must be applied for the learner in order to make it visit various states. Exploration will be discussed later.

2.3.2 Temporal Difference Learning

Temporal difference (TD) learning is a learning approach based on examining a series of state transitions. This series is normally arranged into a set of trials—each trial starts in some state and typically ends in an absorbing state. TD learning provides a fundamental idea for reinforcement learning. Learning is aimed at reducing the difference between the estimated values of a pair of adjacent states (s, s'). A simple TD update, which is noted as TD(0) [Sutton, 1988], works as follows:

$$V(s) := V(s) + \alpha \left[R(s, a, s') + \gamma V(s') - V(s) \right], \tag{2.22}$$

where α is the learning rate.

TD learning is a learning setting for the prediction problem, which computes the value function under a *fixed* policy (i.e., there is only one action for every state). In other words, it is a learning method to approximate the long-term probabilities of a *Markov chain*. TD(0) is a special case of a more sophisticated learning algorithm named Q-learning [Watkins, 1989]—a learning algorithm that learns a policy without knowing the action model. TD(0) works for a model with one action, while Q-learning works for an arbitrary number of actions. Due to the convergence proof of Q-learning provided by [Watkins, 1989], TD(0) also converges for predicting the value function.

Nonetheless, TD learning also works in control problems to produce an optimal policy. TD learning can work on-line *"on-policy"* like RTDP by generating the sequence of states following the current policy—the policy greedy with respect to the current value function. Under this condition, the key difference between RTDP and TD learning is their ways of updating the value function. An RTDP update uses the Bellman update based on the best action according to the current policy while a TD update uses the value of the resulting state. When working on a deterministic problem, the TD update exactly performs the Bellman update, therefore, in this case, TD learning is essentially the same as RTDP. RTDP which updates one value per step actually is on-line on-policy TD learning with learning rate $\alpha = 1$.

A general method for applying TD learning for control problems is the *adaptive heuristic critic* (AHC) algorithm [Barto *et al.*, 1983]. AHC is a learning analog of policy iteration in which TD learning is employed to compute the value function for a policy. In AHC, a precise policy determination step is not adopted, instead, a learning mechanism is developed to learn the mapping from a state and its value to the action to perform at the state. AHC provides a powerful method to solve large MDPs. It can also be applied to solve problems with unknown action models.

TD(0) is a special setting of a more general learning algorithm TD(λ) with $\lambda = 0$. While TD(0) looks only one step ahead when adjusting values, TD(λ) makes use of an *eligibility trace* to make learning more efficient. As introduced by Klopf (1972), the eligibility trace works as follows. Each time a state s_t is visited, a short-term memory process, a trace, is initiated, which makes the states in the trace also eligible for learning. The strength of learning for the states in the trace decays exponentially over time. TD(λ) updates values for every state s in the trace as follows:

$$V(s) := V(s) + \alpha \left[R(s, a, s') + \gamma V(s') - V(s) \right] e_t(s), \tag{2.23}$$

where $e_t(s)$ for every state s is updated based on the recency factor λ ($0 \leq \lambda \leq 1$) and the discount factor γ as follows:

$$e_t(s) = \begin{cases} \gamma \lambda e_{t-1}(s) + 1, & \text{if } s = s_t \\ \gamma \lambda e_{t-1}(s) \quad\;\; , & \text{otherwise} \end{cases} \tag{2.24}$$

Here, s_t is the current state. Initially, $e_0(s)$ is set to 0 for every state s. [Sutton, 1988] and [Dayan, 1992] proved the convergence of TD(λ) for general λ.

Recently, [Singh and Sutton, 1996] presented a new type of trace, the *replacing* eligibility trace, and showed that it performs better than the above accumulating trace. The replacing trace differs the accumulating trace in that it sets $e_t(s) = 1$ when s is revisited. Implementing the eligibility trace is computationally expensive. [Cichosz and Mulawka, 1995] presented the *truncated* TD learning method to overcome this problem.

For undiscounted problems, when $\lambda = 1$, TD(λ) is the same as supervised learning which uses "actual values" as output for updating state-values. The actual value for state s_t in a trial is the sum of all reinforcements received from the current state s_t to the final state s_m:

$$z_t = \sum_{k=t}^{m-1} R(s_k, a, s_{k+1}). \tag{2.25}$$

TD(1) actually performs the following update:

$$V(s_t) := V(s_t) + \alpha \left[z_t - V(s_t) \right]. \tag{2.26}$$

Sutton (1988) showed that on a simple random-walk prediction problem, $\lambda = 0$ works the best. $\lambda = 1$ has trouble coming up with correct predictions.

It is worthwhile to mention that Samuel's checker-playing program [Samuel, 1959] probably was the first system to use a kind of TD learning method. Samuel's program learned an evaluation function based on the difference of estimates between successive board positions. Other earlier TD related works include Holland's bucket brigade and profit-sharing plan [Holland, 1986] and Grefenstette's RUDI system [Grefenstette, 1988].

2.3.3 Sarsa and Q-learning

The *Sarsa* algorithm is a variation of $TD(\lambda)$ [Rummery and Niranjan, 1994, Sutton, 1996, Singh and Sutton, 1996]. Sarsa differs from $TD(\lambda)$ in that it employs state-action pairs to learn the *Q-function* (the mapping from state-action pairs to Q-values)

rather than states to learn the value function. The name "Sarsa" comes from the 5-tuple (s, a, r, s', a'), (or STATE, ACTION, REWARD, RESULTING-STATE, NEXT-ACTION), for update. A Sarsa-update works as follows:

$$Q(s,a) := Q(s,a) + \alpha \left[R(s,a,s') + \gamma Q(s',a') - Q(s,a) \right] e_t(s), \tag{2.27}$$

Sarsa is a learning setting for stochastic control problems. An on-line on-policy Sarsa which always chooses actions greedy to the current Q-values will approximate the optimal policies. Sarsa with $\lambda = 0$ looks similar to Q-learning. The key difference between these two methods is that Sarsa strictly uses the information provided in a temporal sequence for value update while Q-learning uses the predicted value of the state, which is the maximum of the Q-values over all admissible actions in the state. A Q-update works as follows:

$$Q(s,a) := Q(s,a) + \alpha \left[R(s,a,s') + \gamma V(s') - Q(s,a) \right], \tag{2.28}$$

where

$$V(s') = \max_{a \in A(s')} Q(s',a). \tag{2.29}$$

Both Sarsa and Q-learning completely rely on trial-and-error experience to learn to control. However, when there is an action model available that describes the probabilities of state transitions, these two methods may not work as well as RTDP, because they do not take advantage of the information provided by the action model, and also because the Q-value table is much bigger than the state-value table.

2.4 Scale Up: Generalization

By generalization we mean that instead of using tables to store a value function, a Q-function, or a policy, function approximation by a compact representation is employed. Popular representations include various types of neural networks, CMAC (Cerebellar Model Articulation computer) [Albus, 1981, Sutton, 1996], decision trees, and local memory-based representations such as nearest neighbor methods. Generalization is important and useful for reinforcement learning in many situations as follows:

- The state space is often very large. In many cases, the partial-policy learning approach does not work.
- A problem-space is continuous. Many real-world problems involve a continuous problem-space, where continuous variables are used to describe states.
- The table-based representation is inefficient. There may be a more compact representation that also achieves an optimal policy or a very good policy. Typically, we generally expect similar states to have similar values and similar best actions.
- Learning is aimed at solving a set of instances of a problem. For example, we want to learn a problem-solving policy for solving the traveling-salesman problem. We learn a policy by investigating a few instances of the problem and want to make this policy work for all instances of the problem.

The function approximation methods for the above compact representations were originally designed in the supervised learning setting. When applied to reinforcement learning, these methods may encounter several pitfalls. The key problem is that in reinforcement learning based on dynamic programming or temporal difference learning, the estimates for the value function are built on the basis of other estimates. [Sutton, 1996] stated:

> *Because the estimates are imperfect, and because they in turn are used as the targets for other estimates, it seems possible that the ultimate result might be very poor estimates, or even divergence.*

Several results have been reported which showed the cases that function approximation with reinforcement learning is unstable [Thrun and Schwartz, 1993, Tsitsiklis, 1995, Baird, 1995, Bradtke, 1993, Boyan and Moore, 1995, Schraudolph *et al.*, 1994]. In theory, reinforcement learning algorithms converge to optimality only when lookup-tables are used. [Thrun and Schwartz, 1993] pointed out that an

important source of the failures is due to a systematic overestimation of the value function by the "max" operation according to the definition of the value function. Each Bellman backup could convert zero-mean "function approximation noise" into a positive-mean noise. [Tsitsiklis, 1995], [Baird, 1995] and [Boyan and Moore, 1995] showed some simple examples where value function errors grow arbitrarily large when function approximation with reinforcement learning is used. Both [Tsitsiklis, 1995] and [Boyan and Moore, 1995] have suggested using actual values as in the TD(λ) algorithm when $\lambda = 1$ to overcome this problem.

Fortunately, many pitfalls can be avoided by a careful design of reinforcement learning systems. [Gordon, 1995] showed in theory that several methods such as nearest-neighbor will not diverge. [Sutton, 1996] reported that the unstable phenomena that appeared in [Boyan and Moore, 1995]'s examples can be resolved using an on-line Sarsa learning approach based on a sparse-coarse-coded CMAC representation. [Baird, 1995] presented the *residual algorithm* which employs a combined current gradient and "residual gradient" to update weights in a neural network. The residual gradient is the current gradient subtracted by the discounted previous gradient. This algorithm can be stable but slow. Lin [Lin, 1992, Lin, 1993] demonstrated that reinforcement learning with neural networks is able to learn to accomplish various tasks for a simulated robot.

More importantly, applications in large domains have shown great success [Tesauro, 1992, Crites and Barto, 1996, Zhang and Dietterich, 1995, Zhang and Dietterich, 1996]. Tesauro (1992) applied the TD(λ) algorithm using a neural network to approximate the value function. The resulting TD-gammon system is now far better than any other computer program and competitive with the best human players. [Crites and Barto, 1996] presented a multi-agent Q-learning neural network approach to successfully construct a policy for elevator control. The work reported here presents further encouraging results on a combination of back-propagation neural networks and TD learning.

Among various different types of function approximators, neural networks appear most popular in application. A nice result about the neural network approach is that

TD(λ) as originally presented in [Sutton, 1988] has been well combined with the back-propagation method. Other noticeable generalization methods include the tree-based *G-learning* algorithm [Chapman and Kaelbling, 1991], the *kd-tree* method [Moore, 1990, Moore, 1991], and the parti-game algorithm [Moore, 1994].

2.5 Improving Learning Performance

Reinforcement learning has been observed to be slow in its convergence to an optimal or near-optimal policy. By improving learning performance we mean to speedup this convergence or to improve the quality of the constructed policy in a given reasonable amount of training. This section discusses some important results about how to improve the performance of reinforcement learning.

2.5.1 Exploration

A reinforcement learning agent faces two opposing objectives: act to gain information and act to gain rewards. On the one hand, the agent must choose an action to gain reward in each step in order to achieve the largest long-term cumulative reward. This means that actions should be selected greedily according to the best current control knowledge—the current estimated value function. This mechanism, named *exploitation*, makes the agent focus on a local interesting part of the state space, normally leading to more rewards in the short term. On the other hand, *exploration* in the space space is necessary in order for the agent to gain information for further improving the long-run policy. However, an exploration step could encounter a negative reward. Excessive exploration is not efficient, we can not conduct large amounts of exploration if we have only limited time. Proper control of the tradeoff between exploration and exploitation is important in order to construct an efficient learning method.

There are generally two classes of exploration strategies. One class is the random-walk based strategy, which combines random exploration with exploitation. The

strategies can be performed in a variety of ways such as pure random-walk or using the Boltzmann distribution based on the current value function. The other class is more information-directed, named "directed" exploration, which memorizes information about the learning process and utilizes this information for directing exploration. [Thrun, 1992] compared the following directed exploration methods:

- *counter-based exploration:* count the number of visits to each state and go to less explored states.
- *recency-based exploration:* similar to the eligibility trace in Sutton's TD(λ).
- *error-based exploration:* calculate the change of the value estimates and choose an action that could create the biggest change.

Other information-directed exploration methods include the interval estimation method [Kaelbling, 1990], the exploration bonus in Dyna [Sutton, 1990], and exploration based on a priority queue [Moore and Atkeson, 1993].

For a deterministic problem, while learning time with the random-walk based exploration is expected to scale exponentially in the size of state space in order to come up with a complete policy [Whitehead, 1991], most directed exploration methods only involve polynomial time [Thrun, 1992]. Evidently, for small problems directed exploration is more efficient. However, the directed exploration methods have difficulties to scale up for larger problems because these methods generally allocate one counter of information for each state or state-action pair. To make them practical for large problems, generalization should be employed to process this exploration information.

2.5.2 Teaching

For many domains, there are probably good teachers available to give some important lessons for training or to tell in some ways what decision rules are good. Having teachers to guide learning should be the most effective way to accelerate the learning process. A number of forms of teaching methods have been developed for reinforcement learning [Lin, 1991, Lin, 1992, Whitehead, 1991, Utgoff and Clouse, 1991,

Clouse and Utgoff, 1992]. [Lin, 1991] provided an approach by giving the learner *lessons*, which are human-generated sequences of experiences in (s, a, r, s') quadruples that begin with an initial state and proceed to a goal state. To speed learning further, the quadruples are presented to the system in temporally backward order. [Utgoff and Clouse, 1991]'s approach allows the learner to query an expert as to the correct action to take when it has low confidence in its own decisions. [Whitehead, 1991] presented two methods. One is "Learning with an External Critic", where an external agent provides an immediate critic to the learner for its action. This critic is used to learn a "biasing" function over state-action pairs. Decision making then is based on a combination of the Q-value and the bias value. The other method is "Learning by Watching", where the learner gains experience by observing the behaviors of other learners.

The guidance can be either taken on-line or taken in advance as *prior* knowledge. Several researchers have reported that teaching methods combined with some random exploration can scale very efficiently to solve difficult problems [Lin, 1991, Whitehead, 1991].

2.5.3 Experience Replay

As discussed earlier, generalization with neural networks has been the most popular methodology for solving large-scale problems. Experience replay is a very useful technique to improve learning performance under this situation, because with training, a network would tend to forget patterns trained in the beginning. The experience obtained by the trial-and-error procedure is often costly, and some experiences may only appear rarely. To effectively train a neural network, experiences should be reused in an effective way. [Lin, 1992] suggested replaying experiences that follow the current policy. The work reported in this thesis replays good experiences.

2.6 Learning to Search A State Space

In this section, we focus on two important problems related to applying MDPs to state-space search. The first problem is concerned with the design of the reinforcement function. We classify different types of state-space search problems and discuss how to design the reinforcement function according to the characteristics of a search problem.

We noticed another important problem. Unlike the agent learning problem where the agent is willing to stay in a cycle of life as long as the cycle provides the largest cumulative reward, in state-space search we never want to stay in a cycle. With the objective of finding the goal quickly, an ideal search method must employ a policy that never takes a loop. However, an imperfectly learned policy may loop. To make a learned policy useful in search, we must develop some mechanism for the policy to escape loops.

2.6.1 Types of State-Space Search Learning Tasks

In general, problem-solving tasks can be classified into one of the following two classes: decision problems and optimization problems. With the goal of determining whether there is a solution for a task, the decision problem cares about finding one solution. On the other hand, the optimization problem seeks an optimal solution. While finding an optimal solution could be very expensive in time for complex or large-scale problems, in practice, people accept some kinds of satisfactory solutions, which may not be optimal. Whenever search reaches a state with a satisfactory solution, the solution is returned and the search process can terminate.

In the context of state-space search, a satisfactory solution is a goal state in search. Normally, a goal test is defined, and thus all states satisfying this test become the goal states. The goal test can be viewed as a combination of "*hard*" constraints that must be satisfied for a solution. In contrast, the optimization criterion for the optimization problem can be viewed as a combination of "*soft*" constraints which measure the quality of a solution.

There are generally three types of objectives for state-space problem-solving tasks. The first type is concerned with finding a solution for the decision problem or finding a satisfactory solution for the optimization problem. (We can turn the objective around and attempt to prove that there is no solution.) The key question for this task is how to find such a solution quickly. The objective can be generally stated as finding the shortest path to a goal state. The performance of a search method for this task is measured by the number of search steps required to reach the goal. The performance can also be measured by the search time spent to reach the goal, which actually is a weighted shortest path.

The next two types of objective are concerned with optimization problems: one is only concerned with solution quality and the other is concerned with both solution quality and search speed. For the former, the performance of a search method is measured by the solution quality in terms of the optimization criterion. While there exist algorithms such as exhaustive search which are guaranteed to find an optimal solution, this type of task does not present much interest to researchers. For the second type of objective, the performance of a search method is measured by both the solution quality and the length of the search path. A common measure is the quality of the solution obtained under a given search time or search steps. The more time search spends, the better solution it will obtain.

Now we look at the characteristics of the problem-solving state-space. A search space is called *acyclic* if there is no cycle in the space, which means that in any state s in a space, performing any sequence of operations will not make search come back to s. Such a state space represents a *directed acyclic graph* (DAG) and is viewed as a *progressive* state space where search can be progressively conducted in the space from a start state toward the final states by taking any sequence of operators. In an acyclic finite state-space, at least one state is absorbing (the goal state). A state space with absorbing states is also called a *goal-directed* state space.

A search space is called *cyclic* if there exist cycles in the space. For many problem-solving tasks, a common design of operators may not be able to completely eliminate cycles in the space. In other situations, there might be some "dead end" states. When

search reaches such a state, it must perform some kind of "additional operators" such as "backtracking" and "backjumping" to continue the solution-seeking procedure. Search in a cyclic state-space is more complicated than in a space without cycles, because it involves making decisions about how to avoid a cycle, whether to take a cycle, and when to leave the cycle (in case search must come to a cycle). Failure in making such decisions could make the search process extremely inefficient.

Furthermore, in many real-world applications, we may face a problem where it is not practical to specify a goal test. Rather, we would want a search algorithm to find the best solution it can in a limited number of search steps or amount of execution time. In this situation, search can end up at any state when the time bound is reached. When search terminates, it returns the best solution found so far. A common approach to handle this situation is to consider every state in the state space to be a solution, this means that no hard constraints are employed. Note that a problem space that does not have a goal state must be cyclic (otherwise there must be at least one specified terminal state, which is a goal).

In summary, state-space search learning problems can be characterized with two dimensions: one in terms of the objective of a problem and the other in terms of the characteristics of the search space. There are three selections for the objective of a problem: (1) the shortest path, (2) the optimal solution, and (3) the shortest path to the optimal solution. There are three possibilities for the characteristics of the search space: (1) acyclic space, (2) cyclic space with absorbing states, and (3) cyclic space without absorbing states. In combination, there are nine possible configurations.

2.6.2 Designing Reinforcement Functions

We now consider how to design reinforcement functions for each of these configurations.

A. Acyclic space:

1. **Shortest path:** In the acyclic space mode, let us first consider the objective of finding the shortest path. The reinforcement function can be simply designed as follows. For any action in a state, the reinforcement function provides the negative of the cost of an action as the immediate reward.

2. **Best solution:** For the objective of finding the optimal solution, the reinforcement function is defined as follows. For any action that results in a solution, the immediate reward is the quality of the solution. Otherwise, the immediate reward of an action is set to 0.

3. **Shortest path to the best solution:** For the objective of finding the shortest path to the optimal solution, the reinforcement function is defined as follows. For any action that results in a solution, the immediate reward is the quality of the solution subtracted by the cost of the action. Otherwise, the immediate reward of an action is the negative of the cost of the action. Since the reinforcement function combines the solution quality and the cost of each action, we must define the action-cost small enough to make the cumulative reward emphasize the solution quality.

B. Cyclic space with absorbing states:

1. **Shortest path:** We now look at a cyclic space that contains absorbing states. For the objective of finding the shortest path, the reinforcement function would be the same as the one for an acyclic space, which defines the negative action-cost as the immediate reward.

2. **Best solution:** For the objective of finding the optimal solution, besides defining solution quality as the immediate reward, we should also define the action-cost. An action-cost helps search avoid being trapped into a cycle. If a set of neighborhood states are all able to reach an optimal solution, then using zero action-cost will fail to make a policy for the neighborhood states to determine how to choose an action because they all have the same value.

Table 2.1. The reinforcement function.

Objective	*State Space*		
	Acyclic	Cyclic w/Absorbing	Cyclic w/o Absorbing
Shortest Path	Action-Cost	Action-Cost	N/A
Best Solution	Solution-Quality	Solution-Quality + Action-Cost	Solution Quality-Gain
Best Solution + Shortest Path	Solution-Quality + Action-Cost	Solution-Quality + Action-Cost	Solution Quality-Gain + Action-Cost

3. **Shortest path to the best solution:** For the objective of finding the shortest path to the optimal solution, the reinforcement function would be the same as the one for an acyclic space, which defines both solution quality and the negative action-cost as the immediate reward.

C. Cyclic space without absorbing states:

Finally, consider the situation where no absorbing states exist in a cyclic state space. Because no goal states exist, the thing that determines the termination of a process is the process time horizon. Therefore, this setting only works for the optimization problem with the following two objectives: the best solution that could be obtained in a given number of search steps and the shortest path to the best solution. The two objectives have the same effect except that the latter is capable of finding the best solution with a path shorter than the given time horizon.

1. **Best solution:** For the objective of finding the best solution, to design the reinforcement function, we must consider that any state could be treated as a solution. We can define the solution *quality-gain* as the immediate reinforce-

ment, which is the difference in quality between the resulting solution and the current solution under an action. To calculate the value function, the value of a state s should initially be set to the solution quality of s. Unlike in the situation where a space contains absorbing states, here the action-cost is not necessary to set, because the immediate reward has been designed for all states.

2. **Shortest path to the best solution:** For the objective of finding the shortest path to the optimal solution, accordingly, the reinforcement function would be the solution quality-gain combined with the action-cost.

Table 2.1 summarizes the reinforcement functions for the nine configurations.

An important fact about the state-space search learning problems using these reinforcement functions is that future reinforcements should *not* be discounted. For example, in order to find the shortest path to an optimal solution in a goal-directed search, we must take quality of a final solution as the most important reinforcement, so the final reinforcement should not be discounted. We also should not discount the cost of future search steps, because future steps in general are as important as the current step.

To determine the criterion of accumulating future reinforcements for measuring a policy, we also need to consider the time horizon for search. We have discussed this for problems without goal states, where a finite horizon is employed. We now consider other situations. For an acyclic state space, it is convenient to use the infinite horizon. Because search is guaranteed to terminate in a finite number of steps, the value function can not go to infinity. The benefit of using the infinite horizon is that we do not need to worry about the bound for the number of search steps, accordingly, a stationary policy can be employed.

For a cyclic space that contains goal states, it is important for search to escape loops. If we have some mechanism to do this and guarantee that search can terminate with a finite number of steps, then again, we can employ the infinite horizon. We will discuss methods for escaping loops in the next section.

2.6.3 The Looping Policy Problem

The *looping policy problem* says that in some situation where search follows a policy constantly, it can be trapped into an infinite loop. This problem can occur in the following situations:

- **When the reinforcement function is not carefully designed:** For example, for a cyclic state-space, if we do not define a cost for each search step, search could go to an infinite loop.
- **During the learning process:** Policy is gradually improved during learning. An imperfect policy during learning can cause loop.
- **When function approximation is applied:** In this situation, we are not guaranteed to find an optimal policy. However, if we can work out good input representation and function approximation methods, the loop problem could possibly be solved.
- **When we work for a problem containing a set of instances:** In this situation, we must employ function approximation to come up with a generalized policy that works for all instances in the problem. Because learning is conducted only on a subset of all instances, in general, we can not guarantee that the learned policy will not create an infinite loop for new problems.

Loops are detrimental for both learning and execution. An infinite loop during learning and execution is nothing but wasting time. During learning, if learning observes only the experiences in a small cycle, it would not be able to produce a useful problem-solving policy. During execution, an infinite loop prevents search from getting to the goal.

We now consider how to solve the looping policy problem. A policy is said to be *static* if the action to be chosen for a state during execution is determined before execution. A policy is said to be *dynamic* if the action must be determined during execution. Certainly, a dynamic policy is non-stationary because for the same state,

different actions can be taken at different times. To solve the looping policy problem, we develop the following dynamic non-stationary policy method.

Instead of specifying a single action for a state as in a decision rule of a static policy, the dynamic policy specifies an *if-then* structured decision rule. This rule can be represented as a priority list of choices, $(a_1, a_2, ..., a_n)$, where a more preferred choice is listed earlier. If the first choice does not create a loop, then take this choice, otherwise consider the second choice. If the second choice also creates a loop, we then consider the next choice. Usually, we put the last choice as backtracking to the preceding state to make a search process continue. A typical rule considers best two choices as well as backtracking: $(a_1, a_2, backtrack)$, which can be explicitly written as

```
If action a1 does not create a loop, then take a1.
Else if action a2 does not create a loop, then take a2.
Else backtrack.
```

Most reinforcement learning methods learn the value function of an optimal policy rather than directly learn the policy. Once the value function is learned, we can transform it into an optimal policy via a simple one-step lookahead search. However, the learned value function may not be perfect, thus the resulting policy may loop. We can use the dynamic non-stationary policy method to overcome this problem. Rather than obtaining a greedy policy in the form of a single action for each state, a greedy policy for a value function can be in the form of the if-then structured dynamic policy:

```
Choose the best action according to the value function if
     it does not create a loop.
Else choose the second best action according to the value
     function if it does not create a loop.
Else backtrack.
```

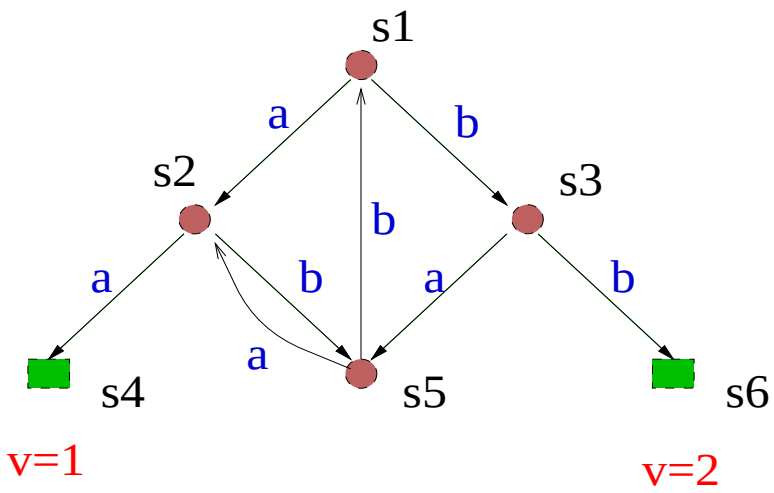


Figure 2.7. A Simple State-Space Search Problem. This is an optimization problem. The goal is to find the optimal solution `s6` quickly. The cost of each action is -0.1. The reward of `s6` is 2, and the reward of `s4` is 1.

2.6.4 An Example

We finish this chapter by showing a simple example to make the context presented in the previous sections more clear.

Example 2.1 *Figure 2.7 shows a simple deterministic decision process with 6 states. There are 2 absorbing states:* `s4` *and* `s6`*. Each non-absorbing state has two actions:* `a` *and* `b`*. This is an optimization problem. The goal is to find the optimal solution quickly. Here the optimal solution is state* `s6`*, which has a quality value 2, larger than the value of the other solution—state* `s4`*, which is 1. The cost of each action is set to be 0.1. The deterministic state transitions are shown in the diagram. For example, taking action* `a` *in state* `s1` *makes the process move to state* `s2` *and at the same time the process receives reward -0.1 as the cost of the action. Because the problem has absorbing states, the infinite time horizon is employed. Also, the process is undiscounted.*

Table 2.2 shows the optimal stationary policy for the problem, which is choosing action `b` at all states. When search follows this policy, it finds the optimal solution with minimum amount of search starting from any state. For example, when search

Table 2.2. The optimal policy.

State	π^*	v^*	Path*
s1	b	1.8	s1 $\rightarrow$ s3 $\rightarrow$ s6
s2	b	1.6	s2 $\rightarrow$ s5 $\rightarrow$ s1 $\rightarrow$ s3 $\rightarrow$ s6
s3	b	1.9	s3 $\rightarrow$ s6
s5	b	1.7	s5 $\rightarrow$ s1 $\rightarrow$ s3 $\rightarrow$ s6

Table 2.3. A bad policy.

State	π	v^π	Path$^\pi$
s1	a	$-\infty$	s1 $\rightarrow$ s2 $\rightarrow$ s5 $\hookrightarrow$ s2 $\hookrightarrow$ s5 $\hookrightarrow$ $\cdot\cdot$ (Loop)
s2	b	$-\infty$	s2 $\rightarrow$ s5 $\hookrightarrow$ s2 $\hookrightarrow$ s5 $\hookrightarrow$ $\cdot\cdot$ (Loop)
s3	b	1.9	s3 $\rightarrow$ s6
s5	a	$-\infty$	s5 $\rightarrow$ s2 $\hookrightarrow$ s5 $\hookrightarrow$ s2 $\hookrightarrow$ $\cdot\cdot$ (Loop)

starts in state `s1`, it first takes action `b` and goes to state `s3`. It then takes action `b` at state `s3` and reaches the solution `s6`. The optimal value function v^* and the optimal paths for all starting states are also shown in the table.

What will happen if the process has a bad stationary policy? Table 2.3 presents a policy which recommends taking action `a` in state `s1` and `s5` and taking action `b` in state `s2` and `s3`. This policy leads to an infinite loop when search starts with state `s1, s2`, and `s5`. The value function following this policy decreases to negative infinity. The "$\hookrightarrow$" sign in a path shows a step which goes to a state that has been visited before.

Table 2.4 presents a dynamic policy for this problem. The first choice of actions for all state in this policy exactly follows the above bad policy. However, in the dynamic policy, for state `s1` and `s4`, the other available action is used as the second

Table 2.4. A dynamic policy.

State	π	v^π	$\texttt{Path}^\pi$
s1	(a, b, `Backtrack`)	1.4	s1 $\rightarrow$ s2 $\rightarrow$ s5 $\hookrightarrow$ s2 $\hookrightarrow$ s1 $\rightarrow$ s3 $\rightarrow$ s6
s2	(b, `Backtrack`)	1.6	s2 $\rightarrow$ s5 $\rightarrow$ s1 $\rightarrow$ s3 $\rightarrow$ s6
s3	(b)	1.9	s3 $\rightarrow$ s6
s5	(a, b, `Backtrack`)	1.5	s5 $\rightarrow$ s2 $\hookrightarrow$ s5 $\rightarrow$ s1 $\rightarrow$ s3 $\rightarrow$ s6

Table 2.5. Value function approximation with a linear function.

State	L	V	$\hat{v} = \frac{100-L-V}{10}$
s1	80	8	1.2
s2	82	2	1.6
s3	80	5	1.5
s5	82	1	1.7
s4	90	0	1.0
s6	80	0	2.0

choice. Also, `Backtrack` is used as the last choice (not necessary for `s3` because its action `b` reaches the optimal solution). Both policies can be generated from the same value function. We see that this dynamic policy performs much better than the previous static policy. This policy is able to come up with the best solution `s6` for any starting state, although the search path starting from some states is much longer than the shortest path generated by the optimal policy.

When a state space is large and function approximation is applied for learning the value function, achieving an optimal policy can become difficult. Again look at this problem. Suppose the states in Figure 2.7 are only a subset of a large state space, and suppose two features L and V are used for representing states and a linear function is used for representing the value function. Table 2.5 presents a value function, $\hat{v}$, that learning could eventually come up with. With this predicted value function, we can obtain a static greedy policy as presented in Table 2.3; we can also obtain a dynamic policy as presented in Table 2.4.

Chapter 3
Domain: The Task, Scheduling, and Learning

3.1 Space Shuttle Payload Processing

The NASA space shuttle payload processing (SSPP) domain requires scheduling the various tasks that must be performed to install and test the payloads that are placed in the cargo bay of the space shuttle. The types of payload carriers include long module (LM), mission peculiar equipment support structure (MPESS), and pallet and igloo. In job-shop scheduling terminology, each shuttle mission is a job. Each job consists of a partially-ordered set of tasks that must be performed. Each task has a duration and a list of resource requirements. For example, the task `MISSION-SEQUENCE-TEST` has a duration of 7200 and requires two quality-control officers, two technicians, one `ATE`, one `SPCDS`, and one `HITS`. These tasks form a hierarchy in which a task may be decomposed into a set of subtasks. The primitive, or "leaf", tasks must each have their duration and resource requirements specified. Non-primitive, parent tasks can have resource requirements in addition to those of their children, but they do not have a duration. Instead, the duration of a parent task is determined by the start time of its earliest subtask and the end time of its latest subtask. The resources required by the parent task must be available for the entire duration of the task, not just during those time intervals when one of its subtasks is being performed.

For example, consider a case where a payload must be disassembled, modified, and then reassembled. Suppose that the modifications involve several tasks (some of which can be performed in parallel), but that throughout the entire process a certain testbed resource is being used. This can be represented by a parent task that has the testbed as one of its required resources. The disassembly, modification, and reassembly tasks can all be (partially-ordered) children of the parent task.

It is possible to require that one parent task must be scheduled prior to another parent task. This is interpreted to mean that all of the children of the first parent task must be scheduled before all of the children of the second parent task.

There are 35 different types of resources. Each may have many units available. For example, there are 6 quality control officers available and 15 technicians. However, these available resources may be split into resource pools, so that, for example, the 6 quality control officers might be subdivided into two pools of size 4 and 2. If a task requires two quality control officers, they must both be drawn from the same pool. Resource pools model multiple work shifts and multiple physical locations. A complete schedule must specify the start time and the end time of each task and the resource pool by which each resource requirement of each task is satisfied. A set of resource types and simulated resource pools are shown in Table 3.1.

Each shuttle mission has a fixed launch date, but no starting date or ending date. The `FLY-MISSION` task must be scheduled on the launch date. Most of mission tasks must be performed prior to launch, but some also take place after the shuttle has landed. Hence, tasks prior to launch have deadlines but no ready times; tasks after landing have ready times but no deadlines. Table 3.2 shows a set of high-level tasks in a Pallet-Igloo payload mission. Table 3.3 shows a set of Pallet-Igloo LEVEL-III-II Tasks. A mission for a single Pallet-Igloo payload has been specified with 82 tasks.

A typical SSPP problem involves the simultaneous scheduling of between two and six shuttle missions; each mission involves up to two payloads and between 32 and 164 tasks. Hence, the SSPP domain requires solving scheduling problems containing several hundred tasks. A key goal of the scheduling system is to find a conflict-free schedule that minimizes the total duration of the schedule, so that more missions could be scheduled in a year. This is much more challenging than simply finding a *feasible* schedule.

The work presented in this dissertation considers problems involving temporal constraints and resource constraints as the hard constraints and minimizing the duration of a schedule as a soft constraint.

Table 3.1. Resource pools. There are 35 different types of resources. This table lists a subset of resource types and simulated resource pools. For example, the 6 quality control officers might be subdivided into two pools of size 4 and 2.

Resource Pool	Resource Type	Capacity
ATE-1	ATE (R1)	2
Crane-1	Crane (R3)	2
Electrical-engineer-1	Electrical-engineer (R4)	4
Electrical-engineer-2	Electrical-engineer (R4)	3
Flight-Payload-1	Flight-Payload (R5)	2
Floor-1	Floor (R6)	1
HITS-1	HITS (R7)	2
Igloo-Carrier-1	Igloo-Carrier (R8)	4
Long-module-1	Long-module (R9)	4
Logistics-1	Logistics (R10)	3
Logistics-2	Logistics (R10)	2
Mechanical-engineer-1	Mechanical-engineer (R12)	4
Mechanical-engineer-2	Mechanical-engineer (R12)	3
MPESS-Carrier-1	MPESS-Carrier (R13)	4
Pallet-carrier-1	Pallet-carrier (R15)	4
PCU-1	PCU (R16)	2
PITS-1	PITS (R17)	2
PSTE-1	PSTE (R18)	2
Quality-control-1	Quality-control (R19)	4
Quality-control-2	Quality-control (R19)	2
Rack-3-1	Rack-3 (R20)	1
Rack-4-1	Rack-4 (R21)	1
Safety-1	Safety (R30)	2
SPCDS-1	SPCDS (R31)	2
Strong-Back-1	Strong-Back (R32)	2
Technician-1	Technician (R33)	10
Technician-2	Technician (R33)	8
Test-conductor-1	Test-conductor (R34)	2
Test-Stand-1	Test-Stand (R35)	2

Table 3.2. Pallet-Igloo high-level tasks. This table includes the high-level tasks of the Pallet-Igloo mission. A parent task is not specified with a task duration. The duration of a parent task is variable during scheduling, which covers the scheduled time of its sub-tasks. The `FLY-MISSION` task must be scheduled on the launch date. The top level Pallet-Igloo processing task has 6 subtasks: `T1`, `T31`, `T38`, `T62`, `FLY-MISSION`, and `T72`. All sub-tasks of `T1`, `T31`, `T38`, and `T62` must be scheduled before launch. All sub-tasks of `T72` must be scheduled after launch.

Task	Duration (Sec)	After	Resource List		Parent
			Type	Qty	
T0 (PALLET-IGLOO-PROCESSING)			IGLOO-CARRIER	1	Top-Level
			PALLET-CARRIER	1	
T1 (LEVEL-IV)					T0
T2 (PRE-EXPERIMENT-INTEGRATION)					T1
T6 (IGLOO-CARRIER-STAGING)					T1
T11 (PALLET-IGLOO-INTEGRATION)		T6			T1
T20 (PREP-CONNECT-GSE)		T11			T1
T25 (LEVEL-IV-TEST)		T22 T23 T24			T20
T31 (LEVEL-III-II)		T1			T0
T34 (LEVEL-III-II-TESTING)		T32			T31
T38 (CITE-OPS)		T31			T0
T39 (CITE-PREPS)					T38
T44 (INSTALL-CONNECT-SLUK-CABLING)		T43			T38
T52 (REMOVE-TRANSFER-D-SSPS-TO-OPF)		T51			T38
T62 (LEVEL-I)		T38			T0
FLY (FLY-MISSION)		T62			T0
T72 (DEINTEGRATION)		FLY			T0

Table 3.3. Pallet-Igloo LEVEL-III-II tasks. A task specified with a work duration is a leaf task. A task may have a list of resource requirements. Only the temporal constraints with respect to tasks at the same level need to be specified. The temporal relationship with the parent tasks is inherited. For example, the task `MISSION-SEQUENCE-TEST` has a duration of 7200 and requires two quality-control officers, two technicians, one `ATE`, one `SPCDS`, and one `HITS`. This task must be scheduled after the `SL-SYS-TEST` task as specified. Also, it must be scheduled after all sub-tasks of `PREP-FOR-TEST` (T33) and `LEVEL-IV` (T1).

Task	Duration (Sec)	After	Resource-list		Parent
			Type	Qty	
T31 (LEVEL-III-II)		T1			T0
T32 (INSTALL-PAYLOAD-IN-TS)	21600				T31
T33 (PREP-FOR-TEST)	4320	T32			T31
T34 (LEVEL-III-II-TESTING)		T33			T31
T35 (SL-SYS-TEST)	14400		QUALITY-CONTROL	2	T34
			TECHNICIAN	2	
			ATE	1	
			SPCDS	1	
			HITS	1	
T36 (MISSION-SEQUENCE-TEST)	7200	T35	QUALITY-CONTROL	2	T34
			TECHNICIAN	2	
			ATE	1	
			SPCDS	1	
			HITS	1	
T37 (CARGO-CLOSEOUT)	8640	T34			T31

Other types of space shuttle scheduling problems were also discussed in [Zweben *et al.*, 1994]. Other practical optimization criteria considered in this domain include minimizing weekend work in a schedule and minimizing schedule perturbation for rescheduling. A more sophisticated specification of the domain would include *state constraints*. Each task may have state prerequisites (e.g., the shuttle power must be turned off) and state effects (e.g., turning on or off the power). The scheduler must then perform a limited amount of planning to find feasible schedules.

3.2 A Repair-Based Scheduling Method

3.2.1 A Repair-Based Scheduling Problem Space

The repair-based scheduling state space defines complete schedules as its states. Other than a partial schedule, every task for a complete schedule must be scheduled at a time and assigned to resources. Since rescheduling is more common in NASA, this state space is more useful than a partial schedule state space.

Zweben et al. (1994) defined a repair-based scheduling problem space as follows. All schedules in the state space must satisfy temporal constraints, but may have resource violations. First, a *critical path schedule* is constructed by working backward and forward from the launch and landing dates. Each task prior to launch is scheduled as late as the temporal partial order will permit; each task after landing is scheduled as early as the temporal partial order will permit. Resource constraints are ignored; resource requests are randomly assigned to resource pools that have enough resources. This critical path schedule is the shortest schedule that satisfies the temporal constraints. This critical path schedule can be constructed very efficiently, and it provides the starting state for the scheduling problem space.

In each state of this problem space, there are two kinds of operators that can be applied. The REASSIGN-POOL operator changes the pool assignment for one of the resource requirements of a task. It is only applied when the pool reassignment would allow the resource requirement to be successfully satisfied. The MOVE operator moves a task to a different time and then reschedules all of the temporal dependents

of the task using the critical path method (leaving the resource pool assignments of the dependents unchanged). The MOVE operator is only applied to move a task to the next earlier or the next later time at which the violated resource requirement can be satisfied. When moving earlier, the temporal dependents of the moved task are all the tasks that must scheduled before the task. When moving later, the temporal dependents are all the tasks that must scheduled after the moved task. When calculating the next earlier or the next later resource-available time, all the temporal dependents must first release their assigned resources for the concerned resource type.

A schedule can involve resource violations at many different times. An easy way to identify the violations is to subdivide the schedule into a sequence of time intervals. A time interval is defined to be a maximal period of time during which the set of currently scheduled tasks does not change. The start time and the end time of a time interval must meet the start time or the finish time of some task. Therefore, within a time interval the resource assignments do not change. A time interval is also called a "block" or a "window".

In repair-based scheduling, time intervals are used to determine the number of available operations for repairing the schedule. In an interval involving a violation with a resource pool, we find all the tasks that compete for the resources in the pool. To reduce the branching factor of the search, we consider only operations that would repair the earliest resource constraint violation in the schedule. For the earliest interval containing a constraint violation, we consider each resource pool that is over-allocated and consider pool reassignments and moves for all tasks involved in the over-allocation. We have experimented with two other ways of reducing the branching factor. One alternative is to consider only operations that would repair the latest resource violation in the schedule. An even more restricted approach is to still focus on the earliest interval that contains a violation, but to choose only one of the over-allocated resource pools and consider only pool reassignments or moves that would repair that over-allocation.

Any feasible solution of a scheduling problem is a final state in the problem space. The quality of a solution is measured in terms of the optimization criterion.

Table 3.4. The repair-based scheduling problem space.

`State`	Complete schedule (may contain resource violations)
`Start State`	Critical path schedule
`Operators`	REASSIGN-POOL and MOVE
`Final State`	Conflict-free schedule (with some quality measure)

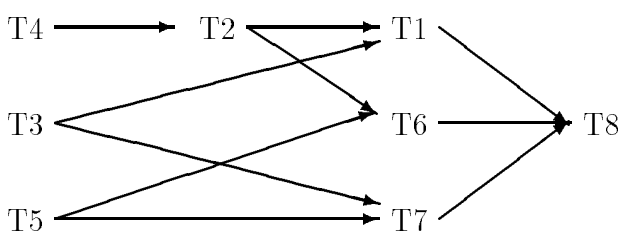
This problem space contains all possible schedules that can be made by performing a series of REASSIGN-POOL and MOVE operations starting from the critical path schedule. Table 3.4 summarizes the repair-based scheduling problem space.

Example 3.1 *Figure 3.1 shows an example of a simple scheduling problem. It is an 8-task job with two types of resources. Each type of resource has two pools with capacity 5 and 3 respectively. Figure 3.2 shows a critical path schedule for the problem. It depicts the Gantt diagram and resource utilization diagram. It shows that for the pool r22 violation in interval 7, where both task T1 and T6 compete for the resource, we can reassign resource R2 to resource pool r21 for either task T1 or T6 to eliminate the violation. To resolve the pool r11 violation in interval 7, we must move either task T1 or T7.*

The drawback of the problem space generated by these operators is that it is not guaranteed to contain the optimal solution. This problem space is much smaller than the total state space that contains all possible schedules. There are other formulations of the state space that can guarantee to contain the optimal solutions. Unfortunately, these spaces all appear to be much larger, which make any search procedure inefficient. A preliminary study compared Zweben's operators to a set of "guaranteed" operators and found that the former lead to a much more efficient search. The iterative repair algorithm (described below) performs only about half as many repairs with Zweben's operators as it does with the "guaranteed" operators to find the solutions of the same quality.

Task	Duration	After	Resource Type R1	Resource Type R2
T1	3	T2, T3	5	3
T2	10	T4		5
T3	4			5
T4	5			4
T5	12		4	
T6	7	T2, T5	1	2
T7	14	T3, T5	5	
T8	5	T1, T6, T7		5

(a) Tasks



(b) Precedence constraint diagram

Resource pool	Type	Capacity
r11	R1	5
r12	R1	3
r21	R2	5
r22	R2	3

(c) Resource pools

Figure 3.1. A simple scheduling problem. (a) Eight tasks in a job. Each task is specified with a duration, precedence relation, and resource requirements. For example, task T1 requires 3 units of time. It must be performed after T2 and T3. To perform it, T1 requires 5 units of resource R1 and 3 units of resource R2. (b) The precedence constraint diagram. (c) Two types of resources R1 and R2 and their available pools. There are two resource pools available for each type of resources with capacity 5 and 3 respectively.

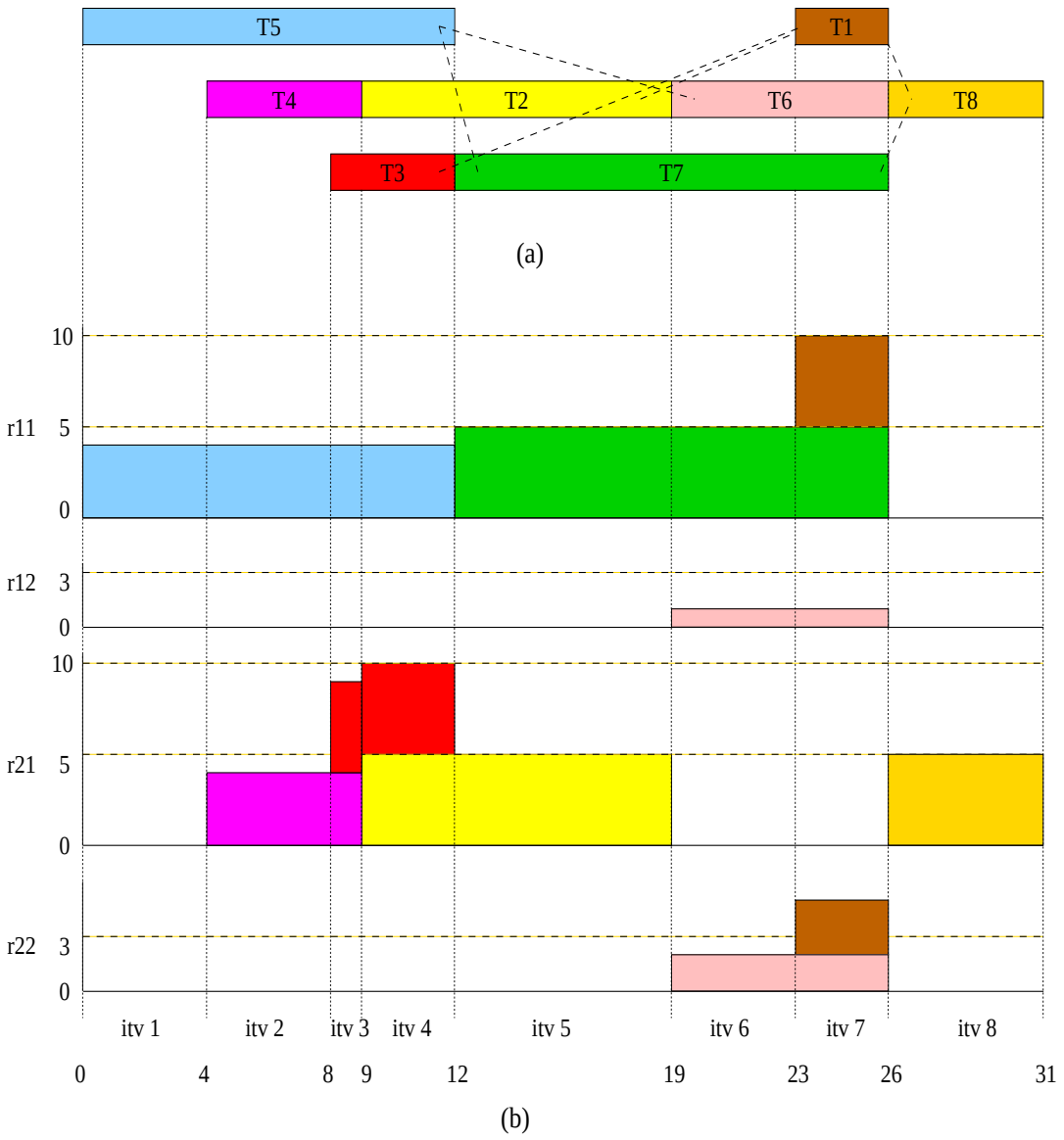


Figure 3.2. A critical path schedule for the 8-task job from Figure 3.1. (a) The Gantt Diagram of the schedule. (b) The resource pool utilization of each pool. There are a total of 8 schedule intervals. Three of them involve resource violations. For example, interval 7 (`itv 7`) contains violations with two pools, `r11` and `r22`. For the violation in pool `r22` where tasks T1 and T6 are competing for the resource, we can perform REASSIGN-POOL of resource R2 to either task T1 or task T6 to eliminate the violation. To resolve the violation in pool `r11`, we must move either task T1 or task T7.

3.2.2 The Iterative Repair Algorithm

Zweben and colleagues developed the following iterative repair method for solving the scheduling problem. First, a critical path schedule is constructed. REASSIGN-POOL and MOVE operators are applied to a schedule as follows. At each step, the earliest resource constraint violation is identified. If a REASSIGN-POOL operator can be applied to reduce this over-allocation, then it is applied. If not, then the MOVE operator is applied to move one of the offending tasks to an earlier or later time. If several different pool reassignments are possible, one is chosen at random. If both an earlier and a later move are possible, then one is chosen at random. Of the several tasks involved in the resource violation, one is chosen 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.

The iterative repair method originally presented by Zweben views all constraints as soft constraints. A *cost function* is defined to measure the quality of a schedule:

$$cost(s) = \sum_{c_i \in constraints} penalty_{c_i}(s) * weight_{c_i}(s), \tag{3.30}$$

which is a weighted sum of constraint violations. The *penalty function* ($penalty_{c_i}$) of a constraint returns a non-negative number reflecting the degree to which the constraint is violated. The *weight function* ($weight_{c_i}$) of a constraint returns a non-negative number representing the importance or utility of a constraint.

The overall control structure of the algorithm applies simulated annealing to minimize constraint violations. Iterative repair begins with a complete schedule of unacceptable quality and iteratively modifies it until its quality is found to be satisfactory or until a designated time bound is reached. After each operator is applied, the cost of the resulting schedule is computed. If this has decreased, the resulting schedule is accepted as the current schedule. If it has increased, the resulting schedule is accepted only with probability $exp(-\Delta cost/T)$, where $\Delta cost$ is the increase in the cost function and T is the current temperature. The temperature is gradually

decreased. An annealing schedule provides a starting temperature and a cooling rate for the temperature.

This procedure can be extended to solving problems with hard constraints. For such problems, search proceeds until no hard constraints are violated. To obtain a better schedule, the algorithm can be run several times, and the best resulting schedule is selected. A restart criterion is defined to determine when the procedure needs to be restarted. A simple restart criterion is to restart whenever a schedule that satisfies all hard constraints has been reached. When restarted, the algorithm works again on the starting schedule at the same starting temperature. Figure 3.3 shows the extended iterative repair algorithm.

3.3 Defining Reinforcement Learning Problem

3.3.1 Resource Utilization and RDF

To view the scheduling problem as a reinforcement learning problem, we must describe the reinforcement function R. A key goal of the scheduling task is to find short conflict-free schedules. The reinforcement function must reflect this goal. Because we are using a learning approach, the reinforcement function should also give the same amount of reward to schedulers of the same relative quality. A simple approach is to define the schedule length of the final schedule as the reinforcement of reaching a final schedule. Unfortunately, this only works for a single scheduling problem. The lower bound of the schedule length of different problems can vary dramatically. For example, a problem with more jobs would require a much longer schedule than one with fewer jobs. A problem with many big jobs would require a much longer schedule than one with few of these jobs.

To remove this variation, we need some kind of normalized measure of schedule length. A good way for doing this is to consider the critical path schedule of a problem. If the critical path schedule is long (because there are many jobs or many big jobs), then a final schedule will also be long (at least as long as the critical path schedule). If the critical path schedule involves a large amount of resource over-allocation (because

Algorithm 6 (IR) *Given: the annealing schedule: T_0, the starting temperature, and r, the temperature cooling rate; P, a scheduling problem; $cost(s)$, the cost function; and a restart criterion.*

1. *Construct the critical path schedule and make it the current schedule $s := cps(P)$; set temperature $T := T_0$.*
2. *Repeat with a given number of restarts or until an acceptable solution is obtained or until a designated time bound is reached.*
   (a) *If pool reassignment is possible, then perform it to create new schedule s'.*
   (b) *Else choose a move probabilistically by heuristics and perform it to create new schedule s'.*
   (c) *If $cost(s') < cost(s)$, then set $s := s'$; $T := T * r$.*
   (if resulting schedule is improved, accept it and decrease the temperature)
   (d) *Else if $rand() > e^{\frac{-(cost(s')-cost(s))}{T}}$, then set $s := s'$; $T := T * r$.*
   (else accept it probabilistically according to $e^{\frac{-(cost(s')-cost(s))}{T}}$)
   (e) *If the repaired schedule satisfies the restart criterion, then restart: set $s := cps(P)$; $T := T_0$.*

Figure 3.3. The iterative repair algorithm.

the jobs have simultaneous or close deadlines), then a final schedule would be largely *dilated* from the critical path schedule in terms of the schedule length, leading to a long schedule. A difficult problem would require a long schedule and much schedule-length dilation, whereas a simple problem would require a short schedule and less dilation.

We now define a number of resource utilization indices related to both schedule length and resource utilization. Based on these, the *resource dilation factor* (RDF) is defined to adjust the schedule length variation across multiple scheduling problems.

Definition 3.1 *Let $capacity(i)$ be the (fixed) capacity of resource type i—that is, the combined capacity of all resource pools of resource type i. At each time t in the schedule, let $u(i,t)$ be the current utilization of resources of type i. If $u(i,t) > capacity(i)$, then the resource of type i is overallocated at time t (no matter how we assign resource requests to resource pools of this type). The resource utilization index $RUI(i,t)$ for resource type i at time t is*

$$RUI(i,t) = \max\left\{1, \frac{u(i,t)}{capacity(i)}\right\}. \tag{3.31}$$

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

Definition 3.2 *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 n resource types and all l times:*

$$TRUI(s) = \sum_{i=1}^{n}\sum_{t=1}^{l} RUI(i,t). \tag{3.32}$$

The total resource utilization index does not consider the resource pool assignments. It is determined 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. A more accurate measure of resource utilization of a schedule is concerned with the resource pool assignment.

Definition 3.3 *Let capacity(i,j) be the (fixed) capacity of resource pool j of resource type i. At each time t in the schedule, let $u(i,j,t)$ be the current utilization of resources of pool j. If $u(i,j,t) > capacity(i,j)$, then the resource pool j is overallocated at time t. The pool utilization index $PUI(i,j,t)$ for resource pool j of resource type i at time t is*

$$PUI(i,j,t) = \max\left\{1, \frac{u(i,j,t)}{capacity(i,j)}\right\}. \tag{3.33}$$

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

Definition 3.4 *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 n resource types and all l times where for each type of resource the average pool utilization index over all pools is computed:*

$$TPUI(s) = \sum_{i=1}^{n}\left[\frac{1}{m_i}\sum_{j=1}^{m_i}\sum_{t=1}^{l} PUI(i,j,t)\right]. \tag{3.34}$$

Example 3.2 *For the critical path schedule shown in Figure 3.2, the total resource utilization index is*

$$\begin{aligned} TRUI(s) &= [23 + \frac{10+1}{8}(26-23) + (31-26)] + \\ &\quad [8 + \frac{9}{8}(9-8) + \frac{10}{8}(12-9) + (31-12)] \\ &= 64. \end{aligned} \tag{3.35}$$

The total pool utilization index is

$$\begin{aligned} TPUI(s) &= \frac{1}{2}\left\{[23 + \frac{10}{5}(26-23) + (31-26)] + 31\right\} + \\ &\quad \frac{1}{2}\left\{[8 + \frac{9}{5}(9-8) + \frac{10}{5}(12-9) + (31-12)] + \right. \\ &\quad \left. [23 + \frac{5}{3}(26-23) + (31-26)]\right\} \\ &= 66.9. \end{aligned} \tag{3.36}$$

Definition 3.5 *The resource dilation factor (RDF) for a schedule s is the ratio of total pool utilization index of s to total resource utilization index of its starting critical*

path scheduling s_0:

$$RDF(s) = \frac{TPUI(s)}{TRUI(s_0)}. \tag{3.37}$$

To understand the rationale behind this formula, first note that in a final schedule s, $TRUI(s)$ is equal to $TPUI(s)$. They are just the number of resource types times the length of the schedule ($n \times l$). This is because in the final schedule, no resource is overallocated, so $RUI(i,t) = PUI(i,j,t) = 1$. The total resource utilization index of the initial schedule, $TRUI(s_0)$, measures the amount of overallocation of resources in the initial state, and hence, provides a crude measure of the difficulty of the scheduling problem. $TRUI(s_0)$ provides a constant value for all critical path schedules no matter how resource pools are assigned. Hence, it is used to normalize the final schedule length to produce the resource dilation factor.

The RDF of a final schedule normally is in the range from 0.8 to 2.0. The value can sometimes be smaller than 1. This is because it is possible for some problem to construct a solution which involves relative less schedule-length dilation with respect to the resolved resource overallocation, therefore, the final $TPUI$ is smaller than the initial $TRUI$. For each repair operation, pool reassignment decreases the $TPUI$ of the schedule. The task-move operation could also decrease the $TPUI$, if the resulting schedule is not lengthened quite dramatically.

3.3.2 A Deterministic Decision Process for Scheduling

We describe the reinforcement learning problem for the scheduling task as a deterministic decision process (DDP). To do this, we must define the problem space, the reinforcement function R, and the optimization criterion (value function). We employ the same problem space as Zweben et al. As described earlier, each state is a complete schedule in which temporal constraints are satisfied but may contain resource violations. The starting state s_0 is the critical path schedule. Repair operations are pool-reassignment and task-move. Final states are conflict-free schedules. Evidently, this process is deterministic. Each repair operation produces a unique resulting schedule.

This problem space contains cycles. A simple situation that causes a cycle would be first moving a task earlier for resolving a violation in one resource type and then moving the task back to the old position because of the violation of another resource type. The cycle length may vary. Based on an investigation over hundreds of scheduling problems, we found that the length of a cycle is generally less than 7. Among the encountered cycles, about 40% are 2-step cycles (moving earlier/later and then immediately moving back). We further found that cycles are quite sparsely distributed in the state space. The chance for the iterative repair algorithm to encounter a cycle in a trial is less than 4%. In order to make a learned policy for the DDP escape from an infinite loop and produce a final schedule in a finite number of repairs, we employ the dynamic policy as discussed in Chapter 2.

Now the task is to define the reinforcement function $R(s, a, s')$. We define the reinforcement function $R(s, a, s')$ to give a small negative reinforcement $-c = -0.001$ constantly for any action that does not create a conflict-free schedule (i.e., where s' still contains constraint violations). This assesses a small penalty for each scheduling action (REASSIGN-POOL or MOVE). It is intended to encourage reinforcement learning to prefer actions that *quickly* find a good schedule. It is also intended to help learning to construct a policy that can escape loops. For any action that produces a conflict-free schedule s', the reinforcement $R(s, a, s')$ is the negative of the resource dilation factor $-RDF(s')$. The RDF provides a scale-independent measure of the schedule length across multiple scheduling problems. The $-RDF(s')$ is intended to encourage the search to find short final schedules. It provides more penalty for reaching a longer schedule. Because we apply a loop avoidance mechanism and every trial must end up in a final schedule, we do not need to worry about using a positive final reward to encourage and guarantee search to pick up actions for achieving final schedules. Note that if we do not apply loop control and we employ a limited search horizon, the optimal policy could suggest to take an infinite loop because its cumulative penalty can be smaller than the policy that suggests to make a final schedule (penalty RDF of a final schedule is much bigger than 0.001). To summarize, the reinforcement function is

$$R(s,a,s') = \begin{cases} -RDF(s'), & s' \text{ is final} \\ -0.001 \quad , & \text{otherwise} \end{cases} \tag{3.38}$$

The use of a constant cost for non-final actions is based not only on the desire to find the shortest path from the starting schedule to a final schedule but also on the desire to minimize the total CPU time to find the schedule. In general, a MOVE operation is more expensive than a REASSIGN-POOL operation. However, the difference between execution times of the two operations is relatively small compared to the time required for a repair. Most of the CPU time in a repair is spent testing and generating all possible repair actions and evaluating these actions. When the evaluation is based on schedules, we must perform one-step lookahead to generate all possible resulting schedules.

The action cost c must be small to make the system more concerned with the final RDF. For example, facing the situation where one action creates a schedule with final RDF v (in 1 step) and the other action followed by two actions (in total 3 steps) creates a schedule with the optimal final RDF $v - 2c$, the system can not distinguish the difference and hence can not select the second choice for achieving the optimal schedule. It views both actions to be optimal because they provide the same total long-term reinforcement. A small c will help the system to find the optimal schedule.

The cost c is determined by two factors: the permitted relative error of the RDF ε_{RDF} for choosing a better final schedule and the difference of the number of steps to take $\Delta Steps$ such that within it a better schedule under ε_{RDF} is selected. It can be written as

$$c = \frac{\varepsilon_{RDF}}{\Delta Steps} \tag{3.39}$$

Certainly, the smaller c is, the more accurate the reinforcement learning model will be, which leads to the optimal policy to find the optimal schedule. However, when performing function approximation, a very small c could eliminate the impact of the cost. In this work, c is set to 0.001. This setting has been intended to work for the situation where $\varepsilon_{RDF} = 0.01$ and $\Delta Steps = 10$. The quality of schedules has

been very close when their RDF difference is under 0.01. In this cost setting, a better schedule with error $\varepsilon_{RDF} < 0.01$ can be detected within 10 future steps.

Finally, consider the definition of the value function. Because the dynamic policy is employed to escape loops, search is guaranteed to get to a final state in a finite number of steps. Thus, we can simply employ an infinite time horizon to take care of the variable length of scheduling paths. We employ the undiscounted cumulative future reward as the value function, because the final reward RDF is most important and should not be discounted. Hence, the value function actually describes a linear combination of the final RDF and the number of repairs needed to construct a final schedule. For a state s, the value under a given policy π is

$$f^{\pi}(s) = \sum_{j=0}^{\infty} R(s_j, a, s_{j+1}) | s_0 = s \tag{3.40}$$

$$= \sum_{j=0}^{N-1} R(s_j, a, s_{j+1}) \tag{3.41}$$

$$= -RDF(s_N) - 0.001(N-1), \tag{3.42}$$

where s_N is a conflict-free schedule and N is the number of steps taken to find s_N.

3.4 Challenges for the Learning Task

We have specified how to view repair-based scheduling as a reinforcement learning problem. Before presenting the learning approach, we now look at several aspects of this task that provide challenging problems for learning methods. First, the state space is huge. Multiple scheduling problems are present in the space, and even a very small scheduling problem (with only 20 tasks) can generate over 10^{15} different schedules as states. The number of states generated by a 50 task problem normally ranges from 10^{20} to 10^{50}. (These are roughly estimated based on the branching factor and the number of repairs required to construct a final schedule.) Therefore, using any table-based "exact" method to compute the value function or policy certainly is out of the question. We must adopt some kind of function approximation methods for this computation.

The next challenge is that we are not learning a policy for solving a single scheduling problem, rather, we are learning a single generalized policy for solving a set of scheduling problems. In a particular application setting, it is believed that there are domain-specific regularities that can be exploited to construct an effective policy. Therefore, a goal of this task is to explore the possibility of applying reinforcement learning algorithms to discover domain-specific regularities in the NASA space shuttle payload processing task to come up with a good scheduling policy. In order to make such a generalized policy, again, function approximation methods must be employed.

Another challenge is the input representation problem. As long as function approximation is applied, we are only allowed to use a fixed-length vector to describe a schedule, because virtually all methods for function approximation can only be applied to a fixed-length vector as input. However, the length of schedules varies depending on the number of tasks and the complexity of their temporal and resource constraints. In previous applications of reinforcement learning, such as Tesauro's [Tesauro, 1992] work on backgammon, the representation was *exact*: every possible state had a unique representation. However, in scheduling this is not possible: we must employ some form of *abstract input representation*. This abstract representation must be designed to make learning easy (or at least to make it possible). In the following two chapters, two approaches will be presented to solve the input representation problem. Chapter 4 presents a feature hand-engineering approach to come up with a set of global schedule features. Chapter 5 presents a time-delay neural network approach to discover features automatically.

3.5 Learning Methods

This section presents the learning methods applied to the scheduling reinforcement learning task.

3.5.1 Applying Value Function Approximation

There have been generally three types of functions that could be applied for learning to achieve a generalized policy: the value function, the Q-function, and the policy function. Since we have a complete deterministic scheduling model for exact state transitions, we can approximate the value function with a value iteration method. Once we have learned the value function, we can transform it into a policy via a simple one-step lookahead search.

The next two types of functions can avoid one-step lookahead in determining a policy. One is the policy function, which directly maps a state to an action as the policy. The other type of function is the Q-function, which maps from a state-action pair to a Q value. Unfortunately, these two types of functions are difficult to apply in the scheduling domain. The key difficulties come from the following problems with the action representation. First, although there are only two types of actions available for repairing a schedule, which are pool-reassignment and task-move, the number of admissible actions can vary dramatically for different schedules. Second, each operation presents its unique repair to a schedule. It is hard to draw similarities between actions. Therefore, actions are difficult to generalize. Finally, the description of an action does not provide much information. Much more useful information about an action is presented in the resulting schedule. All these factors can make approximation on the policy function or the Q-function perform very poorly. Therefore, the work presented in the thesis explores the possibility of value function approximation.

3.5.2 Applying the TD(λ) algorithm

As presented earlier, the reinforcement learning problem for the scheduling task involves a deterministic state space. It employs a critical path schedule as the starting state. The goal of the learning problem is to construct a policy to find short conflict-free schedules quickly. Evidently, we do not need to construct a policy that covers

the whole state space. Rather, our interest is to construct a policy to lead the search to the more valuable parts of state space.

The temporal difference learning algorithm TD(λ) is the best known algorithm for this type of problem. In Tesauro's landmark work on backgammon, he showed that TD(λ) can learn an excellent value function for playing backgammon. Although backgammon can be viewed a stochastic problem, because it involves the roll of the dice and opponent's decisions, Tesauro treated the problem in a deterministic way. After rolling a dice, the effect of each move is unique, leading to a single board position. Therefore, the action to choose can be determined based on evaluating all possible moves, which is the same as evaluating of all possible resulting board positions. Like the scheduling task, backgammon also starts in a single board position. The success of TD-gammon suggests that TD(λ) could also successfully learn a good value function for scheduling.

We employ TD(λ) combined with a feed-forward neural network for the value function. Let us write the value function as $\hat{f}(s, \mathbf{W})$, where $\mathbf{W}$ is the vector of weights in the network. In this situation, TD(λ) works as follows. Let $s_0, s_1, \ldots, s_N$ be a sequence of states visited by following policy π with associated reinforcements $R(s_1), \ldots, R(s_N)$. At step $j+1$, we can compute the temporal difference error at step j as

$$J_j = [\hat{f}(s_{j+1}, \mathbf{W}) + R(s_{j+1})] - \hat{f}(s_j, \mathbf{W}). \tag{3.43}$$

TD(λ) then computes the smoothed gradient

$$e_j = \nabla_{\mathbf{W}} \hat{f}(s_j, \mathbf{W}) + \lambda e_{j-1} \tag{3.44}$$

and updates the weights of the network according to

$$\Delta \mathbf{W} = \alpha J_j e_j. \tag{3.45}$$

The recency factor λ can be viewed as a smoothing parameter that combines previous gradients with the current gradient in e_j.

The TD(λ) algorithm was designed to learn the value function for a stationary Markov random process such as would result from following a fixed policy. In reinforcement learning, however, we want to apply it to learn the value function of the

optimal policy starting with an initial, random policy. To do this, we employ real-time dynamic programming. TD(λ) is applied on-line to the sequences of states and reinforcements that result from choosing actions according to the current estimated value function, $\hat{f}$. At each state s during learning, we conduct a one-step lookahead search using the current estimated value function $\hat{f}$ to evaluate the states resulting from applying each possible operator. We then select the action that maximizes the predicted value of the resulting state s'. After applying this action and receiving the reward, we update our estimate of $\hat{f}$ to reflect the difference between the value of $\hat{f}(s)$ and the more informed value $R(s') + \hat{f}(s')$. This means that the policy is continually changing during the learning process. The actual learning procedure is more complicated. It includes several modifications described in the following sections.

3.5.3 Exploration

For any reinforcement learning algorithm, it is critical to perform some kind of exploration to discover new and better ways of getting from the start state to the goal. The following simple exploration strategy is employed. At each state, with probability β we choose a random action instead of the action recommended by the current value function and policy. Initially, β is set to 1. After each training sequence, β is decreased by an amount $\Delta\beta$ until it reaches a final value of 0.05. (The values used for $\Delta\beta$ are given below.)

3.5.4 Trajectory Backward Learning

We do not perform weight updates in the neural network after each action. Instead, we remember the sequence of states visited along the path from the starting state to the final conflict-free schedule. Then we update the network starting with the final action and working backward to the start of the action sequence. Trajectory backward learning is a fundamental modification to the TD(λ) algorithm. It differs from standard forward TD(λ) in that the eligibility trace records the future states

rather than the history. In the next section, we will present results comparing backward TD(λ) to standard TD(λ). These show that backward learning learns more accurately and more efficiently than forward learning.

3.5.5 Overtraining Big Errors with Multiple Updates

To better reduce the temporal difference error between the values of a pair of adjacent states, we perform multiple weight updates for a state-pair with a big error. The weight-update loop proceeds until the error is reduced to a given threshold ϵ or until a maximum permitted number of loops l_b is reached. We set $\epsilon = 0.05$ and $l_b = 20$. This works better than simply using a single update with a big learning rate, because the weights in the network are changed more smoothly.

Furthermore, we develop a mechanism to set the learning rate automatically and use it during learning. The learning rate α is set according to the initial temporal difference d_0 of the current state-pair. We set $\alpha = d_0/3.0$. In this application, we have $d_0 < 0.5$ and average d_0 for final states is about 0.09 (the final prediction error is much bigger than the error for non-final predictions). By this α setting, the number of loops for weight update of a state-pair on average is controlled in the range between 10 and 20. In general, a large λ parameter requires more iterations. When $\lambda > 0.95$, almost all updates reach the loop bound l_b, because an update does not follow the current gradient (the gradients from the trace dominate the update). Under this situation, the TD error can not be effectively reduced, and TD learning is not really performed.

3.5.6 Best Experience Replay

We employ Lin's *experience replay* method. During learning, the best sequence of moves from start to goal is remembered, which is the shortest path to the so far best-known final schedule. After every four training sequences, we update the net-

work using this best training sequence. This improved learning and performance significantly.

3.5.7 Teaching

To accelerate the learning process, in the beginning of the training process, we train the network using the sequences generated by a good scheduler. This scheduler works like a teacher to make training focus on some more useful part of the state space. The iterative repair procedure can be a good teacher. Another good scheduler is greedy search using the RDF of the current schedule as the heuristic. In experiments, teaching is performed to generate 20 sequences for each problem. These sequences may be re-used later during training as in the experience replay procedure.

3.5.8 Policy Determination Methods

We employ the following two methods to determine how to choose an action. These methods are employed during both training and execution. First, in order to avoid loops in search, we employ the dynamic greedy policy with respect to the current value function. The policy is in the form of the following if-then structure as presented in Section 2.6.3:

```
Choose the best action according to the value function if
     it does not create a loop.
Else choose the second best action according to the value
     function if it does not create a loop.
Else backtrack.
```

Second, when the branching factor is big, we do not employ a full one-step lookahead search to select actions. Instead, we employ *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 actions 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 action is within ϵ of the best possible action. We continue sampling possible actions 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 $A = \{a_1, a_2, \ldots, a_k\}$ is the current set of k sampled actions and $V = \{v_1, v_2 \ldots, v_k\}$ gives the values of these actions. Let v_{max} be the largest value in V. We postulate that the current observed action sample has the same distribution as the complete action sample. Thus, we estimate the value of the best possible action within permitted error ϵ using the current sample, which is

$$\hat{v}_\epsilon = v_{max}(1-\epsilon). \tag{3.46}$$

We then compute the percentage of actions which have values larger or equal to $\hat{v}_\epsilon$:

$$\theta = \frac{|V_\epsilon|}{|V|}, \tag{3.47}$$

where $V_\epsilon = \{v_i | (\forall v_i \in V) v_i \geq \hat{v}_\epsilon\}$, which is a subset of V that contains elements larger or equal to $\hat{v}_\epsilon$. Thus, the probability that the value of a single action in the current action sample is larger or equal to $\hat{v}_\epsilon$ is θ. We can sample m actions and compute the probability that there is at least one action in the m actions with value larger or equal to $\hat{v}_\epsilon$. This probability is $1-(1-\theta)^m$. To determine the sample size m, we solve

$$1-(1-\theta)^m \geq 1-\delta \tag{3.48}$$

and obtain the smallest m that satisfies this inequality:

$$m = \left\lceil \frac{ln(\delta)}{ln(1-\theta)} \right\rceil. \tag{3.49}$$

Note that m is determined incrementally. In each step, we sample one action (without replacement) and add it to the current sample. We then re-compute $\hat{v}_\epsilon$, θ, and m. We continue sampling possible actions until the current sample size exceeds

the current m. We finally choose the best-sampled action. We set $\epsilon = 0.1$ and $\delta = 0.1$ in this application.

Random sample greedy search is based on the statistical assumption that the distribution of the current sample is the same as the distribution of the complete sample. Therefore, this method could perform fairly bad when the sample size is small, either selecting an action far worse than the best action or sampling out of all possible actions without achieving speed gain. In the scheduling application, the action sample size is not very big, which is only about 20. However, we still employ this method. We employ it for the following two further reasons. First, there are several distributions of data for which RSGS can work exceptionally well. For example, for the case that there are 10 actions having a single value and other 10 actions having another value, RSGS chooses a best action with much less time than pure greedy search. We found that scheduling repairs often encounter similar situations, where several actions have very close values. Second, since we perform value function approximation and the function will not be learned optimally, pure greedy search based on complete evaluation of all possible actions may not be very useful. A better-evaluated action is not necessarily better than a worse-evaluated action.

3.6 Backward TD(λ)

This section presents a study comparing of backward TD(λ) and standard TD(λ) under function approximation, which shows that backward learning learns more accurately and more efficiently than forward learning. We consider using TD(λ) to solve deterministic state-space search problems. We typically consider a state space with a fixed starting states and a set of final states.

Before considering function approximation, we first look at the learning scenario with the table representation. Under this pure setting, we see that the following factors are important to the speed of TD(λ) for computing an optimal policy and the optimal value function.

- **Exploration:** The speed of this computation largely depends on the amount of time spent to obtain the best trajectory from a starting state to a desired goal state. Directed exploration usually works better than random-walk based exploration for finding the best trajectory. As long as we get this trajectory, we use the actual value to update the value for the states along this trajectory. If the value function is never over-estimated, then the optimal policy for all states in the trajectory is obtained. In TD learning, exploration must finally be turned off to ensure that learning finally follows the current policy.

- **Initialization:** We should make the initial values small enough that they are not over-estimated. The value-update process gradually increases the values and eventually reaches the optimal values.

- **Loops in a trajectory:** Ideally, we should avoid having loops in a trajectory. Having loops not only is inefficient but also may cause troubles. When using the accumulating eligibility trace, the value function can be over-estimated when a loop is encountered [Singh and Sutton, 1996]. If we can not completely avoid loops, we should employ the replacing eligibility trace to obtain more reliable results.

- λ **parameter:** The standard TD(λ) algorithm updates values strictly on-line. The λ parameter is important to the efficiency of the learning process. A good strategy for choosing the λ parameter should work as follows. When we obtain a best-known trajectory, we use $\lambda = 1$, which works exactly the same way as update using the actual value. When we have a bad trajectory, which can be a very long trajectory or a trajectory ended in a low-quality solution, we use a small value of λ to reduce the impact of the final reward on earlier states. However, this strategy it is difficult to use in on-line learning. We must explore until reaching a final state to determine whether the current trajectory is the best. Therefore, a fixed λ parameter is often used in standard on-line learning.

- **Learning rate** α: For fast computation, learning rate α should be set to 1 when learning with the best trajectory. When learning with a bad trajectory, we could use a small α value to reduce the negative impact from the next state.

Under the table representation, it seems that RTDP should learn more efficiently than TD(λ). In RTDP, values are updated using the Bellman update, which employs the largest value of the state over all possible resulting states to update the value of the current state. RTDP can afford more exploration: the exploration rate β does not need to reduce to 0 in the end of learning. Also, we do not need to worry about learning rate α in RTDP (α is set to 1).

While it is quite clear what we should do for this computation under the table representation, many problems exist when function approximation is applied. With function approximation, the learning performance would largely depend on what kind of approximation method is used. TD-update combined with the use of λ and α parameters provides a robust learning mechanism, which could result in a more smooth and stable function approximation than pure Bellman update. Furthermore, under function approximation, excessive exploration may not be helpful, which could make the value function out off focus. The simplest approach to success probably is to learn a good value function focusing only on states along a good trajectory.

We now presents results that are important to the success and the speed of TD(λ) combined with back-propagation neural networks. The most important factor in determining learning performance probably is to ensure that learning performs properly for the best trajectory. If we can make the estimates for the states in the best trajectory larger than the estimates for its neighborhood, then the greedy policy will pick the best trajectory and get the optimal solution quickly. So let us consider learning under the best trajectory. We know that under the table representation, TD(λ) with $\lambda = 1$ and $\alpha = 1$ learns most efficiently and it is exactly the same as learning using the actual values or TD learning in the backward order. However, under function approximation, does this parameter setting still work well? If not,

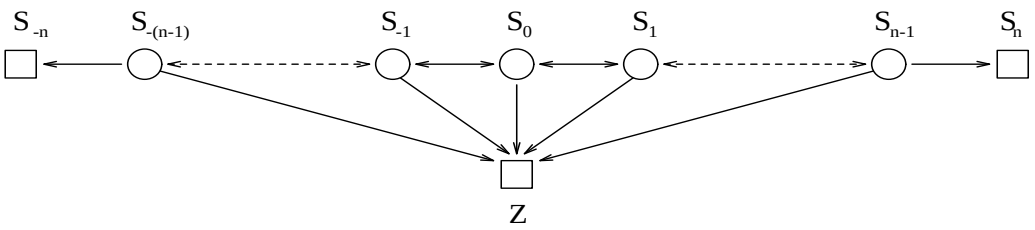


Figure 3.4. A walk-control problem. The agent can move to the left, to the right, or move down to get immediate reward 0.5. The agent receives a reward 1 if it reaches the right boundary s_n, but it receives nothing if it reaches the left boundary s_{-n}.

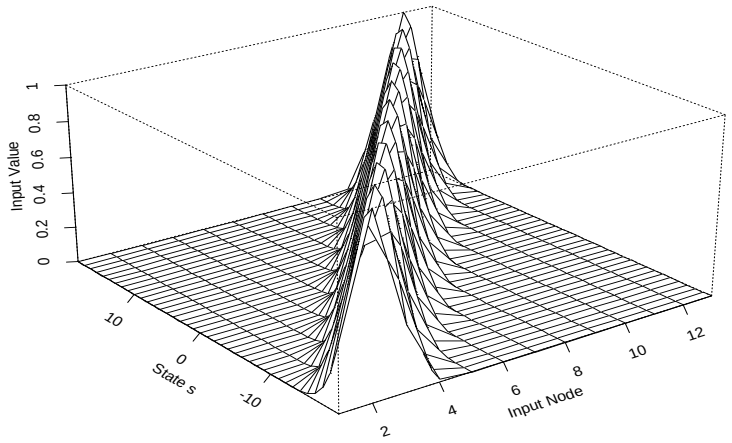


Figure 3.5. Input representation of a walk-control problem with $n = 20$. 13 input nodes are used.

what should be the best setting for these two parameters? Could learning in the backward order perform better?

3.6.1 A Walk-Control Problem

The scheduling task is too complex to analyze these questions in great detail. We consider a simple problem that still contains fundamental characteristics of state-space search, which is a *walk-control* problem—a modification of the random-walk problem.

Figure 3.4 shows a walk-control problem. All walks begin in the center state s_0. States $s_{-(n-1)}, \ldots, s_{-1}, s_0, s_1, \ldots, s_{n-1}$ are non-terminal states. From these states, an agent can move to the left, to the right, or move down to state z to get immediate reward 0.5 and terminate. The agent receives a reward 1 if it terminates at the right boundary s_n, but it receives nothing if it terminates at the left boundary s_{-n}. A move involves a small cost; this tends to discourage to walk back and forth in a pair of states.

To solve large problems, we use overlapping Gaussian ranges [Pomerleau, 1991] as an input representation. Overlapping Gaussian ranges can be considered as an extension of the CMAC representation. While the CMAC uses a boolean value to tell whether a region is considered by a feature, overlapping Gaussian ranges smooth values in the boundary of the region. We write a single dimensional value $v(s)$ for each non-terminal state s from left to right as: $-(n-1), \ldots, -2, -1, 0, 1, 2, \ldots, n-1$ and normalize $v(s)$ into [0,1]. We encode each state as a 13 feature input vector. The value of the ith element ($i = 1, \ldots, 13$) in the input for state s is

$$Input_i = e^{-\frac{([10v(s)-i+2]/12)^2}{0.007}}. \tag{3.50}$$

Let us consider a problem with $n = 20$. This problem contains 42 states, among them 39 are non-terminal states. Figure 3.5 shows the input value with 13 input nodes for the problem. For example, the input value for state s_{19} (the state next to the right boundary) is $(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.017, 0.36, 1, 0.36)$; the input value for s_{18} is $(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.046, 0.575, 0.932, 0.196)$. The cost of an action is set at $c = 0.01$. We employ a linear function to represent the value function and use gradient descent as in back-propagation to approximate the function.

3.6.2 Learning Forward

The conventional TD(λ) algorithm works in the forward fashion. An eligibility trace records all previous states visited from a starting state. For each sequence, we compute the current weight-update amount at each state and accumulate this amount

Table 3.5. Five goals for learning to walk.

	Policy Correctness	Mean Prediction Error
`Goal-1`	> 0.9	< 0.02
`Goal-2`	1	< 0.02
`Goal-3`	1	< 0.01
`Goal-4`	1	< 0.002
`Goal-5`	1	< 0.001

over all states along the sequence. Weight update is performed after each sequence. Under this condition, [Sutton, 1988] proves that TD(1) is exactly the same as using the actual final value as the output value.

Let us consider how to learn the value function given that we have obtained the best trajectory. The best trajectory is the walk starting in s_0 and ending in s_{20} without cycles. Initially, the weights in the linear function are set as follows: the intercept weight is set to 0.5 and the weights for all inputs are set to 0, thus, the initial value estimate for all states is 0.5. This is intended to perform a pure random walk initially. Because we only consider best-trajectory learning, the values for states $s_{-1}, s_{-2}, \ldots, s_{-19}$ are not changed significantly. For states $s_0, s_1, \ldots, s_{19}$, however, after learning their values should become $0.8, 0.81, \ldots, 0.99$, respectively. These are actual values for these states. Learning performance is measured only on this part of state space. We measure the mean error of the prediction of the actual value of a state over states $s_0, s_1, \ldots, s_{19}$. We also measure the percentage of correct decisions in the greedy policy over these states, which we call the *policy correctness*.

Figure 3.6 shows the performance for learning with $\lambda = 0, 0.1, 0.2, \ldots, 0.9, 1$ using the best α for each λ. The horizontal axis shows different λ settings, and the vertical axis gives the log-scaled mean prediction error. Learning with each λ setting is tested with different α values in a fine grain in domain (0, 1]. The upper curve shows the

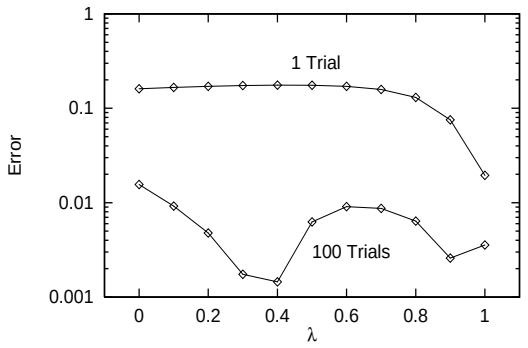


Figure 3.6. The mean error for learning with different λ using the best α.

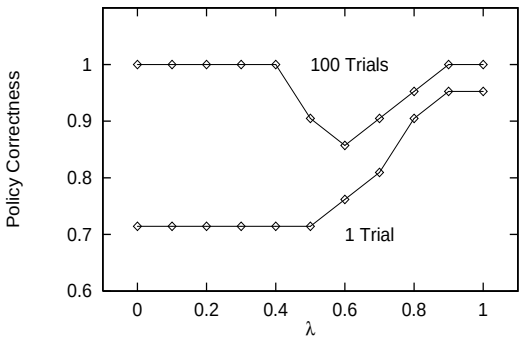


Figure 3.7. The policy correctness for learning with different λ using the best α.

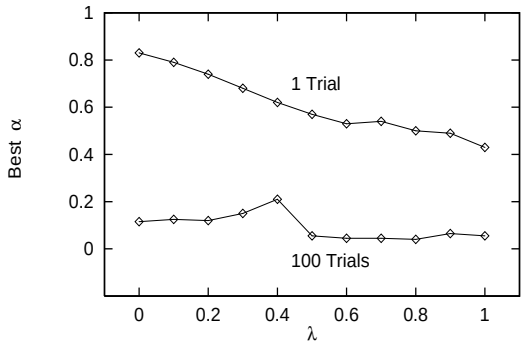


Figure 3.8. The best α for different λ.

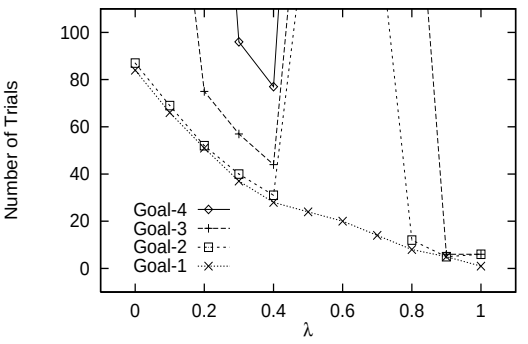


Figure 3.9. The number of trials of update required to reach a set of goals for learning with different λ using the best α.

performance of different λ values after one trial on the best trajectory. We see that $\lambda = 1$ learns the most for one trial. The lower curve presents the performance after 100 trials. Now $\lambda = 0.4$ gives the best performance. We see that λ in the range between 0.5 and 0.8 encountered troubles in learning the value function. $\lambda = 0.9$ and 1 perform better than λ in this range but worse than $\lambda = 0.3$ and 0.4. Figure 3.7 presents the policy correctness for learning with different λ using the best α. We clearly see that λ in the range between 0.5 and 0.8 is not able to learn the correct policy with 100 trials. This tells us that with function approximation, a big λ could encounter some pitfalls and fail to learn an optimal policy. The best learning rate α for different λ values with one trial and 100 trials is presented in Figure 3.8.

We tested the learning speed with five different goals, each goal is measured with a threshold of the policy correctness and a threshold of the mean prediction error. For each goal, if learning performance exceeds the thresholds of the goal, learning is succeeded. These goals are ordered monotonically from the easiest to the hardest. Table 3.5 shows these five goals. Figure 3.9 presents the learning speed for reaching these goals with different λ values using the best α. We see that $\lambda = 1$ and 0.9 learn quickly for reaching the first three goals. For `Goal-4`, $\lambda = 0.4$ is the best setting. It conducts 77 trials of weight update and reaches the goal. $\lambda = 0.3$ needs 96 trials. All other settings are not able to reach the goal within 100 trials. We also see that λ in the range between 0.5 and 0.8 is even not able to reach the second goal. Finally, none of learning settings is able to reach `Goal-5` with 100 trials of learning.

3.6.3 Learning Backward

We conducted analogous experiments with backward TD(λ). In backward TD(λ), the states are presented in reverse order from the final state s_{20} to the starting state s_0. Update for weights is conducted immediately after the update amount is computed for a state-pair. Under this condition, learning with $\lambda = 0$ and $\alpha = 1$ is the same as learning with actual values given that the table representation is employed.

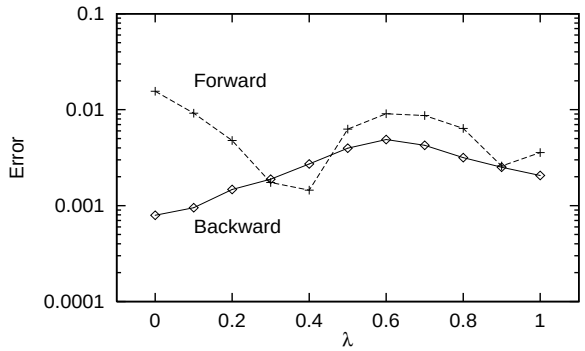


Figure 3.10. Compare forward learning and backward learning in the mean error for learning with different λ using the best α.

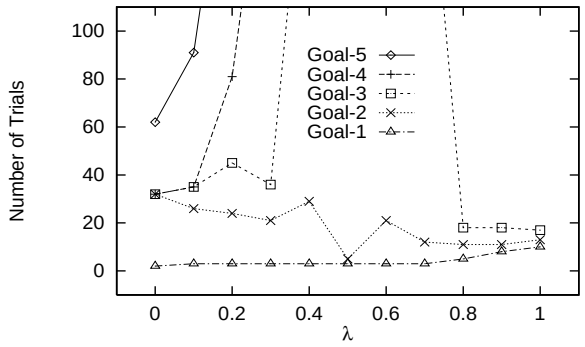


Figure 3.11. The number of trials of update required to reach a set of goals for backward learning with different λ using the best α.

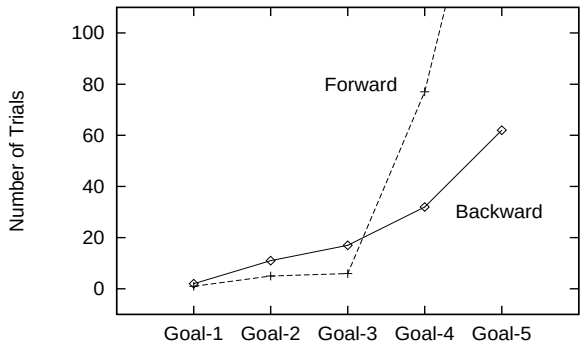


Figure 3.12. Compare forward learning and backward learning in the number of trials of update required to reach a set of goals using the best λ.

When $\lambda > 0$, we have an eligibility trace which covers all states starting from the current state to the final state. While it does not make sense to use the trace under the table representation, because it over-estimates the value for states in the trace, with function approximation, using $\lambda > 0$ may speed up learning an optimal policy. For example, when we learn along the best trajectory in the walk-control problem, a big λ could help to keep the estimate for a later state (one closer to the final state) larger than the estimate for an earlier state. Thus, an optimal policy could be obtained more quickly. For back-propagation neural networks, in general, learning tends to intensify memorizing the current pattern while the patterns trained in the past are gradually lost. Learning by repeatedly presenting the sequence could lead to oscillation if every time we only consider the current pattern. With some $\lambda > 0$, the chance of oscillation can be reduced, because the trace would place some emphasis on the patterns trained in the past. The parameter λ acts as a smoothing factor, which is called a *momentum* term in supervised back-propagation learning.

Figure 3.10 compares forward TD(λ) and backward TD(λ) on the mean prediction error with 100 trials of training along the best trajectory. It shows the performance of different λ settings, each using the best α. We see that backward TD(λ) with $\lambda = 0$ performs the best. Its mean prediction error is much smaller than the error of the best forward TD(λ) setting: $\lambda = 0.4$.

Figure 3.11 shows the speed of backward TD(λ) in learning to reach the five goals. We see the $\lambda = 0$ requires the fewest number of trials to reach the first goal. However, a big λ ($\lambda \geq 0.8$) is faster in reaching `Goal-2` and `Goal-3`. `Goal-1` and `Goal-2` have the same threshold for prediction error but differ in their policy correctness thresholds. `Goal-1` requires 90% correct decisions but `Goal-2` requires 100% correct decisions. Therefore, we see that a big λ make it easier to find the optimal policy. We now look at learning for more challenging goals: `Goal-4` and `Goal-5`. These two goals put high requirements on the mean prediction error. Only learning with small λ can reach these goals within 100 trials, and $\lambda = 0$ performs the best. This mean that if we want to learn a perfect value function, we should employ $\lambda = 0$. However, this could requires more time to obtain an optimal policy than using a bigger λ value.

Figure 3.12 shows the comparison of the speed of forward TD(λ) and the speed of backward TD(λ) for these goals under their best learning setting. Both methods reach the first three goals quickly, whereas forward learning is faster. However, forward learning slows down dramatically in learning for `Goal-4` and `Goal-5`. The best forward learning method requires 77 trials of the best trajectory to reach `Goal-4` and can not reach `Goal-5` in 100 trials. In contrast, the best backward learning method only takes 32 trials to reach `Goal-4` and takes 62 trials to reach `Goal-5`.

Forward learning can be performed in a strict on-line fashion. To perform backward learning, we must store the sequence of states visited along the current trajectory and then update the network. Nonetheless, the overhead of memorizing a single sequence is not very great. Therefore, for application domains that do not require a strict on-line system, we suggest using backward learning to improve learning performance.

Furthermore, the results also suggest that we should employ multiple weight update for an experience in order to learn the most from it. This is important in active learning, because each experience is obtained at some cost and some experiences may appear very rarely. In general, multiple updates with a small learning rate work better than a single update with a big learning rate.

Chapter 4
Learning with Hand-Engineered Features

This chapter is organized as follows. We first demonstrate the main results presented in the [Zhang and Dietterich, 1995] paper. We next describe an earlier preliminary study that was conducted to choose good features for describing schedules and to design the network architecture. The remainder of the chapter describes a series of experiments performed to help us understand how and why TD(λ) worked in the SSPP problem.

4.1 Main Results

4.1.1 Features

We employ TD(λ) (with modifications) to learn a neural network as the value function. The neural network can accept only a fixed vector of feature values describing each state (i.e., each current schedule). Schedules, on the other hand, are variable-length objects. Hence, it was necessary to define a set of useful features that extract important aspects of the current schedule that the neural network can use to predict the value of the state. We defined the following features:

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

Mean and standard deviation of slacks: The slack time between a task and one of its temporal prerequisites is the difference between the end time of the

prerequisite task and the scheduled start time of the task. We measure the minimum slack for each task (and all of its temporal prerequisites) and the average slack for each task. The mean and standard deviation of these two quantities taken over all tasks provide four features.

RDF: We found that RDF is an important predictor for the final value. We use two RDF features, the second feature only gives the RDF for final schedules (and has the value 0 for all other schedules).

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 compute 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 problems: We finally define two features to discriminate different scheduling problems. One is the resource utilization of the initial critical path schedule $Util_{CPS}$. The other is the average of job due-time separations in a problem t_{JDS}.

Each of these features was developed by studying small scheduling problems to find quantities that had some ability to predict final RDF. Section 4.2 describes our preliminary study on features in more detail.

4.1.2 Problem Sets

We constructed two problem sets: an artificial problem set ART-1 and a problem set SSPP based on specifications for the NASA SSPP problem (See further information about problem sets in Appendix). The artificial problems were generated as follows. First, we generated a pool of 20 jobs. From these, we constructed scheduling problems by choosing random subsets of these jobs. This was intended to model the NASA setting where there are only a limited number of possible shuttle-cargo-bay configurations (i.e., jobs), but where each scheduling problem is a unique combination of such shuttle missions. More generally, this models a job shop where each new scheduling interval requires scheduling a unique mix of more-or-less standard jobs.

To generate a synthetic job, we choose the number of tasks randomly in the range 6 to 10. A set of temporal constraints among these tasks is then randomly generated such that approximately 60% of all possible pairwise precedence constraints are asserted.

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

Once the pool of 20 jobs is generated in this way, 50 training problems and 50 test problems are constructed. To generate a problem, we first choose the number of jobs in the problem to be either 3 or 4 (with equal probability). The desired number of

jobs is selected randomly with replacement from the 20-job pool. Each job is assigned a completion deadline with the deadlines randomly separated by between 8 and 15 time units.

Sixteen input features are computed to represent schedules for these problems: 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.

During training, 15 of the 50 training problems were held out as a validation set to determine when to halt training. The remaining 35 problems were repeatedly processed to train the value function networks.

In addition to the 50 test problems, we generated a second test set of 20 larger problems to evaluate the ability of the learned value functions to scale up to larger scheduling problems. Each of these larger problems was generated in the same way as the smaller problems except that the number of jobs was chosen uniformly between 15 and 20.

For the space shuttle payload processing task, a problem consists of a set of shuttle missions with launch dates one to three months apart. Each mission can have one or two payloads. We considered three kinds of payloads: long module (LM), mission peculiar equipment support structure (MPESS), and pallet and igloo (PALLET & IGLOO). These have 65, 32, and 82 tasks, respectively. There are 35 types of resources of which only five are major bottleneck resources.

We randomly generated a training set of 20 problems and a test set of 20 problems. The training problems each contained between two and four shuttle missions. Of the 20 training problems, 5 were held out for validation to determine when to stop training. The test problems each contained 3 to 6 shuttle missions. The test problems thus assess the ability of the learned policy to scale up to larger problems.

For the shuttle problems, 20 input features are used: 10 features for pool capacity, 4 slack features, the RDF, 2 features describing windows in violation, percentage of time units in violation, index of first violated window, and the overallocation index.

4.1.3 Network Architecture and Training Procedure

To represent the value function, we trained feed-forward networks having 40 sigmoidal hidden units and 8 sigmoidal output units. The 8 output units encode the predicted RDF using the technique of overlapping Gaussian ranges [Pomerleau, 1991] as follows. Each output unit represents one assigned RDF value, v_j ($j = 1, \ldots, 8$). For the artificial problems, these RDF values are $v_1 = 0.8, v_2 = 1.0, \ldots, v_8 = 2.2$. For the SSPP problems, the RDF values are $v_1 = 0.9, v_2 = 1.0, \ldots, v_8 = 1.6$. During training, the target output activation for each output unit is set to be

$$target_j = e^{-\frac{(v_j - RDF)^2}{0.08}}. \tag{4.51}$$

For example, when the RDF is 1.32, the value of the vector of 8 output units $(target_1, target_2, \ldots, target_8)$ is $(0.034, 0.278, 0.835, 0.923, 0.375, 0.056, 0.003, 0.000)$. During testing, the predicted RDF value is computed as

$$\hat{f} = (\sum_j act_j \cdot v_j)/(\sum_j act_j), \tag{4.52}$$

where act_j is the actual output activation for output unit j.

For each problem, 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 $\lambda = 0.2$ or 0.7. The training set problems are processed in round-robin fashion. Each problem is solved using one of the networks to obtain a sequence of states and actions. That network is then updated (via backpropagation with $TD(\lambda)$) by processing the state sequence working backward from the final state. For each state-pair, we employ multiple weight update to over-train big errors (but did not use the automatic learning rate mechanism). The best-experience replay strategy is employed. After every 50 passes through the training set, a cross-validation test is conducted to compute the average RDF of the final schedules produced over all cross-validation problems. The best network found during cross-validation (for each of the eight parameter sets) is retained. For each network, training continues until the cross-validated RDF of that network is worse than the previous nine measured values for cross-validated RDF.

Six networks are chosen for testing as follows. The three best networks found during cross-validation are retained along with their corresponding final networks. We retain the final networks to compensate for variance in the cross-validation measurements. Therefore, the resulting scheduling system runs six iterations, one for each of these six selected networks.

The TD scheduler employs a greedy policy using the value function. To avoid loops, the dynamic greedy policy described in Section 2.6.3 is used. When a loop is detected, we apply the learned value function to compute the *second best* action and choose it. If a loop is detected again at the same state, we backtrack to the preceeding state and again take the second best action. If this were to create a loop also, we would continue backtracking to earlier states. The randomness introduced by the random sample greedy procedure also tends to avoid loops, because even when the same state is revisited, the same action may not be chosen.

4.1.4 Performance Results

Figure 4.1 shows the average cross-validation RDF for the four value-function networks trained with $\alpha = 0.1$. The horizontal axis gives the number of training sequences processed. This figure shows that the performance of the trained networks is improving on the cross-validation problems. Figure 4.2 plots the number of repair actions for these same networks. This shows that there is some reduction in the number of actions required to convert the starting schedule into a conflict-free final schedule.

We now present the test performance. Both temporal difference scheduling and the iterative repair method of Zweben are run on the test problems. For the simulated annealing component of the iterative repair method, we set the starting temperature to 100 for the synthetic scheduling task and to 200 for the SSPP task. After every 10 accepted repairs to the schedule, the temperature is reduced according to $T := 0.95T$. The cost function describes the number of constraint violations.

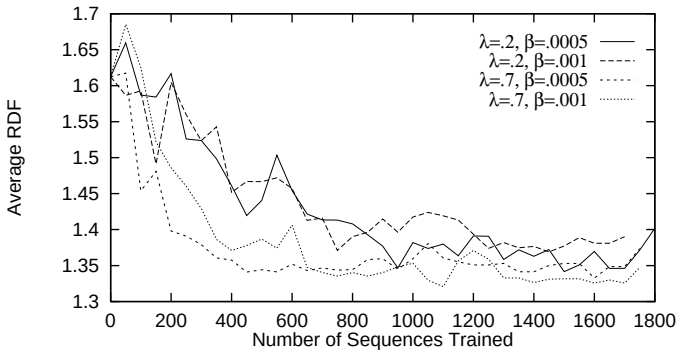


Figure 4.1. Average RDF over 15 CV Problems.

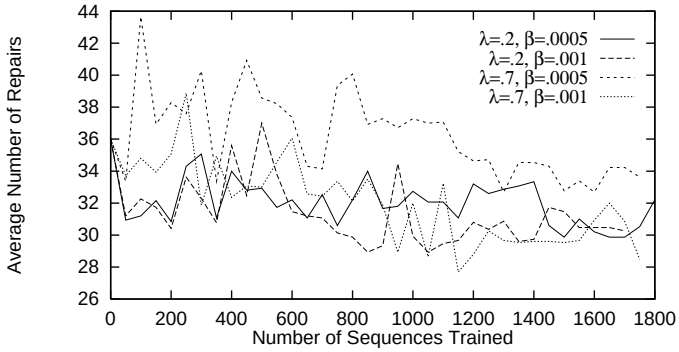


Figure 4.2. Average Number of Repairs over 15 CV Problems.

Figures 4.3 compares the performance of temporal difference (TD) scheduling with the iterative repair (IR) method. The vertical axis is the RDF of the best conflict-free schedule found so far. The horizontal axis is a machine-independent proxy for the amount of CPU time consumed by each method. For IR, the horizontal axis gives the number of restarts of the simulated annealing procedure, and the vertical axis records the RDF of the best conflict-free schedule found so far. The longer IR is run, the better its performance.

For the TD scheduler, the horizontal axis represents the number of neural network evaluation functions employed. When k networks are used to solve a scheduling problem, the problem is solved k times, once with each network, and the schedule having the best RDF is returned as the answer. The best k networks, as determined

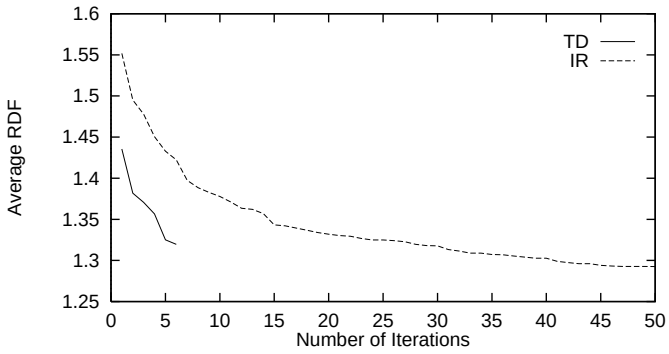


Figure 4.3. Performance Comparison of TD to IR on 50 Small-scale Problems.

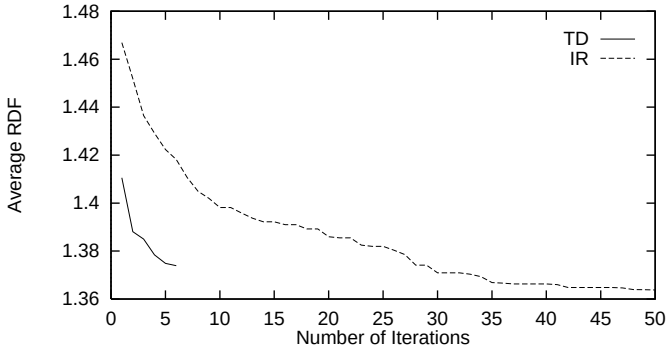


Figure 4.4. Performance Comparison of TD to IR on 20 Medium-scale Problems.

by cross-validation, are used. The curves stop at $k = 6$, because only six networks were used (once each).

Some care must be taken in interpreting the horizontal axis as a measure of CPU time. Each step of the TD scheduler requires more CPU time than a step of the IR scheduler, because the TD scheduler must perform the random sample lookahead search and check for loops. On the average, TD spends 2.2 times as much CPU time per step as IR. On the other hand, TD requires fewer steps to find a conflict-free schedule. The average sequence length for an iteration of TD is 82% as long as an average IR sequence. The net effect is that one iteration of TD is equivalent to approximately 1.8 iterations of IR.

Bearing this in mind, the key point to notice is that the curve for the TD scheduler always lies below the curve for iterative repair. This means that given the same amount of CPU time, TD always finds a better schedule (i.e., with lower RDF). For example, with 6 networks, TD obtains an RDF of 1.320 compared to IR's RDF of 1.371 (at $1.8 \cdot 6 = 11$ iterations). This is a 3.9% improvement, which in a schedule lasting a year is a savings of 14 days (and thousands of dollars). The curve also shows that iterative repair always requires much more time (29 iterations vs. 11) to find a schedule whose quality matches the RDF found by TD.

Figure 4.4 shows a similar comparison for TD and IR on the 20 larger test problems. Here the difference between the algorithms is even more pronounced. Temporal difference scheduling scales better to larger problems, even though it has only been trained on smaller problems.

Figure 4.5 shows analogous results for temporal difference and iterative repair on the 20 test-set SSPP problems. Here the horizontal axis is log CPU time. We see that TD maintains a constant *factor* advantage over iterative repair. Temporal difference scheduling finds better schedules faster than iterative repair.

Note, however, that this figure just gives the average RDF over the whole test set. Because of the random components of both algorithms, this hides considerable variation. Figure 4.6 reveals this variation. Let us say that TD "wins" on a particular problem 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 4.6 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. Eventually, as CPU time becomes larger, TD still wins or ties slightly more than 50% of the time, provided that TD stops running much early after 37,114 seconds.

In subsequent experiments, we found that if we let TD run further with more iterations, the TD solutions will keep improving. This will keep TD win in most problems over time. To let TD run further, we can add more good networks to TD. We can also employ some methods to make more use of a single network. A simple

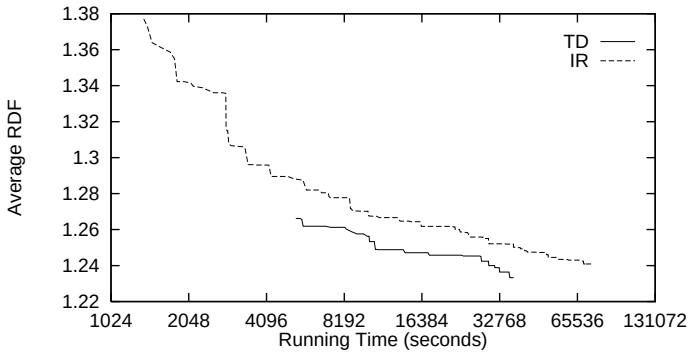


Figure 4.5. Performance Comparison of TD to IR on 20 SSPP Problems (RDF).

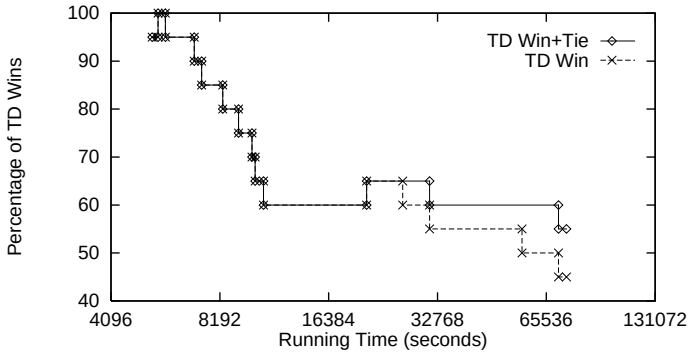


Figure 4.6. Performance Comparison of TD to IR on 20 SSPP Problems (% Wins).

effective method is to allow TD to choose the second best action with some probability according to the difference between the prediction of the best two actions.

4.2 Results of the Preliminary Study

To develop good features and a good network architecture, we conducted extensive preliminary studies. This section reports the results of those studies.

4.2.1 The Approach

Our studies were performed using off-line supervised learning based on a set of the small-scale problems from the ART-1 problem set. We carried out the following steps:

- **Develop a set of features:** As described in Section 4.1.1, we developed features based on the capacity analysis of a schedule. The key features that describe a schedule would include the resource utilization for each bottleneck resource pool and slacks in a schedule. We also developed features in order to make prediction of the final RDF easier. Therefore, we defined features that could be good predictors such as the current RDF and a number of violation-counting features. Furthermore, two features were defined to discriminate different scheduling problems. They are initial resource utilization and the job due-time separation. Table 4.1 lists 22 features for the ART-1 problems.

- **Generate a sample of the state space:** A sample of the state space was generated by running the iterative repair algorithm on a set of ART-1 problems. We then transformed this sample into a set of examples where each example was described with the 22 features and a target value. The target records the value function computed based on all the states in the sample.

- **Test features:** We employed statistical analysis and supervised learning to test the significance of each feature. The major techniques we applied included scatter plots, sequence plots, and a number of modeling methods such as linear regression, regression trees, and neural networks.

- **Test network architectures:** A number of neural network architectures were tested. The key goal was to find a small-size network that can perform well in predicting the value function. A small network is cheap to evaluate and helpful for speeding up the scheduling process.

Table 4.1. Features.

Name	*Feature*	*Description*
F1	$E(Util(pool_1))$	Mean of the 1st pool utilization
F2	$SD(Util(pool_1))$	SD of the 1st pool utilization
F3	$E(Util(pool_2))$	Mean of the 2nd pool utilization
F4	$SD(Util(pool_2))$	SD of the 2nd pool utilization
F5	$E(Util(pool_3))$	Mean of the 3rd pool utilization
F6	$SD(Util(pool_3))$	SD of the 3rd pool utilization
F7	$E(Util(pool_4))$	Mean of the 4th pool utilization
F8	$SD(Util(pool_4))$	SD of the 4th pool utilization
F9	$E(Slack1)$	Mean of the minimum slacks
F10	$SD(Slack1)$	SD of the minimum slacks
F11	$E(Slack2)$	Mean of the average slacks
F12	$SD(Slack2)$	SD of the average slacks
F13	RDF	The current schedule RDF
F14	$Util_{CPS}$	The initial schedule resource utilization
F15	n_v/n_{wds}	Percentage of windows in violation
F16	n_v	Number of windows in violation
F17	$n_{v10}/10$	Percentage of first 10 windows in violation starting I_{v1}
F18	L_v/L	Percentage time units in violation
F19	n_p/n_v	Percentage of violated windows for pool-reassignment
F20	RDF for finals	RDF for final schedules and 0 for all others
F21	I_{v1}/n_{wds}	First violated window index
F22	t_{JDS}	Mean of job due-time separations

4.2.2 Significance of Features

We now present the methods that were employed to test features and some results obtained from the test.

Scatter plots: Scatter plots reveal relationships between a pair of variables. Figure 4.7 presents the scatter plots for the relationship between each feature and the value function target. It plots a sample containing 100 sequences of 5 ART-1 problems, each problem with 20 sequences. To generate this sample, we ran 100 restarts of the iterative repair procedure on each problem and selected the 20 best sequences. This sample, named $\mathcal{S}_{100}$ due to its 100 sequences, includes 2840 distinct states. All features and the target values were normalized.

From the figure, we can see that none of these features alone has high correlation with the target. This means that prediction by using these features could be difficult. The difficulty is due to the characteristics of the value function. The value function actually represents the structure of the state space: how rewards are distributed in a state-space and how state transitions are performed. In scheduling, it is not obvious how to construct a set of features that can capture such structure, particularly, about state transitions. Among these features, we see that F13—the current schedule RDF—has a better correlation with the final value than any of the other features.

Scatter plots can also reveal correlations between features. We found that several features are correlated in some small degree. For example, the mean and the standard deviation statistics of a variable reflect a certain degree of correlation. Analysis of variance is helpful for analyzing these correlations.

Sequence plots: It is important to know how the features and the target change through a sequence from the starting schedule to a final schedule. Figure 4.8 demonstrates sequence plots for all features and the target. We see that the features are generally good at measuring how search is progressing from the starting state to a final state. For example, violations are gradually reduced but slacks are gradually increased.

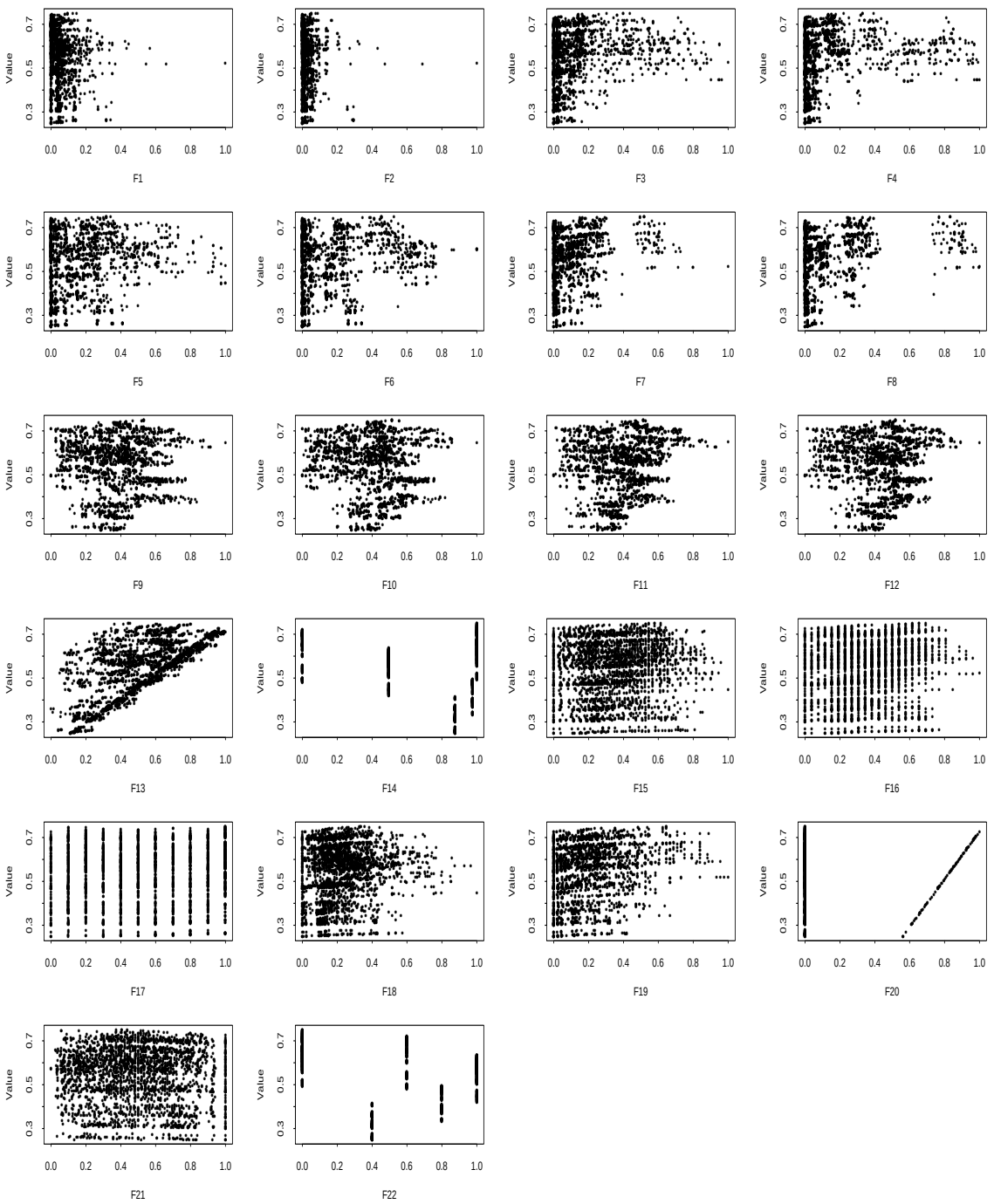


Figure 4.7. Scatter plots for 22 features. The sample contains 100 sequences for 5 ART-1 problems, each problem with 20 sequences. There are 2840 distinct states. This presents the scatter plot of each feature (horizontal axis) against the true value (vertical axis).

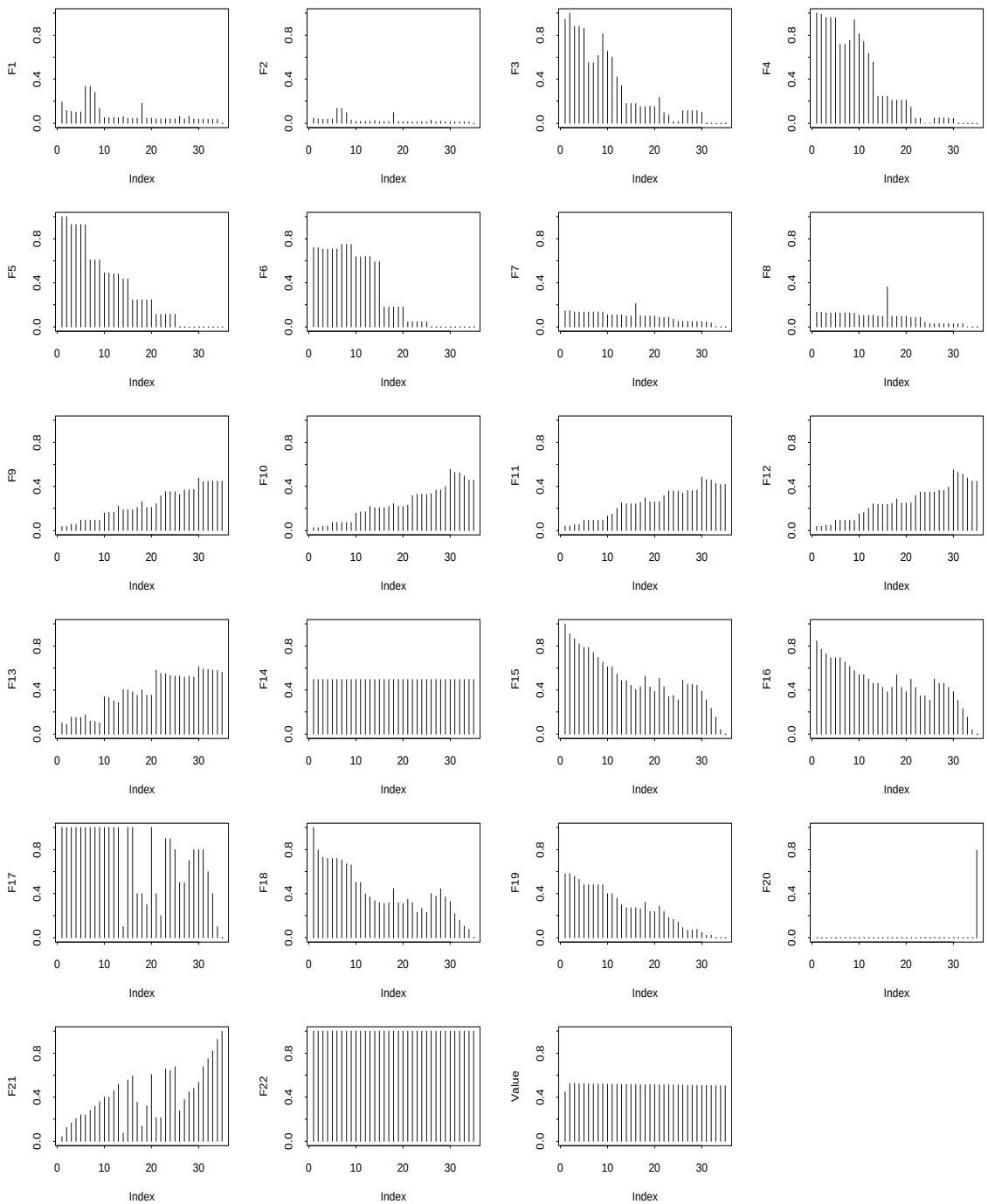


Figure 4.8. Sequence plots for 22 features and the value. The horizontal axis is the index of the states along a sequence from the starting schedule to a final schedule. The vertical axis is the value of the feature.

An important property about sequence plots is that if a sequence is an optimal path, then the target values for the states along the sequence must be smooth: the difference between the values of the adjacent states is 0.001, the cost of an action. The value for a state is the maximal value that can be achieved from this state. When a value jumps at a state, this means that the previous action that resulted in this state was not optimal. In the last plot in Figure 4.8, we see that the value jumped at the second state. Therefore, there exists a better action (other than the action that leads to the second state in the plot) from the starting state in the sample.

Data brushing and spinning: Data visualization tools are very useful in exploring the correlation and the structure in this data. Two important visualization functions are data brushing and data spinning. Data brushing allows one to interactively highlight a point in one scatter plot and see the same case highlighted in the remaining scatter plots. Data spinning allows one to create a three-dimensional data space for any three variables and then to spin the the data to observe the structure.

Linear regression: The weight for a feature in the linear model can tell how important the feature is (assuming independence of features). In order to make a fair comparison of these weights, we normalized all features into the range from 0 to 1. We also normalized the output (the target) into the range from 0.25 to 0.75. (This output normalization is important for learning a neural network with a single sigmoidal output unit, because the sigmoidal activation function is almost linear only in this range. Otherwise the network would only learn Boolean output functions). Table 4.2 shows the weights for all features by linear regression based on sample $\mathcal{S}_{100}$. The most important five features presented by the model are F18, F15, F13, F19, and F11.

Regression trees: Regression trees [Breiman *et al.*, 1984] are another popular model for uncovering the structure in continuous data. Regression tree methods usually produce a function very different from ones produced by classical statistical models such as linear and logistic regression models. The major advantages of regression trees include the following:

Table 4.2. Significance of the features according the linear and regression tree model.

Name	*Linear*	*Tree*	*Name*	*Linear*	*Tree*
F1	-0.0860	0.37 (5)	F12	-0.2343	0.34
F2	-0.2331	0	F13	0.3997 (3)	1.93 (1)
F3	-0.1769	0.34	F14	-0.0807	1.17 (2)
F4	0.1642	0.83 (4)	F15	-0.4618 (2)	0
F5	0.0630	0	F16	0.0347	0
F6	-0.0466	0.17	F17	0.0142	0
F7	0.1714	0.14	F18	0.5043 (1)	0.17
F8	0.0155	0.31	F19	0.2880 (4)	0
F9	0.0388	0	F20	-0.0031	0
F10	-0.0270	0.29	F21	-0.0170	0
F11	0.2618 (5)	0.25	F22	-0.0299	1.08 (3)

- A regression tree is actually composed of a set of decision rules for prediction, where each rule focuses on a certain part of the problem space. The model is invariant to monotone transformations of features. Therefore, the effect of interference would be small.
- The treatment of missing data and noisy data is more satisfactory.
- The model is capable of dealing with a mix of continuous variables and discrete variables.

A regression tree is generated by recursively partitioning the sample space. In each recursive step, a feature is selected to split the current space into half according to a certain utility measure. Only the feature with the largest utility is selected. Therefore, the more times a feature is selected, the more important the feature would be. Furthermore, a split near the root of the tree would be more important than a

split deeper in the tree. Thus, the importance of a feature can be calculated using the sum of the reciprocal of the depth of a split over all split appearances for the feature:

$$\sum_i \frac{1}{depth_i}.$$

The depth of the root feature is defined as 1. Table 4.2 also shows the significance of features according to the regression tree model. It shows that F13, the current RDF, is the most important feature. The next four important features are F14, F22, F4, and F1.

Neural networks: We employed feed-forward back-propagation neural networks with sigmoidal activations. In general, the bigger a network is, the more powerful the network will be. However, a big network is expensive to learn and to evaluate. In the following section, we will present the experimental results about testing the size of networks.

A simple method to find the role that a feature plays in a network is to check the weights associating with the feature. For a two-layer network, we let $w_{ij}^{(o)}$ be the weight connecting between output unit y_i and hidden unit h_j and let $w_{jk}^{(h)}$ be the weight connecting between hidden unit h_j and input x_k. The significance of a feature can be computed as the total magnitude of the weights associated with the feature. For the feature at input position k, its significance is computed as

$$\sum_j |w_{jk}^{(h)}|,$$

or alternatively as the weighted total magnitude of the weights:

$$\sum_j \left(|w_{jk}^{(h)}| \sum_i |w_{ij}^{(o)}| \right).$$

A popular method for presenting the structure of a learned network is to show all weights graphically by a *Hinton diagram*. Figure 4.9 presents a two-layer network trained with $20,000$ epochs on the sample as described above. The network contains 8 output units and 40 hidden units. In the diagram, the left-hand side shows the weights connected to the hidden layer, and the right-hand side shows the weights connected to the output layer. A black box stands for a positive weight, while a

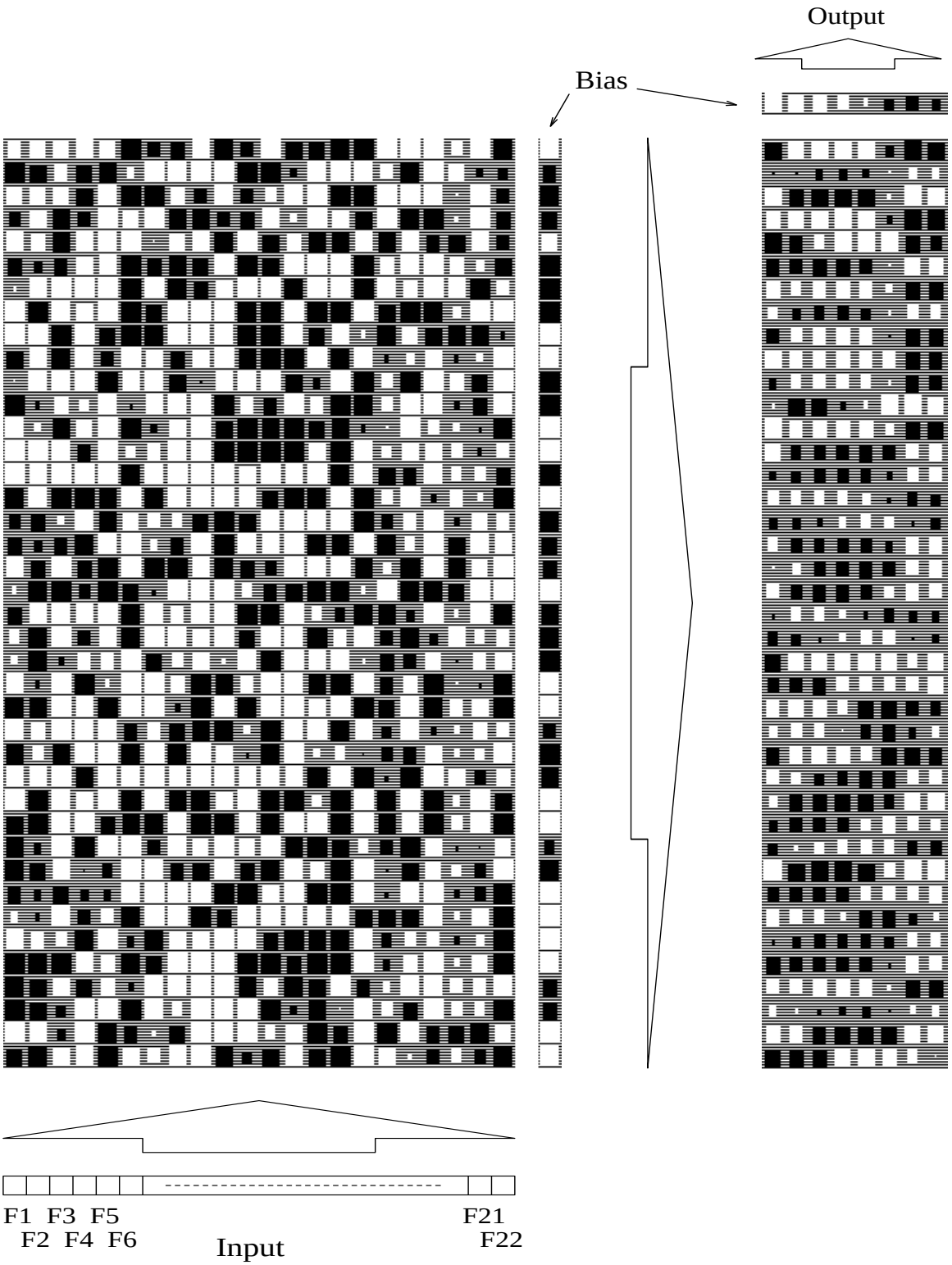


Figure 4.9. The Hinton Diagram. There are 22 inputs, 8 outputs, and one hidden layer with 40 units. The left-hand side shows the weights connected to the hidden layer and the right-hand side shows the weights connected to the output layer.

white box stands for a negative weight. The size of a box tells the magnitude of a weight. From the diagram, we see that almost all features involve some big weights, which means that they all play some important role in predicting the value function.

4.2.3 Network Architectures

We tested neural networks with various numbers of hidden units. Our goal was to find the smallest network that gives good performance. The experimental results showed that a good choice for the number of hidden units in a single hidden layer neural network is in the range between 30 and 40. Learning performance only improves very slightly with a doubling of the number of hidden units from this range. However, a network with 40 hidden units performs significantly better than a network with only 16 hidden units. We also tested whether using more layers of hidden units could help to improve the learning performance. The result is negative. Under about the same number of connections (free parameters) in a network, a network with more hidden layers performs about the same as a network with one hidden layer.

We now turned our attention to the question of how to represent the predicted value at the output units. We use the technique of overlapping Gaussian ranges to make a transformation between a vector of outputs and the single output value. We tested the number of output units needed and the standard deviation σ for the Gaussian function. If σ is small, then the bell shape of a Gaussian becomes narrow, which makes the units away from the center of the Gaussian have target values close to 0. Otherwise, the bell shape becomes more flat and all values are pulled up toward 1. Figure 4.10 presents three overlapping-Gaussian-range transformations with $\sigma = 1/10, 1/14$, and $1/18$ respectively. The output value ranges from 0.25 to 0.75. Note that $\sigma = 1/14$ for this range of output makes a transformation almost equivalent to $\sigma = 0.2$ for output ranging from 0.8 to 2.2, which we used in Section 4.1.

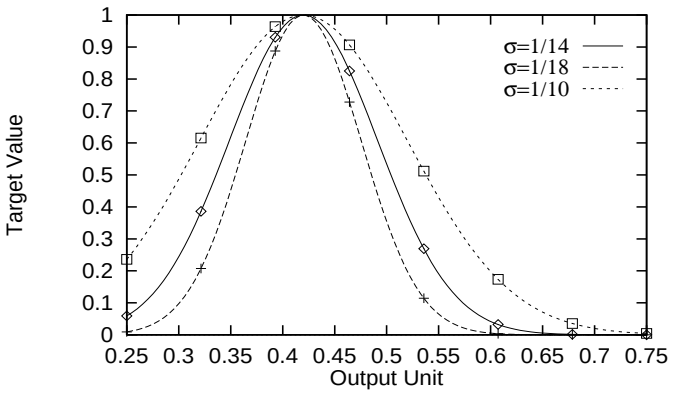


Figure 4.10. Overlapping Gaussian ranges with $\sigma = 1/10, 1/14$, and $1/18$. The plot shows how a single value 0.42 is transformed to the value of an 8-variable vector ranging from 0.25 to 0.75. For example, when $\sigma = 1/14$, the value of the vector is $(0.059, 0.386, 0.930, 0.825, 0.269, 0.032, 0.001, 0.000)$.

To test prediction, we need an accurate transformation from the value of an output vector to a single value. The transformation

$$\hat{f} = (\sum_j act_j \cdot v_j)/(\sum_j act_j)$$

creates errors when the Gaussian of the network output activations is not centered at the middle units. This error can be quite big if the Gaussian is centered at a side unit. Also, a bigger σ tends to make a bigger error. To correct this error, we adjust $\hat{f}$ with a further update as follows:

$$\hat{f} := \hat{f} + c_1(\hat{f} - 0.5) + c_2(\hat{f} - 0.5)^5. \tag{4.53}$$

For $0.05 \leq \sigma \leq 0.1$, the following parameter setting is good:

$$c_1 = 2\sigma - 0.1, \tag{4.54}$$

and

$$c_2 = 2500\sigma - 100. \tag{4.55}$$

Note that this $\hat{f}$ adjustment is needed only for supervised learning, because we need to compute the mean squared error (MSE) of the estimates according to given true values. For reinforcement learning, however, we will not be able to take this error measure, because true values are not available.

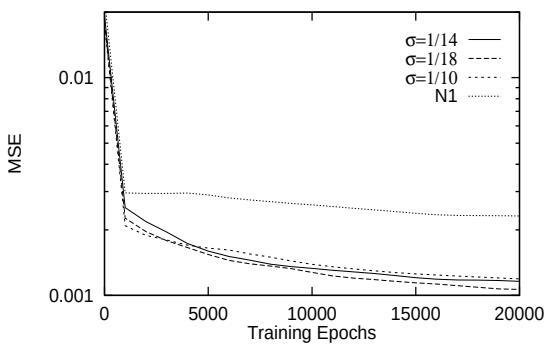


Figure 4.11. Training MSE.

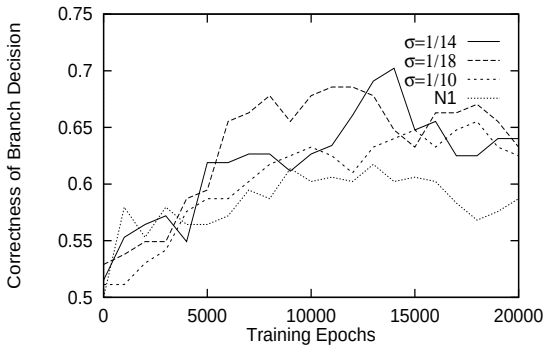


Figure 4.12. Correctness of branch decisions.

We now compare networks with one output unit and networks with eight output units using the overlapping-Gaussian-range transformations with three σ settings: $\sigma = 1/10, 1/14$, and $1/18$. The single-output network contains 60 hidden units, so it has 1441 free parameters (1380 weights + 61 biases). Each eight-output network has 40 hidden units, so each network has 1248 free parameters (1200 weights + 48 biases). We tested these four networks on sample $\mathcal{S}_{100}$.

Figure 4.11 plots the training MSE for the four networks. The horizontal axis gives the number of training sequences processed. This clearly shows that the single-output network N1 performs much worse than the three eight-output networks, even though it uses more hidden units and free parameters. The three eight-output networks perform about the same. Among them, the network with $\sigma = 1/18$ performs slightly better than other two networks. Figure 4.12 plots the correctness of branch decisions for the four networks. We say that a network performs correctly at a state if it chooses the same action as the action recommended by the true values. This measure presents the percentage of correct network choices over all branching states. $\mathcal{S}_{100}$ contains a total of 132 branching states. This plot further shows that network N1 performs worse than the eight-output networks. Among the eight-output networks, the $\sigma = 1/14$ network performs slightly better. Unfortunately, this correctness is quite low and the learning curve on this measure is noisy for all networks. This demon-

strates the difficulty of value function approximation for our reinforcement learning problem. Under the current features, the network is not able to perfectly learn the true values. It is worth mentioning that other function approximation methods such as regression trees and nearest neighbor methods also fail to learn the value function perfectly and they perform worse than the overlapping-Gaussian-range networks.

Nonetheless, the key result presented here is that using overlapping Gaussian ranges improves learning performance significantly. We found that using eight output units is good. When the number of output units increases to 16, the performance gain is quite small.

4.3 Evaluation of Learning Performance

Reinforcement learning has been so far mostly concerned with using convergence and speed of convergence to optimality to measure learning performance. However, for the difficult learning task presented here, these measures are not useful. We are interested in training on a sample of scheduling problems and then applying the learned networks to solve a much larger disjoint set of scheduling problems. Therefore, we employ a cross-validation test to provide several measures to evaluate learning performance. This section presents the performance measures that are employed for this task. Learning behavior under these measures is also demonstrated.

In section 4.1, we presented two cross-validation measures during training:

- **Average RDF of solutions of validation problems:** We have shown that the performance of trained networks is improving. With training, a network is able to find better solutions for validation problems.
- **Average number of repairs needed to construct solutions for validation problems:** We have also shown that with training, a network is able to construct solutions more quickly.

We now introduce three further performance measures to describe the learning behavior.

- **Average training error:** First, we measure the average backpropagation error over a sequence during training. We only use the error in the first update among multiple weight updates for a state pair. Since training involves exploration of the search space, this error can be very noisy. Our measurement takes an average of the error over the past 1,000 sequences.

- **Error of final prediction:** We measure the accuracy of predicted values of states along the search path produced by a learned network. An optimal path produced by the optimal value function has the property that for any state s in this path, the value of the next state s' must equal the value of state s plus the reinforcement assessed by this state-transition, r. Therefore, we can use the prediction error $(\hat{f}(s) - r - \hat{f}(s'))$ for all states s along the path produced by a network to measure how well the value function is learned. We noticed that the biggest error comes from the prediction for the value of the final state, so we separate this measure into two parts: one measures the error for final states and the other measures the error for non-final states. This is the first measure. This measure takes the average of the final prediction errors over all validation problems.

- **Smoothness of prediction:** This is the second measure. It takes the average prediction error for all states except the last one in a solution path over all validation problems.

We now reveal learning behavior using these measures. We performed an experiment slightly different from the previous experiment presented in section 4.1. Here are the differences.

- **Problem set:** We use the ART-2 problem set. This problem set contains four subsets of problems generated using all combinations of the following settings of two parameters: job due-time separations is small or large and resource capacity is small or large. A problem is loosely constrained if both job due-time

separation and resource capacity are large. On the other hand, it is overconstrained if both parameters are small. A problem with one large and one small setting of these two parameters is then moderately constrained. In general, a more constrained problem requires a bigger RDF for its final schedules.

Each subset contains 50 training and 50 test problems. These problems are randomly generated in the same way as ART-1 problems. ART-2 problems have the same resource configuration as ART-1 problems: two resource types and 4 pools. Each problem has either 3 or 4 jobs, so ART-2 contains only small problems. The key difference between an ART-2 problem and an ART-1 problem is that a job in ART-2, like a SSPP job, contains a milestone task, a set of pre-tasks that must be scheduled before the milestone task, and a set of post-tasks that must be scheduled after the milestone task. The number of tasks in a job ranges from 8 to 15. Therefore, an ART-2 problem combines aspects of both ART-1 and SSPP problems. More information about the ART-2 problem set is available in Appendix.

A main purpose of developing the ART-2 problem set is to test the generality of learning performance and the generality of learned knowledge. We want to make sure that learning is not only just working for a single domain, but also works generally for many different types of problems. Furthermore, we hope that by a study of several domains, we can find what was learned as domain-specific scheduling knowledge, specifically, find some type of knowledge that only works in the same domain as where it was learned but is not working well in other domains.

- **Training scheme:** Among 50 training problems, we choose 30 for training and 20 for validation. We employ the big-error-overtraining strategy. Learning rate α is determined automatically according to the current TD error. The exploration rate $\Delta\beta = 0.0005$. We trained eight different networks using all combinations of the following two λ settings at $\lambda = 0.2$ or 0.7 and four training-problem settings: the first 10 problems, the second 10 problems, the last 10

problems, or all 30 problems. We extend training to 5000 epochs (training sequences).

- **Validation:** We let the maximal number of repairs for a problem be 300. When search reaches this deadline, we record 300 as the number of repairs, but we do not count this trial in the average RDF measure and the measure of the error of final prediction. The length of a search path for ART-2 problems normally ranges between 20 and 80. A path exceeding 300 steps must have encountered a terrible number of loops and conducted a lot of backtracking. We often employ this as an additional measure in cross-validation test. We found that TD learning can successfully learn the value function to produce very short search—its search paths for all problems over all runs never reach the maximal bound. We also found that for a few problems in ART-2, iterative repair would have about 6% of chance to reach the deadline.

- **Features:** We employ 22 features. The features are exactly those presented in Section 4.2. Using 22 features improves the TD prediction slightly in comparison with using 16 features as in Section 4.1 for the ART-1 set.

Figure 4.13 shows the average training error for the four training problem settings with $\lambda = 0.2$. P30L2 used 30 training problems, and each of other three settings used a set of 10 training problems. The horizontal axis gives the number of training sequences processed. This shows that training for 10 problems could be faster than training for 30 problems. This also that the speed of training depends training problems. Two 10 problem settings, P10L2b and P10L2c, are easier to train; but for another 10 problem setting, P10L2a, training requires more time to achieve the same performance.

Figure 4.14 shows that the average final schedule RDF is improving with training. It is interesting to see that with an extended training epoch, the networks are still getting improving and finding better solutions. Figure 4.15 plots the average number of repair actions taken to produce a final schedule. This shows that with about 2000 epochs of training, the number of repairs is reduced to a very small magnitude. After that, the networks produce about the same number of repairs.

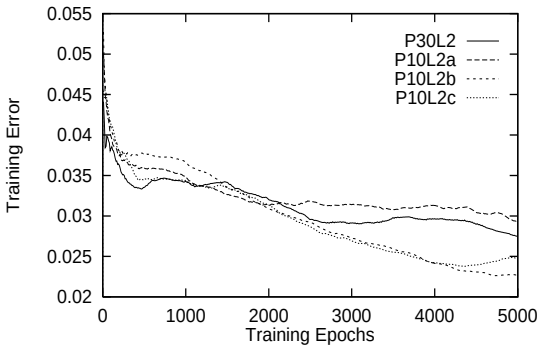


Figure 4.13. Training Error on 30 Training Problems.

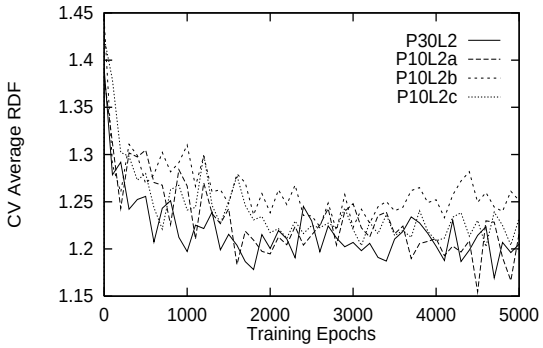


Figure 4.14. Average RDF over 20 Validation Problems.

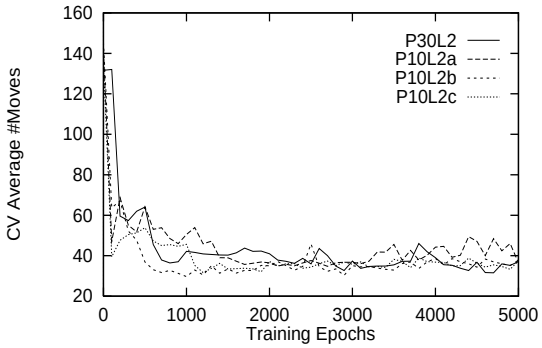


Figure 4.15. Average Number of Repairs over 20 Validation Problems.

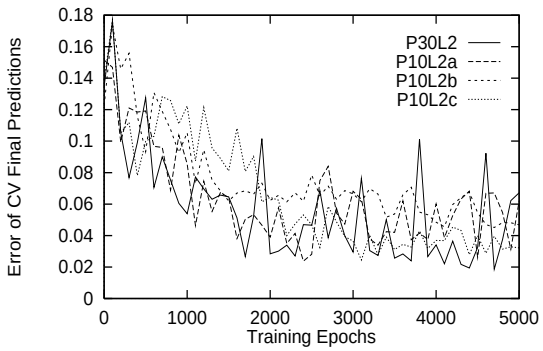


Figure 4.16. MSE of Final Predictions on 20 Validation Problems.

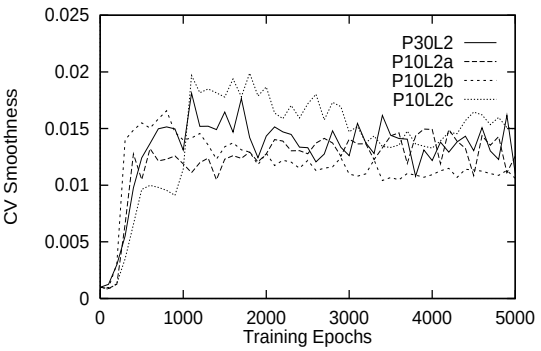


Figure 4.17. Smoothness of Predictions on 20 Validation Problems.

Figure 4.16 clearly shows that the accuracy of final predictions improves with training. We should mention that feature F20—RDF for the final state—has played a very important role in this performance improvement. Figure 4.17 plots the smoothness of predictions during training. Each network is very smooth in the beginning, because all the weights are very small and its predictions are nearly constant. So its smoothness is $|R(s, a, s')|$, which is 0.001. With training, the network begins to distinguish different states, and its smoothness error increases. After 1000 epochs of training, the network begins to slowly reduce this error.

Evidently, the learning behavior revealed by these measures shows that the value function is not completely learned. While the smoothness error (Figure 4.17) over the cross-validation set decreases during learning, it is still much larger than zero. Similarly, the final prediction error (Figure 4.16) is also still substantial, despite the changes we made to the training procedure and to the input features to try to reduce it.

Are these problems an indication of pathologies caused by the TD function approximation method? Or are they a consequence of inadequate input features? Based on the training curves, it seems that the TD function approximation method is not to blame. We know that our features are far from perfect. One piece of evidence supporting this is that when we added the two features that encode final RDF and signal the final state, our performance improved, both for final state RDF prediction accuracy and also for smoothness.

We now turn our attention to understand why the TD scheduler works for this problem. Certainly, a set of good features along with a proper network is essential to this success. A key question is: Is it possible that the use of function approximation alone is responsible for the superiority of TD over IR even ignoring the dynamic-programming component of our approach. If this were true, then it would not be necessary to use TD learning or dynamic programming to achieve this high scheduling performance. This could be true if the main reason for the superiority of TD in this problem is that it can smooth away local minima in the evaluation function. Briefly, the argument goes as follows. Iterative repair must use simulated annealing

because its evaluation function (number of constraint violations) suffers from local minima that must be overcome by the controlled application of randomness. We have confirmed this—pure greedy search using this evaluation function gives miserable results. If these local minima could somehow be smoothed away, then a one-step lookahead greedy algorithm could obtain substantially the same final schedules more quickly (even without using the "true" value function). TD(λ) combined with backpropagation can be viewed as a smoothing operation applied to the network's input features. These input features include 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.

The answer to this question is false. In the next section, we will present clear evidence for the answer, which shows that TD learning that performs dynamic programming is the key to our success. TD learning propagates final rewards backward and learns to predict the final RDF successfully, resulting in a very good scheduling policy.

4.4 Evidence for TD Learning

4.4.1 Learning Is Not Feature Copying

A simple possible explanation of the success of TD is that some input features—particularly current-state-RDF—provide a very good evaluation function and that the network is essentially just copying this input feature to its output. We have tested this in two ways. First, we ran a pure greedy algorithm using only current-state-RDF as the value function. This performs slightly worse than our TD scheduler. Second, we compared the current-state-RDF to the value function that predicts the final RDF. Figure 4.18 shows a comparison of these along a TD scheduling path. We can see that these are *very* different from each other. Current-state-RDF starts very low and increases rather erratically as the schedule is alternately lengthened and then improved. The predicted final RDF generally starts high and decreases fairly monotonically and smoothly toward the final state RDF. This shows that the

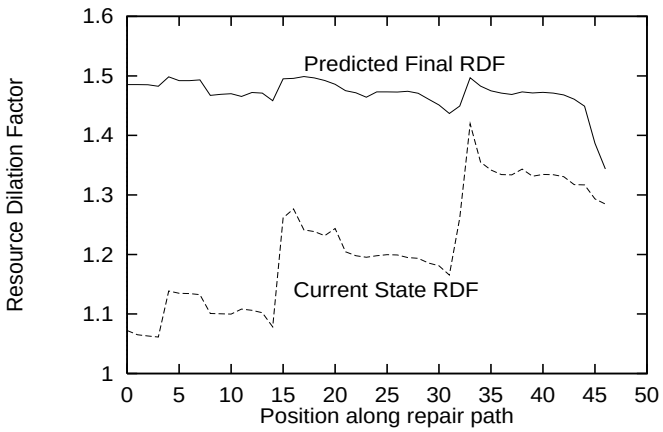


Figure 4.18. Compare the current-state-RDF to the learned value function. This shows how these two values change along a TD scheduling path.

value function has been learned to predict the final RDF. From these experiments, we conclude both that our input features are good and that the networks are doing substantially more than just copying these input features to the output.

4.4.2 Training with No Final Rewards

We now present evidence which shows that TD learning with function approximation works properly in propagating rewards backward to earlier states in a sequence. A simple experiment is to compare a network trained with no final reward to a network trained in the standard way with final reward. We train a network on a set of 10 problems and test the network on the same problems. Each network was trained with $\lambda = 0.2$. The test performs greedy search using the policy suggested by the learned network. The difference in performance of the two networks is clearly presented in Figure 4.19 and Figure 4.20.

Figure 4.19 compares the average RDF of the test solutions for the two networks. Both networks were trained with $\lambda = 0.2$. The figure shows that with training, the network trained without final reward is not able to capture useful scheduling strategies. Its average RDF is much larger than the average RDF produced by the network trained by the standard way. Figure 4.20 plots the number of repairs required

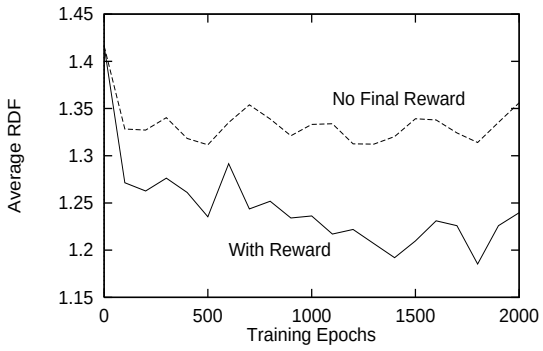


Figure 4.19. Effect of training with no final rewards shown with the average RDF.

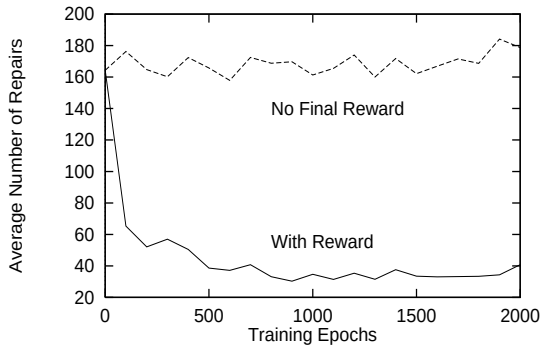


Figure 4.20. Effect of training with no final rewards shown with the average number of repairs.

to construct solutions on the training problems. It shows that the network trained without final reward also fails to learn to construct final solutions quickly. The number of repairs required to construct solutions by this network is much larger than the number of repairs needed by the standard network.

4.4.3 Training with the Last Section of A Sequence

Another experiment which demonstrates that TD learning performs properly in reward back-propagation is to compare a network trained only using the last section of a sequence to the network trained in the standard way. Likewise, we again train a network on a set of 10 problems and test the network on the same problems. The following figures shows the performance of a network trained only using the last six states in each sequence.

Figure 4.21 plots the average RDF tested on the training problems. It shows that with training, the network trained with the last section even finds better solutions than the standard network does. We tested on other sets of problems and also found this property. This tells us that a network only trained with the final section learns better for repairing a schedule in the final stage. However, when we perform cross-validation test (i.e., test on different problems), the difference in performance between

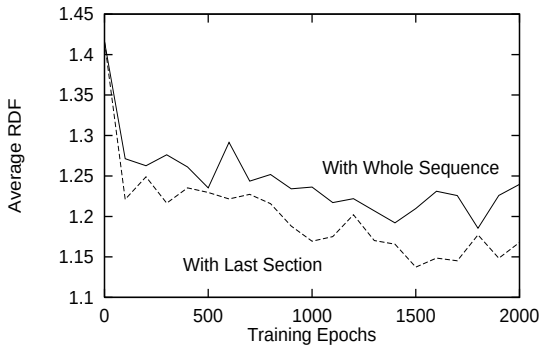


Figure 4.21. Effect of training with the last section shown with the average RDF.

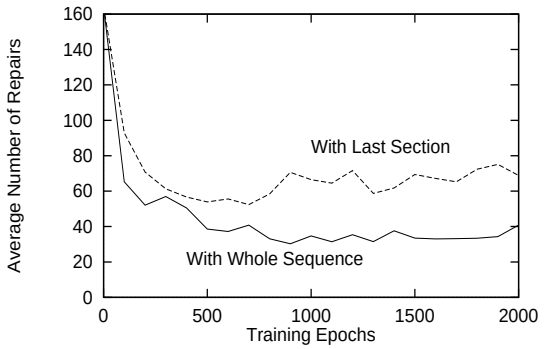


Figure 4.22. Effect of training with the last section shown with the average number of repairs.

these two networks becomes small. While over-training the last section helps quite significantly to find better solutions for the training problems, it is still not clear how much we can benefit from this in general.

Figure 4.22 plots the number of repairs required to construct solutions on the training problems. It shows that with training, the network trained with the last section is not able to learn to construct final solutions quickly. Because the network is trained with only a small portion of a whole sequence, it lacks knowledge about what is the shortest path. The search path generated by this network is much longer than the path generated by the standard network.

The experiment has clearly shown the impact of TD learning. More importantly, the result suggests that we could possibly further improve our scheduling system by training separate networks for repairs in different stages and using different networks in the corresponding stages during scheduling. This is an approach worth exploring in the future.

4.4.4 Learning Early Decisions

We know that state-space search becomes more difficult if it involves a larger space, a larger branching factor, and more cycles. We are now concerned about another

important factor that determines the difficulty of a state-space search problem, named the *importance of early decisions*. A problem would become much easier if its early decisions were not important. To solve this type of problem, we could simply apply backtracking search to find solutions quickly. Planning to solve this type of problem is also easy, because in all states we only need to worry about the near future. However, if early decisions are important, the time horizon considered by a search policy must be very large. The learning task for this type of problems must deal with *delayed rewards*—the rewards received in the distant future.

Note that when we emphasize early decisions, we do not mean to lessen the importance of decisions in the middle and late stages. For most complex real-world problems, we must perform actions correctly throughout the whole search process in order to achieve success.

It has been clear that early decisions are crucial in scheduling. [Fox and Zweben, 1995] states that in order to make good schedules, it is very important to make correct repairs in the early stages of scheduling. They points out that this has been the major reason that backtracking-based scheduling systems fail to construct schedules efficiently. Evidently, to make a good schedule, we need not only to perform correct repairs earlier but also to perform correctly later during scheduling.

Reinforcement learning is concerned with learning to act in an environment with delayed rewards. We now demonstrate results about how a learned TD policy performs in different stages during scheduling. We conduct two types of experiments: one is to insert a TD learned policy into the iterative repair scheduling system; and the other is to insert the IR scheduling policy into the TD scheduling system. The new policy replaces the policy in the base system for one quarter of a whole scheduling trajectory. To do this, before search we need to give an estimated length of the trajectory for a given problem for each of the two scheduling systems. We use the following crude estimate for trajectory length: the number of tasks in a problem n_{tsk} multiplied by 1.6 for TD and n_{tsk} multiplied by 2 for IR. The actual length of a final quarter could be longer or shorter than the estimated length, because the trajectory must terminate at a final schedule.

The performance of each scheduling procedure is presented in terms of the RDF of the constructed solutions on test problems. We use the 50 test problems from the ART-2 set. We select the 12 best networks from the networks trained by TD learning as described in Section 4.3 and construct a TD scheduling system using these networks. We test 10 procedures: Pure TD, Pure IR, 4 embedded TDs, and 4 embedded IRs. Pure TD runs 12 iterations, each using a unique network. The pure IR procedure runs 50 iterations. It runs an improved version of the IR algorithm, which uses the RDF as the cost function. Each of embedded TD procedures and embedded IR procedures runs 20 iterations. Each of these embedded systems uses the 10 best learned networks in its TD component, where each network is run twice. To make the two trials using a single network produce different solutions, we allow search to take the second best action in each step with some probability if values of the two best actions are almost the same.

Figure 4.23 plots that the average RDF of the solutions over all iterations and all test problems. The vertical error bars shows 95% confidence intervals based on t-test. TD1Q, TD2Q, TD3Q, and TD4Q denote four embedded IR scheduling procedures where the TD policy is used in the first, second, third, and final quarter of the IR procedure respectively. We see that TD2Q, TD3Q, and TD4Q can find solutions of close quality. They perform clearly better than Pure IR but worse than Pure TD. This not only demonstrates the importance of the early decisions but also shows that TD learning has captured a good policy for these decisions. Even though the IR random-search based policy is implemented in the second half of each trajectory, TD2Q still makes significant performance improvement over Pure IR. However, we can only see a very slight performance improvement with TD1Q. One possible explanation to this is that a bad policy employed in the last three quarter sections could almost completely remove the effect of a good policy implemented in the beginning quarter. If we apply a better policy in the last three quarter, then we can probably see some impact of TD policy. Another possible explanation is that probably TD has not succeeded in learning a good policy for the first quarter. This figure further shows that using a good policy later has a larger effect in improving performance than

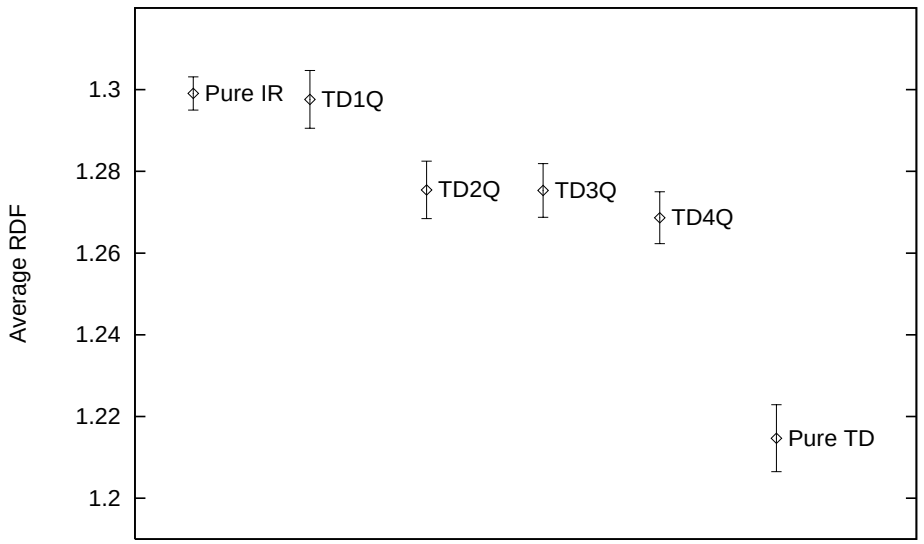


Figure 4.23. Effect of the TD policy inserted in different stages of the IR scheduling procedure.

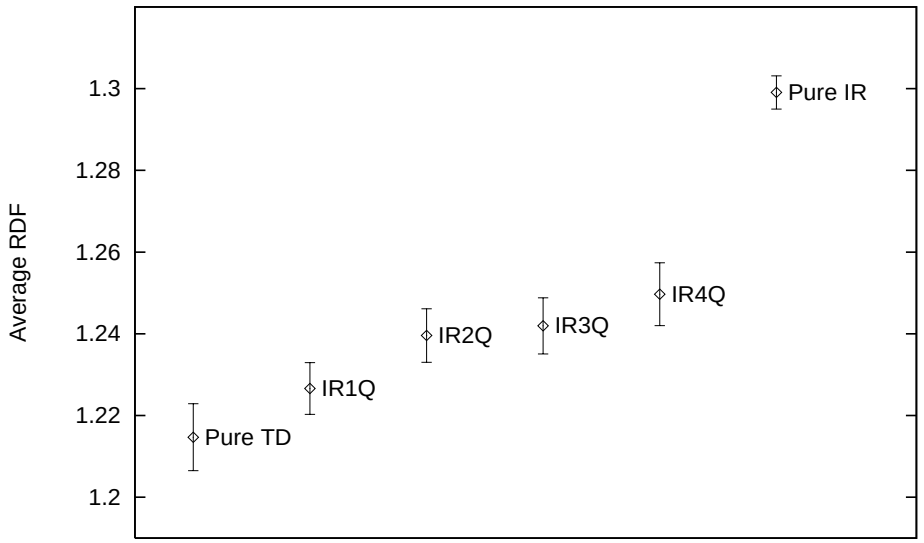


Figure 4.24. Effect of the IR policy inserted in different stages of the TD scheduling procedure.

using a good policy earlier. The performance of TD1Q, TD2Q, TD3Q, and TD4Q increases monotonically. This makes sense in practice, because the later a bad repair is conducted, the more unlikely the negative effect can be recovered.

Analogously, Figure 4.24 shows the performance of scheduling with the IR policy inserted in the TD scheduling system. We now evidently see that IR1Q and other embedded TD procedures perform worse than Pure TD. It is important to see the difference between IR1Q and Pure TD. This tells that a system that employs a good policy in the beginning quarter followed by the same good policy for later repairs performs better than a system that starts with a bad policy followed by the same good policy. This result further shows the importance of early decisions and the quality of the TD policy for early decisions. Bad repairs made in the beginning are not able to be recovered. A high-quality scheduling system must perform well in the early stage. For the four embedded TD procedures, likewise, we see that using a bad policy later can have more negative effect than using a bad policy earlier. In general, the effect of a bad action in a good system is more sensitive and easier to see than that in a bad system.

Note, however, that because of the random components of the procedures, these two figures hide considerable variation. Figure 4.25 reveals this variation. It plots the histogram of the RDF of all solutions for each method. We now look at the embedded TD procedures from IR1Q to IR4Q (the right column in the figure). The figure shows that if the IR policy is implemented in a later quarter, then the histogram shape leans more to the left. This tells that the average RDF measure is not very accurate, and the actual performance of these four embedded procedures is even closer. We believe that under an ideal system which executes an optimal policy, making a bad repair in different stages should result in equally bad effect.

For the embedded IR procedures, however, the histograms are roughly symmetric, so the mean RDF gives an accurate measure of performance.

We only present results on the solution RDF measure. Similar results have also been obtained for the length of scheduling paths, which show that the TD policy also makes good early decisions in finding a short path to good solutions.

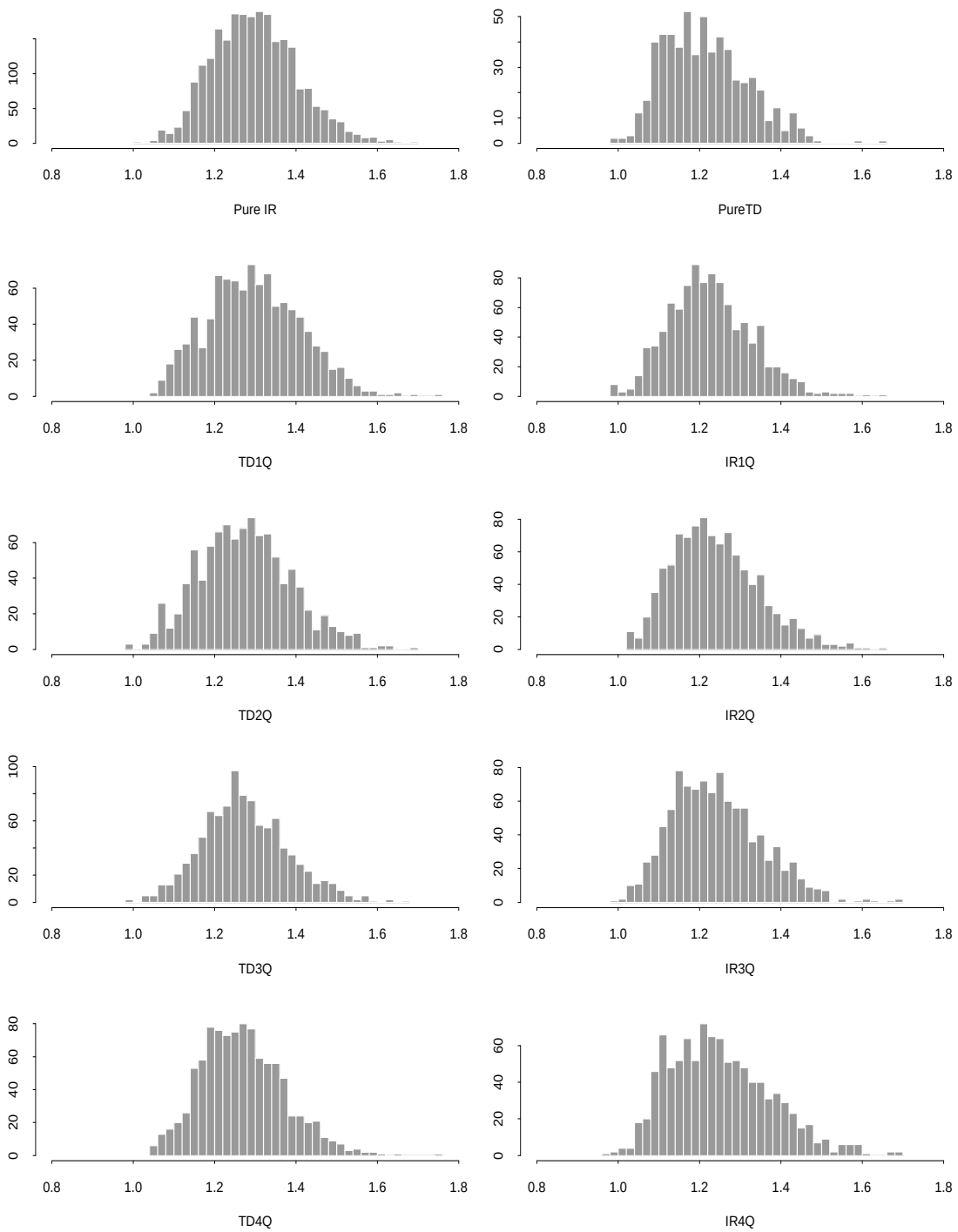


Figure 4.25. Histograms of the average RDF of the solutions produced by 10 different scheduling procedures.

4.5 How to Learn?

In the last section, we presented evidence for TD learning that shows why TD works in this problem. We now turn our attention to algorithm design: What is the best way to perform TD learning in order to achieve good learning performance?

4.5.1 Backward Learning and the λ Parameter

We now analyze learning behavior using different values of λ. Learning is implemented in backward order, which was found to learn better than forward learning. Figure 4.26 shows the performance of learning using the following λ parameter settings: 0, 0.2, 0.7, 0.9, 0.95, and 0.98. It also shows the performance of learning using actual outcomes. For each learning setting, it plots five evaluation measures as presented in Section 4.3 (from left to right): the training error, average solution RDF, average number of repairs, final prediction error, and smoothness. The last four measures are taken from cross-validation test. Training and validation exactly follow the procedure given in Section 4.3. Training is conducted on 30 problems and validation test is conducted on another 20 problems.

Let us look at the learning performance for each setting in detail. We see that learning with $0 \leq \lambda \leq 0.7$ performs very well. The training errors (the first column) are smoothly and gradually reduced. The learning curves for the average RDF and average number of repairs in cross-validation test (the second and third columns) are quite stable and improve with training. These two measures are most important, because they reflect the scheduling capability. None of other learning settings shown in the figure can perform well for both of the measures. We also see the reduction of final prediction error (the fourth column) and a slight improvement in smoothness (the fifth column) for $\lambda = 0, 0.2$, and 0.7. In general, we can not tell among $0 \leq \lambda \leq 0.7$ which λ setting is the best. Their resulting policies are comparable in terms of scheduling capability. Training with a bigger λ could learn slightly faster. Training with a small λ could result in a slightly smoother prediction for the on-policy trajectory (see the fifth measure).

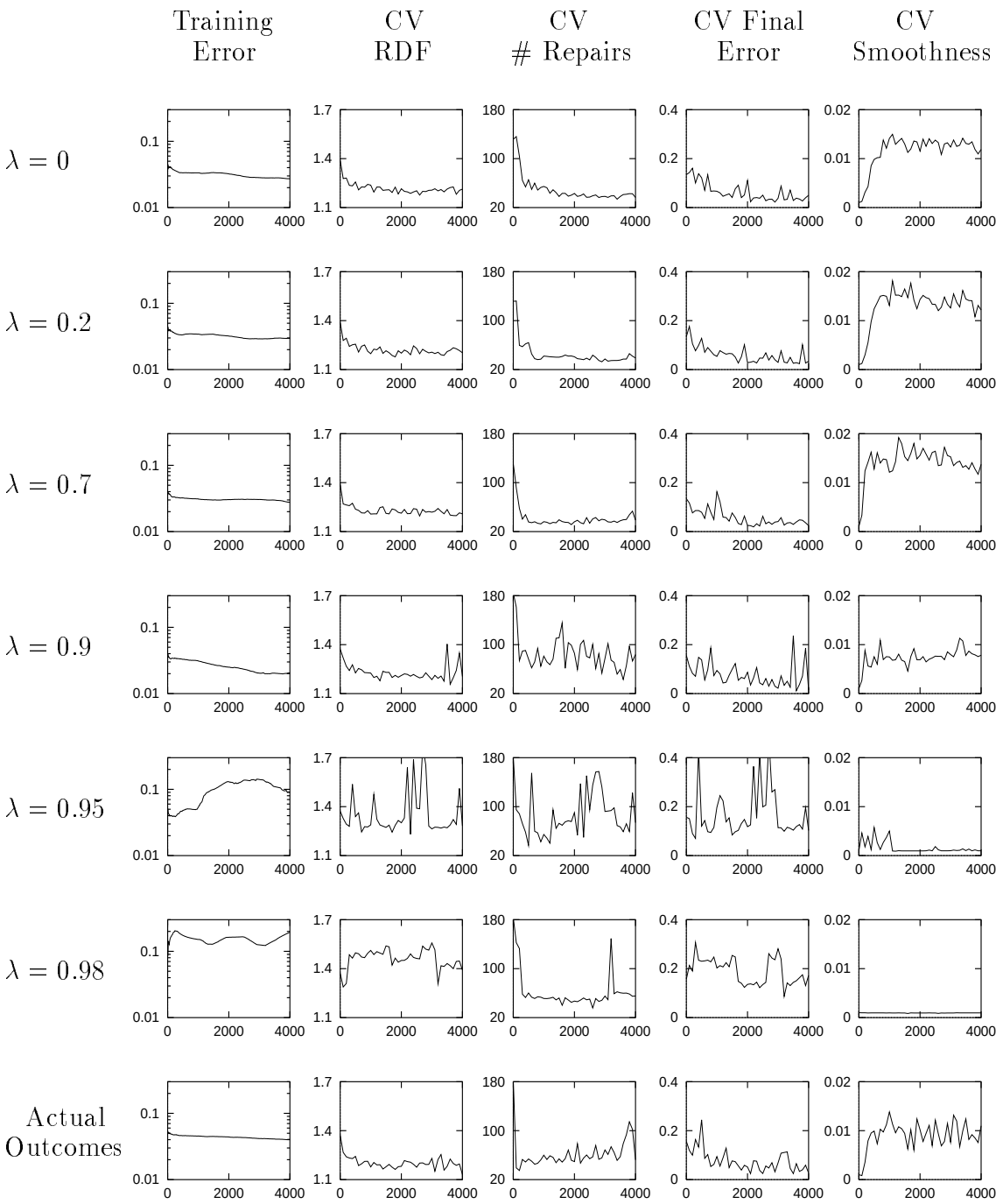


Figure 4.26. The effect of λ and learning with actual outcomes. For each learning setting, it plots the training error, average RDF, average number of repairs, the final prediction error, and the smoothness (from left to right). The horizontal axis is the training epoch processed.

When $\lambda = 0.9$, we start to see some trouble encountered by learning. The RDF measure becomes noisy during training. The number of repairs is very unstable and often becomes substantially large. The error in the final prediction is also unstable and can become very large during training. An interesting phenomenon is that the smoothness error for this situation becomes small. This appears to be a general phenomenon. Under the current schedule representation, we never found a value function that was good for scheduling that also had a very small smoothness error. On the other hand, a very bad value function which predicts the same value over all states has a very small smoothness error, which is 0.001. A small smoothness error very often could be a signal that learning has encountered trouble making good predictions.

When $\lambda = 0.95$ and 0.98, learning becomes messy. None of the five measures shows that it does the right thing. The training error is not reduced, which means that each weight-update step that over-emphasizes a future trace is completely useless. These two learning settings all have a very small smoothness error, however. Their value functions basically predict the same value over all states.

Finally, the figure shows the behavior of learning using actual outcomes. The actual outcome value for a state in a path is the cumulative reward received from the current state to the final state of the path. The learning procedure performs slightly differently from regular forward TD(1). It performs multiple updates for each pattern (state-pair), and weight update is conducted after each pattern rather than after each sequence. The learning behavior does not look too bad. Aside from the number of repairs measure, all other measures show very good performance. The training error is big because for each pattern actual outcome values are used (TD errors are usually smaller).

For a complete comparison, it is worthwhile to show the results for standard forward TD(λ). Figure 4.27 shows the results when $\lambda = 0.2, 0.9$, and 1. The implementation of the standard method differs from the backward implementation in that it calculates the weight update amount for each state-pair from the starting state to the final state under a fixed learning rate and performs weight update after each

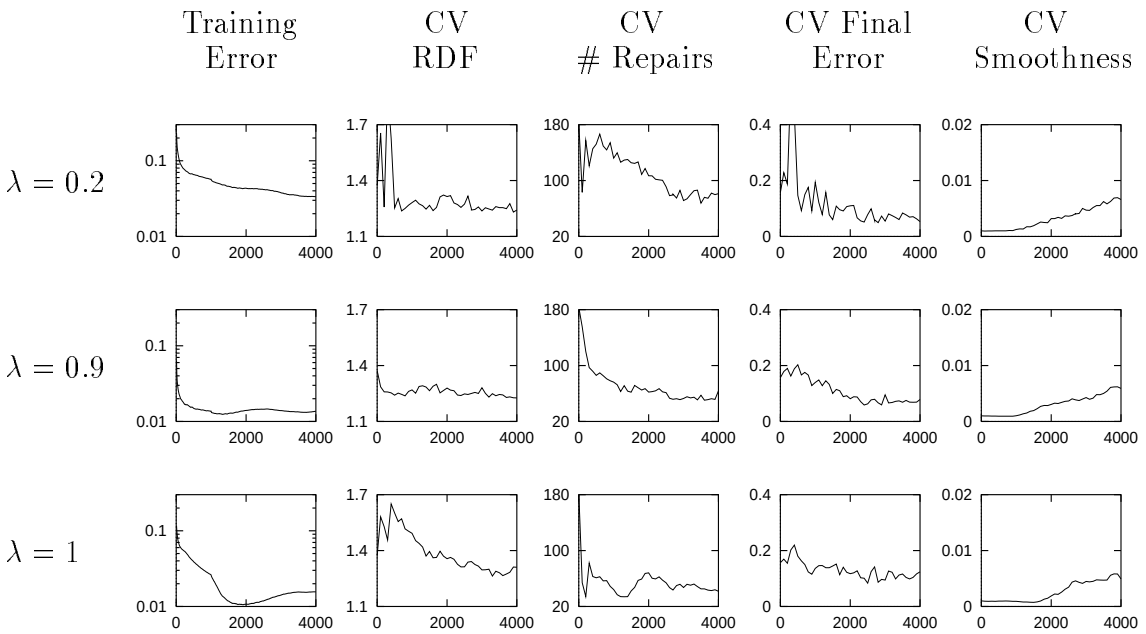


Figure 4.27. Forward learning under $\lambda = 0.2, 0.9$ and 1. For each learning setting, it plots the training error, average RDF, average number of repairs, the final prediction error, and the smoothness (from left to right). The horizontal axis is the training epoch processed.

sequence. (Both methods use the same exploration and best-experience replay.) For fast learning, we use the largest possible learning rate. This makes $\alpha = 0.2$ when $\lambda = 0.2$ and 0.9 and makes $\alpha = 0.1$ when $\lambda = 1$. We can clearly see that standard TD(λ) does not perform as well as backward TD(λ) in both learning accuracy and learning speed. This is not surprising, because the big rewards for scheduling are delayed, and forward learning does not learn much when it processes the early states of each sequence. Learning with big λ values is also not very helpful, because we use function approximation and abstract schedule representation. The figure further shows that with forward learning, using a big λ (0.9) is better than using a small λ (0.2), but $\lambda = 1$ is not good. The training error with forward learning would be small (see $\lambda = 0.9$ and 1) since it trains the early part of a sequence first, which mostly has a very small TD error.

We conclude that backward TD(λ) learns better and faster than learning with actual outcomes and standard forward TD(λ). In backward learning, λ must be set smaller than 0.9 in order to learn properly.

4.5.2 Other Strategies

We have presented the importance of backward learning and the λ parameter. The multiple weight update mechanism is always associated with backward learning, because this produces the best performance. We now turn our attention to other modifications to the TD(λ) algorithm and discuss their roles briefly.

Exploration: We used a very simple exploration strategy during learning. Exploration is necessary during learning. In general, it is not quite clear what is the best schedule for the exploration rate. However, to achieve convergence, learning must end in the low exploration stage. Therefore, a simple and good strategy is to determine exploration based on how long we are willing to train a network. If we plan to train with more epochs (trials), then the exploration rate is reduced slowly, otherwise, it is reduced more quickly. For learning with k epochs, we simply set

$$\Delta\beta = \frac{1}{k}.$$

After each learning epoch, we update exploration rate β as

$$\beta := \beta - \Delta\beta. \tag{4.56}$$

Experience replay: Best experience replay makes a difference in producing a better scheduling policy. We compared learning with this replay and learning without replay and found that the former clearly out-performs the latter in finding better solutions. However, based on our current features, we could not eventually produce solutions for training problems as good as the best experiences obtained under thousands of random trials. This is probably the result of the inability of our neural network and input features to represent the true value function. This means that if we allow iterative repair based on simulated annealing to run an extremely long time,

it will eventually find solutions better than those that can be produced by learned networks.

Loop avoidance: We employed the dynamic policy determination method to avoid loops being taken in each trial. This improves learning performance significantly. In our early work where we did not employ this mechanism, we found that learning was not only slow but also frequently converged to networks with poor performance. The loop avoidance mechanism also improve scheduling performance significantly. Scheduling with this mechanism finds better solutions much more quickly than a system that does not employ it.

Random sample greedy search: The branching factor for SSPP problems is typically about 20. However, it is only about 6 for ART-1 and ART-2 problems. Random sample greedy search is only actually employed for SSPP problems. For this set of problems, this mechanism can reduce search time quite significantly, accordingly, it speeds up the learning process. Scheduling using the mechanism is also more efficient.

Teaching: Our teaching method is very simple and used only during the early stage of training. By teaching, we train a network using the sequences generated by a good scheduler. We mostly employed greedy search using the RDF of the current schedule as the heuristic function as this scheduler. This speeds up the learning process, especially in the early stage. However, we did not always employ this mechanism. In general, learning can be done successfully without teaching.

Learning rate: Another mechanism that we did not always use is the automatic learning rate mechanism. In general, this mechanism performs quite well. But in most situations, we used a fixed learning rate and also achieved success.

4.6 Generality

This section discusses our current understanding about the generality of learning performance and the generality of learned knowledge. Two types of questions are considered: (1) Is it possible to apply the same TD learning approach to many dif-

ferent scheduling domains and achieve similar success? (2) How general is learned scheduling knowledge? Did learning capture some domain-specific scheduling knowledge that can only be applied in certain domains? What kinds of scheduling strategies have been learned?

4.6.1 Generality of Learning Performance

We believe that the TD learning approach should generally work very well for many different scheduling domains. We have experimented with this approach on several problem sets, particularly, on ART-1, four subsets of ART-2, and SSPP, and we found that it worked well in all cases.

From the ART-1 experiment, we learned that we can train a neural network with a set of small problems and use the learned network to solve both small and large problems. This means that there must be some regularity that is shared by both small and large problems. Training with small problems is much faster than training with large problems. The length of a trajectory for small problems is shorter, which makes it easier for TD learning to propagate final rewards backwards to the starting state. Training with very long trajectories not only is inefficient but may sometimes cause troubles, especially when a big λ is used. From the ART-2 experiment, we learned that TD works better than IR for problems with different levels of constraint: loosely constrained, moderately constrained, and overconstrained. The performance curves look very similar with different subsets of problems. From the SSPP experiment, we learned that TD can be applied well in real problems. Real problems are large, but we learn a scheduling policy from smaller problems. Therefore, training a network does not take much time (about 3 days on a Sun SPARCstation-10). The performance test on large-scale real problems shows that TD maintains a constant factor advantage over iterative repair.

4.6.2 Generality of Learned Knowledge

As we mentioned earlier, one purpose of constructing the ART-2 problems was to test the generality of the learned knowledge. We wanted to know whether the learned scheduling strategies are general or domain-specific. We believe that there must be some domain-specific knowledge learned to make the resulting scheduling system work efficiently. However, what is the learned knowledge? What are key factors for scheduling problems that might require different scheduling strategies?

One experiment to investigate this was to cross-test among all four subsets of ART-2. Recall that the four ART-2 subsets have different due-time separations and different resource capacities. So, this test is trying to find potential domain-specific knowledge related to these two types of constraints. The experiment is described as follows. First, for each subset of problems, we trained neural networks on training problems using our TD learning procedure and then constructed a TD scheduler using the best-learned networks. After this, we conducted cross-test which used each TD scheduler to solve the test problems in all four subsets of problems. If TD is learning domain-specific knowledge, we would expect to see better performance when the network is tested on problems from the same ART-2 subset. In contrast, when the learned scheduler is tested on different ART-2 subsets, we would expect to see worse performance.

The result of the experiment, however, is negative. The test did not show a significant difference existed between each of the four TD schedulers. For each of the four problem sets, all TD schedulers perform very similarly. They produce solutions of about the same quality and at about the speed. One explanation for this result is that the ART-2 problem set is probably not suited for this test. Under the problem configuration of ART-2, there might not be domain-specific aspects that require using different scheduling strategies.

We conducted another experiment to test for domain-specific knowledge as follows. We tested on the scheduling problems generated according to the mission component method [Zweben *et al.*, 1992]. By this method, a mission (job) is composed

of tasks and constraints from a generic flow with various "options". A "suite" of scheduling problems is considered to be a domain. For each suite, the generic flow, the set of options, and resource capacities are fixed. We tested whether TD could learn different strategies for different suites of problems. Again, we did not find any significant difference between the scheduling systems learned from different domains.

We feel that TD has probably only learned quite general scheduling strategies. Therefore, it is difficult to find what is learned as domain-specific scheduling knowledge. Our schedule features basically encode information regarding the violations on different resources and factors that affect the schedule length (such as slacks) in a schedule. These features are very general. A learned neural network must reflect a combination of these features. From such features, a network could easily pick up several simple general scheduling strategies such as "choose a repair that resolves more violations", "choose a repair that will not lengthen a schedule too much", and "perform pool-reassignment first if it is available". Test verified this. However, different networks can perform quite differently in combining these strategies. The behavior of a network is too subtle to figure out exactly how it works.

4.7 Summary

A number of results have been presented for TD learning for job-shop scheduling. To view these results more clearly, we now summarize them organized in the order according to the context of features, the network architecture, learning, and scheduling, rather than in the order in which they were presented.

Features:

- Features are critical to the correctness of prediction. The current features primarily describe the following information about a schedule: resource utilizations, slacks, the RDF, various violation counts, and features discriminating different scheduling problems.

- We generated a sample described by the features and the value function. We showed scatter plots and sequence plots for features. The results show that none of the features alone has high correlation with the value function.

- We presented test results for our features using linear regression, regression trees, and neural network models. The results showed that each feature has some capability to predict the value function. Among them, the RDF probably is the most useful feature.

Network Architecture:

- Overlapping-Gaussian-range output units improved learning performance significantly.

- A relatively small network with 40 hidden units and 8 output units is capable of coming up with good predictions and making scheduling efficient.

Learning:

- TD(λ) with modifications works well in predicting the value function.

- Five measures were found to be useful to assess learning behavior: the training error, average solution RDF, average number of repairs, final prediction error, and smoothness. The last four measures were taken from a validation test.

- TD learning works properly in reward back-propagation.

- TD learning learned knowledge for making early decisions.

- Backward TD(λ) is important for the success and the speed of learning. The λ parameter must be smaller than 0.9 in order to learn a scheduling policy properly.

- Exploration, best experience replay, and training with loop-free sequences are also important for improving learning performance.

- TD learning has been tested in several problem sets. They all worked very well.

- The learned scheduling strategies probably are quite general.

Scheduling:

- The TD scheduling system performs better than the best known existing algorithm for this scheduling task—iterative repair.
- The TD scheduling system has strong scale-up capability. A neural network trained on a set of small problems works well in solving larger problems.
- Both loop avoidance and random sample greedy search mechanisms are important for improving the efficiency of the TD scheduling system.

Chapter 5
Learning with A Time-Delay TD(λ) Network

A shortcoming of the work presented in Chapter 4 is its reliance on hand-engineered input features. In this chapter, we show that TD(λ) can be combined with a time-delay neural network (TDNN) architecture to avoid feature hand-engineering. In this chapter, we first show how to extend the time-delay neural network architecture to extract features from irregular-length schedules. Following this, we present experiments on a set of simulated problems and discuss the results. These results show that TDNN-TD(λ) using low level features can not only match the performance of the hand-engineered features—it can actually perform slightly better.

5.1 Schedule Representation

The main challenge for designing a schedule representation is that virtually all methods for learning evaluation functions can only be applied to fixed-length vectors of features. However, the length of schedules varies depending on the number of tasks and the complexity of their temporal and resource constraints. In our previous work, we hand-engineered a fixed set of features that summarized the structure of the schedule. We included such features as the RDF of the current schedule, the mean and standard deviation of the unused resource capacity of each resource pool, the mean and standard deviation of the slack times, and so on. However, hand-engineering increases the cost of creating a new application and reduces the autonomy of the learning system. Therefore, we wish to develop a method that can automatically learn good input features.

The time-delay neural network [Lang *et al.*, 1990, LeCun *et al.*, 1989] has proved to be very effective in learning good position-independent features in visual- and

speech-recognition tasks. In speech recognition, it is applied to convert an input sequence of speech frames into a "hidden sequence" of extracted features. A classifier then walks along this hidden sequence and classifies each position in the sequence. The mapping from a particular position in the input sequence to a particular position in the hidden sequence is performed by a "kernel" neural network, which is scanned along the input sequence. It examines a sliding window of adjacent positions in the sequence. The kernel network has a single set of weights that is applied at all positions in the sequence, although each position may have its own bias weight.

To apply this architecture to scheduling, we must solve two problems. First, we must define what a "position" means in the schedule. Second, we must decide how to use the "hidden sequence" of computed features.

To define "positions", we subdivide the schedule into a sequence of blocks. Each block is defined to be a maximal time interval in which the current tasks and resource assignments do not change. For each block, we compute primitive features such as the number of resource units available in each pool and whether each pool is over-allocated. We also say that a task is "inside" a block if that task starts at the beginning of the block. Given this definition, we compute additional primitive features of the tasks inside the block: minimum and average slack time, number of dependents, and number of tasks inside the block. If there are multiple tasks inside a block, we compute the average values of these features.

We employ the ART-2 problem set as the testbed. These problems are concerned with 4 resource pools. A total of 12 primitive features are computed for each block: two features for each pool (the number of resource units available and whether it is over-allocated) and four features about tasks inside the block as described.

5.2 Network Architecture

We now turn our attention to network design. We define five kernels, each of which examines a single "current block" and computes a hidden "feature" from this block. These five kernels are scanned along the entire schedule, and they create a derived

sequence containing five "hidden features" for each block. How should this derived sequence be processed to compute the estimated value function?

One approach would be to scan a network along the hidden sequence and take the maximum value output by that network. However, because our goal is to estimate the RDF of the final schedule, it seems wiser to view the kernels as learning to recognize "bad conflicts" and "opportunities". By summing (or averaging) the "hidden features", the network can effectively count the number of opportunities and conflicts and estimate how much more the RDF will change before the conflicts are eliminated. Hence, we take the following approach.

The sequence of "hidden features" is split into thirds. Each third uses it own bias weight. Within each third, five features are computed by finding the mean value of each of the five hidden features over the blocks in this third. This gives a total of 15 features. These features, along with two other "global" features—the RDF of the current schedule and the resource utilization of the starting schedule—are input to a network having 40 hidden units. The output of that network is taken as the estimated value for the current schedule.

Figure 5.1 shows the architecture of the time-delay neural network. To recap, the network has three hidden layers named H1, H2, and H3. Layer H1 has 5 kernels (each with 12 weights). There are 15 biases, one for each feature in each third of the schedule. So H1 has a total of 75 parameters. H2 has 17 units; 15 of these are hidden units (3 sets of 5) averaged from H1 and 2 of these are the two global input features. H2 has no adjustable parameters. H3 has 40 hidden units fully connected to H2, for a total of 720 parameters. The output layer has 8 units fully connected to H3 and encoding the predicted RDF using the technique of overlapping Gaussian ranges [Pomerleau, 1991]. The output layer has 328 $(= 8 \cdot (40 + 1))$ parameters. Therefore, this net has a total of 1123 parameters. All units in H1, H3, and the output layer use sigmoidal transfer functions.

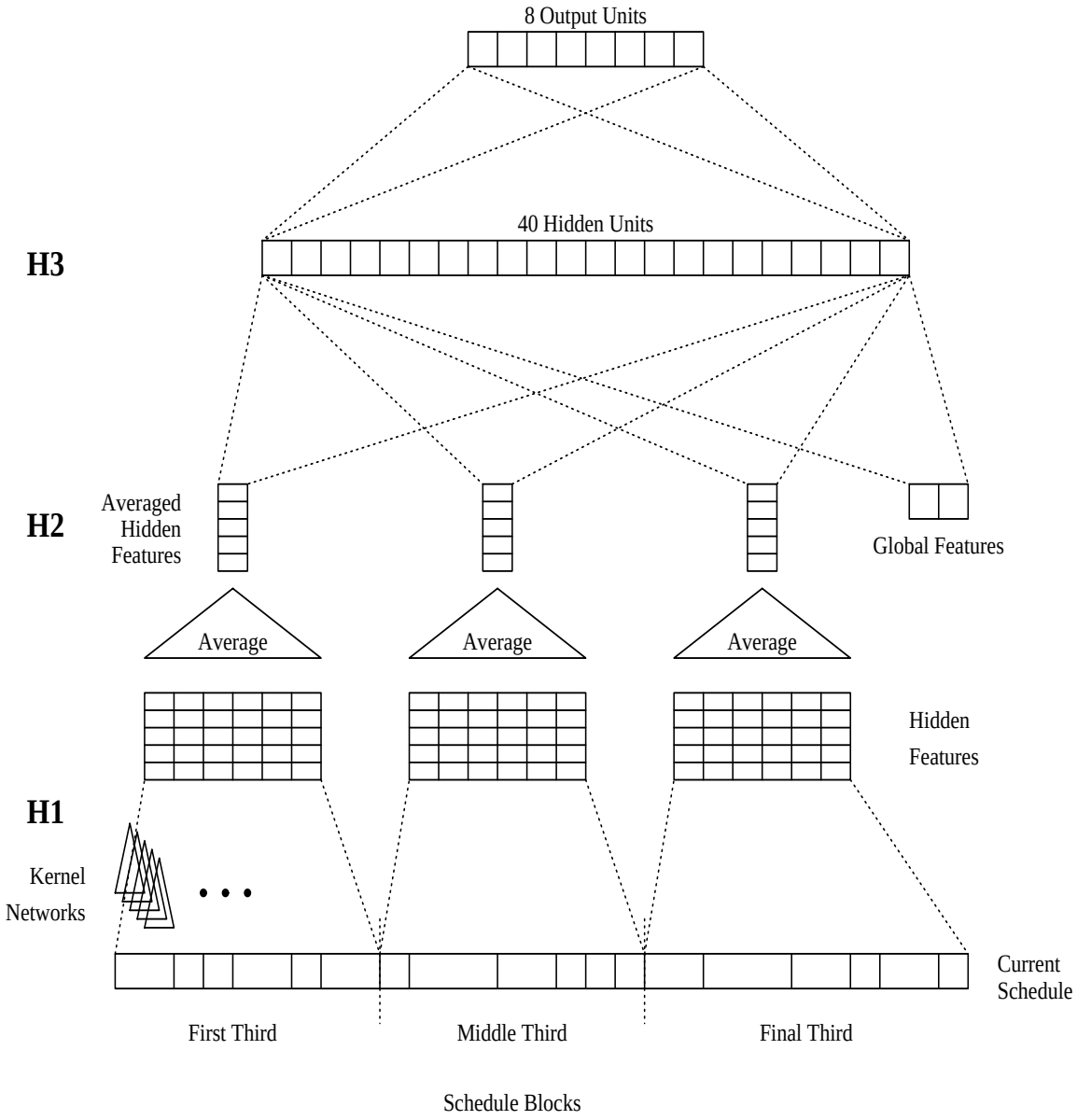


Figure 5.1. Time-Delay Neural Network Architecture.

5.3 Experimental Results

The experiment employs the ART-2 problem set. We followed roughly the same training procedure as presented in Section 4.3. In a set of 100 scheduling problems, we used 50 problems for final testing, 20 problems for validation testing, and three training sets of 10 problems each. An ensemble of 6 networks was constructed by training a separate network for each of these three training sets and for $\lambda = 0.2$ and $\lambda = 0.7$. Training was monitored by testing on the validation set every 100 epochs. To train a good TDNN, we normally need to take about 10,000 trials. Therefore, we employed a smaller $\Delta\beta$, $\Delta\beta = 0.0002$. Training was halted when the validation test showed no further change. For each of the 6 networks, the final set of weights and the set of weights giving the best validation score were retained for a total of 12 networks. The resulting scheduling system performs one-step lookahead search using the learned networks. It also employs the loop avoidance mechanism as described earlier.

We compared three different scheduling methods: TDNN-TD(λ) using low level features, TD(λ) using hand-engineered features, and Zweben's iterative repair method IR. G1TDN and G12TDN are two versions of the TDNN-TD(λ) scheduling procedure. G1TDN employs the best single TDNN network (as determined by validation set performance). To make G1TDN run more iterations based on the single network, we introduce some randomness into the policy determination procedure. G1TDN is allowed to choose the second best action with some probability according to the difference between the prediction of the best two actions. Let $\hat{f}_1$ and $\hat{f}_2$ are the estimates for the best and second-best repairs (which means that $\hat{f}_1 \leq \hat{f}_2$), we let the probability of choosing the best repair be $p = 0.5\left(1 + 1000(\hat{f}_2 - \hat{f}_1)\right)$ (p is set to 1 if this number exceeds 1) and probability of choosing the second-best be $1 - p$. G1TDN was run 10 iterations. Because of the randomness introduced in choosing actions, these 10 repetitions usually generate different schedules. G12TDN was run 12 iterations, using each of the 12 learned networks once. Analogously, G1N and G12N are two versions of the TD(λ) scheduling procedure using hand-engineered

features. G1N was run 10 iterations using the best single network trained using our hand-engineering features and G12N was run 12 iterations using 12 hand-engineered networks (each once). We also ran two versions of IR methods: IR-V and IR-RDF, using the number of violations and the RDF respectively as the cost function to be minimized via simulated annealing. IR-V and IR-RDF were each run 50 iterations. All methods were tested on 50 problems.

Figure 5.2 compares the test set performance of these six different scheduling configurations. It shows for each configuration the mean RDF of the solutions obtained over all iterations. From this we can see that G1TDN produces schedules with better average RDF than any of the other methods. In particular the F test shows that it is significantly better than all other algorithms except G12TDN. We can also see that IR-RDF is a big improvement over IR-V in finding better final schedules.

Figure 5.3 compares the mean number of repairs required by each algorithm to find a solution. For the IR algorithms only repairs accepted by simulated annealing were counted. This shows that the neural network methods have learned very good evaluation functions—they are able to find a good solution much more directly than the simulated annealing methods. According to the F tests, the networks with engineered features are slightly better than the TDNN networks, but all of the networks are significantly better than simulated annealing. From this we can also see that IR-V reaches a final schedule with many fewer steps than IR-RDF.

A shortcoming of these figures, however, is that they only record the mean results of *single runs*. Better results are typically obtained from simulated annealing if the algorithm is run many times and the best solution retained. Figures 5.4 and 5.5 show the mean RDF of the best schedule as a function of the number of schedule repairs and the amount of CPU time expended. Each time a scheduler finds an improved solution to a problem, the mean RDF value is updated. A point is plotted on the graph when an improvement reaches 0.01. G1N and G1TDN are not shown in these two figures. They perform very close to G12N and G12TDN, respectively, under a limited time (with 10 iterations).

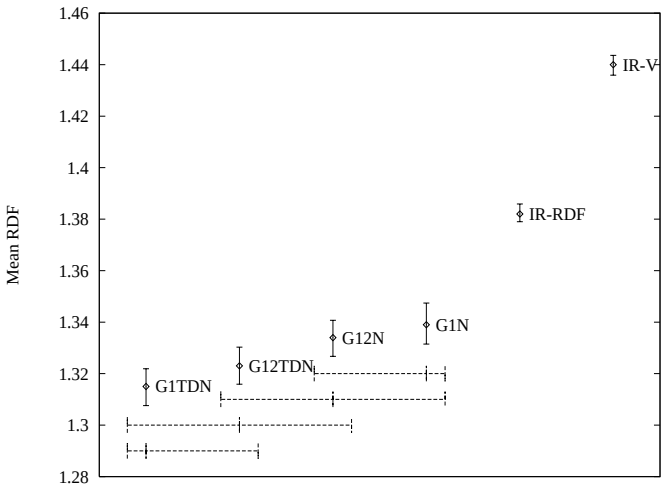


Figure 5.2. Mean RDF values. Vertical error bars show 95% confidence intervals based on t-test. Horizontal bars group together algorithms that cannot be distinguished based on ANOVA F-tests at $p < 0.05$.

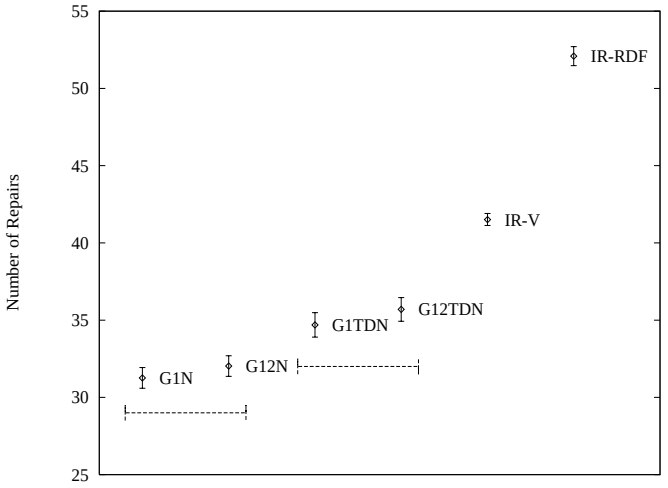


Figure 5.3. Mean number of repairs. Vertical error bars show 95% confidence intervals based on t-test. Horizontal bars group together algorithms that cannot be distinguished based on ANOVA F-tests at $p < 0.05$.

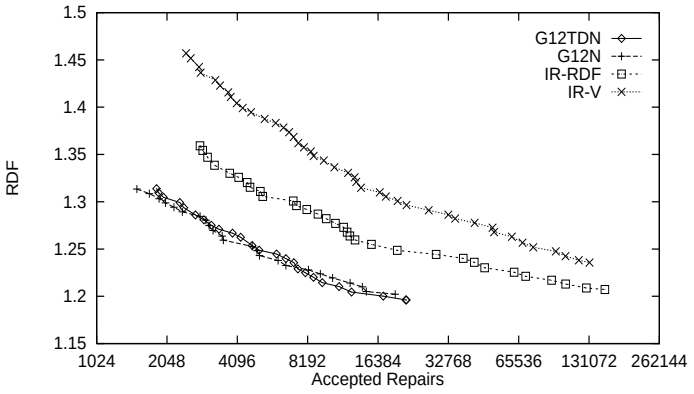


Figure 5.4. Comparison of Accepted Schedule Repairs.

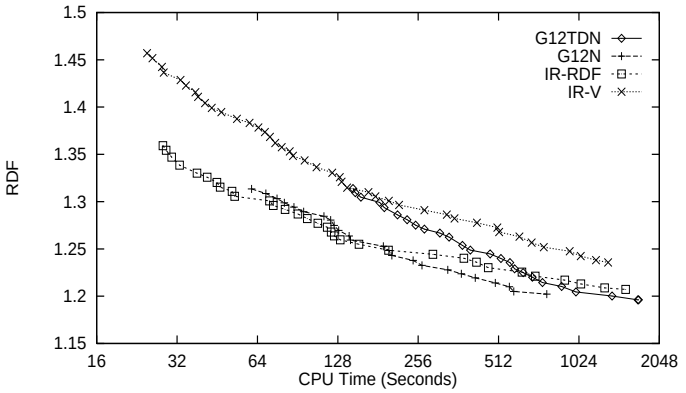


Figure 5.5. Comparison in CPU time.

The graphs show that the learned networks clearly out-perform Zweben's IR algorithms. Figure 5.4 shows that the networks perform many fewer repairs to find schedules of the same quality as the IR algorithms. Note that the horizontal axis is plotted on a log scale—the networks maintain a constant *factor* advantage over IR. For a schedule of a year's duration, this improvement would translate into several days (and hundreds of thousands of dollars) saved.

Ultimately, G12TDN does slightly better than G12N. After 12 iterations and 21,676 repairs, G12TDN produced solutions with an average RDF of 1.196. By comparison, G12N performs 19,406 repairs and produces an average RDF of 1.202.

Figure 5.5 illustrates a problem with the time-delay neural network approach: the networks spend more CPU time selecting each repair. This reduces the differences between the methods. G12N exhibits the best tradeoff between CPU time and schedule quality, although G12TDN attains the best final schedule quality.

The major CPU cost of G12TDN is the cost of breaking the schedule into blocks and convolving the kernel networks with the blocks. There are many opportunities to make this more efficient by taking advantage of the fact that each repair changes only parts of the schedule, and therefore, only parts of the neural network calculation need to be updated. We did not implement these improvements, however.

5.4 Discussion and Summary

TDNN-TD(λ) possesses many of the nice characteristics possessed by the hand-engineered system. The learning settings that work for the hand-engineered system (such as backward learning and experience replay) also perform well for TDNN-TD(λ). The learning curves for the training error and the four measures of cross-validation test are very good. We can learn a TDNN from small problems and then use the learned network as the evaluation function to solve large problems, which also produces high-quality solutions. Furthermore, TDNN-TD(λ) has been tested on many problems, particularly problems in ART-1, ART-2, and SSPP. We found that it generally works very well.

There are several important practical issues that are particularly related to the TDNN-TD(λ) approach. We discuss these as follows.

Learning behavior: Training a time-delay neural network takes much longer than training a regular network using hand-engineering features. While for ART-2 problems we can successfully train a regular network with 5000 trials, to train a TDNN, we normally need to take about 10,000 trials. Furthermore, it takes more time for a TDNN to learn from each pattern. During learning, a kernel must scan through all blocks in a schedule and update weights according to each of these blocks. For ART-2 problems, training a TDNN takes about 8 times as much time as training

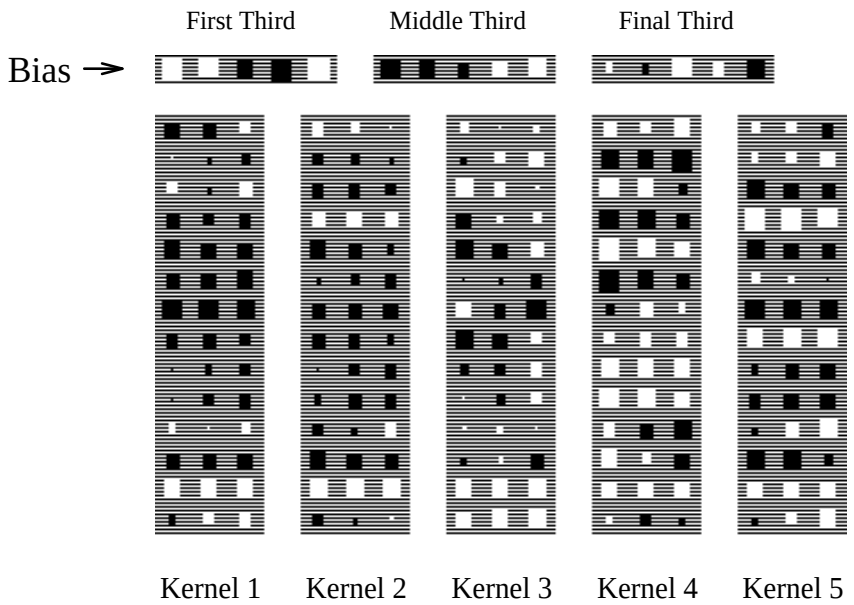


Figure 5.6. Five kernel networks.

a regular net. This becomes even worse (about 12 times) on SSPP problems, because a schedule of a SSPP problem involves many more blocks.

Detected features: Based on the observed scheduling performance, a learned TDNN must have detected very useful schedule features. The detected features perform at least as well as those hand-engineered features. We now discuss our current understanding on the detected features briefly. First, like manual features, the TDNN-learned features also resulted in quite general scheduling strategies such as "choose a repair to resolve more violations", "choose a repair that will not lengthen a schedule too much", and "perform pool-reassignment first if it is available". This was observed by watching how a TDNN chooses repair actions for schedules. Second, rather than using statistics over a whole schedule (such as the hand-crafted features mean of resource pool free-capacity and mean of slacks), the learned features computed a finer grain statistics. A TDNN can learn different statistics over three different sections of a schedule. We found that for each third of a schedule, the network usually learned quite different features.

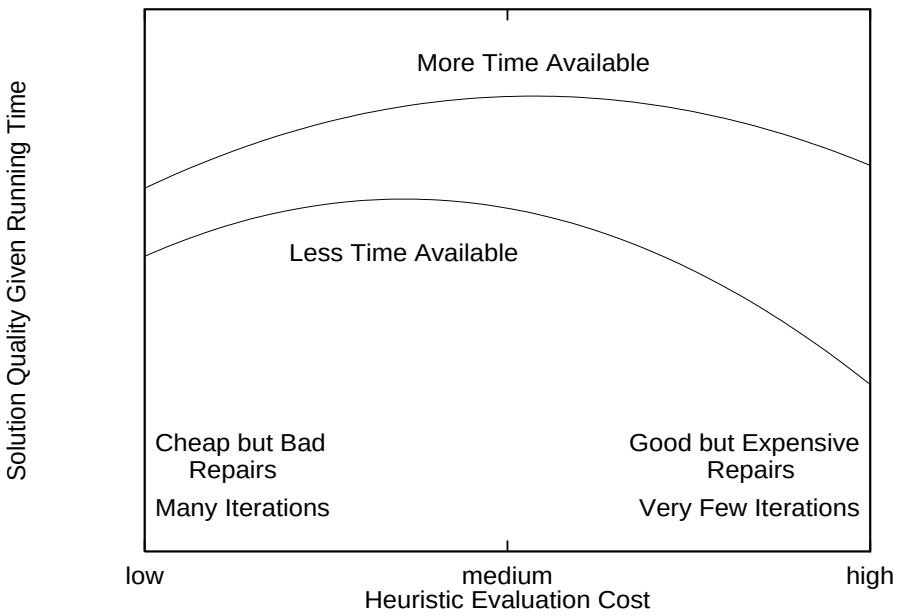


Figure 5.7. The utility problem.

Figure 5.6 shows the weights of five learned kernel networks. The kernels presented here are slightly different from the one presented in Figure 5.1 (This network improved the learned policy slightly). A kernel takes three adjacent blocks as input, where in each block 14 features are used. Two more features are introduced in describing tasks in a block: the average distance of move required in order to resolve the violation created by the tasks (the 13th feature) and the average task duration over all tasks in the block (the 14th feature). Therefore, a kernel uses a total of 42 low-level input features. In the figure, each column of a kernel presents the weights for a block. The bias weights are presented in the top of the graph. We can see that each third has learned very different bias weights. (A black box represents a positive weight and a white represents a negative weight; the size gives the magnitude.) For example, the first third employs big biases on almost all kernels, but the final third employs big biases only on `Kernel 3` (for negative) and `Kernel 5` (for positive).

The utility problem: A key problem for the TDNN-TD(λ) approach is the overhead of network evaluation during scheduling. Although a TDNN can be learned to provide a very effective scheduling policy, the overall running performance of this

scheduler may not be very attractive (see Figure 5.5). At low CPU costs, TDNNs are not able to produce solutions as well as other methods. This problem is referred to as the *utility problem*. We found that the heuristic evaluation cost is essential in determining the speed of a scheduler. This is illustrated in Figure 5.7. At the left side of the graph, we adopt very small neural networks or simple heuristics (such as the ones used in iterative repair) which are cheap to evaluate but of low quality. A scheduler using these kinds of heuristics must rely on some random mechanism (such as simulated annealing) and repeated restarts to improve solutions. In contrast, at the right side of the graph, we use more informed heuristics which require expensive computation but in general select a better repair at each step. These heuristics could be a very big neural network, a network which requires much local computation such as a TDNN, or a many-step lookahead with a relatively cheap heuristic. A scheduler using more informed heuristics does not need many iterations. We found that the best scheduler typically results from heuristics of intermediate cost. However, this depends on how much time we are allowed to prepare a solution. If we have less time, we would tend to choose heuristics with a lower evaluation cost, otherwise, we would tend to choose relatively more informed heuristics.

This chapter showed the results of using TD(λ) combined with a time-delay neural network architecture to avoid feature hand-engineering. We presented a modification of the TDNN architecture appropriate for scheduling problems. The combined TDNN-TD(λ) architecture can learn very powerful search heuristics that significantly out-perform all previous algorithms in terms of the quality of the resulting schedules. The TDNN architecture achieves this high performance with much less "feature-engineering" than our previous neural network approach. This demonstrates once again the superior ability of neural networks to learn useful higher-level features from raw input features.

Both of our neural-net-based methods demonstrate that the impressive performance of Tesauro's TD-gammon system can carry over to an important industrial application. Temporal difference learning is able to learn a very effective evaluation

function for job shop scheduling. Using this function, a scheduler can find better schedules and find them in fewer search steps than the best previous methods.

Chapter 6
Conclusions

We began this thesis by introducing state-of-the-art of technologies for the scheduling problem. After discussing the problems with these technologies, we introduced a novel approach to job-shop scheduling—reinforcement learning. Reinforcement learning is applied to learn a scheduling policy to speed up the scheduling process. We reviewed the mathematical foundations of reinforcement learning—the Markov decision theory—and discussed standard dynamic programming methods to solve Markov decision problems. We then discussed the reinforcement learning methods developed for solving large-scale problems, which learn partial policies and perform generalization. We presented the methods to formulate reinforcement learning problems for various types of state-space search tasks. We drew particular attention to the infinite loop problem in state-space search and introduced the dynamic policy to solve this problem.

Following this, we came to the major topic of the thesis. We introduced a very challenging learning task to reinforcement learning—learning to schedule for a NASA scheduling problem. We are aimed at acquiring a high-quality scheduling policy for this domain. The key challenges include (1) handling extremely an large state space, (2) finding a single policy for a *set* of problems, and (3) handling variable-length input objects. To solve this challenging learning task, we applied the temporal difference learning algorithm TD(λ) to train a neural network to learn an evaluation function over states. Several modifications to the TD(λ) algorithm were made in order to make learning successful and efficient. In this thesis, we particularly presented a study of a key modification to TD(λ)—backward TD(λ)—which shows that backward TD(λ) learns more accurately and more efficiently than standard forward TD(λ).

We presented a detailed study of learning scheduling knowledge using TD(λ) and hand-engineered schedule features. We evaluated this approach on synthetic problems and on problems from a NASA space shuttle payload processing task. The evaluation function is trained on problems involving a small number of jobs and then tested on larger problems. The test showed that the TD scheduler performs better than the best known existing algorithm for this task—the iterative repair method based on simulated annealing.

A large volume of effort has been put into improving learning performance of the system and into understanding why reinforcement learning works for this domain. Several performance measures were employed to investigate learning behavior. In order to achieve high learning performance, we adjusted our learning algorithm in terms of several parameters and strategies. We also engineered our features continually. Whenever we introduced a new feature, we tested it using both supervised learning and TD learning. With much effort, we eventually achieved success, which showed that TD learning is able to learn a very good heuristic evaluation function. We verified that TD learning works properly in capturing the value function. We can confidently say that TD learning along with a set of good features and a proper neural network is the key to this success.

We noticed that the key shortcoming of the above work was its reliance on hand-engineered input features. We then developed a time-delay neural network architecture to avoid feature hand-engineering. The experimental results showed that the combined TDNN-TD(λ) using low level features can not only match the performance of the hand-engineered features—it can actually perform slightly better. However, a TDNN requires much more time to learn and to evaluate. We discussed the utility problem in scheduling; a TDNN is useful only when we have more time available to prepare a schedule.

In the rest of this chapter, we list the contributions of this work. We conclude this thesis with a discussion of directions for extending this research.

6.1 Contributions

- **Reinforcement learning for state-space search:** This dissertation discussed reinforcement learning methods for solving large-scale state-space search problems. It presented methods for formulating different types of state-space search problems as reinforcement learning problems. It identified the looping policy problem and introduced the dynamic policy to solve this problem.

- **Reinforcement learning for scheduling:** A reinforcement learning problem for job-shop scheduling is formulated based on a repair-based scheduling problem space. This work defined the resource dilation factor as the key reinforcement in the learning task. It also employed a small cost for each action as reinforcement. Based on a study of several possible methods, this work selected the approach that applies the TD(λ) algorithm to train a neural network to learn the evaluation function for scheduling.

- **Backward TD(λ):** Backward TD(λ) differs from standard TD(λ) in that the eligibility trace records future states rather than the history. This work showed that backward TD(λ) is more efficient than standard TD(λ) for optimal control problems. Results for this were shown on a simple problem as well as on the complex scheduling problem. Theoretically, $\lambda = 0$ in backward learning should give the best result. Under function approximation, however, using $\lambda > 0$ may help speed up acquiring an optimal policy. Results on the scheduling problem showed that learning under $\lambda \leq 0.7$ all performs very well.

- **TD learning strategies:** Besides backward TD learning, this thesis presented several other modifications to TD learning. The methods for using these modifications were also discussed. In order to make TD learning successfully learn to predict the value function and to speed up the learning process, it is important to employ the following mechanisms during learning: exploration, overtraining big errors with multiple updates, experience replay, and loop avoidance. Teaching methods also help accelerate the learning process.

- **Measures of learning performance:** Reinforcement learning has been concerned with using convergence and speed of convergence to optimality to measure learning performance. However, for the difficult learning task presented in this dissertation, these measures are not applicable. This work employed cross-validation tests to evaluate learning performance. Four measures from these tests are taken: average RDF, average number of repairs, error of final prediction, and smoothness. The back-propagation error on training problems is also measured. Learning behavior for job-shop scheduling is revealed in detail by these measures.

- **Schedule feature hand-engineering:** Features are critical to learning performance. This work developed features that primarily describe the following information about a schedule: resource utilization, slacks, the RDF, various violation counts, and features for discriminating scheduling problems. It showed that these hand-engineered features can be combined with TD learning methods to train a neural network to learn effective scheduling heuristics. In addition, statistical and supervised learning methods for feature hand-engineering were discussed.

- **Overlapping Gaussian ranges for neural networks:** Overlapping Gaussian ranges can be considered as an extension of the CMAC representation. This work showed that a network using overlapping-Gaussian-range output improves learning performance significantly. It predicts the value function much more accurately than a single-output network.

- **Time-delay neural network architecture:** A time-delay neural network architecture was developed to avoid feature hand-engineering. This architecture extends the time-delay neural network to extract features from irregular-length schedules. The test showed that the combined TDNN-TD(λ) can even learn a slightly better scheduling policy than the feature hand-engineered system. However, its computational complexity could make it impractical for use on large problems.

- **Comparison of TD scheduling and iterative repair scheduling:** The iterative repair scheduling method is the best-known previous (non-learning) solution to the space shuttle payload processing task. This work developed two TD(λ)-based scheduling systems using hand-engineered features and the TDNN respectively. It developed a policy determination method that employs the loop avoidance mechanism and random sample greedy search to TD scheduling. Experimental tests showed that the two TD scheduling systems significantly out-perform iterative repair in terms of the quality of the resulting schedules and the number of search steps required to construct them.

- **Understanding why reinforcement learning works:** Our approach combined TD learning with back-propagation neural network approximation. There is no doubt that a set of good features along with a proper network is essential to this success. Nonetheless, this work proved that TD learning is the key to this success. The test showed that TD learning is not simply copying good input features to the network output. TD learning worked properly in propagating reward backward and learned to make early decisions well.

- **Generality:** TD learning has been tested in several problem sets which include synthetic problems and problems from a NASA space shuttle payload processing task. They all worked well very. Under our current understanding, the learned scheduling strategies are quite general.

6.2 Future Work

One important problem that is left open through the course of this research is: What domain-specific scheduling strategies were learned by reinforcement learning? According to the performance of the resulting scheduling system, we believe that reinforcement learning must have learned some domain-specific knowledge. However, we so far have only very limited understanding of this. We tried to find some factors that could cause different scheduling problems to require different scheduling strategies.

We conducted cross-tests of learned networks trained on different domains; but we did not see significant difference of scheduling capabilities among these networks (Section 4.6). We have only seen that learning acquired some general scheduling strategies.

It is important to know what kind of scheduling strategies can be easily acquired by reinforcement learning and what can not. This is essential to understanding our learning system. With better knowledge about the strengths and the weaknesses of the current system, we will be able to improve learning performance further.

We have envisioned several methods that have the potential to further improve our TD scheduling systems. Some of them were suggested in the earlier chapters. One interesting future direction is to train separate networks for repairs in different stages and then to use different networks in the corresponding stages during scheduling (Section 4.4.3). There is evidence that this will further improve scheduling performance.

Industrial scheduling problems abound. Some problems consider different optimization criteria concerned with cost, tardiness, or throughput. Some problems may involve different types of resource constraints (for example, no resource pools). Some problems may even involve more types of constraints such as state constraints. To apply reinforcement learning to these problems, we would need to define different reinforcements for learning. Reinforcement learning is a general approach, it learns policies for state-space search. This research has shown that reinforcement learning has the great potential for quickly finding high-quality solutions to these scheduling problems. This also shows that reinforcement learning has the potential for succeeding in general combinatorial optimization problems.

This research has shown that reinforcement learning methods can successfully learn scheduling knowledge for finding high-quality solutions to the NASA space shuttle payload process task. A key goal of future research must be to improve reinforcement learning methods so that they can be applied with a minimum of domain-specific engineering to produce a new, cost-effective scheduling technology.

Bibliography

[Aarts, 1989] E. H. L. Aarts. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing.* Wiley, 1989.

[Abramson, 1991] D. Abramson. Constructing school timetables using simulated annealing: Sequential and parallel algorithms. *Management Science*, 37(1):98–113, 1991.

[Albus, 1981] J. S. Albus. *Brain, Behavior, and Robotics.* Byte Books, 1981.

[Aytug *et al.*, 1994] H. Aytug, G. J. Koehler, and J. L. Snowdon. Genetic learning of dynamic scheduling within a simulation environment. *Computer and Operations Research*, 21(8):909–926, 1994.

[Baird, 1995] L. Baird. Residual algorithm: reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 30–37. Morgan Kaufmann, 1995.

[Baker, 1974] K. R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, 1974.

[Baluja and Caruana, 1995] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 38–46. Morgan Kaufmann, 1995.

[Baluja, 1995] S. Baluja. An empirical comparison of seven iterative and evolutionary function optimization heuristics. Tech report, cmu-cs-95-193, Carnegie Mellon University, Pittsburgh, PA, 1995.

[Barto *et al.*, 1983] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike element that can solve difficult learning control problem. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846, 1983.

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

[Bellman, 1957] R. E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[Bertsekas and Tsitsiklis, 1989] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[Bertsekas, 1987] D. P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Biegal and Davern, 1990] J. Biegal and J. Davern. Genetic algorithms and job shop scheduling. *Computer and Industrial Engineering*, 19:81–91, 1990.

[Boyan and Moore, 1995] J. A. Boyan and A. W. Moore. Generalization in reinforcement learning: safely approximating the value function. In *Advances in Neural Information Processing Systems 7*, San Mateo, CA, 1995. Morgan Kaufmann.

[Bradtke, 1993] S. J. Bradtke. Reinforcement learning applied to linear quadratic regulation. In *Advances in Neural Information Processing Systems 5*, pages 295–302, San Mateo, CA, 1993. Morgan Kaufmann.

[Breiman *et al.*, 1984] L. Breiman, J. H. Friedman, R. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth Internatinal Group, Belmont, CA, 1984.

[Cassandra *et al.*, 1994] A. R. Cassandra, L. P. Kaelbling, and M. L. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1023–1028, 1994.

[Chapman and Kaelbling, 1991] D. Chapman and L. P. Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 726–731, 1991.

[Chapman, 1989] D. Chapman. Penguins can make cake. *AI Magazine*, 10:45–50, 1989.

[Cichosz and Mulawka, 1995] P. Cichosz and J. J. Mulawka. Fast and efficient reinforcement learning with truncated temporal difference learning. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 99–107, 1995.

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

[Clouse and Utgoff, 1992] J. A. Clouse and P. E. Utgoff. A teaching method for reinforcement learning. In *Proceedings of the Ninth International Workshop on Machine Learning*, pages 92–101. Morgan Kaufmann, 1992.

[Coffman, Jr., 1976] E. G. Coffman, Jr. *Computer and Job-shop Scheduling Theory.* Wiley, 1976.

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

[Davis, 1985] L. Davis. Job shop scheduling with genetic algorithm. In *Proceedings of the an International Conference on Genetic Algorithms and Their Applications*, pages 136–140, 1985.

[Dayan and Hinton, 1993] P. Dayan and G. E. Hinton. Feudal reinforcement learning. In *Advances in Neural Information Processing Systems 5*, San Mateo, CA, 1993. Morgan Kaufmann.

[Dayan and Sejnowski, 1994] P. Dayan and T. J. Sejnowski. TD(λ) converges with probability 1. *Machine Learning*, 14(3):295–302, 1994.

[Dayan, 1992] P. Dayan. The convergence of TD(λ) for general λ. *Machine Learning*, 8(3/4):341–362, 1992.

[Deale *et al.*, 1994] M. Deale, M. Yvanovich, D. Schnitzius, D. Kautz, M. Carpenter, M. Zweben, G. Davis, and B. Daun. The space shuttle ground processing scheduling system. In M. Zweben and M. S. Fox, editors, *Intelligent Scheduling*, chapter 15, pages 423–449. Morgan Kaufmann, San Francisco, CA, 1994.

[Dean and Lin, 1995] T. Dean and S.-H. Lin. Decomposition techniques for planning in stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1121–1127. Morgan Kaufmann, 1995.

[DeJong and Mooney, 1986] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(1), 1986.

[DeJong, 1992] K. DeJong. Genetic algorithms are NOT function optimizers. In Whitley, editor, *FOGA-2 Foundations of Genetic Algorithms-2*, pages 5–17. Morgan Kaufmann, 1992.

[Dell'Amico, 1993] M. Dell'Amico. Applying tabu-search to the job-shop scheduling problem. *Annals of Operations Research*, 41:231–252, 1993.

[Denardo, 1982] E. V. Denardo. *Dynamic Programming Models and Applications.* Prentice-Hall, Inc, 1982.

[Dietterich and Flann, 1995] T. D. Dietterich and N. S. Flann. Explanation-based learning and reinforcement learning: A unified view. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 176–184. Morgan Kaufmann, 1995.

[Fang *et al.*, 1993] H. L. Fang, P. Ross, and D. Corne. A promising genetic algorithm approach to job-shop scheduling, rescheduling, and open-shop scheduling problems. In *Proceedings of the the Fifth International Conference on Genetic Algorithms*, 1993.

[Fox and Zweben, 1995] M. S. Fox and M. Zweben. Constraint directed search: Theory and practice. Totorial MA4, Fourteenth International Joint Conference on Artificial Intelligence, 1995.

[Fox, 1983] M. S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* PhD thesis, Carnegie Mellon University, 1983.

[Fox, 1994] M. S. Fox. ISIS: A retrospective. In M. Zweben and M. S. Fox, editors, *Intelligent Scheduling*, chapter 1, pages 3–28. Morgan Kaufmann, San Francisco, CA, 1994.

[French, 1982] S. French. *Sequencing and Scheduling.* Wiley, 1982.

[Garey and Johnson, 1979] M. R. Garey and D. S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness.* Freeman and Co, 1979.

[Ginsberg, 1989] M. L. Ginsberg. Universal planning: an (almost) universally bad idea. *AI Magazine*, 10(4):40–44, 1989.

[Glover, 1990] F. Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

[Goldberg, 1989] D. E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning.* Addison Wesley, 1989.

[Gordon, 1995] G. Gordon. Stable function approximation in dynamic programming. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 261–268. Morgan Kaufmann, 1995.

[Grefenstette, 1988] J. J Grefenstette. Credit assignment in rule discorvery systems based on genetic algorithm. *Machine Learning*, 3:225–245, 1988.

[Hilliard *et al.*, 1988] M. R. Hilliard, G. E. Piepens, M. Palmer, and D. J. Kejitan. Machine learning application to job shop scheduling. In *Proceedings of the the fiest International Conference on AI and Expert Systems*, pages 728–733. ACM Press, 1988.

[Holland, 1986] J. H. Holland. Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach, Volume II.* Morgan Kaufmann, 1986.

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

[Howard, 1960] R. Howard. *Dynamic Programming and Markov Decision Problem.* The MIT Press, Mass, 1960.

[Hübscher and Glover, 1994] R. Hübscher and F. Glover. Applying tabu search with influential diversification to multiprocessor scheduling. *Computer and Operations Research*, 21(8):877–884, 1994.

[Icmeli and Erenguc, 1994] O. Icmeli and S. S. Erenguc. A tabu search procedure for the resource constrainted project scheduling problem with discounted cash flows. *Computer and Operations Research*, 21(8):841–854, 1994.

[Jaakkola *et al.*, 1994] T. Jaakkola, M. I. Jordan, and S. P. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6(6):1185–1201, 1994.

[Kaelbling *et al.*, 1996] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A servey. *Journal of AI Research (to appear)*, 1996.

[Kaelbling, 1990] L. P. Kaelbling. *Learning in Embedded Systems*. PhD thesis, Stanford University, June 1990.

[Kaelbling, 1993] L. P. Kaelbling. Hierarchical learning in stochastic domains: Preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning*, 1993.

[Karmarkar, 1984] N. Karmarkar. A new polynomial time algorithm for linear programming. *Combinatorica*, 4(4):373–385, 1984.

[Khachian, 1979] L. G. Khachian. A polynomial algorithm for linear programming. *Soviet Math Dokl.*, 20:191–194, 1979.

[Kirkpatrick *et al.*, 1983] S. Kirkpatrick, C. D. Galatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[Klopf, 1972] A. H. Klopf. Brain function and adaptive systems—a heterostatic theory. Tech report afcrl-72-0164, Air Force Cambridge Research Lab., Bedford, MA, 1972.

[Korf, 1990] R. E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, 1990.

[Laird *et al.*, 1986] J. E. Laird, P. S. Rosenbloom, and A. Newell. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[Lang *et al.*, 1990] K. J. Lang, A. H. Waibel, and G. E. Hinton. A time-delay neural network architecture for isolated word recognition. *Neural Networks*, 3:33–43, 1990.

[Lang, 1995] K. J. Lang. Hill climbing beats genetic search on a boolean circuit synthesis problem of koza's. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 340–343. Morgan Kaufmann, 1995.

[LeCun *et al.*, 1989] Y. LeCun, B. Boser, J. S. Deniker, and D. Henderson et al. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551, 1989.

[Lee and Piramuthu, Oct 1994] C. Lee and S. Piramuthu. Global job shop scheduling with genetic algorithm and machine learning. *ORSA/TIMES Detroit*, Oct, 1994.

[Lin, 1991] Longji Lin. Programming robots using reinforcement learning and teaching. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 781–786, 1991.

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

[Lin, 1993] Longji Lin. *Reinforcement Learning for Robots using Neural Networks.* PhD thesis, Carnegie Mellon University, 1993.

[Littman *et al.*, 1995a] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling. Learning policies for partially observable environments: scaling up. In *Proceedings of the Twelfth International Conference on Machine Learning*, 1995.

[Littman *et al.*, 1995b] M. L. Littman, T. L. Dean, and L. P. Kaelbling. On the complexity of solving Markov decision problems. In *Proceedings of the the 11th Annual Conference of Uncertainty and AI (UAI-95)*, 1995.

[Lourenco and Zwijnenburg, 1995] H. R. Lourenco and M. Zwijnenburg. Combining the large-step optimization with tabu-search: Application to the job-shop scheduling problem. Presented in the first AI and OR workshop, Timberline, Oregon, 1995.

[Minton *et al.*, 1992] S. Minton, M. D. Johnson, A. B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problem. *Artificial Intelligence*, 58:161–205, 1992.

[Minton, 1990] S. Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42:363–392, 1990.

[Mitchell *et al.*, 1983] T. M. Mitchell, P. E. Utgoff, and R. B. Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach (Vol I)*. Morgan Kaufmann, 1983.

[Mitchell *et al.*, 1986] T. M. Mitchell, R. M. Keller, and S. t. Kedar-Cabelli. Explanation-based generalization: a unifying view. *Machine Learning*, 1(1), 1986.

[Mittenthal *et al.*, 1993] J. Mittenthal, M. Raghavachari, and A. Rana. A hybrid simulated annealing approach for single machine scheduling problems with non-regular penalty functions. *Computer and Operations Research*, 20(2):103–111, 1993.

[Montana and Davis, 1989] D. Montana and L. Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 762–767, 1989.

[Moore and Atkeson, 1993] A. W. Moore and C. G. Atkeson. Prioritized sweeping: reinforcement learning with less data and less real time. *Machine Learning*, 13(1):103–130, 1993.

[Moore, 1990] A. W. Moore. *Efficient Memory-Based Learning for Robot Control.* PhD thesis, University of Cambridge, UK, 1990.

[Moore, 1991] A. W. Moore. Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 333–337, San Mateo, CA, 1991. Morgan Kaufmann.

[Moore, 1994] A. W. Moore. The parti-game algorithm for variable resolution reinforcement learning. In *Advances in Neural Information Processing Systems 6*, San Mateo, CA, 1994. Morgan Kaufmann.

[Nordström and Tufekci, 1994] A. L. Nordström and S. Tufekci. A genetic algorithm for the talent scheduling problem. *Computer and Operations Research*, 21(8):927–940, 1994.

[Ogbu and Smith, 1990] F. A. Ogbu and D. K. Smith. The application of the simulated annealing algorithm to the solution of the n/m/c max flowshop problem. *Computer and Operations Research*, 17(3):243–253, 1990.

[Parr and Russell, 1995] R. Parr and S. Russell. Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1088–1094, 1995.

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

[Puterman and Shin, 1978] M. L. Puterman and M. C. Shin. Modified policy iteration algorithm for discounted Markov decision problem. *Management Science*, 24(11):1127–1137, 1978.

[Puterman, 1990] M. L. Puterman. Markov decision process. In D. P. Heyman and M. J. Sobel, editors, *Handbooks in Operations Research and Management Science: Stochastic Models*, chapter 8, pages 331–434. North-Holland, Amsterdam, 1990.

[Puterman, 1994] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley, 1994.

[Rich and Knight, 1991] E. Rich and K. Knight. *Artificial Intelligence.* McGraw-Hill, Inc., second edition, 1991.

[Ross, 1983] S. M. Ross. *Introduction to Stochastic Dynamic Programming.* Academic, New York, 1983.

[Rummery and Niranjan, 1994] G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Tech report cued/f-infeng/td 166, Cambridge University, Engineering Dept, 1994.

[Russell and Norvig, 1995] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, New Jersey, 1995.

[Samuel, 1959] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:211–229, 1959.

[Schoppers, 1987] M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1039–1046, 1987.

[Schoppers, 1989] M. J. Schoppers. In defense of reaction plans as caches. *AI Magazine*, 10:51–60, 1989.

[Schraudolph *et al.*, 1994] N. Schraudolph, P. Dayan, and T. Sejnowski. Using TD(λ) to learn an evaluation function for the game of go. In *Advances in Neural Information Processing Systems 6*, San Mateo, CA, 1994. Morgan Kaufmann.

[Schwartz, 1993] A. Schwartz. A reinforcement learning method for maximizing undiscounted reward. In *Proceedings of the Tenth International Conference on Machine Learning*. Morgan Kaufmann, 1993.

[Singh and Sutton, 1996] S. P. Singh and R. S. Sutton. Reinforcement learning with replacing eligibility trace. *Machine Learning (to appear)*, 1996.

[Smith and Ow, 1986] S. F. Smith and P. S. Ow. Constructing and maintaining detailed production plans: Inverstigations into the development of knowledge-based factory scheduling systems. *AI magazine*, 7(4):45–61, 1986.

[Smith *et al.*, 1986] S. F. Smith, P. S. Ow, C. Le Pape, B. McLarn, and N. Muscettola. Integrating mupltiple scheduling perspectives to generate detailed production plans. In *Proceedings 1986 SME Conference o AI in Manufacturing*, pages 123–137, 1986.

[Smith, 1994] S. F. Smith. OPIS: A metholology and architecture for reactive scheduling. In M. Zweben and M. S. Fox, editors, *Intelligent Scheduling*, chapter 2, pages 29–66. Morgan Kaufmann, San Francisco, CA, 1994.

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

[Sutton, 1990] R. S. Sutton. Integrated architecture for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Senventh International Workshop on Machine Learning*, pages 216–224. Morgan Kaufmann, 1990.

[Sutton, 1991] R. S. Sutton. Planning by incremental dynamic programming. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 353–357. Morgan Kaufmann, 1991.

[Sutton, 1996] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, Cambridge, MA, 1996. MIT Press.

[Syswerda, 1991] G. Syswerda. Schedule optimization using genetic algorithms. In L. Davis, editor, *Handbook of Genetic Algorithms*, pages 332–349. Van Nostrand, 1991.

[Tadepalli and Ok, 1994] P. Tadepalli and D. Ok. Discounting considered harmful: A comparison of reinforcement learning methods in automatic guided vehicle scheduling. In *Proceedings of Robot Learning workshop*, New Brunswick, NJ, 1994.

[Tesauro, 1992] G. J. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8(3/4):257–277, 1992.

[Thrun and Schwartz, 1993] S. Thrun and A. Schwartz. Issues in using approximation for reinforcement learning. In *Proceedings of the Fourth Connectionist Models Summer School*, Hillsdale, NJ, 1993. Lawrence Erlbaum Publisher.

[Thrun, 1992] S. Thrun. The role of exploration in learning control. In D. A. White and D. A. Sofge, editors, *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*. Van Nostrand Reinhold, 1992.

[Tseng, 1990] P. Tseng. Solving H-horizon, stationary Markov decision problems in time proportional to log(h). *Operations Research Letters*, 9(5):287–297, 1990.

[Tsitsiklis, 1994] J. Tsitsiklis. Asynchronous stochastic approximation and Q-learning. *Machine Learning*, 16(3):185–202, 1994.

[Tsitsiklis, 1995] J. Tsitsiklis. Counterexample to temporal difference learning. *Neural Computation*, 7(2):270–279, 1995.

[Uckun *et al.*, Oct 1993] S. Uckun, S. Bagchi, K. Kawamura, and Y. Miyabe. Managing genetic search in job shop scheduling. *IEEE Expert*, pages 15–24, Oct, 1993.

[Utgoff and Clouse, 1991] P. E. Utgoff and J. A. Clouse. Two kinds of training information for evaluation function learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 596–600, 1991.

[Van Laarhoven *et al.*, 1992] P. J. M. Van Laarhoven, E. H. L. Aarts, and J. K. Lenstra. Job shop scheduling by simulated annealing. *Operations Research*, 40:113–125, 1992.

[Van Laarhoven, 1988] P. J. M. Van Laarhoven. *Theoretical and Computational Aspects of Simulated Annealing*. PhD thesis, Erasmus University, Rotterdam, Netherlands, 1988.

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

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

[Whitehead, 1991] S. D. Whitehead. A complexity analysis of cooperative mechanisms in reinforcement learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 607–613, 1991.

[Whitley *et al.*, 1989] D. Whitley, T. Starkweather, and D. Fuquay. Scheduling problems and traveling salesmen: the genetic edge recombination operator. In *Proceedings of the the Third International Conference on Genetic Algorithms*, pages 133–140. Morgan Kaufmann, 1989.

[Williams and Baird, 1993] R J. Williams and L. C. Baird. Tight performance bounds on greedy policies based on imperfect value functions. Tech report nu-ccs-93-14, Northeastern University, College of Computer Science, Boston, MA, 1993.

[Woodruff, 1994] D. L. Woodruff. Simulated annealing and tabu search: Lessons from a line search. *Computer and Operations Research*, 21(8):823–840, 1994.

[Zhang and Dietterich, 1995] W. Zhang and T. Dietterich. A reinforcement learning approach to job-shop scheduling. In *IJCAI-95*, pages 1114–1120, 1995.

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

[Zweben *et al.*, 1992] M. Zweben, E. Davis, B. Daun, E. Drascher, M. Deale, and M. Eskey. Learning to improve constraint-based scheduling. *Artificial Intelligence*, 58:271–296, 1992.

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

APPENDIX

Appendix
Experimental Problem Sets

This appendix describes the problem specification for two artificial problem sets ART-1 and ART-2 and for a problem set from a NASA space shuttle payload processing task SSPP. Table A.1, Table A.2, and Table A.3 give the specification for the three problem sets respectively. Table A.4 lists the set of training problems in SSPP which shows how the problems are generated.

Table A.1. ART-1 Problem Specification. ART-1 contains 50 training problems, 50 test problems, and 20 scale-up test problems.

Environment	
Number of Resource Types	2
Number of Jobs	20
Resource	
Number of Pools for Each Type of Resources	2
First Pool Capacity	6
Second Pool Capacity	8
Job	
Number of Tasks in a Job	6 to 10
Density of Precedence Constraints	60%
Task	
Duration	6 to 15
Chance of Choosing One Resource Type	50%
Chance of Choosing Two Resource Types	50%
Maximal Requested Resource Quantity for Each Type	6
Problem	
Job Due Time Separation	8 to 15
Number of Jobs in a Small Problem	3 or 4
Number of Jobs in a Large Problem	15 to 20

Table A.2. ART-2 Problem Specification. ART-2 contains four subsets of problems generated using the combinations of job due-time separation and resource capacity parameter settings. Each subset of problems has 50 training problems and 50 test problems.

Environment		
Number of Resource Types	2	
Number of Jobs	20	
Resource		
Number of Pools for Each Type of Resources	2	
First Pool Capacity	Small:	6
	Large:	8
Second Pool Capacity	Small:	8
	Large:	10
Job		
Number of Tasks in a Job	6 to 10	
Density of Precedence Constraints	60%	
Task		
Duration	6 to 15	
Chance of Choosing One Resource Type	50%	
Chance of Choosing Two Resource Types	50%	
Maximal Requested Resource Quantity for Each Type	6	
Problem		
Number of Jobs	3 or 4	
Job Due Time Separation (t_{JDS})	Small:	4 to 7
	Large:	8 to 15
Problem Set		
	t_{JDS}	*Resource Capacity*
Subset-1	Large	Large
Subset-2	Large	Small
Subset-3	Small	Large
Subset-4	Small	Small

Table A.3. SSPP Problem Specification. SSPP contains 20 training problems and 20 test problems. Each training problem contains between two and four shuttle missions. Each test problem each contains three to six shuttle missions.

Resource		
Number of Resource Types	35	
Payload		
Payload Name	*Number of Tasks*	
Long Module (LM)	65	
MPESS	32	
Pallet and Igloo (PI)	82	
Mission		
Chance of One Payload	70%	
Chance of Two Payloads	30%	
Launch Date Separation	30 to 90 Days	
Problem		
Problem Type	*Number of Payloads*	*Number of Missions*
Training Problem	3 or 4	2 to 4
Test Problem	5 or 6	3 to 6

Table A.4. SSPP Training Problems. It shows a complete list of 20 training problems. Each mission in a problem is described with the names of payloads needed to be processed along with a mission launch date. For example, "LM+MPESS@14(1/14/97)" specifies a mission that contains two payloads LM and MPESS to process and that must be scheduled to launch on 1/14/97.

Problem	*Mission-1*	*Mission-2*	*Mission-3*	*Mission-4*
1	LM@43(2/12/97)	LM@113(4/23/97)	PI@194(7/13/97)	
2	LM+MPESS@14(1/14/97)	LM@72(3/13/97)		
3	PI@28(1/28/97)	MPESS+PI@76(3/17/97)		
4	MPESS@4(1/4/97)	MPESS@58(2/27/97)	LM@128(5/8/97)	
5	LM+MPESS@42(2/11/97)	MPESS@104(4/14/97)	LM@162(6/11/97)	
6	MPESS@41(2/10/97)	MPESS+LM@81(3/22/97)	LM@155(6/4/97)	
7	MPESS@60(3/1/97)	MPESS@107(4/17/97)	MPESS@155(6/4/97)	
8	MPESS@10(1/10/97)	MPESS@72(3/13/97)	PI@111(4/21/97)	
9	MPESS@17(1/17/97)	MPESS@55(2/24/97)	PI@99(4/9/97)	PI@162(6/11/97)
10	LM+MPESS@3(1/3/97)	MPESS@75(3/16/97)		
11	PI+MPESS@21(1/21/97)	LM+PI@107(4/17/97)		
12	LM@2(1/2/97)	LM@44(2/13/97)	MPESS@106(4/16/97)	
13	LM@4(1/4/97)	LM@42(2/11/97)	MPESS@120(4/30/97)	PI@156(6/5/97)
14	MPESS+PI@16(1/16/97)	PI@56(2/25/97)		
15	MPESS+MPESS@53(2/22/97)	LM+PI@90(3/31/97)		
16	PI+MPESS@6(1/6/97)	MPESS@38(2/7/97)	MPESS@84(3/25/97)	
17	PI@37(2/6/97)	LM@69(3/10/97)	LM+MPESS@112(4/22/97)	
18	MPESS@51(2/20/97)	PI+LM@98(4/8/97)	PI@149(5/29/97)	
19	PI@24(1/24/97)	PI+PI@102(4/12/97)	MPESS@177(6/26/97)	
20	MPESS@23(1/23/97)	PI+PI@107(4/17/97)	LM@168(6/17/97)	