Applying Probabilistic Rules To Relational Worlds

by

Natalia Hernandez Gardiol

B.S. Computer Science, Michigan State University (1999)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

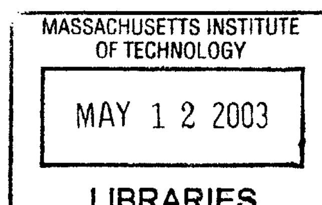
February 2003

© Massachusetts Institute of Technology 2003. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
November 25, 2002

Certified by.........................
Leslie Pack Kaelbling
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
MAY 1 2 2003
LIBRARIES
BARKER

Applying Probabilistic Rules To Relational Worlds

by

Natalia Hernandez Gardiol

Submitted to the Department of Electrical Engineering and Computer Science
on November 25, 2002, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Being able to represent and reason about the world as though it were composed of "objects" seems like a useful abstraction. The typical approach to representing a world composed of objects is to use a relational representation; however, other representations, such as deictic representations, have also been studied. I am interested not only in an agent that is able to represent objects, but in one that is also able to act in order to achieve some task. This requires the ability to learn a plan of action. While value-based approaches to learning plans have been studied in the past, both with relational and deictic representations, I believe the shortcomings uncovered by those studies can be overcome by the use of a world model. Knowledge about how the world works has the advantage of being re-usable across specific tasks. In general, however, it is difficult to obtain a completely specified model about the world. This work attempts to characterize an approach to planning in a relational domain when the world model is represented as a potentially incomplete and/or redundant set of uncertain rules.

Thesis Supervisor: Leslie Pack Kaelbling
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Without question, my thanks first and foremost go to my advisor, Leslie Pack Kaelbling. Without her keen insights, gentle advice, and unending patience, none of this work would have been possible. She is a constant and refreshing source of inspiration to me and to the people around her, and I am grateful to count myself among her students.

I thank my family for their unwavering support of my endeavors, both academic and otherwise, foolish and not. I owe to them the very basic notion that the natural world is an amazing thing about which to be curious. I thank my father especially, for demonstrating by his own example the discipline and the wonder that go with scientific inquiry. Los quiero mucho.

Here at MIT, I owe many people thanks for their support and patience. Sarah Finney has been an awesome collaborator and office-mate, and has taught me a great deal about how to ask questions. Luke Zettlemoyer sat with me on countless late nights discussing the intricacies of well-defined probability distributions. Terran Lane and Tim Oates generously shared their expertise with me and provided countless useful pointers. I thank Marilyn Pierce at the EECS headquarters for her amazing resourcefulness, patience, and personal attention, and for always keeping the students' best interests at heart.

This work was done within the Artificial Intelligence Lab at Massachusetts Institute of Technology. The research was sponsored by the Office of Naval Research contract N00014-00-1-0298, by the Nippon Telegraph & Telephone Corporation as part of the NTT/MIT Collaboration Agreement, and by a National Science Foundation Graduate Research Fellowship.

Contents

1 Introduction: Why Objects? **13**

2 Background **16**

2.1 Representational Approaches to Worlds with Objects 17

2.1.1 Relational Representations 17

2.1.2 Deictic Representations 18

2.2 Acting in Worlds with Objects: Value-Based Approaches 19

2.2.1 Relational Reinforcement Learning 19

2.2.2 Reinforcement Learning With Deictic Representations 22

2.3 The Case for a World-Model 23

2.4 Reasoning with Probabilistic Knowledge 24

2.4.1 An Example: When Rules Overlap 25

2.5 Related Work in Probabilistic Logic 30

2.6 Related Work in Policy Learning for Blocks World 32

3 Reasoning System: Theoretical Basis **34**

3.1 Probabilistic Inference with Incomplete Information 35

3.2 Using Probabilistic Relational Rules as a Generative Model 38

3.2.1 The Sparse-Sampling Planner 39

3.2.2 CLAUDIEN: Learning Probabilistic Rules 40

4 Experiments **42**

4.1 A Deterministic Blocks World 43

4.1.1 Acquiring the Set of Rules 46

4.1.2 Deterministic Planning Experiment 48

4.2 A Stochastic World with Complex Dynamics 50

4.2.1 Acquiring the Rules for the Stochastic Stacking Task 54

4.3 A Stochastic World with Simpler Dynamics 57

4.3.1 Acquiring the Rules for the Simpler Stochastic Stacking Task . 59

4.3.2 Planning Experiments 64

5 Conclusions **77**

5.1 Combining Evidence 77

5.2 Learning and Planning with Rules 78

5.3 Future Work 79

A Rules Learned by CLAUDIEN For the Deterministic Blocks World **84**

List of Figures

2-1 How should we represent the blocks and the relationships between the blocks in this example blocks world? 16

2-2 The DBN induced by the `move(c,d)` action in the example world. . . 26

2-3 The Noisy-OR combination rule. 29

4-1 The RRL blocks world. The goal is to achieve `on(a,b)`. 43

4-2 The deterministic world used by our system. The table is made of three *table* blocks, which may be subject to *move* attempts but are actually fixed. 44

4-3 The DLAB template for the deterministic blocks world. This template describes rules for how actions, in conjunction with pre-conditions and/or a goal predicate, either produce a certain reward (either 1.0 or 0.0) or influence a single state predicate. In other words, it describes a mapping from actions, state elements, and the goal predicate into a reward value of 1.0, a reward value of 0.0, or the post-state of a state element. The expression `1-1` before a set means that exactly one element of the set must appear in the rule; similarly, `3-3` means exactly three elements, `0-len` means zero or more, etc. 47

4-4 Sample output from the planning experiment in the deterministic small blocks world. 49

4-5 The large stochastic blocks world. The table is made up of fixed, indistinguishable table blocks that *may not* be the subject of a move action. 51

4-6 The DLAB template, without background knowledge, used in the complex block-stacking task in the large stochastic world. 55

4-7 The DLAB template, with background knowledge about *next-to* used in the complex block-stacking task. 55

4-8 The DLAB template used in the simpler stochastic block-stacking task. The same template was used in all stochasticity versions of the task. . 60

4-9 CLAUDIEN rules for the high-stochasticity version of the stacking task. 61

4-10 CLAUDIEN rules for the mid-stochasticity version of the stacking task. 61

4-11 CLAUDIEN rules for the low-stochasticity version of the stacking task. 62

4-12 Hand-coded rules, and corresponding probabilities, for the high-stochasticity version of the stacking task. 63

4-13 Hand-coded rules, and corresponding probabilities, for the mid-stochasticity version of the stacking task. 63

4-14 Hand-coded rules, and corresponding probabilities, for the low-stochasticity version of the stacking task. 63

4-15 Plot of cumulative reward for the hand-coded rules in the one-step planning experiments. 65

4-16 Plot of cumulative reward for CLAUDIEN rules in the one-step planning experiment. 66

4-17 A small, example world for illustrating the DBN that results from applying rules from the hand-coded set. 67

4-18 The DBN induced by applying the hand-coded rules to the example world in Figure 4-17 and the action `move(n3,n1)`. It is sparse enough that a combination rule does not need to be applied. 69

4-19 Initial configuration of blocks in all the one-step planning experiments. The configuration can also be seen schematically in Figure 4-5. Each block is represented by a pair of square brackets around the blocks' name and the first letter of the block's color (e.g., block *n20* is red; *n00* is a table block). 70

4-20 Output from the CLAUDIEN rules in the low-stochasticity one-step task. 71

4-21 Output from the hand-coded rules in the low-stochasticity one-step task. 72

4-22 The smaller world used in the three-step planning experiments. . . . 73

4-23 Initial configuration of blocks in the three-step planning experiments. The configuration can also be seen schematically in Figure 4-22. . . . 73

4-24 Output from the CLAUDIEN rules in the low-stochasticity three-step task. 74

4-25 Output from the hand-coded rules in the low-stochasticity three-step task. 74

List of Tables

2.1 The conditional probability table for `ontable(c)`. 27

4.1 Parameters tested in the one- and multi-step planning experiments. . 64

Chapter 1

Introduction: Why Objects?

When human agents interact with their world, they often think of their environment as being composed of objects. These objects (chairs, desks, cars, pencils) are often described in terms of their properties as well as in terms of their relationship to the agent and to other objects. Abstracting the world into a set of "objects" is useful not only for achieving a representation of what the agent perceives, but also for giving the agent a vocabulary for describing the goals and effects of its actions. Any "object" with the right set of properties can be the target of an action that knows how to handle objects with such properties, regardless of the specific object's identity. For example, if I have learned how to drink from a certain cup, then I should expect to be able to transfer my knowledge when presented with a different cup of approximately the same size and shape.

I want to build an artificial agent that can reason about objects in the world. Such an endeavor raises important questions about how objects in the world should be represented. When AI researchers first attempted to build systems that could handle a world with objects, a common approach was to use a first-order representation of the world and to learn properties about the world via inductive logic programming systems [43]. Traditional logic programming systems, such as those based on

the language Prolog, are powerful, but they unfortunately require the world to be deterministic.

In contrast, for dealing with the world probabilistically, Bayesian networks [48] provide an elegant framework. The difficulty with Bayes' nets, however, is that they are only able to manage propositional representations of state; that is, each node in the network stands for a particular attribute of the world and represents it as a random variable. The limitations of propositional representations are well-known: with propositional representations of the world, it is hard to reason about the relationships between concepts in general; all instances of the concept must be articulated and represented explicitly.

Consequently, there has been much recent interest in probabilistic extensions to first-order logic; that is, in ways to represent the world relationally (as in logic programming) and to reason about such representations in a manner that can handle uncertainty (as in Bayesian networks).

Probabilistic logic systems seek to answer queries about the world from probabilistic relational knowledge bases. This is important for an agent who needs to reason in a non-deterministic environment with objects. The concrete question I am interested in is how an agent with probabilistic relational knowledge might use its knowledge in order to act.

In general, there are two basic approaches for an agent to develop courses of action. The first, and simplest, is to directly learn a mapping from observations to actions without explicitly learning about how the world works. This approach, within a relational context, has been investigated by recent work on relational reinforcement learning [23, 21]. The second approach is to try to build up some kind of model of the world, and to use this model to predict the effects of actions.

If the agent has a model of the world, it can query this model and use the resulting predictions to make a plan. Traditionally, the complexity of the planning problem

scales badly with the size of the world representation. Kearns *et al.* give an algorithm for a sparse, approximate planning algorithm that works with a generative model of the world [31]. Unfortunately, a major difficulty is that acquiring a generative model of the world, in practice, is often intractable.

The system described in the following pages is an attempt to build an artificial agent that represents objects with a relational representation, builds a generative world model with probabilistic rules [15], and acts by sparse planning with the (potentially incomplete) model. The probabilistic rules that I consider are quite simple: they probabilistically map a first-order representation of a state situation and action to the first-order representation of the resulting state.

Chapter 2

Background

This chapter discusses some of the previous work that motivated this study. It examines some previous approaches to representing objects, learning in worlds with objects, and reasoning probabilistically about objects.

For the purposes of illustration, let us consider the following running example, seen in Figure 2-1. In this very small blocks world, there are three blocks; the blocks have certain properties and they are related to each other in certain ways. For example, block b is stacked on top of block a, and block c has nothing on top of it. Furthermore, we may have some general knowledge that it is only reasonable to try to move a block if it has no other blocks on top of it, and we might want to apply this knowledge to the block c. How to write down these properties, relations, and rules meaningfully is thus the subject of the following section.

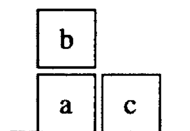


Figure 2-1: How should we represent the blocks and the relationships between the blocks in this example blocks world?

2.1 Representational Approaches to Worlds with Objects

When attempting to represent a world as being composed of *objects*, there are a number of ways to proceed. Below, we take a look at two approaches that have been considered in the past.

2.1.1 Relational Representations

The typical approach to describing objects and their relations is with a *relational representation.* For example, to describe the small blocks world shown in Figure 2-1, we could begin by writing down the following relational description:

```
on(b,a)
nextto(c,a)
ontable(c)
ontable(a)
clear(b)
clear(c)
```

Relational representations have a great deal of expressive power. For example, we can articulate our rule for moving clear blocks onto other clear blocks in a way that is not specific to any individual object:

$$\forall C, B.\ clear(C) \wedge clear(B) \wedge move(C, B) \rightarrow on(C, B)$$

This ability to generalize, to talk about any object satisfying particular properties, is what makes relational representations so appealing as a descriptive language. This is in contrast to *propositional*, or attribute vector, representations, which must refer to each individual object by name. Thus, when we want to represent a piece of knowledge, it must be repeated for every individual object in our universe:

```
clear(a), clear(b), move(a,b) → on(a,b)
clear(a), clear(c), move(a,c) → on(a,c)
clear(b), clear(a), move(b,a) → on(b,a)
clear(b), clear(c), move(b,c) → on(b,c)
...
```

Even though relational representations are very convenient for describing situations succinctly, the question of how best to *learn* policies using a relational representation remains an open problem. The most widely used reinforcement learning algorithms [55] all require propositional representations of state. Thus, even though relations are a compact and powerful way of articulating knowledge and policies, learning in such an expressive space remains difficult for standard methods.

2.1.2 Deictic Representations

Deictic representations hold the promise of potentially bridging the gap between purely propositional and purely relational representations. The term *deictic representation* came into common use with the work of Agre and Chapman on the Pengi domain [2]. It comes from the Greek word *deiktikos*, which means "to point": a deictic expression is one that uses a conceptual marker to "point" at something in the environment, for example, *the-block-that-I'm-looking-at.* An agent with an attentional marker on block *c* might describe the blocks world in Figure 2-1 like this:

```
ontable(the-block-that-I'm-looking-at).
clear(the-block-that-I'm-looking-at).
has-block-nextto(the-block-that-I'm-looking-at).
...
```

Deictic expressions themselves are propositional. Generalization is obtained by actively moving the marker, or focus of attention, about. Depending on whether the agent is looking at block a, b or c, it will be able to use its knowledge about *the-block-that-I'm-looking-at* in an adaptive way: the meaning of the attentional marker is relative to the agent and the context in which it is being used.

At one extreme, essentially replicating a full propositional representation, the agent could have one marker for each object in its environment. In general, however, this is undesirable: the appeal of a deictic representation rests on the notion of a limited attentional resource that the agent actively directs to parts of the world relevant to its task. As a result, a deictic representation generally results in partial observability: the agent can directly observe part of its state space, but not all of it. So, while standard propositional reinforcement learning methods can be applied directly to a deictic representation, the partial observability is problematic for many learning algorithms [24].

2.2 Acting in Worlds with Objects: Value-Based Approaches

This section describes recent work in value-based approaches to learning with relational and deictic representations. As previously noted, a simple way to learn how to act is to learn a mapping from observations to actions without explicitly learning about how the world works. This approach is known as model-free reinforcement learning [55]. Recent work in reinforcement learning has focused on investigating representations that have generalization properties while remaining amenable to model-free approaches [23, 21]. The reason for investigating alternative representations is to avoid the limitations of propositional representations: because propositional representations are grounded in terms of specific objects, as soon as the world or the task changes slightly, the agent is forced to re-learn its strategy from scratch.

2.2.1 Relational Reinforcement Learning

The work of Dzeroski, De Raedt, and Driessens [19] on relational reinforcement learning uses a tree-based function approximator to approximate the Q-values in a rela-

tional blocks world.

The relational reinforcement learning (RRL) algorithm is a logic-based regression algorithm that assumes a deterministic, Markovian domain. The planning task includes:

- A set of states. The states are described in terms of a list of ground facts.
- A set of actions.
- A set of pre-conditions on the action specifying whether an action is applicable in a particular state.
- A *goal* predicate that indicates whether the specified goal is satisfied by the current state.
- A starting state.

In its basic form, the algorithm is a batch algorithm that operates on a group training episodes. An *episode* is a sequence of states and actions from the initial state to the state in which the goal was satisfied. There is also an incremental version of the algorithm, described by Driessens [19]. Here is the basic RRL algorithm:

1. Generate an episode; that is, a sequence of examples where each example consists of a state, and action, and the immediate reward.
2. Compute Q-values [56] for each example in the episode, working backwards from the end of the episode. The last example in an episode (i.e., the one in which the goal is achieved) is defined to have a Q-value of zero. The Q-values are computed with the following rule, where the Q-value for example j is being computed from example $j+1$:

$$Q(s_j, a) \leftarrow r_{j+1} + \gamma max_{a'} Q(s_{j+1}, a').$$

3. Present the episode, labeled with the computed Q-values, to the TILDE-RT [15] algorithm. Given labeled data, TILDE-RT induces logical decision trees. In the case of RRL, the decision trees would specify which predicate symbols in the state description are sufficient to predict the Q-value for that state.

RRL assumes a deterministic domain. It stores all the training examples and regenerates the decision tree (also called regression tree, or Q-tree) from scratch after each episode.

In the longer version of the paper ([21, 22]), Dzeroski *et al.* augment the batch RRL algorithm by learning what are called *P-trees* in addition to the Q-trees. The basic problem is that value functions implicitly encode the "distance to the goal"; in the case of blocks world, this limits the applicability of the learned value function when moving to a domain with a different number of blocks. Thus, given a Q-tree, a P-tree can be constructed that simply lists the optimal action for a state (i.e., the action with the highest Q-value) rather than storing explicitly the Q-value for each action. Dzeroski *et al.* find that policies taken from P-trees generalize best when the Q-tree is initially learned in small domains; then, the P-tree usually generalizes well to domains with a different number of blocks.

Additionally, the authors noted that for more complicated tasks (namely, achieving `on(a,b)`) learning was hindered without a way to represent some intermediate concepts, such as `clear(a)`, or `number-of-blocks-above(a,`*N*`)`. They note that being able to express such concepts seems key for the success of policy learning. The apparently "delicate" nature of these intermediate concepts, their inherent domain dependence, argues strongly for a system that can define such concepts for itself.

The incremental version [19] uses an incremental tree algorithm based on the G algorithm of Chapman and Kaelbling [12] to build the regression trees as data arrives. Apart from the difficulties inherent to growing a decision tree incrementally,[1] the main

[1]Namely, having to commit to a branch potentially prematurely. See the discussions in [19, 23] for details.

challenge presented by the relational formulation is in what *refinement operator* to use for proposing new tree branches. In contrast to the propositional case, where a query branch is refined[2] by simply proposing to extend the branch by a feature not already tested higher in the tree, in the relational case there are any number of ways to refine a query. Proposing and tracking the candidate queries is a difficult task, and it is clearly an important and open problem.

2.2.2 Reinforcement Learning With Deictic Representations

Deictic representation is a way of achieving generalization that appears to avoid some of the difficulties presented by relational representations. Because of their propositional nature, deictic representations can be used directly in standard reinforcement learning approaches. However, the consequence of having limited attentional resources is that the world now becomes partially observable; that is to say, it is described by a partially observable Markov decision process (POMDP) [49].

When the agent is unable to determine the state of the world with its immediate observation, the agent must refer to information outside of the immediate perception in order to disambiguate its situation in the world. The typical approaches are to either add a fixed window of short-term history [42, 41] or to try to build a generative model of the world for state estimation [30, 36]. Reinforcement learning with the history-based approach is simple and appealing, but there can be some severe consequences stemming from this approach [23].

Finney *et al.*[24] showed that using short-term history with deictic representations can be especially problematic. They studied a blocks-world task in which the agent was given one attentional marker and one "place-holding" marker. The environment consisted of a small number of blocks in which the task was to uncover a green block and pick it up. The agent was given actions to pick up blocks, put down blocks,

[2] "Refining a query" is sometimes spoken of as "adding a distinction" to a tree branch.

and move the attentional marker. At each time step, the agent received perceptual information (color, location, etc) about the block on which the attentional marker was focused. Neuro-dynamic programming [7] and a tree-based function approximator [12] were used to map short history sequences (i.e., action and observation pairs) to values.

This approach generates a dilemma. Because the history the agent needs in order to disambiguate its immediate observations depends on the actions it has executed, until it has learned an effective policy, its history is usually uninformative. It is even possible, with a sufficiently poor action choices, to actively push disambiguating information off the end of the history window. Alarmingly, this can happen *regardless* of the length of the history window. This difficulty was also noticed by McCallum in his work on reinforcement learning with deictic representations [41]. McCallum was able to get around this problem by guiding the agent's initial exploration. Nevertheless, the question of how best to use your history when you do not yet know what to do is a very fundamental bootstrapping problem with no obvious solution.

2.3 The Case for a World-Model

The challenges presented by model-free, value-based reinforcement learning, with both relational and deictic representations, argue compellingly for a model with which to estimate the state of the world.

When the agent has a generative model of the world, it can query this model to decide its next action. The idea of an agent with some knowledge of world dynamics is appealing because then the agent is able to hypothesize the effects of its actions, rather than simply reacting to its immediate observation or to windows of history. A model allows the agent to plan its actions based on the current state of the world, rather than having to compute a whole policy in advance. However, the complexity of generating a plan typically scales badly with the size of the world representation and the number of possible actions. To address the scaling problem, Kearns *et al.*

give an algorithm for a sparse, approximate planning algorithm that works with a generative model of the world [31].

A major difficulty is that learning a full generative model of the world is often hopelessly intractable. It is often unreasonable to expect the agent to come up with a full model of the world before it can begin to act. Intuitively, it seems likely that the agent should be able to begin to apply whatever knowledge it has as soon as it is acquired.

This is the major thrust of this work: exploring what it means to plan with a world model that is by necessity incomplete and perhaps inconveniently redundant.

2.4 Reasoning with Probabilistic Knowledge

When an agent's knowledge about the world is incomplete, the theory of probability becomes extremely useful. Probability theory gives a powerful way to address the uncertainty brought on by missing information. The converse of not having enough information is having pieces of evidence that overlap. How to reconcile bits of knowledge when their predictions about the next state of the world overlap is another important issue that requires probability theory.

The approach described in this work adopts the spirit of *direct inference*: it takes general statistical information and uses it to draw conclusions about specific individuals, or situations. Much of the work in direct inference depends on the notion of finding a suitable *reference class* [50] from which to draw conclusions. That is to say, given a number of competing pieces of evidence, which piece of evidence is the correct one to apply in the current specific situation? This is a deep problem, addressed in detail by Bacchus *et al.* [3]. My approach is to adopt some assumptions about the evidence, and then to combine the evidence using standard probability *combination rules* that make those same assumptions. Combining the evidence in this way is appealing for

its simplicity and speed, and it should lead to reasonable performance within the constraints of the assumptions.

2.4.1 An Example: When Rules Overlap

Let us see what this means in the context of our example. Say we have two first-order rules as follows.

1. With probability 0.1: $move(A, B)\ clear(A) \rightarrow ontable(A)$
2. With probability 0.3: $move(A, B)\ slippery(B) \rightarrow ontable(A)$

These rules express the knowledge that

1. In general, moving any block A incurs some small probability of dropping the block onto the table.
2. Moving any block A onto a slippery block can result in block A falling on the table.

If we are considering a `move(c,b)` action, both of these rules apply. What probability should we assign to `ontable(c)`? That is, how should we compute

$$P(\texttt{ontable(c)} \mid \texttt{move(c,b)}, \texttt{clear(c)}, \texttt{slippery(b)})?$$

Let us describe the probability distribution over next states by the dynamic Bayesian network (DBN) shown in Figure 2-2. To be fully specified, the conditional probability table in the DBN for `ontable(c)` needs to have entries for all of its rows.

The basic problem is that the conditional probability tables (CPTs) associated with the logical rules do not define a complete probability distribution in combination. That is, the rules only give us the following information:

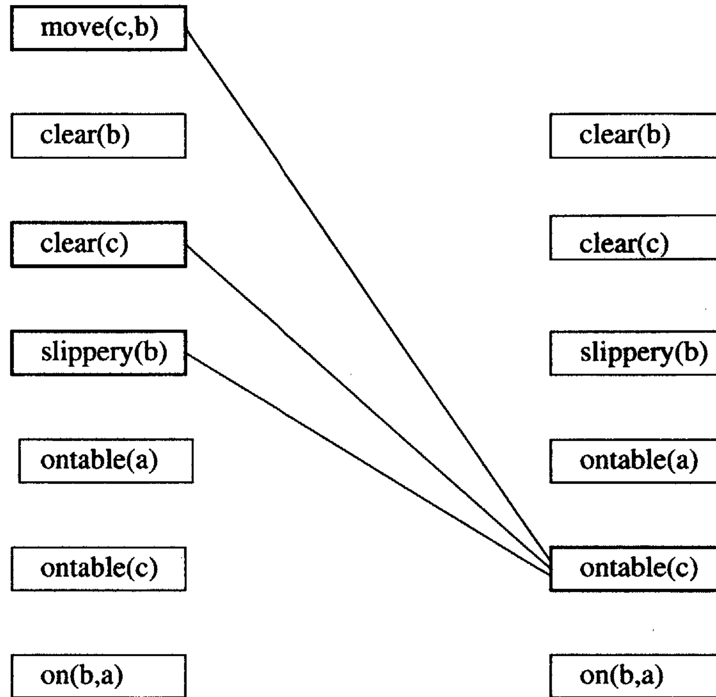


Figure 2-2: The DBN induced by the `move(c,d)` action in the example world.

1. P(`ontable(c)` | `move(c,b)`, `clear(c)`) = 0.01, from Rule 1, and

2. P(`ontable(c)` | `move(c,b)`, `slippery(b)`) = 0.03 from Rule 2.

To simplify the notation, let us refer to the random variable *ontable(c)* as O; *move(c,b)* as M; *slippery(b)* as S; and *clear(c)* as C. If *ontable(c)* ranges over `ontable(c)` and ¬`ontable(c)`, then we abbreviate it by saying O ranges over `o` and ¬`o`. And so on for the other random variables.

One way to compute the probability value we need, i.e., P(`o` | `m`, `s`, `c`) is to adopt a maximum entropy approach, and to search for the least committed joint probability distribution P(O, M, S, C) that satisfies the three constraints:

$$\begin{aligned} \alpha_1 &= P(\mathtt{o} \mid \mathtt{m}, \mathtt{c}) \\ &= P(\mathtt{o}, \mathtt{m}, \mathtt{c}) / P(\mathtt{m}, \mathtt{c}) \\ &= \sum_S P(S, \mathtt{o}, \mathtt{m}, \mathtt{c}) / \sum_{S,O} P(S, O, \mathtt{m}, \mathtt{c}) \end{aligned}$$

move(c,b)	*clear(c)*	*slippery(b)*	P(ontable(c))
¬move(c,b)	¬clear(c)	¬slip(b)	P(· | ¬move(c,b), ¬clear(c), ¬slip(b))
¬move(c,b)	¬clear(c)	slip(b)	P(· | ¬move(c,b), ¬clear(c), slip(b))
¬move(c,b)	clear(c)	¬slip(b)	P(· | ¬move(c,b), clear(c), ¬slip(b))
¬move(c,b)	clear(c)	slip(b)	P(· | ¬move(c,b), clear(c), slip(b))
move(c,b)	¬clear(c)	¬slip(b)	P(· | move(c,b), ¬clear(c), ¬slip(b))
move(c,b)	¬clear(c)	slip(b)	P(· | move(c,b), ¬clear(c), slip(b))
move(c,b)	clear(c)	¬slip(b)	P(· | move(c,b), clear(c), ¬slip(b))
move(c,b)	clear(c)	slip(b)	P(· | move(c,b), clear(c), slip(b))

Table 2.1: The conditional probability table for ontable(c).

$$\alpha_1 \sum_S P(S, \mathtt{o}, \mathtt{m}, \mathtt{c}) = \sum_{S,O} P(S, O, \mathtt{m}, \mathtt{c}), \quad (2.1)$$

$$\begin{aligned} \alpha_2 &= P(\mathtt{o} \mid \mathtt{m}, \mathtt{s}) \\ &= P(\mathtt{o}, \mathtt{m}, \mathtt{s}) / P(\mathtt{m}, \mathtt{s}) \\ &= \sum_C P(C, \mathtt{o}, \mathtt{m}, \mathtt{s}) / \sum_{C,O} P(C, O, \mathtt{m}, \mathtt{s}) \\ \alpha_2 \sum_C P(C, \mathtt{o}, \mathtt{m}, \mathtt{s}) &= \sum_{C,O} P(C, O, \mathtt{m}, \mathtt{s}), \quad \text{and} \end{aligned} \quad (2.2)$$

$$1 = \sum_{S,O,M,C} P(S, O, M, C), \quad (2.3)$$

and maximizes

$$\sum_{S,O,M,C} P(S, O, M, C) \log P(S, O, M, C). \quad (2.4)$$

Having found such a distribution, we could solve for P(o | m, s, c) by setting it equal to

$$P(\mathtt{o} \mid \mathtt{m}, \mathtt{s}, \mathtt{c}) = \frac{P(\mathtt{o}, \mathtt{m}, \mathtt{s}, \mathtt{c})}{\sum_O P(O, \mathtt{m}, \mathtt{s}, \mathtt{c})}.$$

However, the maximum entropy approach requires a maximization over 2^4 unknowns: P(o, m, s, c), P(o, m, s, ¬c), P(o, m, ¬s, ¬c), etc. In general, for a rule with n variables, we will have 2^n unknowns.

But this calculation seems like overkill: all we really want is to predict the value of P(ontable(c)) for the current situation; that is, the situation in which O=o, M=m, S=s, and C=c. Furthermore, it has been observed that maximum entropy approaches exhibit counter-intuitive results when applied to causal or temporal knowledge bases [28, 48]. If we assume we have some background knowledge, there is no need to be as non-committal as maximum entropy prescribes. In fact, a useful way to articulate domain knowledge when it comes to combining rules with under-specified CPTs is through a *combination rule.*

A combination rule is a way to go from a set of CPTs,

$$P(a|a_{11}, \cdots, a_{1n_1})$$
$$P(a|a_{21}, \cdots, a_{2n_2})$$
$$\cdots$$
$$P(a|a_{m1}, \cdots, a_{mn_m})$$

to a single combined CPT,

$$P(a|a_1, \cdots, a_n)$$

where $\{a_1, \cdots, a_n\} \subseteq \bigcup_{i=1}^{m} \{a_{i1}, \cdots, a_{in_i}\}$.

In general, a combination rule is a design choice. In this work, I used the Noisy-OR combination rule and the Dempster combination rule [53].

The Noisy-OR rule is a simple way of describing the probability for an effect given the presence or absence of its causes. It makes a very strong assumption that each causal process is independent of the other causes: the presence of any single cause is sufficient to cause the effect, and absence of the effect is due to the independent "failure" of all causes. Consider an example with four random variables, A, B, C, D (respectively ranging over a and ¬a, b and ¬b, etc.) shown in Figure 2-3. Say we want to compute $P(\mathsf{c} \mid \mathsf{a}, \mathsf{b}, \mathsf{d})$, and that we know:

$$\begin{aligned} P(\mathsf{c} \mid \mathsf{a}, \mathsf{b}) &= \alpha_1, \\ P(\mathsf{c} \mid \mathsf{d}, \mathsf{b}) &= \alpha_2, \\ P(\mathsf{c} \mid \mathsf{a}) &= \alpha_3. \end{aligned}$$

The model in Figure 2-3 describes the idea that the random variables A, B, D each influence intermediate variables, X, Y, Z: conceptually, the probability of X being "on" is α_3, the probability of Y being "on" is α_1, and the probability Z being "on" is α_2. These intermediate variables in turn influence C. Noisy-OR asserts that if any one of X,Y, or Z is "on," that is sufficient to turn C "on".[3] Noisy-OR asserts that the probability of C being "off" is equal to the probability of X,Y, and Z all independently failing to be "on." Therefore, according to Noisy-OR:

$$\begin{aligned} P(\mathsf{c} \mid \mathsf{a}, \mathsf{b}, \mathsf{d}) &= 1 - ((1 - P(\mathsf{y} \mid \mathsf{a}, \mathsf{b}))(1 - P(\mathsf{z} \mid \mathsf{b}, \mathsf{d}))(1 - P(\mathsf{x} \mid \mathsf{a}))) \\ &= 1 - ((1 - \alpha_1)(1 - \alpha_2)(1 - \alpha_3)). \end{aligned}$$

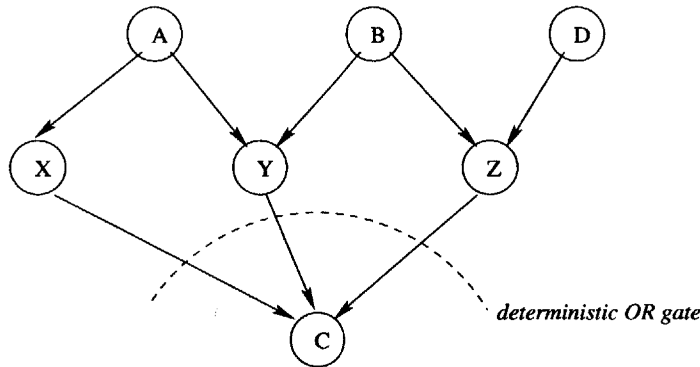


Figure 2-3: The Noisy-OR combination rule.

The Dempster combination rule assumes that the competing rules are essentially independent; that is, it asserts the sets of situations described by each rule overlap only on the current query situation. Each probabilistic rule represents the proportion of the predicted effect in its corresponding set. It has the underlying notion that the proportions of evidence are, in some sense, cumulative. Thus, if α_i is the probability

[3] In other words, "turning C on" means to assign C the value c and not $\neg$c.

that rule i gives to P(c), then the Dempster rule states:

$$P(\mathsf{c} \mid \mathsf{a}, \mathsf{b}, \mathsf{d}) = \frac{\prod_i \alpha_i}{\prod_i \alpha_i + \prod_i (1 - \alpha_i)}.$$

Both combination rules make fairly strong independence assumptions about the competing causes. The Dempster rule assumes an additive quality about the evidence (which seems appropriate in the case where we search for high-probability rules about action effects), but the Noisy-OR rule is simpler to compute. That being said, there are any number of ways to go from a set of rules to a single probability (e.g., choosing the most specific rule, computing a consistent maximum entropy distribution, etc.); in general, combination rules are ad hoc and assume particular characteristics about the domain. Combining evidence in the general case is truly hard; the reader is again referred to Bacchus *et al.* [3] for the analysis and discussion of a more general approach.

2.5 Related Work in Probabilistic Logic

Bayesian networks are well known for their ability to manage probabilities associated with world knowledge, but they generally handle only propositional knowledge. Extending relational, or first-order, representations so that they can be managed probabilistically is therefore an important area of study. Ground has been broken in this area by Ngo and Haddawy [45], Koller and Pfeffer [35], Jaeger [29], Russell and Pasula [47], Lukasiewicz [37]. Kersting and De Raedt provide a nice survey of these approaches [32].

The probabilistic relational models (PRMs) of Koller and Pfeffer, inspiring the subsequent work by Pasula and Russell, bring in elements from object-oriented databases and couch the qualitative state information in terms of the entity/relationship model from that literature, instead of in terms of logical clauses. The PRM framework pro-

vides a mechanism for combining CPTs in the form of "aggregate functions," which again are provided by the designer. In contrast, the work of Lukasiewicz on probabilistic logic programming gets around the question of combination rules by specifying that the CPTs should be combined in a way that produces the combined probability distribution with maximum entropy.

Kersting and De Raedt give an interesting synthesis of these first three approaches and provide a unifying framework, called "Bayesian Logic Programs." A Bayesian Logic program consists of two parts: a set of logical clauses qualitatively describing the world, and a set of conditional probability tables (CPTs), plus a combination rule, encoding quantitative information over those clauses. The way logical queries are answered in this framework is by building up a tree structure that represents the logical proof of the query; the probabilistic computations are propagated along the tree according to the specified combination rule.

I adopt a rule-based approach, rather than the Bayesian Logic or PRM approach, because of the need to express temporal effects of actions. PRMs provide a rich syntax for expressing uncertainty about complex structures; however, the structure I need to capture is dynamic in nature. It is not immediately clear if the extension from static PRMs to dynamic PRMs is as straightforward as the DBN case. More fundamentally, PRMs require a fully described probability distribution. That is, that every row in the conditional probability tables for every slot must be filled in. For the purposes of modeling the immediate effect of a chosen action, however, such detail is hard to come by and may not be actually needed (see discussion above). For these reasons, the simpler approach of probabilistic rules in conjunction with a combination rule seems more appropriate for modeling the effects of actions in a relational setting.

2.6 Related Work in Policy Learning for Blocks World

Apart from the traditional planning work in blocks worlds, there is an interesting line of work in directly learning policies for worlds not represented in a first-order language.

Martín and Geffner [39], and Yoon *et al.* [57] use concept languages to find generalized policies for blocks world. They build on the work of Khardon [33], who first learned generalized policies for blocks world within a first-order context. The basic approach is to start with a set of solved examples (usually obtained by solving a small instance of the target domain with a standard planning algorithm, called the *teacher*). Then, a search through the language space is conducted to find rules which best cover the examples. Khardon's approach is to enumerate all the rules possible within the language, and then to use a version of Rivest's algorithm for learning decision lists [52] to greedily select out the "best" rule, one by one, until all the training examples are covered. The learned policy, then, is articulated in the form of a decision list: the current state is compared to each condition in the list, and the action specified by the first matching rule is taken. The resulting policy is able to solve instances of the problem that the teacher is unable to solve.

Martín and Geffner expand on Khardon's work by moving from a first-order language to a concept language [9, 10, 44]. They notice that the success of Khardon's approach hinges on some intermediate support predicates that had to be coded into the representation by hand. For example, Khardon includes a predicate that indicates whether a block is a member of the set of blocks that are *inplace*; that is, blocks that have been correctly placed so far, given the goal specification. They reason that the ability of a concept language to express knowledge of *sets* or *classes* of objects would make it possible to learn concepts like *inplace* automatically.

Yoon *et al.* take the work further in two steps. First, they use a greedy heuristic

search to find the “best” rules, rather exhaustively enumerating of all the rules in the language space first. Second, they use *bagging* [11] to assemble a set of of decision-list policies (each trained on a subset of the training examples). The ensemble then chooses the action at each time step by voting.

Baum [5, 6] uses evolutionary methods to learn policies for a propositional blocks world. Each individual that is evolved represents a condition-action pair. Then, each individual learns how much its action is worth, when its condition holds, using temporal difference learning. Baum calls this the individual’s *bid*. When it comes time to choose an action, the applicable individuals bid for the privilege of acting; the individual with the highest bid will get to execute its action. The learning of bids is described as an economic system, with strong pressure on each individual to learn an accurate value for itself. Individuals of low worth are eliminated from the pool, and evolutionary methods are applied to the individuals that succeed. What’s interesting about this approach is that the learning, of both the rule conditions, and of the action values, is completely autonomous.

The above work represents successful instances of planning in blocks world: these systems aim to find policy lists that prescribe an action given certain conditions. The approach taken here is different, however. I am interested in modeling the dynamics of a relational domain (rather learning a policy ahead of time). Accordingly, I will pursue the approach of using first-order rules to probabilistically describe the effects of actions.

Chapter 3

Reasoning System: Theoretical Basis

We want to represent the world as a set of first-order relations over symbolic objects in order to learn a definition of the world's dynamics that is independent of the specific domain. The first-order rules in our system describe knowledge in the abstract, like a *skeleton* in the PRM sense [26, 25]. To use abstract knowledge for reasoning about concrete world situations, we take the approach of *direct inference* [50]. That is to say, we ground the rules' symbols by resolving the rules against the current state, and then we make conclusions about the world directly from the grounded rules. In the rest of this document, the use of italic font in a rule denotes abstract, world-independent knowledge, and fixed-width font denotes ground elements.

Let us define a relational, or first-order, MDP as follows. The definition is based on the PSTRIPS MDP as defined by Yoon *et al.* [57].

A relational MDP has a finite set of states and actions. The actions change the state of the world stochastically, according to the probability distribution $P(s, a, s')$. There is a distribution $R(s, a)$ that associates each state and action pair stochastically with the immediate real-valued reward.

- *States*: The set of states is defined in terms of a finite set S of predicate symbols, representing the properties (if unary) and relations (if n-ary) among the domain objects. The set of domain objects is finite, as well.
- *Actions*: The set of actions is defined in terms of action symbols, which are analogous to the predicate symbols. The total number of actions depends on the cardinality (that is, the number of arguments) of each action and the number of objects in the world.
- *Probability and reward distributions*: For each action, we construct a dynamic Bayesian network [16] that defines the probability distribution over next states and rewards.

3.1 Probabilistic Inference with Incomplete Information

Given a set of rules about how the agent's actions affect the world, we would like to turn them into a usable world model. We want to pull out those rules that apply to the current situation and use them to generate, with appropriate likelihood, the expected next situation.

To continue with our example, a first-order MDP description of the world in Figure 2-1 would be as follows.

- Set of predicate symbols:

	Arity	Description	Example
clear	unary	Whether a block is clear.	`clear(a)`
slippery	unary	Whether a block is slippery.	`slippery(a)`
ontable	unary	Whether a block is on the table.	`ontable(a)`
on	binary	Whether a block is on top of another.	`on(a,b)`
nextto	binary	Whether a block is beside another.	`nextto(a,b)`

- Set of action symbols:

	Arity	Description	Example
move	binary	Move a block onto the top of another.	`move(a,b)`

- The probability distribution over next states, P, and over next reward, R, can be represented with a dynamic Bayesian network (DBN) for each action. Say we are considering the the `move(a,b)` action that the applicable rules are

  1. With probability 0.1: $move(A, B)\ clear(A) \rightarrow ontable(A)$, and
  2. With probability 0.3: $move(A, B)\ slippery(B) \rightarrow ontable(A)$

It should be immediately clear that the conditional probability tables on each arc of the DBN are incomplete, as we described in the previous chapter. What should the distribution be over `clear(a)`, `on(a,b)`? What should it be for the state elements with no rules? What should the value of the reward be?

We need a precise definition of what to do when:

1. More than one rule influences a subsequent state element, or
2. No rule influences a subsequent state element.

Both of these problems result from incomplete information. The solution for the first item lies in finding a distribution that is consistent with the information available. The solution to the second problem additionally requires making some assumptions about the nature of the domain.

An appropriate way to express such domain knowledge is through the use of a combination rule, as described in the previous chapter. The system assumes a relatively static world and that the state elements may take on any number of discrete values (up until now, our examples have been with binary-valued state elements). Thus, we adopt the following approach:

1. If there is no information about a particular dimension, the value will not change from one state to the next, except for some small "leak" probability with which the dimension might take on a value randomly. In the experiments that follow, the "leak" probability was 0.01.

2. If there is more than one rule with influence on a particular value for a state element, their probabilities are combined via a combination rule. This yields the probability of generating that value.

Now, here comes a crucial assumption. For a given state element, the system takes all of the pieces of evidence for a value and normalizes their probabilities. This means, for example, that if the applicable rules only mention one particular value, then after normalizing that value gets a probability of one. It a value is not mentioned by any rule, it gets a probability of zero.[1] Knowledge about how the rules are acquired is important here. For example, if we do not have a rule about a particular situation, and we can safely assume that a lack of a rule implies a lack of meaningful regularity—

[1]Perhaps a better alternative would have been to divide the remaining mass evenly among the remaining possible values; however, this puts one into the awkward position of having to articulate all the possible (and potentially never-achieved) values of a perceptual dimension. The system does allow, however, for the user to explicitly supply an "alternative" value – if such a value is available, it can be generated with the leak probability of ϵ, and the rest of the probability mass is normalized to $1 - \epsilon$. In general, however, such "alternate" values are unknown.

then, in this case, it is probably fine not to consider values that do not appear explicitly in any of the rules for the situation in question. The dimension's next value, then, is generated by choosing a value according to the normalized probability distribution.

3.2 Using Probabilistic Relational Rules as a Generative Model

The relational generative model consists of a set of relational rules, represented as Horn clauses, called the *rule-base.* Optionally, it can contain a set of background facts, called the *bgKB.* When queried with a proposed action and a current state, the model outputs a sampled next state and the estimated immediate reward.

Here are the steps for generating the next state and reward:

1. Turn the action and current observation into a set of relational predicates. Call it the *stateKB.* If background knowledge exists, concatenate the bgKB to the stateKB to make a new, larger, stateKB.
2. Compare each rule in the rule-base against each predicate in the stateKB. Given a particular rule, if there is a way to bind its variables such that the rule's antecedents and action unify against the stateKB (i.e., if the rule can be said to *apply* to the current state), then:
   (a) Add the rule, along with all the legal bindings, to the list of *matching rules.*
   (b) "Instantiate" the rule's consequent (i.e., right-hand side) according to each possible binding. This generates a list of one or more instantiated consequences, where each instantiation represents evidence for a particular value for a state element. Associate each instantiation with the probability recorded for the rule.

3. For each state dimension, look through the list of matching rules for instantiations that give evidence for a value for this dimension. Collect the evidence for each value.

    (a) If there is more than one piece of evidence for a value, then compute a probability for the value according to the combination rule.

    (b) Normalize the probabilities associated with each value, so that the probabilities for each value sum to one.

    (c) Generate a sampled next value for this state dimension according to the normalized probabilities.

4. For the reward, look through the list of matching rules for instantiations that give evidence for a reward value. Compute a weighted average across all the reward values.

3.2.1 The Sparse-Sampling Planner

The generative world models produced by the above rules are by necessity stochastic and incomplete; whatever planning algorithm is used must take this uncertainty into consideration. To this end, I implemented the sparse-sampling algorithm by Kearns, Mansour and Ng [31]. In contrast to traditional algorithms where the search examines all the states generated by all the actions, the sparse sampler uses a look-ahead tree to estimate the value of each action by sampling randomly among the next-states generated by the action. As with all planning algorithms, this algorithm has the disadvantage of scaling exponentially with the horizon; but, the sparse-sampling trades that off with the advantage of not having to grow with the size of the state space.

The algorithm's look-ahead tree has two main parameters: the horizon and the width. These parameters depend on the discount factor, the amount of maximum reward, and the desired approximation to optimality. Kearns *et al.* provide guaranteed bounds on

the required horizon and width to achieve the desired degree of accuracy. However, with a discount factor of 0.9, a maximum reward of 1.0, and a tolerance of 0.1, for example, the computed parameters are a horizon of 80 and a width of $2,147,483,647$. Such numbers are, unfortunately, not very practical. As a result, the authors offer a number of suggestions for the use of the algorithm in practice, such as iterative deepening, or standard pruning methods.

However, there is unavoidable computational explosion with the depth of the horizon. It becomes especially acute if the action set size is anything but trivial (i.e., less than 5 actions). Although it is true that the number of state samples required is bounded (by the *width* parameter; in other words, the number of states that are sampled does not grow with the size of the state space) in order to estimate each sample's value, we need to find the maximum Q-value for the sampled state. This involves trying out each action at each level of depth in the tree, resulting in a potentially huge fan-out at each level. While it is possible to try to keep the action set small, it is not clear what solutions are available if one genuinely has a large action set, however. In the case of blocks world, for example, a large number of blocks produces a huge number of $move(A, B)$-type actions; in fact, the number of actions grows quadratically with the number of blocks.

This system does not try to prune the action explosion at all, although this is clearly desirable. The width and depth are set to rather arbitrary small values; the idea is to allow some degree of robustness, while at the same time permitting manageable computation time.

3.2.2 CLAUDIEN: Learning Probabilistic Rules

The CLAUDIEN system is an inductive logic engine for "clausal discovery" [15, 14]. It discovers clausal regularities in unclassified relational data. CLAUDIEN operates by a general-to-specific search to find a hypothesis set of clauses that best characterize the

presented data. It assumes a closed world – predicates that are not explicitly listed as true are assumed to be false.

CLAUDIEN is a batch system that takes as input a set of training examples. It then "data mines" the training set for the most compact set of Horn-clause rules that entail the examples in the set. Any variables in the rules are implicitly universally quantified.

The search space is specified by the user in terms of declarative bias [17]. This bias, which describes the set of clauses that may appear in a hypothesis, is provided according to the rules in a formalism called DLAB. A DLAB grammar is a set of templates that tells CLAUDIEN what the characterizing regularities should look like. Getting this bias right is absolutely crucial: CLAUDIEN will not find anything outside of the described search space; but, a space that is too large results in an intolerably long search.

The main user-definable parameters are called *accuracy* and *coverage*. Coverage specifies the minimum number of examples a potential rule must "explain" in order to be considered. Accuracy specifies the minimum proportion of the examples explained (i.e., examples in which the pre-conditions apply) by the rule, to which the rule's post-conditions must also apply. The choice of parameters was somewhat arbitrary; I wanted to find rules that explained a non-trivial number of cases, and that explained them with high probability. Thus, coverage was set to 10 and accuracy was set to 0.9.

Because of its data-mining nature, CLAUDIEN seemed like the right approach for uncovering regularities about how actions affect the world. Other inductive logic programming systems, such as FOIL, are designed for classification problems, with positive and negative examples. Learning about the effects of actions seems more easily expressed as a data-mining problem than as a classification problem.

Chapter 4

Experiments

There were two main parts to the experiments. The first part examines how well a model-based system could perform compared to a value-based reinforcement learning system in a relational domain. To that end, I implemented a small, deterministic blocks world (see Figure 4-2) in order to draw some comparisons to the RRL system. The second part moves into a stochastic domain to truly exploit the probabilistic nature of the rules.

The main learning problem is in the acquisition of a set of first-order rules about the world dynamics—the world model. Resolving such a set of rules against the current observation and proposed action produces a set of grounded rules; these ground rules can be used to partially specify a DBN describing the effect of the action. We call this step *applying* the model to the world. At each step, to decide its next action, the agent invokes the planner. The planner applies the model to the current state, and successively to each predicted next-state, to ultimately produce an estimated Q-value for each action. The action chosen is the one with the highest estimated value; if there is more than one action at the highest Q-value, the planner chooses randomly among them.

4.1 A Deterministic Blocks World

In order to be able to draw comparisons to the RRL results, I used a blocks-world setup as close as possible to the one in the RRL experiments [21], although there are some slight differences which will be duly noted. The task in the RRL experiment to move block a onto block b. The domain is shown in Figure 4-1.

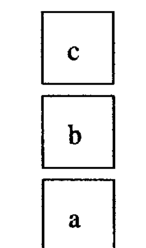


Figure 4-1: The RRL blocks world. The goal is to achieve `on(a,b)`.

For comparison, here is the description of the blocks world used by the RRL system:

- State predicates:

	Arity	Description	Example
clear	unary	Whether a block is clear.	`clear(a)`
on	binary	Whether a block is on top of another block or on top of the table.	`on(a,b)` `on(a,table)`
goal	unary	Specifies the goal predicate.	`goal(on(a,b))`

- Action symbol:

	Arity	Description	Example
move	binary	Move a block onto the top of another. Can only be executed if the appropriate pre-conditions are met.	`move(a,b)`

- Action preconditions: allow *move(A,B)* if $A \neq B$ and
  $A \in$`{r,b,g}`, $B \in$ `{r,b,g,table}`

- Reward: The reward is 1.0 if block **a** is successfully moved onto block **b**, and 0.0 otherwise.

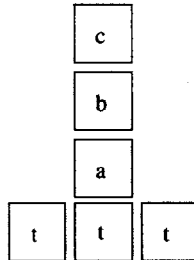


Figure 4-2: The deterministic world used by our system. The table is made of three *table* blocks, which may be subject to *move* attempts but are actually fixed.

Here is the description of the deterministic blocks world used by our system:

- State predicates

	Arity	Description	Example
clear	binary	Whether a block is clear.	`clear(a,false)`
on	binary	Whether a block is on top of another.	`on(a,t)`
goal	unary	Specifies the goal predicate.	`goal(on(a,b))`

- Action Symbol

  The *move* action can be executed on any pair of blocks (even if both arguments refer to the same block, or if the first block refers to a table block).

	Arity	Description	Example
move	binary	Move a block onto the top of another.	`move(a,b)`

- Reward: The reward is 1.0 if the agent successfully moves block *a* onto block *b*, and 0.0 otherwise.

As in the RRL setup, the agent observes the properties of the three colored blocks, as well as the goal. This yields an observation space of size 513. One difference from the RRL system is that the above observation vector includes `clear(b,false)`, whereas in the RRL system if b were not clear then `clear(b)` would simply be absent. An example observation vector for our deterministic planner looks like this:

```
on(c,b).
clear(c,true).
on(b,a).
clear(b,false).
on(a,t)
clear(a,false).
goal(on(a,b)).
```

The reason for predicates like `clear(c,false)` and not like `on(c,a,false)` is that our system does not infer that the absence of `clear(c)` implies that `clear(c)` is false; in other words, it does not make a closed world assumption. Because pre-conditions are not built into the action specifications, knowing explicitly that a block is not clear is needed for estimating the success of a possible move action.

As in the RRL setup, the agent's action space consists of a *move* action with two arguments: the first argument indicates which block is to be moved, and the second argument indicates the block onto which the first one is to be moved. The action only succeeds if both of the blocks in question are clear; that is, they have no other blocks on top of them. Because the agent can consider all combinations of the six blocks as arguments to the *move* action, this yields 36 actions. In contrast, the RRL system only allows *move(A,B)* where $A \neq B$ and $A \in \{\texttt{a,b,c}\}$, $B \in \{\texttt{a,b,c,t}\}$, a total of nine actions. Additionally, the table is three blocks wide, and the agent may attempt to pick up the blocks that make up the table. In the RRL configuration, by contrast, the table is an infinitely long surface and is not subject to any picking up attempts.

4.1.1 Acquiring the Set of Rules

To acquire a set of rules, we must collect a set of training examples for CLAUDIEN. For this, a random agent was set loose in the blocks world for 1,500 steps. The search parameters were set with coverage equal to 10, and accuracy equal to 0.9. Finally, the search space was defined. Figure 4-3 shows the template. This template describes rules about the conditions under which actions either produce a certain reward or affect a single state predicate.

Each step taken by the agent resulted in a training example, which consisted of the current observation (denoted by a `pre` predicate), the goal predicate,[1] the executed action, the resulting observation (denoted by a `post` predicate), and the resulting reward. Here, a typical training example might look like this:

```
begin(model(0)).
goal(on(a,b)).
action(move(t,t)).
pre(on(a,t)).
pre(cl(a,false)).
pre(on(b,a)).
pre(cl(b,false)).
pre(on(c,b)).
pre(cl(c,true)).
post(on(a,t)).
post(cl(a,false)).
post(on(b,a)).
post(cl(b,false)).
post(on(c,b)).
post(cl(c,true)).
reward(0.0).
end(model(0)).
```

After four days, CLAUDIEN returned with 147 rules, shown in appendix A. It is interesting to note that many of the rules learned by CLAUDIEN try to articulate

[1]Some tasks, such as "build the tallest stack you can," are not neatly described by a single, first-order, goal predicate. In such a case, a goal predicate would not be specified.

```
dlab_template('1-1:[1-1:[reward(1.0)],
                    1-1:[reward(0.0)],
                    1-1:[post(on(anyblock,anyblock)),post(cl(anyblock,anytruth)),
                      post(on(D,E)),post(cl(D,anytruth)),post(cl(E,anytruth))] ]
                <--
                3-3:[0-len:[pre(on(a,buta)),pre(on(b,butb)),pre(on(c,butc)),
                     pre(cl(a,anytruth)),pre(cl(b,anytruth)),pre(cl(t,anytruth)),
                     pre(cl(c,anytruth)),pre(cl(D,anytruth)),pre(cl(E,anytruth))],
                     1-1:[action(move(anyblock,anyblock)),action(move(D,E))],
                     0-1:[goal(on(D,E))] ]').

dlab_variable(anytruth,1-1,[true,false]).
dlab_variable(anyblock,1-1,[a,b,t,c]).
dlab_variable(butb,1-1,[a,t,c]).
dlab_variable(buta,1-1,[b,t,c]).
dlab_variable(butc,1-1,[a,b,t]).
```

Figure 4-3: The DLAB template for the deterministic blocks world. This template describes rules for how actions, in conjunction with pre-conditions and/or a goal predicate, either produce a certain reward (either 1.0 or 0.0) or influence a single state predicate. In other words, it describes a mapping from actions, state elements, and the goal predicate into a reward value of 1.0, a reward value of 0.0, or the post-state of a state element. The expression `1-1` before a set means that exactly one element of the set must appear in the rule; similarly, `3-3` means exactly three elements, `0-len` means zero or more, etc.

the pre-conditions for the *move* action as specified in the RRL paper. For example, browsing the rules, one can see that CLAUDIEN imposed $A \neq B$ by making lots of rules that partially instantiate the variables[2]:

```
27: [0.909091] cl(E,true) on(b,t) goal(on(D,E)) move(a,b)-> 1.0
```

This rule means that, with accuracy of 1.0, when E is clear, `b` is on the table,[3] and the goal is $on(D, E)$, then moving `a` onto `b` results in success.

Similarly, without a way to express equality of blocks, CLAUDIEN enumerated the idea that, regardless of preconditions, trying to move the same block onto itself or trying to move the table anywhere resulted in no reward, and in no change of state. For

[2]Recall that the goal predicate is `on(a,b)`, and that the agent receives a reward of 1.0 for success (achieving the goal) and 0.0 otherwise.

[3]Which, presumably, correlates with `b` being clear, since `b` can only have been moved to the table once `c` was no longer on it.

example:

```
0: [1.0] move(a,a)-> 0.0
4: [1.0] move(b,b)-> 0.0
10: [1.0] move(t,c)-> 0.0
14: [1.0] move(c,c)-> 0.0
28: [1.0] on(a,c) move(t,c)-> on(a,c)
30: [1.0] on(a,c) move(t,t)-> on(a,c)
```

Many of the rules, especially the ones that predict reward of 1.0 (goal-achievement), appear to be redundant. When you consider that there are only three blocks on a table that's only three-blocks long, many of the rules are just slightly different ways of saying the same thing. For example:

```
16: [1.0] cl(a,true) on(c,t) move(a,b)-> 1.0
20: [1.0] cl(b,true) cl(a,true) move(a,b)-> 1.0
```

In general, the rule set is not as compact as it could be. However, it does appear to cover the space of action effects fairly completely. Because so many of the attempted actions result in no state change and a reward of 0.0 (because they are either trying to move a non-clear block, or are of the form *move(A,A)*), there are lots of rules about the world staying the same.

4.1.2 Deterministic Planning Experiment

Once the rules were learned, I ran an experiment with the planning parameters horizon= 3, width= 1, discount factor= 0.9, and a fixed random seed. There is no exploration; at each step the agent runs the planner then executes the suggested best action. If there is more than one action at the highest Q-value, we choose randomly among them. After executing the action, we perceive the resulting configuration of blocks and run the planner to choose the next action. Below is the trial output from the experiment.

In step one, the agent correctly considers the `move(c,t)` action— the action is listed multiple times because the three table blocks are considered in turn.
(The `move(a,c)` and `move(c,c)` actions appear incorrectly due to a minor bug in the unification mechanism; this mistake was fixed for the later experiments.)

```
[t]
[t][a][b][c]
[t]


-- Randomly choosing between 5 at value=1.968:
[MOVE(c,t)] [MOVE(c,t)] [MOVE(c,t)]
[MOVE(c,c)] [MOVE(a,c)]
1 [MOVE(c,t)], r: 0.0

[t]
[t][a][b]
[t][c]
```

In step two, the the `move(b,t)` action appears to produce no result. This is in fact due to the way the blocks-world simulator responds to a request for a "table-colored" block: because there is more than one, it returns one randomly. In this case, it returned one of the two table blocks already under a or c, resulting in a failed move.

```
-- Randomly choosing between 3 at value=2.373:
[MOVE(b,t)] [MOVE(b,t)] [MOVE(b,t)]
2 [MOVE(b,t)], r: 0.0

[t]
[t][a][b]
[t][c]
```

In step three, it correctly attempts again to move b onto the table.

```
-- Randomly choosing between 3 at value=2.373:
[MOVE(b,t)] [MOVE(b,t)] [MOVE(b,t)]
3 [MOVE(b,t)], r: 0.0

[t][b]
[t][a]
[t][c]
```

In step four, there is only one best action: a is correctly moved onto b.

```
4 [MOVE(a,b)], r: 1.0

[t][b][a]
[t]
[t][c]
*** Solved at 4
```

Figure 4-4: Sample output from the planning experiment in the deterministic small blocks world.

In general, the planning experiment was successful. It turns out that describing the table as a composition of blocks produced some awkward behavior. For example, in the first step of the trace shown in Figure 4.1.2, the *move(b,t)* action produces no result. This is in fact due to the way the blocks world responds to a request for a "table-colored" block: because there is more than one, it returns one randomly. In this case, the randomly chosen block was either the one under block a or under block c, meaning that an attempt to move the block b onto that table block would fail. The RRL blocks world, by contrast, would return a clear space on the table in every case.

The aim of this experiment was to show that a learned set of stochastic rules could enable an agent, in a deterministic blocks world, to execute a policy that had similar success to that of an agent with a logical regression tree and action pre-conditions. In this sense, the experiment turned out satisfactorily. One complaint is that the size of the rule-set seemed larger than necessary—both by including redundant rules and by including lots of rules about world stasis. Largely, however, this appeared to be a result of some awkward representational choices. The next domain smoothed out some of the representational idiosyncrasies, as well as incorporating some non-determinism.

4.2 A Stochastic World with Complex Dynamics

Given the relative success of the planning approach in the deterministic blocks world, the next step was to move into a more realistic stochastic blocks world. The aim, however, was not to make the domain stochastic arbitrarily; rather, the world should be stochastic in such a way that, with appropriate action choices, the agent can actively *do* something to improve its chances of success.

In that spirit, I considered a task in which the agent should make as tall a stack of blocks as possible. The initial configuration of blocks is shown in Figure 4-5. As

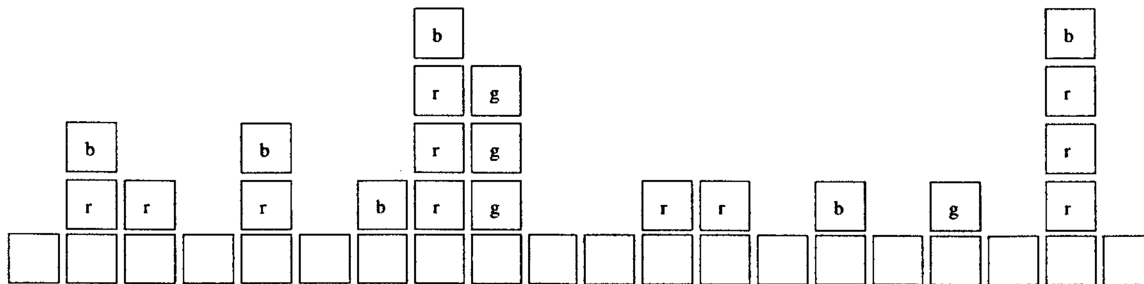


Figure 4-5: The large stochastic blocks world. The table is made up of fixed, indistinguishable table blocks that *may not* be the subject of a move action.

before, the world consists of blue, green, and red blocks. The table is again a surface made up of fixed blocks; this time, however, a request for a table block returns a clear table block randomly.

This world has a fairly high amount of stochasticity, mainly resulting from the fact that stacks of blocks can fall down with some probability. Whether a block falls off the top of a stack is influenced by a number of factors:

1. The block's *width*. Blue blocks are the widest, followed by red and then green. A block has a greater chance of falling the wider and/or higher up it is.

2. The difference between the block's height and the height of its neighboring stack(s). This is supposed to capture the idea that tall stacks can be "buttressed" and supported by adjoining stacks. If a stack towers over its supporting neighbors, it becomes more unstable than if surrounded by stacks of similar height.

3. The stability of the neighboring stacks, in turn. If the supporting stacks themselves are not well buttressed, then they cannot lend much support to the original stack.

4. Where a block lands, should it fall, is also stochastic and depends on how high on the stack it was. Blocks may also fall off the table. If they fall off the table, they cannot be recovered.

Specifically, the dynamics work as follows. Every time a block is moved onto a stack, we topple it according to the computed probability. If the top block falls, we repeat the process for the next block. Thus, it is possible that a stack may entirely collapse.

The topple probability for a block b is computed as:

$$P[b \text{ falls}] = \frac{1}{1 + \exp\frac{(I_b)^2 - K}{2K}} \; .$$

The parameter K is simply a knob for increasing or decreasing the probability of toppling. For these experiments, its value was 350. The block's instability, I_b, is computed from its leftward and rightward instabilities:

$$I_b = RI_b + LI_b \; .$$

Each directional instability is a recursive function of the distance down to the top block in the stack next to it,[4] the block's *width*,[5] the instability of the top neighbor block itself. The recursive calculation ends when we have either reached the end of the table, or there are no more neighbor blocks. To calculate, e.g., the left instability:

$$LI_b = (dist(b, b_l) \times width(b)) \times \gamma \frac{I_{b_l}}{I_{b_l} + (dist(b, b_l) \times width(b))}$$

The discount, γ, encodes the idea that support from buttressing stacks tapers off with distance. It was set to 0.9 for these experiments.

The landing position is computed as:

[4] Let's call this block b_l or b_r, depending on the direction (left or right). If there is no neighboring stack in that direction, this distance, $dist(b, b_l)$, reduces to the block b's height; if the neighboring stack is higher, this distance is zero.

[5] The *width* is 1 for green blocks, 2 for red blocks, 3 for blue blocks.

$$new_position(b) = position(b) + (whichSide) \times P[b \text{ falls}] \times tableSize/2;$$

Basically, how far a block falls from its original location is a function of how high and unsteady it originally was. The parameter *whichSide* is set to -1 or 1 randomly. If the new position is beyond the edge of the table, and if falling off the table is not allowed, then the block will just "land" on the last position in the table.

A description of the state and action predicates is next:

- State predicates:

	Arity	Description	Example
clear	binary	Whether a block is clear.	`clear(a,true)`
on	binary	Whether a block is on top of another.	`on(a,b)`
color	binary	Specifies the color of a block.	`color(b,red)`
tallest	unary	Specifies the highest stack's position.	`tallest(12)`
position	binary	Specifies the block's column position.[6]	`position(b,4)`

- Action symbol:

  The *move* action takes two arguments: the block to be moved, and the block onto which it should go. The action succeeds if both blocks are clear, and fails if one of them is covered. The second argument may be a table block; however, the first argument can only be one of the movable non-table blocks. For the world shown, with 21 non-table blocks, this results in 21 actions per block (i.e., 20 actions with the other non-table blocks as the second arguments, and 1 action with the table as the second argument.), or 441 total actions.

	Arity	Description	Example
move	binary	Move a block onto the top of another.	`move(a,b)`

[6]This lets a block be identified as being in the tallest stack, or on a buttressing stack, etc.

- Reward: The reward at each step, if a successful action is executed, is equal to the height of the highest resulting stack. If the executed action failed, then a reward of 0.0 is given. So if a stack of height 10 exists, and the robot puts a block onto a different, shorter stack, then the robot gets a reward of 10. This is to encourage tall-stack building as well as buttress-stack building.

4.2.1 Acquiring the Rules for the Stochastic Stacking Task

The DLAB template for this experiment is shown in Figure 4-6. The idea behind this template is for the agent to learn:

1. What causes a reward of zero: when will an action fail.
2. What causes a non-zero reward. A higher reward corresponds to building up a stable stack.
3. How the various state elements change as a result of the *move* action.

To acquire the rules, an agent taking random actions was run for 1,500 steps. A typical training example contained 90 predicates and looked like this:

```
begin(model(0)).
action(move(n39,n20)).
pre (tallest(7)).
pre (on(n20,n01)).
pre (cl(n20,false)).
pre (color(n20,r)).
pre (pos(n20,1)).
pre (on(n21,n20)).
...
post (cl(n40,true)).
post (color(n40,b)).
post (pos(n40,18)).
reward(0.0).
end(model(0)).
```

```
dlab_template(' 1-1:[ 1-1:[reward(Z)],
                  1-1:[reward(0)],
                  1-1:[reward(num)],
                  1-1:[post(tallest(Z)), post(tallest(num)),
                       post(color(W, anycolor)), post(color(X, C)),
                       post(pos(V, N)), post(on(D, E)),
                       post(cl(D, anytruth)), post(cl(E, anytruth))] ]
                <--
                2-2 :[0-len:[pre(tallest(Y)), pre(tallest(num)),
                       pre(color(D, anycolor)), pre(color(D, C)),
                       pre(color(E, anycolor)), pre(color(E, B)),
                       pre(pos(D, N)), pre(pos(E, M)), pre(on(D, E)),
                       pre(cl(D, anytruth)), pre(cl(E, anytruth))],
                     1-1:[action(move(D, E))] ] ').

dlab_variable(anytruth, 1-1, [true, false]) .
dlab_variable(anycolor, 1-1, [g, b, t, r]) .
dlab_variable(num, 1-1, [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]).
```

Figure 4-6: The DLAB template, without background knowledge, used in the complex block-stacking task in the large stochastic world.

```
dlab_template('1-1:[ 1-1:[reward(Z)],
     1-1:[reward(num)],
     1-1:[post(tallest(Z)), post(tallest(num)),
                       post(color(W, anycolor)), post(color(X, C)),
                       post(pos(V, N)), post(on(D, E)),
                       post(cl(D, anytruth)), post(cl(E, anytruth))] ]
                <--
                2-2 :[0-len:[pre(tallest(Y)), pre(tallest(num)),
                       pre(color(D, anycolor)), pre(color(D, C)),
                       pre(color(E, anycolor)), pre(color(E, B)),
                       pre(pos(D, N)), pre(pos(E, M)), pre(on(D, E)),
                       pre(cl(D, anytruth)), pre(cl(E, anytruth)),
                       nextto(Y, J), nextto(Z, K),
                       nextto(E, H)],
                     1-1:[action(move(D, E))] ]').

dlab_variable(anytruth, 1-1, [true, false]).
dlab_variable(anycolor, 1-1, [g, b, t, r]).
dlab_variable(num, 1-1, [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]).
```

Figure 4-7: The DLAB template, with background knowledge about *next-to* used in the complex block-stacking task.

After some initial experiments, it was clear that the notion of a stack being "next to" another stack, crucial for the buttressing idea, was missing from the hypothesis space. So, background knowledge of a pairwise *next-to* predicate was included along with the training samples. This file contained a list of facts of the form:

`nextto(0,1)`, `nextto(1,2)` ... `nextto(19,20)`
`nextto(1,0)`, `nextto(2,1)` ...

The addition of these facts, however, apart from creating an overwhelmingly large hypothesis space, was not able to produce rules sufficiently expressive for good performance in the task.

The buttressing idea requires not only that you build onto a stack next to the tallest stack, but also onto the stack next to that, etc, for however many stacks there might be in the set of buttressing stacks. The first-order language that we are using is not equipped to talk about arbitrarily sized sets of things compactly. Its restriction to describing chains of relations by enumerating them is a real limitation in this case.

There are other obvious shortcomings to the hypothesis space as described above in the DLAB templates. For example, because the reward received depends on the height of the tallest stack, some information about the tallest stack's height should probably be included. Recall that CLAUDIEN can only learn Horn clauses; this means that it cannot learn a single rule that expresses the relationship between the reward *and* the resulting height of the stack; it must learn some relationship between the previous height and the reward, which may require further background knowledge about arithmetic.

It is clear that a standard first-order language was insufficient to describe the complex dynamics of this particular blocks world. What other languages would be better suited? For now, however, we bypass some of the problems posed by the language limitations by moving to a simpler world described below.

4.3 A Stochastic World with Simpler Dynamics

In this domain, the task is the same as before: make as tall a stack as possible. The number of blocks and initial configuration is also the same (see Figure 4-5). The difference is that, now, whether a block falls depends only on its color and the color of the block below it. Furthermore, it is not possible for entire stacks to fall down—only the block that was moved may or may not fall.

As before, blue blocks are widest, red blocks are medium, and green blocks are the narrowest. If a block is placed on a block narrower than it, then it will fall with probability P_{fall}, which is the same for all block colors. This boils down to the following behavior:

1. A blue block has a small chance of falling if gets moved onto a green block or a red block, but not on a blue block.
2. A red block has a small chance of falling if moved onto a green block, but not on a red or blue block.
3. A green block will never fall after being moved.

If a block falls, it falls to a completely random table position; it must, however, fall onto a stack that's shorter than the one it came from. Blocks cannot fall off the table. This world lend itself more easily to description in a first-order language, since it does not require generalization over sets of objects or numeric relationships.

I experimented with three versions of this domain: a low stochasticity world with P_{fall} equal to 0.1; a medium stochasticity world with P_{fall} equal to 0.5, and a high stochasticity world with P_{fall} equal to 0.9.

The description for this task is as follows:

- State predicates:

	Arity	Description	Example
clear	binary	Whether a block is clear.	`clear(a,false)`
on	binary	Whether a block is on another.	`on(a,b)`
color	binary	Specifies the color of a block.	`color(b,red)`
intallest	binary	Whether a block is in the tallest stack. If there is more than one tallest stack, this element will be true for all the blocks in those stacks.	`intallest(a,true)`
height	unary	Specifies tallest stack's height.	`height(4)`

- Action symbol:

  The *move* action takes two arguments: the block to be moved, and the block onto which it should go. The action succeeds if both blocks are clear, and fails if one of them is covered. The second argument may be a table block; however, the first argument can only be one of the movable non-table blocks.

	Arity	Description	Example
move	binary	Move a block onto the top of another.	`move(a,b)`

- Reward:

  1. If an action failed, a penalty of -2.0 was assigned.
  2. If an action resulted in a toppled block, a penalty of -1.0 was assigned.
  3. If an action didn't fail, but did not result in increasing the height of the tallest stack, a reward of 0 was given.
  4. If an action resulted in growing the tallest stack, a reward equal to the previous height of the stack was given.

4.3.1 Acquiring the Rules for the Simpler Stochastic Stacking Task

There were two sets of rules learned from the training data: a set of rules induced by CLAUDIEN, and a set of hand-coded rules whose probabilities were learned by counting from the data.

As before, 1,500 training examples were collected from an agent choosing actions randomly. This was done in each of the three low, medium, and high stochasticity versions of the world. A typical training example contained 174 predicates and looked like this:

```
begin(model(0)).
action(move(n32,n26)).
pre(height(4)).
pre(on(n20,n01)).
pre(cl(n20,false)).
pre(color(n20,r)).
pre(intallest(n20,false)).
pre(on(n21,n20)).
pre(cl(n21,true)).
pre(color(n21,b)).
pre(intallest(n21,false)).
...
post(intallest(n38,true)).
post(on(n39,n38)).
post(cl(n39,false)).
post(color(n39,r)).
post(intallest(n39,true)).
post(on(n40,n39)).
post(cl(n40,true)).
post(color(n40,b)).
post(intallest(n40,true)).
reward(0).
end(model(0)).
```

```
dlab_template('1-1:[ 1-1:[reward(R)],
                     1-1:[reward(0)],
                     1-1:[reward(-1)],
                     1-1:[reward(-2)],
                     1-1:[post(on(A,B)),post(cl(A,anytruth)),
                     post(cl(B,anytruth))] ]
                <--
                2-2:[0-len:[pre(height(R)),pre(color(A,anycolor)),
                     pre(color(B,anycolor)),pre(intallest(B,anytruth)),
                     pre(on(A,B)),pre(cl(A,anytruth)),pre(cl(B,anytruth))],
                1-1:[action(move(A,B))] ]').


dlab_variable(anytruth,1-1,[true,false]).
dlab_variable(anycolor,1-1,[g,b,t,r]).
```

Figure 4-8: The DLAB template used in the simpler stochastic block-stacking task. The same template was used in all stochasticity versions of the task.

Rules Learned By CLAUDIEN

The DLAB template is shown in Figure 4-8, and the resulting rules learned for each world are shown in Figures 4-9 to 4-11.

In general, the rules learned by CLAUDIEN capture well the regularities of each task. In the high-stochasticity task, the set contains a number of rules about how the color of the blocks influence the next state. For example, rules 7, 9, 11, and 12 (in Figure 4-9) pick up the tendency of blue blocks to fall, and of green blocks to stay put.

In the mid-stochasticity task, the set includes fewer rules about the block colors, presumably because the blocks do not fall as often in this domain. Rules 20 and 21 (in Figure 4-10) address the impact of moving blue and green blocks. Rules 25 and 26 are an interesting pair—they encode the lack of reward that comes with moving blocks onto stacks that are not the tallest. However, the rule that we would like to see, about actually stacking successfully onto a tallest stack, does not appear; apparently this did not occur often enough in the example set for CLAUDIEN to discover anything.

In the low-stochasticity task, most of the rules centered around how the *clear* and *on* predicates change (or don't) given the preconditions. There is less emphasis on the

```
0: [0.856] move(A,B)->-2.0
1: [1.0] on(A,B) move(A,B)-> cl(B,false)
2: [0.879] cl(A,true) move(A,B)-> cl(B,false)
3: [1.0] cl(B,false) move(A,B)-> cl(B,false)
4: [0.856] color(B,r) color(A,g) move(A,B)-> cl(B,false)
5: [1.0] cl(A,false) move(A,B)-> cl(A,false)
6: [1.0] cl(A,true) move(A,B)-> cl(A,true)
7: [0.888] cl(B,true) color(B,g) move(A,B)-> cl(B,true)
8: [1.0] cl(B,true) cl(A,false) move(A,B)-> cl(B,true)
9: [0.878] cl(B,true) color(A,b) move(A,B)-> cl(B,true)
10: [1.0] on(A,B) move(A,B)-> on(A,B)
11: [1.0] cl(B,true) cl(A,true) color(A,g) move(A,B)-> on(A,B)
12: [1.0] cl(B,true) cl(A,true) color(B,b) move(A,B)-> on(A,B)
13: [1.0] cl(B,true) cl(A,true) color(B,r) color(A,r) move(A,B)-> on(A,B)
14: [0.916] cl(B,true) cl(A,true) color(B,b) color(A,b) move(A,B)->0.0
15: [0.869] cl(B,true) cl(A,true) color(B,g) color(A,r) move(A,B)->-1.0
```

Figure 4-9: CLAUDIEN rules for the high-stochasticity version of the stacking task.

```
0: [0.858] color(A,b) move(A,B)->-2.0
1: [0.856] color(A,r) move(A,B)->-2.0
2: [0.877] color(B,b) move(A,B)->-2.0
3: [0.855] color(B,r) move(A,B)->-2.0
4: [0.933] intallest(B,true) move(A,B)->-2.0
5: [1.0] on(A,B) move(A,B)->-2.0
6: [1.0] cl(A,false) move(A,B)->-2.0
7: [1.0] cl(B,false) move(A,B)->-2.0
8: [0.861] intallest(B,true) move(A,B)-> cl(B,false)
9: [1.0] on(A,B) move(A,B)-> cl(B,false)
10: [0.899] cl(A,true) move(A,B)-> cl(B,false)
11: [1.0] cl(B,false) move(A,B)-> cl(B,false)
12: [1.0] cl(A,false) move(A,B)-> cl(A,false)
13: [1.0] cl(A,true) move(A,B)-> cl(A,true)
14: [1.0] cl(B,true) cl(A,false) color(B,g) move(A,B)-> cl(B,true)
15: [1.0] cl(B,true) cl(A,false) move(A,B)-> cl(B,true)
16: [0.851] cl(B,true) color(B,g) color(A,b) move(A,B)-> cl(B,true)
17: [0.853] cl(B,true) color(B,g) color(A,r) move(A,B)-> cl(B,true)
18: [0.9] cl(B,true) intallest(B,true) color(B,r) color(A,b) move(A,B)->cl(B,true)
19: [0.878] cl(B,true) intallest(B,true) color(A,b) move(A,B)-> cl(B,true)
20: [1.0] on(A,B) move(A,B)-> on(A,B)
21: [1.0] cl(B,true) cl(A,true) color(A,g) move(A,B)-> on(A,B)
22: [1.0] cl(B,true) cl(A,true) color(B,b) move(A,B)-> on(A,B)
23: [0.896] cl(B,true) cl(A,true) color(A,r) move(A,B)-> on(A,B)
24: [0.870] cl(B,true) cl(A,true) color(B,r) move(A,B)-> on(A,B)
25: [0.904] cl(B,true) cl(A,true) intallest(B,false) color(A,g) move(A,B)->0.0
26: [0.860] cl(B,true) cl(A,true) intallest(B,false) color(B,b) move(A,B)->0.0
27: [0.9] cl(B,true) cl(A,true) color(B,b) color(A,b) move(A,B)->0.0
```

Figure 4-10: CLAUDIEN rules for the mid-stochasticity version of the stacking task.

```
0: [0.867333] move(A,B)->-2.0
1: [1.0] on(A,B) move(A,B)-> cl(B,false)
2: [0.953789] cl(A,true) move(A,B)-> cl(B,false)
3: [1.0] cl(B,false) move(A,B)-> cl(B,false)
4: [1.0] cl(A,false) move(A,B)-> cl(A,false)
5: [0.916667] cl(B,true) color(B,g) color(A,g) move(A,B)-> cl(A,false)
6: [1.0] cl(A,true) move(A,B)-> cl(A,true)
7: [0.9] on(A,B) color(B,b) color(A,r) move(A,B)-> cl(A,true)
8: [1.0] cl(B,true) cl(A,false) move(A,B)-> cl(B,true)
9: [0.916667] cl(B,true) color(B,g) color(A,g) move(A,B)-> cl(B,true)
10: [1.0] on(A,B) move(A,B)-> on(A,B)
11: [0.977528] cl(B,true) cl(A,true) move(A,B)-> on(A,B)
12: [0.916667] cl(B,true) cl(A,true) color(B,b) color(A,g) move(A,B)->0.0
```

Figure 4-11: CLAUDIEN rules for the low-stochasticity version of the stacking task.

effect of the colors, as would be expected. Rule 11 (in Figure 4-11) gives the canonical expression for a successful move action.

So, the CLAUDIEN rules capture the domain regularities well. However, they do not pick up on important low-probability events, which will be a problem, as we see later.

Hand-coded Rules with Learned Probabilities

Because of the length of time it took for CLAUDIEN to induce the rules, a quick alternative was to hand code a set of rules and learn their probabilities from the data. Figures 4-12 to 4-14 show the rule set (the same rules for each stochasticity-level of the world) and the learned probabilities. Rules that predict the immediate reward values are depicted with a numeric value in the post-condition; the rule with H in the post-condition means that it predicts a reward value of H (namely, a reward equal to the previous height of the stack).

The probabilities are just simple counts of the number of times in which the antecedent matched and the consequent was true over the total number of times in which the antecedent matched. Thus, the learned probabilities are equivalent to CLAUDIEN's *accuracy* parameter.

```
0: [0.741] cl(A,true) cl(B,true) move(A,B)-> on(A,B)
1: [0.621] cl(A,true) cl(B,true) intallest(B,true) height(H) move(A,B)-> H
2: [0.25] cl(A,true) cl(B,true) color(A,r) move(A,B)-> -1.0
3: [0.587] cl(A,true) cl(B,true) color(A,r) move(A,B)-> 0.0
4: [0.0] cl(A,true) cl(B,true) color(A,g) move(A,B)-> -1.0
5: [0.844] cl(A,true) cl(B,true) color(A,g) move(A,B)-> 0.0
6: [0.632] cl(A,true) cl(B,true) color(A,b) move(A,B)-> -1.0
7: [0.326] cl(A,true) cl(B,true) color(A,b) move(A,B)-> 0.0
8: [1.0] cl(A,false) move(A,B)-> -2.0
9: [1.0] cl(B,false) move(A,B)-> -2.0
```

Figure 4-12: Hand-coded rules, and corresponding probabilities, for the high-stochasticity version of the stacking task.

```
0: [0.829] cl(A,true) cl(B,true) move(A,B)-> on(A,B)
1: [0.812] cl(A,true) cl(B,true) intallest(B,true) height(H) move(A,B)-> H
2: [0.121] cl(A,true) cl(B,true) color(A,r) move(A,B)-> -1.0
3: [0.731] cl(A,true) cl(B,true) color(A,r) move(A,B)-> 0.0
4: [0.0] cl(A,true) cl(B,true) color(A,g) move(A,B)-> -1.0
5: [0.904] cl(A,true) cl(B,true) color(A,g) move(A,B)-> 0.0
6: [0.454] cl(A,true) cl(B,true) color(A,b) move(A,B)-> -1.0
7: [0.436] cl(A,true) cl(B,true) color(A,b) move(A,B)-> 0.0
8: [1.0] cl(A,false) move(A,B)-> -2.0
9: [1.0] cl(B,false) move(A,B)-> -2.0
```

Figure 4-13: Hand-coded rules, and corresponding probabilities, for the mid-stochasticity version of the stacking task.

```
0: [0.977] cl(A,true) cl(B,true) move(A,B)-> on(A,B)
1: [1.0] cl(A,true) cl(B,true) intallest(B,true) height(H) move(A,B)-> H
2: [0.027] cl(A,true) cl(B,true) color(A,r) move(A,B)-> -1.0
3: [0.835] cl(A,true) cl(B,true) color(A,r) move(A,B)-> 0.0
4: [0.0] cl(A,true) cl(B,true) color(A,g) move(A,B)-> -1.0
5: [0.833] cl(A,true) cl(B,true) color(A,g) move(A,B)-> 0.0
6: [0.043] cl(A,true) cl(B,true) color(A,b) move(A,B)-> -1.0
7: [0.804] cl(A,true) cl(B,true) color(A,b) move(A,B)-> 0.0
8: [1.0] cl(A,false) move(A,B)-> -2.0
9: [1.0] cl(B,false) move(A,B)-> -2.0
```

Figure 4-14: Hand-coded rules, and corresponding probabilities, for the low-stochasticity version of the stacking task.

The hand-coded rule set is smaller than each of the learned CLAUDIEN rule sets. There are fewer rules articulating domain regularities, and instead there is more emphasis on rules about the next-state reward. The biggest difference is the rule that predicts the reward for successfully stacking onto the tallest stack (rule 1). Interestingly, as can be seen from the probability counts, the bulk of the hand-coded rules would not have passed CLAUDIEN's 0.9 accuracy threshold.

4.3.2 Planning Experiments

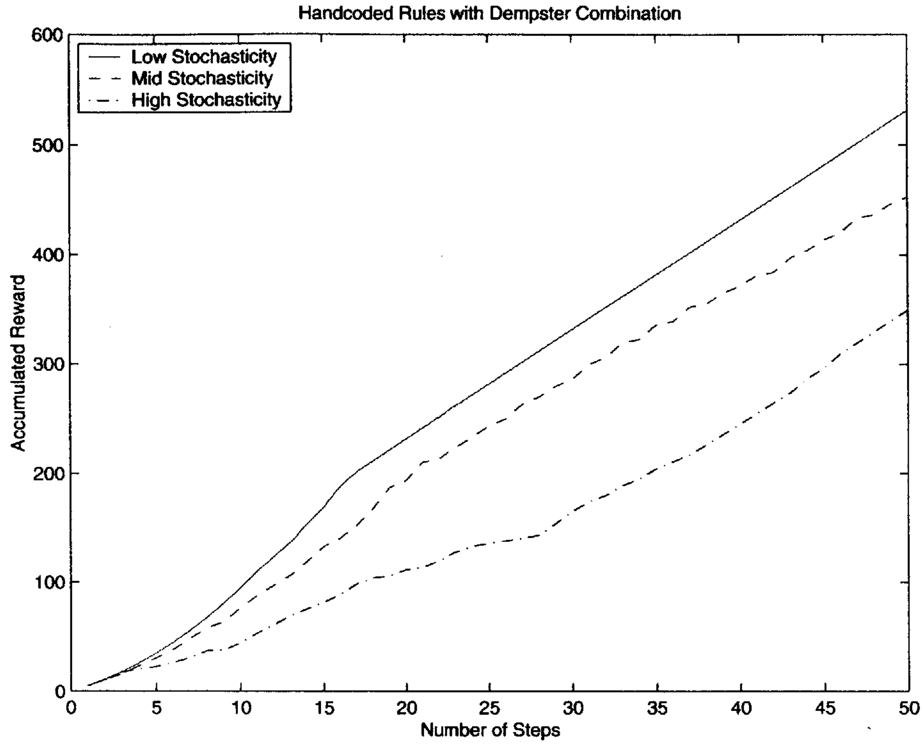
There were two phases of planning experiments. The first set of experiments examines the one-step performance of each rule set on the stacking task in its corresponding domain. This experiment mostly illustrates the ability of each rule set to predict the best greedy action. The second experiment tries to better evaluate how the rules work as an actual model for the world; in this case, the horizon was three steps. The experiment parameters are shown in Table 4.1.

Blocks World	Planning Horizon	Sampling Width	Rule Sets	Combination Rules	P_{fall}
Large stochastic	1	3	CLAUDIEN, Hand-coded	Noisy-OR, Dempster	Low (0.1), Medium (0.5), High (0.9)
Small stochastic	3	3	CLAUDIEN, Hand-coded	Noisy-OR, Dempster	Low (0.1), Medium (0.5), High (0.9)

Table 4.1: Parameters tested in the one- and multi-step planning experiments.

One-step Planning in the Large Stochastic World

In the large stochastic world, the experiment proceeded as follows. For each parameter combination and the same random seed, an experiment of 150 steps was initialized. At each step, the cumulative reward was recorded. The main 150-step experiment was divided into three 50-step episodes: after 50 steps, the blocks were randomly



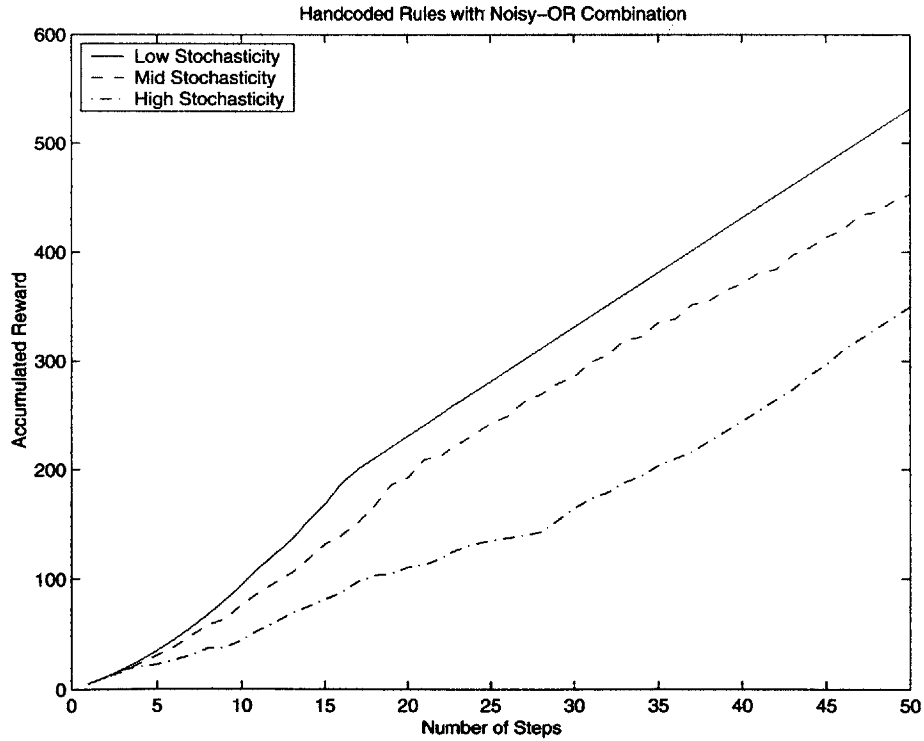
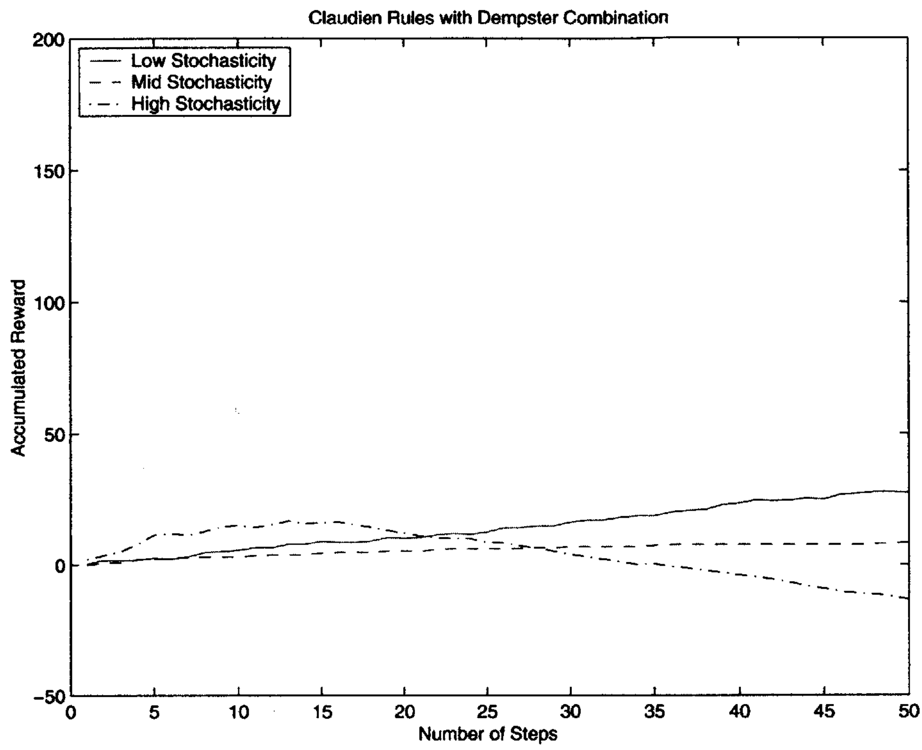


Figure 4-15: Plot of cumulative reward for the hand-coded rules in the one-step planning experiments.



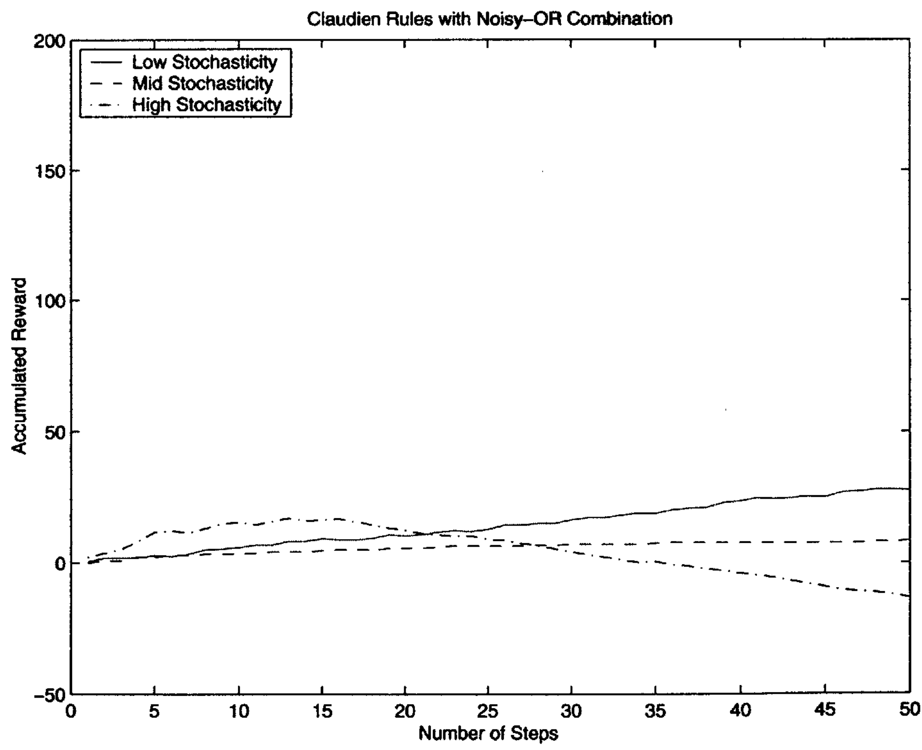


Figure 4-16: Plot of cumulative reward for CLAUDIEN rules in the one-step planning experiment.

re-scrambled and the reward totals were reset to zero. In Figures 4-15 and 4-16 are shown the averages over each 50-step episode for each combination of rule set and combination rule. Each plot is averaged over 5 episodes.

The first thing to note is that the combination rule does not make much of a difference! The graphs for each combination rule look exactly the same. This is because, in fact, the occasions in which there are competing rules for the same value of a state element are quite rare, if they exist at all. The rule sets are just too sparse. Also, as expected, the low-stochasticity world lends itself to higher block stacks than the higher ones. However, even in the high-stochasticity world, the hand-coded rule set is sufficient to build up high stacks.

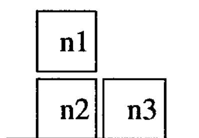


Figure 4-17: A small, example world for illustrating the DBN that results from applying rules from the hand-coded set.

To see how sparse the DBN is, consider the small subset of the stochastic world in Figure 4-17. Let's say we are considering the action `move(n3, n1)`. Then, the rules from hand-coded rule set that apply are:

```
0: cl(A,true) cl(B,true) move(A,B)-> on(A,B)
1: cl(A,true) cl(B,true) intallest(B,true) height(H) move(A,B) -> H
6: cl(A,true) cl(B,true) color(A,b) move(A,B)-> -1.0
7: cl(A,true) cl(B,true) color(A,b) move(A,B)->  0.0
```

The resulting DBN is shown in Figure 4-18. There are not competing rules for the value of, say, `on(n3, n1)`, and so there is no need for a combination rule here.

Looking at the performance plots, the CLAUDIEN rule set seems to be doing quite badly. It appears that the main problem is that, while the rules describe the regularities of the training examples (namely, failure) well, they do not provide any evidence for instances of positive reward. Figure 4.3.2 shows excerpt from the CLAUDIEN

low-stochasticity trial. We can see that the agent avoids failing and topple-inducing actions, but does not know to stack up the blocks in a determined way.

In contrast, the hand-coded rules provide much more directed activity. In the excerpted output in Figure 4.3.2, the agent picks up a *green* block, which is the least likely to fall, and moves it to the top of the tallest stack.

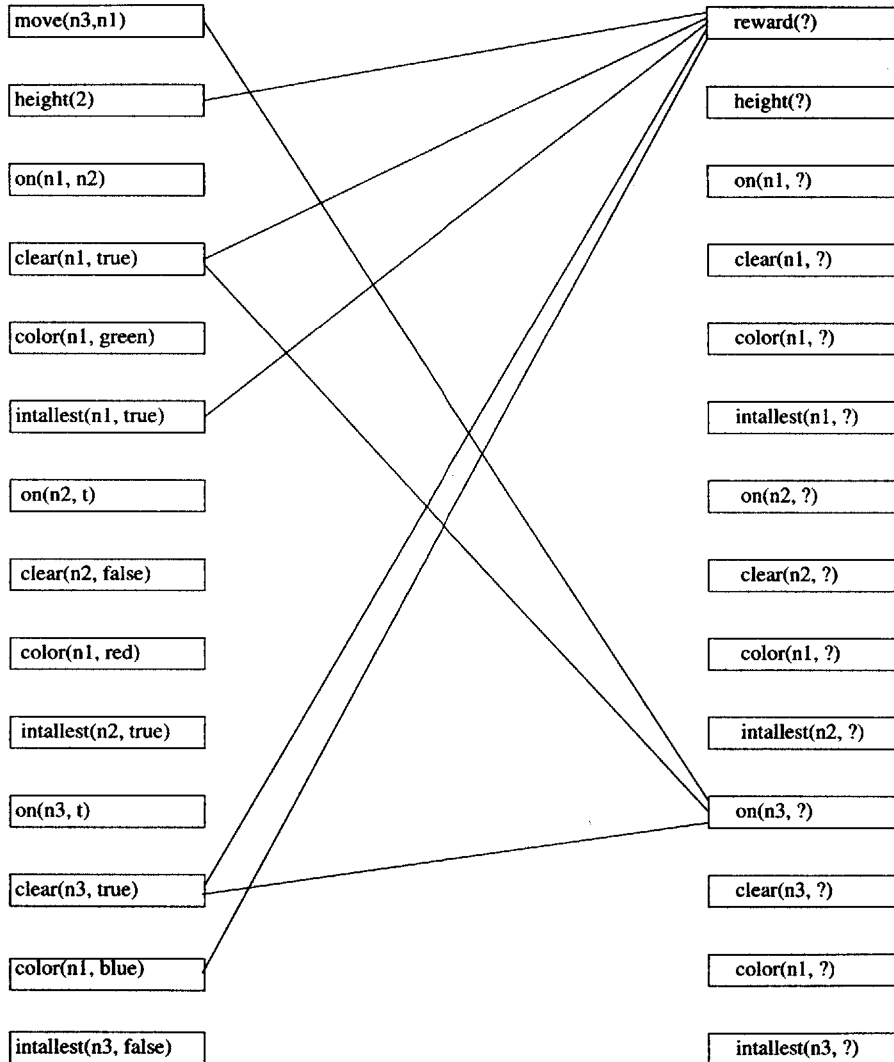


Figure 4-18: The DBN induced by applying the hand-coded rules to the example world in Figure 4-17 and the action `move(n3,n1)`. It is sparse enough that a combination rule does not need to be applied.

```
[n00]
[n01] [n20r] [n21b]
[n02] [n22r]
[n03]
[n04] [n23r] [n24b]
[n05]
[n06] [n26b]
[n07] [n25r] [n29r] [n30r] [n31b]
[n08] [n27g] [n28g] [n32g]
[n09]
[n10]
[n11] [n33r]
[n12] [n34r]
[n13]
[n14] [n35b]
[n15]
[n16] [n36g]
[n17]
[n18] [n37r] [n38r] [n39r] [n40b]
[n19]
```

Figure 4-19: Initial configuration of blocks in all the one-step planning experiments. The configuration can also be seen schematically in Figure 4-5. Each block is represented by a pair of square brackets around the blocks' name and the first letter of the block's color (e.g., block *n20* is red; *n00* is a table block).

In the first step, there are a number of actions of equal value. These actions all involve moving clear green blocks (there are two: n36 and n32) onto clear blue blocks.

```
-- Randomly choosing between 12 items at value=-0.867333:
[MOVE(n32,n21)] [MOVE(n32,n24)] [MOVE(n32,n26)] [MOVE(n32,n31)]
[MOVE(n32,n40)] [MOVE(n36,n21)] [MOVE(n36,n24)] [MOVE(n36,n40)]
[MOVE(n36,n31)] [MOVE(n36,n35)] [MOVE(n36,n26)] [MOVE(n32,n35)]
     Step 0, Chose: [MOVE(n32,n24)], r: 0.0
[n00]
[n01][n20r][n21b]
[n02][n22r]
[n03]
[n04][n23r][n24b] [n32g]
[n05]
[n06][n26b]
[n07][n25r][n29r][n30r][n31b]
[n08][n27g][n28g]
[n09]
[n10]
[n11][n33r]
[n12][n34r]
[n13]
[n14][n35b]
[n15]
[n16][n36g]
[n17]
[n18][n37r][n38r][n39r][n40b]
[n19]
```

In the second step, the best actions are again those that move a clear green block to a clear blue block. Notice that there is no rule to guide the agent toward growing the tallest stack.

```
-- Randomly choosing between 15 items at value=-0.867333:
[MOVE(n28,n21)] [MOVE(n28,n26)] [MOVE(n36,n40)] [MOVE(n28,n31)]
[MOVE(n28,n40)] [MOVE(n32,n21)] [MOVE(n32,n26)] [MOVE(n32,n31)]
[MOVE(n32,n40)] [MOVE(n36,n21)] [MOVE(n36,n26)] [MOVE(n36,n31)]
[MOVE(n36,n35)] [MOVE(n32,n35)] [MOVE(n28,n35)]
     Step 1, Chose: [MOVE(n28,n35)], r: 0.0
[n00]
[n01][n20r][n21b]
[n02][n22r]
[n03]
[n04][n23r][n24b][n32g]
[n05]
[n06][n26b]
[n07][n25r][n29r][n30r][n31b]
[n08][n27g]
[n09]
[n10]
[n11][n33r]
[n12][n34r]
[n13]
[n14][n35b] [n28g]
[n15]
[n16][n36g]
[n17]
[n18][n37r][n38r][n39r][n40b]
[n19]
```

Again, the agent moves a clear green block onto a blue block. The behavior of the CLAUDIEN-rules agents is characterized by this conservative, but aimless, stacking.

```
-- Randomly choosing between 16 items at value=-0.867333:
[MOVE(n36,n26)] [MOVE(n36,n31)] [MOVE(n36,n40)] [MOVE(n27,n21)]
[MOVE(n27,n31)] [MOVE(n27,n40)] [MOVE(n28,n21)] [MOVE(n28,n26)]
[MOVE(n28,n31)] [MOVE(n28,n40)] [MOVE(n32,n21)] [MOVE(n32,n26)]
[MOVE(n32,n31)] [MOVE(n27,n26)] [MOVE(n36,n21)] [MOVE(n32,n40)]
     Step 2, Chose: [MOVE(n32,n26)], r: 0.0
[n00]
[n01][n20r][n21b]
[n02][n22r]
[n03]
[n04][n23r][n24b]
[n05]
[n06][n26b] [n32g]
[n07][n25r][n29r][n30r][n31b]
[n08][n27g]
[n09]
[n10]
[n11][n33r]
[n12][n34r]
[n13]
[n14][n35b][n28g]
[n15]
[n16][n36g]
[n17]
[n18][n37r][n38r][n39r][n40b]
[n19]
```

Figure 4-20: Output from the CLAUDIEN rules in the low-stochasticity one-step task.

In the first step for the hand-coded-rules agent, there are four actions of equal value. They involve moving one of the clear green blocks n36 or n32) onto one of the two tallest stacks (at positions 7 and 18).

```
-- Randomly choosing between 4 items at value=1.33:
[MOVE(n32,n31)] [MOVE(n32,n40)] [MOVE(n36,n31)] [MOVE(n36,n40)]
        Step 0, Chose: [MOVE(n36,n31)], r: 4.0
[n00]
[n01][n20r][n21b]
[n02][n22r]
[n03]
[n04][n23r][n24b]
[n05]
[n06][n26b]
[n07][n25r][n29r][n30r][n31b]{\bf [n36g]}
[n08][n27g][n28g][n32g]
[n09]
[n10]
[n11][n33r]
[n12][n34r]
[n13]
[n14][n35b]
[n15]
[n16]
[n17]
[n18][n37r][n38r][n39r][n40b]
[n19]
```

In the next step, now that there is a single tallest stack, there is only one best action: to move the only clear green block onto the tallest stack.

```
        Step 1, Chose: [MOVE(n32,n36)], r: 5.0
[n00]
[n01][n20r][n21b]
[n02][n22r]
[n03]
[n04][n23r][n24b]
[n05]
[n06][n26b]
[n07][n25r][n29r][n30r][n31b][n36g]{\bf [n32g]}
[n08][n27g][n28g]
[n09]
[n10]
[n11][n33r]
[n12][n34r]
[n13]
[n14][n35b]
[n15]
[n16]
[n17]
[n18][n37r][n38r][n39r][n40b]
[n19]
```

And again in the third step: the agent continues to grow the tallest stack greedily. When it runs out out of green blocks, it goes on to choose among the red blocks.

```
        Step 2, Chose: [MOVE(n28,n32)], r: 6.0
[n00]
[n01][n20r][n21b]
[n02][n22r]
[n03]
[n04][n23r][n24b]
[n05]
[n06][n26b]
[n07][n25r][n29r][n30r][n31b][n36g][n32g]{\bf [n28g]}
[n08][n27g]
[n09]
[n10]
[n11][n33r]
[n12][n34r]
[n13]
[n14][n35b]
[n15]
[n16]
[n17]
[n18][n37r][n38r][n39r][n40b]
[n19]
```

Figure 4-21: Output from the hand-coded rules in the low-stochasticity one-step task.

Multi-step Planning in the Small Stochastic World

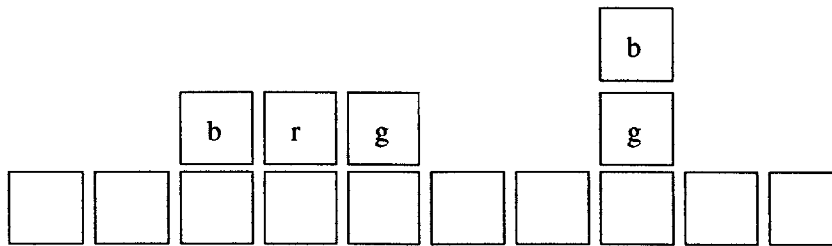


Figure 4-22: The smaller world used in the three-step planning experiments.

Admittedly, a one-step horizon is not much of a "planning" exercise. The hand-coded rule set, for example, is very good at choosing the next greedy action (that is, to pick up a green block if available and put it on top of the tallest stack), but, given a longer horizon, would the agent figure out that it needed to rebuild a stack so that it had a stronger base?

The main obstacle to running experiments with a longer horizon is the computational load of fanning out over 441 actions. Thus, the smaller world shown in Figure 4-22 was used to test out the planning with the horizon set to three. This smaller world, with five blocks, has 25 actions total.

As we can see in Figure 4-24, the CLAUDIEN-rules agent oscillates between the two most conservative actions it has: moving the green block from one blue block to the other. Because it has no rules to tell it how to grow the tallest stack, it simply does the best it can with the rules it has—it avoids toppling a block. This is always achieved by moving a green block.

```
[n00]
[n01]
[n02] [n14b]
[n03] [n12r]
[n04] [n13g]
[n05]
[n06]
[n07] [n10g] [n11b]
[n08]
[n09]
```

Figure 4-23: Initial configuration of blocks in the three-step planning experiments. The configuration can also be seen schematically in Figure 4-22.

```
        Step 0, [MOVE(n13,n14)], r: 1.0
[n00]
[n01]
[n02][n14b] [n13g]
[n03][n12r]
[n04]
[n05]
[n06]
[n07][n10g][n11b]
[n08]
[n09]

        Step 1, [MOVE(n13,n11)], r: 2.0
[n00]
[n01]
[n02][n14b]
[n03][n12r]
[n04]
[n05]
[n06]
[n07][n10g][n11b] [n13g]
[n08]
[n09]

        Step 2, [MOVE(n13,n14)], r: 1.0
[n00]
[n01]
[n02][n14b] [n13g]
[n03][n12r]
[n04]
[n05]
[n06]
[n07][n10g][n11b]
[n08]
[n09]
```

Figure 4-24: Output from the CLAUDIEN rules in the low-stochasticity three-step task.

```
        Step 0, [MOVE(n13,n11)], r: 2.0
[n00]
[n01]
[n02][n14b]
[n03][n12r]
[n04]
[n05]
[n06]
[n07][n10g][n11b] [n13g]
[n08]
[n09]

        Step 1, [MOVE(n14,n13)], r: 3.0
[n00]
[n01]
[n02]
[n03][n12r]
[n04]
[n05]
[n06]
[n07][n10g][n11b][n13g] [n14b]
[n08]
[n09]

        Step 2, [MOVE(n12,n14)], r: 4.0
[n00]
[n01]
[n02]
[n03]
[n04]
[n05]
[n06]
[n07][n10g][n11b][n13g][n14b] [n12r]
[n08]
[n09]
```

Figure 4-25: Output from the hand-coded rules in the low-stochasticity three-step task.

In Figure 4-25, the hand-coded rules agent still starts out with the immediate greedy action: picking up the green block from the table and putting it on the tallest stack. However, what happens next is interesting. It chooses to put the blue block on top of the green, and then the red block on top of that. It was able to model that trying the blue block first was worthwhile: if the blue block sticks, then the red block is guaranteed not to fall; however, if it tries the red block first, it may stick (with the same probability that the blue block has of sticking), but there is still a chance that the blue block might fall off if placed on top of it later.

Ideally, the first action should be to move either blue block onto the other blue block (since the width-effect only applies to the block immediately under a newly placed block), and then go on build a stack with the blue blocks as a base. While the hand-coded rules do a good job of modeling the next-state reward, they do not do quite enough to provide information about the next state itself. Thus, when projecting forward, the model is forced to make a lot of 'static world' assumptions that don't necessarily apply.

Looking a bit deeper, though, the problem cannot be solved by just adding a rule that says:

$$\text{With probability 0.9: } move(A, B) \wedge clear(A) \rightarrow on(A, B).$$

We need to actually say:

$$\text{With probability 0.1: } move(A, B) \wedge clear(A) \rightarrow \neg on(A, B).$$

Our language is presently restricted to Horn clauses, which means we cannot have a negation as a post-condition. But suppose we were in fact able to have negated consequents, how would we use them to generate the next state? Our consequents are not necessarily binary-valued. Presumably, $\neg on(A, B)$ means there is some chance for `on(a,table)`, or `on(a,c)`, `on(a,d)`, and so forth. Currently, without a rule describing what happens in the other 10% of the cases, the system normalizes the

outcomes of the applicable high-probability rules and chooses among them. If the only rule we had was the rule predicting `on(A,B)` 90% of the time, then the system would only ever predict the outcome `on(A,B)`. If we knew what all the outcomes were, presumably we could divide the probability mass among them. But how do we know what all the outcomes are when our consequents can take on any number of discrete values?

Clearly, rules that model important, high-probability events are very useful, especially for taking the best greedy action. The agent can choose the next action optimistically, try it, and then choose the next best action according to the resulting state of the actual world.

However, to accurately model the world, high-probability rules are not enough. While the system is able to learn about regularities, it ignores the exceptions might be important in their own right. This is especially true if the low-probability events carry very large penalties (such as, say, the robot's block stack falling down; or, in the cliff-walking domain [55], possibly falling off the cliff, etc.) Or even, as illustrated in this domain, if the low-probability events carry very high reward. When the events in question are not binary-valued, as they are not in the above blocks world, then our task is two-fold; we must first identify the occasional events before even beginning to learn probabilities for them. These ideas and others for describing the world more accurately are discussed next.

Chapter 5

Conclusions

This work sought to address the following issue: given a set of probabilistic first-order rules describing a world's dynamics, how can one define a probability distribution to estimate the next state from the current state and a proposed action? The approach I studied was to create a partially specified dynamic Bayesian network for each proposed action by applying rules from the set to the current state and by employing a combination rule to resolve any competition among the rules.

I hoped to discover two main things from the experiments. First, are the combination rules a reasonable way to combine pieces of evidence? That is, given the different rule-sets, are the assumptions underlying the Noisy-OR rule and the Dempster rule still valid in each case? And second, if the next-states are in fact generated in a reasonable way, how well does the planning algorithm work?

5.1 Combining Evidence

The Noisy-OR combination rule assumes that a failure to see the expected result is due to independent failure of all the causes. Furthermore, it assumes that the

presence of a cause is enough to trigger the result (and that the absence of a cause does not take away from an existing trigger). The Dempster rule also assumes an additive nature in the contributing causes, as well as independence in the causes.

In the deterministic experiment, Noisy-OR seemed to work reasonably well. This seems to be because the independence and closeness-to-one assumptions underlying Noisy-OR match up well with the bias underlying CLAUDIEN's data mining engine. CLAUDIEN seeks out rules with an accuracy greater than a specified threshold; if the accuracy is close to one (in this case, 0.9) then the rules in fact produce evidence for a state feature with probability close to one. CLAUDIEN also seeks out the most compact set of rules that explains the training examples, which will lead it to prune out redundant rules; thus, because each rule ends up mainly explaining a subset of the training data that is not already explained by some other rule, the assumption of independence among the rules is likely to hold. The conclusion is that the Noisy-OR and Dempster rules are good matches for data mining engines like CLAUDIEN. The Dempster rule is perhaps a bit more general in that it does not assume such a specific causal relationship.

In the stochastic experiments, the rules were so sparse as to essentially obviate the need for a combination rule.

5.2 Learning and Planning with Rules

For acquiring a set of rules, I found that CLAUDIEN is a good approach if the regularities are "uni-modal", in some sense; that is to say, if there are lots of training examples that can be characterized in the same way. However, if some small subset of the data is characterized very tightly in one way, but it it is overshadowed by another (larger) subset of data with different characteristics, then CLAUDIEN will not find rules for the smaller subset; it will get thrown out as low-probability noise. This is why, for example, in the stochastic stacking task CLAUDIEN did not learn a rule

about the tall stacks resulting in high reward: even though the phenomenon was very regular, there just were not enough examples of it.

The hand-coded rules performed very well in the one-step planning experiment, but they were insufficient for the multi-step planning experiment. As probably was to be expected, in hindsight, they were clearly a less complete model of the world dynamics than that derived by a more thorough and patient engine such as CLAUDIEN. Nevertheless, using the hand-coded rules was illuminating for a number of reasons. The hand-coded rules were a very good model of the next state reward: as such, they might initially appear to have little advantage over the value-based approaches that learn mappings from states to values. However, the fact that the rules learned in the large stochastic world directly applied to the smaller stochastic world without any kind of re-training is a clear validation of the relational, forward model-based approach. The problem with learning value functions is that value functions have bundled up inside them the notion of distance to the goal: if the size of the task, or the goal itself, changes, it is difficult to see how one might directly apply a previously learned value function. With a predictive forward model, however, the dependence on the goal and the distance to the goal goes away.

The computational load of planning, even with the sparse sampling over next states, was still significant. In part, this was due to my admittedly simple implementation for unifying the first-order rules against the current state. With more work, this part of the code could certainly have been made more efficient. Nevertheless, the main obstacle is the sheer number of actions that must be tried at each node in the tree.

5.3 Future Work

There are three main ways to extend the work described here: speeding up the planning, enriching the rule language, and improving the rule learning.

To speed up the planning, it is imperative to reduce the number of actions that are looked at at each level in order to reign in the combinatorial explosion. One very old strategy, used by the RRL system, is to adopt STRIPS-like action pre-conditions. As it is, so many of the $move(a, b)$ actions are for blocks that are covered by other blocks, for example. Having a pre-condition, as the RRL system does, that the blocks must be clear would eliminate all of these candidates from the search tree. Another potential strategy would be to do some kind of sampling among the actions. Clearly, it would make little sense to sample among actions in an unguided way: the very best action may be only one in a sea of poor actions. The system could start out considering all actions equally, as it does now, and then gradually sample among them with probability proportional to the number of times the action was selected in the past. This kind of "re-weighting" scheme could be easily combined with a pre-condition scheme, for example.

Another result that was clear from the experiments is that we must move beyond the language of Horn clauses to something richer. In the case of the first stochastic blocks world, for example, it was difficult to describe a couple of important ideas:

- Buttressing stacks: we want to be able to say that building up a set of neighboring stacks around the tallest stack contributes to that stack's stability. We want to be able to learn a rule that says stacking a block onto a neighboring stack may be better in the long run than greedily stacking onto the tallest stack. This requires that we talk about the stack next to the tallest stack, and the stack next to that neighboring stack, and so on, for however many such stacks there happen to be at run time. With standard first-order expressions, we can write down what happens in either of the following situations:
  - `tallest(7)` $\wedge$ `nextto(7,6)` $\wedge$ `pos(B,6)` $\wedge$ `move(A,B)`, or
  - `tallest(7)` $\wedge$ `nextto(7,6)`$\wedge$ `nextto(6,5)` $\wedge$ `pos(B,5)` $\wedge$ `move(A,B)`,

  but we do not have a way to describe the concept of "next-to" chained an arbitrary number of times.

- Towering stacks being more unstable: we would like to say that if a tall stack towers over its nearest buttressing stack, then the set of blocks in the tallest stack that are higher than the neighboring stack are more likely to fall down. The same difficulty crops up as before; we need to talk about the block on a certain block, and the block on top of that, and so on, until the top of the stack. There is no way to write down a rule that addresses an arbitrary number of towering blocks.

One possibility is to use a concept language, as described by Martín and Geffner [39] and Yoon *et al.* [57]. Concept languages are first-order languages that uses class expressions to denote sets, rather than just individuals. Such languages provide the Kleene star operator, $*$, as a primitive. This relieves one of having to write down a different rule for, say, towering stacks of different heights, and lets one talk about "all the blocks above this block" by chaining the **on** predicate arbitrarily.

An interesting idea for learning rules autonomously would be to apply Baum's evolutionary approach [5, 6] in a concept language version of blocks world, rather than in his original propositional one. There are interesting parallels between the work of Baum and the work in decision-list policies of Martín and Geffner, Yoon *et al.*, and Khardon [33]. For example, the decision-list learners order the rules in the list greedily, according to the rule's coverage of the training examples. In contrast, Baum's rules learn their ordering by autonomously learning appropriate *bid* values. Conversely, Baum noticed that there were some concepts he had to hand-code into his propositional representation in order to facilitate learning. For example, he needed to encode *under*, which is the inverse of the given *on* relation and the idea of *top of stack*, which can be expressed in terms of the transitive closure of the *on* relation. Both of these operations, inverse and transitive closure (or Kleene star), are available as primitives in the syntax of a concept language. Furthermore, Baum's system learns concepts autonomously, while the decision-list systems must learn from solved examples. Thus, the combination of a richer language with Baum's economy-style learning could be very interesting.

Finally, as alluded to at the end of the last chapter, it seems necessary for a well-defined world model not only to include information about what happens most of the time, but also to specify something about what might happen the rest of the time. That is to say, if we want to model the instances when $move(A, B) \wedge clear(A) \rightarrow on(A, B)$ does not hold, then we need to know something about the domain over which the *on* relation operates so that we can assign some probability to those situations that are encompassed by $\neg on(A, B)$. This need also exists with the PSTRIPS approach as used by Yoon *et al.*: the set of *action outcomes* must be specified ahead of time. This kind of information is crucial if the method of sparse sampling is to produce an estimated next state consistent with the world dynamics. The important question to answer here is how to go about identifying the domain, which may not be known ahead of time.

One straightforward idea is to start with just the set of high-probability rules. Then, once a rule has been identified as useful, we can go back to the data and look at what happens in the cases when the rule doesn't predict the right thing. This could either be done by re-examining the set of training examples, or by storing such counter-examples as they arise in practice. Once we get a sense for what the alternative, lower-probability outcomes are, we can begin to estimate probabilities for them.

Ultimately, the batch nature of the system described here is somewhat unsatisfying: first we get some rules, then we act. Intuitively, we would want our success or failure at acting to drive the learning of more, or better, rules. For example, we should be able to ask for more rules in settings where we find the current model is poor. One simple way to do this would be to simply cache the state-action-state triplets where we do poorly and then add them to our set of training examples. At some later point, CLAUDIEN or some other rule-inducer can be run to learn rules on the expanded training set. This approach is simple, but requires that we keep around an ever-growing pile of training examples. We could, for example, decide that after each bout of rule-learning, we discard the training examples; then, we would only have to learn rules for the new examples on each iteration. Now we must deal with

the question of how to incorporate the new rules into the existing rule-base: simply tacking on the new rules at the end of the canon could result in redundancies (if, say, we learn a rule that is more specific then a previously learned rule), which would violate any assumptions about independence.

Clearly, there are many ways to improve on the system described here; there is room for future work in the planning, in describing the language for the rules, and in the rule learning itself. Nevertheless, this system has shown the advantages of a model-based approach over a value-based one. It has also shown that it is possible to specify well-defined distributions over next states even with very limited information, given strong assumptions about the world's dynamics. The problem of finding a compact and expressive world model tractably is still wide open, but the approach of using a sparse set of first-order rules, as shown here, is a step along the way.

Appendix A

Rules Learned by CLAUDIEN For the Deterministic Blocks World

```
0: [1.0] move(a,a)-> 0.0
1: [1.0] move(a,t)-> 0.0
2: [1.0] move(a,c)-> 0.0
3: [1.0] move(b,a)-> 0.0
4: [1.0] move(b,b)-> 0.0
5: [1.0] move(b,t)-> 0.0
6: [1.0] move(b,c)-> 0.0
7: [1.0] move(t,a)-> 0.0
8: [1.0] move(t,b)-> 0.0
9: [1.0] move(t,t)-> 0.0
10: [1.0] move(t,c)-> 0.0
11: [1.0] move(c,a)-> 0.0
12: [1.0] move(c,b)-> 0.0
13: [1.0] move(c,t)-> 0.0
14: [1.0] move(c,c)-> 0.0
15: [0.987854] move(D,E)-> 0.0
16: [1.0] cl(a,true) on(c,t) move(a,b)-> 1.0
17: [0.923077] cl(c,false) on(c,t) move(a,b)-> 1.0
18: [0.923077] cl(c,false) cl(b,true) move(a,b)-> 1.0
19: [1.0] cl(D,true) on(c,t) goal(on(D,E)) move(a,b)-> 1.0
20: [1.0] cl(b,true) cl(a,true) move(a,b)-> 1.0
21: [1.0] cl(D,true) cl(b,true) goal(on(D,E)) move(a,b)-> 1.0
22: [0.923077] cl(E,true) cl(c,false) goal(on(D,E)) move(a,b)-> 1.0
23: [1.0] cl(E,true) cl(a,true) goal(on(D,E)) move(a,b)-> 1.0
24: [1.0] cl(E,true) cl(D,true) goal(on(D,E)) move(a,b)-> 1.0
```

```
25: [1.0] on(c,t) on(b,t) move(a,b)-> 1.0
26: [0.909091] cl(b,true) on(b,t) move(a,b)-> 1.0
27: [0.909091] cl(E,true) on(b,t) goal(on(D,E)) move(a,b)-> 1.0
28: [1.0] on(a,c) move(t,c)-> on(a,c)
29: [1.0] cl(c,false) on(b,t) move(t,c)-> on(a,c)
30: [1.0] on(a,c) move(t,t)-> on(a,c)
31: [1.0] cl(E,false) cl(c,false) goal(on(D,E)) move(t,t)-> on(a,c)
32: [1.0] on(c,t) on(b,t) move(t,t)-> on(a,c)
33: [1.0] cl(c,false) on(b,t) move(t,t)-> on(a,c)
34: [1.0] cl(b,true) cl(a,true) on(b,t) move(t,t)-> on(a,c)
35: [1.0] cl(E,true) cl(a,true) on(b,t) goal(on(D,E)) move(t,t)-> on(a,c)
36: [1.0] cl(E,true) cl(D,true) on(b,t) goal(on(D,E)) move(t,t)-> on(a,c)
37: [1.0] cl(c,false) cl(b,false) cl(a,true) move(t,t)-> on(a,c)
38: [1.0] cl(D,false) cl(c,false) on(c,t) goal(on(D,E)) move(t,t)-> on(a,c)
39: [1.0] cl(D,true) cl(b,true) on(b,t) goal(on(D,E)) move(t,t)-> on(a,c)
40: [1.0] cl(c,false) cl(a,false) on(c,t) move(t,t)-> on(a,c)
41: [0.923077] cl(E,true) cl(c,false) goal(on(D,E)) move(D,E)-> on(D,E)
42: [1.0] cl(b,true) cl(a,true) on(a,t) move(a,t)-> cl(c,false)
43: [1.0] cl(a,true) on(c,t) on(a,t) move(a,t)-> cl(c,false)
44: [1.0] cl(D,true) cl(b,true) on(a,t) goal(on(D,E)) move(a,t)-> cl(c,false)
45: [1.0] cl(E,true) cl(a,true) on(a,t) goal(on(D,E)) move(a,t)-> cl(c,false)
46: [1.0] cl(D,true) on(c,t) on(a,t) goal(on(D,E)) move(a,t)-> cl(c,false)
47: [1.0] cl(D,true) cl(c,true) goal(on(D,E)) move(t,a)-> on(b,t)
48: [0.945946] on(a,c) move(D,E)-> on(a,c)
49: [0.931429] cl(c,false) on(b,t) move(D,E)-> on(a,c)
50: [1.0] cl(D,false) on(c,t) on(b,t) move(D,E)-> on(a,c)
51: [1.0] cl(E,false) on(c,t) on(b,t) move(D,E)-> on(a,c)
52: [1.0] cl(D,false) cl(b,true) cl(a,true) on(b,t) move(D,E)-> on(a,c)
53: [1.0] cl(E,false) cl(b,true) cl(a,true) on(b,t) move(D,E)-> on(a,c)
54: [0.930556] cl(c,false) cl(b,false) cl(a,true) move(D,E)-> on(a,c)
55: [1.0] cl(D,true) cl(c,false) on(a,t) goal(on(D,E)) move(t,t)-> on(c,t)
56: [1.0] cl(E,true) cl(c,false) on(b,t) goal(on(D,E)) move(t,t)-> on(c,t)
57: [1.0] on(b,c) move(a,t)-> on(b,c)
58: [1.0] cl(c,false) on(a,t) move(a,t)-> on(b,c)
59: [1.0] cl(b,true) cl(a,true) on(a,t) move(a,t)-> on(b,c)
60: [1.0] cl(a,true) on(c,t) on(a,t) move(a,t)-> on(b,c)
61: [1.0] cl(D,true) cl(b,true) on(a,t) goal(on(D,E)) move(a,t)-> on(b,c)
62: [1.0] cl(E,true) cl(a,true) on(a,t) goal(on(D,E)) move(a,t)-> on(b,c)
63: [1.0] cl(D,true) on(c,t) on(a,t) goal(on(D,E)) move(a,t)-> on(b,c)
64: [1.0] cl(E,true) cl(D,true) on(a,t) goal(on(D,E)) move(a,t)-> on(b,c)
65: [1.0] cl(E,true) cl(D,true) on(a,t) goal(on(D,E)) move(a,t)-> cl(c,false)
66: [0.912621] cl(E,true) cl(b,false) on(a,t) move(D,E)-> cl(E,true)
67: [0.912621] cl(E,true) on(c,b) on(a,t) move(D,E)-> cl(E,true)
68: [0.909091] cl(D,false) cl(a,true) on(c,b) on(a,t) move(D,E)-> cl(E,true)
69: [1.0] on(b,c) move(c,t)-> on(b,c)
```

```
70: [1.0] cl(c,false) on(a,t) move(c,t)-> on(b,c)
71: [1.0] cl(b,true) cl(a,true) on(a,t) move(c,t)-> on(b,c)
72: [1.0] cl(E,true) cl(a,true) on(a,t) goal(on(D,E)) move(c,t)-> on(b,c)
73: [1.0] cl(a,true) on(c,t) on(a,t) move(c,t)-> on(b,c)
74: [1.0] cl(D,true) cl(b,true) on(a,t) goal(on(D,E)) move(c,t)-> on(b,c)
75: [1.0] cl(D,true) on(c,t) on(a,t) goal(on(D,E)) move(c,t)-> on(b,c)
76: [1.0] cl(E,true) cl(D,true) on(a,t) goal(on(D,E)) move(c,t)-> on(b,c)
77: [0.916667] cl(E,true) cl(b,false) move(D,E)-> cl(E,true)
78: [0.916667] cl(E,true) on(c,b) move(D,E)-> cl(E,true)
79: [1.0] cl(c,false) cl(a,false) on(c,t) move(D,E)-> on(a,c)
80: [0.907609] on(b,c) move(D,E)-> on(b,c)
81: [1.0] cl(E,false) cl(c,false) on(c,a) move(D,E)-> on(b,c)
82: [0.907609] cl(c,false) on(a,t) move(D,E)-> on(b,c)
83: [1.0] cl(D,false) cl(b,true) cl(a,true) on(a,t) move(D,E)-> on(b,c)
84: [1.0] cl(E,false) cl(b,true) cl(a,true) on(a,t) move(D,E)-> on(b,c)
85: [1.0] cl(D,false) cl(a,true) on(c,t) on(a,t) move(D,E)-> on(b,c)
86: [1.0] cl(E,false) cl(a,true) on(c,t) on(a,t) move(D,E)-> on(b,c)
87: [1.0] cl(b,true) cl(a,true) on(a,t) move(t,c)-> on(b,c)
88: [1.0] cl(E,true) cl(a,true) on(a,t) goal(on(D,E)) move(t,c)-> on(b,c)
89: [1.0] cl(D,true) cl(b,true) on(a,t) goal(on(D,E)) move(t,c)-> on(b,c)
90: [1.0] cl(a,true) on(c,t) on(a,t) move(t,c)-> on(b,c)
91: [1.0] cl(D,true) on(c,t) on(a,t) goal(on(D,E)) move(t,c)-> on(b,c)
92: [1.0] cl(E,true) cl(D,true) on(a,t) goal(on(D,E)) move(t,c)-> on(b,c)
93: [1.0] cl(c,false) cl(b,false) move(t,t)-> on(a,c)
94: [1.0] cl(c,false) on(c,b) move(t,t)-> on(a,c)
95: [1.0] on(c,a) move(t,c)-> on(c,a)
96: [0.930556] cl(c,false) cl(b,false) move(D,E)-> on(a,c)
97: [0.930556] cl(c,false) on(c,b) move(D,E)-> on(a,c)
98: [1.0] cl(D,true) cl(c,true) goal(on(D,E)) move(t,t)-> on(b,t)
99: [1.0] cl(c,false) on(c,a) move(t,t)-> on(b,c)
100: [1.0] cl(D,true) cl(b,true) on(a,t) goal(on(D,E)) move(t,t)-> on(b,c)
101: [1.0] cl(b,true) cl(a,true) on(a,t) move(t,t)-> on(b,c)
102: [1.0] cl(E,true) cl(a,true) on(a,t) goal(on(D,E)) move(t,t)-> on(b,c)
103: [1.0] cl(a,true) on(c,t) on(a,t) move(t,t)-> on(b,c)
104: [1.0] cl(D,true) on(c,t) on(a,t) goal(on(D,E)) move(t,t)-> on(b,c)
105: [1.0] cl(E,true) cl(D,true) on(a,t) goal(on(D,E)) move(t,t)-> on(b,c)
106: [1.0] cl(c,false) on(c,a) move(D,E)-> on(c,a)
107: [1.0] cl(D,false) on(c,a) move(D,E)-> on(c,a)
108: [1.0] cl(E,false) on(c,a) move(D,E)-> on(c,a)
109: [1.0] on(c,a) on(b,c) move(D,E)-> on(c,a)
110: [1.0] cl(a,false) on(b,c) move(D,E)-> on(c,a)
111: [1.0] cl(D,false) cl(a,false) on(b,t) move(D,E)-> on(c,a)
112: [1.0] cl(E,false) cl(a,false) on(b,t) move(D,E)-> on(c,a)
113: [1.0] cl(D,false) cl(c,true) cl(b,true) on(b,t) move(D,E)-> on(c,a)
114: [1.0] cl(E,false) cl(c,true) cl(b,true) on(b,t) move(D,E)-> on(c,a)
```

```
115: [1.0] cl(D,false) cl(b,true) on(b,t) on(a,t) move(D,E)-> on(c,a)
116: [1.0] cl(E,false) cl(b,true) on(b,t) on(a,t) move(D,E)-> on(c,a)
117: [1.0] cl(c,false) cl(a,false) on(a,t) move(D,E)-> on(c,a)
118: [1.0] cl(c,false) on(b,a) move(t,t)-> on(a,c)
119: [1.0] cl(c,false) on(b,a) move(D,E)-> on(a,c)
120: [1.0] on(c,a) move(t,t)-> on(c,a)
121: [1.0] cl(D,false) on(b,c) goal(on(D,E)) move(t,t)-> on(c,a)
122: [1.0] cl(D,false) cl(c,false) on(a,t) goal(on(D,E)) move(t,t)-> on(c,a)
123: [1.0] cl(E,true) cl(D,true) goal(on(D,E)) move(D,E)-> 1.0
124: [1.0] cl(E,true) cl(a,true) goal(on(D,E)) move(D,E)-> 1.0
125: [1.0] cl(b,true) cl(a,true) goal(on(D,E)) move(D,E)-> 1.0
126: [0.923077] cl(E,true) cl(c,false) goal(on(D,E)) move(D,E)-> 1.0
127: [1.0] cl(a,true) on(c,t) goal(on(D,E)) move(D,E)-> 1.0
128: [1.0] cl(D,true) on(c,t) goal(on(D,E)) move(D,E)-> 1.0
129: [0.923077] cl(c,false) on(c,t) goal(on(D,E)) move(D,E)-> 1.0
130: [1.0] on(c,t) on(b,t) goal(on(D,E)) move(D,E)-> 1.0
131: [0.909091] cl(E,true) on(b,t) goal(on(D,E)) move(D,E)-> 1.0
132: [0.909091] cl(b,true) on(b,t) goal(on(D,E)) move(D,E)-> 1.0
133: [1.0] cl(D,true) cl(b,true) goal(on(D,E)) move(D,E)-> 1.0
134: [0.923077] cl(c,false) cl(b,true) goal(on(D,E)) move(D,E)-> 1.0
135: [1.0] cl(E,true) cl(D,true) goal(on(D,E)) move(D,E)-> on(a,b)
136: [1.0] cl(E,true) cl(a,true) goal(on(D,E)) move(D,E)-> on(a,b)
137: [1.0] cl(b,true) cl(a,true) goal(on(D,E)) move(D,E)-> on(a,b)
138: [0.923077] cl(E,true) cl(c,false) goal(on(D,E)) move(D,E)-> on(a,b)
139: [1.0] cl(a,true) on(c,t) goal(on(D,E)) move(D,E)-> on(a,b)
140: [1.0] cl(D,true) on(c,t) goal(on(D,E)) move(D,E)-> on(a,b)
141: [0.923077] cl(c,false) on(c,t) goal(on(D,E)) move(D,E)-> on(a,b)
142: [1.0] on(c,t) on(b,t) goal(on(D,E)) move(D,E)-> on(a,b)
143: [0.909091] cl(E,true) on(b,t) goal(on(D,E)) move(D,E)-> on(a,b)
144: [0.909091] cl(b,true) on(b,t) goal(on(D,E)) move(D,E)-> on(a,b)
145: [1.0] cl(D,true) cl(b,true) goal(on(D,E)) move(D,E)-> on(a,b)
146: [0.923077] cl(c,false) cl(b,true) goal(on(D,E)) move(D,E)-> on(a,b)
```

Bibliography

[1] Philip E. Agre. *The Dynamic Structure of Everyday Life*. PhD thesis, Massachusetts Institute of Technology, 1985. MIT AI Lab TR-1085.

[2] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, 1987.

[3] Fahiem Bacchus, Adam J. Grove, Joseph Y. Halpern, and Daphne Koller. From statistical knowledge bases to degrees of belief. *Artificial Intelligence*, 87, 1996.

[4] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[5] Eric Baum. Toward a model of mind as a laissez-faire economy of idiots. Unpublished document, NEC Research Institute, December 1995.

[6] Eric Baum. Toward a model of mind as a laissez-gaire economy of idiots. In *Proceedings of the 13th International Conference on Machine Learning*, 1996.

[7] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

[8] Craig Boutilier, Raymond Reiter, and Bob Price. Symbolic dynamic programming for first-order MDPs. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, 2001.

[9] Ronald Brachman and Hector Levesque. The tractability of subsumption in frame-based description languages. In *Proceedings of the 4th National Conference on Artificial Intelligence (AAAI)*, 1984.

[10] Ronald Brachman and James Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2), April-June 1985.

[11] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[12] David Chapman and Leslie Pack Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, 1991.

[13] Luc De Raedt and Hendrik Blockeel. Using logical decision trees for clustering. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, 1997.

[14] Luc De Raedt and Luc Dehaspe. Clausal discovery. Technical Report CW 238, Department of Computing Science, Katholieke Universiteit Leuven, 1996.

[15] Luc De Raedt and Luc Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.

[16] Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5:142–150, 1989.

[17] Luc Dehaspe and Luc De Raedt. DLAB: A declarative language bias formalism. In *Proceedings of the 10th International Symposium on Methodologies for Intelligent Systems (ISMIS96)*, volume 1079 of *Lecture Notes in Artificial Intelligence*, pages 613–622. Springer-Verlag, 1996.

[18] Gary L. Drescher. *Made-up Minds: A Constructivist Approach to Artificial Intelligence.* The MIT Press, Cambridge, Massachusetts, 1991.

[19] Kurt Driessens, Jan Ramon, and Hendrik Blockeel. Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In *European Conference on Machine Learning*, 2001.

[20] Saso Dzeroski, Luc De Raedt, and Hendrik Blockeel. Relational reinforcement learning. In *International Workshop on Inductive Logic Programming*, pages 11–22, 1998.

[21] Saso Dzeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine Learning*, 43, April 2001.

[22] Saso Dzeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. Technical Report CW 311, Katholieke Universiteit Leuven, May 2001.

[23] Sarah Finney, Natalia H. Gardiol, Leslie Pack Kaelbling, and Tim Oates. Learning with deictic representations. Technical Report (AIM-2002-006), A.I. Lab, MIT, Cambridge, MA, 2002.

[24] Sarah Finney, Natalia H. Gardiol, Leslie Pack Kaelbling, and Tim Oates. The thing that we tried didn't work very well: Deictic representation in reinforcement learning. In *Proceedings of the 18th International Conference on Uncertainty in Artificial Intelligence*, 2002.

[25] Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning probabilistic relational models. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, 1999.

[26] Lise Getoor, Nir Friedman, Daphne Koller, and Avi Pfeffer. *Relational Data Mining*, chapter Learning Probabilistic Relational Models. Springer-Verlag, 2001. S. Dzeroski and N. Lavrac, eds.

[27] Joseph Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46, 1990.

[28] Daniel Hunter. Causality and maximum entropy updating. *International Journal of Approximate Reasoning*, 33, 1989.

[29] Manfred Jaeger. Relational Bayesian networks. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Ingelligence*, 1997.

[30] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101, 1998.

[31] Michael J. Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, 1999.

[32] Kristian Kersting and Luc De Raedt. Bayesian logic programs. In J. Cussens and A. Frisch, editors, *Proceedings of the Work-in-Progress Track at the 10th International Conference on Inductive Logic Programming*, pages 138–155, 2000.

[33] Roni Khardon. Learning action strategies for planning domains. *Artificial Intelligence*, 113(1-2):125–148, 1999.

[34] Daphne Koller and Avi Pfeffer. Learning probabilities for noisy first-order rules. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI)*, 1997.

[35] Daphne Koller and Avi Pfeffer. Probabilistic frame-based systems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI)*, 1998.

[36] Michael Littman, Richard Sutton, and Satinder Singh. Predictive representations of state. In *14th Neural Information Processing Systems*, 2001.

[37] Thomas Lukasiewicz. Probabilistic logic programming with conditional constraints. *ACM Transactions on Computational Logic (TOCL)*, July 2001.

[38] Thomas Lukasiewicz and Gabriele Kern-Isberner. Probabilistic logic programming under maximum entropy. In *Fifth European Conference on Logic Programming and Quantitative Approaches to Reasoning with Uncertainty (EC-SQARU'99)*, July 1999.

[39] Mario Martín and Héctor Geffner. Learning generalized policies in planning using concept languages. In *7th International Conference on Knowledge Representation and Reasoning (KR 2000). Morgan Kaufmann.*, 2000.

[40] David McAllester and Robert Givan. Taxonomic syntax for first-order inference. *Journal of the ACM*, 40, April 1993.

[41] Andrew K. McCallum. Instance-based utile distinctions for reinforcement learning with hidden state. In *12th International Conference on Machine Learning*, 1995.

[42] Andrew K. McCallum. *Reinforcement Learning with Selective Perception and Hidden State.* PhD thesis, University of Rochester, Rochester, New York, 1995.

[43] Stephen Muggleton. *Inductive Logic Programming.* Academic Press, London, 1992.

[44] Bernhard Nebel. Computational complexity of terminological reasoning in BACK. *Artificial Intelligence*, 34(3), 1988.

[45] Liem Ngo and Peter Haddawy. Answering queries from context-sensitive probabilistic knowledge bases. *Theoretical Computer Science*, 171, 1997.

[46] Nils J Nilsson. Probabilistic logic. *Artificial Intelligence*, 28, 1986.

[47] Hanna Pasula and Stewart Russell. Approximate inference for first-order probabilistic languages. In *17th Conference on Uncertainty in Artificial Intelligence*, 2001.

[48] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, California, 1988.

[49] Martin L. Puterman. *Markov Decision Processes.* John Wiley & Sons, New York, 1994.

[50] Hans Reichenbach. *Theory of Probability.* University of California Press, Berkeley, CA, 1949.

[51] Raymond Reiter. *Knowledge In Action: Logical Foundations for Describing and Implementing Dynamical Systems.* MIT Press, Cambridge, MA, 2001.

[52] Ron Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.

[53] Glenn Shafer. *A Mathematical Theory of Evidence.* Princeton University Press, Princeton, NJ, 1976.

[54] Wei-Min Shen. Discovery as autonomous learning from the environment. *Machine Learning*, 12, 1993.

[55] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, Cambridge, Massachusetts, 1998.

[56] Chris Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8, 1992.

[57] SungWook Yoon, Alan Fern, and Robert Givan. Inductive policy selection for first-order MDPs. In *Proceedings of the 18th International Conference on Uncertainty in Artificial Intelligence*, 2002.