Learning Probabilistic Relational Planning Rules

Hanna M. Pasula
MIT CSAIL
Cambridge, MA 02139
pasula@csail.mit.edu

Luke S. Zettlemoyer
MIT CSAIL
Cambridge, MA 02139
lsz@csail.mit.edu

Leslie Pack Kaelbling
MIT CSAIL
Cambridge, MA 02139
lpk@csail.mit.edu

Abstract

To learn to behave in highly complex domains, agents must represent and learn compact models of the world dynamics. In this paper, we present an algorithm for learning probabilistic STRIPS-like planning operators from examples. We demonstrate the effective learning of rule-based operators for a wide range of traditional planning domains.

Introduction

Imagine robots that live in the same world as we do. Such robots must be able to predict the consequences of their actions both efficiently and accurately. Programming a robot for advanced problem solving in a complicated environment is an hard problem, for which engineering a direct solution has proven difficult. Even the most sophisticated robot programming paradigms (Brooks, 1991) are difficult to scale to human-like robot behaviors.

If robots could learn to act in the world, then much of the programming burden would be removed from the robot engineer. Reinforcement learning has attempted to solve this problem, but this approach often involves learning to achieve particular goals, without gathering any general knowledge of the world dynamics. As a result, the robots can learn to do particular tasks but have trouble generalizing to new ones. If, instead, robots could learn how their actions affect the world, then they would be able to behave more robustly in a wide range of situations. This type of learning allows the robot to develop a *model* that represents the immediate effects of its action in the world. Once this model is learned, the robot could use it to behave robustly in a wide variety of situations.

There are many different ways of representing action models, but one representation, probabilistic relational rules, stands out. These rules represent situations in which actions will have a set of possible effects. Because they are probabilistic they can model actions that have more than one effect and actions that might fail often. Because they are rules, each situation can be considered independently. Rules can be used individually without having to understand the whole world. Because they are relational, they can generalize over the identities of the objects in the world. Overall, the rules we will explore in this paper, encode a set of assumptions about the world that, as we will see later, improve learning in our example domains.

Once rules have been learned, acting with them is a well-studied research problem. Probabilistic planning approaches are directly applicable (Blum & Langford, 1999) and work in this area has shown that compact representations, like rules, are essential for scaling probabilistic planning to large worlds (Boutilier, Dearden, & Goldszmidt, 2002).

Structured Worlds

When an agent is introduced into a foreign world, it must find the best possible explanation for the world's dynamics within the space of possible models it can represent. This space of models is defined by the agent's representation language. The ideal language would be able to compactly model every world the agent might encounter and no others. Any extra modeling capacity is wasted and will complicate learning since the agent will have to consider a larger space of possible models, and be more likely to overfit its experience. Choosing a good representation language provides a strong *bias* for any algorithm that will learn models in that language. In this paper we explore learning a rule-based language that makes the following assumptions about the world:

- **Frame Assumption:** When an agent takes an action in a world, anything not explicitly changed by that action stays the same.
- **Object Abstraction Assumption:** The world is made up of objects, and the effects of actions on these objects generally depend on their attributes rather than their identities.
- **Action Outcomes Assumption:** Each action can only affect the world in a small number of distinct ways. Each possible effect causes a set of changes to the world that happen together as a single *outcome*.

The first two assumptions have been captured in almost all planning representations, such as STRIPS operators (Fikes & Nilsson, 1971) and more recent variants (Penberthy & Weld, 1992). The third assumption has been made by several probabilistic planning representations, including probabilistic rules (Blum & Langford, 1999), equivalence-classes (Draper, Hanks, & Weld, 1994), and the situation calculus approach of Boutilier, Reiter, and Price (2001). The first and third assumptions might seem too rigid for some real problems: relaxing them is a topic for future work.

This paper is organized as follows. First, we describe how we represent states and action dynamics. Then, we present a rule-learning algorithm, and demonstrate its performance in three different domains. Finally, we go on to discuss some related work, conclusions, and future plans.

Representation

This section presents a formal definition of relational planning rules, as well as of the world descriptions that the rules will manipulate. Both are built using a subset of standard first-order logic that does not include functions, disjunctive connectives, or existential quantification.

State Representation

An agent's description of the world, also called the *state*, is represented syntactically as a conjunction of ground literals. Semantically, this conjunction encodes all of the important aspects of this world. The constants map to the objects in the world. The literals encode the truth values of all the possible properties of all of the objects and all of the relations that are possible between the objects.

For example, imagine a simple blocks world. The objects in this world include blocks, a table and a gripper. Blocks can be on other blocks or on the table. A block that has nothing on it is clear. The gripper can hold one block or be empty. The state description

$$\begin{array}{l} \mathit{on}(\text{B1},\text{B2}), \mathit{on}(\text{B2},\text{TABLE}), \neg\mathit{on}(\text{B2},\text{B1}), \neg\mathit{on}(\text{B1},\text{TABLE}), \\ \mathit{inhand}(\text{NIL}), \mathit{clear}(\text{B1}), \mathit{block}(\text{B1}), \mathit{block}(\text{B2}), \neg\mathit{clear}(\text{B2}), \\ \neg\mathit{inhand}(\text{B1}), \neg\mathit{inhand}(\text{B2}), \neg\mathit{block}(\text{TABLE}) \end{array} \quad (1)$$

represents a blocks world where there are two blocks in a single stack on the table. Block B1 is on top of the stack, while B2 is below B1 and on the TABLE.

Action Representation

Rule sets model the action dynamics of the world. The rule set we will explore in this section models how the simple blocks world changes state as it is manipulated by a robot arm. This arm can attempt to pick up blocks and put them on other blocks or the table. However, the arm is faulty, so its actions can succeed, fail to change the world, or fail by knocking the block onto the table. Each of these possible outcomes changes several aspects of the state. We begin the section by presenting the rule set syntax. Then, the semantics of rule sets is described procedurally.

Rule Set Syntax A rule set, $\mathbf{R}$, is a set of rules. Each $r \in \mathbf{R}$ is a four-tuple, $(r_A, r_C, r_{\mathbf{O}}, r_P)$. The rule's *action*, r_A, is a positive literal. The *context*, r_C, is a conjunction of literals. The *outcome set*, $r_{\mathbf{O}}$, is a non-empty set of outcomes, where each *outcome* $o \in r_{\mathbf{O}}$ is a conjunction of literals that defines a deterministic mapping from previous states to successor states, $f_o : S \to S$, as described shortly. Finally, r_P is a

$$\mathit{pickup}(X,Y) : \mathit{on}(X,Y), \mathit{clear}(X), \mathit{inhand}(\text{NIL}), \mathit{block}(Y)$$
$$\rightarrow \begin{cases} .7: & \mathit{inhand}(X), \neg\mathit{clear}(X), \neg\mathit{inhand}(\text{NIL}), \\ & \neg\mathit{on}(X,Y), \mathit{clear}(Y) \\ .2: & \mathit{on}(X,\text{TABLE}), \neg\mathit{on}(X,Y), \mathit{clear}(Y) \\ .1: & \text{no change} \end{cases}$$
$$\mathit{pickup}(X,\text{TABLE}) : \mathit{on}(X,\text{TABLE}), \mathit{clear}(X), \mathit{inhand}(\text{NIL})$$
$$\rightarrow \begin{cases} .66: & \mathit{inhand}(X), \neg\mathit{clear}(X), \neg\mathit{inhand}(\text{NIL}), \\ & \neg\mathit{on}(X,\text{TABLE}) \\ .34: & \text{no change} \end{cases}$$
$$\mathit{puton}(X,Y) : \mathit{clear}(Y), \mathit{inhand}(X), \mathit{block}(Y)$$
$$\rightarrow \begin{cases} .7: & \mathit{inhand}(\text{NIL}), \neg\mathit{clear}(Y), \neg\mathit{inhand}(X), \\ & \mathit{on}(X,Y), \mathit{clear}(X) \\ .2: & \mathit{on}(X,\text{TABLE}), \mathit{clear}(X), \mathit{inhand}(\text{NIL}), \\ & \neg\mathit{inhand}(X) \\ .1: & \text{no change} \end{cases}$$
$$\mathit{puton}(X,\text{TABLE}) : \mathit{inhand}(X)$$
$$\rightarrow \begin{cases} .8: & \mathit{on}(X,\text{TABLE}), \mathit{clear}(X), \mathit{inhand}(\text{NIL}), \\ & \neg\mathit{inhand}(X) \\ .2: & \text{no change} \end{cases}$$

Figure 1: Four relational rules that model the action dynamics of a simple blocks world.

discrete distribution over the set of outcomes $r_{\mathbf{O}}$. Rules may contain variables; however, every variable appearing in r_C or $r_{\mathbf{O}}$ must also appear in r_A. Figure 1 shows a rule set with four rules for the blocks world domain.

A rule set is a full model of a world's action dynamics. This model can be used to predict the effects of an action, a, when it is performed in a specific state, s, as well as to determine the probability that a transition from s to s' occurred when a was executed. When using the rule set to do either, we must first *select* the rule which governs the change for the state-action pair, (s, a): the $r \in \mathbf{R}$ that *covers* (s, a).

Rule Selection The rule that covers (s, a) is found by considering each candidate $r \in \mathbf{R}$ in turn, and testing it using a three-step process that ensures that r's action models a, that r's context is satisfied by s, and that r is well-formed given a. The first step attempts to unify r_A with a. A successful unification returns an *action substitution* θ that maps the variables in r_A to the corresponding constants in a. This substitution is then applied to r; because of our assumption that all the variables in r are in r_A, this application is guaranteed to ground all literals in r. The second step checks whether the context r_C, when grounded using θ, is a subset of s. Finally, the third step tests the ground outcomes for contradictions. A contradiction occurs when the grounding produces an outcome containing both a literal and its negation.

As an example, imagine an agent wants to predict the effects of executing *pickup*(B1, B2) in the world described in Equation 1 given the model represented by the rule set in Figure 1. The action unifies with the action of the first rule, producing the substitution $\theta = \{X/\text{B1}, Y/\text{B2}\}$; fails to unify with the second rule's action, because B2 doesn't equal TABLE; and fails to unify with the remaining rules since they have different action predicates. When we ap-

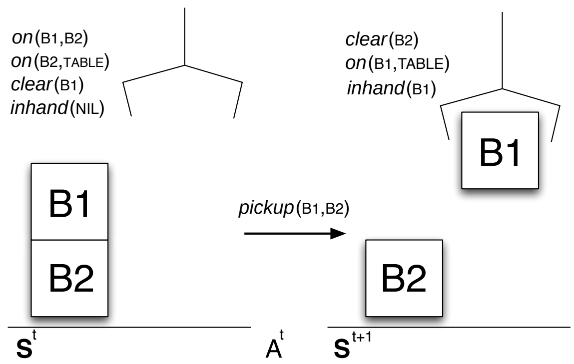

Figure 2: Two subsequent states of the blocks world with two blocks. The pictured states are represented by the neighboring lists of true propositions. Everything not listed is false. The action *pickup*(B1, B2) was performed successfully.

ply θ to the first rule, we can see that its outcomes contain no contradictions; note, however, that if the action a had been *pickup*(B1, B1) then the first outcome would have contained one. The context, meanwhile, becomes $\{$*on*(B1, B2), *clear*(B1), *inhand*(NIL), *block*(B2)$\}$. Since this set is a subset of the state description in Equation 1, the first rule passes all three tests.

In general, the state-action pair (s, a) could be covered by zero, one, or many rules. If there are zero rules, we can fall back on the frame assumption. A rule set is *proper* if every possible state is covered by at most one rule. All of the rule sets in this paper are assumed to be proper.

Successor State Construction An agent can predict the effects of executing action a in state s as follows. If no $r \in \mathbf{R}$ covers (s, a), then, because of the frame assumption, the successor state s' is taken to be simply s. Given an r, an outcome $o \in r_{\mathbf{O}}$ is selected by sampling from r_P and ground using θ. The next state, s', is constructed by applying $f_o(s)$, which combines o with those literals in s that are not directly contradicted by o.

Figure 2 shows an example where the first outcome from the first rule in Figure 1 predicts that effects of *pickup*(B1, B2) to the state of Equation 1. The states are represented pictorially and annotated with only the true literals; all others are assumed to be false. As the outcome predicts, *inhand*(B1) and *clear*(B2) become true while *on*(B1, B2), *clear*(B1), and *inhand*(NIL) become false.

Likelihood Estimation The general probability distribution $P(S'|S, A, R)$ is defined as follows. If no rule in R covers (S, A), then this probability is 1.0 iff $s' = s$. Otherwise, it is defined as

$$\begin{aligned} P(S'|S,A,r) &= \sum_{o \in r_O} P(S', o|S, A, r) \\ &= \sum_{o \in r_O} P(S'|o, S, A, r) P(o|S, A, r) \quad (2) \end{aligned}$$

where r is the covering rule, $P(o|S, A, r)$ is $r_P(o)$, and $P(S'|o, S, A, r)$ is deterministic: it is 1.0 iff $f_o(S) = S'$.

We say that an outcome *covers* an example (s, a, s') if $f_o(s) = s'$. Now, the probability of S' is the sum of all the outcomes in r that cover the transition from S to S'. Notice that a specific S and o uniquely determine S'. This fact guarantees that, as long as r_P is a well-defined distribution, so is $P(S'|S, A, r)$.

Overlapping Outcomes Notice that $P(S'|S, A, r)$ is using the set of outcomes as a hidden variable. This introduces the phenomenon of *overlapping outcomes*. Outcomes overlap when, given a rule r that covers the initial state and action (s, a), several of the outcomes $r_{\mathbf{O}}$ could be used to describe the transition to the successor state s'. As an example, consider a rule for painting blocks,

$$\textit{paint}(X) : \textit{inhand}(X), \textit{block}(X)$$
$$\rightarrow \begin{cases} .8 : \textit{painted}(X), \textit{wet} \\ .2 : \text{no change} \end{cases}$$

When this rule is used to model the transition caused by the action *paint*(B1) in an initial state that contains *wet* and *painted*(B1), there is only one possible successor state: the one where no change occurs, and *painted*(B1) remains true. Both the outcomes describe this one successor state, and so we must sum their probabilities to recover that state's total probability.

Learning

In this section, we describe how a rule set defining the distribution $P(S'|S, A, R)$ may be learned from a training set $\mathbf{D} = D_1 \ldots D_{|\mathbf{D}|}$. Every example $(s, a, s') \in \mathbf{D}$ represents a single action execution in the world, consisting of a previous state s, an action a, and a successor state s'.

The algorithm involves three levels of greedy search: an outermost level, *LearnRules*, which searches through the space of rule sets; a middle level, *InduceOutcomes* which, given a context and an action, constructs the best set of outcomes; and an innermost level, *LearnParameters*, which learns a distribution over a given set of outcomes. These three levels are detailed in the next three sections.

Learning Rules

LearnRules performs a greedy search in the space of proper rule sets. We define a rule set as proper with respect to a data set $\mathbf{D}$ as a set of rules $\mathbf{R}$ that includes exactly one rule that is applicable to every example $D \in \mathbf{D}$ in which some change occurs, and that does not includes any rules that are applicable to no examples.

Scoring Rule Sets As it searches, *LearnRules* must judge which rule sets are the most desirable. This is done with the help of a scoring metric, $S(\mathbf{R}) =$

$$\sum_{(s,a,s') \in \mathbf{D}} \log(P(s'|s, a, \mathbf{R})) - \alpha \sum_{r \in \mathbf{R}} PEN(r) \quad (3)$$

which favors rule sets that assign high likelihood to the data and penalizes rule sets that are overly complex. The complexity of a rule $PEN(r)$ is defined simply as $|r_C| + |r_{\mathbf{O}}|$. The first part of this term penalizes long contexts; the second part penalizes for having too many outcomes. We have chosen this penalty for its simplicity, and also because it performed no worse than any other penalty term we tested in informal experiments. The scaling parameter α is set to 0.5 in our experiments, but it could also be set using cross-validation on a hold-out dataset or some other principled technique.

Initializing the Search We initialize the search by creating the most specific rule set: one that contains, for every unique (s, a) pair in the data, a rule with $r_C = s$ and $r_A = a$. Because the context contains the whole world state, this is the only rule that could possibly cover the relevant examples, and so this rule set is guaranteed to be proper.

Search Operators Given a starting point, *LearnRules* repeatedly finds and applies the operator that will increase the score of the current rule set the most. There are four types of search operators available, based on the four basic syntactic operations used for rule search in inductive logic programming (Lavrač & Džeroski, 1994). Each operator selects a rule r, removes it from the rule set, and creates one or more new rules, which are then introduced back into the rule set in a manner that ensures the rule set remains proper. How this is done for each operator is described below. In each case, the new rules are created by choosing an r_C and an r_A and calling *InduceOutcomes* to complete r.

There are two possible ways to generalize a rule: a literal can be removed from the context, or a constant can be replaced with a variable. Given an old rule, the first generalization operator simply shortens the context by one while keeping the action the same; the second generalization operator picks one of the constant arguments of the action, invents a new variable to replace it, and substitutes that variable for every instance of the original constant both in the action and the context.[1] Both operators then call *InduceOutcomes* to complete the new rule, which is added to the set. At this point, *LearnRules* must ensure that the rule set remains proper. Generalization may increase the number of examples covered by a rule, and so make some of the other rules redundant. The new rule replaces these other rules, removing them from the set. Since this removal can leave some training examples with no rule, new, maximally specific rules are created to cover them.

There are also two ways to specialize a rule: a literal can be added to the context, or a variable can be replaced with a constant. The first specialization operator picks an atom that is absent from the old rule's context. It then constructs two new enlarged contexts, one containing a positive instance of this atom, and one containing a negative instance. A rule is filled in for each of the contexts, with the action remaining the same. The second specialization operator picks one of the variable arguments of the action, and creates a new rule for every possible constant by substituting the constant for the variable in both the action and the body of the rule, and calling *InduceOutcomes* as usual. In either case, the new rules are then introduced into the rule set, and *LearnRules* must, again, ensure that it remains proper. This time the only concern is that some of the new rules might cover no training examples; such rules are left out of the rule set.

All these operators, just like the ILP operators that motivated them (Lavrač & Džeroski, 1994), can be used to create any possible rule set. There are also other advanced rule set search operators, such as least general generalization (Plotkin, 1970), which might be modified to create operators that allow *LearnRules* to search the planning rule set space more efficiently.

LearnRules's search strategy has one large drawback; the set of rules which is learned is only guaranteed to be proper on the training set and not on testing data. Solving this problem, possibly with approaches based on relational decision trees (Blockeel & De Raedt, 1998), is an important area for future work.

[1]During learning, we always introduce variables aggressively wherever possible, based on the intuition that if it is important for any of them to remain a constant, this should become apparent through the other training examples.

Inducing Outcomes

The effectiveness and efficiency of the *LearnRules* algorithm are limited by those of the *InduceOutcomes* subprocedure, which is called every time a new rule is constructed. Formally, the problem of inducing outcomes for a rule r is the problem of finding a set of outcomes $r_{\mathbf{O}}$ and a corresponding set of parameters r_P which maximize the score,

$$\sum_{(s,a,s') \in \mathbf{D}_r} \log(P(s'|s,a,r)) - \alpha PEN(r),$$

where $\mathbf{D}_r$ is the set of examples such that r covers (s, a). This score is simply r's contribution to the overall rule set score of Equation 3.

In general, outcome induction is NP-hard (Zettlemoyer, Pasula, & Kaelbling, 2003). *InduceOutcomes* uses greedy search through a restricted subset of possible outcome sets: those that are *proper* on the training examples, where an outcome set is proper if every training example has at least one outcome that covers it and every outcome covers at least one training example. Two operators, described below, move through this space until there are no more immediate moves that improve the rule score. For each set of outcomes it considers, *InduceOutcomes* calls *LearnParameters* to supply the best r_P it can.

Initializing the Search The initial set of proper outcomes is created by, for each example, writing down the set of atoms that changed truth values as a result of the action, and then creating an outcome to describe every set of changes observed in this way.

As an example, consider the coins domain. Each coins world contains n coins, which can be showing either heads or tails. The action *flip-coupled*, which has no context and no arguments, flips all of the coins to heads half of the time and otherwise flips them all to tails. A set of training data for learning outcomes with two coins might look like part (a) of Figure 3 where $h(C)$ stands for $heads(C)$, $t(C)$ stands for $\neg heads(C)$, and $s \rightarrow s'$ is part of an (s, a, s') example where $a =$ *flip-coupled*. Given this data, the initial set of outcomes has the four entries in part (b) of Figure 3.

$$
\begin{array}{ll}
D_1 = t(c1), h(c2) \rightarrow h(c1), h(c2) & O_1 = \{h(c1)\} \\
D_2 = h(c1), t(c2) \rightarrow h(c1), h(c2) & O_2 = \{h(c2)\} \\
D_3 = h(c1), h(c2) \rightarrow t(c1), t(c2) & O_3 = \{t(c1), t(c2)\} \\
D_4 = h(c1), h(c2) \rightarrow h(c1), h(c2) & O_4 = \{\text{no change}\} \\
\text{(a)} & \text{(b)}
\end{array}
$$

Figure 3: (a) Possible training data for learning a set of outcomes. (b) The initial set of outcomes that would be created from the data in (a).

Search Operators *InduceOutcomes* uses two search operators. The first is an add operator, which picks a pair of non-contradictory outcomes in the set and adds in a new outcome based on their conjunction. For example, it might pick O_1 and O_2 and combine them, adding a new outcome $O_5 = \{h(c1), h(c2)\}$ to the set. The second is a remove operator that drops an outcome from the set. Outcomes can only be dropped if they were overlapping with other outcomes on every example they cover, otherwise the outcome set would not remain proper. Sometimes, *LearnParameters* will return zero probabilities for some of the outcomes. Such outcomes are removed from the outcome set, since they contribute nothing to the likelihood, and only add to the complexity. This optimization greatly improves the efficiency of the search.

In the outcomes of Figure 3, O_4 can be immediately dropped since it covers only D_4, which is also covered by both O_1 and O_2. If we imagine that $O_5 = \{h(c1), h(c2)\}$ has been added with the add operator, then O_1 and O_2 could also be dropped since O_5 covers D_1, D_2, and D_3. This would, in fact, lead to the optimal set of outcomes for the training examples in Figure 3.

Our coins world example has no context and no action. Handling contexts and actions with constant arguments is easy, since they simply restrict the set of training examples the outcomes have to cover. However, when a rule has variables among its action arguments, *InduceOutcomes* must be able to introduce those variables into the appropriate places in the outcome set. This variable introduction is achieved by applying the inverse of the action substitution to each example's set of changes while computing the initial set of outcomes. So, for example, if *InduceOutcomes* were learning outcomes for the action *flip(X)* that flips a single coin, our initial outcome set would be $\{O_1 = \{$h(X)$\}$, $O_2 = \{$t(X)$\}$, $O_3 = \{$no change$\}\}$ and search would progress as usual from there.

Notice that an outcome is always equal to the union of the set of literals that change in every training example it covers. This fact ensures that every proper outcome can be made by merging outcomes from the initial outcome set. *InduceOutcomes* can, in theory, find any set of proper outcomes.

Learning Parameters

Given a rule r with a context r_C and a set of outcomes $r_{\mathbf{O}}$, all that remains to be learned is the distribution over the outcomes, r_P. *LearnParameters* learns the distribution that maximizes the rule score: this will be the distribution that maximizes the log likelihood of the examples $\mathbf{D}_r$ as given by

$$
\begin{aligned}
& \sum_{(s,a,s') \in \mathbf{D}_r} \log(P(s'|s,a,r)) \\
= & \sum_{(s,a,s') \in \mathbf{D}_r} \log \left(\sum_{\{o|D \in \mathbf{D}_o\}} r_P(o) \right) \qquad (4)
\end{aligned}
$$

where $\mathbf{D}_o$ is the set of examples covered by outcome o. When every example is covered by a unique outcome, the problem of minimizing L is relatively simple. Using a Lagrange multiplier to enforce the constraint that r_P must sum to 1.0, the partial derivative of L with respect to $r_P(o)$ is then $|\mathbf{D}_o|/r_P(o) - \lambda$, and $\lambda = |\mathbf{D}|$, so that $r_P(o) = |\mathbf{D}_o|/|\mathbf{D}|$. The parameters can be estimated by calculating the percentage of the examples that each outcome covers.

However, in general, the rule could have overlapping outcomes. In this case, the partials would have sums over os in the denominators and there is no obvious closed-form solution; estimating the maximum likelihood parameters is a nonlinear programming problem. Fortunately, it is an instance of the well-studied problem of maximizing a convex function over a probability simplex. Several gradient ascent algorithms with guaranteed convergence can be found (Bertsekas, 1999). *LearnParameters* uses the *conditional gradient method*, which works by, at each iteration, moving along the axis with the maximal partial derivative. The step-sizes are chosen using the Armijo rule (with the parameters $s = 1.0$, $\beta = 0.1$, and $\sigma = 0.01$.) The search converges when the improvement in L is very small, less than 10^{-6}. If problems are found where this method converges too slowly, one of the other methods could be tried.

Experiments

This section describes experiments that demonstrate that the rule learning algorithm is robust. We first describe our test domains and then we report the experiments we performed.

Domains

The experiments we performed involve learning rules for the domains which are briefly described in the following sections. Please see the technical report by Zettlemoyer et al. (2003) for a formal definition of these domains.

Coin Flipping In the coin flipping domain, n coins are flipped using three atomic actions: *flip-coupled*, which, as described previously, turns all of the coins to heads half of the time and to tails the rest of the time; *flip-a-coin*, which picks a random coin uniformly and then flips that coin; and *flip-independent*, which flips each of the coins independently of each other. Since the contexts of all these actions are empty, every rule set contains only a single rule and the whole problem reduces to outcome induction.

Slippery Gripper The slippery gripper domain, inspired by the work of Draper et al. (1994), is a blocks world with a simulated robotic arm, which can be used to move the blocks around on a table, and a nozzle, which can be used to paint the blocks. Painting a block might cause the gripper to become wet, which makes it more likely that it will fail to manipulate the blocks successfully; fortunately, a wet gripper can be dried.

Trucks and Drivers Trucks and drivers is a logistics domain, adapted from the 2002 AIPS international planning competition (AIPS, 2002), with four types of constants. There are trucks, drivers, locations, and objects. Trucks, drivers and objects can all be at any of the locations. The locations are connected with paths and links. Drivers can board and exit trucks. They can drive trucks between locations that are linked. Drivers can also walk, without a truck, between locations that are connected by paths. Finally, objects can be loaded and unloaded from trucks.

Most of the actions are simple rules which succeed or fail to change the world. However, the walk action has an interesting twist. When drivers try to walk from one location to another, they succeed most of the time, but some of the time they arrive at a randomly chosen location that is connected by some path to their origin location.

Inducing Outcomes

Before we investigate learning full rule sets, we consider how the *InduceOutcomes* sub-procedure performs on some canonical problems in the coin flipping domain. We do this to evaluate *InduceOutcomes* in isolation, and demonstrate its performance on overlapping outcomes. In order to do so, a rule was created with an empty context and passed to *InduceOutcomes*. Table 1 contrasts the number of outcomes in the initial outcome set with the number eventually learned by *InduceOutcomes*. These experiments used 300 randomly created training examples; this rather large training set gave the algorithm a chance of observing many of the possible outcomes, and so ensured that the problem of finding a smaller, optimal, proper outcome set was difficult.

Given n coins, the optimal number of outcomes for each action is well defined. *flip-coupled* requires 2 outcomes, *flip-a-coin* requires $2n$, and *flip-independent* requires 2^n. In this sense, *flip-independent* is an action that violates our basic structural assumptions about the world, *flip-a-coin* is a difficult problem, and *flip-coupled* behaves like the sort of action we expect to see frequently. The table shows that *InduceOutcomes* can learn the latter two cases, the ones it was designed for, but that actions where a large number of independent changes results in an exponential number of outcomes are beyond its reach.

	Number of Coins				
	2	3	4	5	6
flip-coupled initial	7	15	29.5	50.75	69.75
flip-coupled final	2	2	2	2	2
flip-a-coin initial	5	7	9	11	13
flip-a-coin final	4	6.25	8	9.75	12
flip-independent initial	9	25	47.5	-	-
flip-independent final	5.5	11.25	20	-	-

Table 1: The decrease in the number of outcomes found while inducing outcomes in the n-coins world. Results are averaged over four runs of the algorithm. The blank entries did not finish running in reasonable amounts of time.

Learning Rule Sets

Now that we have seen that *InduceOutcomes* can learn rules that don't require an exponential number of outcomes, let us investigate how *LearnRules* performs.

The experiments perform two types of comparisons. The first shows that propositional rules can be learned more effectively than *Dynamic Bayesian Networks* (DBNs), a well-known propositional representation that has traditionally been used to learn world dynamics. The second shows that relational rules outperform propositional ones.

These comparisons are performed for four actions. The first two, paint and pickup, are from the slippery gripper domain while the second two, drive and walk, are from the trucks and drivers domain. Each action presents different challenges for learning. *Paint* is a simple action that has overlapping outcomes. *Pickup* is a complex action that must be represented by more than one planning rule. *Drive* is a simple action that has four arguments. Finally, *walk* is a complicated action uses the path connectivity of the world in its noise model for lost pedestrians. The slippery gripper actions were performed in a world with four blocks. The trucks and driver actions were performed in a world with two trucks, two drivers, two objects, and four locations.

All of the experiments use examples, $(s, a, s') \in \mathbf{D}$, generated by randomly constructing a state s, randomly picking the arguments of the action a, and then executing the action in the state to generate s'. The distribution used to construct s is biased to guarantee that, in approximately half of the examples, a has a chance to change the state: that is, that a hand-constructed rule applies to s.

Thus, the experiments in this paper ignore the problems an agent would face if it had to generate data by exploring the world.

After training on a set of training examples $\mathbf{D}$, the models are tested on a set of test examples $\mathbf{E}$ by calculating the average *variational distance* between the true model P and an estimate $\hat{P}$,

$$VD(P, \hat{P}) = \frac{1}{|\mathbf{E}|} \sum_{E \in \mathbf{E}} |P(E) - \hat{P}(E)|.$$

Variational distance is a suitable measure because it favors similar distributions and is well-defined when a zero probability event is observed, which can happen when a rule is learned from sparse data and doesn't have as many outcomes as it should.

Comparison to DBNs To compare *LearnRules* to DBN learning, we forbid variable abstraction, thereby forcing the rule sets to remain propositional during learning. The BN learning algorithm of Friedman and Goldszmidt (1998), which uses decision trees to represent its conditional probability distributions, is compared to this restricted *LearnRules* algorithm in Figure 4.

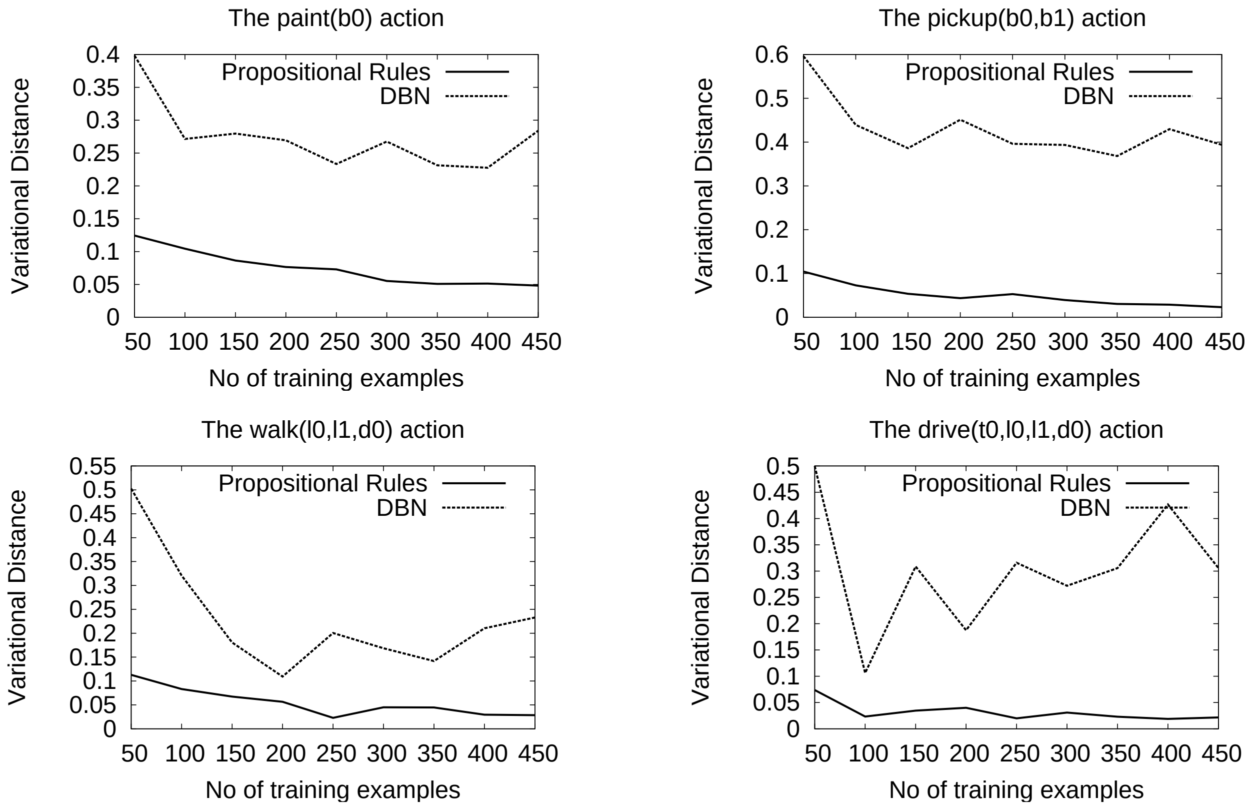

Figure 4: Variational distance as a function of the number of training examples for DBNs and propositional rules. The results are averaged over ten trials of the experiment. The test set size was 300 examples.

Notice that the propositional rules consistently outperform DBNs. In the four blocks world DBN learning consistently gets stuck in local optima and never learns a satisfactory model. We ran other experiments in the simpler two blocks world which showed DBN learning reasonable (VD$<$.07) models in 7 out of 10 trials and generalizing better than the rules in one trial.

The Advantages of Abstraction The second set of experiments demonstrates that when *LearnRules* is able to use variable abstraction, it outperforms the propositional version. Figure 5 shows that the full version consistently outperforms the restricted version.

Also, observe that the performance gap grows with the number of arguments that the action has. This result should not be particularly surprising. The abstracted representation is significantly more compact. Since there are fewer rules, each rule has more training examples and the abstracted representation is significantly more robust in the presence of data sparsity.

We also performed another set of experiments, showing that relational models can be trained in blocks worlds with a small number of blocks and tested in much larger worlds. Figure 6 shows that there is no real increase in test error as the size of the test world is increased. This is one of the major attractions of a relational representation.

Discussion The experiments of this section should not be surprising. Planning rules were designed to efficiently encode the dynamics of the worlds used in the experiments. If they couldn't outperform more general representations and learning algorithms, there would be a serious problem.

However, these experiments are still an important validation that *LearnRules* is a robust algorithm that does leverage the bias that it was designed for. Because no other algorithms have been designed with this bias, it would be difficult to demonstrate anything else. Ultimately, the question of whether this bias is useful will depend on its applicability in real domains of interest.

Related Work

The problem of learning deterministic action models, which is closely related to our work, is well-studied. There are several systems which are, in one way or another, more advanced than ours. The LIVE system (Shen & Simon, 1989) learns operators with quantified variables while incrementally exploring the world. The EXPO system (Gil, 1993, 1994) also learns incrementally, and uses special heuristics to design experiments to test the operators. However, both of these system assume that the learned models are completely deterministic and would fail in the presence of noise.

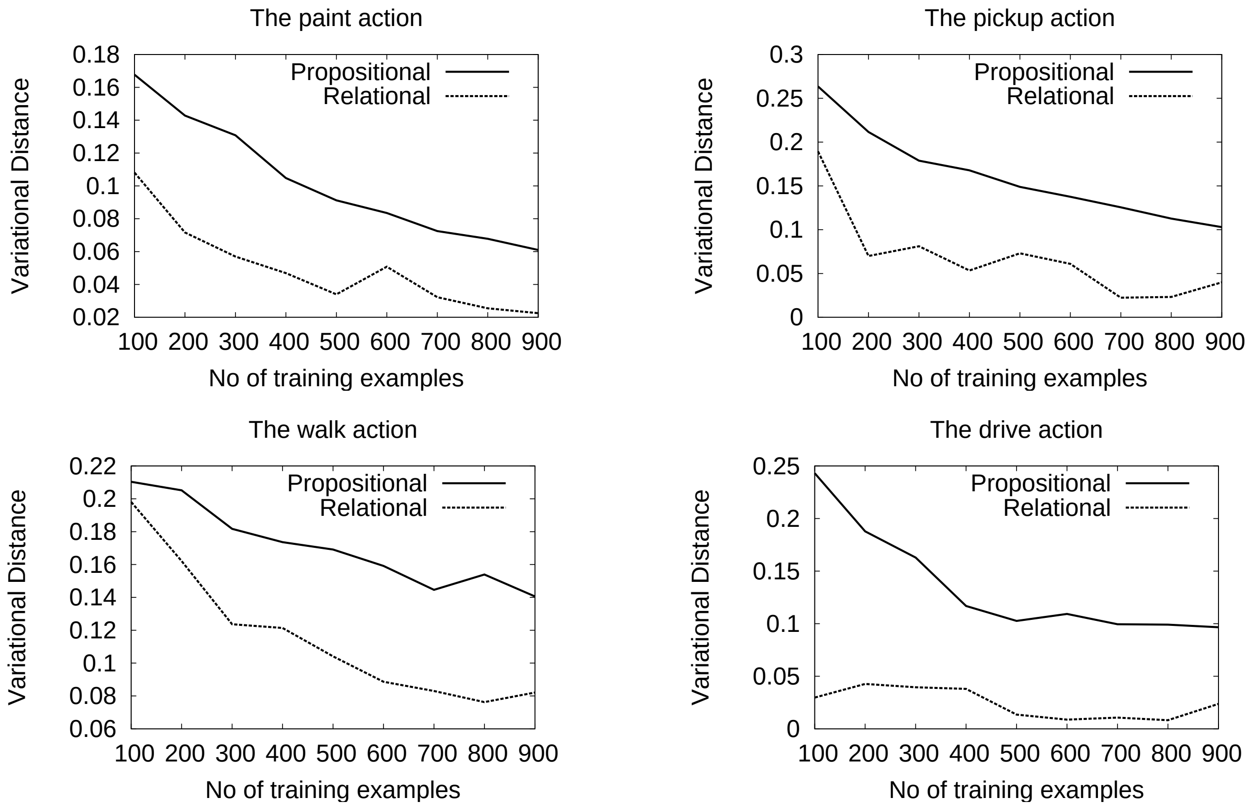

Figure 5: Variational distance as a function of the number of training examples for propositional and relational rules. The results are averaged over ten trials of the experiment. The test set size was 400 examples.

The TRAIL system (Benson, 1996) limits its operators to a slightly-extended version of Horn clauses so that it can apply ILP learning which is robust to noise. Moreover, TRAIL models continuous actions and real-valued fluents, which allow it to represent the most complex models to date, including knowledge used to pilot a realistic flight simulator.

Our search through the space of rule sets, *LearnRules*, is a simple extension of these deterministic rule learning techniques. However, our *InduceOutcomes* and *EstimateParams* algorithms are novel. No previous work has represented the action effects using a set of alternative outcomes. This is an important advance since deterministic operators cannot model even the simplest probabilistic actions, such as flipping a coin. Even in nearly-deterministic domains, actions can have unlikely effects that are worth modeling explicitly.

Literature on learning probabilistic planning rules is relatively sparse: we know of only one method for learning operators of this type (Oates & Cohen, 1996). Their rules are factored and can apply in parallel. However, their representation is strictly propositional and it only allows each rule to contain a single outcome.

Probabilistic world dynamics are commonly represented using graphical models, such as Bayesian networks (BNs) (Friedman & Goldszmidt, 1998), a propositional representation, and probabilistic relational models (PRMs) (Getoor, 2001), a relational generalization. However, these representations do not make any assumptions tailored towards representing action dynamics. In this paper, we test the usefulness of such assumptions by comparing BN learning to our propositional rule-learning algorithm. We would like to have included an comparison to PRM learning but were unable to because of various technical limitations of that representation (Zettlemoyer et al., 2003).

Conclusions and Future Work

Our experiments show that biasing representations towards the structure of the world they will represent significantly improves learning. The natural next question is: how do we bias robots so they can learn in the real world?

Planning operators exploit a general principle in modeling agent-induced change in world dynamics: each action can only have a few possible outcomes. In the simple examples in this paper, this assertion was exactly true in the underlying world. In real worlds, this assertion may not be exactly true, but it can be a powerful approximation. If we are able to abstract sets of resulting states into a single generic "outcome," then we can say, for example, that one outcome of trying to put a block on top of a stack is that the whole stack falls over. Although the details of how it falls over can be very different from instance to instance, the import of its having fallen over is essentially the same.

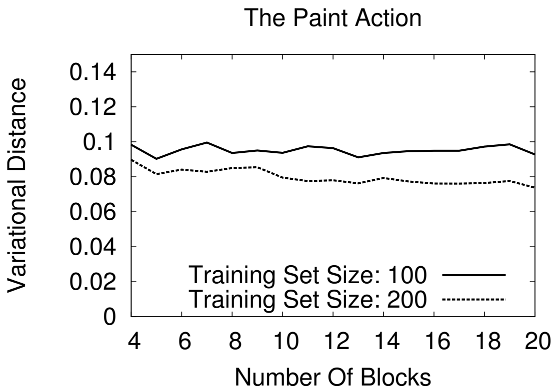

Figure 6: Variational distance of a relational rule set trained in a world four-block world, as a function of the number of blocks in the worlds on which it was tested. Results are given for three different training set sizes. The testing sets were the same size as the training sets.

An additional goal in this work is that of operating in extremely complex domains. In such cases, it is important to have a representation and a learning algorithm that can operate incrementally, in the sense that it can represent, learn, and exploit some regularities about the world without having to capture all of the dynamics at once. This goal originally contributed to the use of rule-based representations.

A crucial further step is the generalization of these methods to the partially observable case. Again, we cannot hope to come up with a general efficient solution for the problem. Instead, algorithms that leverage world structure should be able to obtain good approximate models efficiently.

References

AIPS (2002). International planning competition.. http://www.dur.ac.uk/d.p.long/competition.html.

Benson, S. (1996). *Learning Action Models for Reactive Autonomous Agents*. Ph.D. thesis, Stanford University.

Bertsekas, D. P. (1999). *Nonlinear Programming*. Athena Scientific.

Blockeel, H., & De Raedt, L. (1998). Top-down induction of first-order logical decision trees. *Artificial Intelligence*, *101*(1-2).

Blum, A., & Langford, J. (1999). Probabilistic planning in the graphplan framework. In *Proceedings of the Fifth European Conference on Planning*.

Boutilier, C., Dearden, R., & Goldszmidt, M. (2002). Stochastic dynamic programming with factored representations. *Artificial Intelligence*, *121*(1-2).

Boutilier, C., Reiter, R., & Price, B. (2001). Symbolic dynamic programming for first-order MDPs. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*.

Brooks, R. A. (1991). Intelligence without representation. *Artificial Intelligence*, *47*.

Draper, D., Hanks, S., & Weld, D. (1994). Probabilistic planning with information gathering and contingent execution. In *Proceedings of the Second International Conference on AI Planning and Scheduling*.

Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*(2).

Friedman, N., & Goldszmidt, M. (1998). Learning Bayesian networks with local structure. In *Learning and Inference in Graphical Models*.

Getoor, L. (2001). *Learning Statistical Models From Relational Data*. Ph.D. thesis, Stanford University.

Gil, Y. (1993). Efficient domain-independent experimentation. In *Proceedings of the Tenth International Conference on Machine Learning*.

Gil, Y. (1994). Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the Eleventh International Conference on Machine Learning*.

Lavrač, N., & Džeroski, S. (1994). *Inductive Logic Programming Techniques and Applications*. Ellis Horwood.

Oates, T., & Cohen, P. R. (1996). Searching for planning operators with context-dependent and probabilistic effects. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*.

Penberthy, J. S., & Weld, D. (1992). UCPOP: A sound, complete partial-order planner for ADL. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*.

Plotkin, G. (1970). A note on inductive generalization. *Machine Intelligence*, *5*.

Shen, W.-M., & Simon, H. A. (1989). Rule creation and rule learning through environmental exploration. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*.

Zettlemoyer, L., Pasula, H., & Kaelbling, L. (2003). Learning probabilistic relational planning rules. *MIT CSAIL Technical Report*.