Learning Planning Rules in Noisy Stochastic Worlds

Luke S. Zettlemoyer
MIT CSAIL
lsz@csail.mit.edu

Hanna M. Pasula
MIT CSAIL
pasula@csail.mit.edu

Leslie Pack Kaelbling
MIT CSAIL
lpk@csail.mit.edu

Abstract

We present an algorithm for learning a model of the effects of actions in noisy stochastic worlds. We consider learning in a 3D simulated blocks world with realistic physics. To model this world, we develop a planning representation with explicit mechanisms for expressing object reference and noise. We then present a learning algorithm that can create rules while also learning derived predicates, and evaluate this algorithm in the blocks world simulator, demonstrating that we can learn rules that effectively model the world dynamics.

Introduction

One of the goals of artificial intelligence is to build systems that can act in complex environments as effectively as humans do: to perform everyday human tasks, like making breakfast or unpacking and putting away the contents of an office. Any robot that hopes to solve these tasks must be an integrated system that perceives the world, understands it in an, at least naively, human manner, and commands motors to effect changes to it. Unfortunately, the current state of the art in reasoning, planning, learning, perception, locomotion, and manipulation is so far removed from human-level abilities that we cannot even contemplate working in an actual domain of interest. Instead, we choose to work in domains that are its almost ridiculously simplified proxies.

One popular such proxy, used since the beginning of work in AI planning (Fikes & Nilsson 1971) is a world of stacking blocks. This *blocks world* is typically formalized in some version of logic, using predicates such as *on*(a, b) and *clear*(a) to describe the relationships of the blocks to one another. Blocks are always very neatly stacked; they don't fall into jumbles. In this paper, we will work in a slightly less ridiculous version of the blocks world, one constructed using a three-dimensional rigid-body dynamics simulator (ODE 2004). An example domain configuration is shown in Figure 1. In this simulated blocks world, blocks are not always in tidy piles; blocks sometimes slip out of the gripper; and piles sometimes fall over. We would like to learn models that enable effective action in this world.

Unfortunately, previous approaches to action model learning cannot solve this problem. The algorithms that learn deterministic rule descriptions (Shen & Simon 1989; Gil 1994; Wang 1995) have limited applicability in a stochastic world. One approach (Pasula, Zettlemoyer, & Kaelbling 2004) has extended those methods to learn probabilistic STRIPS rules, but this representation cannot cope with the complexity of the simulated blocks world. The work of Benson (1996), which extends a deterministic ILP (Lavrač & Džeroski 1994) learning algorithm that is robust to noise in the training set, would, perhaps, come the closest, but it lacks the ability to handle complex action effects such as piles of blocks falling over. We address this challenge by developing a more flexible algorithm that creates models that include mechanisms for referring to objects and abstracting away rare or highly complex action outcomes, and also invents new concepts that help determine when actions will have different effects.

When learning these models, we assume that the learner has access to training examples that show how the world changes when an action is executed. The learning problem is then one of density estimation. The learner must estimate the distribution over next states of the world that executing an action will cause.

In the rest of this paper, we first present our representation, showing how these extensions are added to probabilistic STRIPS rules. Then, we develop a learning algorithm for these rules. Finally, we evaluate these learned rules in the simulated blocks world.

Representation

This section describes representations for the set $\mathcal{S}$ of possible states of the world, the set $\mathcal{A}$ of possible actions the agent can take, and the probabilistic transition dynamics $\Pr(s'|s, a)$, where $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$. In each case, we use a subset of a relatively standard first-order logic with equality. States and actions are ground; the rules used to express the transition dynamics quantify over variables.

We begin by defining a language that includes a set of predicates Φ and a set of functions Ω. There are three types of functions in Ω: traditional functions, which range over objects; discrete-valued functions, which range over a predefined discrete set of values; and integer-valued functions, which range over a finite subset of the integers.

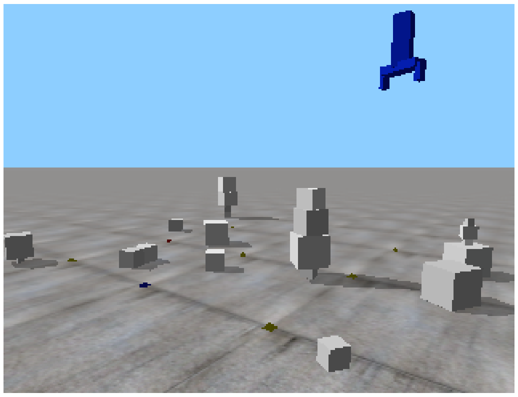

Figure 1: A screen capture of the simulated blocks world. The blocks come in various sizes, visible here, and various colors. The gripper can perform two macro actions: *pickup*, which centers the gripper above a block, lowers it until it hits something, closes it, and raises the gripper; and *puton*, which centers the gripper above a block, lowers until it encounters pressure, opens it, and raises it.

State Representation

In this work, we assume that the environment is completely observable; that is, that the agent is able to perceive an unambiguous and correct description of the current state.[1] Each state consists of a particular configuration of the properties of and relations between objects for all of the objects in the world, where those individual objects are denoted using constants. State descriptions are conjunctive sentences that list the truth values for all of the possible groundings of the predicates and functions with the constants. When writing them down, we will make the closed world assumption and omit the negative literals.

As an example, let us consider representing the state of a simple blocks world, using a language that contains the predicates *on*, *table*, *clear*, *inhand*, and *inhand-nil*. The objects in this world include two blocks, c_1 and c_2, a table t, and a gripper. The sentence

$$\textit{on}(c_1, c_2) \wedge \textit{on}(c_2, t) \wedge \textit{inhand-nil} \wedge \textit{clear}(c_1) \wedge \textit{table}(t) \quad (1)$$

represents a blocks world where the gripper holds nothing and the two blocks are in a single stack on the table.

Action Representation

Actions are represented as positive literals whose predicates are drawn from a special set, α, and whose terms are drawn from the set of constants C associated with the world s where the action is to be executed.

For example, in the simulated blocks world, α contains *pickup*$/1$, an action for picking up blocks, and *puton*$/1$, an action for putting down blocks. The action literal *pickup*(c_1) could represent the action where the gripper attempts to pickup the block c_1 in the state represented in Sentence 1.

[1]This is a very strong, and ultimately indefensible assumption; one of our highest priorities for future work is to extend this to the case when the environment is partially observable.

World Dynamics Representation

We begin by defining probabilistic STRIPS rules (Blum & Langford 1999). Next, we describe the changes we have made to the rules to enable them to model more complex worlds. Then, we explain how the representation language is extended to allow for the construction of additional predicates and functions. Finally, we show how to use a set of rules to provide a model of world dynamics.

Probabilistic STRIPS rules Each probabilistic STRIPS rule specifies the conditions under which it applies, as well as a small number of simple action *outcomes*—sets of changes that might occur in tandem. More formally, a rule for action z has the form

$$\forall \bar{x}.\Psi(\bar{x}) \wedge z(\bar{x}) \rightarrow \bullet \begin{cases} p_1 & \Psi'_1(\bar{x}) \\ \dots & \dots \\ p_n & \Psi'_n(\bar{x}) \end{cases},$$

where $\bar{x}$ is a vector of variables, Ψ is the *context*, a formula that might hold of them at the current time step, $\Psi'_1 \dots \Psi'_n$ are *outcomes*, formulas that might hold in the next step, and $p_1 \dots p_n$ are positive numbers summing to 1, representing a probability distribution over the outcomes. Traditionally, the action $z(\bar{x})$ must contain every $x_i \in \bar{x}$. We constrain Ψ and Ψ' to be conjunctions of literals constructed from the predicates in Φ and the variables $\bar{x}$ as well as equality statements comparing a function (taken from Ω) of these variables to a value in its range. In addition, Ψ is allowed to contain greater-than and less-than statements.

We say that a rule *covers* a state $\Gamma(\mathrm{C})$ and action $a(\mathrm{C})$ if there exists an action substitution σ mapping the variables in $\bar{x}$ to C (note that there may be fewer variables in $\bar{x}$ than constants in C) such that $\Gamma(\mathrm{C}) \models \Psi(\sigma(\bar{x}))$ and $a(\mathrm{C}) = z(\sigma(\bar{x}))$. That is, if there exists a substitution of constants for variables that, when applied to antecedent, grounds it so that it is entailed by the state and, when applied to the rule action, makes it equal the action the rule covers.

Here is an example using the language of Sentence 1:

$$\begin{array}{l} \textit{pickup}(X, Y): \\ \textit{on}(X, Y), \textit{inhand-nil} \\ \quad \rightarrow \begin{cases} .80: & \neg\textit{on}(X, Y), \textit{inhand}(X), \neg\textit{inhand-nil}, \\ & \textit{clear}(Y) \\ .10: & \neg\textit{on}(X, Y), \textit{on}(X, t), \textit{clear}(Y) \\ .10: & \text{no change} \end{cases} \end{array}$$

The context of this rule states that *X* is on *Y*, and there is nothing in the gripper. The rule covers the world of Sentence 1 and action *pickup*(c_1, c_2) under the action substitution $\{X \rightarrow c_1, Y \rightarrow c_2\}$. The first outcome describes the situation where the gripper successfully picks up the block *X*, and the second indicates that *X* falls onto the table.

Let us now consider what a rule that covers the state and action can tell us about the possible subsequent states. Each outcome directly specifies that $\Psi'(\sigma(\bar{x}))$ holds at the next step, but this may be only an incomplete specification of the state. We use the frame assumption to fill in the rest; every literal that would be needed to make a complete description of the state that is not included in $\Psi'(\sigma(\bar{x}))$ is retrieved, with its associated truth value or equality assignment, from $\Gamma(\mathrm{C})$.

Thus, each outcome Ψ'_i can be used to construct a new state s'_i, which will occur with probability p_i. The probability that a rule r assigns to moving from state s to state s' when action a is taken, $P(s'|s,a,r)$, can be calculated as:

$$\begin{aligned} P(s'|s,a,r) &= \sum_{i=1}^{n} P(s', \Psi'_i|s,a,r) \\ &= \sum_{i=1}^{n} P(s'|\Psi'_i, s,a,r) P(\Psi'_i|s,a,r) \end{aligned} \quad (2)$$

where $P(\Psi'_i|s,a,r)$ is p_i, and the outcome distribution $P(s'|\Psi'_i,s,a,r)$ is a deterministic distribution that assigns all of its mass to the relevant s'. If $P(s'|\Psi'_i,s,a,r) = 1.0$, that is, if s' is the state that would be constructed given that rule and outcome, we say that the outcome Ψ'_i *covers* s'.

Noisy Deictic Rules We extend probabilistic STRIPS rules in two ways: by permitting them to refer to objects not mentioned in the action description, and by adding a noise outcome.

Deictic References Relational planning representations use a list of action variables to abstract over the objects in the world. For example, $pickup(X,Y)$ abstracts the identity of the block *X* to be picked up and the block *Y* that *X* will be picked up from. This abstraction allows the rules to compactly encode actions that affect many different objects. Part of the challenge of creating effective rules is to determine what to abstract over. Traditionally, this is done when defining the set of actions, since abstraction can occur only in the action argument list.

We have developed *deictic references*, an extension of a mechanism originally introduced by Benson (1996), as a way of introducing additional variables to the rules. Our rule learning algorithm uses them to learn useful abstractions that were not initially included in the action arguments.

We extend probabilistic STRIPS rules as follows. Each rule is augmented with a list, D, of deictic references. A reference consists of a variable v_i and a restriction ρ_i, which is a set of literals that define v_i with respect to the variables $\bar{x}$ in the action and the other v_j such that $j < i$.

For example, the $pickup(X,Y)$ rule we saw earlier can be rewritten to use deictic references as follows:

$$\begin{array}{l} \mathit{pickup}(X) : \left\{ \; Y : \mathit{on}(X,Y), \;\; Z : \mathit{table}(Z) \; \right\} \\ \mathit{inhand\text{-}nil} \\ \qquad \rightarrow \left\{ \begin{array}{ll} .80: & \neg \mathit{on}(X,Y), \mathit{inhand}(X), \neg \mathit{inhand\text{-}nil}, \\ & \mathit{clear}(Y) \\ .10: & \neg \mathit{on}(X,Y), \mathit{on}(X,Z), \mathit{clear}(Y) \\ .10: & \text{no change} \end{array} \right. \end{array}$$

where *Y* is now defined as a deictic reference that names that unique thing that *X* is *on*. In many ways, this is a more natural encoding because it makes explicit the fact that the only block that *Y* should ever name is the one that *X* is on. This reduces the number of arguments to the action, which can greatly increase planning efficiency (Gardiol & Kaelbling 2003). Note also that, in this representation, different rules for the same action can abstract over different sets of objects.

To use rules with deictic references, we must extend our procedure for computing rule coverage to ensure that all of the deictic references can be resolved. The deictic variables are bound by starting with bindings for $\bar{x}$ and working sequentially through the deictic references D, using their restrictions to determine their unique bindings. If a deictic variable does not have a unique binding—if it has either no possible bindings, or several—it fails to refer, and the rule fails to cover the state and action.

The Noise Outcome Probability models of the type we have seen thus far, ones with a small set of possible outcomes, are not sufficiently flexible to handle noisy domains where there may be a large number of possible action effects that are highly unlikely and yet hard to model—such as all the configurations that may result when a tall stack of blocks topples. It would be inappropriate to model such effects as impossible, and yet we don't have the space or inclination to model each of them as an individual outcome.

We handle this issue by augmenting each rule with an additional *noise outcome*. This outcome has the probability $p_{noise} = 1 - \sum_1^n p_i$, but no associated Ψ'; we are declining to model in detail what happens to the world in such cases.

As an example, consider the rule

$$\begin{array}{l} \mathit{pickup}(X) : \left\{ \; Y : \mathit{on}(X,Y), \;\; Z : \mathit{table}(Z) \; \right\} \\ \mathit{inhand\text{-}nil} \\ \qquad \rightarrow \left\{ \begin{array}{ll} .80: & \neg \mathit{on}(X,Y), \mathit{inhand}(X), \neg \mathit{inhand\text{-}nil}, \\ & \mathit{clear}(Y) \\ .10: & \neg \mathit{on}(X,Y), \mathit{on}(X,Z), \mathit{clear}(Y) \\ .05: & \text{no change} \\ .05: & \text{noise} \end{array} \right. \end{array}$$

where noise can happen with a probability of 0.05. Here, the noise outcome might model the fact that towers sometimes fall over when you are picking up a block.

Since we are not explicitly modeling the effects of noise, we can no longer calculate the transition probability $\Pr(s'|s,a,r)$ using Equation 2: we lack the distribution over next states given the noise outcome, $P(s'|noise,s,a,r)$. Instead, we substitute a worst case constant bound $p_{min} \leq P(s'|noise,s,a,r)$ everywhere this distribution would be required, and bound the transition probability as

$$\begin{aligned} \hat{P}(s'|s,a,r) &= p_{noise} p_{min} + \sum_{i=1}^{n} P(s'|\Psi'_i,s,a,r) p_i \\ &\leq P(s'|s,a,r). \end{aligned}$$

In this way, we create a partial model that allows us to ignore unlikely or overly complex state transitions while still learning and acting effectively. [2]

[2] $P(s'|noise,s,a,r)$ could alternately be any well-defined probability distribution that models the noise of the world. However, we would have to ensure that this distribution does not assign probability to worlds that are impossible (for example, blocks worlds where blocks are floating in midair), because this would complicate planning. We will leave the exploration of this alternative approach to future work.

Background knowledge

In the rule semantics as described so far, the same set of primitive predicates has been used to construct all the elements of the rule. However, it is often useful to divide the predicates and functions of the language into two sets: a set of primitives whose values are observed directly, and represented within a state, and a set of additional predicates and functions that can be derived from these primitives, and so do not need to be represented directly. The derived predicates and functions can then be used in the antecedents, but not in the outcomes—a good thing, since it can be difficult to describe how the values of the derived predicates change directly. (The predicate *above*, the transitive closure of *on*, is an example of a hard-to-update predicate.) This has been found to be essential for representing certain advanced planning domains (Edelkamp & Hoffman 2004).

We define such background knowledge using a *concept language* that includes existential quantification, universal quantification, transitive closure, and counting. Consider the situation where the only primitive predicates are *on* and *table*. Quantification is used for defining predicates such as *inhand*. Transitive closure is included in the language via the Kleene star and plus and defines predicates such as *above*. Finally, counting is included using a special quantifier $\#$ which returns the number of objects for which a formula is true. It is useful for defining integer-valued functions such as *height*. The derived predicates can be used in the context and deictic reference restrictions.

As an example, here is a deictic noisy rule for attempting to pick up block *X* together with the background knowledge used by this rule:

$$\begin{array}{l} \textit{pickup}(X): \left\{ \begin{array}{l} Y: \textit{topstack}(Y,X), \\ Z: \textit{on}(Y,Z), \\ T: \textit{table}(T) \end{array} \right\} \\ \textit{inhand-nil}, \textit{height}(Y) < 9 \\ \quad \rightarrow \left\{ \begin{array}{ll} .80: & \neg\textit{on}(Y,Z) \\ .10: & \neg\textit{on}(Y,Z), \textit{on}(Y,T) \\ .05: & \text{no change} \\ .05: & \text{noise} \end{array} \right. \end{array} \tag{3}$$

$$\begin{array}{rcl} \textit{clear}(V_1) & := & \neg\exists V_2.\textit{on}(V_2,V_1) \\ \textit{inhand}(V_1) & := & \neg\exists V_2.\textit{on}(V_1,V_2) \\ \textit{inhand-nil} & := & \neg\exists V_2 \textit{inhand}(V_2) \\ \textit{above}(V_1,V_2) & := & \textit{on}^*(V_1,V_2) \\ \textit{topstack}(V_1,V_2) & := & \textit{clear}(V_1) \wedge \textit{above}(V_1,V_2) \\ \textit{height}(V_1) & := & \#V_2.\textit{above}(V_1,V_2)) \end{array}$$

The rule is far more complicated than our running example: it deals with the situation when the block to be picked up, *X*, is in the middle of a stack. It is now useful to abstract over even more objects: the deictic variable *Y* identifies the (unique) block on top of the stack, and the deictic variable *Z*—the block under Y. As might be expected, the gripper succeeds in lifting *Y* with a high probability.

$\mathbf{LearnRuleSet}(\mathbf{E})$
Inputs:
  Training examples $\mathbf{E}$
Computation:
  Initialize rule set R to contain only the default rule
  While better rules sets are found
    For each search operator O
      Create new rule sets with O, $R_O = O(R, \mathbf{E})$
      For each rule set $R' \in R_O$
        If the score improves ($S(R') > S(R)$)
          Update the new best rule set, $R = R'$
Output:
  The final rule set R

Figure 2: *LearnRuleSet* Pseudocode. This algorithm performs greedy search through the space of rule sets. At each step a set of search operators each propose a set of new rule sets. The highest scoring rule set is selected and used in the next iteration.

Action Models

Individual rules define the world dynamics only in specific situations; a general description is provided by an *action model*, which consists of some background knowledge and a set of rules R that, together, define the action dynamics of a world. Given an action a and state s, the rule $r \in R$ that covers s and a is used to predict the effects of a in s. When no such rule exists, we use the *default rule*. This rule has an empty context and two outcomes: a no-change outcome (which, in combination with the frame assumption, models the situations where nothing changes), and, again, a noise outcome (modeling all other situations). This rule allows noise to occur in situations where no single non-default rule applies; the probability assigned to the noise outcome in the default rule specifies a kind of "background noise" level. The default rule is also used when more than one rule covers s and a. However, in general, we hope to learn rule sets where the rules are mutually exclusive.

Learning

In this section, we describe an algorithm for learning action models from training examples that describe action effects. More formally, each training example $E \in \mathbf{E}$ is a state, action, next state triple (s, a, s') where states are described in terms of primitive functions and predicates.

We divide the problem of learning action models into two parts: learning background knowledge, and learning a rule set R. First, we describe how to learn a rule set given some background knowledge. Then, we show how to derive new useful concepts.

Learning Rule Sets

The *LearnRuleSet* algorithm takes a set of examples $\mathbf{E}$ and a fixed language of primitive and derived predicates. It then performs a greedy search through the space of possible rule sets as described in the pseudocode in Figure 2.

The search starts with a rule set that contains only the noisy default rule. At every step, we take the current rule set and apply all our search operators to it to obtain a set of new rule sets. We then select the rule set R that maximizes the scoring metric

$$S(R) = \sum_{(s,a,s')\in \mathbf{E}} \log(\hat{P}(s'|s,a,r_{(s,a)})) - \alpha \sum_{r\in \mathbf{R}} PEN(r)$$

where $r_{(s,a)}$ is the rule that covers (s,a), α is a scaling parameter, and the penalty $PEN(r)$ is the number of literals in the rule r. Ties in $S(R)$ are broken randomly.

As a greedy search through the space of rule sets, *LearnRuleSet* is similar in spirit to previous work (Pasula, Zettlemoyer, & Kaelbling 2004). However, adapting that work to handle our representation extensions involved substantial redesign of the algorithm, including changing the initial rule set, the scoring metric, and the search operators.

Search Operators Each search operator O takes as input a rule set R and a set of training examples $\mathbf{E}$, and creates a set of new rule sets R_O to be evaluated by the greedy search loop. There are eight search operators. We first describe the most complex operator, *ExplainExamples*, and then the most simple one, *DropRules*. Finally, we present the remaining six operators which all share a common computational framework, outlined in Figure 4.

- *ExplainExamples* takes as input a training set $\mathbf{E}$ and a rule set R and creates new rule sets that contain additional rules modeling the training examples that were covered by the default rule in R. Figure 3 shows the pseudocode for this algorithm, which considers each training example E that was covered by the default rule in R, and executes a three-step procedure. The first step builds a large and specific rule r' that describes this example; the second step attempts to trim this rule, and so generalize it so as to maximize its score, while still ensuring that it covers E; and the third step creates a new rule set R' by copying R and integrating the new rule r' into this new rule set.

  As an illustration, let us consider how steps 1 and 2 of *ExplainExamples* might be applied to the training example $(s,a,s') = (\{on(a,t), on(b,a)\}, pickup(b), \{on(a,t)\})$, when the background knowledge is as defined for Rule 4. Step 1 builds a rule r. It creates a new variable X to represent the object b in the action; then, the action substitution becomes $\sigma = \{X \to b\}$, and the action of r is set to $pickup(X)$. The context of r is set to the conjunction $\textit{inhand-nil}, \neg inhand(X), clear(X), height(X) = 2, \neg on(X,X), \neg above(X,X), \neg topstack(X,X)$ Then, in Step 1.2, *ExplainExamples* attempts to create deictic references that name the constants whose properties changed in the example. In this case, the only changed literal is $on(b,a)$, so $C = \{a\}$; a new deictic variable Y is created and restricted, and σ is extended to be $\{X \to b, Y \to a\}$. The resulting rule r' looks as follows:

$$pickup(X) : \left\{ Y : \begin{array}{l} \neg inhand(Y), \neg clear(Y), on(X,Y), \\ above(X,Y), topstack(X,Y), \\ \neg above(Y,Y), \neg topstack(Y,Y), \\ \neg on(Y,Y), height(Y) = 1 \end{array} \right\}$$
$$\textit{inhand-nil}, \neg inhand(X), clear(X), height(X) = 2, \neg on(X,X), \neg above(X,X), \neg topstack(X,X)$$
$$\to \{\ 1.0 : \neg on(X,Y)$$

ExplainExamples$(R, \mathbf{E})$
Inputs:
A rule set R
A training set $\mathbf{E}$
Computation:
For each example $(s,a,s') \in \mathbf{E}$ covered by the default rule in R
Step 1: *Create a new rule r*
Step 1.1: *Create an action and context for r*
Create new variables to represent the arguments of a
Use them to create a new action substitution σ
Set r's action to be $\sigma^{-1}(a)$
Set r's context to be the conjunction of boolean and equality literals that can be formed using the variables and the available functions and predicates (primitive and derived) and that are entailed by s
Step 1.2: *Create deictic references for r*
Collect the set of constants C whose properties changed from s to s', but which are not in a
For each $c \in C$
Create a new variable v and extend σ to map v to c
Create ρ, the conjunction of literals containing v that can be formed using the available variables, functions, and predicates, and that are entailed by s
Create deictic reference d with variable v and restriction $\sigma^{-1}(\rho)$
If d uniquely refers to c in s, add it to r
Step 2: *Trim literals from r*
Create a rule set R' containing r and the default rule
Greedily trim literals from r while r still covers (s,a,s') and R''s score improves
Step 3: *Create a new rule set containing r*
Create a new rule set $R' = R$
Add r to R' and remove any rules in R' that cover any examples r covers
Recompute the set of examples that the default rule in R' covers and the parameters of this default rule
Add R' to the return rule sets R_O
Output:
A set of rule sets, R_O

Figure 3: *ExplainExamples* Pseudocode. This algorithm attempts to augment the rule set with new rules covering examples currently handled by the default rule.

  In Step 2, *ExplainExamples* trims this rule to remove the invariably true literals, like $\neg on(X,X)$, and the redundant ones, like $\neg inhand()$ and $\neg clear(Y)$, to give

$$pickup(X) : \{Y : \ on(X,Y), height(Y) = 0\ \}$$
$$\textit{inhand-nil}, clear(X), height(X) = 1$$
$$\to \{\ 1.0 : \neg on(X,Y)$$

  which is then integrated into the rule set.

- *DropRules* cycles through all the rules in the current rule set, and removes each one in turn from the set. It returns a set of rule sets, each one missing a different rule.

The remaining six operators create new rule sets from the input rule set R by repeatedly choosing a rule $r \in R$ and making changes to it to create one or more new rules. These new rules are then integrated into R, just as in *ExplainExamples*, to create a new rule set R'. Figure 4 shows the

$\mathbf{OperatorTemplate}(R, \mathbf{E})$
Inputs:
Rule set R
Training examples $\mathbf{E}$
Computation:
Repeatedly select a rule $r \in R$
Create a copy of the input rule set $R' = R$
Create a new set of rules, N, by making changes to r
For each new rule $r' \in N$
Estimate new outcomes for r' with the *InduceOutcomes* algorithm described by Pasula et al (2004)
Add r' to R' and remove and rules in R' that cover any examples r' covers
Recompute the set of examples that the default rule in R' covers and the parameters of this default rule
Add R' to the return rule sets R_O
Output:
The set of rules sets, R_O

Figure 4: *OperatorTemplate* Pseudocode. This algorithm is the basic framework that is used by six different search operators. Each operator repeatedly selects a rule, uses it to make n new rules, and integrates those rules into the original rule set to create a new rule set.

the general pseudocode for how this is done. The operators vary in the way they select rules and the changes they make to them. These variations are described for each operator below:

- *DropLits* selects every rule $r \in R$ n times, where n is the number of literals in the context of r; in other words, it selects each r once for each literal in its context. It then creates a new rule r' by removing that literal from r's context; N of Figure 4 is simply the set containing r'.
- *DropRefs* selects each rule $r \in R$ once for each deictic reference in r. It then creates a new rule r' by removing that deictic reference from r.
- *ChangeRanges* selects each rule $r \in R$ n times for each equality or inequality literal in the context, where n is the total number of values in the range of each literal. Each time it selects r it creates a new rule r' by replacing the numeric value of the chosen (in)equality with another other possible value from the range. Thus, if $f()$ ranges over $[1 \ldots n]$, *ChangeRange* would, when applied to a rule containing the inequality $f() < i$, construct rule sets in which i is replaced by all other integers in $[1 \ldots n]$.
- *SplitOnLits* selects each rule $r \in R$ n times, where n is the number of literals that are absent from the rule's context. (The set of absent literals is obtained by applying the available predicates and functions—both primitive and derived—to the variables defined in the rule, and removing those already present.) It then constructs a set of new rules. In the case of predicate and inequality literals, it creates one rule in which the positive version of the literal is inserted into the context, and one in which it is the negative version. In the case of equality literals, it constructs a rule for every possible value the equality could take. This time, N contains all these rules.
- *AddLits* selects each rule $r \in R$ n times, where n is the number of predicate-based literals that are absent from the rule's antecedent. It constructs a new rule by inserting that literal into the earliest place in which the its variables are all well-defined. If the literal contains no deictic variables, this will be the context, otherwise this will be the restriction of the last deictic variable mentioned in the literal. (If V_1 and V_2 are deictic variables and V_1 appears first, $p(V_1, V_2)$ would be inserted into the restriction of V_2.)
- *AddRefs* selects each rule $r \in R$ n times, where n is the number of literals that can be constructed from variables in r and a new variable v. It then creates a new rule by adding a deictic reference with the variable v and a restriction defined by one of the literals.

We have found that all of these types of operators are consistently used during learning. While this set of operators is heuristic, it is complete in the sense that every rule set can be constructed from the initial rule set—although, of course, there is no guarantee that the scoring metric will lead the greedy search to the global maximum.

Learning Background Knowledge

We learn background knowledge using an algorithm which iteratively constructs increasingly complex concepts, then tests their usefulness by running *LearnRuleSet* and checking whether they appear in the learned rules. The first set is created by applying the operators in Figure 5 to literals built with the original language. Subsequent sets of concepts are constructed using the literals that proved useful on the latest run; concepts that have been tried before, or that are always true or always false across all examples, are discarded. The search ends when none of the new concepts prove useful.

Since our concept language is quite rich, overfitting (e.g., by learning concepts that can be used to identify individual examples) can be a serious problem. We handle this in the expected way: by introducing a penalty term, $\alpha' c(R)$, to create a new scoring metric

$$S'(R) = S(R) - \alpha' c(R)$$

where $c(R)$ is the number of distinct concepts used in the rule set R and α' is a scaling parameter. This new metric S' is now used by *LearnRuleSet*; it avoids overfitting by favoring rule sets that use fewer derived predicates.

Evaluation

In this section, we demonstrate that noise outcomes and derived predicates are necessary to learn good action models for the physics-based blocks world simulator of Figure 1, and also that our algorithm is capable of discovering the relevant background knowledge. We accomplish this by learning a variety of action models and then comparing their performance on a simple planning task.

All the experiments are set in a world containing twenty blocks. The observed, primitive predicates include $on(X, Y)$ (which is true if block X exerts a downward force on Y), $size(X)$, $color(X)$, and the typing predicate *table*(X). There were five sizes and five colors, both uniformly distributed. The color attribute is a distractor. The sizes complicate the

$$
\begin{array}{rcl}
p(X) & \rightarrow & n := QY.p(Y) \\
p(X_1, X_2) & \rightarrow & n(Y_2) := QY_1.p(Y_1, Y_2) \\
p(X_1, X_2) & \rightarrow & n(Y_1) := QY_2.p(Y_1, Y_2) \\
p(X_1, X_2) & \rightarrow & n(Y_1, Y_2) := p^*(Y_1, Y_2) \\
p(X_1, X_2) & \rightarrow & n(Y_1, Y_2) := p^+(Y_1, Y_2) \\
p_1(X_1), p_2(X_2) & \rightarrow & n(Y_1) := p_1(Y_1) \wedge p_2(Y_1) \\
p_1(X_1), p_2(X_2, X_3) & \rightarrow & n(Y_1, Y_2) := p_1(Y_1) \wedge p_2(Y_1, Y_2) \\
p_1(X_1), p_2(X_2, X_3) & \rightarrow & n(Y_1, Y_2) := p_1(Y_1) \wedge p_2(Y_2, Y_1) \\
p_1(X_1, X_2), p_2(X_3, X_4) & \rightarrow & n(Y_1, Y_2) := p_1(Y_1, Y_2) \wedge p_2(Y_1, Y_2) \\
p_1(X_1, X_2), p_2(X_3, X_4) & \rightarrow & n(Y_1, Y_2) := p_1(Y_1, Y_2) \wedge p_2(Y_2, Y_1) \\
p_1(X_1, X_2), p_2(X_3, X_4) & \rightarrow & n(Y_1, Y_2) := p_1(Y_1, Y_2) \wedge p_2(Y_1, Y_1) \\
p_1(X_1, X_2), p_2(X_3, X_4) & \rightarrow & n(Y_1, Y_2) := p_1(Y_1, Y_2) \wedge p_2(Y_2, Y_2) \\
f(X) = c & \rightarrow & n() := \#Y.f(Y) = c \\
f(X) \leq c & \rightarrow & n() := \#Y.f(Y) \leq c \\
f(X) \geq c & \rightarrow & n() := \#Y.f(Y) \geq c
\end{array}
$$

Figure 5: Operators used to invent a new predicate n. Each operator takes as input one or more literals, listed on the left. The ps represent old predicates; f represents an old function; Q can refer to $\forall$ or $\exists$; and c is a numerical constant. Each operator takes a literal and returns a concept definition. These operators are applied to all of the literals used in rules in a rule set to create new predicates.

action dynamics, both because they influence stack stability, and because the gripper does best with blocks of average size, and is unable to grasp giant blocks at all. The training data were generated by repeatedly attempting to perform random actions in random simulator states and noting the result. The random starting states were generated by randomly placing blocks on each other, or on the table. The last block was sometimes placed in the gripper.

Planning

Since we have no true model to compare the rule sets to, we evaluate them by using them to plan. We implemented a simple planner based on the sparse sampling algorithm (Kearns, Mansour, & Ng 2002), which treats the domain as a Markov Decision Problem (MDP) (Puterman 1999). Given a state s, it creates a tree of states (of predefined depth and branching factor) by sampling forward using a transition model, computes the value of each node using the Bellman equation, and selects the action that has the highest value. In our implementation, the transition function is defined using an action model and the reward function is defined by hand.

We adapt the algorithm to handle noisy outcomes, which do not predict the next state, by estimating the value of the unknown next state as a fraction of the value of staying in the same state: i.e., we sample forward as if we had stayed in the same state and then scale down the value we obtain. Our scaling factor was 0.75, our depth was three, and our branching factor was five.

This scaling method is only a guess at what the value of the unknown next state might be; because noisy rules are

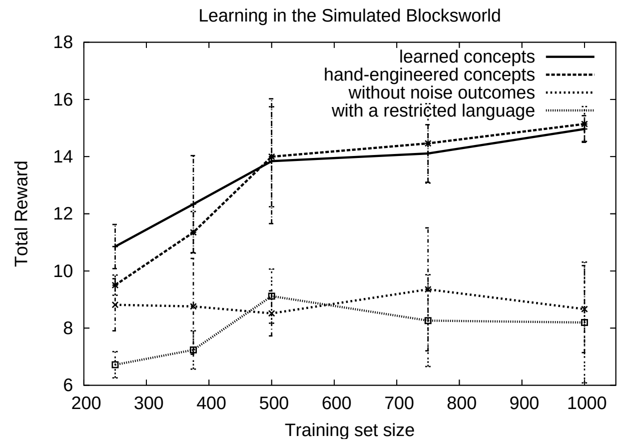

Figure 6: The performance of various action model variants as a function of the number of training examples. All data points were averaged over five runs each of three rule sets learned on different training data sets. For comparison, the average reward for performing no actions is 9.2, and the reward obtained when a human directed the gripper averaged 16.2.

partial models, there is no way to compute the value explicitly. In the future, we would like to explore methods that learn to associate values with noise outcomes. For example, the value of the outcome where a tower of blocks falls over is different if the goal is to build a tall stack of blocks than if the goal is to put all of the blocks on the table.

Experiments

We set our planner the task of building tall stacks: our reward function was the average height of the blocks in the world. The plans were executed for ten time steps. The scaling parameters α and α' (associated respectively with the rule complexity penalty term, and the background knowledge complexity penalty term) were set to 1.0 and 5.0. The noise probability bound p_{min} was set to 0.00001.

To evaluate the overall quality of the learned rules, we did an informal experiment to measure the reward achieved when a human domain expert directed the robot arm. (Note that humans have an advantage over the planner, since they can view the entire 3D world while the planner only has access to the information encoded in the *on*, *height*, and *size* relations.)

Results We tested four action model variants, varying the training set size; the results are shown in Figure 6. The curve labeled 'learned concepts' represents the full algorithm as presented in this paper. Its performance approaches that obtained by a human expert, and is comparable to that of the algorithm labeled 'hand-engineered concepts' that did not do concept learning, but was, instead, provided with hand-coded versions of the concepts *clear*, *inhand*, *inhand-nil*, *above*, *topstack*, and *height*. The concept learner discovered all of these, as well as other useful predicates, e.g., $p(X, Y) := \mathit{clear}(Y) \wedge on(Y, X)$, which we will call *onclear*.

This could be why its action models outperformed the hand-engineered ones slightly on small training sets. In domains less well-studied than the blocks world, it might be less obvious what the useful concepts are; the concept-discovery technique presented here should prove helpful.

The remaining two model variants obtained rewards comparable to the reward for doing nothing at all. (The planner did attempt to act during these experiments, it just did a poor job.) In one variant, we used the same full set of predefined concepts but the rules could not have noise outcomes. The requirement that they explain every action effect led to significant overfitting and a decrease in performance. The other rule set was given the traditional blocks world language, which does not include *above*, *topstack*, or *height*, and allowed to learn rules with noise outcomes. We also tried a full-language variant where noise outcomes were allowed, but deictic references were not: the resulting rule sets contained only a few very noisy rules, and the planner did not attempt to act at all. The poor performance of these ablated versions of our representation shows that all three of our extensions are essential for modeling the simulated blocks world domain.

Example Learned Rules To get a better feel for the types of rules learned, here are two interesting rules learned by the full algorithm.

$$
\begin{array}{l}
\mathit{pickup}(X) : \left\{ \begin{array}{l} Y : \mathit{onclear}(X,Y),\ Z : \mathit{on}(Y,Z), \\ T : \mathit{table}(T) \end{array} \right\} \\
\mathit{inhand\text{-}nil}, \mathit{size}(X) < 2 \\
\quad \rightarrow \left\{ \begin{array}{ll} .80 : & \neg \mathit{on}(Y,Z) \\ .10 : & \neg \mathit{on}(X,Y) \\ .10 : & \neg \mathit{on}(X,Y), \mathit{on}(Y,T), \neg \mathit{on}(Y,Z) \end{array} \right.
\end{array}
$$

This rule applies when the empty gripper is asked to pick up a small block X that sits on top of another block Y. The gripper grabs both with a high probability.

$$
\begin{array}{l}
\mathit{puton}(X) : \left\{ \begin{array}{l} Y : \mathit{topstack}(Y,X),\ Z : \mathit{inhand}(Z), \\ T : \mathit{table}(T) \end{array} \right\} \\
\mathit{size}(Y) < 2 \\
\quad \rightarrow \left\{ \begin{array}{ll} .62 : & \mathit{on}(Z,Y) \\ .12 : & \mathit{on}(Z,T) \\ .04 : & \mathit{on}(Z,T), \mathit{on}(Y,T), \neg \mathit{on}(Y,X) \\ .22 : & \text{noise} \end{array} \right.
\end{array}
$$

This rule applies when the gripper is asked to put its contents, Z, on a block X which is inside a stack topped by a small block Y. Because placing things on a small block is chancy, there is a reasonable probability that Z will fall to the table, and a small probability that Y will follow.

Discussion and Future Work

In this paper, we developed a probabilistic action model representation that is rich enough to be used to learn models for planning in the simulated blocks world. This is a first step towards defining representations and algorithms that will enable learning in more complex worlds.

There remains much work to be done in the context of learning probabilistic planning rules. We plan to expand our approach to handle partial observability, possibly incorporating some of the techniques from work on deterministic learning (Amir 2005). We also plan to learn probabilistic operators in an incremental, online manner, similar to the learning setup in the deterministic case (Shen & Simon 1989; Gil 1994; Wang 1995), which has the potential to help scale this approach to larger domains. Finally, we plan to explore the learning of parallel planning rules.

Acknowledgments

This material is based upon work supported in part by the Defense Advanced Research Projects Agency (DARPA), through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010; and in part by DARPA Grant No. HR0011-04-1-0012 .

References

Amir, E. 2005. Learning partially observable deterministic action models. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*.

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

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

Edelkamp, S., and Hoffman, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. *Technical Report 195, Albert-Ludwigs-Universität, Freiburg, Germany*.

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

Gardiol, N., and Kaelbling, L. 2003. Envelope-based planning in relational MDPs. In *Advances in Neural Information Processing Systems 16*.

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

Kearns, M.; Mansour, Y.; and Ng, A. 2002. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning* 49(2).

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

ODE. 2004. Open dynamics engine toolkit. http://opende.sourceforge.net.

Pasula, H.; Zettlemoyer, L.; and Kaelbling, L. 2004. Learning probabilistic relational planning rules. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*.

Puterman, M. L. 1999. *Markov Decision Processes*. John Wiley and Sons, New York.

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

Wang, X. 1995. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the Twelfth International Conference on Machine Learning*.