Learning Probabilistic Planning Rules

Hanna M. Pasula
MIT AI Lab
Cambridge, MA 02139
pasula@ai.mit.edu

Luke S. Zettlemoyer
MIT AI Lab
Cambridge, MA 02139
lsz@ai.mit.edu

Leslie Pack Kaelbling
MIT AI Lab
Cambridge, MA 02139
lpk@ai.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 a compact representations for STRIPS-like probabilistic planning operators, and develop an algorithm for the novel problem of learning such operators from examples. We also present experiments that demonstrate effective learning of several planning operators.

1 Introduction

Our goal is to make robots that can operate in the same world that we do. In order to do so, a robot must be able to predict the consequences of its actions both efficiently and accurately. The general problem of accurate prediction is very difficult, so we must take advantage of the structure and regularities of the world.

Two of the most important structural principles are that most things stay the same, and that the effects of actions depend on the properties of objects and not their identities. These principles have been captured in planning representations such as STRIPS rules since the earliest days (Fikes & Nilsson, 1971), in the use of the frame assumption that the only changes are those made by the agent, and in the use of planning operator schemata that abstract over particular objects. Probabilistic STRIPS rules (Blum & Langford, 1999) capture another principle: that when action effects are uncertain, it is rarely the case that every effect is uncertain independently of all the others. There are usually a few possible outcomes; actions can either succeed, and have the appropriate effects, or fail, and have no effect—or perhaps some alternative set of less desirable effects. This principle of modeling small sets of action outcomes underlies many probabilistic planning representations, such as the probabilistic situation calculus (Boutilier, Reiter, & Price, 2001), and the equivalence-class approach of Draper, Hanks, and Weld (1994).

We are developing learning algorithms that can leverage these structural properties. In particular, since we want robots to learn in our world, we are investigating rules which can be learned incrementally in pieces as the robot explores different parts of the world. A robot might first learn, and take advantage of the fact that, objects typically move when touched, unless they're very large. With more experience of the world, it could refine that rule, as well as learn rules describing the way objects behave when they're dropped, or what happens when it tries to pick up objects that have something heavy on top of them. In this paper we develop a rule-based representation that can describe different aspects of the world dynamics independently; however, our current algorithms are all non-incremental. We do, however place special emphasis on the performance of the algorithms with small

$$\begin{array}{rcl} \textit{on}(X,Y), \textit{clear}(X), \textit{inhand}(\text{NIL}), \\ \textit{block}(X), \textit{block}(Y), \neg\textit{wet}, \textit{pickup}(X,Y) & \rightarrow & \left\{ \begin{array}{ll} .8: & \textit{inhand}(X), \neg\textit{clear}(X), \neg\textit{inhand}(\text{NIL}), \\ & \neg\textit{on}(X,Y), \textit{clear}(Y) \\ .2: & \textit{on}(X, \text{TABLE}), \neg\textit{on}(X,Y), \neg\textit{inhand}(\text{NIL}) \\ .1: & \text{no change} \end{array} \right. \\ \\ \textit{on}(X,Y), \textit{clear}(X), \textit{inhand}(\text{NIL}), \\ \textit{block}(X), \textit{block}(Y), \textit{wet}, \textit{pickup}(X,Y) & \rightarrow & \left\{ \begin{array}{ll} .33: & \textit{inhand}(X), \neg\textit{clear}(X), \neg\textit{inhand}(\text{NIL}), \\ & \neg\textit{on}(X,Y), \textit{clear}(Y) \\ .33: & \textit{on}(X, \text{TABLE}), \neg\textit{on}(X,Y), \neg\textit{inhand}(\text{NIL}) \\ .34: & \text{no change} \end{array} \right. \end{array}$$

Figure 1: Two example rules for the blocks-world domain, describing the dynamics of the *pickup* action when it attempts to pick up a block which is stacked on another. Notice that, when the gripper is wet, the probability of failing to grasp the block or of dropping it onto the table increases. The *pickup* ruleset combines these rules together with a similar pair, which handle attempts to pick up a block off the table.

amounts of data, which is guaranteed to be the situation of agents learning in very complex domains.

There has been essentially no work on the problem of inducing probabilistic relational representations of world dynamics. Oates and Cohen (1996) learn propositional probabilistic planning operators, but each rule has a single outcome and multiple rules can apply in parallel, making the model more like a dynamic Bayesian network with no arcs in the second slice, and therefore unable to capture correlations in the effects. Shen and Simon (1989) and others in the ILP community have learned deterministic planning operators from data. In this paper, we attempt to tackle this under-studied problem.

We begin by presenting a formal description of our probabilistic relational rules representation. Then, we discuss the difficulty of learning such rules, and describe a greedy search algorithm for the problem. Finally, we present a set of experiments that demonstrate effective learning of several planning operators.

2 Representation

We model worlds that have stationary stochastic dynamics and are completely observable. The world state at time t, $\mathbf{S}^t$, is represented using a conjunction of ground literals. Given an action $A^t = a$ we define the transition to the next state, $\mathbf{S}^{t+1}$, by specifying the conditional distribution $P_a(\mathbf{S}^{t+1}|\mathbf{S}^t)$. Our aim is to model this distribution compactly. We represent it using a set of probabilistic STRIPS rules $\mathbf{r}_a$: every rule $r \in \mathbf{r}_a$ applies in some subset of the possible world states, and defines a distribution over a set of *outcomes*, where each outcome is a set of action effects that occur in tandem. Like all STRIPS-like representations, we make the *frame assumption*, which states that everything not explicitly changed by the rules stays the same. Thus, if no rule in our ruleset $\mathbf{r}_a$ applies in $\mathbf{S}^t$, nothing changes and $\mathbf{S}^{t+1} = \mathbf{S}^t$; and when a rule does apply, the only changes permitted in $\mathbf{S}^{t+1}$ are the effects of the chosen outcome.

Our probabilistic STRIPS rules use a subset of first-order logic which has predicates, constants, and conjunctive connectives. (It does not include functions, disjunctive connectives or quantification.) Each $r \in \mathbf{r}_a$ is a four-tuple $(C, \mathbf{O}, P_\mathbf{O}, A)$. The rule's action $r.A$ is a positive literal, an instance of the predicate a. The *context* $r.C$ is a conjunction. Together, the action and the context select the set of world states in which r applies. The *outcome set* $r.\mathbf{O}$ is a set of conjunctions. Each outcome $O \in r.\mathbf{O}$ contains literals that must be in $\mathbf{S}^{t+1}$ if that particular outcome has occurred. $r.P_\mathbf{O}$ is a discrete distribution over the outcomes. Rules may contain variables; however, we impose the STRIPS constraint that every variable appearing in $r.C$ or $r.\mathbf{O}$ also appears in $r.A$.

As an example, consider the two rules in Figure 1, which describe part of the action dynamics of a probabilistic blocks-world domain. In this domain, inspired by the work of Draper et al. (1994), the agent has both a 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.

When deciding whether a rule applies, or whether an outcome might have occurred, we often need to check whether a conjunction is true in a particular world. We define the *selector function* $\delta(\mathbf{C_1}, \mathbf{C_2})$ on two ground conjunctions to be 1 if the literals in $\mathbf{C_1}$ are a subset of the literals in $\mathbf{C_2}$, $\mathbf{C_1} \subseteq \mathbf{C_2}$, and 0 otherwise. Since rules can contain variables, we may have to ground them using a substitution. We denote the application of a substitution θ as $\theta(\mathbf{C})$. The *argument substitution* θ_a as the substitution that maps each of the variables in $r.A$ to the respective argument of action A^t. This is the only substitution we need here, since the STRIPS constraint we have made guarantees that, given an action, all the variables will be uniquely bound, permitting us to apply the selector function δ to them directly.

Following the above definitions, a rule r applies to a state $\mathbf{S}^t$ and an action A^t if $\theta_a(r.A) = A^t$ and $\delta(\theta_a(r.C), S^t) = 1$: that is, if, after the action substitution has been applied to the rule r, its action matches A^t and its context selects $\mathbf{S}^t$. We assume that there exists at most one rule $r \in \mathbf{r}_a$ which applies to any pair $(\mathbf{S}^t, A^t)$. The distribution over $\mathbf{S}^{t+1}$ is then defined as

$$P_a(\mathbf{S}^{t+1}|\mathbf{S}^t) = \sum_{o \in r.\mathbf{O}} \delta(\theta_a(o), \mathbf{S}^{t+1}) \delta(\mathbf{S}^t - \theta_a(o), \mathbf{S}^{t+1}) r.P_{\mathbf{O}}(o) \,. \tag{1}$$

The two δ functions select the outcomes that describe the transition from $\mathbf{S}^t$ to $\mathbf{S}^{t+1}$: the first one checks whether the outcome is true in the new state, and the second one enforces the frame assumption for all literals not in the outcome. The probability of each $\mathbf{S}^{t+1}$ is the sum of all the selected outcomes. Notice that each outcome will be selected exactly once, since, for a given $\mathbf{S}^t$, each outcome describes a unique $\mathbf{S}^{t+1}$. As a result, P_a is always well defined. Notice also that each $\mathbf{S}^{t+1}$ can be described by more than one outcome. This *overlapping outcomes* phenomenon is seen in rules like

$$\textit{block}(X), \textit{paint}(X) \quad \rightarrow \quad \begin{cases} .25 : \textit{painted}(X), \textit{wet} \\ .25 : \textit{painted}(X) \\ .50 : \text{no change} \end{cases}$$

where, if the block is already painted, and the gripper is already wet, all outcomes apply, and $\mathbf{S}^{t+1} = \mathbf{S}^t$ with probability 1.

3 Learning

In this section, we present our algorithm for learning probabilistic planning rules, given a training set $\mathbf{D} = D_1 \ldots D_{|\mathbf{D}|}$. Every element D of this set represents a single transition in the world, and consists of a previous state $D.\mathbf{S}^t$, an action $D.A^t$, and a next state $D.\mathbf{S}^{t+1}$. It should be clear that a rule applies to an example D if it applies to its state $\mathbf{S}^t$ and action A^t. Our aim is to, for every possible action a, learn a ruleset $\mathbf{r}_a$ defining the distribution $P_a(\mathbf{S}^{t+1}|\mathbf{S}^t)$.

The sections below are organized as follows. In Section 3.1, we describe the outer learning loop which is a search through the space of rule sets. This outer loop assumes the existence of a subroutine *induceOutcomes* which, when given a context and an action, learns the rest of the rule. *induceOutcomes* is discussed in Section 3.2 and makes use of the subroutine *learnParameters*, which learns a distribution over a given set of outcomes as presented in Section 3.3.

3.1 Rule Sets

Our rule-learning algorithm performs a greedy search in the space of proper rulesets, where a proper ruleset for dataset $\mathbf{D}_a$ is a set of rules $\mathbf{r}_a$ in which there is exactly one rule applicable to every example D_i in which some change occurs. (The examples where no change occurs can be handled by the frame assumption, so they do not need to have an explicit rule.) Our scoring function scores each rule r by combining the log likelihood of the set of examples to which the rule applies, $\mathbf{D}_r$, with the penalty term

$$S(r) = \alpha|r.C| + 0.5(|r.\mathbf{O}| - 1)\log(|\mathbf{D}_r|). \quad (2)$$

where α is a scaling parameter (we have been setting $\alpha = 0.5$). The first part of the penalty term penalizes long contexts; the second part keeps track of the description length of the parameters.

Given a dataset $\mathbf{D}_a$, we initialize the search by constructing a maximally specific ruleset. Thus, we create a rule to represent every combination of $\mathbf{S}^t$ and A^t values encountered in the training data. Whenever a rule is created, we pass its action and context to *induceOutcomes* to fill in the rest: in this case, we set the context to $\mathbf{S}^t$, the action to A^t before calling the function. At every step of the search, we greedily find and apply the best operator. There are four types of search operators. Two of them generalize the rule: one drops a predicate from the rule context, and one replaces one of the constant arguments of the action A with a variable (this substitution is then applied throughout the rule.) Generalization may increase the number of examples covered by a rule, and so make some of the other rules unnecessary. We give the new rule precedence over any of these other rules, removing them from the set: however, this removal can leave some training examples with no rule, so new, specific rules are induced to cover them, just as at initialization. The other two operators replace a general rule with several specific ones, either by retrieving a dropped literal and constructing two new contexts (corresponding to the literal's two truth values), or by selecting a variable and creating a set of new rules corresponding to the different constant values the variable takes on in the examples.

This simple algorithm has one large drawback; the set of rules which are learned is only guaranteed to be valid on the training set and not on testing data. We leave it to future work to solve this problem, possibly with approaches based on least general generalization (Plotkin, 1970) or decision trees (Blockeel & Raedt, 1998).

3.2 Inducing Outcomes

Inducing outcomes is the problem of finding a set of outcomes $\mathbf{O}$ and parameters $P_{\mathbf{O}}$ which maximize the score of a rule with a given context C or, alternatively, a rule covering the set of examples $\mathbf{D}$ selected by C. We assume that the subroutine *estimateParams*, presented in Section 3.3, will provide the score maximizing $P_{\mathbf{O}}$ for a given $\mathbf{D}$ and $\mathbf{O}$. Here we first argue that, in the worst case, the problem of inducing outcomes is NP-hard. Then, we provide a greedy search algorithm for finding $\mathbf{O}$.

Our problem is to, given an α, find an $r.\mathbf{O}$ that minimizes $-\log L(\mathbf{D}|r) + \alpha S(r)$. Here, we show that the special case of finding an $r.\mathbf{O}$ than minimizes $|r.\mathbf{O}|$ while assigning the data nonzero likelihood is NP-hard. We do this by reducing the minimal subset problem (Garey & Johnson, 1979) to our problem. Given a finite set $\mathbf{S}$, and a collection $\mathbf{C}$ of subsets of $\mathbf{S}$, the minimal subset problem is to find a minimal subset $\mathbf{C}'$ of $\mathbf{C}$ such that every element in $\mathbf{S}$ belongs to at least one member of $\mathbf{C}'$. Now, consider a set of outcome induction training examples $\mathbf{D} = \{D_1 \ldots D_{|\mathbf{D}|}\}$. Each possible outcome O covers some (possibly empty) subset of the examples in $\mathbf{D}$. Let us define the set of outcomes $\mathbf{O}$ as the set of all the most general outcomes corresponding to non-empty subsets of $\mathbf{D}$. Our reduction creates, for each $C \in \mathbf{C}$, a predicate p_C. We associate a full set of p_Cs with every $S \in \mathbf{S}$: each such p_C is true iff $S \in C$. The Ss can then be treated as examples, and the Cs as outcomes, which cover the examples in the same way that the original subsets covered their

elements. Finding the minimal set of outcomes will also find the minimal set of subsets. Since the conversion can be done in polynomial time ($O(|\mathbf{S}||\mathbf{C}|)$), finding the smallest set of outcomes is NP-hard.

Because our problem is hard, we again use a greedy search algorithm. We will present this algorithm using a running example set in the n-coin domain, in which each world contains n coins which can be showing either heads or tails. Consider *flip-coupled*, a simple action with no context and no parameters, which 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 { h(c1)h(c2) → h(c1)h(c2) , t(c1)h(c2) → h(c1)h(c2), h(c1)t(c2) → h(c1)h(c2), h(c1)h(c2) → t(c1)t(c2) }, where h(C) stands for heads(C), t(C) stands for ¬heads(C), and $D.\mathbf{S}^t \rightarrow D.\mathbf{S}^{t+1}$ is an example with $D.A^t =$ *flip-coupled*.

Our algorithm searches through a restricted subset of possible outcome sets: those that are *valid* on the training examples, where an outcome set is valid if every training example has at least one outcome that covers it and every outcome covers has at least one training example. (Note that an outcome o covers a training example D if $\delta(\theta_a(o), D.\mathbf{S}^{t+1})\delta(\mathbf{S}^t - \theta_a(o), D.\mathbf{S}^{t+1}) = 1$ as we did in Equation 1.) We create an initial set of valid outcomes by, for each example, writing down the set of literals which changed values as a result of the action, and then creating an outcome to describe every set of changes observed in this way. In our running example, the initial outcome set is { h(c1), h(c2), t(c1)t(c2), nil }. We then search through the space of valid outcome sets using two operators.[1] The first is an add operator which picks a pair of outcomes in the set and adds in a new outcome based on their conjunction. The second is a remove operator that drops an outcome from the set. We try all possible moves at each iteration and take the one that improves the score the most, and stop searching when the score can't be improved. In our running example, we would add the outcome h(c1)h(c2) (from conjoining h(c1) and h(c2)) and then drop everything else except for t(c1)t(c2). This example highlights the intuition behind our search strategy. In practice, the outcomes we are looking for are either in the initial outcome set, like t(c1)t(c2), or are merges of initial outcomes, like h(c1) and h(c2).

It is worth recalling that this example has no context and no action. Handling contexts and actions with constant parameters 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 parameters, we may want to introduce those variables into the appropriate places in the outcome set. We do this by applying the inverse of the action substitution to each example's set of changes when we compute the initial set of outcomes. So, for example, if we were learning outcomes for the action *flip(X)* that flips a single coin, our initial outcome set would be {h(X), t(X), nil} and search would progress as usual from there. We introduce parameters aggressively wherever possible, based on the intuition that if any on them should remain a constant, this should become apparent through the other examples.

3.3 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, $r.P_{\mathbf{O}}$. We want to learn the distribution that will maximize the rule score: this will be the distribution that minimizes the negative log likelihood of the examples $\mathbf{D}$ covered by r, as given by

$$L(\mathbf{D}|M) = -\sum_{D \in \mathbf{D}} \log(P_{D.a}(D.\mathbf{S}^{t+1}|D.\mathbf{S}^t)) = -\sum_{D \in \mathbf{D}} \log\left(\sum_{\{o|D \in \mathbf{D}_o\}} r.P_{\mathbf{O}}(o)\right) \quad (3)$$

[1] Whenever we propose a set of outcomes we have to estimate its parameters. Often, many of the outcomes will have zero probability. These zero probability outcomes are immediately removed from the outcome set since they don't contribute to the likelihood of the data and they add complexity. This optimization greatly improves the efficiency of the search.

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

Table 1: The average changes 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.

where $\mathbf{D}_o$ is the set of examples covered by outcome o, selected as in Equation 1. Note that the probability $P_{D.a}$ is defined as in Equation 1, as the sum over the outcomes covering D. When every example is covered by a unique outcome, the problem of minimizing L is relatively simple. We can use a Lagrange multiplier to enforce the constraint that $r.P_{\mathbf{O}}$ must sum to 1.0. The partial derivative of L with respect to $r.P_{\mathbf{O}}(o)$ is then simply $|\mathbf{D}_o|/r.P_{\mathbf{O}}(o) - \lambda$, and $\lambda = |\mathbf{D}|$, so that $r.P_{\mathbf{O}}(o) = |\mathbf{D}_o|/|\mathbf{D}|$ and the parameters can be estimated by simple counting. However, in general, the partials can 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 minimizing a convex function over a probability simplex. Several gradient descent algorithms with guaranteed convergence can be found in Bertsekas (1999). We have implemented the *conditional gradient method*, which works by, at each iteration, moving along the axis with the minimal partial derivative. The step-sizes are chosen using the Armijo rule (with the parameters $s = 1.0$, $\beta = 0.1$, and $\sigma = 0.01$.) We stop the algorithm once the improvement in L is very small, less than 10^{-6}.

4 Experiments

In this section, we present a set of experiments to demonstrate that our algorithm does learn probabilistic planning rules effectively. All of the experiments use data generated by a combination of random and guided exploration. (Here, guided exploration means that only actions satisfying the preconditions of some rule were executed.) Thus, we are ignoring the exploration problem and assuming out data has successful examples of the actions we are trying to learn. After training our models on a set of training examples $\mathbf{D}$, we test them on a set of test examples $\mathbf{E}$ by calculating the average *variational distance* between the true model P and our estimate $\hat{P}$, $\frac{1}{|\mathbf{E}|}\sum_{E\in\mathbf{E}}|P(E) - \hat{P}(E)|$.

4.1 Inducing Outcomes

Consider the the coin-flipping domain, where n coins are flipped using three atomic actions: *flip-coupled*, which, as described before, 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 ruleset contains only a single rule and the whole problem reduces to that of outcome induction: thus, learning rules for coin-flipping actions allows us to focus on how our outcome induction algorithm works. Table 1 contrasts the number of outcomes in the initial rule with the number eventually induced by our algorithm. Notice that, given n coins, the optimal number of outcomes for *flip-coupled* is 2, for *flip-a-coin*—$2n$, and for *flip-independent*—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

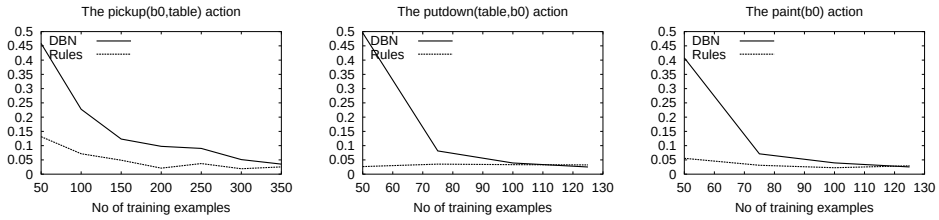

Figure 2: Variational distance as a function of the number of training examples for DBNs and propositional rules in a world with four blocks. The results are averaged over 5 runs of the experiment.

problem, and *flip-coupled* behaves like the sort of action we expect to see frequently. The table shows that our algorithm can handle 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 our reach. An ideal learning algorithm might notice such behaviour and choose to represent it with a factored model.

4.2 Comparison to DBNs

The slippery gripper domain from Section 2 was originally used by Draper et al. (1994) as a domain for propositional planning. Traditionally, *Dynamic Bayesian Networks* (DBNs) have been used to learn this type of world dynamics. To compare our algorithm to DBN learning, we disallow variable abstraction, thereby propositionalizing our rules. We implemented the Bayes net learning algorithm of Friedman and Goldszmidt (1998), which uses decision trees to represent its conditional probability distributions. Figure 2 shows that our algorithm learns better models from less training data. This is not a surprising result, since our representation is tuned for the world structure we are learning. Instead, we consider it a simple validation that our algorithm works as expected.

4.3 The Advantages of Abstraction

To further motivate our approach, we demonstrate that abstracting over objects can greatly facilitate learning. We compare our full algorithm to its propositional version by using it to learn models for the slippery gripper domain based on a fixed number of examples. The variables should allow the model to generalize more effectively, even as the world grows in size and complexity. And, indeed, Figure 3 demonstrates that, as we scale the size of the domain, the full version consistently out-performs the propositional version. In addition, the abstracted representation is clearly more compact. Since each iteration of our algorithm changes a single rule, this makes abstracted rules much faster to learn. (For example, in eight-block worlds, a propositional ruleset takes 7000-odd steps to converge, while an abstracted one takes only around 300.)

One could also imagine running experiments comparing our relational rule learning to learning a model of the dynamics of the world using probabilistic relational models (Getoor, 2001), since it is possible, though quite complicated, to represent our planning domains as PRMs. However, we expect that our representation would be able to leverage its bias much as it does against DBNs in the propositional case.

5 Conclusions and Further Work

Throughout this paper, we have identified many possible improvements for specific algorithms. We plan to investigate these changes, as well as to extend the work to more complex domains.

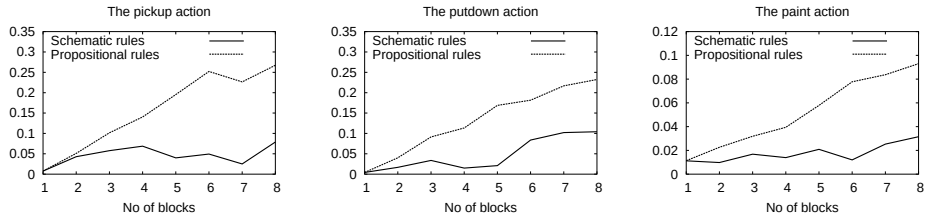

Figure 3: An illustration of how the variational distance of the models learned scales with the size of the world, for propositional and schematic representations. The results were averaged over 500 examples of each action, and over three runs of the experiment.

Our planning operators exploit an important general principle in modeling agent-induced change in world dynamics: each action can only have a few possible outcomes. In the simple examples we worked with in this paper, this assertion was exactly true in the underlying world. In real worlds, this assertion may not be exactly true, but we can still apply this principle. 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. We will investigate the concurrent construction of abstract state characterizations that allow us to continue to model actions as having few outcomes.

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, we plan to find structural leverage that we can exploit to obtain good approximate models efficiently.

References

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

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

Blum, A., & Langford, J. (1999). Probabilistic planning in the graphplan framework. In *ECP*.

Boutilier, C., Reiter, R., & Price, B. (2001). Symbolic dynamic programming for first-order mdps. In *IJCAI*.

Draper, D., Hanks, S., & Weld, D. (1994). Probabilistic planning with information gathering and contingent execution. In *AIPS*.

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

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

Garey, M. R., & Johnson, D. (1979). *Computers and Intractability: A guide to the theory of NP-completeness*. Freeman.

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

Oates, T., & Cohen, P. R. (1996). Searching for planning operators with context-dependent and probabilistic effects. In *AAAI*.

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

Shen, W.-M., & Simon, H. A. (1989). Rule creation and rule learning through environmental exploration. In *IJCAI*.