Multi-Resolution Planning in Large Uncertain Domains
NASA Grant NCC2-1237 Final Report

Leslie Pack Kaelbling, Principal Investigator

September 3, 2005

This project spanned three and one half years of research. This report covers three major lines of work that were done most recently and reported on in a talk given by the PI at NASA/Ames on March 23, 2004. There are have been additional publications related to this work (Lane & Kaelbling, 2001a, 2001b, 2002; Zettlemoyer, Pasula, & Kaelbling, 2003; Gardiol & Kaelbling, 2003; Pasula, Zettlemoyer, & Kaelbling, 2004).

CHAPTER 1

Computing action equivalences for planning under time-constraints

NATALIA H. GARDIOL AND LESLIE PACK KAELBLING

Abstract

In order for autonomous artificial decision-makers to solve realistic tasks, they need to deal with the dual problems of searching through large state and action spaces under time pressure. We study the problem of planning in domains with lots of objects. Structured representations of action can help provide guidance when the number of action choices and size of the state space is large. We show how structured representations of action effects can help us partition the action space in to a smaller set of approximate equivalence classes. Then, the pared-down action space can be used to identify a useful subset of the state space in which to search for a solution. As computational resources permit, we then allow ourselves to elaborate the original solution. This kind of analysis allows us to collapse the action space and permits faster planning in much larger domains than before.

1.1 Introduction

In many logical planning domains, the crux of finding a solution often lies in overcoming an overwhelmingly large action space. Consider, just to illustrate, the classic blocks world domain: the number of ways to make a stack of a certain height grows exponentially with the number of blocks on the table; and if the outcomes of actions are uncertain, this apparently simple task becomes even more daunting. We want planning techniques that can deal with large state spaces and large, stochastic action sets; most compelling, realistic domains have these characteristics.

In order to describe large stochastic domains compactly, we need relational structures that can represent uncertainty in the dynamics. Relational representations allow the structure of the domain to be expressed in terms of object *properties* rather than object identities and thus yield a much more compact representation of a domain than the equivalent propositional version can.

Even planning techniques that use relational representations, however, often end up operating in a fully-ground state and action space when it comes time to find a solution. However, it is often true that many action instances have similar kinds of effects: for example, in a blocks world it often does not matter which block is picked up first as long as a stack of blocks is produced in the end. If it were possible to identify under what conditions actions produce equivalent kinds of effects, the planning problem could be simplified by considering a representative action (from each equivalence class) rather than the whole action space.

This work is about taking advantage of structured, relational action representations. First, we want to identify logically similar effects in order to reduce the effective size of the action space; second, we want to limit the state space under consideration to an informative, reachable subset.

1.2 Relational Envelope-based Planning

Decision-making agents are often faced with complicated problems and not much time in which to find a solution. In such situations, the agent is better off acting quickly – finding *some* reasonable solution fast – than acting perfectly. Relational Envelope-based Planning (REBP) (Gardiol & Kaelbling, 2003) is a planning approach designed for time-pressured decision-making problems.

REBP proceeds in two phases. First, given a planning problem, an initial plan of action must be quickly found. Knowledge about the structure of action effects is used to eliminate potentially redundant information and focus the search onto high-probablility sequences of actions, known as an *envelope* of states (Dean, Kaelbling, Kirman, & Nicholson, 1995). Second, if the agent is given additional time, it can elaborate the original plan by considering lower-probability consequences of its action choices. Figure 1.1 shows a very high-level system diagram of the main parts of the REBP system.

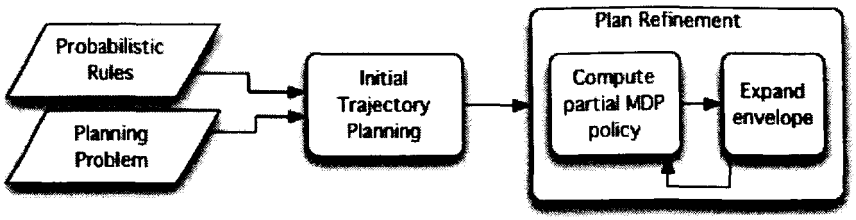

Figure 1.1: A high-level schematic of the REBP system. The input to the system is twofold: a set of probabilistic rules. and a description of the planning problem at hand. The next process is to find an initial plan quickly. The final process is to refine the initial plan as resources permit.

REBP lets us reason with ground states and actions, but, it is structured so as to limit how much of the ground state and action space is considered at a time. REBP explicitly inhabits the space between a plan (a sequence of actions computed for a given state/goal pair) and a policy (a mapping from all states in a space to the appropriate action). This allows an agent to make a plan that hedges against the most likely deviations from the expected course of action, without requiring construction of a complete policy. It produces an initial plan quickly by taking advantage of generalization among action effects, and as a result behaves smarter in a large space much sooner than it could by waiting for a full solution. Using the relational envelope method, we can take real advantage of relational generalization to produce good initial plans efficiently, and use envelope-growing techniques to improve the robustness of our plans incrementally as time permits. More details on REBP are available in our previous work (Gardiol & Kaelbling, 2003).

The trick, at least to start, is in producing the initial envelope efficiently. This paper will address our first steps in that direction.

1.2.1 Relational representation of actions and states

We cast our planning problem in the framework of a Markov decision process (MDP) (Puterman, 1994). An MDP is a tuple, $\langle \mathcal{Q}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$ where: $\mathcal{Q}$ is a set of states; $\mathcal{A}$ is a set of actions; $\mathcal{R}$ is a reward function mapping each state to a real number; and $\mathcal{T}$ is a transition function mapping each state-action-state triple to a probability. A *solution* for an MDP consists in finding the best mapping from states to actions in a way that maximizes long-term reward. This function, π, is called a *policy*.

In the past, much work on finding policies for MDPs considered a state to be an atomic entity; this approach is known not to scale to large state spaces, and much recent literature has been devoted to smarter ways of representing problems.

In this work, we take advantage of a more compact way of representing state transitions (i.e.,

```
(:action pick-up-block-from
 :parameters (?top - block ?bottom - object)
 :precondition
   (and (on-top-of ?top ?bottom) (not (= ?top ?bottom)) (on-top-of ?top ?bottom)
      (forall (?b - block) (not (holding ?b)))
      (forall (?b - block) (not (on-top-of ?b ?top))))
 :effect (probabilistic
   0.9 (and (holding ?top) (not (on-top-of ?top ?bottom)))
   0.1 (and (forall (?b - block)) (not (on-top-of ?b ?top)) (on-top-of ?top table))))
```

Figure 1.2: An example of a probabilistic relational rule for blocks-world dynamics in the PPDDL formalism. Each rule schema contains the action name, arguments, precondition, and a set of outcomes. In this case, the first probabilistic outcome is the "successful" outcome of picking up the block; the second outcome denotes a less likely "failure" outcome, in which the block falls onto the table.

actions). That is, rather than a state being composed of a set of propositional features, we think of it as being composed instead of a set of logical relationships between domain objects. Since these predicates can make assertions about logical *variables*, a single predicate may in fact represent a large number of ground propositions. This lets use a single transition rule to represent many ground state transitions.

We define a relational MDP (RMDP) as a tuple $\langle \mathcal{P}, \mathcal{Z}, \mathcal{O}, \mathcal{T}, R \rangle$:

States: The set of states is defined by a finite set $\mathcal{P}$ of relational predicates, representing the properties and relations that can hold among the finite set of domain objects, $\mathcal{O}$. Each RMDP state is a ground interpretation of the domain predicates over the domain objects.

Actions: The set of ground actions depends on the set of rules $\mathcal{Z}$ and the objects in the world.

Transition Dynamics: For the transition dynamics, we use a compact set of rules similar to probabilistic STRIPS rules (Fikes & Nilsson, 1971). A rule's behavior is defined by a precondition and a probabilistic effect, each expressed in terms of logical predicates. A probabilistic effect describes a distribution over a disjoint set of logical outcomes. A rule applies in a state if its precondition can be matched against some subset of the state ground predicates. Each outcome then describes a possible resulting ground state. An example is shown in Figure 1.2. In our system, we currently use rules that are designed by hand; they may, however, be obtained via learning.

For each action, the distribution over next states is given compactly by the distribution over outcomes encoded in the rule schema. The rule outcomes themselves usually only specify a subset of the domain predicates, effectively describing a set of possible resulting ground states. To fill in the values of the domain predicates not menioned in the outcome, we assume a static frame: state predicates not directly changed by the rule are assumed to remain the same.

Rewards: A state is mapped to a scalar reward according to function $R(s)$.

The original version of envelope-based planning (Dean et al., 1995) used an atomic MDP representation. To extend envelope-based planning to relational domains, then, we need two things. First, we need a set of probabilistic relational rules, which tell us the transition dynamics for a domain; and second, we need a problem description, which tells us the states and reward. These are the two input items in the diagram in Figure 1.1.

A to define a planning problem we have to specify the following elements. The *initial world state* is the set of ground predicates that describes the starting state. The *goal condition* is a logical sentence. The *reward* is specified by list of logical conditions mapping states to a scalar reward value. If a state in the current MDP does not match a reward condition, the default value is 0. Additionally, there must be a penalty associated with falling out of the envelope. This penalty is an estimate of the cost of having to recover from falling out (such as having to replan back to the envelope, for example).

1.2.2 Initial trajectory planning

Given a set of rules and the problem description, the next step in envelope-based planning is finding the initial envelope. In a relational setting, when the underlying MDP space implied by the full instantiation of the representation is potentially huge, a good initial envelope is crucial. It determines the quality of the early envelope policies and sets the stage for more elaborate policies later on.

Blum and Langford (Blum & Langford, 1999) describe a probabilistic extension to the Graphplan algorithm (Blum & Furst, 1997), called TGraphplan (TGP), that can find the shortest straight-line plan quickly from start to goal that satisfies a minimum probability. We use the trajectory found by TGP to populate our initial envelope.

Initial plan construction essentially follows the TGP algorithm described by Blum and Langford (Blum & Langford, 1999). The TGP algorithm starts with the initial world state as the first layer in the graph, a minimum probability cutoff for the plan, and a maximum plan depth. The TGP algorithm produces a sequence of actions.

However, our relational MDP describes a large underlying MDP. In a STRIPS -like rule, every variable in the rule schema appears in the argument list; so, when a STRIPS rule is grounded, it yields an exponential number of actions as the number of domain objects grows. Since large numbers of actions will grind TGP to a halt, we want to avoid considering all the actions during our plan search.

To cope with this problem, we have identified a technique called *equivalence-class sampling*. We partiiton into equivalence classes the actions that produce similar effects on the properties of the variables in their scope. Then, the plangraph can be constructed chaining forward only a sampled

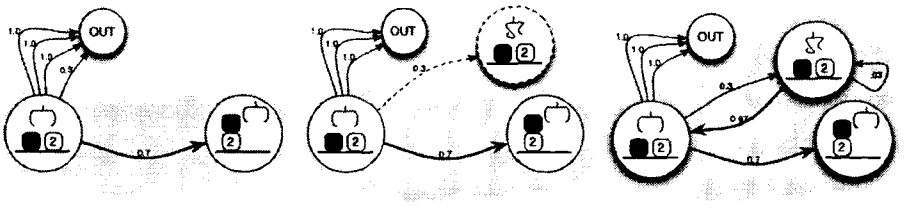

Figure 1.3: An illustration of the envelope-based approach to planning. The task is to making a two-block stack in a domain with two blocks. The initial envelope is shown (far left), followed by a step of deliberation (i.e., consideration of deviations from the initial plan) (middle), and finally after envelope expansion and computation of a new mdp policy (far right).

action from each class. The sampled action is *representative* of the effects of any action from that class. Sampling reduces the branching factor at each step in the plangraph, so significantly larger domains can be handled.

1.2.3 Equivalence in relational domains

Before we can determine whether two actions are equivalent, we must define when two objects are equivalent. This requires the following crucial assumption:

Remark 1 (Sufficiency of Object Properties)**:** We assume an object's identity is determined solely by its properties and relations to other objects, and not by its name.

So, then, what do we mean by equivalent objects? Intuitively, we mean to say that two objects are equivalent to each other if they are related in "similar" ways to other objects that are, in turn, equivalent.

Evaluating equivalence is tricky, of course, because it sets off a "chain reaction": determining whether two objects are equivalent requires looking at the objects that they are related to; we have to "push" through each relation an object participates in. How do we know that this process will stop, and that it will give us an answer? Previous work on object equivalence, or symmetry, has used single, unary predicates as a basis for computing similarity (Ellman, 1993; Fox & Long, 1999, 2002).

We would like to explore object equivalence when more complex relationships come into play.

To generalize our concept of equivalence, we start by considering a relational state description as a graph. We can think of the objects or variables in the domain as nodes in the graph, and the relations between them as directed arcs.

Definition 1 (State Relation Graph)**:** A *state relation graph* is a labeled, directed graph. Nodes represent objects in the domain, and arcs represent relations. The arcs are labeled with the relation

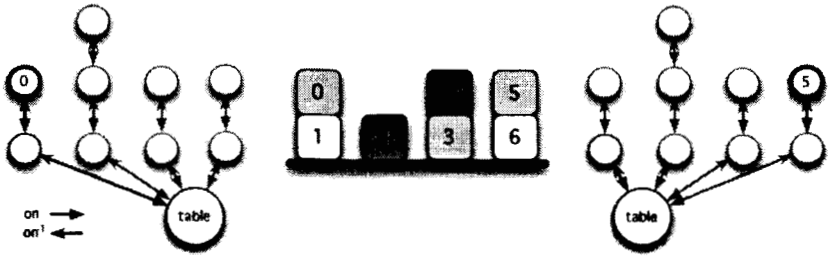

Figure 1.4: A blocks-world arrangement (middle), and corresponding object relation graphs for blocks 0 and 5 (at eft and right, respectively).

name. For every pair of related nodes, we connect them with one directed arc for the original relation and an opposite arc for the inverse of the relation. If the domain is typed, then nodes are labeled with the object's type. See Figure 1.4 for an example.

Definition 2 (Object Relation Graph)**:** An *object relation graph* is constructed from the state relation graph by cutting out all nodes to which the object is not connected by a outgoing directed path. We keep the relation labels on the arcs (and type labels on the nodes, if they exist) and label the object's node with its name. No other nodes are labeled.[1]

Definition 3 (Equivalence)**:** Two *objects* are *equivalent* if their relation graphs are isomorphic to each other. Two *states* are *equivalent* if there exists a bijective function that maps each object in the first state to an equivalent object in the second state. Two *actions* are *equivalent* if the objects in each action's argument list are respectively equivalent in the current state.

Theorem 1: Consider a complete planning procedure P,[2] the graph-isomorphism definition of state equivalence, and the above definition of action equivalence. If P uses such a definition of equivalence to partition its action set into equivalence classes, and then plans with a reduced set of actions consisting of a representative from each equivalence class, then P continues to be a complete planning procedure as long as we are willing to sacrifice plan parallelism.

Proof 1 (Sketch)**:** A plan is a sequence of actions that yields a sequence of states from start to goal. Substituting one action for an equivalent one, then, will replace one state with a state whose relation graph is isomorphic to it. This means all of the objects in the replacement state will be equivalent to the objects in the replaced state, and thus the preconditions for the next action in the sequence will be satisfied. This means the remainder of the plan is still valid. Thus, any serial plan that existed before in the full action space will exist in the new, collapsed action space.

[1] In the case of relation with more than two arguments, we would have to consider a hypergraph to allow for edges of more than 2 nodes.

[2] a *complete* planning procedure is one which is guaranteed to find a solution if it exists

1.2.4 Planning with equivalent actions in TGraphPlan

We would like to use this notion of action equivalence in the REBP planning framework. That is to say, we want to group two instances of an planning operator into the same equivalence class when the ground objects in each argument list are respectively isomorphic to each other in the current state.

When we try to apply this definition into the TGraphplan setting, however, the following issue immediately crops up: in each layer of the plan graph, there is no notion of "current state." In the Graphplan algorithm, the first level in the graph (step 0) contains the propositions corresponding to the facts in the initial state. Each level beyond the first contains two layers: one layer for all the actions that could possibly be enabled based on the propositions on the previous level, and a layer for all of the possible effects of those actions. Thus, each level of the plan graph simply contains a list of all propositions that could conceivably be true. The only information at our disposal is that of which propositions are mutually exclusive from one another.

In order to partition actions into equivalence classes, we adopt the following criteria. We define the *extended state* of an action to be all those propositions in the current layer that are not mutually exclusive with any of the action's preconditions. Thus, we group two actions together if the ground objects in each argument list are isomorphic to each other with respect to each action's *extended state*.

Intuitively, this criteria will create a finer set of equivalence classes than our original one, since the set of propositions that could be possibly true is greater than those that will actually become true. We conjecture, without proof for the moment, that the equivalence classes produced by this criteria will be at least exactly those produced by Theorem 1. Most likely, the resulting equivalence classes will be simple refinements (i.e., finer partitions) of those of Theorem 1.

1.3 Preliminary results

We have very some preliminary, illustrative, results for a small stacking task in the version of blocks world from the ICAPS 2004 probabilistic planning competition (Younes & Littman, 2003). In this version of the blocks world, blocks are related by the *on-top-of* relation; "clear" blocks are those which have no other blocks on top of them. Our experimental domain has seven blocks, arranged as in Figure 1.4.

The goal is to put any three blocks into a single stack; as written in PPDDL:

```
(:goal (and  (exists (?fb0 - block) (exists (?fb1 - block) (exists      (?fb2 - block)
             (on-top-of ?fb2 ?fb1) (on-top-of ?fb1 ?fb0) (on-top-of ?fb0 table))))))
```

A correct plan is a sequence of pick-up and put-down actions. In Figure 1.5, the highlighting in the

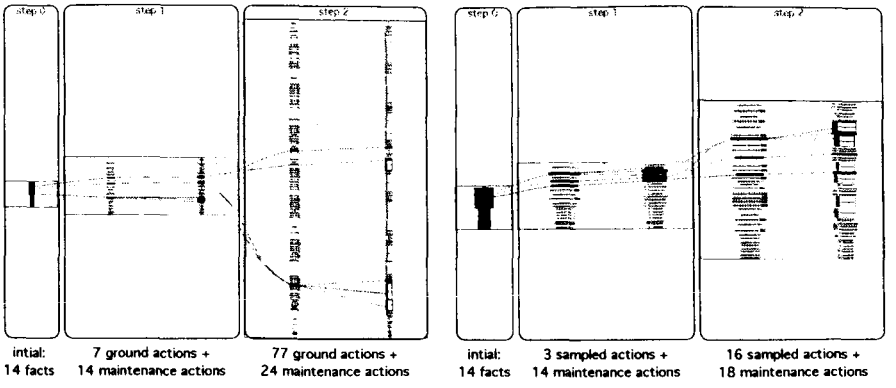

Figure 1.5: Two versions of a plan graph for achieving a 3-block stack in a world with 7 blocks. On the left, we show a plan graph without equivalence sampling: it populates each level with all possible action instantiations. The plan graphs are too small to interpret closely, but it is possible to get a sense for their relative density. Even for such a small task, without equivalence sampling the plan graph becomes dense quickly. In contrast, on the right, a plan graph with equivalence sampling: it uses only representative actions from each equivalence class for each level. The reduction in actions considered at each layer is significantly reduced.

plan graph denotes the selected action in each step[3]: in step 1, pick up block 4 from the table; and in step 2, put down block 4 on block 2. There are many instances of each pick-up and put-down operator in each level, but many acheive the same qualitative result.

In Figure 1.5, we can appreciate the effect of using equivalence class sampling in even the first level of the plan graph. In the step 1, the algorithm identifies two equivalence classes, a reduction from four ground instantiations. In the second layer, it is more dramatic: from 47 instantiations to 11 equivalence classes. Even in this tiny experiment, the effect of reducing the branching factor has a significant impact on the plan computation time.

In layer one, the algorithm identified two classes, shown in Table 1.3. They distinguish between picking a block from a table and picking a block from the top of a stack.

In layer two, 11 classes were found (see Table 1.3. These classes are harder to intepret intuitively, but they are nevertheless revealing. For example, there is a class (5) which groups together all the ways to put down block 4 on the top of a stack. However, there are two classes (1 and 4) distinguishing between the set of circumstances for picking up block 4 from the table, and for picking up blocks 1, 3, or 6 from the table (which were covered by other blocks in the previous step). We are currently experimenting with larger instances of the blocks world domain, as well as

[3]In addition to the explicitly chosen planning action, the highlighting also shows any required "maintenance" or "frame" actions, which maintain the state from one level to the next.

Class	Members
[1]	pick-up-block-from (block4, table)
[2]	pick-up-block-from (block2, block3)
	pick-up-block-from (block5, block6)
	pick-up-block-from (block0, block1)

Figure 1.6: Two equivalence classes found in Step 1 of the plan graph of Figure 1.5. The sampled (representative) action is listed first.

a variety of logistics domains from the ICAPS 2004 competition.

1.4 Conclusion

Our objective is to plan in large domains in which there is time-pressure to begin acting appropriately. To this end, we seek to take advantage of an envelope-based planning framework, which explicitely considers low-complexity plans first, and higher-complexity plans as time permits. The difficulty of envelope-based approaches, however, is always this: what is the best way to populate the initial envelope?

If our domain is represented relationally, then it makes sense to leverage the fast classical planning techniques for finding straight-line plans, such as Graphplan. However, when our domains have many objects in them (which is precisely the setting we want to consider), the number of ground action instantiations creates an insurmountable branching factor for Graphplan and descendent algorithms. They are unable to find the straight-line plan we need in order to create an initial envelope.

Thus, for envelope-based approaches to scale, it is crucial to find a way to prune out action instantiations that do not achieve qualitatively different outcomes.

There is a host of preceeding approaches that attempt to identify symmetries in the problem to collapse actions together (Ellman, 1993; Fox & Long, 1999, 2002). Unfortunately, in the existing work in this area there is a rather weak notion of what makes objects equivalent. These notions rely primarily on unary predicates, and they are hard to reconcile with a relational domain in which predicates have more than one argument.

This is the first work that we know of that explicitly attempts to define what it means for planning operators to be equivalent in the presence of complicated relational structure. This is work is an initial attempt at formalizing such a definition, and to apply it within the context of envelope-based planning.

Class	Members
[1]	pick-up-block-from (block2, table)
[2]	pick-up-block-from (block2, block3) pick-up-block-from (block5, block6) pick-up-block-from (block0, block1)
[3]	pick-up-block-from (block4, table)
[4]	pick-up-block-from (block1, table) pick-up-block-from (block3, table) pick-up-block-from (block6, table)
[5]	put-down-block-on (block4, block2) put-down-block-on (block4, block5) put-down-block-on (block4, block0)
[6]	put-down-block-on (block4, block3) put-down-block-on (block4, block6) put-down-block-on (block4, block1)
[7]	put-down-block-on (block2, block6) put-down-block-on (block2, block1)
[8]	put-down-block-on (block2, block4) put-down-block-on (block2, block3)
[9]	put-down-block-on (block2, block0) put-down-block-on (block2, block5)
[10]	put-down-block-on (block2, table)
[11]	put-down-block-on (block4, table)

Figure 1.7: Eleven equivalence classes found in Step 2 of the plan graph of Figure 1.5. The sampled (representative) action is listed first. The classes in this step less obvious to interpret than in the first, since these classes are based on all possibly true propositions (not necessarily the ultimate state) of Step 1.

CHAPTER 2

Learning Probabilistic Relational Planning Rules

HANNA M. PASULA, LUKE S. ZETTLEMOYER, AND LESLIE PACK KAELBLING

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.

2.1 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).

2.1.1 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 slightly 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.

2.2 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.

2.2.1 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.

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

Figure 2.1: Four relational rules that model the action dynamics of a simple blocks world.

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

$$
\begin{aligned}
&on(a,b), on(b,\text{TABLE}), \neg on(b,a), \neg on(a,\text{TABLE}), \\
&inhand(\text{NIL}), clear(a), block(a), block(b), \neg clear(b), \\
&\neg inhand(a), \neg inhand(b), \neg block(\text{TABLE})
\end{aligned} \tag{2.1}
$$

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

2.2.2 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 simply 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, $f_o : S \rightarrow S$, as described shortly. Finally, r_P is a 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 2.1 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 rule $r \in \mathbf{R}$ 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 all of 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 leads to an outcome containing both a literal and its negation.

As an example, imagine an agent wants to predict the effects of executing $pickup(a, b)$ in the world described in (2.1) given the model represented by the rule set in Figure 2.1. The action unifies with the action of the first rule, producing the substitution $\theta = \{X/a, Y/b\}$; fails to unify with the second rule's action, because b doesn't equal TABLE; and fails to unify with the remaining rules since they have different action predicates. When we apply θ to the first rule, we can see that its outcomes contain no contradictions; note, however, that if the action a had been $pickup(a, a)$ then the first outcome would have contained one. The context, meanwhile, becomes $\{on(a, b), clear(a), inhand(\text{NIL}), block(b)\}$. Since this set is a subset of the state description

in (2.1), 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 $o \in r_O$ 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.

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

$$\begin{aligned} P(S'|S, A, r) &= \sum_{o \in r_O} P(S', o|S, A, r) \\ &= \sum_{o \in r_O} P(S'|o, S, A, r) P(o|S, A, r) \end{aligned} \tag{2.2}$$

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_O could be used to describe the transition to the successor state s'. As an example, consider a rule for painting blocks,

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

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

2.3 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.

2.3.1 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, $S(\mathbf{R}) =$

$$\sum_{(s,a,s') \in \mathbf{D}} \log(P(s'|s, a, \mathbf{R})) - \alpha \sum_{r \in \mathbf{R}} PEN(r) \tag{2.3}$$

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 $|r_C| + |r_O|$. 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 simply left out of the rule set.

[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.

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 & Raedt, 1998), is an important area for future work.

2.3.2 Inducing Outcomes

The effectiveness and efficiency of the *LearnRules* algorithm are limited by those of the *InduceOutcomes* sub-procedure, 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_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 (2.3).

In general, outcome induction is NP-hard (Zettlemoyer et al., 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 2.2 where $h(C)$ stands for

$D_1 = t(c1), h(c2) \rightarrow h(c1), h(c2)$
$D_2 = h(c1), t(c2) \rightarrow h(c1), h(c2)$
$D_3 = h(c1), h(c2) \rightarrow t(c1), t(c2)$
$D_4 = h(c1), h(c2) \rightarrow h(c1), h(c2)$

(a)

$O_1 = \{h(c1)\}$
$O_2 = \{h(c2)\}$
$O_3 = \{t(c1), t(c2)\}$
$O_4 = \{\text{no change}\}$

(b)

Figure 2.2: (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).

$heads(C)$, $t(C)$ 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 2.2.

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 2.2, 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 2.2.

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 $\{O_1 = \{\text{h(X)}\}, O_2 = \{\text{t(X)}\}, O_3 = \{\text{no change}\}\}$ 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.

2.3.3 Learning Parameters

Given a rule r with a context r_C and a set of outcomes r_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

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

$$= \sum_{(s,a,s')\in\mathbf{D}_r} \log\left(\sum_{\{o|D\in\mathbf{D}_o\}} r_P(o)\right) \tag{2.4}$$

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.

2.4 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.

2.4.1 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.

	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	-	-

Figure 2.3: 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.

2.4.2 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 2.3 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.

2.4.3 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 2.4.

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.

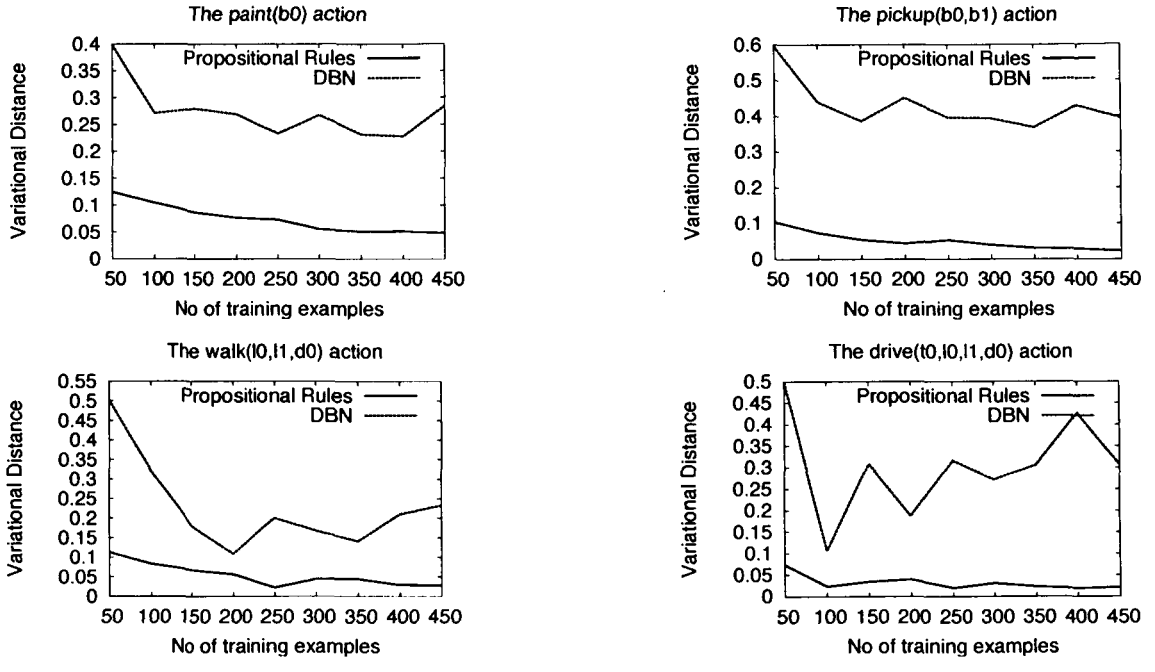

Figure 2.4: Variational distance as a function of the number of training examples for DBNs and propositional rules. The results are averaged over ten trials of the experiment. The test set size was 300 examples.

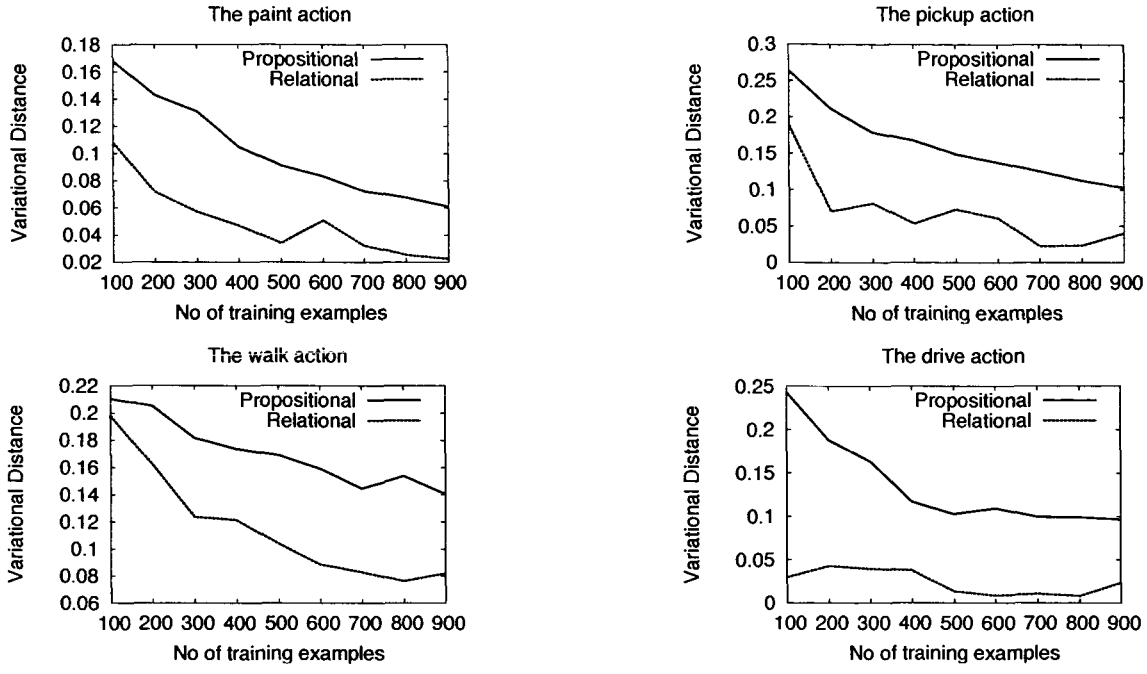

Figure 2.5: Variational distance as a function of the number of training examples for propositional and relational rules. The results are averaged over ten trials of the experiment. The test set size was 400 examples.

The Advantages of Abstraction

The second set of experiments demonstrates that when *LearnRules* is able to use variable abstraction, it outperforms the propositional version. Figure 2.5 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 2.6 shows that there is no real increase in test error. This is one of the major attractions of a relational representation.

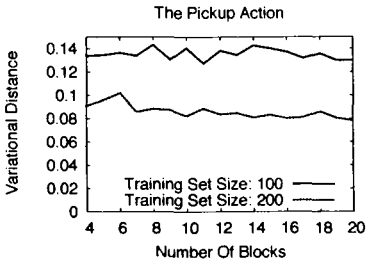

Figure 2.6: Variational distance of a relational rule set trained in a world four-block world, as a function of the number of blocks in the worlds on which it was tested. Results are given for three different training set sizes. The testing sets were the same size as the training sets.

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.

2.5 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. 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 some of the most complex models to date, including the knowledge required 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 consequences 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).

2.6 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.

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, algo-

rithms that leverage world structure should be able to obtain good approximate models efficiently.

CHAPTER 3

Learning Planning Rules in Noisy Stochastic Worlds

LUKE S. ZETTLEMOYER, HANNA M. PASULA AND LESLIE PACK KAELBLING

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.

3.1 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 3.1. 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 et al., 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.

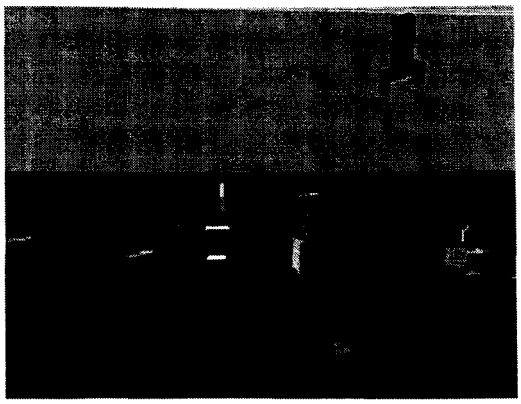

Figure 3.1: A screen capture of the simulated blocks world. The blocks come in various sizes, visible here, and various colors. The gripper can perform two macro actions: *pickup*, which centers the gripper above a block, lowers it until it hits something, closes it, and raises the gripper; and *puton*, which centers the gripper above a block, lowers until it encounters pressure, opens it, and raises it.

3.2 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.

3.2.1 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 con-

[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.

junctive 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

$$\mathit{on}(c_1, c_2) \wedge \mathit{on}(c_2, t) \wedge \mathit{inhand\text{-}nil} \wedge \mathit{clear}(c_1) \wedge \mathit{table}(t) \tag{3.1}$$

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

3.2.2 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 $\mathit{pickup}(c_1)$ could represent the action where the gripper attempts to pickup the block c_1 in the state represented in Sentence 3.1.

3.2.3 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, $\Psi'_1 \ldots \Psi'_n$ are *outcomes*, formulas that might hold in the next step, and $p_1 \ldots p_n$ 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 3.1:

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

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 3.1 and action $\textit{pickup}(c_1, c_2)$ 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:

$$
\begin{aligned}
P(s'|s, a, r) &= \sum_{i=1}^{n} P(s', \Psi'_i|s, a, r) \\
&= \sum_{i=1}^{n} P(s'|\Psi'_i, s, a, r) P(\Psi'_i|s, a, r) \qquad (3.2)
\end{aligned}
$$

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:

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

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

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

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 (3.2): 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]

3.2.4 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:

$$\begin{array}{l} \mathit{pickup}(X) : \left\{ \begin{array}{l} Y : \mathit{topstack}(Y, X), \\ Z : \mathit{on}(Y, Z), \\ T : \mathit{table}(T) \end{array} \right\} \\ \mathit{inhand\text{-}nil}, \mathit{height}(Y) < 9 \\ \quad \rightarrow \left\{ \begin{array}{ll} .80: & \neg \mathit{on}(Y, Z) \\ .10: & \neg \mathit{on}(Y, Z), \mathit{on}(Y, T) \\ .05: & \text{no change} \\ .05: & \text{noise} \end{array} \right. \end{array} \tag{3.3}$$

[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.

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

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.

3.2.5 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.

3.3 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.

LearnRuleSet(E)
Inputs:
Training examples **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 3.2: *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.

3.3.1 Learning Rule Sets

The *LearnRuleSet* algorithm takes a set of examples **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 3.2.

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 et al., 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 3.4.

- *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 3.3 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 3.4.

  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 $\sigma = \{X \rightarrow b\}$, and the action of r is set to $pickup(X)$. The context of r is set to the conjunction $inhand\text{-}nil, \neg inhand(X), clear(X), height(X) = 2, \neg on(X,X), \neg above(X,X), \neg topstack(X,X)$ 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 $\{X \rightarrow b, Y \rightarrow a\}$. The resulting rule r' looks as follows:

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

  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

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

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 3.4 shows the 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 3.4 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

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

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 3.4: *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.

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.

3.3.2 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 ap-

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

Figure 3.5: 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.

pear in the learned rules. The first set is created by applying the operators in Figure 3.5 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.

3.4 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 3.1, 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 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.

3.4.1 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 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.

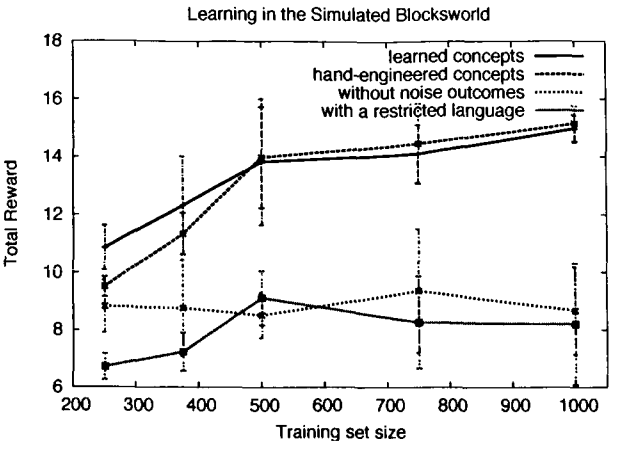

Figure 3.6: The performance of various action model variants as a function of the number of training examples. All data points were averaged over five runs each of three rule sets learned on different training data sets. For comparison, the average reward for performing no actions is 9.2, and the reward obtained when a human directed the gripper averaged 16.2.

3.4.2 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 3.6. 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., $p(X, Y) := clear(Y) \wedge on(Y, X)$, 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.

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

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.

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

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.

3.5 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.

CHAPTER 4

Kurt Steinkraus, Leslie Pack Kaelbling

Abstract

One of the reasons that it is difficult to plan and act in real-world domains is that they are very large. Existing research generally deals with the large domain size using a static representation and exploiting a single type of domain structure. In this paper, we create a framework that encapsulates existing and new abstraction and approximation methods into modules, and combines arbitrary modules into a system that allows for dynamic representation changes. We show that the dynamic changes of representation allow our framework to solve larger and more interesting domains than were previously possible, and while there are no optimality guarantees, suitable module choices gain tractability at little cost to optimality.

4.1 Introduction

Recent planning algorithms for deterministic and stochastic systems have improved considerably, allowing the solution of moderately large problems. When exact solution is required for relatively circumscribed applications, these approaches are clearly appropriate. In this paper, we wish to consider a somewhat different case, analogous to the action selection problem faced by a human, or by a robot operating in a highly complex and open-ended domain, such as disaster relief or general battlefield operations. In such domains, the world model is so big that, ideally, a planning algorithm would run in time that is effectively constant, independent even of the number of state variables in the model.

Our approach will be to use dynamic abstractions, so that the agent only ever needs to solve very small planning problem instances. This approach will, of necessity, give up on achieving optimal performance, instead emphasizing the ability to continue to behave without complete failure in situations of extreme complexity. As the agent moves through the environment, it represents the domain at multiple levels of abstraction. What makes our approach different from a variety of other abstraction methods is the dynamism of the abstraction: the agent's current view of its environment depends on the current state. In the current work, the adaptation methods are designed into the system; in future work, the agent should learn which abstractions are appropriately used in which circumstances.

This paper presents a framework for building agents using a dynamic combination of abstractions, describes a particular concrete set of abstraction methods, shows how they can be combined into a dynamically adapting hierarchy, and applies that hierarchy to the problem of controlling an agent in a moderately complex game domain.

4.2 Module hierarchy framework

We assume that the agent has a domain model expressed as a factored Markov decision process (MDP) and a high-level goal articulated as a reward function over the variables in the factored model. Both the MDP's transition functions and the reward function are represented using algebraic decision diagrams (ADDs). In addition, the agent has a hierarchy of abstraction modules that dynamically create a hierarchy of abstracted versions of the base-level domain model.

To help understand the framework, consider a trivial example of a robot that lives in a 10×10 gridworld (see figure 4.1(a)). The robot can carry packages, and its goal is to move them from one location to another. The robot's movement is stochastic, so that with some small probability it fails to execute the action it attempts, or it moves in the wrong direction. The robot has a battery whose charge gradually runs down from 1000 to 0 and needs to be charged periodically at a charger. Finally, the robot gets reward (discounted over time) for successfully transferring packages. This

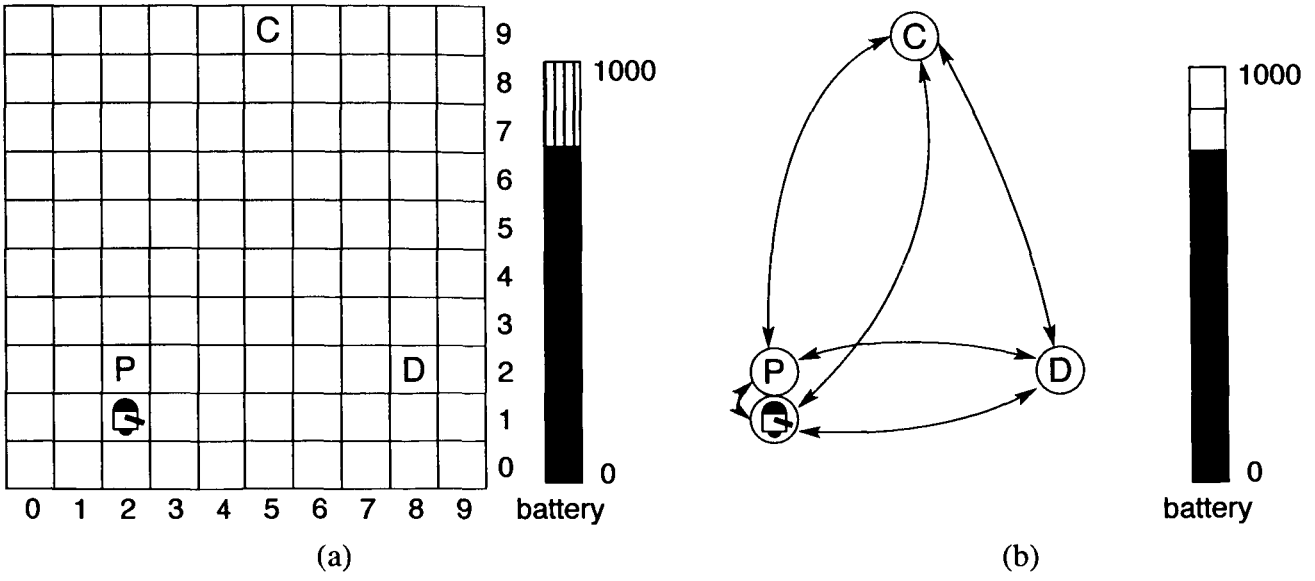

Figure 4.1: (a) The package and charger gridworld example. (b) An abstract version of the gridworld example.

domain has $10 \times 10 \times 2 \times 1000 = 200,000$ states and seven action values (*north*, *south*, *east*, *west*, *get*, *put*, and *charge*) in one action variable.

In this example, there are two different kinds of domain structure to exploit. First, a robot executing optimally will only ever want to go back and forth between three locations: the pickup, drop-off, and charger points, so the robot's view of the map can be abstracted into a smaller, topological version. Second, the battery sensor is more fine-grained than needed, so similar battery levels can be clustered together, say, in groups of 100 (see figure 4.1(b)).

Were the robot to attempt to find a policy for this domain using a single previous abstraction method such as state aggregation or the options framework (Precup, Sutton, & Singh, 1998), it would miss the chance to exploit both types of structure. Our framework, however, allows multiple abstraction methods to be used together, each focusing on the domain structure it is able to simplify.

In our framework, each abstraction method is packaged into an *abstraction module*, and individual modules are combined into a *module hierarchy*, which is then used to plan and act in a domain. The modules in the module hierarchy induce successive *i-models* (see figure 4.2). The top-level i-model is a trivial MDP with one state and one action (the action means "act in the domain"), while the bottom-level i-model, I^1, is identical to the input model.

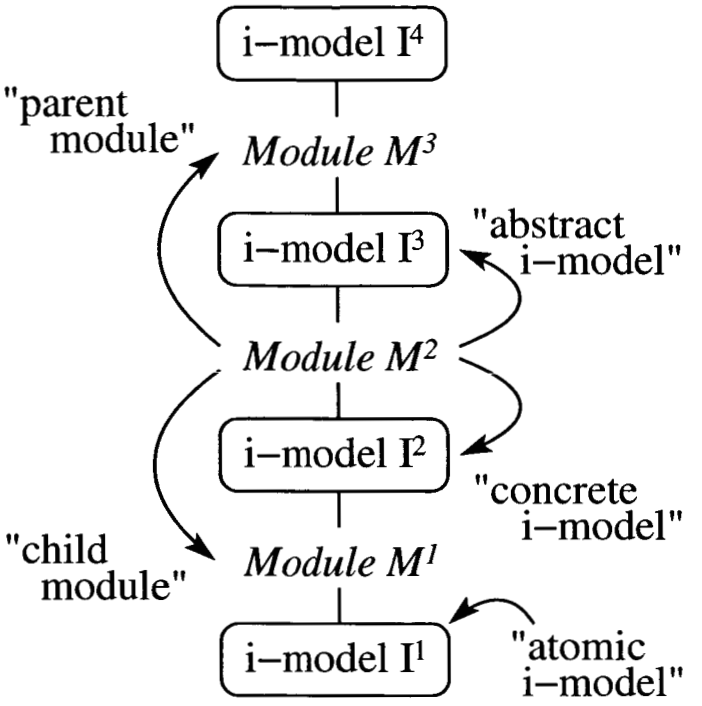

Figure 4.2: Module hierarchy nomenclature, with labels given relative to module M^2.

4.2.1 Intermediate models

Each i-model I^j is an intermediate representation for the domain and is created by module M^{j-1} looking at I^{j-1} and applying some abstraction. We let i-models be *factored semi-MDPs*, since they need to be able to represent both the input model, which is a factored MDP, and also any temporal abstraction information generated by abstraction modules.

It is difficult to come up with consistent, intuitive semantics for factored semi-MDPs when allowing multiple concurrent action variables. Rohanimanesh and Mahadevan (Rohanimanesh & Mahadevan, 2001), for instance, have tackled concurrent options, but the general case remains the subject of future research. For now, we therefore assume that, no matter how many action variables a domain contains, only one action variable can be active (i.e., can have an action selected) at any particular time step.

We define an i-model I as a tuple $(\overline{S}, \overline{A}, \tau, T, r, \gamma)$:

- $\overline{S} = \{S\}$ is a set of state variables; each S ranges over a set of state values $\{s\}$.
- $\overline{A} = \{A\}$ is a set of action variables; each A ranges over a set of action values $\{a\}$.

- τ is a time distribution, where $\tau : \prod_{S \in \overline{S}} S \times \bigcup_{A \in \overline{A}} A \times \mathbb{N} \rightarrow \mathbb{R}$ gives the probability distribution over lengths of time that each action will take in each state.
- $T = \{t_k\}$ is the set of transition probability distributions, one per state variable $S_k \in \overline{S}$, that the overall transition probability distribution is factored into. Each $t_k : \prod_{S \in \overline{S_k}} S \times \bigcup_{A \in \overline{A_k}} A \times \mathbb{N} \times S_k \rightarrow \mathbb{R}$ gives the probability distribution over S_k at time $t+1$, given the relevant portions of the state space, $\overline{S_k} \subseteq \overline{S}$, at time t, and the relevant portions of the action space, $\overline{A_k} \subseteq \overline{A}$. Each t_k is also conditioned on the action's duration.
- r is a reward function, where $r : \prod_{S \in \overline{S}} S \times \bigcup_{A \in \overline{A}} A \rightarrow \mathbb{R}$.
- γ is a discount factor, where $0 < \gamma < 1$.

We assume that the action values in different action variables A are unique. Therefore, $\bigcup_{A \in \overline{A}} A$ gives the set of all possible action choices (remember, only one action variable is active at any one time).

In each i-model, the distributions τ and $\{t_k\}$ as well as the reward function r are represented as ADDs. Using ADDs allows for a much more compact representation than, say, using tables or even using normal decision trees.

4.3 Abstraction modules

The abstraction modules M^j are the heart of the module hierarchy. Each module must conform to the following interface:

- Each module M^j must create an abstract i-model I^{j+1} from i-model I^j. I^{j+1} should contain approximately the same information as I^j, except that some structure or redundancy will have been factored out in an attempt to make the domain simpler.

  M^j must be able to create I^{j+1} so that a specified atomic state is representable in I^{j+1}, given that it is representable in I^j. During initial planning, this will be the starting state, but it may change over the course of execution.

- Each module M^j must respond to requests for re-abstraction, so that I^{j+1} changes appropriately when I^j changes. Ideally, the new I^{j+1} will be only slightly different than the old one, and this allows modules to reuse a lot of the information from the old I^{j+1} when computing the new one.

  The actual approach that each module takes to re-abstraction needs to be lazy; that is, it should create I^{j+1} not when notified that I^j has changed, but rather when the new I^{j+1} is first needed by module M^{j+1}.

- Each module M^j must be able to translate a state from its concrete i-model I^j to its abstract i-model I^{j+1}. There will not always be an abstract state corresponding to every concrete state, but the planning and execution framework guarantees that this translation is only requested when it is possible.

- Each module M^j must provide the proper hooks for the execution framework, which involves expanding actions from I^{j+1} into actions in I^j and storing intermediate execution information. The interface for execution is approximately given by the following pseudocode functions:

```
void    setAbstractAction(Action a);
void    makeAtomicObservation(State s);
boolean isExecutingAbstractAction();
Action  getNextAtomicAction();
```

All existing MDP abstraction and approximation methods that we know of can be made to fit this interface. To solve the gridworld example domain, we package two prior methods into abstraction modules.

4.3.1 Subgoal-options module

The first abstraction discussed in the gridworld example above, having abstract actions that move the robot between the pertinent locations, is similar to the options framework (Precup et al., 1998) and to nearly deterministic abstractions (Lane & Kaelbling, 2002). The idea of the options framework is to create temporally extended actions, in order to speed up value/policy iteration, or in order to create a temporally abstract model that skips past most states by only executing the options (rather than the atomic actions that the options utilize). In this module, the options that it creates are all sub-policies to go from one salient location to another. Applied to the gridworld example, the resulting abstract i-model is a semi-MDP that has 3 locations instead of 100 x-y combinations.

The inputs to a subgoal-options module M^j are

- S_G, a set of goal states, where each goal state $\sigma \in S_G$ specifies values over some subset $\overline{S_{\text{goal}}} \subseteq \overline{S^j}$ of I^j's state variables; and

- $\{a_{\text{goal}}\} \subseteq A^*$, a set of action values, drawn from an action variable $A^* \in \overline{A^j}$ of I^j, that the options are permitted to use when attempting to reach a goal. These actions will be replaced by the options when the abstract model I^{j+1} is created.

In the example, $\overline{S_{\text{goal}}} = \{x, y\}$, $S_G = \{(2,2), (8,2), (5,9)\}$, and $\{a_{\text{goal}}\} = \{north, south, east, west\}$.

This module creates a set O of options, one for each goal $g \in S_G$. An option o_g gives a sub-policy that terminates when the restriction of the current state σ to $\overline{S_{\text{goal}}}$ is g. Each option is built by using policy iteration on a slightly modified version of the domain, where the option's corresponding goal g has absorbing dynamics and a slightly positive pseudo-reward. For each option, the probabilistic expected time transition, state transition, and reward functions (*pett*, *pest*, and *per*, respectively) are calculated for moving from goal state to goal state.

The abstract state and action variable sets that this module creates for I^{j+1} are

- $\overline{S^{j+1}} = \overline{S^j} \setminus \overline{S_{\text{goal}}} \cup \{S_G\}$; and
- $\overline{A^{j+1}} = \overline{A^j} \setminus \{A^*\} \cup \{A'\}$, where A' is an action variable with values $\{a^*\} \setminus \{a_{\text{goal}}\} \cup O$.

In this example, the robot's x and y state variables are replaced with a state variable whose values are the three salient goal locations, and the robot's choices to move in the four compass directions are replaced by actions to move from one goal location to another.

Let $u : \prod_{S \in \overline{S^{j+1}}} S \to \prod_{S \in \overline{S^j}} S$ be a mapping that unpacks the goal part of a state in I^{j+1} into its constituent state variables, giving a state in I^j. In other words, u is the obvious mapping from S_G to $\overline{S_{\text{goal}}}$ extended to be the identity on other state variables. The subgoal-options module maps $I^j \to I^{j+1}$ and defines τ^{j+1}, T^{j+1}, and r^{j+1} in terms of their I^j counterparts as follows:

- $\tau^{j+1}(\sigma^{j+1}, \alpha, n) = \begin{cases} pett(u(\sigma^{j+1}), \alpha, n) & \text{if } \alpha \in O \\ \tau^j(u(\sigma^{j+1}), \alpha, n) & \text{if } \alpha \notin O \end{cases}$
- $t_k^{j+1}(\sigma^{j+1}, \alpha, n, \sigma'^{j+1})$
  $= \begin{cases} pest_k(u(\sigma^{j+1}), \alpha, n, u(\sigma'^{j+1})) & \text{if } \alpha \in O \\ t_k^j(u(\sigma^{j+1}), \alpha, n, u(\sigma'^{j+1})) & \text{if } \alpha \notin O \end{cases}$
- $r^{j+1}(\sigma^{j+1}, \alpha) = \begin{cases} per(u(\sigma^{j+1}), \alpha) & \text{if } \alpha \in O \\ r^j(u(\sigma^{j+1}), \alpha) & \text{if } \alpha \notin O \end{cases}$

4.3.2 State-aggregation module

The second abstraction discussed in the example above, clustering together states that have similar battery levels, is a simple state aggregation. Using the mapping between original and abstract states, transition and reward dynamics for the new states can be formed by taking the average (mean) of the dynamics for the corresponding original states.

The inputs to a state-aggregation module M^j are

- $S_{\text{nonaggr}} \in \overline{S^j}$, the state variable being transformed;

- S_{aggr}, the replacement state variable; and
- $f : S_{\text{nonaggr}} \rightarrow S_{\text{aggr}}$, the aggregation function.

In the gridworld example, S_{nonaggr} = *battery-level*, S_{aggr} = *coarse-battery-level*, and $f(x) = \lceil x/100 \rceil$.

The abstract state and action variable sets that this module creates for I^{j+1} are

- $\overline{S^{j+1}} = \overline{S^j} \backslash \{S_{\text{nonaggr}}\} \cup \{S_{\text{aggr}}\}$
- $\overline{A^{j+1}} = \overline{A^j}$

For any state $\sigma^j \in \prod_{S \in \overline{S^j}} S$, let $f(\sigma^j)$ be the same state but with the value v of S_{nonaggr} in σ^j replaced by $f(v)$. Also, let $c(\sigma^{j+1})$ be the number of states that f maps to σ^{j+1}, i.e., $c(\sigma^{j+1}) = |\{\sigma^j : f(\sigma^j) = \sigma^{j+1}\}|$. The state aggregation module maps $I^j \rightarrow I^{j+1}$ as follows:

- $\tau^{j+1}(\sigma^{j+1}, \alpha, n) = \frac{1}{c(\sigma^{j+1})} \sum_{\sigma^j \in f^{-1}(\sigma^{j+1})} \tau^j(\sigma^j, \alpha, n)$
- $t_k^{j+1}(\sigma^{j+1}, \alpha, n, \sigma'^{j+1}) = \frac{1}{c(\sigma^{j+1})} \sum_{\sigma^j \in f^{-1}(\sigma^{j+1})} \sum_{\sigma'^j \in f^{-1}(\sigma'^{j+1})} t_k^j(\sigma^j, \alpha, n, \sigma'^j)$
- $r^{j+1}(\sigma^{j+1}, \alpha) = \frac{1}{c(\sigma^{j+1})} \sum_{\sigma^j \in f^{-1}(\sigma^{j+1})} r^j(\sigma^j, \alpha)$

In this module, we take the uniform average of the dynamics and reward over the states being aggregated. This is clearly an approximation, since the transition probability distribution at an aggregated state depends very much on the underlying distribution over the states that were aggregated, which in turn depends on the agent's actions to this point. Since we aim to work in huge domains, the large reduction in state space size outweighs the small risk of abstracting out pertinent information. One long-term goal we have is to learn to identify when pertinent information is being abstracted out (in this or other abstraction modules), and to replace such underperforming modules in a module hierarchy.

4.4 Planning and execution

Given a module hierarchy that some domain expert has created, the module hierarchy framework creates various plan or policy pieces and then executes actions while monitoring its execution.

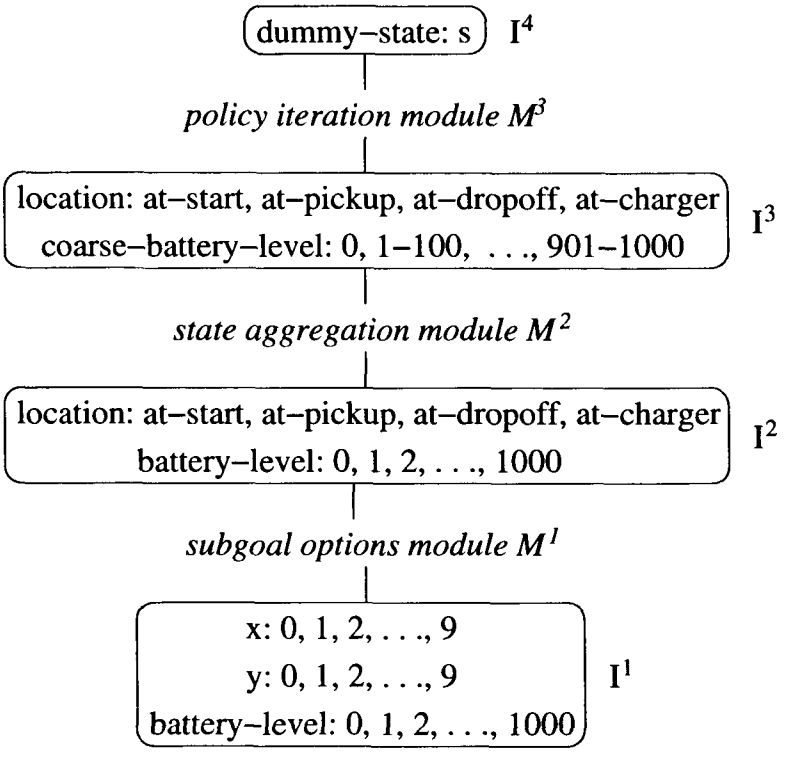

Figure 4.3: A module hierarchy for the gridworld example.

4.4.1 Planning

A module hierarchy for the example gridworld domain is given in figure 4.3. In the figure's i-models, the state variables are listed to demonstrate how the input model is successively abstracted.

The topmost module is always a special module that creates an abstract i-model with only one state and one action. This "abstraction" involves solving the model using policy iteration (Puterman & Shin, 1978), so that the single abstract action is a temporally abstract action meaning "execute the optimal policy that was calculated using policy iteration."

Planning is implicit in the module hierarchy as part of the process of creating the i-models. The i-models are created in order from the bottom to the top, where each module M^j creates I^{j+1} from I^j. Some obvious planning is done by module M^3 as it uses policy iteration to solve the abstract model I^3, but implicit planning also occurs in the subgoal-options module M^1, because creating each option requires creating a plan or policy to get from one location to another. In most module hierarchies, the majority of the planning will happen this latter way, i.e., as part of the process of creating an abstract i-model, rather than in the topmost module.

The overall plan is therefore composed of pieces that are created by and stored in the individual modules. Notice that each piece of planning is done in a much smaller domain than the whole 200,000 state domain: the subgoal-options module creates options on a 1100 state domain, and the policy iteration module creates a policy in an 88 state domain.

An important thing to note is that I^{j+1} is created so that a specified state is representable. When doing initial planning, the state specified to be representable is the starting state. This lets the subgoal-options module, for instance, know that it will have to create a location other than the pickup, drop-off, and charger locations if the initial state is elsewhere.

Another thing to note is that the planning is done in a lazy fashion, where i-model I^{j+1} is created only when it is first needed. This will be important to prevent unnecessary plan change propagation during replanning.

Thus, when receiving a message to represent the world dynamics for a particular starting state, each module M^j follows these steps:

1. Pass the message about representing the world dynamics for a particular state to the child module (if any).
2. Note the particular state to represent.
3. Mark that I^{j+1} is out of date.

When receiving a request for I^{j+1}, each module M^j follows these steps:

1. If I^{j+1} is up to date, return it.

2. Ask M^{j-1} for I^j. If there is no child module, then this module's concrete i-model I^j is simply the input model.

3. Based on I^j, figure out what I^{j+1} should be. What this step actually does is specific to the type of abstraction being performed by M^j.

4. Mark that I^{j+1} is up to date.

5. Return the newly created I^{j+1}.

Since both of these lists of steps begin by asking something of the child module, a request to model a certain starting state or a request for the top i-model will result in a chain of messages being passed from the top to the bottom of the module hierarchy, and then the abstraction work is done in order from the bottom up to the top.

4.4.2 Execution

After the i-models (and the relevant plan pieces) have been created, the module hierarchy begins to execute. This starts by executing the single action in the top i-model.

When an action is executed in any I^{j+1}, module M^j makes observations and chooses concrete actions in I^j to execute until the abstract action from I^{j+1} is done executing. Each time M^j executes an action in I^j, module M^{j-1} makes observations and chooses concrete actions, etc. The actions are translated further and further down the module hierarchy, and eventually they end up in I^1 as atomic actions that can be executed directly in original domain model.

The execution loop consists of two steps: informing all modules of the current state, and then asking the topmost module for the next atomic action to execute. When receiving a request for the next atomic action to execute, each module M^j follows these steps:

1. From M^{j-1}, get the next atomic action to execute.

2. If M^{j-1} returns a `terminated` action,

   (a) Choose the next action in I^j to execute, according to the current action that is executing in I^{j+1}.

   (b) If there is no such action, then the action in I^{j+1} is finished, so return a `terminated` action to M^{j+1}.

   (c) Tell M^{j-1} to execute the action chosen in I^j.

   (d) From M^{j-1}, get the next atomic action to execute.

3. Return the atomic action specified by module M^{j-1}.

Suppose we use the module hierarchy given in figure 4.3 to solve the example gridworld domain, starting as shown in figure 4.1(a). When the single top action is executed, M^3 gains control and executes the optimal policy that it has found (i.e., the top-level action never terminates). The *goto-pickup* action is executed, and control passes to M^2. M^2 passes the action on to M^1, which determines that the current concrete action corresponding to *goto-pickup* is *north*. This is in I^1 and atomic, so the robot takes this action. Suppose this action fails to move the robot; M^1 determines that *north* is again what should be done. This time, the action succeeds, and M^1 determines that *goto-pickup* has terminated. It therefore returns control to M^2, telling it that *goto-pickup* has terminated, and M^2 similarly returns control to M^3. Since *location* is now *at-pickup*, the optimal policy indicates that *pickup* should be executed. Execution continues in a similar manner.

4.4.3 State representation

As part of execution, a module M^j needs to determine the current state in I^j when choosing a new action. The current state is determined by having each module successively translate the observed (atomic) state up from I^1.

Not all atomic states are always representable at all i-models; for instance, only four of the hundred combinations of x and y correspond to values of the *location* state variable. This is not a problem, though, because the only time that the current state needs to be representable in I^j is when a new action is being selected in I^j, and this happens in only two situations. The first is when beginning to execute, and recall that the initial i-models are built so that the initial state is representable. The second is when some action in I^j has just finished, and the current state will necessarily be representable since I^j is assumed to be a valid factored semi-MDP.

4.4.4 Replanning and dynamic representation changes

The module hierarchy so far is a static entity: the decomposition and the modules are chosen, then the framework executes. What makes the module hierarchy different from similar previous methods is that the module hierarchy can change the representation dynamically and update the plan accordingly.

In the gridworld example, the robot can make several deliveries on one charge, and so the robot's battery level isn't important until it gets low. Suppose we insert a new module right below the policy iteration module and have it selectively remove or not remove the *coarse-battery-level* state from the abstract i-model that it creates (see figure 4.4(b)), say, removing *coarse-battery-level* when its value is above *1-100*. Removing *coarse-battery-level* gives the policy iteration module a much smaller model to find a policy for.

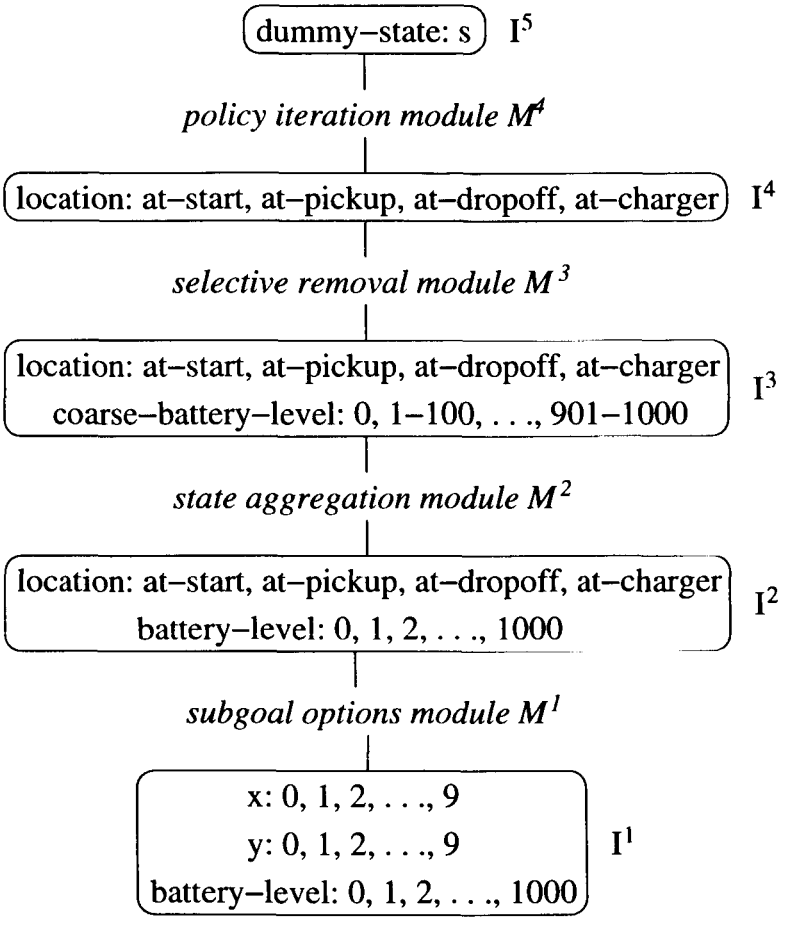

Figure 4.4: The same module hierarchy as in figure 4.3 but with a selective removal module added.

When the battery level gets low enough, we want the selective removal module to notice this and change I^4 to include *coarse-battery-level*, causing the policy iteration module to update the optimal policy it has found to take account of the new *coarse-battery-level* state variable.

In general, when a module M^j changes I^{j+1}, this will cause a cascade of updates up the hierarchy as each module propagates the change by updating its abstract i-model. These updates happen in the same way that initial planning happened: modules are notified that their abstract i-models are out of date, modules are told to create i-models so that the current state is representable, and when the new i-models are requested, each I^{j+1} is created based on I^j.

When a change happens, all currently executing actions are terminated, since they may not be optimal (or may not even exist!) any more. To continue execution after replanning, the top i-model's single action must be executed again, just like the initial execution step.

Note that changes do not have to be propagated down the hierarchy towards I^1, but rather only from the point of the change up to the top of the hierarchy. It could be that the new requirement that the i-models be able to represent the current state is not met by some lower i-models, if they were in the middle of executing a temporally abstract action that was terminated. To ensure that this does not happen, we insist that each module M^j may only make changes to I^{j+1} when an action has just finished executing in I^j.

As each module M^j updates I^{j+1} to take account of changes, it could recalculate I^{j+1} from scratch, but in many cases, it can reuse most of the solution from the previous version of I^{j+1}. This causes the change of representation to happen much faster than the creation of the initial representation. For instance, suppose the robot is periodically instructed to change its drop-off location to one of nine other possible sites. Instead of representing all possible drop-off locations and having nine of them be useless, the drop-off location can be approximated as fixed in one place. When its location changes, the options module only has to create a new *goto-dropoff* option; it can reuse the options to reach other locations. Even better, changes to *location* cause the state aggregation module no new work, since it just copies information about *location* from I^2 to I^3. Reabstraction can therefore occur fairly frequently in this framework without being a burden. For large domains, this ability to keep the current representation small will likely mean the difference between tractability and intractability.

The ability to adapt the representation dynamically can be used in other ways as well. For instance, if more processing power is suddenly available, it may be advantageous to reduce the amount of approximating, in the hopes of getting a better policy. Or, if a better atomic model of the domain's dynamics becomes available (say, because it is being learned online), then that better model can replace the old model without needing to plan from scratch.

4.5 Experiments

In order to test the module hierarchy, we used a simplified version of the computer game `nethack` (http://www.nethack.org). Nethack is a good domain for testing different approaches to solving real-world problems because it contains several different types of structure, some simple and some complex. The varying structure and the interaction between the different parts is representative of even larger, real-world domains, such a disaster relief robot, a Mars rover, or a general-purpose battlefield robot.

In the simplified version of nethack that we used, the goal is to escape from a dungeon. The dungeon is composed of several levels, where each level consists of some large rooms connected by narrow hallways. The levels are connected by stairways to the levels right above and right below them, and the escape stairway is at the top. The player can move north, south, east, west, up, and down, but not diagonally.

The game is not just a path planning problem, because the player has hunger and health. The player gets progressively more hungry as time goes on; if he starves, his health decreases, but there is food available to eat lying around the dungeon, and the player can carry this food with him. The player's health normally stays constant, but it decreases when starving or when attacked by a monster. The player can heal himself by using one of the medkits lying around the dungeon, and the player can carry medkits with him.

We represented the domain as an infinite-horizon discounted factored MDP with 11 primitive actions and a varying number of states (depending on the exact layout of the dungeon). Some of the actions, such as movement and attacking a monster, had probabilistic outcomes (e.g., the monster dies with a certain probability). The reward was set to be positive for escaping from the dungeon, negative for dying, and zero elsewhere.

4.5.1 Implemented module hierarchy

The module hierarchy that was created to solve the nethack domain is shown in figure 4.5. Even though the module hierarchy is a linear alternation of i-models and modules, modules are drawn showing which part of the domain they change, in order to better and more compactly illuminate the structure of the changes that each abstraction module makes.

This module hierarchy uses eighteen modules instantiated from six module types. These modules were arranged in the module hierarchy and parameters were supplied by a domain expert, who tailored the structure and parameters so as to solve the simplified nethack domain as well as possible. It is important to note that, although the above module types were created in order to solve this simplified nethack domain, they are completely general and can be used in other module hierarchies to solve other problems, given appropriate parameter choices.

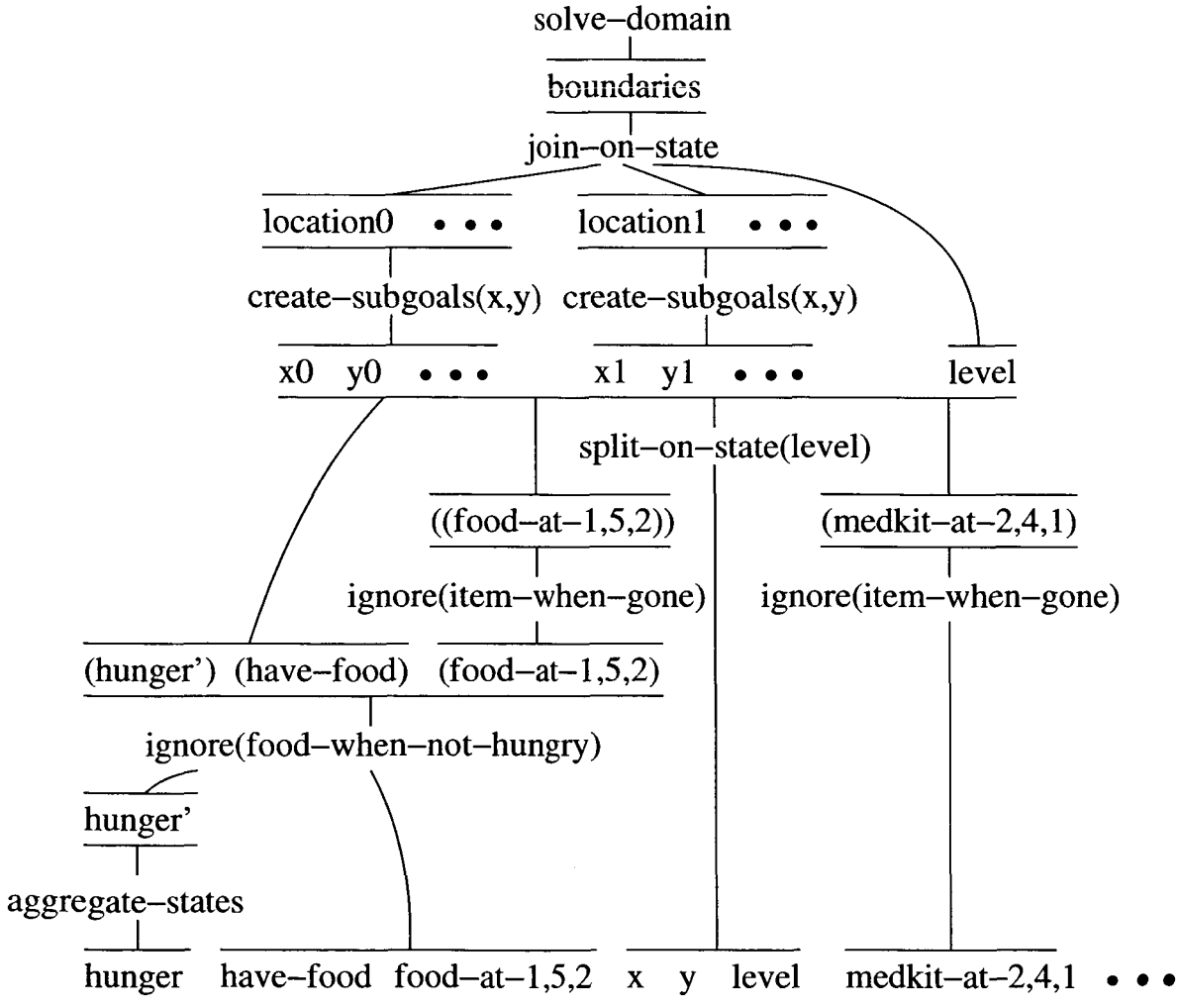

Figure 4.5: The module hierarchy used to solve the simplified nethack domain.

Of the six module types used, four are the modules used in the package and charger gridworld example above (subgoal-options, state-aggregation, policy-iteration, and selective-removal). The remaining two modules allow subparts of the state space to be solved separately and recombined, and there is an extension to the subgoal-options module that allows it to fill in some of its parameters more automatically.

4.5.2 Split-on-state and join-on-state modules

These two modules allow subparts of the state space to be solved separately and recombined, in a way similar to the macro-action framework (Hauskrecht, Meuleau, Kaelbling, Dean, & Boutilier, 1998).

Split-on-state module

The split-on-state module divides up the state space into subparts. Each state is assigned to a subpart based on the state value that it contains for a particular *indicator* state variable. In the nethack domain, for instance, the dungeon was divided up based on the *floor* state variable, so that an escape path could be found separately through each floor. Copies of all the rest of the state variables are created for each subpart, as well as copies of all the action variables.

The transition dynamics for each subpart are simply the original dynamics but with the indicator state variable fixed, if the active action variable is from that subpart. If it the active action variable is from another subpart, then the state for this subpart does not change no matter what action is chosen. The abstract reward function is based on what happens in the active subpart only.

Since there will be other modules between the split-on-state and join-on-state modules that will want to abstract each subpart separately, it would be good to have each subpart be as independent as possible. Unfortunately, because some actions can cause the state to switch into other subparts, the subparts will have to be linked somehow (i.e., transition probability distributions for state variables in a subpart will be dependent on some state or action variables not in that subpart). We make this interdependence as small as possible by creating a new action variable A_{switch} to encapsulate all dynamics that switch subparts, removing such dynamics from each subpart's normal action variable.

A_{switch} contains action values for each combination of state and action that can cause the subpart to switch, and those action values execute the switching action when at states where the subpart could switch but execute a no-op elsewhere. For instance, in the nethack domain, the player can go from floor to floor using stairways. So, A_{switch} contains an action value meaning "go up this stairway" for each stairway; this action value causes the player to ascend when at the bottom of the stairway and to stay put elsewhere. In the split action variable corresponding to the floor at the

bottom of the stairway, the corresponding dynamics (i.e., the result of taking an *up* action at the bottom of the stairway) are changed to staying put. By moving all switching behavior to A_{switch}, each subpart's dynamics are virtually independent, which is a huge boon for modules operating between the split-on-state and join-on-state modules.

The single input to a split-on-state module M^j is $S_{\text{split}} \in \overline{S^j}$, the state variable that indicates the current subpart. In the nethack domain, $S_{\text{split}} =$ *floor*.

The abstract state and action variable sets that the split-on-state module creates for I^{j+1} are

- $\overline{S^{j+1}} = \{S_{\text{split}}\} \cup \bigcup_{subpart \in S_{\text{split}}} \bigcup_{S \in \overline{S^j}} S_{subpart}$
- $\overline{A^{j+1}} = \{A_{\text{switch}}\} \cup \bigcup_{subpart \in S_{\text{split}}} \bigcup_{A \in \overline{A^j}} A_{subpart}$, where A_{switch} is as described above.

For any action value $\alpha^{j+1} \in \bigcup_{A \in \overline{A^{j+1}} \setminus A_{\text{switch}}} A$, let $sp(\alpha^{j+1}) \in S_{\text{split}}$ be the subpart that the action value is drawn from. For any state $\sigma^{j+1} \in \prod_{S \in \overline{S^{j+1}}} S$, let $\lfloor \sigma^{j+1} \rfloor$ be the corresponding state in I^j, where the subpart of σ^{j+1} that is mapped into $\overline{S^j}$ is given by the value of S_{split} in σ^{j+1}. Also let $\lfloor \sigma^{j+1} \rfloor_{s_{\text{sp}}}$ be the same except that the selected subpart of σ^{j+1} is given by $s_{\text{sp}} \in S_{\text{split}}$.

For action values $\alpha \in A_{\text{switch}}$, let $aps(\sigma^j, \alpha) \in \{\text{true}, \text{false}\}$ indicate whether σ^j is an appropriate pre-state for α, and let $u(\alpha)$ be the corresponding normal action value that is executed when α is at an appropriate pre-state.

The split-on-state module maps $I^j \rightarrow I^{j+1}$ as follows:

- $\tau^{j+1}(\sigma^{j+1}, \alpha, n) =$
$$\begin{cases} \tau^j(\lfloor \sigma^{j+1} \rfloor_{sp(u(\alpha))}, u(\alpha), n) & \text{if } \alpha \in A_{\text{switch}} \\ \tau^j(\lfloor \sigma^{j+1} \rfloor_{sp(\alpha)}, \alpha, n) & \text{if } \alpha \notin A_{\text{switch}} \end{cases}$$
- For each subpart s_{sp}, $t^{j+1}_{k,s_{\text{sp}}}(\sigma^{j+1}, \alpha, n, \sigma'^{j+1}) =$
$$\begin{cases} \text{if } \alpha \in A_{\text{switch}}: \\ \quad t^j_k(\lfloor \sigma^{j+1} \rfloor_{sp(u(\alpha))}, u(\alpha), n, \lfloor \sigma'^{j+1} \rfloor) \\ \qquad\qquad \text{if } aps(\lfloor \sigma^{j+1} \rfloor_{sp(u(\alpha))}, \alpha) \\ \quad 1.0 \quad \text{if } \neg aps(\lfloor \sigma^{j+1} \rfloor_{sp(u(\alpha))}, \alpha) \text{ and } \lfloor \sigma^{j+1} \rfloor = \lfloor \sigma'^{j+1} \rfloor \\ \quad 0.0 \quad \text{if } \neg aps(\lfloor \sigma^{j+1} \rfloor_{sp(u(\alpha))}, \alpha) \text{ and } \lfloor \sigma^{j+1} \rfloor \neq \lfloor \sigma'^{j+1} \rfloor \\ \text{else}: \\ \quad t^j_k(\lfloor \sigma^{j+1} \rfloor_{sp(\alpha)}, \alpha, n, \lfloor \sigma'^{j+1} \rfloor_{sp(\alpha)}) \quad \text{if } sp(\alpha) = s_{\text{sp}} \\ \quad 1.0 \quad \text{if } sp(\alpha) \neq s_{\text{sp}} \text{ and } \lfloor \sigma^j \rfloor_{S^j_k} = \lfloor \sigma^{j+1} \rfloor_{S^j_k} \\ \quad 0.0 \quad \text{if } sp(\alpha) \neq s_{\text{sp}} \text{ and } \lfloor \sigma^j \rfloor_{S^j_k} \neq \lfloor \sigma^{j+1} \rfloor_{S^j_k} \end{cases}$$
(Recall that t^j_k gives the post-state for the state variable S^j_k from I^j.)

- $r^{j+1}(\sigma^{j+1}, \alpha) =$
$\begin{cases} r^j(\lfloor \sigma^{j+1} \rfloor_{sp(u(\alpha))}, u(\alpha)) & \text{if } \alpha \in A_{\text{switch}} \\ r^j(\lfloor \sigma^{j+1} \rfloor_{sp(\alpha)}, \alpha) & \text{if } \alpha \notin A_{\text{switch}} \end{cases}$

Join-on-state module

The join-on-state module merges back together the subparts that the split-on-state module created. As with the macro-action framework, the abstract states created by this module are the boundary states, where it is possible to go from one subpart to another, and the abstract actions are subpolicies to travel from one boundary state to another.

Even though the join-on-state module simply undoes the partitioning of the split-on-state module, it cannot get away with only using the same parameters as split-on-state module. This is because there may have been other modules, in between the split- and join-on-state modules, that reworked various parts of the state and action space until it is not recognizable as belonging to a particular subpart. So, the join-on-state module needs parameters that tell it what state and action variables correspond to which subpart.

The inputs to a join-on-state module M^j are

- S_{split}, the state variable that indicates the current subpart;
- A_{switch}, the action variable that switches subparts;
- $sp_s : \overline{S^j} \rightarrow S_{\text{split}}$, a mapping that indicates which subpart each state variable belongs to (if any; S_{split} doesn't belong to a subpart); and
- $sp_a : \overline{A^j} \rightarrow S_{\text{split}}$, a mapping that indicates which subpart each action variable belongs to.

These inputs are used in the construction of the abstract state and action spaces. The abstract state and action variable sets that the join-on-state module creates for I^{j+1} are

- $\overline{S^{j+1}} = \{S_{\text{bdry}}\}$, where S_{bdry} is a state variable whose values are all states that can be reached by taking an action in A_{switch} and attempting to go from one subpart to another. (More precisely, the state values are the restriction of such states to the post-subpart, along with the value of S_{split}.)
- $\overline{A^{j+1}} = \{A_{\text{gotobdry}}\}$, where A_{gotobdry} is an action variable whose values consist of all subpolicies a_{gotobdry} of the following form: given the current subpart and some $a_{\text{switch}} \in A_{\text{switch}}$ that switches from this subpart to another, attempt to go and execute a_{switch} in an optimal way. These policies are built in the same was as the options in the subgoal-options module, by running policy iteration on a domain with a slightly positive reward at the goal. There are

roughly two parts to each policy; the first part attempts to reach a state where a_{switch} could be effective at switching to a different subpart, and the second part executes a_{switch} once. The optimality requirement is with respect to reward gathered along the way.

As with the subgoal-options module, we define the probabilistic expected time transition, state transition, and reward functions (*pett*, *pest*, and *per*, respectively) for each $a_{\text{switch}} \in A_{\text{switch}}$. Similarly, we let $u : S_{\text{bdry}} \rightarrow \prod_{S \in \overline{S^j}} S$ be a mapping that unpacks the state in I^{j+1} into S_{split} and the appropriate subpart's state variables in I^j, filling in the rest of $\overline{S^j}$ randomly.

The join-on-state module maps $I^j \rightarrow I^{j+1}$ as follows:

- $\tau^{j+1}(\sigma^{j+1}, \alpha, n) = pett(u(\sigma^{j+1}), \alpha, n)$
- $t^{j+1}_{\text{bdry}}(\sigma^{j+1}, \alpha, n, \sigma'^{j+1}) =$
  $pest_{\text{bdry}}(u(\sigma^{j+1}), \alpha, n, u(\sigma'^{j+1}))$
- $r^{j+1}(\sigma^{j+1}, \alpha) = per(u(\sigma^{j+1}), \alpha)$

Intervening modules

The point of breaking the macro-action framework into two modules is that there can be other modules in between the division into subparts and the creation of the boundary state semi-MDP. Intervening modules can manipulate, abstract, approximate, and solve different subparts separately, before they get merged back together.

Each intervening module should be careful only modify state and action variables corresponding to a single subpart. They should never modify values of the state variable S_{split} or the action variable A_{switch}, and they should only modify the transition/reward dynamics associated with these in order to reflect the changes that happened in some subpart's state space. For instance, consider the case of using a subgoal-options module M^j to abstract a particular level l of the nethack domain, in between the split-on-state and join-on-state modules. The dynamics of the action values in A^j_{switch} that switch into and out of level l refer to x-y coordinates, but the dynamics of comparable A^{j+1}_{switch} action values need to refer to *locations*.

4.5.3 Auto-subgoal-options module

This module is an extension of the subgoal-options module that automatically determines what the subgoals should be. It takes a set of state variables to find goals in, and it creates goals wherever the transition or reward dynamics are "interesting."

The inputs to an auto-subgoal-options module M^j are

- $\overline{S_{\text{subsume}}} \subseteq \overline{S^j}$, a set of state variables to create goals in and then subsume; and
- $\{a_{\text{goal}}\} \subseteq A^*$, a set of action values drawn from an action variable $A^* \in \overline{A^j}$ of I^j; this is the same set of action values as is given as input to a subgoal-options module.

The states of $\overline{S_{\text{subsume}}}$ that are added as goals are the ones where transition and reward dynamics are mostly the same but are different at a few states. The algorithm to find subgoals is as follows:

1. Vary the transition distribution t_k over all transition distributions for non-subsumed state variables $\overline{S^j} \backslash \overline{S_{\text{subsume}}}$. Recall that $\overline{S_k}$ is the set of state variables that are relevant to the transition.
2. Vary the action a over all non-subsumed action values $\left[\bigcup_{A \in \overline{A^j}} A\right] \backslash \{a_{\text{goal}}\}$.
3. Vary the partial state s_{sub} over the relevant subsumed state variables $\overline{S_{\text{subsume}}} \cap \overline{S_k}$.
4. For each combination of t_k, a, and s_{sub},
   (a) Vary the partial state s_{nonsub} over the relevant non-subsumed state variables.
   (b) Notice how the conditional probability distribution over post-states, given by $cpd(n, s') = t_k(s_{\text{sub}} \cup s_{\text{nonsub}}, a, n, s')$, varies as s_{nonsub} is changed.
   (c) If there is more than one cpd as s_{nonsub} is varied, and if one is more than twice as frequent as all the rest, then add all non-zero state outcomes in the infrequent cpds as goals.

The algorithm to find interesting combinations based on reward is very similar. As an example, this module would note that, for most x-y coordinates, when a player chooses the *up* action the *level* state variable doesn't change. When the player happens to be standing at an up staircase, however, the level changes. This indicates that the stairway's x-y coordinates should be thought of as an interesting subgoal.

This process seems as though it could potentially iterate over many transition functions, partial states and action values, and indeed it would if the representation of the t_k were flat. However, since the t_k are given by ADDs, this algorithm takes time proportional to the size of the ADD rather than the number of state and action variables, and this produces substantial savings in execution time.

4.5.4 Experimental results

The running time and solution quality of the module hierarchy were compared both to policy iteration on the original domain and to several abstraction methods used in its modules, operating

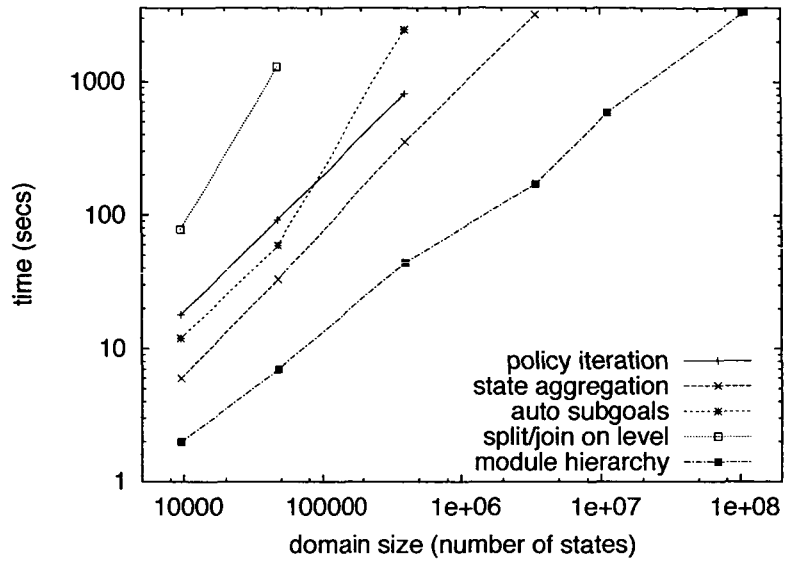

Figure 4.6: Experimental results.

individually. Each method was run on a sequence of progressively more complicated instances of the nethack domain until it failed to escape from the dungeon within one hour. In successive domains, the number of items in the domain, the number of levels, and the x-y size of each level were gradually increased, so that the state space size ranged from 9,600 to 108,900,000. The number of primitive action steps required to escape from the dungeon increased sub-logarithmically with the size of the domain.

The running time results are given in figure 4.6. As expected, the module hierarchy is the only method that scales up to problems with very large numbers of states.

The most important point of comparison between the different methods is the size of the domains that each works with after having applied pertinent abstractions. The previous methods end up attempting to work in models with hundreds of thousands of states by the fourth or fifth test domain. In contrast, the largest model that the module hierarchy needs to solve has just 450 states and 15 actions. Granted, quite a few 450-state domains are solved during the course of execution, but this is certainly preferable to intractability.

4.5.5 Optimality

In the test runs described above, wherever two methods managed to produce a solution for the same test domain, the reward gained by those methods was the same (within the margin of error caused by the stochasticity of the domain and thus needing to average over several trial runs). In other words, all methods that succeeded in escaping from the dungeon in one CPU hour did so in approximately the same amount of time.

Of course, that no reward was lost by approximately solving the domains is due entirely to an appropriate choice of modules and their parameters. If, for instance, the selective-removal module that operates on the hunger portion of the domain were to have its threshold parameter set too low, then the player might die from hunger before being able to reach food. But an appropriate choice of modules is not unreasonable to assume, because any approximation method is dependent on the quality of its approximation. In addition, even if the approximation turns out to be wildly sub-optimal, a system that makes a very large domain tractable is still be better than one that can't handle the domain at all.

What would be ideal is for each module to come with a bound on the possible reward loss when using it, and we can derive a loose such bound, for instance, using value functions. Suppose we consider solving the given model $\mathcal{M}$ by first solving an approximate model $\hat{\mathcal{M}}$ and then using its optimal policy, $\pi^\dagger$, instead of the optimal policy π^*, in the original model. The triangle inequality gives us a bound on the loss,

$$L^{\pi^\dagger}(s) \leq \left|V^{\pi^*}(s) - \hat{V}^{\pi^\dagger}(s)\right| + \left|\hat{V}^{\pi^\dagger}(s) - V^{\pi^\dagger}(s)\right|,$$

where V^{π^*} gives the optimal values in $\mathcal{M}$, $\hat{V}^{\pi^\dagger}$ gives the optimal values in $\hat{\mathcal{M}}$, and $V^{\pi^\dagger}(s)$ gives the values when using $\pi^\dagger$ in $\mathcal{M}$.

Unfortunately, only loose loss bounds like the one above can be derived in general, if we wish to allow modules powerful enough to drastically reduce the complexity of the original domain. Individual estimates of reward loss might be possible for different modules, and this is an ongoing area of our research.

4.6 Comparison with prior work

There are several previous approaches to hierarchical planning under uncertainty that are similar to module hierarchies, but most use a single abstraction method and exploit only one kind of domain structure. MAXQ (Dietterich, 1998), for instance, uses parameterized tasks arranged in a hierarchy to constrain the policy that it searches for, and it works very well on domains that

decompose hierarchically into subtasks, but it does not support arbitrary other abstractions like state aggregation.

The most similar previous work is the hierarchies of abstract machines (HAMs) framework (Parr & Russell, 2000). This method allows for combining multiple abstraction types represented as non-deterministic finite state machines. The module hierarchy framework has two advantages over this methods.

The first advantage is the representation of the domain MDP. The factoring of state and action spaces into variables, and the representation of functions as ADDs, allows for a large amount of structure to be stored explicitly with each i-model. This structure allows abstraction modules to operate on large chunks of state and action space at once; thus, a simple abstraction such as state approximation can operate on a domain in time proportional to the domain complexity rather than the domain size.

A lot of the benefit from the structured representation actually comes automatically from using ADDs to represent functions; for instance, ADDs never have redundant choices, so a function's independence of certain state variables manifests itself in the ADD structure with no special processing. Such independence is noticed and exploited in the split-on-state and join-on-state abstractors, for instance, when they determine that certain state variables are irrelevant to the workings of particular subparts and therefore do not need to be added to that subpart's state space.

The second and larger advantage of the module hierarchy comes in the ability to change the representation dynamically, at any time, and update the plan accordingly. This allows for partial plans to be created and then amended as needed. As long as the current representation is accurate enough to show the gist of far-future dynamics, the details can wait until the present situation has been dealt with and currently useful information is no longer relevant.

(Not planning the far future in detail has the added bonus that no work is wasted if the far future turns out differently than currently expected.)

As an example, in nethack, a player rarely has to worry about simultaneous imminent death from starvation and imminent death from monsters. By removing and then selectively adding back these aspects of the domain only when necessary, it is possible to deal with them separately. This effect is even more pronounced in larger, real-world domains, which have more areas of knowledge that are fairly specialized and thus do not interact much; such areas of knowledge produce a combinatorial explosion in the size of the state and action spaces unless dealt with separately.

Unfortunately, in order to gain benefits over these previous methods, it is necessary to give up any hope at optimality. Each of the previous methods mentioned guarantees optimality, or perhaps a limited version such as hierarchical or recursive optimality, if certain criteria are met. Similar criteria could be formulated for the module hierarchy framework; for instance, if all modules have a bound on the reward loss from using the module, then those bounds could be summed or multiplied

to give an overall reward loss bound. Such hard criteria are unlikely to be found for the most useful of modules, however.

Our usage of ADDs to represent probability distributions and reward functions is reminiscent of SPUDD (Hoey, St-Aubin, Hu, & Boutilier, 1999) and APRICODD (St-Aubin, Hoey, & Boutilier, 2000). These algorithms attempt to improve value iteration by exploiting the domain structure exposed by ADDs, which is an orthogonal approach to that of the module hierarchy framework. Though we could have used SPUDD and APRICODD algorithms when, say, creating options in the subgoal-options module, the sub-domains being solved were small enough that this would have brought no significant improvement.

4.7 Conclusions and future work

The module hierarchy trades optimality and high speed on small domains for tractability on large domains. Although the module hierarchy makes no guarantees about optimality, the nethack domain results show that it may not be necessary to sacrifice much optimality in order to gain tractability. Tractability is gained because modules can be chosen to exploit the specific structure of different parts of the domain, and because those modules have the ability to reabstract dynamically, changing the representation to focus domain solving on the (small) currently relevant portions of state space.

The execution times show that the module hierarchy can handle larger domains than any single static abstraction method; even so, there is room for considerable improvement. For instance, ways to improve on this first module hierarchy system include monitoring abstract action execution and interrupting when low probability occurrences can be exploited, and extension to other models like POMDPs and relational MDPs.

Bibliography

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

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. thesis, Stanford University.

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

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

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

Blum, A. L., & Furst, M. L. (1997). Fast plannning through planning graph analysis. *Artificial Intelligence, 90,* 281–300.

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

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

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

Dean, T., Kaelbling, L. P., Kirman, J., & Nicholson, A. (1995). Planning under time constraints in stochastic domains. *Artificial Intelligence, 76.*

Dietterich, T. (1998). The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning.*

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

Edelkamp, S., & 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.*

Ellman, T. (1993). Abstraction via approximate symmetry. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence.*

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

Fox, M., & Long, D. (1999). The detection and exploitation of symmetry in planning problems. In *16th International Joint Conference on Artificial Intelligence.*

Fox, M., & Long, D. (2002). Extending the exploitation of symmetries in planning. In *AIPS.*

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

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

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

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.*

Hauskrecht, M., Meuleau, N., Kaelbling, L., Dean, T., & Boutilier, C. (1998). Hierarchical solution of markov decision processes using macro-actions. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence.*

Hoey, J., St-Aubin, R., Hu, A., & Boutilier, C. (1999). Spudd: Stochastic planning using decision diagrams. In *Fifteenth Conference on Uncertainty in Artificial Intelligence.*

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

Lane, T., & Kaelbling, L. (2002). Nearly deterministic abstractions of markov decision processes. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence.*

Lane, T., & Kaelbling, L. P. (2001a). Approaches to macro decompositions of large markov decision process planning problems. In *Proceedings of the 2001 SPIE Conference on Mobile Robotics.*

Lane, T., & Kaelbling, L. P. (2001b). Toward hierarchical decomposition for planning in uncertain environments. In *Proceedings of the 2001 IJCAI Workshop on Planning under Uncertainty and Incomplete Information.*

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 *AAAI.*

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

Parr, R., & Russell, S. (2000). Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing 10.*

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

Penberthy, J. S., & Weld, D. (1992). UCPOP: A sound, complete partial-order planner for adl. In *KR.*

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

Precup, D., Sutton, R., & Singh, S. (1998). Theoretical results on reinforcement learning with temporally abstract options. In *In Proceedings of the Tenth European Conference on Machine Learning.*

Puterman, M. (1994). *Markov decision processes: discrete stochastic dynamic programming.* John Wiley & Sons.

Puterman, M., & Shin, M. (1978). Modified policy iteration algorithms for discounted markov decision problems. *Management Science, 24(11).*

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

Rohanimanesh, K., & Mahadevan, S. (2001). Decision theoretic planning with concurrent temporally extended actions. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence.*

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.*

St-Aubin, R., Hoey, J., & Boutilier, C. (2000). Apricodd: Approximate policy construction using decision diagrams. In *Advances in Neural Information Processing 13.*

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*.

Younes, H., & Littman, M. (2003). PPDDL1.0: An extension to pddl for expressing planning domains with probabilistic effects. In *In Proceedings of the 14th International Conference on Automated Planning and Scheduling*.

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