Learning Partially Observable Action Models: Efficient Algorithms

Dafna Shahaf Allen Chang Eyal Amir
Computer Science Department
University of Illinois, Urbana-Champaign
Urbana, IL 61801, USA
{dshahaf2,achang6,eyal}@uiuc.edu

Abstract

We present tractable, exact algorithms for learning actions' effects and preconditions in partially observable domains. Our algorithms maintain a propositional logical representation of the set of possible action models after each observation and action execution. The algorithms perform exact learning of preconditions and effects in any deterministic action domain. This includes STRIPS actions and actions with conditional effects. In contrast, previous algorithms rely on approximations to achieve tractability, and do not supply approximation guarantees. Our algorithms take time and space that are polynomial in the number of domain features, and can maintain a representation that stays compact indefinitely. Our experimental results show that we can learn efficiently and practically in domains that contain over 1000's of features (more than 2^{1000} states).

1 Introduction

Complex, real-world environments afford only partial observations and limited apriori information about their dynamics. Acting in such domains is difficult for that reason, and it is very important to try to acquire as much information about the environment dynamics as possible. Most current planning approaches assume that the action model is specified fully in a formalism such as STRIPS (Fikes & Nilsson 1971) or PDDL (Ghallab *et al.* 1998). This holds even for algorithms for planning in partially observable domains (Cimatti & Roveri 2000; Bertoli *et al.* 2001).

We present efficient algorithms for learning general deterministic action models in partially observable domains. Unlike previous approaches, our algorithms do not approximate the learned model. Our algorithms take as input a propositional logical formula that describes partial knowledge about the initial state of the world and the action model. The algorithms also take as input a sequence of actions that were taken and observations that were received by an agent acting in the world. The algorithms work by computing the set of all hypothetical action models that are consistent with the actions taken and observations received.

There have been previous approaches that attempt to learn deterministic action models automatically, but only recently have approaches which learn action models in the presence of partial observability appeared. Of these previous approaches, our work is closest to (Amir 2005). In this previous work an algorithm tracks a logical representation that encodes the set of possible action models. The approach assumes that actions never fail (in which case, no preconditions are learned), or that actions map states 1:1 (in which case, the size of the logical representation may grow exponentially with the number of steps). In addition, the approach cannot learn preconditions of actions efficiently.

We relax these assumptions using several technical advances, and provide algorithms that are *exact* and tractable (linear in time steps and domain size) for all deterministic domains. Our algorithms maintain an alternative logical representation of the partially known action model. They present two significant technical insights and advances over (Amir 2005):

Our most general algorithm (Section 3) represents logical formulas as directed acyclic graphs (DAGs, versus "flat" formulas) in which leaves are propositional symbols and internal nodes are logical connectives. This way, subformulas may be re-used or shared among parent nodes multiple times. This data structure prevents exponential growth of the belief state, guarantees that the representation always uses a fixed number of propositional symbol nodes, and guarantees that the number of connective nodes grows only polynomially with the domain size and number of time steps (compare this representation with Binary Decision Diagrams (BDDs), which can explode exponentially in size).

We also present efficient algorithms that represent formulas in conjunctive normal form (CNF) (Section 4). These algorithms consider tractable subcases for which we can polynomially bound the size of the resulting CNF formulas. They are applicable to non-failing STRIPS actions and possibly failing STRIPS actions for which action preconditions are known. That is, they trade off some expressivity handled by the first algorithm in order to maintain the belief state compactly in a CNF representation. Such representations are advantageous because they can be readily used in conjunction with modern inference tools, which generally expect input formulas to be in CNF.

Related Work A number of previous approaches to learning action models automatically have been studied in addition to aforementioned work (Amir 2005). Approaches such

Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

as (Wang 1995; Gil 1994; Pasula, Zettlemoyer, & Kaelbling 2004) are successful for fully observable domains, but do not handle partial observability. In partially observable domains, the state of the world is not fully known, so assigning effects and preconditions to actions becomes more complicated.

One previous approach in addition to (Amir 2005) that handles partial observability is (Qiang Yang & Jiang 2005). In this approach, example plan traces are encoded as a weighted maximum satisfiability problem, from which a candidate STRIPS action model is extracted. A data-mining style algorithm is used in order to examine only a subset of the data given to the learner, so the approach is approximate by nature.

Hidden Markov Models can be used to estimate a *stochastic* transition model from observations. However, the more complex nature of the problem prevents scaling up to large domains. These approaches represent the state transition matrix explicitly, and can only handle relatively small state spaces. Likewise, structure learning approaches for Dynamic Bayesian Networks are limited to small domains (e.g., 10 features (Ghahramani & Jordan 1997; Boyen, Friedman, & Koller 1999)) or apply multiple levels of approximation. Importantly, DBN based approaches have unbounded error in deterministic settings. In contrast, we take advantage of the determinism in our domain, and can handle significantly larger domains containing over 1000 features (i.e., approximately 2^{1000} states).

2 A Transition Learning Problem

In this section we describe the combined problem of learning the transition model and tracking the world. First, we give an example of the problem; we will return to this example after introducing some formal machinery.

EXAMPLE Consider a Briefcase domain, consisting of several rooms and objects. The agent can lock and unlock his briefcase, put objects inside it, get them out and carry the briefcase from room to room. The objects in the briefcase move with it, but the agent does not know it.

The agent performs a sequence of actions, namely putting *book1* in the briefcase and taking it to *room3*. His goal is to determine the effects of these actions (to the extent he can, theoretically), while also tracking the world.

We now define the problem formally (borrowed from (Amir 2005)).

Definition 2.1 *A **transition system** is a tuple $\langle P,S,A,R \rangle$*

- *P is a finite set of fluents.*
- *$S \subseteq$ Pow(P) is the set of world states; a state $s \in S$ is the subset of P containing exactly the fluents true in s.*
- *A is a finite set of actions.*
- *$R \subseteq S \times A \times S$ is the (deterministic) transition relation.*

$\langle s, a, s' \rangle \in R$ means that state s' is the result of performing action a in state s.

Our Briefcase world has fluents of the form *locked, is-at(room), at(object,room), in-BC(object),* and actions such as *putIn(object)*, *getOut(object)*, *changeRoom(from,to)*, *press-BC-Lock*.

This domain includes actions with conditional effects: Pressing the lock causes the briefcase to lock if it was unlocked, and vice versa. Also, if the agent is in room1, the result of performing *changeRoom(room1,room3)* is always *is-at(room3)*. Other outcomes are conditional– if *in-BC(apple)* holds, then the apple will move to room3 as well. Otherwise, its location will not change. This is *not* a STRIPS domain.

Our agent cannot observe the state of the world completely, and he does not know how his actions change it; in order to solve it, he can maintain a set of possible world states and transition relations that might govern the world.

Definition 2.2 *A **transition belief state** Let $\mathcal{R}$ be the set of all possible transition relations on S, A. Every $\rho \subseteq S \times \mathcal{R}$ is a transition belief state.*

Informally, a transition belief state ρ is the set of pairs $\langle s, R \rangle$ that the agent considers possible. The agent updates his belief state as he performs actions and receives observations. We now define semantics for Simultaneous Learning and Filtering (tracking the world state).

Definition 2.3 (Simultaneous Learning and Filtering) *$\rho \subseteq S \times \mathcal{R}$ a transition belief state, a_i are actions. We assume that observations o_i are logical sentences over P.*

1. *SLAF$[\epsilon](\rho) = \rho$ (ϵ: an empty sequence)*
2. *SLAF$[a](\rho) = \{\langle s', R \rangle \mid \langle s, a, s' \rangle \in R, \langle s, R \rangle \in \rho\}$*
3. *SLAF$[o](\rho) = \{\langle s, R \rangle \in \rho \mid o$ is true in $s\}$*
4. *SLAF$[\langle a_j, o_j \rangle_{i \le j \le t}](\rho) =$ SLAF$[\langle a_j, o_j \rangle_{i < j \le t}]$ (SLAF$[o_i]$(SLAF$[a_i](\rho)$))*

We call step 2 progression with a and step 3 filtering with o. In short, we maintain a set of pairs that we consider possible; the intuition behind this definition is that every pair $\langle s', R \rangle$ becomes a new pair $\langle \tilde{s}, R \rangle$ as the result of an action. If an observation discards a state $\tilde{s}$, then all pairs involving $\tilde{s}$ are removed from the set. We conclude that R is not possible when all pairs including it have been removed.

EXAMPLE (CONT.) Our agent put *book1* in the briefcase and took it to *room3*. Assume that one of the pairs in his belief state is $\langle s, R \rangle$: s is the actual state before the actions, and R is a transition relation which assumes that both actions do not affect *at(book1,room3)*. That is, after progressing with both actions, *at(book1,room3)* should not change, according to R. When receiving the observation *¬at(book1,room3)*, the agent will eliminate this pair from his belief state.

3 Update of Possible Transition Models

In this section, we present an algorithm for updating the belief state. The naïve approach (enumerating) is clearly intractable; we apply logic to keep the representation compact and the learning tractable. Later on, we show how to solve *SLAF* as a logical inference problem.

We define a vocabulary of *action propositions* which can be used to represent transition relations as propositional formulas. Let $L^0 = \{a_G^F\}$, where $a \in A$, F a literal, G a conjunction of literals (a term). We call a_G^F a transition rule, G its *precondition* and F its *effect*.

a_G^F means "if G holds, executing a causes F to hold". Any deterministic transition relation can be described by a finite set of such propositions (if G is a formula, we can take

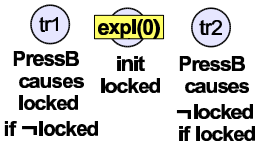

Figure 2: The DAG at time 0

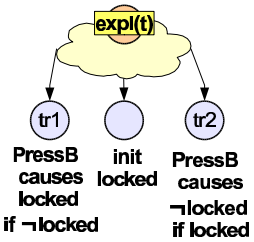

Figure 3: The DAG at time t

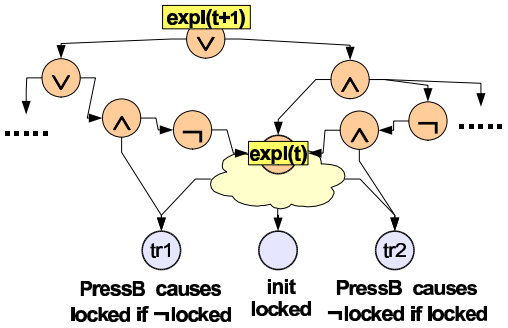

Figure 4: Building the DAG at time t+1, using time t

PROCEDURE DAG-SLAF($\langle a_i, o_i \rangle_{0<i\leq t}, \varphi$)
input: an action-observation sequence and a formula over P

1: **for** $f \in P$ **do**
2: $expl_f :=$ a new proposition $init_f$
3: $kb := \varphi$, where each $f \in P$ is replaced with $init_f$
{Lines 1-3: Preprocess φ}
4: **for** i=1...t **do** DAG-SLAF-STEP(a_i, o_i)
{Process the Sequence}
5: **return** $\bigwedge_{f\in P}(f \leftrightarrow expl_f) \wedge kb \wedge base$

PROCEDURE DAG-SLAF-STEP(a, o)
input: o an observation (a logical formula over P)

1: **for** $f \in P$ **do**
2: $kb := kb \wedge$ Poss(a,f) {see eq. (1)}
3: $expl'_f :=$ SuccState(a,f) {see eq. (2)}
4: ReplaceFluents($expl'_f$)
5: ReplaceFluents(kb)
{Lines 1-5: maintain kb, and keep the fluents' new status in $expl'$}
6: **for** $f \in P$ **do** $expl_f := expl'_f$
{Lines 1-6: Progress with action a}
7: $kb := kb \wedge o$
8: ReplaceFluents(kb) {Lines 7-8: Filter with observation o}

PROCEDURE ReplaceFluents(ψ)
input: ψ a formula

1: **for** $f \in P$ **do** replace f with $expl_f$ in ψ

Figure 1: *DAG-SLAF*

its DNF form and split to rules with term preconditions. F must be a term, so again we can split the rule into several rules with a literal as their effect).

Definition 3.1 (Transition Rules Semantics) *Given $s \in S, a \in A$ and R, a transition relation represented as a set of transition rules, we define s', the result of performing action a in state s: if $a^F_G \in R$, and $s \models G$, $s' \models F$. The rest of the literals do not change. If there such s', we say that action a is* possible *in s.*

We can now use logical formulas to encode belief states, using $L = L^0 \cup P$. A belief state φ is equivalent to $\{\langle s, R\rangle \mid s \wedge R \models \varphi\}$. Logical formulas and set-theoretic notions of belief states will be used interchangeably.

Algorithm Overview (see Figure 1): We are given φ, a formula over L. First, we convert it to a formula of the form $\bigwedge_{f\in P}(f \leftrightarrow expl_f) \wedge kb$, where kb and $expl_f$ do not include any fluents. We do it by adding new propositions to the language (see below). φ will remain in this form throughout the algorithm, and we will update $expl_f$ and kb.

We iterate through the action-observation sequence (procedure DAG-SLAF-STEP), while maintaining several formulas in a directed acyclic graph (DAG) structure. (1) For any fluent f, we maintain the logical formula, $expl_f$. This formula represents the explanations of why f is true. Every time step, we update $expl_f$, such that it is true if and only if f currently holds. The formula is updated according to successor-state axioms (line 3, and see details below). (2) We also maintain the formula kb, which stores additional knowledge that we gain. When we perform an action, kb asserts that the action was possible (line 2); when we receive an observation, we add it to kb (line 7).

We make sure that both formulas do not involve any fluent, but only propositions from $L^0 \cup I_P$. Updating the formulas inserts fluents into them, so we need to call *ReplaceFluents*. This procedure replaces the fluents with their (equivalent) explanations. We use a *DAG* implementation, so we add an edge to the relevant node– no need to copy the explanation formula; This allows us recursive sharing of formulas, and helps us maintain compactness.

After processing the sequence, we return the updated φ. In order to make sure that the models of φ are indeed valid models, we conjoin it with another formula, *base*:

$$base := [\textstyle\bigwedge_{a,F,G} \neg(a^F_G \wedge a^{\neg F}_G)] \wedge [\bigwedge_{a,F,G\to G'} (a^F_{G'} \rightarrow a^F_G)]$$

Note that we assume the actions in the sequence were possible; we can also handle the general case, if the agent can observe which actions were possible.

EXAMPLE– BUILDING THE DAG: In Figures 2-4 we see how the DAG is constructed. We focus on the way $expl_{locked}$ is updated throughout the sequence. A node marked *expl(i)* is the root of the formula that represents $expl_{locked}$ at time i.

At time 0 the explanation is just a leaf, $init_{locked}$. There are leafs for every proposition in $I_P \cup L^0$. At time t, $expl_{locked}$ is a DAG with the same leafs: we show how to progress with the action *press-BC-Lock* to the DAG of time $t+1$ using the graph of time t. $expl_{locked}$ of time $t+1$ is the top node in Figure 4. It is true iff one of its children is true. The left one ($\vee$ node) represents the case that a transition rule that causes *locked* was activated (its precondition held in time t). We show one transition rule, with precondition $\neg$*locked*.
The right child ($\wedge$ node) corresponds to the case that *locked*

held in time t, and no rule that causes $\neg$*locked* was activated. Note that we use the previous explanation of *locked* whenever we refer to its status at time t (in this figure, we reuse it three times).

Correctness of the Algorithm:

Theorem 3.2 *DAG-SLAF is correct. For any formula φ and a sequence of actions and observations $\langle a_i, o_i\rangle_{0<i<\leq t}$,*

$$\{\langle s, R\rangle \textit{ that satisfy DAG-SLAF}(\langle a_i, o_i\rangle_{0<i<\leq t}, \varphi)\} = \textit{SLAF}[\langle a_i, o_i\rangle_{0<i<\leq t}](\{\langle s, R\rangle \textit{ that satisfy } \varphi\}).$$

INTUITION: we define an effect model for action a at time t, $T_{\textit{eff}}(a,t)$, which is a logical formula consisting of Situation-Calculus-like axioms (Reiter 2001). It describes the ways in which performing the action at time t affects the world. We then show that *SLAF*$[a](\varphi)$ is equivalent to consequence finding (in a restricted language) of $\varphi \wedge T_{\textit{eff}}(a,t)$. Finally, we show that *DAG-SLAF* calculates those consequences.

Definition 3.3 ***Effect Model**:*
*For $a \in A$, define **the effect model** of a at time t to be:*

$$T_{\textit{eff}}(a,t) = a_t \rightarrow \bigwedge_{f\in P} \textit{Poss}(a,t,f) \wedge [f_{t+1} \leftrightarrow \textit{SuccState}(a,t,f)]$$

$$\textit{Poss}(a,t,f) = \neg[(\bigvee_G (a^f_G \wedge G_t)) \wedge (\bigvee_{G'} (a^{\neg f}_{G'} \wedge G'_t))] \quad (1)$$

$$\textit{SuccState}(a,t,f) = [(\bigvee_G (a^f_G \wedge G_t)) \vee (f_t \wedge (\bigwedge_{G'} \neg (a^{\neg f}_{G'} \wedge G'_t)))] \quad (2)$$

a_t asserts that action a occurred at time t, and f_{t+1} means that f held after performing that action.

The effect model corresponds to effect axioms and explanation closure axioms from Situation Calculus: a fluent f holds at time t iff either (a) there exists a transition rule a^f_G which is true, and was activated (G held in time t). (b) f held at time t, and there is no rule $a^{\neg f}_{G'}$ that is true and was activated. In other words, f was true and no transition rule changed it. Also, it asserts that the action was possible: there cannot be two activated rules with contradicting effects.

Definition 3.4 *We reuse the notation of SLAF:*
- *SLAF*$[a](\varphi) = Cn^{L_{t+1}}(\varphi_t \wedge a_t \wedge T_{\textit{eff}}(a,t))$
- *SLAF*$[o](\varphi) = \varphi \wedge o$

$Cn^{L_{t+1}}(\psi)$ denotes the consequences of ψ in the vocabulary that includes only fluents of time $t+1$ and L^0; $L_{t+1} = L^0 \cup P_{t+1}$, where $P_{t+1} = \{f_{t+1} | f \in P\}$.

Lemma 3.5 *If φ is a belief state formula, and $a \in A$*
SLAF$[a](\{\langle s, R\rangle \in S \times \mathcal{R} \mid \langle s, R\rangle$ *satisfies* $\varphi\}) =$
$\{\langle s, R\rangle \in S \times \mathcal{R} \mid \langle s, R\rangle$ *satisfies SLAF*$[a](\varphi)\}$

Therefore, applying *SLAF* to a transition belief formula is equivalent to applying *SLAF* to a transition belief state (Proof idea: we show that the two sets have the same elements, using Craig's Interpolation Theorem for Propositional Logic). We now show that *DAG-SLAF* computes exactly the consequences in the restricted language. We denote by $\varphi[\psi/f]$ the formula φ, where every occurrence of fluent f is replaced with the formula ψ.

We need to compute the consequences of $\varphi_t \wedge a_t \wedge T_{\textit{eff}}(a,t)$. W.l.g., φ is of the form $kb \wedge \bigwedge_{f\in P} f \leftrightarrow \textit{expl}_f$, where kb and $\textit{expl}_f$ do not involve fluents. To do this, we add new propositions to our language, $\textit{init}_f$ for every fluent f. Every φ is equivalent to
$\varphi[\textit{init}_f/f \mid f \in P] \wedge \bigwedge_{f\in P}(f \leftrightarrow \textit{init}_f)$.
We replace any fluent g in $T_{\textit{eff}}(a,t)$ with its explanation.
$\varphi_t \wedge a_t \wedge T_{\textit{eff}}(a,t) \equiv$
$kb \wedge \bigwedge_{f\in P}(f_t \leftrightarrow \textit{expl}_f) \wedge a_t \wedge T_{\textit{eff}}(a,t) \equiv$
$kb \wedge \bigwedge_{f\in P}(f_t \leftrightarrow \textit{expl}_f) \wedge a_t \wedge T_{\textit{eff}}(a,t)[\textit{expl}_g/g_t \mid g \in P]$.
Consequence finding for L_{t+1} is the same as performing resolution on P_t and a_t.
$Cn^{L_{t+1}}(\varphi_t \wedge a_t \wedge T_{\textit{eff}}(a,t)) =$
$kb \wedge \bigwedge_{f\in P} \textit{Poss}(a,t,f)[\textit{expl}_g/g_t \mid g \in P] \wedge$
$\bigwedge_{f\in P}(f_{t+1} \leftrightarrow \textit{SuccState}(a,t,f)[\textit{expl}_g/g_t \mid g \in P])$
This is *exactly* what *DAG-SLAF* computes. The proof for handling observations is similar. ■

COMPLEXITY In order to keep the representation compact and the algorithm tractable, we maintain a *DAG* instead of a formula. That is, when *ReplaceFluents* replaces f with $\textit{expl}_f$, we only need to update a pointer, rather than copying the expression again.
The space and time complexities of the algorithm are $O(|\varphi_0| + |\textit{Obs}| + tk(2|P|)^{k+1})$, where φ_0 is the initial belief state, $|\textit{Obs}|$ is the total length of the observations throughout the sequence (can be omitted if observations are always conjunctions of literals), t is the length of the sequence, and k is a parameter of the domain- the minimum number such that preconditions are k-DNF.

If there are no preconditions (always executable actions), we can maintain a "flat" formula (instead of a DAG) with complexity $O(|\varphi_0| + |\textit{Obs}| + t|P|)$.

Inference on DAGs The *DAG-SLAF* algorithm returns a formula represented as a DAG. We would like to perform inference on this formula. We can always "flatten" the DAG and use a SAT solver, but then we will lose our compact representation. Instead, we used an algorithm which is a generalization of the PDDL algorithm. It is a recursive algorithm: in each iteration, it chooses an uninstantiated variable p. It updates the graph for the case of $p =$ *TRUE* (propagating this assignment recursively throughout the DAG). We get a smaller graph, and call the function recursively on it. If it finds a satisfying assignment, we return *TRUE*. Otherwise, we try $p =$ *FALSE*. If both did not succeed, we backtrack.

For a DAG of size m, with n leafs, the algorithm takes space $O(m)$, and time exponential in n and linear in m.

4 CNF-based Algorithms

The algorithm presented in the previous section encodes logical formulas using arbitrary directed-acyclic graphs. In some cases, however, it can be convenient to directly maintain transition belief states in CNF. Many powerful algorithms which we would like to be able to leverage for performing logical inference assume that the input formula is in CNF. Included in particular are modern satisfiability check-

ing algorithms such as zChaff, which can handle some formulas containing as many as one million clauses and ten million variables (Moskewicz *et al.* 2001).

PROCEDURE CNF-SLAF($\langle a, o\rangle, \varphi$)
input: successful action a, observation term o, fluent-factored transition belief formula φ (see Definition 4.1)
1: **for** $f \in \mathcal{P}$ **do**
2: $\quad kb_f := (\neg a^{[f]} \vee expl_f) \wedge (\neg a^{[\neg f]} \vee expl_{\neg f}) \wedge kb_f$
3: $\quad expl_f := (a^f \vee (a^{f\circ} \wedge \neg a^{[\neg f]} \wedge expl_f))$
4: $\quad expl_{\neg f} := (a^{\neg f} \vee (a^{f\circ} \wedge \neg a^{[f]} \wedge expl_{\neg f}))$
5: $\quad$ **if** $o \models f$ (f is observed) **then**
6: $\quad\quad \varphi_f := (\neg f \vee \top) \wedge (f \vee \bot) \wedge kb_f \wedge expl_f$
7: $\quad$ **else if** $o \models \neg f$ **then**
8: $\quad\quad$ set $\varphi_f := (\neg f \vee \bot) \wedge (f \vee \top) \wedge kb_f \wedge expl_{\neg f}$
9: Eliminate subsumed clauses in φ
10: **return** φ

PROCEDURE CNF-FSLAF($\langle a, o\rangle, \varphi$)
input: action a with known precondition term p, observation term o, φ a transition belief formula with the following factored form: $\varphi = \bigwedge_i \bigvee_j \varphi_{i,j}$, where each $\varphi_{i,j}$ is a fluent-factored formula.
1: **if** action a did not fail **then**
2: $\quad \varphi := \varphi \wedge \bigvee_i F(\neg l_i)$ where l_i are the literals appearing in p, and $F(l)$ is the fluent-factored formula equivalent to l (i.e., $F(l) = ((l \Rightarrow \top) \wedge (\neg l \Rightarrow \bot) \wedge \top) \wedge \bigwedge_{f\in\mathcal{P}}((f \Rightarrow \top) \wedge (\neg f \Rightarrow \top) \wedge \top)$).
3: $\quad \varphi_{i,j} :=$ AE-STRIPS-SLAF$[o](\varphi_{i,j})$
4: **else**
5: $\quad$ **for** $\varphi_{i,j}$ **in** φ **do**
6: $\quad\quad \varphi_{i,j} :=$ AE-STRIPS-SLAF$[p, \langle a, o\rangle](\varphi_{i,j})$
7: Each $\varphi_{i,j}$ is factored into $A_{i,j} \wedge B_{i,j}$ where $B_{i,j}$ contains all (and only) clauses containing a fluent from $\mathcal{P}$. For any i such that there exists B such that for all j, $B_{i,j} \equiv B$, replace $\bigvee_j \varphi_{i,j}$ with $B \wedge \bigvee_j A_{i,j}$
8: Eliminate subsumed clauses in φ
9: **return** φ

Figure 5: Single-step CNF SLAF algorithms.

4.1 STRIPS Actions

In this section, we present an efficient algorithm, CNF-SLAF (Figure 5), for progressing CNF transition belief formulas. Because CNF is an inherently less flexible knowledge representation language, these algorithms assume that the action model to be learned is unconditional STRIPS in order to still provide compactness guarantees. In particular, under certain conditions we show that transition belief formulas stay *indefinitely* compact.

For STRIPS actions, we consider a simpler propositional vocabulary than the one defined in Section 3. Define action propositions $L_f = \bigcup_{a\in\mathcal{A}}\{a^f, a^{f\circ}, a^{\neg f}, a^{[f]}, a^{[\neg f]}\}$ for every $f \in \mathcal{P}$. Let the vocabulary for the formulas representing transition belief states be defined as $L = \mathcal{P} \cup \bigcup_{f\in\mathcal{P}} L_f$. Intuitively: a^f ($a^{\neg f}$) is true if and only if action a in the transition relation causes f ($\neg f$) to hold after a is executed. $a^{f\circ}$ is true if and only if action a does not affect fluent f. $a^{[f]}$ ($a^{[\neg f]}$) is true if and only if f ($\neg f$) is in the precondition of a. Also, let the formula $base_{CNF}$ encode the axioms that "inconsistent" models are not possible. That is, models in which $a^{[f]}$ and $a^{[\neg f]}$ both hold, or models where it is not the case that exactly one of a^f, $a^{\neg f}$, or $a^{f\circ}$ hold are disallowed.

The algorithm CNF-SLAF maintains transition belief formulas in *fluent-factored* form:

Definition 4.1 *A transition belief formula φ is **fluent-factored** if it is in the form $\varphi = base_{CNF} \wedge \bigwedge_{f\in\mathcal{P}} \varphi_f$ with $\varphi_f = (\neg f \vee expl_f) \wedge (f \vee expl_{\neg f}) \wedge kb_f$ where $expl_f$, $expl_{\neg f}$, and kb_f contain only propositions from L_f and $A_f \models Cn^{L_f}(expl_f \vee expl_{\neg f})$.*

The subformula $expl_f$ ($expl_{\neg f}$) represents knowledge that is learned when the fluent f ($\neg f$) is observed. Note that the fluent-factored form for the transition belief state containing no prior knowledge about the state of the world or its action model is given by $\bigwedge_{f\in\mathcal{P}}(\neg f \vee \top) \wedge (f \vee \top) \wedge \top$.

The following theorem shows the correctness of CNF-SLAF. It also gives time and space complexity results. In particular, note that in the case that every fluent is observed every at most k steps, the transition belief formula stays in k-CNF (i.e., indefinitely compact).

Theorem 4.2 *SLAF$[\langle a, o\rangle](\varphi) \equiv$ CNF-SLAF$[\langle a, o\rangle](\varphi)$ for any fluent-factored formula φ, successfully executed action a, and observation term o. Moreover:*
1. *The procedure takes time linear in the size of the formula.*
2. *If each fluent is observed every at most k steps and the input formula is in k-CNF, the formula stays in k-CNF.*

4.2 Action Failures

In this section we will explicitly consider the notion of action failure. When an action fails, all worlds where the the preconditions were not met are to be removed from the belief state. That is, for a failed action a, Definition 2.3 becomes $SLAF[a](\varphi) = \{\langle s, R\rangle\}|\langle s, a, \cdot\rangle \notin R, \langle s, R\rangle \in \varphi\}$. Note that in the general case of learning any deterministic action model (Section 3), action failures can be handled implicitly by adding deterministic rules which cause a state to map to itself if the preconditions of the taken action do not hold. However, such transition rules cannot be compactly expressed in the restricted vocabulary from Section 3.

The algorithm CNF-FSLAF (Figure 5) handles the case of failed actions. It assumes that such failures are observable, and it also assumes that the preconditions of the failed actions are known beforehand. That is, the algorithm learns actions' effects but not actions' preconditions. The algorithm utilizes another algorithm, AE-STRIPS-SLAF (Amir 2005), as a subroutine. The following theorem shows that CNF-FSLAF produces a safe approximation of SLAF (that is, the correct action model is never lost) and gives conditions under which CNF-FSLAF produces an exact result. Additionally, time and space complexity are given.

Theorem 4.3 *$SLAF[\langle a, o\rangle](\varphi) \models$ CNF-FSLAF$[\langle a, o\rangle](\varphi)$, and if any of the conditions below hold then CNF-FSLAF$[\langle a, o\rangle](\varphi) \equiv$ SLAF$[\langle a, o\rangle](\varphi)$.*
1. *For every transition relation in φ, a maps states 1:1.*
2. *φ contains all its prime implicates.*
3. *There is at most one state s where $\langle s, R\rangle \in \varphi$ for any R.*

Theorem 4.4 *(Time and space complexity of CNF-FSLAF)*

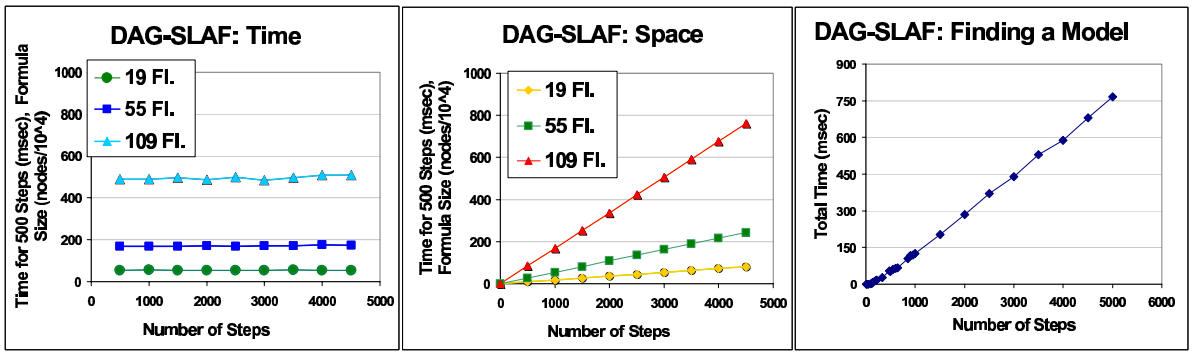

Figure 6: Experimental results for the DAG SLAF algorithm showing time required to process actions, space required to process actions, and time required to extract an action model.

1. *CNF-FSLAF takes time linear in the size of the formula for a single action, observation pair.*
2. *If every fluent is observed every at most k steps and the input formula is in km-CNF where m is the maximum size of any precondition, then the formula stays in km-CNF.*

5 Experimental Results

We evaluated our DAG-SLAF and CNF-SLAF algorithms on several domains, including Blocks world, Briefcase world, a variation of the Safe world, Bomb-in-toilet world, and the Driverlog domain from the 2002 International Planning Competition [1]. In the variation of the Safe world domain tested, the agent must try a number of different possible combinations in order to open a safe. In addition, there is an action `CHANGE` which permutes the right combination. Note that many of these domains (Briefcase world, Safe world, Bomb-in- toilet world) feature conditional actions, which cannot be handled by previous learning algorithms.

Figure 6 shows the time (always in milliseconds) and space results for instances of Block World of various sizes. The time plots show the time required to process every 500 actions. Note, that DAG-SLAF requires less than 2 milliseconds per action. The space plots show the total space needed in bytes. The rightmost graph shows the time taken by our inference procedure to find an action model of the Blocksworld domain (19 fluents) from the formula produced by DAG-SLAF. Finding an action model is accomplished by simply finding a model of the logical formula produced by the algorithm using our inference procedure; inference times are generally longer for more complex queries.

The middle graph in Figure 7 shows time taken to process actions for the CNF-SLAF algorithm compared to the AE-STRIPS-SLAF (series labeled (AE)) algorithm from (Amir 2005). The algorithms were tested on instances of various size of the Driverlog domain. The CNF-SLAF algorithm is comparable in speed, but slower by a constant factor. Note that CNF-SLAF solves a more difficult problem because AE-STRIPS-SLAF does not learn action preconditions.

Figure 8 shows part of an action model learned after running the DAG-SLAF algorithm on the Safe world domain

```
(CHANGE) causes (RIGHT COM2)
    if (RIGHT COM1)
(CHANGE) causes (RIGHT COM3)
    if (RIGHT COM2)
(CHANGE) causes (NOT (RIGHT COM2))
    if (RIGHT COM2)
(CLOSE) causes (NOT (SAFE-OPEN))
    if (TRUE)
(TRY COM1) causes (SAFE-OPEN)
    if (RIGHT COM1)
```

Figure 8: A Model of the Safe World (after 20 steps)

for 20 steps. The model shown was the first model extracted by our inference procedure on the logical formula returned by the algorithm. In this particular model, the algorithm was able to partially learn how the correct combination is permuted after the `CHANGE` action executes.

The learning rate graphs in Figure 7 show learning rates for the DAG-SLAF and CNF-SLAF algorithms. The graphs show the number of transition rules (corresponding to action propositions) successfully learned out of 50 randomly selected rules from the domain. Initially, the algorithms have *no* knowledge of the state of the world or the action model. The first graph shows results for the Safe world domain containing 20 fluents and the latter graph shows results for the Driverlog domain containing 15 fluents. Note that in the domains tested, it is not possible for any learning algorithm to completely learn how every action affects every fluent. This is because in these domains certain fluents never change, and thus it is impossible to learn how these fluents are affected by any action. Nevertheless, our algorithms demonstrate relatively fast learning rate under different degrees of partial observability. Unsurprisingly, the results show that learning rate increased as the degree of observability increased. It is also worth noting that inference time generally decreased as the degree of observability increased.

We also ran our CNF-SLAF algorithm on an instance of the Driverlog domain containing 50 objects and nearly 2000 fluents. 200 plans that solve 200 randomly created problems for this domain were generated. Each plan contained on average 21 actions. Both CNF-SLAF and the algorithm of (Qiang Yang & Jiang 2005) were run on these plans. The

[1] http://planning.cis.strath.ac.uk/competition/

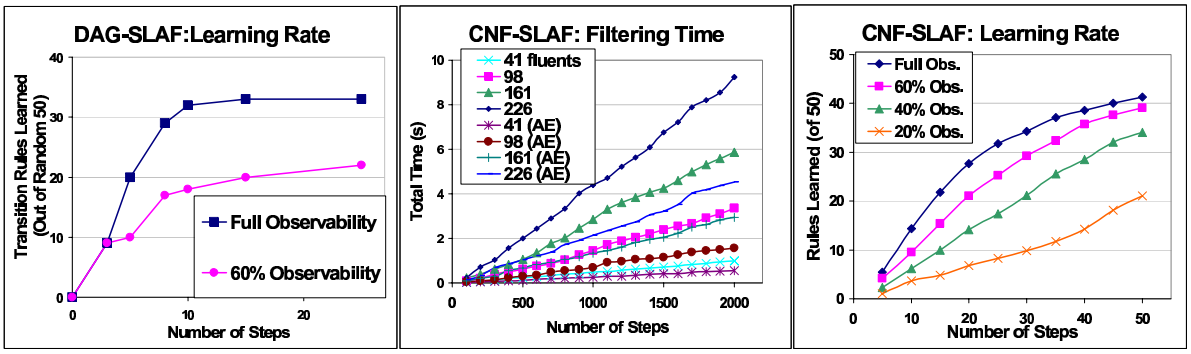

Figure 7: (Left, right) Experimental results showing learning rates for the DAG-SLAF and CNF-SLAF algorithms under different degrees of observability (the percentage of randomly selected fluents that are observed at each step). (Middle) Results showing time to process actions for the CNF-SLAF algorithm and AE-STRIPS-SLAF algorithms (series labeled (AE)).

algorithms were given only the initial state of each plan, the actions taken in the plans, and the goal condition of each problem. A simple extension to the CNF-SLAF algorithm was made to handle schematized actions. The CNF-SLAF algorithm took 61 seconds to process the plans, and using the SAT solver zChaff, 25 seconds were required to find a model. On the other hand, the algorithm of (Qiang Yang & Jiang 2005) required 83 seconds to find a model. Thus, the running times of the algorithms are comparable.

Our algorithm finds exactly all action models that are consistent with the actions and observations, while the algorithm of (Qiang Yang & Jiang 2005) only guesses a (possibly incorrect) action model heuristically. The goal of this paper is to perform exact learning where there is no assumed probabilistic prior on the set of action models. In this case, there is no principled way to favor one possible action model over another. However, introducing bias by preference between models was studied by the nonmonotonic reasoning community, and can be applied here as well.

6 Conclusion

We have presented new algorithms for learning deterministic action models in partially observable domains. Unlike previous algorithms, our algorithms handle the fully general case of arbitrary deterministic action models, perform exact learning, and are tractable. We have given theoretical analyses for our algorithms as well as empirical results which demonstrate effectiveness. Moreover, manual construction of action models for domain descriptions is tedious and painstaking, even for experts. Thus, our algorithms may help alleviate such knowledge-engineering bottlenecks. We plan to extend our work to stochastic domains.

The algorithms presented deal with purely propositional domains. Therefore in relational domains that contain many ground actions, these algorithms, applied naively, do not scale well. In such cases, algorithms that build upon the presented algorithms that take advantage of relational structure are better suited (Shahaf & Amir 2006).

Acknowlegement This work was supported by a Defense Advanced Research Projects Agency (DARPA) grant HR0011-05-1-0040, and by a DAF Air Force Research Laboratory Award FA8750-04-2-0222 (DARPA REAL program). The second author was supported by a fellowship from the University of Illinois Graduate College.

References

Amir, E. 2005. Learning partially observable deterministic action models. In *IJCAI '05*. MK.

Bertoli, P.; Cimatti, A.; Roveri, M.; and Traverso, P. 2001. Planning in nondeterministic domains under partial observability via symbolic model checking. In *IJCAI '01*, 473–478. MK.

Boyen, X.; Friedman, N.; and Koller, D. 1999. Discovering the hidden structure of complex dynamic systems. In *Proc. UAI '99*.

Cimatti, A., and Roveri, M. 2000. Conformant planning via symbolic model checking. *JAIR* 13:305–338.

Fikes, R., and Nilsson, N. J. 1971. Strips: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*

Ghahramani, Z., and Jordan, M. I. 1997. Factorial hidden markov models. *Machine Learning* 29:245–275.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language, version 1.2. Technical report, Yale center for computational vision and control.

Gil, Y. 1994. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proc. ICML-94*.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*.

Pasula, H. M.; Zettlemoyer, L. S.; and Kaelbling, L. P. 2004. Learning probabilistic relational planning rules. In *Proc. ICAPS'04*.

Qiang Yang, K. W., and Jiang, Y. 2005. Learning action models from plan examples with incomplete knowledge. In *Proc. ICAPS'05*. AAAI Press.

Reiter, R. 2001. *Knowledge In Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press.

Shahaf, D., and Amir, E. 2006. Learning partially observable action schemas. In *Proc. AAAI '06*.

Wang, X. 1995. Learning by observation and practice: an incremental approach for planning operator acquisition. In *Proc. ICML-95*, 549–557. MK.