Learning Partially Observable Action Schemas

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

Abstract

We present an algorithm that derives actions' effects and preconditions in partially observable, relational domains. Our algorithm has two unique features: an expressive relational language, and an exact tractable computation. An action-schema language that we present permits learning of preconditions and effects that include implicit objects and unstated relationships between objects. For example, we can learn that replacing a blown fuse turns on all the lights whose switch is set to on. The algorithm maintains and outputs a relational-logical representation of all possible action-schema models after a sequence of executed actions and partial observations. Importantly, our algorithm takes polynomial time in the number of time steps and predicates. Time dependence on other domain parameters varies with the action-schema language. Our experiments show that the relational structure speeds up both learning and generalization, and outperforms propositional learning methods. It also allows establishing apriori-unknown connections between objects (e.g. light bulbs and their switches), and permits learning conditional effects in realistic and complex situations. Our algorithm takes advantage of a DAG structure that can be updated efficiently and preserves compactness of representation.

1 Introduction

Agents that operate in unfamiliar domains can act intelligently if they learn the world's dynamics. Understanding the world's dynamics is particularly important in domains whose complete state is hidden and only partial observations are available. Example domains are active Web crawlers (that perform actions on pages), robots that explore buildings, and agents in rich virtual worlds.

Learning domain dynamics is difficult in general partially observable domains. An agent must learn how its actions affect the world as the world state changes and it is unsure about the exact state before or after the action. Current methods are successful, but assume full observability (e.g., learning planning operators (Gil 1994; Wang 1995; Pasula *et al.* 2004) and reinforcement learning (Sutton and Barto 1998)), or do not scale to large domains (reinforcement learning in POMDPs (Jaakkola *et al.* 1994; Littman 1996; Even-Dar *et al.* 2005)), or approximate the problem (Wu *et al.* 2005).

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

In this paper we present a relational-logical approach to scaling up action learning in deterministic partially observable domains. Focusing on deterministic domains and the relational approach yields a strong result. The algorithm that we present learns relational schema representations that are rich and surpass much of PDDL (Ghallab *et al.* 1998). Many of the benefits of the relational approach hold here, including faster convergence of learning, faster computation, and generalization from objects to classes.

Our learning algorithm uses an innovative boolean-circuit formula representation for possible transition models and world states (*transition belief states*). The learning algorithm is given a sequence of executed actions and perceived observations together with a formula representing the initial transition belief state. It updates this formula with every action and observation in the sequence in an online fashion. This update makes sure that the new formula represents exactly all the transition relations that are consistent with the actions and observations. The formula returned at the end includes all consistent models, which can be retrieved then with additional processing.

We show that updating such formulas using actions and observations takes polynomial time, is exact (it includes all consistent models and only them), and increases the formula size by at most a constant additive (without increasing the number of state variables). We do so by updating a directed acyclic graph (DAG) representation of the formula. We conclude that the overall exact learning problem is tractable, when there are no stochastic interferences; it takes time $O(t \cdot p^{k+1})$, for t time steps, p predicates, and k the maximal precondition length. Thus, this is the first tractable relational learning algorithm for partially observable relational domains.

These results are useful in deterministic domains that involve many objects, relations, and actions, e.g., Web mining, learning planning operator schemas from partially observed sequences, and exploration agents in virtual domains. In those domains, our algorithm determines how actions affect the world, and also which objects are affected by actions on other objects (e.g., associating light bulbs with their switches). The understanding developed in this work is also promising for relational structure in real-world partially observed stochastic domains. It might also help enabling reinforcement learning research to extend its reach beyond ex-

plicit or very simply structured state spaces.

Related Work Our approach is closest to (Amir 2005). There, a formula-update approach learns the effects (but not preconditions) of STRIPS actions in propositional, deterministic partially observable domains. In contrast, our algorithm learns models (preconditions and effects) that include conditional effects in a very expressive relational language. Consequently, our representation is significantly smaller, and the algorithm scales to much larger domains. Finally, our algorithm can generalize across instances, resulting in significantly stronger and faster learning results.

Another close approach is (Wu *et al.* 2005), which learns action models from plans. There, the output is a single model, which is built heuristically in a hill-climbing fashion. Consequently, the resulting model is sometimes inconsistent with the input. In contrast, our output is *exact*, and the formula that we produce accounts for exactly all of the possible transition models (within the chosen representation language). Furthermore, our approach accepts observations and observed action failures.

Another related approach is structure-learning in Dynamic Bayes Nets (Friedman *et al.* 1998). This approach addresses a more complex problem (stochastic domain), and applies hill-climbing EM. It is a propositional approach, and consequently it is limited to small domains. Also, it could have unbounded errors in discrete deterministic domains.

In recent years, *Relational* Paradigm is achieving important advances in learning and reasoning (Friedman *et al.* 1999; Dzeroski and Luc De Raedt 2001; Pasula *et al.* 2004; Getoor 2000). This approach takes advantage of the underlying structure of the data, in order to be able to generalize and scale up well. We incorporate those ideas into Logical Learning and present a *relational* logical approach.

We present our problem in Section 2, propose several representation languages in Section 3, present our algorithm in Section 4, and evaluate it experimentally in Section 5.

2 A Relational Transition Learning Problem

Consider the example in Figure 1. It presents a three-room domain. It is a *partially observable* domain – the agent can only observe the state of his current room. There are two switches in the middle room, and light bulbs in the other rooms; unbeknownst to our agent, the left and right switches affect the light bulbs in the left and right rooms, respectively.

The agent performs a sequence of actions: switching up the left switch and entering the left room. After each action, he gets some (partial) observations. The agent's goal is to determine the effects of these actions (to the extent he can), while also tracking the world. Furthermore, we want our agent to *generalize*: once he learns that switching up the left switch causes it to be up, he should guess that the same might hold for the other switch.

We define the problem formally as follows.

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

- *Obj, Pred, and Act are finite sets of objects in the world, predicate symbols, and action names, respectively. Predicates and actions also have an arity.*

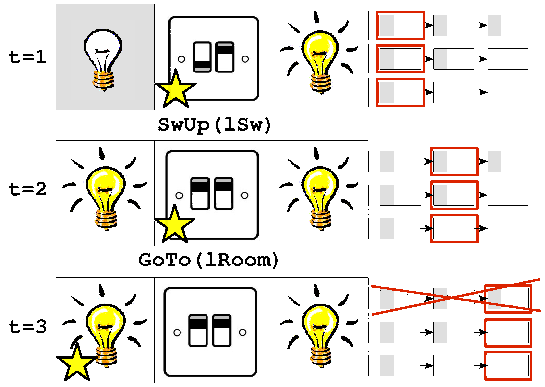


Figure 1: An action-observation sequence. The left part presents the actions and actual states timeline, and the right illustrates some possible ⟨World-State,Transition-Relation⟩ pairs at times 1,2,3. respectively. Every row is a transition-relation fragment related to the action sequence. A star indicates the agent's location.

- *P is a finite set of fluents of the form $p(c_1, ..., c_m)$, where $p \in Pred$, $c_1, .., c_m \in Obj$.*
- *$S \subseteq Pow(P)$ is the set of world states; a state $s \in S$ is the subset of P containing exactly the fluents true in s.*
- *$A \subseteq \{a(\bar{c}) \mid a \in Act, \bar{c} = (c_1, .., c_n), c_i \in Obj, \}$, ground instances of Act.*
- *$R \subseteq S \times A \times S$ is the transition relation.*

$\langle s, a(\bar{c}), s' \rangle \in R$ means that state s' is the result of performing action $a(\bar{c})$ in state s. In our light bulb world,

Obj={lSw, rSw, lBulb, rBulb, lRoom,...}, Act={GoTo$^{(1)}$, SwUp$^{(1)}$, SwDown$^{(1)}$}, Pred={On$^{(1)}$, Up$^{(1)}$, At$^{(1)}$}, P={ On(lBulb), On(rBulb), Up(lSw), Up(rSw), At(lRoom),..}

Our agent cannot observe the state of the world completely, and he does not know how his actions change it. One way to determine this is to maintain a set of possible world-states and transition relations that might govern the world.

Definition 2.2 (transition belief state) *Let $\mathcal{R}$ be the set of transition relations on S, A. A **transition belief state** $\rho \subseteq S \times \mathcal{R}$ is a set of pairs $\langle s, R \rangle$, where s is a state and R a transition relation.*

The agent updates his transition belief state as follows after he performs actions and receives observations.

Definition 2.3 ***Simultaneous Learning and Filtering of Schemas (SLAFS)*** *$\rho \subseteq S \times \mathcal{R}$ a transition belief state, $a(\bar{c}) \in A$ an action. We assume that observations $\bar{o}$ are logical sentences over P.*

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

We call step 2 *progression* with $a(\bar{c})$ and step 3 *filtering* with o. The intuition here is that every pair $\langle s', R \rangle$ transitions to a new pair $\langle \tilde{s}, R \rangle$ after 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. Note that we are interested in deterministic domains, i.e. for every $s, a(\bar{c})$ there is exactly one s'; if the action fails, we stay in the same state. An extension to domains in which there is *at most* one s' is easy.

EXAMPLE The right part of Figure 1 illustrates a (simplified) transition belief state, consisting of three $\langle s, R\rangle$ pairs, and the way it is updated. At the beginning, the agent considers three possibilities: all pairs agree on the initial state (the light is currently off in the left room), but they suggest different transition relations. Each chain illustrates the way the state of the world changes according to one of the relations (with respect to the specific action sequence): The first pair suggests that both actions do not affect the light, the second suggests that entering the room turns on the light, and the third associates the light with flipping the switch. After performing both actions, the agent observes that the light is on in the left room. This observation contradicts the first pair, so we eliminate it from his belief state.

3 Representing Transition Belief States

The naïve approach for representing transition belief states, enumeration, is intractable for non-trivial domains. We apply logic to represent transition belief states more compactly and to make learning tractable; later, we show how to solve *SLAFS* as a logical inference problem, while maintaining a compact representation of our transition belief state.

For humans, the action of opening a door and opening a book is the same meta-action; in both cases, the object will be opened. We try to capture this intuition.

Our logical languages represent transition belief states, using ground relational fluents from P (representing the state of the world), and *action-schemas*, which are propositions that represent the possible transition relations. Informally, schemas correspond to if-then rules; together, they are very similar to actions' specification in PDDL (Ghallab *et al.* 1998). For example, a schema in our language is **causes**(*SwUp(x)*, *Up(x)*, *TRUE*) (switching up an object causes it to be up, if *TRUE*). This schema represents a set of *instances*– ground transition rules, e.g. **causes**(*SwUp(lSw)*, *Up(lSw)*, *TRUE*) and **causes**(*SwUp(rSw)*, *Up(rSw)*, *TRUE*).

Definition 3.1 (Schemas) *A schema is a* proposition *of the form* ***causes***$(a(x_1,...,x_n), F, G)$ *(read: $a(\bar{x})$ causes F if G). $a \in$ Act is an n-ary action name, $\bar{x}$ are n different symbols, F (the effect) is a literal and G (the precondition) is a sentence, both over P_{Pat} which we now define. W.l.g., G is a conjunction of literals; otherwise, we can take its DNF form and split it to several schemas of this form.*

Let Pat be a set of symbols that includes $\{x_1, x_2, ...\}$. P_{Pat} is the set of patterned fluents over Pat:

$P_{Pat} = \{p(y_1,...,y_m) \mid p \in Pred, y_1,..,y_m \in Pat\}$.

In other words, a schema is a transition rule containing variables. Its instances can be calculated by assigning objects to these variables; the result is a ground transition rule **causes**$(a(\bar{c}), F, G)$ for $a \in A$,F, G over P. That is, every patterned fluent (*Up(x)*) becomes a fluent (*Up(lSw)*) after the assignment. In order to compute the instances, we may need to know some relations between objects, e.g. which switch controls which bulb. The set of possible relations is denoted by *RelatedObjs* (see *SL-H* below).

Definition 3.2 (Transition Rules Semantics) *Given a state s and a ground action $a(\bar{c})$, the resulting state s' satisfies every literal F which is the effect of an activated rule (a rule whose precondition held in s). The rest of the fluents do not change– in particular, if no precondition held, the state stays the same. If two rules with contradicting effects are activated, we say that the action is not possible.*

We now present several languages to represent schemas, starting from our most basic language.

SL$_0$: The Basic Language In this language, *Pat* $= \{x_1, x_2, ...\}$. For any schema **causes**$(a(x_1,...,x_n), F, G)$, F and G include only symbols from $x_1,...,x_n$. An instance is an assignment of objects to $x_1,...,x_n$. No related objects are needed (we set *RelatedObjs* $= \{$*TRUE*$\}$).
Examples include **causes**(*SwUp*(x_1), *Up*(x_1), *TRUE*), **causes**(*PutOn*(x_1, x_2), *On*(x_1, x_2), *Has*(x_1) $\wedge$ *Clear*(x_2)) (you can put a block that you hold on another, clear one). Any STRIPS domain can be represented in *SL*$_0$. Note that *SL*$_0$ can only describe domains in which every action $a(\bar{c})$ can affect only the elements in $\bar{c}$; the following extensions are more expressive.

SL-V: Adding Quantified Variables Some domains permit quantified variables in effects and preconditions; there, an action can affect objects other than its parameters. For example, **causes**(*sprayColor*(x_1), *Color*(x_2, x_1), *RobotAt*(x_3) $\wedge$ *At*(x_2, x_3)) (spraying color x_1 causes everything at the robot's location to be painted).

Pat is still $\{x_1, x_2, ...\}$, but F, G can include *any* symbol from *Pat*. $\{x_1,...,x_n\}$ are the action's parameters, $\bar{c}$ (thus, they are specified by the action $a(\bar{c})$). $\{x_{n+1}, ...\}$ represent free variables. Similarly to PDDL, free variables that appear in the effect part of the schema are considered to be universally quantified, and those that appear only in the precondition part are considered existentially quantified. In the previous example, x_2 is universally quantified and x_3 is existentially quantified. No related objects are needed.

In a more expressive variant of *SL-V*, the variables range only over objects that *cannot* be described any other way– that is, they do not range over the action's parameters, $\bar{c}$ (and in richer languages, not over their functions). This allows us to express defaults and exceptions, as in "blowing a fuse turns every light bulb off, except for a special (emergency) bulb, which is turned on", or "moving the rook to (c_1, c_2) causes it to attack every square (x, c_2) except for (c_1, c_2)". If *Pred* includes equality, we can use a simpler variant.

SL-H: Adding Hidden Object Functions *Pat* $= \{x_1, x_2, ...\} \cup \{h_1, h_2, ...\}$. We write h_1 as a shorthand for $h_1(\bar{x})$. This extension can handle hidden objects– objects that are affected by an action, although they do not appear in the action's parameters. For example, the rules **causes**(*SwUp(lSw)*, *On(lBulb)*, *TRUE*), **causes**(*SwUp(rSw)*, *On(rBulb)*, *TRUE*) are instances of the schema **causes**(*SwUp*(x_1), *On*($h_1(x_1)$), *TRUE*) (flipping up switch c_1 causes its light bulb, $h_1(c_1)$, to turn on. Note

that h_1 is a function of the action's parameters, which does not change over time). *SL-H* includes related object propositions, which specify these functions: $\{h_j(\vec{d}) = d' \mid d_i \in Obj, d' \in Obj \cup \perp\}$. $\perp$ means 'undefined'. Every *relobjs* $\in$ *RelatedObjs* completely specifies those functions.

Other Possible Extensions: *Extended Hidden Objects:* in *SL-H*, the hidden objects depended only on the action's parameters. We add to the language new functions, that can depend on the quantified variables as well. We add their specifications to *RelatedObjs*. (example schema: *OpenAll* causes all the doors for which we have a key to open)

Invented Predicates: sometimes the representation of the world does not enable us to learn the transition model. For example, consider a Block-world with predicates *On(x,y),Has(x)*; this suffices for describing any world state, but we cannot learn the precondition of *Take(x)*: it involves universal quantification, $\forall y.\neg$*On(y,x)*. If we add a predicate *Clear(x)*, it is easy to express *all* of the transition rules (including those that affect *Clear*) in our language. This idea is similar to the ones used in Constructive Induction and Predicate Invention (Muggleton and Buntine 1988).

We can also combine the languages mentioned above. For example, *SL-VH* allows both variables and hidden objects.

4 Learning Via Logical Inference

In this section we present a tractable algorithm that solves *SLAFS* exactly. The algorithm maintains a formula that represents the agent's transition belief state. In order to maintain compactness, the formula is represented as a DAG (directed acyclic graph). The algorithm can be applied to any schema language.

4.1 Update of Possible Transition Models

Algorithm Overview (see Figure 2): We are given φ, a formula over P and a schema language. φ represents the initial belief state (if we know nothing, φ =*TRUE*). For every fluent, f, we maintain a formula, $expl_f$ (intuition: the explanation of f's value). This formula is updated every time step, s.t. it is true if and only if f currently holds. Another formula, *kb*, stores the knowledge gained so far (by φ, the observations, and the actions that were performed). We make sure that those formulas *do not* involve any fluent (proposition from P). To do this, we add new propositions to the language, $init_f$. Those propositions represent the initial state of each fluent. *kb* and $expl_f$ can only involve those propositions and schema propositions.

At the beginning (steps 1-2 in DAG-SLAFS), we initialize *kb* and $expl_f$ according to φ, using those new propositions. Then we iterate: every time step, we progress with the action and filter with the observation. Procedure DAG-SLAFS-STEP updates $expl_f$ according to successor-state axioms (see below, and in procedure *ExplAfterAction*), and adds the assertion that the action was possible (procedure *PossibleAct*) to *kb*. Both updates insert fluents into our formulas; we use *ReplaceFluents* to replace the fluents for their (equivalent) explanations. This is done using pointers to the relevant node, so there is no need to copy the whole formula.

DAG-SLAFS-STEP also adds the observation to *kb*, and uses *ReplaceFluents* to eliminate fluents.

After every iteration, the updated φ is $kb \wedge \bigwedge_{f \in P}(f \leftrightarrow expl_f)$. At the end, we return it conjoined with *base*, which is a formula that each transition relation must satisfy; we use it to ensure that we return only legal relations.

base := $base_{relobjs} \wedge$

PROCEDURE DAG-SLAFS($\langle a_i(\bar{c}_i), o_i \rangle_{0<i\leq t}, \varphi$)
input: an action-observation sequence and a formula over P
1: **for** $f \in P$ **do** $expl_f$ = a new proposition $init_f$
2: kb = replace every occurrence of $f \in P$ by $init_f$ in φ
{Lines 1-2: Preprocessing of φ}
3: **for** i=1...t **do**
4: DAG-SLAFS-STEP($a_i(\bar{c}_i), o_i$) { Process Sequence }
5: **return** $\bigwedge_{f \in P}(f \leftrightarrow expl_f) \wedge kb \wedge base$

PROCEDURE DAG-SLAFS-STEP($a(\bar{c}), o$)
input: $a(\bar{c})$ an action, o an observation
1: **for** $f \in P$ **do**
2: $kb = kb \wedge$ PossibleAct(f,a($\bar{c}$))
3: $expl'_f$ = ExplAfterAction(f,a($\bar{c}$))
4: ReplaceFluents($expl'_f$)
5: ReplaceFluents(kb)
6: **for** $f \in P$ **do** $expl_f = expl'_f$ {1-6: Progress with action}
7: $kb = kb \wedge o$
8: ReplaceFluents(kb) {7-8: Filter with observation}

PROCEDURE PossibleAct(f,a($\bar{c}$))
input: $f \in P$, $a(\bar{c})$ an action
1: $\psi = TRUE$
2: **for** *relobjs* $\in$ *RelatedObjs* [1] **do**
3: Compute all schema-instance pairs with effect f, {(*sch+*,*inst+*)}, and those with effect $\neg f$, {(*sch-*,*inst-*)}, regarding action $a(\bar{c})$ and *relobjs*. [2]
4: $\psi = \psi \wedge relobjs \rightarrow$ $\neg[(\bigvee sch+ \wedge prec(inst+)) \wedge (\bigvee sch- \wedge prec(inst-))]$ {the action is possible if *relobjs* is true}
5: **return** ψ

PROCEDURE ExplAfterAction(f,a($\bar{c}$))
input: $f \in P$, $a(\bar{c})$ an action
1: $\psi = TRUE$
2: **for** *relobjs* $\in$ *RelatedObjs* **do**
3: Compute all schema-instance pairs with effect f, {(*sch+*,*inst+*)}, and those with effect $\neg f$, {(*sch-*,*inst-*)}, regarding action $a(\bar{c})$ and *relobjs*.
4: $\psi = \psi \wedge relobjs \rightarrow$ $[\bigvee sch+ \wedge prec(inst+)] \vee [f \wedge \neg(\bigvee sch- \wedge prec(inst-))]$ {f's value after the action if *relobjs* is true}
5: **return** ψ

PROCEDURE ReplaceFluents(ψ)
input: ψ a formula
1: **for** $f \in P$ **do** replace f by a pointer to $expl_f$ in ψ

Figure 2: *DAG-SLAFS*

[1] If the language does not involve related objects, assume *RelatedObjs*={*TRUE*}.

[2] Implementation depends on the schema language used.

$\bigwedge_{a,F,G} \neg(\textbf{causes}(a,F,G) \wedge \textbf{causes}(a,\neg F,G))$
$\bigwedge_{a,F,G\to G'} [\textbf{causes}(a,F,G') \to \textbf{causes}(a,F,G)\ ]$

and $base_{relobjs}$ is a formula that the related objects must satisfy. It depends on the schema language used.

(EXAMPLE– BUILDING THE DAG:) In Figure 3 we see how $expl_{On(lBulb)}$ is updated after the first action, *SwUp(lSw)*. The DAG in Figure 3 *is* the formula $expl'_{On(lBulb)}$ (after update). The node labeled "expl" is the root of the DAG before the update. The bottom nodes (the leaves) are the propositions: This is a simplified example, so we only show two *relobjs* nodes– (*p1,p2*), and two schemas– *tr1,tr2*. *tr1* claims that switching up an object x causes its hidden object, $h(x)$ to become on. *tr2* claims that it turns off everything that is currently on. *p1,p2* relate the left switch with the left and right light bulbs, respectively.
The $\to$ nodes (second layer) correspond to different cases of *relobjs*. The $\vee$ node is the explanation of *On(lBulb)* in case *p1* holds. Its left branch describes the case that the action caused the fluent to hold– *tr1* is true, and its preconditions hold; the right branch deals with the case that *On(lBulb)* held before the action, and the action did not change it (that is, either *tr2* is false, or its precondition does not hold). The formula in Figure 3 can be simplified, but this is an optimization that is not needed for our algorithm.

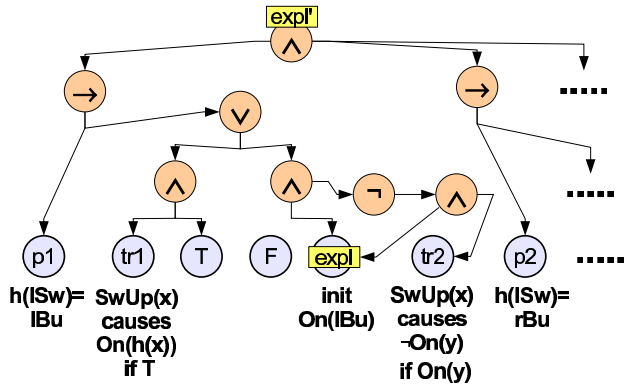


Figure 3: a (simplified) update of *On(lBulb)* after the first action, *SwUp(lSw)*

UPDATING THE FORMULAS– A CLOSER LOOK: Given *relobjs*, a ground action $a(\bar{c})$ and a fluent f, we want to update $expl_f$ and *kb*. To do this, we first identify the instances that can affect fluent f, and the schemas they correspond to.

Denote by (*sch+*,*inst+*) a schema-instance pair that can cause f. *inst+* is a transition rule with effect f, which is an instance of schema *sch+*. In other words– if the schema is true in our domain, and the precondition of the instance (*prec(inst)*) holds, f will hold after the action. Similarly, (*sch-*,*inst-*) is a pair that can cause $\neg f$. We need *relobjs* and $a(\bar{c})$ to match schemas and instances.

Fluent f is true after the action if either (1) a schema *sch+* is true and the precondition of its instance holds, or (2) f holds, and for every schema *sch-* that is true, no precondition holds. *kb* asserts that the action was possible; it cannot be the case that there are two schema-instance pairs, such that their effects are f and $\neg f$, and both preconditions hold.

We assume that the sequence consists of possible actions; if the agent has a way to know whether the action was possible, we do not need this assumption.

Theorem 4.1 *DAG-SLAFS is correct. For any formula φ and a sequence of actions and observations*

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

PROOF OVERVIEW we define an effect model for action $a(\bar{c})$ at time t, $T_{\textit{eff}}(a(\bar{c}),t)$, 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 $\textit{SLAFS}[a(\bar{c})](\varphi)$ is equivalent so consequence finding in a restricted language of $\varphi \wedge T_{\textit{eff}}(a(\bar{c}),t)$. Consequence finding can be done by resolving all fluents that are not in the language; we show that *DAG-SLAFS* calculates exactly those consequences.

COMPLEXITY In order to keep the representation compact and the algorithm tractable, we implement the algorithm to maintain a *DAG* instead of a flat formula. This way, when *ReplaceFluents* replaces f by $expl_f$, we only need to update a pointer, rather than copying the expression again. This allows us to recursively share subformulas.
Let φ_0 be the initial belief state, $|Obs|$ the total length of the observations (if observations are always conjunctions of literals, we can omit it), t is the length of the sequence. The maximal precondition length, k, is at most $\min\{k' \mid$ preconditions are k'-DNF$\}$. Let *pairs* be the maximal number of schema-instance pairs for an action $a(\bar{c})$. Let r_a, r_p be the maximal arities of actions and predicates, respectively.

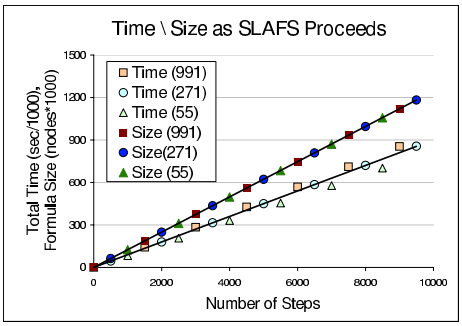
Theorem 4.2 *With the DAG implementation, DAG-SLAFS's time and space (formula size) complexities are $O(|\varphi_0| + |Obs| + t \cdot k \cdot pairs)$ We can maintain an NNF-DAG (no negation nodes) with the same complexity.*

If we allow preprocessing (allocating space for the leafs): In SL_0, $pairs = (2|Pred| \cdot {r_a}^{r_p})^{k+1}$. In *SL-H* with f functions, $pairs = |RelatedObjs|(2|Pred| \cdot (r_a+f)^{r_p})^{k+1}$. In *SL-V* without existential quantifiers, $|P|^{r_p} \cdot (2|Pred| \cdot (r_a + r_p)^{r_p})^{k+1}$, and with them- $|P|^{(k+1)r_p} \cdot (2|Pred| \cdot (r_a+(k+1)r_p)^{r_p})^{k+1}$. If we add invented predicates, we increase $|Pred|$ accordingly.

Since r_a, r_p and k are usually small, this is tractable. Interestingly, SL_0 (and some cases of *SL-H*) allow runtime that *does not* depend on the domain size (requiring a slightly different implementation). Importantly, SL_0 includes STRIPS.
If there are no preconditions (always executable actions), we can maintain a flat formula with the same complexity.

Note: The inference on the resulting DAG is difficult (SAT with $|P|$ variables). The related problem of temporal projection is coNP-hard when the initial state is not fully known.

Using the model Our algorithm computes a solution to *SLAFS* as a logical formula; we can use a SAT solver in order to answer queries about the world state and the transition model. If the formula is represented as a DAG, we



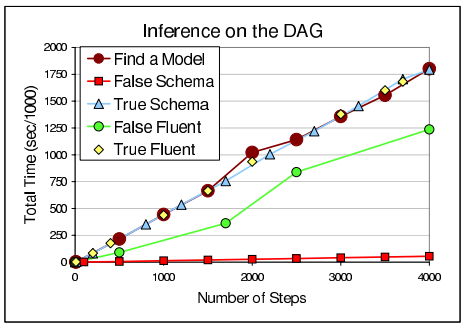


Figure 4: Left: Time and space for several Block-Worlds (numbers represent $|P|$). As can be seen, the time and space do not depend on the size of the domain. Slight time differences are due to a hash table. Right: Inference time on DAG-SLAFS' output, for several simple queries.

use an algorithm which adapts DPLL for DAGs (we have created such an implementation– see Section 5). Note that the number of variables in the formula is independent of the length of the sequence. Therefore, we can use DPLL-DAG and SAT solvers for very long sequences. We can also use bias (McCarthy 1986) to find minimal models. Preferential bias is well studied and fits easily with logical formula.

5 Experimental Results

We implemented and tested *DAG-SLAFS* for SL_0 and for a variant of *SL-V*; we also implemented a version for the case of always-executable actions, which returns a flat formula. In addition, we implemented a DPLL SAT-search algorithm for DAGs. It finds satisfying assignments for the algorithm's output. We tested the *SLAFS* algorithms on randomly generated partially observable action sequences, including STRIPS domains (Block-Worlds, Chess, Driverlog), and ADL, PDDL domains (Briefcase, Bomb-In-Toilet, Safe, Grid) of various size, ranging from tens to thousands of propositional fluents.

Figures 4, 6 present some of our results. We measured the time, space, *knowledge rate* (percentage of schemas that were learned, out of all the schemas currently in the knowledge base), and *learning rate* (percentage of schemas that were learned, out of all of the schemas that could have been learned from the given sequence). A schema is *learned*, if all models assign the same truth value to it.

As expected, the algorithm takes linear time and space in the sequence length, and does not depend on the domain's size (Figure 4). Importantly, simple SAT queries return relatively fast, especially regarding schemas which were contradicted by the sequence. Naturally, more complicated queries take longer time.

Decreasing the number of observations also resulted in long inference time: in the Bomb-In-Toilet domain, we generated several action-observation sequences, with different degrees of observability (from full observability to 10% of the fluents). We then chose randomly 40 transition rules, and checked how many of them were learned for each sequence. Not surprisingly, both learning and inference were faster when the number of observations was higher.

Another important observation is that, most of the schemas that one could learned were learned very quickly, even for larger domains. In most domains, more than 98% of schemas were learned after 200 steps (Figure 6). This is mainly because the number of action schemas *does not* depend on the size of the domain, e.g. all Block-Worlds have exactly the same number of schemas. Compare this with the decreasing knowledge rate in the propositional approach of (Amir 2005). The latter does not generalize across instances, and the number of encountered (propositional) transition rules grows faster than those that are learned.

```
(dunk ?bomb ?toilet) causes (NOT (armed
?bomb)) if (NOT (clogged ?toilet))
(dunk ?bomb ?toilet) causes (clogged
?toilet) if (TRUE)
(dunk ?bomb ?toilet) causes (toilet
?toilet) if (TRUE)
(flush ?toilet) causes (not (clogged
?toilet)) if (TRUE)
----------------
(dunk ?bomb ?toilet) causes (NOT (armed
?bomb)) if (AND (bomb ?bomb) (toilet
?toilet) (NOT (clogged ?toilet)))
(dunk ?bomb ?toilet) causes (clogged
?toilet) if (TRUE)
(flush ?toilet) causes (not (clogged
?toilet)) if (toilet ?toilet)
```

Figure 5: Possible Models of the Bomb-Toilet World (Top: after 5 steps. Bottom: after 20 steps)

In another comparison, we ran sequences from (Wu *et al.* 2005), and each one took a fraction of a second to process. We also cross-validated our output and the output of (Wu *et al.* 2005) with a known model. We found that several outputs of (Wu *et al.* 2005) were inconsistent with the action-observation sequence, while the true model was consistent with our final transition belief state.

Note that their algorithm returns one (approximate) model, whereas our algorithm return a formula that represents all consistent models. Figure 5 shows two models of Bomb-In-Toilet world. Those models were found by running our DPLL algorithm on the resulting DAG after 5 and 20 steps, and returning the first satisfying assignment. The

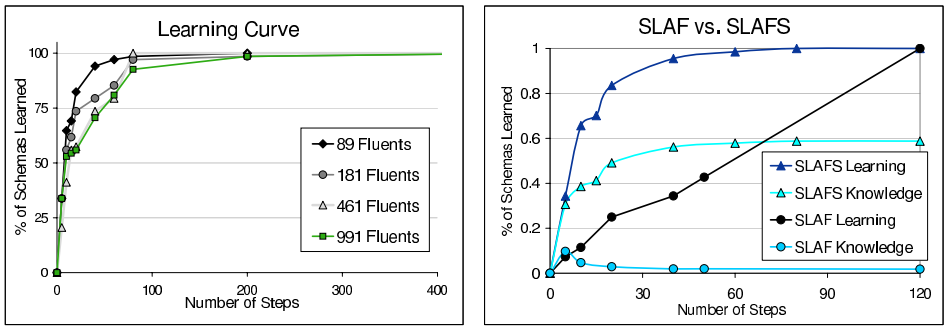


Figure 6: Block Worlds. Left: SLAFS learning rate. Right: SLAF (Amir 2005) and SLAFS knowledge and learning rates.

second model (20 steps) is more refined than the first one, and is quite close to the real model.
Trying a schema language that is too weak for the model (for example, trying SL_0 for the Briefcase World) resulted in no models, eventually.

6 Conclusions

We presented an approach for learning action schemas in partially observable domains. The contributions of our work are a formalization of the problem, the schema languages, and the tractable algorithm. Our results compare favorably with previous work, and we expect to apply and specialize them to agents in text-based adventure games, active Web crawling agents, and extensions of semantic Web services.

Significantly, our approach is a natural bridge between machine learning and logical knowledge representation. It shows how learning can be seen as logical reasoning in commonsense domains of interest to the KR community. It further shows how restrictions on one's knowledge representation language gives rise to efficient learning algorithms into that language. Its use of logical inference techniques (especially resolution theorem proving served in proving correctness of our theorems) and knowledge representation techniques makes it applicable to populating commonsense knowledge bases automatically.

6.1 Criticism and Future Directions

Noise: Because of the logical nature of our work, it is not robust to noise. This limits its potential utility. The naïve ways to handle noise (adding propositions) will affect the efficiency of the inference.

Data Structure: Although the representation of formulas as DAGs maintains compactness (compare with BDDs), we are not sure that it is the best representation possible. We are considering some alternative representations.

Open Questions: How does observability affect learning? How does the choice of schema language affect it? Also, a more detailed analysis of convergence to the correct underlying model is needed.

Acknowledgements This work was supported by a Defense Advanced Research Projects Agency (DARPA) grant HR0011-05-1-0040.

References

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

Saso Dzeroski and K. Driessens Luc De Raedt. Relational reinforcement learning. *Machine Learning*, 43(1-2):7–52, 2001.

E. Even-Dar, S. M. Kakade, and Y. Mansour. Reinforcement learning in POMDPs. In *IJCAI '05*, 2005.

N. Friedman, K. Murphy, and S. Russell. Learning the structure of dynamic probabilistic networks. In *Proc. UAI '98*. MK, 1998.

Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning probabilistic relational models. In *IJCAI '99*, pages 1300–1307. MK, 1999.

Lise Getoor. Learning probabilistic relational models. *Lecture Notes in Computer Science*, 1864:1300–1307, 2000.

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

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

T. Jaakkola, S. P. Singh, and M. I. Jordan. Reinforcement learning algorithm for partially observable Markov decision problems. In *Proc. NIPS'94*, volume 7, 1994.

M. L. Littman. *Algorithms for sequential decision making*. PhD thesis, Department of Computer Science, Brown University, 1996. Technical report CS-96-09.

J. McCarthy. Applications of circumscription in formalizing common sense knowledge. *Artificial Intelligence*, 28:89–116, 1986.

S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proc. ICML-88*, 1988.

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

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

R. S. Sutton and A. G. Barto. *Reinforcement Learning: an introduction*. MIT Press, 1998.

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

K. Wu, Q. Yang, and Y. Jiang. Arms: Action-relation modelling system for learning action models. Proc. ICAPS'05, 2005.