Learning Probabilistic Action Models from Interpretation Transitions

David Martínez
Institut de Robotica i Informatica Industrial (CSIC-UPC)
(*e-mail:* dmartinez@iri.upc.edu),

Tony Ribeiro
The Graduate University for Advanced Studies (Sokendai),
(*e-mail:* tony_ribeiro@nii.ac.jp),

Katsumi Inoue
National Institute of Informatics,
(*e-mail:* inoue@nii.ac.jp)

Guillem Alenyà, Carme Torras
Institut de Robotica i Informatica Industrial (CSIC-UPC)
(*e-mail:* {galenya,torras}@iri.upc.edu)

submitted 29 April 2015; accepted 5 June 2015

Abstract

There have been great advances in the probabilistic planning community during recent years, and planners can now provide solutions for very complex probabilistic tasks. However, planners require to have a model that represents the dynamics of the system, and in general these models are built by hand. In this paper, we present a framework to automatically infer probabilistic models from observations of the state transitions of a dynamic system. We propose an extension of previous works that perform learning from interpretation transitions. These works consider as input a set of state transitions and build a logic program that realizes the given transition relations. Here we extend this method to learn a compact set of probabilistic planning operators that capture probabilistic dynamics. Finally, we provide experimental validation of the quality of the learned models.
KEYWORDS:inductive logic programming, learning from interpretation transitions, probabilistic planning operators, action model learning, probabilistic planning

1 Introduction

Lately the performance of probabilistic task planners has improved to the point where they can find solutions in many complex scenarios (Kolobov et al. 2012; Keller and Eyerich 2012). Planners have been applied successfully in several fields, such as robotics (Chanel et al. 2014; Martínez et al. 2015) and aerospace (Surovik and Scheeres 2015). However, in most cases an expert is required to define the action model to be used by the planner. Creating a model is an error-prone task that requires in-depth knowledge about the problem and the planner, as well as extensive testing to assure the correctness of the model. We propose a new method to automatically learn action models based on a set of given state transitions.

Most previous approaches have only tackled the problem of learning deterministic action models (Zhuo et al. 2010), including uncertainty in the perceptions (Mourao et al. 2012). Regarding logic programming, abductive action learning has been studied based on *abductive event calculus* (Eshghi 1988), an abductive extension of event calculus, and has been extended for applications to planning, e.g., (Shanahan 2000). (Moyle 2003) uses an ILP technique to learn a causal theory based on event calculus (Kowalski and Sergot 1989), given examples of input-output relations. Probabilistic logic programs to maximize the probabilities of observations are learned by (Corapi et al. 2011; Sykes et al. 2013). They use parameter estimation to find the probabilities associated with each atom and rule, but it requires to code manually the restrictions for the initial set of rule candidates. Action models with uncertain effects have also been learned by (Rodrigues et al. 2012; Pasula et al. 2007) in STRIPS-like languages. Their optimization looks for planning operators that define as many effects as possible at once, which makes it a better choice for preconditions-effects based PPDDL domains (Younes and Littman 2004), such as the domains in past planning competitions (IPPC 2004-2008). In contrast our approach finds planning operators for each single action effect (i.e. rule head), which makes it a better choice to learn variable-based RDDL domains (Sanner 2010) used in newer planning competitions (IPPC 2011-2014).

Our approach learns an action model encoded as a set of planning operators. Each operator describes how the value of a predicate changes based on a set of preconditions given that an action is executed. We present a novel method to learn in two levels as shown in Fig. 1. On the right, the Learning From Interpretation Transitions (LFIT) framework (Inoue et al. 2014; Ribeiro and Inoue 2014) is used to generate rules. Given a set of state transitions of a system, it can learn a normal logic program that captures the system dynamics. The resulting rules are all candidates that have to be considered to select the best planning operators. On the left, the data is generalized between different objects by using a relational representation, and an optimization method is used to select the best subsets of planning operators that explain input transitions while maintaining generality. The dependency relations between the planning operators are used to efficiently select the best candidates. Our method is designed to learn RDDL-like (Younes and Littman 2004) operators, where each variable is updated separately based on a set of preconditions. To that end, in this paper we present the following novel improvements:

- The *LUST* algorithm: an extension of the *LFIT* framework to learn probabilistic dynamics from uncertain state transitions. The new algorithm that we propose can construct a model of a non-deterministic system by learning probabilistic rules.
- LUST also integrates multi-valued variables, which allows us to represent actions more efficiently since every transition has only one action of the many possible actions.
- The integration of logic programming to efficiently limit the number of rule candidates with a relational representation that provides better generalization, and an optimization method to select the best subsets of planning operators.

The paper is organized as follows. First we will introduce some background required to understand the logic rule learning, and the algorithms proposed to that end. Afterwards, the action model and the translations needed between logic programs and planning domains

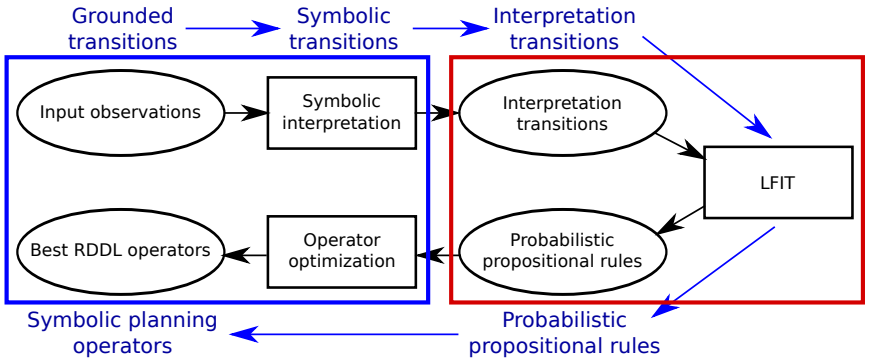

Fig. 1. Overview of the learning framework. The modules that interact with planning data and operators are in the blue rectangle on the left, while the modules related to logic programming are included inside the red rectangle on the right. The input of the method are grounded transitions, which are then converted to symbolic transitions to generalize between different objects. To obtain rule candidates, *LFIT* requires that the symbolic transitions are represented with propositional atoms. Finally, after rule candidates are obtained, they are transformed to planning operators to select the best set of operators.

are explained. Finally we show experimental results and conclusions of the presented approach.

2 Preliminaries

In this section we recall some preliminaries of logic programming. We also explain the basis of learning from interpretation transitions in order to make easier the understanding of its extension to multi-valued variable and probabilistic rule learning.

2.1 Logic Programming

We consider a first-order language and denote the Herbrand base (the set of all ground atoms) as $\mathcal{B}$. A (*normal*) *logic program* (NLP) is a set of *rules* of the form

$$A \leftarrow A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n \tag{1}$$

where A and A_i are atoms ($n \geq m \geq 0$). For any rule R of the form (1), the atom A is called the *head* of R and is denoted as $h(R)$, and the conjunction to the right of $\leftarrow$ is called the *body* of R. An (*Herbrand*) *interpretation* I is a subset of $\mathcal{B}$. For a logic program P and an Herbrand interpretation I, the *immediate consequence operator* (or T_P *operator*) (Apt et al. 1988) is the mapping $T_P : 2^{\mathcal{B}} \rightarrow 2^{\mathcal{B}}$:

$$T_P(I) = \{\, h(R) \mid R \in ground(P),\, b^+(R) \subseteq I,\, b^-(R) \cap I = \emptyset \}. \tag{2}$$

2.2 Learning from Interpretation Transitions

LF1T (Inoue et al. 2014) is an *any time algorithm* that takes a set of one-step state transitions E as input. These one-step state transitions can be considered as positive examples.

From these transitions, the algorithm learns a logic program P that represents the dynamics of E. To perform this learning process, we can iteratively consider one-step transitions. In *LF1T*, the set of all atoms $\mathcal{B}$ is assumed to be finite. In the input E, a state transition is represented by a pair of interpretations. The output of *LF1T* is a logic program that realizes all state transitions of E.

Learning from 1-Step Transitions (LF1T)

Input: E a set of state transitions (I, J) of a system S and $\mathcal{B}$ the set of all possible atoms that can appear in I and J.
Output: A logic program P such that $J = next(I)$ holds for any $(I, J) \in E$.

To build a logic program with *LF1T*, in (Ribeiro and Inoue 2014) we used a bottom-up method that generates hypotheses by *specialization* from the most general rules, that are fact rules, until the logic program is consistent with all input state transitions. Learning by specialization ensures to output the most general valid hypothesis.

2.3 Learning Multi-valued logic programs

Research in multi-valued logic programming has proceed along three different directions (Kifer and Subrahmanian 1992): bilattice-based logics (Fitting 1991; Ginsberg 1988), quantitative rule sets (Van Emden 1986) and annotated logics (Blair and Subrahmanian 1989; Blair and Subrahmanian 1988). The multi-valued logic representation used in our new algorithm is based on annotated logics. Here, to each atom corresponds a given set of values. In a rule, a literal is an atom annotated with one of these values. It allows us to represent annotated atoms simply as classical atoms and thus to remain in the normal logic program semantics.

In order to represent multi-valued variables, we now restrict all atoms of a logic program to the form var^{val}. The intuition behind this form is that var represents some variable of the system and val represents the value of this variable. In annotated logics, the atom var is said to be annotated by the constant val. We consider a *multi-valued logic program* as a set of *rules* of the form

$$var^{val} \leftarrow var_1^{val_1} \wedge \cdots \wedge var_n^{val_n} \tag{3}$$

where var^{val} and $var_i^{val_i}$'s are atoms $(n \geq 1)$. Like before, for any rule R of the form (3), left part of $\leftarrow$ is called the *head* of R and is denoted as $h(R)$, and the conjunction to the right of $\leftarrow$ is called the *body* of R. We represent the set of literals in the body of R of the form (3) as $b(R) = \{var_1^{val_1}, \ldots, var_n^{val_n}\}$. A rule R of the form (3) is interpreted as follows: the variable var takes the value val in the next state if all variables var_i have the value val_i in the current state. An interpretation of a multi-valued program provides the value of each variable of the system and is defined as follows.

Definition 1 (*Multi-valued Interpretation*)
Let $\mathcal{B}$ be a set of atoms where each element has the form var^{val}. An *interpretation* I of a set of atoms $\mathcal{B}$ is a subset of $\mathcal{B}$ where $\forall var^{val} \in \mathcal{B}, \exists var^{val'} \in I$ and $\forall var^{val'} \in I, \nexists var^{val''} \in I, val' \neq val''$.

For a system S represented by a multi-valued logic program P and a state s_1 represented

by an interpretation I, the successor of s_1 is represented by the interpretation:

$$next(I) = \{h(R) \mid R \in P, b(R) \subseteq I\}$$

The state transitions of a logic program P are represented by a set of pairs of interpretations $(I, next(I))$.

Definition 2 (*Multi-valued Consistency*)
Let R be a rule and (I, J) be a state transition. R is *consistent* with (I, J) iff $b(R) \subseteq I$ implies $h(R) \in J$. Let E be a set of state transitions, R is consistent with E if R is consistent with all state transitions of E. A logic program P is *consistent* with E if all rules of P are *consistent* with E.

Definition 3 (*Subsumption*)
Let R_1 and R_2 be two rules. If $h(R_1) = h(R_2)$ and $b(R_1) \subseteq b(R_2)$ then R_1 subsumes R_2. Let P be a logic program and R be a rule. P subsumes R if there exists a rule $R' \in P$ that subsumes R.

We say that a rule R_1 is *more general* than another rule R_2 if R_1 subsumes R_2. In particular, a rule R is *most general* if there is no rule $R'(\neq R)$ that subsumes R ($b(r) = \emptyset$).

Example 1
Let R_1 and R_2 be the two following rules: $R_1 = (a^1 \leftarrow b^1)$, $R_2 = (a^1 \leftarrow a^0 \wedge b^1)$, R_1 subsumes R_2 because $(b(R_1) = \{b^1\}) \subset (b(R_2) = \{a^0, b^1\})$.When R_1 appears in a logic program P, R_2 is useless for P, because whenever R_2 can be applied, R_1 can be applied.

To learn multi-valued logic programs with **LF1T** we need to adapt the ground resolution of (Inoue et al. 2014) and the least specialization of (Ribeiro and Inoue 2014) to handle non-boolean variables.

Definition 4 (*complement*)
Let R_1 and R_2 be two rules, R_2 is a complement of R_1 on var^{val} if $var^{val} \in b(R_1)$, $var^{val'} \in b(R_2), val \neq val'$ and $(b(R_2) \setminus \{var^{val'}\}) \subseteq (b(R_1) \setminus \{var^{val}\})$.

Definition 5 (*multi-valued ground resolution*)
Let R be a rule, P be a logic program and $\mathcal{B}$ be a set of atoms, R can be generalized on var^{val} if $\forall var^{val'} \in \mathcal{B}, val \neq val'$, $\exists R' \in P$ such that R' is a complement of R on var^{val}:

$$generalise(R, P) = h(R) \leftarrow b(R) \setminus var^{val}$$

Definition 6 (*Multi-valued least specialization*)
Let R_1 and R_2 be two rules such that $h(R_1) = h(R_2)$ and R_1 subsumes R_2. Let $\mathcal{B}$ be a set of atoms. The least specialization $ls(R_1, R_2, \mathcal{B})$ of R_1 over R_2 w.r.t $\mathcal{B}$ is

$$ls(R_1, R_2, \mathcal{B}) = \{h(R_1) \leftarrow b(R_1) \wedge var^{val'} | var^{val} \in b(R_2) \setminus b(R_1), var^{val'} \in \mathcal{B}, val' \neq val\}$$

Least specialization can be used on a rule R to avoid the subsumption of another rule with a minimal reduction of the generality of R. By extension, least specialization can be used on the rules of a logic program P to avoid the subsumption of a rule with a minimal reduction of the generality of P. Let P be a logic program, $\mathcal{B}$ be a set of atoms, R be a rule

and S be the set of all rules of P that subsume R. The least specialization $ls(P, R, \mathcal{B})$ of P by R w.r.t $\mathcal{B}$ is as follows:

$$ls(P, R, \mathcal{B}) = (P \setminus S) \cup (\bigcup_{R_P \in S} ls(R_P, R, \mathcal{B}))$$

LF1T starts with an initial logic program $P = \{var^{val} \leftarrow \mid \ var^{val} \in \mathcal{B}\}$. Then *LF1T* iteratively analyzes each transition $(I, J) \in E$. For each labeled atom A that **does not appear** in J, *LF1T* infers an **anti-rule** R_A^I:

$$R_A^I = A \leftarrow \bigwedge_{B_i \in I} B_i$$

A rule of P that subsumes such an anti-rule is not consistent with the transitions of E and must be revised. The idea is to use least specialization to make P consistent with the new transitions (I, J) by avoiding the subsumption of all anti-rules R_A^I inferred from (I, J). After least specialization, P becomes consistent with the new transition while remaining consistent with all previously analyzed transitions (theorem 3 of (Ribeiro and Inoue 2014)). When all transitions of E have been analyzed, *LF1T* outputs the rules of the system that realize E.

3 Learning from Uncertain State Transitions

In this section we extend the *LFIT* framework to learn probabilistic dynamics by proposing an extension of *LFIT* for learning from uncertain state transitions. Where other work like (Gutmann et al. 2011) perform inferences from a probabilistic logic program, what we do is inferring the rules of such logic program. The programs infered by our new algorithm are similar to paraconsistent logic program of (Blair and Subrahmanian 1988). The use of annotated atoms allows the learned programs to induce multiple values for the same represented variable. It allows us to represent multi-valued models and capture non-deterministic state transitions. Our semantics differs from previous work like (Fierens et al. 2013). There, the authors consider probabilistic logic programs as logic programs in which some of the facts are annotated with probabilities. But in our method, its the rules that have probabilities and they are independent.

3.1 Formalization

An non-deterministic system can be represented by a set of logic programs where the rules have the following form:

$$R = value(var, val, i, j) \leftarrow var_1^{val_1} \wedge \cdots \wedge var_n^{val_n}$$

where var^{val} and $var_i^{val_i}$ are atoms $(n \geq 1)$, $value(var, val, i, j)$ is the head of R again denoted $h(R)$ and i, j are natural numbers, $i \leq j$. Let I be a multi-valued interpretation. R means that i times over j, var takes the value val in the successor of I if $b(R) \subset I$.

Definition 7 (*Non-deterministic successors*)
Let I be the multi-valued interpretation of a state of a non-deterministic system S represented by a set of logic programs P. Let P' be a logic program, one of the successors of I

according to P' is

$$next(I, P') = \{var^{val} | R \in P', b(R) \subseteq I, h(R) = value(var, val, i, j), 0 < i \leq j\}$$

The set of *successors* of I in S according to P is

$$successor(I, P) = \{J | J \in next(I, P_i), P_i \in P\}$$

Example 2
Let R_1 and R_2 be two rules such that $R_1 = (value(a, 1, 10, 100) \leftarrow a^1)$ and $R_2 = (value(a, 2, 90, 100) \leftarrow a^1)$. Let S be a probabilistic system represented by a set of logic programs P such that $P = \{\{R_1\}, \{R_2\}\}$. The possible next states of I in S are $successor(I, P) = \{\{a^1\}, \{a^2\}\}$. The likelihood of having a^1 in the next state of I is 10% and the one of having a^2 is 90%.

3.2 Algorithm

We now present **LUST**, an extension of *LFIT* for Learning from Uncertain State Transition. **LUST** learns a set of deterministic logic programs. The main idea is that when two transitions are not consistent we need two different programs to realize them. The first program will realize the first transition and the second one will realize the second transition. The algorithm will output a set of logic programs such that every transition given as input is realized by at least one of those programs. The rules learned also provide the probability of the variable values in the next state. The probability of each rule $R = value(var, val, i, j) \leftarrow b(R)$ is simply obtained by counting how many transitions (I, J) it realizes (when $b(R) \subseteq I$ and $var^{val} \in J$), represented by i, over how many transitions it matches (when $b(R) \subseteq I$), represented by j.

LUST algorithm:

- **Input**: a set of pairs of interpretations E and a set of atoms $\mathcal{B}$.
- Step 1: Initialize a set of logic programs P with one program P_1 with fact rules for each atom of $\mathcal{B}$.
- Step 2: Pick (I, J) in E, check consistency of (I, J) with all programs of P:
- if there is no logic program in P that realizes (I, J) then
  - — copy one of the logic programs P_i into a P'_i and add rules in P'_i to realize (I, J).
  - — Use full ground resolution to generalize P'_i.
- Step 3: Revise all logic programs that realize (I, J) by using least specialization.
- Step 4: If there is a remaining transition in E, go to step 2.
- Step 5: Compute the probability of each rule of all programs P_i according to E.
- **Output**: P a set of multi-valued logic programs that realizes E.

4 Integration of Logic Programming and Planning Domains

In this section we describe the formalization used to represent planning operators, as well as the data conversions needed to learn symbolic operators from grounded transitions while using propositional logic programming.

4.1 Planning Model

Although *LUST* uses a propositional representation, the planning model uses a relational representation to provide a better generalization between different states. Relational domains represent the state structure and objects explicitly. These domains are described by using a vocabulary of predicates $\mathcal{P}$ and actions $\mathcal{A}$, and a set of objects C_π. Predicates and actions take objects as arguments to define their ground counterparts.

Example 3
Let *on(X,Y)* be a symbolic predicate, and { *box1*,*box2*,*box3*} be a set of objects. Three possible groundings of the symbolic predicate *on(X,Y)* with the given atoms are *on(box1, box2)*, *on(box1, box3)* and *on(box3, box1)*.

A ground predicate or action is equivalent to an atom. In a planning domain, a planning state s is a set of ground predicates $s = \{p_1, p_2, ..., p_n\}$ that is equivalent to an Herbrand interpretation.

Our planner operators represent a subset of RDDL domains (Sanner 2010). For each variable, we define the probability that it will become true in the next state based on a set of preconditions. In contrast to the full RDDL specification, these preconditions can only consist of an action and a set of predicates. Thus we define a planning operator as a tuple $o = \langle o_{p^{val}}, o_{act}, o_{prec}, o_{prob} \rangle$ where:

- $o_{p^{val}}$ is a predicate p whose value can change to val by applying the operator. It is equivalent to the head of a logic rule.
- o_{act} is the action that has to be executed for the operator to be applicable.
- o_{prec} is a set of predicates that have to be satisfied in the current state so that the planning operator is applicable. It is equivalent to the body of a logic rule.
- o_{prob} is the probability that p^{val} will be true.

4.2 Data Representation

To have a more general and compact model, we are using a relational representation at the planning level. The input of our method consists of state transitions that are tuples $t = \langle s, act, s' \rangle$ where s and s' are the states before and after executing the action act. The states consist of sets of ground predicates, and act is a grounded action. On the other hand, the output is a set of symbolic (i.e. non-grounded) planning operators. Therefore, our method transforms initial grounded data to symbolic planning operators. Moreover, *LUST* works with propositional atoms, so a transformation from symbolic predicates to atoms and back to symbolic predicates is also needed. Figure 1 shows the needed data conversions, which are explained in more detail below.

Transform grounded transitions to symbolic transitions:

- **Input**: a set of grounded transitions $T = [t_1, t_2, ..., t_n]$.
- For each transition $t = \langle s, act, s' \rangle$:
  - — Take every argument χ of the action act.
  - — Substitute χ for a default symbolic parameter in s, act, and s'.
  - — Create a new transition t' with the symbolic predicates in s and s' and the symbolic action act.

— Add the new transition t' to T'.

- **Output**: set of symbolic transitions T'.

Transform a symbolic transition to an interpretation transition:

- **Input**: a symbolic transition $t = \langle s, act, s' \rangle$.
- Assign an atom to each predicate in s, act and s'.
- $I =$ atoms that correspond to s.
- Add act as an atom in I that represents the action.
- $J =$ atoms that correspond to s'.
- **Output**: interpretation transition (I, J).

To transform a planning symbolic transition to an interpretation transition, a labeled atom that encodes the action is added to the body of the interpretation transition. As each transition has exactly one action, a multi-valued variable represents it more efficiently than boolean, otherwise every action would have to be represented as different variables with only one being true. Moreover, each symbolic predicate value is represented by one labeled atom. After the logic programs are generated, the labeled atoms are translated back to symbolic predicates by using the same conversion.

Transform a logic program to a set of planning operators:

- **Input**: a logic program P.
- For every rule $R \in P$ such that $h(R) = value(var, val, i, j)$, a planning operator o is created so that:
  — $o_{p^{val}} = var^{val}$.
  — $o_{act} =$ the action in the atoms of $b(R)$.
  — $o_{prec} =$ the set of atoms in $b(R)$ that represent predicates.
  — $o_{prob} = i/j$.
  — Add o to $\mathcal{O}$
- **Output**: set of planning operators $\mathcal{O}$.

5 Selecting the Set of Planning Operators

In this section we present the method to select the best subset of probabilistic planning operators by using the set of logic programs generated by *LF1T*. First, the requirements that the planning operators have to satisfy are presented. Afterwards, we explain the preferences to decide which are the best planning operators. Finally, the method to obtain the desired planning operators is described.

5.1 Planning Operators Requirements

Probabilistic planners require that only one planning operator can be applied in each state-action pair. Therefore the model has to be defined with a set of non-conflicting operators, so the planner can always decide which operator to apply for each state-action pair.

Definition 8 (*Conflicting planning operators*)
Let o_1 and o_2 be two planning operators that represent the same action $o_{1,act} = o_{2,act}$ and change the same predicate $o_{1,p^{val}} = o_{2,p^{val}}$ with different probabilities $o_{1,prob} \neq o_{2,prob}$. A conflict exists between both planning operators if $\exists s \mid o_{1,prec} \subset s,\ o_{2,prec} \subset s$.

5.2 Score Function

LUST provides the minimal set of rules that describe all possible transitions. However, the best subset of non-conflicting planning operators has to be selected to create the model. To decide which are the best set of operators the algorithm uses a score function. The algorithm prefers operators with a high likelihood (i.e. that can successfully explain the state transitions), but also has a regularization term to avoid overfitting to the training data. The regularization is based on the number of planning operators and preconditions in those operators, so that general operators are preferred when the likelihood of both is similar. This regularization penalty is bigger when there are few training transitions to estimate each operator, as general operators are preferred to poorly estimated specific ones. On the other hand, the regularization penalty decreases as our estimate improves with more transitions because the method can be more confident about the operators. The following functions are used to define the score function.

- The likelihood $P(t|\mathcal{O}) = P(s'|s, act, o) \mid (o \in \mathcal{O}, o_{prec} \in s, o_{act} = act)$ is the probability that the transition t is covered by the set of planning operators $\mathcal{O}$.
- The penalty term $Pen(\mathcal{O}) = |\mathcal{O}| + \frac{1}{|\mathcal{O}|}\sum_{o\in\mathcal{O}} |o_{prec}|$ is the number of planning operators plus the average number of preconditions that they have.
- The confidence in a planning operator $Conf(\mathcal{O}, T)$ is bounded by using the Hoeffding's inequality. The probability that our estimate $\widehat{o_{prob}}$ is accurate enough $|\widehat{o_{prob}} - o_{prob}| \leq \epsilon$ with a number of samples $|T|$ is bounded by $Conf(\mathcal{O}, T) \leq 1 - 2e^{-2\epsilon^2|T|}$.

Finally, the proposed score function is defined as

$$S(\mathcal{O}, T) = \frac{1}{|T|}\left(\sum_{t\in T} P(t|\mathcal{O})\right) - \alpha\frac{Pen(\mathcal{O})}{Conf(\mathcal{O}, T)} \tag{4}$$

where α is a scaling parameter for the penalty term. Also note that $Conf(\mathcal{O}, T) \simeq 1$ when the number of input transitions is large, so the penalty term will always be present to ensure general operators are preferred.

5.3 Selecting the Best Planning Operator Subset

In this section we present the algorithm used to select the subset of non-conflicting planning operators that maximizes the score function. To do it efficiently we make use of the dependency relations between operators by using the subsumption relation graph defined below.

Definition 9 (*Planning operator subsumption relation*)
Let o_1 and o_2 be two planning operators that represent the same action $o_{1,act} = o_{2,act}$ and the same effect $o_{1,p^{val}} = o_{2,p^{val}}$. o_1 subsumes the operator o_2 if $o_{1,prec} \subset o_{2,prec}$.

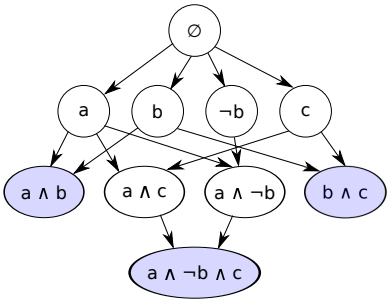

Fig. 2. Example of a subsumption graph. In each node, the planning operator preconditions are shown. Leaves are the nodes painted in blue.

Definition 10 (*Subsumption graph*)
The subsumption graph $G_{\mathcal{O}}$ of a set of planning operators $\mathcal{O} = \{o_1, ..., o_n\}$ is a directed graph with arcs (o_i, o_j) when o_i subsumes of o_j and $|o_{j,prec}| - |o_{i,prec}| = 1$. Figure 2 shows an example of a subsumption graph. We will call *leaves* $L(G_{\mathcal{O}})$ of the graph all nodes that do not have a child.

The subsumption graph orders the rules in levels that represent the generality of the operator: the less predicates in the preconditions the more general the operator is. This graph provides two advantages: generalizing operators is easier as we only have to follow the arcs, and it reduces the number of conflicts to check because general operators won't be checked if other more specific operators that represent the same dynamics exist. Below a greedy approach to efficiently obtain a set of good planning operators is provided.

Select set of planning operators:

- Create the subsumption graph.
- Repeat until nothing changes:
  - — Select the graph leaves, and choose the subset of non-conflicting leafs with the highest score. Remove the other leaves.
  - — Check if replacing any operator by a more general one (i.e. the operator parents in the subsumbtion graph) improves the score. If so, remove the specific operator.
- The result is the set of leaves in the remaining graph.

6 Evaluation

In this section we provide an experimental evaluation of our approach by learning two domains of the 2014 International Probabilistic Planning Competition (IPPC). The experiments use transitions $t = \langle s, act, s' \rangle$ generated by randomly constructing a state s, randomly picking the arguments of the action act, and then applying the action to generate the state s'. The distribution of samples is biased to guarantee that half of the samples have a chance to change the state. To measure the quality of the learned models, the errors shown represent the differences between the learned operators and the ground truth ones. For each incorrect predicate in an operator preconditions, the number was increased by 1.

The left plot in Fig. 3 shows the results obtained in the Triangle Tireworld domain. In this domain, a car has to move to its destination, but it has a probability of getting a

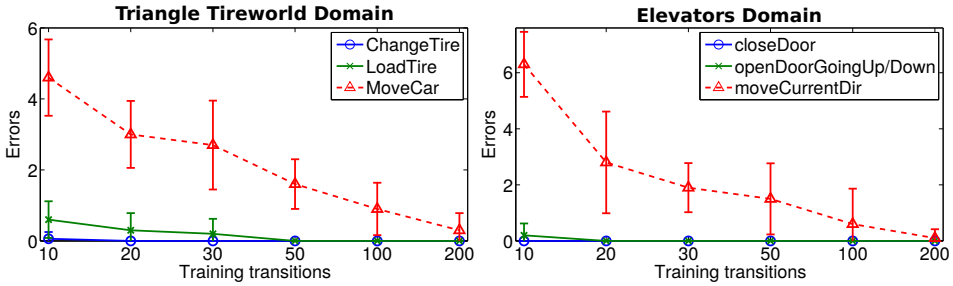

Fig. 3. Results of learning the Triangle Tireworld domain and the Elevators domain from IPPC 2014. The results shown are the means and standard deviations obtained from 10 runs. The number of training transitions per action are shown.

flat tire while it moves. The domain has 3 actions. A "Change Tire" action that has only one precondition, a "Load Tire" action that has 2 preconditions to change one predicate, and a "MoveCar" action that changes 3 predicates based on 3 different preconditions. The results show that easy actions can be learned with just a few transitions, while very complex actions require 100 transitions until average errors below 1 are obtained.

The right plot in Fig. 3 shows the results obtained in the Elevators domain. In this domain, we are only learning the actions that interact with the elevators, and not the dynamic effects related to people using them. The "openDoorGoing*" and "closeDoor" actions are easy to learn, but the "moveCurrentDir" action has two effects with 3 different preconditions each, and another two effects with 4 preconditions. Therefore, to learn successfully the dynamics of "moveCurrentDir" a large number of input transitions is required.

7 Conclusions

We have presented a new approach to learn planning operators. In contrast to previous approaches, it learns the dynamics of every predicate independently, and thus can be applied to learn RDDL domains. The presented approach can find good sets of planning operators efficiently, as it combines logic programming to restrict the set of candidates, and then a optimization method to select a good subset of operators (i.e. a set that generalizes as much as possible while explaining well enough the input transitions). To that end, the LFIT algorithm has been extended to learn probabilistic dynamics and multi-valued variables. Finally, experimental validation is provided to show that the planning operators can be learned with a reduced number of transitions.

As future work, we would like to work on planning operators independent of actions. This would require to have a more intelligent method to generate symbolic transitions from grounded transitions so that changes not represented by actions could be learned. Moreover, optimal solutions for selecting the best subset of planning operators should be analyzed.

8 Acknowledgments

This research is supported in part by the JSPS 2014-2017 Grants-in-Aid for Scientific Research (B) No. 26540122 and 2014-2015 Challenging Exploratory Research No. 26280092. D. Martínez is also supported by the Spanish Ministry of Education, Culture and Sport via a FPU doctoral grant (FPU12-04173).

References

APT, K. R., BLAIR, H. A., AND WALKER, A. 1988. Towards a theory of declarative knowledge. Foundations of deductive databases and logic programming, 89.

BLAIR, H. A. AND SUBRAHMANIAN, V. 1988. Paraconsistent foundations for logic programming. Journal of non-classical logic 5, 2, 45–73.

BLAIR, H. A. AND SUBRAHMANIAN, V. 1989. Paraconsistent logic programming. Theoretical Computer Science 68, 2, 135 – 154.

CHANEL, C. P. C., LESIRE, C., AND TEICHTEIL-KÖNIGSBUCH, F. 2014. A robotic execution framework for online probabilistic (re) planning. In Twenty-Fourth International Conference on Automated Planning and Scheduling. 454–462.

CORAPI, D., SYKES, D., INOUE, K., AND RUSSO, A. 2011. Probabilistic rule learning in non-monotonic domains. In Computational Logic in Multi-Agent Systems. Springer, 243–258.

ESHGHI, K. 1988. Abductive planning with event calculus. In ICLP/SLP. 562–579.

FIERENS, D., VAN DEN BROECK, G., RENKENS, J., SHTERIONOV, D., GUTMANN, B., THON, I., JANSSENS, G., AND DE RAEDT, L. 2013. Inference and learning in probabilistic logic programs using weighted boolean formulas. Theory and Practice of Logic Programming, 1–44.

FITTING, M. 1991. Bilattices and the semantics of logic programming. The Journal of Logic Programming 11, 2, 91 – 116.

GINSBERG, M. L. 1988. Multivalued logics: A uniform approach to reasoning in artificial intelligence. Computational intelligence 4, 3, 265–316.

GUTMANN, B., THON, I., KIMMIG, A., BRUYNOOGHE, M., AND DE RAEDT, L. 2011. The magic of logical inference in probabilistic programming. Theory and Practice of Logic Programming 11, 4-5, 663–680.

INOUE, K., RIBEIRO, T., AND SAKAMA, C. 2014. Learning from interpretation transition. Machine Learning 94, 1, 51–79.

KELLER, T. AND EYERICH, P. 2012. PROST: Probabilistic Planning Based on UCT. In Proceedings of the International Conference on Automated Planning and Scheduling. 119–127.

KIFER, M. AND SUBRAHMANIAN, V. 1992. Theory of generalized annotated logic programming and its applications. Journal of Logic Programming 12, 4, 335–367.

KOLOBOV, A., MAUSAM, AND WELD, D. S. 2012. Lrtdp versus uct for online probabilistic planning. In Proceedings of AAAI Conference on Artificial Intelligence.

KOWALSKI, R. AND SERGOT, M. 1989. A logic-based calculus of events. In Foundations of knowledge base management. Springer, 23–55.

MARTÍNEZ, D., ALENYÀ, G., AND TORRAS, C. 2015. Planning robot manipulation to clean planar surfaces. Engineering Applications of Artificial Intelligence 39, March 2015, 23–32.

MOURAO, K., ZETTLEMOYER, L. S., PETRICK, R., AND STEEDMAN, M. 2012. Learning strips operators from noisy and incomplete observations. In Proceedings of the Conference on Uncertainty in Artificial Intelligence. 614–623.

MOYLE, S. 2003. Using theory completion to learn a robot navigation control program. In Inductive Logic Programming. Springer, 182–197.

PASULA, H. M., ZETTLEMOYER, L. S., AND KAELBLING, L. P. 2007. Learning symbolic models of stochastic domains. Journal of Artificial Intelligence Research 29, 1, 309–352.

RIBEIRO, T. AND INOUE, K. 2014. Learning prime implicant conditions from interpretation transition. In The 24th International Conference on Inductive Logic Programming. To appear (long paper) (http://tony.research.free.fr/paper/ILP2014long).

RODRIGUES, C., GÉRARD, P., ROUVEIROL, C., AND SOLDANO, H. 2012. Active learning of relational action models. In Inductive Logic Programming. Springer, 302–316.

SANNER, S. 2010. Relational dynamic influence diagram language (rddl): Language description. Unpublished ms. Australian National University.

SHANAHAN, M. 2000. An abductive event calculus planner. The Journal of Logic Programming 44, 1, 207–240.

SUROVIK, D. A. AND SCHEERES, D. J. 2015. Heuristic search and receding-horizon planning in complex spacecraft orbit domains. In Proceedings of the International Conference on Automated Planning and Scheduling.

SYKES, D., CORAPI, D., MAGEE, J., KRAMER, J., RUSSO, A., AND INOUE, K. 2013. Learning revised models for planning in adaptive systems. In Proceedings of the International Conference on Software Engineering. 63–71.

VAN EMDEN, M. H. 1986. Quantitative deduction and its fixpoint theory. The Journal of Logic Programming 3, 1, 37–53.

YOUNES, H. L. AND LITTMAN, M. L. 2004. Ppddl1. 0: An extension to pddl for expressing planning domains with probabilistic effects. Techn. Rep. CMU-CS-04-162.

ZHUO, H. H., YANG, Q., HU, D. H., AND LI, L. 2010. Learning complex action models with quantifiers and logical implications. Artificial Intelligence 174, 18, 1540–1569.