Markov Decision Processes Specified by Probabilistic Logic Programming: Representation and Solution

Thiago P. Bueno, **Denis D. Mauá**, **Leliane N. de Barros**
Instituto de Matemática e Estatística
Universidade de São Paulo
Rua do Matão, 1010
São Paulo, SP, Brazil
{tbueno,ddm,leliane}@ime.usp.br

Fabio G. Cozman
Escola Politécnica
Universidade de São Paulo
Av. Prof. Mello Moraes, 2231
São Paulo, SP, Brazil
fgcozman@usp.br

***Abstract*—Probabilistic logic programming combines logic and probability, so as to obtain a rich modeling language. In this work, we extend PROBLOG, a popular probabilistic logic programming language, with new constructs that allow the representation of (infinite-horizon) Markov decision processes. This new language can represent relational statements, including symmetric and transitive definitions, an advantage over other planning domain languages such as RDDL. We show how to exploit the logic structure in the language to perform Value Iteration. Preliminary experiments demonstrate the effectiveness of our framework.**

***Index Terms*—probabilistic planning, Markov decision process, sequential decision making, probabilistic logic programming.**

I. INTRODUCTION

Successfully solving real-world probabilistic planning problems relies on representing and manipulating structured knowledge involving relations, recursion and context-dependent information. This is usually achieved by specifying the transition model by some sort of relational dynamic Bayesian network [1], thus excluding the explicit representation of symmetric and transitive definitions of ground predicates. For example, suppose that a person has a prior probability of 0.2 of buying a certain product, but if one or more of his or her trustees has also bought the product, then the probability increases (syntax and semantics defined later):

$$\begin{aligned} &\text{buys}(X) \leftarrow \text{aux1}. && \mathbb{P}(\text{aux1}) = 0.2 \\ &\text{buys}(X) \leftarrow \text{trusts}(X, Y), \text{buys}(Y), \text{aux2}. && \mathbb{P}(\text{aux2}) = 0.3 \end{aligned}$$

These rules and assessments induce a probabilistic model where the probability of buys(ANN) depends on buys(BOB) and vice-versa if one trusts each other. Moreover, the probability of buys(BOB) can depend on buys(JOHN), even if BOB does not trust JOHN directly. This type of symmetric and transitive knowledge can induce *cycles in the transition and reward models* of a Markov Decision Process (MDP) and therefore cannot be modeled straightforwardly with standard planning languages such as PPDDL [2] and RDDL [1].

Probabilistic logic programming extends logic programming languages with random variables, thus allowing the specification of complex probabilistic distributions over the models of a logic program. Some examples include PRISM [3], ICL [4], BLOG [5], DDC [6], CLP(BN) [7] and PROBLOG [8]. These formalisms inherit from their logical counterparts the ability to represent relational knowledge, often allowing the description of symmetric and transitive definitions, typical of cyclic feedback systems. PROBLOG is particularly interesting, as it has a simple and yet powerful syntax and semantics, and counts with an efficient toolset of inference techniques, implemented in an open-source package[1].

In this work, we develop MDP-PROBLOG, a probabilistic programming framework, based on PROBLOG, that can be used to represent and solve probabilistic planning problems with rich domains, *including the representation of cyclic ground models*. While the use of probabilistic logic programming languages for planning is not novel [6], [9], our framework provides a simpler syntax and a more clear semantics than existing proposals. To solve an MDP in our framework, we combine standard Value Iteration [10] with state-of-the-art techniques developed for PROBLOG; most notably, the reduction of inference task to weighted model counting [11] over a weighted propositional formula that represents PROBLOG's ground logic program with weights associated with its probabilities.

The paper starts with some background knowledge on probabilistic logic programming and probabilistic planning, in Section II. We then define our description language in Section III, and show how to take advantage of PROBLOG capabilities to perform Value Iteration. In Section IV we discuss empirical results over an extended version of the viral marketing domain [12], as well as the sysadmin domain, a more traditional probabilistic planning problem [13]. Finally, we present a discussion of related work and how to address some interesting venues for further development in Sections V and VI.

[1] http://dtai.cs.kuleuven.be/problog/

978-1-5090-3566-3/16 $31.00 © 2016 IEEE
DOI 10.1109/BRACIS.2016.59

II. BACKGROUND

A. Probabilistic Logic Programming

Syntax. We assume a fixed vocabulary of relations, logical variables and constants. An atom is of the form $r(X_1, \ldots, X_n)$, where r is a predicate of arity n, and each X_i is either a constant or a logical variable. The *grounding* of a predicate is all the atoms derived by the *substitution* of logical variables by constants, therefore a *ground atom* contains only constants. A probabilistic fact is of the form $\theta :: f$, where $\theta \in [0, 1]$ and f is a ground atom; it represents a probability assessment $\mathbb{P}(f) = \theta$. A ground probabilistic logic program is a triple $L_p = (At, F_p, R)$ where:

(i) At is a finite set of ground atoms;
(ii) F_p is a finite set of probabilistic facts;
(iii) R is a finite set of *normal logic rules*[2] of the form
$$h :- b_1, ..., b_m, not(b_{m+1}), ..., not(b_n).,$$
where $h \in At \setminus F_p$ is the head, and $b_i \in At$ forms the body.

Semantics. We adopt Sato's distribution semantics [14], which specifies a distribution over logic programs induced by joint realizations of the probabilistic facts. The probabilistic facts are assumed to be stochastically independent. Every realization (total choice) of probabilistic facts induces a logic program L which includes the rules in the original program and the probabilistic facts assigned "true" (i.e., probabilities are discarded). For $F_p = \{\theta_1 :: f_2, ..., \theta_k :: f_k\}$ we have:

$$\mathbb{P}(L|L_p) = \prod_{f_i \in L} \theta_i \prod_{f_i \in F_p \setminus L} (1 - \theta_i). \quad (1)$$

The interpretation is that each probabilistic fact $\theta :: f$ appears in a logical program with probability θ. The assumption of independent probabilistic facts is not restrictive, since probabilistic logic programs can represent (possibly with the inclusion of additional atoms) any distribution over binary variables [15].

Success probability. The semantics of a logic program is given by its well-founded semantics [11]. We assume that each realization of probabilistic facts induces a logic program with a complete (i.e., two-valued) model. The *success probability* of a query $q \in At$ is:

$$\mathbb{P}(q|L_p) = \sum_{L : L \models q} \mathbb{P}(L|L_p), \quad (2)$$

where $L \models q$ denotes that q is true in the well-founded model of L.

B. Probabilistic Planning

MDP. A Markov Decision Problem [10] is defined by the 5-tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$ where:

(i) $\mathcal{S}$ is a finite set of completely observable states.
(ii) $\mathcal{A}$ is a finite set of actions.
(iii) $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is a transition model such that $\mathcal{T}(s, a, s') = \mathbb{P}(s'|s, a)$. Note that function $\mathcal{T}$ satisfies the first-order Markovian assumption, i.e., the next state s' is independent of all past states given the current state s and action a to be executed.
(iv) $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward model that represents the immediate return of the execution of an action in the current state.
(v) $\gamma \in [0, 1]$ is the discount factor for future returns.

The objective of solving an MDP is to select a *policy* $\pi : \mathcal{S} \to \mathcal{A}$ that maximizes the expectation of a utility function defined over a sequence of returns $\langle r_t \rangle_{t=0,...,H}$ induced by state transitions from an initial state s_0. In the particular case where $H \to \infty$, i.e., an MDP with infinite horizon, it is commonplace to define this utility function as the discounted sum of future returns: $E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0\right]$.

Value function. The value function of a state $s \in \mathcal{S}$ w.r.t. a policy π is defined by the function $V_\pi : \mathcal{S} \to \mathbb{R}$ such that:

$$V_\pi(s) = \mathcal{R}(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, \pi(s)) V_\pi(s'). \quad (3)$$

Bellman optimality. The Bellman optimality theorem shows that an infinite-horizon discounted MDP with $0 \leq \gamma < 1$ admits an optimal value function V^* such as for all $s \in \mathcal{S}$:

$$V^*(s) = \max_{a \in \mathcal{A}} \left\{ \mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|s, a) V^*(s') \right\}. \quad (4)$$

Factored state representation. In a factored state representation a state s is described by a set of *state fluents* $x_1, ..., x_n$. As a consequence of the conditional independence of state fluents of next state given current state and action, the transition probability distribution $\mathbb{P}(s'|s, a)$ factorizes as follows:

$$\mathbb{P}(s'|s, a) = \prod_{i=1}^{n} \mathbb{P}(x'_i|x_1, ..., x_n, a). \quad (5)$$

III. MDP-PROBLOG

In this work we propose an extension of the probabilistic programming languages PROBLOG [8], [11] and DT-PROBLOG [9] to represent and solve probabilistic planning problems modeled as infinite-horizon discounted MDPs. In this section we first define the representational language and then present a simple way of solving an MDP by means of dynamic programming built on top of PROBLOG inference mechanism.

Throughout this section we use an extended version of the `viral marketing` decision problem [9], [12] to illustrate the language concepts. The original problem is to decide for which individuals of a known social network it is worth marketing a product, given the costs and rewards involved in a marketing process and the fact that a person might buy the product after being marketed or because he or she trusts someone who already bought it. In our version, we turned the episodic decision problem into a sequential decision problem by adding a transition that models a delayed influence of marketing.

[2]Although only facts are annotated with probabilities in our definition, a probabilistic rule $\theta :: h :- b_1, not(b_2).$ is allowed, since it would just be a syntactic sugar for $h :- b_1, not(b_2), aux.$ and $\theta :: aux.$

A. Language definition

Syntax. An MDP-PROBLOG program is a valid PROBLOG program defined by the triple $L_{\text{MDP}} = (At, F_p, R)$ where:

(i) At is a finite set of atoms partitioned in
- $\mathcal{SF}$: a finite set of state fluents;
- $\mathcal{A}$: a finite set of action predicates;
- $\mathcal{U}$: a finite set of utility attribute predicates.

(ii) F_p is a finite set of auxiliary probabilistic facts.

(iii) R is a finite set of rules partitioned in
- $\mathcal{T}_r$: finite set of transition rules;
- $\mathcal{R}_r$: finite set of reward rules.

In general terms, the syntax of a MDP-PROBLOG program is based on the syntax of PROBLOG programs, but some restrictions are necessary to explicitly describe an MDP. First, we define special purposes predicates for declaring state fluents, actions and utility predicates. Second, we restrict the probabilistic logic rules so as to attend to the form of state-transition distribution programs in order to represent the transition and reward models. In addition, auxiliary facts can be defined as intermediate atoms to help composing the transition and reward rules or to define invariant conditions.

State fluents. In order to define the state variables $x_1, ..., x_n$ that represent the factored state $s \in \mathcal{S}$ of an MDP, we introduce the reserved predicate `state_fluent/1`. Such a predicate should have a single argument representing some state variable x_i. Optionally, one can compactly define all the set of state variables by means of intentional rules restricting the range of the logical variables in the head by qualifiers atoms in the body. Figure 1 illustrates this possibility for the `viral marketing` domain.

```
person(bob). person(ann). person(john).
state_fluent(marketed(P)) :- person(P).
```

Fig. 1. State fluents for a `viral marketing` problem: the state variable represented by the predicate `marketed(P)` indicates that some person `P` has been marketed in a given state.

Action fluents. In order to define the available actions $a \in \mathcal{A}$, we introduce the reserved predicate `action/1`. Such a predicate should have as its only argument an atom representing some action. As in the case for the state fluents, one can optionally use intentional rules to compactly define the set of actions. Figure 2 illustrates a possible action definition for the `viral marketing` domain.

```
action_fluent(market(L)) :- subset([bob,ann,john],L).
```

Fig. 2. Action predicates for a `viral marketing` problem: the action atom `market(L)` represents the action of marketing to a subset `L` of individuals of the network.

Utility predicates. The reward model $\mathcal{R}$ is specified using special-purpose utility attributes of the form `utility(`u_i`,`r_i`)` where u_i is a state fluent or an action predicate such that r_i is a numerical value of reward or cost, respectively. The set of all utility attributes $\mathcal{U}$ represents the overall immediate return of an action executed in the current state. Optionally, intentional rules are allowed to qualify and restrict variable terms in predicate u_i or in real value r_i through Prolog's arithmetic mechanism. Figure 3 illustrates a possible definition of costs and rewards for a `viral marketing` problem.

```
utility(buys(P,1), 5) :- person(P).
utility(market(L), Cost) :- subset([bob,ann,john],L),
                            length(L,S), Cost is -0.75*S.
```

Fig. 3. Utility predicates for a `viral marketing` problem: if a person buys, an immediate reward of 5 is given; for each possible action of marketing to a list of people its cost is proportional to the length of the list.

State-transition distribution programs. Because probabilistic logic programs encode distributions, we can use them to represent the state-transition distributions necessary for the definition of the transition function $\mathcal{T}$ of planning problems. To do so, we increment the arity of atoms in the probabilistic logic program by introducing labels t and $t+1$ in order to define the subsets At_t and At_{t+1}, such that $At_t \cap At_{t+1} = \emptyset$ and atoms in At_t never appear in the head of rules. Consequently, by restricting the probabilistic program in this particular way, we define a two-time slice transition in which the atoms in At_t represent the current state fluents (or derived predicates) and possible actions, and the atoms in At_{t+1} represent the successor state fluents (or derived predicates).

In Figure 4 and Figure 5 we present an example of such scheme for set of rules $\mathcal{T}_r$ and $\mathcal{R}_r$ describing the state transition and reward models of a `viral marketing` domain. Note that the set of atoms At is partitioned by the probabilistic rules into the subsets $At_t = \{$`marketed(P,0)`, `market(L,0)`$\}$ and $At_{t+1} = \{$`marketed(P,1)`, `buys(P,1)`$\}$. Note that other atoms such as `trusts(P,P2)`, `marketed_before`, `buy_from_marketing` and `buy_from_trust` are simply auxiliary or non-fluent atoms and need not to be considered in the time partition of the set At.

```
0.5::marketed_before.
marketed(P,1) :- market(L,0), member(P,L).
marketed(P,1) :- market(L,0), not(member(P,L))
                 marketed(P,0), marketed_before.
```

Fig. 4. State transition rules for the `viral marketing` problem: `marketed(P,0)` and `market(L,0)` are the current state fluents and actions respectively, `marketed(P,1)` are the next state fluents. The first rule defines that person `P` is deterministically marketed at time $t+1$ if a marketing action is targeted at `P` at time t. Second rule defines a stochastic marketing effect at time $t+1$ if a person `P` was not market at time t but has been marketed before. Auxiliary atom `marketed_before` sets the probability of this residual effect.

Semantics. The semantics of an MDP-PROBLOG program is defined in terms of the dependency graph of the ground program augmented by implicit value function nodes. In Figure 6 we show the ground dependency graph for a `viral marketing` problem consisting of only two individuals in the social network. The graph encodes the necessary dependencies to compute successive approximations of the expected future

```
0.2::buy_from_marketing(P) :- person(P).
0.3::buy_from_trust(P) :- person(P).
buys(P,1) :- marketed(P,1), buy_from_marketing(P).
buys(P,1) :- trusts(P,P2), buys(P2,1), buy_from_trust(P).
```

Fig. 5. Reward rules for the `viral marketing` domain: `buys(P,1)` represents the fact that a person `P` buys the product after being marketed or because someone `P` trusts bought the product. Each case has a corresponding probability given by the auxiliary probabilistic facts `buy_from_marketing` and `buy_from_trust`. Auxiliary predicate `trusts(P,P2)` helps define the social network by means of topology invariants not shown in the example.

returns by means of solving the following episodic decision-theoretic problem:

$$\max_{a\in\mathcal{A}} \left\{ \sum_{\mathcal{U}} r_i \mathbb{P}(u_i|L_p; s) \right\}, \tag{6}$$

where $\mathcal{U}$ is the set of all utility predicates u_i associated with its immediate return r_i, i.e. `utility`$(u_i, r_i) \in \mathcal{U}$, and s is the current state observed as evidence in the probabilistic program.

Formally, the semantics of an MDP-PROBLOG program is that solving the episodic decision-theoretic problem in Equation 6 is equivalent to solving a Bellman's backup for state s. Therefore, running the probabilistic program iteratively with updated value function utility nodes for every $s \in S$, the solver approximates an optimum solution of the infinite-horizon MDP by means of dynamic programming.

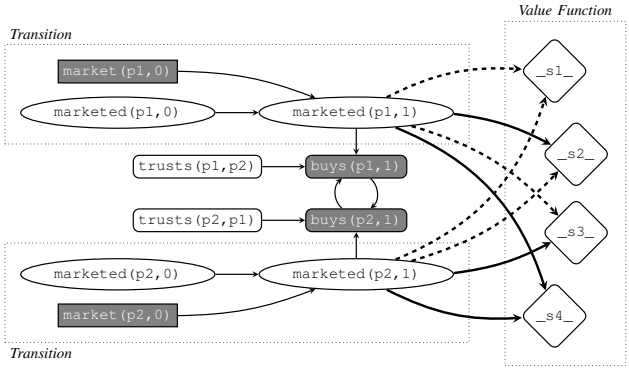

Fig. 6. Ground dependency graph encoding the transition and reward models with implicit value function nodes for the `viral marketing` domain for a two-individuals social network. The atoms `marketed(.,t)`, $t = 0, 1$, correspond to state fluents and `market(.,0)` correspond to actions. Auxiliary predicates `trusts(.,.)` and `buys(.,1)` help define the reward model. Darkened nodes have utility attributes associated. The implicit value function atoms correspond to the `_sj_` nodes, $j = 1, .., 4$. Thick arrows correspond to the functionally determined dependencies between next-state fluents and state labels. Dashed edges represent false values and normal edges represent true values. Note the cyclic dependency between atoms `buys(p1,1)` and `buys(p2,1)`.

It is important to note that the implicit value function nodes are not integral part of the user-defined probabilistic program used as input, but is automatically added by the solver in the internal representation so as to handle the approximations of the value function $V^{(i+1)}(s)$.

In Figure 7 we show an example for the `viral marketing` problem with just two individuals, hence with only two state variables and 4 state labels.

```
_s1_ :- not(marketed(p1,1)), not(marketed(p2,1)).
_s2_ :- marketed(p1,1),      not(marketed(p2,1)).
_s3_ :- not(marketed(p1,1)), marketed(p2,1).
_s4_ :- marketed(p1,1),      marketed(p2,1).
utility(_s1_, γV^(i)(s1)). utility(_s2_, γV^(i)(s2)).
utility(_s3_, γV^(i)(s3)). utility(_s4_, γV^(i)(s4)).
```

Fig. 7. Definition rules and utility predicates of value function nodes for a `viral marketing` problem: it specifies the value function of states s_1, s_2, s_3 and s_4 given the corresponding definitions composed of state fluents `marketed(p1,1)` and `marketed(p2,1)`

Equivalence of MDP-PROBLOG and Bellman's backup. To show this equivalence, we first note that $\mathcal{R}(s, a)$ can generally be decomposed as a sum in factored state representation $x_1, ..., x_n$, so as $\mathcal{R}(s, a) = \sum_{i=1}^{n} \mathcal{U}(x_i) + \mathcal{U}(a)$, where $\mathcal{U}(.)$ corresponds to the utility values of state fluents and action predicates in the program. Since $\mathbb{P}($`state_fluent`$(x_i)|L_p; s)$ and $\mathbb{P}($`action`$(a)|L_p; s)$ are either 1.0 or 0.0, because these predicates are always observed in each iteration, we can write:

$$\mathcal{R}(s,a) = \sum_{i=1}^{n} \mathbb{P}(\texttt{state\_fluent}(x_i)|L_p; s)\ \mathcal{U}(x_i) + \sum_{a\in\mathcal{A}} \mathbb{P}(\texttt{action}(a)|L_p; s)\ \mathcal{U}(a). \tag{7}$$

Moreover, by the construction of value function nodes and the transitivity of the dependence graph, we know that $\mathbb{P}(s'_j|s, a)$ equals the success probability $\mathbb{P}(\_s_j\_|L_p; s, a)$, given by:

$$\mathbb{P}(\_s_j\_|L_p; s, a) = \mathbb{P}(\_s_j\_|x'_1, ..., x'_n) \prod_{i=1}^{n} \mathbb{P}(x'_i|L_p; s, a). \tag{8}$$

Additionally, by the definition of the utility attributes of value function nodes, we verify that the expected future reward over all next states represented by state variables $x'_1, ..., x'_n$ can be computed by $\sum_{j=1}^{2^n} \mathbb{P}(\_s_j\_|L_p; s, a)\ \mathcal{U}(\_s_j\_)$, where $\mathcal{U}(\_s_j\_)$ is conveniently set to $\gamma V^{(i)}(s_j)$.

Finally, combining Equations (7) and (8), we conclude that

$$\begin{aligned} V^{(i+1)}(s) &= \max_{a\in\mathcal{A}} \left\{ \mathcal{R}(s,a) + \gamma \sum_{s'\in\mathcal{S}} \mathbb{P}(s'|s,a) V^{(i)}(s') \right\} \\ &= \max_{a\in\mathcal{A}} \left\{ \sum_{\mathcal{U}} r_i \mathbb{P}(u_i|L_p; s, a) \right\}. \end{aligned} \tag{9}$$

B. Solver

Our MDP-PROBLOG solver is implemented in Python3 and freely available at a public repository.[3] It solves the MDP problem using the built-in capabilities of PROBLOG as follows:

1) **preprocessing:** Each `state_fluent` gives rise to propositional facts representing state variables later used to set state evidence. Each `action` is translated to a set of facts and rules that constrain the program to disjointly consider only one action at a time. At this point,

[3] https://github.com/thiagopbueno/mdp-problog

all intentional rules are resolved and the implicit value function nodes are attached to the program.

2) **compilation:** it performs the relevant grounding of the augmented program with respect to the utility attributes and converts the ground program into a formulae and then compiles it to a specialized data structure used to solve the inference task by weighted model counting [11].
3) **value iteration**: in each iteration it sets the evidence of state s and runs the inference engine of PROBLOG to compute the transition probabilities used during the Bellman's backup for each possible action. By means of dynamic programming it approximates the value function $V(s)$ until ϵ-convergence.

In Algorithm 1 we outline the procedure that implements value iteration over the MDP-PROBLOG representation. It receives as inputs the program L_p, an initial value $V^{(0)}$ and the discount factor γ and parameter ϵ for the convergence test. It outputs the final approximation of value function V and the optimal policy π. Note that in line 10 the weighted model counting mechanism is invoked to compute the success probabilities of all atoms with associated utilities and in line 19 the utility attribute of nodes $\_sj\_$ are updated accordingly.

Algorithm 1: VI-MDP-PROBLOG($L_p, V^{(0)}, \gamma, \epsilon$)

```
1  L'_p ← PREPROCESS(L_p),  formulae ← COMPILE(L'_p)
2  V ← V^(0),  π ← NIL
3  while true do
4      foreach val(x_1, ..., x_n) do
5          s ← val(x_1, ..., x_n)
6          bestValue ← −∞,  bestAction ← NIL
7          foreach a ∈ A do
8              weights ← EVIDENCE(s, a)
9              score ← 0
10             foreach (u_i, p_i) ∈ EVAL(formulae, weights) do
11                 score ← score + p_i U(u_i)
12             end
13             if bestValue < score then
14                 bestValue ← score,  bestAction ← a
15             end
16         end
17         error(s) ← |bestValue − V(s)|
18         V(s) ← bestValue,  π(s) ← bestAction
19         U(s) ← γV(s)
20     end
21     if max(error) ≤ ε(1 − γ)/(2γ) then
22         break
23     end
24 end
25 return V, π
```

IV. EXPERIMENTAL RESULTS

We tested our implementation in a 2.4 GHz Intel Core i5 4GB RAM machine. Our goals with the experiments are twofold: (i) to empirically validate the theoretic equivalence between MDP-PROBLOG and Bellman's backup for acyclic and cyclic programs; and (ii) to establish a performance baseline for future developments of our framework.

As to confirm the correctness of our implementation, we ran MDP-PROBLOG on particular instances of the `sysadmin` [13] problem with a star topology (i.e., all computers are connected to a central computer), for which the optimal solutions always involve trying to maximize the running time of the central computer in the network, and therefore are easy to manually check. Figure 8 shows an example of the convergence of the VI-MDP-PROBLOG algorithm for a simple problem used for validation.

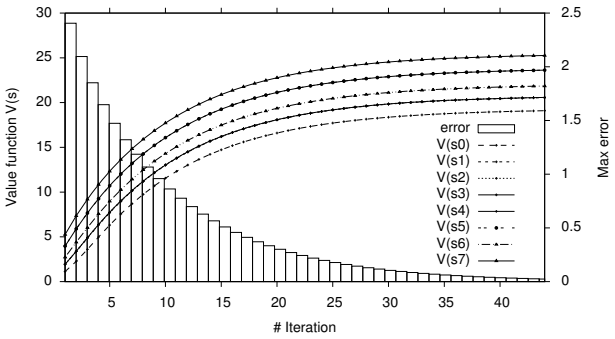

Fig. 8. VI-MDP-PROBLOG convergence for the `sysadmin` problem with 3 computers in a star topology. The problem has 8 states, but due to context-sensitive independencies between state variables derived from the symmetry of the network, 3 curves are overwritten and therefore only 5 curves are visible.

Additionally, we ran our current implementation against 4 different models of the `viral marketing` [12] problem.[4] We used the same network over all models but change the transition rules and state fluents to consider an increasing complexity in the transition function. Models 2 and 4 handle the atoms `buys(P,0)` as part of state representation and add transitions that increases the chances of buying the product in the next step if already bought it previously. Models 1 and 2 consider as valid actions only to market to single individuals at a time. We report in Table I the execution times as well as a problem characterization in terms of number of states, number of actions and maximum number of calls to weight model counting per state/action.

TABLE I
RESULTS FOR THE `VIRAL MARKETING` PROBLEM

model	# states	# actions	# WMC	total (s)	per iter. (s)
1	16	5	24	0.788	0.019
2	256	5	264	95.686	2.225
3	16	16	36	7.683	0.183
4	256	16	276	585.061	13.297

V. RELATED WORK

Our approach relates to previous works on probabilistic programming based on logic languages. The syntactical structure of our language as well as the inference mechanisms we use come directly from PROBLOG [8]. However, by defining special-purpose predicates and restricting the overall form of programs to state distribution programs, we provide a more clear semantics to represent and solve MDPs.

Moreover, in contrast to another decision-theoretic extension, namely DTPROBLOG [9], which only solves episodic

[4]All models are available in the public repository.

(i.e., one-shot) decision problems, our framework addresses the task of sequential decision problems required to solve MDPs. In principle, one can attempt to encode in DT-PROBLOG, perhaps in a mixture of models, a sequential decision problem, but it is likely that it would be memory and/or time-consuming to solve infinite-horizon MDPs directly using its inference engine due to combinatorial explosion of state and actions to consider simultaneously. Alternatively, other direct extensions of PROBLOG to solve sequential decision problems have been attempted. It is worthy to mention a preliminary work [16] which attempts to solve MDPs by means of parameter learning in an online planning setting. Yet, it falls short from our approach since it disregards the reward model and therefore does not find optimal policies for the MDP. Another promising proposal of a probabilistic logic programming system dedicated to solving MDPs uses the language of Dynamic Distributional Clauses (DDC) [6]. This work is much more general in the sense that its main objective is to handle problems with uncountable domains involving mixtures of discrete and continuous variables. In the restricted case of boolean variables, one can easily verify the existence of an homomorphism between DDC language and MDP-PROBLOG (Section 4.2 [17]). Nevertheless, an important difference still resides between the systems: DDC uses as its default inference mechanism importance sampling and Monte-Carlo methods to solve finite-horizon problems whereas MDP-PROBLOG uses state-of-the-art weighted model counting techniques [11] to solve infinite-horizon problems.

In a different perspective, MDP-PROBLOG radically differs from other representation formalisms such as Bayesian networks for probabilistic modeling and PPDDL [2] and RDDL [1] for probabilistic planning. In allowing to encode symmetric and transitive probabilistic dependencies expressed by (stratified) cyclic programs, MDP-PROBLOG is able to represent a broader class of inference and planning problems.

VI. CONCLUSION

We presented a novel framework for representing and solving infinite-horizon MDPs by probabilistic programming. We showed how to extend a probabilistic version of Prolog to compactly represent the logical and probabilistic structure of planning domains. In particular, our techniques are useful to handle rich domains with symmetric and transitive probabilistic dependencies between ground predicates that cannot be modeled straightforwardly with traditional formalisms.

In this work, we only considered a simple value iteration scheme for solving the MDP problem, nonetheless PROBLOG also allows probabilistic sampling and parameter learning. This can enable more sophisticated approaches such as Real-Time Dynamic Programming (RTDP) [18] and planning as inference using Expectation-Maximization (EM) [19].

On a rather different note, another very interesting possibility for future work is to allow more than one stable model per induced logic program. This is in direct relation with more expressive models such as MDP-ST problems [20] and might trigger a considerable change in the language semantics. Finally, one interesting idea we are currently investigating is to use logical inference allowed by our underlying logic mechanisms to reduce the space of policy search and to accelerate the convergence of dynamic programming.

ACKNOWLEDGMENT

This work was partially supported by CNPq (grants 870666/1998-3, 308433/2014-9) and FAPESP (grants 2015/01587-0, 2016/01055-1).

REFERENCES

[1] S. Sanner, "Relational dynamic influence diagram language (RDDL): Language description," 2010, http://users.cecs.anu.edu.au/ ssanner/IPPC_2011/RDDL.pdf.

[2] H. L. Younes and M. L. Littman, "PPDDL1.0: An extension to PPDDL for expressing planning domains with probabilistic effects," in *Proceedings of the 14th International Conference on Automated Planning and Scheduling*, 2004.

[3] T. Sato and Y. Kameya, "PRISM: a language for symbolic-statistical modeling," in *IJCAI*, vol. 97, 1997, pp. 1330–1339.

[4] D. Poole, "The independent choice logic and beyond," in *Probabilistic inductive logic programming*. Springer, 2008, pp. 222–243.

[5] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov, "BLOG: Probabilistic models with unknown objects," *Statistical relational learning*, p. 373, 2007.

[6] D. Nitti, V. Belle, and L. De Raedt, "Planning in discrete and continuous markov decision processes by probabilistic programming," in *Machine Learning and Knowledge Discovery in Databases*. Springer, 2015, pp. 327–342.

[7] V. S. Costa, D. Page, M. Qazi, and J. Cussens, "CLP (BN): Constraint logic programming for probabilistic knowledge," in *Proceedings of the Nineteenth conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann Publishers Inc., 2002, pp. 517–524.

[8] L. De Raedt, A. Kimmig, and H. Toivonen, "PROBLOG: A probabilistic prolog and its application in link discovery." in *IJCAI*, vol. 7, 2007, pp. 2462–2467.

[9] G. Van den Broeck, I. Thon, M. Van Otterlo, and L. De Raedt, "DTPROBLOG: A decision-theoretic probabilistic prolog," in *Proceedings of the twenty-fourth AAAI conference on artificial intelligence*. AAAI Press, 2010, pp. 1217–1222.

[10] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[11] D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt, "Inference and learning in probabilistic logic programs using weighted boolean formulas," *Theory and Practice of Logic Programming*, vol. 15, no. 03, pp. 358–401, 2015.

[12] P. Domingos and M. Richardson, "Mining the network value of customers," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2001, pp. 57–66.

[13] C. E. Guestrin, "Planning under uncertainty in complex structured environments," Ph.D. dissertation, Stanford University, 2003.

[14] T. Sato, "A statistical learning method for logic programs with distribution semantics," in *Proceedings of the 12th International Conference on Logic Programming (ICLP'95*. Citeseer, 1995.

[15] D. Poole, "Probabilistic programming languages: Independent choices and deterministic systems," *Heuristics, probability and causality: A tribute to Judea Pearl*, pp. 253–269, 2010.

[16] I. Thon, B. Gutmann, and G. Van den Broeck, "Probabilistic programming for planning problems," 2010.

[17] L. De Raedt and A. Kimmig, "Probabilistic (logic) programming concepts," *Machine Learning*, vol. 100, no. 1, pp. 5–47, 2015.

[18] A. G. Barto, S. J. Bradtke, and S. P. Singh, "Learning to act using real-time dynamic programming," *Artificial Intelligence*, vol. 72, no. 1, pp. 81–138, 1995.

[19] M. Toussaint and A. Storkey, "Probabilistic inference for solving discrete and continuous state markov decision processes," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 945–952.

[20] F. W. Trevizan, F. G. Cozman, and L. N. D. Barros, "Planning under risk and knightian uncertainty," in *IJCAI-07*, 2007.