Planning in Discrete and Continuous Markov Decision Processes by Probabilistic Programming

Davide Nitti(✉), Vaishak Belle, and Luc De Raedt

Department of Computer Science, KU, Leuven, Belgium
{davide.nitti,vaishak.belle,luc.deraedt}@cs.kuleuven.be

Abstract. Real-world planning problems frequently involve mixtures of continuous and discrete state variables and actions, and are formulated in environments with an unknown number of objects. In recent years, probabilistic programming has emerged as a natural approach to capture and characterize such complex probability distributions with general-purpose inference methods. While it is known that a probabilistic programming language can be easily extended to represent Markov Decision Processes (MDPs) for planning tasks, solving such tasks is challenging. Building on related efforts in reinforcement learning, we introduce a conceptually simple but powerful planning algorithm for MDPs realized as a probabilistic program. This planner constructs approximations to the optimal policy by importance sampling, while exploiting the knowledge of the MDP model. In our empirical evaluations, we show that this approach has wide applicability on domains ranging from strictly discrete to strictly continuous to hybrid ones, handles intricacies such as unknown objects, and is argued to be competitive given its generality.

1 Introduction

Real-world planning problems frequently involve mixtures of continuous and discrete state variables and actions. Markov Decision Processes (MDPs) [28] are a natural and general framework for modeling such problems. However, while significant progress has been made in developing robust planning algorithms for discrete and continuous MDPs, the more intricate hybrid (i.e., mixtures) domains and settings with an unknown number of objects have received less attention.

The recent advances of probabilistic programming languages (e.g., BLOG [15], Church [6], ProbLog [10], distributional clauses [7]) has significantly improved the expressive power of formal representations for probabilistic models. While it is known that these languages can be extended for decision problems [27,29], including MDPs, it is less clear if the inbuilt general-purpose inference system can cope with the challenges (e.g., scale, time constraints) posed by actual planning problems, and compete with existing state-of-the-art planners.

In this paper, we consider the problem of effectively planning in domains where reasoning and handling unknowns may be needed in addition to coping with mixtures of discrete and continuous variables. In particular, we adopt

A. Appice et al. (Eds.): ECML PKDD 2015, Part II, LNAI 9285, pp. 327–342, 2015.
DOI: 10.1007/978-3-319-23525-7_20

dynamic distributional clauses (DDC) [17,18] (an extension of distributional clauses for temporal models) to describe the MDP and perform inference. In such general settings, exact solutions may be intractable, and so approximate solutions are the best we can hope for. Popular approximate solutions include Monte-Carlo methods to estimate the expected reward of a policy (i.e., policy evaluation). Monte-Carlo methods provide state-of-the-art results in probabilistic planners [9,11]. Monte-Carlo planners have been mainly applied in discrete domains (with some notable exceptions, such as [1,13], for continuous domains). Typically, for continuous states, function approximation (e.g., linear regression) is applied. In that sense, one of the few Monte-Carlo planners that works in arbitrary MDPs with no particular assumptions is Sparse Sampling (SST) [8]; but as we demonstrate later, it is often slow in practice. We remark that most, if not all, Monte-Carlo methods require only a way to sample from the model of interest. While this property seems desirable, it prevents us from exploiting the actual probabilities of the model, as discussed (but unaddressed) in [9].

In this work, we introduce HYPE: a conceptually simple but powerful planning algorithm for a given MDP in DDC. However, HYPE can be adapted for other languages, such as RDDL [22]. The proposed planner exploits the knowledge of the model via importance sampling to perform policy evaluation, and thus, policy improvement. Importance sampling has been used in off-policy Monte-Carlo methods [20,24,25], where policy evaluation is performed using trajectories sampled from another policy. We remark that standard off-policy Monte-Carlo methods have been used in reinforcement learning, which are essentially model-free settings. In our setting, given a planning domain, the proposed planner introduces a new off-policy method that exploits the model and works, under weak assumptions, in discrete, continuous, hybrid domains as well as those with an unknown number of objects.

We provide a detailed derivation on how the approximation is obtained using importance sampling. Most significantly, we test the robustness of the approach on a wide variety of probabilistic domains. Given the generality of our framework, we do not challenge the plan times of state-of-the-art planners, but we do successfully generate meaningful plans in all these domains. We believe this performance is competitive given the algorithm's applicability. Indeed, the results show that our system at best outperforms SST [8] and at worst produces similar results, where SST is an equally general planner; in addition, it obtains reasonable results with respect to state-of-the-art discrete (probabilistic) planners.

2 Preliminaries

In a MDP, a putative agent is assumed to interact with its environment, described using a set S of *states*, a set A of *actions* that the agent can perform, a *transition function* $p : S \times A \times S \to [0, 1]$, and a *reward function* $R : S \times A \to \mathbb{R}$. That is, when in state s and on doing a, the probability of reaching s' is given by $p(s' \mid s, a)$, for which the agent receives the reward $R(s, a)$. The agent is taken to operate over a finite number of time steps $t = 0, 1, \ldots, T$, with the goal of maximizing the expected reward: $\mathbb{E}[\sum_{t=0}^{T} \gamma^t R(s_t, a_t)]$, where s_0 is the start state, a_0 the first action, and $\gamma \in [0, 1]$ is a discount factor.

This paper focuses on maximizing the reward in a finite horizon MDP; however the same ideas are extendable for infinite horizons. This is achieved by computing a (deterministic) policy $\pi : S \times D \rightarrow A$ that determines the agent's action at state s and remaining steps d (horizon). The expected reward starting from state s_t and following a policy π is called the *value function* (V-function):

$$V_d^{\pi}(s_t) = \mathbb{E}\left[\sum_{k=t}^{t+d} \gamma^{k-t} R(s_k, a_k) \mid s_t, \pi\right]. \tag{1}$$

Furthermore, the expected reward starting from state s_t while executing action a_t and following a policy π is called the *action-value function* (Q-function):

$$Q_d^{\pi}(s_t, a_t) = \mathbb{E}\left[\sum_{k=t}^{t+d} \gamma^{k-t} R(s_k, a_k) \mid s_t, a_t, \pi\right]. \tag{2}$$

Since $T = t + d$, in the following formulas we will use T for compactness. An *optimal policy* π^* is a policy that maximizes the V-function for all states. A sample-based planner uses Monte-Carlo methods to solve an MDP and find a (near) optimal policy. The planner simulates (by sampling) interaction with the environment in episodes $E^m = <s_0^m, a_0^m, s_1^m, a_1^m, ..., s_T^m, a_T^m>$, following some policy π. Each episode is a trajectory of T time steps, and we let s_t^m denote the state visited at time t during episode m. (So, after M episodes, $M \times T$ states would be explored). After or during an episode generation, the sample-based planner updates $Q_d(s_t^m, a_t^m)$ for each t according to a backup rule, for example, averaging the total rewards obtained starting from (s_t^m, a_t^m) till the end. The policy is improved using a strategy that trades-off exploitation and exploration, e.g., the ϵ-greedy strategy. In this case the policy used to sample the episodes is not deterministic; we indicate with $\pi(a_t|s_t)$ the probability to select action a_t in state s_t under the policy π. Under certain conditions, after a sufficiently large number of episodes, the policy converges to a (near) optimal policy, and the planner can execute the greedy policy $argmax_a Q_d(s, a)$.

3 Dynamic Distributional Clauses

We assume some familiarity with standard terminology of statistical relational learning and logic programming [2]. We represent the MDP using *dynamic distributional clauses* [7,17], an extension of logic programming to represent continuous and discrete random variables. A *distributional clause* (DC) is of the form $\mathtt{h} \sim \mathcal{D} \leftarrow \mathtt{b_1}, \ldots, \mathtt{b_n}$, where the $\mathtt{b_i}$ are literals and $\sim$ is a binary predicate written in infix notation. The intended meaning of a distributional clause is that each ground instance of the clause $(\mathtt{h} \sim \mathcal{D} \leftarrow \mathtt{b_1}, \ldots, \mathtt{b_n})\theta$ defines the random variable $\mathtt{h}\theta$ as being distributed according to $\mathcal{D}\theta$ whenever all the $\mathtt{b_i}\theta$ hold, where θ is a substitution. Furthermore, a term $\simeq(d)$ constructed from the reserved functor $\simeq/1$ represents the value of the random variable d.

Example 1. Consider the following clauses:

$$\mathtt{n} \sim \mathtt{poisson}(6). \tag{3}$$

$$\mathtt{pos}(\mathtt{P}) \sim \mathtt{uniform}(1, 10) \leftarrow \mathtt{between}(1, \simeq(\mathtt{n}), \mathtt{P}). \tag{4}$$

$$\mathtt{left}(\mathtt{A}, \mathtt{B}) \leftarrow \simeq(\mathtt{pos}(\mathtt{A})) > \simeq(\mathtt{pos}(\mathtt{B})). \tag{5}$$

Capitalized terms such as $\mathtt{P}, \mathtt{A}$ and $\mathtt{B}$ are logical variables, which can be substituted with any constant. Clause (3) states that the number of people $\mathtt{n}$ is governed by a Poisson distribution with mean 6; clause (4) models the position $\mathtt{pos}(\mathtt{P})$ as a random variable uniformly distributed from 1 to 10, for each person $\mathtt{P}$ such that $\mathtt{P}$ is between 1 and $\simeq(\mathtt{n})$. Thus, if the outcome of $\mathtt{n}$ is two (i.e., $\simeq(n) = 2$) there are 2 independent random variables $\mathtt{pos}(1)$ and $\mathtt{pos}(2)$. Finally, clause (5) shows how to define the predicate $\mathtt{left}(\mathtt{A}, \mathtt{B})$ from the positions of any $\mathtt{A}$ and $\mathtt{B}$. Ground atoms such as $\mathtt{left}(1, 2)$ are binary random variables that can be true or false, while terms such as $\mathtt{pos}(1)$ represent random variables that can take concrete values from the domain of their distribution.

A *distributional program* is a set of distributional clauses (some of which may be deterministic) that defines a distribution over possible worlds, which in turn defines the underlying semantics. A possible world is generated starting from the empty set $S = \emptyset$; for each distributional clause $\mathtt{h} \sim \mathcal{D} \leftarrow \mathtt{b}_1, ..., \mathtt{b}_\mathtt{n}$, whenever the body $\{\mathtt{b}_1\theta, ..., \mathtt{b}_\mathtt{n}\theta\}$ is true in the set S for the substitution θ, a value v for the random variable $\mathtt{h}\theta$ is sampled from the distribution $\mathcal{D}\theta$ and $\simeq(h\theta) = v$ is added to S. This is repeated until a fixpoint is reached, i.e., no further variables can be sampled. *Dynamic distributional clauses* (DDC) extend distributional clauses in admitting temporally-extended domains by associating a time index to each random variable.

Example 2. Let us consider an object search scenario (*objsearch*) used in the experiments, in which a robot looks for a specific object in a shelf. Some of the objects are visible, others are occluded. The robot needs to decide which object to remove to find the object of interest. Every time the robot removes an object, the objects behind it become visible. This happens recursively, i.e., each new uncovered object might occlude other objects. The number and the types of occluded objects depend on the object covering them. For example, a box might cover several objects because it is big. This scenario involves an unknown number of objects and can be written as a partially observable MDP. However, it can be also described as a MDP in DDC where the state is the type of visible objects; in this case the state grows over time when new objects are observed or shrink when objects are removed without uncovering new objects. The probability of observing new objects is encoded in the state transition model, for example:

$$\mathtt{type}(\mathtt{X})_{\mathtt{t+1}} \sim \mathtt{val}(\mathtt{T}) \leftarrow \simeq(\mathtt{type}(\mathtt{X})_\mathtt{t}) = \mathtt{T},\ \mathtt{not}(\mathtt{removeObj}(\mathtt{X})). \tag{6}$$

$$\mathtt{numObjBehind}(\mathtt{X})_{\mathtt{t+1}} \sim \mathtt{poisson}(1) \ \leftarrow \simeq(\mathtt{type}(\mathtt{X})_\mathtt{t}) = \mathtt{box}, \mathtt{removeObj}(\mathtt{X}). \tag{7}$$

$$\begin{aligned}&\mathtt{type}(\mathtt{ID})_{\mathtt{t+1}} \sim \mathtt{finite}([0.2:\mathtt{glass}, 0.3:\mathtt{cup}, 0.4:\mathtt{box}, 0.1:\mathtt{can}]) \leftarrow\\ &\quad \simeq(\mathtt{type}(\mathtt{X})_\mathtt{t}) = \mathtt{box}, \mathtt{removeobj}(\mathtt{X}), \simeq(\mathtt{numObjBehind}(\mathtt{X})_{\mathtt{t+1}}) = \mathtt{N}, \mathtt{getLastID}(\mathtt{Last})_\mathtt{t},\\ &\quad \mathtt{NewID}\ \mathtt{is}\ \mathtt{Last} + 1, \mathtt{EndNewID}\ \mathtt{is}\ \mathtt{NewID} + \mathtt{N}, \mathtt{between}(\mathtt{NewID}, \mathtt{EndNewID}, \mathtt{ID}).\end{aligned} \tag{8}$$

Clause (6) states that the type of each object remains unchanged when we do not perform a remove action. Otherwise, if we remove the object, its type is removed from the state at time $t+1$ because it is not needed anymore. Clauses (7) and (8) define the number and the type of objects behind a box `X`, added to the state when we perform a remove action on `X`. Similar clauses are defined for other types. The predicate $\mathtt{getLastID(Last)_t}$ returns the highest object `ID` in the state and is needed to make sure that any new object has a different `ID`.

To complete the MDP specification we need to define a reward function $R(s_t, a_t)$, the terminal states that indicate when the episode terminates, and the applicability of an action a_t is a state s_t as in PDDL. For *objsearch* we have:

$$\mathtt{stop_t} \leftarrow \simeq(\mathtt{type(X)_t}) = \mathtt{can}.$$
$$\mathtt{reward(20)_t} \leftarrow \mathtt{stop_t}.$$
$$\mathtt{reward(-1)_t} \leftarrow \mathtt{not(stop_t)}.$$

That is, a state is terminal when we observe the object of interest (e.g., a can), for which a reward of 20 is obtained. The remaining states are nonterminal with reward -1. To define action applicability we use a set of clauses of the form

$$\mathtt{applicable(action)_t} \leftarrow \mathtt{preconditions_t}.$$

For example, action `removeobj` is applicable for each object in the state, that is when its type is defined with an arbitrary value `Type`:

$$\mathtt{applicable(removeobj(X))_t} \leftarrow \simeq(\mathtt{type(X)_t}) = \mathtt{Type}.$$

4 Planning by Importance Sampling

Our approach to plan in MDPs described in DDC is an *off-policy strategy* [28] based on importance sampling and *derived* from the transition model. Related work is discussed more comprehensively in Section 5, but as we note later, sample-based planners typically only require a generative model (a way to generate samples) and do not exploit the declarative model of the MDP (i.e., the actual probabilities) [9]. In our case, this knowledge leads to an effective planning algorithm that works in discrete, continuous, hybrid domains, and domains with an unknown number of objects under weak assumptions.

In a nutshell, the proposed approach samples episodes E^m and stores for each visited state s_t^m an estimation of the V-function (e.g., the total reward obtained from that state). The action selection follows an ϵ-greedy strategy, where the Q-function is estimated as the immediate reward plus the weighted average of the previously stored V-function points at time $t+1$. This is justified by the means of importance sampling as explained later. The essential steps of our planning system HYPE (= *hybrid episodic planner*) are given in Algorithm 1.

The algorithm realizes the following key ideas:

- $\tilde{Q}$ and $\tilde{V}$ denote approximations of the Q and V-function respectively.

Algorithm 1. Hype

```
 1: function SampleEpisode($d, s_t^m, m$)          ▷ Horizon $d$, state $s_t^m$ in episode $m$
 2:     if $d = 0$ then
 3:         return 0
 4:     end if
 5:     for each applicable action $a$ in $s_t^m$ do          ▷ $Q$-function estimation
 6:         $\tilde{Q}_d^m(s_t^m, a) \leftarrow R(s_t^m, a) + \gamma \frac{\sum_{i=0}^{m-1} w^i \tilde{V}_{d-1}^i(s_{t+1}^i)}{\sum_{i=0}^{m-1} w^i}$
 7:     end for
 8:     sample $u \sim uniform(0, 1)$          ▷ $\epsilon$-greedy strategy
 9:     if $u < 1 - \epsilon$ then
10:         $a_t^m \leftarrow argmax_a \tilde{Q}_d^m(s_t^m, a)$
11:     else
12:         $a_t^m \sim uniform(actions\ applicable\ in\ s_t^m)$
13:     end if
14:     sample $s_{t+1}^m \sim p(s_{t+1} \mid s_t^m, a_t^m)$          ▷ sample next state
15:     $G_d^m \leftarrow R(s_t^m, a_t^m) + \gamma\ \cdot$SampleEpisode $(d - 1, s_{t+1}^m, m)$          ▷ recursive call
16:     $\tilde{V}_d^m(s_t^m) \leftarrow G_d^m$
17:     store $(s_t^m, \tilde{V}_d^m(s_t^m), d)$
18:     return $\tilde{V}_d^m(s_t^m)$          ▷ V-function estimation for $s_t^m$ at horizon $d$
19: end function
```

- Lines 14-17 sample the next step and recursively the remaining episode of total length T, then stores the total discounted reward G_d^m starting from the current state s_t^m. This quantity can be interpreted as a sample of the expectation in formula (1), thus an estimator of the V-function. For this and other reasons explained later, G_d^m is stored as $\tilde{V}_d^m(s_t^m)$.
- Lines 8-13 implement an ϵ-greedy exploratory strategy for choosing actions.
- Most significantly, line 6 approximates the Q-function using the *weighted average* of the stored $\tilde{V}_{d-1}^i(s_{t+1}^i)$ points:

$$\tilde{Q}_d^m(s_t^m, a) \leftarrow R(s_t^m, a) + \gamma \frac{\sum_{i=0}^{m-1} w^i \tilde{V}_{d-1}^i(s_{t+1}^i)}{\sum_{i=0}^{m-1} w^i}, \tag{9}$$

  where w^i is a *weight function* for episode i at state s_{t+1}^i. The weight exploits the transition model and is defined as:

$$w^i = \frac{p(s_{t+1}^i \mid s_t^m, a)}{q(s_{t+1}^i)} \alpha^{(m-i)}. \tag{10}$$

Here, for evaluating an action a at the current state s_t, we let w^i be the ratio of the transition probability of reaching a stored state s_{t+1}^i and the probability used to sample s_{t+1}^i, denoted using q. Recent episodes are considered more significant than previous ones, and so α is a parameter for realizing this. We provide a detailed justification for line 6 below.

We note that line 6 requires us to go over a finite set of actions, and so in the presence of continuous action spaces (e.g., real-valued parameter for a move action), we can discretize the action space or sample from it. More sophisticate approaches are possible [5,26].

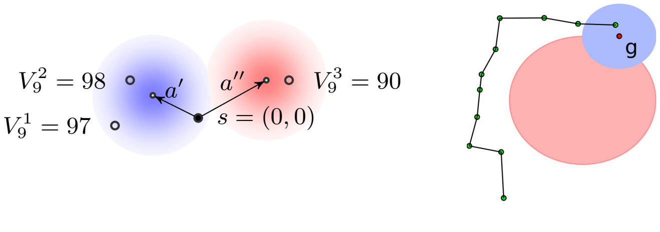

Fig. 1. Left: weight computation for the objpush domain. Right: a sampled episode that reaches the goal (blue), and avoids the undesired region (red).

Example 3. As a simple illustration, consider the following example called *objpush*. We have an object on a table and an arm that can push the object in a set of directions; the goal is to move the object close to a point g, avoiding an undesired region (Fig. 1). The state consists of the object position (x, y), with push actions parameterized by the displacement $(\mathtt{DX}, \mathtt{DY})$. The state transition model is a Gaussian around the previous position plus the displacement:

$$\mathtt{pos(ID)_{t+1}} \sim \mathtt{gaussian}(\simeq\!(\mathtt{pos(ID)_t}) + (\mathtt{DX}, \mathtt{DY}), \mathtt{cov}) \leftarrow \mathtt{push}(\mathtt{ID}, (\mathtt{DX}, \mathtt{DY})). \quad (11)$$

The clause is valid for any object ID; nonetheless, for simplicity, we will consider a scenario with a single object. The terminal states and rewards in DDC are:

$$\begin{aligned} \mathtt{stop_t} &\leftarrow \mathtt{dist}(\simeq\!(\mathtt{pos(A)_t}), (0.6, 1.0)) < 0.1. \\ \mathtt{reward(100)_t} &\leftarrow \mathtt{stop_t}. \\ \mathtt{reward(-1)_t} &\leftarrow \mathtt{not(stop_t)}, \mathtt{dist}(\simeq\!(\mathtt{pos(A)_t}), (0.5, 0.8)) >= 0.2. \\ \mathtt{reward(-10)_t} &\leftarrow \mathtt{not(stop_t)}, \mathtt{dist}(\simeq\!(\mathtt{pos(A)_t}), (0.5, 0.8)) < 0.2. \qquad (12) \end{aligned}$$

That is, a state is terminal when there is an object close to the goal point $(0.6, 1.0)$ (i.e., distance lower than 0.1), and so, a reward of 100 is obtained. The nonterminal states have reward -10 whether inside an undesired region centered in $(0.5, 0.8)$ with radius 0.2, and $R(s_t, a_t) = -1$ otherwise.

Let us assume we previously sampled some episodes of length $T = 10$, and we want to sample the m = 4-th episode starting from $s_0 = (0, 0)$. We compute $\tilde{Q}_{10}^m((0, 0), a)$ for each action a (line 6). Thus we compute the weights w^i using (10) for each stored sample $\tilde{V}_9^i(s_1^i)$. For example, Figure 1 shows the computation of $\tilde{Q}_{10}^m((0, 0), a)$ for action $a' = (-0.4, 0.3)$ and $a'' = (0.9, 0.5)$, where we have three previous samples $i = \{1, 2, 3\}$ at depth 9. A shadow represents the likelihood $p(s_1^i | s_0 = (0, 0), a)$ (left for a' and right for a''). The weight w^i (10) for each sample s_1^i is obtained by dividing this likelihood by $q(s_1^i)$ (with $\alpha = 1$). If $q(s_1^i)$ is uniform over the three samples, sample $i = 2$ with total reward $\tilde{V}_9^2(s_1^2) = 98$

will have higher weight than samples $i = 1$ and $i = 3$. The situation is reversed for a''. Note that we can estimate $\tilde{Q}^m_d(s^m_t, a)$ using episodes i that may never encounter s^m_t, a_t provided that $p(s^i_{t+1}|s^m_t, a_t) > 0$.

Computing the (Approximate) *Q*-Function

The purpose of this section is to motivate our approximation to the Q-function using the weighted average of the V-function points in line 6. Let us begin by expanding the definition of the Q-function from (2) as follows:

$$Q^\pi_d(s_t, a_t) = R(s_t, a_t) + \gamma \int_{s_{t+1:T}, a_{t+1:T}} G_{d-1} p(s_{t+1:T}, a_{t+1:T}|s_t, a_t, \pi) ds_{t+1:T}, a_{t+1:T}, \quad (13)$$

where G_{d-1} is the total (discounted) reward from time $t + 1$ for $d - 1$ steps: $G_{d-1} = \sum_{k=1}^{d-1} \gamma^{k-1} R(s_{t+k}, a_{t+k})$. Given that we sample trajectories from the target distribution $p(s_{t+1:T}, a_{t+1:T}|s_t, a_t, \pi)$, we obtain the following approximation to the Q-function equaling the true value in the sampling limit:

$$Q^\pi_d(s_t, a_t) \approx R(s_t, a_t) + \frac{1}{N} \gamma \sum_i G^i_{d-1}. \quad (14)$$

Policy evaluation can be performed sampling trajectories using another policy, this is called *off-policy* Monte-Carlo [28]. For example, we can evaluate the greedy policy while the data is generated from a randomized one to enable exploration. This is generally performed using (normalized) *importance sampling* [25]. We let w^i be the ratio of the target and proposal distributions to restate the sampling limit as follows:

$$Q^\pi_d(s_t, a_t) \approx R(s_t, a_t) + \frac{1}{\sum w^i} \gamma \sum_i w^i G^i_{d-1}. \quad (15)$$

In standard off-policy Monte-Carlo the proposal distribution is of the form:

$$p(s_{t+1:T}, a_{t+1:T}|s_t, a_t, \pi') = \prod_{k=t}^{T-1} \pi'(a_{k+1}|s_{k+1}) p(s_{k+1}|s_k, a_k)$$

The target distribution has the same form, the only difference is that the policy is π instead of π'. In this case the weight becomes equal to the policy ratio because the transition model cancels out. This is desirable when the model is not available, for example in model-free Reinforcement Learning. The question is whether the availability of the transition model can be used to improve off-policy methods. This paper shows that the answer to that question is positive.

We will now describe the proposed solution. Instead of considering only trajectories that start from s_t, a_t as samples, we consider all sampled trajectories from time t+1 to T. Since we are ignoring steps before t+1, the proposal distribution for sample i is the marginal

$$p(s_{t+1:T}, a_{t+1:T}|s_0, \pi^i) = q(s_{t+1}) \pi^i(a_{t+1}|s_{t+1}) \prod_{k=t+1}^{T-1} \pi^i(a_{k+1}|s_{k+1}) p(s_{k+1}|s_k, a_k),$$

where q is the marginal probability $p(s_{t+1}|s_0, \pi^i)$. To compute $\tilde{Q}_d^m(s_t^m, a)$ we use (15), where the weight w^i (for $0 \le i \le m-1$) becomes the following:

$$\frac{p(s_{t+1}^i|s_t^m, a)\pi^m(a_{t+1}^i|s_{t+1}^i)\prod_{k=t+1}^{T-1}\pi^m(a_{k+1}^i|s_{k+1}^i)p(s_{k+1}^i|s_k^i, a_k^i)}{q(s_{t+1}^i)\pi^i(a_{t+1}^i|s_{t+1}^i)\prod_{k=t+1}^{T-1}\pi^i(a_{k+1}^i|s_{k+1}^i)p(s_{k+1}^i|s_k^i, a_k^i)}$$

$$= \frac{p(s_{t+1}^i|s_t^m, a)}{q(s_{t+1}^i)}\frac{\prod_{k=t}^{T-1}\pi^m(a_{k+1}^i|s_{k+1}^i)}{\prod_{k=t}^{T-1}\pi^i(a_{k+1}^i|s_{k+1}^i)} \qquad (16)$$

$$\approx \frac{p(s_{t+1}^i|s_t^m, a)}{q(s_{t+1}^i)}\alpha^{(m-i)}. \qquad (17)$$

Thus, we obtain line 6 in the algorithm given that $\tilde{V}_{d-1}^i(s_t^i) = G_{d-1}^i$. In our algorithm the target (greedy) policy π^m is not explicitly defined, therefore the policy ratio is hard to compute. We replace the unknown policy ratio with a quantity proportional to $\alpha^{(m-i)}$ where $0 < \alpha \le 1$; thus, formula (16) is replaced with (17). The quantity $\alpha^{(m-i)}$ becomes smaller for an increasing difference between the current episode index m and the i-th episode. Therefore, the recent episodes are weighted (on average) more than the previous ones, as in *recently-weighted average* applied in on-policy Monte-Carlo [28]. This is justified because the policy is improved over time, thus recent episodes should have higher weight.

Since we are performing policy improvement, each episode is sampled from a different policy. It has been shown [20,25] that samples from different distributions can be considered as sampled from a single distribution that is the mixture of the true distributions. Therefore, for a given episode

$$q(s_{t+1}^i) = \frac{1}{m-1}\sum_j p(s_{t+1}^i|s_0, \pi_j) = \frac{1}{m-1}\sum_j \int_{s_t}\int_{a_t} p(s_{t+1}^i|s_t, a_t)p(s_t, a_t|s_0, \pi_j)ds_t da_t$$
$$\approx \frac{1}{m-1}\sum_j p(s_{t+1}^i|s_t^j, a_t^j),$$

where for each j the integral is approximated with a single sample (s_t^j, a_t^j) from the available episodes. Since each episode is sampled from $p(s_{0:T}, a_{0:T}|s_0, \pi_j)$, samples (s_t^j, a_t^j) are distributed as $p(s_t, a_t|s_0, \pi_j)$ and are used in the estimation of the integral.

The likelihood $p(s_{t+1}^i|s_t^m, a)$ is required to compute the weight. This probability can be decomposed using the chain rule, e.g., for a state with 3 variables we have: $p(s_{t+1}^i|s_t^m, a) = p(v_3|v_2, v_1, s_t^m, a)p(v_2|v_1, s_t^m, a)p(v_1|s_t^m, a)$, where $s_{t+1}^i = \{v_1, v_2, v_3\}$. In DDC this is performed evaluating the likelihood of each variable in v_i following the topological order defined in the DDC program. The target and the proposal distributions might be mixed distributions of discrete and continuous random variables; importance sampling can be applied in such distributions as discussed in [19, Chapter 9.8].

To summarize, for each state s_t^m, $Q(s_t^m, a_t)$ is evaluated as the immediate reward plus the weighted average of stored G_{d-1}^i points. In addition, for each state s_t^m the total discounted reward G_d^m is stored. We would like to remark that we can estimate the Q-function also for states and actions that have never been visited, as shown in example 1. This is possible without using function approximations (beyond importance sampling).

Extensions

Our derivation follows a Monte-Carlo perspective, where each stored point is the total discounted reward of a given trajectory: $\tilde{V}_d^m(s_t^m) \leftarrow G_d^m$. However, following the Bellman equation, $\tilde{V}_d^m(s_t^m) \leftarrow max_a \tilde{Q}_d^m(s_t^m, a)$ can be stored instead. The Q estimation formula in line 6 is not affected; indeed we can repeat the same derivation using the Bellman equation and approximate it with importance sampling:

$$Q_d^\pi(s_t, a_t) = R(s_t, a_t) + \gamma \int_{s_{t+1}} V_{d-1}^\pi(s_{t+1}) p(s_{t+1}|s_t, a_t) ds_{t+1}$$
$$\approx R(s_t, a) + \gamma \sum \frac{w^i}{\sum w^i} \tilde{V}_{d-1}^i(s_{t+1}^i) = \tilde{Q}_d^m(s_t, a_t), \qquad (18)$$

with $w^i = \frac{p(s_{t+1}^i|s_t,a_t)}{q(s_{t+1}^i)}$ and s_{t+1}^i the state sampled in episode i for which we have an estimation of $\tilde{V}_{d-1}^i(s_{t+1}^i)$, while $q(s_{t+1}^i)$ is the probability with which s_{t+1}^i has been sampled. This derivation is valid for a fixed policy π; for a changing policy we can make similar considerations to the previous approach and add the term $\alpha^{(m-i)}$. If we choose $\tilde{V}_{d-1}^i(s_{t+1}^i) \leftarrow G_{d-1}^i$, we obtain the same result as in (9) and (17) for the Monte-Carlo approach. Instead of choosing between the two approaches we can use a linear combination, i.e., we replace line 16 with $\tilde{V}_d^m(s_t^m) \leftarrow \lambda G_d^m + (1-\lambda) max_a \tilde{Q}_d^m(s_t^m, a)$. The analysis from earlier applies by letting $\lambda = 1$. However, for $\lambda = 0$, we obtain a local value iteration step, where the stored $\tilde{V}$ is obtained maximizing the estimated $\tilde{Q}$ values. Any intermediate value balances the two approaches (this is similar to, and inspired by, TD(λ) [28]). Another strategy consists in storing the maximum of the two: $\tilde{V}_d^m(s_t^m) \leftarrow max(G_d^m, max_a \tilde{Q}_d^m(s_t^m, a))$. In other words, we alternate Monte-Carlo and Bellman backup according to which one has the highest value. This strategy works often well in practice; indeed it avoids a typical issue in Monte Carlo methods: bad policies or exploration lead to low rewards, averaged in the estimated Q/V-function. For this reason it may occur that optimal actions are rarely chosen. The mentioned strategy avoids this, and a high ϵ value (line 9) is possible without affecting the performance.

5 Related Work

There is an extensive literature on MDP planners, we will focus mainly on Monte-Carlo approaches. The most notable sample-based planners include Sparse Sampling (SST) [8], UCT [11] and their variations. SST creates a lookahead tree of depth D, starting from state s_0. For each action in a given state, the algorithm samples C times the next state. This produces a near-optimal solution with theoretical guarantees. In addition, this algorithm works with continuous and discrete domains with no particular assumptions. Unfortunately, the number of samples grows exponentially with the depth D, therefore the algorithm is extremely slow in practice. Some improvements have been proposed [31], although the worst-case performance remains exponential. UCT [11] uses *upper confidence bound* for multi-armed bandits to trade off between exploration and exploitation in the tree search, and inspired successful Monte-Carlo tree search methods. Instead of building the full tree, UCT chooses the action a that maximizes an upper confidence bound of $Q(s, a)$, following the principle of optimism in the face of uncertainty. Several improvements and extensions for UCT have been proposed, including handling continuous actions [13] (see [16] for a review), and continuous states [1] with a simple Gaussian distance metric; however the knowledge of the probabilistic model is not directly exploited. For continuous states, parametric function approximation is often used (e.g., linear regression), nonetheless the model needs to be carefully tailored for the domain to solve [32].

There exist algorithms that exploit instance-based methods (e.g. [3,5,26]) for model-free reinforcement learning. They basically store Q-point estimates, and then use e.g., neighborhood regression to evaluate $Q(s, a)$ given a new point (s, a). While these approaches are effective in some domains, they require the user to design distance metric that takes into account the domain. This is straightforward in some cases (e.g., in Euclidean spaces), but it might be harder in others. We argue that the knowledge of the model can avoid (or simplify) the design of a distance metric in several cases, where the importance sampling weights and the transition model, can be considered as a kernel.

The closest related works include [20,21,24,25], they use importance sampling to evaluate a policy from samples generated with another policy. Nonetheless, they adopt importance sampling differently without the knowledge of the MDP model. Although this property seems desirable, the availability of the actual probabilities cannot be exploited, apart from sampling, in their approaches. The same conclusion is valid for practically any sample-based planner, which only needs a sample generator of the model. The work of [9] made a similar statement regarding PROST, a state-of-the-art discrete planner based on UCT, without providing a way to use the state transition probabilities directly. Our algorithm tries to alleviate this, exploiting the probabilistic model in a sample-based planner via importance sampling.

For more general domains that contain discrete and continuous (hybrid) variables several approaches have been proposed under strict assumptions. For example, [23] provide exact solutions, but assume that continuous aspects of the transition model are deterministic. In a related effort [4], hybrid MDPs are solved using dynamic programming, but assuming that transition model and reward is piecewise constant or linear. Another planner HAO* [14] uses heuristic search to find an optimal plan in hybrid domains with theoretical guarantees. However, they assume that the same state cannot be visited again (i.e., they assume plans do not have loops, as discussed in [14, sec.5]), and they rely on the availability of methods to solve the integral in the Bellman equation related to the continuous part of the state. Visiting the same state in our approach is a benefit and not a limit; indeed a previous visited state s' is useful to evaluate $Q_d(s, a)$, when the weight is positive (i.e., when s' is reachable from s with action a).

There exists several languages specific for planning, the most recent is RDDL [22]. A RDDL domain can be mapped in DDC and solved with HYPE. Nonetheless, RDDL does not support a state space with an unknown number of variables as in Example 2. Some planners are based on probabilistic logic programming, for example DTProbLog [29] and PRADA [12], though they only support discrete action-state spaces. For domains with an unknown number of objects, some probabilistic programming languages such as BLOG [15], Church [6], and DC [7] can cope with such uncertainty. To the best of our knowledge DTBLOG [27] and [30] are the only proposals that are able to perform decision making in such domains using a POMDP framework. Furthermore, BLOG is one of the few languages that explicitly handles data association and identity uncertainty. The proposed paper does not focus on POMDP, nor on identity uncertainty; however, interesting domains with unknown number of objects can be easily described as an MDP that HYPE can solve.

Among the mentioned sample-based planners, one of the most general is SST, which does not make any assumption on the state and action space, and only relies on Monte-Carlo approximation. In addition, it is one of the few planners that can be easily applied to any DDC program, including MDPs with an unknown number of objects. For this reason SST was implemented for DDC and used as baseline for our experiments.

6 Experiments

This section answers the following questions: (Q1) Does the algorithm obtain the correct results? (Q2) How is the performance of the algorithm in different domains? (Q3) How does it compare with state-of-the-art planners?

The algorithm was implemented in YAP Prolog and C++, and run on a Intel Core i7 Desktop.

To answer (Q1) we tested the algorithm on a nonlinear version of the hybrid mars rover domain (called *simplerover1*) described in [23] for which the exact V-function is available (depth $d = 3$ and 2 variables: a two-dimensional continuous position and one discrete variable to indicate if the picture was taken). We choose 31 initial points and ran the algorithm for 100 episodes each. Each point took on average $1.4s$. Fig. 2 shows the results where the line is the exact V, and dots are estimated V points. The results show that the algorithm converges to the optimal V-function with a negligible error. This domain is deterministic, and so, to make it more realistic we converted it to a probabilistic MDP adding Gaussian noise to the state transition model. The resulting MDP (*simplerover2*) is hard (if not impossible) to solve exactly. Then we performed experiments for different horizons, number of pictures points (1 to 4, each one is a discrete variable) and summed the rewards. For each instance the planner searches for an optimal policy and executes it, and after each executed action it samples additional episodes to refine the policy (replanning). The proposed planner is compared with SST

Table 1. Experiments: d is the horizon used by the planner, T the total number of steps, M is the maximum number of episodes sampled for HYPE, while C is the SST parameter (number of samples for each state and action). Time limit of $1800s$ per instance. PROST results refer to IPPC2011.

	Domain	game1	game2	sysadmin1	sysadmin2
Planner		$T = 40$	$T = 40$	$T = 40$	$T = 40$
HYPE	reward	0.87 ± 0.11	$\mathbf{0.77 \pm 0.22}$	0.94 ± 0.07	$\mathbf{0.87 \pm 0.11}$
	time (s)	622	608	422	475
	param	$M = 1200$	$M = 1200$	$M = 1200$	$M = 1200$
		$d = 5$	$d = 5$	$d = 5$	$d = 5$
SST	reward	0.34 ± 0.15	0.14 ± 0.20	0.47 ± 0.13	0.31 ± 0.12
	time (s)	986	1000	1068	1062
	param	$C = 1$	$C = 1$	$C = 1$	$C = 1$
		$d = 5$	$d = 5$	$d = 5$	$d = 5$
HYPE	reward	$\mathbf{0.89 \pm 0.07}$	0.76 ± 0.19	$\mathbf{0.98 \pm 0.06}$	0.86 ± 0.11
	time (s)	312	582	346	392
	param	$M = 1200$	$M = 1200$	$M = 1200$	$M = 1200$
		$d = 4$	$d = 4$	$d = 4$	$d = 4$
SST	reward	0.79 ± 0.08	0.27 ± 0.22	0.66 ± 0.08	0.46 ± 0.12
	time (s)	1538	1528	1527	1532
	param	$C = 2$	$C = 2$	$C = 2$	$C = 2$
		$d = 4$	$d = 4$	$d = 4$	$d = 4$
PROST	reward	0.99 ± 0.02	1.00 ± 0.19	1.00 ± 0.05	0.98 ± 0.09

	Domain	objpush	simplerover2	marsrover	objsearch
Planner		$T = 30$	$d = T$	$T = 40$	$d = T$
HYPE	reward	83.7 ± 7.6	11.8 ± 0.2	249.8 ± 33.5	2.53 ± 1.03
	time (s)	472	38	985	13
	param	$M = 4500$	$M = 200$	$M = 6000$	$M = 500$
		$d = 9$	$d = T = 8$	$d = 6$	$d = T = 5$
SST	reward	82.7 ± 2.7	11.4 ± 0.3	227.7 ± 27.3	1.46 ± 1.0
	time (s)	330	48	787	45
	param	$C = 1$	$C = 1$	$C = 1$	$C = 5$
		$d = 9$	$d = T = 8$	$d = 6$	$d = T = 5$
HYPE	reward	86.4 ± 1.0	11.7 ± 0.2	269.0 ± 29.4	$\mathbf{3.64 \pm 1.09}$
	time (s)	1238	195	983	17
	param	$M = 4500$	$M = 500$	$M = 6000$	$M = 600$
		$d = 10$	$d = T = 9$	$d = 7$	$d = T = 5$
SST	reward	82.4 ± 1.9	11.3 ± 0.3	N/A	2.48 ± 1.0
	time (s)	1574	238	timeout	138
	param	$C = 1$	$C = 1$	$C = 1$	$C = 6$
		$d = 10$	$d = T = 9$	$d = 7$	$d = T = 5$
HYPE	reward	$\mathbf{87.5 \pm 0.5}$	$\mathbf{11.9 \pm 0.3}$	$\mathbf{296.3 \pm 19.5}$	3.3 ± 1.6
	time (s)	373	218	1499	20
	param	$M = 2000$	$M = 500$	$M = 4000$	$M = 600$
		$d = 12$	$d = T = 10$	$d = 10$	$d = T = 6$
SST	reward	N/A	11.2 ± 0.3	N/A	0.58 ± 1.4
	time (s)	timeout	1043	timeout	899
	param	$C = 1$	$C = 1$	$C = 1$	$C = 5$
		$d \geq 11$	$d = T = 10$	$d \geq 8$	$d = T = 6$

that requires replanning every step. The results for both planners are always comparable, which confirms the empirical correctness of HYPE (Table 1).

To answer (Q2) and (Q3) we studied the planner in a variety of settings, from discrete, to continuous, to hybrid domains, to those with an unknown number of objects. We performed experiments in a more realistic mars rover domain that is publicly available[1], called *marsrover* (Fig. 2). In this domain we consider one robot and 5 picture points that need to be taken, the movement of the robot causes a negative reward proportional to the displacement and the pictures can be taken only close to the interest point. Each taken picture provides a different reward. Other experiments were performed in the continuous *objpush* MDP described in Section 4 (Fig. 1), and in discrete benchmark domains of the IPPC 2011 competition. In particular, we tested a pair of instances of game of life and sysadmin domains. The results are compared with PROST [9], the IPPC 2011 winner, and shown in table 1 in terms of scores, i.e., the average reward normalizated with respect to IPPC 2011 results; score 1 is the highest result obtained, score 0 is the maximum between the random and the no operation policy.

As suggested by [9], limiting the horizon of the planner increases the performance in several cases. We exploited this idea for HYPE as well as SST (*simplerover2* excluded). For SST we were forced to use small horizons to keep plan time under 30 minutes. In all experiments we followed the IPPC 2011

[1] http://users.cecs.anu.edu.au/~ssanner/IPPC_2014/index.html

schema, that is each instance is repeated 30 times (*objectsearch* excluded), the results are averaged and the 95% confidence interval is computed. However, for every instance we replan from scratch for a fair comparison with SST. In addition, time and number of samples refers to the plan execution of one instance. The results (Table 1) highlight that our planner obtains generally better results than SST, especially at higher horizons. HYPE obtains good results in discrete domains but does not reach state-of-art results (score 1) for two main reasons. The first is the lack of a heuristic, that can dramatically improve the performance, indeed, heuristics are an important component of PROST [9], the IPPC winning planner. The second reason is the time performance that allows us to sample a limited number of episodes and will not allow to finish all the IPPC 2011 domains in 24 hours. This is caused by the expensive Q-function evaluation; however, we are confident that heuristics and other improvements will significantly improve performance and results.

Finally, we performed experiments in the *objectsearch* scenario (Section 3), where the number of objects is unknown. The results are averaged over 400 runs, and confirm better performance for HYPE with respect to SST.

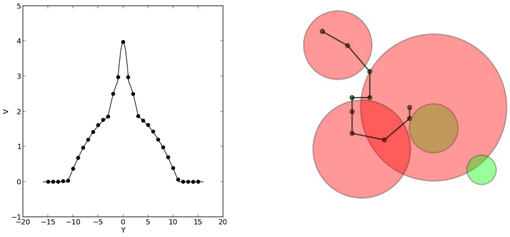

Fig. 2. V-function for different rover positions (with fixed $X = 0.16$) in *simplerover1* domain (left). A possible episode in *marsrover* (right): each picture can be taken inside the respective circle (red if already taken, green otherwise).

7 Practical Improvements

In this section we briefly discuss issues and improvements of HYPE. To evaluate the Q-function the algorithm needs to query all the stored examples, making the algorithm potentially slow. This issue can be mitigated with solutions used in instance-based learning, such as hashing and indexing. For example, in discrete domains we avoid multiple computations of the likelihood and the proposal distribution for samples of the same state. In addition, assuming policy improvement over time, only the N_{store} most recent episodes are kept, since older episodes are generally sampled with a worse policy.

The algorithm HYPE relies on importance sampling to estimate the Q-function, thus we should guarantee that $p > 0 \Rightarrow q > 0$, where p is the target and q is the proposal distribution. This is not always the case, like when we sample the first episode. Nonetheless we can have an indication of the estimation reliability. In our algorithm we use $\sum w^i$ with expectation equals to the number of samples: $\mathbb{E}[\sum w^i] = m$. If $\sum w^i < \mathit{thres}$ the samples available are considered insufficient to compute $Q_d^m(s_t^m, a)$, thus action a can be selected to perform exploration.

A more problematic situation is when, for some action a_t in some state s_t, we always obtain null weights, that is $p(s_{t+1}^i|s_t, a_t) = 0$ for each of the previous episodes i, no matter how many episodes are generated. This issue is solved by adding noise to the state transition model, e.g., Gaussian noise for continuous random variables. This is equivalent to adding a smoothness assumption to the V-function. Indeed the Q-function is a weighted sum of V-function points, where the weights are proportional to a noisy version of the state transition likelihood.

8 Conclusions

We proposed a sample-based planner for MDPs described in DDC under weak assumptions, and showed how the state transition model can be exploited in off-policy Monte-Carlo. The experimental results show that the algorithm produces good results in discrete, continuous, hybrid domains as well as those with an unknown number of objects. Most significantly, it challenges and outperforms SST. For future work, we will consider heuristics and hashing to improve the implementation.

References

1. Couetoux, A.: Monte Carlo Tree Search for Continuous and Stochastic Sequential Decision Making Problems. Université Paris Sud - Paris XI, Thesis (2013)
2. De Raedt, L., Kersting, K.: Probabilistic inductive logic programming. In: De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S.H. (eds.) Probabilistic Inductive Logic Programming. LNCS (LNAI), vol. 4911, pp. 1–27. Springer, Heidelberg (2008)
3. Driessens, K., Ramon, J.: Relational instance based regression for relational reinforcement learning. In: Proc. ICML (2003)
4. Feng, Z., Dearden, R., Meuleau, N., Washington, R.: Dynamic programming for structured continuous Markov decision problems. In: Proc. UAI (2004)
5. Forbes, J., André, D.: Representations for learning control policies. In: Proc. of the ICML Workshop on Development of Representations (2002)
6. Goodman, N., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: A language for generative models. In: Proc. UAI, pp. 220–229 (2008)
7. Gutmann, B., Thon, I., Kimmig, A., Bruynooghe, M., De Raedt, L.: The magic of logical inference in probabilistic programming. Theory and Practice of Logic Programming (2011)
8. Kearns, M., Mansour, Y., Ng, A.Y.: A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes. Machine Learning (2002)
9. Keller, T., Eyerich, P.: PROST: probabilistic planning based on UCT. In: Proc. ICAPS (2012)

10. Kimmig, A., Santos Costa, V., Rocha, R., Demoen, B., De Raedt, L.: On the efficient execution of problog programs. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 175–189. Springer, Heidelberg (2008)
11. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
12. Lang, T., Toussaint, M.: Planning with Noisy Probabilistic Relational Rules. Journal of Artificial Intelligence Research **39**, 1–49 (2010)
13. Mansley, C.R., Weinstein, A., Littman, M.L.: Sample-Based planning for continuous action markov decision processes. In: Proc. ICAPS (2011)
14. Meuleau, N., Benazera, E., Brafman, R.I., Hansen, E.A., Mausam, M.: A heuristic search approach to planning with continuous resources in stochastic domains. Journal of Artificial Intelligence Research **34**(1), 27 (2009)
15. Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D., Kolobov, A.: BLOG: probabilistic models with unknown objects. In: Proc. IJCAI (2005)
16. Munos, R.: From Bandits to Monte-Carlo Tree Search: The Optimistic Principle Applied to Optimization and Planning. Foundations and Trends in Machine Learning, Now Publishers (2014)
17. Nitti, D., De Laet, T., De Raedt, L.: A particle filter for hybrid relational domains. In: Proc. IROS (2013)
18. Nitti, D., De Laet, T., De Raedt, L.: Relational object tracking and learning. In: Proc. ICRA (2014)
19. Owen, A.B.: Monte Carlo theory, methods and examples (2013)
20. Peshkin, L., Shelton, C.R.: Learning from scarce experience. In: Proc. ICML, pp. 498–505 (2002)
21. Precup, D., Sutton, R.S., Singh, S.P.: Eligibility traces for off-policy policy evaluation. In: Proc. ICML (2000)
22. Sanner, S.: Relational Dynamic Influence Diagram Language (RDDL): Language Description (unpublished paper)
23. Sanner, S., Delgado, K.V., de Barros, L.N.: Symbolic dynamic programming for discrete and continuous state MDPs. In: Proc. UAI (2011)
24. Shelton, C.R.: Policy improvement for POMDPs using normalized importance sampling. In: Proc. UAI, pp. 496–503 (2001)
25. Shelton, C.R.: Importance Sampling for Reinforcement Learning with Multiple Objectives. Ph.D. thesis, MIT (2001)
26. Smart, W.D., Kaelbling, L.P.: Practical reinforcement learning in continuous spaces. In: Proc. ICML (2000)
27. Srivastava, S., Russell, S., Ruan, P., Cheng, X.: First-order open-universe POMDPs. In: Proc. UAI (2014)
28. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press (1998)
29. Van den Broeck, G., Thon, I., van Otterlo, M., De Raedt, L.: DTProbLog: a decision-theoretic probabilistic prolog. In: Proc. AAAI (2010)
30. Vien, N.A., Toussaint, M.: Model-Based relational RL when object existence is partially observable. In: Proc. ICML (2014)
31. Walsh, T.J., Goschin, S., Littman, M.L.: Integrating sample-based planning and model-based reinforcement learning. In: Proc. AAAI (2010)
32. Wiering, M., van Otterlo, M.: Reinforcement learning: state-of-the-art. In: Adaptation, Learning, and Optimization. Springer (2012)