Reinforcement Learning for Golog Programs with First-Order State-Abstraction

DANIEL BECK, *Knowledge-Based Systems Group, RWTH Aachen University, Ahornstraße 55, 52074 Aachen, Germany.*
E-mail: beck@kbsg.rwth-aachen.de

GERHARD LAKEMEYER, *Knowledge-Based Systems Group, RWTH Aachen University, Ahornstraße 55, 52074 Aachen, Germany.*
E-mail: gerhard@kbsg.rwth-aachen.de

Abstract

A special feature of programs in the action language Golog are non-deterministic constructs such as non-deterministic choice of actions or arguments. It has been shown that in the presence of stochastic actions and rewards reinforcement learning techniques can be applied to obtain optimal choices for those choice-points. In order to avoid an explosion of the state space an abstraction mechanism is employed that computes first-order state descriptions for the given program. Intuitively, the idea is to generate abstract descriptions that group together states for which the expected reward of executing the program is the same. A current limitation is that a non-deterministic choice of arguments can be handled only if the possible candidates are known in advance. In this paper we show how this restriction can be lifted. We also show how a first-order variant of binary decision diagrams (BDDs) can be used to efficiently compute first-order state abstractions. Moreover, we give a completely declarative specification of a learning Golog interpreter that incorporates the presented state-abstraction mechanisms.

Keywords: Situation Calculus, Golog, Reinforcement Learning

1 Introduction

In classical reinforcement learning (RL) and Markov Decision Processes (MDPs), we are given a set of states, actions which stochastically take us from a given state into one of a number of states, and a reward function over states. The goal of learning is to find the optimal policy, which tells us for each state which action to select to maximize our expected reward. In principle this is well understood with methods such as Q-learning solving the problem. However, for most practical applications the huge state and action space is a concern, as explicit representations usually are not viable computationally. To address this problem, state abstraction mechanisms have been explored [2], including FOMDPs [4], which employ first-order logic to characterize a possibly infinite state space using a finite set of formulas.

In this paper, we take this idea further by also constraining the action space using programs written in the action language Golog. Roughly, instead of a state and a set of primitive actions to choose from, we are given a formula describing the current state and a program we need to follow. In the extreme case, when the program is completely deterministic, there is nothing to learn, as the program tells us exactly what the next action is. However, in general the program allows for non-deterministic choices, and here we again need to learn what choices are the best ones in terms of maximizing expected rewards. As we will see, the idea of Q-learning can be adapted to this setting.

More precisely, based on earlier work [4, 10], we start by presenting a method to compute, for a given reward function and Golog program, first-order state formulas describing the possible states before the program is executed. Roughly, these formulas specify sets of states which are equivalent in the sense that the expected rewards are identical when following a policy which is compliant with the program. Moreover, only those properties of the states which are relevant to the expected reward are reflected in those state formulas.

Though the definition of the partitions induced by programs are quite intuitive, these are not practical for actually computing them. This is because huge numbers of unsatisfiable formulas are generated that lead to an even faster increase of the size of the induced partitions. We show how a first-order variant of BDDs can be used as a concise representation for partitions and how these allow to efficiently compute the induced partitions in many cases.

Similar to [2], we then construct a joint semi-MDP (SMDP) over a state space. The states are tuples which consist of the remaining program (starting with a choice point) and a corresponding state formula.

Lastly, we give the semantics for our new Golog dialect QGolog which incorporates reinforcement learning techniques to learn the optimal decisions for the choice points of a program by means of executing it and observing the outcomes. In essence, we integrate a *Q*-learning algorithm for the SMDP described above.

The rest of the paper is organized as follows. After giving a very brief introduction to the situation calculus and Golog, we present how Golog programs induce partitions and how the induced partition can be directly computed using BDDs. In the following section, we discuss the SMDP induced by a Golog program and specify how Q-learning works in this setting. We then present some experimental results, discuss related work and conclude.

2 The Situation Calculus and Golog

The *situation calculus* is a sorted first-order language with equality and sorts of type action and situation. A situation is a history of executed actions; the initial situation is denoted by S_0; the successor situation which results from executing action a in situation s is denoted as $do(a, s)$. Properties of the world that might change from situation to situation are described by means of (relational) *fluents,* which are ordinary predicate symbols that have a situation term as their last argument. We also have some special symbols in our theory which are $Poss$, $choice$, $prob$, and SR (more on these later). A *fluent formula* is any formula not mentioning any of those special symbols. (Fluent-) formulas which mention only a single situation term σ and which do not quantify over situations are called *uniform* in σ. We sometimes consider *situation-suppressed* formulas which are obtained by removing all situation arguments from the fluents. If ϕ is a situation-suppressed formula, then $\phi[\sigma]$ denotes the formula which results from restoring the situation σ in all the fluents mentioned by ϕ. To simplify matters, we assume that there are no object terms other than variables and a countably infinite set of object constants. (We do allow arbitrary function symbols for actions.)

The state of the world is changed by executing actions. For each such *primitive action* $A(\vec{x})$ the preconditions are given by $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$ where Π_A is a fluent formula with variables among $\vec{x}$ and s.[1] According to Reiter's solution of the frame problem [19] the effects of actions are encoded as so-called *successor-state axioms (SSAs)*, one for each fluent:

$$F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$$

where the Φ_F are fluent formulas with free variables among $\vec{x}$, a, and s.

A *basic action theory (BAT)* $\mathcal{D}$ consists of the foundational axioms Σ, which define the space of situations, the successor state axioms $\mathcal{D}_{ssa}$, the action preconditions D_{ap}, the unique name axioms for actions $\mathcal{D}_{una}$, and a set $\mathcal{D}_{S_0}$ of first-order sentences without action terms and uniform in S_0 which describe the fluent values in the initial situation.

For the purposes of this paper we make the following simplifying assumptions for $\mathcal{D}_{S_0}$.

1. $\mathcal{D}_{S_0}$ makes the unique names assumption for object constants, that is, $\mathcal{D}_{S_0}$ contains the set $\mathcal{D}_{una}^{obj} = \{(a \neq b) \mid a \text{ and } b \text{ are distinct object constants}\}$.
2. $\mathcal{D}_{S_0}$ is complete for fluent formulas , that is, for every fluent formula $\phi(S_0)$ uniform in S_0 and which does not mention action terms, either $\mathcal{D}_{S_0} \models \phi(S_0)$ or $\mathcal{D}_{S_0} \models \neg\phi(S_0)$.

A useful property of such $\mathcal{D}_{S_0}$ is that an existential is entailed iff a witness is.

Lemma 2.1
Let $\mathcal{D}_{S_0}$ and $\exists x.\, \phi(x, S_0)$ be as above. Then $\mathcal{D}_{S_0} \models \exists x.\, \phi(x, S_0)$ iff $\mathcal{D}_{S_0} \models \phi(a, S_0)$ for some constant a.

Proof. The if direction is immediate. Also, if $\mathcal{D}_{S_0}$ is unsatisfiable, the only-if direction holds vacuously.

So suppose $\mathcal{D}_{S_0}$ is satisfiable. In [15] (Theorem 2) it is shown that any such $\mathcal{D}_{S_0}$ is satisfiable iff it has a "standard" model M, where by standard we mean a model where the constants are precisely the universe of discourse for objects, that is, there are no unnamed objects. Assume, to the contrary, that $\mathcal{D}_{S_0} \models \exists x.\, \phi(x, S_0)$ yet $\mathcal{D}_{S_0} \not\models \phi(a, S_0)$ for all constant a. Since $\mathcal{D}_{S_0}$ is complete, we therefore have $\mathcal{D}_{S_0} \models \neg\phi(a, S_0)$ for all constant a. In particular, this means for the standard model M that $M \models \forall x.\, \neg\phi(x, S_0)$. By completeness, it follows that $\mathcal{D}_{S_0} \models \forall x.\, \neg\phi(x, S_0)$, a contradiction. ■

Example 2.2
Consider the blocks world domain. The fluent $on(x, y, s)$ expresses that block x is on top of block y in situation s. The action $move(x, y)$ moves block x onto block y. It can be performed iff there is, in the current situation, no other block on x or on y except if y is the table. Furthermore, x and y have to be distinct.

$$Poss(move(x, y), s) \equiv \neg\exists z.on(z, x, s) \wedge (y \neq table \supset \neg\exists z.on(z, y, s)) \wedge x \neq y$$

[1] In formulas like these free variables are understood to be implicitly universally quantified.

A block x is on top of y iff it has just been moved there or iff it has been there before and wasn't moved away with the last action.

$$on(x, y, do(a, s)) \equiv a = move(x, y) \vee on(x, y, s) \wedge \neg\exists z. a = move(x, z)$$

A possible $\mathcal{D}_{S_0}$ as above might then be:

$$\mathcal{D}_{S_0} = \{\forall x, y.\, on(x, y, S_0) \equiv x = b_1 \wedge y = table \vee x = b_2 \wedge y = b1\} \cup \mathcal{D}^{obj}_{una}$$

Besides deterministic primitive actions like *move* we also include stochastic actions. The idea is that, when a stochastic action is executed, nature chooses one of a finite number of deterministic actions [20]. Formally, for a stochastic action a_{st} the possible choices of primitive actions $o_1, \ldots, o_k$ are defined as

$$choice(a_{st}(\vec{x}), a) \equiv \bigvee_{i=1}^{k} a = o_i(\vec{x}).$$

We denote the probability with which $o_i(\vec{x})$ is chosen as the outcome of action $a_{st}(\vec{x})$ in situation s by $prob(o_i(\vec{x}), a_{st}(\vec{x}), s)$. Axioms of the form

$$\sum_{i=1}^{k} prob(n_i(\vec{x}), a_{st}(\vec{x}), s) = 1$$

ensure that we indeed obtain proper probability distributions. Note, we freely use real numbers and assume that they have the intended interpretation.

If the probability distribution with which nature chooses is known, this can also be specified. In our setting, the distribution is generally not known. Nevertheless, we assume that conditions under which the probability distribution are different are known. In particular, we assume that there are sets situation-suppressed formulas $\theta_1, \ldots, \theta_r$ (one fore every stochastic action) which indicate that in all situations in which θ_j holds the probability distribution over the outcome actions is the same. Moreover, these formulas partition the set of situations (see Definition 4.1 below for a formal description of this concept). Formally, we include axioms of the form

$$\theta_j(\vec{x}, s) \wedge \theta_j(\vec{x}, s') \supset prob(N_i(\vec{x}), A_{st}(\vec{x}), s) = prob(N_i(\vec{x}), A_{st}(\vec{x}), s').$$

Note that in the simple case where the distribution does not change at all, there is only one $\theta = true$.

To ensure full observability it has to be possible to determine the actual outcome of a stochastic action. Therefore, sensing conditions $senseCond(N_i) \equiv \varphi_i$ are defined such that if in the situation resulting from executing a stochastic action A_{st} the condition φ_i holds, then the outcome of A_{st} was N_i.

Furthermore, we define sensing functions *SR*. The idea behind those is that when a Golog program is executed in the real world it is necessary to feed back, for instance, sensor values to the Golog program. The *SR*-functions model the possible return values. For example, we assume that the actual system controlled by the Golog program is able to determine the actual outcome of executing a stochastic action in the real world. This is then modeled by

$$SR(A_{st}(\vec{x})) = r \equiv \exists n.\, choice(A_{st}(\vec{x}), n) \wedge r = n$$

The regression of a formula ϕ through an action a is a formula $\phi' = Regr(\phi[do(a,s)])$. The idea is that, for a given BAT, ϕ holds after executing a only in case ϕ' held before the execution of a. Formally: $\mathcal{D} \models \phi[do(a,s)] \equiv \phi'[s]$ (Theorem 4.5.4 in [20]). The regression operator is defined as follows:

$$\begin{aligned}
&Regr(F(\vec{x}, do(a,s))) = \Phi_F(\vec{x}, a, s)\\
&Regr(Poss(A(\vec{x}), do(a,s))) = Regr(\Pi_A(\vec{x}, do(a,s)))\\
&Regr(\neg\phi) = \neg Regr(\phi)\\
&Regr(\phi_1 \wedge \phi_2) = Regr(\phi_1) \wedge Regr(\phi_2)\\
&Regr(\exists x.\, \phi) = \exists x.\, Regr(\phi)
\end{aligned}$$

The high-level agent programming language Golog [16] is based on the situation calculus. Roughly, Golog allows us to write programs where the primitive actions are those defined by a basic action theory. The available language constructs according to [16] are:

- primitive actions a,
- test actions $\varphi?$,
- sequences of programs $[\delta_1; \delta_2]$,
- conditional branchings **if** φ **then** δ_1 **else** δ_2 **end**,
- loops **while** φ **do** δ **end**,
- nondeterministic branchings **nondet**$(\delta_1, \ldots, \delta_n)$,
- nondeterministic choice of arguments **pick**(v, η),
- nondeterministic iteration δ^*,
- and procedures **proc** $P(\vec{x})$ δ **end**.

The meaning of a Golog program can be defined with the help of two special predicates $Final(\delta, s)$ and $Trans(\delta, s, \delta', s')$, which can be read as "$\delta$ can legally terminate in situation s" and "executing the first action of program δ in situation s leads to situation s' with remaining program δ'." For example, if a is a primitive action, then $Trans([a;\rho], s, \delta', s')$ holds iff $Poss(a,s)$ holds, $s' = do(a,s)$, and $\delta' = \rho$. The definition of the $Trans$-predicate requires to reify programs as terms (cf. [11]). We will make use of this too, later, to use programs as arguments of fluents. We will define $Trans$ only for the new constructs introduced in this paper and refer to [11] for the others. To start with, for a stochastic action a_{st}, $Trans$ is defined as

$$Trans(a_{st}, s, \delta', s') \equiv \exists n.\, choice(a_{st}, n) \wedge Trans(n, s, \delta', s')$$

This allows to reason about all possible outcomes of a stochastic action regardless of the probability with which o_i will be the actual outcome of executing a_{st} in the real world. Certainly, if we really want to execute a stochastic action in the real world this has to be handled differently since the actual outcome has to be determined (cf. Section 8.1).

The configurations reachable by executing program δ in situation s are those in the reflexive transitive closure of the transition relation. Formally:

$$Trans^*(\delta, s, \delta', s') \overset{def.}{=} \forall T.\, [(\ldots) \supset T(\delta, s, \delta', s')]$$

where $(\dots)$ stands for the conjunction of the universal closure of the following implications (cf. [11]):

$$True \supset T(\delta, s, \delta, s)$$
$$Trans(\delta, s, \delta'', s'') \wedge T(\delta'', s'', \delta', s') \supset T(\delta, s, \delta', s')$$

3 Markov Decision Processes

In the Markov decision process (MDP) model the agent has to decide on the next action to be executed in the current state which is fully observable to the agent. Nature then determines the subsequent state and a reward the agent obtains for getting from the previous state to the current state by means of the selected action. Formally, a MDP is described by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$. The set $\mathcal{S}$ describes the (possibly infinite) state space. $\mathcal{A}$ is the set of actions available to the agent by which means the agent may change the state of the system. $\mathcal{T}$ is a transition function assigning probabilities to $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$. $\mathcal{T}(s, a, s')$ represents the probability of ending up in state s' after executing action a in state s. The reward function $\mathcal{R}$ maps $\mathcal{S} \times \mathcal{A}$ into the reals; it returns the reward for taking a specific action in a specific state. Solving a MDP refers to determining the optimal policy that in each state returns the action that maximizes the expected future rewards. This is equivalent to finding the optimal state-value function V^* or the optimal action-value function Q^*, respectively. The optimal state-value function $V^*(s)$ returns the expected cumulative future reward when starting in state s and following the optimal policy. Similar for $Q^*(a, s)$: it returns the expected cumulative future reward when performing action a in state s and following the optimal policy thereafter.

The specific solution method we are concerned with in this paper is Q-learning [26]. It is a model-free, off-policy temporal difference learning technique. As such it learns from interacting with the environment, observing the state transitions and the obtained rewards. Interacting with the system is necessary since the model of the system is not known in advance. Furthermore, Q-learning directly approximates the optimal action-value function. After executing the action a_t in the state s_t and thereby reaching the state s_{t+1} and receiving the reward r_{t+1} the Q-function for the state-action pair a_t, s_t is updated according to the following update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

where α is the learning rate and γ the discount factor.

For Q-learning it is known that the Q-function converges to the optimal Q-function Q^* under the assumption that all state-action pairs continue to be updated infinitely often.

4 First-Order Partitions

For a set of situation suppressed formulas $S(\vec{x}) = \{\phi_1(\vec{x}), \dots, \phi_n(\vec{x})\}$, $S(\vec{x})[s]$ denotes the set of formulas $\{\phi_i(\vec{x})[s] \mid 1 \le i \le n\}$.

Definition 4.1
A set $\{\phi_1(\vec{x}), \dots, \phi_n(\vec{x})\}$ of formulas ϕ_i is a partition iff the following conditions hold for all completely specified $\mathcal{D}_{S_0}$ and an arbitrary ground situation term σ:

1. $\mathcal{D} \models \forall \vec{x}. \bigvee_{i=1}^{n} \phi_i(\vec{x})[\sigma]$ and (collectively exhaustive)
2. $\mathcal{D} \models \forall \vec{x}. \phi_i(\vec{x})[\sigma] \supset \neg \left(\bigvee_{j \neq i} \phi_j(\vec{x})[\sigma]\right)$ (mutually exclusive)

Definition 4.2
Let $S_1(\vec{x}_1) = \{\phi_1(\vec{x}_1), \ldots, \phi_n(\vec{x}_1)\}$ and $S_2(\vec{x}_2) = \{\psi_1(\vec{x}_2), \ldots, \psi_m(\vec{x}_2)\}$ be sets of state formulas.

- The binary $\otimes$-operation is defined as:

$$S_1(\vec{x}_1) \otimes S_2(\vec{x}_2) = \{\phi_i(\vec{x}_1) \wedge \psi_j(\vec{x}_2) \,|\, 1 \leq i \leq n, 1 \leq j \leq m\}$$

- The binary $\cup$-operation is defined as:

$$S_1(\vec{x}_1) \cup S_2(\vec{x}_2) = \{\phi_1(\vec{x}_1), \ldots, \phi_n(\vec{x}_1), \psi_1(\vec{x}_2), \ldots, \psi_m(\vec{x}_2)\}$$

Let $S(\vec{x}) = \{\phi_1(\vec{x}), \ldots, \phi_n(\vec{x})\}$ be a set of state formulas.

- The unary $\mathfrak{E}_x$-operation is defined as:

$$\mathfrak{E}_x\, S(\vec{x}) = \left\{ \bigwedge_{i=1}^{n} \psi_i \,|\, \psi_i = \exists x.\, \phi_i(\vec{x}) \text{ or } \psi_i = \neg\exists x.\, \phi_i(\vec{x}) \right\}$$

  The set of formulas resulting from applying the $\mathfrak{E}_x$-operator on $S(\vec{x})$ contains all possible combinations of the $\exists x.\, \phi_i(\vec{x})$ and $\neg\exists x.\, \phi_j(\vec{x})$.

Example 4.3
This example illustrates the operation of the $\mathfrak{E}_x$-operator.

$$\begin{aligned}\mathfrak{E}_x\, \{\phi(x), \neg\phi(x)\} = \{&\exists x.\, \phi(x) \wedge \exists x.\, \neg\phi(x), \neg\exists x.\, \phi(x) \wedge \exists x.\, \neg\phi(x)\} \\ &\exists x.\, \phi(x) \wedge \neg\exists x.\, \neg\phi(x), \neg\exists x.\, \phi(x) \wedge \neg\exists x.\, \neg\phi(x)\}\end{aligned}$$

Lemma 4.4
If $S_1(\vec{x}_1)$ and $S_2(\vec{x}_2)$ are partitions then $S_1(\vec{x}_1) \otimes S_2(\vec{x}_2)$ is a partition. Also, $(\{\phi\} \otimes S(\vec{x})) \cup \{\neg\phi\}$ and $(\{\phi\} \otimes S_1(\vec{x}_1)) \cup (\{\neg\phi\} \otimes S_2(\vec{x}_2))$ are partitions.

Lemma 4.5
If $S(\vec{x})$ is a partition then $\mathfrak{E}_x\, S(\vec{x})$ is a partition.

Proof. Since $S(\vec{x})$ is a partition there exists for each interpretation $\mathcal{M}$ and each variable mapping ν a $\phi_i(\vec{x}) \in S(\vec{x})$ such that $\mathcal{M}, \nu \models \phi_i(\vec{x})$. For all $j \in \{1, \ldots, n\} \backslash \{i\}$, either $\mathcal{M}, \nu \models \exists x.\, \phi_j(\vec{x})$ or $\mathcal{M}, \nu \models \neg\exists x.\, \phi_j(\vec{x})$. Consequently, there is exactly one combination $(\neg)\exists x.\, \phi_1(\vec{x}) \wedge \ldots \wedge \exists x.\, \phi_i(\vec{x}) \wedge \ldots \wedge (\neg)\exists x.\, \phi_n(\vec{x})$ for which $\mathcal{M}, \nu$ is a model. ∎

4.1 Having a Closer Look at the $\mathfrak{E}_x$-Operator

The idea behind the $\mathfrak{E}_x$-operator is to build a partition consisting of formulas that make assumptions about the existence and non-existence, respectively, of satisfying assignments of x for the formulas of the input partition. More precisely, every formula in $\mathfrak{E}_x\, S(x)$ states that for every formula in $S(x) = \{\phi_1(x), \ldots, \phi_n(x)\}$ whether there exists a satisfying x or whether there exists no such x. Thus, if $|S(x)| = n$ then $|\mathfrak{I}_x\, S(x)| = 2^n$. If there exists a formula ψ independent of x and formulas $\tau_i(x)$ (one for each $\phi_i(x)$) such that for every $\phi_i(x) \in S(x)$ it holds that either $\phi_i(x) \equiv \psi \wedge \tau_i(x)$ or $\phi_i \equiv \neg\psi \wedge \tau_i(x)$ the exponential growth can be avoided. Let N^+ be the set of indices for which $\phi_i(x) \equiv \psi \wedge \tau_i(x)$, $i \in N^+$, and N^- be the set of indices for which $\phi_j(x) \equiv \neg\psi \wedge \tau_j(x)$, $j \in N^-$. For every $\phi_i(x)$, $i \in N^+$, it then holds that $\exists x.\, \phi_i(x) \equiv \psi \wedge \exists x.\, \tau_i(x)$. Analogously for every $\phi_j(x)$, $j \in N^-$. Clearly, every formula in $\mathfrak{I}_x\, S(x)$ that mentions a $\phi_i(x)$ and a $\phi_j(x)$ where either

- one is negated and the other one is not and $i \in N^+$ and $j \in N^+$ or
- one is negated and the other one is not and $i \in N^-$ and $j \in N^-$ or
- both are negated or not negated and $i \in N^+$ and $j \in N^-$ (or vice versa)

is (trivially) unsatisfiable, since it contains the subformula $\psi \wedge \neg\psi$. Given those formulas ψ and the $\tau_i(x)$ the set

$$A = \left\{ \psi \wedge \bigwedge_{i \in N^+} (\neg)\tau_i(x) \right\} \cup \left\{ \neg\psi \wedge \bigwedge_{j \in N^-} (\neg)\tau_j(x) \right\}$$

contains for every satisfiable formula $\phi_i(x) \in S(x)$ a formula $\phi'_i(x)$ such that $\phi_i(x) \equiv \phi'_i(x)$. Let $|N^+| = p$ and, consequently, $|N^-| = n - p$. Then, $|A| = 2^p + 2^{n-p} \leq 2^n$. By re-iterating this process for the sets $\{\phi_i(x) \,|\, i \in N^+\}$ and $\{\phi_i(x) \,|\, i \in N^-\}$ a partition that contains even less (trivially) unsatisfiable formulas can be computed. Still, these partitions with less formulas are equivalent to $\mathfrak{I}_x\, S(x)$, i.e., for every satisfiable formula in $\mathfrak{I}_x\, S(x)$ there exists an equivalent formula in the smaller partitions.

Below, when we detail the implementation of the $\mathfrak{I}_x$-operator, we will come back to this idea to avoid generating unsatisfiable formulas by applying the $\mathfrak{I}_x$-operator. Especially, we show how adequate ψ and $\tau_i(x)$ can be determined efficiently.

4.2 The Reward Partition

Throughout this paper we assume the reward function to be $rew(s)$ and that it can be presented in the following form:

$$rew(s) = r \quad \equiv \quad \bigvee_{i=1}^{k} \phi_i^{rew}[s] \wedge r = r_i$$

where the r_i are distinct numerical constants. Since $rew(s)$ is a function (i.e., it assigns a unique reward to every situation) it induces a partition $\{\phi_1^{rew}, \ldots, \phi_n^{rew}\}$, the so-called *reward partition* P^{rew}.

5 Partitions Induced by Golog Programs

Given a reward function (or more precisely the induced reward partition) and the dynamics of the system (the preconditions and effects of actions) one can compute for a Golog program a partition such that the formulas in that partition distinguish states from each other for which the expected reward for executing the program is different. We call the partition the partition *induced* by the program. Below we detail how such partitions induced by Golog programs are computed.

Since this process of computing the partition induced by a program is recursive we are limited to *finite* Golog programs. A Golog program is called finite if every one of its possible execution traces is finite. Consequently, we have to disallow recursive procedure calls, have to transform while-loops into a finite number of nested conditional branchings, and have to replace nondeterministic iterations by a nondeterministic branching over zero to n many iterations. Certainly, this restricts the expressive power of the programs. For instance, a program that, while there is a green block on the table, picks a block and removes it from the table terminates only if there are no more green blocks on the table—no matter how many green blocks have been there initially. The finite version of such a program that only checks n times whether there is a green block on the table could maximally remove n green blocks from the table. Nevertheless, since we only have to restrict the programs we want to learn the optimal execution strategy for, we still can formulate a program that tells the agent to learn to pick a block and remove it from the table and repeat this until there are no more green blocks on the table. Without loss of generality we further assume that every program is *nil*-terminated, that is, every program is of the form $[\delta; nil]$.

For a finite Golog program, $\mathcal{P}(\delta)$ is the partition induced by the program δ. Then, the partition induced by the empty program *nil* is the reward partition:

$$\mathcal{P}(nil) = \mathcal{P}^{rew}$$

The expected reward for executing the empty program only depends on the reward in the current situation since the empty program doesn't change the state.

The partition induced by a program which starts with a sequence $[\delta_1; \delta_2]$ is defined as the partition induced by the program $[\delta_1; [\delta_2; \delta']]$ where δ' is the remaining program:

$$\mathcal{P}([[\delta_1; \delta_2]; \delta']) = \mathcal{P}([\delta_1; [\delta_2; \delta']])$$

If the program starts with a primitive action a the induced partition is defined as:

$$\mathcal{P}([a; \delta]) = (\{Poss(a, s) \land Regr(\phi_i[do(a, s)]) | \phi_i \in \mathcal{P}(\delta)\} \otimes \mathcal{P}^{rew}) \\ \cup \neg Poss(a, s) \quad (5.1)$$

The set of states in which the preconditions for the action a are given is further subdivided according to the formulas ϕ_i in the partition induced by the remaining program regressed through the action a. Additionally, these are split up according to their current reward value. The set of states where the preconditions do not hold are not differentiated any further.

In case the program starts with a stochastic action a_{st} whose possible outcomes are defined to be $n_1, \dots, n_k$ and the conditions under which the probability distribution over the outcomes changes are $\theta_1, \dots, \theta_r$ then

$$\mathcal{P}([a_{st}; \delta]) = \mathcal{P}^{pr}_{a_{st}} \otimes \bigotimes_{i=1}^{k} \mathcal{P}([n_i; \delta]) \quad (5.2)$$

where $\mathcal{P}^{pr}_{a_{st}} = \{\theta_1, \dots, \theta_r\}$.

The partition induced by a program starting with a test action $\vartheta?$ basically is the partition induced by the remaining program where ϑ is conjunctively added to each formula in the partition and a further formula $\neg\vartheta$ is added to the partition.

$$\mathcal{P}([\vartheta?;\delta]) = (\{\vartheta\} \otimes \mathcal{P}(\delta)) \cup \{\neg\vartheta\}$$

Quite similar, the partition induced by a program starting with a conditional branching is computed only that the set of states where $\neg\vartheta$ holds is further subdivided by the partition induced by the program in the else-branch:

$$\mathcal{P}([\textbf{if } \vartheta \textbf{ then } \delta_1 \textbf{ else } \delta_2;\delta]) = (\{\vartheta\} \otimes \mathcal{P}([\delta_1;\delta])) \cup (\{\neg\vartheta\} \otimes \mathcal{P}([\delta_2;\delta]))$$

For programs starting with a nondeterministic branching the induced partition is computed by multiplying the partitions induced by all the possible remaining programs with each other.

$$\mathcal{P}([\textbf{nondet}(\delta_1,\ldots,\delta_n);\delta]) = \bigotimes_{i=1}^{n} \mathcal{P}([\delta_i;\delta])$$

For a nondeterministic choice of argument the formulas in the induced partition express the existence and non-existence of objects that satisfy the formulas in the partition induced by the remaining program. This is exactly what the $\boldsymbol{\exists}_v$-operator achieves:

$$\mathcal{P}([\textbf{pick}(v,\eta);\delta]) = \boldsymbol{\exists}_v\, \mathcal{P}([\eta;\delta]) \tag{5.3}$$

For a procedure call $P(\vec{t})$ where the procedure is defined as **proc** $P(\vec{x})$ δ_P the partition is computed by replacing the formal parameters in the body δ_P of the procedures with the actual parameters and computing the induced partition for the program consisting of the body and the remaining program:

$$\mathcal{P}([P(\vec{t});\delta]) = \mathcal{P}([\delta_P{}^{\vec{x}}_{\vec{t}};\delta])$$

Theorem 5.1
For any finite Golog program δ and a reward function $rew(s)$ that induces a reward partition $\mathcal{P}^{rew}$ the set of of formulas $\mathcal{P}(\delta)$ describes a partition.

Proof. Directly follows from Lemma 4.4 and 4.5. ∎

In the remainder of this section we examine further properties of the partitions induced by programs. We intent to use these partitions as the basis of an abstraction mechanism that, informally speaking, allows to generalize the assessment of a program from ground situations to all situations in which a particular formula in the induced partition holds. Consequently, we need make sure that this generalization is valid. In particular it has to be shown that if the agent chooses a particular execution strategy for a program δ (i.e., resolves the nondeterminism in the program in a particular way) then for any two situations σ_1 and σ_2 such that there is a formula in $\mathcal{P}(\delta)$ with $\mathcal{D} \models \phi[\sigma_1] \wedge \phi[\sigma_2]$ the expected reward for executing the program according to the selected execution strategy is the same. First, we formally define the set of all possible execution strategies for a program δ.

Definition 5.2
For any program δ the (possibly infinite) set $Det(\delta)$ contains all deterministic versions of δ:

$$\begin{aligned}
&Det(nil) = \{nil\}, \quad Det(a) = \{a\}, \quad Det(a_{st}) = \{a_{st}\}, \quad Det(\vartheta?) = \{\vartheta?\}\\
&Det(\textbf{if } \vartheta \textbf{ then } \delta_1 \textbf{ else } \delta_2) = \\
&\qquad\qquad\qquad \{\textbf{if } \vartheta \textbf{ then } \delta_1' \textbf{ else } \delta_2' \,|\, \delta_1' \in Det(\delta_1) \text{ and } \delta_2' \in Det(\delta_2)\}\\
&Det(\textbf{nondet}(\delta_1, \ldots, \delta_n)) = \bigcup_{i=1}^{n} Det(\delta_i)\\
&Det(\textbf{pick}(v, \eta)) = \bigcup_{c \in \mathfrak{C}} Det(\eta_c^v)\\
&Det(P(\vec{t})) = Det(\delta_P{}^{\vec{x}}_{\vec{t}}) \text{ with } \textbf{proc } P(\vec{x})\ \delta_P(\vec{x})\\
&Det([\delta_1; \delta_2]) = \{[\delta_1'; \delta_2'] \,|\, \delta_1' \in Det(\delta_1) \text{ and } \delta_2' \in Det(\delta_2)\}
\end{aligned}$$

where $\mathfrak{C}$ is the set of all constants.

Lemma 5.3
For each $\delta_D \in Det(\delta)$ and $\mathcal{D}$ it holds that if δ_D is *legally* executable in a ground situation σ then so is δ. Formally:

$$\begin{aligned}
&\text{if } \mathcal{D} \models \exists \delta', \sigma'. Trans^*(\delta_D, \sigma, \delta', \sigma') \wedge Final(\delta', \sigma')\\
&\qquad\qquad\qquad \text{then } \mathcal{D} \models \exists \delta', \sigma'. Trans^*(\delta, \sigma, \delta', \sigma') \wedge Final(\delta', \sigma')
\end{aligned}$$

Definition 5.4
For a given $\mathcal{D}$, let Δ be a probability distribution over the outcomes of the stochastic actions:

$$\Delta(a_{st}, n_i, s) \rightarrow [0, 1] \text{ such that } \mathcal{D} \models choice(a_{st}, n_i)$$

Then, for a $\delta_D \in Det(\delta)$ the function $\hat{R}(\delta_D, \sigma, \Delta)$ returns a tuple $\langle r, pr\rangle$ where r is the expected reward of executing δ_D in the ground situation σ and pr is the probability that δ_D can be successfully executed in σ.

$$\begin{aligned}
&\hat{R}(nil, \sigma, \Delta) = \langle r, 1.0\rangle \text{ for } \mathcal{D} \models rew(\sigma) = r\\
&\hat{R}([a; \delta'], \sigma, \Delta) = \begin{cases} \langle r + r', pr'\rangle & \text{if } \mathcal{D} \models Poss(a, \sigma); \text{ for } \mathcal{D} \models rew(\sigma) = r\\ & \text{and } \hat{R}(\delta', do(a, \sigma), \Delta) = \langle r', pr'\rangle\,,\\ \langle r, 0.0\rangle & \text{otherwise; for } \mathcal{D} \models rew(\sigma) = r \end{cases}\\
&\hat{R}([\vartheta?; \delta'], \sigma, \Delta) = \begin{cases} \hat{R}(\delta', \sigma, \Delta) & \text{if } \mathcal{D} \models \vartheta[\sigma],\\ \langle r, 0.0\rangle & \text{otherwise; for } \mathcal{D} \models rew(\sigma) = r \end{cases}\\
&\hat{R}([a_{st}; \delta'], \sigma, \Delta) = \left\langle \sum_i \Delta(a_{st}, n_i, \sigma) \cdot r_i, \sum_i \Delta(a_{st}, n_i, \sigma) \cdot pr_i \right\rangle,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for } \hat{R}([n_i; \delta'], \sigma, \Delta) = \langle r_i, pr_i\rangle\\
&\hat{R}([\textbf{if } \vartheta \textbf{ then } \delta_1 \textbf{ else } \delta_2; \delta'], \sigma, \Delta) = \begin{cases} \hat{R}([\delta_1; \delta'], \sigma, \Delta) & \text{if } \mathcal{D} \models \vartheta[\sigma],\\ \hat{R}([\delta_2; \delta], \sigma, \Delta) & \text{otherwise} \end{cases}
\end{aligned}$$

We say that the partition $\mathcal{P}_{a_{st}}^{pr} = \{\vartheta_1, \ldots, \vartheta_r\}$ and the probability distribution Δ conform to each other iff for all ground situations σ, σ' it holds that if $\mathcal{D} \models \vartheta_j[\sigma] \wedge \vartheta_j[\sigma']$ then $\Delta(a_{st}, n_i, s) = \Delta(a_{st}, n_i, s')$ for all n_i and ϑ_j.

With the definition of the set of deterministic programs for any nondeterministic program and the definition of $\hat{R}$ we can now formulate a theorem that states that the formulas in a partition induced by a program differentiate situations in which the expected future reward for executing the same deterministic version of a program is different.

Theorem 5.5
For a given $\mathcal{D}$, any program δ, all Δ that conform to $\mathcal{P}_{a_{st}}^{pr}$ for all stochastic actions a_{st}, and ground situations σ_1, σ_2 it holds that if, for any $\delta_D \in Det(\delta)$, $\hat{R}(\delta_D, \sigma_1, \Delta) \neq \hat{R}(\delta_D, \sigma_2, \Delta)$ [2] then there exist distinct $\phi_i, \phi_j \in \mathcal{P}(\delta)$ with $\mathcal{D} \models \phi_i[\sigma_1] \wedge \phi_j[\sigma_2]$.

Proof. By induction on the program structure of δ with the empty program nil being the base case. $\hat{R}(nil, \sigma_1, \Delta) = \langle rew(\sigma_1), 1.0\rangle$, $\hat{R}(nil, \sigma_2, \Delta) = \langle rew(\sigma_2), 1.0\rangle$, and by assumption $rew(\sigma_1) \neq rew(\sigma_2)$. Since $\mathcal{P}(nil) = \mathcal{P}^{rew}$ there clearly have to be distinct $\phi_i, \phi_j \in \mathcal{P}(nil)$.

The inductive cases have to be examined according to the beginning of the program.

- $\delta = [a; \delta']$ and $\delta_D = [a; \delta'_D]$, $\delta'_D \in Det(\delta')$:
  If $\hat{R}(\delta_D, \sigma_1, \Delta) \neq \hat{R}(\delta_D, \sigma_1, \Delta)$ then $rew(\sigma_1) \neq rew(\sigma_2)$, $\mathcal{D} \models Poss(a, \sigma_1) \not\equiv Poss(a, \sigma_2)$, or $\hat{R}(\delta', do(a, \sigma_1), \Delta) \neq \hat{R}(\delta', do(a, \sigma_2), \Delta)$.
  In the first case there are distinct $\phi_i^{rew}, \phi_j^{rew} \in \mathcal{P}^{rew}$ with $\mathcal{D} \models \phi_i^{rew}[\sigma_1] \wedge \phi_j^{rew}[\sigma_2]$. In the second case either $\phi_i^{rew}[\sigma_1] \equiv Poss(a, \sigma_1)$ and $\phi_j^{rew}[\sigma_2] \equiv \neg Poss(a, \sigma_2)$ or vice versa. In the third case there have to be distinct $\phi'_i, \phi'_j \in \mathcal{P}(\delta')$ with $\mathcal{D} \models \phi'_i[do(a, \sigma_1)] \wedge \phi'_j[do(a, \sigma_2)]$ according to the inductive assumption.
  According to Equation 5.1 there are for each $\phi \in \mathcal{P}(\delta)$ unique $\phi' \in \mathcal{P}(\delta')$, $\phi^{rew} \in \mathcal{P}^{rew}$, and $\phi^{poss} \in \{Poss(a, s), \neg Poss(a, s)\}$ such that $\mathcal{D} \models \forall s. \phi[s] \supset (\phi'[do(a, s)] \wedge \phi^{rew}[s] \wedge \phi^{poss}[s])$. Consequently, in each of the above cases there have to be distinct $\phi_i, \phi_j \in \mathcal{P}(\delta)$.
- $\delta = [a_{st}; \delta']$ and $\delta_D = [a_{st}; \delta'_D]$, $\delta'_D \in Det(\delta')$:
  If $\hat{R}(\delta_D, \sigma_1, \Delta) \neq \hat{R}(\delta_D, \sigma_2, \Delta)$ then $\Delta(a_{st}, n_i, \sigma_1) \neq \Delta(a_{st}, n_i, \sigma_2)$ or $\hat{R}([n_i; \delta'], \sigma_1, \Delta) \neq \hat{R}([n_i; \delta'], \sigma_2, \Delta)$ for at least one of the outcomes n_i.
  In the former case there are distinct $\theta_i, \theta_j \in \mathcal{P}_{a_{st}}^{pr}$ with $\mathcal{D} \models \theta_i[\sigma_1] \wedge \theta_j[\sigma_2]$. In the latter case there are distinct $\phi'_i, \phi'_j \in \mathcal{P}([n_i; \delta'])$ with $\mathcal{D} \models \phi'_i[\sigma_1] \wedge \phi'_j[\sigma_2]$ according to the inductive assumption.
  According to Equation 5.2 there are for each $\phi \in \mathcal{P}(\delta)$ unique $\theta \in \mathcal{P}_{a_{st}}^{pr}$ and unique $\phi_i \in \mathcal{P}([n_i; \delta'])$ for each outcome n_i such that $\mathcal{D} \models \forall s. \phi[s] \supset (\theta[s] \wedge \bigwedge_i \phi_i[s])$. Consequently, in each of the above two cases there have to be distinct $\phi_i, \phi_j \in \mathcal{P}(\delta)$.
- $\delta = [\varphi?; \delta']$ and $\delta_D = [\varphi?; \delta'_D]$, $\delta'_D \in Det(\delta')$:
  Then $\mathcal{D} \models \varphi[\sigma_1] \wedge \neg\varphi[\sigma_2]$ (or the other way around) or $\hat{R}(\delta'_D, \sigma_1, \Delta) \neq \hat{R}(\delta'_D, \sigma_2, \Delta)$. In the former case there have to be distinct $\phi_i, \phi_j \in \mathcal{P}(\delta)$ with $\mathcal{D} \models \phi_i[\sigma_1] \supset \varphi[\sigma_1]$ and $\mathcal{D} \models \phi_j[\sigma_2] \supset \neg\varphi[\sigma_2]$ (or vice versa). In the latter case there are distinct $\phi'_i, \phi'_j \in \mathcal{P}(\delta')$ with $\mathcal{D} \models \phi'_i[\sigma_1] \wedge \phi'_j[\sigma_2]$ and then also distinct $\phi_i, \phi_j \in \mathcal{P}(\delta)$ with $\phi_i \equiv \varphi \wedge \phi'_i$ and $\phi_j \equiv \varphi \wedge \phi'_j$.

[2] $\langle r, pr\rangle \neq \langle r', pr'\rangle$ iff $r \neq r'$ or $pr \neq pr'$.

- $\delta = [\mathbf{if}\ \varphi\ \mathbf{then}\ \delta^1\ \mathbf{else}\ \delta^2; \delta']$ and $\delta_D = [\mathbf{if}\ \varphi\ \mathbf{then}\ \delta^1_D\ \mathbf{else}\ \delta^2_D; \delta'_D]$, $\delta^1_D \in Det(\delta^1)$, $\delta^2_D \in Det(\delta^2)$, $\delta'_D \in Det(\delta')$:
  Then $\varphi[\sigma_1] \not\equiv \varphi[\sigma_2]$ or if $\mathcal{D} \models \varphi[\sigma_1] \wedge \varphi[\sigma_2]$ then $\hat{R}([\delta^1_D; \delta'_D], \sigma_1, \Delta) \neq \hat{R}([\delta^2_D, \delta'_D], \sigma_2, \Delta)$ or if $\mathcal{D} \models \neg\varphi[\sigma_1] \wedge \neg\varphi[\sigma_2]$ then $\hat{R}([\delta^2_D; \delta'_D], \sigma_1, \Delta) \neq \hat{R}([\delta^2_D; \delta'_D], \sigma_2, \Delta)$. By the same argument as in the previous case there have to be distinct $\phi_i, \phi_j \in \mathcal{P}(\delta)$ with $\mathcal{D} \models \phi_i[\sigma_1] \wedge \phi_j[\sigma_2]$.
- $\delta = [\mathbf{pick}(v, \eta); \delta']$, $\delta = [\mathbf{nondet}(\delta_1, \dots, \delta_n); \delta']$, or $\delta = [P(\vec{t}); \delta'])$ for a procedure $P(\vec{x})$:
  In each of these cases the $\delta_D \in Det(\delta)$ are of one of the above forms and thus by the same arguments as given above there are distinct $\phi_i, \phi_j \in \mathcal{P}(\delta)$ with $\mathcal{D} \models \phi_i[\sigma_1] \wedge \phi_j[\sigma_2]$.

■

Roughly speaking, what we are after is to find out what the optimal deterministic version of a given program is. But instead of doing this for every situation we intent to do it only once for each formula in the partition induced by the program. The following theorem establishes that if there is one deterministic version of the program which is optimal in one situation and another one optimal in another situation and each of these deterministic versions is suboptimal in the other situation then those situations are distinguished by the induced partition. This means that for every formula in the induced partition there is a one or more deterministic versions that are all equally good in all situations in which the formula holds.

Theorem 5.6
For a given $\mathcal{D}$ and for all δ, Δ that conform to $\mathcal{P}^{pr}_{a_{st}}$ for all stochastic actions a_{st}, and ground situations σ_1 and σ_2 it holds that if there are distinct $\delta^r_D, \delta^s_D \in Det(\delta)$ which maximize[3] $\hat{R}(\delta^r_D, \sigma_1, \Delta)$ and $\hat{R}(\delta^s_D, \sigma_2, \Delta)$, respectively, with $\hat{R}(\delta^r_D, \sigma_1, \Delta) \neq \hat{R}(\delta^s_D, \sigma_1, \Delta)$ or $\hat{R}(\delta^r_D, \sigma_2, \Delta) \neq \hat{R}(\delta^s_D, \sigma_2, \Delta)$ then there exist distinct $\phi_i, \phi_j \in \mathcal{P}(\delta)$ such that $\mathcal{D} \models \phi_i[\sigma]$ and $\mathcal{D} \models \phi_j[\sigma']$.

Proof. Again, for the proof we differentiate according to the beginning of δ.

- $\delta = [\mathbf{nondet}(\delta_1, \dots, \delta_n); \delta']$, $\delta^r_D \in Det([\delta_i; \delta'])$, $\delta^s_D \in Det([\delta_j; \delta'])$, $i \neq j$:
  Then there have to be distinct $\phi'_i, \phi''_i \in \mathcal{P}([\delta_i; \delta'])$ with $\mathcal{D} \models \phi'_i[\sigma_1] \wedge \phi''_i[\sigma_2]$ or distinct $\phi'_j, \phi''_j \in \mathcal{P}([\delta_j; \delta'])$ with $\mathcal{D} \models \phi'_j[\sigma_1] \wedge \phi''_j[\sigma_2]$. Otherwise $\hat{R}([\delta_i; \delta'], \sigma_1, \Delta) = \hat{R}([\delta_i; \delta'], \sigma_2, \Delta)$ and $\hat{R}([\delta_j; \delta'], \sigma_1, \Delta) = \hat{R}([\delta_j; \delta'], \sigma_2, \Delta)$. In such a case either continuing in σ_1 as well as σ_2 with $[\delta_i; \delta']$ or $[\delta_j; \delta']$ as the remaining program is the better choice and consequently the other one is not the maximizing choice.
  Assume there are distinct $\phi'_i, \phi''_i \in \mathcal{P}([\delta_i; \delta']$ with $\mathcal{D} \models \phi'_i[\sigma_1] \wedge \phi''_i[\sigma_2]$. Then there also have to be distinct $\phi_i, \phi_j \in \mathcal{P}(\delta)$ such that $\mathcal{D} \models \phi_i[\sigma_1] \wedge \phi'_i[\sigma_1]$ and $\mathcal{D} \models \phi_j[\sigma_2] \wedge \phi''_i[\sigma_2]$. Likewise for distinct $\phi'_j, \phi''_j \in \mathcal{P}([\delta_j; \delta])$ with $\mathcal{D} \models \phi'_j[\sigma_1] \wedge \phi''_j[\sigma_2]$.
- $\delta = [\mathbf{pick}(v, \eta); \delta']$, $\delta^r_D \in Det([\eta^v_{c_1}; \delta'])$, $\delta^s_D \in Det([\eta^v_{c_2}; \delta'])$ such that for any $\delta^{c_1}_D, \delta^{c_2}_C \in Det(\delta)$ that only differ in the choice for v it holds that the expected rewards for executing $\delta^{c_1}_D$ and $\delta^{c_2}_D$ in σ_1 as well as σ_2 are different. That is, choosing c_1 and c_2 actually makes a difference w.r.t. $\hat{R}$.

[3] $\delta_D \in Det(\delta)$ maximizes $\hat{R}(\delta_D, s, Delta)$ iff $\delta_D = \arg\max_{\delta'_D \in Det}(\delta) f(\hat{R}(\delta'_D, s, \Delta))$ for arbitrary but fixed $f : \langle r, pr \rangle \to v$.

Consequently, there have to exist distinct $\phi'_i(v), \phi'_j(v) \in \mathcal{P}([\eta; \delta'])$ with $\mathcal{D} \models \phi'_i(c_1)[\sigma_1] \wedge \phi'_j(c_2)[\sigma_2]$. Then, there are also distinct $\phi_i, \phi_j \in \mathcal{P}(\delta)$ such that $\mathcal{D} \models \phi_i[\sigma_1] \wedge \exists v.\, \phi'_i(v)[\sigma_1]$ and $\mathcal{D} \models \phi_j[\sigma_2] \wedge \neg\exists v.\, \phi'_i(v)[\sigma_2]$. Otherwise δ^s_D could make the same choice for v in σ_2 as δ^r_D in σ_1 and thus could obtain the same reward.

- For all other cases a combination of the arguments given above and in the proof for Theorem 5.5 suffices to prove the assertion. Precisely, the "other cases" are programs that begin with deterministic constructs or nondeterministic constructs where the choices made for these by δ^r_D and δ^s_D, respectively, are irrelevant w.r.t. $\hat{R}$.

■

6 Representing Partitions by BDDs

Although the above definition of $\mathcal{P}(\delta)$ has the desired properties, it has a severe drawback that prohibits a direct implementation: possibly, the number of formulas in a partition which are trivially unsatisfiable becomes prohibitively large. For the $\exists_v$-operator this problem was already discussed above and a hint at a solution was presented (cf. Section 4.1). But the problem might also occur when the $\otimes$-operator is applied.

We show how a variant of binary decision diagrams (BDDs) can be used to compactly represent partitions. But not only does this improve the representational efficiency, by directly manipulating BDDs instead of partitions the computational efficiency is improved, too.

A BDD representing a Boolean function $f(x_1, \ldots, x_n)$ is a DAG consisting of n decision nodes and two terminal nodes labeled with '0' and '1', respectively. Every decision node is labeled with one of the Boolean variables $x_1, \ldots, x_n$ and has exactly two child nodes, the high child and the low child. For a node v in the BDD the Boolean function represented by the sub-BDD rooted in v is defined as $f_v(x_1, \ldots, x_n) = x_i \cdot f_{high(v)}(x_1, \ldots, x_n) + \bar{x}_i \cdot f_{low(v)}(x_1, \ldots, x_n)$ where x_i is the label of node v.

The variant of BDDs [6] that represent partitions consisting of n situation suppressed formulas are defined by the following BNF grammar:

$$B ::= L \,|\, \texttt{if } \varphi \texttt{ then } B^h \texttt{ else } B^l$$

where L is the label of a terminal node. The decision nodes are represented as `if` φ `then` B^h `else` B^l where φ is a situation-suppressed formula associated with this decision node; B^h is the high-child and B^l is the low-child of the decision node. For any ground situation σ if $\mathcal{D} \models \varphi[\sigma]$ holds then the high branch leading to B^h is followed; otherwise the low branch leading to B^l is followed. The terminal nodes are labeled with subsets of $\{1, \ldots, n\}$.

The formulas associated with the decision nodes are either atomic formulas or they are quantified formulas where the scope of the quantifiers is minimized as much as possible. In order to construct the BDD representing a partition $P = \{\phi_1, \ldots, \phi_n\}$ the ϕ_i need to be transformed first such that they can be written as conjunctions or disjunctions over sub-formulas that conform to the above description. This is achieved by means of the rewrite rules shown below which push the quantifiers inwards as far as possible and thus reveal the *"propositional structure"* [22] of the formulas.

$$\begin{aligned}
&\exists x.\, [\phi(x,\star) \lor \psi(x,\star)] \to \exists x.\, [\phi(x,\star)] \lor \exists x.\, [\psi(x,\star)]\\
&\forall x.\, [\phi(x,\star) \land \psi(x,\star)] \to \forall x.\, [\phi(x,\star)] \land \forall x.\, [\psi(x,\star)]\\
&\exists x.\, [\phi(x,\star) \circ \psi(\star)] \to \exists x.\, [\phi(x,\star)] \circ \psi(\star), \quad \circ \in \{\land, \lor\}\\
&\forall x.\, [\phi(x,\star) \circ \psi(\star)] \to \forall x.\, [\phi(x,\star)] \circ \psi(\star), \quad \circ \in \{\land, \lor\}
\end{aligned}$$

The $\star$ denotes other free variables which are not bound by the quantifiers.

The partition represented by a BDD B consists of n distinct formulas ϕ_i where n is the number of distinct terminal nodes in B. A formula ϕ_i is represented by all paths that lead from the root node to the terminal node i. ϕ_i is reconstructed by disjunctively combining the formulas that are represented by each of these paths. The formula represented by a single path is the conjunction of the formulas associated with the decision nodes on that path. Inherently, the BDD representation facilitates to identify distinguishing subformulas: if the ψ is the formula associated with the root node of the BDD then it holds that for all ϕ_i represented by the BDD either $\models \phi_i \supset \psi$ or $\models \phi_i \supset \neg\psi$.

Lemma 6.1
For a partition P let $\mathfrak{F}_P$ be the set of sub-formulas that result from applying the above rewrite formulas on the formulas of the partition until the quantifiers cannot be pushed in any further. For any total ordering over $\mathfrak{F}_P$ there exists a unique reduced BDD representing the partition P.

A proof for BDDs that only have two distinct terminal nodes is given in [6]. It trivially generalizes to BDDs with n distinct terminal nodes.

Example 6.2
This example illustrates the generation of the BDD representing the partition $\{Poss(move(x,y),s), \neg Poss(move(x,y),s)\}$. The resulting BDD is shown in Figure 1. The application of the rewrite rules is demonstrated below. The boxed subformulas cannot be broken down any further and, consequently, are the formulas that are associated with the decision nodes in the BDD.

$$\begin{aligned}
Poss(move(x,y),s) &\equiv \neg\exists z.\, [on(z,x,y) \lor y \neq table \land on(z,y,s)] \land x \neq y\\
&\equiv \forall z.\, [\neg(on(z,x,s) \lor y \neq table \land on(z,y,s))] \land x \neq y\\
&\equiv \forall z.\, [\neg on(z,x,s) \land \neg(y \neq table \land on(z,y,s))] \land x \neq y\\
&\equiv \forall z.\, [\neg on(z,x,s) \land (y = table \lor \neg on(z,y,s))] \land x \neq y\\
&\equiv \forall z.\, [\neg on(z,x,s)] \land \forall z.\, [y = table \lor \neg on(z,y,s)] \land x \neq y\\
&\equiv \boxed{\forall z.\, [\neg on(z,x,s)]} \land \left(\boxed{y = table} \lor \boxed{\forall z.\, [\neg on(z,y,s)]} \right)\\
&\quad \land \boxed{\neg x = y}
\end{aligned}$$

6.1 Limitations of the Rewrite Rules

Certainly, the above rewrite rules are not capable of generating formulas such that all equivalent sub-formulas become syntactically alike. And, consequently, it might

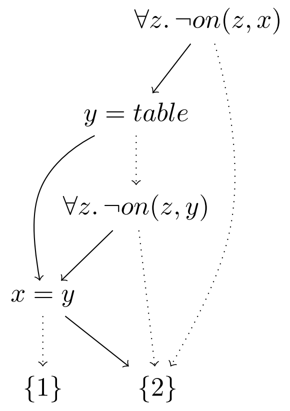
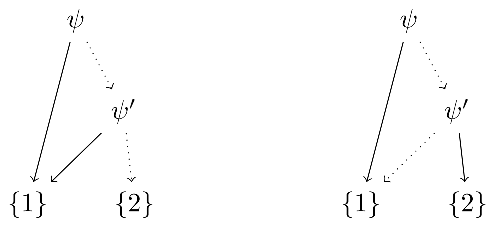


FIG. 1: BDD representing the partition $\{Poss(move(x,y),s), \neg Poss(move(x,y),s)\}$. The high branches are indicated by solid lines, the low branches by dotted lines.

be that after applying the above rewrite rules on the formulas of the partition these contain syntactically different sub-formulas ψ and ψ' but $\models \psi \equiv \psi'$ or $\models \psi \equiv \neg\psi'$. Although this means that there might be no common sub-formula ψ such that for every ϕ_i either $\phi_i \equiv \psi \wedge \tau_i$ or $\phi_i \equiv \neg\psi \wedge \tau_i$, a BDD representing the partition can still be constructed. Failing to identify equivalent sub-formulas, though, leads to a BDD that is larger in size than the minimal BDD. What the BDDs look like in the cases where the equivalence between ψ and ψ' and ψ and $\neg\psi'$, respectively, could not be made out is illustrated below. For syntactically different ψ, ψ', the BDD on the left represents the partition $\{\psi, \neg\psi'\}$ (where $\psi \equiv \psi'$); the BDD on the right represents the partition $\{\psi, \psi'\}$ (where $\psi \equiv \neg\psi'\}$).



Although there is no formula in the partition that mentions $\neg\psi$ as a sub-formula with the knowledge that $\{\psi, \neg\psi'\}$ is a partition we know that it is correct to differentiate the state where ψ does not hold according to ψ'. This "solution method" scales to cases where there are more formulas in the partition and the formulas are more complex.

7 BDDs Induced by Golog Programs

Not only do BDDs allow for a compact representation of a partition but by directly manipulating BDDs instead of the partitions they represent we will show how the generation of (trivially) unsatisfiable formulas can be reduced drastically. So, eventually, we strive to directly compute the BDD representing the partition induced by a Golog program without computing the partition first and then generate the BDD representing it—at least where this is possible. As we will see there are two exceptions. For short, we call the BDD representing the partition by a program δ the BDD induced by δ.

Analogously to the operations on partitions we used to compute the partition induced by a program (cf. Definition 4.2) we define similar operations on BDDs. In particular these are the binary operator $B_1 \otimes B_2$, the ternary operator $B_1 \oslash^{\phi} B_2$, and the unary operator $\mathfrak{I}_v\, B$.

The data structure used to store the BDDs builds on a cache that contains mappings of the form $\langle \phi, B^h, B^l \rangle \to id$, i.e., a unique identifier is assigned to BDDs. By storing the identifiers of the high- and low-child instead of the corresponding BDDs allows to efficiently store common sub-BDDs. In order to keep the presentation of the following algorithms simple we assume that the function calls to query and to modify the cache happen transparently every time a new (sub-) BDD is constructed. Additionally, we assume a globally given ordering over the formulas associated with the decision nodes and that the BDDs operated on conform to that ordering.

With $\mathcal{B}(P)$ we denote the BDD representing the partition P; with $\mathcal{P}(B)$ we denote the partition represented by the BDD B.

The Operator $B_1 \otimes B_2$

The operator $B_1 \otimes B_2$ is the equivalent to the $\otimes$ operator for partitions: it computes the BDD that represents the partition $\mathcal{P}(B_1) \otimes \mathcal{P}(B_2)$.

In Algorithm 1 the implementation for this operator is sketched. If both input BDDs only consist of terminal nodes the algorithm returns a terminal node whose label is the union of the labels of the input terminal nodes. If only one of the input BDDs is a terminal node then the output BDD basically coincides with the other BDD but the labels of the terminal nodes are modified by taking the union of the original label and the label of the input terminal node. If both input BDDs are non-terminal nodes the ordering of the formulas associated with the respective root nodes is compared and the process is continued accordingly. The formula preceding in the ordering is associated with the root node of the newly generated BDD and the high and low-child are computed by combining the high and the low-child, respectively, of the preceding input BDD with the other input BDD. In case of equality (by which we mean that there is a unifier for the two formulas) both high and low-children are combined with each other. Given that the two input BDDs conform to the ordering the generated BDD will do so, too.

The Operator $B_1 \oslash^{\phi} B_2$

The ternary operator $B_1 \oslash^{\phi} B_2$ is not a direct equivalent to any operator defined on partitions. As inputs it takes two BDDs, B_1 and B_2, and a formula ϕ. From these it computes a BDD that represents the partition $\{\phi\} \otimes \mathcal{P}(B_1) \cup \{\neg\phi\} \otimes \mathcal{P}(B_2)$.

In the presentation of the algorithm below we assume wlog that in a preprocessing step a BDD representing the partition $\{\phi, \neg\phi\}$ is computed whose terminal nodes are labeled with "$\{+\}$" (for the case where ϕ holds) and "$\{-\}$" (otherwise).

Algorithm 1: $B_1 \otimes B_2$

```
if $B_1$ and $B_2$ are terminal nodes then
    return $B_1 \cup B_2$
if $B_1$ is a non-terminal node and $B_2$ is a terminal node then
    $\varphi \longleftarrow \varphi_1$
    $B^h \longleftarrow B_1^h \otimes B_2$
    $B^l \longleftarrow B_1^l \otimes B_2$
    return $B$
if $B_1$ is a terminal node and $B_2$ is a non-terminal node then
    [...]
if $\varphi_1 = \varphi_2$ then
    $\varphi \longleftarrow \varphi_1$
    $B^h \longleftarrow B_1^h \otimes B_2^h$
    $B^l \longleftarrow B_1^l \otimes B_2^l$
    return $B$
else if $\varphi_1 \prec \varphi_2$ then
    $\varphi \longleftarrow \varphi_1$
    $B^h \longleftarrow B_1^h \otimes B_2$
    $B^l \longleftarrow B_1^l \otimes B_2$
    return $B$
else                                                  // $\varphi_2 \prec \varphi_1$
    [...]
```

Then, basically, the operator combines the three BDDs in a similar fashion as the $\otimes$-operator. The only exception is if B_ϕ, the BDD representing $\{\phi, \neg\phi\}$, is a terminal node. In that case either B_1 or B_2 is returned depending on the label of the terminal node.

The Operator $\exists_v B$

The $\exists_v B$ operator produces a BDD that represents the partition generated by $\exists_v \mathcal{P}(B)$. Contrary to the two operators for BDDs introduced above, $\otimes$ and $\oslash$, the $\exists_v B$ operator requires to first reconstruct the partition represented by the BDD B. This necessity arises from the definition of the $\exists_v$-operator for partitions. Each formula in the resulting partition makes a statement about the existence or non-existence of objects such that each ϕ_i in the input partition can be satisfied. Transferred to the BDD representation this means that the formulas associated with the decision nodes of the output BDD have to be constructed on the basis of all paths through the input BDD and not just from its nodes (an example is given in Figure 2).

For reasons of exposition we refrain from presenting a formal algorithm for the $\exists_v B$ operator. Instead we give an informal explanation that holds on to the structure of the actual algorithm.

1. Reorder the BDD B such that along all paths of the BDD the nodes labeled with

Algorithm 2: $B_1 \oslash^{\mathcal{B}(\{\phi,\neg\phi\})} B_2)$

```
if $B_\phi = \{+\}$ then
    return $B_1$
if $B_\phi = \{-\}$ then
    return $B_2$
if $B_1$ and $B_2$ are non-terminal nodes then
    if $\varphi_\phi \prec \varphi_1$ and $\varphi_\phi \prec \varphi_2$ then
        $\varphi \longleftarrow \varphi_\phi$
        $B^h \longleftarrow B_1 \oslash^{B^h_\phi} B_2$
        $B^l \longleftarrow B_1 \oslash^{B^l_\phi} B_2$
    // all other possibilities for the ordering of $\varphi_1$, $\varphi_2$, and $\varphi_\phi$
    else if ... then
        [...]
    return $B$
else if $B_1$ is a non-terminal node and $B_2$ is a terminal node then
    // combine $B_1$ and $B_\phi$ with each other
    [...]
else if $B_1$ is a terminal node and $B_2$ is a non-terminal node then
    // combine $B_2$ and $B_\phi$ with each other
    [...]
else if $B_1$ and $B_2$ are terminal nodes then
    return $B_\phi$
```

formulas that do not mention v as a free variable come first. Let the resulting BDD be B'.

2. On each path through B' identify the first node N that is labeled with a formula that mentions v as a free variable. Let the sub-BDD that has N as its root node be B_N. For every B_N perform the following:

   (a) Assume the labels of the terminal nodes of B_N are $l_1, \ldots, l_n$. For reasons of readability we refer to those labels as $1, \ldots, n$ in the following. Let $\mathcal{P}(B_N) = \{\phi_1(v, \star), \ldots, \phi_n(v, \star)\}$ where other free variables mentioned by the ϕ_i are denoted by $\star$. Each $\phi_i(v, \star)$ is equivalent to $\bigvee_{j=1}^{m_i} \psi^i_j(v, \star)$ where the formula $\psi^i_j(v, \star)$ is the conjunction of the formulas associated with the decision nodes along the jth path from the root node of B_N to a terminal node with the label i. Then, $\exists v.\, \phi_i(v, \star) \equiv \bigvee_{j=1}^{m_i} \exists v.\, \psi^i_j(v, \star)$.

   (b) Construct a new BDD incrementally over the $\{\phi_i(v, \star)\}$: For every $i \in \{1, \ldots, n\}$, B_i^C denotes a BDD that represents ϕ_i, $C \subseteq \{1, \ldots, n\}$, and it looks like this:

$$\begin{tikzcd}
\exists v.\,\psi_1^i(v,\star) \arrow[dr, dotted] \arrow[ddddr] & & & \\
& \exists v.\,\psi_2^i(v,\star) \arrow[dr, dotted] \arrow[ddd] & & \\
& & \cdots \arrow[dr, dotted] \arrow[ddl] & \\
& & & \exists v.\,\psi_{m_i}^i(v,\star) \arrow[dll] \arrow[d, dotted] \\
& \diamond_{i+1}^{C\cup\{i\}} & & \diamond_{i+1}^{C}
\end{tikzcd}$$

The idea behind the interim nodes $\diamond_{i+1}^{C\cup\{i\}}$ and $\diamond_{i+1}^{C}$ in B_i^C is to memorize the index for the next iteration $(i+1)$ and the indices of the ϕ_j for which a satisfying v exists. More precisely, at a node $\diamond_i^C$ it is known that for each $j \in C$ the formula $\exists v.\,\phi_j(v,\star)$ holds. Note, that in B_i^C the set of indices for the left interim node (where at least one of the $\exists v.\,\psi_j^i(v,\star)$ holds) is $C \cup \{i\}$.

Now, the new BDD is constructed by starting with $B_1^\emptyset$ and then iteratively replacing every occurrence of an interim node $\diamond_i^C$ with $i \leq n$ by the BDD B_i^C.

(c) After this process is completed for all ϕ_i the remaining interim nodes $\diamond_{n+1}^C$ are replaced by terminal nodes labeled C. The resulting BDD is called B'_N.

(d) Replace the sub-BDD B_N in B' with B'_N.

An example is given in Figure 2. There, $\psi_1^1(v) = \neg\vartheta(v)$, $\psi_2^1(v) = \vartheta(v) \wedge \varphi(v)$, and $\psi_1^2(v) = \vartheta(v) \wedge \neg\varphi(v)$.

The implementation of the $\mathbf{\exists}_v$-operator for BDDs picks up the idea presented above (cf. Section 4.1) to reduce the number of (trivially) unsatisfiable formulas in the partitions generated by the $\mathbf{\exists}_v$-operator for partitions. The common sub-formulas can be easily identified due to the representation as a BDD and the exponential blow-up in the size of the partition/BDD is diminished by moving down the decision nodes in the BDD which are associated with formulas that mention v as a free variable. Let $|\mathcal{P}(B)| = n$ and let there by m sub-BDDs B_{N_i} (cf. step 2 in the algorithm) with $|\mathcal{P}(B_{N_i})| = n_i$. Then, the number of formulas in the partition represented by the BDD $\mathbf{\exists}_v\, B$ is $n + \sum_{i=1}^{m} 2^{n_i} - n_i$.

7.1 Computing the Induced BDDs

With the previously defined operators at hand the BDD induced by a Golog program δ can be defined. In particular, we show how to directly compute the BDD representing the partition induced by a program. With the exception of the $\mathbf{\exists}_v$-operator and the regression of all formulas in the partition this can be achieved by directly operating on BDDs.

In the following we use a number of abbreviations:

- $\mathcal{B}^{rew}$ is an abbreviation for $\mathcal{B}(\mathcal{P}^{rew})$ and
- $\mathcal{B}(\delta)$ is an abbreviation for $\mathcal{B}(\mathcal{P}(\delta))$.

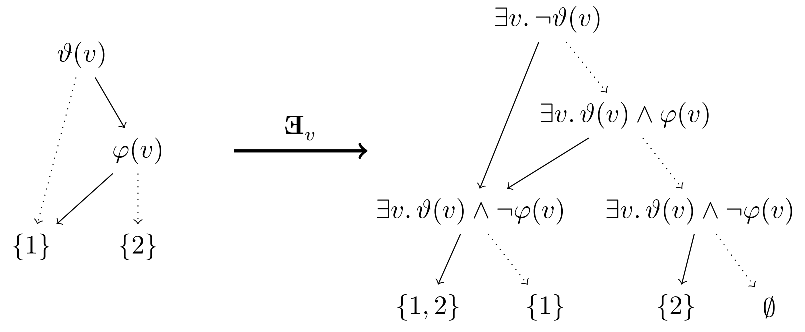

$\vartheta(v)$
$\varphi(v)$
$\{1\}$
$\{2\}$
$\mathfrak{I}_v$
$\exists v.\, \neg\vartheta(v)$
$\exists v.\, \vartheta(v) \wedge \varphi(v)$
$\exists v.\, \vartheta(v) \wedge \neg\varphi(v)$
$\exists v.\, \vartheta(v) \wedge \neg\varphi(v)$
$\{1,2\}$
$\{1\}$
$\{2\}$
$\emptyset$

FIG. 2: Illustration of the BDD produced by the $\mathfrak{I}_v$-operator. The input BDD is shown on the left side; it represents the partition $\{\neg\vartheta(v) \vee \psi(v), \vartheta(v) \wedge \neg\psi(v)\}$.

Analogously to the definition of $\mathcal{P}(nil)$, the BDD induced by the empty program is defined to be the BDD representing the reward partition:

$$\mathcal{B}(nil) = \mathcal{B}^{rew}$$

The BDD induced by a program starting with a primitive action a is defined as:

$$\mathcal{B}([a;\delta]) = B' \oslash^{Poss(a,s)} \{\star\}$$

where $B' = \mathcal{B}^{rew} \otimes \mathcal{B}(\{Regr(\phi[do(a,s)]) \mid \phi \in \mathcal{P}(\mathcal{B}^{\delta})\}$ and $\{\star\}$ is a terminal node with a new label $\star$.

For the remaining language constructs the function $\mathcal{B}(\delta)$ is defined in analogy to $\mathcal{P}(\delta)$.

- The program starts with a stochastic action a_{st} which has the outcomes $n_1, \ldots, n_k$ and the partitions induced by the conditions under which the probability distribution over the outcomes varies is $\mathcal{P}^{pr}_{a_{st}}$:

$$\mathcal{B}([a_{st};\delta]) = \mathcal{B}(\mathcal{P}^{pr}_{a_{st}}) \otimes \bigotimes_{i=1}^{k} \mathcal{B}([n_i;\delta])$$

- The program starts with a test action $\vartheta?$:

$$\mathcal{B}([\vartheta?;\delta]) = \mathcal{B}(\delta) \oslash^{\vartheta} \{\star\}$$

  Again, the $\star$ denotes a “fresh” label.
- The program starts with a conditional branching statement:

$$\mathcal{B}([\textbf{if } \vartheta \textbf{ then } \delta_1 \textbf{ else } \delta_2;\delta]) = \mathcal{B}([\delta_1;\delta]) \oslash^{\vartheta} \mathcal{B}([\delta_2;\delta])$$

- The program starts with a nondeterministic branching:

$$\mathcal{B}([\textbf{nondet}(\delta_1, \ldots, \delta_n);\delta]) = \bigotimes_{i=1}^{n} \mathcal{B}([\delta_i;\delta])$$

- The program starts with a nondeterministic choice of argument:

$$\mathcal{B}([\mathbf{pick}(v,\eta);\delta]) = \mathfrak{A}_v\, \mathcal{B}([\eta;\delta])$$

- The program starts with a procedure call:

$$\mathcal{B}([P(\vec{t});\delta]) = \mathcal{B}([\delta_P{}^{\vec{x}}_{\vec{t}};\delta])$$

8 The QGolog Interpreter

The programs we consider may contain nondeterministic constructs, the nondeterministic branching and the nondeterministic choice of arguments, in particular. These are the *choice points* in the program—here the agent needs to make a decision on how to proceed with the execution of the program. Our intention is to learn what the best decisions at the choice points of a program are by integrating Q-learning into the action language Golog.

The partition induced by a program beginning with a nondeterministic construct allows for first-order state as well as action abstraction. The formulas in the partition induced by a program differentiate situations in which the expected reward for executing the program differs. For instance, assume the program $\mathbf{nondet}(\delta_1,\ldots,\delta_n);\delta$ that allows the agent to first follow one of the δ_i, $1 \le i \le n$, and then the program δ. Let $\{\phi_1,\ldots,\phi_m\}$ be the partition induced by that program. Since we know that the ϕ_j distinguish situations in which any possible continuation of the program leads to different expected rewards it is not necessary to maintain a Q-table that contains entries for choosing to continue with δ_i in any possible ground state. Instead it suffices to store Q-values for all combinations of δ_i and ϕ_j.

Action abstraction (together with state abstraction) can be achieved for programs beginning with a nondeterministic choice of argument, e.g. $\mathbf{pick}(v,\eta);\delta$. Here, the partition induced by the program frees the agent from having entries in the Q-table for every possible choice for v. Not only that there might be infinitely many, but it might not always be known in advance what choices are available. Assume $\{\psi_1(v),\ldots,\psi_l(v)\}$ is the partition induced by the program $\eta;\delta$. Then, choosing a v that satisfies ψ_1 potentially leads to another expected reward than choosing a v that satisfies ψ_2. The Q-table can thus be reduced to combinations of the abstract states given by $\mathfrak{A}_v\,\{\psi_1(v),\ldots,\psi_l(v)\}$ (cf. Eq. 5.3) and the abstract choices for v given by $\psi_1(v),\ldots,\psi_l(v)$. Not in every state described by a formula in $\mathfrak{A}_v\,\{\psi_1(v),\ldots,\psi_l(v)\}$ it is possible to choose a v satisfying each of the $\psi_i(v),\ldots,\psi_l(v)$—it might be the case that there exists no v satisfying a certain ψ_i. Since the formulas in $\mathfrak{A}_v\,\{\psi_1(v),\ldots,\psi_l(v)\}$ make assumptions about the existence and nonexistence of v's satisfying the $\psi_i(v)$ we only allow the agent to choose a v satisfying a ψ_i if the (unique) formula in $\mathfrak{A}_v\,\{\psi_1(v),\ldots,\psi_l(v)\}$ that holds in the current situation guarantees the existence of a v satisfying ψ_i. Formally, if $\phi[\sigma]$ holds with $\phi \in \mathfrak{A}_v\,\{\psi_1(v),\ldots,\psi_l(v)\}$ and $\phi \models \exists v.\, \psi_i(v)$ for an i with $1 \le i \le l$ then choosing a v satisfying ψ_i is a licit choice.

Whereas for programs that begin with a nondeterministic branching the possible choices can be read off of the program this is not that simple for programs beginning with a nondeterministic choice of argument. But with the BDD-based representation of the induced partitions presented in Section 7 this can be achieved fairly easily. Remember that the terminal nodes in the BDD induced by a program $[\mathbf{pick}(v,\eta);\delta]$

are labeled with subsets of the labels of the terminal nodes of the BDD induced by the program $[\eta; \delta]$. Precisely, these sets contain the labels of the terminal nodes which correspond to formulas in the partition induced by $[\eta; \delta]$ for which a satisfying v exists.

We only intend to learn Q-values for the choices that the agent can make at the choice points of a program—in between two successive choice points no decisions need to be made since the program is deterministic. Consequently, the execution of the program can be regarded as transitioning from one choice point to the next, making a decision at each of these. Due to stochastic actions the transitions from one choice point to the next may be probabilistic. Semi-MDPs (SMDPs) are like MDPs only that they additionally consider the duration between state changes. In our case the number of actions (deterministic or stochastic) may vary between choice points. In order to take this into account we make use of the SMDP-version of the Q-update:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \Big(R_t + \gamma^k \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\Big)$$

The reward R_t obtained for executing the SMDP-action a_t (not to be confused with a situation calculus action) in state s_t is computed as the discounted sum of rewards obtained for executing the k actions in the sequence that let from the previous choice point to the current choice point:

$$R_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \cdots + \gamma^k \cdot r_{t+k-1}$$

where the r_i correspond to the value of the reward function $rew(do([a_1, \ldots, a_{i-1}], \sigma))$ with σ being the situation at the previous choice point.

Contrary to the general (S)MDP-setting where the agent may freely choose between all available actions, the choices are limited by the given program the agent has to follow in our setting. Consequently, we need to store Q-values for all triplets δ, ϕ, and A where δ is the remaining program, $\phi \in \mathcal{P}(\delta)$, and A identifies one of the possible ways to continue with the execution of δ in a situation where ϕ holds. If δ begins with $\mathbf{nondet}(\delta_1, \ldots, \delta_n)$ then A can take on the values $\delta_1, \ldots, \delta_n$; if δ begins with $\mathbf{pick}(v, \eta)$ then A is of the form $\mathbf{pick}(v, [\varphi?; \eta])$ where φ is a formula that describes one of the licit choices for v. Formally, the set of possible choices for a choice point δ if $\phi \in \mathcal{P}(\delta)$ holds in the current situation is defined by $\mathcal{A}(\delta, \phi)$:

- $\mathcal{A}(\mathbf{nondet}(\delta_1, \ldots, \delta_n); \delta, \phi) = \{\delta_1, \ldots, \delta_n\}$
- $\mathcal{A}(\mathbf{pick}(v, \eta); \delta, \phi) = \{\mathbf{pick}(v, [\varphi?; \eta]) \mid \varphi \in \mathcal{P}([\eta; \delta]) \text{ and } \phi \models \exists v.\, \varphi\}$

The interpreter manages the Q-table by means of the fluent $q(\phi, \delta, A, v, s)$. It states that the Q-value of proceeding with the remaining program δ according to A if ϕ holds in the current situation s, $\phi \in \mathcal{P}(\delta)$, is v. With the action $setQ(\phi, \delta, A, v)$ the entries in the Q-table can be manipulated. The successor state axiom for the q-fluent is defined as

$$\begin{aligned} q(\delta, \phi, A, v, do(a, s)) \equiv a = setQ(\delta, \phi, A, v) & \\ \vee\, q(\delta, \phi, A, v) \wedge \neg\exists v'.\, a = setQ(\delta, \phi, A, v') & \end{aligned}$$

We assume that the q-fluent is properly initialized in $\mathcal{D}_{S_0}$. That is, for every program δ which begins with a nondeterministic construct and is a remaining program of the program for which the optimal execution shall be learned, for every $\phi \in \mathcal{P}(\delta)$, and for every possibility A to continue with δ, the q-fluent holds for a unique, numerical constant v in S_0. Precisely, we assume that we have sentences of the following form in $\mathcal{D}_{S_0}$ for every such δ and ϕ

$$q(\delta, \phi, A, v, S_0) \equiv \bigvee_{A' \in \{A_1, \ldots, A_n\}} A = A' \wedge v = v_{A'}$$

where $A_1, \ldots, A_n$ are the possible choices to continue with the program δ in a situation where ϕ holds and the v_{A_i} are numerical constants.

We introduce the new language construct **learn**(δ) that resolves the nondeterminism in δ and records how good the choices are that have been made. The semantics of QGolog programs is mostly defined with the help of the *QDo* macro:

$$QDo(\delta^*, \phi^*, A^*, \delta, s, r, k, \pi)$$

where δ^* is the remaining program at the last encountered choice point, $\phi^* \in \mathcal{P}(\delta^*)$ held in the situation current at the last choice point, and A^* is the choice the agent made at the previous choice point. These values have to be remembered for the next update of the respective Q-value. The remaining arguments of the *QDo*-macro are the remaining program δ, the current situation s, the reward r accumulated since the last choice point, the number of primitive actions k performed since the last choice point, and the computed policy π. The *Trans*-predicate for **learn**(δ) is defined as:

$$Trans(\mathbf{learn}(\delta), s, \delta', s') \equiv s' = s \wedge QDo(\delta, \phi_{start}, A_{start}, \delta, s, 0, 0, \delta')$$

That is, the program may proceed from a configuration $\langle \mathbf{learn}(\delta), s\rangle$ to a configuration $\langle \delta', s'\rangle$ where $s' = s$ and δ' is the policy computed by the *QDo*-macro. Since not every program has to begin with a choice point we add a distinguished start state ϕ_{start} (for instance, $\phi_{start} = true$) and start choice A_{start}. We also assume that the q-fluent is properly initialized in $\mathcal{D}_{S_0}$ for the parameters δ, ϕ_{start}, and A_{start}.

The predicate *QDo* is defined depending on the beginning of the remaining program. If the remaining program is the empty program *nil* everything that remains to be done is to compute the Q-update for the choice made at the last choice point and add a corresponding action to the policy:

$$\begin{aligned}
&QDo(\delta^*, \phi^*, A^*, nil, s, k, r, \pi) \overset{def.}{=} \\
&\quad \exists q_t, q_{t+1}.\, q(\delta^*, \phi^*, A^*, q_t, s) \wedge q_{t+1} = q_t + \alpha \cdot (r - q_t) \\
&\quad \wedge \pi = setQ(\delta^*, \phi^*, A^*, q_{t+1})
\end{aligned}$$

In case the remaining program starts with a primitive, deterministic action and the preconditions for that action do hold in the current situation, the remaining policy π' is determined by *QDo* where the number of actions performed since the last choice point increased by one and the reward obtained in the successor situation $do(a, s)$ is added to the accumulated reward. If the preconditions are not given in the current situation a negative reward r_{fail} is obtained. It is necessary to choose a value for r_{fail} such that every successful execution of the program leads to a higher accumulated reward than any unsuccessful one.

$$\begin{aligned}
&QDo(\delta^*, \phi^*, A^*, [a; \delta], s, k, r, \pi) \overset{def.}{=} \\
&\quad Poss(a, s) \land \exists r', \pi'.\, r' = rew(do(a, s)) \\
&\quad \land QDo(\delta^*, \phi^*, A^*, \delta, do(a, s), k + 1, r + \gamma^k \cdot r', \pi') \land \pi = [a; \pi'] \\
&\quad \lor \neg Poss(a, s) \land \exists q_t, q_{t+1}.\, q(\delta^*, \phi^*, A^*, q_t, s) \\
&\quad \land q_{t+1} = q_t + \alpha \cdot (r + \gamma^k \cdot r_{fail} - q_t) \\
&\quad \land \pi = setQ(\delta^*, \phi^*, A^*, q_{t+1})
\end{aligned}$$

If the program starts with a test action and the test holds QDo is called recursively for the remaining program with unchanged parameters for s, k, and r. Otherwise the choice made at the last choice point is penalized in the same fashion as above.

$$\begin{aligned}
&QDo(\delta^*, \phi^*, A^*, [\vartheta?; \delta], s, k, r, \pi) \overset{def.}{=} \\
&\quad \vartheta[s] \land QDo(\delta^*, \phi^*, A^*, \delta, s, k, r, \pi) \\
&\quad \lor \neg\vartheta[s] \land \exists q_t, q_{t+1}.\, q(\delta^*, \phi^*, A^*, q_t, s) \\
&\quad \land q_{t+1} = q_t + \alpha \cdot (r + \gamma^k \cdot r_{fail} - q_t) \\
&\quad \land \pi = setQ(\delta^*, \phi^*, Q^*, q_{t+1})
\end{aligned}$$

The policy generated for programs that start with a stochastic action are somewhat different. Since the outcome of the stochastic action is not known in advance the computation of the policy needs to be interrupted, the stochastic action needs to be executed online, and the actual outcome needs to be observed. Afterwards the computation of the policy needs to be picked up again. In order to perform the next Q-update it is necessary to remember the choice made at the last choice point, the number of executed actions, and the accumulated reward. This is exactly what the language construct **l** is for. It is similar to **learn**(δ) only that it allows to additionally specify the aforementioned parameters.

$$Trans(\mathbf{l}(\delta^*, \phi^*, A^*, \delta, s, k, r), s, \delta', s') \equiv QDo(\delta^*, \phi^*, A^*, \delta, s, k, r, \delta') \land s = s'$$

With this the QDo-macro for programs beginning with a stochastic action can be defined as:

$$\begin{aligned}
&QDo(\delta^*, \phi^*, A^*, [a_{st}; \delta], s, k, r, \pi) \overset{def.}{=} \\
&\quad \exists r_1, \ldots, r_k.\, \bigwedge_{i=1}^{k} r_i = rew(do(n_i, s)) \\
&\quad \pi = [SR(a_{st}); \textbf{if } senseCond(n_1) \textbf{ then} \\
&\qquad\qquad \mathbf{l}(\delta^*, \phi^*, A^*, \delta, k + 1, r + \gamma^k \cdot r_1) \\
&\qquad\quad \textbf{else if } senseCond(n_2) \textbf{ then} \\
&\qquad\qquad \ldots \\
&\qquad\quad \textbf{else if } senseCond(n_k) \textbf{ then} \\
&\qquad\qquad \mathbf{l}(\delta^*, \phi^*, A^*, \delta, k + 1, r + \gamma^k \cdot r_k)]
\end{aligned}$$

The policy π begins with the function $SR(a_{st})$ that returns one of the defined outcomes n_i:

$$SR(a_{st}(\vec{x}), s) = r \equiv r = n_1 \vee \ldots \vee r = n_k$$

During the execution in the real world $SR(a_{st})$ is replaced by the observed outcome (cf. Section 8.1).

The definition of *QDo* for programs that start with a conditional branching is straightforward:

$$\begin{aligned}
&QDo(\delta^*, \phi^*, A^*, [\textbf{if } \vartheta \textbf{ then } \delta_1 \textbf{ else } \delta_2; \delta], s, k, r, \pi) \stackrel{def.}{=} \\
&\quad \vartheta[s] \wedge QDo(\delta^*, \phi^*, A^*, [\delta_1; \delta], s, k, r, \pi) \\
&\quad \vee \neg\vartheta[s] \wedge QDo(\delta^*, \phi^*, A^*, [\delta_2; \delta], s, k, r, \pi)
\end{aligned}$$

as it is the case for programs starting with a procedure call:

$$\begin{aligned}
&QDo(\phi^*, \delta^*, A^*, [P(\vec{t}); \delta], s, k, r, \pi) \stackrel{def.}{=} \\
&\quad QDo(\phi^*, \delta^*, A^*, [\delta_P{}^{\vec{x}}_{\vec{t}}; \delta], s, k, r, \pi)
\end{aligned}$$

where the procedure is defined as **proc** $P(\vec{x})\, \delta_P$.

The remaining two cases, namely programs that begin with either a nondeterministic branching or a nondeterministic choice of argument, represent the choice points in the program. Here, a Q-update needs to be performed for the Q-value of the choice made at the last choice point. In both cases *QDo* first determines which of the formulas in the induced partition holds in the current situation and then updates the Q-value for the choice made at the previous choice point. The latter consists of looking up the current Q-value of the choice made at the previous choice point, determining the maximizing choice for the current choice point and its Q-value, and to compute the updated Q-value for the choice made at the last choice point. *QDo* is then called recursively with the current program, the formula in the induced partition that holds in the current situation, and the choice made for the current choice point. Here, the interpreter follows a greedy policy, i.e., it always chooses the maximizing action. How state space exploration can be incorporated into the interpreter is shown in Section 8.2.

$$\begin{aligned}
&QDo(\delta^*, \phi^*, A^*, [\textbf{nondet}(\delta_1, \ldots, \delta_n); \delta], s, k, r, \pi) \stackrel{def.}{=} \\
&\quad \bigvee_{\phi \in \mathcal{P}([\textbf{nondet}(\delta_1, \ldots, \delta_n); \delta])} \phi[s] \wedge \exists q_t.\, q(\delta^*, \phi^*, A^*, q_t, s) \\
&\quad \wedge \exists A, q_{max}.\, QMax([\textbf{nondet}(\delta_1, \ldots, \delta_n); \delta], \phi, A, q_{max}, s) \\
&\quad \wedge \exists q_{t+1}.\, q_{t+1} = q_t + \alpha \cdot (r + \gamma^k \cdot q_{max} - q_t) \wedge \\
&\quad \wedge QDo([\textbf{nondet}(\delta_1, \ldots, \delta_n); \delta], \phi, A, [\delta_A; \delta], s, 0, 0, \pi') \\
&\quad \wedge \pi = [setQ(\delta^*, \phi^*, A^*, q_{t+1}); \pi']
\end{aligned}$$

$QMax(\phi, \delta, A, q_{max}, s)$ is an auxiliary macro that determines the choice A maximizing the Q-value that can be made at the choice point δ if ϕ holds and its Q-value q_{max}.

The definition is given below.

$$QMax(\delta,\phi,A,q_{max},s) \overset{def.}{=}$$
$$q(\delta,\phi,A,q_{max},s) \wedge \bigwedge_{\substack{B\in\mathcal{A}(\delta,\phi),\\ B\neq A}} \exists q_B.\, q(\delta,\phi,B,q_B,s) \wedge q_B \leq q_{max}$$

Note that the choices for a choice point beginning with a nondeterministic choice of argument are identified by Golog programs of the form $\mathbf{pick}(v,[\vartheta?;\eta]);\delta)$. By computing a $Trans$-step for such a program the interpreter binds the variable v in η such that ϑ holds. Particularly, the $Trans$ step for the program $[\mathbf{pick}(v,[\vartheta?;\eta]);\delta]$ is computed by finding an adequate binding x for v such that there exists a $Trans$-step for $[\vartheta?;\eta^v_x]$ and thus the x has to satisfy ϑ. Compare with the definition of the $Trans$ predicate:

$$Trans(\mathbf{pick}(v,\eta),s,\delta',s') \equiv \exists x.\, Trans(\eta^v_x,s,\delta',s')$$

The policy π first updates the q fluent and then continues with the policy π' computed for the remaining program which is computed by recursively calling QDo.

$$QDo(\delta^*,\phi^*,A^*,[\mathbf{pick}(v,\eta);\delta],s,k,r,\pi) \overset{def.}{=}$$
$$\bigvee_{\phi\in\mathcal{P}([\mathbf{pick}(v,\eta);\delta])} \phi[s] \wedge \exists q_t.\, q(\delta^*,\phi^*,A^*,q_t,s)$$
$$\wedge \exists A,q_{max}.\, QMax([\mathbf{pick}(v,\eta);\delta],\phi,A,q_{max},s)$$
$$\wedge \exists q_{t+1}.\, q_{t+1} = q_t + \alpha\cdot(r+\gamma^k\cdot q_{max} - q_t)\wedge$$
$$\wedge \exists \delta'.\, Trans(A,s,\delta',s)$$
$$\wedge\, QDo([\mathbf{pick}(v,\eta);\delta],\phi,A,\delta',s,0,0,\pi')$$
$$\wedge\, \pi = [setQ(\delta^*,\phi^*,A^*,q_{t+1});\pi']$$

Since it is necessary to observe the execution of a program δ in the real world several times in order to learn the optimal execution of the program this either requires to embed the learning in a loop:

while φ **do learn**(δ) **end**

or to progress the fluents relevant to for the Q-learning (especially the q fluent and others; see below) to the situation that was reached after executing the program and replace any occurrence of these fluents in another $\mathcal{D}'_{S_0}$ with the progressed fluents. Since we assumed a completely specified $\mathcal{D}_{S_0}$ the progression is computable (cf. [17]). Note, that since the Q-values are accessible within the program it is possible to formulate an exit condition for the loop above that depends on the current Q-values. e.g.:

while $\exists v.\, q(\delta,\phi_{start},A_{start},v) \wedge v < 5.0$ **do learn**(δ) **end**

8.1 Online Execution

The online execution of a QGolog program is performed similarly to [24, 9]. As long as the program is not final in the current situation a Trans-step $Trans(\delta,s,\delta',s')$ is computed and if $\exists a.\, s' = do(a, s)$ then the action a is executed in the real world. An exception is if $\delta = [SR(a_{st}); \delta']$. Then, the stochastic action a_{st} is executed in the real world and by sensing the resulting state of the world the outcome of a_{st} that models those changes is determined. Assume that after executing a_{st} in situation s in the real world (at least as far as the agent perceives it) is in a state that coincides with $do(n_i, s)$ where n_i is a modeled outcome of a_{st}. In the next step the execution system then computes Trans-step for the remaining program δ' in situation $do(n_i, s)$: $Trans(\delta', do(n_i, s), \delta'', s'')$

8.2 State Space Exploration

The interpreter presented above follows a greedy policy, i.e., at every choice point it chooses to continue with the execution of the program according to the choice that maximizes the Q-function for the current choice point. In order to prove the convergence of the Q-learning process it is required, however, that with a certain probability that decreases over time a non-maximizing choice is selected. This encourages the exploration of the state space and ensures that, eventually, every possible choice at every choice point is tried out infinitely many times.

In order to integrate the state space exploration into the interpreter we introduce the fluent $\epsilon(s)$ storing the probability with which a non-maximizing choice is selected. The action *setEpsilon*(v) allows to manipulate the fluent $\epsilon(s)$ similar to the *setQ*-action and the q-fluent. Additionally, we assume that the sensing action *senseRnd* accesses a random number generator and sets the fluent $rnd(s)$ to a value between 0 and 1.

To incorporate the state space exploration the definition of the QDo macro needs to be modified for programs that start with either a nondeterministic branching or a nondeterministic choice of argument. Therefore, we introduce two new macros that compute programs that depending on the value of the fluent $rnd(s)$ select one of the possible choices. More specifically, the remaining program $\mathbf{l}(\delta, \phi, A_i, \delta, 0, 0)$ reflects the selection of the choice A_i. The definition of the macros is similar to the definition of the $BestDoAux$ macro in [5].

$$\begin{aligned}
&chooseSmdpActionNdet(\{A_1, \ldots, A_n\}, \phi, \delta, i, \pi) \overset{def.}{=} \\
&\quad chooseSmdpActionNdet(\{A_1, \ldots, A_{n-1}\}, \phi, \delta, i+1, \pi') \\
&\quad \wedge \pi = \mathbf{if}\ rnd \leq \frac{i}{n+i-1}\ \mathbf{then}\ \mathbf{l}(\phi, \delta, A_n, \delta, 0, 0)\ \mathbf{else}\ \pi'
\end{aligned}$$

When called with the empty set as the first argument, *chooseSmdpActionNdet* as well as *chooseSmdpActionPick* (see below) return $\pi = nil$.

The macro that generates a program to select a choice for a choice point that begins with a nondeterministic choice of arguments is only slightly different. It additionally determines the remaining program δ' by computing a $Trans$ step on the program that is associated with the choice A_i.

$$\begin{aligned}
&\mathit{chooseSmdpActionPick}(\{A_1,\ldots,A_n\},\phi,\delta,i,s,\pi) \overset{def.}{=} \\
&\quad \mathit{Trans}(A_n,s,\delta',s) \\
&\quad \wedge \mathit{chooseSmdpActionPick}(\{A_1,\ldots,A_{n-1}\},\phi,\delta,i+1,s,\pi') \\
&\quad \wedge \pi = \textbf{if } \mathit{rnd} \leq \frac{i}{n+i-1} \textbf{ then } \mathbf{l}(\phi,\delta,A_n,\delta',0,0) \textbf{ else } \pi'
\end{aligned}$$

Lastly, the QDo macro for the nondeterministic branching and the nondeterministic choice of argument needs to be modified. The set of choices $\{A_1,\ldots,A_n\}$ corresponds to $\mathcal{A}([\textbf{nondet}(\delta_1,\ldots,\delta_n);\delta],\phi)$ and $\mathcal{A}([\textbf{pick}(v,\eta);\delta],\phi)$, respectively.

$$\begin{aligned}
&QDo(\delta^*,\phi^*,A^*,[\textbf{nondet}(\delta_1,\ldots,\delta_n);\delta],s,k,r,\pi) \overset{def.}{=} \\
&\quad \bigvee_{\phi\in\mathcal{P}([\textbf{nondet}(\delta_1,\ldots,\delta_n);\delta])} \phi[s] \wedge \exists q_t.\, q(\delta^*,\phi^*,A^*,q_t,s) \\
&\quad \wedge \exists A, q_{max}.\, QMax([\textbf{nondet}(\delta_1,\ldots,\delta_n);\delta],\phi,A,q_{max},s) \\
&\quad \wedge \exists q_{t+1}.\, q_{t+1} = q_t + \alpha\cdot(r+\gamma^k\cdot q_{max} - q_t)\wedge \\
&\quad \wedge QDo([\textbf{nondet}(\delta_1,\ldots,\delta_n);\delta],\phi,A,[A;\delta],s,0,0,\pi') \\
&\quad \wedge \mathit{chooseSmdpActionNdet}(\{A_1,\ldots,A_n\},\phi,[\textbf{nondet}(\delta_1,\ldots,\delta_n);\delta],1,\pi'') \\
&\quad \wedge \pi = [\mathit{setQ}(\delta^*,\phi^*,A^*,q_{t+1}); \mathit{senseRnd}; \\
&\qquad \textbf{if } \mathit{rnd} > \epsilon \textbf{ then } \pi'' \textbf{ else } \pi']
\end{aligned}$$

$$\begin{aligned}
&QDo(\delta^*,\phi^*,A^*,[\textbf{pick}(v,\eta);\delta],s,k,r,\pi) \overset{def.}{=} \\
&\quad \bigvee_{\phi\in\mathcal{P}([\textbf{pick}(v,\eta);\delta])} \phi[s] \wedge \exists q_t.\, q(\phi^*,\delta^*,A^*,q_t,s) \\
&\quad \wedge \exists A, q_{max}.\, QMax([\textbf{pick}(v,\eta);\delta],\phi,A,q_{max},s) \\
&\quad \wedge \exists q_{t+1}.\, q_{t+1} = q_t + \alpha\cdot(r+\gamma^k\cdot q_{max} - q_t)\wedge \\
&\quad \wedge \exists\delta'.\, \mathit{Trans}(A,s,\delta',s) \\
&\quad \wedge QDo([\textbf{pick}(v,\eta);\delta],\phi,A,\delta',s,0,0,\pi') \\
&\quad \wedge \mathit{chooseSmdpActionPick}(\{A_1,\ldots,A_n\},\phi,[\textbf{pick}(v,\eta);\delta],1,s,\pi'') \\
&\quad \wedge \pi = [\mathit{setQ}(\delta^*,\phi^*,A^*,q_{t+1}); \mathit{senseRnd}; \\
&\qquad \textbf{if } \mathit{rnd} > \epsilon \textbf{ then } \pi'' \textbf{ else } \pi']
\end{aligned}$$

The above extension of the interpreter implements ϵ-greedy action selection. In a similar fashion other action selection methods can be realized (e.g. softmax action selection; cf. [25]).

9 Evaluation

We implemented the QGolog-interpreter using the BDD representation for partitions and a small simulation environment to actually run programs and simulate the interaction with the environment in ECLiPSe prolog. The purpose of this evaluation is twofold. One, we want to examine whether the abstraction mechanisms in QGolog facilitates learning in domains with large state spaces. Two, we intend to explore the practicability of the BDD-based representation and manipulation of first-order formulas. Therefor, we performed experiments in two different domains, the blocksworld and the logistics domain. In the latter boxes can be transported from one city to another by loading them on trucks, driving the trucks to the other city, and unloading the boxes there.

9.1 Blocksworld

For the experiments we extended the formalization of the blocksworld scenario. In particular we added the stochastic action $move_{st}(x, y)$. Its outcomes are described by the (deterministic) action $move(x, y)$ and by a noop-action. That is, moving one block on top of another block might fail and in that case nothing changes. Formally:

$$choice(move_{st}(x, y), a) \;\equiv\; a = move(x, y) \vee a = noop.$$

The probability distribution over the outcome actions depends on whether the box to be moved is heavy or not:

$$\mathcal{P}^{pr}_{move_{st}(x,y)} = \{\; heavy(x), \neg heavy(x) \;\}$$

In the (simulated) execution the action $move_{st}(x, y)$ failed with a probability of 0.1 if the box x is not heavy and with a probability of 0.9 if the box x is heavy.

The QGolog-program we ran is the following:

$$\mathbf{learn}(\mathbf{pick}(x, \mathbf{pick}(y, move_{st}(x, y))))$$

Although, quite simple it allows to assess the effectiveness of the state space as well as the action space abstraction mechanisms. In a domain instance with n blocks a flat Q-learner would need to consider up to n^2 (ground) actions and a state space exponential in n. The partition induced by the reward function differentiates situation in which there exists a green block on top of a red block from those where there is no green block on top of a red block:

$$\begin{aligned}\mathcal{P}^{rew} = \{\; & \exists x, y.\, green(x) \wedge red(y) \wedge on(x, y),\\ & \neg\exists x, y.\, green(x) \wedge red(y) \wedge on(x, y) \;\}\end{aligned}$$

The agent received a reward of 5 if there is a green block on top of red block and -1 otherwise.

Our (non-optimized) implementation generated a BDD representing $\mathcal{P}(\mathbf{pick}(x, \mathbf{pick}(y, move_{st}(x, y))))$ which consists of 505 decision and 504 terminal nodes. That is, by exploiting common sub-structures in the formulas (cf. Sect. 4.1) we could effectively reduce the number of formulas in $\mathcal{P}(\mathbf{pick}(x, \mathbf{pick}(y, move_{st}(x, y))))$ from its theoretical maximum of 2^{32} to 504. By optimizing the orderings of the nodes in the BDDs this number can be reduced even further.

During learning we ran the program given above in randomly generated domain instances—for every iteration we used a previously unseen instance. In every instance

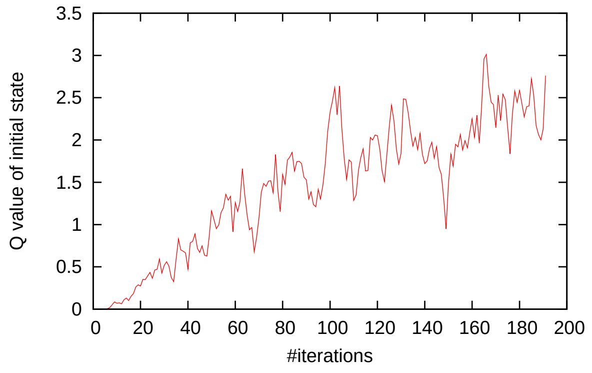


FIG. 3: The result of executing the program $\mathbf{learn}(\mathbf{pick}(x, \mathbf{pick}(y, move_{st}(x, y))))$ again and again, every time in a previously unseen instance of the domain. The results are average over 10 runs.

there are 5 to 50 blocks arbitrarily arranged on the table. A block can either be red, green, or blue. It has to be noted that not in each of the generated instances a situation in which there is a green on top of a red block can be reached by executing the program. In this case the best choice for the agent is to select any two blocks that can be moved on top of each other. The learning rate as well as the exploration probability were set to 0.1; the discount factor was set to 0.9. These values were kept constant throughout the experiments.

In Fig. 3 the development of the Q-value for the distinguished initial state $(\phi_{start}, \delta_{start})$ and the start action A_{start} is shown. The results are averaged over 10 runs. One can see that after only 10 iterations on average the agent starts to make the "right" decisions. A flat Q-learner would be absolutely chanceless here, since in every iteration a new instance of the domain is presented to the agent.

9.2 Logistics Domain

The objective in our version of the logistics domain was to have a box in city c_1, i.e., in situation where there is a box in c_1 a reward of 5 is obtained; else the reward is -1. We encoded this in the program which allows the agent to choose between loading a box on a truck, driving that truck to c_1, and unloading the box there or to skip the first step and directly drive a truck to c_1 or to just unload a box from a truck:

$$\begin{aligned}\mathbf{learn}(\mathbf{nondet}(&\mathbf{pick}(b, \mathbf{pick}(t, [load(b,t), drive(t,c_1), unload(b,t)])),\\ &\mathbf{pick}(b, \mathbf{pick}(t, [drive(t,c_1), unload(b,t)])),\\ &\mathbf{pick}(b, \mathbf{pick}(t, unload(b,t)))))\end{aligned}$$

A box can be loaded on a truck iff both are in the same city:

$$Poss(load(b,t), s) \equiv \exists c.\, boxIn(b,c,s) \wedge truckIn(t,c,s)$$

A box can be unloaded from a truck iff the box is on the truck:

$$Poss(unload(b,t), s) \equiv loaded(b,t,s)$$

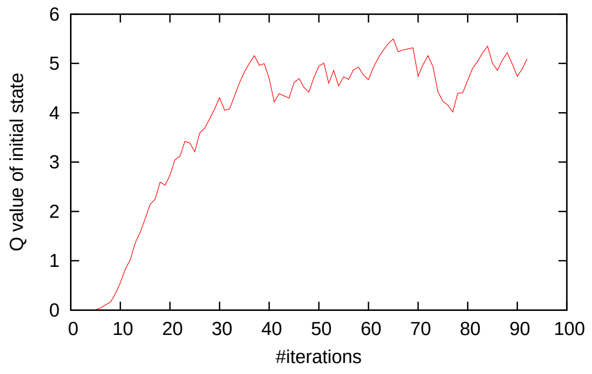


FIG. 4. Learning in the logistics domain. The results are averaged over 10 runs.

And lastly, a truck can always be driven to any city:

$$Poss(drive(t, c), s) \equiv true$$

The successor-state axioms for the fluents $boxIn$, $truckIn$, and $loaded$ are given below:

$$\begin{aligned} boxIn(b, c, do(a, s)) \equiv\ & \exists t_1 .\, truckIn(t_1, c, s) \land a = unload(b, t_1) \\ & \lor boxIn(b, c, s) \land \neg\exists t_2 .\, a = load(b, t_2) \end{aligned}$$

$$\begin{aligned} truckIn(t, c, do(a, s)) \equiv\ & a = drive(t, c) \\ & \lor truckIn(t, c) \land \neg\exists c.\, a = drive(t, c) \end{aligned}$$

$$\begin{aligned} loaded(b, t, do(a, s)) \equiv\ & a = load(b, t) \\ & \lor loaded(b, t, s) \land a \neq unload(b, t) \end{aligned}$$

The BDD induced by the program above as it was computed by our implementation contains 2568 decision nodes and 1904 terminal nodes. The values for the learning rate, the exploration probability, and the discount factor were as above. Again, we used randomly generated instances during learning. These contained between 3 and 10 cities, between 2 and 8 trucks, and between 3 and 12 boxes that could either be in one of the cities or on one of the trucks. The results are shown in Fig. 4. Again, after only a couple of iterations the agent makes repeatedly "good" decisions.

Although the programs presented above and the corresponding induced BDDs are of moderate size, only slight extensions or variations of the program or the formalization of the domain can lead to unmanageable large BDDs. For instance, for a variation of the program given above, where the destination is not given but has to be chosen by the agent, we were not able to compute the induced BDD (though it should be possible if the ordering of the nodes of the BDDs is optimized).

$$\begin{aligned} \mathbf{learn}(\mathbf{nondet}(&\mathbf{pick}(b, \mathbf{pick}(t, [load(b, t), \mathbf{pick}(c, drive(t, c)), unload(b, t)])), \\ &\mathbf{pick}(b, \mathbf{pick}(t, [\mathbf{pick}(c, drive(t, c)), unload(b, t)])), \\ &\mathbf{pick}(b, \mathbf{pick}(t, unload(b, t))))) \end{aligned}$$

The problem is the increased number of nested nondeterministic choices of arguments. Each leads to an exponential growth of the induced partition/BDD (not necessarily in the size of the partition/BDD induced by the remaining program, but still exponential).

10 Related Work

The work presented here, basically combines two different ideas that have been taken up by various approaches to handle the problem of using reinforcement learning for problems that have quite large state spaces. In particular, these ideas are to restrict the space of the policies that is considered when searching for the optimal policy and the abstraction of the state (and action-) space. The intuition behind the latter concept is to generalize a value function over groups of ground states that are similar to each other w.r.t. the value function.

The approach presented in [18] follows the idea of constraining policies. It describes the HAM language that allows to define machines by a set of states and transitions between the states. A state might either be an action state, a call state, a choice state, or a stop state. For a model M and a HAM H the "induced MDP $H \circ M$" is defined to have a set of states consisting of the cross-product of states in H and M. For every state in $H \circ M$ where the machine component is an action state the model and machine transitions are combined; for every state in $H \circ M$ where the machine component is a choice state actions that only change the machine component of the state are introduced. It is shown that from an optimal policy for $H \circ M$ an optimal policy for H can be derived. In a further step $H \circ M$ is reduced to the states where the machine is in a choice state. This produces an equivalent SMDP and the optimal policy for this SMDP will be the same as for $H \circ M$. Note, that these ideas are quite similar to what we did: the state in our decision process consist of tuples $\langle \phi, \delta \rangle$ where the remaining program δ is comparable to the machine state and the state of the environment is described by ϕ. Also, we reduced the set of states to choice states. Compared to Golog the HAM-language is not quite as expressive (although it was extended in [1] by means of parametrization, aborts, interrupts, and memory variables).

The ALisp programming language [2] allows to constrain the policies as well as it allows for state abstraction. Also, it includes the approach to learn the policy for a procedure independent of its calling-context as it was presented in [8]. The state abstraction, though, requires the user to specify the features that are relevant w.r.t. the value function.

DTGolog presented in [5] constitutes another approach that uses the Golog programming language to restrict the policies for the MDP underlying the program. The decision-theoretic approach that is implemented in the DTGolog interpreter, though, requires the complete model of the environment including the probability distributions over the outcomes of the stochastic actions. The solution computed by the interpreter is specific to the current situation, i.e., no state space abstraction is performed.

The approaches mentioned above use programs to limit search-space for the optimal policy of an MDP. There exists intentions to explore the relationship between the BDI architecture and (PO)MDPs [23, 21]. Roughly, the idea is to translate policies for MDPs into plans the BDI framework can handle and vice-versa. The intuition behind this is that the BDI model scales better than MDPs for certain problems. In these cases it is advantageous to determine a BDI plan, first, and then translate that into a policy for the MDP.

State-space abstraction, that is, aggregating ground states and treating them all alike during the search for an optimal (or a close to optimal) policy, is an appropriate way to reduce the computational complexity. An abstraction mechanism for a state-space can either be *safe* or *approximative.* The former means that any optimal policy computed for the abstracted state-space corresponds to an optimal policy in the original state-space. For instance, in [8] certain criteria are mentioned that allow for a safe state abstraction in the MAXQ-framework. Approximative abstraction mechanisms on the other hand result from a simplification of the original problem. These approaches are not guaranteed to find optimal solutions, some even cannot ensure that the solution found incorporating abstraction is applicable in the original problem. Numerous approximative abstraction mechanisms are described in the literature; two examples are [7] and [3]. A further characteristic of such an abstraction mechanism is the level of abstraction. Whereas the approaches mentioned above all work with propositional descriptions of the state, we present some works that employ first-order logic (or some restricted subset of it) for the description of the states.

The symbolic dynamic programming (SDP) approach presented in [4] also relies on the situation calculus to describe the dynamics of the model. The result of applying SDP is a first-order definition of the state-value function. Particularly, this state-value function is computed by a first-order variant of the value iteration algorithm.

In [22] it is shown how to overcome the representational complexity that arises from dealing with the expressiveness of first-order logic and that made a practical application of SDP method infeasible. A representation of the (first-order) value function by means of a first-order algebraic decision diagram (FOADD) is proposed. This inspired the use of BDDs in our approach. FOADDs allow to detect context-specific independence (CSI) and thereby simplify the representation by removing decision nodes that represent conditions that have no influence on the represented value function under their specific context.

A symbolic dynamic programming approach for the Fluent Calculus is presented in [12]. In [13] this approach is refined by introducing a normalization algorithm that discovers and prunes redundant states which might result from performing regression. Neglecting such redundant states avoids unnecessary computations in the further steps of the value iteration algorithm.

Other approaches to deal with the computational complexity of first-order logic is to avoid it by using a less expressive language to describe the abstract states. An example for this is presented in [14]. Instead of using full first-order logic the approach employs a relational logic that only allows for (implicit) existential quantification.

The work presented in [10] inspired the work presented in this paper. Whereas the approaches above exploit the logical definition of the actions to induce abstract definitions of states, in [10] the authors show how such abstract state description can be derived in the presence of a program that constrains the set possible policies. Specifically, the programs are programs in the language GTGolog which is an extension of Golog and incorporates game-theoretic optimization theory. Although, the basic idea of what we presented here is very similar, there are some notable differences:

- We used BDDs to represent the partitions induced by the program and introduced operations on BDDs that allow to computed the BDD representing the partition induced by a program.

- The nondeterministic choice of arguments is not restricted to a choice between a specified, finite list of ground objects. Instead we allow for an unrestricted version of the **pick**(v, η) construct and partition the choice for v by means of first-order formulas such that these capture the influence of the choice made for v on the future expected reward.
- Our approach is based on the "reduced" SMDP (cf. [18]). That is, each state in the state space corresponds to a choice state in the program. Specifically, the state is not changed with every (primitive) action (as it is done in [10]) but only at the choice points in the program which is reasonable since the execution of the program between those choice points is deterministic.
- We integrated Q-learning into the interpreter of our QGolog dialect. This has the advantage that it allows to reason about the expected rewards for executing a program in a certain situation. For example, in a high-risk situation the agent might deliberate about whether it is better to follow the program for which a policy has been learned or to follow a possibly non-optimal hand-coded program that has been especially written with safety in mind.

11 Conclusions

The work presented in this paper builds up on the idea presented in [10] to understand the execution of the program as a process over a state space where the states can be described as a combination of the world state and the machine state (similar to [18]). The machine states are identified with the remaining program and the (abstract) world states are described by means of first-order formulas. These can be derived given the program and the axiomatization of the primitive actions mentioned by the program. This allows to derive descriptions of the states which are limited to features that are of relevance for the further execution of a program w.r.t. the expected reward. QGolog exploits this kind of abstraction in two ways: first, it generalizes experiences made in a particular situation to all situations in which first-order formula describing the abstract state holds. Second, this abstraction extends to actions. When the program allows the agent to choose an argument, the Q-update not only happens for the particular choice the agent makes but for all possible choices the satisfy a specific first-order formula.

On the one hand the abstraction mechanisms allow to obtain quite good policies after fairly few iterations in cases where non-abstracting approaches would be hopelessly lost. On the other hand this requires to deal with the full complexity of first-order logic. This was already pointed out in [4] and to a certain degree this problem could be alleviated by using suitable data-structures. But for larger problems this is not sufficient. Here, we see two somewhat related approaches (that might also be combined with each other) which we intend to explore in future work. The first one is a hierarchical decomposition of the problem in a way similar to the MAXQ framework [8]. In the context of Golog programs, procedures quite "naturally" define a hierarchical structure. The goal then is to learn the optimal execution of a procedure's body independent of its calling context. The second way to reduce the inherent complexity are approximative methods. In [22] an approach for first-order MDPs that approximates the true value function by a linear combination of basis function has already been discussed but it is not directly applicable to the case where a program is given.

References

[1] David Andre and Stuart J. Russell. Programmable reinforcement learning agents. In *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000*, pages 1019–1025. MIT Press, 2000.

[2] David Andre and Stuart J. Russell. State abstraction for programmable reinforcement learning agents. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI) and Fourteenth Conference on Innovative Applications of Artificial Intelligence (IAAI)*, pages 119–125. AAAI Press, 2002.

[3] Jennifer Barry, Leslie Kaelbling, and Tomás Lozano-Pérez. Hierarchical solution of large markov decision processes. In *ICAPS Workshop on Planning and Scheduling in Uncertain Domains*, 2010.

[4] Craig Boutilier, Raymond Reiter, and Bob Price. Symbolic dynamic programming for first-order MDPs. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001*, pages 690–700. Morgan Kaufmann, 2001.

[5] Craig Boutilier, Raymond Reiter, Mikhail Soutchanski, and Sebastian Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI) and Twelfth Conference on Innovative Applications of Artificial Intelligence (IAAI)*, pages 355–362. AAAI Press, 2000.

[6] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[7] Thomas Dean, Rober Givan, and Sonia Leach. Model reduction techniques for computing approximately optimal solutions for markov decision processes. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pages 124–131, 1997.

[8] Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[9] Alexander Ferrein, Christian Fritz, and Gerhard Lakemeyer. On-line decision-theoretic golog for unpredictable domains. In *KI 2004: Advances in Artificial Intelligence, 27th Annual German Conference on AI*, volume 3238 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2004.

[10] Alberto Finzi and Thomas Lukasiewicz. Adaptive multi-agent programming in GTGolog. In *KI 2006: Advances in Artificial Intelligence, 29th Annual German Conference on AI*, volume 4314 of *Lecture Notes in Computer Science*, pages 389–403. Springer, 2007.

[11] Guiseppe De Giacomo, Yves Lespérance, and Hector Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.

[12] Axel Großmann, Steffen Hölldobler, and Olga Skvortsova. Symbolic dynamic programming within the fluent calculus. In *Proceedings of the IASTED International Conference on Artificial and Computational Intelligence*, pages 378–383. ACTA Press, 2002.

[13] Steffen Hölldobler and Olga Skvortsova. A logic-based approach to dynamic programming. In *Proceedings of the Workshop on "Learning and Planning in Markov Processes–Advances and Challenges" at the Nineteenth National Conference on Artificial Intelligence (AAAI)*, pages 31–36. AAAI Press, 2004.

[14] Kristian Kersting, Martijn Van Otterlo, and Luc De Raedt. Bellman goes relational. In *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004)*, volume 69 of *ACM International Conference Proceeding Series*, pages 465–472. ACM, 2004.

[15] Hector Levesque. A completeness result for reasoning with incomplete knowledge bases. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 14–28. Morgan Kaufmann, 1998.

[16] Hector Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard Scherl. Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1-3):59–83, 1997.

[17] Fangzhen Lin and Ray Reiter. How to progress a database. *Artificial Intelligence*, 92(1-2):131–167, 1997.

[18] Ronald Parr and Stuart J. Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 10 (NIPS 1997)*, pages 1043–1049. MIT Press, 1998.

[19] Ray Reiter. The frame problem in situation the calculus: a simple solution (sometimes) and a completeness result for goal regression. *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy*, pages 359–380, 1991.

[20] Raymond Reiter. *Knowledge in Action.* MIT Press, 2001.

[21] Gavin Rens, Alexander Ferrein, and Etienne van der Poel. Bdi agent architecture for a POMDP planner. In *Proccedings of the 9th International Symposium on Logical Formalization of Commonsense Reasoning: Commonsense 2009*, 2009.

[22] Scott Sanner and Craig Boutilier. Practical solution techniques for first-order MDPs. *Artificial Intelligence*, 173(5-6):748–788, 2009.

[23] Gerardo I. Simari and Simon Parsons. On the relationship between mdps and the bdi architecture. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1041–1048. ACM, 2006.

[24] Mikhail Soutchanski. An on-line decision-theoretic golog interpreter. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001*, pages 19–24. Morgan Kaufmann, 2001.

[25] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[26] Christopher J. C. H. Watkins. *Learning from Delayed Rewards.* PhD thesis, Cambridge University, 1989.

Received October 7, 2011