ECAI 2012
Luc De Raedt et al. (Eds.)

doi:10.3233/978-1-61499-098-7-139

Representing Value Functions with Recurrent Binary Decision Diagrams

Daniel Beck and **Gerhard Lakemeyer**[1]

Abstract. The agent programming language Golog features nondeterministic constructs such as nondeterministic branching. Given an optimization theory the nondeterminism can be resolved optimally. There are techniques that allow to derive an abstract first-order description of a value function which is valid across all possible domain instances. The size of the domain may be unknown or even infinite. A finite horizon is assumed, though. That is, although the value function makes no assumptions about the size of the domain, the plans generated on the basis of the value functions are restricted to a certain length. In this paper we present a solution for this dilemma for a specific class of programs. In particular, we present a solution that allows to compute a representation of the value function for non-nested loops without requiring any restrictions on the number of loop iterations. A pleasing side effect is that our new representation of the value function is usually smaller than the representations for fixed horizons.

1 INTRODUCTION

The general problem we are concerned with, is that of finding an optimal execution strategy (wrt a given optimality criterion) for a program containing nondeterministic constructs. Assuming the agent obtains a reward after executing an action, the problem can be solved with the help of a value function which returns for every program the sum of accumulated, discounted rewards. Then, the nondeterminism in the program can be resolved by always opting for the choice which maximizes the value function. In the context of the situation calculus and Golog programs, it is possible to construct a first-order representation of the value function [1] which makes it possible to derive an execution strategy which is optimal in all possible domain instances. In particular, no assumptions about the size of the domain are made, it may be unknown or even infinite. On the downside, that value function is specific to a horizon h which means that it only considers the first h steps of the programs. So, on the one hand there is the potential to deal with arbitrarily sized domains but on the other hand the execution strategies that can be derived from the value functions are limited in length by the horizon. In cases where the program execution directly depends on the size of the domain this is particularly limiting since for every horizon, no matter how large it is, a domain instance can be found that requires a policy of greater length. Programs of this kind are for instance loops that iterate over all domain objects.

The motivating observation for this work is that in some cases the first-order representation of the value function for programs containing loops evolves in a predictable way when the horizon is increased. Consequently, we attempt to identify these patterns that allow to predict how the formulas evolve and encode them explicitly in the representation of the value function. The result is that we can find a finite representation of the value function for loops that does not require a horizon. First-order binary decision diagrams, a variant of regular BDDs, are a beneficial data structure for representing first-order value functions [9]. We propose an extension to FOBDDs which allows for recurrent edges and show how these recurrent FOBDDs can be used for the representation of value functions for certain kinds of loops in a way such that a value can be determined if the loop terminates after a finite but arbitrary number of iterations.

In Sections 2 and 3 we give a short introduction to the agent programming language Golog and its underlying framework, the situation calculus, and show how for a given Golog program a first-order description of the value function can be derived. The extension of FOBDDs we propose is introduced in Section 4. There we also describe under what conditions such recurrent representations can be found. Afterwards we discuss the limitations of our approach and conclude in Section 5.

2 SITUATION CALCULUS AND GOLOG

The agent programs we consider in this paper are given in the agent programming language Golog [6] which is built on top of the situation calculus [8]. It features the usual constructs such as conditionals and loops but also nondeterministic constructs such as nondeterministic branching $(\delta_1 \mid \delta_2)$. Here, the agent may either continue with δ_1 or with δ_2. The situation calculus allows to reason about actions and change and its actions are the primitives of Golog programs. For every action $A(\vec{x})$ a precondition axiom $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$ is given. A situation is a history of actions, $do(a, s)$ being the situation reached after executing action a in situation s. S_0 denotes the initial situation. Fluents represent properties of the domain which may change from situation to situation. The dynamics of the fluents are specified by so-called *successor state axioms* which are of the form $F(\vec{x}, do(a, s)) \equiv \gamma_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s)$. By the regression of a formula ϕ, denoted as $\mathcal{R}[\phi]$, we mean ϕ with all occurrences of every fluent $F(\vec{t}, do(A, s))$ replaced by the right-hand side of the respective successor state axiom, with $\vec{t}$ substituted for $\vec{x}$ and A for a. A *basic action theory* $\mathcal{D}$ (BAT) contains among others the precondition axioms $\mathcal{D}_{ap}$, the successor state axiom $\mathcal{D}_{SSA}$, and a description of the initial situation $\mathcal{D}_{S_0}$.

Before a Golog program can be executed the nondeterminism needs to be resolved. Given an optimization theory the optimal execution among all possible executions can be determined (cf. [4, 1]). The optimization theory contains the definition of a reward function $rew(s)$ which assigns values to situations and a horizon. We assume

[1] RWTH Aachen University, Germany, email: {dbeck, gerhard}@cs.rwth-aachen.de

$rew(s)$ to be of the following form where the r_i are numeric constants:

$$rew(s) = r \equiv \phi_1^{rew}(s) \wedge r = r_1 \vee \cdots \vee \phi_l^{rew}(s) \wedge r = r_l \quad (1)$$

In such a setting the semantics of a program can be viewed as the execution of the first h steps of that program (where h is the horizon) which maximizes the accumulated reward. [2]

3 VALUE FUNCTIONS AND THEIR REPRESENTATION

We adopt the approach of [1] which explicitly represents the value function $V_h^\delta(s)$. For a program δ and a horizon h, $V_h^\delta(s)$ returns the maximal accumulated reward for executing the first h steps of δ in s. Since $V_h^\delta(s)$ is piece-wise constant, it is especially well suited to be given in the case notation [3]:

$$case[\phi_i : v_i \,|\, 1 \leq i \leq n] \stackrel{def.}{=} \bigvee_{i=1}^{n} \phi_i \wedge \mu = v_i \quad (2)$$

where μ is a special variable that is reserved for the use within case statements and must not be used anywhere else. The following macro allows to embed a case statement in a formula without explicitly referring to μ:

$$v = case[\phi_i : v_i \,|\, 1 \leq i \leq n] \stackrel{def.}{=} case[\phi_i : v_i \,|\, 1 \leq i \leq n]_v^\mu \quad (3)$$

Case statements can be combined in different ways. Two are defined by the $\oplus$-operator and the $\cup$-operator:

$$case[\phi_i : v_i \,|\, 1 \leq i \leq n] \oplus case[\psi_j : w_j \,|\, 1 \leq j \leq m] \stackrel{def.}{=} case[\phi_i \wedge \psi_j : v_i + w_j \,|\, 1 \leq i \leq n, 1 \leq j \leq m] \quad (4)$$

$$case[\phi_1 : v_i \,|\, 1 \leq i \leq n] \cup case[\psi_j : w_j \,|\, 1 \leq j \leq m] \stackrel{def.}{=} case[\phi_1 : v_1, \ldots, \phi_n : v_n, \psi_1 : w_1, \ldots, \psi_m : w_m] \quad (5)$$

3.1 Value Functions for Programs

The construction of the case statement representing $V_h^\delta(s)$ depends on δ. For instance, if δ begins with a primitive action a (i.e., $\delta = a; \delta'$ for some δ'), then $V_h^\delta(s)$ is constructed as follows:

$$V_h^{a;\delta}(s) \stackrel{def.}{=} \{\neg Poss(a,s) \wedge rCase(s)\} \cup \{Poss(a,s) \wedge rCase(s) \oplus \gamma \cdot \mathcal{R}\left[V_{h-1}^\delta(do(a,s))\right]\} \quad (6)$$

$rCase(s)$ denotes the case notation of the reward function $rew(s)$ (cf. Eq. 1). The above case statement differentiates situations in which a is executable from those in which it is not executable. In the former case the value is made up of the immediate reward and the future reward discounted by γ. In the latter case only the immediate reward is taken into account since the program cannot be executed any further. A complete definition of $V_h^\delta(s)$ can be found in [1].

[2] Actually, the nondeterminism is resolved with the goal of maximizing the reward as well as the probability of successfully executing the program. For reasons of simplifying the presentation we ignore the latter.

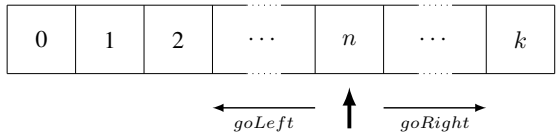

Figure 1. Hallway domain.

3.2 First-Order BDDs

A variant of binary decision diagrams (BDDs), called first-order BDD (FOBDD), has been proposed in [9] as a data structure to represent case statements as $V_h^\delta(s)$, for instance. Not only does this have advantages from a representational point of view in comparison to dealing with plain formulas, also many of the operations can be directly carried out on FOBDDs without changing the representation back to plain formulas.

As for BDDs, every decision node in a FOBDDs has two children, the high child and the low child. FOBDDs may have arbitrarily many terminal nodes labeled with real-valued numbers. The decision nodes of a FOBDD are associated with first-order formulas. Examples of FOBDDs can be seen in Fig. 2. From every FOBDD F a case statement can be constructed. This case statement is denoted by $case[F]$ and is constructed as follows: for every terminal node v, add a formula ϕ_v^F with associated value v to the case statement. ϕ_v^F is obtained by disjunctively combining the formulas corresponding to all paths from the root node to v. Let $x_1, \ldots, x_t, v$ be such a path. The corresponding formula is built by conjunctively combining the formulas associated with the x_i. If x_{i+1} is the low child of x_i then the negation of the formula associated with x_i is taken. A FOBDD F is said to represent a case statement $case[\phi_i : v_i \,|\, 1 \leq i \leq n]$ if there are n terminal nodes labeled $v_1, \ldots, v_n$ and $\models \phi_{v_i}^F \equiv \phi_i$ for all $1 \leq i \leq n$.

The FOBDDs representing $V_h^\delta(s)$ are valuated wrt a BAT $\mathcal{D}$ and a ground situation σ. Beginning at the root node of the FOBDD, it is tested whether $\mathcal{D} \models \phi(\sigma)$ where ϕ is the formula associated with the root node. If the entailment holds the valuation is continued at the high child otherwise at the low child. If a terminal node is reached its value is returned.

3.3 Notation

In the following we make use of some abbreviations to keep the notation concise. In particular these are:

$$\psi \wedge case[\phi_i : v_i \,|\, i \leq n] \stackrel{def.}{=} case[\psi \wedge \phi_i : v_i \,|\, i \leq n] \quad (7)$$

$$v + case[\phi_i : v_i \,|\, i \leq n] \stackrel{def.}{=} case[\phi_i : v + v_i \,|\, i \leq n] \quad (8)$$

$$v \cdot case[\phi_i : v_i \,|\, i \leq n] \stackrel{def.}{=} case[\phi_i : v \cdot v_i \,|\, i \leq n] \quad (9)$$

For a terminal node t of F $case[F \backslash t]$ is the case notation corresponding to F but without the case with the value t.

For FOBDDs F and G and a terminal node t of F, F_G^t is the FOBDD that results from replacing the terminal node t with G. We write $v + F$ to denote that the value v is added to the value returned when valuating F, and similar for $v \cdot F$. For a substitution θ, $F\theta$ means that θ is applied to the formulas associated with the decision nodes of F.

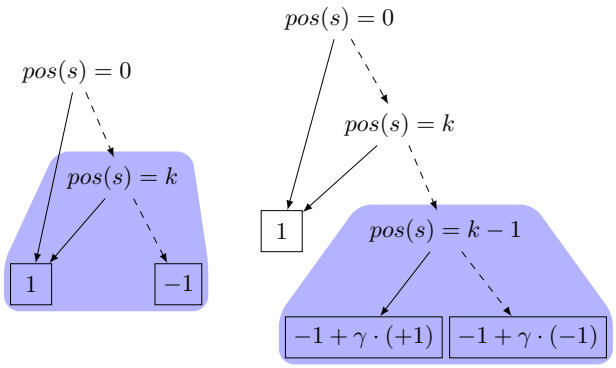

Figure 2. FOBDDs representing the value functions for at most one iteration through the loop (left) and at most two iterations through the loop (right), respectively.

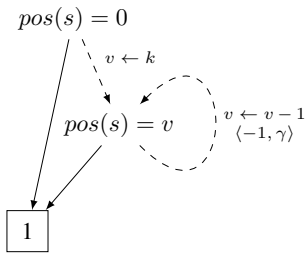

Figure 3. Recurrent FOBDD representing the value function for **while** $pos \neq 0 \wedge pos \neq k$ **do** $goRight$ **end** for any (finite) value of k.

4 RECURRENT DECISION DIAGRAMS

Let us consider a hallway-like domain as it is depicted in Fig. 1. The space is divided into tiles which are numbered from 0 to k where k is a constant. The agent can move to the adjacent tiles by means of the actions $goLeft$ and $goRight$. Assume the agent is executing the following program which instructs it to either repeatedly go to the left or to the right until it reaches either end.

$$\begin{array}{l}(\ \textbf{while}\ pos \neq 0 \wedge pos \neq k\ \textbf{do}\ goLeft\ \textbf{end}\ \mid \\ \quad \textbf{while}\ pos \neq 0 \wedge pos \neq k\ \textbf{do}\ goRight\ \textbf{end}\ )\end{array} \tag{10}$$

In the context of an optimization theory the nondeterminism in the program can be resolved optimally. In the example the agent needs to determine whether repeatedly going to the right or to the left maximizes the value function and decide accordingly for the one or the other branch. The agent receives a reward of $+1$ in situations where it is either at position 0 or at position k; otherwise the reward is -1. The discount factor is γ. For a given domain instance, that is, for a known value for k, a horizon can be chosen such that it is guaranteed that the agent can reach one of the ends within the horizon. But generally, if no particular value for k is assumed, it is not possible to compute a solution with a given horizon. In a nutshell, the horizon limits the domain instances for which a solution can be computed to instances of a certain size. On the other hand it is quite simple to specify a formula such that for every value of k the correct value for each of the two branches is computed. For instance, if the agent is at position n the value for going right is

$$\sum_{i=0}^{k-n-1} \gamma^i \cdot (-1) + \gamma^{k-n} \cdot (+1). \tag{11}$$

The problem though is that the pattern underlying this computation is not made explicit in the representation of the value function $V_h^\delta(s)$.

Let F_1 be the left FOBDD depicted in Fig. 2 and F_2 be the right one. Let G_1 be the highlighted sub-FOBDD of F_1 and G_2 be the highlighted sub-FOBDD of F_2. F_1 and F_2 represent the value functions for the nondeterministic branch of the program above that instructs the agent to repeatedly go to the right for a horizon of 1 and 2, respectively (which corresponds to at most one or up to two iterations through the loop). F_2 is an extension of F_1 in the sense that one of the terminal nodes of F_1, namely -1, is replaced with G_2 in F_2. Additionally, G_2 exhibits certain similarities to G_1: the formulas associated with the decision nodes only differ in the value of the functional fluent $pos(s)$; the values of the terminal nodes of G_2 are the result of adding -1 to γ times the value of the corresponding terminal node in G_1. That is, the values of the terminal nodes of G_2 are obtained by applying the affine transformation defined by $\langle -1, \gamma \rangle$ on the values of the terminal nodes of G_1. So, we see a certain pattern here and if we construct FOBDDs F_3, F_4, $F_5, \ldots$ according to this pattern it can be confirmed that these are in fact representations of the value function for $h = 3, 4, 5, \ldots$. In particular, F_3 would be constructed by replacing the low child of the node labeled with $pos(s) = k - 1$ in F_2 with a FOBDD G_3 which is similar to G_2 only that it is now tested whether $pos(s) = k - 2$ and the terminal nodes are obtained by one more application of the affine transformation $\langle -1, \gamma \rangle$ on the values of the terminal nodes of G_2. The idea now is to explicitly capture this pattern in a data structure representing the value function. Therefore, we propose an extension of FOBDDs called *recurrent FOBDDs*. Fig. 3 shows the recurrent FOBDD resulting from implementing the observations about the evolution of F_2 wrt F_1 made above.

Contrary to BDDs (and also FOBDDs) which are acyclic, recurrent FOBDDs allow for cycles. The formulas associated with the nodes within loops have to mention special variables which are initialized upon the entry in the loop and updated every time a loop is completed. Therefore, initialization as well as update rules are associated with the incoming edges to those nodes which are entry points to a loop: the initialization rule with the edges along the paths from the root node and the update rules along all other of these edges. Additionally, the latter edges are also annotated with an affine transformation. Associating affine transformations with edges has already been proposed in [10].

The valuation of recurrent FOBDDs is explained on the basis of the recurrent FOBDD in Fig. 3. Let $\mathcal{D}$ be a BAT (containing a value for k) and σ be a ground situation. If $\mathcal{D} \models pos(\sigma) = 0$ then the value $+1$ is returned. Otherwise the valuation moves on to the low child and thereby initializes v to the value k. Next, it is determined whether $\mathcal{D} \models pos(\sigma) = k$, that is, every occurrence of v is replaced by its current value k. If so, $+1$ is returned. Otherwise, v is decremented by one, now having the value $k - 1$. If now $D \models pos(\sigma) = k - 1$ holds a terminal node with the value $+1$ is reached. But since on the path that the valuation followed there is an edge annotated with the affine transformation $\langle -1, \gamma \rangle$ not $+1$ but $-1 + \gamma \cdot (+1)$ is returned. With such a valuation strategy the recurrent FOBDD in Fig. 3 concisely represents the value function for repeatedly going right until one of the ends is reached for any (finite) value for any k. It is even smaller than the FOBDD F_2. A similar recurrent FOBDD can be constructed representing the value function for repeatedly going to the left. Then, for every possible (finite) value of k the values for both programs in the current situation can be determined and the nondeterminism in

the program above can be resolved.

In the following we argue why the method we used to construct the recurrent FOBDD in the example above is correct in general. First, we need to define what it means that one FOBDD is a *variant* of another FOBDD. In the example above, we assumed that G_2 is a variant of G_1.

Definition 1 *A FOBDD F' is considered to be a* variant *of a FOBDD F if*

- *F and F' are structurally identical, that is, F' can be obtained from F by simply renaming the nodes and vice versa,*
- *the formulas associated with the nodes are either equivalent or are similar (see below), and*
- *there exists an affine transformation $\langle a, b\rangle$ such that $t' = a + b \cdot t$ for all terminal node nodes t' of F' and the corresponding terminal nodes t in F.*

Before two formulas can be tested for similarity we assume that they are transformed into a form such that every functional fluent f mentioned by the formula occurs in an expression of the form $f(\vec{x}, s) + c_f = \ldots$ (or $f(\vec{x}, s) + c_f > \ldots$) where the c_f are numerical constants. Then, two formulas ϕ_1 and ϕ_2 are considered similar if they can be transformed into formulas of the form described above and only differ in the c_f's. For similar formulas ϕ_1 and ϕ_2 with numerical constants c_f^1 and c_f^2 a formula $\hat{\phi}$ can be constructed by replacing every occurrence of the c_f's with new variables v_f. Then, for substitutions $\theta_1 = \{v_f/c_f^1\}$ and $\theta_2 = \{v_f/c_f^2\}$ for the variables v_f it holds that $\phi_1 = \hat{\phi}\,\theta_1$ and $\phi_2 = \hat{\phi}\,\theta_2$. In the example above, $pos(s) = k$ and $pos(s) = k - 1$ would be transformed into the equivalent formulas $pos(s) + 0 = k$ and $pos(s) + 1 = k$. Then $pos(s) + v = k$ with the substitutions $\{v/0\}$ and $\{v/-1\}$ is equal to the former and latter formula, respectively.

What we intend to show now is that detecting such a pattern is actually sufficient to justify the existence of a recurrent FOBDD. That is, it is correct to assume that the FOBDD continues to evolve according to the identified pattern when the horizon is increased further. For this, we proceed as follows. Theorem 1 shows that this is true for a very limited case. Namely for the case where the body of the loop only consists of a single primitive action and the FOBDD representing the value function for $h+1$ extends the FOBDD representing the value function for h only at a single terminal node as it was the case in the example above. Afterwards we show how these limitations can be extended such that the claim also holds for loops whose bodies are finite, deterministic programs and when the FOBDDs are extended at multiple terminal nodes.

Theorem 1 *Let $\delta =$ **while** ϕ **do** A **end** where A is a primitive action and F_h and F_{h+1} ($h \geq 1$) be FOBDDs representing $V_h^\delta(s)$ and $V_{h+1}^\delta(s)$, respectively. If F_h can be composed from FOBDDs F and G, i.e. $F_h = F_G^t$ for a terminal node t of F, and there exists a terminal node t_* of G such that $F_{h+1} = F_h\,{}_{G'}^{t_*}$ and G' is a variant of G then there exists a recurrent FOBDD representing the value function for any finite number of iterations through the loop.*

Proof sketch. As a general remark, if F_h and F_{h+1} are of a form as described above, this means that the $(h+1)$st iteration through the loop is only possible in situations where $\phi_{t_*}^{F_h}$ holds. Otherwise either the preconditions for A are not given or the loop condition does not hold either right now or after executing A up to h times.

$V_{h+1}^\delta(s)$ is computed from $V_h^\delta(s)$ like this:

$$\begin{aligned} V_{h+1}^\delta(s) \stackrel{def.}{=} \Big\{ & \underbrace{\Big\{ \phi[s] \wedge \quad Poss(A,s) \wedge rCase(s) \oplus \gamma \cdot \mathcal{R}\left[V_h^\delta(do(A,s))\right] \Big\}}_{S_1} \cup \\ & \underbrace{\Big\{ \phi[s] \wedge \neg Poss(A,s) \wedge rCase(s) \Big\}}_{S_2} \cup \\ & \underbrace{\Big\{ \neg\phi[s] \wedge rCase(s) \Big\}}_{S_3} \end{aligned} \tag{12}$$

S_2 and S_3 are already present in $V_h^\delta(s)$ (since $h > 0$) and therefore these cases are represented by F_h and also by F_{h+1}. Thus, the "new" cases $\phi_{t_*}^{F_h} \wedge case[G']$ have to stem from S_1. G' is a variant of G and consequently there exists an affine transformation transforming values of the terminal nodes of G to values of the corresponding terminal nodes of G'. Looking at how the values are manipulated in S_1, this affine transformation has to be $\langle r_*, \gamma\rangle$, $* \in \{1, \ldots, l\}$ (cf. Eq. 1). Then, S_1 can be decomposed as follows:

$$\begin{aligned} S_1 = & \underbrace{\Big\{ \phi[s] \wedge Poss(A,s) \wedge \phi_*^{rew}(s) \wedge r_* + \gamma \cdot \mathcal{R}\left[\phi_t^F \wedge case[G]\right] \Big\}}_{=\phi_{t_*}^{F_h} \wedge case[G']} \cup \\ & \Big\{ \phi[s] \wedge Poss(A,s) \wedge case[\phi_i^{rew} : r_i \mid i \neq *] \oplus \gamma \cdot \mathcal{R}\left[\phi_t^F \wedge case[G]\right] \Big\} \cup \\ & \Big\{ \phi[s] \wedge Poss(A,s) \wedge rCase(s) \oplus \gamma \cdot \mathcal{R}\left[case[F/t]\right] \Big\} \end{aligned} \tag{13}$$

The cases in the first set are precisely the "new" cases $\phi_{t_*}^{F_h} \wedge case[G']$ in $V_{h+1}^\delta(s)$. The cases in the second set are all unsatisfiable: these cases do not lead to new paths in F_{h+1} (these are the cases in the first set) and all the other paths in F_{h+1} which are also in F_h imply that either the preconditions of A do not hold or the loop condition does not hold right now or after executing A up to h times (cf. initial observation). The cases in the third set are already in $V_h^\delta(s)$ (since $h \geq 1$).

Since G' is a variant of G there has to be a FOBDD $\hat{G}$ such that with appropriate substitutions $\theta = \{v_f/c_f\}$ and $\theta' = \{v_f/c_f'\}$ for all variables v_f it holds that $\hat{G}\,\theta = G$ and $r_* + \gamma \cdot \hat{G}\,\theta' = G'$. The formulas associated with the nodes of G' result from regressing the formulas associated with the nodes of G through A (cf. Eq. 13). Due to the construction of $V_h^\delta(s)$ the formula $\phi_{t_*}^{F_h}$ uniquely determines how the values of the fluents mentioned by the formulas associated with the nodes of G change after executing A. Let this change be described by a function τ_f. Then $c_f' = \tau_f(c_f)$ and $G' = r_* + \gamma \cdot \hat{G}\,\{v_f/\tau_f(c_f)\}$.

Similar to Eq. 12, $V_{h+2}^\delta(s)$ can be split up into three case statements S_1', S_2', and S_3' with $S_2' = S_2$ and $S_3' = S_3$. Consequently, these are already contained in $V_{h+1}^\delta(s)$. The case statement S_1' is like S_1 only that $V_h^\delta(do(A,s))$ is replaced by $V_{h+1}^\delta(do(A,s))$. It can be decomposed in a similar fashion as S_1 (cf. Eq. 13) singling out the case statement combining the reward case $\phi_*^{rew}(s) : r_*$ with the regression of the case statement $\phi_t^F \wedge \phi_{t_*}^G \wedge case[G']$. Knowing the structure of $\phi_{t_*'}^{F_{h+1}} = \phi_{t_*}^{F_h} \wedge \phi_{t_*'}^{G'}$ (cf. Eq. 13) helps to

make certain assumptions about the formulas in the case statement mentioned in the previous sentence. Using these and the fact that $G' = r_* + \gamma \cdot \hat{G}\{v_f/\tau_f(c_f)\}$ it can be concluded that this case statement has to be of the form

$$\{\phi_{t'_*}^{F_{h+1}} \wedge \underbrace{\gamma \cdot r_* + \gamma^2 \cdot case[\hat{G}\{v_f/\tau_f(\tau_f(c_f))\}]}_{=:case[G'']}\}. \quad (14)$$

That is, the regression leads to another application of τ_f and another affine transformation $\langle r_*, \gamma\rangle$ on the values of $case[\hat{G}]$. Since the remaining case in S'_1 can be shown to be either unsatisfiable or already present in $V^{\delta}_{h+1}(s)$ for similar arguments as above, we have thereby shown that $F^{h+1}{}_{G''}^{t'_*}$ is actually a representation of $V^{\delta}_{h+2}(s)$. Since h can be arbitrarily chosen, the value functions for any horizon greater than h can be represented by a FOBDD that is constructed according to the identified pattern detected by comparing F_h and F_{h+1}.

A recurrent FOBDD representing the value function for any finite number of iterations through the loop can then be constructed as follows:

1. Label every incoming edge to the root node of $\hat{G}$ in $F^t_{\hat{G}}$ with $v_f \leftarrow c_f$.
2. Add an edge from t_* in $\hat{G}$ to its root node and label it with the update rules $v_f \leftarrow \tau_f(v_f)$ and the affine transformation $\langle r_*, \gamma\rangle$.

□

The extension to cases where F_{h+1} representing $V^{\delta}_{h+1}(s)$ extends F_h at several terminal nodes (i.e., $F_{h+1} = F_h{}^{t_1}_{G_1} \cdots {}^{t_n}_{G_n}$) is straightforward as is the construction of the recurrent FOBDD. The idea behind extending the scope of the theorem to loops over finite, deterministic programs is to replace every finite, deterministic program occuring within a loop with a newly defined primitive action that behaves identically to the program it replaces. Then, Theorem 1 applies again.

4.1 Emulating Deterministic Programs

Finite, deterministic programs are made up of primitive actions, test actions, conditionals, and sequences. An action α *emulates* a finite, deterministic program δ if α can only be executed in situations in which δ can be completely executed; if executing α affects the fluents in the same way as executing δ; and if the reward after executing α is the same as the discounted, accumulated reward accrued while executing δ.

We refrain from providing a complete account of actions emulating arbitratry finite, deterministic programs. Instead we introduce the concepts by means of defining a new action emulating a sequence. Assume the agent needs to assess the program

$$\textbf{while } pos \neq 0 \wedge pos \neq k \textbf{ do } goRight; goRight \textbf{ end}. \quad (15)$$

The BAT needs to be extended to incorporate a new action α which emulates $goRight; goRight$. In particular, a precondition axiom for α is added expressing that α can only be executed if the sequence can be legally executed:

$$Poss(\alpha, s) \equiv Poss(goRight, s) \wedge \mathcal{R}[Poss(goRight, do(goRight, s))] \quad (16)$$

Further, the successor state axiom for the fluent pos, $pos(do(a, s)) = y \equiv \phi_{pos}(y, a, s)$, is replaced by:

$$pos(do(a, s)) = y \equiv a = \alpha \wedge pos(s) = y - 2 \vee a \neq \alpha \wedge \phi_{pos}(y, a, s) \quad (17)$$

The definition of $V^{\delta}_h(s)$ for the case where δ begins with a primitive action needs be updated slightly. It becomes:

$$\begin{aligned} V^{a;\delta}_h(s) \stackrel{def}{=} \Big\{ & Poss(a, s) \wedge rCase(s) \oplus \\ & \gamma^{\kappa(s)} \cdot \mathcal{R}\left[V^{\delta}_{h-\kappa(s)}(do(a, s))\right]\Big\} \cup \\ \Big\{ & \neg Poss(a, s) \wedge rCase(s) \oplus \\ & \gamma^{\kappa(s)} \cdot \mathcal{R}\left[rCase(do(a, s))\right]\Big\}. \end{aligned} \quad (18)$$

The discount of future rewards is not constant but depends on $\kappa(s)$. The rewards obtained after executing $goRight; goRight$ are discounted by γ^2 and so have to be the rewards obtained after executing α. The function $\kappa(do(a, s))$ returns the correct exponent for the action a in situation s. For $goRight$ that is always 1 and for α it is always 2. Corresponding axioms are added to the theory. In situations where the first $goRight$ can be executed but not the second one the agent still receives a reward after executing the first action. Since α is intended to behave identically to the sequence, rewards obtained by a partial execution of the seqeuence need to be considered. The changes made in $V^{\delta}_h(s)$ to that respect (cf. last two lines of Eq. 18) require to further change the definition of the reward function such that it returns 0 for a situation $do(goRight, s)$ if $goRight$ is not executable in s in order to stay comparable to the old definition of $V^{\delta}_h(s)$ (cf. Eq. 6). The reward function then might be defined by axioms of the form

$$rew(S_0) = r \equiv \Phi^{rew}(r, S_0) \quad (19)$$

$$rew(do(goRight, s)) = r \equiv \Phi^{rew}_{goRight}(r, s). \quad (20)$$

For the new action α the following axiom is added:

$$\begin{aligned} & rew(do(\alpha, s)) = r \equiv \\ & \quad \mathcal{R}\big[Poss(\alpha, s) \wedge r = rew(s) + \gamma \cdot rew(do(goRight, s)) \vee \\ & \qquad Poss(goRight, s) \wedge \neg Poss(goRight, do(goRight, s)) \wedge \\ & \qquad\quad r = rew(s) \vee \neg Poss(goRight, s) \wedge r = 0\big] \end{aligned} \quad (21)$$

Note that even with these changes the reward function as well as $V^{\delta}_h(s)$ can still be represented by case statements since $\kappa(s)$ can be given in the case notation.

Lemma 1 *Let $\mathcal{D}^{\alpha}$ be the BAT extended to include the new action α as outlined above. Similar for O^{α}. Then*

$$\mathcal{D}^{\alpha} \cup O^{\alpha} \models \forall s.\, V^{goRight;goRight}(s) = V^{\alpha}(s).$$

(We omitted the horizon since we are only interested in complete executions, i.e., for a horizon greater than 2.)

The recurrent FOBDD constructed according to Theorem 1 for the program **while** $pos \neq 0 \wedge pos \neq k$ **do** α **end** is shown in Fig. 4. As a consequence of Lemma 1 this recurrent FOBDD is also a representation of the value function for arbitrarily many iterations through the loop of the program looping over the sequence of going right twice.

4.2 Limitations

Apart from the obvious limitation that our method can only handle finite, deterministic programs within the loop, there are more subtle limitations. For instance, the criteria that have to hold for F' being a variant of F imply certain limitations. For one thing, F' can only be a

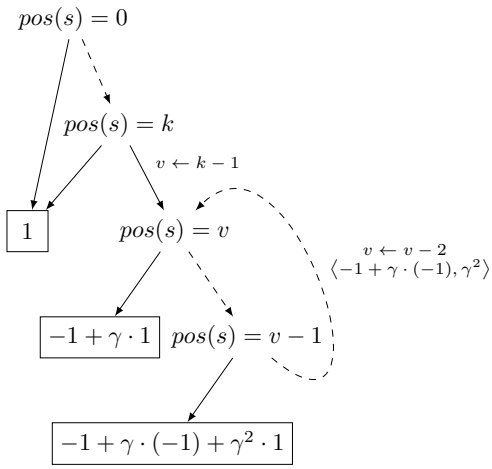

Figure 4. Recurrent FOBDD representing the value function for a loop over going right twice.

variant of F if the differences between the formulas associated with the decision nodes can be limited to numerical, functional fluents having different values. Consequently, there have to be numerical, functional fluents that change with every further iteration through the loop. For another, how these values change underlies certain restrictions by itself. For example, if the value of a fluent f after executing an action a would be set to the maximal value of $g(x)$ for any x, then this would not lead to a recurrent FOBDD.

5 CONCLUSIONS

Other areas of research which show interest in loops are for instance program verification [2], algorithmic design [7], and a special discipline in the area of planning which is concerned with finding plans that contain loops [5]. In these areas it is usually of interest what properties change respectively do not change when the loop is executed and whether or even after how many iterations it terminates and what conditions do hold then. In that respect these interests are different from ours: we are not interested in the invariants per se but in the invariance with which the representation of the value functions changes.

In [11] data structures similar to FOBDDs are used to represent the value function for Relational Markov Decision Processes. Their semantics though is defined wrt single interpretations (it is not based on entailments as in our case) and the formulas associated with the nodes have to be quantifier-free. It might be worthwhile to investigate whether similar recurrent extensions can be integrated into their approach.

We introduced an extension to FOBDDs that allows for recurrent edges and showed that under certain conditions these recurrent FOBDDs can represent the value function for a loop program without the limitations of a horizon: the value function provides a value in all cases where after an arbitrary finite number of iterations the loop terminates. For now, only finite, deterministic programs are allowed within the loops. But we are currently working on extending this approach to also allow for nondeterminism within the loop, thus increasing the range of programs amenable to our approach. Finally, another open question is how to handle nested loops.

ACKNOWLEDGEMENTS

We would like to thank the reviewers for their helpful comments which helped to improve the quality of this paper.

REFERENCES

[1] D. Beck and G. Lakemeyer, 'Decision-theoretic planning for golog programs with action abstraction', in *Proceedings of the Ninth International Workshop on Non-Monotonic Reasoning, Action and Change (NRAC-11)*, pp. 39–46, (2011).
[2] S. Bensalem, Y. Lakhnech, and H. Saïdi, 'Powerful techniques for the automatic generation of invariants', in *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pp. 323–335, (1996).
[3] C. Boutilier, R. Reiter, and B. Price, 'Symbolic dynamic programming for first-order MDPs', in *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, pp. 690–700, (2001).
[4] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun, 'Decision-theoretic, high-level agent programming in the situation calculus', in *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pp. 355–362, (2000).
[5] H. Levesque, 'Planning with loops', in *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, volume 19, pp. 509–515, (2005).
[6] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl, 'GOLOG: A logic programming language for dynamic domains', *The Journal of Logic Programming*, **31**(1-3), 59–83, (1997).
[7] Y. Liu, S. Stoller, and T. Teitelbaum, 'Strengthening invariants for efficient computation', *Science of Computer Programming*, **41**(2), 139 – 172, (2001).
[8] R. Reiter, *Knowledge in action: logical foundations for specifying and implementing dynamical systems*, 2001.
[9] S. Sanner and C. Boutilier, 'Practical solution techniques for first-order MDPs', *Artificial Intelligence*, **173**(5-6), 748–788, (2009).
[10] S. Sanner and D. McAllester, 'Affine algebraic decision diagrams (aadds) and their application to structured probabilistic inference', in *International Joint Conference on Artificial Intelligence*, volume 19, p. 1384, (2005).
[11] C. Wang, S. Joshi, and R. Khardon, 'First order decision diagrams for relational MDPs', *Journal of Artificial Intelligence Research*, **31**(1), 431–472, (2008).