Symbolic Dynamic Programming for First-Order MDPs

Craig Boutilier
Dept. of Computer Science
University of Toronto
Toronto, ON, M5S 3H5
cebly@cs.toronto.edu

Ray Reiter
Dept. of Computer Science
University of Toronto
Toronto, ON. M5S 3H5
reiter@cs.toronto.edu

Bob Price
Department of Computer Science
University of British Columbia
Vancouver, BC, V6T 1Z4
bprice@cs.toronto.edu

Abstract

We present a dynamic programming approach for the solution of first-order Markov decisions processes. This technique uses an MDP whose dynamics is represented in a variant of the situation calculus allowing for stochastic actions. It produces a *logical description* of the optimal value function and policy by constructing a set of first-order formulae that *minimally* partition state space according to distinctions made by the value function and policy. This is achieved through the use of an operation known as *decision-theoretic regression*. In effect, our algorithm performs value iteration without explicit enumeration of either the state or action spaces of the MDP. This allows problems involving relational fluents and quantification to be solved without requiring explicit state space enumeration or conversion to propositional form.

1 Introduction

Markov decision processes (MDPs) have become the *de facto* standard model for decision-theoretic planning problems. However, classic dynamic programming algorithms for MDPs [Puterman, 1994] require explicit state and action enumeration. For example, the classical representation of a value function is as a table or vector associating a value with each system state; these are produced by iterating over the state space. Since state spaces grow exponentially with the number of domain features, the direct application of these models to AI planning problems is limited. As a consequence, much MDP research in AI has focussed on representations and algorithms that allow complex planning problems to be specified concisely and solved effectively. Techniques such as function approximation [Bertsekas and Tsitsiklis, 1996] and state aggregation [Boutilier *et al.*, 1999] have proven reasonably effective at solving MDPs with very large state spaces.

One such approach with a strong connection to classical planning is the *decision-theoretic regression (DTR)* model [Boutilier *et al.*, 2000a]. The state space of an MDP is characterized by a number of random variables (e.g., propositions) and the domain is specified using logical representations of actions that capture the regularity in the effects of actions. For instance, Bayesian networks, decision trees, algebraic decision diagrams (ADDs), and probabilistic extensions of STRIPS can all be used to concisely represent stochastic actions in MDPs. These representations are exploited in the construction of a logical representation of the optimal value function and policy, thereby obviating the need for explicit state space enumeration. This process can be viewed as automatic state space abstraction and has been able to solve fairly substantial problems. For instance, the SPUDD algorithm [Hoey *et al.*, 1999] has been used to solve MDPs with hundreds of millions of states optimally, producing logical descriptions of value functions that involve only hundreds of distinct values. This works suggests that very large MDPs, if described in a logical fashion, can often be solved optimally by exploiting the logical structure of the problem.

Unfortunately, existing DTR algorithms are all designed to work with *propositional* representations of MDPs, while many realistic planning domains are best represented in first-order terms, exploiting the existence of domain objects, relations over those objects, and the ability to express objectives and action effects using quantification. Existing DTR algorithms can only be applied to these problems by grounding or "propositionalizing" the domain.[1] Unfortunately such an approach is impractical: the number of propositions grows very quickly with the number of domain objects and relations, and even relatively simple domains can generate incredibly large numbers of propositions when grounded. The number of propositions has a dramatic impact on the complexity of these algorithms. Specifying and reasoning with intuitively simple domain properties involving quantification becomes problematic in a propositional setting. For instance, a simple objective such as $\exists x \phi(x)$ (e.g., we want some widget at Factory 1) becomes the unwieldy $\phi(c_1) \vee \ldots \vee \phi(c_n)$, where the c_i are (relevant) constants (e.g., widget-1 is at Factory 1, or ...). Thus grounding our domain description deprives one of the naturalness and expressive power of relational representations and quantification in specifying dynamics and objective functions. Finally, existing DTR algorithms require explicit action enumeration when performing dynamic programming, which is also problematic in first-order domains, since the number of ground actions also grows dramatically with domain size.

[1] This assumes a *finite domain*: if the domain is infinite, these algorithms cannot generally be made to work.

In this paper we address these difficulties by proposing a decision-theoretic regression algorithm for solving *first-order MDPs (FOMDPs)*. We adopt the the representation for FOMDPs presented in [Reiter, 2001; Boutilier *et al.*, 2000b], in which stochastic actions and objective functions are specified using the situation calculus. We derive a version of value iteration [Bellman, 1957] that constructs first-order representations of value functions and policies by exploiting the logical structure of the MDP. The algorithm constructs a minimal partitioning of state space, represented by a set of first-order formulae, and associates values (or action choices) with each element of the partition.

As a consequence, our dynamic programming algorithm solves first-order MDPs without explicit state space *or* action enumeration, and without propositionalizing the domain. Furthermore, the technique we propose can be used to reason purely *symbolically* about value and optimal action choice. Our model can be viewed as providing a tight, seamless integration of classic knowledge representation techniques and reasoning methods with solution algorithms for MDPs.

This paper should be viewed as providing the theoretical foundations for first-order decision-theoretic regression. We are encouraged by the success of DTR methods for propositional MDPs, where it has been demonstrated that many MDPs have value functions and policies that can be represented very concisely using logical techniques. We have no doubt that the use of relations and quantification will ultimately enhance these methods tremendously.

We review MDPs in Section 2, and briefly describe our representation of FOMDPs in Section 3. We derive our symbolic dynamic programming technique in detail in Section 4 and discuss various implementation issues in Section 5. We conclude with a discussion of future directions.

2 Markov Decision Processes

We begin with the standard state-based formulation of MDPs. We assume that the domain of interest can be modeled as a fully-observable MDP [Bellman, 1957; Puterman, 1994] with a finite set of states $\mathcal{S}$ and actions $\mathcal{A}$. Actions induce stochastic state transitions, with $\Pr(s,a,t)$ denoting the probability with which state t is reached when action a is executed at state s. We also assume a real-valued reward function R, associating with each state s its immediate utility $R(s)$.[2]

A *stationary policy* $\pi : \mathcal{S} \rightarrow \mathcal{A}$ describes a particular course of action to be adopted by an agent, with $\pi(s)$ denoting the action to be taken in state s. The decision problem faced by the agent in an MDP is that of forming an *optimal policy* that maximizes expected total accumulated reward over an infinite horizon (i.e., the agent acts indefinitely). We compare policies by adopting *expected total discounted reward* as our optimality criterion, wherein future rewards are discounted at a rate $0 \leq \gamma < 1$, and the *value of a policy* π, denoted $V_\pi(s)$, is given by the expected total discounted reward accrued, that is, $E(\sum_{t=0}^{\infty} \gamma^t R(s^t)|\pi, s)$. Policy π is *optimal* if $V_\pi \geq V_{\pi'}$ for all $s \in \mathcal{S}$ and policies π'. The *optimal value function* V^* is the value of any optimal policy.

[2]We ignore actions costs for ease of exposition. These impose no additional complications on our model.

Value iteration [Bellman, 1957] is a simple iterative approximation algorithm for constructing optimal policies. It proceeds by constructing a series of n-stage-to-go *value functions* V^n. Setting $V^0 = R$, we recursively define n-stage-to-go *Q-functions*:

$$Q^n(a,s) = R(s) + \left\{ \gamma \sum_{t \in \mathcal{S}} \Pr(s,a,t) \cdot V^{n-1}(t) \right\} \qquad (1)$$

and value functions:

$$V^n(s) = \max_a Q^n(a,s) \qquad (2)$$

The Q-function $Q^n(a,s)$ denotes the expected value of performing action a at state s with n stages to go and acting optimally thereafter. The sequence of value functions V^n produced by value iteration converges linearly to V^*. For some finite n, the actions that maximize Eq. (2) form an optimal policy, and V^n approximates its value. We refer to Puterman [1994] for a discussion of stopping criteria.

The definition of a Q-function can be based on any value function. We define $Q^V(a,s)$ exactly as in Eq. (1), but with arbitrary value function V replacing V^{n-1} on the right-hand side. $Q^V(a,s)$ denotes the value of performing a at state s, then acting in such a way as to obtain value V subsequently.

3 First-Order Representation of MDPs

Most planning domains are specified in terms of a set of random variables, which jointly determine the state of the system. For example, the system state may be the assignment of truth values to a set of propositional variables. In addition, these variables may themselves be structured, built from various relations, functions, and domain objects, that naturally lend themselves to a first-order representation. Representing and solving MDPs under such circumstances is generally impractical using classic state-based transition matrices and dynamic programming algorithms. The difficulty lies in the need to explicitly enumerate state and action spaces. State spaces grow exponentially with the number of propositional variables need to characterize the domain. Furthermore, in a first-order domain, the number of induced propositional variables can grow dramatically with the number of domain objects of interest.[3] Moreover, we are often interested in solving planning problems with infinite domains.

Several representations for propositionally-factored MDPs have been proposed, including probabilistic variants of STRIPS and dynamic Bayes nets [Boutilier *et al.*, 1999]. First-order representations have also been proposed for MDPs, including those of Poole [1997], and Geffner and Bonet [1998]. In this paper we adopt the first-order, situation calculus MDP representation developed by Reiter [2001], and by Boutilier *et al.* [2000b] for use in the DTGolog framework. This model has several unique features that make dynamic programming techniques viable. We first review this representational language and methodology, and then show how stochastic actions can be represented in this framework. We also introduce some notation to ease the specification of MDPs.

[3]An n-ary relation over a domain of size d induces d^n atoms.

3.1 The Situation Calculus

The situation calculus [McCarthy, 1963] is a first-order language for axiomatizing dynamic worlds. In recent years, it has been considerably extended beyond the "classical" language to include processes, concurrency, time, etc., but in all cases, its basic ingredients consist of *actions*, *situations* and *fluents*.

Actions

Actions are first-order terms consisting of an action function symbol and its arguments. For example, the action of putting block b on the table might be denoted by the action term $\mathit{putTbl}(b)$.

Situations

A *situation* is a first-order term denoting a sequence of actions. These are represented using a binary function symbol *do*: $\mathit{do}(\alpha, s)$ denotes the sequence resulting from adding the action α to the sequence s. The special constant S_0 denotes the *initial situation*, namely the empty action sequence. Thus, $\mathit{do}(\alpha, s)$ is like LISP's $\mathit{cons}(\alpha, s)$ and S_0 is like LISP's ( ). In a blocks world, the situation term

$$\mathit{do}(\mathit{stack}(A, B), \mathit{do}(\mathit{putTbl}(B), \mathit{do}(\mathit{stack}(C, D), S_0)))$$

denotes the sequence of actions

$$[\mathit{stack}(C, D), \mathit{putTbl}(B), \mathit{stack}(A, B)].$$

Foundational axioms for situations are given in [Pirri and Reiter, 1999].

Fluents

Relations whose truth values vary from state to state are called *fluents*, and are denoted by predicate symbols whose last argument is a situation term. For example, $\mathit{BIn}(b, Paris, s)$ is a relational fluent meaning that in that state reached by performing the action sequence s, box b is in Paris.

Axiomatizing a Domain Theory

A domain theory is axiomatized in the situation calculus with four classes of axioms [Pirri and Reiter, 1999]:

1. **Action precondition axioms**: There is one axiom for each action function $A(\vec{x})$, with syntactic form

   $$\mathit{Poss}(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$$

   Here, $\Pi_A(\vec{x}, s)$ is a formula with free variables among $\vec{x}, s$. These characterize the preconditions of action A.
2. **Successor state axioms**: There is one such axiom for each fluent $F(\vec{x}, s)$, with syntactic form

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

   where $\Phi_F(\vec{x}, a, s)$ is a formula with free variables among $a, s, \vec{x}$. These characterize the truth values of the fluent F in the next situation $\mathit{do}(a, s)$ in terms of the current situation s, and they embody a solution to the frame problem for deterministic actions [Reiter, 1991].
3. **Unique names axioms for actions**: These state that the actions of the domain are pairwise unequal.
4. **Initial database**: This is a set of first-order sentences whose only situation term is S_0 and it specifies the initial state of the domain. The initial database will play no role in this paper.

Regression in the Situation Calculus

The regression of a formula ψ through an action a is a formula ψ' that holds prior to a being performed iff ψ holds after a. Successor state axioms support regression in a natural way. Suppose that fluent F's successor state axiom is $F(\vec{x}, \mathit{do}(a, s)) \equiv \Phi_F(\vec{x}, a, s)$. We inductively define the regression of a formula whose situation arguments all have the form $\mathit{do}(a, s)$ as follows:

$$\mathit{Regr}(F(\vec{x}, \mathit{do}(a, s))) = \Phi_F(\vec{x}, a, s)$$

$$\mathit{Regr}(\neg\psi) = \neg \mathit{Regr}(\psi)$$

$$\mathit{Regr}(\psi_1 \wedge \psi_2) = \mathit{Regr}(\psi_1) \wedge \mathit{Regr}(\psi_2)$$

$$\mathit{Regr}((\exists x)\psi) = (\exists x)\mathit{Regr}(\psi)$$

3.2 Stochastic Actions and the Situation calculus

For the purposes of representing probabilistic uncertainty, the above ontology and axiomatization for the situation calculus might appear to be inadequate, because all actions must be deterministic. One can see this requirement most clearly in the syntactic form of successor state axioms where a fluent's truth value in the next situation is uniquely determined by the current situation; thus, the next state is uniquely determined by the present state and the action performed. How then can stochastic actions be represented in the situation calculus? The trick is to decompose stochastic actions into deterministic primitives under nature's control—she chooses the deterministic action that actually gets executed, with some specified probability, when an agent performs a stochastic action. We then formulate situation calculus domain axioms using these deterministic choices [Bacchus *et al.*, 1995; Reiter, 2001; Boutilier *et al.*, 2000b].

We illustrate this approach with a simple example in a logistics domain consisting of cities, trucks, and boxes: boxes can be loaded onto and unloaded from trucks, and trucks can be driven between cities.

Nature's Choices for Stochastic Actions: For each stochastic action we must specify the deterministic choices available to nature. For instance, the stochastic *load* action can succeed (denoted by *loadS*) or fail (*loadF*):

$$\mathit{choice}(\mathit{load}(b, t), a) \equiv a = \mathit{loadS}(b, t) \vee a = \mathit{loadF}(b, t)$$

Similarly, the stochastic *unload* and *drive* actions also decompose into successful or unsuccessful alternatives chosen by nature with known probabilities.

$$\mathit{choice}(\mathit{unload}(b, t), a) \equiv a = \mathit{unloadS}(b, t) \vee a = \mathit{unloadF}(b, t)$$

$$\mathit{choice}(\mathit{drive}(t, c), a) \equiv a = \mathit{driveS}(t, c) \vee a = \mathit{driveF}(t, c)$$

Probabilities for Nature's Choices: For each of nature's choices $n(\vec{x})$ associated with action $A(\vec{x})$, we specify the probability $\mathit{prob}(n(\vec{x}), A(\vec{x}), s)$ with which it is chosen, given that $A(\vec{x})$ was performed in situation s:

$$\mathit{prob}(\mathit{loadS}(b, t), \mathit{load}(b, t), s) = 0.99$$

$$\mathit{prob}(\mathit{loadF}(b, t), \mathit{load}(b, t), s) = 0.01$$

$$\mathit{prob}(\mathit{unloadS}(b, t), \mathit{unload}(b, t), s) = p \equiv$$

$$Rain(s) \wedge p = 0.7 \vee \neg Rain(s) \wedge p = 0.9$$

$$prob(unloadF(b,t), unload(b,t), s) = 1 - prob(unloadS(b,t), unload(b,t), s)$$

$$prob(driveS(t,c), drive(t,c), s) = 0.99$$

$$prob(driveF(t,c), drive(t,c), s) = 0.01$$

Here we see that unloading is less likely to succeed when it is raining.

Action Preconditions for Deterministic Actions:

$$Poss(loadS(b,t), s) \equiv (\exists c).BIn(b,c,s) \wedge TIn(t,c,s)$$

$$Poss(loadF(b,t), s) \equiv (\exists c).BIn(b,c,s) \wedge TIn(t,c,s)$$

$$Poss(unloadS(b,t), s) \equiv On(b,t,s)$$

$$Poss(unloadF(b,t), s) \equiv On(b,t,s)$$

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

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

Nature's choices $n_j(\vec{x})$ for action $A(\vec{x})$ need not have common preconditions, but often they do, as above.

Successor State Axioms:

$$BIn(b,c,do(a,s)) \equiv (\exists t)[TIn(t,c,s) \wedge a = unloadS(b,t)] \vee BIn(b,c,s) \wedge \neg(\exists t) a = loadS(b,t)$$

$$TIn(t,c,do(a,s)) \equiv a = driveS(t,c) \vee TIn(t,c) \wedge \neg(\exists c') a = driveS(t,c')$$

$$On(b,t,do(a,s)) \equiv a = loadS(b,t) \vee On(b,t,s) \wedge a \neq unloadS(b,t)$$

$$Rain(do(a,s)) \equiv Rain(s)$$

There are two important points to note about this example:

1. By virtue of decomposing stochastic actions into deterministic primitives under nature's control, we get perfectly conventional situation calculus action precondition and successor state axioms *that do not refer to stochastic actions.* Stochastic actions have a status different from deterministic actions, and cannot participate in situation terms.[4]
2. *Nowhere do these axioms restrict the domain of discourse to some prespecified set of trucks, boxes, or cities.* There are even models of these axioms with infinitely many—even uncountably many—individuals. If one were to solve an MDP for which this axiomatization is valid, one would obtain, in fact, a solution that applies to an entire class of MDPs with arbitrary domains of trucks, boxes and cities.

3.3 Some Additional Notation

In what follows we use the notion of a *state formula*, $\psi(\vec{x}, s)$, whose only free variables are non-situation variables $\vec{x}$ and a situation variable s. Intuitively, a state formula refers only to properties of the situation s. A set of state formulae $\{\psi_i(\vec{x}, s)\}$ *partitions* state space iff $\models (\forall \vec{x}, s).\psi_i(\vec{x}, s) \supset \neg\psi_j(\vec{x}, s)$, for all $i, j \neq i$, and $\models (\forall \vec{x}, s). \bigvee_i \psi_i(\vec{x}, s)$.

[4] Note that when nature's choices for a specific action do not have identical preconditions, care must be taken in the axiomatization to ensure the probabilities sum to one in every situation.

The Case Notation

To simplify the presentation, we introduce the notation

$$t = case[\phi_1, t_1; \cdots; \phi_n, t_n]$$

as an abbreviation for the formula

$$\bigvee_{i \leq n} \{\phi_i \wedge t = t_i\}$$

where the ϕ_i are state formulae and the t_i are terms. We sometimes write this $case[\phi_i, t_i]$. Often the t_i will be constants and the ϕ_i will partition state space. We introduce the following operators on case statements (whose use will be important in the next section):

$$case[\phi_i, t_i : i \leq n] \otimes case[\psi_j, v_j : j \leq m] = case[\phi_i \wedge \psi_j, t_i \cdot v_j : i \leq n, j \leq m]$$

$$case[\phi_i, t_i : i \leq n] \oplus case[\psi_j, v_j : j \leq m] = case[\phi_i \wedge \psi_j, t_i + v_j : i \leq n, j \leq m]$$

$$case[\phi_i, t_i : i \leq n] \ominus case[\psi_j, v_j : j \leq m] = case[\phi_i \wedge \psi_j, t_i - v_j : i \leq n, j \leq m]$$

$$case[\phi_i, t_i : i \leq n] \cup case[\psi_j, v_j : j \leq m] = case[\phi_1, t_1; \cdots \phi_n, t_n; \psi_1, v_1; \cdots \psi_m, v_m]$$

Representing Probabilities with Case Notation

Let $A(\vec{x})$ be a stochastic action type with possible outcomes $n_1(\vec{x}), \cdots, n_k(\vec{x})$. We assume the probabilities of these outcomes are specified using case notation. Specifically, the choice probabilities for $n_j(\vec{x})$ are given as: $prob(n_j(\vec{x}), A(\vec{x}), s) = case[\phi_1^j(\vec{x}, s), p_1^j; \cdots; \phi_n^j(\vec{x}, s), p_n^j]$,

where the ϕ_i^j partition state space, and p_i^j is the probability of choice $n_i(\vec{x})$ being realized under condition $\phi_i^j(\vec{x}, s)$ when the agent executes stochastic action $A(\vec{x})$.

Our *unload* stochastic action above is represented in case notation as:

$$prob(unloadS(b,t), unload(b,t), s) = case[Rain(s), 0.7; \neg Rain(s), 0.9]$$

$$prob(unloadF(b,t), load(b,t), s) = case[Rain(s), 0.3; \neg Rain(s), 0.1].$$

Notice that when the probability of nature's choice is situation-independent, (e.g., as in *loadS*), then only a single "case" is present (e.g., $case[true, 0.99]$).

Specifying Rewards and Values with Case Notation

An MDP optimization theory contains axioms specifying the reward function. In their simplest form, reward axioms use the function $R(s)$ to assert costs and rewards as a function of the action taken, properties of the current situation, or both (note that the action taken can be recovered from the situation term). In what follows, we assume a simple "state-based" reward model in which only relational fluents determine reward, and we assume that this reward function is specified using case notation:

$$R(s) = case[\xi_1(s), r_1; \cdots; \xi_m(s), r_m],$$

where the $\xi_i(s)$ partition state space. For example, rewarding the presence of *some* box in Paris can be specified using

$$R(s) = case[(\exists b) BIn(b, Paris, s), 10; \\ \neg(\exists b) BIn(b, Paris, s), 0]$$

The restriction to state-based reward is simply to keep the exposition simple. Action costs are easily modeled and are used in our prototype implementation.

We also use the case notation to represent value functions in a similar fashion, concisely writing V in the form

$$V(s) = case[\beta_1(s), v_1; \cdots \beta_n(s), v_n].$$

This use of case statements can be viewed as embodying a form of state space *abstraction*: rather than assigning values on a state-by-state basis, we distinguish states according to the conditions β_i. Those states satisfying β_i can be treated as an *abstract state*. In this way, we can often represent value functions (and policies and Q-functions similarly) without state enumeration, exploiting the logical structure of the function. This is similar to the abstraction models discussed in [Boutilier *et al.*, 1999], but with the ability to partition state space using first-order formulae.

4 Dynamic Programming with FOMDPs

Logical representations for MDPs provide natural and compact specifications of planning domains, obviating the need for explicit state space enumeration. Logical descriptions exploiting regularities in value functions and policies can also be very compact. Solving an FOMDP can be made much more efficient if the logical structure of value functions can be discovered through inference using the the logical MDP specification, with expected value computations performed once per abstract state instead of once per state. Thus a dynamic programming algorithm that works directly with symbolic representations of value functions offers great potential computational benefit. In this section, we generalize the notion of decision-theoretic regression from propositional MDPs to FOMDPs, and construct a *first-order value iteration* algorithm.

4.1 First-Order Decision-Theoretic Regression

Suppose we are given a value function V. The *first-order decision theoretic regression (FODTR)* of V through action type $A(\vec{x})$ is a logical description of the Q-function $Q^V(A(\vec{x}), s)$. In other words, given a set of abstract states corresponding to regions of state space where V is constant, we wish to produce a corresponding abstraction for $Q^V(A(\vec{x}), s)$. This is analogous to classical goal regression, the key differences being that action $A(\vec{x})$ is stochastic.

Let $A(\vec{x})$ be a stochastic action with corresponding nature's choices $n_j(\vec{x}), j \le k$. Ignoring preconditions momentarily, $Q^V(A(\vec{x}), s)$ is defined classically as

$$Q^V(A(\vec{x}), s) = R(s) + \gamma \cdot \{\textstyle\sum_{t \in \mathcal{S}} Pr(s, A(\vec{x}), t) \cdot V(t)\}$$

Since different successor states arise only through different nature's choices, the situation calculus analog of this is:

$$Q^V(A(\vec{x}), s) = R(s) + \\ \gamma \cdot \sum_j prob(n_j(\vec{x}), A(\vec{x}), s) \cdot V(do(n_j(\vec{x}), s)) \quad (3)$$

As described earlier, we assume that the functions $R(s)$, $prob(n, A, s)$ and $V(s)$ are all described with case statements. Respectively denote these by $rCase(s)$, $pCase(n, s)$ and $vCase(s)$. Then after substituting these case expressions into Eq. (3) and appealing to the case addition and multiplication operators of Section 3.3, we obtain

$$Q^V(A(\vec{x}), s) = rCase(s) \oplus \\ \gamma \cdot [\oplus_j \{pCase(n_j(\vec{x}), s) \otimes vCase(do(n_j(\vec{x}), s))\}]$$

The only problem with this expression is that the formula $vCase(do(n_j(\vec{x}), s))$ refers not to the current situation s, but to the future situation $do(n_j(\vec{x}), s))$, but this is easily remedied with regression:

$$Q^V(A(\vec{x}), s) = rCase(s) \oplus \\ \gamma \cdot [\oplus_j pCase(n_j(\vec{x}), s) \otimes Regr(vCase(do(n_j(\vec{x}), s)))]$$

We emphasize the critical nature of this step. The representational methodology we adopt—treating stochastic actions using *deterministic* nature's choices—allows us to apply regression directly to derive *properties of the pre-action state that determine the value-relevant properties of the post-action state*. Specifically, classical regression can be applied *directly* to the case statement $vCase(do(n_j(\vec{x}), s))$ because the $n_j(\vec{x})$ are deterministic.

Because sums and products of case statements are also case statements, the above expression for $Q^V(A(\vec{x}), s)$ is a case statement, say $case[\alpha_i(\vec{x}, s), q_i]$, that characterizes the Q-function for action $A(\vec{x})$ with respect to V. Thus from a logical description of V we can derive one for Q. Conceptually, this can be viewed as transforming the abstraction of state space suitable for V into one suitable for Q. It is not hard to show that if the state formulae in V's case statement partition the state space, then so do the α_i defining Q. This is key to avoiding state and action enumeration in dynamic programming.

The above derivation ignores action preconditions. To handle preconditions, $Q^V(A(\vec{x}), s)$ can no longer be treated as a function, but must be represented by a *relation* $Q^V(A(\vec{x}), q, s)$, meaning that A's Q-value in s is q. This relation holds only if $Poss(n_i(\vec{x}), s)$ holds for at least one of A's choices n_i; otherwise the Q-value is undefined:

$$Q^V(A(\vec{x}), q, s) \equiv \\ [\textstyle\bigvee_i Poss(n_i(\vec{x}), s)] \wedge q = case[\alpha_i(\vec{x}, s), q_i]$$

Since $\bigvee_i Poss(n_i(\vec{x}), s)$ can be distributed into the case statement (by conjoining it with the α_i), the result is again a case statement for the Q-relation.

As an example consider value function V^0:

$$V(s) = case[\exists b.BIn(b, Rome, s), 10 \; ; \; \neg\exists t.BIn(b, Rome, s), 0]$$

That is, if some box b is in *Rome*, value is 10; otherwise value is 0. Suppose that reward R is identical to V^0 and our discount rate is 0.9. We use the $unload(b, t)$ action, described above, to illustrate FODTR. The regression of V^0 through $unload(b, t)$ results in a case statement (after simplification), denoting $Q^1(unload(b, t), q, s)$ with four elements:

$$\alpha_1(b, t, s) \equiv \exists b' BIn(b', Rome, s)$$

$$\alpha_2(b, t, s) \equiv Rain(s) \wedge TIn(t, Rome, s) \wedge$$

$$On(b,t,s) \wedge \neg\exists b' BIn(b', Rome, s)$$

$$\alpha_3(b,t,s) \equiv \neg Rain(s) \wedge TIn(t, Rome, s) \wedge On(b,t,s) \wedge \neg\exists b' BIn(b', Rome, s)$$

$$\alpha_4(b,t,s) \equiv (\neg TIn(t, Rome, s) \vee \neg On(b,t,s)) \wedge \exists b' BIn(b', Rome, s)$$

and the associated Q-values: $q_1 = 19$; $q_2 = 6.3$; $q_3 = 8.1$; $q_4 = 0$. Before simplification, the case statement consisted of 8 formulae, two of which were inconsistent and two pairs of which had identical Q-values.

An important property of FODTR is that it not only produces an abstraction of state space to describe $Q^V(A(\vec{x}), q, s)$, it also abstracts the action space as well. With a small number of logical formulae, it captures the Q-values $Q(a, q, s)$ for each situation s and *each instantiation* a of $A(\vec{x})$. While state space abstraction has been explored in the context of decision-theoretic regression for propositional representations of MDPs, little work has focused on abstracting the action space in this way.

Finally, although our example works with specific numerical values in the case statements, purely *symbolic* descriptions of value can also be reasoned with in this way. For example, if the Q-value of action $drive(t, c)$ depends on the weight of truck t in the current situation, the value term in a case statement can be made to depend on this property of the situation (i.e., $weight(t, s)$). This can prove especially useful for reasoning with continuous (or hybrid) state and action spaces.

4.2 Symbolic Dynamic Programming

Value iteration consists of setting $V^0 = R$ and repeatedly applying Eq. (1) and Eq. (2) until a suitable termination condition is met. Since R is described symbolically and FODTR can be used to implement Eq. (1) logically, we need only derive a "logical implementation" of Eq. (2) in order to have a form of dynamic programming that can compute optimal policies for FOMDPs without explicit state or action enumeration (together with a method for termination testing and policy extraction).

In what follows, we assume that all values occurring in the case statements for $Q(A, v, s)$ are numerical constants, which means that the case statements for $prob(n, A, s)$, $V(s)$ and $R(s)$ all have this property.

Suppose we have computed n-stage-to-go Q-relations $Q(A(\vec{x}), v, s)$, one for each action type A, of the form $case[\alpha_i^A(\vec{x}, s), q_i^A]$, where the q_i^A are numerical constants. Letting $V(s)$ denote the n-stage-to-go value function, Eq. (2) can be written

$$V(s) = v \equiv (\exists a).Q(a, v, s) \wedge (\forall b) Q(b, w, s) \supset w \leq v \quad (4)$$

We assume that *some* stochastic action (e.g., a deterministic no-op) is executable in every situation, so that $V(s)$ will be a function. (If not, we can easily define it as a relation.) We now derive a series of expressions for the r.h.s. of this equivalence. Assuming domain closure for action types (i.e., all actions a are instances of some $A_i(\vec{x}_i)$, we have

$$V(s) = v \equiv [\bigvee_i (\exists \vec{x}_i) Q(A_i(\vec{x}_i), v, s)] \wedge \bigwedge_j (\forall \vec{y}_j, v').Q(A_j(\vec{y}_j), v', s) \supset v' \leq v$$

To minimize notational clutter, represent this generically by

$$V(s) = v \equiv [\bigvee_A (\exists \vec{x}) Q(A(\vec{x}), v, s)] \wedge \bigwedge_B (\forall \vec{y}, v').Q(B(\vec{y}), v', s) \supset v' \leq v$$

We are supposing that we have already determined the Q-values for each action type A, in the form of a case statement:

$$Q(A(\vec{x}), q, s) \equiv q = case[\alpha_i^A(\vec{x}, s), q_i^A] \quad (5)$$

Substitute Eq. (5) into the previous expression to get

$$V(s) = v \equiv \left[\bigvee_A (\exists \vec{x}) v = case[\alpha_i^A(\vec{x}, s), q_i^A]\right] \wedge \bigwedge_B (\forall \vec{y}, v').Q(B(\vec{y}), v', s) \supset v' \leq v$$

Since the q_i^A are constants, we can distribute the existential quantifiers into the case expression:

$$V(s) = v \equiv \left[\bigvee_A v = case[(\exists \vec{x}) \alpha_i^A(\vec{x}, s), q_i^A]\right] \wedge \bigwedge_B (\forall \vec{y}, v').Q(B(\vec{y}), v', s) \supset v' \leq v$$

Writing $(\exists \vec{x}) \alpha_i^A(\vec{x}, s)$ as $\gamma_i^A(s)$, and recalling the definition of the case union operator $\cup$ of Section 3.3, we have

$$V(s) = v \equiv v = \left[\bigcup_A case[\gamma_i^A(s), q_i^A]\right] \wedge \bigwedge_B (\forall \vec{y}, v').Q(B(\vec{y}), v', s) \supset v' \leq v$$

Suppose $\bigcup_A case[\gamma_i^A(s), q_i^A]$ has the form $case[\gamma_i(s), V_i : i \leq k]$. Therefore,

$$V(s) = v \equiv [\bigvee_{i=1}^{k} \gamma_i(s) \wedge v = V_i] \wedge \bigwedge_B (\forall \vec{y}, v').Q(B(\vec{y}), v', s) \supset v' \leq v$$

This simplifies to

$$V(s) = v \equiv \bigvee_{i=1}^{k} \gamma_i(s) \wedge \bigwedge_B (\forall \vec{y}, v')[Q(B(\vec{y}), v', s) \supset v' \leq V_i] \wedge v = V_i$$

Recalling the definition of the case notation, we get

$$V(s) = v \equiv v = case[\gamma_i(s) \wedge \bigwedge_B (\forall \vec{y}, v').Q(B(\vec{y}), v', s) \supset v' \leq V_i, V_i : i \leq k]$$

The only remaining task is to characterize the expressions $Q(B(\vec{y}), v', s) \supset v' \leq V_i$ in terms of the case statement for $Q(B(\vec{y}), v', s)$. Suppose this case statement is:

$$Q(B(\vec{y}), v', s) \equiv v' = case[\beta_j^B(\vec{y}, s), q_j^B]$$

Then it is easy to show that

$$Q(B(\vec{y}), v', s) \supset v' \leq V_i \equiv \bigwedge_j [\beta_j^B(\vec{y}, s) \supset q_j^B \leq V_i]$$

Substituting this last expression for $Q(B(\vec{y}), v', s) \supset v' \leq V_i$ in the above expression for $V(s)$ gives us

$$V(s) = v \equiv v = case[\gamma_i(s) \wedge \bigwedge_B (\forall \vec{y}). \bigwedge_j [\beta_j^B(\vec{y}, s) \supset q_j^B \leq V_i], V_i : i \leq k]$$

Next, because the q_j^B and V_i are numerical constants, we can distribute the universal quantifier as an existential quantifier in the antecedent of the implications, to get

$$V(s) = v \equiv v = case[\gamma_i(s) \wedge \bigwedge_B \bigwedge_j [(\exists \vec{y}) \beta_j(\vec{y}, s) \supset q_j^B \leq V_i], V_i : i \leq k]$$

Next, recalling how the γ_i were introduced by unioning the case expressions for all the Q-values, we get

$$V(s) = v \equiv$$

$$v = case[\gamma_i(s) \wedge \bigwedge_j [\gamma_j(s) \supset V_j \le V_i], V_i : i \le k]$$

Finally, we can again exploit the fact that the Vs are numerical constants (as opposed to symbolic terms), and therefore can be compared. This allows us to write our final expression for V:

$$V(s) = case \begin{bmatrix} \gamma_1(s) \wedge \bigwedge_{\{i|V_i > V_1\}} \neg\gamma_i(s) & V_1 \\ \vdots & \\ \gamma_k(s) \wedge \bigwedge_{\{i|V_i > V_k\}} \neg\gamma_i(s) & V_k \end{bmatrix}$$

If we modify the definition of the $\cup$ operator so that it sorts the rows according to their V values, and merges rows with identical V values, we get the pleasing expression

$$V(s) = \\ case \begin{bmatrix} \gamma_1(s) & V_1 \\ \gamma_2(s) \wedge \neg\gamma_1(s) & V_2 \\ \vdots & \\ \gamma_k(s) \wedge \neg\gamma_1(s) \wedge \neg\gamma_2(s) \wedge \cdots \wedge \neg\gamma_{k-1}(s) & V_k \end{bmatrix} \quad (6)$$

This determines a simple case statement that completely defines the value function $V^n(s)$ given the logical description of the relations $Q^n(A(\vec{x}), v, s)$. Together with the FODTR algorithm for producing Q-relations, this provides the means to construct the sequence of value functions that characterize value iteration in a purely symbolic fashion, eliminating the need for state and action enumeration. It is not hard to show that the case conditions defining V^n partition state space.

Finally, notice that we obtained the case expression (6) by a sequence of equivalence-preserving transformations from the definition (3) of the Q-function (suitably modified to accommodate action preconditions), and the definition (4) of the value function. Therefore, we have:

Theorem 1 *The case expression (6) is a correct representation for $V(s)$ relative to the specifications (3) and (4) for the Q-function and value function respectively.*

With these pieces in place, we can summarize first-order value iteration as follows: given as input a first-order representation of $R(s)$ (a case statement) and our action model, we set $V^0(s) = R(s)$, $n = 1$ and perform the following steps until termination:

1. For each action type $A(\vec{x})$ compute the case representation of $Q^n(A(\vec{x}), q, s)$ (using $V^{n-1}(s)$ as in Eq. (3)).
2. Compute the case representation of $V^n(s)$ (using the $Q^n(A(\vec{x}), q, s)$ as in Eq. (6)).
3. Increment n.

Termination of first-order value iteration is straightforward. Given the case statements C^n and C^{n-1} for value functions V^n and V^{n-1}, we form $C^n \ominus C^{n-1}$ and simplify the resulting case statement by removal of any inconsistent elements. If each case has a value term less that specified threshold ε, value iteration terminates. Extraction of an optimal policy is also straightforward: one simply needs to extract the maximizing actions from the set of Q-functions derived from the optimal value function. The optimal policy will thus be represented symbolically with a case statement.

4.3 An Illustration

To give a flavor of the form of first-order value functions, consider an example where the reward function is given by three statements:

$(\exists b).BIn(b, Paris, s) \wedge TypeA(b); r = 10$
$(\exists b).BIn(b, Paris, s) \wedge \neg TypeA(b); r = 5$
$\neg(\exists b)BIn(b, Paris, s); r = 0$

That is, we want a box of Type A in Paris, but will accept a box of another type if a Type A box is unavailable. Actions include the load, unload, and drive actions described above. We include action costs: the action *unload*(b, t) has cost 4, *load*(b, t) has cost 1, and *drive*(t, c) has cost 3. The optimal one-stage policy chooses only unloading or no-op (since with only one stage to go, driving and loading have no value). Our algorithm derives the following conditions for *unload*(b, t) to be executed:

$$\begin{aligned} & On(b,t,s) \wedge TIn(t, Paris, s) \wedge \\ & \quad [\neg(\exists b')BIn(b', Paris, s) \vee \\ & \qquad TypeA(b) \wedge \neg Rain(s) \wedge \\ & \qquad \neg(\exists b').BIn(b', Paris, s) \wedge \neg TypeA(b')] \end{aligned}$$

Thus a box b is unloaded if there is a box on some truck in Paris, and there is no box currently in Paris, or b is a Type A box and it's not raining, and there's no Type A box in Paris. No-op is executed if the negation of the condition above holds (since for a one-step backup there is no value yet discovered for driving or loading). It is important to note that this partitioning remains fixed (as does the partitioning for the resultant value function) regardless of the number of domain objects and extraneous relations in the problem description. Thus we get stronger abstraction than would be possible using a propositionalized version of the problem. Also note that this describes the conditions under which one performs any *instance* of the unload action. In this way our algorithm allows for action abstraction, allowing one to produce value functions and policies without explicit enumeration of action instances.

5 A (Very) Preliminary Implementation

We have implemented (in Prolog) the basic Bellman backup operator (i.e., single iterations of one-step value iteration) defined by Eq (6). The implementation is based entirely on a rewrite interpreter that applies programmer specified rewrite rules to situation calculus formulae until no further rewrites are possible. The program first computes the case statements for the Q-values for all the stochastic actions. Next, from these it computes the $\langle\gamma_i(s), V_i\rangle$ pairs required by the case statement (6), and finally, the case statement of (6) itself. Throughout, logical simplification is applied (also specified by rewrite rules) to all subformulas of the current formula.

From a practical point of view, the key component in efficiently implementing first-order DTR is logical simplification to ensure manageable formulae describing the partitions. Our current implementation performs only the most rudimentary logical simplification and does not always produce concise descriptions of the cases within partitions. Neither can it eliminate all inconsistent partitions. The main reason for these limitations is that the current implementation lacks a first-order theorem-prover. For the example MDPs we have looked at, sophisticated theorem-proving appears not to be necessary, but simple-minded simplification rules that don't know very much about quantifiers are simply too weak.

We ran value iteration to termination under our implementation using the reward function that gives a reward of 10 for having any box in Paris, and zero reward otherwise (for simplicity, it is treated as a terminal reward, and is received only once). Because our simplifier did not include a theorem-prover, some of the intermediate computations were hand-edited to further simplify the resulting expressions. We obtained the following optimal value function:

$$\begin{array}{l}
\exists b BIn(Paris, b, s) : 10 \\
\neg Rain(s) \wedge \exists b, t(On(b, t, s) \wedge TIn(t, Paris, s)) \\
\quad \wedge \neg\exists b BIn(Paris, b, s) : 5.56 \\
Rain(s) \wedge \exists b, t(On(b, t, s) \wedge TIn(t, Paris, s)) \\
\quad \wedge \neg\exists b.BIn(Paris, b, s) : 4.29 \\
\neg Rain(s) \wedge \exists b, tOn(b, t, s) \wedge \neg\exists b BIn(Paris, b, s) \\
\quad \wedge \neg\exists b, t(On(b, t, s) \wedge TIn(t, Paris, s)) : 2.53 \\
\neg Rain(s) \wedge \exists b, t, c(BIn(c, s) \wedge TIn(c, s)) \\
\quad \wedge \neg\exists b, tOn(b, t, s) \wedge \neg\exists b BIn(Paris, b, s) : 1.52 \\
Rain(s) \wedge \exists b, tOn(b, t, s) \wedge \neg\exists b, t(On(b, t, s) \wedge TIn(t, Paris, s)) \\
\quad \wedge \neg\exists b BIn(Paris, b, s) : 1.26 \\
\neg\exists b BIn(Paris, b, s) \wedge \neg\exists b, tOn(b, t, s) \wedge \\
\quad [Rain(s) \vee \neg\exists b, t, c(BIn(c, s) \wedge TIn(c, s)) : 0.0
\end{array}$$

We emphasize again that this value function applies no matter how many domain objects there are.

Our algorithm is not competitive with state of the art propositional MDP solvers, largely because solvers such as SPUDD [Hoey *et al.*, 1999] use very efficient implementations of logical reasoning software. We are currently developing a version of the FODTR algorithm that uses a first-order theorem-prover to enhance its performance. Of course, at another level, one can argue that propositional MDP solvers cannot even get off the ground when (even trivial) planning problems have a large number of domain objects.

An important issue we hope to address in the near future is the use of hybrid representations of MDPs and value functions that allow one to adopt efficient data structures like ADDs or decision trees, but instantiate these structures with first-order formulae. This would allow the expressive power of our first-order model, but restrict the syntactic form of formulae somewhat so that simplification and consistency checking could be implemented more effectively for "typical" problem instances.

6 Concluding Remarks

We have described the first approach for solving MDPs specified in first-order logic by dynamic programming. By the careful integration of sophisticated KR methods with classic MDP algorithms, we have developed a framework in which MDPs can be specified concisely and naturally and solved without explicit state and action enumeration. Indeed, nothing in our model prevents its direct application to infinite domains. Furthermore, it permits the symbolic representation of value functions and policies.

A number of interesting directions remain to be explored. As mentioned, the practicality of this approach depends on the use of sophisticated simplification methods. We are currently incorporating several of these into our implementation. Other dynamic programming algorithms (e.g., modified policy iteration) can be implemented directly within our framework. Approximation methods based on merging partitions with similar values can also be applied with ease. Finally, the investigation of symbolic dynamic programming to continuous and hybrid domains offers exciting possibilities.

Acknowledgements: This research was supported by NSERC and IRIS Project BAC "Dealing with Actions." Thanks to the referees for their helpful suggestions on the presentation of this paper.

References

[Bacchus *et al.*, 1995] F. Bacchus, J. Y. Halpern, and H. J. Levesque. Reasoning about noisy sensors in the situation calculus. *IJCAI-95*, 1933–1940, Montreal, 1995.

[Bellman, 1957] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, 1957.

[Bertsekas and Tsitsiklis, 1996] D. P. Bertsekas and J.. N. Tsitsiklis. *Neuro-dynamic Programming*. Athena, Belmont, MA, 1996.

[Boutilier *et al.*, 1999] C. Boutilier, T. Dean, and S. Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *J. Art. Intel. Res.*, 11:1–94, 1999.

[Boutilier *et al.*, 2000a] C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic dynamic programming with factored representations. *Art. Intel.*, 121:49–107, 2000.

[Boutilier *et al.*, 2000b] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. *AAAI-2000*, 355–362, Austin, TX, 2000.

[Geffner and Bonet, 1998] H. Geffner and B. Bonet. High-level planning and control with incomplete information using POMDPs. *Fall AAAI Symp. on Cognitive Robotics*, Orlando, FL, 1998.

[Hoey *et al.*, 1999] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. SPUDD: Stochastic planning using decision diagrams. *UAI-99*, 279–288, Stockholm, 1999.

[McCarthy, 1963] J. McCarthy. Situations, actions and causal laws. Tech. report, Stanford Univ., 1963. Repr. *Semantic Information Processing* (M. Minsky ed.), MIT Press, Cambridge, 1968, 410-417.

[Pirri and Reiter, 1999] F. Pirri and R. Reiter. Some contributions to the metatheory of the situation calculus. *JACM*, 46(3):261–325, 1999.

[Poole, 1997] D. Poole. The independent choice logic for modelling multiple agents under uncertainty. *Art. Intel.*, 94(1–2):7–56, 1997.

[Puterman, 1994] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.

[Reiter, 1991] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, ed., *Artificial Intelligence and Mathematical Theory of Computation (Papers in Honor of John McCarthy)*, 359–380. Academic Press, 1991.

[Reiter, 2001] R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, Cambridge, MA, 2001.