Bellman Equations for Stochastic Programs

David McAllester

AT&T Labs-Research
180 Park Ave
Florham Park NJ 07932
dmac@research.att.com
http://www.research.att.com/~dmac

This is a revised version of a paper entitled "World Modeling vs. World Axiomatizing" which accompanied an invited talk at LPNMR-99.

Abstract. This paper gives a model-checking style algorithm for verifying the decision-theoretic utility of a hand-written robot controller. The paper presents a simple Turing-complete programming language for expressing robot control programs. The robot control language supports stochastic nondeterminism, complex data structures, conditional expressions, and recursive procedures. A similar Turing-complete programming language for writing models of the robot's environment is also given. The paper shows that if a given controller running in a given environment has only finitely many behaviors (as defined in the paper) then the value of that controller in that environment can be computed by iterating appropriately generalized Bellman equations.

1 Introduction

In complex robot control problems, such as robosoccer or museum tours, robot control software is often written by hand. Hand-written controllers seem appropriate for partially observable environments — environments where sensors provide only partial information about the current situation. A partially observable environment can be formalized as a partially observable Markov decision process (POMDP). In general, optimal control for POMDPs requires time exponential in the number of underlying states (unless P=PSPACE) [10]. Here we are interested in POMDPs, such as factored POMDPs, with a large set of underlying states. For a factored POMDP with, say, 20 Boolean state variables, optimal planning is out of the question.[1] This paper assumes a given hand-written robot control program and considers the more modest problem of verifying that this control program achieves a desired expected utility when running in a given POMDP environment. The POMDP environment is also assumed to be defined by a stochastic program. Factored POMDPs are easily represented by stochastic programs. The analysis algorithm is intended to handle POMDP environments

[1] For a special case where approximately optimal planning for factored POMDPs is possible see [7].

for which the underlying state space is small enough to be enumerated by computer. The technical content of this paper consists of Bellman equations for computing the expected utility achieved by a given robot controller. There is one such Bellman equation for each primitive construct in the controller programming language.

The Bellman equations developed here are similar to those given by Sutton, Precup and Singh [11]. There are three main differences between this paper and theirs. First, here we are concerned with partially observable rather than fully observable environments. Second, here we provide a Turing-complete high-level programming language for expressing both the robot controller and the model of the environement. Sutton, Precup and Singh do not view options as programs but rather as choice functions specifying an action for each state in an MDP. A program can express such choice functions very compactly at seems useful to give Bellman equations directly for compact representations. Finally, this paper focuses on verifying hand-written controllers while Sutton, Precup, and Singh focus on making the planning problem more tractable by introducing composite actions (options). In spite of these differences, the Bellman equations given here can be viewed as generalizations of the equations given by Sutton, Precup and Singh.

The Bellman equations developed here are also related to methods for computing the distribution of values of a stochastic program developed by Koller, McAllester and Pfeffer [6]. There are several differences however. First, here we compute the expected utility of a program as well the probability distribution over its values. Second, the programming languages studied here use the more familiar call by value semantics rather than call by need. Third, the algorithm given here requires a somewhat different termination condition than the algorithm of Koller et al. Consider a program that flips a coin and, depending on the outcome, either terminates or starts over. This program has only one state and terminates with probability 1 but there is no upper bound on the running time. The procedure given in this paper can handle finite state procedures which terminate with probability 1. The algorithm of Koller et all handles the more restricted class of finite state programs for which there exists an upper bound on the running time. On the other hand, the analysis algorithm of Koller et al. is more efficient in some cases — there are examples where it runs in polynomial time while the algorithm given here takes exponential time.

Thrun has developed a programming language, CES, specialized to the construction of robot control software [12]. Thrun's CES consists of C augmented with primitives for constructing and manipulating probability distributions and machine learning primitives such as back-propagation on neural networks. The ability to represent probability distributions facilitates the representation and manipulation of "belief states" — probabiity distributions over unkown aspects of the environment. Machine learning primitives facilitate the construction of robot control software that tunes parameters from experience with the world. The motivations of Thrun's work are considerably different from the motivation given here. Here we are concerned with the verification of robot controllers —

proving that a given controller achieves a desired return. Thrun is concerned with writing very sophisticated controllers — ones that manipulate belief states and tune parameters from experience. Thrun does not consider the problem of calculating the expected return of controllers written in CES. It seems likely that robot controllers that explicitly manipulate belief states and explicitly tune parameters will be difficult to automatically verify using the methods developed here. The development of verification methods for controlers that explicitly manipulate belief states seems a potentially fruitful direction for future work. Another direction for future work might be methods of automatically tuning parameters in given controllers. Note that this is different from writing controllers that explicitly update (tune) parameters.

Another programming language designed for the construction of robot controllers is the Golog language developed by Hector Levesque, Ray Reiter and others [5, 4]. The Golog language, unlike Thrun's CES, is designed to support verification. There are three significant differences between Golog and the approach taken here. First, Golog takes an axiomatic approach to modeling the robot environment. In the Golog approach the effect of primitive actions on the environment is specified axiomatically using a set of "frame axioms". Converting frame axioms to actual operations on states requires solving the so-called McCarthy frame problem [8]. In the approach taken here each action is defined by a stochastic program. When actions are defined by programs in a programming language the McCarthy frame problem does not arise. A second difference between this paper and the Golog work is stochasticity — Golog does not support stochastic controllers or stochastic world models. Finally, although both approaches allow robot controllers expressed as (essentially) computer programs, quite difference technical approaches are taken. The Golog programming language is formalized as a translation of declarative statements about programs into formulas of second order logic. Here the programming language is formalized in a more traditional manner — a traditional syntax and operational semantics is given.

It is interesting to contrast world-modeling and model-checking with world-axiomatizing and theorem-proving. The situation calculus [8] represents knowledge about states and actions in the form of first order axioms. In the formal verification community, however, there has been a growing movement away from this axiomatic approach toward a modeling approach [1, 9, 2]. In the modeling approach one specifies a particular model (structure) which approximates, or "models", the world. Model checking, often done with BDDs, is the process of evaluating expressions with respect to a given model. Modeling is also widely used in the area of reasoning under uncertainty. A stochastic model is one in which probabilities occur as part of the fundamental data of the model. Bayesian networks, Markov Decision processes (MDPs), Hidden Markov Models (HMMs), n-gram language models, and probabilistic context free grammars (PCFGs) are all examples of stochastic models. In each case some aspect of the world is modeled rather than axiomatized.

First order logic, or even higher order logic, seems inappropriate for modeling. Modeling languages, such as the PROMELA language for constructing models in the SPIN system [3], are similar to programming languages. A computer program can be viewed as a system of definitions. To see the fundamental difference between first order logic and a programming language, consider the following first order axiom for the predicate PATH where PATH(x, y) is intended to mean that there is a path from x to y in the graph defined by the predicate ARC.

$$\forall x, y, z \quad \mathtt{PATH}(x,\ y) \leftarrow \mathtt{ARC}(x, y) \vee [\mathtt{PATH}(x, z) \wedge \mathtt{PATH}(z, y)]$$

Note that this formula is true if we interpret PATH as the universally true predicate. The formula expresses a *constraint* on the meaning of PATH but fails to *define* it. Of course, any set of Horn clauses has a unique least fixed point which gives, in this case, the intended meaning of PATH. So Horn clauses under least fixed point semantics are a better modeling language than general first order logic. Horn clauses under least fixed point semantics can be viewed as a kind of programming language. Of course one could use a second order axiom to say that PATH is required to be the least predicate satisfying the above formula. But this seems much more awkward than simply defining PATH in an appropriate definitional language, such as Horn clauses under least fixed point semantics. In general, programming languages are designed to expression definitions while logics are designed to express constraints.

Semantics, as studied by logicians, is the relationship between statements and models — a given statement is either true or false when interpreted in a given model. Philosophers tend to think of semantics as a relationship between statements and the world. But it seems clearer to work with abstract formally defined models rather than the world itself. This is not just a technical convenience. Most, and perhaps all, real objects seem to reside in conceptual world models rather than the world itself. For example, consider this paper. I see words on a computer screen as I type this, but certainly the pixels on the screen are not "the paper". The paper is a document that can be emailed. Perhaps only the pixels on my screen and the bits in the computer's memory really exist. But the pixels themselves are only models of the real stuff — phosphor excited by electrons. But electrons are only a model of the true reality — the unified electroweak field. And so on, to some hypothetical ultimate model of physics. The world consists of models all the way down. Does this paper *really* exist? I don't care. I use a conceptual model where it does. The model is constructed, perhaps even defined, in my head.

2 Stochastic Programs

This section defines a simple first order stochastic programming language. This language serves two purposes in this paper. First, it is a kind of warm-up exercise for the action language of section 3. Second, as shown in section 4, it provides a way of compactly representing world models. The programming language is designed to be as simple as possible while allowing an exposition of the conceptual issue of stochasticity. More sophisticated languages are clearly possible and desirable but are not discussed here.

The data values are taken to be binary trees whose leaves are labeled by symbolic constants. These trees can be defined by the following grammar where A ranges over symbolic constants (character strings).

$$v ::= A \mid \texttt{CONS}[v_1,\ v_2]$$

I will use $\texttt{LIST}[v_1,\ v_2,\ \ldots,\ v_n]$ to abbreviate

$$\texttt{CONS}[v_1,\ \texttt{CONS}[v_2,\ \ldots\ \texttt{CONS}[v_n,\ \texttt{NIL}]\ldots]]$$

and $f[v_1,\ \ldots,\ v_n]$ where f is a constant other than CONS or LIST to abbreviate $\texttt{LIST}[f,\ v_1,\ \ldots,\ v_n]$. The syntax of the programming language is defined by the following grammar.

$$p ::= x \mid A \mid \texttt{CONS}[p_1,\ p_2]$$

$$e ::= x \mid A \mid \texttt{CONS}[e_1,\ e_2] \mid \texttt{FLIP}(\alpha) \mid g(e) \mid \texttt{CASE}\ e_0\ \texttt{OF}\ p_1 : e_1,\ \ldots,\ p_n : e_n$$

$$\mathcal{P} ::= [] \mid g(x) \equiv e\ ;\ \mathcal{P}$$

The grammar for p defines the set of "patterns" used in case statements. A pattern is just binary tree whose leaves are either symbolic constants or variables. In an expression of the form $\texttt{FLIP}(\alpha)$ we must have that α is a floating point number in the open interval $(0,\ 1)$. In the grammar g ranges over non-constructor function symbols. Only single argument functions are directly supported. However, one can easily simulate functions of many arguments. I will use $g(e_1,\ \ldots,\ e_n)$, with $n > 1$, as an abbreviation for $g(\texttt{LIST}[e_1,\ \ldots,\ e_n])$. Note that $\mathcal{P}$ ranges over programs, i.e., sequences of definitions for non-constructor function symbols. I will use $g(x_1,\ \ldots,\ x_n) \equiv b$, with $n > 1$, as an abbreviation for $g(y) \equiv \texttt{CASE}\ y\ \texttt{OF}\ \texttt{LIST}[x_1,\ \ldots,\ x_n] : b$. The following is a simple example of a syntactically well formed program.

$$\begin{aligned}\texttt{APPEND}(x,\ y) \equiv\ & \texttt{CASE}\ x\ \texttt{OF} \\ & \quad \texttt{CONS}[z,\ w] : \texttt{CONS}[z,\ \texttt{APPEND}(w,\ y)], \\ & \quad z : y\end{aligned}$$

$$\texttt{AT-LEAST}(x) \equiv \texttt{IF}(\texttt{FLIP}(1/2),\ x,\ \texttt{AT-LEAST}(\texttt{SUCC}[x]))$$

The case construct does pattern matching, i.e., the value of the expression $\texttt{CASE}\ e_0\ \texttt{OF}\ p_1 : e_1,\ \ldots,\ p_n : e_n$ is derived by finding the first pattern p_i that matches the value of e_0 and then taking the value of e_i under a variable interpretation derived from the match. If no pattern matches then an error occurs and no value is produced. In the second program $\texttt{IF}(e_0,\ e_1,\ e_2)$ is used as an

$$P(\langle x,\ \rho\rangle \to \rho(x)) = 1 \qquad P(\langle A,\ \rho\rangle \to A) = 1$$

$$P(\langle \texttt{FLIP}(\alpha),\ \rho\rangle \to T) = \alpha \qquad P(\langle \texttt{FLIP}(\alpha),\ \rho\rangle \to F) = 1 - \alpha$$

$$P(\langle \texttt{CONS}[e_1,\ e_2],\ \rho\rangle \to \texttt{CONS}[v_1,\ v_2]) = P(\langle e_1,\ \rho\rangle \to v_1)P(\langle e_2,\ \rho\rangle \to v_2)$$

$$P(\langle g(e),\ \rho\rangle \to w) = \sum_v P(\langle e,\ \rho\rangle \to v)P(\langle \texttt{body}(g),\ [\texttt{param}(g) \mapsto v]\rangle \to w)$$

$$P(\langle \texttt{CASE}\ e_0\ \texttt{OF}\ p_1 : e_1,\ \ldots,\ p_n : e_n,\ \rho\rangle \to w) =$$

$$\sum_{v \text{ matching some } p_i} \begin{array}{l} P(\langle e_0,\ \rho\rangle \to v)\ P(\langle e_i,\ \texttt{match}(p_i,\ v,\ \rho)\rangle \to w) \\ \text{where } p_i \text{ is the first pattern matching } v \end{array}$$

Fig. 1. Semantic equations for program expressions. If None of the above equations apply then $P(\langle e,\ \rho\rangle \to v) = 0$. The expressions $\texttt{param}(g)$ and $\texttt{body}(g)$ denote the variable x and expression b respectively such that the program contains $g(x) \equiv b$. The expression $\texttt{match}(p,\ v,\ \rho)$ denotes the substitution resulting from updating the substitution ρ with variable bindings derived from matching pattern p to value v.

abbreviation for $\texttt{CASE}\ e_0\ \texttt{OF}\ T : e_1,\ F : e_2$. The value of an expression of the form $\texttt{FLIP}(\alpha)$ is selected nondeterministically by selecting the constant $\texttt{T}$ with probability α and the constant F with probability $1 - \alpha$. A variable x occurs free in $\texttt{CASE}\ e_0\ \texttt{OF}\ p_1 : e_1,\ \ldots,\ p_n : e_n$ if either x occurs free in e_0 or there is some $p_i : e_i$ such that x does not occur in the pattern p_i but does occur free in e_i. In a definition $g(x) \equiv e$ we must have that e is an expression (as defined by the grammar) with no free variables other than x. In a program $\mathcal{P}$ we must have that no non-constructor symbol is defined twice and that every non-constructor symbol appearing in the program is defined in the program, i.e., if $\mathcal{P}$ contains the definition $g(x) \equiv e$, and g' appears in e, then $\mathcal{P}$ also contains a definition of the form $g'(x) \equiv e'$. Whenever a particular program $\mathcal{P}$ is being discussed we will implicitly restrict our discussion to program expressions e such that every non-constructor function symbol appearing in e is defined in $\mathcal{P}$.

A binding environment is a mapping ρ from a finite set of program variables, called the domain of ρ, to values. Given a program expression e, and a binding environment ρ whose domain includes the free variables of e, we can compute a value for e in environment ρ in the standard way. This evaluation process either generates an error, fails to terminate, or computes a particular value. When the program $\mathcal{P}$ is clear from context we let $P(\langle e,\ \rho\rangle \to v)$ denote the probability that evaluating e under binding ρ and program $\mathcal{P}$ terminates without error and returns v as the value. These probabilities satisfy the equations shown in figure 1.

The equations in figure 1 have a unique least fixed point which gives the intended meaning of $P(\langle e,\ \rho\rangle \to v)$. More formally, we define $P^0(\langle e,\ \rho\rangle \to v)$ to be 0 and define $P^{i+1}(\langle e,\ \rho\rangle \to v)$ by the equations in figure 1 where P^{i+1} is used on the left hand side of each equation and P^i is used on the right hand side. One can prove by induction on i that $P^{i+1}(\langle e,\ \rho\rangle \to v) \geq P^i(\langle e,\ \rho\rangle \to v)$ and that for any e and ρ we have $\sum_v P^i(\langle e,\ \rho\rangle \to v) \leq 1$. We can then define $P(\langle e,\ \rho\rangle \to v)$ to be $\lim_{i\to\infty} P^i(\langle e,\ \rho\rangle \to v)$. The fact that the values of P^i are monotonically increasing and bounded implies that this limit exists. Furthermore, the limit satisfies the equations in figure 1 and is the intended meaning of P.

Because of errors and nontermination it is possible that $\sum_v P(e \to v)$ is strictly less than one. The program $\mathcal{P}$ is called consistent if for all non-constructor functions symbols g defined in $\mathcal{P}$, and all values v, we have that $\sum_v P(g(v) \to v) = 1$. Note that the definition of `AT-LEAST` given above is consistent — the probability of error or nontermination is zero.

3 World Models and Robot Controllers

A robot must interact with an environment that is only partially under its control. Here we model the environment (the world) as a partially observable semi-Markovian decision process (POSDP). More formally, we adopt a discrete notion of time and assume the environment passes through an infinite sequence of states $s_0,\ s_1,\ s_2,\ \ldots$. The robot is assumed to have a certain set of "primitive actions" where a particular execution of a primitive action takes some integral amount of time during which the environment is in some sequence of states. We take a primitive action to be an expression of the form $h\{v\}$ where h is a primitive action function and v is a value. We use $h\{v_1,\ \ldots,\ v_n\}$, with $n > 1$,as an abbreviation for $h\{\texttt{LIST}[v_1,\ \ldots,\ v_n]\}$. For example, if `PUT-ON` is a primitive action function, and `A` and `B` are constants, then $\texttt{PUT-ON}\{A,\ B\}$ is a primitive action. We define a POSDP to be a set of possible states together with an interpretation of each primitive action as a stochastic function mapping a state s_i at time i to a pair $\langle\langle s_{i+1},\ \ldots\ s_{i+n}\rangle,\ w\rangle$ where n is the time taken by the action and w is a value called the observation yielded by the action. For example, the primitive action $\texttt{POT-ON}\{\texttt{A},\ \texttt{B}\}$ can take some integral amount of time and yield an observation such as the constant `SUCCEEDED` or the constant `FAILED`. A POSDP must be Markovian in the sense that the stochastic behavior of an action depends only on the state of the environment at the time the action is performed. The robot's selection of the next action can depend only on the observations returned by previous actions — the robot has no direct knowledge of the environment state. If $\mathcal{M}$ is a POSDP we write $P(\langle s,\ h\{v\}\rangle \to \langle\langle u_1,\ \ldots,\ u_m\rangle,\ w\rangle \mid \mathcal{M})$ to denote the probability that taking action $h\{v\}$ in state s takes m time steps during which the environment is in the states $u_1, \ldots, u_m$ and then yields the observation w. If $m = 0$ then the action is interpreted as a pure sensing action and is modeled as taking no time at all.

POSDPs generalizes Markov decision processes (MDPs), partially observable Markov decision process (POMDPs), and semi-Markovian decision process. A POMDP is (essentially) a POSDP in which every action takes one time step. A semi-Markovian decision process is a POSDP where the observation always uniquely determines the resulting state. An MDP is a POSDP with both these properties, i.e., every action takes one time step and the observation uniquely determines the resulting state. Here we work with general POSDPs.

Complex actions can be constructed from primitive actions. For example, consider the following *action expression*.

$$\texttt{CASE PUT-ON}(x,\ y)\ \texttt{OF}\ \ \texttt{SUCCEEDED} : \texttt{PUT-ON}(u,\ x),\ \texttt{FAILED} : \texttt{FAILED}$$

This action attempts to put x on y and if that succeeds then attempts to put u on x. A constant such as A can be viewed as a trivial action that takes no time, and hence leaves the state unchanged, and always yields itself as the observation. A variable such as x can also be viewed as an action which takes no time and yields its value as the observation. The syntax of action expressions is defined by the following grammar where patterns p are defined as before.

$$\begin{aligned} a ::= x \ \mid\ A \ \mid\ \texttt{CONS}[a_1,\ a_2] \ \mid\ g(a) \ \mid\ h\{a\} \ \mid \\ \texttt{FLIP}(\alpha) \ \mid\ \texttt{CASE}\ a_0\ \texttt{OF}\ p_1 : a_1,\ \ldots,\ p_n : a_n \end{aligned}$$

$$\mathcal{Q} ::= [] \ \mid\ g(x) \equiv a \ ;\ \ \mathcal{Q}$$

Note that, except for the introduction of a new form of function application for primitive action functions, the grammar is identical to the grammar for stochastic programs. Again we require that an action program $\mathcal{Q}$ have the property that the body of a definition does not have free variables other than the argument of the defined function, that no defined action function is defined more than once, and that every defined action function appearing in the program is defined in the program. Note that by including actions of the form FLIP(α) we allow for stochastic action selection. Action expressions can be executed in essentially the same way as program expressions except that calls to primitive action functions are handled by the POSDP environment rather than by the robot.

Consider the following composite action function for clearing a block. Here we assume a primitive sensing action function BLOCK-ON which takes a block and returns the block on top of that block as an observation about the world.

$$\begin{aligned} \texttt{CLEAR}(x) \equiv \texttt{CASE BLOCK-ON}(x)\ \texttt{OF} & \\ & \texttt{NONE} : \texttt{SUCCESS} \\ & \texttt{FAILURE[m]} : \texttt{FAILURE[m]} \\ & \texttt{BLOCK}[y] : \texttt{SEQ}(\texttt{CLEAR}(y),\ \texttt{PUT-ON}(y,\ \texttt{TABLE})) \end{aligned}$$

In this program we let SEQ(a_1, a_2) abbreviate the following.

$$\begin{aligned} &\texttt{CASE}\ a_1\ \texttt{OF} \\ &\quad \texttt{FAILURE[m]} : \texttt{FAILURE}[m] \\ &\quad \texttt{CASE}\ a_2\ \texttt{OF} \\ &\quad\quad \texttt{FAILURE}[m] : \texttt{FAILURE}[\texttt{AFTER}[\lceil a_1 \rceil,\ m]] \\ &\quad\quad z : z \end{aligned}$$

$$P(\langle s,\ x,\ \rho\rangle \rightarrow \langle\langle\rangle,\ \rho(x)\rangle) = 1 \qquad P(\langle s,\ A,\ \rho\rangle \rightarrow \langle\langle\rangle,\ A\rangle) = 1$$

$$P(\langle s,\ \mathtt{FLIP}(\alpha),\ \rho\rangle \rightarrow \langle\langle\rangle,\ T\rangle) = \alpha \qquad P(\langle s,\ \mathtt{FLIP}(\alpha),\ \rho\rangle \rightarrow \langle\langle\rangle,\ F\rangle) = 1 - \alpha$$

$$P(\langle s,\ \mathtt{CONS}[e_1,\ e_2],\ \rho\rangle \rightarrow \langle\sigma,\ \mathtt{CONS}[v_1,\ v_2]\rangle) =$$

$$\sum_{\gamma\delta=\sigma} P(\langle s,\ e_1,\ \rho\rangle \rightarrow \langle\gamma,\ v_1\rangle)\ P(\langle \mathtt{last}(s\gamma),\ e_2,\ \rho\rangle \rightarrow \langle\delta,\ v_2\rangle)$$

$$P(\langle s,\ h\{e\},\ \rho\rangle \rightarrow \langle\sigma,\ w\rangle) =$$

$$\sum_{v,\gamma\delta=\sigma} P(\langle s,\ e,\ \rho\rangle \rightarrow \langle\gamma,\ v\rangle)\ P(\langle \mathtt{last}(s\gamma),\ h\{v\}\rangle \rightarrow \langle\delta,\ w\rangle \mid \mathcal{M})$$

$$P(\langle s,\ g(e),\ \rho\rangle \rightarrow \langle\sigma,\ w\rangle) =$$

$$\sum_{v,\gamma\delta=\sigma} P(\langle s,\ e,\ \rho\rangle \rightarrow \langle\gamma,\ v\rangle)\ P(\langle \mathtt{last}(s\gamma),\ \mathtt{body}(g),\ [\mathtt{param}(g) \mapsto v]\rangle \rightarrow \langle\delta,\ w\rangle)$$

$$P(\langle s,\ \mathtt{CASE}\ e_0\ \mathtt{OF}\ p_1 : e_1,\ \ldots,\ p_n : e_n,\ \rho\rangle \rightarrow \langle\sigma,\ w\rangle) =$$

$$\sum_{\gamma\delta=\sigma,\ v \text{ mathing some } p_i} \begin{array}{l} P(\langle s,\ e_0,\ \rho\rangle \rightarrow \langle\gamma,\ v\rangle)\ P(\langle \mathtt{last}(s\gamma),\ e_i,\ \mathtt{match}(p_i,\ v,\ \rho)\rangle \rightarrow \langle\delta,\ w\rangle) \\ \text{where } p_i \text{ is the first pattern matching } v \end{array}$$

Fig. 2. Semantic equations for action expressions. If α is a non-empty sequence of states, then $\mathtt{last}(\alpha)$ denotes the last state in the sequence α.

We have that the action $\mathtt{SEQ}(a_1,\ a_2)$ first performs a_1. If the observation from a_1 is a failure message then that message is returned. Otherwise the action a_2 is performed and if that fails a failure message is returned noting that a_1 succeeded before the failure occurred. The expression $\lceil a \rceil$ is intended to denote an expression whose value is a representation of the action term a. For example, if a is $\mathtt{CLEAR}(y)$ then $\lceil a \rceil$ is $\mathtt{CLEAR}[y]$.

Given a POSDP $\mathcal{M}$, it is possible to formally assign a meaning to action expressions so that each such expression denotes a stochastic function which takes a state and stochastically produces a pair of a sequence of states and an observation. When $\mathcal{Q}$ and $\mathcal{M}$ are clear from context we write $P(\langle s,\ a,\ \rho\rangle \rightarrow \langle\alpha,\ w\rangle)$ to denote the probability that executing action a in state s under binding environment ρ results in the sequence of states α and the observation w. These probabilities can be formally defined in a manner similar to the definition of $P(\langle e,\ \rho\rangle \rightarrow v)$ in section 2. These probabilities are rigorously defined as the least fixed point of the equations in figure 2.

4 Defining POSDPs with Stochastic Programs

In this section we give a method of constructing POSDPs. We define a stochastic program POSDP to consist of a stochastic program as defined in section 2 and a set of non-constructor function symbols defined in $\mathcal{P}$ which we identify with the primitive action functions. In the program $\mathcal{P}$ a primitive action function is implemented as a function taking a state plus an argument. More precisely, the semantics of a stochastic program POSDP $\mathcal{M}$ is defined by the following equation where v must be a value.

$$P(\langle s_0,\ h\{v\}\rangle \rightarrow \langle\langle s_1,\ \ldots,\ s_m\rangle,\ w\rangle \mid \mathcal{M})$$

$$= P(\langle h(\texttt{CONS}(s_0,\ v)),\ \emptyset\rangle \rightarrow \texttt{PAIR}[\texttt{LIST}[s_1,\ \ldots,\ s_m],\ w])$$

Note that it is possible that the sum over all α and w of $P(\langle s_0,\ h\{v\}\rangle \rightarrow \langle \alpha,\ w\rangle \mid \mathcal{M})$ is strictly less than one. A stochastic program POSDP will be consistent if for all s_0 and $h\{v\}$ this sum equals one. Technically, only consistent stochastic program POSDPs are actually POSDPs.

It is interesting to note that the McCarthy frame problem does not arise in this approach to constructing world models. For example, a natural representation of a state of the world is a list of assertions — a list of expressions each of which intuitively represents some aspect of the world. A blocks world state might include assertions such as $\texttt{ON}[\texttt{A},\ B]$ and $\texttt{COLOR}[\texttt{A},\ \texttt{GREEN}]$. We can implement a primitive action MOVE-FROM-TO so that if s contains $\texttt{CLEAR}[x]$, $\texttt{ON}[x,\ y]$ and $\texttt{CLEAR}[z]$, then $\texttt{MOVE-FROM-TO}(s,\ x,\ y,\ z)$ returns $\texttt{PAIR}[\texttt{LIST}[u],\ \texttt{SUCCESS}]$ where u is the state resulting from removing the assertions $\texttt{ON}[x,\ y]$ and $\texttt{CLEAR}[z]$ and adding the assertions $\texttt{CLEAR}[y]$ and $\texttt{ON}[x,\ z]$. We can also easily arrange that if the required conditions on the input state are not met then $\texttt{MOVE-FROM-TO}(s,\ x,\ y,\ z)$ returns $\texttt{PAIR}[\texttt{NIL},\ \texttt{FAIL}]$. Note that this action will automatically not affect assertions about colors — there is no need to list the properties unaffected by the action.

5 Policies for POSDPs

It is standard in the MDP literature to call a robot control program a "policy". Here we define a policy (control program) for a POSDP $\mathcal{M}$ to be a pair $\langle \mathcal{Q},\ g\rangle$ where $\mathcal{Q}$ is an action program as defined above and g is a defined action function satisfying the condition that for any value v we have, that with probability 1, the action $g(v)$ takes time greater than zero and terminates without error. We run a policy g as follows. Let s_0 and w_0 be the constants START-STATE and START-OBS respectively. Then stochastically compute state s_{i+1} and observation w_{i+1} by running the action $g(w_i)$ in state s_i. Note that since observations are arbitrary values, this definition allows the observation w_i to be the entire history of primitive actions and their observations. If $\mathcal{M}$ is a finite MDP then w_i might be a single symbol naming the current state. In an elevator controller or a robosoccer controller the action $g(w)$ might rely entirely on sensing and ignore the observation w.

Since the initial state and initial observation are fixed by convention, a given policy and for a given POSDP determines a probability distribution over infinite sequences of states. We can evaluate the utility of a policy by introducing a reward function on states. A reward function maps each state to a real number called the reward of that state. Intuitively, the reward function expresses the goal of the robot — it should behave so as to maximize reward. This papers considers two notions of utility — discounted reward and average reward. The discounted reward is defined as follows where the expectation is taken over the stochastic generation of the infintie state sequence.

$$V \equiv E \sum_{i=1}^{\infty} r(s_i)\gamma^i$$

The average reward is defined as follows where again the expectation is over the stochastic generation of the infinite state sequence.

$$\rho \equiv E \lim_{k \to \infty} \frac{1}{k} \sum_{i=0}^{k} r(s_i)$$

In general there can be a nonzero probability over the choice of the infinite sequence that this limit does not exist (even for bounded reward functions). However, if the set of states is finite and the set of observations passed between runs of the policy h is also finite, then with probability 1 over the generation of the sequence the limit exists. We are now interested in the problem of computing V and ρ as defined by the these equations when given a policy defined by a stochastic action program to be run in a given stochastic program POSDP.

6 Computing Value Distributions for Program Expressions

Our ultimate objective is to compute a utility for a given policy running in a given stochastic program POSDP. As a first step this section gives a method for computing value distributions for stochastic programs. This algorithm will be used in later sections to compute probabilities of the form $P(\langle s,\ g\{v\}\rangle \to \langle \delta,\ w\rangle)$ where $g\{v\}$ is a primitive action in a stochastic program POSDP. In this section, however, we consider arbitrary stochastic programs as defined in section 2.

The semantic equation in figure 1 for procedure calls and `CASE` expressions include infinite summations — the equation for $f(e)$ includes a summation over all possible values of e and the equation for `CASE` e_0 `OF` $p_1 : e_1,\ \ldots,\ p_n : e_n$ contains a summation over all possible values of e_0. Because of these infinite summations, the equations in figure 1 can not be used directly to compute probabilities. This section gives a method which can, in many cases, replace the infinite summations in figure 1 with finite summations which can then be used to effectively compute probability distributions.

We start by defining the computation graph of a pair $\langle e,\ \rho\rangle$ where e is a program expression and ρ is a binding environment. The computation graph of

$$\frac{Eval(\langle x,\ \rho\rangle)}{\langle x,\ \rho\rangle \to \rho(x)}$$

$$\frac{Eval(\langle A,\ \rho\rangle)}{\langle A,\ \rho\rangle \to A}$$

$$\frac{Eval(\langle \mathtt{CONS}[e_1,,\ e_2],\ \rho\rangle)}{Eval(\langle e_1,\ \rho\rangle),\ Eval(\langle e_2,\ \rho\rangle)}$$

$$\frac{\begin{array}{l} Eval(\langle \mathtt{CONS}[e_1,\ e_2],\ \rho\rangle) \\ \langle e_1,\ \rho\rangle \to v_1,\ \langle e_2,\ \rho\rangle \to v_2 \end{array}}{\begin{array}{l} \langle \mathtt{CONS}[e_1,,\ e_2],\ \rho\rangle \\ \quad \to \mathtt{CONS}[v_1,\ \ v_2] \end{array}}$$

$$\frac{\begin{array}{l} Eval(\alpha) \\ \alpha \Rightarrow \beta \\ \beta \to \gamma \end{array}}{\alpha \to \gamma}$$

$$\frac{Eval(\langle \mathtt{FLIP}(\alpha),\ \rho\rangle)}{\langle \mathtt{FLIP}(\alpha),\ \rho\rangle \to \mathtt{T},\ \langle \mathtt{FLIP}(\alpha),\ \rho\rangle \to \mathtt{F}}$$

$$\frac{Eval(\langle g(e),\ \rho\rangle)}{Eval(\langle e,\ \rho\rangle)}$$

$$\frac{\begin{array}{l} Eval(\langle g(e),\ \rho\rangle) \\ \langle e,\ \rho\rangle \to v \end{array}}{\begin{array}{l} Eval(\langle \mathtt{body}(g),\ [\mathtt{param}(g) \mapsto v]\rangle) \\ \langle g(e),\ \rho\rangle \Rightarrow \langle \mathtt{body}(g),\ [\mathtt{param}(g) \mapsto v]\rangle \end{array}}$$

$$\frac{Eval(\langle \mathtt{CASE}\ e_0\ \mathtt{OF}\ p_1 : e_1,\ \ldots,\ p_n : e_n,\ \rho\rangle)}{Eval(\langle e_0,\ \rho\rangle)}$$

$$\frac{\begin{array}{l} Eval(\langle \mathtt{CASE}\ e_0\ \mathtt{OF}\ p_1 : e_1,\ \ldots,\ p_n : e_n,\ \rho\rangle) \\ \langle e_0,\ \rho\rangle \to v \\ p_i\ \mathtt{is\ the\ first\ pattern\ matching}\ v \end{array}}{\begin{array}{l} Eval(\langle e_i,\ \mathtt{match}(p_i,\ v,\ \rho)\rangle) \\ \langle \mathtt{CASE}\ e_0\ \mathtt{OF}\ p_1 : e_1,\ \ldots,\ p_n : e_n,\ \rho\rangle \\ \quad \Rightarrow \langle e_i,\ \mathtt{match}(p_i,\ v,\ \rho)\rangle \end{array}}$$

Fig. 3. Computation graph inference rules. The graph of a pair $\langle e,\ \rho\rangle$ consists of all those assertions derivable using the above rules from the assertion $Eval(\langle e,\ \rho\rangle)$. Rules with multiple conclusions are an abbreviation for multiple rules, one for each conclusion.

$\langle e,\ \rho\rangle$ consists of all assertions derivable from the initial statement $Eval(\langle e,\ \rho\rangle)$ using the inference rules in figure 3. The graph contains all assertions of the form $Eval(\langle e',\ \rho'\rangle)$ such that there exists a nonzero probability that the evaluation of e under environment ρ will cause a recursive evaluation of $\langle e',\ \rho'\rangle$. For each assertion $Eval(\langle e',\ \rho'\rangle)$ in the graph, the graph also includes all assertions of the form $\langle e',\ \rho'\rangle \rightarrow v$ such that $P(\langle e',\ \rho'\rangle \rightarrow v) > 0$.

The inference rules in figure 3 can be implemented as a bottom-up logic program, i.e., the rules are run in a forward chaining manner starting from the initial assertion $Eval(\langle e,\ \rho\rangle)$. This forward chaining deduction process terminates if and only if the number of derivable assertions is finite. So, by definition, this inference process terminates if and only if e has a finite computation graph. Intuitively, e has a finite computation graph if computing the value of e leads to the evaluation of only a finite set of pairs of the form $\langle e',\ \rho'\rangle$ and each such pair encountered in the computation has only a finite set of values with nonzero probability.

As an example, suppose that we have defined a function `NEXT-STATE` such that for any value v we have that `NEXT-STATE`(v) stochastically computes one of a finite set of constants representing a finite set of states. Now suppose we define the following program.

$$\begin{array}{lll} \texttt{FINAL-STATE}(s) \equiv \texttt{CASE}\ s\ \texttt{OF} & A : A \\ & B : B \\ & z : \texttt{FINAL-STATE}(\texttt{NEXT-STATE}(s)) \end{array}$$

Now suppose we take the top level assertion to be `FINAL-STATE(C)`. If the function `NEXT-STATE` is such that the constants `A` and `B` are (transitively) reachable from any state then the procedure `FINAL-STATE` terminates with probability 1. Furthermore, it has only two possible values — the constants `A` and `B`. We wish to compute the relative probabilities of these two possible outcomes. Assuming that calls to `NEXT-STATE` produce finite computation graphs, calls to `FINAL-STATE` also produce finite computation graphs. The graph consists, essentially, of assertions of the form `NEXT-STATE`$(s_i) \rightarrow s_j$ where s_i and s_j are reachable states plus assertions of the form `FINAL-STATE`$(s) \rightarrow$ `A` and `FINAL-STATE`$(s) \rightarrow$ `B`.

Now consider an arbitrary pair $\langle e,\ \rho\rangle$ with a finite computation graph. To compute the probability distribution over the values $\langle e,\ \rho\rangle$ one computes a probability for each edge $\langle e',\ \rho'\rangle \rightarrow v$ in the computation graph. This can be done by iterating the equations in figure 1. The summations in the equations for procedure call and case statements can be restricted to the finite set of terms where all the arrow expressions involved are contained in the computation graph. Note that the equations must be iterated, starting from a value of zero for all probabilities, until a numerical fixed point is reached.

7 The Computation Graph of a Policy

The first step in computing a utility for a given policy running in a given stochastic program POSDP is to compute an appropriate computation graph. The com-

$$\frac{Eval(\langle s,\ x,\ \rho\rangle)}{\langle s,\ x,\ \rho\rangle \to \langle s,\ \rho(x)\rangle}$$

$$\frac{Eval(\langle s,\ A,\ \rho\rangle)}{\langle s,\ A,\ \rho\rangle \to \langle s,\ A\rangle}$$

$$\frac{Eval(\langle s,\ \mathtt{CONS}[a_1,\ a_2],\ \rho\rangle)}{Eval(\langle s,\ a_1,\ \rho\rangle)}$$

$$\frac{\begin{array}{l} Eval(\langle s,\ \mathtt{CONS}[a_1,\ a_2],\ \rho\rangle) \\ \langle s,\ a_1, \rho\rangle \to \langle s',\ v_1\rangle \end{array}}{Eval(\langle s',\ a_2,\ \rho\rangle)}$$

$$\frac{\begin{array}{l} Eval(\langle s,\ \mathtt{CONS}[a_1,\ a_2],\ \rho\rangle) \\ \langle s,\ a_1, \rho\rangle \to \langle s',\ v_1\rangle \\ \langle s',\ a_2, \rho\rangle \to \langle s'',\ v_2\rangle \end{array}}{\begin{array}{l} \langle s,\ \mathtt{CONS}[a_1,\ a_2],\ \rho\rangle \\ \quad \to \langle s'',\ \mathtt{CONS}[v_1,\ v_2]\rangle \end{array}}$$

$$\frac{Eval(\langle s,\ \mathtt{FLIP}(\alpha),\ \rho\rangle)}{\begin{array}{l} \langle s,\ \mathtt{FLIP}(\alpha),\ \rho\rangle \to \langle s,\ \mathtt{T}\rangle \\ \langle s,\ \mathtt{FLIP}(\alpha),\ \rho\rangle \to \langle s,\ \mathtt{F}\rangle \end{array}}$$

$$\frac{Eval(\langle s,\ g(a),\ \rho\rangle)}{Eval(\langle s,\ a,\ \ \rho\rangle)}$$

$$\frac{\begin{array}{l} Eval(\langle s,\ g(a),\ \rho\rangle) \\ \langle s,\ a,\ \rho\rangle \to \langle s',\ v\rangle \\ g(y) \equiv b \end{array}}{\begin{array}{l} Eval(\langle b,\ [y \mapsto v]\rangle) \\ \langle s,\ g(a),\ \rho\rangle \Rightarrow \langle s',\ b,\ [y \mapsto v]\rangle \end{array}}$$

$$\frac{Eval(\langle s,\ \mathtt{CASE}\ a_0\ \mathtt{OF}\ p_1 : a_1,\ \ldots,\ p_n : a_n,\ \rho\rangle)}{Eval(\langle s,\ a_0,\ \rho\rangle)}$$

$$\frac{\begin{array}{l} Eval(\langle s,\ \mathtt{CASE}\ a_0\ \mathtt{OF}\ p_1 : a_1,\ \ldots,\ p_n : a_n,\ \rho\rangle) \\ \langle s,\ a_0,\ \rho\rangle \to \langle s',\ v\rangle \\ p_i\ \mathtt{is\ the\ first\ patterm\ matching}\ v \end{array}}{\begin{array}{l} Eval(\langle s',\ a_i,\ \mathtt{match}(p_i,\ v,\ \rho)\rangle) \\ \langle s,\ \mathtt{CASE}\ a_0\ \mathtt{OF}\ p_1 : a_1,\ \ldots,\ p_n : a_n,\ \rho\rangle \\ \quad \Rightarrow \langle s',\ a_i,\ \mathtt{match}(p_i,\ v,\ \rho)\rangle \end{array}}$$

Fig. 4. Computation graph inference rules for action expressions. The graph of a triple $\langle s,\ q,\ \rho\rangle$ consists of all those assertions derivable from the assertion $Eval(\langle s,\ a,\ \rho\rangle)$ using rules in figures 3, 4 and 5.

$$\frac{Eval(\langle s,\ h\{a\},\ \rho\rangle)}{Eval(\langle s,\ a,\ \rho\rangle)} \qquad \frac{Eval(\texttt{primapp}(h,\ v,\ s))}{Eval(\langle h(\texttt{CONS}(s,\ v)),\ \emptyset\rangle)}$$

$$\frac{\begin{array}{l} Eval(\langle s,\ h\{a\},\ \rho\rangle) \\ \langle s,\ a,\ \rho\rangle \to \langle s',\ v\rangle \end{array}}{\begin{array}{l} Eval(\texttt{primapp}(h,\ v,\ s)) \\ \langle s,\ h\{a\},\ \rho\rangle \\ \quad \Rightarrow \texttt{primapp}(h,\ v,\ s) \end{array}} \qquad \frac{\begin{array}{l} Eval(\texttt{primapp}(h,\ v,\ s)) \\ \langle h(\texttt{CONS}(s,\ v)),\ \emptyset\rangle \to \texttt{PAIR}(\alpha,\ w) \end{array}}{\texttt{primapp}(h,\ v,\ s) \to \langle \texttt{last}(s\alpha),\ w\rangle}$$

$$\frac{\begin{array}{l} \texttt{policy}(g) \\ Eval(\langle s,\ g\{w\},\ \emptyset\rangle) \\ \langle s,\ g\{w\},\ \emptyset\rangle \to \langle s',\ w'\rangle \end{array}}{Eval(\langle s',\ g(w'),\ \emptyset\rangle)}$$

Fig. 5. Computation graph inference rules for primitive applications and policies.

putation graph will include assertions describing the evaluation of action expressions as well as program expressions. Action expression assertions have the form $Eval(\langle s,\ a,\ \rho\rangle)$ and $\langle s,\ a,\ \rho\rangle \to \langle s',\ w\rangle$ where s and s' are states, a is an action expression, ρ is a binding environment and w is a value. Note that, unlike the arrow expressions in figure 2, these arrow expressions produce pairs $\langle s',\ w\rangle$ where s' is a state rather than a sequence of states. The computation graph of a given policy g consists of all assertions derivable from the intial assertions $\texttt{poligy}(g)$ and $Eval(\texttt{START-STATE},\ g(\texttt{START-OBS}),\ \emptyset)$ using the inference rules in figures 3, 4, and 5. The graph includes all assertions of the form $Eval(\langle s,\ a,\ \rho\rangle)$ such that the running poligy g from the initial state and initial observation leads to the evaluation of $\langle s,\ a,\ \rho\rangle$. For each such triple in the graph the graph also includes all arcs of the form $\langle s,\ a,\ \rho\rangle \to \langle s',\ w\rangle$ such that there exists a state sequence σ with $\mathrm{P}(\langle s,\ a,\ \rho\rangle \to \langle \sigma,\ w\rangle) > 0$ and $\texttt{last}(s\sigma) = s'$.

8 Computing Discounted Reward

This section gives an algorithm for computing the discounted reward of a given policy running in a given stochastic program POSDP. The algorithm requires that the computation graph of the given policy and POSDP is finite and that any single iteration of the policy takes at least one time step and terminates without error with probability 1. Given a policy g, observation w, and a state s, let $V^{g(w)}(s)$ be the discounted reward running policy g starting in state s

$$R_\gamma(\langle s,\ x,\ \rho\rangle) = 0$$

$$R_\gamma(\langle s,\ \mathtt{FLIP}(\alpha),\ \rho\rangle) = 0$$

$$R_\gamma(\langle s,\ A,\ \rho\rangle) = 0$$

$$R_\gamma(\langle s,\ \mathtt{CONS}[a_1,\ a_2],\ \rho\rangle) = R_\gamma(\langle s,\ a_1,\ \rho\rangle) + \sum_{v,\ s'} M_\gamma(\langle s,\ a_1,\ \rho\rangle \to \langle s',\ v\rangle)\ R_\gamma(\langle s',\ a_2,\ \rho\rangle)$$

$$R_\gamma(\langle s,\ h\{a\},\ \rho\rangle) = R_\gamma(\langle s,\ a,\ \rho\rangle) + \sum_{v,\ s'} M_\gamma(\langle s,\ a,\ \rho\rangle \to \langle s',\ v\rangle)\ R_\gamma(\mathtt{primapp}(h,\ v,\ s'))$$

$$R_\gamma(\mathtt{primapp}(h,\ v,\ s)) = \sum_{\sigma, w} P(\langle h(\mathtt{CONS}(s,\ v)),\ \emptyset\rangle \to \mathtt{PAIR}(\sigma,\ w)) r_\gamma(\sigma)$$

$$R_\gamma(\langle s,\ g(a),\ \rho\rangle) = R_\gamma(\langle s,\ a,\ \rho\rangle) + \sum_{v, s'} M_\gamma(\langle s,\ a,\ \rho\rangle \to \langle s',\ v\rangle)\ R_\gamma(\langle s',\ \mathtt{body}(g),\ [\mathtt{param}(g) \mapsto v]\rangle)$$

$$R_\gamma(\langle s,\ \mathtt{CASE}\ a_0\ \ \mathtt{OF}\ \ p_1 : a_1,\ \ldots,\ p_n : a_n,\ \rho\rangle) = R_\gamma(\langle s,\ a_0,\ \rho\rangle)$$

$$+ \sum_{v \text{ matching some } p_i,\ s'} \begin{array}{l} M_\gamma(\langle s,\ a_0,\ \rho\rangle \to \langle s',\ v\rangle)\ R_\gamma(\langle s',\ a_i,\ \mathtt{match}(p_i,\ v,\ \rho)\rangle) \\ \text{where } p_i \text{ is the first pattern matching } v \end{array}$$

Fig. 6. Bellman Equations for computing $R_\gamma(\langle s,\ a,\ \rho\rangle)$

and with initial observation w. We are interested in computing the quantity $V = V^{g(\mathtt{START\text{-}OBS})}(\mathtt{START\text{-}STATE})$. Assuming any single iteration takes at least one time step, and terminates without error with probability 1, we have the following where σ ranges over sequences of states and where $r_\gamma(\langle s_1,\ \ldots,\ s_m\rangle)$ denotes $\sum_{i=1}^{m} r(s_i)\gamma^i$.

$$V^{g(w)}(s) = \sum_{\sigma,\ w'} P(\langle s,\ g(w),\ \emptyset\rangle \to \langle \sigma,\ w'\rangle) \left[r_\gamma(\sigma) + \gamma^{|\sigma|} V^{g(w')}(\mathtt{last}(s\sigma)) \right]$$

This equation can be rewritten as follows.

$$\begin{aligned} V^{g(w)}(s) = & \sum_{\sigma,\ w'} P(\langle s,\ g(w),\ \emptyset\rangle \to \langle \sigma,\ w'\rangle) r_\gamma(\sigma) \\ & + \sum_{s', w'} \left[\sum_{\mathtt{last}(s\sigma) = s'} P(\langle s,\ g(w),\ \emptyset\rangle \to \langle \sigma,\ w'\rangle)\gamma^{|\sigma|} \right] V^{g(w')}(s') \end{aligned}$$

$$M_\gamma(\langle s,\ x,\ \rho\rangle \to \langle s,\ \rho(x)\rangle) = 1$$

$$M_\gamma(\langle s,\ \texttt{FLIP}(\alpha),\ \rho\rangle \to \langle s,\ T\rangle) = \alpha$$

$$M_\gamma(\langle s,\ \texttt{FLIP}(\alpha),\ \rho\rangle \to \langle s,\ F\rangle) = 1 - \alpha$$

$$M_\gamma(\langle s,\ A,\ \rho\rangle \to \langle s,\ A\rangle) = 1$$

$$M_\gamma(\langle s,\ \texttt{CONS}[a_1,\ a_2],\ \rho\rangle \to \langle s',\ \texttt{CONS}[v_1,\ v_2]\rangle) =$$

$$\sum_{s''} M_\gamma(\langle s,\ a_1,\ \rho\rangle \to \langle s'',\ v_1\rangle)\ M_\gamma(\langle s'',\ a_2,\ \rho\rangle \to \langle s',\ v_2\rangle)$$

$$M_\gamma(\langle s,\ h\{a\},\ \rho\rangle \to \langle s',\ w\rangle) =$$

$$\sum_{v,\ s''} M_\gamma(\langle s,\ a,\ \rho\rangle \to \langle s'',\ v\rangle)\ M_\gamma(\texttt{primapp}(h,\ v,\ s'') \to \langle s',\ w\rangle)$$

$$M_\gamma(\texttt{primapp}(h,\ v,\ s) \to \langle s',\ w\rangle) = \sum_{\texttt{last}(s\sigma)=s'} P(\langle h(\texttt{CONS}(s,\ v)),\ \emptyset\rangle \to \langle \sigma,\ w\rangle)\gamma^{|\sigma|}$$

$$M_\gamma(\langle s,\ g(a),\ \rho\rangle \to \langle s',\ w\rangle) =$$

$$\sum_{v, s''} M_\gamma(\langle s,\ a,\ \rho\rangle \to \langle s'',\ v\rangle)\ M_\gamma(\langle s'',\ \texttt{body}(g),\ [\texttt{param}(g) \mapsto v]\rangle \to \langle s',\ w\rangle)$$

$$M_\gamma(\langle s,\ \texttt{CASE}\ a_0\ \texttt{OF}\ p_1 : a_1,\ \ldots,\ p_n : a_n,\ \rho\rangle \to \langle s',\ w\rangle) =$$

$$\sum_{v \text{ matching some } p_i, s''} \begin{array}{l} M_\gamma(\langle s,\ a_0,\ \rho\rangle \to \langle s'',\ v\rangle)\ M_\gamma(\langle s'',\ a_i,\ \texttt{match}(p_i,\ v,\ \rho)\rangle \to \langle s',\ w\rangle) \\ \text{where } p_i \text{ is the first pattern matching } v \end{array}$$

Fig. 7. Bellman Equations for computing $M_\gamma(\langle s,\ a,\ \rho\rangle \to \langle s',\ w\rangle)$.

We now define the quantities $R_\gamma(\langle s,\ a,\ \rho\rangle)$ and $M_\gamma(\langle s,\ a,\ \rho\rangle \rightarrow \langle s',\ w\rangle)$ as follows.

$$R_\gamma(\langle s,\ a,\ \rho\rangle) \equiv \sum_{\sigma,\ w} P(\langle s,\ a,\ \rho\rangle \rightarrow \langle \sigma,\ w\rangle)\ r_\gamma(\sigma)$$

$$M_\gamma(\langle s,\ a,\ \rho\rangle \rightarrow \langle s',\ w\rangle) \equiv \sum_{\mathtt{last}(s\sigma)=s'} P(\langle s,\ a,\ \rho\rangle \rightarrow \langle \sigma,\ w\rangle)\ \gamma^{|\sigma|}$$

Using these definitions we have the following.

$$V^{h(w)}(s) = R_\gamma(\langle s,\ h(w),\ \emptyset\rangle) + \sum_{s',\ w'} M_\gamma(\langle s,\ h(w),\ \emptyset\rangle \rightarrow \langle s',\ w'\rangle) V^{h(w')}(s')$$

This is essentially the Bellman equation for policies of options developed by Sutton, Precup and Singh. Given values for $R_\gamma(\langle s,\ a,\ \rho\rangle)$ and $M_\gamma(\langle a,\ a,\ \rho\rangle \rightarrow \langle s',\ w\rangle)$ this Bellman equation can be iterated to yield values for $V^{g(w)}(s)$ and hence a value for V. Values for $R_\gamma(\langle s,\ a,\ \rho\rangle)$ and $M_\gamma(\langle a,\ a,\ \rho\rangle \rightarrow \langle s',\ w\rangle)$ for all triples $\langle s,\ a,\ \rho\rangle$ and arcs $\langle s,\ a,\ \rho\rangle \rightarrow \langle s',\ w\rangle$ in the computation graph of the policy g can be computed by iterating the Bellman equations in figures 6 and 7. In iterating these quations the computation graph can be used to limit summations to terms where all arcs in the body of the summation are present in the comptuation graph. All values are initialized to zero.

9 Computing Average Reward

This section gives an algorithm for computing average reward. The algorithm requires that the computation graph be finite and that each iteration of the policy takes at least one time step, terminates without error with probability 1, and has finite expected running time. A program can terminate with probability 1 but still have infinte expected running time. In particular, if `WAIT{}` always takes exactly one time step then the following program terminates with probability 1 but has infinite expected running time.

$$\mathtt{WEIRD}() \equiv \mathtt{IF}(\mathtt{FLIP}(1/2),\ \mathtt{WAIT}\{\},\ \mathtt{SEQ}(\mathtt{WEIRD}(),\ \mathtt{WEIRD}())))$$

For the undiscounted case we will need $R_1(\langle s,\ a,\ \rho\rangle)$ and $M_1(\langle s,\ a,\ \rho\rangle \rightarrow \langle s',\ w\rangle)$, i.e., the undiscounted rewards and transition masses for actions in the computation graph. These quantities can be computed using the Bellman equations in figures 6 and 7 with $\gamma = 1$. Note that $M_1(\langle s,\ a,\ \rho\rangle \rightarrow \langle s',\ w\rangle)$ is just the probability that $\langle s,\ a,\ \rho\rangle$ produces the pair $\langle s',\ w\rangle$. The requirement that the computation graph be finite and that each iteration of the policy has finite expected running time guarantees that the quantities $R_1(\langle s,\ a,\ \rho\rangle)$ are finite. We will also need the expected running time of each action in the computation graph, i.e., the quantities $T(\langle s,\ a,\ \rho\rangle)$ defined as follows.

$$T(\langle s,\ a,\ \rho\rangle) \equiv \sum_{\sigma,\ w} P(\langle s,\ a,\ \rho\rangle \rightarrow \langle \sigma,\ w\rangle)\ |\sigma|$$

The quantities $T(\langle s,\ a,\ \rho\rangle)$ can be computed using Bellman equations identical to those in figure 6 with R_γ replaced by T, M_γ replaced by M_1, and with the equation for `primapp` replaced by the following.

$$T(\texttt{primapp}(h,\ v,\ s)) = \sum_{\sigma,w} P(\langle h(\texttt{CONS}(s,\ v)),\ \emptyset\rangle \rightarrow \texttt{PAIR}(\sigma,\ w))\ |\sigma|$$

Each iteration of the policy starts with a state-observation pair $\langle s,\ w\rangle$ and generates a next state-observation pair $\langle s',\ w'\rangle$. I will say that a pair $\langle s,\ w\rangle$ is reachable if the computation graph contains $Eval(\langle s,\ g(w),\ \emptyset\rangle)$. Let N be the transition matrix on reachable pairs where the probability of the transition from $\langle s,\ v\rangle$ to $\langle s',\ v'\rangle$, denoted $N(\langle s,\ w\rangle \rightarrow \langle s',\ w'\rangle)$, is $M_1(\langle s,\ g(w),\ \emptyset\rangle \rightarrow \langle s',\ w'\rangle)$. Similarly, let $T(\langle s,\ w\rangle \rightarrow \langle s',\ w'\rangle)$ and $R_1(\langle s,\ w\rangle \rightarrow \langle s',\ w'\rangle)$ abbreviate $T(\langle s,\ g(w),\ \emptyset\rangle \rightarrow \langle s',\ w'\rangle)$ and $R_1(\langle s,\ g(w),\ \emptyset\rangle \rightarrow \langle s',\ w'\rangle)$ repsectively.

We first consider the case where the transition matrix N is ergodic, i.e., has a unique stationary distribution D over the pairs $\langle s,\ w\rangle$. Let $\overline{T}$ be the average time per transition, i.e., the quantitny

$$\overline{T} \equiv \sum_{\langle s,\ w\rangle} D(\langle s,\ w\rangle) \sum_{\langle s',\ w'\rangle} N(\langle s,\ w\rangle \rightarrow \langle s',\ w'\rangle) T(\langle s,\ w\rangle \rightarrow\ \langle s',\ w'\rangle).$$

Similarly, let $\overline{R}$ be the average transition reward, i.e., the quantity

$$\overline{R} \equiv \sum_{\langle s,\ w\rangle} D(\langle s,\ w\rangle) \sum_{\langle s',\ w'\rangle} N(\langle s,\ w\rangle \rightarrow \langle s',\ w'\rangle) R_1(\langle s,\ w\rangle \rightarrow\ \langle s',\ w'\rangle).$$

The average reward ρ is now just $\overline{R}/\overline{T}$. To see this consider sampling an infinite run of the policy. For any finite number k let $R(k)$ be sum of the rewards up to time k and let $n(k)$ be the number of top level iterations of the policy up to time k. We now have the following.

$$\rho = \lim_{k\rightarrow\infty} \frac{R(k)}{k} = \lim_{k\rightarrow\infty} \frac{R(k)/n(k)}{k/n(k)} = \frac{\lim_{k\rightarrow\infty} R(k)/n(k)}{\lim_{k\rightarrow\infty} k/n(k)} = \overline{R}/\overline{T}$$

Finally we consider the case where the matrix N is non-ergodic. We say that $\langle s',\ v'\rangle$ is reachable from $\langle s,\ v\rangle$ if there is a (possibly empty) path of transitions in N with nonzero probability leading from $\langle s,\ v\rangle$ to $\langle s',\ v'\rangle$. We say that two pairs are inter-reachable if each is reachable from the other. Inter-reachability is an equivalnce relation and we call the equivalence classes of this relation the strongly connected components (SCCs). An SCC will be called terminal if all pairs reachable from elements of that component are also in that component. With probability 1 over the choice of an infinite run we have that the pairs eventually enter a terminal SCC. Let $k \geq 1$ be the number of terminal SCCs. Let N_i be the restriction of the transition matrix N to the ith terminal SCC. Let P_i be the probability that an infinite sequence of pairs eventually enters the ith terminal SCC and let D_i be the unique stationary distribution for that SCC, i.e., the stationary distribution of N_i. Let $\overline{R_i}$ and $\overline{T_i}$ be defined as above but for stationary distribution D_i of the transition matrix N_i. We now have that the average reward ρ is the following.

$$\rho = \sum_i P_i \frac{\overline{R_i}}{\overline{T_i}}$$

10 Conclusions

This paper gives a model-checking style algorithm for verifying the utility of a hand-written robot controller. Both the controller and the environment model are assumed to be written in a Turing-complete programming language supporting complex data structures, stochastic nondeterminism, conditional expressions and recursive procedures. There are several possible areas for further work. It seems likely that, as with other areas of model checking, a variety of heuristic techniques can be developed to improve the efficiency and scope of algorithms for verifying decision-theoretic utility. It also seems likely that methods can be developed to optimize the biases of the stochastic `FLIP` expressions in a given stochastic controller. Finally, the Bellman equations for stochastic programs, which underly the verification algorithm, seem to be of independent interest and may have unforseen applications.

References

1. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking:10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, June 1990.
2. J.Y. Halpern and M. Y. Vardi. Model checking vs. theorem proving – a manifesto. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*, pages 151–176. Academic Press, 1991. An Earlier version appeared in KR-91.
3. Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
4. H.J. Levesque and R. Reiter. High-level robotic control: beyond planning. A position paper for AAAI 1998 Spring Symposium: Integrating Robotics Research: Taking the Next Big Leap. Available at http://www.cs.toronto.edu/cogrobo, 1998.
5. H.J. Levesque, R. Reiter, Y. Lesprance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.
6. David McAllester, Daphne Koller, and Avi Pfeffer. Effective bayesian inference for stochastic programs. In *AAAI-97*, 1997.
7. David McAllester and Satinder Singh. Approximate planning for factored pomdps using beleif state simplification. In *UAI-99*, 1999.
8. John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *machine intelligence 4*, pages 463–502. Edinburgh University Press, 1969.
9. Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic, July 1993.

10. C. H. Papadimitriou and J. N. Tsitsiklis. The complexity of markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.
11. R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *AIJ*, 1999. to appear.
12. Sebastian Thrun. A framework for programming embedded systems: Initial design and results. Technical Report CMU-CS-98-142, Carnegie Mellon University, Pittsburgh, PA, 1998.