Stochastic Dynamic Programming with Factored Representations*

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

Richard Dearden
Department of Computer Science
University of British Columbia
Vancouver, BC, V6T 1Z4, CANADA
dearden@cs.ubc.ca

Moisés Goldszmidt
Computer Science Department
Stanford University
Stanford, CA 94305-9010, USA
moises@robotics.stanford.edu

Abstract

Markov decision processes (MDPs) have proven to be popular models for decision-theoretic planning, but standard dynamic programming algorithms for solving MDPs rely on explicit, state-based specifications and computations. To alleviate the combinatorial problems associated with such methods, we propose new representational and computational techniques for MDPs that exploit certain types of problem structure. We use dynamic Bayesian networks (with decision trees representing the local families of conditional probability distributions) to represent stochastic actions in an MDP, together with a decision-tree representation of rewards. Based on this representation, we develop versions of standard dynamic programming algorithms that directly manipulate decision-tree representations of policies and value functions. This generally obviates the need for state-by-state computation, aggregating states at the leaves of these trees and requiring computations only for each *aggregate* state. The key to these algorithms is a decision-theoretic generalization of classic regression analysis, in which we determine the features relevant to predicting expected value. We demonstrate the method empirically on several planning problems,

*Some parts of this report appeared in preliminary form in "Exploiting Structure in Policy Construction," *Proc. of Fourteenth International Joint Conf. on Artificial Intelligence (IJCAI-95)*, Montreal, pp.1550–1556 (1995); and "Correlated Action Effects in Decision-Theoretic Regression," *Proc. of Thirteenth Conf. on Uncertainty in Artificial Intelligence (UAI-97)*, Providence, pp.30–37 (1997).

†Communicating author

showing significant savings for certain types of domains. We also identify certain classes of problems for which this technique fails to perform well and suggest extensions and related ideas that may prove useful in such circumstances. We also briefly describe an approximation scheme based on this approach.

Keywords: decision-theoretic planning, Markov decision processes, Bayesian networks, regression, decision trees, abstraction

1 Introduction

Decision-theoretic planning (DTP) has attracted a considerable amount of attention recently as AI researchers seek to generalize the types of planning problems that can be tackled in computationally effective ways. DTP is primarily concerned with problems of sequential decision making under conditions of uncertainty and where there exist multiple, often conflicting, objectives whose desirability can be quantified. *Markov decision processes* (MDPs) have been adopted as the model of choice for DTP problems in much recent work [12, 26, 28, 30, 61, 78], and have also provided the underlying foundations for most work in reinforcement learning [48, 76, 77, 84]. MDPs allow the introduction of uncertainty into the effects of actions, the modeling of uncertain exogenous events, the presence of multiple, prioritized objectives, and the solution of nonterminating process-oriented problems.[1]

The foundations and the basic computational techniques for MDPs [3, 5, 44, 62] are well-understood and in certain cases can be used directly in DTP. These methods exploit the dynamic programming principle and allow MDPs to be solved in time polynomial in the size of the state and action spaces that make up the planning problem. Unfortunately, these classical dynamic programming methods are formulated so as to require explicit state space enumeration. As such, AI planning systems that solve MDPs are faced with Bellman's so-called *curse of dimensionality*: the number of states grows exponentially with the number of variables that characterize the planning domain. This has an impact on the feasibility of both the specification and solution of large MDPs.

The curse of dimensionality plagues not only DTP, but also classical planning techniques. However, methods have been developed that, in many instances, circumvent this problem. In classical planning one typically does not specify actions and goals explicitly using the underlying state space, but rather "intensionally" using *propositional* or *variable-based* representations. For instance, a STRIPS representation of an action describes very concisely the transitions induced by that action over a large number of states. Similarly, classical planning techniques such as regression planning [83] or nonlinear planning [22, 54, 58, 66] exploit these representations to great effect, never requiring that one search (or implement "shortest-path" dynamic programming techniques) explicitly through state space. Intuitively, such methods aggregate states that behave identically under a given action sequence with respect to a given goal.

[1] One form of uncertainty cannot be handled in the framework we adopt, specifically, *partial observability*, or uncertain knowledge about the state of the system being controlled. *Partially observable MDPs* (or POMDPs) [52, 53, 75, 73] can be used in such cases. We will make further remarks on POMDPs at the end of this article.

In this paper, we develop similar techniques for solving certain classes of large MDPs. We first describe a representation for actions with stochastic effects that uses Bayesian networks (and decision trees to represent the required families of conditional probability distributions) to provide the same type of compact representation of actions that, say, STRIPS affords in deterministic settings. We also use decision trees to represent reward functions. This representation lays bare — indeed, exploits — certain structural regularities in transition probabilities and reward functions. We then describe algorithms that use this representation to compute value functions and solve MDPs without generally requiring explicit enumeration of the state space. Much like regression in classical planning, we focus attention on the variables that, under a particular action, influence the outcome of this action with respect to *relevant* variables. In addition, policies and value functions will be represented compactly using decision trees, with the structure inherent in the policy or value function being preserved to a large extent by our algorithmic operations. Indeed, under certain assumptions one can show that the degree of preserved structure is maximal (e.g., subject to variable reordering in trees).

1.1 Decision-Theoretic Regression

The key to each of our algorithms is a process we call *decision-theoretic regression*. In classical planning the *regression* of a set of conditions C through an action a is the weakest set of conditions *regr*(C, a) such that performing action a under conditions *regr*(C, a) ensures that C is made true [83].[2] This is the key step in any *backchaining* (or subgoaling) planner, including least-commitment planners [54, 58]. Given a (sub)goal set G, regression of G through a produces a new subgoal whose achievement with plan P' assures us of a plan P to achieve G: simply append a to P' to form P.

Decision-theoretic regression generalizes this process in two ways. First, we can't always speak of goal achievement in MDPs; rather, we concern ourselves with the *value* associated with certain conditions. Thus, we regress a *set of conditions*, each associated with a distinct value, through an action. As such, the decision theoretic regression of such a set of conditions through an action will result in a new set of conditions. Second, stochastic actions rarely guarantee achievement of any particular condition—so rather than producing the conditions that, when the action is applied, lead to a specific "target" condition, we instead produce a set of conditions under which the action will make each of the regressed conditions true with identical probability. It follows that, since each condition in the regressed set is associated with a single value, the new conditions produced by decision-theoretic regression each have the same *expected value* under action a.

With such an operation in hand, we can implement classical algorithms for solving MDPs, such as *value iteration* [3] or *modified policy iteration* [63] in a highly structured way. Our structured versions of these algorithms will cluster together states that at each stage in the computation have the same estimated value or same optimal choice of action. This partitioning of state space into such regions will be represented by decision trees that test the values of specific variables. The computational advantage provided by such an

[2] Regression is also a concept of fundamental importance in program synthesis and verification [24, 34].

approach is that value need only be computed once for each region instead of once per state.

1.2 State Aggregation and Function Approximation

The approach we take to solving large MDPs is a specific *state aggregation* method. Other types of state aggregation techniques have been proposed, in which states with similar characteristics are grouped together. Such methods are reported in, for instance, [4, 68, 81], and can vary as to whether states are *statically* or *dynamically* aggregated (that is, do the groupings of states stay fixed or can they change during computation). Other compact representations of value functions have also been proposed, such as linear function representations or neural networks [1, 6, 80, 81]. These techniques do not seek to exploit regions of uniformity in value functions, but rather compact functions of state features that reflect value. As such they are distinguished from strict aggregation methods.

In much of this previous work, the goal is the approximate solution of large MDPs. Our proposal can be distinguished from other aggregation methods, and other compact representations of values functions, in two major ways. First, our aggregations are determined dynamically using features that are easily extracted from the model. In this sense, the intuitions that underly our approach are much more closely aligned with those exploited in classical planning. Indeed, states are implicitly aggregated by a process of *abstraction*—the removing of certain variables from the state space description. Second, our methods are not (inherently) approximation techniques—the basic procedures produce exact solutions and value functions.[3] We will, however, describe modifications of our techniques that allow approximate solutions to be constructed.

There are two approaches to state aggregation that bear similarity to our method. The first is the model minimization approach of Givan and Dean [26, 27, 39]. In this work, the notion of automaton minimization [42, 51] is extended to MDPs and is used to analyze abstraction techniques such as those presented in [30]. More closely related to the specific model we propose in the current paper is that of Dietterich and Flann [32, 33]. They apply regression methods to the solution of MDPs (and consider this problem in the context of reinforcement learning in addition). Their original proposal [32] is restricted to MDPs with goal regions and deterministic actions (represented using STRIPS operators), thus rendering true goal-regression techniques directly applicable. This is extended in [33] to allow stochastic actions, thus providing a stochastic generalization of goal regression. We discuss these models in more detail in Section 4.7.

1.3 Outline

In Section 2 we describe the basic MDP model, various concepts that are used in the solution of MDPs, as well as several classical algorithms for solving MDPs.

In Section 3, we define a particular compact representation of an MDP, using *dynamic Bayesian networks* [25, 29]—a special form of Bayesian network [57]—to represent the dependence between variables before

[3] More accurately, they produce solutions that are identical to their standard state-based counterparts, which may be ϵ-optimal.

and after the occurrence of actions. In addition, we use decision trees to represent the conditional probability matrices quantifying the network to exploit *context-specific* independence [14], that is, independence given a particular variable *assignment*. We note that this representation is somewhat related to the probabilistic variants of STRIPS operators introduced in [40] and augmented in [30]. We also describe the decision-tree representation of reward functions, value functions and policies.

In Section 4, we describe the basic decision-theoretic regression operator which, given a particular tree-structured value function and action network, regresses the value function through that action to produce a new value function. With this operation in hand, we develop structured analogs of classical MDP algorithms like value and policy iteration. In Section 5 we present an empirical analysis of these methods and suggest the types of problems for which it is likely to work well, unlikely to work well, and what possible approaches may help with the latter. In Section 6 we describe an extension of the algorithms presented in Section 4 to deal with correlations in action effects. We also briefly describe some work that leverages the structured methods described in Section 4 to provide *approximate solutions* for structured MDPs. We conclude in Section 7 with some brief discussion of recent work that is related to, or extends, these ideas, and describe some promising directions for future research.

2 Markov Decision Processes

MDPs can be viewed as stochastic automata in which actions have uncertain effects, inducing stochastic transitions between states, and in which the precise state of the system is known only with a certain probability. In addition, the expected value of a certain course of action is a function of the transitions it induces, allowing rewards to be associated with different aspects of the problem rather than with an all-or-nothing goal proposition. Finally, plans can be optimized over a fixed finite period of time, or over an infinite horizon, the latter suitable for modeling ongoing processes. These make MDPs ideal models for many decision-theoretic planning problems (for further discussion of the desirable features of MDPs from the perspective of modeling DTP problems, see [11, 17, 28, 35]).

In this section, we describe the basic MDP model and consider several classical solution procedures. Primarily for reasons of presentation, we do not consider action costs in our formulation of MDPs. All utilities are associated with states (or propositions). However, more general cost/reward models could easily be incorporated with our framework. Furthermore, we restrict our attention to finite state and action spaces. Finally, we make the assumption of *full observability*: despite the uncertainty associated with action effects, the planning (or plan-executing) agent can observe the exact outcome of any action it has taken and knows the precise state of the system at any time. *Partially observable MDPs* (POMDPs) [21, 53, 73] are much more computationally demanding than fully observable MDPs. However, we will make a few remarks on the application of our techniques to POMDPs at the conclusion of this article.[4]

[4] See [16] for more detailed investigations of this type.

We refer the reader to [5, 11, 62] for further material on MDPs.

2.1 The Basic Model

A Markov decision process can be defined as a tuple $\langle \mathcal{S}, \mathcal{A}, T, R\rangle$, where $\mathcal{S}$ is a finite set of *states* or possible worlds, $\mathcal{A}$ is a finite set of *actions*, T is a *state transition function*, and R is a *reward function*. A state is a description of the system of interest that captures all information about the system relevant to the problem at hand. In typical planning applications, the state is a possible world, or truth assignment to the logical propositions with which the system is described. The agent can control the state of the system to some extent by performing actions $a \in \mathcal{A}$ that cause *state transitions*, movement from the current state to some new state. Actions are stochastic in that the actual transition caused cannot generally be predicted with certainty. The transition function T describes the effects of each action at each state. $T(s_i, a)$ is a probability distribution over $\mathcal{S}$: $T(s_i, a)(s_j)$ is the probability of ending up in state $s_j \in \mathcal{S}$ when action a is performed at state s_j. We will denote this quantity by $\Pr(s_i, a, s_j)$. We require that $0 \leq \Pr(s_i, a, s_j) \leq 1$ for all s_i, s_j, and that for all s_i, $\sum_{s_j \in \mathcal{S}} \Pr(s_i, a, s_j) = 1$. The components $\mathcal{S}$, $\mathcal{A}$ and T determine the dynamics of the system being controlled. We assume that each action can be performed at each state. In more general models, each state can have a different *feasible action set*, but this is not crucial here.[5]

The states that the system passes through as actions are performed correspond to the *stages* of the process. The system starts in a state s_i at stage 0. After t actions are performed, the system is at stage t. Given a fixed "course of action," the state of the system at stage t can be viewed as a random variable S^t; similarly, we denote by A^t the action executed at stage t. Stages provide a rough notion of time for MDPs. The system is *Markovian* due to the nature of the transition function; that is,

$$\Pr(S^t | A^{t-1}, S^{t-1}, A^{t-2}, S^{t-2}, \cdots A^0, S^0) = \Pr(S^t | A^{t-1}, S^{t-1})$$

The fact that the system is fully observable means that the agent knows the true state at each stage t (once that stage is reached), and its decisions can be based solely on this knowledge.

A *stationary, Markovian policy* $\pi : \mathcal{S} \rightarrow \mathcal{A}$ describes a course of action to be adopted by an agent controlling the system and plays the same role as a plan in classical planning. An agent adopting such a policy performs action $\pi(s)$ whenever it finds itself in state s. Such policies are Markovian in the sense that action choice at any state does not depend on the previous system history, and are stationary since action choice does not depend on the stage of the decision problem. For the problems we consider, optimal stationary, Markovian policies always exist. In a sense, π is a conditional and *universal plan* [67], specifying an action

[5] We could model the applicability conditions for actions using preconditions in a way that fits within our framework below. However, we prefer to think of actions as action attempts, which the agent can execute (possibly without effect or success) at any state. Preconditions may be useful to restrict the planning agent's attention to potentially "useful" actions, and thus can be viewed as a form of heuristic guidance (e.g., don't bother considering attempting to open a locked door). This will not impact what follows in any important ways.

to perform in every possible circumstance. An agent following policy π can also be thought of as a reactive system.

A number of optimality criteria can be adopted to measure the value of a policy π. We assume a bounded, real-valued reward function $R : \mathcal{S} \to \Re$. $R(s)$ is the instantaneous reward an agent receives for occupying state s. More general reward models are possible, though none of these introduce any special complications for our algorithms. One common generalization allows $R(s)$ to be a random variable—if this is the case, taking its expectation as the (deterministic) reward for state s has no impact on value or policy calculations.[6] Often one allows reward $R(s, a)$ to depend on the action taken, so as to model the costs of various actions. To keep the presentation simple, we will not consider this possibility in the development of our algorithms; but we will point out the very minor adjustments one must make to account for action costs at appropriate points in the presentation of our methods.

We take a *Markov decision problem* to be an MDP together with a specific optimality criterion. We will use the abbreviation *MDP* to refer to the specific problem (process with optimality criterion) as well as the process, with context distinguishing the precise meaning. Optimality criteria vary with the horizon of the process being controlled and the manner in which future reward is valued. In this paper, we focus on *discounted infinite-horizon* problems: the *current* value of a reward received t stages in the future is discounted by some factor $\beta^t (0 \leq \beta < 1)$. This allows simpler computational methods to be used, as discounted total reward will be finite.[7] The infinite-horizon model is important because, even if a planning problem does not proceed for an infinite number of stages, the horizon is usually *indefinite*, and can only be bounded loosely. Furthermore, solving an infinite-horizon problem is often more computationally tractable than solving a very long finite-horizon problem. Discounting has certain other attractive features, such as encouraging plans that achieve goals quickly, and can sometimes be motivated on economic grounds, or can be justified as modeling expected total reward in a setting where the process has probability $1 - \beta$ of terminating (e.g., the agent breaks down) at each stage. We refer to [62] for further discussion of MDPs and different optimality criteria.

The *value* of a policy π (under this optimality criterion) is simply the expected sum of discounted future rewards obtained by executing π. Since this value depends on the state in which the process begins, we use $V_\pi(s)$ to denote the value of π at state s. A policy π^* is *optimal* if, for all $s \in \mathcal{S}$ and all policies π, we have $V_{\pi^*}(s) \geq V_\pi(s)$. We are guaranteed that such optimal (stationary) policies exist in our setting [62]. The (optimal) value of a state $V^*(s)$ is its value $V_{\pi^*}(s)$ under any optimal policy π^*. We take the problem of decision-theoretic planning to be that of determining an optimal policy (or an approximately optimal or satisficing policy).

[6] Similarly, if rewards depend on the transition from s_i to s_j (i.e., take the form $R(s_i, s_j)$) expectations can be used if we allow reward to depend on actions, as we discuss below.

[7] Our methods apply directly to finite-horizon problems as well, and with suitable modification can be used in the computation of average-optimal policies. We do not pursue this here.

2.2 Solution Methods

Policy Evaluation and Successive Approximation

Given a fixed policy π, the function V_π can be computed using a straightforward iterative algorithm know as *successive approximation* [5, 62]. We proceed by constructing a sequence of *n-stage-to-go* value functions V_π^n. The quantity $V_\pi^n(s_i)$ is the expected sum of discounted future rewards received when π is executed for n stages starting at state s_i. We set $V_\pi^0(s_i) = R(s_i)$ and recursively compute

$$V_\pi^n(s_i) = R(s_i) + \beta \sum_{s_j \in \mathcal{S}} \Pr(s_i, \pi(s_i), s_j) V_\pi^{n-1}(s_j) \tag{1}$$

As $n \to \infty$, $V_\pi^n \to V_\pi$; and the convergence rate and error for a fixed n can be bounded [62]. We note that the right-hand side of this equation determines a contraction operator so that: (a) the algorithm converges for any starting estimate V_π^0; and (b) if we set $V_\pi^0 = V_\pi$, then the computed V_π^n for any n is equal to V_π (i.e., V_π is a fixed-point of this operator). We can also compute the value function V_π exactly using the following formula due to Howard [44]:

$$V_\pi(s_i) = R(s_i) + \beta \sum_{s_j \in \mathcal{S}} \Pr(s_i, \pi(s_i), s_j) V_\pi(s_j) \tag{2}$$

We can find the value of π for all states by solving this set of linear equations $V_\pi(s),\ \forall s \in \mathcal{S}$.

Value Iteration

By *solving* an MDP, we refer to the problem of constructing an optimal policy. *Value iteration* [3] is a simple iterative approximation algorithm for optimal policy construction that proceeds much like successive approximation, except that at each stage we choose the action that maximizes the right-hand side of Equation 1:

$$V^n(s_i) = R(s_i) + \max_{a \in \mathcal{A}} \{\beta \sum_{s_j \in \mathcal{S}} \Pr(s_i, a, s_j) V^{n-1}(s_j)\} \tag{3}$$

The computation of $V^n(s)$ given V^{n-1} is known as a *Bellman backup*. The sequence of value functions V^n produced by value iteration converges linearly to V^*. Each iteration of value iteration requires $O(|\mathcal{S}|^2|\mathcal{A}|)$ computation time, and the number of iterations is polynomial in $|\mathcal{S}|$.

For some finite n, the actions a that maximize the right-hand side of Equation 3 form an optimal policy, and V^n approximates its value.

One simple stopping criterion requires termination when

$$||V^{i+1} - V^i|| \leq \frac{\varepsilon(1-\beta)}{2\beta} \tag{4}$$

(where $||X|| = \max\{|x| : x \in X\}$ denotes the supremum norm). This ensures the resulting value function

V^{i+1} is within $\frac{\varepsilon}{2}$ of the optimal function V^* at any state, and that the induced policy is ε-optimal (i.e., its value is within ε of V^*) [62]. Another stopping criterion uses the *span seminorm*, $||V^{i+1} - V^i||_s$, where $||X||_s = \max\{x : x \in X\} - \min\{x : x \in X\}$. Similar bounds on the quality of the induced policy can be provided.[8]

A concept that will be useful later is that of a *Q-function*. Given an arbitrary value function V, we define $Q_a^V(s)$ as

$$Q_a^V(s_i) = R(s_i) + \beta \sum_{s_j \in \mathcal{S}} \Pr(s_i, a, s_j) V(s_j) \tag{5}$$

Intuitively, $Q_a^V(s)$ denotes the value of performing action a at state s and then acting in a manner that has value V [84]. In particular, we define Q_a^* to be the Q-function defined with respect to V^*, and Q_a^n to be the Q-function defined with respect to V^{n-1}. In this manner, we can rewrite Equation 3 as:

$$V^n(s) = \max_{a \in \mathcal{A}} \{Q_a^n(s)\} \tag{6}$$

Policy Iteration

Policy iteration [44] is another optimal policy construction algorithm that produces exact policies and value functions. It proceeds as follows:

1. Let π' be any policy on $\mathcal{S}$
2. While $\pi \neq \pi'$ do
   (a) $\pi := \pi'$
   (b) For all $s \in \mathcal{S}$, calculate $V_\pi(s)$ by solving the set of $|\mathcal{S}|$ linear equations given by Equation 2
   (c) For all $s_i \in \mathcal{S}$, if there is some action $a \in \mathcal{A}$ such that
   $$R(s_i) + \beta \sum_{s_j \in \mathcal{S}} \Pr(s_i, a, s_j) V_\pi(s_j) > V_\pi(s_i)$$
   then $\pi'(s_i) := a$; otherwise $\pi'(s_i) := \pi(s_i)$
3. Return π

The algorithm begins with an arbitrary policy and alternates repeatedly (in Step 2) between an evaluation phase (Step b) in which the current policy is evaluated, and an improvement phase (Step c) in which local improvements are made to the policy. This continues until no local policy improvement is possible. The algorithm converges quadratically and in practice tends to do so in relatively few iterations compared to

[8] We refer to [62] for a detailed discussion of more refined stopping criteria and error bounds for value iteration, and how assurances of optimality (rather than ε-optimality) can be provided using techniques like action elimination.

value iteration [62]. However, each evaluation step requires roughly $O(|\mathcal{S}|^3)$ computation (using the most naive methods for solving the system of equations) and each improvement step is $O(|\mathcal{S}|^2|\mathcal{A}|)$.

The policy evaluation step can also be implemented using successive approximation rather than solving the linear system directly.

Modified Policy Iteration

While policy iteration tends to converge faster in practice than value iteration, the cost per iteration is rather high due to the system of linear equations that must be solved. Puterman and Shin [63] have observed that the exact value of the current policy is typically not needed to check for improvement. Their *modified policy iteration* algorithm is exactly like policy iteration except that the evaluation phase uses some (usually small) number of successive approximation steps instead of the exact solution method. This algorithm tends to work extremely well in practice and can be tuned so that both policy iteration and value iteration are special cases [62, 63]. Few acceptable formal criteria exist for choosing the number of successive approximation steps to invoke, this quantity generally being determined empirically.

3 Bayesian Network Representations of MDPs

While the MDP framework provides a suitable semantic and conceptual foundation for DTP problems, the direct representation of planning problems as MDPs—and the direct implementation of dynamic programming algorithms to solve them—often proves problematic due to the size of the state spaces of many planning problems. Generally, planning problems are described in terms of a set of domain *features* sufficient to characterize the state of the system. Unfortunately, state spaces grow exponentially in the number of features of interest. Because of Bellman's so-called "curse of dimensionality," both the specification of an MDP—in particular, the specification of system dynamics and a reward function—and the computational methods used to solve MDPs must be tailored to resolve this difficulty. In this section we focus on the representation of MDPs in *factored* (or feature-based) problems. In the following section we describe how to exploit our proposed representations computationally in dynamic programming algorithms.

To illustrate our representational methodology, we will use the following example of a feature-based, stochastic, sequential decision problem. We suppose a robot is charged with the task of going to a café to buy coffee and delivering the coffee to its owner in her office. It may rain on the way, in which case the robot will get wet, unless it has an umbrella. The umbrella is kept in the office, and the robot is able to move between the two locations (café and office), buy coffee, deliver (hand over) coffee to its owner, and pick up the umbrella—all under suitable conditions. We have six boolean propositions that characterize this domain:

- O: the robot is located at the office: O means the robot at the office, $\overline{O}$ means it is at the café;
- W: the robot is wet;
- U: the robot has its umbrella;

- R: it is raining;
- *HCR*: the robot has coffee in its possession; and
- *HCO*: the robot's owner has coffee

We also have four actions:

- *Go*: moves the robot to the opposite (of the current) location;
- *BuyC*: buy coffee, which provides the robot with coffee if it is at the café
- *DelC*: the robot hands coffee over to the user, if it is in the office;
- *GetU*: the robot picks up the umbrella if it is in the office

The effect of these actions may be noisy (i.e., with a certain probability may not have the intended or prescribed effect).

3.1 Bayesian Network Action Representation

It has long been recognized in the planning community that explicitly specifying the effects of actions in terms of state transitions is problematic. The intuition underlying the earliest representational mechanisms for reasoning about action and planning—the situation calculus [55] and STRIPS [36] being two important examples—is that actions can often be more compactly and more naturally specified by describing their effects on state variables. For example, in the STRIPS action representation, the state transitions induced by actions are represented implicitly by describing only the effects of actions on features that *change* value when the action is executed. Factored representations can be very compact when individual actions affect relatively few features, or when their effects exhibit certain regularities.

To deal with stochastic actions, we must extend these intuitions somewhat. Rather than stating what value a variable takes when an action is performed, we must provide a distribution over the possible values a variable can take, perhaps conditional on properties of the state in which the action was performed. To exploit the potential independence of an action's effects, and regularities in these effects when the action is performed in different states, we will adopt dynamic Bayesian networks as our representation scheme. We note that other representations are possible, such as the stochastic STRIPS rules described in [40, 41, 50]. However, we will see below that the Bayesian network methodology offers certain advantages.

3.1.1 The Basic Graphical Model

Formally, we assume that the system state can be characterized by a finite set of random variables $\mathbf{X} = \{X_1, \cdots, X_n\}$, each with a finite domain $\mathit{val}(X_i)$ of possible values it can take. We often use *propositional* or *boolean* variables in our examples, which can take the values $\top$ (true) or $\bot$ (false). The possible states of the system are simply the possible assignments of values to variables; that is:

$$\mathcal{S} = \mathit{val}(X_1) \times \mathit{val}(X_2) \times \cdots \times \mathit{val}(X_n)$$

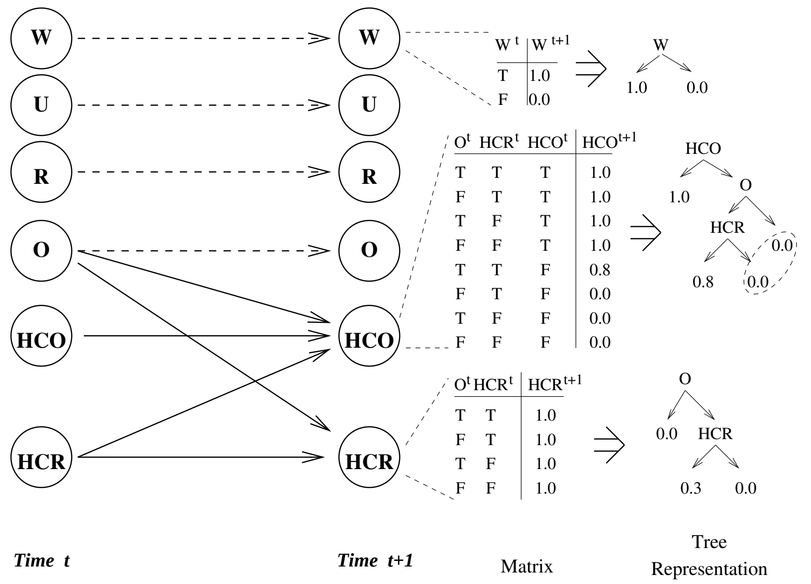


Figure 1: Action Network for *DelC* with three of six CPTs shown

Just as we use the random variable S^t to denote the state of the system at stage t, so too we use X_i^t to denote that value taken by state variable X_i at time t. In our example, *HCR*^{t} is a variable taking value $\top$ or $\perp$ depending on whether the robot has coffee at stage t of the decision process.

A *Bayesian network* [57] is a representational framework for compactly representing a probability distribution in factored form. Although these networks have most typically been used to represent atemporal problem domains, we can apply the same techniques to capture temporal distributions [29], as well as the effects of stochastic actions. Formally, a Bayes net is a directed acyclic graph with vertices corresponding to random variables and an edge between two variables indicating a direct probabilistic dependency between them. A network so constructed also reflects implicit *independencies* among the variables. The network must be *quantified* by specifying a probability distribution for each variable (vertex) conditioned on all possible values of its immediate parents in the graph. In addition the network must include a marginal distribution for each vertex that has no parents. Together with the independence assumptions defined by the graph, this quantification defines a unique joint distribution over the variables in the network. The probability of any event over this space can then be computed using algorithms that exploit the independencies represented in the graph structure. We refer to Pearl [57] for details.

We can use *dynamic Bayesian networks (DBNs)* [29] to represent the transition probabilities associated with a specific action as illustrated in Figure 1. The nodes in the network correspond to state variables X_i, and can be partitioned into two sets: those representing the state of the system before the action is performed, X_i^t; and those representing the state after the action is executed, X_i^{t+1}. Arcs between these nodes represent direct probabilistic (or causal) influence among the corresponding variables under the action in question. Arcs are only directed from pre-action variables to post-action variables—we call these *diachronic arcs*—or from post-action variables to other post-action variables—we call these *synchronic arcs*. Note that the network must, however, be acyclic. The network in Figure 1 represents the effects of the action *DelC* (deliver coffee). We see that the effect of that action on the variable HCO^{t+1} depends directly on the variables O^t, HCO^t and HCR^t.[9]

The network for action a is quantified by providing a family of conditional probability distributions for each post-action variable X_i^{t+1}. More precisely, let $\Pi(X_i^{t+1})$ be the *parents* (i.e., the predecessors) of X_i^{t+1}, and partition the parents into two sets: those parents $\Pi^t(X_i^{t+1})$ that occur among the stage t variables, and those $\Pi^{t+1}(X_i^{t+1})$ that occur among the stage $t+1$ variables. For any instantiation $\mathbf{x}$ of the variables $\Pi(X_i^{t+1})$, we must specify a probability distribution $\Pr(X_i^{t+1}|\mathbf{x})$. This family of distributions is usually referred to as the *conditional probability table (CPT)* for variable X_i^{t+1}, since these distributions are often represented in tabular form (see below). We write $CPT(X_i, a)$ to denote this family.

Let $\mathbf{x} = \mathbf{x}^t \cup \mathbf{x}^{t+1}$, where $\mathbf{x}^t$ (resp. $\mathbf{x}^{t+1}$) denotes the assignment $\mathbf{x}$ restricted to $\Pi^t(X_i^{t+1})$ (resp. $\Pi^{t+1}(X_i^{t+1})$), and let $\mathbf{y}^t$ be any instantiation of the variables $\{X_j^t : X_j^t \notin \Pi^t(X_i^{t+1})\}$. The semantics of this family of conditional distributions is given by:

$$\Pr(X_i^{t+1} = x|\mathbf{x}^t, \mathbf{y}^t, \mathbf{x}^{t+1}, A^t = a) = \Pr(X_i^{t+1} = x|\mathbf{x})$$

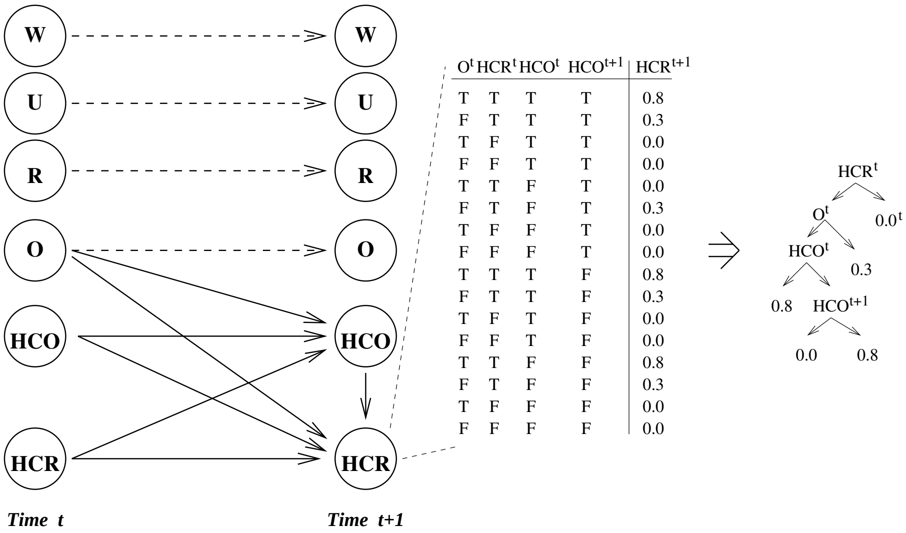
In other words, the distribution governing state variable X^{t+1} when action a is performed at stage t depends on its parents; furthermore, once the state variables $\Pi(X_i^{t+1})$ are known, X_i^{t+1} is independent of other variables at stage t.

Figure 1 illustrates these points in a simple case of the *DelC* action, where there are no synchronic arcs. The family of conditional distributions for HCO^{t+1} is given by a table: for each instantiation of variables O^t, HCO^t and HCR^t, the probability that $HCO^{t+1} = \top$ is provided.[10] This CPT can be explained by observing that: if the owner has coffee prior to the action, she still has coffee after the action; if the robot has coffee and is in the office, it will successfully hand over the coffee with probability 0.8; and if the robot does not have coffee, or is in the wrong location, it will not cause its owner to have coffee. Notice that this representation allows one to specify the *conditional effects* of a stochastic action: the effect of an action on a specific variable can vary with conditions on the pre-action state.

The effect of the action on W (wet) is also shown, and is especially simple: the robot is wet (resp. dry) with probability one if it was wet (resp. dry) before the action was performed. This variable is said to *persist* under the action *DelC*. The effects on U, O and R are captured by similar *persistence relations* (not shown). Finally, the effect of *DelC* on *HCR* is explained as follows: if the robot attempts *DelC* when it isn't in the

[9] While this example has no synchronic arcs, we will see below an example where these occur.

[10] With boolean variables, we adopt the usual convention of specifying only the probability of truth, with the probability of falsity given by one minus this value.



O^t	HCR^t	HCO^t	HCO^{t+1}	HCR^{t+1}
T	T	T	T	0.8
F	T	T	T	0.3
T	F	T	T	0.0
F	F	T	T	0.0
T	T	F	T	0.0
F	T	F	T	0.3
T	F	F	T	0.0
F	F	F	T	0.0
T	T	T	F	0.8
F	T	T	F	0.3
T	F	T	F	0.0
F	F	T	F	0.0
T	T	F	F	0.8
F	T	F	F	0.3
T	F	F	F	0.0
F	F	F	F	0.0

Figure 2: A Modified Action Network for *DelC* with a Synchronic Arc

office, there is a 0.7 chance a passerby will take the coffee; if the robot is in the office it will lose the coffee with certainty. Since there is a 0.8 chance of the user getting coffee, the 0.2 chance of the user not getting coffee can be attributed to spillage.

Because there are no synchronic arcs, the action's effect on each of the state variables is independent, given knowledge of the state S^t. In particular, for any state S^t, we have

$$\begin{aligned}&\Pr(W^{t+1}, U^{t+1}, R^{t+1}, O^{t+1}, \mathit{HCR}^{t+1}, \mathit{HCO}^{t+1} | S^t) = \\ &\quad \Pr(W^{t+1}|S^t)\Pr(U^{t+1}|S^t)\Pr(R^{t+1}|S^t)\Pr(O^{t+1}|S^t)\Pr(\mathit{HCR}^{t+1}|S^t)\Pr(\mathit{HCO}^{t+1}|S^t) \qquad (7)\end{aligned}$$

Furthermore, the terms on the right-hand side rely only on the parent variables at time t; for example, $\Pr(W^{t+1}|S^t) = \Pr(W^{t+1}|\Pi(W^{t+1})) = \Pr(W^{t+1}|W^t)$. Thus, we can easily determine state transition probabilities given the compact specification provided by the DBN.

When action networks have synchronic arcs, the calculation of transition probabilities is complicated slightly. Figure 2 shows a variant of the *DelC* action: a synchronic arc between *HCR* and *HCO* indicates a dependence between the robot losing the coffee and its owner getting her coffee when the robot is in the office. Specifically, the probability with which the robot loses the coffee depends on whether the owner successfully accepts the coffee: if the owner gets the coffee, the robot loses it; but if the owner does not (or if she already had coffee), the robot loses the coffee with probability 0.2. Such synchronic dependencies reflect *correlations* between an action's effect on different variables.

The independence of the $t+1$ variables given S^t does not hold in DBNs with synchronic arcs. Determining the probability of a resulting state requires some simple probabilistic reasoning, for example, application of the chain rule.[11] In this example, we can write

$$\Pr(\mathit{HCR}^{t+1}, \mathit{HCO}^{t+1} | S^t) = \Pr(\mathit{HCR}^{t+1} | \mathit{HCO}^{t+1}, S^t) \Pr(\mathit{HCO}^{t+1} | S^t)$$

The joint distribution over $t+1$ variables given S^t can then be computed with a slightly modified version of Equation 7:

$$\begin{aligned} &\Pr(W^{t+1}, U^{t+1}, R^{t+1}, O^{t+1}, \mathit{HCR}^{t+1}, \mathit{HCO}^{t+1} | S^t) = \\ &\quad \Pr(W^{t+1}|S^t) \Pr(U^{t+1}|S^t) \Pr(R^{t+1}|S^t) \Pr(O^{t+1}|S^t) \Pr(\mathit{HCR}^{t+1}|\mathit{HCO}^{t+1}, S^t) \Pr(\mathit{HCO}^{t+1}|S^t) \end{aligned} \tag{8}$$

Notice that only the two variables *HCR* and *HCO* are correlated—the remaining independencies allow the computation to be factored with respect to the other four variables.

We make a few observations about this representation.

1. Unlike normal Bayes nets or DBNs, we do not provide a marginal distribution over the pre-action variables. In solving fully observable MDPs, we are only concerned with the prediction of the resulting state distribution under some action *given* knowledge of the current state. As such, an action network provides a schematic representation of all $|\mathcal{S}|$ transition distributions: instantiating the pre-action variables to represent any state s allows the straightforward computation of $\Pr(\cdot, a, s)$.

2. We must specify an action network for each action. In this way, an action network can be seen as a compact specification of a transition matrix for that action. It is sometimes convenient to provide a single network with the choice of action represented as a variable, and the distributions over post-action variables conditioned on this action node. This type of representation, common in influence diagrams [69], can sometimes be more compact than a set of individual networks for each action (for example, when a variable's value persists for most or all actions); see [15] for a discussion of the relative advantages of the two approaches. We will not consider the single network representation in this paper.

3. Because of the Markov property, we need only specify the relationship between variables $\mathbf{X}^t$ and $\mathbf{X}^{t+1}$: knowledge of variables X_i^{t-k} $(k > 0)$ is irrelevant to the prediction of the values of variables X_j^{t+1} given $\mathbf{X}^t$. Furthermore, stationarity allows us to specify the dynamics schematically, with one DBN for each action characterizing its effects given state S^t for any $t \geq 0$.

4. Typically, the DBN representation of an action is considerably smaller than the corresponding transition matrix. In the example above, the system has $2^6 = 64$ states, hence each transition matrix

[11] Note that this rationale relies on the basic semantics Bayesian networks. Given two states s_i and s_j, determining $\Pr(s_i, a, s_j)$ involves simple table lookup and multiplication, the presence of synchronic arcs notwithstanding.

requires the specification of $64^2 = 4096$ parameters. The DBN in Figure 1 requires the specification of only 36 parameters, while that in Figure 2 has only 64 parameters.[12] We will see below that suitable representations of CPTs can make DBNs even more compact.

In the worst case, a "maximally connected" DBN will require the same number of parameters as a transition matrix. However, when the effects of actions exhibit certain regularities (e.g., they have the same effect on a given variable under a wide variety of circumstances) or when the effects on subsets of variables are independent, DBNs will generally be much more compact. See [11, 15] for a more detailed discussion of this point. This representation (when augmented with the CPT representations described below) also compares favorably with probabilistic variants of STRIPS operators with respect to representation size [11].

5. In a certain sense, the DBN representation might be seen to fall prey to the *frame problem* [55]: one must specify explicitly that a variable that is intuitively "unaffected" by an action persists in value. In Figure 1, for instance, an arc relating W^t and W^{t+1}, together with the corresponding CPT for W^{t+1}, are required so that one can infer that W^{t+1} has the same value as W^t (when *DelC* is executed). However, it is not hard to allow the specification of an action's effects to focus only on those variables that change, leaving the distributions over unaffected variables unspecified. Such unspecified CPTs can be filled in by default, and unspecified arcs (e.g., the dashed arcs in Figure 1) can be added automatically. The frame problem in DBNs (including aspects related to variables that change values under some conditions and not others) is discussed in detail in [15].

3.1.2 Structured Representations of Conditional Distributions

The DBN representation of an action a exploits certain regularities in the transition function induced by the action. Specifically, the effect of a on a variable X_i, given any assignment of values to its parents $\Pi(X_i^{t+1})$, is identical no matter what values are taken by other state variables at time t (or earlier). However, this representation does not allow one to exploit regularities in the distributions corresponding to *different assignments* to $\Pi(X_i^{t+1})$.

We can view the CPT for variable X_i in a DBN as a function mapping *val*$(\Pi(X_i^{t+1}))$—the set of value assignments to X_i's parents—into $\Delta($*val*$(X_i))$—the set of distributions over X_i. This function is traditionally represented in a tabular form: one explicitly lists each assignment in *val*$(\Pi(X_i^{t+1}))$ in a table along with the corresponding distribution for X_i (the tables in Figures 1 and 2 are examples of this).

In many cases, this function can be more compactly represented by exploiting the fact that the distribution over X_i is identical for several elements of *val*$(\Pi(X_i^{t+1}))$. For instance, in the CPT for *HCO* in

[12] If we exploit the fact that probabilities sum to one, we can remove one entry from each row of a transition matrix and one from each row of a CPT (as we have done in the figures). In this case, a transition matrix would require 4032 entries, while the DBNs above have only 18 and 32 parameters, respectively.

Figure 1, we see that only three distinct distributions are mapped to from the eight assignments of *HCO*'s parents. This suggests that a more compact function representation for this mapping might be useful.

In this paper, we consider the use of decision trees [64] to represent these functions. The CPT for a variable X_i in an action network will be represented as a decision tree: the interior nodes of the tree are labeled with parents of X_i; the edges of the tree are labeled with values of the parent variable from which those edges emanate; and the leaves of the tree are labeled with distributions for X_i. The semantics of such a tree is straightforward: the conditional distribution over X_i determined by any assignment $\mathbf{x}$ to its parents is given by the distribution at the leaf node on the unique branch of the tree whose (partial) assignment to parent variables is consistent with $\mathbf{x}$.

Examples of such trees are shown in both Figures 1 and 2, next to the corresponding CPTs. The mapping from *HCO*'s parents into distributions over *HCO* in Figure 1 is represented more compactly in the decision-tree format than the usual tabular fashion. The structure of the tree corresponds to our intuitions regarding the effects of *DelC*. If *HCO* was true, it remains true;[13] but if *HCO* was false, then it becomes true with probability 0.8 if O is true and *HCR* is true; otherwise it remains false. In a sense, decision trees reflect the "rule-like" structure of action effects. The tree for *HCR* in Figure 2 relies on a synchronic parent.[14]

We focus on decision trees in this paper because of their familiarity and the ease with which they can be manipulated. Furthermore, they are often quite compact when used to describe actions. However, other representations may be suitable, and more compact, in certain circumstances. CPTs could sometimes be more compactly represented using rules [60, 64], decision lists [65] or boolean decision diagrams [19]. The algorithms we provide in the next section are designed to exploit the decision-tree representation, but we see no fundamental difficulties in developing similar algorithms to exploit these other representations. Indeed, we will briefly point out extensions of the work described in this paper that exploit decision diagram representations.[15]

We note that this representation can be viewed as exploiting what is known as *context-specific independence* in Bayesian networks [14]. Just as the independence of two variables given knowledge of some subset of variables can be determined using the graphical structure of a Bayes net, additional independence can be inferred given certain *assignments* to a subset of variables (or a specific context). Algorithms for detecting these context-specific independencies using CPT representations such as decision trees and decision graphs are described in [14]. Related notions can be found in [38, 59, 70]. We note that *asymmetric representations* of conditional distributions in influence diagrams have also been proposed and investigated in [74].

[13] We adopt the convention that, for boolean variables, left edges denote $\top$ and right edges denote $\perp$.

[14] Unless a node has synchronic parents, we will omit t and $t+1$ superscripts at the interior node labels of the decision-tree CPT; all such nodes will be understood to refer to variables at time t, not $t+1$.

[15] Deterministic, goal-based regression algorithms have been developed for such representations in many circumstances; e.g., see [20] for a discussion of regression using boolean decision diagrams. Decision-theoretic generalizations of these techniques, using ideas developed in the following section, should prove useful.

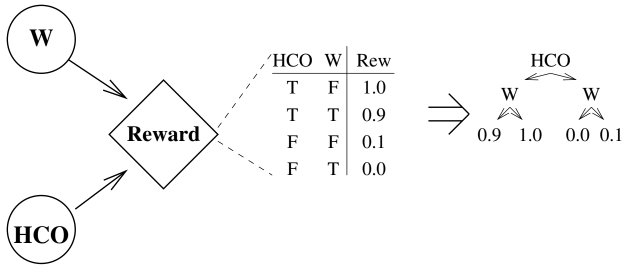


Figure 3: The Reward Tree for the Coffee Example

3.2 Reward Representation

Reward functions can be represented in a similarly compact fashion. Rather than specify a vector of reward values $R(s)$ of size $|\mathcal{S}|$, we can exploit the fact that reward is generally determined by a subset of system features. We represent the dependence of reward on specific state features using a diagram such as that shown in Figure 3: here a *reward node* (the diamond in the figure) depends only on the values of the variables W and *HCO*. The matrix represents this reward as function of the values taken by the two variables. Here we see that the best states are those in which the owner has coffee and the robot is dry, while the worst states are those in which the variables take the opposite values. Note that there is a preference for states in which the robot is wet and the owner has coffee over those where the robot stays dry and its owner is without coffee: thus, delivering coffee is a higher priority objective for the robot than staying dry.

The reward node in this example is related to the *value nodes* of influence diagrams [45, 69]. In influence diagrams, these nodes generally represent (long-term) value, whereas we use them to represent immediate reward (note that we assume stationarity of the reward process). In both cases, the independence of reward and certain state variables is exploited. Some work on influence diagrams has considered the use of reward nodes such as these, which are combined using some function (e.g., summation) to determine overall value (see, e.g., [79]). If action costs need to be modeled (i.e., reward has the form $R(s, a)$), a node representing the chosen action can be included, as they are in influence diagrams, or a separate reward function can be specified for each action, just as we specified a distinct DBN for each action to capture the process dynamics.

As with CPTs for actions, this conditional reward function can also be represented using a decision tree. In the example shown, the decision tree is no more compact than the full table; but in many instances, a decision-tree representation can be considerably more compact.

The representation of reward functions can sometimes be more compact if the reward function is comprised of a number of independent components whose values are combined with some simple function to determine overall reward. These ideas are common in the study of multi-attribute utility theory [49]. In our example, the reward function can be broken into two *additive, independent* components: one component determines the "sub-reward" determined by *HCO*—0.9 if *HCO*, 0 if $\overline{HCO}$; and the other determines the

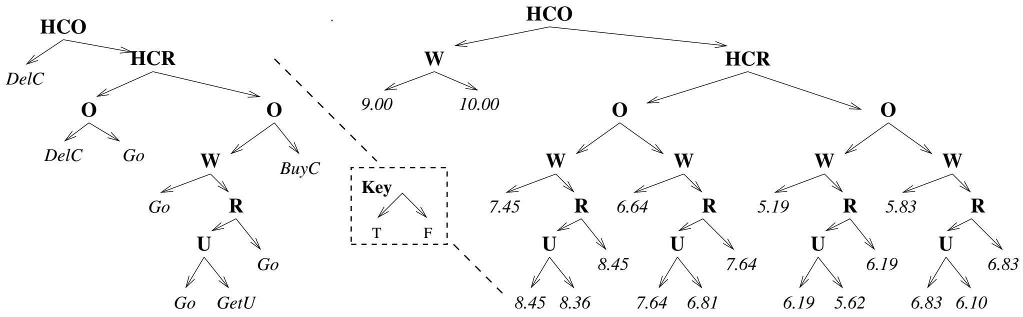


Figure 4: Examples of a Policy Tree and a Value Tree

sub-reward for W— 0.1 of $\overline{W}$ and 0 if W. The reward at any state is determined by summing the two sub-rewards at the state. If there are a number of such component reward functions, specification of reward in terms of these components (together with a combination function) can often be considerably more compact.

While we permit a number of "independent" decision trees to be specified, our algorithms do not exploit the utility independence inherent in such a reward specification. Instead, we simply combine the component functions into a single decision tree representing the true (global) reward function. How best to exploit such utility independence in MDPs in general is still an open question, though it has received some attention. For discussion of these issues, see [9, 37, 56, 71].

3.3 Value Function and Policy Representation

It is clear that value functions and policies can also be represented using decision trees (or other compact function representations). Again, these exploit the fact that value or optimal action choice may not depend on certain state variables, or may only depend on certain variables given that other variables take on specific values.

The algorithms we develop in the next section construct tree-structured representations of value functions and policies. Both *value trees* and *policy trees* have internal nodes labeled by state variables and edges labeled with (corresponding) variable values. The leaves of value trees are labeled with real values, denoting the value of any state consistent with the labeling of the corresponding branch. The leaves of policy trees are labeled with actions, denoting the action to be performed at any state consistent with the labeling of the corresponding branch. Tree representations of policies are sometimes used in reinforcement learning as well [23], though in a somewhat different fashion. Examples of a policy and value tree are given in Figure 4.

In our implementation of decision-theoretic regression and structured dynamic programming algorithms described in the next section, we allow our trees to be slightly more sophisticated in the case of multi-valued variables. When a tree splits on a variable with more than two domain values, we require only that the domain be split into two or more *subsets* of values, with each subset labeling an edge directed from the corresponding interior node. In this way, if the distinctions between the values, say, $\{x_1, x_2\}$ and $\{x_3, x_4\}$ from the domain *val*(X) are important for value function or policy prediction, but the distinction between x_1 and x_2 (or x_3 and x_4) are not, we are not forced to create four distinct subtrees under node X. We will, however, for ease of presentation describe our algorithms as if splits at any interior node of a tree are exhaustive.

These value function and policy trees can understood as effecting a form of *state aggregation*; every state satisfying the conditions labeling a particular branch of the tree are assigned the same value or action choice. In particular, this form of aggregation can be viewed as *state space abstraction*, since at any state we are generally ignoring certain features and using only the value of others in predicting, say, the value or optimal action choice at that state. It is clear that similar remarks can be applied to both the DBN action representation, where states with similar dynamics under a specific action are grouped together, and to the decision-tree reward representation.

Categorizing these types of abstraction along the dimensions described in [11, 30], our methods use *nonuniform abstraction*; that is, different features are ignored in different parts of the state space. In particular, these decision trees capture a conditional form of relevance, where certain variables are deemed to be relevant to value function prediction under certain conditions, but irrelevant under others. Compared to linear function approximators or neural network representations of value functions, our representations aggregate states in a "piecewise constant" fashion.

As we will see below, our abstraction scheme can also be classified as *adaptive* in that the aggregation of states varies over time as our algorithms progress. Finally, our main algorithm implements an *exact* abstraction process, whereby states are aggregated only when they agree exactly on the quantity being represented (e.g., value, reward, optimal action or transition distribution). We will see in Section 6.2 an *approximate* variant of this abstraction method.

4 Decision-Theoretic Regression

The decision-tree representations described in the previous section provide a means of representing value functions and policies more compactly than the straightforward table-based representations. In particular, if a tree can be constructed to represent the optimal value function or policy in which the number of internal nodes labeled by state variables is polynomial in the number of variables, then this representation will be exponentially smaller than the corresponding tabular representation. If we were given the structure of such a tree for (say) the value function by an oracle, one might imagine partitioning state space into the abstract states given by the structure of the tree, and performing dynamic programming—let's assume value iteration—over the reduced state space. In this case, each dynamic-programming iteration would require only one Bellman backup per abstract state, thus, a number of backups which is a polynomial function of the logarithm of the number of states.

Unfortunately, even if the true value function can be compactly represented using a decision tree, the regions of state space over which an intermediate value function generated by value iteration is constant need not match those of the optimal value function. Furthermore, we don't usually have access to oracles who provide us with suitable abstractions. What we need are algorithms that, for example, infer the proper *structure* of the sequence of value functions produced by value iteration, and perform Bellman backups once per abstract state *once this structure has been deduced*. One could use similar ideas in (regular or modified) policy iteration.

In this section we develop methods to do just this. These techniques exploit the structure inherent in the MDP that has been made explicit by the DBN and decision-tree representations of the system dynamics and the reward function. Specifically, given a tree-structured representation of a value function V, we derive algorithms that produce tree-structured representations of the following functions: Q-functions with respect to V; the value function obtained by performing a Bellman backup with respect to V; the value function obtained by successive approximation with respect to a fixed policy π, where π is represented with a decision tree; and the greedy policy with respect to V. These algorithms infer (to varying degrees) the appropriate *structure* of the underlying value function or policy before performing any decision-theoretic calculations (e.g., maximizations or expected value calculations). In this way, operations such as computing the expected value of an action are computed only *once per abstract state* (or region of state space, or leaf of the tree), instead of once per system state. If the size of the trees is substantially smaller than the size of the original state space, the computational savings can also be substantial.

The key to all of the operations mentioned above is the first—the computation of a Q-function Q_a^V for action a with respect to a given value function V. This operation can be viewed as the decision-theoretic analog of regression, as described in Section 1.1.

In this section, we assume that none of the action networks describing our domain contain synchronic arcs; that is, an action's effects on distinct variables are uncorrelated (given knowledge of the current state). This assumption is valid in many domains, including those we experimented with, but may be unrealistic in others. We do this primarily for reasons of exposition. Our algorithms are conceptually simple in the case where correlations are absent. As described in Section 3.1.1, determining the probability of a state variable taking a certain value after an action is performed is straightforward. When the action network has no synchronic arcs, these can be combined by multiplication to determine state transition probabilities due to their independence; but this combination requires some simple probabilistic inference when synchronic arcs are present. To avoid having the intuitions get lost in the details of this inference, we present our algorithms under the assumption of uncorrelated effects. We discuss the requisite amendments to the decision-theoretic regression algorithm when correlations are present in Section 6.1.

In Section 4.1 we describe the basic decision-theoretic regression algorithm. We describe regression of a tree-structured value function through a (tree-structured) policy in Section 4.2 and the maximization step needed for Bellman backups in Section 4.3. Section 4.4 treats the policy improvement step of policy iteration and puts the pieces above together to form the tree-structured version of (modified) policy iteration. In Section 4.5 we use these component algorithms to implement structured value iteration.

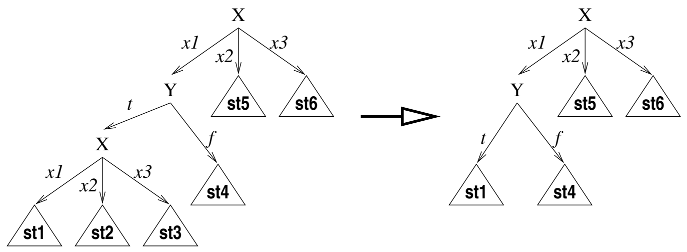


Figure 5: A tree simplified by removal of redundant nodes (triangles denote subtrees).

In the sequel we frequently use the following standard operations on decision trees:

- *Tree Simplification:* This refers to the process of removing any redundant interior nodes in a tree (i.e., meaningless splits). If two or more interior nodes lying on a single branch b of tree T are labeled by the same variable, all but the topmost such node is removed from the tree.[16] For any node so removed, we retain exactly one of its subtrees: the subtree consistent with the edge label (on b) of the topmost node (see Figure 5). In addition, if an interior node splits a tree into two or more subtrees, all of which are identical, that interior node can be removed, and the parent arc of the removed node redirected to a single "copy" of the subtree. We use *Simplify*(T) to denote the tree resulting from simplification.
- *Appending Trees:* By *appending* tree T_2 to leaf l of tree T_1, we refer to extending T_1 with the structure of T_2 at the leaf l. The new leaves added to the tree (i.e., those leaves of T_2) are labeled using some function of the label of l and the labels of the corresponding leaves in T_2 (see Figure 6). We primarily consider the following functions: the union of the information in two labels; the sum of two labels; or the maximum of two labels. We denote by *Append*(T_1, l, T_2) the tree resulting from this process (the label-combining function will always be clear from context). We denote by *Append*(T_1, T_2) the tree obtained by appending T_2 to each leaf of T_1. In other words, *Append*(T_1, T_2) denotes a tree whose branches partition state space as determined by the intersection of the partitions induced by T_1 and T_2. We usually assume the resulting trees are simplified without explicitly mentioning this fact.
- *Merging Trees:* By merging a set of trees $\{T_1, \cdots, T_n\}$, we refer to the process of producing a single tree that makes all distinctions occurring in any of the trees, and whose leaves are labeled using some function of the labels of the corresponding leaves in the original tree. This can be accomplished by repeated appending of successive trees to the merge of the earlier trees in the sequence (any append ordering will result in a semantically equivalent result, assuming the label-combination function is associative and commutative). We refer to the resulting tree as *Merge*$(\{T_1, \cdots, T_n\})$. We usually assume the resulting trees are simplified.

[16] Tree simplification is only slightly more involved when multivalued variables are allowed to split nonexhaustively. In this case, a certain variable may legitimately appear on a branch of a tree more than once, not unlike continuous splits in classification and regression trees.

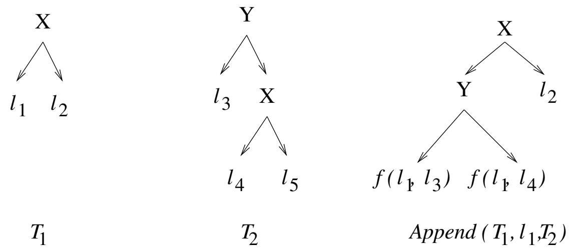


Figure 6: Appending tree T_2 to leaf l_1 of T_1, with labels combined using function f. Note that the resulting tree has been simplified.

4.1 Regression through a Single Action

A key component in all dynamic programming algorithms is the computation of the expected value of performing action a at state s_i, with respect to a given value function V. Recalling Equation 5:

$$Q_a^V(s_i) = R(s_i) + \beta \sum_{s_j \in \mathcal{S}} \Pr(s_i, a, s_j) V(s_j)$$

Notice that this computation can be divided into three phases: (a) the computation of the *expected future value* of performing a at s, $\sum_{s_j \in \mathcal{S}} \Pr(s_i, a, s_j)V(s_j)$; (b) the discounting of this future value by β; and (c) the addition of the immediate reward $R(s_i)$. If V is represented compactly using *Tree*(V), we would like to produce a compact tree-structured representation *Tree*(Q_a^V) of Q_a^V itself. We can do this by exploiting structure in the reward function (given by *Tree*(R)), the structure in the action network for a, and the structure given by *Tree*(V). Intuitively, Q_a^V takes the same value at states s_i and s_j if these states have the same reward and the same expected discounted future value.[17] That s_i and s_j have identical reward can be verified easily by examining *Tree*(R). We now focus on the latter condition.

Recall that the branches of *Tree*(V) correspond to regions of state space in which V is constant. Two states s_i, s_j will have identical expected future value (with respect to V and a) if a causes both states to transition to any "constant region of V" with the same probability. This is equivalent to saying that a, when executed at either state, makes the conditions labeling *any branch* b of *Tree*(V) true with identical probability. Thus *Tree*(Q_a^V) should (only) distinguish conditions under which action a makes some branch of *Tree*(V) true with differing odds. These conditions can be determined by examining the conditions under which the impact of a on any variable X_i occurring in *Tree*(V) varies, which in turn can be determined by examining the action network for a, specifically *Tree*(a, X_i).

We illustrate the intuitions with an example before describing the algorithm in detail. Consider a domain with four boolean variables W, X, Y and Z, with action a shown in Figure 7(a) and reward function

[17]This is a sufficient condition, not a necessary condition; more on this below.

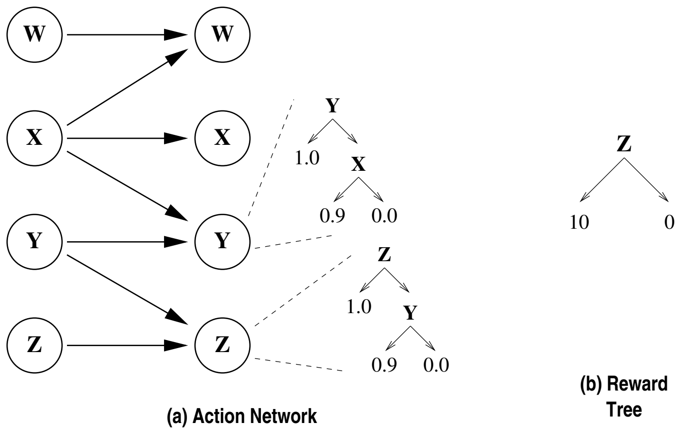


Figure 7: (a) A simple action network; and (b) a reward function.

R shown in Figure 7(b). Action a has no influence on variables W or X, but makes Y true with high probability if X is true, and makes Z true with high probability if Y is true (both Y and Z are unaffected if they are true before a is performed).

If we assume that V^0 is the reward function R, then *Tree*(V^0) is simply the tree shown in the figure and V^0 is represented very compactly as a tree with two leaves instead of a table with 16 entries. Figure 8 illustrates how a tree representing Q_a^1 (i.e., $Q_a^{V^0}$) is constructed. Building *Tree*(Q_a^1) requires that we delineate the conditions under which a will have distinct *expected future value* with respect to V^0. Because future value (with 0 stages to go) depends *only* on the truth of Z, expected future value with one stage to go depends *only* on conditions that influence the probability of Z being true or false after a is performed. These conditions are given *directly* by the network for action a, specifically by the tree representing *CPT*(Z, a). The action network in Figure 7 tells us that Z's post-action probability is influenced only by the pre-action truth values of Y and Z (and that it is independent of Y if Z is true). Each branch of this tree thus corresponds to a set of conditions on the state with 1 stage-to-go under which action a will lead with *fixed probability* to each of the regions of state space determined by *Tree*(V^0), and is denoted in Figure 8(b) as *PTree*(Q_a^1), the "probability tree" for Q_a^1. We can view this as *regressing* the tree representing V^0 through the action a to obtain the conditions (with 1 stage-to-go) under which a has identical effects with respect to the conditions relevant to prediction of V^0.

Each leaf of *PTree*(Q_a^1) is labeled with a distribution over Z—this of course dictates a distribution over the branches of *Tree*(V^0). As such, we can compute the expected future value of performing action a under any of the conditions labeling any branch of *PTree*(Q_a^1), as shown in Figure 8(c). For example, when Z is false and Y is true, a makes Z true with probability 0.9 and false with probability 0.1; so the expected future value (as dictated by V^0) of executing a under those conditions is 9. We denote by *FVTree*(Q_a^1) the "future value tree" obtained by converting the distributions over branches of $PTree(Q_a^1)$ into expected values with

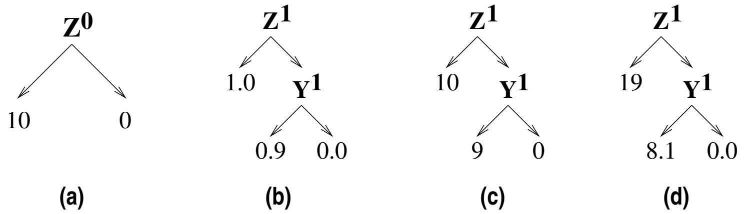


Figure 8: The various intermediate trees constructed during the regression of *Tree*(V^0) through action a to obtain *Tree*(Q^1_a) are shown here. (a) *Tree*(V^0): the tree representation of V^0; (b) *PTree*(Q^1_a): this denotes the probability of making variable Z—the only variable mentioned in *Tree*(V^0)—true with zero stages-to-go when action a is performed with one stage-to-go; (c) *FVTree*(Q^1_a): this denotes the undiscounted expected future value associated with performing a with one stage-to-go; and (d) *Tree*(Q^1_a): the tree representation of Q^1_a, obtained by discounting *FVTree*(Q^1_a) and adding to it the reward function (structured as *Tree*(R).

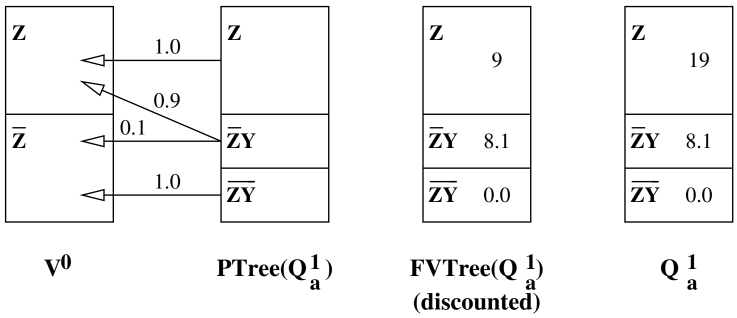


Figure 9: The state aggregation induced by the regression of *Tree*(V^0) through action a.

respect to V^0.

Finally, since Q^1_a is given by immediate reward plus discounted expected future value, the final tree is obtained by applying the discount factor to each leaf of *FVTree*(Q^1_a) and appending the reward tree to the resulting tree (using addition to combine the leaves). In our example, since only variable Z occurs in *Tree*(R), the final *Tree*(Q^1_a), illustrated in Figure 8(d), does not grow in size when *Tree*(R) is appended. In general, however, this step can cause further growth of the tree. Notice that the algorithm is unchanged if action costs are involved: a reward function of the form $R(s,a)$ can easily be accommodated in the construction of *Tree*(Q^1_a).

A slightly more direct view of the state aggregation induced by this abstraction mechanism is shown in Figure 9. Working from the left, we see the partitioning of state space induced by the series of operations described above. To the left, we have the original value function V^0, which depends only on variable Z. *PTree*(Q^1_a) can be viewed as a new partitioning of state space: each region in this partitioning contains only

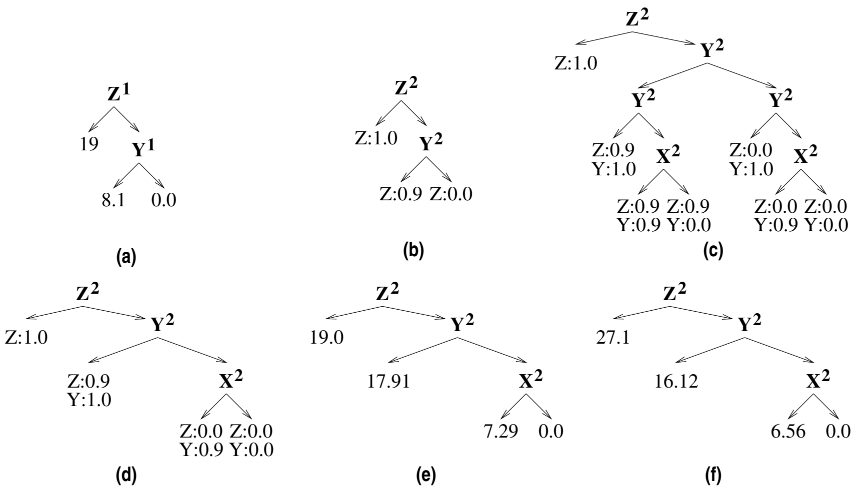


Figure 10: Regression of *Tree*(V^1) through a to obtain *Tree*(Q_a^2): (a) *Tree*(V^1); (b) Partial completion of *PTree*(Q_a^2); (c) Unsimplified version of *PTree*(Q_a^2); (d) *PTree*(Q_a^2); (e) *FVTree*(Q_a^2); and (f) *Tree*(Q_a^2).

states that have identical probability of reaching the different regions of V^0 under action a, as suggested by the figure. Within these regions of *PTree*(Q_a^1), each state has the same expected future value (in the figure the discounted future value is shown). The final step involves adding the immediate reward R to each state to obtain *Tree*(Q_a^1). In this example, this causes no further state splitting, since R depends only on variable Z (and the partition already reflects the Z, $\neg Z$ distinction).

More interesting aspects of the regression operation emerge when the value tree being regressed through a has more structure. Imagine that $V^1(s) = Q_a^1(s)$, so that *Tree*(Q_a^1) as produced above is now *Tree*(V^1) (e.g., suppose a has maximum expected value with 1 stage to go at each state during a value iteration computation). To obtain *Tree*(Q_a^2), we perform the steps shown in Figure 10. In order to predict expected future value, we must predict the probability of making any branch in *Tree*(V^1) true. Thus we must know the probability of making Z true and—if Z is possibly false—the probability of making Y true. To do this, we "regress" the individual variables occurring in *Tree*(V^1) through a and piece together the appropriate conditions.

In Step 1, the variable Z is regressed through a as above, producing a tree (simply the tree representing *CPT*(Z, a)) whose leaves are labeled with distributions over Z (see Figure 10(b)). In Step 2, we regress the variable Y through a, which returns *CPT*(Y, a) from the DBN with leaves labeled with distributions over Y. This tree is appended to the first tree (that in Figure 10(b)) at every leaf where $\Pr(Z) < 1.0$; at leaves where Z is certain to become true, the value of Y is irrelevant to prediction of V^1. The leaves of the appended tree are labeled by the union of the original labels—each leaf is now labeled by a distribution over Y and one over Z (see Figure 10(c)). Of course, this tree can be simplified by redundant node removal to give *PTree*(Q_a^2) as shown in Figure 10(d).

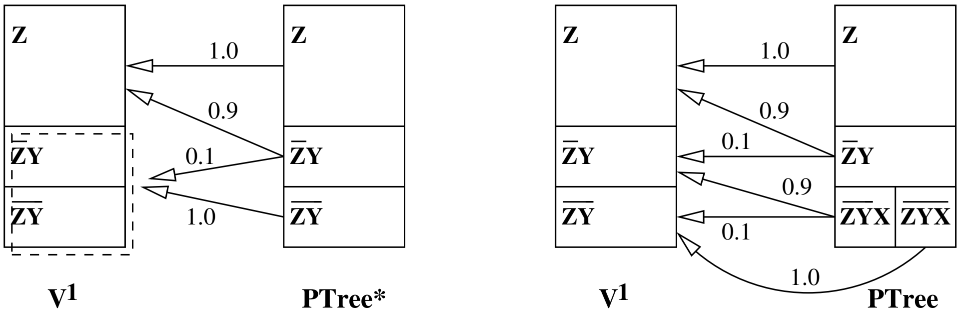


Figure 11: The state aggregation induced by the construction of *PTree*(Q_a^2) from V^1.

Once again, an alternative perspective on the construction of *PTree*(Q_a^2) is illustrated in Figure 11. Regressing V^1 through a first requires regressing Z through a and then Y through a. The regression of Z produces the state partitioning shown at the left of Figure 11, *PTree*$*$: this is sufficient to determine the probability of ending up in either the "Z-half" or "$\neg Z$-half" of V^1. Given that we end up in the $\neg Z$-half, we must then determine the probability of making Y true or false (i.e., ending up in the $\neg ZY$-quarter, or the $\neg Z \neg Y$-quarter of V^1). This is achieved by regressing Y through a. The conditions relevant to predicting Y are "overlayed" on top of the $\neg Z$ region just created to produce the more refined partitioning at the right of Figure 11, *PTree*. Notice that the prediction of Y is *not* relevant for the Z region of *PTree*(Q_a^2) since all states where Z is true will move, with probability one, to the Z-region of V^1.

Note that each leaf of *PTree*(Q_a^2) induces a joint product distribution over Y and Z. The semantics of our DBNs, under our assumption that there are no synchronic arcs, ensures that the probabilities of Y and Z becoming true under action a are independent given relevant aspects of the state. Specifically, the conditions labeling the branches of *PTree*(Q_a^2) are sufficient to ensure this independence. Thus, the product of these two distributions provides an accurate prediction of the probability of making any of the conditions labeling the branches of *Tree*(V^1) come about. For example, *PTree*(Q_a^2) tells us that if Z is false and Y is true, then a will make Z true (and leave Y true) with probability 0.9 and make Z false (and leave Y true) with probability 0.1. Each of these conditions is sufficient to "navigate" *Tree*(V^1). Thus, expected future value can be easily determined for each branch of *PTree*(Q_a^2), giving rise to *FVTree*(Q_a^2) as shown in Figure 10(e). Discounting the future value tree and adding the immediate reward, we obtain the final form of *Tree*(Q_a^2), as shown in Figure 10(f).

This example illustrates the main intuitions underlying our regression algorithm. Given *Tree*(V), the tree *Tree*(Q_a^V) is constructed in three stages. The most important phase is the first, the construction of *PTree*(Q_a^V), which provides a tree whose leaves are labeled with a set of distributions over a subset of the variables occurring in *Tree*(V).[18] The corresponding joint distribution (simply the product distribution ob-

[18]In general, not all variables will occur at the leaves. A distribution over X_i is needed only in those conditions under which X_i

Input: *Tree*(V), action a; **Output:** *Tree*(Q^V_a)

1. Let *PTree*(Q^V_a) be the tree returned by *PRegress*(*Tree*(V), a); we assume *PRegress*(*Tree*(V), a) is simplified (contains no redundant nodes).
2. Construct *FVTree*(Q^V_a) as follows. For each branch $b \in$ *PRegress*(*Tree*(V), a) (with leaf node l_b):
   (a) Let $\Pr^b$ be the joint distribution obtained from the product of the individual variable distributions labeling l_b.
   (b) Compute $v_b = \sum_{b' \in Tree(V)} \Pr^b(b')V(b')$. (Here b' are branches in *Tree*(V), $\Pr^b(b')$ is the probability of the conditions labeling that branch as given by the distribution $\Pr^b$, and $V(b')$ is the value labeling the leaf l'_b in *Tree*(V)).
   (c) Re-label leaf l_b with v_b.
3. Discount *FVTree*(Q^V_a) with discount factor β (multiply each leaf label by β).
4. Append (and simplify) the reward tree, *Tree*(R), to *FVTree*(Q^V_a), using addition as the combination function. The resulting tree is *Tree*(Q^V_a).

Figure 12: Algorithm *Regress*(*Tree*(V), a)

tained from the individual distributions) will fix a probability distribution over the branches of *Tree*(V), and thus over the values occurring in value function V. This tree is constructed by regressing each of the variables occurring in *Tree*(V) through a, in turn, and piecing together the resulting trees. The trees for the individual variables are themselves taken directly from the DBN for a.

The second phase of the algorithm simply involves using the joint distribution at each leaf of *PTree*(Q^V_a) to compute *FVTree*(Q^V_a), the expected future value tree with respect to V. The final phase involves adding the reward function to the discounted version of *FVTree*(Q^V_a) to obtain *Tree*(Q^V_a). This last step may involve expanding *FVTree*(Q^V_a) by appending *Tree*(R) to it and doing some simplification.

The algorithm *Regress*(*Tree*(V), a)—which accepts as input *Tree*(V) and action a, and returns *Tree*(Q^V_a)—is detailed in Figure 12. The bulk of the tree structure is produced in the call to the recursive algorithm *PRegress*(*Tree*(V), a), which produces *PTree*(Q^V_a). *PRegress*(*Tree*(V), a) is described in Figure 13.

Both algorithms are described in reasonably straightforward terms; and each could be implemented in slightly more complicated ways to minimize repeated operations and tree traversals. For instance, Steps 3 and 4 of *Regress* need not wait until *FVTree*(Q^V_a) is completely constructed in Step 2. Opportunities for optimization of *PRegress* include: exploiting shared substructure in the subtrees of *Tree*(V), thus intertwining the (currently) independent recursive calls to *PRegress* in Step 3(b); and combining the merging operations at different leaves in Step 4(b) so that leaves labeled by distributions with overlap in their support sets can share computation. We note that several other ways of constructing *PTree*(Q^V_a) may also prove useful. In what follows, we will not discuss in detail the fine-tuning of these tree-manipulation algorithms.[19]

The soundness of the algorithm is ensured by the following result.

Theorem 4.1 *Let PTree*(Q^V_a) *be the tree produced by the algorithm PRegress*(*Tree*(V), a). *For any branch*

is relevant to future value. In our example, a distribution over Y was not needed when the distribution over Z was concentrated on $Z = \top$.

[19] The specific performance of different approaches to tree manipulation will likely be closely tied to many domain specific features, such as the noise in actions' effects, the ordering of variables in the input representation, the problem specific structure, and so on. Until more empirical experience is obtained with these algorithms, we are unable to offer deep insights into these factors.

Input: *Tree*(V), action a; **Output:** *PTree*(Q_a^V)

1. If *Tree*(V) contains a single (leaf) node (necessarily labeled with a value), return an empty tree *PTree*(Q_a^V).
2. Let X be the variable labeling the root of *Tree*(V). Let T_X^P be the tree-structured CPT for X in the DBN for a (with leaves labeled by distributions over *val*(X)).
3. For each $x_i \in$ *val*(X) that occurs with positive probability in the distribution at some leaf in T_X^P:
   (a) Let $T_{x_i}^V$ be the subtree in *Tree*(V) attached to the root X by arc x_i.
   (b) Let $T_{x_i}^P$ be the tree produced by calling *PRegress*$(T_{x_i}^V, a)$.
4. For each leaf $l \in T_X^P$, labeled with distribution $\Pr^l$:
   (a) Let $val_l(X) = \{x_i \in val(X) : \Pr^l(x_i) > 0\}$
   (b) Let $T_l = Merge(\{T_{x_i}^P : x_i \in val_l(X)\})$, using union to combine the leaf labels (which are distributions over sets of variables).
   (c) Revise T_X^P by appending T_l to leaf l, using union to combine the leaf labels (distributions) of T_l with the label (distribution over X) of leaf l.
5. Return *PTree*$(Q_a^V) = T_X^P$.

Figure 13: Algorithm *PRegress*(*Tree*$(V), a)$

b of PTree(Q_a^V), let $\mathcal{B}$ denote the event determined by its edge labels, and assume the leaf of b is labeled by distributions $P_i(X_i)$ for $1 \le i \le k$. Let $\Pr^b$ denote the joint product distribution over $\mathbf{X} = \{X_1, \cdots, X_k\}$ induced by $\{P_1, \cdots, P_k\}$. Then:

(a) Any $\mathbf{x} \in val(\mathbf{X})$ such that $\Pr^b(\mathbf{x}) > 0$ corresponds to a unique branch of Tree(V). That is, any assignment to $\mathbf{X}$ that has positive probability is sufficient to "traverse" Tree(V) to a unique leaf node.

(b) Let $\mathbf{x} \in val(\mathbf{X})$ and s_i be any state satisfying $\mathcal{B}$. Then

$$\sum\{\Pr(s_i, a, s_j) : s_j \models \mathbf{x}\} = \Pr^b(\mathbf{x})$$

In other words, $\Pr(S^{t+1} \models \mathbf{x} \mid S^t \models \mathcal{B}, A^t = a) = \Pr^b(\mathbf{x})$.

Proof We prove part (a) inductively on the depth of tree *Tree*(V). The base case for a tree of depth 0—i.e., when *Tree*(V) consists of a single leaf labeled with a value—is immediate, since *PRegress* returns an empty tree, which is sufficient to traverse *Tree*(V) to a leaf. Now assume that the result holds for all value trees with depth less than d. Let *Tree*(V) have depth d with root labeled by variable X and subtrees *Tree*(x_i) for each $x_i \in$ *val*(X). Since all subtrees *Tree*(x_i) have depth less than d, by the inductive hypothesis *PRegress* will return a probability tree *PTree*(x_i) capturing a joint distribution over the variables in *Tree*(x_i) such that any assignment to those variables given nonzero probability allows subtree *Tree*(x_i) to be traversed to a leaf. Now *PTree*(Q_a^V) is constructed by appending to each leaf l of the tree *CPT*(X, a) those trees *PTree*(x_i) for which the distribution over X labeling l

assigns $\Pr(x_i) > 0$, and maintaining the subsequent product distribution at each resulting leaf. If this resulting product distribution at any leaf of *PRegress*$(\textit{Tree}(V), a)$ assigns $\Pr(x_i) > 0$, then we may traverse the x_i arc from the root of *Tree*(V). The fact that this distribution must include information from *PTree*(x_i) means that any event with nonzero probability permits the navigation of the subtree *Tree*(x_i) of *Tree*(V). If the resulting product distribution assigns $\Pr(x_i) = 0$, then we will never traverse the x_i arc, and the fact that the distributions from *PTree*(x_i) are not included in the product distribution is irrelevant.

To prove part (b), let leaf l_b of *PTree*(Q_a^V) be labeled by a distribution over the set of variables $\mathbf{X} = \{X_1, \cdots, X_k\}$ and the corresponding branch b labeled with conditions $\mathcal{B}$. By construction, the conditions labeling b entail the conditions labeling exactly one branch b_{X_i} of the tree for *CPT*(X_i, a), for each X_i. Denote these conditions $\mathcal{B}_{X_i}$. The semantics of the DBN, given the absence of synchronic arcs, ensures that

$$\Pr(X_i^{t+1} | \mathcal{B}_{X_i}^t, A^t = a) = \Pr(X_i^{t+1} | \mathcal{B}_{X_i}^t, A^t = a, C^{t+1}, C^t)$$

where C^t is any event over the variables X_j^t consistent with $\mathcal{B}_{X_i}$ and C^{t+1} is any event over the variables $X_j^{t+1}, j \neq i$. Since, for each X_i the distribution labeling l_b is exactly $\Pr(X_i^{t+1} | \mathcal{B}_{X_i}^t, A^t = a)$, the corresponding product distribution is

$$\Pr(X_1^{t+1} | \mathcal{B}^t, A^t = a) \Pr(X_2^{t+1} | \mathcal{B}^t, A^t = a) \cdots \Pr(X_k^{t+1} | \mathcal{B}^t, A^t = a)$$

Since the X_i^{t+1} are independent given $\mathcal{B}^t$, this is exactly:

$$\Pr(X_1^{t+1}, X_2^{t+1}, \cdots X_k^{t+1} | \mathcal{B}^t, A^t = a)$$

Since the X_i^{t+1} are independent of any event C^t consistent with $\mathcal{B}^t$, the result follows. ■

It follows almost immediately that the algorithm *Regress*$(\textit{Tree}(V), a)$ is sound, since it uses *PTree*(Q_a^V) to the determine the distribution over values in *Tree*(V), and the conditions under which to use that distribution to compute the expected future value of performing a. Adding the immediate reward to the discounted future value is straightforward.

Corollary 4.2 *Let Tree*(Q_a^V) *be the tree produced by the algorithm Regress*$(\textit{Tree}(V), a)$*. For any branch* b *of Tree*(Q_a^V)*, let* $\mathcal{B}$ *denote the event determined by its edge labels, and assume the leaf of* b *is labeled by the value* v_b*. For any state* $s_i \models \mathcal{B}$*, we have* $Q_a^V(s_i) = v_b$*. In other words, Tree*(Q_a^V) *accurately represents* Q_a^V*.*

Input: *Tree*(V), *Tree*(π); **Output:** *Tree*(Q_π^V)

1. For each action a occurring in *Tree*(π), call *Regress*(*Tree*(V), a) to produce *Tree*(Q_a^V).
2. At each leaf of *Tree*(π) labeled by a, append *Tree*(Q_a^V) to *Tree*(π), retaining the leaf labels (values) from *Tree*(Q_a^V) (deleting the action labels from *Tree*(π)).
3. Return the (simplified) resulting tree, *Tree*(Q_π^V).

Figure 14: Algorithm *Regress*(*Tree*(V), *Tree*(π))

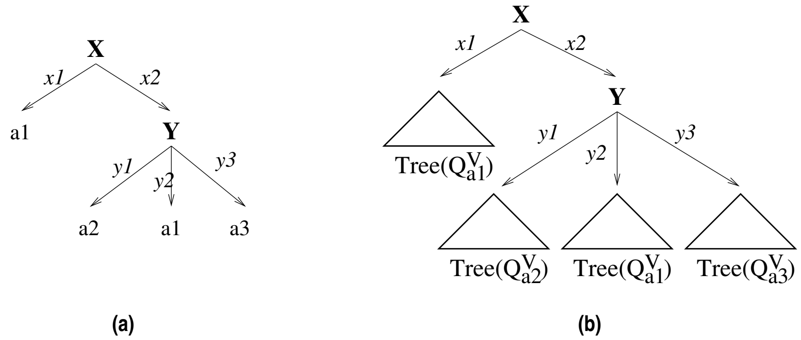


Figure 15: Regression of *Tree*(V) through *Tree*(π) to obtain *Tree*(Q_π^V): (a) *Tree*(π); and (b) *Tree*(Q_π^V).

4.2 Regression through a Policy

The algorithm for regressing a value function through an action can be generalized to regress a value function through a policy. Specifically, given value function V, we wish to produce a new value function Q_π^V that represents the value of executing policy π for one step and receiving terminal value V. This is, for instance, the key step in successive approximation for policy evaluation: given V_π^k, V_π^{k+1} is just $Q_\pi^{V_\pi^k}$.

Given a tree *Tree*(V) representing V and a tree *Tree*(π) representing π, our goal is to produce a tree *Tree*(Q_π^V) capturing Q_π^V. The algorithm *Regress*(*Tree*(V), *Tree*(π)) that does this is conceptually similar to *Regress*(*Tree*(V), a), the key difference being that different actions are performed in different regions of state space, as dictated by *Tree*(π). Our algorithm, detailed in Figure 14, reflects the most straightforward approach to producing *Regress*(*Tree*(V), *Tree*(π)). The algorithm *Regress*(*Tree*(V), a) is applied to each action occurring in *Tree*(π), and the resulting tree is appended to *Tree*(π) at each leaf where a occurs (see Figure 15 for an illustration). The tree is then suitably simplified.

Since $Q_\pi^V(s)$ is just $Q_a^V(s)$ at any state where $\pi(s) = a$, it is quite obvious that the algorithm produces a sound representation of Q_π^V.

Proposition 4.3 *Let Tree*(Q_π^V) *be the tree produced by the algorithm Regress*(*Tree*(V), *Tree*(π)). *For any branch b of Tree*(Q_π^V), *let $\mathcal{B}$ denote the event determined by its edge labels, and assume the leaf of b is labeled by the value v_b. For any state $s_i \models \mathcal{B}$, we have $Q_\pi^V(s_i) = v_b$. In other words, Tree*(Q_π^V) *accurately*

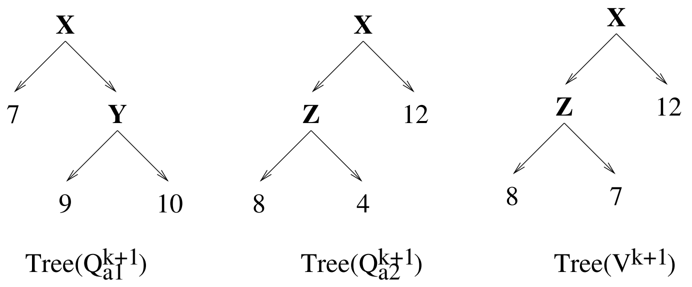


Figure 16: Merging two Q-trees *Tree*(Q_a^{k+1}) and *Tree*(Q_b^{k+1}) to obtain an improved value function, *Tree*(V^{k+1}).

represents Q_π^V.

As in the previous subsection, our algorithm is reasonably straightforward, and could be optimized somewhat to work more efficiently under certain conditions. For example, if action a occurs in *Tree*(π) only under specific conditions C, these conditions could be passed to *Regress*$($*Tree*$(V), a)$ to incrementally prune the subtrees generated by that algorithm.

4.3 Merging Q-trees

A fundamental step in both value iteration and policy iteration is the maximization step involved in a Bellman backup. In value iteration, V^{k+1} is computed by setting $V^{k+1}(s) = \max_a Q_a^{k+1}$, where Q_a^{k+1} is itself computed with respect to V^k. In policy iteration, an improved policy π' is constructed by setting $\pi'(s) = \arg\max Q_a^{V_\pi}$. Given tree representations of the appropriate Q-functions, these maximization steps can be implemented by merging these Q-trees and using maximization as the leaf-label-combining function (substituting the maximizing action names for values in the case of policy improvement).

In the case of value iteration, assume we have been given *Tree*(Q_a^{k+1}) for each action a (this is obtained by regressing *Tree*(V^k) through action a). *Tree*(V^{k+1}) can then be obtained by merging these trees and simplifying (i.e., obtaining *Merge*$(\{$*Tree*$(Q_a^{k+1}) : a \in \mathcal{A}\})$), taking each leaf label in *Tree*(V^{k+1}) to be the maximum over the corresponding labels in the trees in $\{$*Tree*$(Q_a^{k+1}) : a \in \mathcal{A}\}$. This is illustrated in Figure 16.

As mentioned earlier, there are a number of ways in which this merge operation can be implemented. Our approach is straightforward, implemented via repeated appending and simplification. Another possibility would be to reorder all Q-trees to have a common variable ordering (e.g., the ordering of the tree with highest "average" value could be retained), and simultaneously traversing all trees to find the maximizing values (and tree structure).

With policy improvement, we are given *Tree*$(Q_a^{V_\pi})$ for each a, and must produce *Tree*(π'), a tree-structured representation of some policy π' that is greedy with respect to V_π. This is achieved in a nearly identical fash-

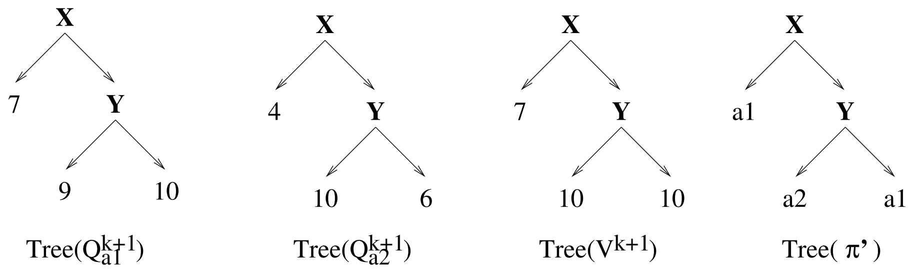


Figure 17: Merging two Q-trees *Tree*(Q_a^{k+1}) and *Tree*(Q_b^{k+1}) to obtain an improved policy, *Tree*(π').

Input: *Tree*(R), *Tree*(π); **Output:** *Tree*(V_π)

1. Set *Tree*(V_π^0) to be *Tree*(R) (immediate reward).
2. Until termination, compute *Tree*$(V_\pi^{k+1}) =$ *Regress*(*Tree*(V_π^k), *Tree*(π)).
3. Return the final tree *Tree*$(V_\pi) =$ *Tree*(V_π^n).

Figure 18: The Structured Successive Approximation (SSA) Algorithm

ion, by merging the Q-trees; but rather than labeling leaves with the maximizing Q-values, we label them with the corresponding (i.e., maximizing) action names. Once these merged values are replaced with action labels, the trees may be further simplified, since a subtree with distinct (maximizing) value labels on the leaves may have identical action labels at these leaves. Figure 17 illustrates this process. We note that the merged tree (as it exists before the maximizing Q-values are replaced by action labels) should not be simplified by collapsing identical subtrees. In the example illustrated, both leaves under node Y have identical values in the intermediate merged value tree; but these values are produced by different maximizing actions. Hence, the split of Y is relevant to the representation of π'.

4.4 Structured Successive Approximation

With the tree-structured implementations of basic operations such as expected value computation, maximization, and Bellman backups, we can now implement standard dynamic programming algorithms in a structured fashion. The first such algorithm is the successive approximation algorithm, which, given a fixed policy π, computes the value function V_π.

We assume we have been given a tree-structured policy π, represented as *Tree*(π). *Structured successive approximation (SSA)*, described in Figure 18, proceeds by constructing a sequence of value functions $V_\pi^0, V_\pi^1, \cdots$, each represented as a tree, *Tree*(V_π^k). The initial value function is simply the reward function *Tree*(R) itself, while successive value approximations *Tree*(V_π^{k+1}) are produced by regressing *Tree*(V_π^k) through *Tree*(π).

Termination is determined by some standard criterion, such as supremum norm or span seminorm. In

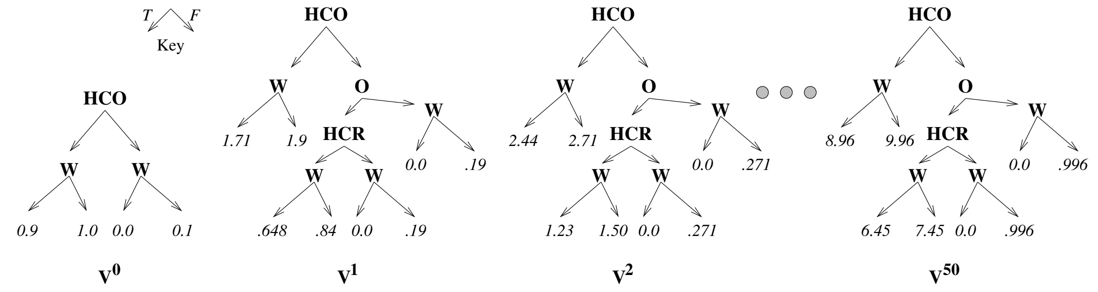


Figure 19: Fifty iterations of SSA with fixed policy *DelC*.

our implementation, we use span seminorm. To determine convergence, we must compute $||V_\pi^{k+1} - V_\pi^k||_s$ at each iteration. We do this by first merging *Tree*(V_π^{k+1}) and *Tree*(V_π^k), using difference as the combination function, to obtain *Tree*$(V_\pi^{k+1} - V_\pi^k)$. The span of this tree is determined by one tree traversal to find its maximal and minimal elements, denoted m^+ and m^-, respectively. Given a termination threshold ε, if $m^+ - m^- \leq \varepsilon$, then *Tree*$(V_\pi^{k+1})$ is returned as *Tree*(V_π), the approximate value of the policy π, with bounds given by the usual formulae.[20]

Since the termination test clearly reproduces the classical test in our tree-structured setting, it follows immediately from Proposition 4.3 that *Tree*(V_π) accurately reflects V_π.

Theorem 4.4 *Let Tree(V_π) be the tree produced by the SSA algorithm. For any branch b of Tree(V_π)), let $\mathcal{B}$ denote the event determined by its edge labels, and assume the leaf of b is labeled by the value v_b. For any state $s_i \models \mathcal{B}$, we have $V_\pi(s_i) = v_b \pm f(\varepsilon)$, where $f(\varepsilon)$ is the standard error introduced by the termination criterion. In other words, Tree(V_π) accurately represents V_π.*

Note that the approximation error is due to the nature of successive approximation itself, not the use of the decision-tree representation. The usual error terms for either the span seminorm or supremum norm stopping criteria, as described in Section 2.2, apply here directly.

Figure 19 illustrates the sequence of value trees produced by fifty iterations of SSA using our running example, where the policy being used is, for simplicity, the uniform application of action *DelC* at every state. After fifty iterations, the estimated value $V^\pi(s)$ is within 0.04 of its true value. The SSA algorithm has discovered that there are only eight distinct values in this value function, and has abstracted the state space appropriately.[21] Thus, it performs eight expected value computations, or backups, per iteration, rather than the 64 required by the standard state-based successive approximation algorithm.

One thing we notice immediately about the sequence of value trees in this example is that its structure stabilizes very quickly. How quickly this occurs depends on a number of specific problem characteristics,

[20] Again, span semi-norm is generally used for early stopping with a good *policy*, not for accurate estimation of the value function.

[21] In fact, this value function has only six distinct values, but the minimal decision-tree representation requires eight leaves. A representation such as a decision graph, or ADD, would be able to represent this value function more concisely still.

but we can be assured that once the structure of the tree stabilizes in this fashion — that is, once the structure persists in two successive iterations — its structure will not change in any subsequent iteration.[22]

Theorem 4.5 *Let Tree*(V_π^k) *and Tree*(V_π^{k+1}) *be two trees produced by successive iterations of SSA. If Tree*(V_π^k) *and Tree*(V_π^{k+1}) *have identical structure (i.e., are identical except possibly for the value labels at their leaves), then Tree*(V_π^{k+j}) *will have the same structure for any* $j \geq 0$.

Proof Suppose *Tree*(V_π^k) and *Tree*(V_π^{k+1}) have the same structure. The algorithm *Regress*(*Tree*(V_π^k), *Tree*(π)) produces the structure of *Tree*(V_π^{k+1}) based on the structure of *Tree*(V_π^k) without regard to the values at the leaves. Since *Tree*(V_π^{k+1}) has identical structure (it differs from *Tree*(V_π^k) only in its leaf values), the algorithm will produce *Tree*(V_π^{k+2}) to have identical structure to *Tree*(V_π^{k+1}). A simple inductive argument proves the result. ■

The significance of this result lies in the fact that, once the decision tree structure stabilizes, SSA can proceed exactly as standard successive approximation. Specifically, there is no need to recompute the structure of the decision tree at subsequent iterations. SSA can reuse the same decision tree and simply perform one expected value calculation per leaf (in contrast to the standard one calculation per state) without additional overhead.

4.5 Structured Policy Iteration

Policy iteration can be implemented in a way that exploits tree structure by simply piecing together some of the components described above. *Structured policy iteration (SPI)* is detailed in Figure 20 and works by alternating phases of SSA and structured policy improvement. Policy improvement is implemented using the "maximization merge" described in Section 4.3, where action names replace values labeling the leaves. Termination is tested by comparing *Tree*(π) and *Tree*(π') to see if the policies are identical.[23]

The soundness of the component algorithms ensures that the SPI algorithm produces trees that accurately reflect the optimal policy and value function.

Theorem 4.6 *Let Tree*(π^*) *and Tree*(V^*) *be the trees produced by the SPI algorithm. For any branch* b *of Tree*(V^*)*, let* $\mathcal{B}$ *denote the event determined by its edge labels, and assume the leaf of* b *is labeled by the value* v_b*. For any state* $s_i \models \mathcal{B}$*, we have* $V^*(s_i) = v_b \pm f(\varepsilon)$*, where* $f(\varepsilon)$ *is the standard error introduced by the termination crtierion. Similarly, the policy* π^* *represented by Tree*(π^*) *is* $f(\varepsilon)$*-optimal.*

[22] If identical values are collapsed at subtrees, the structure can in fact become simpler. The following result ignores this possibility. It seems to rarely occur in practice except at the earliest stages of dynamic programming. Such collapsing can be ignored until the end of a sequence of iterations in any case, thus the practical import of the results remains.

[23] As is usual with policy iteration, if more than one action can be chosen for the greedy policy π' during policy improvement and one of the candidate actions is the same as the action taken in π, the action used by π is retained.

Input: $Tree(R)$, $Tree(\pi)$ for random initial π; **Output:** $Tree(\pi^*)$ and $Tree(V^*)$.

1. Set $Tree(\pi') = Tree(\pi)$.
2. Repeat
   (a) Set $Tree(\pi) = Tree(\pi')$.
   (b) Compute $Tree(V_\pi)$ using SSA.
   (c) Compute $Tree(Q_a^{V_\pi}) = Regress(Tree(V_\pi), a)$ for each action a.
   (d) Merge the trees $Tree(Q_a^{V_\pi})$ to obtain $Tree(\pi')$ (where π' is the greedy policy w.r.t. V_π).

   Until $\pi' = \pi$.
3. Return $Tree(\pi^*) = Tree(\pi)$ and $Tree(V^*) = Tree(V_\pi)$.

Figure 20: The Structured Policy Iteration (SPI) Algorithm

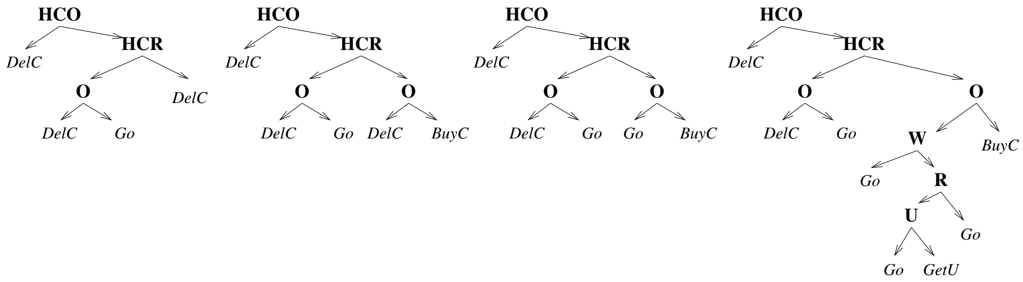


Figure 21: The sequence of improved policies produced by SPI.

Notice that the potential approximation error introduced in policy iteration is due to the fact that policy evaluation is obtained by means of successive approximation. Procedures such as action elimination [62] can be used to ensure that the obtained policy is in fact optimal.[24]

Figure 21 illustrates the sequence of four policy trees produced by SPI using our running example (with an initial policy that uniformly delivers coffee at each state). A fifth policy tree is created and compared to the fourth, but is found to be identical; thus the final (fourth) tree is the optimal policy for this problem.[25] The final value function is shown in Figure 22. Notice that the final policy consists of a tree with eight leaves showing that SPI is capable of discovering inherent structure in optimal policies. Furthermore, the optimal value function consists of 18 distinct leaves. Thus each policy evaluation and improvement computation involves no more (and generally fewer) than 18 expected value or maximization computations, rather than the 64 required in the standard, state-based version of policy iteration.

[24] Optimality can also be assured if policy evaluation is performed exactly by solution of the corresponding linear equations. We conjecture that a means of doing so in a way that exploits structure may be possible, but have not explored this possibility.

[25] The action *DelC* is essentially a "no-op" for this domain when the robot does not have coffee. Thus it remains as the action selected (since it was the sole action in the initial policy) when nothing of interest is to be done. If *DelC* incurred some cost and a no-op were included in the set of actions, the no-op would be optimal at all branches where *DelC* occurs other than $\langle \overline{HCO}, HCR, O \rangle$.

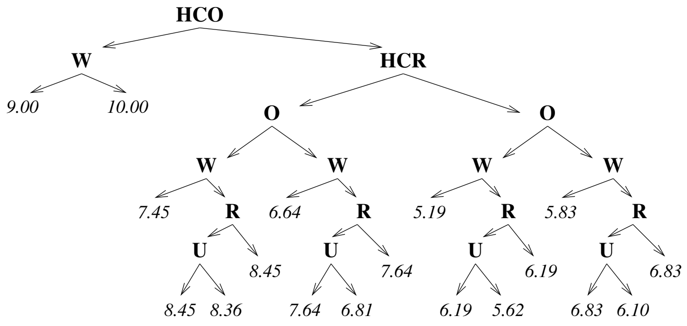


Figure 22: The optimal value function produced by SPI.

Input: *Tree*(R). **Output:** *Tree*(π^*) and *Tree*(V^*).

1. Set *Tree*$(V^0) = $ *Tree*(R).
2. Repeat
   (a) Compute *Tree*$(Q_a^{V^k}) = $ *Regress*(*Tree*$(V^k), a)$ for each action a.
   (b) Merge (via maximization) the trees *Tree*$(Q_a^{V^k})$ to obtain *Tree*(V^{k+1}).

   Until termination criterion holds (w.r.t. *Tree*(V^{k+1}), *Tree*(V^k)).
3. Set *Tree*$(V^*) = $ *Tree*(V^{k+1}).
4. Compute *Tree*$(Q_a^{V^*}) = $ *Regress*(*Tree*$(V^*), a)$ for each action a.
5. Merge the trees *Tree*$(Q_a^{V^*})$ to obtain *Tree*(π^*) (where π^* is the greedy policy w.r.t. V^*).
6. Return *Tree*(π^*) and *Tree*(V^*).

Figure 23: The Structured Value Iteration (SVI) Algorithm

Finally, we point out that modified policy iteration can be implemented in exactly the same fashion. The only change required of the SPI algorithm as presented is that one would perform a fixed number of steps of SSA in order to evaluate a policy rather than implementing SSA until convergence.

4.6 Structured Value Iteration

For completeness, we also describe a *structured value iteration (SVI)* algorithm. The results described below are all based on SPI, but SVI plays an important role in approximation, as we discuss in Section 6.2. SVI is shown in Figure 23 and works by repeatedly constructing Q-trees for the current estimated value function and performing a maximization merge to obtain an improved estimate of the value function. Once convergence according to some termination criterion is attained, the greedy policy with respect to that value function is produced via another maximization merge (where values are replaced by action names). The algorithm is obviously sound given the soundness of its components.

4.7 Related Work

We have already pointed out a number of techniques for solving large MDPs, but two approaches to state aggregation warrant further discussion due to their similarity to our method.

Dietterich and Flann [32, 33] also consider the application of regression methods to the solution of MDPs in the context of reinforcement learning. Their original proposal [32] is restricted to MDPs with goal regions and deterministic actions (represented using STRIPS-like operators), thus rendering true goal-regression techniques directly applicable. They extend their approach in [33] to allow stochastic actions, thus providing a stochastic generalization of goal regression. One key difference between their model and ours is that they deal exclusively with goal-based problems whereas we allow general reward functions. Thus we might classify their work as stochastic regression and ours as decision-theoretic regression. The general motivation and spirit of their proposal is very similar to ours, but focuses on different representations. In the abstract, Dietterich and Flann simply require operators (actions) that can be inverted, and they develop grid-world navigation and chess end-games as examples of deterministic regression. In the stochastic case, Dietterich and Flann place an emphasis is on algorithms for manipulating rectangular regions of grid worlds. In contrast, our approach deals with general DBN/decision-tree representations of discrete, multi-variable systems. Our decision-tree representation has certain advantages in multi-variable domains (e.g., we will see below that it provides leverage for approximation). In navigation domains (to take one example), the region-based representation is clearly superior as they offer very little structure that can be exploited by a decision tree. Both approaches can be seen as particular instances of a more general approach to regression in MDPs.

The model minimization approach of Givan and Dean [26, 27, 39] is also related to our model. In this work, the notion of automaton minimization [42, 51] is extended to MDPs and is used to analyze abstraction techniques such as those presented in [30]. As such this technique can be viewed as providing a more abstract view of the type of work we describe here. The emphasis is not on specific algorithms for regression, but rather a development of a theoretical framework in which abstraction methods such as those proposed here, as well as others, are viewed as minimization algorithms for stochastic automata. Intuitively, a minimized automaton is one in which states are aggregated if they agree on a certain property of interest. For example, before solving an MDP, it can be minimized by discovering blocks of states such that each state in a given block agrees on reward, and agrees on the transition probabilities for each action with respect to the block structure. Specifically, when an action is taken, each state in a block must have the same probability of moving to any other *block* (not necessarily any other state). An aggregate MDP formed this way (i.e., by replacing states with blocks) can be solved optimally, but more quickly due to the reduction in state space size. Lee and Yannikakis [51] describe an algorithm for for minimizing stochastic automata, though it relies on state space enumeration and is not directed toward decision processes. As pointed out by Dean and Givan [26], the MDP abstraction method of [30] can be viewed directly in this way, explicitly minimizing the MDP before solving it.

Of course, minimization can involve abstraction with respect to weaker properties, such as value function differences [26]. The SPI algorithm can be viewed in this light: it dynamically constructs a "minimal model" based on the current estimate of the value function. For instance, Theorem 4.4, pertaining to the stabilization of the value function *structure*, can be interpreted as confirming the discovery of a minimal model (with respect to value of a fixed policy).

5 Analysis

In this section we describe some empirical results with our structured dynamic programming algorithms. We focus on the SPI algorithm, show its performance on several problems with slightly different features, and attempt to characterize the types of problems for which SPI will and will not work well. Some of the reasons for poor performance will suggest directions for future development of MDP decomposition and abstraction techniques.

In the following, we describe a series of problems and compare the running time of SPI with that of flat (state-based) modified policy iteration (MPI). In all comparisons, we use the same number of iterations for policy evaluation or the same termination criterion for both the structured and unstructured algorithms. The MPI algorithm is optimized to exploit any sparseness in the transition matrices: sparse matrix representations are used for probability matrices and the sparseness is used to avoid "state enumeration" of zero probability states in expected value computations. In this sense, we compare SPI to the "(conceptually) best" implementation of a general-purpose unstructured algorithm. We also describe the size of the resulting structured policies and value function in terms of the number of leaves the corresponding tree contain, and compare this to the state space size.[26]

5.1 Synthetic MDPs: Best and Worst Cases

SPI was tested on two sets of synthetic MDPs designed to illustrate its performance under best-case and worst-case scenarios, as compared to unstructured MPI, which enumerates the state space explicitly. Worst-case behavior was tested on a series of MDPs whose tree-structured value function requires a full tree. Specifically, the MDP is designed so that the optimal value function has a distinct value at each state. The MDP consists of n boolean variables, $X_1, X_2, \cdots X_n$, and n deterministic actions $a_1, a_2, \cdots a_n$. A positive reward r is associated with the single state where each X_k is true, while a reward of zero is assigned to all other states. The problem is discounted with discount factor β. The kth action a_k sets the kth variable X_k true if all preceding variables $X_i \, (i < k)$ are true, otherwise it has no effect; but it also makes all preceding variables false. The DBN for the kth action is illustrated schematically in Figure 24(a).

[26] All of our results were obtained using an implementation written in C++ running under Linux on a Pentium II 400 MHz with 640MB of memory.

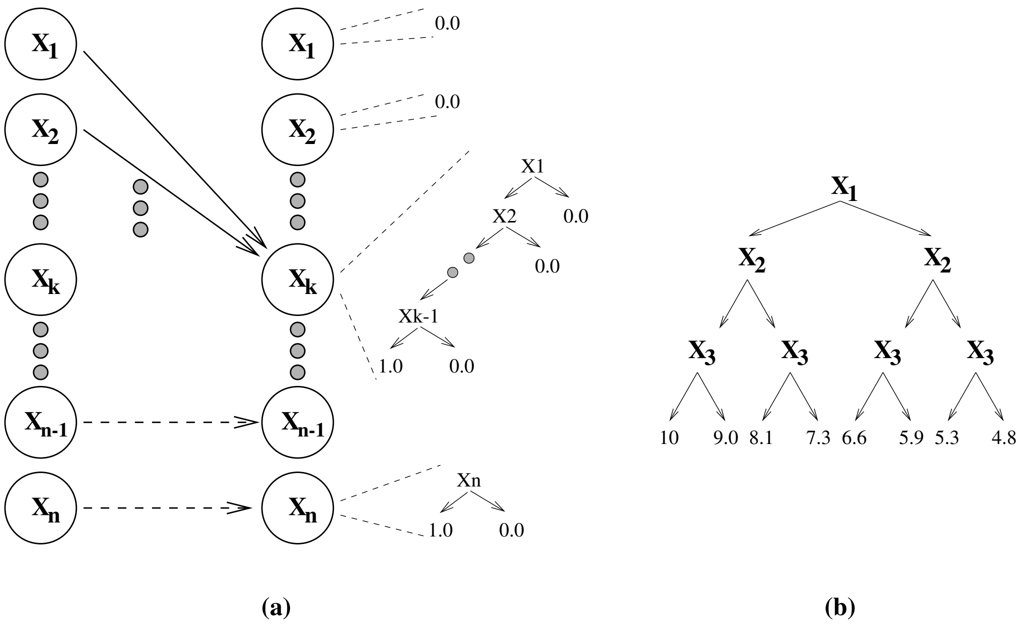


Figure 24: (a) The DBN for the kth action a_k in the worst-case examples; and (b) the worst-case value function for the three-feature ($n = 3$) version of this MDP.

If the state space is viewed as a binary number, the optimal policy requires choosing an action to set the highest bit (largest variable X_k) whose predecessors are already set. Since this sets the predecessors to false, the optimal policy induces a path from any given starting state that enumerates all binary numbers in order until the number $11 \cdots 1$ is reached (i.e., all variables are true). Because of discounting, the state corresponding to number j has value $\beta^{2^n-j-1}r$. Thus, each state has a unique value. Though this MDP can be represented using DBNs and decision trees in $O(n^2)$ space, its value function requires $O(2^n)$ space when represented as a decision tree.[27] An example of the optimal value function with three variables ($n = 3, r = 10, \beta = 0.9$) is illustrated in Figure 24(b). This, then, represents something of a "worst case" for SPI: it must enumerate the entire state space, exactly like MPI, yet pays the additional overhead associated with constructing trees before doing the expected value calculations.

Figure 25 compares the performance of the SPI algorithm with unstructured MPI on a series of four worst-case problems with six to twelve variables (64 to 4096 states).[28] From the plot on the right of Figure 25 we see that the overhead associated with the SPI algorithm causes the algorithm to run roughly one hundred times slower than the corresponding flat dynamic programming algorithm. The roughly constant

[27]This isn't to say the value function can't be represented compactly in some other way: the functional expression above offers a compact representation!

[28]The data corresponding to all of the problems described in Section 5 can be found at the SPI web site, `http://www.cs.ubc.ca/spider/dearden/spi.html`.

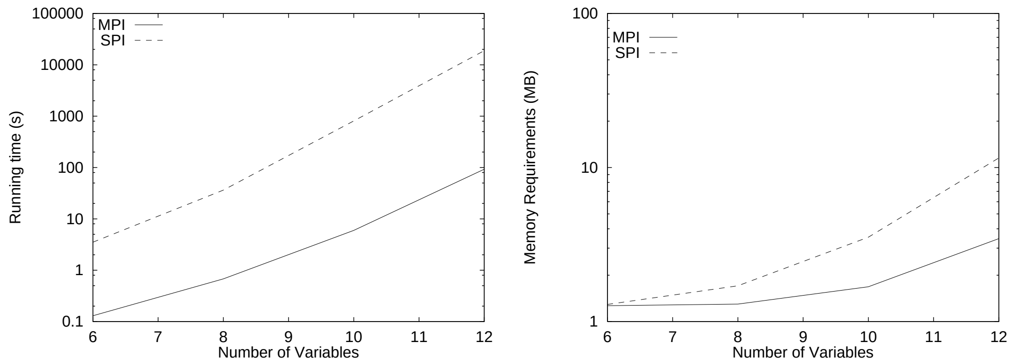


Figure 25: Time and space performance for SPI and MPI on the worst-case series

overhead is to be expected given that the *complete* tree representing a value function is roughly twice as large as the corresponding flat (tabular) value function. Thus the number of operations required to construct the tree-structured value function is bounded by the size of the tree and the number of operations required to traverse (partial) trees while building it.[29]

Note that as the number of variables in this type of problem increases, the space required to compute the optimal policy increases at a significantly greater rate with SPI, as seen in the plot on the right of Figure 25. Again, this is due to the fact that a tree representation of the value function requires storage of interior nodes in the value tree.

While the overhead for SPI is quite large in the worst-case examples, such examples are designed in an adversarial fashion to illustrate the worst-case. The constant-factor overhead in computation time may not be a serious price to pay if worst-case bahavior is unlikely to arise in practice, as long as there are benefits in the best or typical cases. We have designed a different set of abstract, synthetic examples to illustrate "best-case" behavior. The best-case examples are designed so that the optimal value function, when represented as a tree, has size linear in the number of problem variables; specifically, each variable occurs exactly once in the tree. This is "best-case" in the sense that no problem in which all variables play a role in the final value function can be smaller.[30]

Each best-case MDP consists of n boolean variables, $X_1, X_2, \cdots X_n$, and n deterministic actions $a_1, a_2, \cdots a_n$. A positive reward r is associated with the single state where each X_k is true, while a reward of zero is assigned to all other states, much like the worst-case problems. The problem is again discounted with discount factor β. The kth action a_k sets the kth variable X_k true if all preceding variables $X_i (i < k)$ are true, otherwise it has no effect; but it also makes all succeeding variables $X_j (j > k)$ false. The DBN for the kth

[29]The slight separation of the log plots as the number of variables increases is due to some slight inefficiencies in our prototype implementation, not due to the decision-theoretic regression approach itself. We discuss this further at the end of this section.

[30]Of course, if there are completely irrelevant variables, SPI will recognize this as have an even greater advantage over unstructured algorithms, as we discuss below.

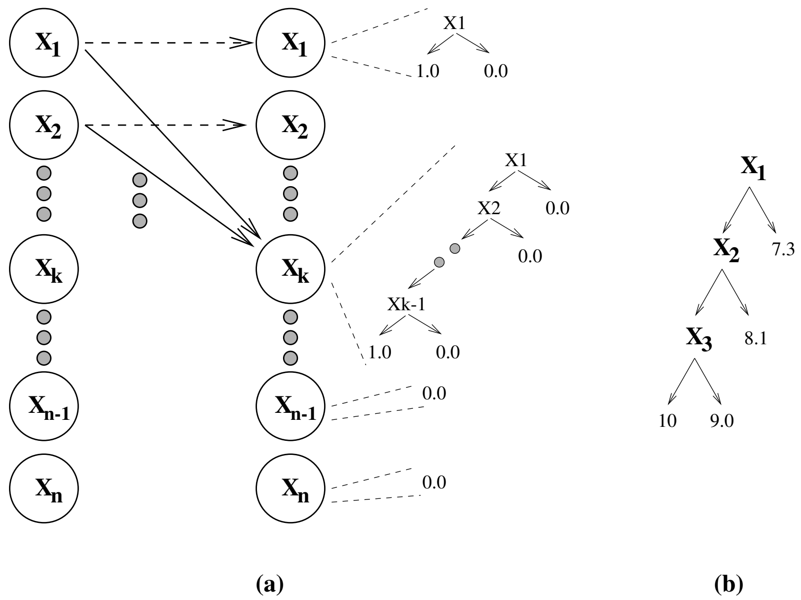


Figure 26: (a) The DBN for the kth action a_k in the best-case examples; and (b) the best-case value function for the three-feature ($n = 3$) version of this MDP.

action is illustrated schematically in Figure 26(a).

If the state space is viewed as a binary number, the optimal policy requires choosing an action to set the highest bit (largest variable X_k) whose predecessors are already set. This turns off higher bits, each of which must in turn be set by subsequent actions. Thus for any state whose lowest false variable is X_k, a sequence of $n - k + 1$ actions (setting variables X_k through X_n) is required to reach the goal; the value of such a state is $\beta^{n-k+1}r$. Thus the value function contains only $n + 1$ distinct values and is represented as a tree with n internal nodes, one for each variable. An example of the optimal value function with three variables ($n = 3, r = 10, \beta = 0.9$) is illustrated in Figure 26(b). This MDP can be represented using DBNs and decision trees in $O(n^2)$ space, and its value function requires $O(n)$ space.

Figure 27 shows a comparison of SPI and MPI on this series of examples, ranging from 6 to 20 variables. As expected, the time and space requirements for MPI grow exponentially with number of variables (and is unable to solve the 20 variable problem due to memory demands), while SPI outperforms MPI considerably with respect to both time and space. For example, SPI solves the 18-variable problem in 1.4 seconds, while MPI requires 2923 seconds to solve the same problem.

We note that it is largely the inherent structure of the problems above that dictates the *differences* in performance between SPI and MPI. While the problems are deterministic, the performance differences are virtually identical when noise of various types is added to both the best- and worst-case problems. We illustrate this with one simple form of noise (though similar phenomena arise with other noise models). In

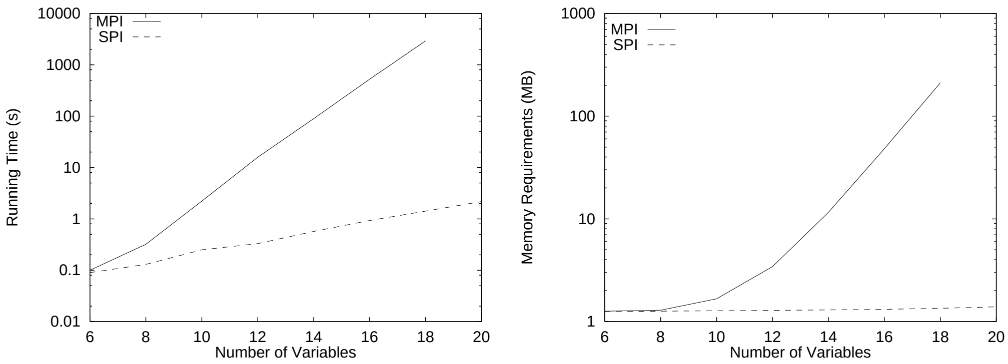


Figure 27: Time and space performance for SPI and MPI on the best-case series

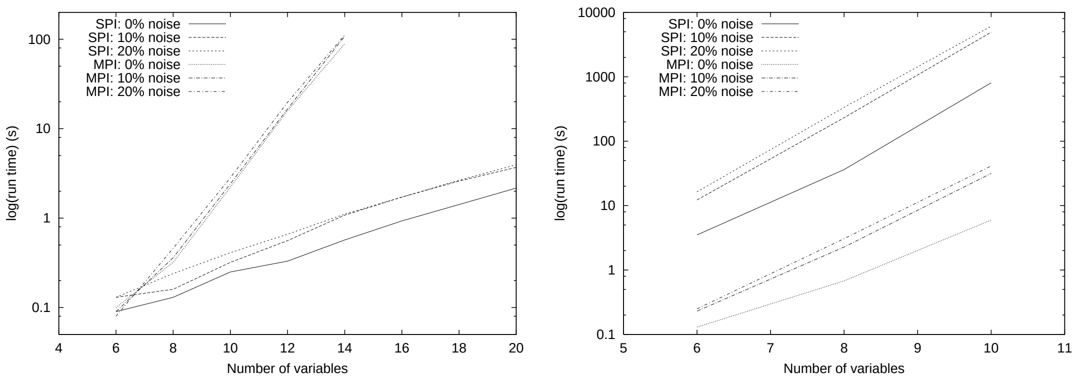


Figure 28: Solution time for SPI and MPI with various noise levels, on best-case problems (left) and worst-case problems (right).

our worst-case problems, a $k\%$ noise level indicates a $k\%$ chance that action a_k fails to set variable X_k true (otherwise the effect is exactly as above). In the best-case problems, a $k\%$ noise level indicates a $k\%$ chance that action a_k will make the variable X_{k-1} false (otherwise the effect is unchanged). In the noisy variants of these problems, the structure of the value function and optimal policy is identical; but noise generally causes longer convergence times. Figure 28 compares the effect of noise for $k = 10$ and $k = 20$, on both SPI and MPI, on best- and worst-case examples of various sizes. While noise makes both types of problems harder to solve, the increase in difficulty is no worse for SPI than for unstructured MPI.

Problem	Algorithm	States	Actions	SPI Leaves	Time (s)	Memory (M)
Manufacturing1	MPI	55,296	14		349	78
	SPI			5786	650	22
Manufacturing2	MPI	221,184	14		1775	160
	SPI			14,117	1969	54
Manufacturing3	MPI	1,769,472	14		>9000	-
	SPI			40,278	5155	129

Figure 29: Comparison of SPI and MPI on process-planning problems

5.2 Process-Planning Problems

The results above illustrate the extreme points of SPI's performance. To test more "typical case" performance, we ran SPI on a set of process-planning problems from a synthetic manufacturing domain. These domains are based on a manufacturing problem in which a product is produced by attaching finished component parts. The parts must be milled, polished, painted, and attached by either bolting or gluing them together. There are two types of finished product, high-quality and low-quality, and the policies for producing then are quite different. For example, high-quality products should be hand-painted, drilled and bolted together, which requires skilled labour, a drill-press, and a supply of bolts. This process is too expensive for producing low-quality products, which should be spray-painted and glued, thus requiring a spray gun, glue, and clamps. The reward function is designed to capture the need for high-quality versus low-quality products: specifically, if high-quality is required, then little reward is given for producing a low-quality product; and if low-quality is all that is necessary, high-quality production is given a small added reward, but generally not enough to pay for the cost of producing high-quality.

Figure 29 shows the comparison of SPI and MPI on three such problems (again, full descriptions can be found at the Web site mentioned above), with size ranging from 55,000 states to 1.8 million states. In the largest of these problems, MPI is unable to run to completion due to memory limitations, but SPI solves the problem in what we extrapolate to be roughly one-third the time required by MPI. In the smallest problem, there is not enough structure in the value function to permit the tree-construction overhead of SPI to pay off. In the medium-sized problem the methods are roughly comparable with respect to solution time. Because of SPI's ability to ignore irrelevant variables, new variables can be added to these problems that have no impact on optimal actions, thus increasing state space size without having anything but a trivial impact on SPI's running time. Each new boolean variable added to the problem effectively doubles the running time of MPI.

Notice that SPI discovers considerable regularity in the value functions for these problems. For instance, in the largest problem SPI produces a value function tree with roughly 40,278 distinct leaves, thus discovering that there are only (no more than) that many distinct values in the optimal value function among the

1.8 million states.[31] This regularity has a strong impact on the memory requirements for SPI, which are considerably lower than those for MPI.

A key feature of this problem class that allows SPI to perform well is the fact that in certain parts of the state space certain variables are completely irrelevant to the prediction of value. For instance, if high-quality products are required, then a number of variables, such as the availability of glue, are irrelevant to value function prediction. Similarly, low-quality products do not require the availability of skilled labour. This is where SPI gains its computational and space advantages, since it discovers this *conditional irrelevance*, effectively abstracting the state space by ignoring certain variables conditional on other variables taking certain values. This type of irrelevance will hold in many types of domains. For example, in any domain where there are several methods of achieving various objectives, but only one should be chosen under any specific set of circumstances, the variables relevant to the execution of those methods that are not optimal will be ignored by SPI (under the given circumstances).

5.3 Taskable Robot Problems and Exogenous Events

We have also run SPI on more elaborate versions of the robot coffee-delivery scenario. We report on these here because they point out certain problems for SPI when dealing with event-driven processes. More specifically, they point out conditions under which the type of "irrelevance" exploited by SPI is less likely to exist.

The problem domain is one in which the robot can move among five different locations, can pick up and delivery coffee, can pick up and deliver mail and can tidy a lab. Penalties are imposed in states where there is an outstanding request for coffee, there is undelivered mail, or the lab is untidy (there are several degrees of untidiness).[32] The problem is not designed to have any irrelevant features—for example, all features relating to a specific objective are relevant to the value function if that objective needs to be filled. Thus there are few irrelevant details that can be exploited by SPI. However, one feature of the problem makes the problem especially difficult for SPI, namely, the presence of *exogenous events*.

Without exogenous events to drive the process, at any state there is some subset of the three objectives that needs to be satisfied. However, once the objective is satisfied, it will never need to be considered again; and if the objective was not relevant it will never become relevant. For instance, if there is no coffee request outstanding in the initial state, no future coffee requests will be issued and no variables relevant to coffee delivery are needed to predict value. In such an MDP, under the optimal policy the robot will reach an absorbing state (or class of states) where all objectives are satisfied and never need to be considered again. A more realistic version of the problem contains exogenous events that continually drive the robot to achieve objectives that arise over time. For example, even if there is no outstanding coffee request, a coffee request event can occur with some small probability. This requires that the robot constantly assess it ability to deliver coffee, even when in states where no coffee request is outstanding. Similarly, the realistic model contains

[31] The optimal policy tree is much smaller than the value function tree in all problems we consider.

[32] This domain is described in some detail in [11] and can be found at the Web site mentioned above.

exogenous events that cause mail to arrive and the lab to become messier.

The problem domain has six variables (four of which are five-valued, hence 400 states) and eight actions. Without exogenous events, SPI runs to completion in 11.9 seconds, producing a final value tree with 291 leaves and a policy tree with 196 leaves, a requires 1.85Mb of memory. Notice that the value tree does not contain significantly fewer entries than the flat, tabular representation. This is because all variables are relevant under most circumstances. For comparison, MPI runs in 0.31 seconds and requires 1.5Mb of memory.

When we add the three exogenous events to the domain, SPI produces slightly larger value and policy trees with 300 and 219 leaves, respectively, and requires 2.0Mb of memory. Though not substantially larger than without exogenous events, the trees become larger because variables associated with various tasks are now relevant even in states where the task is not "active." For example, in a state with no outstanding coffee requests, the variables relevant to coffee delivery are now relevant to predicting value—this is because the exogenous "coffee request" event could, at some future point, make the task active, and the speed with which it is accomplished depends on the status of coffee-delivery variables. Thus, exogenous events tend to make trees larger by rendering variables relevant because of *future possibilities*.[33] Even worse, SPI takes nearly ten times as long (100.6 seconds) to run in the presence of the three exogenous events. This is not so much because the final trees are much larger, but because the value trees produced in *early* phases on policy iteration get much more complex; that is, the trees get larger earlier.

We note that these exogenous events cause difficulty primarily in cases where all variables are relevant to the (optimal) performance of *some* task, and where these tasks can each arise at any time. When certain problem variables are irrelevant—for example, because we discover, while solving the MDP, that they are irrelevant to suboptimal (hence, unselected) methods of task achievement under certain conditions—SPI still discovers these irrelevancies and can take advantage in the usual way, even in the presence of exogenous events. Although exogenous events do increase the degree of (stochastic) connectedness of an MDP, this is not the primary contributor to the difficulties faced by SPI in such domains. Rather it is the fact that the complexity of the "abstract state description" required to predict the value function and optimal action choice can depend on variables relevant to *any* task that *could* arise. This suggests that a form of task descomposition could be used to help alleviate these difficulties (we return to this possibility in Section 7).

5.4 Discussion

In a some loose sense, SPI can be viewed as preserving as much structure in the value function representation as possible, subject to certain restrictions. For example, given the DBN for action a and the tree representation of value function V, the regression of V through a will produce a regressed tree that makes distinctions that could *all* be necessary given the *structure* of the inputs V and a. Whether the distinctions

[33] The effect on tree size is not dramatic in this example, but it is very easy to construct realistic scenarios where the effect is considerably more severe.

actually *are* necessary depends in large part on the specific values and probabilities labeling the leaves of the input structures. But SPI produces output of minimal size for an algorithm that use only the *structure* of input trees to make abstraction decisions.

Of course, the use of trees restricts how compactly certainly reward functions, value functions, and CPTs can be represented. The smallest tree representation of a given value function may be exponentially larger than the smallest representation using some other technique (like an ordered decision diagram, which can handle disjunction much more effectively, or a decision list, or a set of Horn clauses). Variable ordering also plays an important role in just how small a decision tree is. Since no representation can represent all polynomial-sized functions (i.e., those with only polynomially-many distinct values) over a set of variables compactly (i.e., with polynomial size), the potential blowup is unavoidable. Furthermore, no representation is universally more compact than another; for instance, with some functions the best decision tree will be exponentially smaller than the best ordered decision diagram, and for others it will be exponentially larger. The choice of appropriate representation will generally depend on the structure of a given domain. We conjecture that decision trees offer a suitable choice for many problems. However, the basic conception of decision-theoretic regression can be applied to any representation: one simply needs algorithms to manipulate that representation, as in the region-based approach of Dietterich and Flann [32, 33] or the decision diagram model of Hoey *et al.* [43].

The empirical results above suggest some possible directions for enhancing SPI and suggest conditions under which SPI may and may not work well. Results on the process-planning domain suggest that the overhead associated with SPI will pay off if we are able to eliminate the equivalent of roughly four to five variables from the description of the value function. That is the tree representation of the value function should have 15 to 30 times fewer leaves than there are states of the system. We expect that for large problems such a reduction factor is very easy to obtain. Note that we refer only to computation time above; SPI offers more dramatic savings in memory usage even when the time savings are minimal.

This time reduction estimate is based on the simple implementation described here. We notice that the overhead in the worst-case examples is roughly constant at first glance. However, upon closer examination, the overhead factor is increasing slowly with problem size. This suggests that the implementation tested has certain inefficiencies. It also suggests that improved algorithms for manipulating the structured representations of value functions and policies could greatly improve the applicability of SPI. Both of these facts have been confirmed in subsequent work [43] that extends SPI using algebraic decision diagrams (ADDs) [2]. This improved structured representation and implementation (SPUDD) has been tested on the problems described above and has proven the benefit of decision-theoretic regression to be more substantial than suggested here.[34] For example, SPUDD solves the 12-variable worst-case problem in just over 1500 seconds, reducing worst-case overhead to a factor of 15 (from a factor of better than 100 with SPI), and actually shows a decrease in overhead factor with problem size. On the largest process-planning problem (with 1.8 million

[34]SPUDD is based on structured value iteration rather than modified policy iteration; a version based on modified policy iteration would show even better performance.

states), SPUDD runs in 462 seconds, 12 times faster than SPI (and we conjecture about 35-40 times faster than MPI). In the smaller process-planning examples, where SPI fails to beat MPI, SPUDD runs much faster than MPI as well (in 78 seconds and 111 seconds, compared to 349 seconds and 1775 seconds for MPI). We take these results to confirm the intuition that decision-theoretic regression can pay off even with *much* smaller variable reduction factors.

The difficulty with exogenous events is something that cannot be addressed directly within the SPI model. There are two methods we can use to deal with this however. The first is to use a form of approximation. We discuss approximation withing the SPI framework in the following section; but we simply note here that it may be a suitable way to handle the specific problem with exogenous events in certain situations. If the events have reasonably small probability, knowledge of variables relevant to the corresponding objective will have a small impact on value. The approximation scheme outlined later can ignore such distinctions.

This problem arises in the taskable robot domain largely because there are multiple objectives that may be simultaneously active (or may become active in the future). One way to deal with this problem is to treat the different objectives separately and construct optimal policies or value functions for the *individual objectives*. This might be appropriate in the domain described above since each objective makes an independent contribution to the reward function. A deeper discussion of such a model takes us beyond the scope of this paper, and SPI specifically. However, we note that there have been several models of MDPs that exploit this type of independence [9, 56, 71]. Decision-theoretic regression methods such as SPI can be used in conjunction with such techniques to great effect since they are largely orthogonal.

6 Extensions of the Basic Algorithm

6.1 Handling Synchronic Constraints

The key operation defined in Section 4, namely the decision-theoretic regression operator *Regress*$(\mathit{Tree}(V), a)$, was justified by assuming that the effects an action has on different post-action variables are independent. Specifically, the joint distributions produced by the algorithm *PRegress*$(\mathit{Tree}(V), a)$ (see Figure 13) are product distributions. These independence assumptions are valid in DBNs without synchronic arcs (arcs between post-action variables); this fact in used in the proof of Theorem 4.1.

Unfortunately, this independence assumption no longer holds when action networks have synchronic arcs. In a network representing an action a like that shown in Figure 30(a), the effect of a on variable Y^{t+1} is not independent of its effect on X^{t+1} given the previous state. This causes two distinct problems for decision-theoretic regression.

Regression involves computing $\mathit{Tree}(Q_a^V) = \mathit{Regress}(\mathit{Tree}(V), a)$, where $\mathit{Tree}(Q_a^V)$ denotes the value of executing a with $k+1$ stages-to-go assuming V represents k-stage-to-go value. The first problem that occurs in our standard regression algorithm is a result of the fact that it pieces together CPT-trees from the DBN for a for each of the variables occuring in $\mathit{Tree}(V)$. When there are synchronic constraints, these CPT-trees

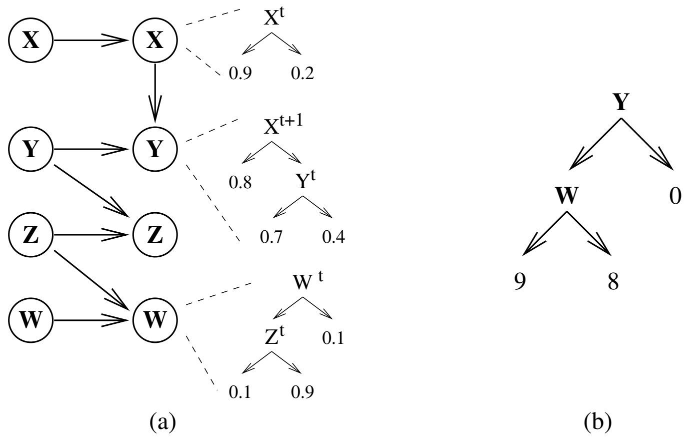


Figure 30: (a) An action network with synchronic arcs denoting a correlation between the effects on variables X and Y; (b) An example value tree over the same domain

may have post-action variables occurring in them (e.g., X^{t+1} occurs in the CPT for Y^{t+1} in Figure 30(a)), leading to the occurrence of post-action variables in the tree *Tree*(Q_a^V). This means that *Tree*(Q_a^V) no longer represents the value of executing a with $k+1$-stages-to-go as a function of the state at that time (it now refers to properties of the state with k-stages-to-go). This can be fixed rather easily by summing out the influence of X^{t+1}, replacing the dependence of Y^{t+1} on X^{t+1} with a direct dependence on the parents of X^{t+1}.

The second problem that occurs when constructing *Tree*(Q_a^V) from *Tree*(V) arises because the effect of a on the variables occuring in *Tree*(V) may be correlated. For example, if the variables X and Y both occur on a single branch of *Tree*(V), the probability of attaining the value in that branch (using action a from Figure 30) cannot be specified by the independent probabilities of making X true and making Y true. Unlike our earlier algorithm, where the lack of synchronic arcs ensured independence, we must keep track of the *joint distribution* over X^{t+1} and Y^{t+1} explicitly when constructing *Tree*(Q_a^V).

We illustrate how one deals with these issues by means of simple examples, and describe the intuitions needed to extend our decision theoretic regression algorithm to deal with synchronic constraints. We do not provide a formal algorithm, or proof of correctness; instead we refer to [8] for a more detailed description of the necessary amendments.

6.1.1 Summing out Post-Action Influences

Consider action a in Figure 30(a) and the example value tree *Tree*(V) in Figure 30(b). Using the algorithm *PRegress*(*Tree*$(V), a)$ from Section 4 to produce *Tree*(Q_a^V), we would first regress Y through a to obtain the tree shown in Figure 31(a). Continuation of the algorithm will not lead to a legitimate Q-tree, since it involves a post-action variable X^{t+1}. Our revised algorithm will establish the dependence of $\Pr(Y^{t+1})$ on the previous state s by "summing out" the influence of X^{t+1} on Y^{t+1}, letting the probability of Y^{t+1}

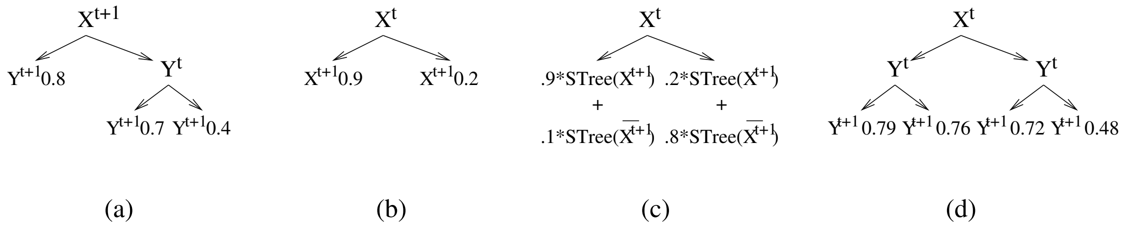


Figure 31: Summing out the influence of post-action parents: (a) The (partial) tree obtained by the original *PRegress* algorithm; (b) The effect of a on X^{t+1}; (c) A conceptual view of how the influence of X^{t+1} is summed out; and (d) The resulting (partial) *PRegress*-tree.

depend directly on the *parents* of X^{t+1} instead of on X^{t+1} itself. Specifically, we compute

$$\Pr(Y^{t+1}|s) = \sum_{x' \in val(X^{t+1})} \Pr(Y^{t+1}|x', s) \cdot \Pr(x'|X^t) = \sum_{x' \in val(X^{t+1})} \Pr(Y^{t+1}|x', Y^t) \cdot \Pr(x'|X^t)$$

This computation can exploit the tree structure as follows. Once we have regressed Y through a, we will replace the node X^{t+1} by the tree representing *CPT*(X, a). This dictates $\Pr(X^{t+1}|\Pi(X^{t+1}))$, and this tree is duplicated in Figure 31(b). Denote the subtree of the replaced node corresponding to each value x_i of X^{t+1} by *STree*(x_i). In Figure 31(a), *STree*(x') is the single leftmost leaf node, while *STree*$(\overline{x}')$ is the right subtree rooted at variable Y. At each leaf l of *CPT*(X, a), we have the label $\Pr(x_i)$. For those values of x_i that have positive probability, we merge the trees *STree*(x_i) and copy these at l. Specifically, the merge operation proceeds as illustrated in Figure 31(c): we weight each subtree *STree*(x_i) by $\Pr(x_i)$ labeling the leaf of *CPT*(X, a), and merge these weighted subtrees using addition as the combination operation. The resulting tree, shown in Figure 31(d), shows the probability of Y^{t+1} as a function of the previous state only, with dependence on X^{t+1} removed completely. Once completed, it is easy to see that regression of W through a can proceed unhindered as in Section 4.

We now consider a second example (see Figure 32) that illustrates that the order in which these post-action variables are replaced in a tree can be crucial. Suppose that we have an action a similar to the one just described, except now we have that variable Y^{t+1} depends on both X^{t+1} and Z^{t+1} (i.e., a's effect on X, Y and Z is correlated). When we regress Y through a, we will introduce a tree in which both X^{t+1} and Z^{t+1} appear, and we assume that X^{t+1} and Z^{t+1} appear together on at least one branch of *CPT*(Y, a) that is present in *Tree*(Q_a^V). Now let us suppose that Z^{t+1} also depends on X^{t+1}, as in Figure 32. In such a case, it is important to substitute *CPT*(Z, a) for Z^{t+1} before substituting *CPT*(X, a) for X^{t+1}. If we replace X^{t+1} first, we will compute

$$\Pr(Y^{t+1}|Z^{t+1}, \Pi(X^{t+1})) = \sum_{x' \in val(X^{t+1})} \Pr(Y^{t+1}|x', Z^{t+1}) \Pr(x'|\Pi(X^{t+1}))$$

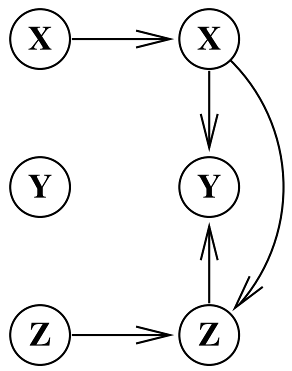


Figure 32: Ordering variables for replacement

(we suppress mention of other parents of Y^{t+1}, if any). Subsequently, we would replace occurrences of Z^{t+1} with *CPT*(Z, a) and compute

$$\Pr(Y^{t+1}|\Pi(Z^{t+1}), \Pi(X^{t+1})) = \sum_{z' \in val(Z^{t+1})} \Pr(Y^{t+1}|z', \Pi(X^{t+1})) \Pr(z'|\Pi(Z^{t+1}))$$

This ordering has two problems. First, since X^{t+1} is a parent of Z^{t+1}, this approach would reintroduce X^{t+1} into the tree, requiring the wasted computation of summing out X^{t+1} again. Even worse, for any branch of *Tree*(Z, a) on which X^{t+1} occurs, the computation above is not valid, for Y^{t+1} is not independent of X^{t+1} (an element of $\Pi(Z^{t+1})$) given Z^{t+1} and $\Pi(X^{t+1})$ (since X^{t+1} directly influences Y^{t+1}).

Because of this, we require that when a variable Y is regressed through a, if any two of its post-action parents lie on the same branch of *CPT*(Y, a), these variables in *CPT*(Y, a) must be replaced by their trees in an order that respects the dependence among post-action variables in a's network. More precisely, let a *post-action ordering* O_P for action a be any ordering of variables such that, if X^{t+1} is a parent of Z^{t+1}, then Z^{t+1} occurs *before* X^{t+1} in this ordering (so the ordering goes against the direction of the synchronic arcs in the DBN for a). Post-action variables in *CPT*(Y, a), or any tree obtained by recursive replacement of post-action variables, must be replaced according to some post-action ordering O_P.

6.1.2 Computing Local Joint Distributions

Consider again *Tree*(V) shown in Figure 30(b) and its regression through the action a shown in Figure 33(a). Figure 33(b) shows the first step of this regression, the regression of Y through a. The second step of the regression appends *CPT*(W, a) to each leaf of this partial tree; but since Y^{t+1} occurs in *CPT*(W, a), we replace Y^{t+1} with *CPT*(Y, a) as described in the previous subsection, resulting in a tree *PRegress*$($*Tree*$(V), a)$ that has the *structure* of the tree shown in Figure 33(c).

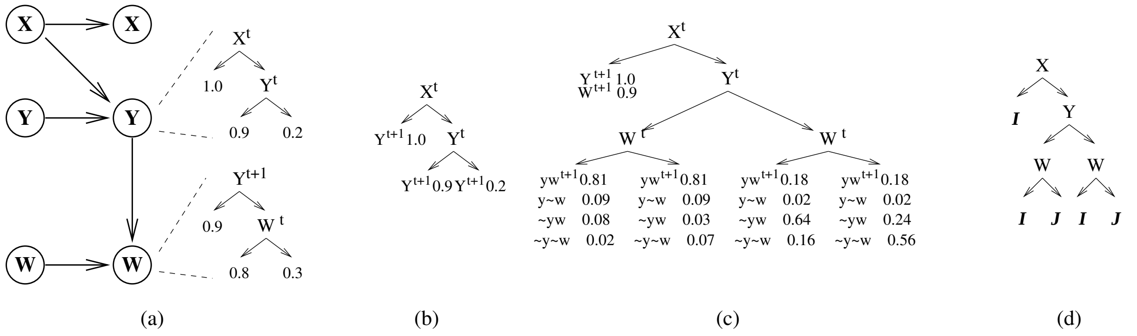


Figure 33: Capturing correlations in *PTree*(Q_a^V): (a) An action network with synchronic arcs; (b) The first stage of regression; (c) The final version of *PTree*(Q_a^V) with joint distributions over certain effects labeling the leaves; (d) An alternative structure for *PTree*(Q_a^V).

Once we have this structure, if we were to proceed as above, we would simply sum out the influence of Y^{t+1} on W^{t+1} to determine $\Pr(W^{t+1})$ at each leaf. That is, we would compute

$$\Pr(W^{t+1}|W^t, X^t, Y^t) = \sum_{y' \in val(Y^{t+1})} \Pr(W^{t+1}|y', W^t) \cdot \Pr(y'|X^t, Y^t)$$

and obtain *PTree*(Q_a^V). This, unfortunately, does not provide an accurate picture of the probability of attaining the conditions b labeling the branches of *Tree*(V). If we labeled the leaves of *PTree*(Q_a^V) in Figure 33(c) with $\Pr(Y^{t+1})$ and $\Pr(W^{t+1})$ so computed, these probabilities, while correct, are not sufficient to determine $\Pr(Y^{t+1}, W^{t+1})$: Y^{t+1} and W^{t+1} are *not independent* given X^t, Y^t and W^t. The synchronic arc between Y^{t+1} and W^{t+1} means that the effect of a on these two variables is correlated, even given knowledge of the prior state.

To ensure correct expected future values are computed when constructing *FVTree*(Q_a^V), we must instead maintain the correlation between Y^{t+1} and W^{t+1} in the construction of *PTree*(Q_a^V). To do this, we explicitly label the leaves of *PTree*(Q_a^V) with the joint distribution $\Pr(Y^{t+1}, W^{t+1})$, as shown in Figure 33(c). We note that this joint is obtained in a very simple fashion. At each leaf of *PTree*(Q_a^V), we have easy access to the labels both $\Pr(Y^{t+1})$ and $\Pr(W^{t+1}|Y^{t+1})$ (given the conditions on the previous state leading to that leaf). Instead of summing out the influence of Y^{t+1} on W^{t+1}, we explicitly store the terms $\Pr(Y^{t+1}, W^{t+1})$ we compute.[35]

This approach—explicitly representing the joint probability of different action effects instead of summing out the influence of synchronic parents—allows us to accurately capture the correlations among action

[35] We should emphasize that this local joint distribution does not need to be computed or represented explicitly. Any *factored* representation, e.g., storing directly $\Pr(Y^{t+1})$ and $\Pr(W^{t+1}|Y^{t+1})$, can be used. In fact, when a number of variables are correlated, we generally expect this to be the approach of choice. However, we will continue to speak as if the local joint were explicitly represented for ease of exposition.

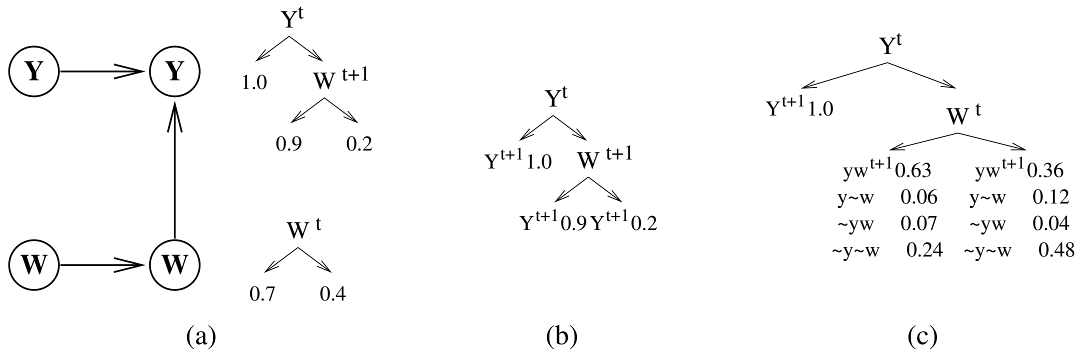


Figure 34: Deciding which correlations to record *PTree*(Q_a^V): (a) An action network with synchronic arcs; (b) The first stage of regression; (c) The final version of *PTree*(Q_a^V) with joint distributions labeling the leaves.

effects that directly impact the value function. It is important to note, however, that we need only compute the joint distribution between two relevant variables in those contexts in which they are actually correlated. For instance, suppose that we switched the locations of variables Y^{t+1} and W^t in *CPT*(W, a) in Figure 33(a). We see then that W^{t+1} only depends on Y^{t+1} when W^t is false. In this case, *PTree*(Q_a^V) would have a similar structure, but we could maintain independent estimates of $\Pr(Y^{t+1})$ and $\Pr(W^{t+1})$ at certain leaves. In particular, referring to Figure 33(d), independent distributions could be maintained at those leaves labeled I (since Y^{t+1} and W^{t+1} are independent given w^t), while joint distributions must be maintained at those leaves labeled J (since Y^{t+1} and W^{t+1} are not independent given $\overline{w}^t$). Since representing a joint distribution explicitly requires a number of parameters exponential in the number of variables involved (and is strongly impacted by the domain size of those variables), maintaining the (independent) product form of the joint wherever possible is important.

The last piece in the puzzle pertains to the decision of when to sum out a variable's influence on one of its synchronic descendents and when to retain the (local) joint representation of the distribution over the two variables. Consider again the value tree from Figure 30(b) and suppose action a has the form shown in Figure 34(a); notice that the dependence of W^{t+1} on Y^{t+1} has been reversed in this action. When regressing *Tree*(V) through a, the first stage where Y^{t+1} is regressed leads to the tree in Figure 34(b). When removing the influence of variable W^{t+1} on Y^{t+1}, we obtain the tree shown in Figure 34(c). Using the ideas above, we would be tempted to sum out the influence of W^{t+1} on Y^{t+1}, computing

$$\Pr(Y^{t+1}|Y^t, W^t) = \sum_{w' \in val(W^{t+1})} \Pr(Y^{t+1}|w', Y^t) \cdot \Pr(w'|W^t)$$

However, if we "look ahead," we see that the second stage of the regression algorithm requires us to regress W through a as well, since W also occurs in *Tree*(V). Specifically, we will have to regress W at both of the

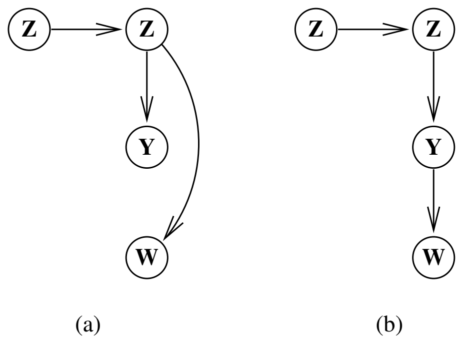


Figure 35: Detecting future need for parents

leaf nodes for which we are attempting to compute $\Pr(Y^{t+1})$ as function of the parents of W^{t+1}. Clearly, since W^{t+1} and Y^{t+1} are correlated, we should leave $\Pr(Y^{t+1})$ uncomputed (explicitly), leaving the joint representation of $\Pr(Y^{t+1}, W^{t+1})$ as shown in Figure 34(c). When subsequently regressing W^{t+1} at each leaf where $\Pr(Y^{t+1}) > 0$, our work is already done at these points.

This leads to an obvious question: when removing a post-action variable Z^{t+1} from the tree produced when regressing another variable Y^{t+1} which depends on it, under what circumstances should we sum out the influence of Z^{t+1} on Y^{t+1}, and under what circumstances should we retain the explicit joint representation of $\Pr(Z^{t+1}, Y^{t+1})$? Intuitively, we want to retain the "expansion" of Y^{t+1} in terms of Z^{t+1} (i.e., retain the joint) if we will *need* to worry about the correlation between Y^{t+1} and Z^{t+1} later on. As we saw above, this notion of *need* is easily noticed when one of the variables in directly involved in the value tree, and will be regressed explicitly afterward (under the conditions that label the current branch of course). However, variables that may be *needed* subsequently are not restricted to those that have to be regressed directly (i.e., they needn't be part of *Tree*(V)); instead, variables that *influence* those in *Tree*(V) can sometimes be retained in expanded form.

Consider the action in Figure 35(a) (we again use the same value function). When we regress Y through a, we obtain a tree containing node Z^{t+1}, which subsequently gets replaced by ***CPT***(Z, a). The term $\Pr(Y^{t+1})$ should be computed explicitly by summing the terms $\Pr(Y^{t+1}|z') \cdot \Pr(z'|Z^t)$ over values z'. However, looking at *Tree*(V), we see that W will be regressed wherever $\Pr(Y^{t+1}) > 0$, and that W^{t+1} also depends on Z^{t+1}. This means that (ignoring any specific structure in the CPT-trees) W^{t+1} and Y^{t+1} are correlated given the previous state s. This dependence is mediated by Z^{t+1}, so we will need to explicitly use the joint probability $\Pr(Y^{t+1}, Z^{t+1})$ to determine the joint probability $\Pr(Y^{t+1}, W^{t+1})$. In such a case, we say that Z^{t+1} is *needed* and we do not sum out its influence on Y^{t+1}. In an example like this, however, once we have determined $\Pr(Y^{t+1}, Z^{t+1}, W^{t+1})$ we can decide to sum out Z^{t+1} to obtain the reduced joint distribution $\Pr(Y^{t+1}, W^{t+1})$ if Z^{t+1} will not be needed further.

Finally, suppose that W^{t+1} depends indirectly on Z^{t+1}, but that this dependence is mediated by Y^{t+1}, as in Figure 35(b). In this case, we can sum out Z^{t+1} and claim that Z^{t+1} is *not needed*: Z^{t+1} can only influence W^{t+1} through its effect on Y^{t+1}. This effect is adequately summarized by $\Pr(Y^{t+1}|Z^t)$; and the terms $\Pr(Y^{t+1}, Z^{t+1}|Z^t)$ are not needed to compute $\Pr(Y^{t+1}, W^{t+1}|Z^t)$ since W^{t+1} and Z^{t+1} are independent given Y^{t+1}.

These considerations are formalized in detail in [8]. Specifically, there we provide the formal definitions and algorithms needed to operationalize the intuitions described in this section.

The use of actions with correlated effects leads to two difficulties. First, the overhead of tree construction is increased. Essentially, certain minimal probabilistic inference must be performed in order to accurately predict the effects of an action and to compute expected future value of performing an action with respect to a given value function. The second difficulty lies in maintaining the *PTree*s themselves. Certain leaves of these trees must be labeled by explicit (local) joint distributions. However, two considerations suggest that this may not be problematic in practice. The first is the possibility that these joint distributions can themselves be factored in certain ways and computed as needed. The second lies in the fact that while many actions will exhibit correlations in their effects, these correlations tend to involve a small number of variables. Our algorithm requires only that a joint be maintained over variables that are actually correlated. It is clear, however, that practical experience is needed with this algorithm before a realistic assessment can be made.

6.2 Approximation within SVI

One advantage of using decision trees to structure value functions is the ease with which one can specify approximation schemes. The tree $\mathit{Tree}(V)$ representing a value function V reflects all conditions relevant to differences in value at different states. However, some of these distinctions may have a small impact on value. That is, certain leaves of the tree may correspond to (clusters of) states whose values differ only marginally. For example, referring to the optimal value tree produced by SPI in our running example (Figure 22), we see that the states abstracted by the three branches of the tree $\langle \overline{\mathit{HCO}}, \mathit{HCR}, O, \overline{W}, R, U\rangle$, $\langle \overline{\mathit{HCO}}, \mathit{HCR}, O, \overline{W}, R, \overline{U}\rangle$, and $\langle \overline{\mathit{HCO}}, \mathit{HCR}, O, \overline{W}, \overline{R}\rangle$ all have values that differ by at most 0.09 (as compared to the total range of values of 5.19 to 10.0). If we include the fourth branch $\langle \overline{\mathit{HCO}}, \mathit{HCR}, O, W\rangle$, the values differ by at most 1.0. Tree-structured value functions make it easy to detect such regions of *similar* value.

If we are willing to live with a certain amount of approximation error, a value tree can be made smaller by *pruning* the tree in order to coalesce regions of similar value. More precisely, by replacing a subtree whose leaves are labeled with value all within some small factor δ of one another with a single leaf, we obtain an approximation of the original value tree, but one which is (perhaps considerably) smaller.

There are several ways to label the leaves of a pruned value tree. We could label each leaf with the (possibly weighted) average of the values within the subtree it replaced, or possibly with the midpoint of the range of values it replaced. In our work, we have opted to label these leaves with the *range* of values

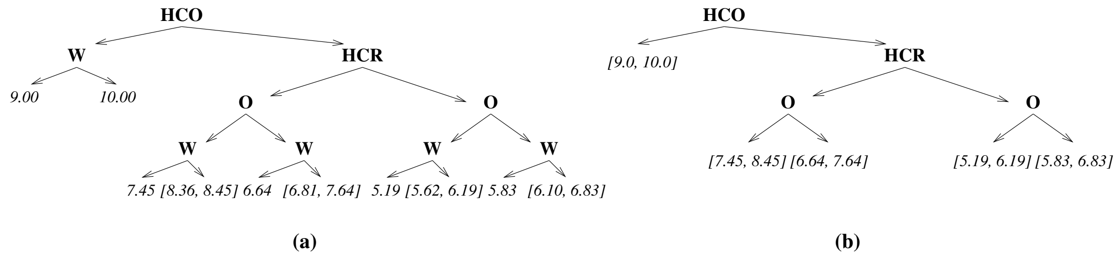


Figure 36: Two ranged value trees: (a) A more cautious pruning; and (b) a more aggressive pruning.

itself. Specifically, if a subtree is pruned, the replacing leaf is labeled with the maximum and minimum values labeling the leaves of the pruned subtree. Two such *ranged value trees*, or R-trees, are illustrated in Figure 36. The first tree corresponds to pruning the tree of Figure 22 by removing all distinctions referring to variables R and U (i.e., removing subtrees rooted at R). The maximum range of values at any leaf is 0.83, thus little error can be introduced by acting "optimally" with respect to the pruned value function. The second tree is produced by the more aggressive pruning of all subtrees rooted at variable W, giving a smaller tree with slightly larger ranges at the leaves.

There is no immediate computational utility in constructing a value tree and then pruning it—all of the computational effort has already been expended to construct a larger tree. However, if the pruned tree is one is a sequence of value trees constructed by, say, structured successive approximation or structured value iteration (SVI), this pruning can be computationally beneficial. For instance, suppose we use SVI to solve a structured MDP and *Tree*(V^k) has been generated. By pruning *Tree*(V^k) to obtain an approximate version of V^k, say *Tree*$(\widetilde{V}^k)$, the pruned tree will be smaller, containing fewer interior nodes. Subsequent decision-theoretic regression used to generate *Tree*$(\widetilde{V}^{k+1})$—the approximate version of V^{k+1} generated from the approximation of V^k—will proceed more quickly due to the fact that the tree being regressed in smaller.

In [13] we develop an approximate variant of SVI in which each value tree in the sequence produced by SVI is pruned before the next tree in the sequence in constructed. The result is an algorithm that solves structured MDPs approximately but generally does so considerably more efficiently than exact SVI or SPI. The value functions that are regressed are, in fact, ranged trees, not just simple value trees. The basic operations defined in Section 4 are extended to deal with value ranges: backing up maximum and minimum values through the basic *Regress* operator and merging R-trees to keep track of upper and lower bounds. The result is an algorithm that produces a sequence of R-trees with a guarantee that the true value at any state lies within the range labeling the appropriate leaf of the R-tree.

Apart from extending the algorithms to deal with value ranges, a number of other issues must be dealt with to satisfactorily implement such an approximate version of SVI [13].

1. We must decide how best to prune a ranged tree. We may opt for the most accurate pruned tree of some fixed maximum size or the smallest pruned tree of a fixed minimum accuracy. In [13] we present an

algorithm for finding the optimal pruning sequence for a given R-tree; that is, an algorithm in which each pruning step introduces the least error. This allows one to adopt whatever pruning criterion is most suitable. The problem is strongly related to work on pruning decision trees in classification, and our algorithm draws ideas from the work of Bohanec and Bratko [7].[36] We are able to provide error bounds for the algorithm as well in a way that allows on-line, anytime tradeoffs between tree size and solution quality [13].[37]

2. The ability to prune is strongly influenced by the variable ordering in the tree. Again this issue arises in research on classification [64, 82]. Finding the smallest decision tree representing a given function is NP-hard [46], but in [13] we discuss certain feasible heuristics suitable for reordering an R-tree to make it smaller and/or more amenable to pruning.

3. Termination of SVI requires care when approximations are introduced. While value iteration is assured to terminate due to the contraction property of the Bellman backup operator, this property fails to hold when approximations are introduced (in fact, we can easily construct examples where pruning with *midpoint* replacement causes nontermination). Fortunately, since the ranged value trees we construct are guaranteed to bracket the true value function, we can adopt rather cautious stopping criteria based on the closeness of the ranges.

We refer to [13] for further details on approximation within these structured decision-theoretic regression operations. There we discuss these issues in more detail, present the various algorithms, describe results on error bounds, and provide empirical evidence suggesting that approximate SVI can provide significant computational savings over SVI, SPI and standard dynamic programming techniques with minimal introduction of error in a variety of problems. For instance, on the worst-case MDPs described in Section 5.1, approximate SVI provides significant savings over exact SVI with very little introduction of error, at many levels of pruning. As one example, on the 10-variable worst-case domain approximate SVI at a cautious level of pruning solves the problem in less than 1% of the time required by SVI (roughly the same amount of time required by MPI), but introduces an average error in the value function of under 0.47%. At a more aggressive pruning level it solves the problem in less than 0.01% of the time required by SVI (roughly one one-hundredth the time required by MPI), yet introduces an average error of 0.77%. Similar results obtain on other problems, such as the taskable robot problems (with and without exogenous events).

7 Concluding Remarks

We have proposed the notion of *decision-theoretic regression* as a generalization of classical regression planning that deals with stochastic domains with conflicting objectives. Viewed as a form of state-space

[36] This work is concerned with pruning for the sake of simplifying the resulting decision tree with little loss in accuracy, in contrast to pruning for the purpose of preventing overfitting [64].

[37] The approximation is thus careful enough to avoid the problems of approximation described in [18].

abstraction, decision-theoretic regression groups together states that have identical value or policy choice at various points in the dynamic programming computations required to solve an MDP. We have designed a specific decision-theoretic regression operator that works with DBNs and decision trees representing transition and reward functions and that uses decision trees to represent value functions and policies. This operation exploits uniformity in the value function and policy, specifically, the fact that certain variables—under certain conditions—are not relevant to the optimal choice of action or the prediction of value.

Our SPI algorithm was shown to offer some significant advantages in certain problems, both in terms of time and space requirements, compared to unstructured dynamic programming. We also described some problems where the overhead of SPI fails to pay off. Generally, speaking, the larger the problem the more likely that overhead of tree construction associated with SPI will be more than compensated by the reduction in the number of expected value and maximization computations induced by abstraction. We discussed certain problem properties that are likely to benefit SPI, with our best- and worst-case examples giving us some sense of the boundaries of performance. Even in the worst case, SPI's overhead is not overwhelming. Our tests suggest that the overhead will be compensated by (the equivalent of) the removal of only a few variables. Finally, SPI lends itself readily to approximation, which offers additional computational benefits—often with only a small introduction of error—providing the ability to construct error bounds that can be used to make anytime computational decisions.

We note that decision-theoretic regression is a general concept whose applicability is not restricted to decision-tree representations of transition functions, value functions and the like. The same principles apply to any structured representation as long as one can develop a suitable regression operator for that representation. To wit, the SPUDD system [43] applies the same decision-theoretic regression techniques to the solution of MDPs by value iteration, but does so using algebraic decision diagrams [2] to represent inputs and output. Because these representations are often more compact than decision trees, the performance of SPUDD is considerably better than that of SPI; but it adopts the same general conceptualization of the problem described here. We note that much SPUDD's improved performance is due to the use of optimized code. On the worst-case examples, SPUDD outperforms SPI from two- to twelve-fold (with larger performance differences on larger problems). This occurs despite the fact the the decision diagram representation of the value functions in the worst-case problem set is exactly a full decision tree. This provides further evidence of the utility of decision-theoretic regression.

There are many interesting directions in which the work described here can be extended. One is the integration of decision-theoretic regression with other concepts that can be used to solve MDPs effectively. This includes the use of reachability analysis, other abstraction methods, and other structured value function representations (e.g., those that support some type of functional decomposition of the value function such as neural networks [6, 80] or additive structure [9, 31, 37, 47, 56, 72, 71]). This should prove possible because the structure assumed by SPI can be exploited in a way that is orthogonal to the types of structure assumed by many other solution methods. One example of this is the integration of abstraction methods like SPI with reachability analysis [10].

SPI and other decision-theoretic regression methods need to be tested empirically on more realistic domains. Further testing will give an idea as to the types of problem structure that exist in naturally-occurring MDPs. This will also suggest the types of representations (and associated regression algorithms) that can best exploit this strcuture.

Finally, we hope to extend our decision-theoretic regression algorithms to more sophisticated forms of MDPs that lend themselves to more realistic modeling of domains. This includes consideration of first-order representations of stochastic decision problems that allow objects and relations over them to be specified. Such an extension is crucial in the modeling of real-world planning problems. Also of interest is the extension of these methods to semi-Markov and hybrid (continuous-discrete) models. The application of decision-theoretic regression to partially observable settings is important for realistic modeling as well. Investigations into the application of SPI to POMDPs is reported in [16], where vectors corresponding to the usual piecewise linear representation of value functions for POMDPs are treated as decision trees, produced by decision-theoretic regression. Further investigations into compatible structured belief state representations is needed to make the approach more practical.

References

[1] Christopher G. Atkeson, Andrew W. Moore, and Stefan Schaal. Locally weighted learning for control. *Artificial Intelligence Review*, 11:75–113, 1997.

[2] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *International Conference on Computer-Aided Design*, pages 188–191. IEEE, 1993.

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

[4] D. P. Bertsekas and D. A. Castanon. Adaptive aggregation for infinite horizon dynamic programming. *IEEE Transactions on Automatic Control*, 34:589–598, 1989.

[5] Dimitri P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Englewood Cliffs, 1987.

[6] Dimitri P. Bertsekas and John. N. Tsitsiklis. *Neuro-dynamic Programming*. Athena, Belmont, MA, 1996.

[7] Marko Bohanic and Ivan Bratko. Trading accuracy for simplicity in decision trees. *Machine Learning*, 15:223–250, 1994.

[8] Craig Boutilier. Correlated action effects in decision theoretic regression. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pages 30–37, Providence, RI, 1997.

[9] Craig Boutilier, Ronen I. Brafman, and Christopher Geib. Prioritized goal decomposition of Markov decision processes: Toward a synthesis of classical and decision theoretic planning. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1156–1162, Nagoya, 1997.

[10] Craig Boutilier, Ronen I. Brafman, and Christopher Geib. Structured reachability analysis for Markov decision processes. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 24–32, Madison, WI, 1998.

[11] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[12] Craig Boutilier and Richard Dearden. Using abstractions for decision-theoretic planning with time constraints. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1016–1022, Seattle, 1994.

[13] Craig Boutilier and Richard Dearden. Approximating value trees in structured dynamic programming. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 54–62, Bari, Italy, 1996.

[14] Craig Boutilier, Nir Friedman, Moisés Goldszmidt, and Daphne Koller. Context-specific independence in Bayesian networks. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 115–123, Portland, OR, 1996.

[15] Craig Boutilier and Moisés Goldszmidt. The frame problem and Bayesian network action representations. In *Proceedings of the Eleventh Biennial Canadian Conference on Artificial Intelligence*, pages 69–83, Toronto, 1996.

[16] Craig Boutilier and David Poole. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1168–1175, Portland, OR, 1996.

[17] Craig Boutilier and Martin L. Puterman. Process-oriented planning and average-reward optimality. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1096–1103, Montreal, 1995.

[18] Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*. MIT Press, Cambridge, 1995.

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

[20] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Conference on Logic in Computer Science*, pages 428–439, 1990.

[21] Anthony R. Cassandra, Leslie Pack Kaelbling, and Michael L. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1023–1028, Seattle, 1994.

[22] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[23] David Chapman and Leslie Pack Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 726–731, Sydney, 1991.

[24] E. M. Clarke, E. A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *Symposium on Principles of Programming Languages*, pages 117–126. ACM, 1983.

[25] Adnan Darwiche and Moisés Goldszmidt. Action networks: A framework for reasoning about actions and change under uncertainty. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 136–144, Seattle, 1994.

[26] Thomas Dean and Robert Givan. Model minimization in Markov decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 106–111, Providence, 1997.

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

[28] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Planning with deadlines in stochastic domains. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 574–579, Washington, D.C., 1993.

[29] Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150, 1989.

[30] Richard Dearden and Craig Boutilier. Abstraction and approximate decision theoretic planning. *Artificial Intelligence*, 89:219–283, 1997.

[31] Thomas G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 118–126, Madison, WI, 1998.

[32] Thomas G. Dietterich and Nicholas S. Flann. Explanation-based learning and reinforcement learning: A unified approach. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 176–184, Lake Tahoe, 1995.

[33] Thomas G. Dietterich and Nicholas S. Flann. Explanation-based learning and reinforcement learning: A unified view. *Machine Learning*, 28(2):169–210, 1997.

[34] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[35] Jerome A. Feldman and Robert F. Sproull. Decision theory and artificial intelligence II: The hungry monkey. *Cognitive Science*, 1:158–192, 1977.

[36] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[37] Zoltán Gábor, Zsolt Kalmár, and Csaba Szepesvári. Multi-criteria reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 197–205, Madison, WI, 1998.

[38] Dan Geiger and David Heckerman. Advances in probabilistic reasoning. In *Proceedings of the Seventh Conference on Uncertainty in Artificial Intelligence*, pages 118–126, Los Angeles, 1991.

[39] Robert Givan and Thomas Dean. Model minimization, regression, and propositional STRIPS planning. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1163–1168, Nagoya, Japan, 1997.

[40] Steve Hanks and Drew V. McDermott. Modeling a dynamic and uncertain world i: Symbolic and probabilistic reasoning about change. *Artificial Intelligence*, 1994.

[41] Steven John Hanks. *Projecting Plans for Uncertain Worlds*. PhD thesis, Yale University, 1990.

[42] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, Englewood Cliffs, 1966.

[43] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 279–288, Stockholm, 1999.

[44] Ronald A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, 1960.

[45] Ronald A. Howard and James E. Matheson, editors. *Readings on the Principles and Applications of Decision Analysis*. Strategic Decision Group, Menlo Park, CA, 1984.

[46] L. Hyafil and R. L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5:15–17, 1976.

[47] Leslie Pack Kaelbling. Hierarchical reinforcement learning: Preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 167–173, Amherst, MA, 1993.

[48] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[49] R. L. Keeney and H. Raiffa. *Decisions with Multiple Objectives: Preferences and Value Trade-offs*. Wiley, New York, 1976.

[50] Nicholas Kushmerick, Steve Hanks, and Daniel Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76:239–286, 1995.

[51] D. Lee and M. Yannakakis. Online miminization of transition systems. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing (STOC-92)*, pages 264–274, Victoria, BC, 1992.

[52] Michael Lederman Littman. Algorithms for sequential decision making. Ph.D. Thesis CS–96–09, Brown University, Department of Computer Science, March 1996.

[53] William S. Lovejoy. A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, 28:47–66, 1991.

[54] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 634–639, Anaheim, 1991.

[55] John McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[56] Nicolas Meuleau, Milos Hauskrecht, Kee-Eung Kim, Leonid Peshkin, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Solving very large weakly coupled Markov decision processes. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 165–172, Madison, WI, 1998.

[57] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, 1988.

[58] J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for adl. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 103–114, Cambridge, MA, 1992.

[59] David Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1):81–129, 1993.

[60] David Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1–2):7–56, 1997.

[61] Doina Precup, Richard S. Sutton, and Satinder Singh. Theoretical results on reinforcement learning with temporally abstract behaviors. In *Proceedings of the Tenth European Conference on Machine Learning*, pages 382–393, Chemnitz, Germany, 1998.

[62] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York, 1994.

[63] Martin L. Puterman and M.C. Shin. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24:1127–1137, 1978.

[64] J. Ross Quinlan. *C45: Programs for Machince Learning*. Morgan Kaufmann, San Mateo, 1993.

[65] Ronald L. Rivest. Learning decision lists. *Machine Learning*, 2:229–246, 1987.

[66] Earl D. Sacerdoti. The nonlinear nature of plans. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 206–214, 1975.

[67] M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1039–1046, Milan, 1987.

[68] Paul L. Schweitzer, Martin L. Puterman, and Kyle W. Kindle. Iterative aggregation-disaggregation procedures for discounted semi-Markov reward processes. *Operations Research*, 33:589–605, 1985.

[69] Ross D. Shachter. Evaluating influence diagrams. *Operations Research*, 33(6):871–882, 1986.

[70] Solomon E. Shimony. The role of relevance in explanation I: Irrelevance as statistical independence. *International Journal of Approximate Reasoning*, 8(4):281–324, 1993.

[71] Satinder P. Singh and David Cohn. How to dynamically merge Markov decision processes. In *Advances in Neural Information Processing Systems 10*, pages 1057–1063. MIT Press, Cambridge, 1998.

[72] Satinder Pal Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8:323–339, 1992.

[73] Richard D. Smallwood and Edward J. Sondik. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, 21:1071–1088, 1973.

[74] James E. Smith, Samuel Holtzman, and James E. Matheson. Structuring conditional relationships in influence diagrams. *Operations Research*, 41(2):280–297, 1993.

[75] Edward J. Sondik. The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research*, 26:282–304, 1978.

[76] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, Austin, 1990.

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

[78] Jonathan Tash and Stuart Russell. Control strategies for a stochastic planner. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1079–1085, Seattle, 1994.

[79] Joseph A. Tatman and Ross D. Shachter. Dynamic programming and influence diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, 20(2):365–379, 1990.

[80] Gerald J. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

[81] John H. Tsitsiklis and Benjamin Van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59–94, 1996.

[82] Paul E. Utgoff. Decision tree induction based on efficient tree restructuring. Technical Report 95–18, University of Massachusetts, March 1995.

[83] Richard Waldinger. Achieving several goals simultaneously. In E. Elcock and D. Mitchie, editors, *Machine Intelligence 8: Machine Representations of Knowledge*, pages 94–136. Ellis Horwood, Chichester, England, 1977.

[84] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

Acknowledgements

Thanks to Tom Dean, David Poole and Marty Puterman for discussions and comments on various aspects of this research. We also thank the referees for their suggestions. Craig Boutilier was supported by NSERC Research Grant OGP0121843, IRIS Phase II Project IC-7 and IRIS Phase III Project BAC. This work was done in part while Boutilier was at the University of British Columbia. Richard Dearden was supported by a UBC University Graduate Fellowship, a Killam Predoctoral Scholarship, IRIS Phase II Project IC-7 and IRIS Phase III Project BAC. This work was done in part while Moisés Goldszmidt was at Rockwell International Science Center and supported by DARPA contract F30602-95-C-0251.