To appear, *Proc. Fourteenth Inter. Conf. on AI (IJCAI-95),* Montreal, August, 1995

Exploiting Structure in Policy Construction

Craig Boutilier
Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4, CANADA
cebly@cs.ubc.ca

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

Moisés Goldszmidt
Rockwell Science Center
444 High Street
Palo Alto, CA 94301, U.S.A.
moises@rpal.rockwell.com

Abstract

Markov decision processes (MDPs) have recently been applied to the problem of modeling decision-theoretic planning. While traditional methods for solving MDPs are often practical for small states spaces, their effectiveness for large AI planning problems is questionable. We present an algorithm, called *structured policy iteration* (SPI), that constructs optimal policies without explicit enumeration of the state space. The algorithm retains the fundamental computational steps of the commonly used modified policy iteration algorithm, but exploits the variable and propositional independencies reflected in a temporal Bayesian network representation of MDPs. The principles behind SPI can be applied to any structured representation of stochastic actions, policies and value functions, and the algorithm itself can be used in conjunction with recent approximation methods.

1 Introduction

Increasingly research in planning has been directed towards problems in which the initial conditions and the effects of actions are not known with certainty, and in which multiple potentially conflicting objectives must be traded against one another to determine optimal courses of action. For this reason, there has been much interest in *decision theoretic planning* (Dean and Wellman 1991). In particular, the theory of *Markov decision processes* (MDPs) has found considerable popularity recently both as a conceptual and computational model for DTP (Dean et al. 1993; Boutilier and Dearden 1994; Tash and Russell 1994).

While MDPs provide firm semantic foundations for much of DTP, the question of their computational utility for AI remains. Many robust methods for optimal policy construction have been developed in the operations research (OR) community, but most of these methods require explicit enumeration of the underlying state space of the planning problem, which grows exponentially with the number of variables relevant to the problem at hand. This severely affects the performance of these methods, the storage required to represent the problem, and the amount of effort required by the user to specify the problem. Much emphasis in DTP research has been placed on the issue of speeding up computation, and several solutions proposed, including local search methods (Dean et al. 1993; Dearden and Boutilier 1994; Barto, Bradtke and Singh 1995; Tash and Russell 1994) or reducing the state space via abstraction (Boutilier and Dearden 1994). Both approaches reduce the state space in a way that allows MDP solution techniques to be used, and generate approximately optimal solutions (whose accuracy can sometimes be bounded *a priori* (Boutilier and Dearden 1994)). While approximation is no doubt crucial, two questions remain: a) what if optimal solutions are required? b) what if the state space reduction afforded by these methods is not great enough to admit feasible solution?

The approach we propose is orthogonal to the approximation techniques mentioned above. It is based on a structured representation of the domain that allows the exploitation of regularities and independencies in the domain to reduce the "effective" state space. This reduction has an immediate effect on the computation of the solution, the storage required, and on the effort required to specify the problem. The approach has the following benefits:

- It computes an optimal, rather than an approximate solution. Thus, it can be applied in instances where optimality is strictly required.
- It employs representations of actions and uncertainty that are well known in the AI literature.
- It is orthogonal to, and can be used in conjunction with, many of the approximation techniques mentioned above.

This third point is especially significant because approximation methods such as abstraction often require that one optimally solve a smaller problem.

In this paper, we describe our investigations of a commonly used algorithm from OR called *modified policy iteration* (MPI) (Puterman and Shin 1978). We present a new algorithm called *structured policy iteration* (SPI) which uses the same computational mechanism as MPI. As in (Boutilier and Dearden 1994), we assume a compact representation of an MDP, in this case using a "two-slice" temporal Bayesian network (Dean and Kanazawa 1989; Darwiche and Goldszmidt 1994) to represent the dependence between variables before and after the occurrence of an action. In addition, we use a structured decision tree representation of the conditional probability matrices quantifying the network to exploit "propositional" independence, that is, independence given a particular variable *assignment* (see also (Smith, Holtzman and Matheson 1993)). Propositional independence is reflected in the specific quantification of the network, in contrast to the *vari-*

able independence captured by the network structure. Such representations allow problems to be specified in a natural and concise fashion; and they have the added advantage of allowing problem structure to be easily identified.

Using this representation, we can exploit the structure and regularities of a domain in order to obviate explicit state space enumeration. Roughly, at any point in our computation, states are partitioned in two distinct ways: those states assigned the same action by the "current" policy are grouped together, forming one partition of the state space; and those state whose "current" estimated value is the same are grouped, forming a second partition. MPI-style computations can be performed, but need only be considered once for each partition, rather than for each state. The motivation for our method is similar to that underlying Bayes nets and influence diagrams, namely, that many problems seem to exhibit tremendous structure. Just as network algorithms have proven practical for reasoning under uncertainty, we expect SPI to be quite useful in practice.[1]

In Section 2 we briefly describe MDPs and the MPI algorithm; we refer to (Puterman 1994) for a more detailed description of MDPs and solution techniques. In Section 3 we discuss our representation of MDPs using decision trees, and in Section 4 we describe the structured policy iteration algorithm. The two phases of the algorithm, *structured successive approximation* and *structured policy improvement*, are described individually. We illustrate the algorithm on a detailed example, and describe the results of our implementation. We refer to the full paper (Boutilier, Dearden and Goldszmidt 1994) for a much more detailed description of the algorithm and implementation, and discussion of additional issues.

2 Modified Policy Iteration

We assume a DTP problem can be modeled as a *completely observable MDP*. We assume a finite set of states S and actions $\mathcal{A}$, and a reward function R. While an action takes an agent from one state to another, the effects of actions cannot be predicted with certainty; hence we write $Pr(s_1, a, s_2)$ to denote the probability that s_2 is reached given that action a is performed in state s_1. These transition probabilities can be encoded in an $|S| \times |S|$ matrix for each action. Complete observability entails that the agent always knows what state it is in. We assume a bounded, real-valued *reward function* R, with $R(s)$ denoting the (immediate) utility of being in state s. For our purposes an MDP consists of S, $\mathcal{A}$, R and the set of transition distributions $\{Pr(\cdot, a, \cdot) : a \in \mathcal{A}\}$.

A plan or *policy* is a mapping $\pi : S \rightarrow \mathcal{A}$, where $\pi(s)$ denotes the action an agent will perform whenever it is in state s.[2] Given an MDP, an agent ought to adopt an optimal policy that maximizes the expected rewards accumulated as it performs the specified actions. We concentrate here on *discounted infinite horizon* problems: the current value of future rewards is discounted by some factor β ($0 < \beta < 1$); and we want to maximize the expected accumulated discounted rewards over an infinite time period. The expected *value* of a fixed policy π at any given state s is given by:

$$V_\pi(s) = R(s) + \beta \sum_{t \in S} Pr(s, \pi(s), t) \cdot V_\pi(t) \qquad (1)$$

The value of π at any initial state s can be computed by solving this system of linear equations. A policy π is *optimal* if $V_\pi(s) \geq V_{\pi'}(s)$ for all $s \in S$ and policies π'.

Howard's (1971) policy iteration algorithm constructs an optimal policy by improving the "current" (initially random) policy by finding for each state some action better than the current action for that state. Each iteration of the algorithm involves two steps, *policy evaluation* and *policy improvement*:

1. For each $s \in S$, compute $V_\pi(s)$.
2. For each $s \in S$, find the action a that maximizes

$$V_a(s) = R(s) + \beta \sum_{t \in S} Pr(s, a, t) \cdot V_\pi(t) \qquad (2)$$

If $V_a(s) > V_\pi(s)$, let policy π' be such that $\pi'(s) = a$; otherwise $\pi'(s) = \pi(s)$.

The algorithm iterates on each new policy π' until no improvement is found. The algorithm will terminate with an optimal policy, and in practice tends to converge in a reasonable number of iterations.

Policy evaluation requires the solution of a set of $|S|$ linear equations in $|S|$ unknowns. This can be computationally prohibitive for very large state spaces. However, one can estimate V_π through several steps of *successive approximation*. V_π is approximated by a sequence of vectors $V^0, V^1, \cdots$, each a successively better estimate. The initial estimate V^0 is any random $|S|$-vector. The estimate $V^i(s)$ is given by

$$V^i(s) = R(s) + \beta \sum_{t \in S} Pr(s, \pi(s), t) \cdot V^{i-1}(t) \qquad (3)$$

Modified policy iteration (Puterman and Shin 1978) uses some number of successive approximation steps to produce an estimate of $V_\pi(s)$ at step 1. We refer to (Puterman 1994) for theoretical and practical advice for choice of good *stopping criteria*. MPI is used frequently in practice for large state space problems with good results (Puterman 1994).

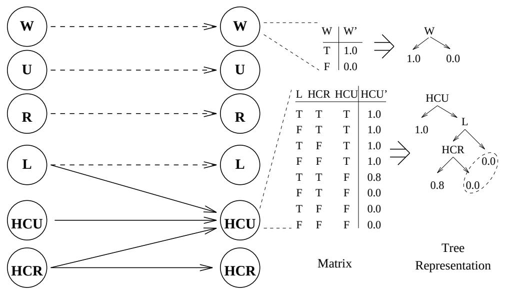
3 Representation of MDPs

It is unreasonable to expect that DTP problems, while formulable as MDPs, will be specified in the manner described above. Since state spaces grow exponentially with the number of propositions relevant to a problem, one should not expect a user to provide an explicit $|S| \times |S|$ probability matrix for each action, or a $|S|$-vector of immediate rewards. Regularities in action effects and reward structure will usually permit more natural and concise representations.

We will illustrate our representational methodology (and algorithm) on a simple example: a robot is charged with the task of going to a café to buy coffee and delivering it to a user in their office. It may rain on the way and the robot will get wet unless it has an umbrella. We have six propositions (hence 64 states) – L (location of robot: L at office, $\overline{L}$ at café), W (robot is wet), U (robot has umbrella), R (raining), *HCR* (robot has coffee), *HCU* (user has coffee) – and four actions —

[1]The analogy in fact is quite strong: Tatman and Shachter (1990) have shown that influence diagram methods perform dynamic programming steps on MDP problems in a way that "compacts" the state space somewhat. However, their method is restricted to finite-horizon problems, and adopts *value iteration*, which converges much too slowly on the infinite (or indefinite) horizon problems frequently encountered in planning (Puterman 1994).

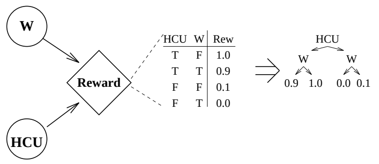
[2]Thus we restrict attention to *stationary* policies. For the problems we consider, optimal stationary policies always exist.



W	W'
T	1.0
F	0.0

L	HCR	HCU	HCU'
T	T	T	1.0
F	T	T	1.0
T	F	T	1.0
F	F	T	1.0
T	T	F	0.8
F	T	F	0.0
T	F	F	0.0
F	F	F	0.0

Figure 1: Action Network for *DelC*



HCU	W	Rew
T	F	1.0
T	T	0.9
F	F	0.1
F	T	0.0

Figure 2: Reward Function Network

Go (to opposite location), *BuyC* (buy coffee), *DelC* (deliver coffee to user), *GetU* (get umbrella). Each of these actions has the obvious effect on a state, but may fail with some probability (see (Boutilier, Dearden and Goldszmidt 1994) for a full problem specification).

We discuss one possible representation for actions and utilities, Bayesian networks, and in the next section show how this information can be exploited in MPI. While our algorithm depends on the particular representation given, the nature of our method does not and could be used with, say, the probabilistic STRIPS representation of (Boutilier and Dearden 1994; Kushmerick, Hanks and Weld 1994).

We assume a set $\mathcal{P}$ of atomic propositions characterizing the relevant features of our domain. Because of the Markov assumption, the effect of a given action a is completely determined by the current state of the world, and can be represented by a "two-slice" temporal Bayes net (Dean and Kanazawa 1989; Darwiche and Goldszmidt 1994): we have one set of nodes representing the state of the world prior to the action (one node for each $P \in \mathcal{P}$), another set representing the world after the action has been performed, and directed arcs representing causal influences between these sets.[3] Figure 1 illustrates this network representation for the action *DelC* (deliver coffee); we will have one such network for each action.

The post-action nodes have the usual matrices describing the probability of their values given the values of their parents, under action A. We assume that these conditional probability matrices are represented using a *decision tree* (or if-then rules) (Smith, Holtzman and Matheson 1993). This allows independence among variable *assignments* to be represented, not just variable independence (as captured by the network structure), and is exploited to great effect below. A tree representation of the matrices for variables *HCU* and W in the *DelC* network is shown in Figure 1 along with the induced matrix (our convention is to use left-arrows for "true" and right-arrows for "false"). Each branch thus determines a partial assignment to the parents of that variable (in the network) with some parents unmentioned. The leaf at each branch denotes the probability of the variable being true after the action is executed given any conditions consistent with that branch. In this case $\Pr(HCU) = .8$ when $\overline{HCU}$, L and *HCR* hold prior to the action. The tree associated with proposition P in the network for an action a is denoted *Tree*$(P|a)$. Since we are interested only in the transition probabilities $P(s_1, a, s_2)$ for a *known* state s_1, we do not require prior probabilities (or matrices) for pre-action variables (the roots of the network).

We note that many actions affect only a small number of variables; to ease the burden on the user, we allow unaffected variables to be left out of the network *specification* for that action. Persistence is assumed and the arcs (indicated by broken arrows) and trees can be constructed automatically. For instance, all of L, R, W and U are unaffected by the action *DelC*. It is easy to see how such an action representation induces a transition matrix over the state space.

We assume that the immediate reward R is solely a function of the state of the world. As such, we can use a simple "atemporal influence diagram" to capture the regularities in such a function. Since (immediate) reward is independent of stage and the action performed, we need only one network to capture reward. Figure 2 illustrates such a network. Only variables that influence reward need be specified. One may also use a tree-structured representation for R as shown (where leaves now indicate the reward associated with any state consistent with the branch). It is easy to see how such a tree determines the reward function $R(s)$. In this example, the robot gets a reward of 0.9 if the user has coffee and 0.1 if it stays dry, which are added to determine $R(s)$.

4 Structured Policy Iteration

Given a network formulation of an MDP, one might compute an optimal policy by constructing the appropriate transition matrices and reward vector, and solving with standard techniques. But as the number of propositions increase, state spaces grow exponentially, and these methods quickly become infeasible. In addition, although these algorithms may converge in relatively few iterations, memory requirements are quite intensive.[4] If a problem can be represented compactly, the representation must exploit certain regularities and structure in the problem domain. Therefore one can often expect that optimal policies themselves have certain structure, as do value functions V_π. The optimal policy for our example problem can be expressed quite concisely. For example, *DelC* is the best action whenever *HCR* and L are both true, regardless of the truth values of the other four variables. Thus by associating the action *DelC* with the proposition *HCR* $\wedge L$, we capture the policy for 16 states with one assertion.

We propose a method for optimal policy construction that eliminates the need to construct explicit transition matrices, reward and value vectors, and policy vectors. Our method is based on MPI, but exploits the fact that at any stage in the

[3]To simplify the exposition, we only consider binary variables, and assume that there are no arcs between post-action variables. Relaxing these assumptions (as long as the network is acyclic) does not complicate our algorithm in any essential ways.

[4]Puterman (1994) describes this as a potential bottleneck; our previous experiences also suggests this is often problematic.

computation: (a) The current policy π may be structured; and (b) The value function for a policy V_π, or some estimate V^i thereof, may be structured. Rather than having a policy vector of size $|S|$ individually associating an action with each state, one can use a structured representation.

Definition A *structured policy* is any set of formula-action pairs $\pi = \{\langle \psi_i, a_i \rangle\}$ such that the set of propositional formulae $\{\psi_i\}$ partitions the state space. This induces the "explicit" policy $\pi(s) = a_i$ iff $s \models \psi_i$.

Structured policies can be represented in many ways (e.g., with *decision lists* (Rivest 1987)). We adopt a *decision tree* representation similar to the representation of probability matrices above. Leaves are labeled with the action to be performed given the partial assignment corresponding to the branch. Thus, if there are k leaf nodes, the state space is partitioned into k subsets or *clusters*. Figure 6 (the second tree) illustrates a structured policy: we have 4 clusters and 4 action assignments rather than one action assignment for each of 64 states.[5]

A *structured value vector* can be represented similarly as a set of formula-value pairs $V = \{\langle \varphi_i, v_i \rangle\}$, where states satisfying φ_i have value v_i. Again, we will use a tree-structured representation for such a partition. In this case, each leaf is annotated with the value associated with that partition (see Figure 4).

The insights crucial to our algorithm are the following:

(a) If we have structured policy π and a structured value estimate V^i for π, an improved estimate V^{i+1} can often preserve much of this structure;

(b) If we have a structured value estimate V_π, we can construct a structured improving policy π'.

The first observation suggests a structured form of successive approximation, while the second suggests that one can improve a policy in a way that exploits structure. This gives rise to the SPI algorithm for structured policy iteration:

1. Choose a random *structured policy* π, then LOOP through 2,3:
2. Approximate value function $V_\pi(s)$ using *structured successive approximation*.
3. Produce an improved *structured policy* π' (if no improvement is possible, terminate).

We describe these components of the algorithm in turn below.

Initial structured policy selection is fairly unconstrained. In the example below, we adopt the greedy "one-step" policy $\pi = \{\langle \top, \textit{DelC} \rangle\}$ (deliver coffee no matter what the state). Simpler policies should be preferred.

4.1 Structured Successive Approximation

Phase 1 of each iteration of SPI invokes *structured successive approximation* (SSA): we assume we have been given a structured policy and an initial structured estimate of that policy's value. During the first iteration of SPI, SSA may use the immediate reward tree as its initial structured estimate. In subsequent iterations the initial estimate is the computed value-tree for the previous policy.

[5]We note that tree representations of policies are sometimes used in reinforcement learning as well (Chapman and Kaelbling 1991); however, the motivation there is somewhat different. In addition, the ordering of variables in the tree can have a dramatic impact on the size of the representation (see Section 5).

```
Input: Tree(V^i), action a; Output: Tree(V^{i+1})
1. Determine a total ordering O of variables in Tree(V^i)
2. Set Tree(V^{i+1}) = ∅
3. For each variable X in Tree(V^i) (using ordering O):
    (a) Determine partial branch in Tree(V^i) that make X relevant
    (b) For each leaf in Tree(V^{i+1}) that assigns nonzero probability to
        some such branch: 1) Append Tree(X|a) to leaf, collapsing redundant
        nodes; 2) Copy probability labels from leaf to each 'new'
        leaf (from Tree(X|a))
4. At each leaf of Tree(V^{i+1}), use prob labels to compute expected future value
5. Add reward/action costs (expanding leaves if needed)
```

Figure 3: Algorithm Explain$(a, \textit{Tree}(V^i))$

Given a policy π, we wish to determine its value V_π. The basic step of successive approximation involves producing a more accurate estimate of a policy's value V^{i+1} given some previous estimate V^i using Equation 3. Successively better estimates are produced until the difference between V^{i+1} and V^i is (componentwise) below some threshold.[6]

SSA embodies the intuition that, given a structured value vector V^i, the conditions under which two states can have different values V^{i+1} can be readily determined from the action representation. In particular, although an action may have different effects at two states, if this difference is only in variables or variable assignments *not relevant to the structured value vector* V^i, then these states must have identical values in V^{i+1}. Since V^i is tree-structured, the SSA algorithm can easily determine what assignments are relevant to value at stage i. The crucial feature of the algorithm is its use of the action representation to cluster the states (i.e., form a new tree V^{i+1}) under which the policy must have the same value at stage $i+1$. By doing so, we can calculate V^{i+1} once for each leaf of the tree rather than for each state in the state space. This may have a significant impact on both time and memory requirements in many cases.

We first describe the main loop of the SSA assuming a single action (uniform policy) to be executed (see Algorithm Explain$(a, \textit{Tree}(V^i))$ in Figure 3), and give a detailed example. We then describe how general policies are dealt with.

We accept a structured value vector $\textit{Tree}(V^i)$, the current estimated value for the current policy π, and an action a to be performed at stage $i+1$ (as if we were computing V_a in Eq. 2). Given an action a to be performed, the states that can have different values V^{i+1} are those that lead (under action a) to different partitions of $\textit{Tree}(V^i)$ with different probability. Roughly, $\textit{Tree}(V^i)$ describes not only what is relevant to the value of the policy (executed for i stages), but also how its value depends on (or is independent of) the particular variable assignments in the tree. To generate the $\textit{Tree}(V^{i+1})$ we want to *explain* the partitions in $\textit{Tree}(V^i)$. That is, we want to generate the conditions that, if known prior to the action, would cause a state transition with some fixed probability to fixed partitions (or leaves) of $\textit{Tree}(V^i)$.

Since the probability of reaching a given partition in V^i is a function of the probabilities of the individual variables on its branch, we can build this explanation componentwise:

[6]Better stopping criteria are possible (Puterman 1994), but have no bearing on our algorithm.

we consider the variables in *Tree*(V^i) individually. More precisely, explanations are generated by a process we call *abductive repartitioning*, quite similar in spirit to probabilistic Horn abduction (Poole 1993). A given traversal of *Tree*(V^i) induces an ordering of relevant (post-action) variables; we "explain" variables in *Tree*(V^i) according to this order (Step 3 of Figure 3).

For each variable X in *Tree*(V^i), the conditions under which the probability of X varies when action a is executed is given by *Tree*$(X|a)$, which is simply read from the network for a (see Step 3b.1). Hence explanation generation for the individual variables is trivial – an explanation consists of the tree whose branches are partial truth assignments and whose leaves reflect the probability that the variable becomes true. This explanation must be added to the current (partial) tree for V^{i+1} (Step 3b.1). However, it need not be added to the end of every partial branch. *Tree*(V^i) asserts the conditions under which X is relevant to value; the explanation for X need only be added to the leaves where those conditions are possible (Steps 3a and 3b). Since the tree is generated in the order dictated by *Tree*(V^i), the probabilities of the relevant variables are already on the leaves of the partial tree. Once *Tree*$(X|a)$ is added to the required leaves, the new leaves of the tree now have *Pr*(X) attached in addition to the probabilities of the previous variables (Step 3b.2), and these can be used to determine where the explanation for the next variable must be placed. Should the variable labeling a node of *Tree*$(X|a)$ occur earlier in the partial tree for V^{i+1}, that node in *Tree*$(X|a)$ can be deleted (since the assignments to that node in *Tree*$(X|a)$ must be either redundant or inconsistent – Step 3b.1). Thus, much shrinkage is possible (see below).[7]

Figure 4 illustrates the value trees that result for two successive approximation steps, V^1 and V^2, as well as the fiftieth step in our example, using the initial policy *DelC*. The generation of *Tree*(V^1) from *Tree*(V^0) is straightforward. The first relevant variable *HCU* will have different outcome probabilities as dictated by *Tree*(*HCU*|*DelC*) in Figure 1; this requires the addition of variables *HCU*, L and *HCR* to *Tree*(V^1). The other relevant variable W has its tree added to each of the leaves of *Tree*(*HCU*|*DelC*), since it is relevant no matter how *HCU* turns out. This results in the tree structure shown in Figure 4 labeled V^1.

More interesting is the generation of *Tree*(V^2) using *Tree*(V^1), illustrated in Figure 5, which we now describe in some detail. The four variables in *Tree*(V^1) are ordered *HCU*, L, *HCR*, W. We start by inserting *Tree*(*HCU*|*DelC*) (Stage1) which explains *HCU*. The probability of *HCU* given the relevant assignment labels the leaf nodes. The next variable L is "explained" by *Tree*$(L|$*DelC*$)$; however, from *Tree*(V^1) we notice that L is only relevant when *HCU* is false. Therefore, we only add L to those leaves where *Pr*(*HCU*) < 1.[8] We notice that such leaves only exist below the node L in our partial tree. We also notice that *Tree*$(L|$*DelC*$)$ contains only the variable L (by persistence); thus, no additional branches need to be added to the tree (any further partitioning on L is either redundant or inconsistent). The net result is the simple addition of a probability label for L on these leaves (see Stage2). In general, for more complicated trees, we will add a tree of the form *Tree*$(X|a)$ to a leaf node, and eliminate some (but perhaps not all) of its nodes.

The next variable is *HCR*, which is only relevant when $\overline{HCU}$ and L hold, and *Tree*(*HCR*|*DelC*) is added only at leaves where *Pr*(*HCU*) < 1 and *Pr*$(L) > 0$. However, as with L, the addition of *Tree*(*HCR*|*DelC*) (containing only the variable *HCR*) is redundant since leaves satisfying this condition lie below node *HCR* is the partial tree (see Stage3). Finally, the variable W must be explained. W is relevant at all points in the partial tree, so its is added to each leaf node (Stage4).

Finally, with the probability labels, the value tree V^1, the reward tree, and the discounting factor (here $\beta = 0.9$), the leaf nodes can be labeled with the values for V^2 (Figure 4). In this example, abductive repartitioning gives rise to six distinct values in the estimates V^1 and V^2. Our mechanism generates a tree with eight leaves, but two pairs of these can be identified as necessarily having the same value (indicated by the broken ovals); see Section 5. Thus, in principle, only six value computations need be performed rather than 64.

To deal with general policies instead of single actions, SSA proceeds as follows. We assume the policy π is represented structurally as *Tree*(π). For each action a that occurs in π, the explanation algorithm is run, as above. The tree generated for an action a is then appended to the leaves of *Tree*(π) at which a occurs. Since *Tree*(π) may make certain distinctions that occur in the appended "action trees," we delete any redundant nodes to simplify the tree (either during or after its construction).

The SSA algorithm requires some number of rounds of successive approximation before a reasonable estimate of the policy's value can be determined and the policy improvement phase can be invoked. While the number of backups per step can potentially be reduced exponentially, there may be considerable overhead involved in determining the appropriate partitions (or the reduced "state space"). We first note that the reduction will often be worth the additional computation, especially as state spaces become (realistically) large — even as domains increase, the effects of actions may be quite localized in many problem settings. We can expect this reduction for a particular policy to be quite valuable. In addition, and more importantly, this repartitioning need not be performed at each successive approximation step. As the following theorem indicates, once the partition stabilizes for two successive approximations, it cannot change subsequently.

Theorem 1 *Let* $\{\varphi^i_j\}$ *and* $\{\varphi^{i+1}_j\}$ *be the partitions constructed for value estimates* V^i *and* V^{i+1}*, respectively. If* $\{\varphi^i_j\} = \{\varphi^{i+1}_j\}$*, then* $V^k(s) = V^k(s')$ *for any* $k \geq i$*, and states* s, s' *such that* $s \models \varphi^i_j$, $s' \models \varphi^i_j$ *for some* j*.*

Thus, the backups for successive estimates can proceed without any repartitioning. Essentially, the very same computations are performed for each partition as are performed for each state in (unstructured) SA, with no additional overhead. In our example, we reach such convergence quickly. When repartitioning to compute V^2, we discover that the partition is unchanged: the tree-representation of V^2 is identical to that for V^1. Thus, after one repartitioning the partition of our value vector has stabilized and backups can proceed apace.

[7]In general, one has to consider also the impact of the immediate reward function, whose tree can be incorporated into the tree V^{i+1} (see Step 5); however, this is often unnecessary when the reward function is used as an initial value estimate.

[8]We don't use *Pr*(*HCU*) $= 0$; L is relevant to value whenever *HCU* is less than certain.

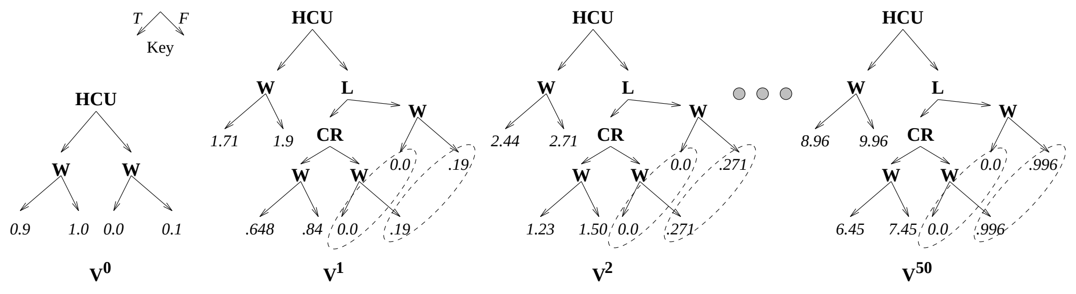


Figure 4: Fifty Iterations of SSA

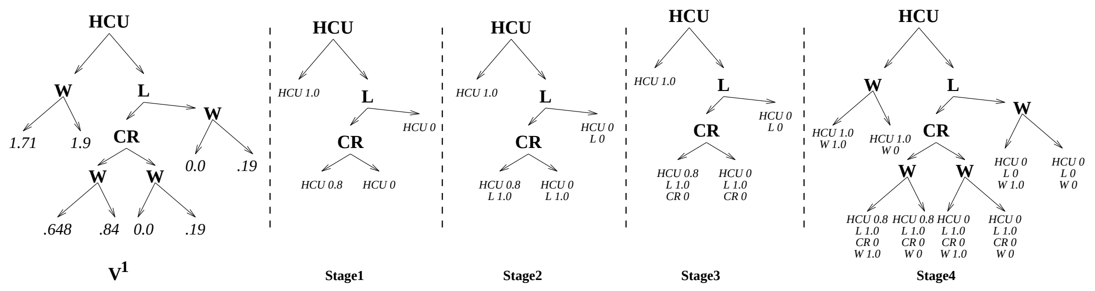


Figure 5: Generation of Abductively Repartitioned Vector

The value vector V^{50} contains the values that are within 0.4% of the true value vector V_π, as shown in Figure 4.

It is important to note that while the value vector approximates the true value function for the specified policy, the approximation is an inherent part of the policy evaluation algorithm. No loss of information results from our partitioning. The structured value estimates V^i are the same as those generated in the classic successive approximation (or MPI) algorithm: they are simply expressed more compactly.

4.2 Structured Policy Improvement

When we enter Phase 2 of the SPI algorithm, we have a current structured policy π and a structured value vector V_π. The policy improvement phase of MPI requires that we determine possible "local" improvements in the policy — for each state we determine the action that maximizes the one-step expected reward, using our value estimate V_π as a terminal reward function; that is, the a that maximizes $V_a(s)$. Should a be different from $\pi(s)$, we replace $\pi(s)$ by a in our new policy.

Once again, we want to exploit the structure in the network to avoid explicit calculation of all $|S||A|$ values. While there are several ways one might approach this problem, one rather simple method is based on the observation made above. For any fixed structured value vector V_π and action a, we can determine the value vector V_a using the algorithm Explain$(a, \mathit{Tree}(V^i))$ described above. Abductive repartitioning is used to identify the relevant pre-action conditions that influence this outcome, and provides us with a new partition of the state space for a, dividing the space into clusters of states whose value $\sum_{t\in S} \mathit{Pr}(s,a,t)\cdot V_\pi(t)$ is identical. We determine one such partitioning of the state space for each action a and compute the value of a for each partition.

Figure 6 illustrates the value trees generated by the abductive repartitioning scheme for the two actions *DelC* (the tree V^{50} from Figure 4). and *Go*. The values labeling the leaves indicate the values V^{DelC} and V^{Go} in the policy improvement phase of the algorithm — these are determined by using the probability labels generated by abductive repartitioning and the tree for V_π. (We ignore actions *BuyC* and *GetU* which generate similar trees to *DelC* but which are dominated by *DelC* at this stage of the algorithm.) We note that these values are undiscounted and do not reflect immediate reward. Without action costs, these factors cannot cause a change in the relative magnitude of total reward for an action. Thus, actions need only be compared for expected *future* rewards (although action costs are easily incorporated).[9]

With these values, we must now determine a new locally improved policy choosing either *Go* or *DelC* for each state. Given that the expected value of the action only varies between partitions, we can use the trees to quickly determine which action is best in each partition. In the worst case, where the partitions are orthogonal, we must consider their cross-product. Here, however, the trees share much structure. Our current algorithm performs a reasonably straightforward merging of the trees in question, keeping the coarsest partitions possible to produce a tree with the dominating values and action choices. The result of this process is illustrated in Figure 6 which shows the maximally improved policy. While there are a number of methods for merging the trees for each action, our current implementation reorders the trees so that the variable orderings are consistent (with the current value

[9]We should point out that while *Tree*(Go) has 20 value partitions, in fact, there are only 12 distinct values among these twenty. Moreover, these coincident values are due to structural properties of the problem and can be identified beforehand as necessarily having the same value. For legibility, we omit the ovals that join these leaves with the same value.

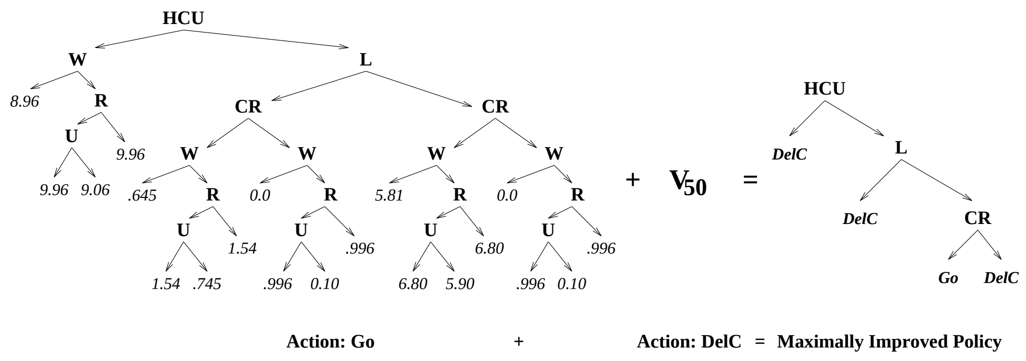


Figure 6: Value Trees for Candidate Actions and the Improved Policy

ordering if feasible); this allows a straightforward comparison of leaves to determine dominance and the structure of the new policy tree.

We note that if both *Go* and *DelC* provide equal value at a given state, the action of the earlier policy π is chosen for that state, as is usual in policy iteration. Thus, some of the states where *DelC* is chosen may have had equal value if *Go* had been chosen; but the persistence condition dictates our choice of *DelC*. The net effect is often a simpler policy tree. It also allows the easier detection of the termination condition.

4.3 Analysis

The final policy produced by the SPI algorithm (i.e., some number of SSA and improvement steps) is shown in Figure 7 along with the value function it produces. This policy is produced on the fourth iteration. A fifth policy improvement step is attempted, but no improvement is possible (note that for this fifth iteration no value approximation of the policy's value is performed). Thus, the state space is partitioned into 8 clusters allowing a very compact specification of the policy. The value tree corresponds to 50 backups of successive approximation. Since all value trees for prior iterations are (strictly) smaller than this, we see that we had to consider at most 18 distinct "states" at any given time (the first three policies peaked at 8, 10 and 14 partitions), rather than the 64 states of the original state space. In fact, the 4 sets of duplicate values are easily show to have exactly the same value because of structural properties of the problem. Though we have not yet implemented the algorithm to take advantage of this property, in principle we could use at most 14 clusters and compute at most 14 value estimates instead of 64.

The intent of this paper is to suggest methods by which structure implicit in problem representations can be imposed on policy and value vectors in policy construction. Clearly we cannot expect SPI or related methods to work well on all MDPs, for not all problems have compact representations that can be exploited; or even if a compact representation is possible, the optimal policy may not be compact. The most we might hope for is that a problem with a compact description (input) and optimal policy (output) can be computed efficiently using SPI (e.g., in polytime in the size of the input/output). Unfortunately, even this cannot be guaranteed. Consider the following example, whose network is diagramed in Figure 8. We assume n propositions P_i and actions A_i ($i \leq n$). Action A_i will cause P_i to become true as

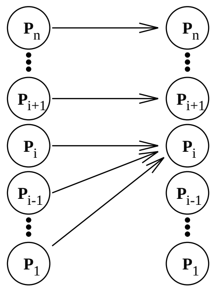


Figure 8: Description of Action A_i

long as $P_1, \cdots, P_{i-1}$ are all true; but it has the side effect of setting each of $P_1, \cdots, P_{i-1}$ false. Each of the n actions can be represented in $O(n)$ space. Our reward function assigns 1 to the state making all P_i true and 0 to all other states. Thus the problem is representable in $O(n^2)$ space. With a discounting rate $\beta < 1$, the optimal policy has the form "If $\neg P_1$ do A_1; else if $\neg P_2$ do A_2, else . . . " which in tree form requires $O(n)$ space. But the value $V_\pi(s)$ is different at each state s. Since SPI makes all distinctions relevant to value, it must produce a complete tree of size $O(2^n)$ (requiring time exponential in the problem description and solution).[10]

Our initial experiments have provided some suggestions about the types of problems on which SPI should work well, and where further optimizations can be made. Not surprisingly, SPI spends relatively more time on policy improvement vs. evaluation compared to MPI. This is due to the overhead involved in merging action trees. Good tree manipulation algorithms will play a role here. The evaluation phase of SPI compares favorably to MPI even when overhead is accounted for. When many iterations of successive approximation are needed, the overhead is amortized (due to Theorem 1) and SPI outperforms MPI considerably.

Naturally, as trees become larger the overhead of SPI becomes a crucial factor. Our experience with small (64–800 state) problems suggests that SPI requires problems whose structure allows (the equivalent of) deletion of roughly 4–6 variables (i.e., overhead appears to be proportional to tree-

[10]Essentially, the optimal policy winds through the state space like a binary counter. $V_\pi(s)$ could be compactly encoded with an appropriate of "parameterized" metric representation.

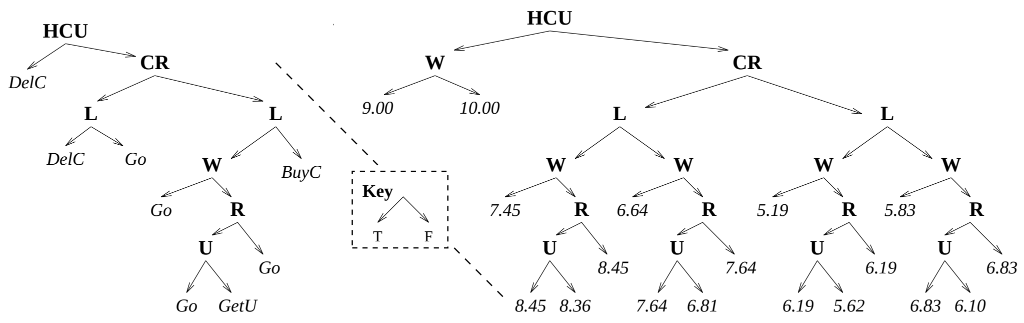


Figure 7: The Optimal Policy and Value Function

size); further experimentation with large problems is necessary to verify this conjecture. The performance of SPI with relatively small trees suggests that a hybrid approach (using SPI and switching to MPI if trees become too large) may work well in practice. In other words, we can use SPI until the effort required to discover irrelevant distinctions ceases to pay off. As a general remark, goal-based problems with competing objectives seem to be especially well-suited to SPI (as opposed to process-oriented problems), since choosing a particular goal to pursue tends to render features related to other objectives irrelevant.

5 Concluding Remarks

We have presented a natural representational methodology for MDPs that lays bare much of a problem's structure, and have presented an algorithm that exploits this structure in the construction of optimal policies. The key component of our algorithm uses an abductive mechanism that generates partitions of the state space that, at any point in the computation, group together states with the same estimated value or best action. This allows the computation of value estimates and best actions to be performed for partitions as a whole rather than for individual states. This work contributes both to AI (a specific DTP algorithm) and OR (a representational methodology and clustering technique for MDPs).

Currently we are investigating a number of extensions to this model. Our experimental results point out two important bottlenecks. The first involves multivalued variables, which if relevant, cause branching on all values. In many domains, features are relevant if a variable has one value, but not others. Using decision lists (or hybrids) to quantify Bayes nets will help in this regard; they can be used also to represent policies and value vectors. The second involves the ordering of variables in trees: good heuristics may help keep the size of policy and value trees close to optimal. Related is the use of acyclic graph representations instead of trees.

One of the most promising aspects of this work is the fact that it provides structures that should help in approximating optimal policies. The conditional relevance of variables can be quantified and trees can be pruned by deleting nodes having the least impact on value, even at intermediate stages. In this way, abstraction methods such as those of (Boutilier and Dearden 1994) can be made far more "adaptive."

Acknowledgements: Thanks to Tom Dean, David Poole and Marty Puterman and an anonymous reviewer for discussions and comments on this paper. This research was supported by NSERC Research Grant OGP0121843, NCE IRIS-II Project IC-7 and Rockwell International. The first two authors also acknowledge the kind support of Rockwell International.

References

Barto, A., Bradtke, S., and Singh, S. 1995. Learning to Act using Realtime Dynamic programming. *Artif. Intel.*, 72:81–138.

Boutilier, C. and Dearden, R. 1994. Using abstractions for decision-theoretic planning with time constraints. *AAAI-94*, pp.1016–1022, Seattle.

Boutilier, C., Dearden, R., and Goldszmidt, M. 1994. Exploiting structure in optimal policy construction. Technical Report 94-23, University of British Columbia, Vancouver.

Chapman, D. and Kaelbling, L. P. 1991. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. *IJCAI-91*, pp.726–731, Sydney.

Darwiche, A. and Goldszmidt, M. 1994. Action networks: A framework for reasoning about actions and change under uncertainty. *UAI-94*, pp.136–144, Seattle.

Dean, T., Kaelbling, L. P., Kirman, J., and Nicholson, A. 1993. Planning with deadlines in stochastic domains. *AAAI-93*, pp.574–579, Washington, D.C.

Dean, T. and Kanazawa, K. 1989. A model for reasoning about persistence and causation. *Comp. Intel.*, 5(3):142–150.

Dean, T. and Wellman, M. 1991. *Planning and Control*. Morgan Kaufmann, San Mateo.

Dearden, R. and Boutilier, C. 1994. Integrating planning and execution in stochastic domains. *UAI-94*, pp.162–169, Seattle.

Howard, R. A. 1971. *Dynamic Probabilistic Systems*. Wiley.

Kushmerick, N., Hanks, S., and Weld, D. 1994. An algorithm for probabilistic least-commitment planning. *AAAI-94*, pp.1073–1078, Seattle.

Poole, D. 1993. Probabilistic Horn abduction and Bayesian networks. *Artif. Intel.*, 64(1):81–129.

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

Puterman, M. and Shin, M. 1978. Modified policy iteration algorithms for discounted Markov decision problems. *Mgmt. Sci.*, 24:1127–1137.

Rivest, R. 1987. Learning decision lists. *Mach. Learn.*, 2:229–246.

Smith, J., Holtzman, S., and Matheson, J. 1993. Structuring conditional relationships in influence diagrams. *Op. Res.*, 41(2):280–297.

Tash, J. and Russell, S. 1994. Control strategies for a stochastic planner. *AAAI-94*, pp.1079–1085, Seattle.

Tatman, J. and Shachter, R. 1990. Dynamic programming and influence diagrams. *IEEE Trans. Sys., Man, Cyber.*, 20:365–379.