

Exploiting Structure in Policy Construction

Craig Boutilier and Richard Dearden

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

Moisés Goldszmidt

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

From: AAAI Technical Report SS-95-07. Compilation copyright © 1995, AAAI (www.aaai.org). All rights reserved.

Abstract: Markov decision processes (MDPs) have recently been applied to the problem of modeling decision-theoretic planning. While such 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 in reflected in a temporal Bayesian network representation of MDPs. The principles behind SPI can be applied to any structured representation of stochastic actions, and 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* [6]. In particular, the theory of *Markov decision processes* (MDPs) has found considerable popularity recently both as a conceptual and computational model for DTP [3, 1, 12].

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 through approximation [3, 1, 5, 12], allowing standard algorithms to be used on reduced state spaces; but two questions remain: a) what if optimal solutions are required? b) what if the state space reduction afforded by existing 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 opti-

mality is strictly required.

- It employs representations of actions and uncertainty modeling techniques that are well known in the AI literature.
- By virtue of being orthogonal to the approximations studied in the literature, it can be combined with any of these techniques.

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) [10]. We present a new algorithm called *structured policy iteration* (SPI) which uses the same computational mechanism as MPI. As in [1], we assume a compact representation of an MDP, in this case using a “two-slice” temporal Bayesian network [4, 2] 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 [11]). 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 grouped by two distinct partitions: 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. Since many problems seem to exhibit tremendous structure, we expect SPI and similar algorithms to be extremely useful in practice (see also [13] for who use influence diagrams to represent MDPs).

We briefly describe MDPs and the MPI algorithm in Sect. 2, followed by our representation of MDPs in Sect. 3. We then describe the SPI algorithm in Sect. 4.

2 Modified Policy Iteration

We assume a DTP problem can be modeled as a *completely observable MDP*. We assume a set of states S , a set of 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 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 . 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:

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

Since the factors $V_\pi(s)$ are mutually dependent, 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 [7] policy iteration algorithm constructs an optimal policy by starting with a random policy and trying to improve this policy by finding for each world some action better than the action specified by the policy. 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

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

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 converge on 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. This can be computationally prohibitive for very large state spaces. However, one can estimate V_π through several steps of *successive approximation*. The value vector V_π is approximated by a sequence of vectors V^0, V^1, \dots that are successively better estimates. The initial estimate V^0 is any random $|S|$ -vector. The estimate $V^i(s)$ is given by

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

Modified policy iteration [10] uses some number of successive approximation steps to produce an estimate of $V_\pi(s)$ at step 1 (see [9] for discussion of *stopping criterion*).

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

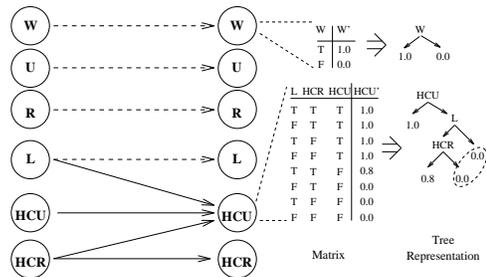


Figure 1: Action Network for DelC

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 his 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, \bar{L} at café), W (robot is Wet), U (robot has Umbrella), R (Raining), HCR (Robot Has Coffee), HCU (User Has Coffee) — and four actions — 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.

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 [4, 2]: we have one set of nodes representing the state of the world at prior to the action, another set representing the world after the action has been performed, and directed arcs representing causal influences between them.¹ Figure 1 illustrates this network representation for the action *DelC* (deliver coffee); we will have one such network for each action.

We assume that the conditional probability matrices of post-action nodes are represented using a *decision tree* (or if-then rules) [11]. 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. Each branch thus determines a partial assignment to $Par(HCU|DelC)$, with some variables unmentioned; the leaf at each branch denotes the probability of HCU being true after *DelC* is executed given any conditions consistent with that branch. 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 $P(s_1, a, s_2)$ that characterize an MDP, we do not require prior probabilities for pre-action variables.

We note that many actions affect only a small number of

¹For simplicity, we only consider binary variables and arcs directed from pre-action variables to post-action variables. Relaxing these assumptions (as long as the network is acyclic) does not complicate our algorithm in any essential ways.

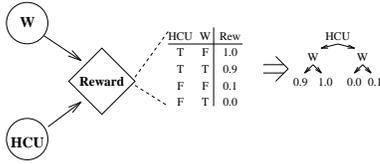


Figure 2: Reward Function Network

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. Figure 2 illustrates such a network. 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).

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 by the standard techniques. But as the number of propositions increase, these methods quickly become infeasible. In addition, although these algorithms may converge in relatively few iterations, memory requirements are quite intensive. If a problem can be represented compactly, the representation must exploit certain regularities and structure in the problem domain. Therefore one can 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.

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 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. 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 an “explicit” policy $\pi(s) = a_i$ iff $s \models \psi_i$. Again, we adopt a tree-structured 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 *clusters*. Figure 5 (the second tree) illustrates a structured policy: we have 4 clusters and 4 action assignments rather one action assignment for each of

64 states.

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 use a tree-structured representation for the partitions. In this case, each leaf is annotated with the value associated with that partition (see Figure 3).

The insights crucial to our algorithm are the following:

- If we have structured policy π and a structured value estimate V^i for π , an improved estimate V^{i+1} can preserve much of this structure;
- 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 and preserves structure. This gives rise to the SPI algorithm for structured policy iteration. We first choose a random structured policy π and proceed as follows:

- Approximate $V_\pi(s)$ using *structured successive approximation*.
- Produce an improved *structured policy* π' (if no improvement is possible, terminate).

We describe these components of the algorithm in turn below.

Rather than start with a random policy (whether structured or not), we initiate the algorithm with a greedy “one-step” policy, and an initial value estimate equivalent to the immediate reward function R . The mechanism for determining this policy will be exactly that used for policy improvement, as we describe below. For now, we adopt the greedy policy $\pi = \{\langle \top, DelC \rangle\}$ (deliver coffee no matter what the state). The initial structured value estimate V^0 (Figure 3) is simply the reward function (Figure 2).

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. This estimate can be a random value tree; but usually will be more informed (e.g., the immediate reward tree). 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 . Recall that

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

Successively better estimates are produced until the difference between V^{i+1} and V^i is (componentwise) below some threshold.²

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

²Better stopping criteria are possible [9], but have no bearing on our algorithm.

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.

Let $Tree(\pi)$ be the tree for a structured policy π and let $Tree(V^i)$ be the tree for a structured value vector, the current estimate of the value of π . We perform the following considerations for each cluster or partition of the state space dictated by the policy tree $Tree(\pi)$. Given an action a to be performed in a certain segment of the state space, we note that the states in that partition can have different values V^{i+1} only if these states can lead (under action a) to different partitions of $Tree(V^i)$ with different probability. Roughly, $Tree(V^i)$ 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 $Tree(V^{i+1})$ we want to *explain* the partitions in $Tree(V^i)$. 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 $Tree(V^i)$.

For example, consider our initial policy $DelC$ and the initial value estimate V^0 (Figure 3). We first notice that V^1 may have different values for states at which $DelC$ affects HCU differently. The partitioning of such states is easily determined: we simply examine the probability tree $Tree(HCU|DelC)$ (see Figure 1). This shows that HCU is affected by the action differently *only under certain conditions*, therefore that V^{i+1} can differ (in this regard) only under these conditions. The second variable that influences V^i is W , and it has impact regardless of whether HCU is true or false. The tree for W (given $DelC$) is appended to each branch of $Tree(HCU|DelC)$. This results in the tree structure shown in Figure 3 labeled V^1 . The values at the leaves are easily determined by the probabilities associated with variables HCU and W under the conditions indicated by the branches (and the values in $Tree(V^0)$).

More precisely, explanations are generated by a process we call *abductive repartitioning*, quite similar in spirit to probabilistic Horn abduction [8]. The algorithm repeats the following procedure for each policy partition (with action a) in $Tree(\pi)$. $Tree(V^i)$ determines an ordering of the relevant (post-action) variables. We construct $Tree(V^{i+1})$ using this ordering. For each relevant variable X , we add an explanation for that X to the leaves of the current tree. The explanation for X under action a is simply the probability tree $Tree(X|a)$ for X in the network for a . 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} . 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. 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, and these can be used to determine where the explanation for the next variable must be placed. We note that if the variable labeling a node of $Tree(X|a)$ occurs 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).

Figure 4 illustrates the steps of SSA in the more complicated construction of $Tree(V^2)$ using $Tree(V^1)$. There are four variables in $Tree(V^1)$, in the order HCU, L, HCR, W . We start by inserting $Tree(HCU|DelC)$ (Step 1) 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 $Pr(HCU) < 1$.³ 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 Step 2). 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 HCU and L hold (as indicated by $Tree(V^1)$). Thus, the addition of $Tree(HCR|DelC)$ must take place 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 in the partial tree (see Step 3). Finally, the variable W must be explained. W is relevant at all points in the partial tree, so $Tree(W|DelC)$ is added (Step 4).

Finally, with the probability labels, the value tree V^1 , the reward tree, and the discounting factor (we assume $\beta = 0.9$), the leaf nodes can be labeled with the values for V^2 (Figure 3). In this example, abductive repartitioning gives rise to six distinct values in the estimates V^1 and V^2 . We note that our tree 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). Thus, only six value computations need be performed rather than 64.

The 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 typically be worth the additional computation, especially as state spaces become (realistically) large — even as domains increase, the effects of actions tend to be localized. 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

³We don’t use $Pr(HCU) = 0$; L is relevant to value whenever HCU is less than certain.

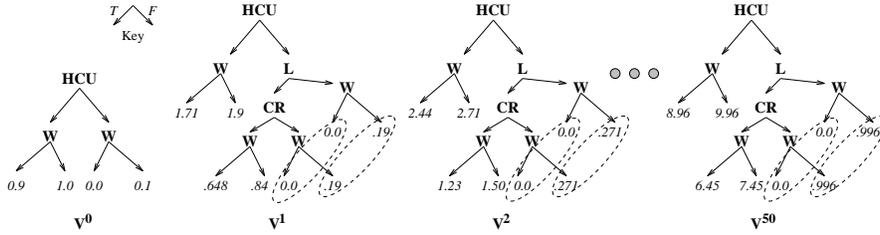


Figure 3: Fifty Iterations of SSA

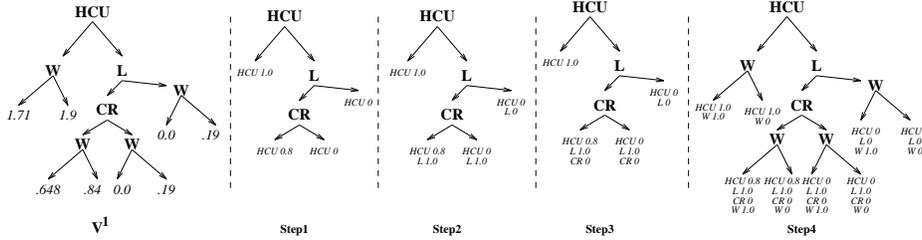


Figure 4: Generation of Abductively Repartitioned Vector

cannot change subsequently.

Theorem 1 Let $\{\varphi_j^i\}$ and $\{\varphi_j^{i+1}\}$ be the partitions constructed for value estimates V^i and V^{i+1} , respectively. If $\{\varphi_j^i\} = \{\varphi_j^{i+1}\}$, then $V^k(s) = V^k(s')$ for any $k \geq i$, and states s, s' such that $s \models \varphi_j^i, s' \models \varphi_j^i$ 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. The value vector V^{50} contains the values that are within 0.4% of the true value vector V_π , as shown in Figure 3.

It is important to note that while the value vector is an approximation the true value function for the specified policy, the approximation is an inherent part of successive approximation. No loss of information results from our partitioning. The structured value estimates V^i are exactly 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. As with SA, this can be a computation- and memory-intensive operation: 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 upon which structured SA was based. For any fixed structured value vector V_π and action a , we can quickly determine the propositions that influence the value of performing that action. As above, actions have different values only when they lead to different clusters of the value partition V_π with different probabilities. 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 $\sum_{t \in S} \Pr(s, a, t) \cdot V_\pi(t)$ is identical. We determine one such partitioning of the state space for each action a and compute the value of a for each partition. The repartitioning algorithm above can be used unmodified.

Figure 5 illustrates the value trees generated by the abductive repartitioning scheme for the two actions *DelC* (the tree V^{50} from Figure 3). 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 as in the algorithm for successive approximation, by using the probability labels generated by abductive repartitioning and the tree for V_π . (We ignore *BuyC* and *GetU*, which are dominated at this stage.) 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).

Space limitations preclude a full discussion of our algorithm, but roughly, we determine the locally improved policy by traversing the two trees, and comparing the values of each action at various clusters. We construct a new policy tree reflecting the dominant actions, as shown in Figure 5. We currently reorder the variables in the trees to facilitate this comparison.

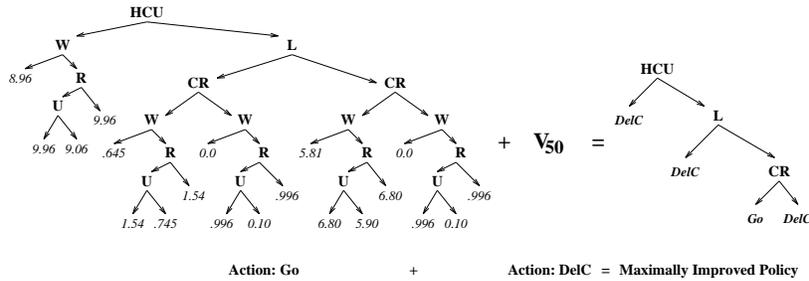


Figure 5: Value Trees for Candidate Actions and the Improved Policy

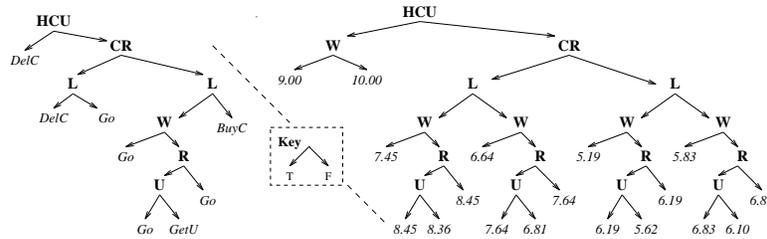


Figure 6: The Optimal Policy and Value Function

The final policy produced by the SPI algorithm (i.e., some number of SSA and improvement steps) is shown in Figure 6 along with the value function it produces. This policy is produced on the fourth iteration. 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 shown 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.

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 embodies 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).

We are currently conducting tests using various strategies for tree generation to verify empirically the cost of the overhead involved in constructing structured tree representations and its tradeoff with reduction in “essential state space.” As state spaces become larger we expect the overhead costs to be-

come insignificant given the state space reductions afforded by eliminating even a small number of variables. We are also exploring acyclic directed graphs instead of trees to represent values and policies, providing further savings.

Acknowledgements: Thanks to Tom Dean, David Poole and Marty Puterman and Nir Friedman for their comments.

References

- [1] C. Boutilier and R. Dearden. Using abstractions for decision-theoretic planning with time constraints. *AAAI-94*, pp.1016–1022, Seattle, 1994.
- [2] A. Darwiche and M. Goldszmidt. Action networks: A framework for reasoning about actions and change under uncertainty. *UAI-94*, pp.136–144, Seattle, 1994.
- [3] T. Dean, L. P. Kaelbling, J. Kirman, and A. Nicholson. Planning with deadlines in stochastic domains. *AAAI-93*, pp.574–579.
- [4] T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Comp. Intel.*, 5(3):142–150, 1989.
- [5] R. Dearden and C. Boutilier. Integrating planning and execution in stochastic domains. *UAI-94*, pp.162–169, Seattle, 1994.
- [6] S. Hanks, S. Russell, and M. Wellman (eds.). *Notes of AAAI Spr. Symp. on Dec. Theor. Planning*. Stanford, March 1994.
- [7] R. A. Howard. *Dynamic Probabilistic Systems*. Wiley, 1971.
- [8] D. Poole. Representing diagnostic knowledge for probabilistic horn abduction. *IJCAI-91*, pp.1129–1135, Sydney, 1991.
- [9] M. L. Puterman. *Markov Decision Processes*. Wiley, 1994.
- [10] M. L. Puterman and M. C. Shin. Modified policy iteration algorithms for discounted Markov decision problems. *Mgmt. Sci.*, 24:1127–1137, 1978.
- [11] J. E. Smith, S. Holtzman, and J. E. Matheson. Structuring conditional relationships in influence diagrams. *Op. Res.*, 41(2):280–297, 1993.
- [12] J. Tash and S. Russell. Control strategies for a stochastic planner. *AAAI-94*, pp.1079–1085, Seattle, 1994.
- [13] J. A. Tatman and R. D. Shachter. Dynamic programming and influence diagrams. *IEEE SMC*, 20(2):365–379, 1990.