

Chapter 2

Markov Decision Processes

The Markov decision process (MDP) model was first introduced in the field of operations research [Bellman, 1957] and significantly developed in subsequent years [Howard, 1960]. An excellent recent text on MDPs is that of Puterman [1994]. The MDP has since been adopted as a model for decision-theoretic planning with fully observable state in the field of artificial intelligence [Bertsekas, 1987; Bertsekas and Tsitsiklis, 1996; Boutilier *et al.*, 1999].

In the MDP model we use in this thesis, an agent is allowed to fully observe the current state and choose an action to execute from that state. Based on that state and action, Nature then chooses a next state according to some fixed probability distribution and the agent receives a corresponding reward. This process repeats itself for some horizon of time steps, possibly infinite. The goal of the agent is to choose its actions so as to maximize the sum of expected discounted future rewards in this model.¹

Given this high-level description of the MDP model, we now proceed to provide a more detailed mathematical definition of an MDP followed by a description of various algorithmic approaches for making optimal sequential decisions in this model. Except where otherwise noted, the following presentation derives from Puterman [1994].

2.1 MDP Representation

Formally, a finite state and action MDP is specified by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, h, \gamma \rangle$. We now describe each of these components in turn, noting that in practice, each must be specified by a domain expert or learned from data.

¹While an agent may seek to maximize other objectives in a general MDP model, we focus on maximizing the sum of expected discounted reward in this thesis.

State space \mathcal{S}

The world is modeled by a set of distinct states \mathcal{S} . In the most general MDP models, \mathcal{S} can be infinite or continuous, but throughout the thesis, we assume a discrete (possibly infinite) state space.

Action space \mathcal{A}

An agent in an MDP can effect changes to its state by executing actions from the set \mathcal{A} . In more general MDP models, \mathcal{A} can be infinite or continuous, but again, we assume a discrete (possibly infinite) action space throughout the thesis. Actions are the only way that an agent can interact with the state and thus the choice of action to take in each state comprises the main decision-theoretic task of the agent.

Transition function \mathcal{T}

In an MDP model, the effects of actions can be uncertain such that for any action $a \in \mathcal{A}$ executed, the world has a fixed probability distribution over transitions to any state in \mathcal{S} . For the purpose of this thesis, the transition function \mathcal{T} will be modeled as a probability distribution $\mathcal{T}(s, a, s') = P(s'|a, s)$, which denotes the probability that the world makes a transition from state $s \in \mathcal{S}$ to $s' \in \mathcal{S}$ given that action a was executed in state s . We note that this representation of the transition function satisfies the Markovian assumptions of an MDP, which require that the distribution over states s_{t+1} at time $t+1$ is independent of any previous state s_{t-i} and action a_{t-i} for $i \geq 1$ given the state s_t and the action a_t taken at time t .

While we typically think of the transition function as dependent only upon the agent's action and the state from which it was taken, there can also be *exogenous events* that are not directly influenced by the agent. For example, as discussed in the SYSADMIN problem in Chapter 1, any computer not explicitly rebooted can independently fail according to some probability distribution. In order to model such exogenous events in this thesis, we will simply fold these implicit probabilistic effects into the transition distribution for each action.

Reward function \mathcal{R}

The preferences of the agent are encoded in a reward function, which for the purpose of this thesis will be restricted to a real-valued range, that is $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. This form of reward function is much more flexible than goal-oriented notions in classical planning; for example,

one can easily model multiple objectives and decision-theoretic reward tradeoffs using different reward values for different states and actions. In a classical planning model, one is typically restricted to specifying a set of equally preferred goal states with state-independent action costs.

Horizon h and discount factor γ

In an MDP, the objective of the agent will be to maximize expected utility accumulated over some time horizon h representing the number of decision steps until termination. While we cover the case for finite h in this chapter, for all subsequent chapters of the thesis, we will assume $h = \infty$ unless otherwise noted.

In the calculation of accumulated reward, we allow for the discounting of rewards t time steps into the future by a discount factor γ^t where $\gamma \in [0, 1]$. Throughout this thesis, we will assume that $\gamma < 1$ unless specifically noted. The use of $\gamma < 1$ allows one to model the notion that an immediate reward r is worth more than the equivalent reward delayed one or more time steps in the future. Such a discounting assumption has both an economic justification as well as an implicit modeling justification for a process that has a $1 - \gamma$ probability of terminating at each step.

Practically, $\gamma < 1$ is required to ensure that the total expected reward is bounded in the case of infinite horizon MDPs. However, if we can make the assumption that the only non-zero reward states in our MDP model are a set of goal states and the system transitions into a zero-reward absorbing state after reaching a goal state, then we can use $\gamma = 1$ in the infinite horizon setting since the total future reward is guaranteed to be bounded.

2.2 Policy Representation

The goal of an agent is to take the action in each state that maximizes the expected accumulated discounted reward criterion over a specified time horizon h . A sequence of actions to be taken can be specified as $\langle \pi_h, \pi_{h-1}, \dots, \pi_1 \rangle$ where each $\pi_t : \mathcal{S} \rightarrow \mathcal{A}$ is a time-dependent action *policy* that specifies an action to take from each state s_t with t -stages-to-go. An important result following from the Markovian property of MDPs is that any policy conditioned on the state or action history from previous decision stages can be represented by an equivalent policy conditioned on only the current state. This follows from the fact that the fully observed state at any stage renders the previous history irrelevant.

An optimal policy $\langle \pi_h^*, \pi_{h-1}^*, \dots, \pi_1^* \rangle$ is a sequence of action policies to be taken that max-

imize the agent's total expected discounted reward over horizon h . Conveniently, for the case of $h = \infty$, there always exists an optimal stationary policy [Howard, 1960]. Thus, no loss of expected discounted reward is incurred for infinite horizon MDPs by restricting our policy representation to a single policy π denoting the action to take from all states at all time stages.

2.3 Optimal Solution Criteria

If the agent's objective is to find the policy that maximizes the expected sum of discounted rewards over a specified time horizon, this objective can be formally expressed as

$$E_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t \cdot r^t \mid s_0 \right] \quad (2.1)$$

where r^t is a reward obtained at time t , γ is a discount factor as defined above, π is a policy as defined previously, and s_0 is the initial starting state. Based on this reward criterion, we define the *value function* for a policy π as the following:

$$V_{\pi}(s) = E_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t \cdot r^t \mid s_0 = s \right]. \quad (2.2)$$

Intuitively, the value function for a policy π is the expected sum of discounted rewards accumulated while executing that policy when starting from state s .

A *greedy policy* π_V w.r.t. a value function V is simply the action policy that takes an action in each state that maximizes expected value w.r.t to V defined as follows:

$$\pi_V(s) = \arg \max_{a \in \mathcal{A}} \left\{ R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) V(s') \right\} \quad (2.3)$$

Thus, from any value function, we can derive a corresponding greedy policy that represents the best action choice w.r.t. that value estimation.

An *optimal policy* π^* in an infinite horizon MDP maximizes the value function for all states. An optimal policy π^* is the greedy policy w.r.t. an optimal value function V^* and likewise the optimal value function is the value under an optimal policy, $V_{\pi^*}(s) = V^*(s)$. We note that V^*

satisfies the following fixed-point equality:

$$V^*(s) = \max_a \left\{ R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s') \cdot V^*(s') \right\}. \quad (2.4)$$

2.4 Exact Solution Techniques

In this section we will discuss exact solution techniques primarily for the case of infinite horizon MDPs. Before we discuss these techniques though, we introduce an alternative matrix notation for MDPs that will simplify portions of the following presentation.

2.4.1 Vector and Matrix Notation

We sometimes write the MDP in vector and matrix form. For each $a \in \mathcal{A}$, we can represent the reward $R(s, a)$ as a column vector R_a indexed by state $s \in \mathcal{S}$. We can represent the value function $V(s)$ as a column vector V indexed by state s . And we can represent the transition function $T(s, a, s')$ for each action $a \in \mathcal{A}$ as a transition matrix T_a row-indexed by current state s and column-indexed by next state $s' \in \mathcal{S}$. In this case, equation 2.4 can be restated as the following:

$$V^* = \max_{\pi} \{ R_{\pi} + \gamma T_{\pi} V^* \} \quad (2.5)$$

In some cases, we will refer to the reward vector and the transition matrix with respect to a policy π as R_{π} and T_{π} , respectively; here the reward value and transition probability for each state corresponds to the action choice indicated by π . Or we may refer to the reward vector and transition matrices restricted to a specific action $a \in \mathcal{A}$ as R_a and T_a , respectively. If needed, π itself can be represented as a vector of actions $a \in \mathcal{A}$ indexed by state and we let Π denote the set of all possible policy vectors.

2.4.2 Dynamic programming

We begin our discussion of dynamic programming by providing two equations that form the basis of the stochastic dynamic programming algorithms used to solve MDPs.

We define $V_{\pi}^0 = R(s)$ and then inductively define the *t-stage-to-go value function* for a policy π as follows:

$$V_{\pi}^t(s) = R(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} T(s, \pi(s), s') \cdot V_{\pi}^{t-1}(s') \quad (2.6)$$

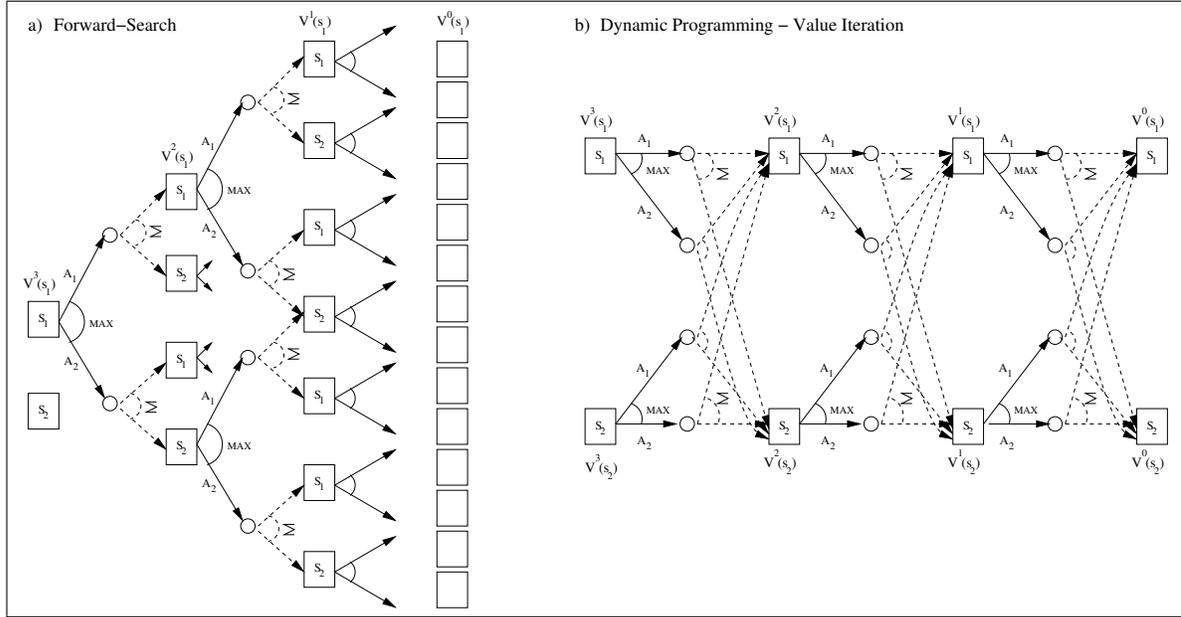


Figure 2.1: A diagram demonstrating a) forward evaluation of the MDP value function and b) dynamic programming regression evaluation of the MDP value function. Both methods return the same value for $V^3(s)$, but the forward evaluation requires exponential time in the search depth $O((|S| \cdot |\mathcal{A}|)^d)$ and only calculates the value for one initial state whereas dynamic programming caches its results on each backup thus requiring only polynomial time in the search depth $O(|S| \cdot |\mathcal{A}| \cdot d)$ and solving for the value function at *every* state.

Based on this definition, Bellman’s *principle of optimality* [Bellman, 1957] establishes the following relationship between the optimal value function at stage t and the optimal value function at the previous stage $t - 1$:

$$V^{t,*}(s) = \max_{a \in \mathcal{A}} \left\{ R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s') \cdot V^{t-1,*}(s') \right\} \quad (2.7)$$

Value iteration

We start with an algorithm known as value iteration that directly implements Equation 2.7. Here, we start with $V^0(s) = \max_a R(s, a)$ and perform the Bellman backup given in Equation 2.7 for each state $V^1(s)$ using the value of $V^0(s)$. We repeat this process for each stage t , producing the backed up value function for $V^t(s)$ from $V^{t-1}(s)$ until we have computed the intended t -stage-to-go value function. This algorithm is demonstrated graphically in Figure 2.1(b).

Often, the Bellman backup is rewritten in two steps to separate out the backup of a value function through a single action and the maximization of this value over all actions. In this

case, we first compute the t -stage-to-go Q-function for every action and state:

$$Q^t(s, a) = R(s, a) + \gamma \cdot \sum_{s' \in \mathcal{S}} T(s, a, s') \cdot V^{t-1}(s') \quad (2.8)$$

Then we maximize over each action to determine the value of the regressed state:

$$V^t(s) = \max_{a \in A} \{Q^t(s, a)\} \quad (2.9)$$

This is clearly equivalent to equation 2.7 but is in a form that we will refer to later since it separates the algorithm into its two conceptual components.

Puterman [1994] shows that terminating once the following condition is met

$$\|V^t - V^{t-1}\|_\infty < \frac{\epsilon(1 - \gamma)}{2\gamma} \quad (2.10)$$

guarantees ϵ -optimality, i.e., $\max_s |V^t(s) - V^*(s)| < \epsilon$. Thus, the greedy policy derived from V^t iteration loses no more than ϵ in value over the infinite horizon in comparison to the optimal policy.

We note that the value iteration approach requires time polynomial in the search depth d , i.e., $O(|S| \cdot |A| \cdot d)$, and solves for the value function at *every* state. Puterman [1994] provides a proof that value iteration converges at a linear rate in terms of the number of iterations.

Policy iteration

At each step of the value iteration backup, we are implicitly performing a policy update, determining the best action to take from every state in order to maximize reward. Another approach to dynamic programming is known as policy iteration [Howard, 1960] and is summarized in the following algorithm:

1. *Initialization:* Pick an arbitrary initial decision policy $\pi_0 \in \Pi$ and set $i = 0$.
2. *Policy Evaluation:* Solve for V_{π_i} (see below).
3. *Policy Improvement:* Find a new policy π_{i+1} that is a greedy policy w.r.t. V_{π_i} (i.e., $\pi_{i+1} \in \arg \max_{\pi \in \Pi} \{R_\pi + \gamma T_\pi V_{\pi_i}\}$ with ties resolved via a total precedence order over actions).
4. *Termination Check:* If $\pi_{i+1} \neq \pi_i$ then increment i and go to step 2 else return π_{i+1} .

We note that the policy evaluation of a *fixed* policy π reduces to the solution of a linear system since the MDP reduces to a simple Markov chain. Thus, we can solve for V_π by computing the right-hand side of the following equation:

$$V_\pi = R_\pi(I - \gamma T_\pi)^{-1} \quad (2.11)$$

We note that a unique solution for V_π always exists since the Markovian properties of T_π guarantee that $I - \gamma T_\pi$ is invertible. We note that solving for V_π directly using matrix inversion takes time $O(|S|^3)$. Alternately, we can solve for V_π using *successive approximation*, which initializes $V_\pi^0 = R_\pi$ and iteratively computes V_π^t from V_π^{t-1} using Equation 2.6 until $V_\pi^t = V_\pi^{t-1}$ (where $V_\pi = V_\pi^t$).

Once policy iteration has terminated, the final policy returned is the optimal policy π^* and the value function corresponding to this policy is the optimal value function V^* . Puterman [1994] provides conditions and a proof of a superlinear rate of convergence for policy iteration.

So far, we have implicitly assumed that the above algorithms perform synchronous updates, that is, we are updating the value function in value iteration for all states and that we are improving the policy in policy iteration for all states. We additionally note that there are a number of asynchronous variants of value and policy iteration that do not update the value or improve the policy at every state on all iterations, yet still retain similar convergence properties. These algorithm variants are discussed by Puterman [1994] and Bertsekas and Tsitsiklis [1996] and are extremely useful for proving convergence properties of the reinforcement learning [Barto and Sutton, 1998] and real-time search [Barto *et al.*, 1993] approaches to solving MDPs. However, we do not discuss asynchronous methods further as they are not directly relevant to the methods we employ throughout the rest of the thesis.

Modified policy iteration

A comparison of the two previous algorithms reveals that they occupy two extremes in terms of policy updates: value iteration performs an implicit policy update in order to compute every intermediate value function whereas policy iteration performs an update only after solving directly for V_π .

If we interpolate between these two approaches, we arrive at an algorithm known as modified policy iteration [Puterman and Shin, 1978]. In this algorithm, we simply iterate between policy evaluation and policy improvement phases until our policy is ϵ -optimal using the same

terminating criteria as value iteration. The algorithm is very similar to policy iteration with the exception of the policy evaluation phase replaced by an approximate version:

1. *Initialization:* Pick an arbitrary initial decision policy vector $\pi_0 \in \Pi$ and set $i = 0$.
2. *Approximate Policy Evaluation:* Solve for V_{π_i} using some number of steps of successive approximation.
3. *Policy Improvement:* Find a new policy π_{i+1} that is the greedy policy w.r.t. V_{π_i} .
4. *Termination Check:* If $\pi_{i+1} \neq \pi_i$ then increment i and go to step 2 else return π_{i+1} .

Algorithm convergence requires only that the policy approximation phase does not increase the error of the value estimate from the previous iteration, i.e.,

$$\|V^* - V_{\pi_{i+1}}\| \leq \|V^* - V_{\pi_i}\| \quad (2.12)$$

Such a property holds, for example, by initializing the value estimate with V_{π_i} and then performing one or more steps of successive approximation under the policy π_{i+1} .

A proof of superlinear convergence rate for modified policy iteration under certain conditions is given by Puterman [1994]. Puterman also notes that modified policy iteration often empirically requires less computation time than both value and policy iteration.

2.4.3 Forward-search

If we reexamine Equation 2.7, we note that we could compute this recurrence in a forward-search manner by starting at an initial state and unfolding the recurrence to horizon h and then computing the expectation and maximization as we return to the initial state. A graphical representation of the unfolding of this computation is shown in Figure 2.1(a). We note that determining the value $V^h(s)$ for a *single* state using this method requires time exponential in the search depth h , that is, $O((|\mathcal{S}| \cdot |\mathcal{A}|)^h)$.

Since we are performing forward search to a fixed *a priori* search depth, we can determine the minimum horizon h to search if we want an ϵ bound on the maximum error of our value function, given knowledge of our discount factor γ and our maximum reward $R_{max} = \max_{s,a} R(s, a)$:

$$h \geq \log_{\gamma} \left(\frac{\epsilon(1 - \gamma)}{R_{max}} \right) - 1 \quad (2.13)$$

2.4.4 Real-time dynamic programming

The real-time dynamic programming (RTDP) framework [Barto *et al.*, 1993] is a hybrid approach that combines real-time forward search with dynamic programming. This approach uses limited depth, forward-search backups to update the value function of the set of states visited during on-line trials, assuming that initial states were generated according to some fixed distribution. The policy used for the trials is the optimal policy for the current value function. Since backed-up and cached values from one step are used by other steps, this approach mixes the forward-search and dynamic programming paradigms. It is provably convergent and has the advantage that it only derives the value function for the set of states reachable from the initial state distribution. This can often be more efficient than synchronous dynamic programming approaches when the set of reachable states is small compared to the total number of states.

2.4.5 Linear programming

An MDP can also be solved by formulating it as the optimization of a linear program (LP). The fact that such a solution exists follows from the notion that the optimal policy and value function must satisfy the following inequalities for all states as implied by Equation 2.4:

$$V^*(s) \geq \max_{a \in \mathcal{A}} \left(R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V^*(s') \right); \forall s \in \mathcal{S} \quad (2.14)$$

This equality in turn implies the following conditions:

$$V^*(s) \geq R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V^*(s'); \forall a \in \mathcal{A}, s \in \mathcal{S} \quad (2.15)$$

While this latter set of inequalities only enforces one side of the optimal value function fixed-point equality given in Equation 2.4, it turns out that finding the minimal V^* under an \mathcal{L}_1 metric that satisfies these constraints suffices to enforce the other side of the inequality. Thus, the optimal value function can be computed by the following primal specification of a linear

program [Puterman, 1994]:

$$\begin{aligned}
 &\text{Variables: } V \\
 &\text{Minimize: } \|V\|_1 \\
 &\text{Subject to: } 0 \geq R_a + \gamma T_a V - V, \forall a \in \mathcal{A}
 \end{aligned} \tag{2.16}$$

Puterman [1994] provides a proof that this formulation is guaranteed to produce an optimal value function for an MDP. Puterman also notes that solving the dual LP formulation is often more efficient than solving the primal LP formulation. However, we do not present the dual formulation here as we work directly with the primal formulation and its variants throughout the thesis.

2.5 Approximate Solution Techniques

As the number of states and actions in an MDP grows, it often becomes necessary to explore approximate solutions in the face of intractability of exact solutions. While approximation in MDPs can take many forms, it is frequently carried out by considering restricted representations of the value function. Some methods for restricting the value function representation will become relevant once we introduce structured descriptions of our MDP models and solution algorithms. However, a very general and popular approximate solution technique for MDPs is that of linear-value function approximation [Schweitzer and Seidmann, 1985; Tsitsiklis and Van Roy, 1996; Koller and Parr, 1999a; Koller and Parr, 1999b; Schuurmans and Patrascu, 2001; Guestrin *et al.*, 2002], which we discuss at length in this section.

2.5.1 Linear-value Function Representation

Representing value functions as a linear combination of basis functions has many convenient computational properties, many of which will become evident as we incorporate factored and relational structure in our MDP model. However, perhaps one of the most important aspects for the work we present here is that linear-value function representations lead to MDP solution formulations using optimization w.r.t. linear objectives and linear constraints — that is, the well-studied case of linear program (LP) optimization. Since many robust off-the-shelf LP solvers are available, this makes such approaches attractive for practical implementation purposes.

If we have n states in our MDP, the exact value function can be specified as a vector in \mathbb{R}^n . This vector can be approximated by a value function $\tilde{V}_{\vec{w}}$ that is a linear combination of k fixed basis function vectors denoted $b_i(s)$ as follows:

$$\tilde{V}_{\vec{w}}(s) = \sum_{i=1}^k w_i \cdot b_i(s) \quad (2.17)$$

The linear subspace spanned by the basis set might not include the actual value function, but one can use projection methods to minimize some error measure between the actual value function and the linear combination of basis functions.

The basis functions themselves can be specified by domain experts, constructed or learned in an automated fashion (e.g., [Poupart *et al.*, 2002a; Mahadevan, 2005]). We will consider more structured forms of automated basis function construction as we introduce structured MDP representations in subsequent chapters.

On a final note, we mention that there are a variety of other general function approximation such as nonlinear functions or neural nets [Bertsekas and Tsitsiklis, 1996] but it is generally difficult to provide useful convergence properties for such approximation architectures so we do not discuss them further in this thesis.

2.5.2 Error Bounds on Approximate Value Functions

Once a set of basis functions has been specified, the problem of finding an approximate value function reduces to the problem of finding a good set of weights that closely approximates the optimal value function. One way of measuring the *a posteriori* quality of an approximated value function $\tilde{V}_{\vec{w}}$ is by evaluating the Bellman error β (i.e., the L_∞ norm of the Bellman residual) of the value function under the MDP dynamics:

$$\beta = \max_{s \in \mathcal{S}} \left| \tilde{V}_{\vec{w}}(s) - \max_{a \in \mathcal{A}} \left(R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) \tilde{V}_{\vec{w}}(s') \right) \right| \quad (2.18)$$

Of course we note that when the Bellman error is zero, this equation satisfies the fixed-point equation for the optimal value function given in Equation 2.4 and thus $\beta = 0$ indicates that $\tilde{V}_{\vec{w}} = V^*$.

Let $\tilde{\pi}$ be the greedy policy w.r.t. the value function approximation $\tilde{V}_{\vec{w}}$. Once β is known for $\tilde{V}_{\vec{w}}$, it is then easy to bound the max-norm (\mathcal{L}_∞) error of $V_{\tilde{\pi}}$ w.r.t. the optimal value function

using the following inequality [Williams and Baird, 1994]:

$$\|V^* - V_{\tilde{\pi}}\|_{\infty} \leq \frac{2\gamma\beta}{1-\gamma} \quad (2.19)$$

Thus, in all of the following approximation techniques, we will have some way of determining a maximum bound on the loss of our approximation.

2.5.3 Approximate Dynamic Programming

Approximate dynamic programming techniques are simply extensions of the previous dynamic programming algorithms with additional approximation steps. While these approximation steps do not guarantee convergence, an *a posteriori* analysis of the Bellman error of a value function can show that the value function estimate has converged within some error bound.

Approximate Value Iteration

Approximate value iteration (AVI) is precisely the value iteration algorithm previously presented with the additional step that after each Bellman backup, the value function may be projected onto a more compact representation. Since we are focusing on linear-value function approximation in this section, we will cover the case of projecting the one-step Bellman backup onto a linear-value function representation.

In AVI using linear-value approximation, we begin by initializing the weights \vec{w}^0 of our initial linear-value function representation $\tilde{V}_{\vec{w}}^0$ in some way — perhaps with $\vec{w}^0 = \vec{0}$ or with \vec{w}^0 set so that $\tilde{V}_{\vec{w}}^0 = \max_a R_a$ (if our linear-value function representation permits this). Then we perform the standard Bellman backup given in Equation 2.7 to obtain V^1 . Since the dynamics of our MDP do not guarantee that our linear-value function representation spans the space of V^1 , it will be necessary to project V^1 onto the space of $\tilde{V}_{\vec{w}}^1$, which we discuss in a moment. This process can proceed indefinitely in AVI, obtaining V^t from $\tilde{V}_{\vec{w}}^{t-1}$ and projecting V^t to obtain $\tilde{V}_{\vec{w}}^t$ until some predefined stopping criterion such as a maximum limit on iterations or Bellman error bound has been met.

Perhaps the most obvious choice for projecting V^t to obtain $\tilde{V}_{\vec{w}}^t$ in AVI is the following where \vec{w}^* represents the weights for the optimal projection and n is the error norm \mathcal{L}_n being minimized in the projection:

$$\vec{w}^* = \arg \min_{\vec{w}} \left\| V^t - \tilde{V}_{\vec{w}}^t \right\|_n \quad (2.20)$$

Tsitsiklis and Van Roy [1996] show that minimizing the Euclidean-distance (\mathcal{L}_2) error can diverge — even when $\tilde{V}_{\vec{w}}^t$ spans the space of the optimal value function. Likewise, Guestrin, Koller, and Parr [2001] discuss similar issues with the divergence of AVI for the case of the max-norm (\mathcal{L}_∞) error minimizing projection. However, these divergence issues can be mitigated in practice if additional basis functions are introduced to minimize the projection error.

Approximate Policy Iteration

Approximate policy iteration (API) with linear-value function approximation is another variant of dynamic programming that uses a different projection step. The benefit of API is that under an \mathcal{L}_∞ projection step, its error can be shown to be bounded for all iterations, thus avoiding the divergence issues of AVI [Guestrin *et al.*, 2001].

The API algorithm follows the policy iteration algorithm provided previously, except that the value determination step is now approximate rather than exact. After starting with an initial arbitrary policy π_0 , policy iteration iterates between the following two steps where the projection is in terms of the \mathcal{L}_n norm:

$$\vec{w}^i = \arg \min_{\vec{w}} \left\| R_{\pi_i} + T_{\pi_i} \tilde{V}_{\vec{w}} - \tilde{V}_{\vec{w}} \right\|_n \quad (2.21)$$

$$\pi_{i+1} = \arg \max_{\pi \in \Pi} \left\{ R_\pi + \gamma T_\pi \tilde{V}_{\vec{w}^i} \right\} \quad (2.22)$$

Koller and Parr [1999b] provide an API algorithm based on minimizing a weighted \mathcal{L}_2 norm in the projection step. In subsequent work, Guestrin, Koller and Parr [2001] presented the following LP intended to directly minimize the \mathcal{L}_∞ norm in the projection step:

$$\begin{aligned} &\text{Variables: } \vec{w} \\ &\text{Minimize: } \beta \\ &\text{Subject to: } \beta \geq \left| R_{\pi_i} + T_{\pi_i} \tilde{V}_{\vec{w}} - \tilde{V}_{\vec{w}} \right| \end{aligned} \quad (2.23)$$

One nice advantage of directly minimizing the L_∞ norm in the projection step is that when API converges (i.e., $\pi_i = \pi_{i-1}$ or equivalently $\vec{w}^i = \vec{w}^{i-1}$), the objective β for the final LP solution of Equation 2.23 is the Bellman error of the approximated value function. Thus a bound on the error of the approximated value function is immediately available by plugging β directly into Equation 2.19 [Guestrin *et al.*, 2001].

2.5.4 Approximate Linear Programming

Approximate linear programming (ALP) is simply an extension of the linear programming solution of MDPs to the case where the value function is approximated. In a linear-value function representation, the objective and constraints will be linear in the weights being optimized and thus the linear programming framework can still be used. Consequently, we arrive at the following variant of the LP in Equation 2.16 that simply takes into account the linear-value function representation:

$$\begin{aligned}
 \text{Variables: } & \vec{w} \\
 \text{Minimize: } & \|\tilde{V}_{\vec{w}}\|_1 \\
 \text{Subject to: } & 0 \geq R_a + \gamma T_a \tilde{V}_{\vec{w}} - \tilde{V}_{\vec{w}}, \forall a \in \mathcal{A}
 \end{aligned} \tag{2.24}$$

2.6 Application to AI Planning Problems

We focus on MDPs as a model for decision-theoretic planning since they generalize many of the planning paradigms found in the literature. First we review some of these planning paradigms and then proceed to a discussion of two general classes of MDP problems, one oriented towards a decision-theoretic extension of classical task-oriented planning and the other oriented towards a non-terminating process model with a long-term reward optimization objective, but no clear definition of a single task or goal.

2.6.1 Common AI Planning Paradigms

As mentioned previously, classical planning can be viewed as a restricted case of decision-theoretic planning in MDPs where all actions are deterministic and the reward is goal-oriented, that is, there is only one non-zero reward value that is specified for a set of absorbing goal states. Typically the initial state is known, thus making observability a moot issue — with a known initial state and deterministic actions, the state of the world after *any* action sequence will be known with certainty.

In classical planning the objective is simply to find a sequence of actions that will lead to a goal state from the initial state. There may be an emphasis placed on finding shorter plans, or more generally there may be costs associated with actions and the use of an objective criterion that minimizes cost-to-goal. Nonetheless, all of these variants can be modeled in the MDP

framework. However, this does not mean that standard MDP solution algorithms are particularly well-suited for classical planning; while standard exact MDP solution algorithms will provide an optimal policy in the case of classical planning, this optimal policy is provided for *all* states. However due to the known initial state and determinism of action effects, solutions to classical planning can be specified via straight-line sequences of actions that may touch on only a very small subset of the total state space. Thus, the full policy provided by exact MDP solution algorithms will be inefficient compared to deterministic planners in the classical planning paradigm that can exploit knowledge of the initial state and action constraints to avoid searching through all states. Weld [1999] provides an excellent overview of many recent advances in classical deterministic AI planning along these lines.

A related topic is that of optimal deterministic planning, which uses a similar framework as classical AI planning (i.e., known initial state and deterministic action effects), but relaxes the goal-oriented notions to a much richer set of preferences over goals and resource constraints (see e.g., [Haddawy and Hanks, 1998; Williamson and Hanks, 1994; Brafman and Chernyavsky, 2005]) and even temporally extended preferences (see e.g., [Bienvenu *et al.*, 2006; Baier *et al.*, 2007] for some recent work and a discussion of related approaches in this area). The task here is to find an optimal plan that takes into account the preferences and constraints. Since these approaches use a rich notion of preferences and assumptions, there does not necessarily exist a direct correspondence to the scalar-reward MDP framework discussed in this chapter. Nonetheless, notions of reward in the MDP framework defined here can capture some aspects of optimal deterministic planning.

A number of planning problems in AI involve partial observability and thus cannot be solved in the MDP framework presented here. Two notable problems are variants of conformant planning. In conformant planning [Cimatti and Roveri, 1999] the initial state is restricted, but strictly unknown and actions have non-deterministic effects with no (or in some variants, partial) observability. Probabilistic conformant planning is similar except that strict uncertainty in the initial state and action effects are replaced with known probability distributions [Kushmerick *et al.*, 1995]. Nonetheless, the partial observability assumptions of conformant planning and many other partially observable problems prevent them from being modeled or solved within the MDP framework presented here.

2.6.2 Task- vs. Process-oriented Planning

Most classical AI planning problems exhibit the characteristic of being goal-oriented, even when there are multiple goals and relative preferences over those goals. The BOXWORLD problem from Chapter 1 is a good example of a such a task-oriented problem: there are a number of boxes that need to be delivered to their destination and once this is achieved, the problem terminates. While many task-oriented decision-theoretic planning problems can be modeled as MDPs with some form of absorbing goal state, this is only one possible class of problems.

There are many problems that are continuous processes without a clearly defined notion of goal or termination, but rather a continuously accumulating reward over an infinite horizon. The SYSADMIN problem from Chapter 1 is an exemplar of this class of problems. Recalling the SYSADMIN description, the objective was to maximize the count of computers running per time step under an infinite horizon discounted reward criterion. However, given that any computer can independently fail at any time step if not rebooted due to exogenous events, the task has no clear criterion for termination since no state can persist indefinitely. Fortunately, this ongoing process-oriented problem is well-modeled as the optimization and solution of an infinite-horizon discounted reward MDP.

As mentioned in Boutilier *et al.* [1999], many real-world problems exhibit both task- and process-oriented behavior. And the beauty of the MDP framework is that it can accommodate both forms of MDP models and it can seamlessly combine them, if needed. Thus, we can not only accurately model decision-theoretic planning problems based on the classical task-oriented paradigm, but we can encapsulate these task-oriented problems in a more realistic ongoing optimization process with random exogenous events. These types of combined task- and process-oriented models more accurately reflect the problems than an agent would likely have to contend with while acting in a realistic world model.

2.7 Summary

We have motivated the decision-theoretic planning paradigm and cast the framework in an MDP setting. And we have covered all of the groundwork for the MDP solution techniques that we develop in this thesis. Among these solutions, there are two important choices to consider. The first choice is whether to use iterative dynamic programming methods or direct linear program optimization techniques. The second choice is whether to use exact or approximate

solution methods.

It is not entirely clear when to use dynamic programming algorithms vs. direct linear program optimization techniques. While Puterman [1994] cites Koehler [1976] in reporting that dynamic programming based modified policy iteration techniques can outperform direct linear programming techniques by as much as 10 times, Trick and Zin [1997] report exactly the opposite case, perhaps owing to their use of the more recently available and highly optimized ILOG CPLEX LP solver.

The second choice of exact vs. approximate is almost invariably determined by the size of the state space. If the state space is relatively small then one can easily resort to exact methods. However, if the state space is sufficiently large, approximate solution techniques are the only viable option. But this last statement depends critically on *how one measures the size of the state space*.

Looking ahead to future chapters, we note that there is only so much computational advantage that can be gained by using the approximate solution techniques in place of the exact techniques covered in this chapter. That is, all exact and approximate solution techniques mentioned here must represent the value function and policy (if required) as vectors or functions over an explicitly enumerated state space. As it turns out, there are many representations well suited to decision-theoretic planning tasks that do not require explicit state enumeration in the problem representation or in the solution. As such, the use and exploitation of structured representations is complementary to the choice of exact vs. approximate solution method or dynamic programming vs. direct linear program optimization. That is, the exploitation of structure can help all of these methods scale far beyond what is possible with approaches that rely on explicit state enumeration.

Thus, the modeling and exploitation of decision-theoretic planning structure in the MDP framework will be the core focus of the remainder of this thesis.

Chapter 3

Factored MDPs

In the MDP representation of the previous chapter we expressed the reward, transition distribution, policy, and value function all in terms of an explicitly enumerated state space. However, this is neither the most natural nor the most compact representation one can choose, nor can it be easily exploited in solution methods.

Intuitively, we often think of states in terms of various state properties. That is, a state representation can be factored into a number of properties that we will call *state variables* where each of these state variables can take assignments from a set of possible values. For example, a state variable may be the location of an object and it may take assignments from a small set of locations (e.g., office, hallway, or cafeteria). If there are a number of objects, we may choose to represent the location of each object with a different state variable. In this case of multiple state variables, states can be considered to be a joint configuration of all state variables. As we will show in the first half of this chapter, it is not only natural to represent MDPs in this factored manner, but state variable factoring can also result in compact representations that can be exploited by solution methods to avoid explicit state enumeration.

In the second half of this chapter, we will review a number of methods for exploiting factored MDP structure in extensions of solution algorithms from the previous chapter. We will also introduce the first contribution of this thesis, which is a compact data structure termed the affine algebraic decision diagram (AADD) that can compactly and simultaneously exploit multiple forms of independence in the representation and solution of factored MDPs.

3.1 Factored MDP Representations

While the MDP solution techniques from the previous chapter all require time at least polynomial in $|\mathcal{S}|$ and $|\mathcal{A}|$, we note that $|\mathcal{S}|$ can be very large. To see this, recall the SYSADMIN problem from Chapter 1 where the state can be represented by n binary state variables x_1, \dots, x_n where each state variable $x_i \in \mathcal{X}_i$ (with $\mathcal{X}_i = \{true, false\}$) represents whether computer i is running or not. In this problem, the total number of states is 2^n (i.e., $|\mathcal{S}| = |\{\mathcal{X}_1 \times \mathcal{X}_2 \times \dots \times \mathcal{X}_n\}|$). This is Bellman’s well-known curse of dimensionality [Bellman, 1957] and it unfortunately implies that the enumerated state solution methods discussed in the last chapter require time exponential in the number of state variables n . Obviously, such enumerated state solution approaches would be computationally prohibitive for as few as 50 computers.¹

Consequently, efficient representations and algorithms are extremely important for the solution of MDPs for realistic problems. This is especially true for fields such as decision-theoretic planning where 50 binary state variables would be considered at most an intermediate-sized problem.² In the following sections, we describe structured representations and algorithms that mitigate the problems associated with enumerated state MDP representations and solution algorithms, thus vastly increasing the size of MDPs that can be practically solved exactly or approximately.

3.1.1 Factored Transition and Reward Dynamics

One of the major representational bottlenecks in MDPs stems from representing the transition matrices. For example, with a state in SYSADMIN formed from $n = 3$ binary variables, the joint transition distribution would be of the form $P(x'_1, x'_2, x'_3, x_1, x_2, x_3, a)$ (with the x'_i variables representing the next-state variables and $a \in \mathcal{A}$ representing the three actions to reboot each of the three computers). If this probability distribution was represented in an enumerated manner, it would require $|\mathcal{A}| = 3$ matrices of row and column dimension 2^3 for a total of 2^6 entries per matrix. Clearly, it would become prohibitively difficult to store these matrices as more variables were added as the number of matrix entries scales exponentially

¹For reference, using one byte of storage per enumerated state in an MDP with 50 variables would require one petabyte of storage, far beyond what could reasonably be stored in primary or secondary storage on a modern desktop computer.

²For example, in the 2006 ICAPS International Probabilistic Planning Competition, the largest problems in the ELEVATOR domain had well over 350 binary state variables if a binary variable were instantiated for each ground relational fluent. This amounts to over 2^{350} distinct states.

with the number of state variables n .

However, from an intuitive standpoint, most actions affect only a small subset of state variables, which can be exploited in a factored representation of the transition distribution. A *dynamic Bayes net (DBN)* [Dean and Kanazawa, 1989] serves as an appropriate representation in this case. Using a DBN, we can specify the effects of an action on an individual computer conditioned on the relevant state variables. Let us assume that our three computers in SYSADMIN are connected in a unidirectional ring³, thus having the network configuration and DBN transition function representation in Figure 3.1(a). We can then specify the *conditional probability tables (CPTs)* in the DBN where the next state of each computer x'_i is conditioned on the computer's previous state x_i , the computer x_{i-1} to which it has an upstream connection, and the action (specifically whether x_i was rebooted by the action $reboot(i)$):⁴

$$P(x'_i = true | \vec{x}_i, a) = \begin{cases} a = reboot(i) : & 1 \\ a \neq reboot(i) \wedge x_i = true : & .475 \cdot (\mathbb{I}[x_{i-1}] + 1) \\ a \neq reboot(i) \wedge x_i = false : & .025 \cdot (\mathbb{I}[x_{i-1}] + 1) \end{cases} \quad (3.1)$$

In words, this states that a computer is running in the next state with probability 1 if it was rebooted, or otherwise with a probability that is most impacted by the computer's previous state and somewhat less by the previous state of its upstream connection. The exact numerical values chosen here are taken from the SYSADMIN specification in *Guestrin et al.* [2002].

We can use a factored representation in the spirit of influence diagram [Howard and Matheson, 1984] representations to model the state variables that influence the reward function. This is also shown in Figure 3.1(a).

For this DBN, we can then write the full conditional joint transition distribution in the following factored form:

$$P(x'_1, x'_2, x'_3 | x_1, x_2, x_3, a) = P(x'_1 | x_1, x_3, a) \cdot P(x'_2 | x_1, x_2, a) \cdot P(x'_3 | x_2, x_3, a)$$

We note that the full conditional joint distribution for a single action would take 192 entries to represent as a fully enumerated CPT while the factored representation requires tables with a total number of 72 entries given the conditional independence assumptions. As the number

³Formally, in a unidirectional ring, each computer x_i has one incoming connection from x_{i-1} where subtraction is modulo n .

⁴The notation $\mathbb{I}[\cdot]$ is an indicator function that takes the value 1 when its argument evaluates to true and 0 when it evaluates to false.

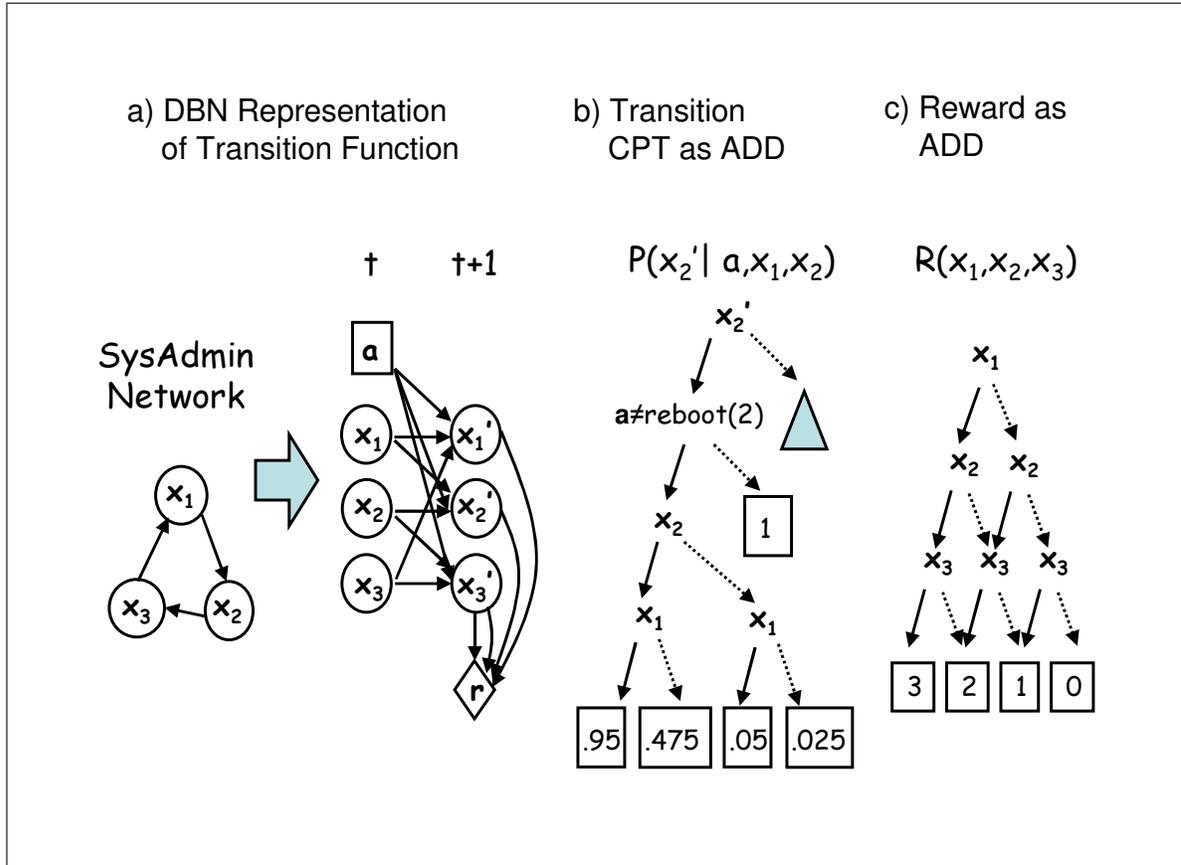


Figure 3.1: a) A dynamic Bayes network and decision diagram representing a transition function and a reward function for SYSADMIN with $n = 3$ and a unidirectional ring network topology. b) An compact encoding of the transition function CPT for the DBN as an ADD. Note that x'_3 sums to one over all possible previous states. c) An ADD representation of the additive reward function for SYSADMIN. For all ADDs, the high (true) edge is solid, the low (false) edge is dotted.

of computers n in this unidirectional network topology increases, the size of the full joint representation will scale exponentially in n while the size of the DBN representation will scale only quadratically in n (requiring n CPTs each with $8n$ entries).

Throughout this exposition, we assume that the DBN representation of the transition function does not have synchronic arcs that specify dependences between post-action variables. However, if needed, it is easy to modify our DBN notation to permit such arcs and the forthcoming algorithms to take such arcs into account during inference. Or alternately, one may choose to modify the problem description to use joint variables in place of variables connected via synchronic arcs. This approach incurs a representational blowup exponential in the number of variables joined, but converts a DBN with synchronic arcs to an equivalent (but larger) DBN

without synchronic arcs.

We also note that there are alternative representations to the DBN transition representation such as probabilistic generalizations of STRIPS operators [Boutilier *et al.*, 1995a]. However, Littman [1997] proved that this representation can be converted to a dynamic Bayes net representation with only a polynomial blowup in size. This effectively demonstrates that both formalisms are representationally equivalent.

In the general case, using DBN and influence diagram structures to efficiently represent transition and reward dependencies often saves a considerable amount of space in these representations. Defining the *parents* of a next-state variable x'_i in the DBN representation as the set of current-state variables $\{x_j\}$ appearing in a CPT with x'_i , we note that in the worst case, every x'_i has all $\{x_1, \dots, x_n\}$ as parents, thus requiring a number of parameters exponential in n . In the best case, every state variable x'_i has only x_i as a parent, requiring a number of parameters linear in n . However, even in the typical case, if the number of parents of any state variable is bounded by some constant $k < n$, this requires $O(n \cdot 2^k)$ parameters in the case of binary state variables — still an exponential reduction over the worst case. While a factored transition and reward representation can yield substantial savings for the MDP representation, we note that this factoring cannot often be preserved in the value function due to the correlation of action effects over sufficiently extended periods of time [Boutilier *et al.*, 1995b]. Nevertheless, representing large MDPs is a first step toward solving them and subsequent techniques will take advantage of this factored structure for efficient computation and approximation.

3.1.2 Context-specific Independence and ADDs

Even if we can represent the joint transition probability as a Bayes net with a conditional probability table (CPT) for each next-state variable, we can often represent these tables more efficiently than by enumerating all state configurations of the variables in that table. Quite often, we find that certain values of variables in a CPT render the other values irrelevant. This is known as *context-specific independence (CSI)* [Boutilier *et al.*, 1996].

For the example DBN in Figure 3.1(a), given that the value of x'_2 depends on x_1, x_2 and a in $P(x'_2|x_1, x_2, a)$ but that in the context of $a \neq \text{reboot}(2)$, the value of x'_2 depends on no other variables, we say that in the context of $a \neq \text{reboot}(2)$, x'_2 is independent of all other variables and thus $P(x'_2|x_1, x_2, a \neq \text{reboot}(2)) = P(x'_2|a \neq \text{reboot}(2))$. In order to represent this CSI compactly, we can use a decision tree or an algebraic decision diagram (ADD) [Bahar *et al.*, 1993], which is similar to a tree except that it is a canonical *directed acyclic graph (DAG)* with

all variable decision tests following a strict order from the root to the leaves. An example ADD for this probability distribution showing the above CSI is given in Figure 3.1(b). Effectively, CSI performs automatic state aggregation in that all possible state contexts under the condition $a \neq \text{reboot}(2)$ are effectively grouped together and assigned a common value. An example ADD for the reward is given in Figure 3.1(c), here there is no explicit CSI, but the reconvergent DAG structure of the ADD does allow sharing of common substructure that reduces what would be a tabular representation exponentially sized in n to an ADD representation quadratically sized in n .

In addition to the representational efficiency of state aggregation in ADDs, we note that computation with ADDs can also be very efficient. When we perform operations on factors represented as ADDs, we can just replace these operations with their ADD-based versions [Bahar *et al.*, 1993], allowing us to exploit CSI and shared substructure not only in the representation of factored MDPs, but also in the computations required for their solution.

Since the ADD will be a crucial data structure for our subsequent presentation of factored MDP solution algorithms, we provide a formal definition of ADDs and algorithms to construct and manipulate them in the following subsections. The following discussion draws on the work of Bahar *et al.* [1993], which is itself a slight variant of the original work on ordered *binary decision diagrams (BDDs)* of Bryant [1986].

Canonical Reduced ADDs

An ADD is a decision diagram with a fixed variable ordering of all decision tests on paths from the root to the leaves that is capable of representing functions from $\mathbb{B}^n \rightarrow \mathbb{R}$. We define ADDs with the following simple BNF grammar:

$$F ::= C \mid \text{if } (F^{var}) \text{ then } F_h \text{ else } F_l \quad (3.2)$$

Here, $C \in \mathbb{R}$ is a constant-valued terminal node. Each internal decision node is represented as $\text{if } (F^{var}) \text{ then } F_h \text{ else } F_l$ and is associated with a single variable var that indicates the high branch leading to node F_h should be taken when $var = \text{true}$ and the low branch leading to F_l should be taken when $var = \text{false}$.

Let $Val(F, \rho)$ be the value of ADD F under variable value assignment ρ . Then the valua-

tion of an ADD can be defined recursively by the following equation:

$$Val(F, \rho) = \begin{cases} F = C : & C \\ F \neq C \wedge \rho(F^{var}) = true : & Val(F_h, \rho) \\ F \neq C \wedge \rho(F^{var}) = false : & Val(F_l, \rho) \end{cases}$$

Formally, we define a *variable ordering* as a total ordering over all variables such that for all variable pairs x_i, x_j ($i \neq j$) either $x_i \succ x_j$ or $x_j \succ x_i$. We say that F satisfies a given variable ordering if $F = C$ or F is of the form *if* (F^{var}) *then* F_h *else* F_l where (1) F^{var} does not occur in F_h or F_l , (2) F^{var} is the earliest variable under the given ordering occurring in F and (3) F_l and F_h satisfy the variable ordering. We discuss choices for variable order later in the context of variable reordering.

Then we obtain the following lemma where we define a *reduced* ADD to be the minimally-sized ordered decision diagram representation a function $f(x_1, \dots, x_n)$.

Lemma 3.1.1. *Fix a variable ordering over x_1, \dots, x_n . For any function $f(x_1, \dots, x_n)$ mapping $\mathbb{B}^n \rightarrow \mathbb{R}$, there exists a unique reduced ADD F over variable domain x_1, \dots, x_n satisfying the given variable ordering such that for all $\rho \in \mathbb{B}^n$ we have $f(\rho) = Val(F, \rho)$.*

Bryant [1986] provides a proof of this lemma for BDDs, which only have two distinct terminal values. The proof trivially generalizes to ADDs, which can have more than two distinct terminal values. This lemma shows that there is a unique canonical ADD representation of all functions from $\mathbb{B}^n \rightarrow \mathbb{R}$.

Given that there exists a unique reduced ADD for any function from $\mathbb{B}^n \rightarrow \mathbb{R}$, we next describe how this reduced ADD can be constructed from an arbitrary ordered decision diagram. All algorithms that we will define rely on the helper function *GetNode* in Algorithm 1, which returns a canonical representation of a single internal decision node. Using *GetNode*, the *Reduce* procedure in Algorithm 2 takes any ordered decision diagram and returns its reduced, canonical ADD representation (necessarily removing any redundant structure in the process). The control flow of *Reduce* is very simple in that it uses the *GetNode* procedure to recursively build a reduced ADD from the bottom up (i.e., from the terminal leaf nodes all the way up to the root node). An example application of the *Reduce* algorithm is given in Figure 3.9.

Binary Operations on ADDs

Given functions $\mathbb{B}^n \rightarrow \mathbb{R}$ represented as ADDs, we now want to apply operations to these functions that work directly on the ADD representation. Additionally, we would prefer that

Algorithm 1: $GetNode(v, F_h, F_l) \longrightarrow F_r$

```

input   :  $v, F_h, F_l$  : Var and node ids for high/low branches
output  :  $F_r$  : Return values for offset,
           multiplier, and canonical node id
begin
  // If branches redundant, return child
  if ( $F_l = F_h$ ) then
     $\sqsubset$  return  $F_l$ ;

  // Make new node if not in cache
  if ( $\langle v, F_h, F_l \rangle \rightarrow id$  is not in node cache) then
     $\sqsubset$   $id :=$  currently unallocated id;
     $\sqsubset$  insert  $\langle v, F_h, F_l \rangle \rightarrow id$  in cache;

  // Return the cached, canonical node
  return  $id$ ;
end

```

Algorithm 2: $Reduce(F) \longrightarrow F_r$

```

input   :  $F$  : Node id
output  :  $F_r$  : Canonical node id for reduced ADD
begin
  // Check for terminal node
  if ( $F$  is terminal node) then
     $\sqsubset$  return canonical terminal node for value of  $F$ ;

  // Check reduce cache
  if ( $F \rightarrow F_r$  is not in reduce cache) then
    // Not in cache, so recurse
     $F_h := Reduce(F_h)$ ;
     $F_l := Reduce(F_l)$ ;

    // Retrieve canonical form
     $F_r := GetNode(F^{var}, F_h, F_l)$ ;

    // Put in cache
     $\sqsubset$  insert  $F \rightarrow F_r$  in reduce cache;

  // Return canonical reduced node
  return  $F_r$ ;
end

```

these operations avoid enumerating all possible variable assignments whenever possible.

To do this, we first define the *Apply* function that applies a binary operation to two operands represented as ADDs and returns the result as an ADD. We let op denote a binary operator on ADDs with possible operations being addition, subtraction, multiplication, division, min, and max denoted respectively as \oplus , \ominus , \otimes , \oslash , $\min(\cdot, \cdot)$, and $\max(\cdot, \cdot)$. We also define binary

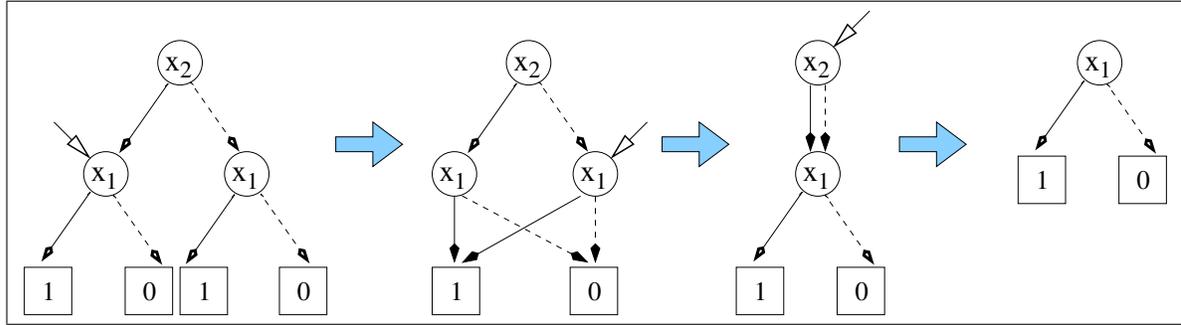


Figure 3.2: An example application of the *Reduce* algorithm. The input is the leftmost diagram. From left to right, the hollow arrow shows the node F currently being evaluated by *Reduce* just after the recursive *Reduce* calls to the high branch F_h and low branch F_l but before $GetNode(F^{var}, F_h, F_l)$ is called and the canonical representation of F is returned (see Algorithm 2). The next diagram in the sequence shows the result after the previous *Reduce* call. The rightmost diagram is the final canonical ADD representation of the input.

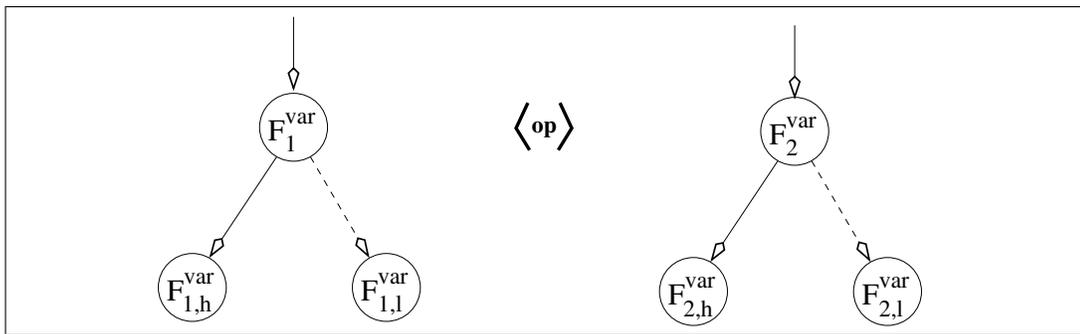


Figure 3.3: Two ADD nodes F_1 and F_2 and a binary operation op with the corresponding notation used in the presentation of the *Apply* function.

comparison functions $\geq, >, \leq, <$ that return an indicator function represented as an ADD that takes the value 1 when the comparison is satisfied and 0 otherwise.

The high-level control flow of the *Apply* routine in Algorithm 3 is straightforward: we first check whether we can compute the result immediately by calling *ComputeResult*, otherwise we check if we can reuse the result of a previously cached *Apply* computation. If we can do neither of these, we then choose a variable to branch on and recursively call the *Apply* routine for each instantiation of the variable. We cover these steps in-depth in the following sections and note that Figure 3.4 provides an example of the *Apply* operation.

Terminal computation The function *ComputeResult* given in Table 3.1, determines if the result of a computation can be immediately computed without recursion. The first entry in

Algorithm 3: $Apply(F_1, F_2, op) \longrightarrow F_r$

```

input    :  $F_1, F_2, op$  : ADD nodes and op
output   :  $F_r$  : ADD result node to return
begin
  // Check if result can be immediately computed
  if ( $ComputeResult(F_1, F_2, op) \rightarrow F_r$  is not null ) then
     $\sqsubset$  return  $F_r$ ;
  // Check if result already in apply cache
  if ( $\langle F_1, F_2, op \rangle \rightarrow F_r$  is not in apply cache) then
    // Not terminal, so recurse
    if ( $F_1$  is a non-terminal node) then
      if ( $F_2$  is a non-terminal node) then
        if ( $F_1^{var}$  comes before  $F_2^{var}$ ) then
           $\sqsubset$   $var := F_1^{var}$ ;
        else
           $\sqsubset$   $var := F_2^{var}$ ;
        else
           $\sqsubset$   $var := F_1^{var}$ ;
      else
         $\sqsubset$   $var := F_2^{var}$ ;
    // Set up nodes for recursion
    if ( $F_1$  is non-terminal  $\wedge$   $var = F_1^{var}$ ) then
       $\sqsubset$   $F_l^{v1} := F_{1,l}; F_h^{v1} := F_{1,h}$ ;
    else
       $\sqsubset$   $F_{l/h}^{v1} := F_1$ ;
    if ( $F_2$  is non-terminal  $\wedge$   $var = F_2^{var}$ ) then
       $\sqsubset$   $F_l^{v2} := F_{2,l}; F_h^{v2} := F_{2,h}$ ;
    else
       $\sqsubset$   $F_{l/h}^{v2} := F_2$ ;
    // Recurse and get cached result
     $F_l := Apply(F_l^{v1}, F_l^{v2}, op)$ ;
     $F_h := Apply(F_h^{v1}, F_h^{v2}, op)$ ;
     $F_r := GetNode(var, F_h, F_l)$ ;
    // Put result in apply cache and return
     $\sqsubset$  insert  $\langle F_1, F_2, op \rangle \rightarrow F_r$  into apply cache;
  return  $F_r$ ;
end

```

this table is required for proper termination of the algorithm as it computes the result of an operation applied to two terminal constant nodes. However, the other entries denote a number of pruning optimizations that immediately return a node without recursion. For example, we know that $F_1 \oplus 0 = F_1$ and $F_1 \otimes 1 = F_1$. If a result cannot be immediately determined in *ComputeResult* then we must continue recursing on the substructure of the operands until a

$ComputeResult(F_1, F_2, op) \longrightarrow F_r$	
Operation and Conditions	Return Value
$F_1 \text{ op } F_2; F_1 = C_1; F_2 = C_2$	$C_1 \text{ op } C_2$
$F_1 \oplus F_2; F_2 = 0$	F_1
$F_1 \oplus F_2; F_1 = 0$	F_2
$F_1 \ominus F_2; F_2 = 0$	F_1
$F_1 \otimes F_2; F_2 = 1$	F_1
$F_1 \otimes F_2; F_1 = 1$	F_2
$F_1 \oslash F_2; F_2 = 1$	F_1
$\min(F_1, F_2); \max(F_1) \leq \min(F_2)$	F_1
$\min(F_1, F_2); \max(F_2) \leq \min(F_1)$	F_2
similarly for max	
$F_1 \leq F_2; \max(F_1) \leq \min(F_2)$	1
$F_1 \leq F_2; \max(F_2) \leq \min(F_1)$	0
similarly for $<, \geq, >$	
other	<i>null</i>

Table 3.1: Input and output summaries of *ComputeResult*. If *ComputeResult* receives two constant ADD nodes as input, the constant resulting from the direct evaluation of *any* possible binary operation is returned. In other cases where at least one node is non-terminal, special operand structure and specific operator properties sometimes permit the computation of the result without further recursion. Some computations rely on the unary $\min(F)$ and $\max(F)$ operators that are discussed directly following the *Apply* algorithm.

result can be computed.

Recursive computation If a call to *Apply* is unable to immediately compute a result or reuse a previously cached computation, we must recursively compute the result. For this we have two cases (the third case where both operands are constant terminal nodes having been taken care of in the previous section). These algorithms assume the notation given in Figure 3.3 for the structure of the operands.

- F_1 or F_2 is a constant terminal node, or $F_1^{var} \neq F_2^{var}$: For simplicity of exposition, we assume the operation is commutative and reorder the operands so that F_1 is the constant node or the operand whose variable comes *later* in the variable ordering so that we know to branch on F_2^{var} first.⁵ Thus, we compute the operation applied separately to

⁵We note that the first case prohibits the use of the non-commutative \ominus and \oslash operations. However, a simple solution would be to recursively descend on either F_1 or F_2 rather than assuming commutativity and swapping operands to ensure descent on F_2 . To accommodate general non-commutative operations, we have used this alternate approach in our specification of the *Apply* routine.

F_1 and *each* of F_2 's high and low branches. We then build an internal *if* decision node conditional on F_2^{var} and get its canonical representation for the result:

$$F_h = Apply(F_1, F_{2,h}, op)$$

$$F_l = Apply(F_1, F_{2,l}, op)$$

$$F_r = GetNode(F_2^{var}, F_h, F_l)$$

- F_1 and F_2 are constant nodes and $F_1^{var} = F_2^{var}$: Since the variables for each operand match, we know the result F_r is simply an *if* statement branching on F_1^{var} ($= F_2^{var}$) with the true case being the operator applied to the high branches of F_1 and F_2 and likewise for the false case and the low branches:

$$F_h = Apply(F_{1,h}, F_{2,h}, op)$$

$$F_l = Apply(F_{1,l}, F_{2,l}, op)$$

$$F_r = GetNode(F_1^{var}, F_h, F_l)$$

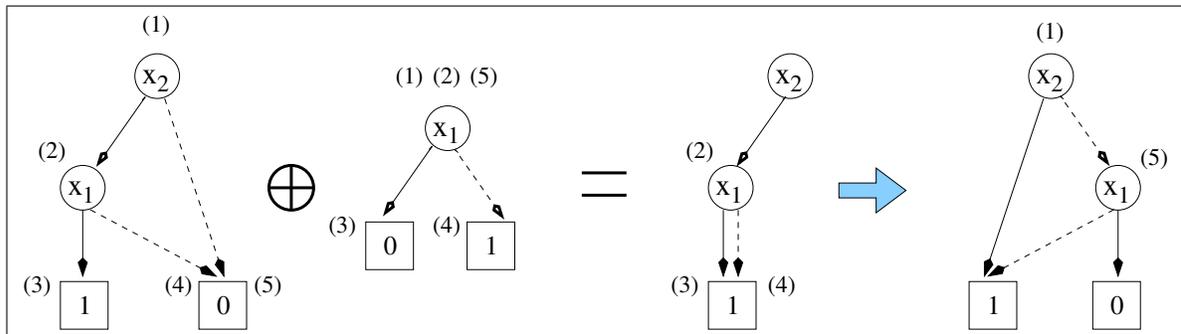


Figure 3.4: An example application of the *Apply* algorithm. The indices (i) in the diagram correspond to successive (recursive) calls to the *Apply* algorithm: for the operands the indices denote which node of each operand is passed as a parameter to the call to *Apply* (the op is always \oplus); for the result the indices indicate the node that is returned by the call to *Apply*. For example, the initial call to *Apply* takes the arguments corresponding to the node marked (1) x_2 on the LHS of the \oplus and the node (1) x_1 on the RHS of the \oplus (as well as the operation \oplus itself) and returns the node marked (1) on the RHS of the equality.

Other Operations Above we covered binary operations on ADDs, but we will also need to perform a variety of unary operations on ADDs such as determining the min and max value of

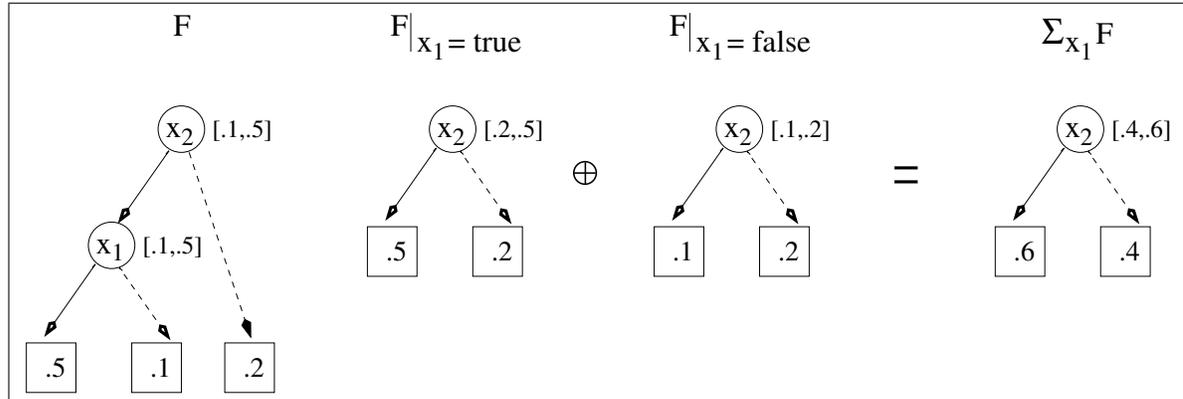


Figure 3.5: An example application of the unary *restriction* and *marginalization* operations. Each ADD has all of its internal nodes annotated with $[\min, \max]$, which can be recursively computed from the children of each internal node.

an ADD and marginalization over variables. Here we cover some unary operations that can be performed (efficiently) on ADDs:

- min and max computation:** During the *Reduce* operation, it is easy to maintain the minimum and maximum values for each internal decision node. Exploiting the fact that an ADD is a DAG, $\min F = \min(F_l, F_h)$ and likewise for \max . A simple example of this annotation and its recursive relationship is shown in Figure 3.5.
- Restriction:** The restriction of a variable x_i in an ADD F to either *true* or *false* (i.e. $F|_{x_i=true/false}$) can be computed by replacing all decision nodes for variable x_i with the branch corresponding to the variable restriction. Then *Reduce* can be applied on the resulting decision diagram to convert it to a canonical ADD. Two examples of restriction are given in Figure 3.5.
- Sum out/marginalization:** A variable x_i can be summed (or marginalized) out of a function F simply by computing the sum of the restricted functions (i.e. $\sum_{x_i} F = F|_{x_i=T} \oplus F|_{x_i=F}$). An example of this is given in Figure 3.5.
- Negation/reciprocation:** Negation can be performed using the binary *Apply* operation on $0 \ominus F$. Likewise, reciprocation (i.e., $\frac{1}{F}$) can be computed using the binary *Apply* operation $1 \oslash F$.
- Variable reordering:** Rudell [1993] provides an ADD variable reordering algorithm that casts a general variable reordering in terms of a sequence of pairwise reorderings

of neighboring variables. Then, the basic idea is that two variables x_i and x_j can be reordered locally (i.e., rotated) in the ADD DAG without requiring the modification of any internal nodes other than those involving x_i and x_j . Furthermore, Rudell describes how this can be done without requiring extra storage for backpointers from children to parents if *GetNode*'s canonical node cache is allowed to be modified.

As an addendum to this final operation, we note that the MDP solution algorithms based on ADDs (and their extensions) that we introduce in this chapter could dynamically reorder variables in an attempt to maintain even more compact representations than possible with a fixed variable ordering. However, we do not employ such dynamic variable ordering techniques in this thesis as they prevent the reuse of cached computations that underly one of the major sources of efficiency of ADDs when used in MDP solution algorithms. Furthermore, searching for compact ADD representations requires search and is computationally expensive. Such results are reflected in experiments using ADDs to perform value iteration in factored MDPs [St-Aubin *et al.*, 2000], which demonstrate that dynamic variable reordering does not pay off and that a natural fixed variable ordering derived from the MDP description tends to be compact and preserves structure. As a consequence of these observations, all of the algorithms used in this thesis use a natural fixed variable ordering derived from the order that variables appear in an MDP problem description, unless otherwise noted.

3.1.3 Additive Independence

Additive independence in reward structure is a common assumption in utility theory and related fields [Keeney and Raiffa, 1976; Bacchus and Grove, 1995]. In Figure 3.1(c), we note that we could represent the additive reward structure of SYSADMIN using an ADD whose size scales quadratically in the number of computers n . But if we can explicitly model additive rewards as sums of (potentially non-linear) factors, then we trivially note that the SYSADMIN reward can be expressed compactly in a form whose size scales linearly in n :

$$R(\vec{x}, a) = \sum_{i=1}^n \mathbb{I}[x_i] \quad (3.3)$$

Furthermore, if we permit the use of similar expressions in the CPTs that we specify for our transition DBN, we can also exploit additive independence in their representation. For example, letting $Conn(i, j)$ denote that there is an incoming network connection to computer j from computer i , we note that the CPTs for the transition function for any SYSADMIN network

topology can be specified in the following additive manner:

$$P(x'_i = \text{true} | \vec{x}_i, a) = \begin{cases} a = \text{reboot}(c_i) : & 1 \\ a \neq \text{reboot}(c_i) : & (0.05 + 0.9 \cdot \mathbb{I}[x_i]) \cdot \frac{\sum_j \mathbb{I}[j \neq i \wedge x_j \wedge \text{Conn}(j, i)]}{|\{x_j | j \neq i \wedge \text{Conn}(j, i)\}| + 1} \end{cases}$$

Here we see that the success probability of a computer running scales proportionally to the number of its incoming connections that are also running. And we also note that the previous CPT we gave for the unidirectional ring in Figure 3.1(b) is just a special case of this CPT where computer i is connected only to computer $i + 1$ (where addition is modulo n).

In our subsequent discussion of solution methods for factored MDPs, we note that some recent approaches can exploit additive reward structure while others cannot. In fact, it will only be in the final part of this chapter when we introduce affine ADDs (AADDs) that we will be able to fully exploit CSI and additive independence in both the reward and transition functions, not to mention multiplicative independence as it happens to naturally occur in many value functions.⁶

3.1.4 Structured Policy Representation

Just as the reward and transition function may be represented in a factored manner in propositional MDPs, so can the policy. To do this, we adapt the following definition from Boutilier *et al.* [1995b]:

Definition 3.1.2. *A structured policy is any set of function-action pairs $\pi = \{\langle \phi_a, a \rangle\}$ such that ϕ_a is a structured representation of an indicator function and $\{\phi_a\}$ partitions the state space. This induces the explicit policy $\pi_a(\vec{x}) = a$ iff $\phi_a(\vec{x}) = 1$.*

To ensure that the policy partitions the state space, one must ensure that it is exhaustive and that all action indicator functions are pairwise disjoint. To ensure that the policy exhausts the entire state space, one can simply ensure that the sum of all indicator functions is the constant 1 (i.e., $\sum_{a \in \mathcal{A}} \phi_a = 1$). To ensure that all action policies are pairwise disjoint, one can ensure $\phi_a \cdot \phi_b = 0$ for all action pairs $a \in \mathcal{A}$, $b \in \mathcal{A}$ such that $a \neq b$.

There are a variety of structured methods for representing the $\{\phi_a\}$ indicator functions ranging from decision lists [Koller and Parr, 1999a], to trees [Boutilier *et al.*, 1995b], to ADDs. Throughout our presentation here, we will use ADDs. Then policy evaluation is simply the task

⁶Multiplicative independence is just the multiplicative generalization of additive independence.

of evaluating each ADD ϕ_a under a given state assignment \vec{x} to see if $\phi_a(\vec{x}) = 1$ (meaning do action a). This structured policy representation will play an important role in our description of structured policy iteration.

3.1.5 Putting it all Together

Before we cover exact solution methods in the factored MDP framework, let us quickly recapitulate the factored MDP representation. In a factored MDP, states will be represented by vectors \vec{x} of length n , where for simplicity we assume all state variables x_1, \dots, x_n are binary-valued;⁷ hence the total number of states is $N = 2^n$. We also assume a finite set of actions $A = \{a_1, \dots, a_m\}$. As usual, we assume a discount factor γ , $0 \leq \gamma \leq 1$ where appropriate steps have been taken to ensure bounded reward in the case of $\gamma = 1$.

To generalize the MDP model from the previous chapter, we specify a propositionally factored MDP by the following:

1. *Factored Transition Function:* A DBN-factored state transition model which specifies the probability of the next state \vec{x}' given the current state \vec{x} and action a . The transition function can be factored as a dynamic Bayes net (DBN) with CPTs $P(x'_i | \vec{x}_i, a)$ where each next state variable x'_i is only dependent upon the action a and its direct parents \vec{x}_i in the DBN. Then the transition model can be compactly specified as $P(\vec{x}' | \vec{x}, a) = \prod_{i=1}^n P(x'_i | \vec{x}_i, a)$.
2. *Factored Reward Function:* An additive reward function $\sum_{i=1}^r R_i(\vec{x}_i, a)$ over r reward factors $R_i(\vec{x}_i, a)$ dependent on action a and relevant state \vec{x}_i , which specifies the immediate reward obtained by taking action a in state \vec{x} .

The individual factors can be expressed as tabular representations, or as trees and ADDs that exploit CSI, or even as additive expressions that exploit additive independence. Finally, when needed, a structured policy $\pi = \{\langle \phi_a, a \rangle\}$ uses indicator functions ϕ_a to specify the states where action a should be taken.

⁷However, all of the methods here can be easily generalized to non-binary variables through known transformations [Rossi *et al.*, 1990; Stergiou and Walsh, 1999].

3.2 Exact Solution Methods

In our specification of our solution methods, it will be notationally useful to define a *backup* operator B^a for action a as follows:⁸

$$B^a[V(\vec{x})] = \gamma \sum_{\vec{x}'} \prod_{i=1}^n P(x'_i | \vec{x}_i, a) V(\vec{x}') \quad (3.4)$$

This is essentially the factored representation of the Q-function computation for action a in Equation 2.8 from Chapter 2 without adding in the reward. We note that the backup $B^a[\cdot]$ operator can exploit both additive structure since it is a *linear operator* as well as efficient factored computation due to the transition DBN structure.

If π^* denotes the optimal policy and V^* its value function, then we have the following factored representation of the fixed-point Equation 2.4 from Chapter 2:

$$V^*(\vec{x}) = \max_{a \in \mathcal{A}} \left\{ \sum_{i=1}^r R_i(\vec{x}_i, a) + B^a[V^*(\vec{x})] \right\}. \quad (3.5)$$

Having done this, we first present the basic factored variants of the relevant MDP equations from the previous thesis chapter and proceed to show that the previous MDP solution methods can be easily redefined in terms of these factored equations. This allows us to exploit the factored structure and any CSI therein during the application of the MDP solution algorithms.

3.2.1 Structured Value Iteration

ADD-based Value Iteration

The value iteration algorithm from Chapter 2 can be easily extended to exploit factored MDP structure in a structured value iteration setting. Initializing $V^0(\vec{x})$ to some value, we generalize Equation 2.7 from Chapter 2 to the factored form:

$$V^{t+1}(\vec{x}) = \max_{a \in \mathcal{A}} \left\{ \sum_{i=1}^r R_i(\vec{x}_i, a) + B^a[V^t(\vec{x})] \right\}. \quad (3.6)$$

It will be extremely important to use a compact data structure such as a tree or ADD to exploit CSI in the representation of the value function in structured value iteration. If we were

⁸Technically, this should be written $(B^a V)(\vec{x})$, but we abuse notation for readability when V itself is structured and for consistency with subsequent first-order MDP notation.

to simply use a tabular representation, we would find that in typical MDPs, all variables in the value function become correlated after some number of backups *if* the graphical model underlying the DBN cannot be decomposed into disjoint components [Boutilier *et al.*, 1995b]. Thus, a tabular representation will typically need to represent a value function over all state variables and in the absence of some method for compactly representing value function structure, this representation will require full state enumeration.

Fortunately, as described previously, ADDs are ideal for exploiting CSI and functions with shared substructure, both of which may occur in the value functions of highly structured factored MDPs. As such, representing all factors in Equation 3.6 using ADDs and carrying out its computation in terms of ADD operations as done in the SPUDD algorithm of Hoey *et al.* [1999] has proved to be a promising method in comparison to the enumerated state value iteration approach of the previous chapter. While SPUDD may scale comparably to enumerated state value iteration in the worst case (e.g., when all states have distinct values), the authors demonstrate that there is much potential for computational and space savings using the SPUDD algorithm to perform value iteration on many factored MDPs.

Decomposition-based Value Iteration

In a different vein of research, there are alternate (but not incompatible) approaches to structured value iteration that exploit decomposable task structure in MDPs [Meuleau *et al.*, 1998a; Singh and Cohn, 1998]. If a problem domain consists of many independent subprocesses that only interact via their dependence on globally shared resources and/or constraints on joint action choices, one can often factor these MDPs into tasks represented as independent subMDPs with global resource and action constraints. We could take the cross-product of all the subMDP state spaces and solve the resulting joint MDP, but this would discard a lot of the structure inherent in the task decomposition of the initial problem. Alternately we can focus on algorithms that directly exploit the decomposed structure of the MDP directly.

An exact structured value iteration approach for a subclass of MDPs with highly decomposable structure is provided by Singh and Cohn [Singh and Cohn, 1998]. In this model, an MDP must decompose into a set of subMDPs where each subMDP has its own independent state space but an action set that is globally constrained. The reward objective is to maximize the sum of rewards for each subMDP. The solution approach they advocate is a value iteration method based on maintaining upper and lower bounds on the value function. The upper bounds simply come from assuming that actions are unconstrained across subMDPs (which

can be achieved in the best case) and the lower bounds come from taking the maximal reward for an individual subMDP (which could be achieved in the worst case). These upper and lower bounds allow various actions to be pruned from consideration during value iteration and with enough iterations will provably converge on an optimal solution. This decomposed value iteration algorithm is empirically found to be more efficient than value iteration in the joint cross-product MDP.

3.2.2 Structured Policy Iteration

Structured policy iteration (SPI) in factored MDPs was first defined in Boutilier *et al.* [1995b] using trees as a method of state aggregation. Here we describe a similar version using ADDs.⁹ Recalling the definition of modified policy iteration from the previous chapter, first we initialize a random policy $\pi_0 = \{\langle \phi_a, a \rangle\}$ and then we iterate between approximate policy evaluation and policy improvement steps. For approximate policy evaluation, we can simply use the following factored extension of the successive approximation update (c.f. Section 2.4.2):

$$V_{\pi_i}^{t+1}(\vec{x}) = \sum_{a \in \mathcal{A}} \phi_a(\vec{x}) \cdot \left\{ \sum_{j=1}^r R_j(\vec{x}_j, a) + B^a[V_{\pi_i}^t(\vec{x})] \right\}. \quad (3.7)$$

Here, the policy indicator function ϕ_a ensures that the value for a state is only updated for action a if the policy indicates that action a should be taken from that state. We note that this entire computation can be carried out in terms of efficient operations on ADDs. Correctness follows from the fact that π_i is a partitioning of the state space.

Then, given $V^{\pi_i}(\vec{x})$, we need only produce a new policy π_{i+1} that is greedy w.r.t. V^{π_i} . In order to break ties for actions having equal value, we require a total preference ordering (perhaps random) over actions, that is, for all actions a and b such that $a \neq b$, either $a \succ b$ or $b \succ a$. Recalling the definition of the ADD “ $>$ ” and “ \geq ” comparison functions that produce an ADD taking the value 1 in states where the LHS operand is greater than (or equal to) the RHS operand and 0 otherwise, we can produce the ADD representation of ϕ_a for all $a \in \mathcal{A}$ in the following iterative fashion:

1. Initialize $\phi_a = 1$ (the constant 1 ADD)
2. For each $a \in \mathcal{A}$, let $Q(\vec{x}, a) = \sum_{j=1}^r R_j(\vec{x}_j, a) + B^a[V_{\pi_i}(\vec{x})]$

⁹We will implicitly assume throughout the text that all operations in the following equations such as $+$, $-$, \times , *etc.* are performed on ADDs in terms of their corresponding operations \oplus , \ominus , \otimes , *etc.*

3. For each $b \in \mathcal{A}$ s.t. $b \neq a$ update ϕ_a as follows:

$$\phi_a := \begin{cases} a \succ b : \phi_a \cdot (Q(\vec{x}, a) \geq Q(\vec{x}, b)) \\ b \succ a : \phi_a \cdot (Q(\vec{x}, a) > Q(\vec{x}, b)) \end{cases}$$

Thus we have defined a structured version of policy iteration. While our algorithm presentation here differs from the original presentation [Boutilier *et al.*, 1995b], it is consistent with the overall structured approach to policy iteration. Furthermore, we will build on this approach when we extend policy iteration to exploit other types of structure in future chapters.

3.2.3 Difficulty of Structured Linear Programming

We do not present a structured variant of the exact linear programming solution to factored MDPs as this method requires *a priori* knowledge of the structure of the value function and in this case we are talking about the exact value function. Typically, we cannot determine the structure of an optimal value function from the structure of a factored MDP. Consequently, for the exact case, we would have no choice but to use a fully enumerated state representation of the value function, thus preventing the exploitation of factored structure. However, as we will see shortly, the approximate variant of linear programming is in fact quite useful for solving factored MDPs and there is much opportunity to exploit factored structure in that case.

3.3 Approximate Solution Methods

While some factored MDPs do exhibit considerable structure in their optimal value functions or policies, sometimes these representations are still too large for practical representation or computation as the size of the problem scales. Thus, in this section we focus on approximate variants of previously described solution algorithms.

3.3.1 Approximate Value Iteration Methods

ADD-based Approximation

One additional benefit of the use of ADDs to specify factored MDPs is that it allows one to prune internal nodes in an ADD and replace these nodes with the minimum and maximum value of the ADD rooted at that node [Dearden and Boutilier, 1997]. An example of this is

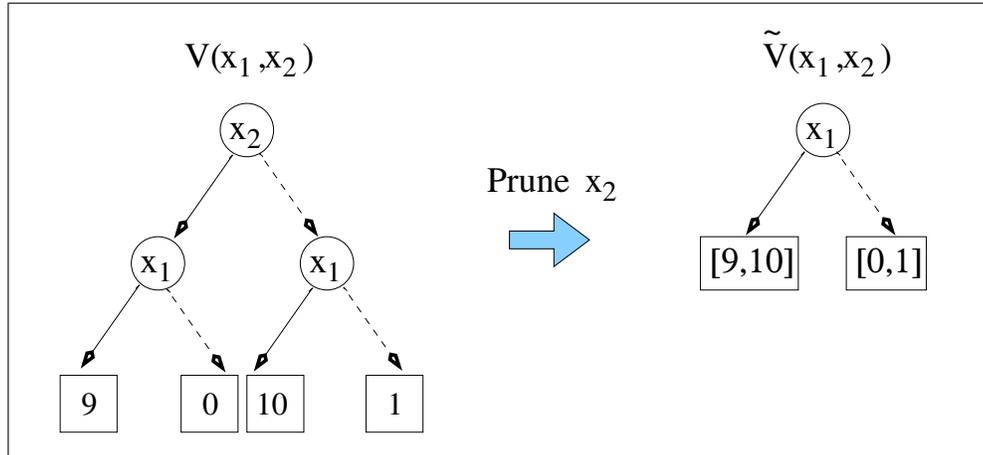


Figure 3.6: An example of approximating an ADD representation of a value function $V(x_1, x_2)$ as $\tilde{V}(x_1, x_2)$ by pruning out the decision node for variable x_2 and replacing leaf values with their respective ranges.

shown in Figure 3.6. One can then perform value iteration maintaining these upper and lower bounds. Since the Bellman backup is a known contraction operator [Puterman, 1994], this algorithm will still converge, albeit within some error bound of the optimal value function. This is the idea behind the APRICODD [St-Aubin *et al.*, 2000] algorithm that is essentially the SPUDD value iteration algorithm with an extra step for approximation by pruning the value function ADD. We note that APRICODD represents a completely automated approach to approximate value iteration that autonomously derives the approximated value function. It should be noted that this contrasts sharply with the linear-value approximation approach to approximate value iteration discussed in Section 2.5.3 that relies on the pre-specification of a fixed set of basis functions.

Decomposition-based Approximation

In the vein of exploiting structure in decomposable MDPs, Mealeau *et al.* [1998a] describe an approximate value iteration technique for solving weakly coupled subMDPs having global resource and action constraints. Their algorithm is referred to as Markov Task Decomposition (MTD) and is an approximately optimal approach to solving the joint MDP that divides the solution into local and global optimization steps. MTD first determines the optimal value function for each subMDP. Following this local optimization, a global optimization phase then chooses a joint action at each time step that enforces the global resource constraint while trading off local action choices for each task in order to maximize the expected reward. Since an

optimal sequential solution in this case would be equivalent to solving the full joint MDP, a heuristic resource allocator is used in this work.

While we don't exploit the same approximate decomposition ideas in the contributions of this thesis, we do note that the general framework of additive value decomposition and approximate solution methods within this framework serves as motivation for our work in future chapters.

3.3.2 Linear-value Approximation Solution Methods

We next introduce three efficient approximate solution methods for factored MDPs based on linear-value approximation [Guestrin *et al.*, 2002; Schuurmans and Patrascu, 2001]. These methods are effectively factored extensions of the approximate policy iteration and approximate linear programming techniques from the previous chapter. The key to the efficiency of these approaches over their enumerated state counterparts will be to show how the structure of the factored representation can be exploited by algorithms such as *variable elimination* [Zhang and Poole, 1996] that scale exponentially in the induced tree-width of the representation rather than exponentially in the total number of state variables. This exploitation of structure will be most apparent when solving the linear programs for error-minimizing max-norm projections that are at the heart of these techniques.

In a linear-value function representation, we represent V as a linear combination of k basis functions $b_j(\vec{x})$ where the b_j are typically dependent upon small subsets of \vec{x} :

$$V(\vec{x}) = \sum_{j=1}^k w_j b_j(\vec{x}) \quad (3.8)$$

Our goal is to find weights that approximate the optimal value function as closely as possible. In doing this, all of our solution methods will need to compute the backup of the value function through an action a . To compute this, we recall that the backup operator $B^a[\cdot]$ previously defined is a linear operator such that it distributes into a sum:

$$B^a[V(\vec{x})] = B^a\left[\sum_{j=1}^k w_j b_j(\vec{x})\right] \quad (3.9)$$

$$= \sum_{j=1}^k w_j B^a[b_j(\vec{x})] \quad (3.10)$$

Thus, if the basis functions are defined over small sets of variables, and the backup introduces an additional small set of variables that causally affect this basis function according to the DBN representation of the transition distribution, this sum will be over factors of small sets of variables. This factored structure will be exploited in the methods we define in this section.

Next, we explore factored extensions of approximate policy iteration and linear programming. However, we do not cover approximate value iteration approaches due to their possibility of divergence as noted in the last chapter.

Approximate Policy Iteration

We can generalize policy iteration (API) to the factored linear-value approximation case by calculating successive iterations of weights $w_j^{(i)}$ that represent the best approximation of the fixed point value function for policy $\pi^{(i)}$ at iteration i . The method we present here is a slight variant of that given in Guestrin *et al.* [2002]¹⁰ and is an approach that we will generalize in the next chapter. To apply this approach, we need to introduce the $B^\pi[\cdot]$ operator which is just the backup operator under a fixed policy:

$$B^\pi[V(\vec{x})] = \gamma \sum_{\vec{x}'} \prod_{i=1}^n P(x'_i | \vec{x}_i, \pi(\vec{x})) V(\vec{x}') \quad (3.11)$$

In the context of the following algorithm, we will discuss how $B^\pi[\cdot]$ can be efficiently computed in structured cases.

We perform API by carrying out the following two steps on each iteration i : (1) derive the greedy policy: $\pi^{(i+1)} \leftarrow \pi_{gre}(\sum_{j=1}^k w_j^{(i)} b_j(s))$ using the approach outlined in Section 3.2.2 and (2) use the following LP to determine the weights for the L_∞ minimizing projection of the

¹⁰Other than the ordering of action comparisons in the greedy policy derivation method of Section 3.2.2, this presentation of API follows that of Guestrin *et al.* [2002]. For greedy policy derivation, they use a special ordering of action comparisons that first compares all actions to a *noop* action and then to each other, arguing that this approach is advantageous for domains such as SYSADMIN.

approximate value function for policy $\pi^{(i+1)}$:

$$\begin{aligned}
& \text{Variables: } w_1^{(i+1)}, \dots, w_k^{(i+1)} \\
& \text{Minimize: } \beta^{(i+1)} \\
& \text{Subject to: } \beta^{(i+1)} \geq \left| \sum_{i=1}^r R_i(\vec{x}_i, \pi(\vec{x})) + \sum_{j=1}^k (w_j^{(i+1)}) B^{\pi^{(i+1)}} [b_j(\vec{x})] \right. \\
& \quad \left. - \sum_{j=1}^k [w_j^{(i+1)} b_j(\vec{x})] \right| ; \forall \vec{x},
\end{aligned} \tag{3.12}$$

We note that this LP is just the factored form of the LP defined for approximate policy iteration in Equation 2.23 from the previous chapter where we have exploited linearity of the backup operator. Consequently, when the policy converges (i.e., $\pi^{(i+1)} = \pi^{(i)}$ or equivalently $\vec{w}^{(i+1)} = \vec{w}^{(i)}$), we can derive an error bound on the approximated value function by plugging the projection error β of the final LP solution directly into Equation 2.19 since β is the Bellman error of the approximated value function in this case.

However, we note that in the factored framework, $B^{\pi^{i+1}}[\cdot]$ cannot easily be computed according to Equation 3.11 since our structured policy $\pi(\vec{x})$ takes the form of indicator functions. However, we need only enforce that an LP constraint for an action a is satisfied in the states where ϕ_a^{i+1} takes the value 1. To do this, we can ensure that the constraint for action a is trivially satisfied when ϕ_a is 0. So we introduce the following policy factor as a summand in our constraint:

$$\hat{\phi}_a^{i+1} = (\phi_a^{i+1} - 1) \cdot \infty \tag{3.13}$$

Clearly, $\hat{\phi}_a^{i+1}$ will take the value 0 in states where a should be taken according to π^{i+1} and the value $-\infty$ otherwise.

To see how this allows us to perform the backup under a policy, let us rewrite the constraints we have expressed above:

$$\beta^{(i+1)} \geq \hat{\phi}_a^{i+1} + \left| \sum_{i=1}^r R_i(\vec{x}_i, a) + \sum_{j=1}^k (w_j^{(i+1)}) B^a [b_j(\vec{x})] - \sum_{j=1}^k [w_j^{(i+1)} b_j(\vec{x})] \right| ; \forall \vec{x}, a \in A \tag{3.14}$$

Effectively, the constraint for action a will be trivially satisfied when the policy factor $\hat{\phi}_a^{i+1}$ should not be applied and takes the value $-\infty$. Otherwise, $\hat{\phi}_a^{i+1}$ takes the value 0 in states where the policy should be applied and then the remainder of the constraint must be satisfied.

In a subsequent section, we discuss efficient methods for solving the above form of LP with

a factored $\max\text{-}\sum$ form of the constraints.

Approximate Linear Programming

In the extension of approximate linear programming (ALP) to factored models, we simply replace the enumerated state representation from the previous chapter in Equation 2.24 with the factored representation introduced in this chapter following Schuurmans and Patrascu [2001] where we have again exploited linearity of the backup operator:

$$\begin{aligned}
 &\text{Variables: } w_1, \dots, w_k \\
 &\text{Minimize: } \sum_{\vec{x}} \sum_{j=1}^k w_j b_j(\vec{x}) \\
 &\text{Subject to: } 0 \geq \sum_{i=1}^r R_i(\vec{x}_i, a) + \sum_{j=1}^k (w_j B^a[b_j(\vec{x})]) - \sum_{j=1}^k w_j b_j(\vec{x}) ; \forall a, \vec{x}
 \end{aligned} \tag{3.15}$$

We can exploit the factored nature of the basis functions to simplify the objective to the following compact form where we assume each basis function explicitly depends on the subset of state variables in \vec{x}_j :

$$\sum_{\vec{x}} \sum_{j=1}^k w_j b_j(\vec{x}_j) = \sum_{j=1}^k w_j y_j \tag{3.16}$$

where $y^j = 2^{n-|\vec{x}_j|} \sum_{\vec{x}_j} b_j(\vec{x}_j)$.

Finally, we note that exploiting linearity of the backup operator again provides us with a factored $\max\text{-}\sum$ form of the ALP LP constraints from Equation 3.15 as it did similarly for the final form of the API LP constraints in Equation 3.14. We discuss an efficient solution to LPs with such $\max\text{-}\sum$ factored constraints in the next section.

Constraint Generation

In the above LP, both forms for the constraints take on the generic form of a sum of m factors $F_i(\vec{x})$ over (ideally) small sets of variables:

$$0 \geq w_1 \cdot F_1(\vec{x}) + \dots + w_n \cdot F_m(\vec{x}) ; \forall \vec{x} \tag{3.17}$$

Not every factor must have a weight w_i , but we note that each factor has at most one linear weight owing to the structure of the original basis functions and the properties of the backup

operators.

To view the constraints in a more concrete form, we note that for every possible instantiation \vec{x}^* of the state, we could simply instantiate the factor F_i to its constant value $c_i = F_i(\vec{x}^*)$ under that state assignment and come up with a corresponding linear constraint:

$$0 \geq w_1 \cdot c_1 + \dots + w_n \cdot c_m \quad (3.18)$$

We could generate constraints for *all* possible state assignments \vec{x}^* and solve our LP in this manner. However, we would obviously lose the benefits of our factored representation in that we would have to specify a number of constraints that scales exponentially in the number of state variables.

However, if we rewrite the constraints from Equation 3.17 in the following equivalent form where we enforce all constraints simultaneously with one maximization then we can see how to exploit the factored constraint structure:

$$0 \geq \max_{\vec{x}} [w_1 \cdot F_1(\vec{x}) + \dots + w_n \cdot F_m(\vec{x})] \quad (3.19)$$

This *cost network* [Dechter, 1999] form of these constraints lends itself to very efficient evaluation methods such as variable elimination. The question is how to exploit this property in our LP solution. Fortunately, as it turns out, there are at least two approaches to exploiting this structure.

The first solution, due to Guestrin *et al.* [2002] is to directly simulate variable elimination in the LP encoding of the constraints of Equation 3.19. This leads to a total number of constraints $O(\exp(TW))$ where TW is the induced tree-width of the cost-network under the variable elimination order that was used. This is an attractive method because the structure of the factored MDP and the basis functions should lead to $TW \ll n$ when (a) the basis functions range over small sets of variables with little or no overlap and (b) the backups of each basis function have similar characteristics due to the property that only small sets of variables affect each other causally in the Bayes net. Thus, simulating variable elimination in the LP variable encoding to produce $O(\exp(TW))$ constraints would be a much more efficient solution than generating $O(\exp(n))$ constraints as would be done in the enumerated state case.

However, a simpler approach and often an empirically faster method in practice¹¹ is the technique of constraint generation [Schuermans and Patrascu, 2001; Trick and Zin, 1997]. In

¹¹This is despite the lack of similar guarantees on the maximum number of constraints generated.

this case, we perform the following solution procedure where we have specified some solution tolerance ϵ :

1. Initialize LP with $\vec{w}^i = \vec{0}$, $i = 0$, and empty constraint set.
2. For each constraint in the cost-network form of Equation 3.19 instantiated with the current solution \vec{w}^i , find the maximally violated constraint C (if one exists) using variable elimination.
3. If C 's constraint violation is larger than ϵ , add C to LP constraint set, otherwise return \vec{w}^i as solution.
4. Solve LP for new solution \vec{w}^{i+1} , goto step 2

Using these constraint generation techniques, one can now efficiently apply either API or ALP with linear-value approximation to factored MDPs. However in comparing API and ALP, we note that in practice, one cannot always guarantee a compact structure for the policies generated during API. In addition, API requires one optimization of an LP on each iteration until convergence or some stopping criterion is reached. In contrast, ALP does not require a representation of the policy and tends to have a lower tree-width in its constraints. ALP also solves the problem with one LP optimization. Consequently, ALP tends to be much faster than API as noted by Schuurmans and Patrascu [2001], but they also note in their experiments that API produced better policies.

Basis Function Generation

One additional difficulty with linear value function approximation is that of generating a good set of basis functions. Certainly, a set of basis functions that poorly approximate the optimal value function can have an adverse impact on decision quality. Consequently, one can take a number of approaches to generating basis functions such as finding subtasks with additive reward [Poupart *et al.*, 2002a], performing branch-and-bound search to find Bellman-error minimizing basis functions [Poupart *et al.*, 2002b], or analyzing the dual of the LP solution to heuristically generate basis function candidates [Poupart *et al.*, 2002b]. Unfortunately, at this point in time, generating a good basis function set is still more of an art than a science, and there are no currently known methods that allow one to attain *a priori* guarantees on the decision quality for a given set of basis functions.