6

State Uncertainty

Mykel J. Kochenderfer

The previous two chapters discussed sequential decision-making problems in which the current state is known by the agent. Because of sensor limitations or noise, the state might not be perfectly observable. This chapter discusses sequential decision problems with state uncertainty and methods for computing optimal and approximately optimal solutions.

6.1 Formulation

A sequential decision problem with state uncertainty can be modeled as a *partially observable Markov decision process* (POMDP). A POMDP is an extension to the MDP formulation introduced in Chapter 4. In a POMDP, a model specifies the probability of making a particular observation given the current state.

6.1.1 Example Problem

Suppose we are assigned the task of taking care of a baby. We decide when to feed the baby on the basis of whether the baby is crying. Crying is a noisy indication that the baby is hungry. There is a 10% chance the baby cries when not hungry, and there is a 80% chance the baby cries when hungry.

The dynamics are as follows. If we feed the baby, then the baby stops being hungry at the next time step. If the baby is not hungry and we do not feed the baby, then 10% of the time the baby may become hungry at the next time step. Once hungry, the baby continues being hungry until fed.

The cost of feeding the baby is 5 and the cost of the baby being hungry is 10. These costs are additive, and so if we feed the baby when the baby is hungry, then there is a cost of 15. We want to find the optimal strategy assuming an infinite horizon with a discount factor of 0.9. Figure 6.1 shows the structure of the crying-baby problem as a dynamic decision network.

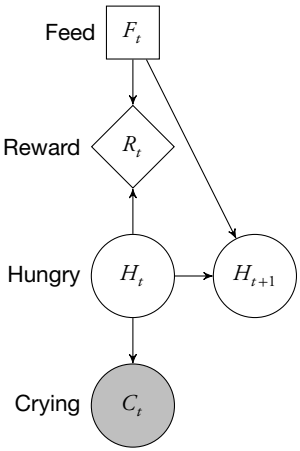

Figure 6.1 Crying-baby problem structure.

6.1.2 Partially Observable Markov Decision Processes

A POMDP is an MDP with an observation model. The probability of observing o given state s is written $O(o \mid s)$. In some formulations, the observation can also depend on the action a, and so we can write $O(o \mid s, a)$. The decisions in a POMDP at time t can only be based on the history of observations $o_{1:t}$. Instead of keeping track of arbitrarily long histories, it is common to keep track of the *belief state*. A belief state is a distribution over states. In belief state b, probability $b(s)$ is assigned to being in state s. A policy in a POMDP is a mapping from belief states to actions. The structure of a POMDP can be represented using the dynamic decision network in Figure 6.2.

6.1.3 Policy Execution

Algorithm 6.1 outlines how a POMDP policy is executed. We choose actions on the basis of the policy evaluated at the current belief state. Different ways to represent policies will be discussed in this chapter. When we receive a new observation and reward, we update our belief state. Belief-state updating is discussed in Section 6.2.

6.1.4 Belief-State Markov Decision Processes

A POMDP is really an MDP in which the states are belief states. We sometimes call the MDP over belief states a *belief-state MDP*. The state space of a belief-state MDP is simply the set of all possible beliefs, $\mathcal{B}$, in the POMDP. If there are n discrete states, then $\mathcal{B}$ is a subset of $\mathbb{R}^n$. The set of actions in the belief-state MDP is exactly the same

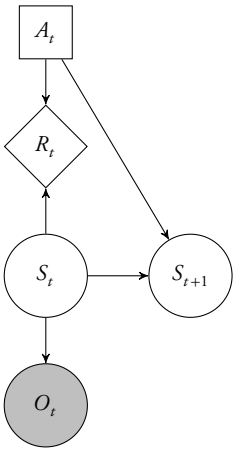

Figure 6.2 POMDP problem structure.

Algorithm 6.1 POMDP policy execution

```
function POMDPPolicyExecution(π)
    b ← initial belief state
    loop
        Execute action a = π(b)
        Observe o and reward r
        b ← UpdateBelief(b, a, o)
```

as for the POMDP. The state transition function $\tau(b' \mid b, a)$ is given by

$$\tau(b' \mid b, a) = P(b' \mid b, a) \tag{6.1}$$

$$= \sum_o P(b' \mid b, a, o) P(o \mid b_t, a_t) \tag{6.2}$$

$$= \sum_o P(b' \mid b, a, o) \sum_{s'} P(o \mid b, a, s') P(s' \mid b, a) \tag{6.3}$$

$$= \sum_o P(b' \mid b, a, o) \sum_{s'} O(o \mid s') \sum_s P(s' \mid b, a, s) P(s \mid b, a) \tag{6.4}$$

$$= \sum_o P(b' \mid b, a, o) \sum_{s'} O(o \mid s') \sum_s T(s' \mid s, a) b(s). \tag{6.5}$$

Here, $P(b' \mid b, a, o) = \delta_{b'}(\textsc{UpdateBelief}(b, a, o))$, with δ being the Kronecker delta function. The immediate reward in the belief-state MDP is

$$R(b, a) = \sum_s R(s, a) b(s). \tag{6.6}$$

Solving belief-state MDPs is challenging because the state space is continuous. We can use the approximate dynamic programming techniques presented in Section 4.5, but we can often do better by taking advantage of the structure of the belief-state MDP. This chapter will discuss such techniques after elaborating on how to update beliefs.

6.2 Belief Updating

Given an initial belief state, we can update our belief state using recursive Bayesian estimation based on the last observation and action executed. The update can be done exactly for problems with discrete states and for problems with linear-Gaussian dynamics and observations. For general problems with continuous state spaces, we often have to rely on approximation methods. This section presents methods for belief updating.

6.2.1 Discrete State Filter

In problems with a discrete state space, applying recursive Bayesian estimation is straightforward. Suppose our initial belief state is b, and we observe o after executing a. Our

new belief state b' is given by

$$
\begin{aligned}
b'(s') &= P(s' \mid o, a, b) && (6.7)\\
&\propto P(o \mid s', a, b) P(s' \mid a, b) && (6.8)\\
&\propto O(o \mid s', a) P(s' \mid a, b) && (6.9)\\
&\propto O(o \mid s', a) \sum_s P(s' \mid a, b, s) P(s \mid a, b) && (6.10)\\
&\propto O(o \mid s', a) \sum_s T(s' \mid a, s) b(s). && (6.11)
\end{aligned}
$$

The observation space can be continuous without posing any difficulty in computing the equation above exactly. The value $O(o \mid s', a)$ would represent a probability density rather than a probability mass.

To illustrate belief state updating, we will use the crying-baby problem. We simply apply Equation (6.11) to the model outlined in Section 6.1.1. Here are the first six steps of one potential scenario.

1. We begin with an initial belief state that assigns $b(h^0) = 0.5$ and $b(h^1) = 0.5$; in other words, uniform probability is assigned to whether the baby is hungry. If we had some prior belief that babies tend to be hungry more often than not, then we could have chosen a different initial belief. For compactness, we will represent our beliefs as tuples, $(b(h^0), b(h^1))$. In this case, our initial belief state is $(0.5, 0.5)$.
2. We do not feed the baby and the baby cries. According to Equation (6.11), the new belief state is $(0.0928, 0.9072)$. Although the baby is crying, it is only a noisy indication that the baby is actually hungry.
3. We feed the baby and the baby stops crying. Because we know that feeding the baby deterministically makes the baby not hungry, the result of our belief update is $(1, 0)$.
4. We do not feed the baby and the baby does not cry. In the prior step, we were certain that the baby was not hungry, and the dynamics specify that the baby becomes hungry at the next time step only 10% of the time. The fact that the baby is not crying further reduces our belief that the baby is hungry. Our new belief state is $(0.9759, 0.0241)$.
5. Again, we do not feed the baby and the baby does not cry. Our belief that the baby is hungry increases slightly. The new belief state is $(0.9701, 0.0299)$.
6. We do not feed the baby and the baby begins to cry. Our new belief is then $(0.4624, 0.5376)$. Because we were fairly certain that the baby was not hungry to begin with, our confidence that the baby is now hungry is significantly lower than when the baby started crying in the second step.

6.2.2 Linear-Gaussian Filter

If we generalize the linear-Gaussian dynamics in Section 4.4 to partial observability, we find that we can perform exact belief updates by using what is known as a *Kalman filter*. The dynamics and observations have the following form:

$$T(\mathbf{z} \mid \mathbf{s}, \mathbf{a}) = \mathcal{N}(\mathbf{z} \mid \mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a}, \Sigma_s) \tag{6.12}$$

$$O(\mathbf{o} \mid \mathbf{s}) = \mathcal{N}(\mathbf{o} \mid \mathbf{O}_s \mathbf{s}, \Sigma_o). \tag{6.13}$$

Hence, the continuous dynamics and observation models are specified using matrices $\mathbf{T}_s$, $\mathbf{T}_a$, Σ_s, $\mathbf{O}_s$, and Σ_o.

We assume that the initial belief state is represented by a Gaussian:

$$b(\mathbf{s}) = \mathcal{N}(\mathbf{s} \mid \mu_b, \Sigma_b). \tag{6.14}$$

Under the linear-Gaussian assumptions for the dynamics and observations, it can be shown that the belief state can be updated as follows:

$$\Sigma_b \leftarrow \mathbf{T}_s(\Sigma_b - \Sigma_b \mathbf{O}_s^\top (\mathbf{O}_s \Sigma_b \mathbf{O}_s^\top + \Sigma_o)^{-1} \mathbf{O}_s \Sigma_b)\mathbf{T}_s^\top + \Sigma_s \tag{6.15}$$

$$\mathbf{K} \leftarrow \mathbf{T}_s \Sigma_b \mathbf{O}_s^\top (\mathbf{O}_s \Sigma_b \mathbf{O}_s^\top + \Sigma_o)^{-1} \tag{6.16}$$

$$\mu_b \leftarrow \mathbf{T}_s \mu_b + \mathbf{T}_a \mathbf{a} + \mathbf{K}(\mathbf{o} - \mathbf{O}_s \mu_b). \tag{6.17}$$

The matrix $\mathbf{K}$ used for computing μ_b is called the *Kalman gain*. Kalman filters are often applied to systems that do not actually have linear-Gaussian dynamics. A variety of different modifications to the basic Kalman filter have been proposed to better accommodate nonlinear dynamics, as discussed in Section 6.7.

6.2.3 Particle Filter

If the state space is large or continuous and the dynamics are not well approximated by a linear-Gaussian model, then a sampling-based approach can be used to perform belief updates. The belief state is represented as a collection of *particles*, which are simply samples from the state space. The algorithm for adjusting these particles based on observations is known as a *particle filter*. There are many different variations of the particle filter, including versions in which the particles are assigned weights.

Our belief b is simply a set of samples from the state space. Updating b is based on a generative model G. We can draw a sample $(s', o') \sim G(s, a)$, which gives the next state s' and observation o' given the current state s and action a. The generative model can be implemented as a black-box simulator, without any explicit knowledge of the actual transition or observation probabilities.

Algorithm 6.2 returns the updated belief state b' based on the current belief state b, action a, and observation o. The process for generating a set of $|b|$ new particles is simple. Each sample is generated by randomly selecting a sample in b and then drawing samples $(s', o') \sim G(s, a)$ until the sampled o' matches the observed o. The sampled s' is then added to the new belief state b'.

Algorithm 6.2 Particle filter with rejection

```
function UpdateBelief(b, a, o)
    b' ← ∅
    for i ← 1 to |b|
        s ← random state in b
        repeat
            (s', o') ~ G(s, a)
        until o' = o
        Add s' to b'
    return b'
```

The problem with the version of the particle filter in Algorithm 6.2 is that many draws from the generative model may be required until the sampled observation is consistent with actual observation. The problem of rejecting many observation draws becomes especially apparent when the observation space is large or continuous. This issue was observed in Section 2.2.5 where we were trying to perform inference using direct sampling from a Bayesian network. The remedy presented in that section was to not sample the observed values but to weight the results using the likelihood of the observations.

Algorithm 6.3 is a version of a particle filter that does not involve rejecting samples. In this version, the generative model only returns states, instead of both states and observations. An observation model is required that specifies $O(o \mid s, a)$, which can be either a probability mass function or a probability density function depending on whether the observation space is continuous.

The algorithm is broken into two stages. The first stage involves generating $|b|$ new samples by randomly selecting samples in b and then propagating them forward using the generative model. For each of the new samples s'_i, we compute a corresponding weight w_i based on $O(o \mid s'_i, a)$, where o is the actual observation and a is the action taken. The second stage involves building the updated belief state b' by drawing $|b|$ samples from the set of new state samples with probability proportional to their weights.

In both versions of the particle filter presented here, it can be shown that as the number of particles increases, the distribution represented by the particles approaches the true posterior distribution. However, in practice, particle filters can fail. Because of the random nature of the particle filter, it is possible that a series of samples can lead

Algorithm 6.3 Particle filter without rejection

```
function UpdateBelief(b, a, o)
    b' ← ∅
    for i ← 1 to |b|
        s_i ← random state in b
        s'_i ~ G(s_i, a)
        w_i ← O(o | s'_i, a)
    for i ← 1 to |b|
        Randomly select k with probability proportional to w_k
        Add s'_k to b'
    return b'
```

to no particles near the true state. This problem, known as *particle deprivation*, can be mitigated to some extent by introducing additional noise to the particles.

6.3 Exact Solution Methods

As discussed earlier, a policy for a POMDP is a mapping from belief states to actions. This section explains how to compute and represent optimal policies.

6.3.1 Alpha Vectors

For now, let us assume that we are interested in computing the optimal policy for a discrete state POMDP with a one-step horizon. We know $U^*(s) = \max_a R(s,a)$, but because we do not know the state exactly in a POMDP, we have

$$U^*(b) = \max_a \sum_s b(s) R(s,a), \tag{6.18}$$

where b is our current belief state. If we let $\boldsymbol{\alpha}_a$ represent $R(\cdot, a)$ as a vector and $\mathbf{b}$ represent our belief state as a vector, then we can rewrite Equation (6.18) as

$$U^*(\mathbf{b}) = \max_a \boldsymbol{\alpha}_a^\top \mathbf{b}. \tag{6.19}$$

The α_a in the equation above is often referred to as an *alpha vector*. We have an alpha vector for each action in this single-step POMDP. These alpha vectors define hyperplanes in belief space. As can be seen in Equation (6.19), the optimal value function is piecewise-linear and convex.

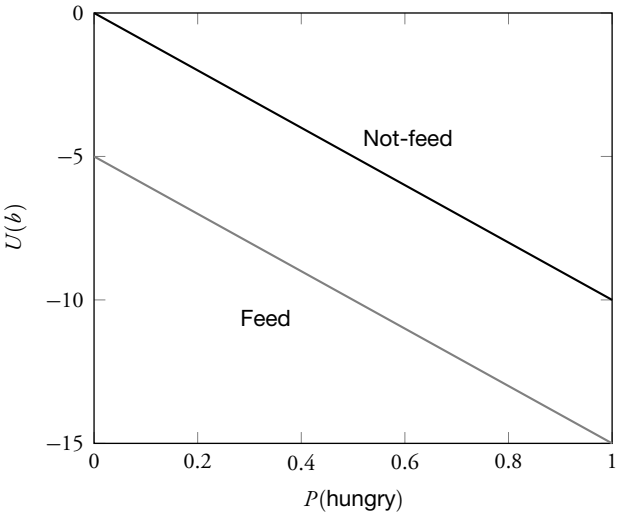

Figure 6.3 Alpha vectors for a one-step version of the crying-baby problem.

Figure 6.3 shows the alpha vectors associated with the crying-baby problem. If the belief vector is represented by the pair $(b(\text{not-hungry}), b(\text{hungry}))$, then the two alpha vectors are

$$\alpha_{\text{not-feed}} = (0, -10) \tag{6.20}$$

$$\alpha_{\text{feed}} = (-5, -15). \tag{6.21}$$

What is apparent from the plot is that regardless of our current beliefs, the one-step optimal policy is to not feed the baby. Given the dynamics of the problem, we do not see any potential benefit of feeding the baby until at least one step later.

6.3.2 Conditional Plans

The alpha vectors introduced in the computation of the optimal one-step policy can be generalized to an arbitrary horizon. In multistep POMDPs, we can think of a policy as a *conditional plan* represented as a tree. We start at the root node, which tells us what action to take at the first time step. After taking that action, we transition to one of the child nodes depending on what we observe. That child node tells us what action to take. We then proceed down the tree.

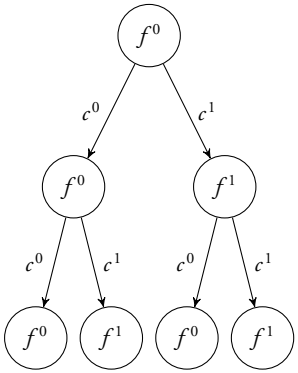

Figure 6.4 Example three-step conditional plan.

Figure 6.4 shows an example of a three-step plan for the crying-baby problem. The root node specifies that we do not feed the baby at the first step. For the second time step, as indicated by the directed edges, we feed the baby if there is crying; otherwise, we do not. At the third time step, we again feed the baby only if there is crying.

We can recursively compute $U^p(s)$, the expected utility associated with conditional plan p when starting in state s:

$$U^p(s) = R(s,a) + \sum_{s'} T(s' \mid s,a) \sum_{o} O(o \mid s',a) U^{p(o)}(s'), \tag{6.22}$$

where a is the action associated with the root node of p and $p(o)$ represents the subplan associated with observation o. We can compute the expected utility associated with a belief state as follows:

$$U^p(b) = \sum_{s} U^p(s) b(s). \tag{6.23}$$

We can use the alpha vector $\boldsymbol{\alpha}_p$ to represent the vectorized version of U^p. If $\mathbf{b}$ is the belief vector, then we can write

$$U^p(\mathbf{b}) = \boldsymbol{\alpha}_p^\top \mathbf{b}. \tag{6.24}$$

If we maximize over the space of all possible plans up to the planning horizon, then we can find

$$U^*(\mathbf{b}) = \max_p \boldsymbol{\alpha}_p^\top \mathbf{b}. \tag{6.25}$$

Hence, the finite horizon optimal value function is piecewise-linear and convex. We simply execute the action at the root node of the plan that maximizes $\boldsymbol{\alpha}_p^\top \mathbf{b}$.

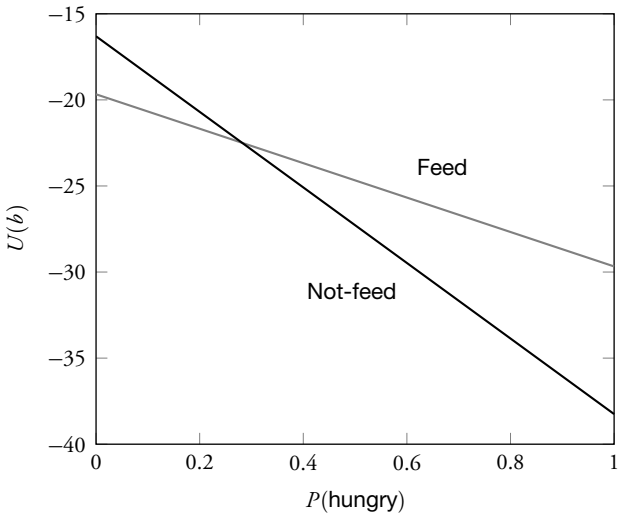

Figure 6.5 Optimal policy for the crying-baby problem.

6.3.3 Value Iteration

It is generally infeasible to enumerate every possible h-step plan to find the one that maximizes Equation (6.25) from the current belief state; the number of nodes in an h-step conditional plan is $(|O|^h - 1)/(|O| - 1)$. Associated with each node are $|A|$ actions. Hence, we have $|A|^{(|O|^h-1)/(|O|-1)}$ possible h-step plans. Even for our crying-baby problem with two actions and two observations, there are 2^{63} six-step conditional plans—too many to enumerate.

The idea in POMDP value iteration is to iterate over all the one-step plans and toss out the plans that are not optimal for any initial belief state. The remaining one-step plans are then used to generate potentially optimal two-step plans. Again, plans that are not optimal for any belief state are discarded. The process repeats until the desired horizon is reached. Identifying the plans that are *dominated* at certain belief states by other plans can be done using linear programming.

Figure 6.5 shows the two, nondominated, alpha vectors for the crying-baby problem with a discount factor of 0.9. The two alpha vectors intersect when $P(\text{hungry}) = 0.28206$. As indicated in the figure, we only want to feed the baby if $P(\text{hungry}) > 0.28206$. Of course, for this problem, it would have been easier to simply store this threshold instead of using alpha vectors, but for higher dimensional problems, alpha vectors often provide a compact representation of the policy.

Discarding dominated plans can significantly reduce the computation required to find the optimal set of alpha vectors. For many problems, the majority of the potential plans are dominated by at least one other plan. However, in the worst case, an exact solution for a general finite-horizon POMDP is *PSPACE-complete*, which is a complexity class that includes NP-complete problems and is suspected to include problems even more difficult. General infinite-horizon POMDPs have been shown to be *uncomputable*. Hence, there has been a tremendous amount of research recently on approximation methods, which will be discussed in the remainder of this chapter.

6.4 Offline Methods

Offline POMDP solution methods involve performing all or most of the computation prior to execution. In practice, we are generally restricted to finding only approximately optimal solutions. Some methods represent policies as alpha vectors, whereas others use finite-state controllers.

6.4.1 Fully Observable Value Approximation

A simple approximation technique is called *QMDP*. The idea is to create a set of alpha vectors, one for each action, based on the state-action value function $Q(s,a)$ under full observability. We can use value iteration to compute the alpha vectors. If we initialize $\alpha_a^{(0)}(s) = 0$ for all s, then we can iterate

$$\alpha_a^{(k+1)}(s) = R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \max_{a'} \alpha_{a'}^{(k)}(s'). \tag{6.26}$$

Each iteration requires $O(|A|^2|S|^2)$ operations. As $k \to \infty$, the resulting set of $|A|$ alpha vectors can be used to estimate the value function. The value function at belief state $\mathbf{b}$ is given by $\max_a \boldsymbol{\alpha}_a^\top \mathbf{b}$, and the approximately optimal action is given by $\arg\max_a \boldsymbol{\alpha}_a^\top \mathbf{b}$.

The QMDP method assumes all state uncertainty disappears at the next time step. It can be shown that QMDP provides an upper bound on the value function. In other words, $\max_a \boldsymbol{\alpha}_a^\top \mathbf{b} \geq U^*(\mathbf{b})$ for all $\mathbf{b}$. QMDP tends to have difficulty with problems with information-gathering actions, such as "look over your right shoulder when changing lanes." However, the method performs extremely well in many real problems in which the particular choice of action has little impact on the reduction in state uncertainty.

6.4.2 Fast Informed Bound

Just as with the QMDP approximation, the *fast informed bound* (FIB) method computes a single alpha vector for each action. However, the fast informed bound takes into account, to some extent, partial observability. Instead of using the iteration of

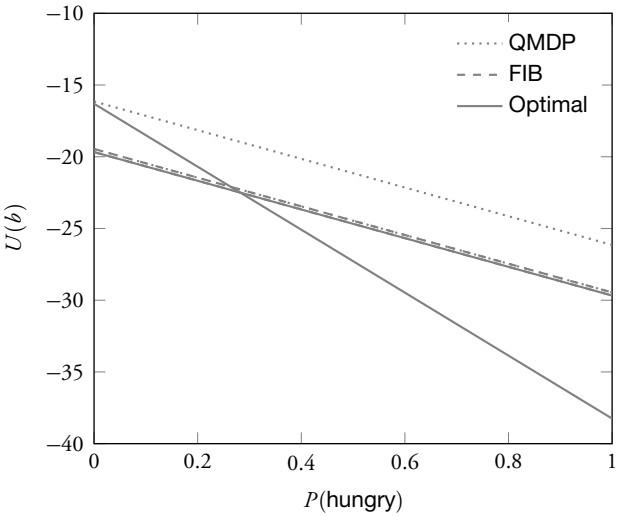

Figure 6.6 QMDP, FIB, and optimal alpha vectors for the crying-baby problem.

Equation (6.26), we use

$$\alpha_a^{(k+1)}(s) = R(s,a) + \gamma \sum_o \max_{a'} \sum_{s'} O(o \mid s', a) T(s' \mid s, a) \alpha_{a'}^{(k)}(s'). \tag{6.27}$$

Each iteration requires $O(|A|^2|S|^2|O|)$ operations, only a factor of $|O|$ more than QMDP. At all belief states, the fast informed bound provides an upper bound on the optimal value function that is no higher than that of QMDP. Figure 6.6 compares the alpha vectors associated with QMDP, FIB, and the optimal policy for the crying baby problem.

6.4.3 Point-Based Value Iteration

There is a family of approximation methods that involves backing up alpha vectors associated with a limited number of points in the belief space. Let us denote the set of belief points $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ and their associated alpha vectors $\Gamma = \{\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n\}$. Given these n alpha vectors, we can estimate the value function at any new point $\mathbf{b}$ as follows:

$$U^\Gamma(\mathbf{b}) = \max_{\boldsymbol{\alpha} \in \Gamma} \boldsymbol{\alpha}^\top \mathbf{b} = \max_{\alpha \in \Gamma} \sum_s \alpha(s) b(s). \tag{6.28}$$

For the moment, let us assume that these belief points are given to us; we will discuss how to choose these later in Section 6.4.5. We would like to initialize the alpha vectors in Γ such that $U^{\Gamma}(\mathbf{b}) \leq U^{*}(\mathbf{b})$ for all $\mathbf{b}$. One way to compute such a lower bound is to initialize all the components of all n alpha vectors to

$$\max_a \sum_{t=0}^{\infty} \gamma^t \min_s R(s,a) = \frac{1}{1-\gamma} \max_a \min_s R(s,a). \tag{6.29}$$

As we perform backups starting with this initial set of alpha vectors, we guarantee that $U(\mathbf{b})$ at each iteration never decreases for any $\mathbf{b}$.

We may update the value function at belief b based on the n alpha vectors

$$U(b) \leftarrow \max_a \left[R(b,a) + \gamma \sum_o P(o \mid a, b) U(b') \right], \tag{6.30}$$

where b' is as determined by UpdateBelief(b,a,o), $U(b')$ is as evaluated by Equation (6.28), and

$$P(o \mid b, a) = \sum_s O(o \mid s, a) b(s). \tag{6.31}$$

We know from Bayes' rule that

$$b'(s') = \frac{O(o \mid s', a)}{P(o \mid b, a)} \sum_s T(s' \mid s, a) b(s). \tag{6.32}$$

Combining Equations (6.28), (6.30), and (6.32) and simplifying, we get the update

$$U(b) \leftarrow \max_a \left[R(b,a) + \gamma \sum_o \max_{\alpha \in \Gamma} \sum_s b(s) \sum_{s'} O(o \mid s', a) T(s' \mid s, a) \alpha(s') \right]. \tag{6.33}$$

Besides simply updating the value at $\mathbf{b}$, we can compute an alpha vector at $\mathbf{b}$ by using Algorithm 6.4. Point-based value iteration approximation algorithms update the alpha vectors at the n belief states until convergence. These n alpha vectors can then be used to approximate the value function anywhere in the belief space.

6.4.4 Randomized Point-Based Value Iteration

The point-based value iteration approach discussed in Section 6.4.3 associates an alpha vector for each of the selected points in the belief space. To reduce the amount of computation required to perform an update of all the belief points, we can attempt to limit the number of alpha vectors representing the value function using the approach outlined in Algorithm 6.5.

Algorithm 6.4 Backup belief

```
function BackupBelief(Γ, b)
    for a ∈ A
        for o ∈ O
            b′ ← UpdateBelief(b, a, o)
            α_{a,o} ← arg max_{α∈Γ} α⊤b′
        for s ∈ S
            α_a(s) ← R(s,a) + γ Σ_{s′,o} O(o | s′, a) T(s′ | s, a) α_{a,o}(s′)
    α ← arg max_{α_a} α_a⊤ b
    return α
```

The algorithm begins by initializing Γ with a single alpha vector with all components set to Equation (6.29), which lower bounds the value function. Given this Γ and our belief points B, we call RandomizedPointBasedUpdate(B, Γ) to create a new set of alpha vectors that provides a tighter lower bound on the value function. These new alpha vectors can be improved on further through another call to this function. The process repeats until convergence.

Each update involves finding a set of alpha vectors Γ' that improves on the value function represented by Γ at the points in B. In other words, the update finds a set Γ' such that $U^{\Gamma'}(\mathbf{b}) \geq U^{\Gamma}(\mathbf{b})$ for all $\mathbf{b} \in B$. We begin by initializing Γ' to the empty set and the set B' to B. We then take a point $\mathbf{b}$ randomly from B' and call BackupBelief$(\mathbf{b}, \Gamma)$ to get a new alpha vector $\boldsymbol{\alpha}$. If this alpha vector improves the value at $\mathbf{b}$, then we add it to Γ'; otherwise, we find the alpha vector in Γ that dominates at $\mathbf{b}$ and add it to Γ'. The set B' then becomes the set of points that still have not been improved by Γ'. At each iteration, B' becomes smaller, and the process terminates when B' is empty.

6.4.5 Point Selection

Many point-based value iteration algorithms involve starting with B initialized to a set containing only the initial belief state b_0 and then iteratively expanding that set. One of the simplest ways to expand a set B is to select actions from each belief state B (based on some exploration strategy from Section 5.1.3) and then add the resulting belief states to B (Algorithm 6.6). This process requires sampling observations from a belief state given an action (Algorithm 6.7).

Other approaches attempt to disperse the points throughout the reachable state space. For example, Algorithm 6.8 iterates through B, tries each available action, and adds the new belief states that are furthest from any of the points already in the set. There are many ways to measure distance between two belief states; the algorithm shown uses the L_1 distance metric, where the distance between b and b' is given by $\sum_s |b(s) - b'(s)|$.

Algorithm 6.5 Randomized point-based backup

```
function RandomizedPointBasedUpdate(B, Γ)
    Γ' ← ∅
    B' ← B
    repeat
        b ← belief point sampled uniformly at random from set B'
        α ← BackupBelief(b, Γ)
        if α⊤b ≥ U^Γ(b)
            Add α to Γ'
        else
            Add α' = arg max_{α∈Γ} α⊤b to Γ'
        B' ← {b ∈ B | U^Γ'(b) < U^Γ(b)}
    until B' = ∅
    return Γ'
```

Algorithm 6.6 Expand belief points with random actions

```
function ExpandBeliefPoints(B)
    B' ← B
    for b ∈ B
        a ← random action selected uniformly from action space A
        o ← SampleObservation(b, a)
        b' ← UpdateBelief(b, a, o)
        Add b' to B'
    return B'
```

Algorithm 6.7 Sample observation

```
function SampleObservation(b, a)
    s ~ b
    s' ← random state selected with probability T(s' | s, a)
    o ← random observation selected with probability O(o | a, s')
    return o
```

Algorithm 6.8 Expand belief points with exploratory actions

```
function ExpandBeliefPoints(B)
    B′ ← B
    for b ∈ B
        for a ∈ A
            o ← SampleObservation(b, a)
            b_a ← UpdateBelief(b, a, o)
        a ← arg max_a min_{b′∈B′} Σ_s |b′(s) − b_a(s)|
        Add b_a to B′
    return B′
```

6.4.6 Linear Policies

As discussed in Section 6.2.2, the belief state in a problem with linear-Gaussian dynamics can be represented by a Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}_b, \boldsymbol{\Sigma}_b)$. If the reward function is quadratic, as assumed in Section 4.4, then it can be shown that the optimal policy can be computed exactly offline. In fact, the solution is identical to the perfect observability case outlined in Section 4.4, but the $\boldsymbol{\mu}_b$ computed by the Kalman filter is used in place of the true state. With each observation, we simply use the Kalman filter to update our $\boldsymbol{\mu}_b$, and then we matrix multiply $\boldsymbol{\mu}_b$ with the policy matrix given in Section 4.4 to determine the optimal action.

6.5 Online Methods

Online methods determine the optimal policy by planning from the current belief state. The belief states reachable from the current state are typically small compared with the full belief space. Many online methods use a depth-first tree-based search up to some horizon. The time complexity of these online algorithms is generally exponential in the horizon. Although online methods require more computation per decision step during execution than offline approaches, online methods are sometimes easier to apply to high-dimensional problems.

6.5.1 Lookahead with Approximate Value Function

We can use the one-step lookahead strategy online to improve on a policy that has been computed offline. If b is the current belief state, then the one-step lookahead policy is given by

$$\pi(b) = \arg\max_a \left[R(b,a) + \gamma \sum_o P(o \mid b, a) U(\textsc{UpdateBelief}(b, a, o)) \right], \tag{6.34}$$

where U is an approximate value function. This approximate value function may be represented by alpha vectors computed offline using strategies such as QMDP, fast informed bound, or point-based value iteration discussed earlier. Experiments have shown that for many problems, a one-step lookahead can significantly improve performance over the base offline strategy.

The approximate value function may also be estimated by sampling from a rollout policy as introduced in Section 4.6.4 but with modifications to handle partial observability as shown in Algorithm 6.9. The generative model G returns a sampled next state s' and reward r given state s and action a. This algorithm uses a single rollout policy π_0, but we can use a set of rollout policies and evaluate them in parallel. The policy that results in the largest value is the one that is used to estimate the value at that particular belief state.

Algorithm 6.9 Rollout evaluation

```
function ROLLOUT(b, d, π_0)
    if d = 0
        return 0
    a ~ π_0(b)
    s ~ b
    (s', o, r) ~ G(s, a)
    b' ← UPDATEBELIEF(b, a, o)
    return r + γROLLOUT(b', d − 1, π_0)
```

An alternative to summing over all possible observations in Equation (6.34) is to use sampling. We can generate n observations for each action through independent calls to SAMPLEOBSERVATION(b, a) and then compute

$$\pi(b) = \arg\max_a \left[R(b,a) + \gamma \frac{1}{n} \sum_{i=1}^{n} U(\text{UPDATEBELIEF}(b, a, o_{a,i})) \right]. \tag{6.35}$$

This strategy is particularly useful when the observation space is large.

6.5.2 Forward Search

The one-step lookahead strategy can be extended to look an arbitrary depth into the future. Algorithm 6.10 defines the function SELECTACTION(b, d, U), which returns the pair (a^*, u^*) defining the best action and the expected utility, given the current belief b, depth d, and approximate value function U.

When $d = 0$, no action is able to be selected, and so a^* is NIL and the utility is $U(b)$. When $d > 0$, we compute the value for every available action and return the best action and its associated value. To compute the value for an action a, we evaluate

$$R(b,a) + \gamma \sum_o P(o \mid b,a) U_{d-1}(\text{UpdateBelief}(b,a,o)). \tag{6.36}$$

In the equation above, $U_{d-1}(b')$ is the expected utility returned by the recursive call SelectAction$(b', d-1)$. The complexity is given by $O(|A|^d |O|^d)$. Algorithm 6.10 can be modified to sample n observations instead of enumerating all $|O|$ of them, similar to what is done in Equation (6.35). The complexity then becomes $O(|A|^d n^d)$.

Algorithm 6.10 Forward search online policy

```
function SelectAction(b, d)
    if d = 0
        return (NIL, U(b))
    (a*, u*) ← (NIL, −∞)
    for a ∈ A
        u ← R(b, a)
        for o ∈ O
            b' ← UpdateBelief(b, a, o)
            (a', u') ← SelectAction(b', d − 1)
            u ← u + γ P(o | b, a) u'
        if u > u*
            (a*, u*) ← (a, u)
    return (a*, u*)
```

6.5.3 Branch and Bound

The branch and bound technique originally introduced in Section 4.6.2 in the context of MDPs can be easily extended to POMDPs. As with the POMDP version of forward search, we have to iterate over the observations and update beliefs—otherwise the algorithm is nearly identical to the MDP version. Again, the ordering of the actions in the for loop is important. To prune as much of the search space as possible, the actions should be enumerated such that their upper bounds decrease in value. In other words, action a_i comes before a_j if $\overline{U}(b,a_i) \geq \overline{U}(b,a_j)$.

We can use QMDP or the fast informed bound as the upper bound function $\overline{U}$. For the lower bound function $\underline{U}$, we can use the value function associated with a *blind policy*, which selects the same action regardless of the current belief state. This value

function can be represented by a set of $|A|$ alpha vectors, which can be computed as follows:

$$\alpha_a^{(k+1)}(s) = R(s,a) + \gamma \sum_{s'} T(s' \mid s,a)\alpha_a^{(k)}(s'), \tag{6.37}$$

where $\alpha_a^{(0)} = \min_s R(s,a)/(1-\gamma)$. Equation (6.37) is similar to the QMDP equation in Equation (6.26) except that it does not have a maximization over the alpha vectors on the right-hand side.

So long as $\underline{U}$ and $\overline{U}$ are true lower and upper bounds, the result of the branch and bound algorithm will be the same as the forward search algorithm with $\underline{U}$ as the approximate value function. In practice, branch and bound can significantly reduce the amount of computation required to select an action. The tighter the upper and lower bounds, the more of the search space branch and bound can prune. However, in the worst case, the complexity of branch and bound is no better than that of forward search.

Algorithm 6.11 Branch and bound online policy

```
function SelectAction(b, d)
    if d = 0
        return (nil, U(b))
    (a*, u) ← (nil, −∞)
    for a ∈ A
        if Ū(b, a) ≤ u
            return (a*, u)
        u ← R(b, a)
        for o ∈ O
            b′ ← UpdateBelief(b, a, o)
            (a′, u′) ← SelectAction(b′, d − 1)
            u ← u + γP(o | b, a)u′
        if u > u
            (a*, u) ← (a, u)
    return (a*, u)
```

6.5.4 Monte Carlo Tree Search

The Monte Carlo tree search approach for MDPs can be extended to POMDPs, as outlined in Algorithm 6.12. The input to the algorithm is a belief state b, depth d, and rollout policy π_0. The main difference between the POMDP algorithm and the MDP algorithm in Section 4.6.4 is that the counts and values are associated with *histories* instead of states. A history is a sequence of past observations and actions. For example,

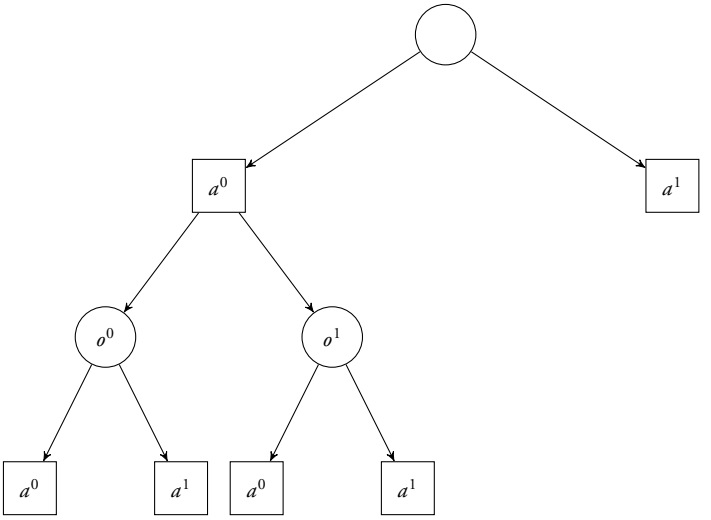

Figure 6.7 Example Monte Carlo tree search history tree.

if we have two actions a^0 and a^1 and two observations o^0 and o^1, then a possible history could be the sequence $h = a^0 o^1 a^1 o^1 a^0 o^0$. During the execution of the algorithm, we update the value estimates $Q(h, a)$ and counts $N(h, a)$ for a set of history-action pairs.

The histories associated with Q and N may be organized in a tree like the one in Figure 6.7. The root node represents the empty history starting from the initial belief state b. During the execution of the algorithm, the tree structure expands. The layers of the tree alternate between action nodes and observation nodes. Associated with each action node are values $Q(h, a)$ and $N(h, a)$, where the history is determined by the path to the root node. In the algorithm, $N(h) = \sum_a N(h, a)$.

As with the MDP version, the Monte Carlo tree search algorithm is an anytime algorithm. The loop in SelectAction(b, d) can be terminated at any time, and some solution will be returned. It has been shown that, with a sufficient number of iterations, the algorithm converges to the optimal action.

Prior knowledge can be incorporated into this algorithm through the choice of initialization parameters N_0 and Q_0 as well as the rollout policy. The algorithm does not need to be reinitialized with each decision. The history tree and associated counts and value estimates can be maintained between calls. The observation node associated with the selected action and actual observation becomes the root node at the next time step.

Algorithm 6.12 Monte Carlo tree search

```
function SelectAction(b, d)
    h ← ∅
    loop
        s ~ b
        Simulate(s, h, d)
    return arg max_a Q(h, a)
function Simulate(s, h, d)
    if d = 0
        return 0
    if h ∉ T
        for a ∈ A(s)
            (N(h, a), Q(h, a)) ← (N_0(h, a), Q_0(h, a))
        T = T ∪ {h}
        return Rollout(s, d, π_0)
    a ← arg max_a Q(h, a) + c √(log N(h) / N(h, a))
    (s′, o, r) ~ G(s, a)
    q ← r + γ Simulate(s′, hao, d − 1)
    N(h, a) ← N(h, a) + 1
    Q(h, a) ← Q(h, a) + (q − Q(h, a)) / N(h, a)
    return q
```

6.6 Summary

- POMDPs are MDPs over belief states.
- POMDPs are difficult to solve exactly in general but can often be approximated well.
- Policies can be represented as alpha vectors.
- Large problems can often be solved online.

6.7 Further Reading

It was observed in the 1960s that POMDPs can be transformed into MDPs over belief states [1]. Belief state updating in discrete state spaces is a straightforward application of Bayes' rule. A thorough introduction to the Kalman filter and its variants is provided in *Estimation with Applications to Tracking and Navigation* by Bar-Shalom, Li, and Kirubarajan [2]. Arulampalam et al. provide a tutorial on particle filters [3]. *Probabilistic Robotics* by Thrun, Burgard, and Fox discusses different methods for belief updating in the context of robotic applications [4].

Exact solution methods for POMDPs were originally proposed by Smallwood and Sondik [5] and Sondik [6] in the 1970s. There are several surveys of early work on POMDPs [7]–[9]. Kaelbling, Littman, and Cassandra present techniques for identifying dominated plans to improve the efficiency of exact solution methods [10]. Computing exact solutions to POMDPs is intractable in general [11], [12].

Approximation methods for POMDPs have been the focus of considerable research recently. Hauskrecht discusses the relationship between QMDP and the fast informed bound and presents empirical results [13]. Offline approximate POMDP solution algorithms have focused on point-based approximation techniques, as surveyed by Shani, Pineau, and Kaplow [14]. The point-based value iteration (PBVI) algorithm was proposed by Pineau, Gordon, and Thrun [15]. There are other, more involved, point-based value iteration algorithms. Two of the best algorithms, Heuristic Search Value Iteration (HSVI) [16], [17] and Successive Approximations of the Reachable Space under Optimal Policies (SARSOP) [18], involve building search trees through belief space and maintaining upper and lower bounds on the value function. The randomized point-based value iteration algorithm discussed in Section 6.4.4 is based on the Perseus algorithm presented by Spaan and Vlassis [19]. Controller-based solutions have also been explored for concisely representing policies for infinite-horizon problems and removing the need for belief updating during execution [20], [21]. Several online solution methods are surveyed by Ross et al. [22]. Silver and Veness present a Monte Carlo tree search algorithm for POMDPs called Partially Observable Monte Carlo Planning (POMCP) [23].

References

1. K.J. Åström, "Optimal Control of Markov Processes with Incomplete State Information," *Journal of Mathematical Analysis and Applications*, vol. 10, no. 1, pp. 174–205, 1965. DOI: 10.1016/0022-247X(65)90154-X.
2. Y. Bar-Shalom, X.R. Li, and T. Kirubarajan, *Estimation with Applications to Tracking and Navigation*. New York: Wiley, 2001.
3. M.S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A Tutorial on Particle Filters for Online Nonlinear / Non-Gaussian Bayesian Tracking," *IEEE Transactions on Signal Processing*, vol. 50, no. 2, pp. 174–188, 2002. DOI: 10.1109/78.978374.
4. S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. Cambridge, MA: MIT Press, 2006.
5. R.D. Smallwood and E.J. Sondik, "The Optimal Control of Partially Observable Markov Processes Over a Finite Horizon," *Operations Research*, vol. 21, no. 5, pp. 1071–1088, 1973.
6. E.J. Sondik, "The Optimal Control of Partially Observable Markov Processes Over the Infinite Horizon: Discounted Costs," *Operations Research*, vol. 26, no. 2, pp. 282–304, 1978.
7. C.C. White III, "A Survey of Solution Techniques for the Partially Observed Markov Decision Process," *Annals of Operations Research*, vol. 32, no. 1, pp. 215–230, 1991. DOI: 10.1007/BF02204836.
8. W.S. Lovejoy, "A Survey of Algorithmic Methods for Partially Observed Markov Decision Processes," *Annals of Operations Research*, vol. 28, no. 1, pp. 47–65, 1991. DOI: 10.1007/BF02055574.
9. G.E. Monahan, "A Survey of Partially Observable Markov Decision Processes: Theory, Models, and Algorithms," *Management Science*, vol. 28, no. 1, pp. 1–16, 1982.
10. L.P. Kaelbling, M.L. Littman, and A.R. Cassandra, "Planning and Acting in Partially Observable Stochastic Domains," *Artificial Intelligence*, vol. 101, no. 1–2, pp. 99–134, 1998. DOI: 10.1016/S0004-3702(98)00023-X.
11. C. Papadimitriou and J. Tsitsiklis, "The Complexity of Markov Decision Processes," *Mathematics of Operation Research*, vol. 12, no. 3, pp. 441–450, 1987. DOI: 10.1287/moor.12.3.441.
12. O. Madani, S. Hanks, and A. Condon, "On the Undecidability of Probabilistic Planning and Related Stochastic Optimization Problems," *Artificial Intelligence*, vol. 147, no. 1-2, pp. 5–34, 2003. DOI: 10.1016/S0004-3702(02)00378-8.

13. M. Hauskrecht, "Value-Function Approximations for Partially Observable Markov Decision Processes," *Journal of Artificial Intelligence Research*, vol. 13, pp. 33–94, 2000. DOI: 10.1613/jair.678.

14. G. Shani, J. Pineau, and R. Kaplow, "A Survey of Point-Based POMDP Solvers," *Autonomous Agents and Multi-Agent Systems*, pp. 1–51, 2012. DOI: 10.1007/s10458-012-9200-2.

15. J. Pineau, G.J. Gordon, and S. Thrun, "Anytime Point-Based Approximations for Large POMDPs," *Journal of Artificial Intelligence Research*, vol. 27, pp. 335–380, 2006. DOI: 10.1613/jair.2078.

16. T. Smith and R.G. Simmons, "Heuristic Search Value Iteration for POMDPs," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2004.

17. ——, "Point-Based POMDP Algorithms: Improved Analysis and Implementation," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2005.

18. H. Kurniawati, D. Hsu, and W.S. Lee, "SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces," in *Robotics: Science and Systems*, 2008.

19. M.T.J. Spaan and N.A. Vlassis, "Perseus: Randomized Point-Based Value Iteration for POMDPs," *Journal of Artificial Intelligence Research*, vol. 24, pp. 195–220, 2005. DOI: 10.1613/jair.1659.

20. P. Poupart and C. Boutilier, "Bounded Finite State Controllers," in *Advances in Neural Information Processing Systems (NIPS)*, 2003.

21. C. Amato, D.S. Bernstein, and S. Zilberstein, "Solving POMDPs Using Quadratically Constrained Linear Programs," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2007.

22. S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa, "Online Planning Algorithms for POMDPs," *Journal of Artificial Intelligence Research*, vol. 32, pp. 663–704, 2008. DOI: 10.1613/jair.2567.

23. D. Silver and J. Veness, "Monte-Carlo Planning in Large POMDPs," in *Advances in Neural Information Processing Systems (NIPS)*, 2010.