21 *Offline Belief State Planning*

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 that are even more difficult.[1] General infinite-horizon POMDPs have been shown to be uncomputable.[2] Hence, there has been a tremendous amount of research recently on approximation methods. This chapter discusses various offline POMDP solution methods, which involve performing all or most of the computation prior to execution. We focus on methods that represent the value function as alpha vectors and different forms of interpolation.

21.1 *Fully Observable Value Approximation*

One of the simplest offline approximation techniques is *QMDP*, which derives its name from the action value function associated with a fully observed MDP.[3] This approach, as well as several others discussed in this chapter, involve iteratively updating a set Γ of alpha vectors, as shown in algorithm 21.1. The resulting set Γ defines a value function and a policy that can be used directly or with one-step lookahead as discussed in the previous chapter, though the resulting policy will only be an approximation of the optimal solution.

```
function alphavector_iteration(𝒫::POMDP, M, Γ)
    for k in 1:M.k_max
        Γ = update(𝒫, M, Γ)
    end
    return Γ
end
```

Algorithm 21.1. Iteration structure for updating a set of alpha vectors `Γ` used by several of the methods in this chapter. The various methods, including QMDP, differ in their implementation of `update`. After `k_max` iterations, this function returns a policy represented by the alpha vectors in `Γ`.

[1] C. Papadimitriou and J. Tsitsiklis, "The Complexity of Markov Decision Processes," *Mathematics of Operation Research*, vol. 12, no. 3, pp. 441–450, 1987.

[2] 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.

[3] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling, "Learning Policies for Partially Observable Environments: Scaling Up," in *International Conference on Machine Learning (ICML)*, 1995. A proof that QMDP provides an upper bound on the optimal value function is given by M. Hauskrecht, "Value-Function Approximations for Partially Observable Markov Decision Processes," *Journal of Artificial Intelligence Research*, vol. 13, pp. 33–94, 2000.

QMDP (algorithm 21.2) constructs a single alpha vector $\boldsymbol{\alpha}_a$ for each action a using value iteration. Each alpha vector is initialized to zero, and then we iterate:

$$\alpha_a^{(k+1)}(s) = R(s,a) + \gamma \sum_{s'} T(s' \mid s, a) \max_{a'} \alpha_{a'}^{(k)}(s') \tag{21.1}$$

Each iteration requires $O(|\mathcal{A}|^2|\mathcal{S}|^2)$ operations. Figure 21.1 illustrates the process.

```
struct QMDP
    k_max # maximum number of iterations
end

function update(𝒫::POMDP, M::QMDP, Γ)
    𝒮, 𝒜, R, T, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.T, 𝒫.γ
    Γ′ = [[R(s,a) + γ*sum(T(s,a,s′)*maximum(α′[j] for α′ in Γ)
        for (j,s′) in enumerate(𝒮)) for s in 𝒮] for a in 𝒜]
    return Γ′
end

function solve(M::QMDP, 𝒫::POMDP)
    Γ = [zeros(length(𝒫.𝒮)) for a in 𝒫.𝒜]
    Γ = alphavector_iteration(𝒫, M, Γ)
    return AlphaVectorPolicy(𝒫, Γ, 𝒫.𝒜)
end
```

Algorithm 21.2. The QMDP algorithm, which finds an approximately optimal policy for an infinite-horizon POMDP with a discrete state and action space, where `k_max` is the number of iterations. QMDP assumes perfect observability.

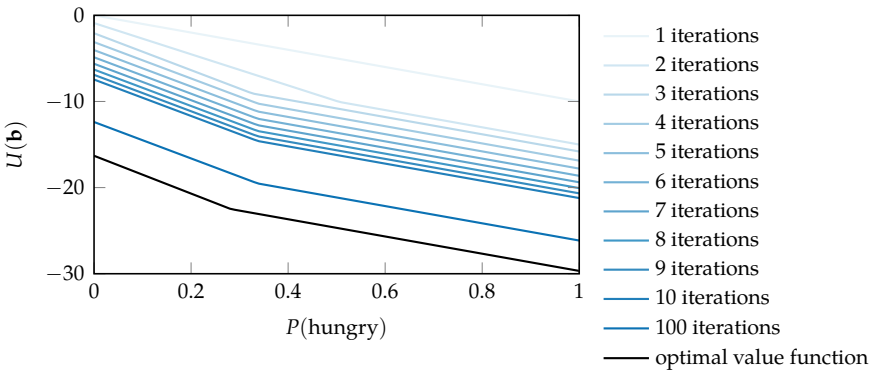


Figure 21.1. Value functions obtained for the crying baby problem (appendix F.7) using QMDP. In the first iteration, a single alpha vector dominates. In subsequent iterations, two alpha vectors dominate.

When QMDP is run to the horizon in finite horizon problems or to convergence for infinite-horizon problems, the resulting policy is equivalent to assuming that there will be full observability after taking the first step. Because we can do better

only if we have full observability, QMDP will produce an upper bound on the true optimal value function $U^*(\mathbf{b})$. In other words, $\max_a \boldsymbol{\alpha}_a^\top \mathbf{b} \geq U^*(\mathbf{b})$ for all $\mathbf{b}$.[4]

If QMDP is not run to convergence for infinite-horizon problems, it might not provide an upper bound. One way to guarantee that QMDP will provide an upper bound after a finite number of iterations is to initialize the value function to some upper bound. One rather loose upper bound is the *best-action best-state upper bound*, which is the utility obtained from taking the best action from the best state forever:

$$\overline{U}(b) = \max_{s,a} \frac{R(s,a)}{1-\gamma} \tag{21.2}$$

The assumption of full observability after the first step can cause QMDP to poorly approximate the value of *information-gathering* actions, which are actions that significantly reduce the uncertainty in the state. For example, looking over one's shoulder before changing lanes when driving is an information-gathering action. QMDP can perform well in problems where the optimal policy does not include costly information gathering.

We can generalize the QMDP approach to problems that may not have a small, discrete state space. In such problems, the iteration in equation (21.1) may not be feasible, but we may use one of the many methods discussed in earlier chapters for obtaining an approximate action value function $Q(s,a)$. This value function might be defined over a high-dimensional, continuous state space using, for example, a neural network representation. The value function evaluated at a belief point is, then,

$$U(b) = \max_a \int Q(s,a)b(s)\,\mathrm{d}s \tag{21.3}$$

The integral above may be approximated through sampling.

21.2 Fast Informed Bound

As with QMDP, the *fast informed bound* computes one alpha vector for each action. However, the fast informed bound takes into account, to some extent, the observation model.[5] The iteration is

$$\alpha_a^{(k+1)}(s) = R(s,a) + \gamma \sum_o \max_{a'} \sum_{s'} O(o \mid a, s')T(s' \mid s, a)\alpha_{a'}^{(k)}(s') \tag{21.4}$$

which requires $O(|\mathcal{A}|^2|\mathcal{S}|^2|\mathcal{O}|)$ operations per iteration.

[4] Although the value function represented by the QMDP alpha vectors upper-bounds the optimal value function, the utility realized by a QMDP policy will not exceed that of an optimal policy in expectation, of course.

[5] The relationship between QMDP and the fast informed bound, together with empirical results, are discussed by M. Hauskrecht, "Value-Function Approximations for Partially Observable Markov Decision Processes," *Journal of Artificial Intelligence Research*, vol. 13, pp. 33–94, 2000.

The fast informed bound provides an upper bound on the optimal value function. That upper bound is guaranteed to be no looser than that provided by QMDP, and it also tends to be tighter. The fast informed bound is implemented in algorithm 21.3 and is used in figure 21.2 to compute optimal value functions.

```
struct FastInformedBound
    k_max # maximum number of iterations
end

function update(𝒫::POMDP, M::FastInformedBound, Γ)
    𝒮, 𝒜, 𝒪, R, T, O, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.R, 𝒫.T, 𝒫.O, 𝒫.γ
    Γ′ = [[R(s, a) + γ*sum(maximum(sum(O(a,s′,o)*T(s,a,s′)*α′[j]
        for (j,s′) in enumerate(𝒮)) for α′ in Γ) for o in 𝒪)
        for s in 𝒮] for a in 𝒜]
    return Γ′
end

function solve(M::FastInformedBound, 𝒫::POMDP)
    Γ = [zeros(length(𝒫.𝒮)) for a in 𝒫.𝒜]
    Γ = alphavector_iteration(𝒫, M, Γ)
    return AlphaVectorPolicy(𝒫, Γ, 𝒫.𝒜)
end
```

Algorithm 21.3. The fast informed bound algorithm, which finds an approximately optimal policy for an infinite-horizon POMDP with discrete state, action, and observation spaces, where `k_max` is the number of iterations.

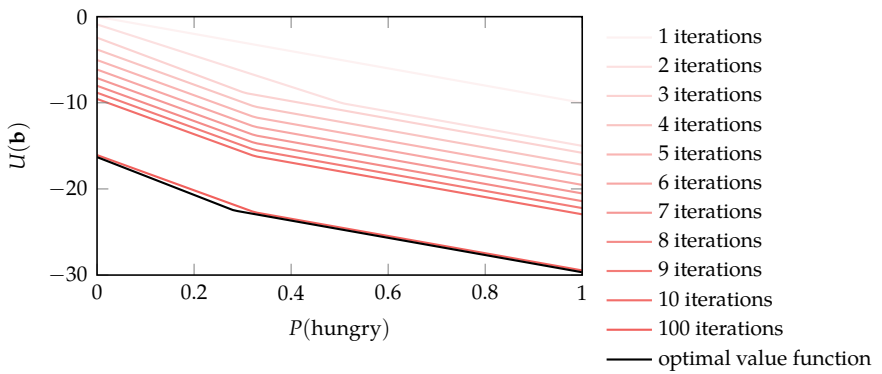


Figure 21.2. Value functions obtained for the crying baby problem using the fast informed bound. The value function after 10 iterations is noticeably lower than that of the QMDP algorithm.

21.3 *Fast Lower Bounds*

The previous two sections introduced methods that can be used to produce upper bounds on the value function represented as alpha vectors. This section introduces

a couple of methods for quickly producing lower bounds represented as alpha vectors without any planning in the belief space. Although the upper-bound methods can often be used directly to produce sensible policies, the lower bounds discussed in this section are generally only used to seed other planning algorithms. Figure 21.3 plots the two lower-bound methods discussed in this section.

A common lower bound is the *best-action worst-state* (*BAWS*) *lower bound* (algorithm 21.4). It is the discounted reward obtained by taking the best action in the worst state forever:

$$r_{\text{baws}} = \max_a \sum_{k=1}^{\infty} \gamma^{k-1} \min_s R(s,a) = \frac{1}{1-\gamma} \max_a \min_s R(s,a) \tag{21.5}$$

This lower bound is represented by a single alpha vector. This bound is typically very loose, but it can be used to seed other algorithms that can tighten the bound, as we will discuss shortly.

```
function baws_lowerbound(𝒫::POMDP)
    𝒮, 𝒜, R, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.γ
    r = maximum(minimum(R(s, a) for s in 𝒮) for a in 𝒜) / (1-γ)
    α = fill(r, length(𝒮))
    return α
end
```

The *blind lower bound* (algorithm 21.5) represents a lower bound with one alpha vector per action. It makes the assumption that we are forced to commit to a single action forever, blind to what we observe in the future. To compute these alpha vectors, we start with another lower bound (typically the best-action worst-state lower bound) and then perform a number of iterations:

$$\alpha_a^{(k+1)}(s) = R(s,a) + \gamma \sum_{s'} T(s' \mid s, a) \alpha_a^{(k)}(s') \tag{21.6}$$

This iteration is similar to the QMDP update in equation (21.1), except that it does not have a maximization over the alpha vectors on the right-hand side.

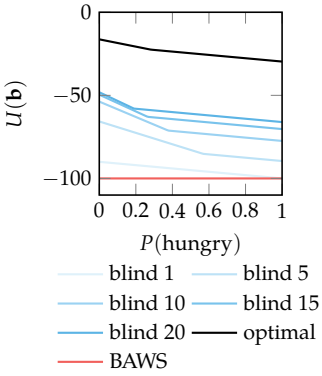


Figure 21.3. Blind lower bounds with different numbers of iterations and the BAWS lower bound applied to the crying baby problem.

Algorithm 21.4. Implementation of the best-action worst-state lower bound from equation (21.5) represented as an alpha vector.

21.4 Point-Based Value Iteration

QMDP and the fast informed bound generate one alpha vector for each action, but the optimal value function is often better approximated by many more

```
function blind_lowerbound(𝒫, k_max)
    𝒮, 𝒜, T, R, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.T, 𝒫.R, 𝒫.γ
    Q(s,a,α) = R(s,a) + γ*sum(T(s,a,s′)*α[j] for (j,s′) in enumerate(𝒮))
    Γ = [baws_lowerbound(𝒫) for a in 𝒜]
    for k in 1:k_max
        Γ = [[Q(s,a,α) for s in 𝒮] for (α,a) in zip(Γ, 𝒜)]
    end
    return Γ
end
```

Algorithm 21.5. Implementation of the blind lower bound represented as a set of alpha vectors.

alpha vectors. *Point-based value iteration*[6] computes m different alpha vectors $\Gamma = \{\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_m\}$, each associated with different belief points $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_m\}$. Methods for selecting these beliefs will be discussed in section 21.7. As before, these alpha vectors define an approximately optimal value function:

$$U^{\Gamma}(\mathbf{b}) = \max_{\boldsymbol{\alpha} \in \Gamma} \boldsymbol{\alpha}^{\top} \mathbf{b} \tag{21.7}$$

The algorithm maintains a lower bound on the optimal value function, $U^{\Gamma}(\mathbf{b}) \leq U^{*}(\mathbf{b})$ for all $\mathbf{b}$. We initialize our alpha vectors to start with a lower bound and then perform a *backup* to update the alpha vectors at each point in B. The backup operation (algorithm 21.6) takes a belief $\mathbf{b}$ and a set of alpha vectors Γ and constructs a new alpha vector. The algorithm iterates through every possible action a and observation o and extracts the alpha vector from Γ that is maximal at the resulting belief state:

$$\boldsymbol{\alpha}_{a,o} = \arg\max_{\boldsymbol{\alpha} \in \Gamma} \boldsymbol{\alpha}^{\top} \text{Update}(\mathbf{b}, a, o) \tag{21.8}$$

Then, for each available action a, we construct a new alpha vector based on these $\boldsymbol{\alpha}_{a,o}$ vectors:

$$\alpha_a(s) = R(s,a) + \gamma \sum_{s',o} O(o \mid a, s') T(s' \mid s, a) \alpha_{a,o}(s') \tag{21.9}$$

The alpha vector that is ultimately produced by the backup operation is

$$\boldsymbol{\alpha} = \arg\max_{\boldsymbol{\alpha}_a} \boldsymbol{\alpha}_a^{\top} \mathbf{b} \tag{21.10}$$

If Γ is a lower bound, the backup operation will produce only alpha vectors that are also a lower bound.

[6] A survey of point-based value iteration methods are provided by G. Shani, J. Pineau, and R. Kaplow, "A Survey of Point-Based POMDP Solvers," *Autonomous Agents and Multi-Agent Systems*, vol. 27, pp. 1–51, 2012. That reference provides a slightly different way to compute a belief backup, though the result is the same.

Repeated application of the backup operation over the beliefs in B gradually increases the lower bound on the value function represented by the alpha vectors until convergence. The converged value function will not necessarily be optimal because B typically does not include all beliefs reachable from the initial belief. However, so long as the beliefs in B are well distributed across the reachable belief space, the approximation may be acceptable. In any case, the resulting value function is guaranteed to provide a lower bound that can be used with other algorithms, potentially online, to further improve the policy.

Point-based value iteration is implemented in algorithm 21.7. Figure 21.4 shows several iterations on an example problem.

```
function backup(𝒫::POMDP, Γ, b)
    𝒮, 𝒜, 𝒪, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.γ
    R, T, O = 𝒫.R, 𝒫.T, 𝒫.O
    Γa = []
    for a in 𝒜
        Γao = []
        for o in 𝒪
            b′ = update(b, 𝒫, a, o)
            push!(Γao, argmax(α->α⋅b′, Γ))
        end
        α = [R(s, a) + γ*sum(sum(T(s, a, s′)*O(a, s′, o)*Γao[i][j]
            for (j,s′) in enumerate(𝒮)) for (i,o) in enumerate(𝒪))
            for s in 𝒮]
        push!(Γa, α)
    end
    return argmax(α->α⋅b, Γa)
end
```

Algorithm 21.6. A method for backing up a belief for a POMDP with discrete state and action spaces, where `Γ` is a vector of alpha vectors and `b` is a belief vector at which to apply the backup. The `update` method for vector beliefs is defined in algorithm 19.2.

21.5 *Randomized Point-Based Value Iteration*

Randomized point-based value iteration (algorithm 21.8) is a variation of the point-based value iteration approach from the previous section.[7] The primary difference is the fact that we are not forced to maintain an alpha vector at every belief in B. We initialize the algorithm with a single alpha vector in Γ, and then update Γ at every iteration, potentially increasing or decreasing the number of alpha vectors in Γ as appropriate. This modification of the update step can improve efficiency.

[7] 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.

```
struct PointBasedValueIteration
    B     # set of belief points
    k_max # maximum number of iterations
end

function update(𝒫::POMDP, M::PointBasedValueIteration, Γ)
    return [backup(𝒫, Γ, b) for b in M.B]
end

function solve(M::PointBasedValueIteration, 𝒫)
    Γ = fill(baws_lowerbound(𝒫), length(𝒫.𝒮))
    Γ = alphavector_iteration(𝒫, M, Γ)
    return LookaheadAlphaVectorPolicy(𝒫, Γ)
end
```

Algorithm 21.7. Point-based value iteration, which finds an approximately optimal policy for an infinite-horizon POMDP with discrete state, action, and observation spaces, where `B` is a vector of beliefs and `k_max` is the number of iterations.

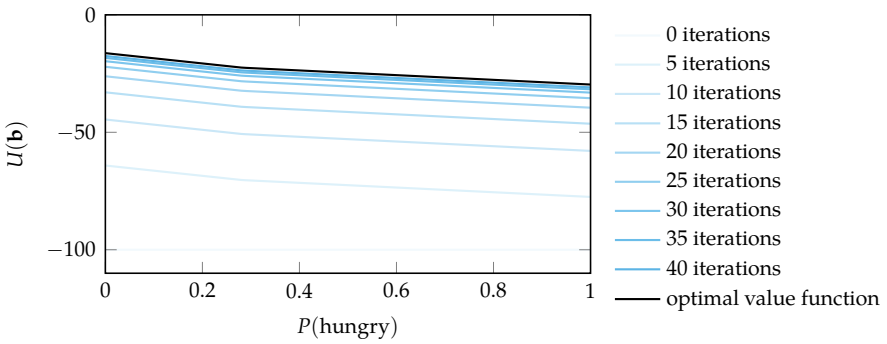


Figure 21.4. Approximate value functions obtained using point-based value iteration on the crying baby problem with belief vectors $[1/4, 3/4]$ and $[3/4, 1/4]$. Unlike QMDP and the fast informed bound, the value function of point-based value iteration is always a lower bound of the true value function.

Each update takes a set of alpha vectors Γ as input and outputs a set of alpha vectors Γ' that improve on the value function represented by Γ at the beliefs in B. In other words, it outputs 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 initializing B' to B. We then remove a point $\mathbf{b}$ randomly from B' and perform a belief backup (algorithm 21.6) on $\mathbf{b}$, using Γ to get a new alpha vector, $\boldsymbol{\alpha}$. We then find the alpha vector in $\Gamma \cup \{\boldsymbol{\alpha}\}$ that dominates at $\mathbf{b}$ and add it to Γ'. All belief points in B' whose value is improved with this alpha vector is then removed from B'. As the algorithm progresses, B' becomes smaller and contains the set of points that have not been improved by Γ'. The update finishes when B' is empty. Figure 21.5 illustrates this process with the crying baby problem.

```
struct RandomizedPointBasedValueIteration
    B     # set of belief points
    k_max # maximum number of iterations
end

function update(𝒫::POMDP, M::RandomizedPointBasedValueIteration, Γ)
    Γ′, B′ = [], copy(M.B)
    while !isempty(B′)
        b = rand(B′)
        α = argmax(α→α⋅b, Γ)
        α′ = backup(𝒫, Γ, b)
        if α′⋅b ≥ α⋅b
            push!(Γ′, α′)
        else
            push!(Γ′, α)
        end
        filter!(b→maximum(α⋅b for α in Γ′) <
            maximum(α⋅b for α in Γ), B′)
    end
    return Γ′
end

function solve(M::RandomizedPointBasedValueIteration, 𝒫)
    Γ = [baws_lowerbound(𝒫)]
    Γ = alphavector_iteration(𝒫, M, Γ)
    return LookaheadAlphaVectorPolicy(𝒫, Γ)
end
```

Algorithm 21.8. Randomized point-based value iteration, which finds an approximately optimal policy for an infinite-horizon POMDP with discrete state, action, and observation spaces, where `B` is a vector of beliefs and `k_max` is the number of iterations.

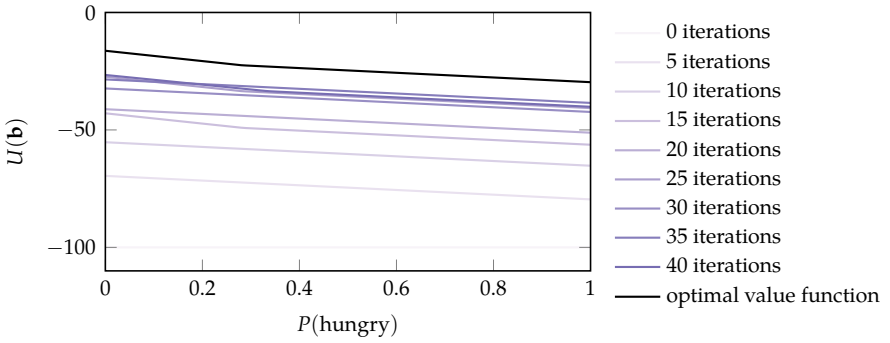


Figure 21.5. Approximate value functions obtained using randomized point-based value iteration on the crying baby problem with belief points at $[1/4, 3/4]$ and $[3/4, 1/4]$.

21.6 Sawtooth Upper Bound

The *sawtooth upper bound* is an alternative way to represent the value function. Instead of storing a set of alpha vectors Γ, we store a set of belief-utility pairs:

$$V = \{(\mathbf{b}_1, \mathbf{u}_1), \ldots, (\mathbf{b}_m, \mathbf{u}_m)\} \tag{21.11}$$

where $u_i = U(\mathbf{b}_i)$. These pairs must include all the standard basis beliefs:

$$E = \{\mathbf{e}_1 = [1, 0, \ldots, 0], \ldots, \mathbf{e}_n = [0, 0, \ldots, 1]\} \tag{21.12}$$

In other words,

$$\{(\mathbf{e}_1, U(\mathbf{e}_1)), \ldots, (\mathbf{e}_n, U(\mathbf{e}_n))\} \subseteq V \tag{21.13}$$

If these utilities are upper bounds (e.g., as obtained from the fast informed bound), then the way that we use V to estimate $U(\mathbf{b})$ at an arbitrary belief $\mathbf{b}$ will result in an upper bound.[8]

The ''sawtooth'' name comes from the way that we estimate $U(\mathbf{b})$ by interpolating points in V. For each belief-utility pair $(\mathbf{b}, U(\mathbf{b}))$ in V, we form a single, pointed ''tooth.'' When multiple pairs are considered, it forms a ''sawtooth'' shape. If the belief space is n-dimensional, each tooth is an inverted, n-dimensional pyramid. The base of each pyramid is formed by the standard basis beliefs $(\mathbf{e}_i, U(\mathbf{e}_i))$. The apex of each pyramid corresponds to one of the belief-utility pairs $(\mathbf{b}, U(\mathbf{b})) \in V$. The walls of each pyramid can be defined by hyperplanes. The combination of multiple pyramids forms the n-dimensional sawtooth. The sawtooth upper bound at any belief is the minimum value among these pyramids at that belief.

[8] The relationship between sawtooth and other bounds are discussed by M. Hauskrecht, "Value-Function Approximations for Partially Observable Markov Decision Processes," *Journal of Artificial Intelligence Research*, vol. 13, pp. 33–94, 2000.

Consider the sawtooth representation in a two-state POMDP, such as in the crying baby problem as shown in figure 21.6. The corners of each tooth are the values $U(\mathbf{e}_1)$ and $U(\mathbf{e}_2)$ for each standard basis belief $\mathbf{e}_i$. The sharp lower point of each tooth is the value $U(\mathbf{b})$, since each tooth is a point-set pair $(\mathbf{b}, U(\mathbf{b}))$. The linear interpolation from $U(\mathbf{e}_1)$ to $U(\mathbf{b})$, and again from $U(\mathbf{b})$ to $U(\mathbf{e}_2)$, form the tooth. To combine multiple teeth and form the upper bound, we take the minimum interpolated value at any belief, creating the distinctive sawtooth shape. As is apparent in figure 21.6, the sawtooth upper bound is a piecewise linear function, but it is not convex, in contrast with the alpha vector representations discussed earlier in this chapter.

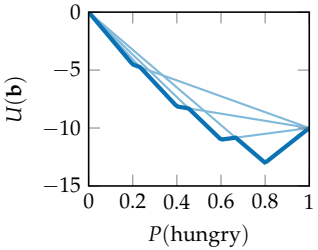


Figure 21.6. The sawtooth upper bound representation applied to the crying baby problem.

We can entend the intuition from the two-state case to allow us to compute the sawtooth upperbound $U^V(\mathbf{b})$ for any belief $\mathbf{b}$ over an arbitrary number of states from the set of pairs in V. First, we will define how to compute the upperbound for a single tooth associated with a nonbasis pair $(\mathbf{b}', u')$ in V. In order to define this upperbound $U_{(\mathbf{b}',u')}(b)$, we will use $\mathbf{u}_E = [U(\mathbf{e}_1), \ldots, U(\mathbf{e}_n)]$, which is a vector containing the utilities at all of the basis beliefs. We will also use what is called the minimum ratio:

$$\rho(\mathbf{b}, \mathbf{b}') = \min_i \frac{b_i}{b'_i} \tag{21.14}$$

It turns out that the upperbound can be written

$$U^{(\mathbf{b}',u')}(\mathbf{b}) = \mathbf{u}_E^\top \mathbf{b} + \rho(\mathbf{b}, \mathbf{b}')(\mathbf{u}' - \mathbf{u}_E^\top \mathbf{b}') \tag{21.15}$$

To compute the sawtooth upperbound with all the teeth defined by V, we take the minimum over all the teeth:

$$U^V(\mathbf{b}) = \min_{(\mathbf{b}',u') \in V | \mathbf{b}' \notin E} U^{(\mathbf{b}',u')}(\mathbf{b}) \tag{21.16}$$

Algorithm 21.9 provides an implementation. We can also derive a policy using greedy one-step lookahead.

We can iteratively apply greedy one-step lookahead at a set of beliefs B to tighten our estimates of the upper bound. The beliefs in B can be a superset of the beliefs in V. Algorithm 21.10 provides an implementation of this. Example 21.1 shows the effect of multiple iterations of the sawtooth approximation on the crying baby problem.

```
struct SawtoothPolicy
    𝒫 # POMDP problem
    V # dictionary mapping beliefs to utilities
end

function basis(𝒫)
    n = length(𝒫.𝒮)
    e(i) = [j == i ? 1.0 : 0.0 for j in 1:n]
    return [e(i) for i in 1:n]
end

function utility(π::SawtoothPolicy, b)
    𝒫, V = π.𝒫, π.V
    if haskey(V, b)
        return V[b]
    end
    uE = [V[e] for e in E]
    ρ(b′) = minimum(b[i] / b′[i] for i in eachindex(b) if b′[i] > 0)
    U(b′, u′) = uE⋅b + ρ(b′)*(u′ - uE⋅b′)
    return minimize(U(b′, u′) for (b′, u′) in V if b′ ∉ E)
end

(π::SawtoothPolicy)(b) = greedy(π, b).a
```

Algorithm 21.9. The sawtooth upper bound representation for value functions and policies. It is defined using a dictionary `V` that maps belief vectors to upper bounds on their utility obtained, such as, from the fast informed bound. A requirement of this representation is that `V` contain belief-utility pairs at the standard basis beliefs, which can be obtained from the `basis` function. We can use one-step lookahead to obtain greedy action-utility pairs from arbitrary beliefs `b`.

```
struct SawtoothIteration
    V     # initial mapping from beliefs to utilities
    B     # beliefs to compute values including those in V map
    k_max # maximum number of iterations
end

function solve(M::SawtoothIteration, 𝒫::POMDP)
    E = basis(𝒫)
    π = SawtoothPolicy(𝒫, M.V)
    for k in 1:M.k_max
        V = Dict(b ⇒ (b ∈ E ? M.V[b] : greedy(π, b).u) for b in M.B)
        π = SawtoothPolicy(𝒫, V)
    end
    return π
end
```

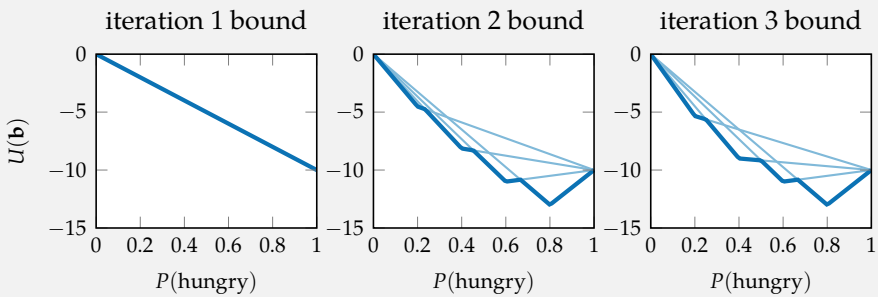
Algorithm 21.10. Sawtooth iteration iteratively applies one-step lookahead at points in `B` to improve the utility estimates at the points in `V`. The beliefs in `B` are a superset of those contained in `V`. To preserve the upper bound at each iteration, updates are not made at the standard basis beliefs stored in `E`. We run `k_max` iterations.

Example 21.1. An illustration of sawtooth's ability to maintain an upper bound at regularly spaced beliefs for the crying baby problem.

Suppose that we want to maintain an upper bound of the value for the crying baby problem with regularly spaced belief points with a step size of 0.2. To obtain an initial upper bound, we use the fast informed bound. We can then run sawtooth iteration for three steps as follows:

```
n = length(𝒫.𝒮)
πfib = solve(FastInformedBound(1), 𝒫)
V = Dict(e ⇒ utility(πfib, e) for e in basis(𝒫))
B = [[p, 1 - p] for p in 0.0:0.2:1.0]
π = solve(SawtoothIteration(V, B, 2), 𝒫)
```

The sawtooth upper bound improves as follows:



21.7 Point Selection

Algorithms like point-based value iteration and sawtooth iteration require a set of beliefs B. We want to choose B so that there are more points in the relevant areas of the belief space; we do not want to waste computation on beliefs that we are not likely to reach under our (hopefully approximately optimal) policy. One way to explore the potentially reachable space is to take steps in the belief space (algorithm 21.11). The outcome of the step will be random because the observation is generated according to our probability model.

```
function randstep(𝒫::POMDP, b, a)
    s = rand(SetCategorical(𝒫.𝒮, b))
    s′, r, o = 𝒫.TRO(s, a)
    b′ = update(b, 𝒫, a, o)
    return b′, r
end
```

Algorithm 21.11. A function for randomly sampling the next belief `b′` and reward `r`, given the current belief `b` and action `a` in problem $\mathcal{P}$.

We can create B from the belief states reachable from some initial belief under a random policy. This *random belief expansion* procedure (algorithm 21.12) may explore much more of the belief space than might be necessary; the belief space reachable by a random policy can be much larger than the space reachable by an optimal policy. Of course, computing the belief space that is reachable by an optimal policy generally requires knowing the optimal policy, which is what we want to compute in the first place. One approach that can be taken is to use successive approximations of the optimal policy to iteratively generate B.[9]

In addition to wanting our belief points to be focused on the reachable belief space, we want those points to be spread out to allow better value function approximation. The quality of the approximation provided by the alpha vectors associated with the points in B degrades as we evaluate points farther from B. We can take an *exploratory belief expansion* approach (algorithm 21.13), where we try every action for every belief in B and add the resulting belief states that are farthest from the beliefs already in the set. Distance in belief space can be measured in different ways. This algorithm uses the L_1-norm.[10] Figure 21.7 shows an example of the belief points added to B using this approach.

[9] This is the intuition behind the algorithm known as *Successive Approximations of the Reachable Space under Optimal Policies* (*SARSOP*). 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.

[10] The L_1 distance between $\mathbf{b}$ and $\mathbf{b}'$ is $\sum_s |b(s) - b'(s)|$ and is denoted as $\|\mathbf{b} - \mathbf{b}'\|_1$. See appendix A.4.

```
function random_belief_expansion(𝒫, B)
    B′ = copy(B)
    for b in B
        a = rand(𝒫.𝒜)
        b′, r = randstep(𝒫, b, a)
        push!(B′, b′)
    end
    return unique!(B′)
end
```

Algorithm 21.12. Randomly expanding a finite set of beliefs `B` used in point-based value iteration based on reachable beliefs.

```
function exploratory_belief_expansion(𝒫, B)
    B′ = copy(B)
    for b in B
        best = (b=copy(b), d=0.0)
        for a in 𝒫.𝒜
            b′, r = randstep(𝒫, b, a)
            d = minimum(norm(b - b′, 1) for b in B′)
            if d > best.d
                best = (b=b′, d=d)
            end
        end
        push!(B′, best.b)
    end
    return unique!(B′)
end
```

Algorithm 21.13. Expanding a finite set of beliefs `B` used in point-based value iteration by exploring the reachable beliefs and adding those that are farthest from the current beliefs.

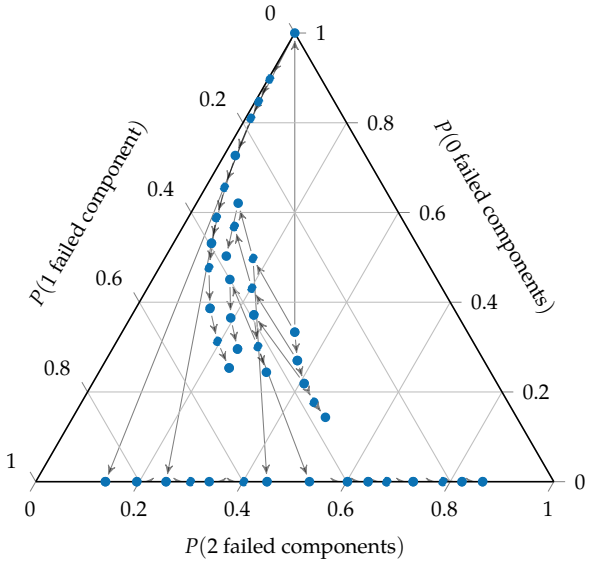


Figure 21.7. Exploratory belief expansion run on the three-state machine replacement problem, starting with an initial uniform belief $\mathbf{b} = [1/3, 1/3, 1/3]$. New beliefs were added if the distance to any previous belief was at least 0.05.

21.8 Sawtooth Heuristic Search

Chapter 9 introduced the concept of heuristic search as an online method in the fully observable context. This section discusses *sawtooth heuristic search* (algorithm 21.14) as an offline method that produces a set of alpha vectors that can be used to represent an offline policy. However, like the online POMDP methods discussed in the next chapter, the computational effort is focused on beliefs that are reachable from some specified initial belief. The heuristic that drives the exploration of the reachable belief space is the gap between the upper and lower bounds of the value function.[11]

The algorithm is initialized with an upper bound on the value function represented by a set of sawtooth belief-utility pairs V, together with a lower bound on the value function represented by a set of alpha vectors Γ. The belief-utility pairs defining the sawtooth upper bound can be obtained from the fast informed bound. The lower bound can be obtained from the best-action worst-state bound, as shown in algorithm 21.14, or some other method, such as point-based value iteration.

[11] The *heuristic search value iteration* (*HSVI*) algorithm introduced the concept of using the sawtooth-based action heuristic and gap-based observation heuristic. T. Smith and R. G. Simmons, "Heuristic Search Value Iteration for POMDPs," in *Conference on Uncertainty in Artificial Intelligence* (*UAI*), 2004. The SARSOP algorithm built on this work. 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.

```
struct SawtoothHeuristicSearch
    b     # initial belief
    δ     # gap threshold
    d     # depth
    k_max # maximum number of iterations
    k_fib # number of iterations for fast informed bound
end

function explore!(M::SawtoothHeuristicSearch, 𝒫, πhi, πlo, b, d=0)
    𝒮, 𝒜, 𝒪, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.γ
    ϵ(b′) = utility(πhi, b′) - utility(πlo, b′)
    if d ≥ M.d || ϵ(b) ≤ M.δ / γ^d
        return
    end
    a = πhi(b)
    o = argmax(o -> ϵ(update(b, 𝒫, a, o)), 𝒪)
    b′ = update(b, 𝒫, a, o)
    explore!(M, 𝒫, πhi, πlo, b′, d+1)
    if b′ ∉ basis(𝒫)
        πhi.V[b′] = greedy(πhi, b′).u
    end
    push!(πlo.Γ, backup(𝒫, πlo.Γ, b′))
end

function solve(M::SawtoothHeuristicSearch, 𝒫::POMDP)
    πfib = solve(FastInformedBound(M.k_fib), 𝒫)
    Vhi = Dict(e => utility(πfib, e) for e in basis(𝒫))
    πhi = SawtoothPolicy(𝒫, Vhi)
    πlo = LookaheadAlphaVectorPolicy(𝒫, [baws_lowerbound(𝒫)])
    for i in 1:M.k_max
        explore!(M, 𝒫, πhi, πlo, M.b)
        if utility(πhi, M.b) - utility(πlo, M.b) < M.δ
            break
        end
    end
    return πlo
end
```

Algorithm 21.14. The sawtooth heuristic search policy. The solver starts from belief `b` and explores to a depth `d` for no more than `k_max` iterations. It uses an upper bound obtained through the fast informed bound computed with `k_fib` iterations. The lower bound is obtained from the best-action worst-state bound. The gap threshold is `δ`.

At each iteration, we explore beliefs that are reachable from our initial belief to a maximum depth. As we explore, we update the set of belief-action pairs forming our sawtooth upper bound and the set of alpha vectors forming our lower bound. We stop exploring after a certain number of iterations or until the gap at our initial state is below a threshold $\delta > 0$.

When we encounter a belief b along our path from the initial node during our exploration, we check whether the gap at b is below a threshold δ / γ^d, where d is our current depth. If we are below that threshold, then we can stop exploring along that branch. We want the threshold to increase as d increases because the gap at b after an update is at most γ times the weighted average of the gap at the beliefs that are immediately reachable.

If the gap at b is above the threshold and we have not reached our maximum depth, then we can explore the next belief, b'. First, we determine the action a recommended by our sawtooth policy. Then, we choose the observation o that maximizes the gap at the resulting belief.[12] We recursively explore down the tree. After exploring the descendants of b', we add (b', u) to V, where u is the one-step lookahead value of b'. We add to Γ the alpha vector that results from a backup at b'. Figure 21.8 shows the tightening of the bounds.

[12] Some variants simply sample the next observations. Others select the observation that maximizes the gap weighted by its likelihood.

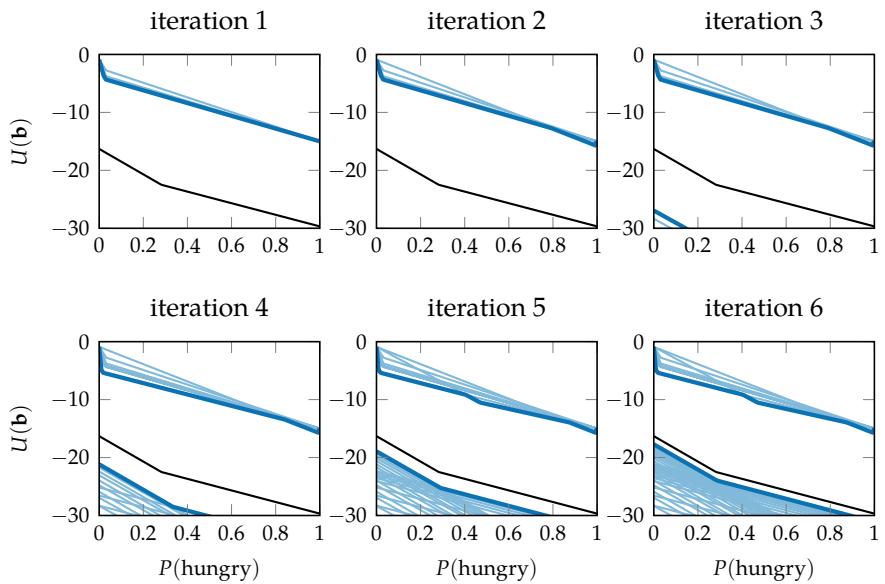


Figure 21.8. The evolution of the upper bound, represented by sawtooth pairs, and the lower bound, represented by alpha vectors for the crying baby problem. The optimal value function is shown in black.

21.9 Triangulated Value Functions

As discussed in section 20.1, a POMDP can be converted to a belief-state MDP. The state space in that belief-state MDP is continuous, corresponding to the space of possible beliefs in the original POMDP. We can approximate the value function in a way similar to what was described in chapter 8 and then apply a dynamic programming algorithm such as value iteration to the approximation. This section discusses a particular kind of local value function approximation that involves *Freudenthal triangulation*[13] over a discrete set of belief points B. This triangulation allows us to interpolate the value function at arbitrary points in the belief space. As with the sawtooth representation, we use a set of belief-utility pairs $V = \{(b, U(b)) \mid b \in B\}$ to represent our value function. This approach can be used to obtain an upper bound on the value function.

Freudenthal interpolation in belief space involves spreading the belief points in B evenly over the space, as shown in figure 21.9. The number of beliefs in B depends on the dimensionality n and granularity m of the Freudenthal triangulation:[14]

$$|B| = \frac{(m+n-1)!}{m!(n-1)!} \tag{21.17}$$

We can estimate $U(\mathbf{b})$ at an arbitrary point b by interpolating values at the discrete points in B. Similar to the simplex interpolation introduced in section 8.5, we find the set of belief points in B that form a simplex that encloses b and weight their values together. In n-dimensional belief spaces, there are up to $n+1$ vertices whose values need to be weighted together. If $b^{(1)}, \ldots, b^{(n+1)}$ are the enclosing points and $\lambda_1, \ldots, \lambda_{n+1}$ are their weights, then the estimate of the value at b is

$$U(b) = \sum_i \lambda_i U(b^{(i)}) \tag{21.18}$$

Algorithm 21.15 extracts this utility function and policy from the pairs in V.

Algorithm 21.16 applies a variation of approximate value iteration (introduced in algorithm 8.1) to our triangulated policy representation. We simply iteratively apply backups over our beliefs in B using one-step lookahead with our value function interpolation. If U is initialized with an upper bound, value iteration will result in an upper bound even after a finite number of iterations. This property holds because value functions are convex and the linear interpolation between vertices on the value function must lie on or above the underlying convex function.[15] Figure 21.10 shows an example of a policy and utility function.

[13] H. Freudenthal, "Simplizialzerlegungen von Beschränkter Flachheit," *Annals of Mathematics*, vol. 43, pp. 580–582, 1942. This triangulation method was applied to POMDPs in W. S. Lovejoy, "Computationally Feasible Bounds for Partially Observed Markov Decision Processes," *Operations Research*, vol. 39, no. 1, pp. 162–175, 1991.

[14] `FreudenthalTriangulations.jl` provides an implementation for generating these beliefs.

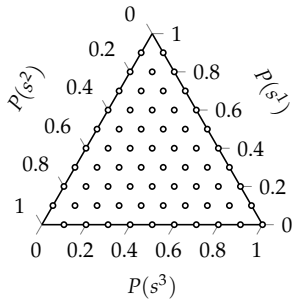


Figure 21.9. A belief state discretization using Freudenthal triangulation in $n = 3$-dimensional belief space with granularity $m = 10$.

[15] See lemma 4 of W. S. Lovejoy, "Computationally Feasible Bounds for Partially Observed Markov Decision Processes," *Operations Research*, vol. 39, no. 1, pp. 162–175, 1991.

```
struct TriangulatedPolicy
    𝒫 # POMDP problem
    V # dictionary mapping beliefs to utilities
    B # beliefs
    T # Freudenthal triangulation
end

function TriangulatedPolicy(𝒫::POMDP, m)
    T = FreudenthalTriangulation(length(𝒫.𝒮), m)
    B = belief_vertices(T)
    V = Dict(b ⇒ 0.0 for b in B)
    return TriangulatedPolicy(𝒫, V, B, T)
end

function utility(π::TriangulatedPolicy, b)
    B, λ = belief_simplex(π.T, b)
    return sum(λi*π.V[b] for (λi, b) in zip(λ, B))
end

(π::TriangulatedPolicy)(b) = greedy(π, b).a
```

Algorithm 21.15. A policy representation using Freudenthal triangulation with granularity `m`. As with the sawtooth method, we maintain a dictionary that maps belief vectors to utilities. This implementation initializes the utilities to 0, but if we want to represent an upper bound, then we would need to initialize those utilities appropriately. We define a function to estimate the utility of a given belief using interpolation. We can extract a policy using greedy lookahead. The Freudenthal triangulation structure is passed the dimensionality and granularity at construction. The `FreudenthalTriangulations.jl` package provides the function `belief_vertices`, which returns B, given a particular triangulation. It also provides `belief_simplex`, which returns the set of enclosing points and weights for a belief.

```
struct TriangulatedIteration
    m     # granularity
    k_max # maximum number of iterations
end

function solve(M::TriangulatedIteration, 𝒫)
    π = TriangulatedPolicy(𝒫, M.m)
    U(b) = utility(π, b)
    for k in 1:M.k_max
        U′ = [greedy(𝒫, U, b).u for b in π.B]
        for (b, u′) in zip(π.B, U′)
            π.V[b] = u′
        end
    end
    return π
end
```

Algorithm 21.16. Approximate value iteration with `k_max` iterations using a triangulated policy with granularity `m`. At each iteration, we update the utilities associated with the beliefs in `B` using greedy one-step lookahead with triangulated utilities.

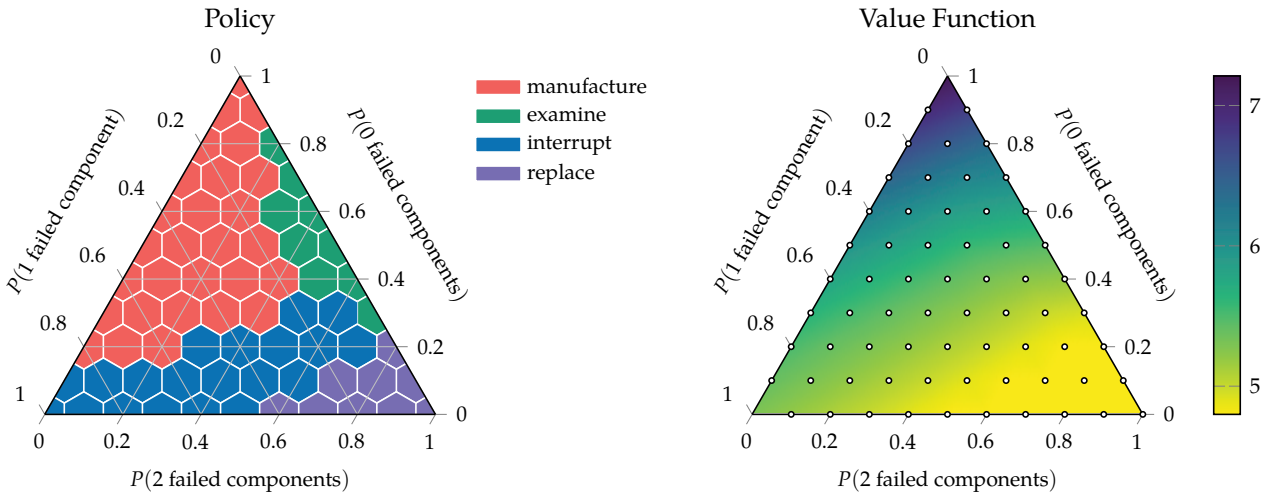


Figure 21.10. The policy and value function for the maintenance problem with granularity $m = 10$ after 11 iterations. The value function plot shows the discrete belief points as white dots. This policy approximates the exact policy given in appendix F.8.

21.10 Summary

- The QMDP algorithm assumes perfect observability after the first step, resulting in an upper bound on the true value function.
- The fast informed bound provides a tighter upper bound on the value function than QMDP by accounting for the observation model.
- Point-based value iteration provides a lower bound on the value function using alpha vectors at a finite set of beliefs.
- Randomized point-based value iteration performs updates at randomly selected points in the belief set until the values at all points in the set are improved.
- The sawtooth upper bound allows iterative improvement of the fast informed bound using an efficient point-set representation.
- Carefully selecting which belief points to use in point-based value iteration can improve the quality of the resulting policies.
- Sawtooth heuristic search attempts to tighten the upper and lower bounds of the value function represented by sawtooth pairs and alpha vectors, respectively.

- One approach to approximately solving POMDPs is to discretize the belief space, and then to apply dynamic programming to extract an upper bound on the value function and a policy.

21.11 Exercises

Exercise 21.1. Suppose that we are in a variation of the straight-line hex world problem (appendix F.1) consisting of four cells corresponding to states $s_{1:4}$. There are two actions: move left (ℓ) and move right (r). The effects of those actions are deterministic. Moving left in s_1 or moving right in s_4 gives a reward of 100 and ends the game. With a discount factor of 0.9, compute alpha vectors using QMDP. Then, using the alpha vectors, compute the approximately optimal action, given the belief $\mathbf{b} = [0.3, 0.1, 0.5, 0.1]$.

Solution: We denote the alpha vector associated with moving left as α_ℓ and the alpha vector associated with moving right as α_r. We initialize the alpha vectors to zero:

$$\boldsymbol{\alpha}_\ell^{(1)} = [R(s_1, \ell), R(s_2, \ell), R(s_3, \ell), R(s_4, \ell)] = [0, 0, 0, 0]$$
$$\boldsymbol{\alpha}_r^{(1)} = [R(s_1, r), R(s_2, r), R(s_3, r), R(s_4, r)] = [0, 0, 0, 0]$$

In the first iteration, since all the entries in the alpha vectors are zero, only the reward term contributes to the QMDP update (equation (21.1)):

$$\boldsymbol{\alpha}_\ell^{(2)} = [100, 0, 0, 0]$$
$$\boldsymbol{\alpha}_r^{(2)} = [0, 0, 0, 100]$$

In the next iteration, we apply the update, which leads to new values for s_2 for the left alpha vector and for s_3 for the right alpha vector. The updates for the left alpha vector are as follows (with the right alpha vector updates being symmetric):

$$\alpha_\ell^{(3)}(s_1) = 100 \quad \text{(terminal state)}$$
$$\alpha_\ell^{(3)}(s_2) = 0 + 0.9 \times \max(\alpha_\ell^{(2)}(s_1), \alpha_r^{(2)}(s_1)) = 90$$
$$\alpha_\ell^{(3)}(s_3) = 0 + 0.9 \times \max(\alpha_\ell^{(2)}(s_2), \alpha_r^{(2)}(s_2)) = 0$$
$$\alpha_\ell^{(3)}(s_4) = 0 + 0.9 \times \max(\alpha_\ell^{(2)}(s_3), \alpha_r^{(2)}(s_3)) = 0$$

This leads to the following:

$$\boldsymbol{\alpha}_\ell^{(3)} = [100, 90, 0, 0]$$
$$\boldsymbol{\alpha}_r^{(3)} = [0, 0, 90, 100]$$

In the third iteration, the updates for the left alpha vector are

$$\begin{aligned}
\alpha_\ell^{(4)}(s_1) &= 100 \quad \text{(terminal state)} \\
\alpha_\ell^{(4)}(s_2) &= 0 + 0.9 \times \max(\alpha_\ell^{(3)}(s_1), \alpha_r^{(3)}(s_1)) = 90 \\
\alpha_\ell^{(4)}(s_3) &= 0 + 0.9 \times \max(\alpha_\ell^{(3)}(s_2), \alpha_r^{(3)}(s_2)) = 81 \\
\alpha_\ell^{(4)}(s_4) &= 0 + 0.9 \times \max(\alpha_\ell^{(3)}(s_3), \alpha_r^{(3)}(s_3)) = 81
\end{aligned}$$

Our alpha vectors are, then,

$$\begin{aligned}
\boldsymbol{\alpha}_\ell^{(4)} &= [100, 90, 81, 81] \\
\boldsymbol{\alpha}_r^{(4)} &= [81, 81, 90, 100]
\end{aligned}$$

At this point, our alpha vector estimates have converged. We now determine the optimal action by maximizing the utility associated with our belief over all actions:

$$\begin{aligned}
\boldsymbol{\alpha}_\ell^\top \mathbf{b} &= 100 \times 0.3 + 90 \times 0.1 + 81 \times 0.5 + 81 \times 0.1 = 87.6 \\
\boldsymbol{\alpha}_r^\top \mathbf{b} &= 81 \times 0.3 + 81 \times 0.1 + 90 \times 0.5 + 100 \times 0.1 = 87.4
\end{aligned}$$

Thus, we find that moving left is the optimal action for this belief state, despite a higher probability of being on the right half of the grid world. This is due to the relatively high likelihood that we assign to being in state s_1, where we would receive a large, immediate reward by moving left.

Exercise 21.2. Recall the simplified hex world problem from exercise 21.1. Compute alpha vectors for each action using the blind lower bound. Then, using the alpha vectors, compute the value at the belief $\mathbf{b} = [0.3, 0.1, 0.5, 0.1]$.

Solution: The blind lower bound, shown in equation (21.6), is like the QMDP update, but it lacks the maximization. We initialize the components of the alpha vectors to zero and run to convergence as follows:

$$\begin{aligned}
\boldsymbol{\alpha}_\ell^{(2)} &= [100, 0, 0, 0] \\
\boldsymbol{\alpha}_r^{(2)} &= [0, 0, 0, 100]
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{\alpha}_\ell^{(3)} &= [100, 90, 0, 0] \\
\boldsymbol{\alpha}_r^{(3)} &= [0, 0, 90, 100]
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{\alpha}_\ell^{(4)} &= [100, 90, 81, 0] \\
\boldsymbol{\alpha}_r^{(4)} &= [0, 81, 90, 100]
\end{aligned}$$

$$\boldsymbol{\alpha}_\ell^{(5)} = [100, 90, 81, 72.9]$$
$$\boldsymbol{\alpha}_r^{(5)} = [72.9, 81, 90, 100]$$

At this point, our alpha vector estimates have converged. We now determine the value by maximizing the utility associated with our belief over all actions:

$$\boldsymbol{\alpha}_\ell^\top \mathbf{b} = 100 \times 0.3 + 90 \times 0.1 + 81 \times 0.5 + 72.9 \times 0.1 = 86.79$$
$$\boldsymbol{\alpha}_r^\top \mathbf{b} = 72.9 \times 0.3 + 81 \times 0.1 + 90 \times 0.5 + 100 \times 0.1 = 84.97$$

Thus, the lower bound at $\mathbf{b}$ is 86.79.

Exercise 21.3. What is the complexity of a backup at a single belief point in point-based value iteration assuming that $|\Gamma| > |S|$?

Solution: In the process of doing a backup, we compute an $\boldsymbol{\alpha}_{a,o}$ for every action a and observation o. Computing $\boldsymbol{\alpha}_{a,o}$ in equation (21.8) requires finding the alpha vector $\boldsymbol{\alpha}$ in Γ that maximizes $\boldsymbol{\alpha}^\top \text{Update}(\mathbf{b}, a, o)$. A belief update, as shown in equation (19.7), is $O(|\mathcal{S}|^2)$ because it iterates over all initial and successor states. Hence, computing $\boldsymbol{\alpha}_{a,o}$ requires $O(|\Gamma||S| + |S|^2) = O(|\Gamma||S|)$ operations for a specific a and o, resulting in a total of $O(|\Gamma||S||A||O|)$ operations. We then compute $\boldsymbol{\alpha}_a$ in equation (21.9) for every action a using these values for $\boldsymbol{\alpha}_{a,o}$, requiring in a total of $O(|S|^2|A||O|)$. Finding the alpha vector $\boldsymbol{\alpha}_a$ that maximizes $\boldsymbol{\alpha}_a^\top \mathbf{b}$ requires $O(|S||A|)$ operations once we have the $\boldsymbol{\alpha}_a$ values. Together, we have $O(|\Gamma||S||A||O|)$ operations for a backup at belief $\mathbf{b}$.

Exercise 21.4. Consider the set of belief-utility pairs given by

$$V = \{([1, 0], 0), ([0, 1], -10), ([0.8, 0.2], -4), ([0.4, 0.6], -6)\}$$

Using weights $w_i = 0.5$ for all i, determine the utility for belief $\mathbf{b} = [0.5, 0.5]$ using the sawtooth upper bound.

Solution: We interpolate with the belief-utility pairs. For each nonbasis belief, we start by finding the farthest basis belief, $\mathbf{e}_i$. Starting with $\mathbf{b}_3$, we compute as follows:

$$\begin{aligned}
i_3 &= \arg\max_j \left\|\mathbf{b} - \mathbf{e}_j\right\|_1 - \left\|\mathbf{b}_3 - \mathbf{e}_j\right\|_1 \\
\|\mathbf{b} - \mathbf{e}_1\|_1 - \|\mathbf{b}_3 - \mathbf{e}_1\|_1 &= \left\|\begin{bmatrix}0.5\\0.5\end{bmatrix} - \begin{bmatrix}1\\0\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}0.8\\0.2\end{bmatrix} - \begin{bmatrix}1\\0\end{bmatrix}\right\|_1 \\
&= \left\|\begin{bmatrix}-0.5\\0.5\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}-0.2\\0.2\end{bmatrix}\right\|_1 \\
&= 0.6 \\
\|\mathbf{b} - \mathbf{e}_2\|_1 - \|\mathbf{b}_3 - \mathbf{e}_2\|_1 &= \left\|\begin{bmatrix}0.5\\0.5\end{bmatrix} - \begin{bmatrix}0\\1\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}0.8\\0.2\end{bmatrix} - \begin{bmatrix}0\\1\end{bmatrix}\right\|_1 \\
&= \left\|\begin{bmatrix}0.5\\-0.5\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}0.8\\-0.8\end{bmatrix}\right\|_1 \\
&= -0.6 \\
i_3 &= 1
\end{aligned}$$

Thus, $\mathbf{e}_1$ is the farthest basis belief from $\mathbf{b}_3$.

For $\mathbf{b}_4$, we compute the following:

$$\begin{aligned}
i_4 &= \arg\max_j \left\|\mathbf{b} - \mathbf{e}_j\right\|_1 - \left\|\mathbf{b}_4 - \mathbf{e}_j\right\|_1 \\
\|\mathbf{b} - \mathbf{e}_1\|_1 - \|\mathbf{b}_3 - \mathbf{e}_1\|_1 &= \left\|\begin{bmatrix}0.5\\0.5\end{bmatrix} - \begin{bmatrix}1\\0\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}0.4\\0.6\end{bmatrix} - \begin{bmatrix}1\\0\end{bmatrix}\right\|_1 \\
&= \left\|\begin{bmatrix}-0.5\\0.5\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}-0.6\\0.6\end{bmatrix}\right\|_1 \\
&= -0.2 \\
\|\mathbf{b} - \mathbf{e}_2\|_1 - \|\mathbf{b}_3 - \mathbf{e}_2\|_1 &= \left\|\begin{bmatrix}0.5\\0.5\end{bmatrix} - \begin{bmatrix}0\\1\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}0.4\\0.6\end{bmatrix} - \begin{bmatrix}0\\1\end{bmatrix}\right\|_1 \\
&= \left\|\begin{bmatrix}0.5\\-0.5\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}0.4\\-0.4\end{bmatrix}\right\|_1 \\
&= 0.2 \\
i_4 &= 2
\end{aligned}$$

Thus, $\mathbf{e}_2$ is the farthest basis belief from $\mathbf{b}_4$.

We can compute $U(\mathbf{b})$ using our weights, along with the appropriate corners and utility pairs $(\mathbf{e}_2, \mathbf{b}_3)$ and $(\mathbf{e}_1, \mathbf{b}_4)$:

$$U_3(\mathbf{b}) = 0.5 \times -4 + 0.5 \times (-10) = -7$$
$$U_4(\mathbf{b}) = 0.5 \times -6 + 0.5 \times 0 = -3$$

Finally, we compute $U(\mathbf{b})$ by taking the minimum of $U_3(\mathbf{b})$ and $U_4(\mathbf{b})$. Thus, $U(\mathbf{b}) = -7$.

Exercise 21.5. Suppose that we have a valid lower bound represented as a set of alpha vectors Γ. Is it possible for a backup at a belief state b to result in an alpha vector $\boldsymbol{\alpha}'$, such that $\boldsymbol{\alpha}'^\top \mathbf{b}$ is lower than the utility function represented by Γ? In other words, can a backup at a belief b result in an alpha vector that assigns a lower utility to b than the value function represented by Γ?

Solution: It is possible. Suppose we have only one action, observations are perfect, there is no discounting, and the state space is $\{s^0, s^1\}$. The reward is $R(s^i) = i$ for all i, and states transition deterministically to s^0. We start with a valid lower bound, $\Gamma = \{[-1, +1]\}$, as shown in red in figure 21.11. We choose $\mathbf{b} = [0.5, 0.5]$ for the belief where we do the backup. Using equation (21.9), we obtain

$$\alpha(s^0) = R(s^0) + U^\Gamma(s^0) = 0 + (-1) = -1$$
$$\alpha(s^1) = R(s^1) + U^\Gamma(s^0) = 1 + (-1) = 0$$

Hence, the alpha vector that we get after a backup is $[-1, 0]$, shown in blue in figure 21.11. The utility at $\mathbf{b}$ with that alpha vector is -0.5. However, $U^\Gamma(\mathbf{b}) = 0$, showing that backing up a belief can result in an alpha vector that represents a lower utility at that belief. This fact motivates the use of the if statement in randomized point-based value iteration (algorithm 21.8). That if statement will use either the alpha vector from the backup or the dominating alpha vector in Γ at belief $\mathbf{b}$, whichever gives the greatest utility estimate.

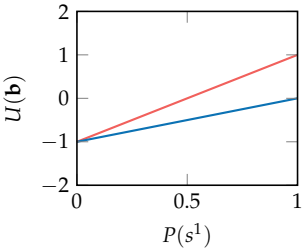


Figure 21.11. An example of how a backup at a belief can result in an alpha vector that, on its own, lowers the value at that belief compared to the original value function. The belief $\mathbf{b}$ where we do the update corresponds to $P(s^1) = 0.5$. The original value function, represented by Γ, is shown in red. The alpha vector resulting from a backup at $\mathbf{b}$ is shown in blue.