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 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 different 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.

[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.

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 and 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.

[3] M. Hauskrecht, "Value-Function Approximations for Partially Observable Markov Decision Processes," *Journal of Artificial Intelligence Research*, vol. 13, pp. 33–94, 2000. This reference contains a proof that QMDP provides an upper bound on the optimal value function.

```
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`.

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 (α,a) in zip(Γ, 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.

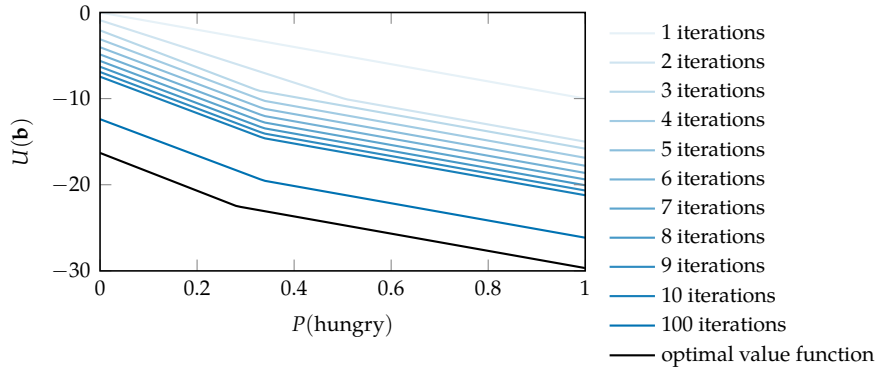


Figure 21.1. Value functions obtained for the crying baby problem using QMDP. In the first iteration, a single alpha vector dominates. In subsequent iterations, two alpha vectors dominate.

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 only do

better 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 our 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

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

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 is 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 tends to be tighter. The fast informed bound is implemented in algorithm 21.3 and is used to compute optimal value functions in figure 21.2.

```
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.

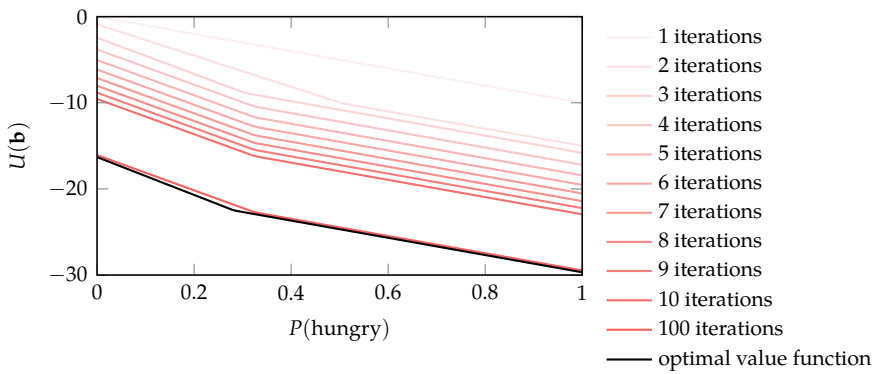


Figure 21.2. Value functions obtained for the crying baby problem using the fast informed bound. The value function after 10 iterations is noticeably lower than that of the QMDP algorithm.

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 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 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) also 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 some other lower bound, typically the best-action worst-state lower bound, and then perform some 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.

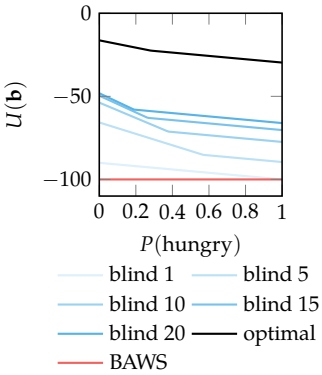


Figure 21.3. Blind lower bounds with different numbers of iterations and the best-action worst-state lower bound applied to the crying baby problem.

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

21.4 Point-Based Value Iteration

QMDP and the fast informed bound generates 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:

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

[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*, pp. 1–51, 2012.

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 only produce alpha vectors that are still a lower bound.

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 in 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 it 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.

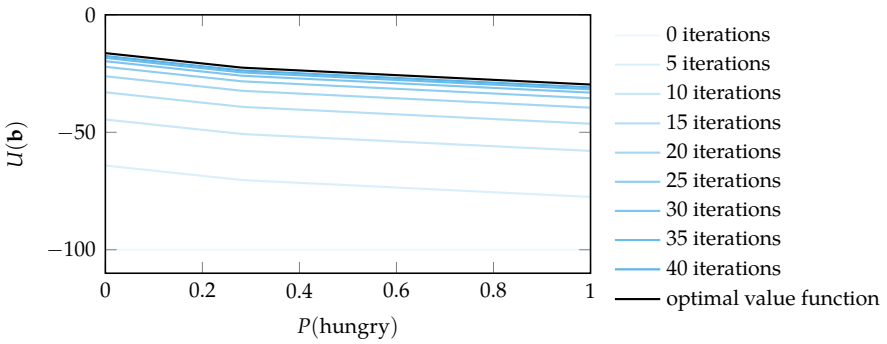


Figure 21.4. Approximate value functions obtained using point-based value iteration on the crying baby problem with belief vectors $[1/4, 3/4]$ and $[3/4, 1/4]$. Unlike QMDP and the fast informed bound, point-based value iteration's value function is always a lower bound of the true value function.

Each update takes a set of alpha vectors Γ as input and outputs a set of alpha vectors Γ' that improves on the value function represented by Γ at the beliefs in B. In other words, it outputs a 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 b randomly from B' and perform a belief backup (algorithm 21.6) on 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'. 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 on 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 backup, which updates the alpha vectors `Γ` associated with beliefs `B` for the $\mathcal{P}$. This backup can be used in place of `point_based_update` in algorithm 21.7.

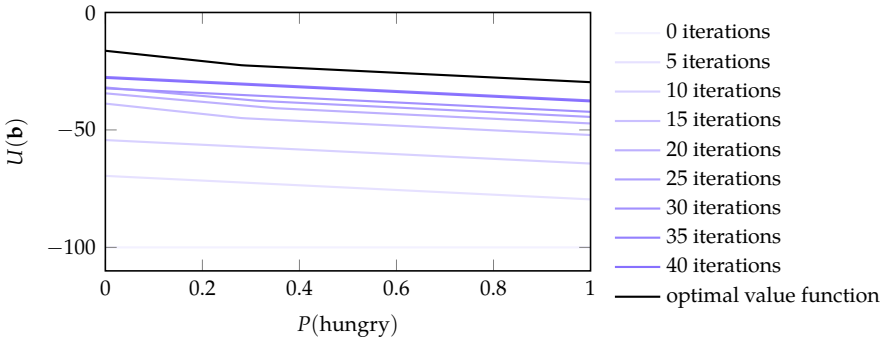


Figure 21.5. Approximate value functions obtained using randomized point-based value iteration on the crying baby problem with belief points at $[1/4, 3/4]$ and $[3/4, 1/4]$.

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:

$$V = \{(b_1, U(b_1)), \ldots, (b_m, U(b_m))\} \tag{21.11}$$

with the requirement that V contains all of the standard basis beliefs:

$$E = \{e_1 = [1, 0, \ldots, 0], \ldots, e_n = [0, 0, \ldots, 1]\} \tag{21.12}$$

such that

$$\{(e_1, U(e_1)), \ldots, (e_n, U(e_n))\} \subseteq V \tag{21.13}$$

If these utilities are upper bounds (for example, as obtained from the fast informed bound), then the way we use V to estimate $U(b)$ at arbitrary beliefs b will result in an upper bound.[8]

The ''sawtooth'' name comes from the way we estimate $U(b)$ by interpolating points in V. For each belief-utility pair $(b, U(b))$ in V, we form a single pointed ''tooth.'' If the belief space is n-dimensional, each tooth is an inverted n-dimensional pyramid. When multiple pairs are considered, it forms a ''sawtooth'' shape. The bases of the pyramids are formed by the standard basis beliefs $(e_i, U(e_i))$. The apex point of each tooth corresponds to each belief-utility pair $(b, U(b)) \in V$. Since these are pyramids in general, each tooth has walls equivalently defined by n-hyperplanes with bounded regions. These hyperplanes can also be interpreted as alpha vectors that act over a bounded region of the 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.

space, not the entire belief space as in normal alpha vectors. The combination of multiple pyramids forms the n-dimensional sawtooth. The sawtooth upper bound at any belief is similarly the minimum value among these pyramids at that belief.

Consider the sawtooth representation in a two-state POMDP, such as in the crying baby problem shown in figure 21.6. The corners of each tooth are the values $U(e_1)$ and $U(e_2)$ for each standard basis belief e_i. The sharp lower point of each tooth is the value $U(b)$, since each tooth is a point-set pair $(b, U(b))$. The linear interpolation from $U(e_1)$ to $U(b)$ and again from $U(b)$ to $U(e_2)$ forms the tooth. To combine multiple teeth and form the upper bound, we take the minimum interpolated value at any belief, forming the distinctive sawtooth shape.

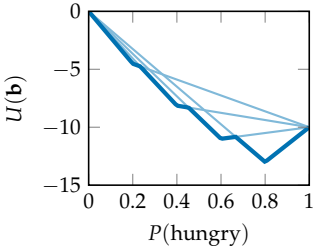


Figure 21.6. The sawtooth upper bound representation applied to the crying baby problem.

To compute the sawtooth at any belief b, we iterate over each belief-utility pair $(b', U(b'))$ in V. The key idea is to compute the utility $U'(b)$ for this hyperpyramid, first by finding the farthest basis point, then using this to determine the matching hyperplane from the hyperpryamid, and finally computing a utility using a rescaled version of the hyperplane. The farthest basis belief e_i is computed using L_1 distances from b and b'.

$$i \leftarrow \arg\max_j \|b - e_j\|_1 - \|b' - e_j\|_1 \tag{21.14}$$

This e_i uniquely identifies the particular hyperplane among those forming the hyperpyramid for $U(b')$. Specifically, this hyperplane is defined by all corners $e_j \neq e_i$ and using b' as a replacement for e_i. At this point, we know this is the hyperplane for the region of the utility that b is contained within. The hyperplane's utilities are $U(e_j)$ for $e_j \neq e_i$ and $U(b')$ as a replacement for $U(e_i)$. However, we cannot directly compute the desired utility $U'(b)$ using a dot product because this is not the standard simplex. We instead compute the weight w of b in terms of the weighted distance from the hyperplane's corners $e_j \neq e_i$ and b'. This allows us to be able to compute $U'(b)$, essentially creating a simplex amenable to a dot product with $U(e_j)$ and $U(b')$:

$$U'(b) = w_i U(b') + \sum_{j \neq i} w_j U(e_j) \tag{21.15}$$

This entire process is done among all $(b', U(b'))$, resulting in

$$U(b) = \min_{(b', U(b')) \in V} U'(b) \tag{21.16}$$

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

```
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
    n = length(𝒫.𝒮)
    E = basis(𝒫)
    u = sum(V[E[i]] * b[i] for i in 1:n)
    for (b′, u′) in V
        if b′ ∉ E
            i = argmax([norm(b-e, 1) - norm(b′-e, 1) for e in E])
            w = [norm(b - e, 1) for e in E]
            w[i] = norm(b - b′, 1)
            w /= sum(w)
            w = [1 - wi for wi in w]
            α = [V[e] for e in E]
            α[i] = u′
            u = min(u, w⋅α)
        end
    end
    return u
end

function (π::SawtoothPolicy)(b)
    U(b) = utility(π, b)
    return greedy(π.𝒫, U, b)
end
```

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, for example, 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`.

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. Example 21.1 shows the effect of multiple iterations of the sawtooth approximation on the crying baby problem.

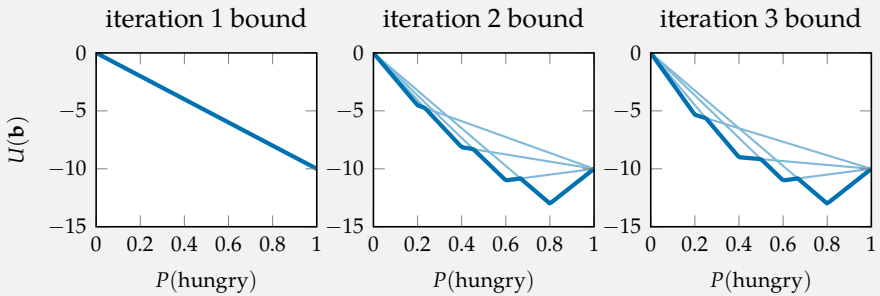
```
struct SawtoothIteration
    V     # initial mapping from beliefs to utilities
    B     # beliefs to compute values including those in V map
    k_max # maximum number of iterations
end

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] : 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.

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

```
n = length(𝒫.𝒮)
πfib = solve(FastInformedBound(1), 𝒫)
V = Dict(e ⇒ πfib(e).u 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:



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

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 (algorithm 21.11) in the belief space. 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 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 also want those points to be spread out to allow for 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 further 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 furthest away 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 $\|\mathbf{b} - \mathbf{b}'\|_1$. 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. An algorithm for 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. An algorithm for expanding a finite set of beliefs `B` used in point-based value iteration by exploring the reachable beliefs and adding those that are furthest from the current beliefs.

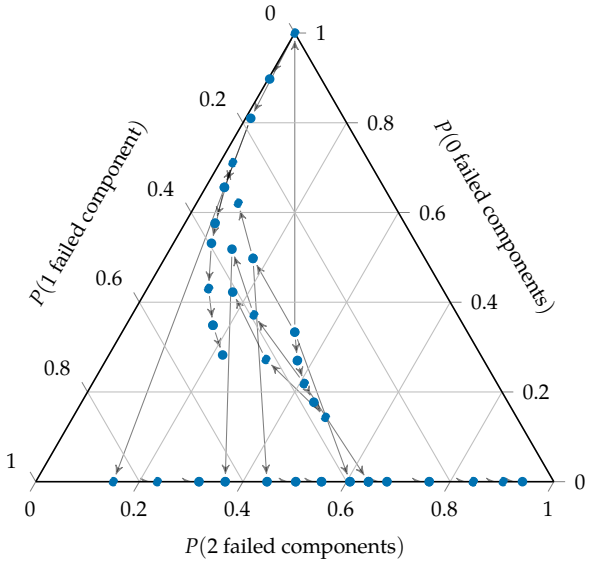


Figure 21.7. Exploratory belief expansion run on the three-state machine replacement problem, starting with an initial uniform belief $\mathbf{b} = [1/3, 1/3, 1/3]$. New beliefs were added if the distance to any previous belief was at least 0.05.

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 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 upon 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.

At each iteration, we explore beliefs that are reachable from our initial belief to some maximum depth. As we do these explorations, 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. As the depth d increases, so does the threshold—in other words, the deeper we are the more tolerant we are of the gap. The reason for this is that 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', which is selected as follows. 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 also add to Γ the alpha vector that results from a backup at b'. Figure 21.8 shows how the upper and lower bounds become tighter with each iteration.

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

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 discussed 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.

The Freudenthal triangulation produces a set of vertices defined by integer components, which can later be converted to real-valued belief vectors for use

[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.

```
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).a
    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′] = 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 ⇒ πfib(e).u 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`.

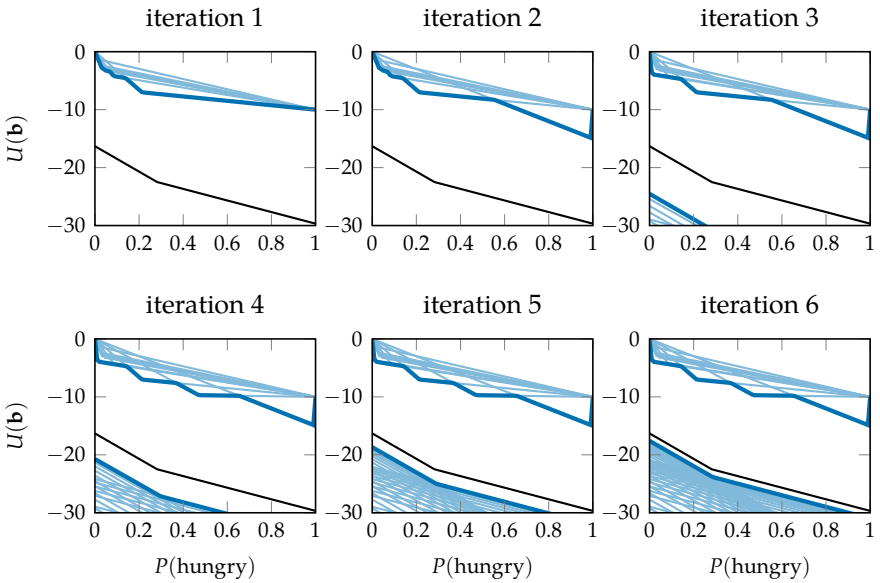


Figure 21.8. The evolution of the upper bound represented by sawtooth pairs and the lower bound represented by alpha vectors for the crying baby problem. The optimal value function is shown in black.

in our value function approximation. The number of vertices depends upon the dimensionality n and granularity m. Given a granularity $m > 0$, the vertices in the Freudenthal triangulation are the n-dimensional vectors $\mathbf{v}$ with integer components that satisfy:

$$m = v_1 \geq v_2 \geq v_3 \geq \cdots \geq v_n \geq 0 \tag{21.17}$$

Algorithm 21.15 generates this set of vertices. Figure 21.9 shows how the number of vertices varies with n and m.

```
function freudenthal_vertices!(vertices, v, i)
    n = length(v)
    if i > n
        push!(vertices, copy(v))
        return
    end
    for k in 0:v[i-1]
        v[i] = k
        freudenthal_vertices!(vertices, v, i+1)
    end
end

function freudenthal_vertices(n, m)
    vertices = Vector{Int}[]
    v = zeros(Int, n)
    v[1] = m
    freudenthal_vertices!(vertices, v, 2)
    return vertices
end
```

Algorithm 21.15. A method for computing the set of integer vertices for a Freudenthal triangulation of `n` dimensions. The positive integer `m` determines the granularity of the discretization, with larger values resulting in a finer discretization.

To be used in belief-state planning, these integer vertices must be transformed to form a triangulation of the belief over n discrete states. This transformation changes vertex $\mathbf{v}$ into:

$$\mathbf{v}' = \frac{1}{m}[v_1 - v_2, v_2 - v_3, v_3 - v_4, \ldots, v_{n-1} - v_n, v_n] \tag{21.18}$$

where $\mathbf{v}'$ represents a valid probability distribution that sums to 1. The probability assigned to the ith state is given by v_i'. Example 21.2 shows how to generate the vertices and transform them into points in the belief-space.

This transformation is equivalent to the matrix multiplication $\mathbf{v}' = \mathbf{B}\mathbf{v}$ with

$$\mathbf{B} = \frac{1}{m}\begin{bmatrix} 1 & -1 & 0 & 0 & & & 0 \\ 0 & 1 & -1 & 0 & \cdots & & 0 \\ 0 & 0 & 1 & -1 & & & 0 \\ & \vdots & & & & & \vdots \\ 0 & 0 & \cdots & & & 1 & -1 \\ 0 & 0 & & & & 0 & 1 \end{bmatrix} \tag{21.19}$$

This matrix $\mathbf{B}$ is invertible, so the opposite transformation can be done. Hence, any belief $\mathbf{b}$ can thus be mapped into the Freudenthal space to obtain its counter-

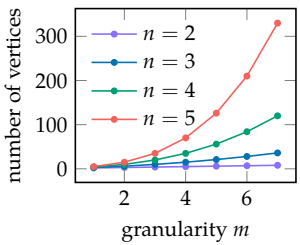


Figure 21.9. The number of beliefs in the Freudenthal triangulation for beliefs over n discrete states as a function of the granularity m: $(m + n - 1)!/m!(n - 1)!$.

part $\mathbf{x} = \mathbf{B}^{-1}\mathbf{b}$. It turns out that the ith component of $\mathbf{B}^{-1}\mathbf{b}$ is simply $m \sum_{k=i}^{n} b_k$. Algorithm 21.16 converts between these two forms.

```
to_belief(x) = [x[1:end-1] - x[2:end]; x[end]]./x[1]
to_freudenthal(b, m) =
    [sum(b[k] for k in i:length(b))*m for i in 1:length(b)]
```

Algorithm 21.16. The method `to_belief` takes a vector `x` and a granularity integer `m` and returns $\mathbf{Bx}$, a vector in belief space. The method `to_freudenthal` takes a belief vector `b` and a granularity integer `m` and returns $\mathbf{B}^{-1}\mathbf{b}$, a vector in the space of the Freudenthal triangulation.

If we know the value function U at a set of discrete belief points B, we can use the triangulation to estimate the value function at arbitrary belief points through interpolation. We will first discuss how to interpolate a general function f in Freudenthal space at a point $\mathbf{x}$. To interpolate in belief space, we use the transformation we just discussed to map between the Freudenthal space to belief space.

If we know the values of a function f at the integer Freudenthal vertices, we can use triangulation to estimate the value at an arbitrary point $\mathbf{x}$ from the $n+1$ vertices of the simplex enclosing $\mathbf{x}$:

$$f(\mathbf{x}) = \sum_{i=1}^{n+1} \lambda_i f(\mathbf{v}^{(i)}) \tag{21.20}$$

The vertices of the simplex containing $\mathbf{x}$ are given by $\mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(n+1)}$. The scalars $\lambda_{1:n+1}$ are the *barycentric coordinates* of $\mathbf{x}$ with respect to the simplex vertices. They are non-negative weights that sum to 1, such that $\mathbf{x} = \sum_{i=1}^{n+1} \lambda_i \mathbf{v}^{(i)}$.

The first vertex consists of the componentwise floor of $\mathbf{x}$:[14]

$$\mathbf{v}^{(1)} = [\lfloor x_1 \rfloor, \lfloor x_2 \rfloor, \ldots, \lfloor x_n \rfloor] \tag{21.21}$$

[14] Recall that $\lfloor a \rfloor$ returns the greatest integer less than or equal to a.

We then compute the difference $\mathbf{d} = \mathbf{x} - \mathbf{v}^{(1)}$, and sort the components in descending order:

$$d_{p_1} \geq d_{p_2} \geq \ldots \geq d_{p_n} \tag{21.22}$$

where $\mathbf{p}$ is a permutation of $1:n$.

The remaining n simplex vertices can be constructed according to:

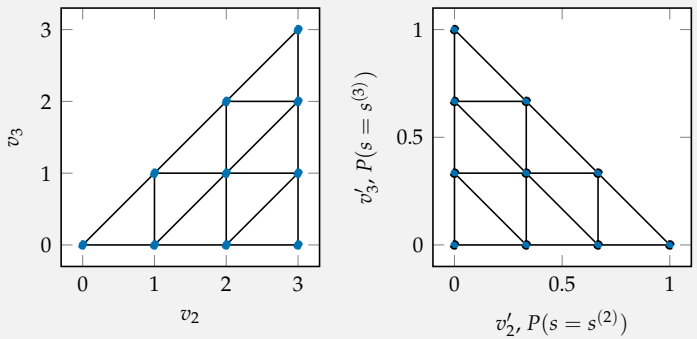
$$\mathbf{v}^{(k+1)} = \mathbf{v}^{(k)} + \mathbf{e}_{p_k} \tag{21.23}$$

Example 21.2. The Freudenthal discretization obtained for an example belief space and resolution.

The Freudenthal discretization of a belief space over 3 states with $m = 3$ has 10 vertices:

$$\left\{\begin{array}{cccc} [3,0,0] & [3,1,0] & [3,2,0] & [3,3,0] \\ & [3,1,1] & [3,2,1] & [3,3,1] \\ & & [3,2,2] & [3,3,2] \\ & & & [3,3,3] \end{array}\right\}$$

The first component in every vertex is m. The other two dimensions, along with the projected triangulation, are shown in the left image below.



In the right image, we have the same triangulation mapped to the 3-state belief space. These vertices are:

$$\left\{\begin{array}{cccc} [1,0,0] & [2/3,1/3,0] & [1/3,2/3,0] & [0,1,0] \\ & [2/3,0,1/3] & [1/3,1/3,1/3] & [0,2/3,1/3] \\ & & [1/3,0,2/3] & [0,1/3,2/3] \\ & & & [0,0,1] \end{array}\right\}$$

```
function freudenthal_simplex(x)
    n = length(x)
    vertices = Vector{Vector{Int}}(undef, n+1)
    vertices[1] = floor.(Int, x)
    d = x - vertices[1]
    p = sortperm(d, rev=true)
    for i in 2:n+1
        vertices[i] = copy(vertices[i-1])
        vertices[i][p[i-1]] += 1
    end
    return vertices
end
```

Algorithm 21.17. A method for computing the set of integer vertices for a Freudenthal triangulation. If the vector `x` is in $\mathbb{R}^n$, then $n+1$ vertices will be returned.

Suppose we want to find the simplex vertices for the Freudenthal triangulation of $\mathbf{x} = [1.2, -3.4, 2]$. The first vertex is $\mathbf{v}^{(1)} = [1, -4, 2]$, resulting in a difference $\mathbf{d} = [0.2, 0.6, 0.0]$. We arrange the components of $\mathbf{d}$ in descending order, $0.6 \geq 0.2 \geq 0.0$ according to the permutation $\mathbf{p} = [2, 1, 3]$. The remaining simplex vertices are:

$$\begin{aligned}
\mathbf{v}^{(2)} &= [1, -4, 2] + [0, 1, 0] = [1, -3, 2] \\
\mathbf{v}^{(3)} &= [1, -3, 2] + [1, 0, 0] = [2, -3, 2] \\
\mathbf{v}^{(4)} &= [2, -3, 2] + [0, 0, 1] = [2, -3, 3]
\end{aligned}$$

Example 21.3. Computing the simplex vertices for a sample vector in $\mathbb{R}^3$.

where $\mathbf{e}_i$ represents the ith standard basis vector. Algorithm 21.17 computes the Freudenthal simplex containing a vector, and example 21.3 illustrates how it works for an arbitrary vector.

The barycentric coordinates $\boldsymbol{\lambda}$ are obtained using:[15]

$$\begin{aligned}
\lambda_{n+1} &= d_{p_n} \\
\lambda_n &= d_{p_{n-1}} - d_{p_n} \\
\lambda_{n-1} &= d_{p_{n-2}} - d_{p_{n-1}} \\
&\vdots \\
\lambda_2 &= d_{p_1} - d_{p_2} \\
\lambda_1 &= 1 - \sum_{i=1}^{n+1} \lambda_i
\end{aligned} \tag{21.24}$$

[15] This process is derived in exercise 21.4.

Algorithm 21.18 computes barycentric coordinates. Example 21.4 illustrates how to perform this computation.

```
function barycentric_coordinates(x, vertices)
    d = x - vertices[1]
    p = sortperm(d, rev=true)
    n = length(x)
    λ = zeros(n+1)
    λ[n+1] = d[p[n]]
    for i in n:-1:2
        λ[i] = d[p[i-1]] - d[p[i]]
    end
    λ[1] = 1 - sum(λ[2:end])
    return λ
end
```

Algorithm 21.18. A method for computing the barycentric coordinates for a vector `x` in a Freudenthal simplex given by vertices `V`. These vertices can be computed using algorithm 21.17.

So far we have discussed how to interpolate in Freudenthal space with integer simplex vertices. To interpolate in belief space, we simply use the transformation to the belief space. Suppose we know the values $U(\mathbf{b})$ at all of the vertices $\mathbf{v}'$ in the belief triangulation. Given any new belief $\mathbf{b}$, we can compute $\mathbf{x} = \mathbf{B}^{-1}\mathbf{b}$, find the vertices $\mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(n+1)}$ of the corresponding simplex in the original Freudenthal triangulation, and then compute the barycentric coordinates $\boldsymbol{\lambda}$ for $\mathbf{x}$. The interpolated value is then:

$$U(\mathbf{b}) = \sum_{i=1}^{n+1} \lambda_i U(\mathbf{v}'^{(i)}) = \sum_{i=1}^{n+1} \lambda_i U(\mathbf{B}\mathbf{v}^{(i)}) \tag{21.25}$$

Example 21.4. Computing the barycentric coordinates for a sample vector in $\mathbb{R}^3$.

Suppose we want to find the barycentric coordinates for the Freudenthal triangulation of $\mathbf{x} = [1.2, -3.4, 2]$. Recall that the vertices for $\mathbf{x}$ were obtained in example 21.3. We compute:

$$\begin{aligned}\lambda_4 &= d_3 = 0.0\\ \lambda_3 &= d_1 - d_3 = 0.2\\ \lambda_2 &= d_2 - d_1 = 0.4\\ \lambda_1 &= 1 - 0.0 - 0.2 - 0.4 = 0.4\end{aligned}$$

This interpolation is implemented in algorithm 21.19.

Algorithm 21.20 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.[16] Figure 21.10 shows an example policy and utility function.

[16] 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. 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.

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 for 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.

```
struct TriangulatedPolicy
    𝒫 # POMDP problem
    V # dictionary mapping beliefs to utilities
    B # beliefs
    m # granularity
end

function TriangulatedPolicy(𝒫::POMDP, m)
    vertices = freudenthal_vertices(length(𝒫.𝒮), m)
    B = to_belief.(vertices)
    V = Dict(b ⇒ 0.0 for b in B)
    return TriangulatedPolicy(𝒫, V, B, m)
end

function utility(π::TriangulatedPolicy, b)
    x = to_freudenthal(b, π.m)
    vertices = freudenthal_simplex(x)
    B = to_belief.(vertices)
    λ = barycentric_coordinates(x, vertices)
    return sum(λi < √eps() ? 0.0 : λi*π.V[b] for (λi, b) in zip(λ, B))
end

function (π::TriangulatedPolicy)(b)
    U(b) = utility(π, b)
    return greedy(π.𝒫, U, b)
end
```

Algorithm 21.19. A policy representation using Freudenthal triangulation with granularity `m`. As with the sawtooth method, we maintain a dictionary `V` that map belief vectors in `B` to utilities. At construction, we initialize the utilities to 0 in this implementation, but if we want to use this policy representation for an upper bound, then we would need to initialize those utilities appropriately. We define a `utility` function to perform the necessary interpolation to compute the utility for an arbitrary belief `b`. We can extract a policy from this representation using greedy lookahead, as with the other policy representations introduced in this chapter.

```
struct TriangulatedIteration
    m     # granulatiry
    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 21.20. 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.

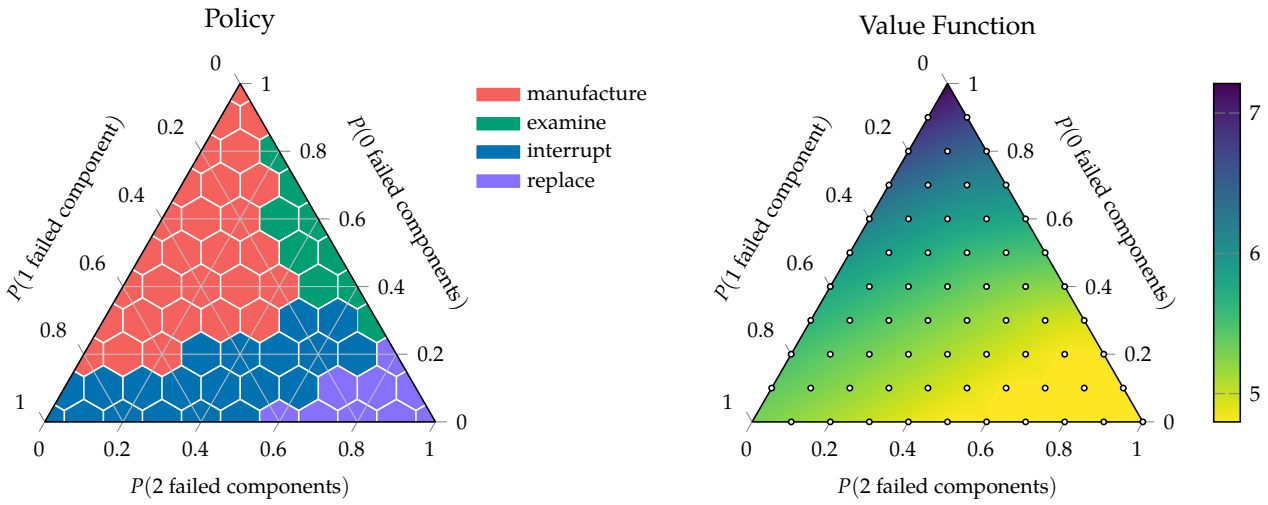


Figure 21.10. The policy and value function for the maintenance problem with granularity $m = 10$ after 11 iterations. The value function plot shows the discrete belief points as white dots. This policy approximates the exact policy given in appendix F.8.

- 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 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: The QMDP update is given by equation (21.1):

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

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 of the entries in the alpha vectors are zero, only the reward term contributes to the QMDP update:

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

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):

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

This leads to:

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

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:

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

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.

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

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

Exercise 21.2. Consider the environment from the previous exercise. 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 is like the QMDP update, but lacks the maximization. It is given by equation (21.6):

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

We initialize the components of the alpha vectors to zero and run to convergence:

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

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

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

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

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:

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

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

Exercise 21.3. 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 non-basis belief, we start by finding the farthest basis belief $\mathbf{e}_i$. Starting with $\mathbf{b}_3$, we compute:

$$\begin{aligned}
i_3 &= \arg\max_j \left\|\mathbf{b} - \mathbf{e}_j\right\|_1 - \left\|\mathbf{b}_3 - \mathbf{e}_j\right\|_1 \\
\|\mathbf{b} - e_1\|_1 - \|\mathbf{b}_3 - \mathbf{e}_1\|_1 &= \left\|\begin{bmatrix}0.5\\0.5\end{bmatrix} - \begin{bmatrix}1\\0\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}0.8\\0.2\end{bmatrix} - \begin{bmatrix}1\\0\end{bmatrix}\right\|_1 \\
&= \left\|\begin{bmatrix}-0.5\\0.5\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}-0.2\\0.2\end{bmatrix}\right\|_1 \\
&= 0.6 \\
\|\mathbf{b} - \mathbf{e}_2\|_1 - \|\mathbf{b}_3 - \mathbf{e}_2\|_1 &= \left\|\begin{bmatrix}0.5\\0.5\end{bmatrix} - \begin{bmatrix}0\\1\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}0.8\\0.2\end{bmatrix} - \begin{bmatrix}0\\1\end{bmatrix}\right\|_1 \\
&= \left\|\begin{bmatrix}0.5\\-0.5\end{bmatrix}\right\|_1 - \left\|\begin{bmatrix}0.8\\-0.8\end{bmatrix}\right\|_1 \\
&= -0.6 \\
i_3 &= 1
\end{aligned}$$

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

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

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

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 $(\mathbf{e}_1, \mathbf{b}_4)$:

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

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 21.4. Derive equation (21.24), used to obtain the barycentric coordinates $\boldsymbol{\lambda}$ for a vector $\mathbf{x} \in \mathbb{R}^n$ in a Freudenthal triangulation simplex with $n+1$ vertices $\mathbf{v}^{(i)}$.

Solution: The barycentric coordinates are a convex combination of the simplex vertices that produce $\mathbf{x}$:

$$\mathbf{x} = \sum_{i=1}^{n+1} \lambda_i \mathbf{v}^{(i)}$$

The components of $\boldsymbol{\lambda}$ are non-negative and must sum to 1.

Recalling the definition of the vertices for the Freudenthal triangulation, we expand:[17]

$$\begin{aligned}
\mathbf{x} &= \sum_{i=1}^{n+1} \lambda_i \mathbf{v}^{(i)} \\
&= \sum_{i=1}^{n+1} \lambda_i \left(\mathbf{v}^{(1)} \sum_{j=1}^{i-1} \mathbf{e}_{p_j} \right) \\
&= \left(\sum_{i=1}^{n+1} \lambda_i \right) \mathbf{v}^{(1)} + \left(\sum_{i=2}^{n+1} \lambda_i \right) \mathbf{e}_{p_1} + \left(\sum_{i=3}^{n+1} \lambda_i \right) \mathbf{e}_{p_2} + \ldots + \left(\sum_{i=n}^{n+1} \lambda_i \right) \mathbf{e}_{p_{n-1}} + \lambda_{n+1} \mathbf{e}_{p_n} \\
&= \mathbf{v}^{(1)} + \left(\sum_{i=2}^{n+1} \lambda_i \right) \mathbf{e}_{p_1} + \left(\sum_{i=3}^{n+1} \lambda_i \right) \mathbf{e}_{p_2} + \ldots + \left(\sum_{i=n}^{n+1} \lambda_i \right) \mathbf{e}_{p_{n-1}} + \lambda_{n+1} \mathbf{e}_{p_n}
\end{aligned}$$

where in the last step we used the fact that the components sum to 1.

Recall that $\mathbf{d} = \mathbf{x} - \mathbf{v}^{(1)}$. We thus have:

$$\mathbf{d} = \left(\sum_{i=2}^{n+1} \lambda_i \right) \mathbf{e}_{p_1} + \left(\sum_{i=3}^{n+1} \lambda_i \right) \mathbf{e}_{p_2} \ldots + \left(\sum_{i=n}^{n+1} \lambda_i \right) \mathbf{e}_{p_{n-1}} + \lambda_{n+1} \mathbf{e}_{p_n}$$

This can be solved starting with λ_{n+1} to obtain equation (21.24).

Exercise 21.5. Consider Freudenthal value interpolation for the belief $\mathbf{b} = [4/6, 1/6, 1/6]$ with a granularity constant $m = 3$. Is there a unique permutation of the components of the direction $\mathbf{d}$ associated with this belief? Geometrically, what does this say about $\mathbf{b}$? Which permutation of $\mathbf{d}$ should be used?

Solution: We begin by converting the belief vector to the space of the Freudenthal triangulation, resulting in:

$$\mathbf{x} = \mathbf{B}^{-1}\mathbf{b} = [3, 1, 0.5]$$

The first vertex in the Freudenthal triangulation is formed by the floor of the components of $\mathbf{x}$, and is thus $\mathbf{v}^{(1)} = [3, 1, 0]$. We can calculate $\mathbf{d}$:

$$\mathbf{d} = \mathbf{x} - \mathbf{v}^{(1)} = [0, 0, 0.5]$$

The components of $\mathbf{d}$ are non-unique, so both permutations $p = [3, 1, 2]$ and $p = [3, 2, 1]$ are valid. Geometrically, this means that $\mathbf{b}$ lies on an edge shared by two simplexes. Both permutations are equally valid for value interpolation, resulting in the same barycentric coordinates that zero out the problematic components. Here, the permutations both produce $\boldsymbol{\lambda} = [0.5, 0.5, 0, 0]$.

[17] This derivation can be found in W. S. Lovejoy, ''Computationally Feasible Bounds for Partially Observed Markov Decision Processes,'' *Operations Research*, vol. 39, no. 1, pp. 162–175, 1991.