20 Exact Belief State Planning

Because states are not directly observable, the agent must use its past history of actions and observations to inform its belief, which can be represented as a probability distribution over states as discussed in the previous chapter. The objective in a POMDP is to choose actions that maximize the accumulation of reward while interacting with the environment. There are different approaches for computing an optimal policy that maps beliefs to actions given models of the transitions, observations, and rewards.[1] One approach is to convert a POMDP into an MDP and apply dynamic programming. Other approaches include representing policies as conditional plans or as piecewise linear value functions over the belief space. The chapter concludes with an algorithm for computing an optimal policy that is analogous to value iteration for MDPs.

[1] A discussion of exact solution methods is provided by L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and Acting in Partially Observable Stochastic Domains," *Artificial Intelligence*, vol. 101, no. 1–2, pp. 99–134, 1998.

20.1 Belief-State Markov Decision Processes

Any POMDP can be viewed as an MDP that uses beliefs as states, also called a *belief-state MDP*.[2] The state space of a belief-state MDP is the set of all beliefs $\mathcal{B}$. The action space is identical to that of the POMDP. If the state and observation spaces are discrete, the belief-state transition function for a belief-state MDP is

[2] K. J. Åström, "Optimal Control of Markov Processes with Incomplete State Information," *Journal of Mathematical Analysis and Applications*, vol. 10, no. 1, pp. 174–205, 1965.

given by:

$$\begin{aligned}
T(b' \mid b, a) &= P(b' \mid b, a) && (20.1)\\
&= \sum_o P(b' \mid b, a, o) P(o \mid b, a) && (20.2)\\
&= \sum_o P(b' \mid b, a, o) \sum_s P(o \mid b, a, s) P(s \mid b, a) && (20.3)\\
&= \sum_o P(b' \mid b, a, o) \sum_s P(o \mid b, a, s) b(s) && (20.4)\\
&= \sum_o P(b' \mid b, a, o) \sum_{s'} \sum_s P(o \mid b, a, s, s') P(s' \mid b, s, a) b(s) && (20.5)\\
&= \sum_o (b' = \text{Update}(b, a, o)) \sum_{s'} O(o \mid a, s') \sum_s T(s' \mid s, a) b(s) && (20.6)
\end{aligned}$$

Above, $\text{Update}(b, a, o)$ returns the updated belief using the deterministic process discussed in the previous chapter.[3] For continuous problems, we replace the summations with integrals.

[3] As a reminder, we use the convention where a logical statement in parentheses is treated numerically as 1 when true and 0 when false.

The reward function for a belief-state MDP depends on the belief and action taken. It is simply the expected value of the reward:

$$R(b, a) = \sum_s R(s, a) b(s) \tag{20.7}$$

Solving belief-state MDPs is challenging because the state space is continuous. We can use the approximate dynamic programming techniques presented in earlier chapters, but we can often do better by taking advantage of the structure of the belief-state MDP, as will be discussed in the remainder of this chapter.

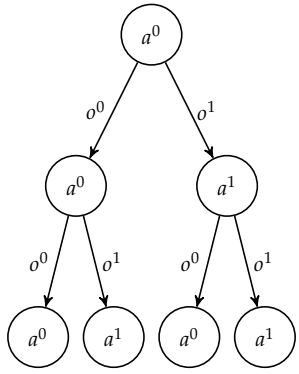


Figure 20.1. An example 3-step conditional plan.

20.2 Conditional Plans

There are different ways to represent policies for POMDPs. One approach is to use a *conditional plan* represented as a tree. Figure 20.1 shows an example of a three-step conditional plan with binary action and observation spaces. The nodes correspond to belief states. The edges are annotated with observations, and the nodes are annotated with actions. If we have a plan π, the action associated with the root is denoted $\pi()$ and the subplan associated with observation o is denoted $\pi(o)$. Algorithm 20.1 provides an implementation.

A conditional plan tells us what to do in response to our observations up to the horizon represented by the tree. To execute a conditional plan, we start with

```
struct ConditionalPlan
    a        # action to take at root
    subplans # dictionary mapping observations to subplans
end

ConditionalPlan(a) = ConditionalPlan(a, Dict())

(π::ConditionalPlan)() = π.a
(π::ConditionalPlan)(o) = π.subplans[o]
```

Algorithm 20.1. The conditional plan data structure consisting of an action and a mapping from observations to subplans. The `subplans` field is a `Dict` from observations to conditional plans. For convenience, we have created a special constructor for plans consisting of a single node.

the root node and execute the action associated with it. We proceed down the tree according to our observations, taking the actions associated with the nodes through which we pass.

Suppose we have a conditional plan π, and we want to compute its expected utility when starting from state s. This computation can be done recursively:

$$U^\pi(s) = R(s, \pi()) + \gamma \left[\sum_{s'} T(s' \mid s, \pi()) \sum_o O(o \mid \pi(), s') U^{\pi(o)}(s') \right] \tag{20.8}$$

An implementation for this procedure is given in algorithm 20.2.

```
function lookahead(𝒫::POMDP, U, s, a)
    𝒮, 𝒪, T, O, R, γ = 𝒫.𝒮, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R, 𝒫.γ
    u′ = sum(T(s,a,s′)*sum(O(a,s′,o)*U(o,s′) for o in 𝒪) for s′ in 𝒮)
    return R(s,a) + γ*u′
end

function evaluate_plan(𝒫::POMDP, π::ConditionalPlan, s)
    U(o,s′) = evaluate_plan(𝒫, π(o), s′)
    return isempty(π.subplans) ? 𝒫.R(s,π()) : lookahead(𝒫, U, s, π())
end
```

Algorithm 20.2. A method for evaluating a conditional plan `π` for MDP `𝒫` starting at state `s`. Plans are represented as tuples consisting of an action and a dictionary mapping observations to subplans.

If we do not know the current state exactly, we can compute the utility of our belief b as follows:

$$U^\pi(b) = \sum_s b(s) U^\pi(s) \tag{20.9}$$

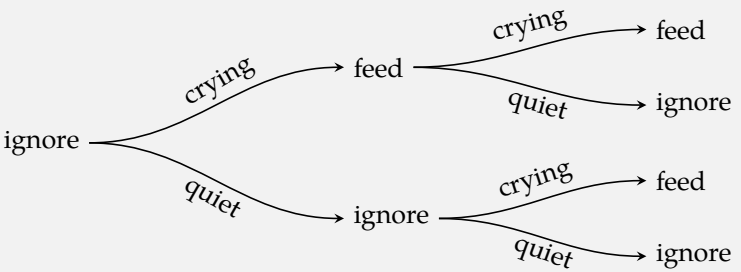
Example 20.1 shows how to compute the utility associated with a three-step conditional plan.

Now that we have a way to evaluate conditional plans up to some horizon h, we can compute the optimal h-step value function:

$$U^*(b) = \max_\pi U^\pi(b) \tag{20.10}$$

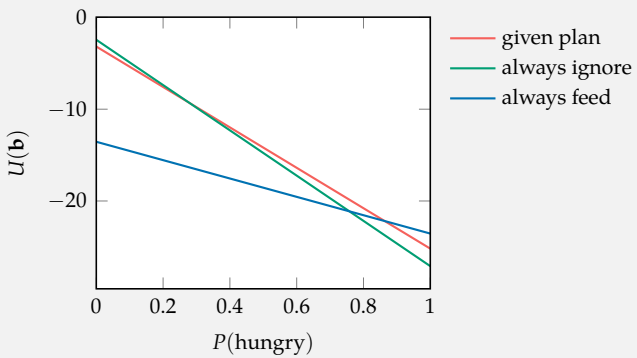
Example 20.1. A conditional plan for the three-step crying baby problem (appendix F.7), evaluated and compared to two simpler conditional plans.

Consider the following 3-step conditional plan for the crying baby problem:



In this plan, we begin by ignoring the baby. If we observe any crying, we feed the baby. If we do not observe any crying, we ignore the baby. Our third action again feeds if there is crying.

The expected utility for this plan in belief space is plotted alongside a 3-step plan that always feeds the baby and one that always ignores the baby.



We find that the given plan is not universally better than either always ignoring or always feeding the baby.

An optimal action can be generated from the action associated with the root of a maximizing π.

Solving an h-step POMDP by directly enumerating all h-step conditional plans is generally computationally intractable as shown in figure 20.2. There are $(|\mathcal{O}|^h - 1)/(|\mathcal{O}| - 1)$ nodes in an h-step plan. In general, any action can be inserted into any node, resulting in $|\mathcal{A}|^{(|\mathcal{O}|^h-1)/(|\mathcal{O}|-1)}$ possible h-step plans. This exponential growth means enumerating over all plans is intractable even for modest values of h. As will be discussed later in this chapter, there are alternatives to explicitly enumerating over all possible plans.

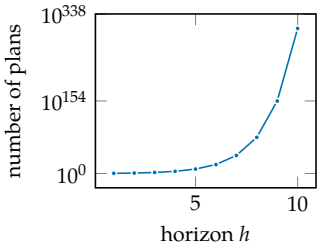


Figure 20.2. Even for extremely simple POMDPs such as the crying baby problem with three actions and two observations, the number of plans grows extremely quickly with the planning horizon. In comparison, only 2 plans dominate in each step for the crying baby problem, so only 8 plans need be evaluated with each iteration: the 2 actions as roots and the 2 dominant plans as subtrees for each observation.

20.3 Alpha Vectors

We can rewrite equation (20.9) in vector form:

$$U^\pi(b) = \sum_s b(s) U^\pi(s) = \boldsymbol{\alpha}_\pi^\top \mathbf{b} \tag{20.11}$$

The vector $\boldsymbol{\alpha}_\pi$ is called an *alpha vector* and contains the expected utility under plan π for each state. As with belief vectors, alpha vectors have dimension $|\mathcal{S}|$. Unlike beliefs, the components in alpha vectors represent utilities and not probability masses. Algorithm 20.3 shows how to compute an alpha vector.

```
function alphavector(𝒫::POMDP, π::ConditionalPlan)
    return [evaluate_plan(𝒫, π, s) for s in 𝒫.𝒮]
end
```

Algorithm 20.3. We can generate an alpha vector from a conditional plan by calling `evaluate_plan` from all possible initial states.

Each alpha vector defines a hyperplane in belief space. The optimal value function equation (20.11) is the maximum over these hyperplanes,

$$U^*(\mathbf{b}) = \max_\pi \boldsymbol{\alpha}_\pi^\top \mathbf{b} \tag{20.12}$$

making the value function piecewise-linear and convex.[4]

An alternative to using a conditional plan to represent a policy is to use a set of alpha vectors Γ, each annotated with an action. Although not practical, one way to generate the set Γ is to enumerate the set of h-step conditional plans and compute their alpha vectors. The action associated with an alpha vector is the action at the root of the associated conditional plan. We execute a policy represented by Γ by updating our belief state and executing the action associated with the dominating

[4] The optimal value function for continuous-state POMDPs is also convex, as can be seen by approximating the POMDP through state space discretization and taking the limit as the number of discrete states approaches infinity.

alpha vector at the new belief $\mathbf{b}$. The dominating alpha vector $\boldsymbol{\alpha}$ at $\mathbf{b}$ is the one that maximizes $\boldsymbol{\alpha}^\top \mathbf{b}$. This strategy can be used to select actions beyond the horizon of the original conditional plans. Algorithm 20.4 provides an implementation.

```
struct AlphaVectorPolicy
    𝒫 # POMDP problem
    Γ # alpha vectors
    a # actions associated with alpha vectors
end

function utility(π::AlphaVectorPolicy, b)
    return maximum(α⋅b for α in π.Γ)
end

function (π::AlphaVectorPolicy)(b)
    u, i = findmax([α⋅b for α in π.Γ])
    return (a=π.a[i], u=u)
end
```

Algorithm 20.4. An alpha vector policy is defined in terms of a set of alpha vectors `Γ` and an array of associated actions `a`. Given current belief `b`, it will find the alpha vector that gives the highest value at that belief point. It will return the associated action and its utility.

If we use *one-step lookahead*, we do not have to keep track of the actions associated with the alpha vectors in Γ. The one-step lookahead action from belief b using the value function represented by Γ, denoted U^Γ, is

$$\pi^\Gamma(b) = \arg\max_a \left[R(b,a) + \gamma \sum_o P(o \mid b,a) U^\Gamma(\text{Update}(b,a,o)) \right] \tag{20.13}$$

where

$$P(o \mid b,a) = \sum_s P(o \mid s,a) b(s) \tag{20.14}$$

$$P(o \mid s,a) = \sum_{s'} T(s' \mid s,a) O(o \mid s',a) \tag{20.15}$$

Algorithm 20.5 provides an implementation. Example 20.2 demonstrates one-step lookahead on the crying baby problem.

20.4 Pruning

If we have a collection of alpha vectors Γ, we may want to *prune* alpha vectors that do not contribute to our representation of the value function or plans that are not optimal for any belief. Removing such alpha vectors or plans can improve computational efficiency. We can check whether an alpha vector $\boldsymbol{\alpha}$ is *dominated* by

```
function lookahead(𝒫::POMDP, U, b::Vector, a)
    𝒮, 𝒪, T, O, R, γ = 𝒫.𝒮, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R, 𝒫.γ
    r = sum(R(s,a)*b[i] for (i,s) in enumerate(𝒮))
    Posa(o,s,a) = sum(O(a,s′,o)*T(s,a,s′) for s′ in 𝒮)
    Poba(o,b,a) = sum(b[i]*Posa(o,s,a) for (i,s) in enumerate(𝒮))
    return r + γ*sum(Poba(o,b,a)*U(update(b, 𝒫, a, o)) for o in 𝒪)
end

function greedy(𝒫::POMDP, U, b::Vector)
    u, a = _findmax(a->lookahead(𝒫, U, b, a), 𝒫.𝒜)
    return (a=a, u=u)
end

struct LookaheadAlphaVectorPolicy
    𝒫 # POMDP problem
    Γ # alpha vectors
end

function utility(π::LookaheadAlphaVectorPolicy, b)
    return maximum(α⋅b for α in π.Γ)
end

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

Algorithm 20.5. A policy represented by a set of alpha vectors Γ. It uses one-step lookahead to produce an optimal action and associated utility. Equation (20.13) is used to compute the lookahead.

Example 20.2. An example of a lookahead policy applied to the crying baby problem.

Consider using one-step lookahead on the crying baby problem with a value function given by the alpha vectors $[-3.7, -15]$ and $[-2, -21]$. Suppose our current belief is $b = [0.5, 0.5]$, meaning that we believe it is equally likely the baby is hungry as not hungry. We apply equation (20.13):

$$
\begin{aligned}
b \;&\to\; R(b, \text{feed}) = -10 \\
&\to\; \gamma P(\text{crying} \mid b, \text{feed}) U(\text{Update}(b, \text{feed}, \text{crying})) = -0.18 \\
&\to\; \gamma P(\text{quiet} \mid b, \text{feed}) U(\text{Update}(b, \text{feed}, \text{quiet})) = -1.62 \\
&\to\; Q(b, \text{feed}) = -11.8 \\
&\to\; R(b, \text{ignore}) = -5 \\
&\to\; \gamma P(\text{crying} \mid b, \text{ignore}) U(\text{Update}(b, \text{ignore}, \text{crying})) = -6.09 \\
&\to\; \gamma P(\text{quiet} \mid b, \text{ignore}) U(\text{Update}(b, \text{ignore}, \text{quiet})) = -2.81 \\
&\to\; Q(b, \text{ignore}) = -13.9 \\
&\to\; R(b, \text{sing}) = -5.5 \\
&\to\; \gamma P(\text{crying} \mid b, \text{sing}) U(\text{Update}(b, \text{sing}, \text{crying})) = -6.09 \\
&\to\; \gamma P(\text{quiet} \mid b, \text{sing}) U(\text{Update}(b, \text{sing}, \text{quiet})) = -1.85 \\
&\to\; Q(b, \text{sing}) = -14.0
\end{aligned}
$$

We use $Q(b, a)$ to represent the action value function from a belief state. The policy predicts that feeding the baby will result in the highest expected utility, so it takes that action.

the alpha vectors in a set Γ by solving a linear program to maximize the utility gap δ that vector achieves over all other vectors:[5]

$$\begin{aligned} \underset{\delta, \mathbf{b}}{\text{maximize}} \quad & \delta \\ \text{subject to} \quad & \mathbf{b} \geq \mathbf{0} \\ & \mathbf{1}^\top \mathbf{b} = 1 \\ & \boldsymbol{\alpha}^\top \mathbf{b} \geq \boldsymbol{\alpha}'^\top \mathbf{b} + \delta, \qquad \boldsymbol{\alpha}' \in \Gamma \end{aligned} \tag{20.16}$$

[5] Constraints of the form $\mathbf{a} \geq \mathbf{b}$ are element-wise. That is, we mean $a_i \geq b_i$ for all i.

The first two constraints ensure that $\mathbf{b}$ is a categorical distribution, and the final set of constraints ensures that we find a belief vector for which $\boldsymbol{\alpha}$ has a higher expected reward than all alpha vectors in Γ. If, after solving the linear program, the utility gap δ is negative, then $\boldsymbol{\alpha}$ is dominated. If δ is positive, then $\boldsymbol{\alpha}$ is not dominated and $\mathbf{b}$ is a belief at which $\boldsymbol{\alpha}$ is not dominated. Algorithm 20.6 provides an implementation for solving equation (20.16) to determine a belief, if one exists, where δ is most positive.

```
function find_maximal_belief(α, Γ)
    m = length(α)
    if isempty(Γ)
        return fill(1/m, m) # arbitrary belief
    end
    model = Model(GLPK.Optimizer)
    @variable(model, δ)
    @variable(model, b[i=1:m] ≥ 0)
    @constraint(model, sum(b) == 1.0)
    for a in Γ
        @constraint(model, (α-a)⋅b ≥ δ)
    end
    @objective(model, Max, δ)
    optimize!(model)
    return value(δ) > 0 ? value.(b) : nothing
end
```

Algorithm 20.6. A method for finding the belief vector `b` for which the alpha vector `α` improves the most compared to the set of alpha vectors `Γ`. Nothing is returned if no such belief exists. The packages `JuMP.jl` and `GLPK.jl` provide a mathematical optimization framework and a solver for linear programs, respectively.

Algorithm 20.7 provides a procedure that uses algorithm 20.6 for finding the dominating alpha vectors in a set Γ. Initially, all of the alpha vectors are candidates for being dominating. We then choose one of the candidates and determine the belief b where the candidate leads to the greatest improvement in value compared to all other alpha vectors in the dominating set. If the candidate does not bring improvement, we remove it as a candidate. If it does bring improvement, we move an alpha vector from the candidate set that brings the greatest improvement

at b to the dominating set. The process continues until there are no longer any candidates. We can prune away any alpha vectors and associated conditional plans that are not dominating at any belief point. Example 20.3 demonstrates pruning on the crying baby problem.

```
function find_dominating(Γ)
    n = length(Γ)
    candidates, dominating = trues(n), falses(n)
    while any(candidates)
        i = findfirst(candidates)
        b = find_maximal_belief(Γ[i], Γ[dominating])
        if b === nothing
            candidates[i] = false
        else
            k = argmax([candidates[j] ? b⋅Γ[j] : -Inf for j in 1:n])
            candidates[k], dominating[k] = false, true
        end
    end
    return dominating
end

function prune(plans, Γ)
    d = find_dominating(Γ)
    return (plans[d], Γ[d])
end
```

Algorithm 20.7. A method for pruning dominated alpha vectors and associated plans. The `find_dominating` function identifies all of the dominating alpha vectors in the set `Γ`. It uses binary vectors `candidates` and `dominating` to track which alpha vectors are candidates for inclusion in the dominating set and which are currently in the dominating set, respectively.

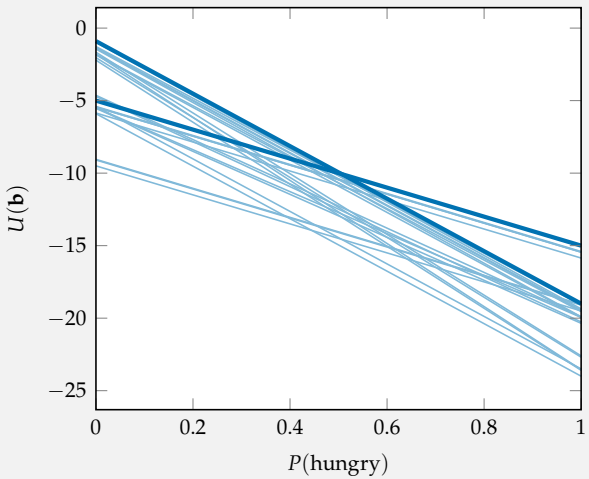
20.5 *Value Iteration*

The value iteration algorithm for MDPs can be adapted for POMDPs.[6] POMDP *value iteration* (algorithm 20.8) begins by constructing all one-step plans. We prune any plans that are never optimal for any initial belief. Then, we expand all combinations of one-step plans to produce two-step plans. Again, we prune any suboptimal plans from consideration. This procedure of alternating between expansion and pruning is repeated until the desired horizon is reached. Figure 20.3 demonstrates value iteration on the crying baby problem.

[6] This section describes a version of value iteration in terms of conditional plans and alpha vectors. For a version that only uses alpha vectors, see A. R. Cassandra, M. L. Littman, and N. L. Zhang, "Incremental Pruning: A Simple, Fast, Exact Method for Partially Observable Markov Decision Processes," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 1997.

We can construct all two-step plans for the crying baby problem. There are $3^3 = 27$ such plans.

The expected utility for each plan in belief space is plotted below. We find that two plans dominate all others. These dominating plans are the only ones that need be considered as subplans for optimal three-step plans.



Example 20.3. The expected utility over the belief space for all two-step plans for the crying baby problem (appendix F.7). The thick lines are optimal for some beliefs, whereas the thin lines are dominated.

```
function value_iteration(𝒫::POMDP, k_max)
    𝒮, 𝒜, R = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R
    plans = [ConditionalPlan(a) for a in 𝒜]
    Γ = [[R(s,a) for s in 𝒮] for a in 𝒜]
    plans, Γ = prune(plans, Γ)
    for k in 2:k_max
        plans, Γ = expand(plans, Γ, 𝒫)
        plans, Γ = prune(plans, Γ)
    end
    return (plans, Γ)
end

function solve(M::ValueIteration, 𝒫::POMDP)
    plans, Γ = value_iteration(𝒫, M.k_max)
    return LookaheadAlphaVectorPolicy(𝒫, Γ)
end
```

Algorithm 20.8. Value iteration for POMDPs, which finds the dominating h-step plans for a finite-horizon POMDP of horizon `k_max` by iteratively constructing optimal subplans. The `ValueIteration` structure is the same as what was defined in algorithm 7.8 in the context of MDPs.

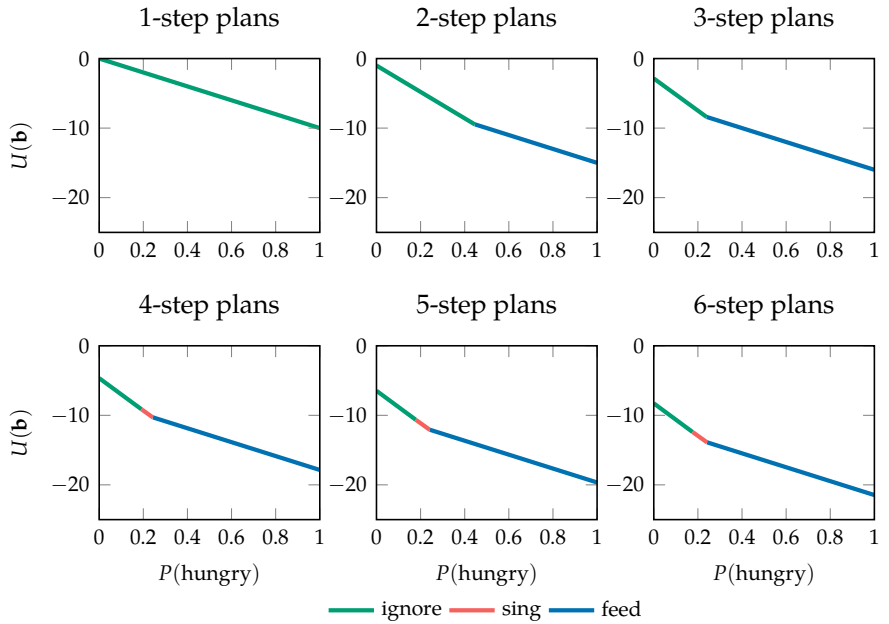


Figure 20.3. POMDP value iteration used to find the optimal value function for the crying baby problem to various horizons.

The expansion step (algorithm 20.9) in this process constructs all possible $k+1$-step plans from a set of k-step plans. New plans can be constructed using a new first action and all possible combinations of the k-step plans as subplans, as shown in figure 20.4. While plans can also be extended by adding additional actions to the ends of subplans, top-level expansion allows alpha vectors constructed for the k-step plans to be used to efficiently construct alpha vectors for the $(k+1)$-step plans.

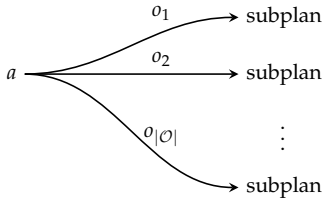


Figure 20.4. A $k+1$ step plan can be constructed using a new initial action leading to any combination of k-step subplans.

Computing the alpha vector associated with a plan π from a set of alpha vectors associated with its subplans can be done as follows. We use $\boldsymbol{\alpha}_o$ to represent the alpha vector associated with subplan $\pi(o)$. The alpha vector associated with π is then:

$$\alpha(s) = R(s, \pi()) + \gamma \sum_{s'} T(s' \mid s, \pi()) \sum_{o} O(o \mid \pi(), s') \alpha_o(s') \tag{20.17}$$

Even for relatively simple problems to shallow depths, computing alpha vectors from subplans in this way is much more efficient than computing them from scratch as in algorithm 20.2.

20.6 Linear Policies

As discussed in section 19.3, the belief state in a problem with linear-Gaussian dynamics can be represented by a Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}_b, \boldsymbol{\Sigma}_b)$. If the reward function is quadratic, then it can be shown that the optimal policy can be computed exactly offline using a process that is often called *linear-quadratic-Gaussian* (*LQG*) control. The optimal action is obtained in an identical manner as in section 7.8, but the $\boldsymbol{\mu}_b$ computed using the linear Gaussian filter is treated as the true state. With each observation, we simply use the filter to update our $\boldsymbol{\mu}_b$ and obtain an optimal action by multiplying $\boldsymbol{\mu}_b$ with the policy matrix from algorithm 7.11. Example 20.4 demonstrates this process.

20.7 Summary

- Exact solutions for POMDPs can typically only be obtained for finite-horizon discrete POMDPs.

```
function ConditionalPlan(𝒫::POMDP, a, plans)
    subplans = Dict(o⇒π for (o, π) in zip(𝒫.𝒪, plans))
    return ConditionalPlan(a, subplans)
end

function combine_lookahead(𝒫::POMDP, s, a, Γo)
    𝒮, 𝒪, T, O, R, γ = 𝒫.𝒮, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R, 𝒫.γ
    U′(s′,i) = sum(O(a,s′,o)*α[i] for (o,α) in zip(𝒪,Γo))
    return R(s,a) + γ*sum(T(s,a,s′)*U′(s′,i) for (i,s′) in enumerate(𝒮))
end

function combine_alphavector(𝒫::POMDP, a, Γo)
    return [combine_lookahead(𝒫, s, a, Γo) for s in 𝒫.𝒮]
end

function expand(plans, Γ, 𝒫)
    𝒮, 𝒜, 𝒪, T, O, R = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R
    plans′, Γ′ = [], []
    for a in 𝒜
        # iterate over all possible mappings from observations to plans
        for inds in product([eachindex(plans) for o in 𝒪]...)
            πo = plans[[inds...]]
            Γo = Γ[[inds...]]
            π = ConditionalPlan(𝒫, a, πo)
            α = combine_alphavector(𝒫, a, Γo)
            push!(plans′, π)
            push!(Γ′, α)
        end
    end
    return (plans′, Γ′)
end
```

Algorithm 20.9. The expansion step in value iteration, which constructs all $k+1$-step conditional plans and associated alpha vectors from a set of k-step conditional plans and alpha vectors. The way we combine alpha vectors of subplans follows equation (20.17).

Example 20.4. An example of an optimal policy used for a POMDP with linear Gaussian dynamics and quadratic reward.

Consider a satellite navigating in two dimensions, neglecting gravity, drag, or other external forces. The satellite can use its thrusters to accelerate in any direction with linear dynamics:

$$\begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} + \begin{bmatrix} \frac{1}{2}\Delta t^2 & 0 \\ 0 & \frac{1}{2}\Delta t^2 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \begin{bmatrix} \ddot{x} \\ \ddot{y} \end{bmatrix} + \boldsymbol{\epsilon}$$

where Δt is the duration of a time-step and $\boldsymbol{\epsilon}$ is zero-mean Gaussian noise with covariance $\Delta t/20\mathbf{I}$.

We seek to place the satellite in its orbital slot at the origin, while minimizing fuel use. Our quadratic reward function is:

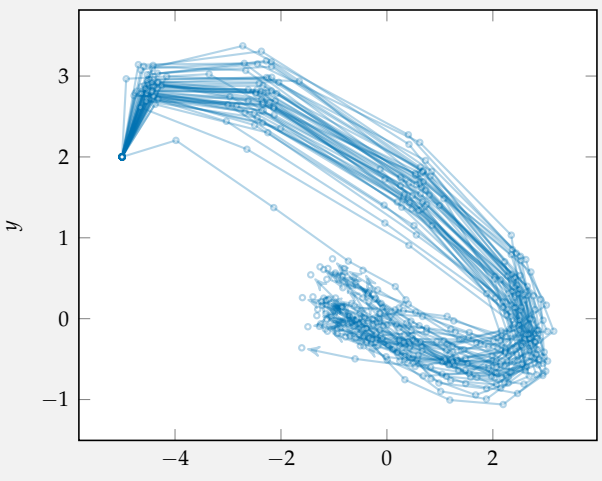
$$R(\mathbf{s}, \mathbf{a}) = -\mathbf{s}^\top \begin{bmatrix} \mathbf{I}_{2\times 2} & \mathbf{0}_{2\times 2} \\ \mathbf{0}_{2\times 2} & \mathbf{0}_{2\times 2} \end{bmatrix} \mathbf{s} - 2\mathbf{a}^\top \mathbf{a}$$

The satellite's sensors only estimate its position with the linear observation:

$$\mathbf{o} = \begin{bmatrix} \mathbf{I}_{2\times 2} & \mathbf{0}_{2\times 2} \end{bmatrix} \mathbf{s} + \boldsymbol{\varepsilon}$$

where $\boldsymbol{\varepsilon}$ is zero-mean Gaussian noise with covariance $\Delta t/10\mathbf{I}$.

Below are 50 trajectories from 10-step rollouts using the optimal policy for $\Delta t = 1$ and a Kalman filter to track the belief. In each case, the satellite was started at $\mathbf{s} = \boldsymbol{\mu}_b = [-5, 2, 0, 1]$ with $\boldsymbol{\Sigma}_b = [\mathbf{I}\,\mathbf{0};\ \mathbf{0}\,0.25\mathbf{I}]$.



- Policies for these problems can be represented as conditional plans, which are trees that describe the actions to take based on the observations.
- Alpha vectors contain the expected utility when starting from different states and following a particular conditional plan.
- Alpha vectors can also serve as an alternative representation of a POMDP policy.
- POMDP value iteration can avoid the computational burden of enumerating all conditional plans by iteratively computing subplans and pruning those that are suboptimal.
- Linear-Gaussian problems with quadratic reward can be solved exactly using methods very similar to those derived for the fully observable case.

20.8 Exercises

Exercise 20.1. Can every POMDP be framed as an MDP?

Solution: Yes. Any POMDP can equivalently be viewed as a belief-state MDP whose state space is the space of beliefs in the POMDP, whose action space is the same as that of the POMDP, and whose transition function is given by equation (20.1).

Exercise 20.2. What are the alpha vectors for the one-step crying baby problem (appendix F.7)? Are all actions dominant?

Solution: There are three one-step conditional plans, one for each action, resulting in three alpha vectors. The optimal one-step policy must choose between these actions given the current belief. The one-step alpha vectors for a POMDP can be obtained from the optimal one-step belief value function:

$$U^*(b) = \max_a \sum_s b(s) R(s, a)$$

Feeding the baby yields an expected reward:

$$\begin{aligned} &R(\text{hungry}, \text{feed}) P(\text{hungry}) + R(\text{sated}, \text{feed}) P(\text{sated}) \\ &\quad = -15 P(\text{hungry}) - 5(1 - P(\text{hungry})) \\ &\quad = -10 P(\text{hungry}) - 5 \end{aligned}$$

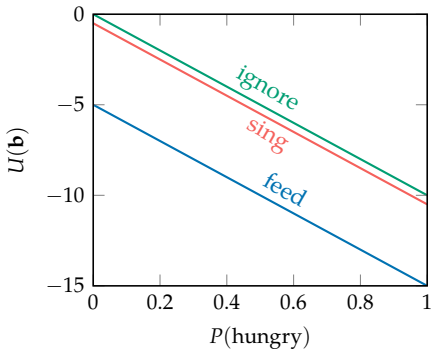
Singing to the baby yields an expected reward:

$$\begin{aligned} &R(\text{hungry}, \text{sing})P(\text{hungry}) + R(\text{sated}, \text{sing})P(\text{sated}) \\ &\quad = -10.5P(\text{hungry}) - 0.5(1 - P(\text{hungry})) \\ &\quad = -10P(\text{hungry}) - 0.5 \end{aligned}$$

Ignoring the baby yields an expected reward:

$$\begin{aligned} &R(\text{hungry}, \text{ignore})P(\text{hungry}) + R(\text{sated}, \text{ignore})P(\text{sated}) \\ &\quad = -10P(\text{hungry}) \end{aligned}$$

The expected reward for each action is plotted below over the belief space:



We find that under a one-step horizon it is never optimal to feed or sing to the baby. The ignore action is dominant.

Exercise 20.3. Why does the implementation of value iteration in algorithm 20.8 call expand plans in algorithm 20.9 rather than evaluating the plan in algorithm 20.2 to obtain alpha vectors for each new conditional plan?

Solution: The plan evaluation method applies equation (20.8) recursively to evaluate the expected utility for a conditional plan. Conditional plans grow very large as the horizon increases. POMDP value iteration can save computation by using the alpha vectors for the subplans from the previous iteration:

$$U^{\pi}(s) = R(s, \pi()) + \gamma \left[\sum_{s'} T(s' \mid s, \pi()) \sum_{o} O(o \mid \pi(), s') \alpha_{s'}^{\pi(o)} \right]$$

Exercise 20.4. Does the number of conditional plans increase faster with the number of actions or with the number of observations?

Solution: Recall that there are $|\mathcal{A}|^{(|\mathcal{O}|^h-1)/(|\mathcal{O}|-1)}$ possible h-step plans. Exponential growth (n^x) is faster than polynomial growth (x^n), and we have better-than exponential growth in $|\mathcal{O}|$ and polynomial growth in $|\mathcal{A}|$. The number of plans thus increases faster with respect to the number of observations. To demonstrate, let us use $|\mathcal{A}| = 3$, $|\mathcal{O}| = 3$, and $h = 3$ as a baseline. The baseline has 1,594,323 plans. Incrementing the number of actions results in 67,108,864 plans, whereas incrementing the number of observations results in 10,460,353,203 plans.

Exercise 20.5. Suppose we have a patient, and we are unsure whether or not they have a particular disease. We do have three diagnostic tests, each with different probabilities that they will correctly indicate whether or not the disease is present. While the patient is in our office, we have the option to administer potentially multiple diagnostic tests in sequence. We observe the outcome of each diagnostic test immediately. In addition, we can repeat any diagnostic test multiple times, with the outcomes of all tests being conditionally independent of each other given the presence or absence of the disease. When we are done with the tests, we decide whether to treat the disease or send the patient home without treatment. Explain how you would define the various components of a POMDP formulation.

Solution: We have three states:

1. s_{healthy}: the patient does not have the disease
2. s_{disease}: the patient has the disease
3. s_{terminal}: the interaction is over (terminal state)

We have five actions:

1. a_1: administer test 1
2. a_2: administer test 2
3. a_3: administer test 3
4. a_{treat}: administer treatment and send patient home
5. a_{stop}: send patient home without treatment

We have three observations:

1. o_{healthy}: the outcome of the test (if administered) indicates the patient is healthy
2. o_{disease}: the outcome of the test (if administered) indicates the patient has the disease
3. o_{terminal}: a test was not administered

The transition model would be deterministic with:

$$T(s' \mid s, a) = \begin{cases} 1 & \text{if } a \in \{a_{\text{treat}}, a_{\text{stop}}\} \wedge s' = s_{\text{stop}} \\ 1 & \text{if } s = s' \\ 0 & \text{otherwise} \end{cases}$$

The reward function would be a function of the cost of administering treatment and each test as well as the cost of not treating the disease if it is indeed present. The reward available from s_{terminal} is 0. The observation model assigns probabilities to correct and incorrect observations of the disease state as a result of a diagnostic test from one of the nonterminal states. The initial belief would assign our prior probability to whether or not the patient has the disease, with zero probability assigned to the terminal state.

Exercise 20.6. Why might we want to perform the same test multiple times in the previous exercise?

Solution: Depending on the probability of incorrect results, we may want to perform the same test multiple times to improve our confidence in whether the patient has the disease or is healthy. The results of the tests are independent given the disease state.

Exercise 20.7. Suppose we have three alpha vectors: $[1, 0]$, $[0, 1]$, and $[\theta, \theta]$, for some constant θ. Under what conditions on θ can we prune alpha vectors?

Solution: We can prune alpha vectors if $\theta < 0.5$ or $\theta > 1$. If $\theta < 0.5$, then $[\theta, \theta]$ is dominated by the other two alpha vectors. If $\theta > 1$, then $[\theta, \theta]$ dominates the other two alpha vectors.

Exercise 20.8. We have $\mathit{Gamma} = \{[1, 0], [0, 1]\}$ and $\boldsymbol{\alpha} = [0.7, 0.7]$. What belief $\mathbf{b}$ maximizes the utility gap δ as defined by the linear program in equation (20.16)?

Solution: The alpha vectors in Γ are shown in blue and the alpha vector $\boldsymbol{\alpha}$ is shown in red. We care only about the region where $0.3 \leq b_2 \leq 0.7$ where $\boldsymbol{\alpha}$ dominates the alpha vectors in Γ; in other words, where the red line is above the blue lines. The point where the gap between the red line and the maximum of the blue lines occurs at $b_2 = 0.5$ with a gap of $\delta = 0.2$. Hence, the belief that maximizes this gap is $\mathbf{b} = [0.5, 0.5]$.