22 *Online Belief State Planning*

Online methods determine the optimal policy by planning from the current belief state. The belief space reachable from the current state is typically small compared with the full belief space. As introduced in the fully observable context, many online methods use variations of tree-based search up to some horizon.[1] Various strategies can be used to try to avoid the exponential computational growth with the tree depth. Although online methods require more computation per decision step during execution than offline approaches, online methods are sometimes easier to apply to high-dimensional problems.

[1] A survey is provided by S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa, ''Online Planning Algorithms for POMDPs,'' *Journal of Artificial Intelligence Research*, vol. 32, pp. 663–704, 2008.

22.1 *Lookahead with Rollouts*

Section 9.2 introduced lookahead with rollouts as an online method in fully observed problems. The algorithm provided in that section can be used directly for partially observed problems. It uses a function for randomly sampling the next state, which corresponds to a belief state in the context of partial observability. This function was already introduced in algorithm 21.11. Because we can use a generative model rather than an explicit model for transitions, rewards, and observations, we can accommodate problems with high dimensional state and observation spaces.

22.2 *Forward Search*

We can apply the forward search strategy from algorithm 9.2 to partially observed problems without modification. The difference between MDPs and POMDPs is encapsulated by the one-step lookahead, which branches on actions and observations as shown in figure 22.1. The value of taking action a from belief b can be

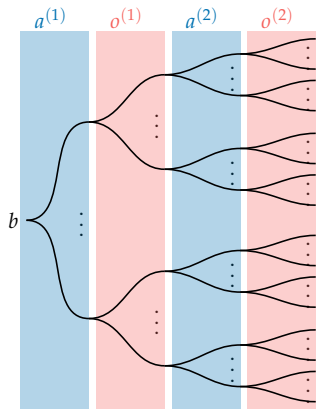


Figure 22.1. Forward search searches the action-observation-belief graph to an arbitrary finite depth in order to select the action that produces the highest expected reward. This figure shows a search to depth 2.

defined recursively to some depth d:

$$Q_d(b,a) = \begin{cases} R(b,a) + \gamma \sum_o P(o \mid b,a) U_{d-1}(\text{Update}(b,a,o)) & \text{if } d > 0 \\ U(b) & \text{otherwise} \end{cases} \tag{22.1}$$

where $U_d(b) = \max_a Q_d(b,a)$. When $d = 0$, we have reached maximum depth and return the utility using the approximate value function $U(b)$. This approximate value function may be obtained from one of the methods discussed in the previous chapter, heuristically chosen, or estimated from one or more rollouts. When $d > 0$, we continue to search deeper, recursing down another level. Example 22.1 shows how to combine QMDP with forward search for the machine replacement problem. Example 22.2 demonstrates forward search on the crying baby problem.

Example 22.1. An example of applying forward search to a POMDP problem

Consider applying forward search to the machine replacement problem. We can first obtain an approximate value function through QMDP (algorithm 21.2). We can then construct a `ForwardSearch` object, which was originally defined in algorithm 9.2. The call to `lookahead` within that function will use the one defined for POMDPs in algorithm 20.5. The following code applies forward search to the problem $\mathcal{P}$ from belief state $[0.5, 0.2, 0.3]$ to depth 5 using our estimate of the utility obtained from QMDP at the leaf nodes.

```
k_max = 10 # maximum number of iterations of QMDP
πQMDP = solve(QMDP(k_max), 𝒫)
d = 5 # depth
U(b) = utility(πQMDP, b)
π = ForwardSearch(𝒫, d, U)
π([0.5,0.2,0.3])
```

The computation associated with the recursion in equation (22.1) grows exponentially with depth, $O(|\mathcal{A}|^d |\mathcal{O}|^d)$. Hence, forward search is generally limited to a relatively shallow depth. To go deeper, we can limit the action or observation branching. For example, if we have some domain knowledge, we may restrict the action set either at the root or further down the tree. For the observation branching, we may restrict our consideration to a small set of likely observations—or even just the most likely observation.[2] Branching can be avoided entirely by adopting the open loop or hindsight optimization methods described in section 9.9.3 with states sampled from the current belief.

[2] R. Platt Jr., R. Tedrake, L. P. Kaelbling, and T. Lozano-Pérez, "Belief Space Planning Assuming Maximum Likelihood Observations," in *Robotics: Science and Systems*, 2010.

Example 22.2. An example of forward search applied to the crying baby problem.

Consider forward search in the crying baby problem with an approximate value function given by the alpha vectors $[-3.7, -15]$ and $[-2, -21]$. Running forward search to depth 2 from the initial belief $b = [0.5, 0.5]$ proceeds as follows:

$$\begin{aligned}
Q_2(b, a_{\text{feed}}) &= R(b, a_{\text{feed}}) + \gamma(P(\text{crying} \mid b, \text{feed})U_1([1.0, 0.0]) \\
&\qquad + P(\text{quiet} \mid b, \text{feed})U_1([1.0, 0.0])) \\
&= -10 + 0.9(0.1 \times -3.2157 + 0.9 \times -3.2157) \\
&= -12.894 \\
Q_2(b, a_{\text{ignore}}) &= R(b, a_{\text{ignore}}) + \gamma(P(\text{crying} \mid b, \text{ignore})U_1([0.093, 0.907]) \\
&\qquad + P(\text{quiet} \mid b, \text{ignore})U_1([0.786, 0.214])) \\
&= -5 + 0.9(0.485 \times -15.872 + 0.515 \times -7.779) \\
&= -15.534 \\
Q_2(b, a_{\text{sing}}) &= R(b, a_{\text{sing}}) + \gamma(P(\text{crying} \mid b, \text{sing})U_1([0.0, 1.0]) \\
&\qquad + P(\text{quiet} \mid b, \text{sing})U_1([0.891, 0.109])) \\
&= -5.5 + 0.9(0.495 \times -16.8 + 0.505 \times -5.543) \\
&= -15.503
\end{aligned}$$

Recall that feeding the baby always results in a sated baby ($b = [1, 0]$), and singing to the baby ensures that it only cries if it is hungry ($b = [0, 1]$). Each U_1 value must be evaluated by recursively looking ahead and eventually evaluating the approximate value function. The policy predicts that feeding the baby will result in the highest expected utility, so it recommends that action.

22.3 Branch and Bound

The *branch and bound* technique originally introduced in the context of MDPs can be extended to POMDPs. The same algorithm in section 9.4 can be used without modification (see example 22.3), relying on the appropriate look-ahead implementation to account for the observations and updating beliefs. The efficiency of the algorithm still depends upon the quality of the upper and lower bounds for pruning.

Although we can use domain-specific heuristics for the upper and lower bounds as we did in the fully observed case, we can alternatively use one of the methods introduced in the previous chapter for discrete state spaces. For example, we can use the fast informed bound for the upper bound and point-based value iteration for the lower bound. So long as the lower bound $\underline{U}$ and upper bound $\overline{Q}$ are true lower and upper bounds, the result of the branch and bound algorithm will be the same as the forward search algorithm with $\underline{U}$ as the approximate value function.

Example 22.3. An application of branch and bound to the crying baby problem.

In this example, we apply branch and bound to the crying baby problem with a depth of 5. The upper bound comes from the fast informed bound and the lower bound comes from point-based value iteration. We compute the action from belief $[0.4, 0.6]$.

```
k_max = 10 # maximum number of iterations for bounds
πFIB = solve(FastInformedBound(k_max), 𝒫)
d = 5 # depth
Ulo(b) = utility(πFIB, b)
B = [[p, 1 - p] for p in 0.0:0.2:1.0]
πPBVI = solve(PointBasedValueIteration(B, k_max), 𝒫)
Uhi(b) = utility(πPBVI, b)
Qhi(b,a) = lookahead(𝒫, Uhi, b, a)
π = BranchAndBound(𝒫, d, Ulo, Qhi)
π([0.4,0.6])
```

22.4 Sparse Sampling

Forward search sums over all possible observations, resulting in a runtime exponential in $|\mathcal{O}|$. As introduced in section 9.5, we can can use sampling to avoid exhaustive summation. We can generate m observations for each action and then

compute

$$Q_d(b,a) = \begin{cases} \frac{1}{m}\sum_{i=1}^{m}\left(r_a^{(i)} + \gamma U_{d-1}\left(\text{Update}(b,a,o_a^{(i)})\right)\right) & \text{if } d > 0 \\ U(\text{Update}(b,a,o_a^{(i)})) & \text{otherwise} \end{cases} \tag{22.2}$$

where $r_a^{(i)}$ and $o_a^{(i)}$ are the ith sampled observation and reward associated with action a from belief b. We may use algorithm 9.4 without modification. The resulting complexity is $O(|\mathcal{A}|^d m^d)$.

22.5 Monte Carlo Tree Search

The *Monte Carlo tree search* approach for MDPs can be extended to POMDPs, though we cannot use the same exact implementation.[3] The input to the algorithm is a belief state b, depth d, exploration factor c, and a rollout policy π.[4] The main difference between the POMDP algorithm (algorithm 22.1) and the MDP algorithm is that the counts and values are associated with *histories* instead of states. A history is a sequence of past actions and observations. For example, if we have two actions a_1 and a_2 and two observations o_1 and o_2, then a possible history could be the sequence $h = a_1 o_2 a_2 o_2 a_1 o_1$. During the execution of the algorithm, we update the value estimates $Q(h,a)$ and counts $N(h,a)$ for a set of history-action pairs.[5]

The histories associated with Q and N may be organized in a tree similar to the one in figure 22.2. The root node represents the empty history starting from the initial belief state b. During the execution of the algorithm, the tree structure expands. The layers of the tree alternate between action nodes and observation nodes. Associated with each action node are values $Q(h,a)$ and $N(h,a)$, where the history is determined by the path from the root node. As with the MDP version, when searching down the tree, the algorithm takes the action that maximizes

$$Q(h,a) + c\sqrt{\frac{\log N(h)}{N(h,a)}} \tag{22.3}$$

where $N(h) = \sum_a N(h,a)$ is the total visit count for history h and c is an exploration parameter. Importantly, c augments the value of actions that are unexplored and underexplored, thus representing the relative tradeoff between exploration and exploitation.

[3] Silver and Veness present a Monte Carlo tree search algorithm for POMDPs called *Partially Observable Monte Carlo Planning* (*POMCP*) and show its convergence. D. Silver and J. Veness, "Monte-Carlo Planning in Large POMDPs," in *Advances in Neural Information Processing Systems* (*NIPS*), 2010.

[4] Monte Carlo tree search can be implemented with a POMDP rollout policy that operates on beliefs or on an MDP rollout policy that operates on states. Random policies are commonly used.

[5] There are many variations of the basic algorithm introduced here, including some that incorporate aspects of double progressive widening, discussed in section 9.6. Z. N. Sunberg and M. J. Kochenderfer, "Online Algorithms for POMDPs with Continuous State, Action, and Observation Spaces," in *International Conference on Automated Planning and Scheduling* (*ICAPS*), 2018.

```
struct HistoryMonteCarloTreeSearch
    𝒫     # problem
    N     # visit counts
    Q     # action value estimates
    d     # depth
    k_max # number of simulations
    c     # exploration constant
    π     # rollout policy
end

function explore(π::HistoryMonteCarloTreeSearch, h)
    𝒜, N, Q, c = π.𝒫.𝒜, π.N, π.Q, π.c
    Nh = sum(get(N, (h,a), 0) for a in 𝒜)
    return _argmax(a->Q[(h,a)] + c*bonus(N[(h,a)], Nh), 𝒜)
end

function simulate(π::HistoryMonteCarloTreeSearch, s, h, d)
    if d ≤ 0
        return 0.0
    end
    𝒫, N, Q, c = π.𝒫, π.N, π.Q, π.c
    𝒮, 𝒜, TRO, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.TRO, 𝒫.γ
    if !haskey(N, (h, first(𝒜)))
        for a in 𝒜
            N[(h,a)] = 0
            Q[(h,a)] = 0.0
        end
        b = [s == s′ ? 1.0 : 0.0 for s′ in 𝒮]
        return rollout(𝒫, b, π.π, d)
    end
    a = explore(π, h)
    s′, r, o = TRO(s,a)
    q = r + γ*simulate(π, s′, vcat(h, (a,o)), d-1)
    N[(h,a)] += 1
    Q[(h,a)] += (q-Q[(h,a)])/N[(h,a)]
    return q
end

function (π::HistoryMonteCarloTreeSearch)(b, h=[])
    for i in 1:π.k_max
        s = rand(SetCategorical(π.𝒫.𝒮, b))
        simulate(π, s, h, π.d)
    end
    return _argmax(a->π.Q[(h,a)], π.𝒫.𝒜)
end
```

Algorithm 22.1. Monte Carlo tree search for POMDPs from belief `b`. The initial history `h` is optional. This implementation is similar to the one in algorithm 9.5.

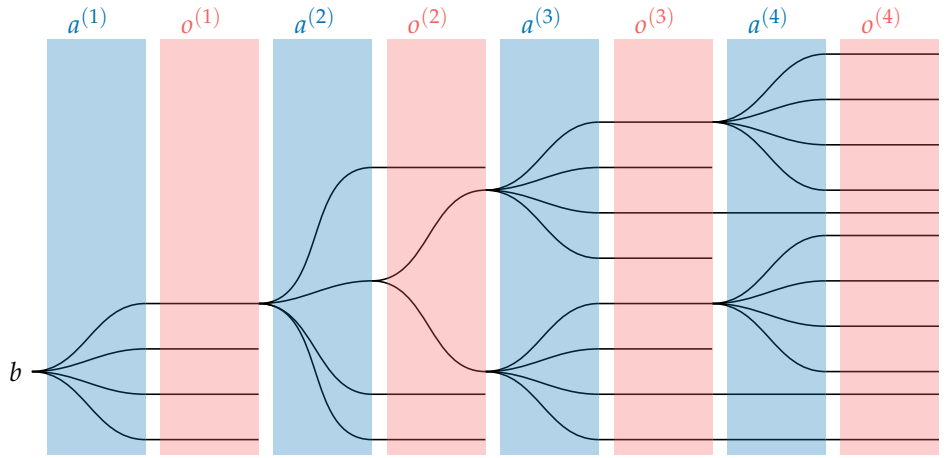


Figure 22.2. A search tree containing all histories covered when running Monte Carlo tree search with 100 samples on the machine replacement problem. This search used an exploration constant $c = 0.5$, a maximum depth $d = 5$, and a uniform random rollout policy. The initial belief is certainty in a fully working system. Monte Carlo tree search is able to avoid certain actions and instead focus samples on more promising paths.

As with the MDP version, the Monte Carlo tree search algorithm is an anytime algorithm. The loop in algorithm 22.1 can be terminated at any time, and the best solution found up to that point will be returned. With a sufficient number of iterations, the algorithm converges to the optimal action.

Prior knowledge can be incorporated into Monte Carlo tree search in how we initialize N and Q. Our implementation uses zero, but other choices are possible, including having the initialization of the action values be a function of history. The rollout policy can also incorporate prior knowledge, with better rollout policies tending to produce better value estimates.

The algorithm does not need to be reinitialized with each decision. The *history tree* and associated counts and value estimates can be maintained between calls. The observation node associated with the selected action and actual observation becomes the root node at the next time step.

22.6 Determinized Sparse Tree Search

Determinized sparse tree search strives to reduce the overall amount of sampling in both sparse sampling and Monte Carlo tree search by making the observation resulting from performing an action deterministic.[6] It does so by building a *determinized belief tree* from a special particle belief representation to form a sparse

[6] Ye et al. present a determinized sparse tree search algorithm for POMDPs called *Determinized Sparse Partially Observable Tree*, or *DESPOT* N. Ye, A. Somani, D. Hsu, and W.S. Lee, "DESPOT: Online POMDP Planning with Regularization," *Journal of Artificial Intelligence Research*, vol. 58, pp. 231–266, 2017. In addition, the algorithm includes branch and bound, heuristic search, and regularization techniques.

approximation of the true belief tree. Each particle refers to one of m *scenarios*, each of depth d. A scenario represents a fixed history that the particle will follow for any given sequence of actions $a^{(1)}, a^{(2)}, \ldots, a^{(d)}$. Every distinct action sequence produces a distinct history under a particular scenario.[7] This determinization reduces the size of the search tree to $O(|\mathcal{A}|^d m)$. An example of a history is given in example 22.4. A determinized tree is shown in figure 22.3.

[7] A similar idea was discussed in section 9.9.3 and is related to the PEGASUS algorithm mentioned in section 11.1.

Example 22.4. An example of a history and a scenario in the context of determinized sparse tree search.

Suppose we have two states s_1 and s_2, two actions a_1 and a_2, and two observations o_1 and o_2. A possible history of depth $d = 2$ for the particle with initial state s_2 is the sequence $h = s_2 a_1 o_2 s_1 a_2 o_1$. If this history is used as a scenario, then this history is returned every time the belief tree is traversed from the initial state with the action sequence $a^{(1)} = a_1$ and $a^{(2)} = a_2$.

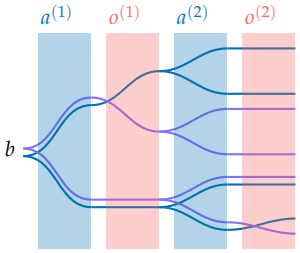


Figure 22.3. A determinized sparse search tree with two scenarios, shown in blue and purple. The line traces show the possible paths for each scenario under different action sequences.

A search tree with m scenarios up to depth d can be fully specified by a compact $m \times d$ *determinizing matrix* $\mathbf{\Phi}$ containing probability masses. The element Φ_{ij} contains the information needed for a particle following the ith scenario at depth j to identify its successor state and observation. Specifically, Φ_{ij} is a uniformly distributed random number that can generate the successor pair (s', o) from a state-action pair (s, a), following the distribution $P(s', o \mid s, a) = T(s' \mid s, a)O(o \mid a, s')$. We can generate a determinizing matrix by filling it with values sampled uniformly between 0 and 1.

Beliefs are represented as vectors of belief particles. Each belief particle ϕ contains a state s and indices i and j into the determinizing matrix $\mathbf{\Phi}$ corresponding to a scenario i and current depth j. Given a particular action a, Φ_{ij} is used to deterministically transition to successor state s' and observation o. The successor particle $\phi' = (s', i, j + 1)$ receives s' as its state and increments j by 1. Example 22.5 demonstrates this tree traversal process. The particle belief representation is implemented in algorithm 22.2 and is used in forward search in algorithm 22.3.

22.7 Gap Heuristic Search

Similar to the offline heuristic search presented in section 21.8, we can use the gap between the upper and lower bounds to guide our search toward beliefs that have uncertainty in their associated value and as an indication of when we can stop exploring. The gap $\epsilon(b)$ at a belief b is the difference between the upper bound

Example 22.5. Determinized sparse tree search uses a matrix to make tree traversal deterministic for a given particle.

Suppose we generate a determinizing matrix $\mathbf{\Phi}$ for a problem with 4 histories up to depth 3:

$$\mathbf{\Phi} = \begin{bmatrix} 0.393 & 0.056 & 0.369 \\ 0.313 & 0.749 & 0.273 \\ 0.078 & 0.262 & 0.009 \\ 0.969 & 0.598 & 0.095 \end{bmatrix}$$

Suppose we take action a_3 in state s_2 when at depth 2 while following history 3. The corresponding belief particle is $\phi = (2, 3, 2)$, and the determinizing value in $\mathbf{\Phi}$ is $\Phi_{3,2} = 0.262$.

The deterministic successor action and observation are given by iterating over all successor state-observation pairs and accumulating their transition probabilities. We begin with $p = 0$ and evaluate $s' = s_1, o = o_1$. Suppose we get $T(s_1 \mid s_2, a_3)O(o_1 \mid a_3, s_1) = 0.1$. We increase p to 0.1, which is less than $\Phi_{3,2}$, so we continue.

Next we evaluate $s' = s_1, o = o_2$. Suppose we get $T(s_1 \mid s_2, a_3)O(o_2 \mid a_3, s_2) = 0.17$. We increase p to 0.27, which is greater than $\Phi_{3,2}$. We thus deterministically proceed to $s' = s_1, o = o_2$ as our successor state, resulting in a new particle $\phi' = (1, 3, 3)$.

```
struct DeterminizedParticle
    s # state
    i # scenario index
    j # depth index
end

function successor(𝒫, Φ, ϕ, a)
    𝒮, 𝒪, T, O = 𝒫.𝒮, 𝒫.𝒪, 𝒫.T, 𝒫.O
    p = 0.0
    for (s′, o) in product(𝒮, 𝒪)
        p += T(ϕ.s, a, s′) * O(a, s′, o)
        if p ≥ Φ[ϕ.i, ϕ.j]
            return (s′, o)
        end
    end
    return last(𝒮), last(𝒪)
end

function possible_observations(𝒫, Φ, b, a)
    𝒪 = []
    for ϕ in b
        s′, o = successor(𝒫, Φ, ϕ, a)
        push!(𝒪, o)
    end
    return unique(𝒪)
end

function update(b, Φ, 𝒫, a, o)
    b′ = []
    for ϕ in b
        s′, o′ = successor(𝒫, Φ, ϕ, a)
        if o == o′
            push!(b′, DeterminizedParticle(s′, ϕ.i, ϕ.j + 1))
        end
    end
    return b′
end
```

Algorithm 22.2. The determinized particle belief update produces a determinized sparse tree search for POMDPs. Each belief `b` consists of particles `ϕ` that each encode a particular scenario and depth along the scenario. Their scenario's trajectory is determinized through a matrix `Φ` containing random values in $[0, 1]$. Each particle `ϕ` represents a particular scenario `i` at a particular depth `j`, referring to the `i`th row and `j`th column of `Φ`.

```
struct DeterminizedSparseTreeSearch
    𝒫 # problem
    d # depth
    Φ # m×d determinizing matrix
    U # value function to use at leaf nodes
end

function determinized_sparse_tree_search(𝒫, b, d, Φ, U)
    𝒮, 𝒜, 𝒪, T, R, O, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.T, 𝒫.R, 𝒫.O, 𝒫.γ
    if d == 0
        return (a=nothing, u=U(b))
    end
    best = (a=nothing, u=-Inf)
    for a in 𝒜
        u = sum(R(ϕ.s, a) for ϕ in b) / length(b)
        for o in possible_observations(𝒫, Φ, b, a)
            Poba = sum(sum(O(a,s′,o)*T(ϕ.s,a,s′) for s′ in 𝒮)
                       for ϕ in b) / length(b)
            b′ = update(b, Φ, 𝒫, a, o)
            u′ = determinized_sparse_tree_search(𝒫,b′,d-1,Φ,U).u
            u += γ*Poba*u′
        end
        if u > best.u
            best = (a=a, u=u)
        end
    end
    return best
end

function determized_approximate_belief(b, 𝒫, m)
    particles = []
    for i in 1:m
        s = rand(SetCategorical(𝒫.𝒮, b))
        push!(particles, DeterminizedParticle(s, i, 1))
    end
    return particles
end

function (π::DeterminizedSparseTreeSearch)(b)
    particles = determized_approximate_belief(b, π.𝒫, size(π.Φ,1))
    return determinized_sparse_tree_search(π.𝒫,particles,π.d,π.Φ,π.U).a
end
```

Algorithm 22.3. An implementation of determinized sparse tree search, a modification of forward search, for POMDPs. The policy takes a belief `b` in the form of a vector of probabilities, which is approximated by a vector of determized particles by `determized_approximate_belief`.

and lower bound values $\epsilon(b) = \overline{U}(b) - \underline{U}(b)$. Search algorithms with the gap heuristic select the observation that maximizes the gap because they are more likely to benefit from a belief backup. Actions are often selected according to a lookahead using an approximate value function.

The initial upper and lower bound values used in heuristic search play an important role in the algorithm's performance. Algorithm 22.4 uses the best-action best-state upper bound from equation (21.2). It uses a rollout policy for the lower bound $\underline{U}(b)$. A rollout, of course, is not guaranteed to produce a lower bound because it is based on a single trial up to a fixed depth. As the number of samples of increases, then it will converge to a true lower bound. Many other forms of upper and lower bounds exist that can provide faster convergence at the cost of run time and implementation complexity. For example, using the fast informed bound (algorithm 21.3) for the upper bound can improve exploration and help reduce the gap. For the lower bound, we can use a problem-specific rollout policy to better guide the search. Example 22.6 demonstrates this process.

22.8 Summary

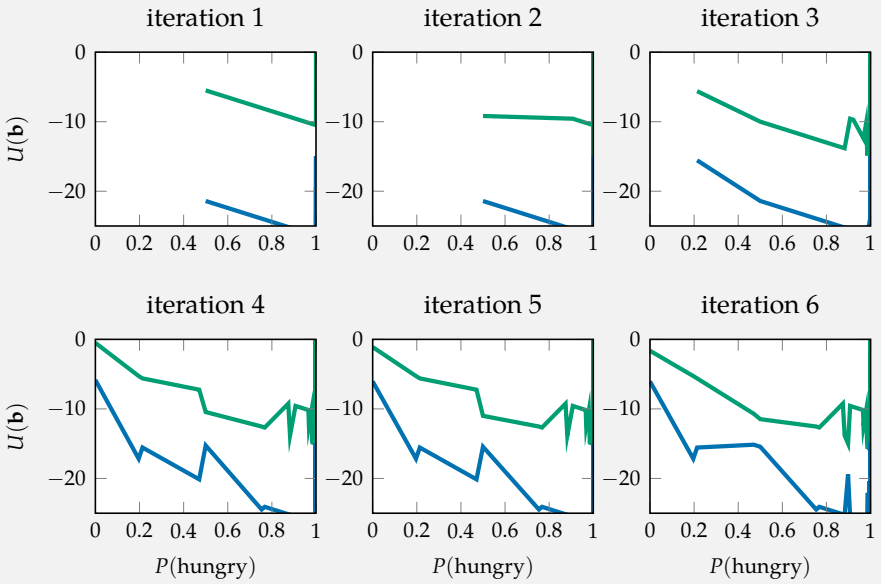
- A simple online strategy is to perform a one-step lookahead, which considers each action taken from the current belief and estimates its expected value using an approximate value function.
- Forward search is a generalization of lookahead to arbitrary horizons, which can lead to better policies but its computational complexity grows exponentially with the horizon.
- Branch and bound is a more efficient version of forward search that can avoid searching certain paths by using upper and lower bounds on the value function.
- Sparse sampling is an approximation method that can reduce the computational burden of iterating over the space of all possible observations.
- Monte Carlo tree search can be adapted to POMDPs by operating over histories rather than states.
- Determinized sparse tree search uses a special form of particle belief that ensures observations are determinized, greatly reducing the search tree.

```
struct GapHeuristicSearch
    𝒫     # problem
    Uhi   # upper bound on value function
    Ulo   # lower bound on value function
    π     # rollout policy
    δ     # gap threshold
    k_max # maximum number of simulations
    d_max # maximum depth
end

function heuristic_search(π::GapHeuristicSearch, b, d)
    𝒫, Uhi, Ulo, δ = π.𝒫, π.Uhi, π.Ulo, π.δ
    𝒮, 𝒜, 𝒪, R, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.R, 𝒫.γ
    B = Dict((a,o)⇒update(b,𝒫,a,o) for (a,o) in product(𝒜,𝒪))
    B = merge(B, Dict(()⇒copy(b)))
    Rmax = maximum(R(s,a) for (s,a) in product(𝒮,𝒜))
    for (ao, b′) in B
        if !haskey(Uhi, b′)
            Uhi[b′], Ulo[b′] = Rmax/(1.0-γ), rollout(𝒫,b′,π.π,d)
        end
    end
    if d == 0 || Uhi[b] - Ulo[b] ≤ δ
        return
    end
    a = _argmax(a -> lookahead(𝒫,b′->Uhi[b′],b,a), 𝒜)
    o = _argmax(o -> Uhi[B[(a, o)]] - Ulo[B[(a, o)]], 𝒪)
    b′ = update(b,𝒫,a,o)
    heuristic_search(π,b′,d-1)
    Uhi[b] = maximum(lookahead(𝒫,b′->Uhi[b′],b,a) for a in 𝒜)
    Ulo[b] = maximum(lookahead(𝒫,b′->Ulo[b′],b,a) for a in 𝒜)
end

function (π::GapHeuristicSearch)(b)
    Uhi, Ulo, k_max, d_max, δ = π.Uhi, π.Ulo, π.k_max, π.d_max, M.δ
    for i in 1:k_max
        heuristic_search(π, b, d_max)
        if Uhi[b] - Ulo[b] < δ
            break
        end
    end
    return _argmax(a -> lookahead(𝒫,b′->Ulo[b′],b,a), 𝒫.𝒜)
end
```

Algorithm 22.4. An implementation of heuristic search that uses bounds, a gap criterion, and a rollout policy as its heuristic. The upper and lower bounds are maintained by `Uhi` and `Ulo`, respectively. The rollout policy `π` must be a function that returns an action given a belief `π(b)`. At belief `b`, the gap is `Uhi[b] - Ulo[b]`. Exploration stops when the gap is smaller than the threshold `δ` or the maximum depth `d_max` is reached. A maximum number of iterations `k_max` is allotted to search.

Example 22.6. An example illustrating the use of heuristic search upper and lower bounds for the crying baby problem over iterations of heuristic search.

Consider applying online heuristic search to the crying baby problem. On each iteration of heuristic search, it explores the belief space following the bounds and improves. Below we show six iterations of heuristic search with an initial belief `b` of $[0.5, 0.5]$, maximum depth `d_max` of 10, and a random action rollout policy. In each iteration, the upper bound is shown in greenand the lower bound is shown in blue.



We see that not all beliefs are explored in the first iteration. The iteration starts at $[0.5, 0.5]$ and heuristic search explores to the right first, towards belief $[0.0, 1.0]$. As the iterations progress, it slowly explores more towards $[1.0, 0.0]$. The jagged bounds are due to some beliefs not being re-explored based on the action and observation selection. There are computational savings due to pruning of the search tree. On the bottom row, we see that it has explored many of the beliefs once, but the bounds are still loose. Heuristic search guides itself towards the goal of reducing the gap.

- Heuristic search intelligently selects action-observation pairs to explore regions with a high gap between the upper and lower bounds on the value function that it maintains.

22.9 Exercises

Exercise 22.1. Suppose we have $\mathcal{A} = \{a^1, a^2\}$ and a belief $\mathbf{b} = [0.5, 0.5]$. The reward is always 1. The observation function is given by $P(o^1 \mid a^1) = 0.8$ and $P(o^1 \mid a^2) = 0.4$. We have an approximate value function represented by an alpha vector $\boldsymbol{\alpha} = [-3, 4]$. With $\gamma = 0.9$, use forward search to a depth of 1 to compute $U(\mathbf{b})$. Use the following updated beliefs in the calculation:

a	o	Update$(\mathbf{b}, a, o)$
a^1	o^1	$[0.3, 0.7]$
a^2	o^1	$[0.2, 0.8]$
a^1	o^2	$[0.5, 0.5]$
a^2	o^2	$[0.8, 0.2]$

Solution: We need to calculate the action value function at depth 1 according to equation (22.1):

$$Q_d(\mathbf{b}, a) = R(\mathbf{b}, a) + \gamma \sum_o P(o \mid b, a) U_{d-1}(\text{Update}(\mathbf{b}, a, o))$$

We first calculate the utility for the updated beliefs.

$$\begin{aligned}
U_0(\text{Update}(\mathbf{b}, a^1, o^1)) &= \boldsymbol{\alpha}^\top \mathbf{b}' = 0.3 \times -3 + 0.7 \times 4 = 1.9 \\
U_0(\text{Update}(\mathbf{b}, a^2, o^1)) &= 0.2 \times -3 + 0.8 \times 4 = 2.6 \\
U_0(\text{Update}(\mathbf{b}, a^1, o^2)) &= 0.5 \times -3 + 0.5 \times 4 = 0.5 \\
U_0(\text{Update}(\mathbf{b}, a^2, o^2)) &= 0.8 \times -3 + 0.2 \times 4 = -1.6
\end{aligned}$$

Second, we compute the action value function for both actions:

$$\begin{aligned}
Q_1(\mathbf{b}, a^1) &= 1 + 0.9((P(o^1 \mid \mathbf{b}, a^1) U_0(\text{Update}(\mathbf{b}, a^1, o^1)) + (P(o^2 \mid \mathbf{b}, a^1) U_0(\text{Update}(\mathbf{b}, a^1, o^2))) \\
&= 1 + 0.9(0.8 \times 1.9 + 0.2 \times 0.5) = 2.458 \\
Q_1(\mathbf{b}, a^2) &= 1 + 0.9((P(o^1 \mid \mathbf{b}, a^2) U_0(\text{Update}(\mathbf{b}, a^2, o^1)) + (P(o^2 \mid \mathbf{b}, a^2) U_0(\text{Update}(\mathbf{b}, a^2, o^2))) \\
&= 1 + 0.9(0.4 \times 2.6 + 0.6 \times -1.6) = 1.072
\end{aligned}$$

Finally, we have $U_1(\mathbf{b}) = \max_a Q_1(\mathbf{b}, a) = 2.458$

Exercise 22.2. Using the following trajectory samples, compute the action value function for belief $\mathbf{b}$ and actions a^1 and a^2 based on sparse sampling to depth 1. Use the following updated beliefs, discount factor $\gamma = 0.9$, and approximate value function represented by an alpha vector $\boldsymbol{\alpha} = [10, 1]$.

a	o	r	Update$(\mathbf{b}, a, o)$
1	1	0	$[0.47, 0.53]$
2	1	1	$[0.22, 0.78]$
1	2	1	$[0.49, 0.51]$
2	1	1	$[0.22, 0.78]$
2	2	1	$[0.32, 0.68]$
1	2	1	$[0.49, 0.51]$

Solution: We first calculate the utility for the updated beliefs:

a	o	r	Update$(\mathbf{b}, a, o_a)$	$U_0(\text{Update}(\mathbf{b}, a, o))$
1	1	0	$[0.47, 0.53]$	5.23
2	1	1	$[0.22, 0.78]$	2.98
1	2	1	$[0.49, 0.51]$	5.41
2	1	1	$[0.22, 0.78]$	2.98
2	2	1	$[0.32, 0.68]$	3.88
1	2	1	$[0.49, 0.51]$	5.41

Then, we can compute the action value function over all actions using equation (22.2):

$$Q_1(\mathbf{b}, a^1) = \frac{1}{3}(0 + 1 + 1 + 0.9(5.23 + 5.41 + 5.41)) = 5.48$$
$$Q_1(\mathbf{b}, a^2) = \frac{1}{3}(1 + 1 + 1 + 0.9(2.98 + 2.98 + 3.88)) = 3.95$$

Exercise 22.3. Consider example 22.5. Give the new particle associated with $\phi = (1, 4, 2)$. Suppose we take action a_3, and we have the following transition functions:

$$T(s_2 \mid s_1, a_3) = 0.4 \qquad O(o_1 \mid s_1, a_3) = 0.6$$
$$T(s_3 \mid s_1, a_3) = 0.45 \qquad O(o_2 \mid s_1, a_3) = 0.5$$

Solution: From the determinizing matrix, our determinizing value is $\Phi_{4,2} = 0.598$ and we are in state s_1. Then, we calculate p:

$$p \leftarrow T(s_2 \mid s_1, a_3)O(o_1 \mid s_1, a_3) = 0.4 \times 0.6 = 0.24 \tag{22.4}$$
$$p \leftarrow p + T(s_2 \mid s_1, a_3)O(o_2 \mid s_1, a_3) = 0.24 + 0.4 \times 0.5 = 0.44 \tag{22.5}$$
$$p \leftarrow p + T(s_3 \mid s_1, a_3)O(o_1 \mid s_1, a_3) = 0.44 + 0.45 \times 0.6 = 0.71 \tag{22.6}$$

We stop our iteration because $p > 0.598$. Thus, from our final iteration, we proceed to (s_3, o_1).