23 Controller Abstractions

This chapter introduces controller representations for POMDP policies, which allow policies to maintain their own internal state. These representations can improve scalability over previous methods that enumerate over belief points. This chapter presents algorithms that construct controllers using policy iteration, mathematical programming, and gradient ascent.

23.1 Controllers

A *controller* is a policy representation that maintains its own internal state. This controller is represented as a graph consisting of a finite set of nodes X.[1] The active *node* changes as actions are taken and new observations are made. Having a finite set of nodes makes these controllers more computationally tractable than belief-point methods that must consider the reachable belief space.

[1] Such a policy representation is also called a *finite state controller*. We will endeavor to refer to the controller states as nodes rather than states to reduce ambiguity with the environment state.

Actions are selected according to an *action distribution* $\psi(a \mid x)$ that depends on the current node. When selecting an action, in addition to transitioning to an unobserved s' and receiving an observation o, the control state also advances according to a *successor distribution* $\eta(x' \mid x, a, o)$. Figure 23.1 shows how these distributions are used as a controller policy is followed. Algorithm 23.1 provides an implementation, and example 23.1 shows a controller for the crying baby problem.

Controllers generalize conditional plans, which were introduced in section 20.2. Conditional plans represent policies as trees, with each node deterministically assigning an action and each edge specifying a unique successor node. Controllers represent policies as directed graphs, and actions may have stochastic transitions to multiple successor nodes. Example 23.2 compares these two representations.

$$\begin{tikzcd}
x_1 \arrow[rr, "\eta(x' \mid x, a, o)"] \arrow[d, "\psi(a \mid x)"] & & x_2 \arrow[rr, "\eta(x' \mid x, a, o)"] \arrow[d, "\psi(a \mid x)"] & & x_3 \arrow[rr, "\eta(x' \mid x, a, o)"] \arrow[d, "\psi(a \mid x)"] & & x_4 \\
a_1 \arrow[urr, bend right] & & a_2 \arrow[urr, bend right] & & a_3 \arrow[urr, bend right] & & \\
& o_1 \arrow[uur, bend right] & & o_2 \arrow[uur, bend right] & & o_3 \arrow[uur, bend right] &
\end{tikzcd}$$

Figure 23.1. In a controller representation, the action is sampled from the action selection distribution. This action, and the subsequent observation it produces, are used alongside the previous node x to produce the successor node x'.

We can construct a simple controller for the crying baby problem (appendix F.7). This policy is shown below as a graph with two nodes, x^1 and x^2. When in x^1, the controller always ignores the baby. When in x^2, the controller always feeds the baby. If the baby cries, we always transition to x^2, and if the baby is quiet, we always transition to x^1.

$$\begin{tikzcd}
x^1 \arrow[loop left, "o = \text{quiet}"] \arrow[r, bend left, "o = \text{crying}"] & x^2 \arrow[l, bend left, "o = \text{quiet}"] \arrow[loop right, "o = \text{crying}"]
\end{tikzcd}$$

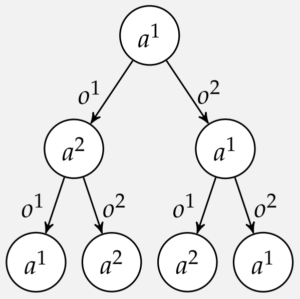
$\psi(\text{ignore} \mid x^1) = 1$ $\psi(\text{feed} \mid x^2) = 1$

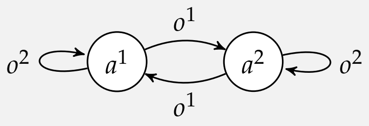
Example 23.1. A two-node example controller for the crying baby problem. This compact representation captures a straightforward solution to the crying baby problem, namely to react immediately to the most recent observation.

```
mutable struct ControllerPolicy
    𝒫 # problem
    X # set of controller nodes
    ψ # action selection distribution
    η # successor selection distribution
end

function (π::ControllerPolicy)(x)
    𝒜, ψ = π.𝒫.𝒜, π.ψ
    dist = [ψ[x, a] for a in 𝒜]
    return rand(SetCategorical(𝒜, dist))
end

function update(π::ControllerPolicy, x, a, o)
    X, η = π.X, π.η
    dist = [η[x, a, o, x′] for x′ in X]
    return rand(SetCategorical(X, dist))
end
```

Algorithm 23.1. A finite state controller policy representation for a POMDP `𝒫`. The nodes in `X` are an abstract representation of reachable beliefs. Actions and controller successor nodes are selected stochastically. Given a node `x`, actions are selected following the distribution `ψ`. The function `π(x)` implements this mechanism to stochastically select actions. After performing action `a` in node `x` and observing observation `o`, the successor is selected following the distribution `η`. The function `update` implements this mechanism to stochastically select successor nodes.

Consider a three-step conditional plan (below left) compared with the more general two-node finite state controller (below right) from example 23.1. In this case, actions and successors are selected deterministically. The deterministic action is marked in the center of a node, and the outgoing edges represent the deterministic successor nodes. This problem has two actions (a^1 and a^2) and two observations (o^1 and o^2).





$o^1 = \text{quiet}$

$o^2 = \text{crying}$

$a^1 = \text{ignore}$

$a^2 = \text{feed}$

The conditional plan performs action a^1 first, toggles the previously chosen action if it observes o^1, and preserves the previously chosen action if it observes o^2. The controller performs the same logic, with five fewer controller nodes. Moreover, the controller represents the described infinite horizon policy perfectly with only two nodes (compared to seven). The conditional plan cannot capture this infinite horizon policy because it would require a tree of infinite depth.

Example 23.2. A comparison of a simple conditional plan with a simple deterministic controller.

Controllers have several advantages over conditional plans. First, controllers can provide a more compact representation. The number of nodes in a conditional plan grows exponentially with depth, but this need not be the case with finite state controllers. The approximation methods from previous chapters might also not be as efficient because they must maintain a large set of beliefs and corresponding alpha vectors. Controllers can be much more compact, considering infinitely many possible reachable beliefs with a small, finite number of nodes. Another advantage of controllers is that they do not require that a belief be maintained.[2] The controller itself selects a new node based on each observation, rather than relying on a belief update, which can be expensive for some domains.

[2] Each controller node corresponds to a subset of the belief space. A controller transitions between these subsets that together cover the reachable belief space.

The utility when following a controller can be computed by forming a product MDP whose state space is $X \times \mathcal{S}$. The value of being in state s with node x active is:

$$U(x,s) = \sum_a \psi(a \mid x)\left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \sum_o O(o \mid a,s') \sum_{x'} \eta(x' \mid x,a,o) U(x',s')\right) \tag{23.1}$$

Policy evaluation involves solving the system of linear equations in equation (23.1). Alternatively, we can apply iterative policy evaluation as shown in algorithm 23.2.

```julia
function utility(π::ControllerPolicy, U, x, s)
    𝒮, 𝒜, 𝒪 = π.𝒫.𝒮, π.𝒫.𝒜, π.𝒫.𝒪
    T, O, R, γ = π.𝒫.T, π.𝒫.O, π.𝒫.R, π.𝒫.γ
    X, ψ, η = π.X, π.ψ, π.η
    U′(a,s′,o) = sum(η[x,a,o,x′]*U[x′,s′] for x′ in X)
    U′(a,s′) = T(s,a,s′)*sum(O(a,s′,o)*U′(a,s′,o) for o in 𝒪)
    U′(a) = R(s,a) + γ*sum(U′(a,s′) for s′ in 𝒮)
    return sum(ψ[x,a]*U′(a) for a in 𝒜)
end

function iterative_policy_evaluation(π::ControllerPolicy, k_max)
    𝒮, X = π.𝒫.𝒮, π.X
    U = Dict((x, s) => 0.0 for x in X, s in 𝒮)
    for k in 1:k_max
        U = Dict((x, s) => utility(π, U, x, s) for x in X, s in 𝒮)
    end
    return U
end
```

Algorithm 23.2. An algorithm for performing iterative policy evaluation to compute the utility of a finite state controller `π` with `k_max` iterations. The utility function performs a single step evaluation for the current controller node `x` and state `s` following equation (23.1). This algorithm was adapted from algorithm 7.3, which applies to MDPs.

If a belief is known, then the current value is

$$U(x,b) = \sum_s b(s) U(x,s) \tag{23.2}$$

We can think of $U(x, s)$ as defining a set of alpha vectors, one for each node x in X. Each alpha vector $\boldsymbol{\alpha}_x$ is defined by $\alpha_x(s) = U(x, s)$. The current value for a given alpha vector is $U(x, b) = \mathbf{b}^\top \boldsymbol{\alpha}_x$.

Given a controller and an initial belief, we can select an initial node by maximizing:

$$x^* = \arg\max_x U(x, b) = \arg\max_x \mathbf{b}^\top \boldsymbol{\alpha}_x \tag{23.3}$$

23.2 *Policy Iteration*

Section 20.5 showed how to incrementally add nodes in a conditional plan to arrive at optimal finite-horizon policy (algorithm 20.8). This section shows how to incrementally add nodes to a controller to optimize for infinite horizon problems. Although the policy representation is different, the version of policy iteration for partially observable problems introduced in this section[3] has some similarities with the policy iteration algorithm for fully observed problems (section 7.4).

Policy iteration (algorithm 23.3) begins with any initial controller, and then iterates between policy evaluation and policy improvement. In policy evaluation, we evaluate the utilities $U(x, s)$ by solving equation (23.1). In policy improvement, we introduce new nodes to our controller. Specifically, we introduce a new node x' for every combination of deterministic action assignments $\psi(a_i \mid x') = 1$ and deterministic successor selection distributions $\eta(x \mid x', a, o)$. This process adds $|\mathcal{A}||X^{(k)}|^{|\mathcal{O}|}$ new controller nodes to the set of nodes $X^{(k)}$ at iteration k.[4] An improvement step is demonstrated in example 23.3.

Policy improvement cannot worsen the expected value of the controller policy. The value of any nodes in $X^{(k)}$ remain unchanged, as they and their reachable successor nodes remain unchanged. It is guaranteed that if $X^{(k)}$ is not an optimal controller, then at least one of the new nodes introduced in policy improvement will have better expected values for some states, and thus the overall controller must be improved.

Many of the nodes added during policy improvement tend to not improve the policy. Pruning is conducted after policy evaluation to eliminate unnecessary nodes. Pruning does not degrade the optimal value function of the controller and can help reduce the exponential growth in nodes that comes with the improvement step.

[3] The policy iteration method given here was given by E. A. Hansen, "Solving POMDPs by Searching in Policy Space," in *Conference on Uncertainty in Artificial Intelligence* (*UAI*), 1998.

[4] Adding all possible combinations is often not feasible. An alternative algorithm called *bounded policy iteration* adds only one node. P. Poupart and C. Boutilier, "Bounded Finite State Controllers," in *Advances in Neural Information Processing Systems* (*NIPS*), 2003. Algorithms can also add a number in between. *Monte Carlo value iteration*, for example, adds $O(n|\mathcal{A}||X^{(k)}|)$ new nodes at each iteration k, where n is a parameter. H. Bai, D. Hsu, W. S. Lee, and V. A. Ngo, "Monte Carlo Value Iteration for Continuous-State POMDPs," in *International Workshop on the Algorithmic Foundations of Robotics* (*WAFR*), 2011.

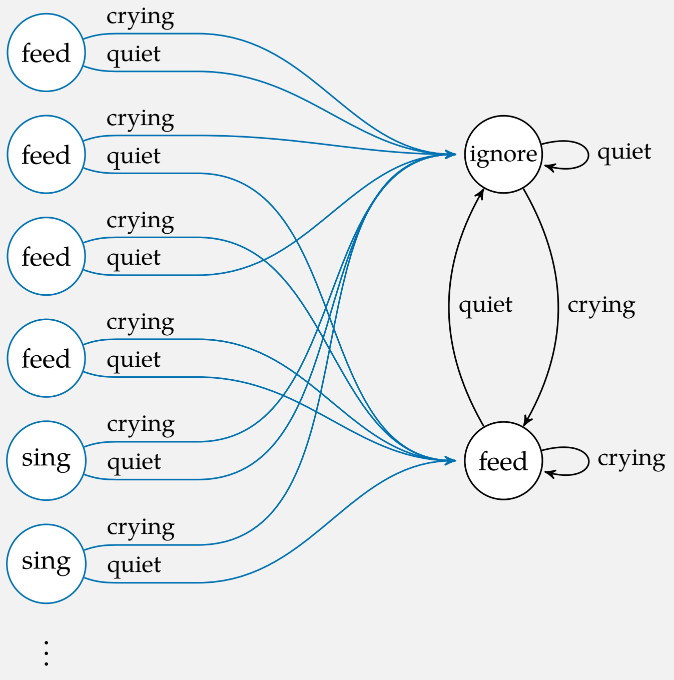
```
struct ControllerPolicyIteration
    k_max    # number of iterations
    eval_max # number of evaluation iterations
end

function solve(M::ControllerPolicyIteration, 𝒫::POMDP)
    𝒜, 𝒪, k_max, eval_max = 𝒫.𝒜, 𝒫.𝒪, M.k_max, M.eval_max
    X = [1]
    ψ = Dict((x, a) => 1.0 / length(𝒜) for x in X, a in 𝒜)
    η = Dict((x, a, o, x′) => 1.0 for x in X, a in 𝒜, o in 𝒪, x′ in X)
    π = ControllerPolicy(𝒫, X, ψ, η)
    for i in 1:k_max
        prevX = copy(π.X)
        U = iterative_policy_evaluation(π, eval_max)
        policy_improvement!(π, U, prevX)
        prune!(π, U, prevX)
    end
    return π
end

function policy_improvement!(π::ControllerPolicy, U, prevX)
    𝒮, 𝒜, 𝒪 = π.𝒫.𝒮, π.𝒫.𝒜, π.𝒫.𝒪
    X, ψ, η = π.X, π.ψ, π.η
    repeatX𝒪 = fill(X, length(𝒪))
    assign𝒜X′ = vec(collect(product(𝒜, repeatX𝒪...)))
    for ax′ in assign𝒜X′
        x, a = maximum(X) + 1, ax′[1]
        push!(X, x)
        successor(o) = ax′[findfirst(isequal(o), 𝒪) + 1]
        U′(o,s′) = U[successor(o), s′]
        for s in 𝒮
            U[x, s] = lookahead(π.𝒫, U′, s, a)
        end
        for a′ in 𝒜
            ψ[x, a′] = a′ == a ? 1.0 : 0.0
            for (o, x′) in product(𝒪, prevX)
                η[x, a′, o, x′] = x′ == successor(o) ? 1.0 : 0.0
            end
        end
    end
    for (x, a, o, x′) in product(X, 𝒜, 𝒪, X)
        if !haskey(η, (x, a, o, x′))
            η[x, a, o, x′] = 0.0
        end
    end
end
```

Algorithm 23.3. Policy iteration for a POMDP 𝒫 given a fixed number of iterations `k_max` and number of policy evaluation iterations `eval_max`. The algorithm iteratively applies policy evaluation from algorithm 23.2 and policy improvement. Pruning is implemented in algorithm 23.4.

We can apply policy improvement to the crying baby controller from example 23.1. The actions are $\mathcal{A} = \{\text{feed}, \text{sing}, \text{ignore}\}$ and observations are $\mathcal{O} = \{\text{crying}, \text{quiet}\}$. The policy improvement backup step results in $|\mathcal{A}||X^{(1)}|^{|\mathcal{O}|} = 3 \times 2^2 = 12$ new nodes. The new controller policy has nodes $\{x^1, \ldots, x^{14}\}$ and distributions as follows:

Node	Action	Successors (for all a below)
x^3	$\psi(\text{feed} \mid x^3) = 1$	$\eta(x^1 \mid x^3, a, \text{crying}) = \eta(x^1 \mid x^3, a, \text{quiet}) = 1$
x^4	$\psi(\text{feed} \mid x^4) = 1$	$\eta(x^1 \mid x^4, a, \text{crying}) = \eta(x^2 \mid x^4, a, \text{quiet}) = 1$
x^5	$\psi(\text{feed} \mid x^5) = 1$	$\eta(x^2 \mid x^5, a, \text{crying}) = \eta(x^1 \mid x^5, a, \text{quiet}) = 1$
x^6	$\psi(\text{feed} \mid x^6) = 1$	$\eta(x^2 \mid x^6, a, \text{crying}) = \eta(x^2 \mid x^6, a, \text{quiet}) = 1$
x^7	$\psi(\text{sing} \mid x^7) = 1$	$\eta(x^1 \mid x^7, a, \text{crying}) = \eta(x^1 \mid x^7, a, \text{quiet}) = 1$
x^8	$\psi(\text{sing} \mid x^8) = 1$	$\eta(x^1 \mid x^8, a, \text{crying}) = \eta(x^2 \mid x^8, a, \text{quiet}) = 1$
$\vdots$	$\vdots$	$\vdots$

We have the following controller, with new nodes in blue and the original two nodes in black:



Example 23.3. An example illustrating an improvement step as part of policy iteration on the crying baby problem with a controller policy representation.

We prune any new nodes that are identical to existing nodes. We also prune any new nodes that are *dominated* by other nodes. A node x is dominated by another node x' when:

$$U(x, s) \leq U(x', s) \quad \text{for all } s \tag{23.4}$$

Existing nodes can be pruned as well. Whenever a new node dominates an existing node, we prune the existing node from the controller. Any transitions to the deleted node are instead rerouted to the dominating node.[5] Example 23.4 demonstrates evaluation, expansion, and pruning on the crying baby problem.

[5] This process is identical to pruning the new node instead, and updating the dominated node's action and successor links to those of the new node.

23.3 *Nonlinear Programming*

The policy improvement problem can be framed as a single large *nonlinear programming* formulation (algorithm 23.5) that involves simultaneously optimizing for ψ and η across all nodes.[6] This formulation allows general purpose solvers to be applied. The nonlinear programming method directly searches the space of controllers to maximize the utility of a given initial belief while satisfying the Bellman equation, equation (23.1). There is no alternating between policy evaluation and policy improvement steps, and the controller node count remains fixed.

[6] C. Amato, D. S. Bernstein, and S. Zilberstein, "Optimizing Fixed-Size Stochastic Controllers for POMDPs and Decentralized POMDPs," *Autonomous Agents and Multi-Agent Systems*, vol. 21, no. 3, pp. 293–320, 2010.

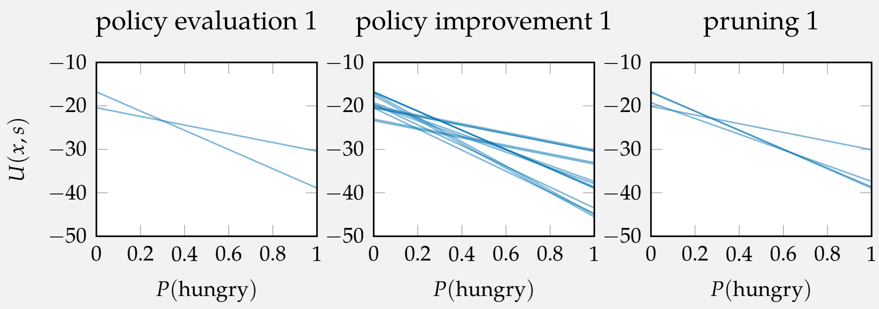
We use x^1 to denote the initial node corresponding to the given initial belief b. The optimization problem is then:

$$\begin{aligned}
\underset{U,\psi,\eta}{\text{maximize}} \quad & \sum_s b(s) U(x^1, s) \\
\text{subject to} \quad & U(x,s) = \sum_a \psi(a \mid x)\left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \sum_o O(o \mid a, s') \sum_{x'} \eta(x' \mid x,a,o) U(x',s')\right) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for all } x, s \\
& \psi(a \mid x) \geq 0 \quad \text{for all } x, a \\
& \sum_a \psi(a \mid x) = 1 \quad \text{for all } x \\
& \eta(x' \mid x,a,o) \geq 0 \quad \text{for all } x, a, o, x' \\
& \sum_{x'} \eta(x' \mid x,a,o) = 1 \quad \text{for all } x, a, o
\end{aligned} \tag{23.5}$$

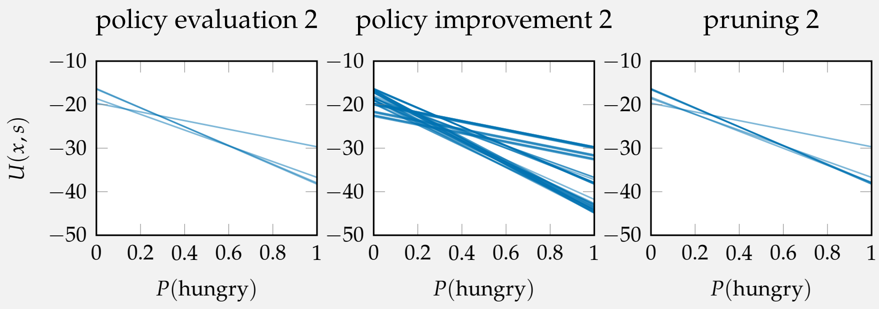
```
function prune!(π::ControllerPolicy, U, prevX)
    𝒮, 𝒜, 𝒪, X, ψ, η = π.𝒫.𝒮, π.𝒫.𝒜, π.𝒫.𝒪, π.X, π.ψ, π.η
    newX, removeX = setdiff(X, prevX), []
    # prune dominated from previous nodes
    dominated(x,x′) = all(U[x,s] ≤ U[x′,s] for s in 𝒮)
    for (x,x′) in product(prevX, newX)
        if x′ ∉ removeX && dominated(x, x′)
            for s in 𝒮
                U[x,s] = U[x′,s]
            end
            for a in 𝒜
                ψ[x,a] = ψ[x′,a]
                for (o,x′′) in product(𝒪, X)
                    η[x,a,o,x′′] = η[x′,a,o,x′′]
                end
            end
            push!(removeX, x′)
        end
    end
    # prune identical from previous nodes
    identical_action(x,x′) = all(ψ[x,a] ≈ ψ[x′,a] for a in 𝒜)
    identical_successor(x,x′) = all(η[x,a,o,x′′] ≈ η[x′,a,o,x′′]
            for a in 𝒜, o in 𝒪, x′′ in X)
    identical(x,x′) = identical_action(x,x′) && identical_successor(x,x′)
    for (x,x′) in product(prevX, newX)
        if x′ ∉ removeX && identical(x,x′)
            push!(removeX, x′)
        end
    end
    # prune dominated from new nodes
    for (x,x′) in product(X, newX)
        if x′ ∉ removeX && dominated(x′,x) && x ≠ x′
            push!(removeX, x′)
        end
    end
    # update controller
    π.X = setdiff(X, removeX)
    π.ψ = Dict(k => v for (k,v) in ψ if k[1] ∉ removeX)
    π.η = Dict(k => v for (k,v) in η if k[1] ∉ removeX)
end
```

Algorithm 23.4. The pruning step of policy iteration. It reduces the number of nodes in the current policy `π`, using the utilities `U` computed by policy evaluation, and the previous node list `prevX`. Its first step replaces any pointwise dominated previous nodes by their improved nodes, marking the redundant node as now dominated. The second step marks any newly added nodes that are identical to previous nodes. The third step marks any pointwise dominated new nodes. Finally, all marked nodes are pruned.

Recall example 23.3. Below we show the first iteration of policy iteration using same initial controller. It consists of the two main steps: policy evaluation (left) and policy improvement (center), as well as the optional pruning step (right).



The second iteration of policy iteration follows the same pattern:



The utility has greatly improved after the second iteration to near optimal values. We see that the prune step removes dominated and duplicate nodes from previous iterations as well as the current iteration's new nodes.

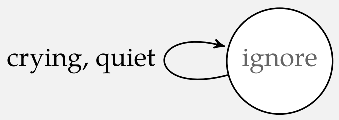
Example 23.4. An example of policy iteration, illustrating the evaluation, improvement, and pruning steps on the crying baby domain with a controller policy representation.

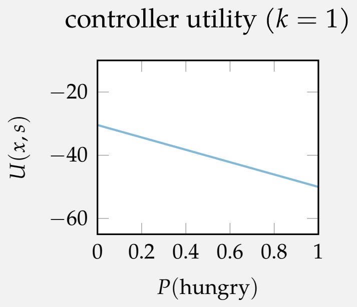
This problem can be written as a *quadratically constrained linear program* (*QCLP*), which can be solved efficiently using a dedicated solver. Example 23.5 demonstrates this approach.

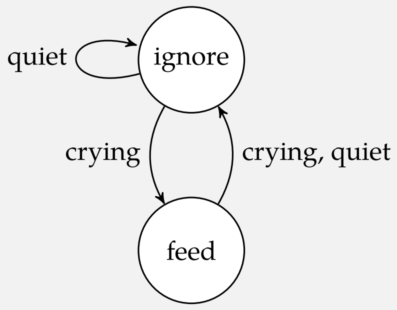
```
struct NonlinearProgramming
    b # initial belief
    ℓ # number of nodes
end

function tensorform(𝒫::POMDP)
    𝒮, 𝒜, 𝒪, R, T, O = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.R, 𝒫.T, 𝒫.O
    𝒮′ = eachindex(𝒮)
    𝒜′ = eachindex(𝒜)
    𝒪′ = eachindex(𝒪)
    R′ = [R(s,a) for s in 𝒮, a in 𝒜]
    T′ = [T(s,a,s′) for s in 𝒮, a in 𝒜, s′ in 𝒮]
    O′ = [O(a,s′,o) for a in 𝒜, s′ in 𝒮, o in 𝒪]
    return 𝒮′, 𝒜′, 𝒪′, R′, T′, O′
end

function solve(M::NonlinearProgramming, 𝒫::POMDP)
    x1, X = 1, collect(1:M.ℓ)
    𝒫, γ, b = 𝒫, 𝒫.γ, M.b
    𝒮, 𝒜, 𝒪, R, T, O = tensorform(𝒫)
    model = Model(Ipopt.Optimizer)
    @variable(model, U[X,𝒮])
    @variable(model, ψ[X,𝒜] ≥ 0)
    @variable(model, η[X,𝒜,𝒪,X] ≥ 0)
    @objective(model, Max, b⋅U[x1,:])
    @NLconstraint(model, [x=X,s=𝒮],
        U[x,s] == (sum(ψ[x,a]*(R[s,a] + γ*sum(T[s,a,s′]*sum(O[a,s′,o]
        *sum(η[x,a,o,x′]*U[x′,s′] for x′ in X)
        for o in 𝒪) for s′ in 𝒮)) for a in 𝒜)))
    @constraint(model, [x=X], sum(ψ[x,:]) == 1)
    @constraint(model, [x=X,a=𝒜,o=𝒪], sum(η[x,a,o,:]) == 1)
    optimize!(model)
    ψ′, η′ = value.(ψ), value.(η)
    return ControllerPolicy(𝒫, X,
        Dict((x, 𝒫.𝒜[a]) => ψ′[x, a] for x in X, a in 𝒜),
        Dict((x, 𝒫.𝒜[a], 𝒫.𝒪[o], x′) => η′[x, a, o, x′]
             for x in X, a in 𝒜, o in 𝒪, x′ in X))
end
```

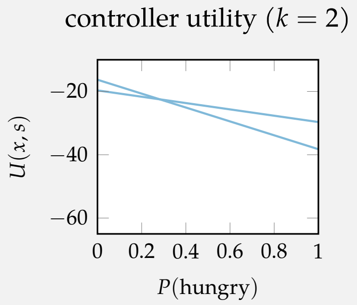
Algorithm 23.5. A nonlinear programming approach to compute the optimal fixed-size controller policy for POMDP $\mathcal{P}$ starting at initial belief b. The size of the finite state controller is specified by number of nodes ℓ.

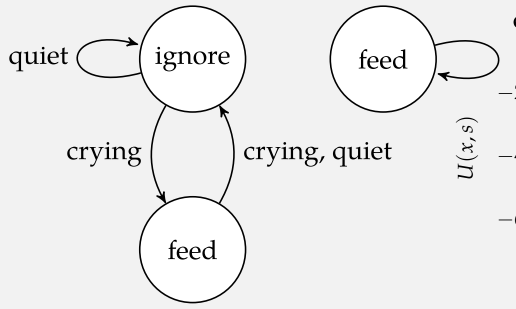
Below are optimal fixed-size controllers computed using nonlinear programming for the crying baby problem with $b_0 = [0.5, 0.5]$. The top node is x_1.

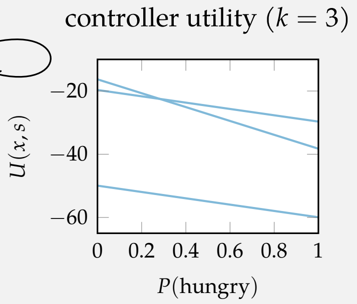












With $k = 1$, the optimal policy is to simply ignore forever. With $k = 2$, the optimal policy is to ignore until crying is observed, at which point the best action is to feed the child, and then return to ignoring. This policy is close to optimal for the infinite horizon crying baby POMDP. With $k = 3$, the optimal policy essentially remains unchanged from when $k = 2$.

Example 23.5. An example of the nonlinear programming algorithm for controllers with a fixed size of k set to 1, 2, and 3. Each row shows the policy and its corresponding utilities (alpha vectors) on the left and right, respectively. The stochastic controllers are shown as circles, with the most likely action in the middle. The outgoing edges show successor node selections given an observation. The stochasticity in node actions and successors are shown as opacity (more opaque is higher probability, more transparent is lower probability).

23.4 Gradient Ascent

A fixed-size controller policy can be iteratively improved using gradient ascent (covered in appendix A.11).[7] Though the gradient is challenging to compute, this opens up controller optimization to a wide variety of gradient-based optimization techniques. Algorithm 23.6 implements controller gradient ascent using algorithm 23.7.

[7] N. Meuleau, K.-E. Kim, L. P. Kaelbling, and A. R. Cassandra, "Solving POMDPs by Searching the Space of Finite Policies," in *Conference on Uncertainty in Artificial Intelligence* (*UAI*), 1999.

Consider an explicit description of the nonlinear problem from section 23.3. For initial belief b and an arbitrary initial controller node x^1, we seek to maximize:

$$\sum_s b(s) U(x^1, s) \tag{23.6}$$

with the utility $U(x,s)$ defined by the Bellman equation for all x and s:

$$U(x,s) = \sum_a \psi(a \mid x) \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \sum_o O(o \mid a, s') \sum_{x'} \eta(x' \mid x,a,o) U(x',s') \right) \tag{23.7}$$

In addition, ψ and η must also be proper probability distributions. To apply gradient ascent, it is more convenient to rewrite this problem using linear algebra.

We define the transition function with a controller, which has a state space $X \times \mathcal{S}$. For any fixed-size controller policy parameterized by $\boldsymbol{\theta} = (\psi, \eta)$, the transition matrix $\mathbf{T}_{\boldsymbol{\theta}} \in \mathbb{R}^{|X \times \mathcal{S}| \times |X \times \mathcal{S}|}$ is:

$$\mathbf{T}_{\boldsymbol{\theta}}((x,s),(x',s')) = \sum_a \psi(x,a) T(s,a,s') \sum_o O(a,s',o) \eta(x,a,o,x') \tag{23.8}$$

The reward for a parameteried policy is represented as a vector $\mathbf{r}_{\boldsymbol{\theta}} \in \mathbb{R}^{|X \times \mathcal{S}|}$:

$$\mathbf{r}_{\boldsymbol{\theta}}((x,s)) = \sum_a \psi(x,a) R(s,a) \tag{23.9}$$

The Bellman equation for utility $\mathbf{u}_{\boldsymbol{\theta}} \in \mathbb{R}^{|X \times \mathcal{S}|}$ is then:

$$\mathbf{u}_{\boldsymbol{\theta}} = \mathbf{r}_{\boldsymbol{\theta}} + \gamma \mathbf{T}_{\boldsymbol{\theta}} \mathbf{u}_{\boldsymbol{\theta}} \tag{23.10}$$

We can consider an initial node-belief vector $\boldsymbol{\beta} \in \mathbb{R}^{|X \times \mathcal{S}|}$ with $\boldsymbol{\beta}_{xs} = b(s)$ if $x = x^1$ and $\boldsymbol{\beta}_{xs} = 0$ otherwise. A utility vector $\mathbf{u}_{\boldsymbol{\theta}} \in \mathbb{R}^{|X \times \mathcal{S}|}$ also is defined over the nodes X and states $\mathcal{S}$ for any of these fixed-size parameterized controller policies $\boldsymbol{\theta} = (\psi, \eta)$. We now seek to maximize:

$$\boldsymbol{\beta}^\top \mathbf{u}_{\boldsymbol{\theta}} \tag{23.11}$$

We begin by rewriting equation (23.10):

$$\mathbf{u}_\theta = \mathbf{r}_\theta + \gamma \mathbf{T}_\theta \mathbf{u}_\theta \tag{23.12}$$
$$(\mathbf{I} - \gamma \mathbf{T}_\theta)\mathbf{u}_\theta = \mathbf{r}_\theta \tag{23.13}$$
$$\mathbf{u}_\theta = (\mathbf{I} - \gamma \mathbf{T}_\theta)^{-1}\mathbf{r}_\theta \tag{23.14}$$
$$\mathbf{u}_\theta = \mathbf{Z}^{-1}\mathbf{r}_\theta \tag{23.15}$$

with $\mathbf{Z} = \mathbf{I} - \gamma \mathbf{T}_\theta$ for convenience. In order to perform gradient ascent, we need to know the partial derivatives of equation (23.15) with respect to the policy parameters:

$$\frac{\partial \mathbf{u}_\theta}{\partial \theta} = \frac{\partial \mathbf{Z}^{-1}}{\partial \theta}\mathbf{r}_\theta + \mathbf{Z}^{-1}\frac{\partial \mathbf{r}_\theta}{\partial \theta} \tag{23.16}$$
$$= -\mathbf{Z}^{-1}\frac{\partial \mathbf{Z}}{\partial \theta}\mathbf{Z}^{-1}\mathbf{r}_\theta + \mathbf{Z}^{-1}\frac{\partial \mathbf{r}_\theta}{\partial \theta} \tag{23.17}$$
$$= \mathbf{Z}^{-1}\left(\frac{\partial \mathbf{r}_\theta}{\partial \theta} - \frac{\partial \mathbf{Z}}{\partial \theta}\mathbf{Z}^{-1}\mathbf{r}_\theta\right) \tag{23.18}$$

with $\partial\theta$ referring to both $\partial\psi(\hat{x},\hat{a})$ and $\partial\eta(\hat{x},\hat{a},\hat{o},\hat{x}')$ for convenience.

Computing the partial derivatives of $\mathbf{Z}$ and $\mathbf{r}_\theta$ results in four equations:

$$\frac{\partial \mathbf{r}_\theta((x,s))}{\partial \psi(\hat{x},\hat{a})} = \begin{cases} R(s,a) & \text{if } x = \hat{x} \\ 0 & \text{otherwise} \end{cases} \tag{23.19}$$
$$\frac{\partial \mathbf{r}_\theta((x,s))}{\partial \eta(\hat{x},\hat{a},\hat{o},\hat{x}')} = 0 \tag{23.20}$$
$$\frac{\partial \mathbf{Z}((x,s),(x',s'))}{\partial \psi(\hat{x},\hat{a})} = \begin{cases} -\gamma T(s,\hat{a},s')\sum_o O(\hat{a},s',o)\eta(\hat{x},\hat{a},o,x') & \text{if } x = \hat{x} \\ 0 & \text{otherwise} \end{cases} \tag{23.21}$$
$$\frac{\partial \mathbf{Z}((x,s),(x',s'))}{\partial \eta(\hat{x},\hat{a},\hat{o},\hat{x}')} = \begin{cases} -\gamma\psi(\hat{x},\hat{a})T(s,\hat{a},s')O(\hat{a},s',\hat{o})\eta(\hat{x},\hat{a},\hat{o},x') & \text{if } x = \hat{x} \text{ and } x' = \hat{x}' \\ 0 & \text{otherwise} \end{cases} \tag{23.22}$$

Finally, these four gradients are used in the utility gradients from equation (23.18):

$$\frac{\partial \mathbf{u}_\theta}{\partial \psi(\hat{x},\hat{a})} = \mathbf{Z}^{-1}\left(\frac{\partial \mathbf{r}_\theta}{\partial \psi(\hat{x},\hat{a})} - \frac{\partial \mathbf{Z}}{\partial \psi(\hat{x},\hat{a})}\mathbf{Z}^{-1}\mathbf{r}_\theta\right) \tag{23.23}$$
$$\frac{\partial \mathbf{u}_\theta}{\partial \eta(\hat{x},\hat{a},\hat{o},\hat{x}')} = \mathbf{Z}^{-1}\left(\frac{\partial \mathbf{r}_\theta}{\partial \eta(\hat{x},\hat{a},\hat{o},\hat{x}')} - \frac{\partial \mathbf{Z}}{\partial \eta(\hat{x},\hat{a},\hat{o},\hat{x}')}\mathbf{Z}^{-1}\mathbf{r}_\theta\right) \tag{23.24}$$

We finally can return to the original objective in equation (23.11). Controller gradient ascent starts with a fixed number of nodes in X and an arbitrary policy ψ and η. At iteration k, it updates these parameters as follows:

$$\psi^{k+1}(x,a) = \psi^k(x,a) + \alpha \boldsymbol{\beta}^\top \frac{\partial \mathbf{u}_{\boldsymbol{\theta}^k}}{\partial \psi^k(\hat{x},\hat{a})} \tag{23.25}$$

$$\eta^{k+1}(x,a,o,x') = \eta^k(x,a,o,x') + \alpha \boldsymbol{\beta}^\top \frac{\partial \mathbf{u}_{\boldsymbol{\theta}^k}}{\partial \eta^k(\hat{x},\hat{a},\hat{o},\hat{x}')} \tag{23.26}$$

with gradient step size $\alpha > 0$. The distributions ψ^{k+1} and η^{k+1} may no longer be valid after this update. They may need non-negativity to be enforced and to be normalized.[8] Example 23.6 demonstrates this process.

The optimization objective in equation (23.6) is not necessarily convex.[9] Hence, normal gradient ascent can converge to a local optimum depending on the initial controller. Adaptive gradient algorithms can be applied to help smooth and speed convergence.

[8] This procedure is more generally called projecting to the probability simplex.

[9] This objective is distinct from the utility $U(x,b) = \sum_s b(s)U(x,s)$, which is guaranteed to be piecewise linear and convex with respect to the belief state b as discussed in section 20.3.

23.5 Summary

- Controllers are a policy representation that do not rely on exploring or maintaining beliefs.
- Controllers consist of nodes, an action selection function, and a successor selection function.
- Nodes and the controller graph are abstract, however, they can be interpreted as sets of the countably infinite reachable beliefs; the value function for a node can also be interpreted as an alpha vector.
- Policy iteration iterates between policy evaluation that computes the utilities for each node, and policy improvement that adds new nodes, with the option to introduce pruning.
- Nonlinear programming reformulates finding the optimal fixed-sized controller as a general optimization problem , allowing for off-the-shelf solvers and techniques to be used.
- Controller gradient ascent climbs in the space of policies to improve the value function directly, benefiting from an explicit POMDP-based gradient step.

```
struct ControllerGradient
    b       # initial belief
    ℓ       # number of nodes
    α       # gradient step
    k_max   # maximum iterations
end

function solve(M::ControllerGradient, 𝒫::POMDP)
    𝒜, 𝒪, ℓ, k_max = 𝒫.𝒜, 𝒫.𝒪, M.ℓ, M.k_max
    X = collect(1:ℓ)
    ψ = Dict((x, a) => rand() for x in X, a in 𝒜)
    η = Dict((x, a, o, x′) => rand() for x in X, a in 𝒜, o in 𝒪, x′ in X)
    π = ControllerPolicy(𝒫, X, ψ, η)
    for i in 1:k_max
        improve!(π, M, 𝒫)
    end
    return π
end

function improve!(π::ControllerPolicy, M::ControllerGradient, 𝒫::POMDP)
    𝒮, 𝒜, 𝒪, X, x1, ψ, η = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, π.X, 1, π.ψ, π.η
    n, m, z, b, ℓ, α = length(𝒮), length(𝒜), length(𝒪), M.b, M.ℓ, M.α
    ∂U′∂ψ, ∂U′∂η = gradient(π, M, 𝒫)
    UIndex(x, s) = (s - 1) * ℓ + (x - 1) + 1
    E(U, x1, b) = sum(b[s]*U[UIndex(x1,s)] for s in 1:n)
    ψ′ = Dict((x, a) => 0.0 for x in X, a in 𝒜)
    η′ = Dict((x, a, o, x′) => 0.0 for x in X, a in 𝒜, o in 𝒪, x′ in X)
    for x in X
        ψ′x = [ψ[x, a] + α * E(∂U′∂ψ(x, a), x1, b) for a in 𝒜]
        ψ′x = project_to_simplex(ψ′x)
        for (aIndex, a) in enumerate(𝒜)
            ψ′[x, a] = ψ′x[aIndex]
        end
        for (a, o) in product(𝒜, 𝒪)
            η′x = [(η[x, a, o, x′] +
                    α * E(∂U′∂η(x, a, o, x′), x1, b)) for x′ in X]
            η′x = project_to_simplex(η′x)
            for (x′Index, x′) in enumerate(X)
                η′[x, a, o, x′] = η′x[x′Index]
            end
        end
    end
    π.ψ, π.η = ψ′, η′
end

function project_to_simplex(y)
    u = sort(copy(y), rev=true)
    i = maximum([j for j in eachindex(u)
                 if u[j] + (1 - sum(u[1:j])) / j > 0.0])
    δ = (1 - sum(u[j] for j = 1:i)) / i
    return [max(y[j] + δ, 0.0) for j in eachindex(u)]
end
```

Algorithm 23.6. An implementation of a controller gradient ascent algorithm for POMDP 𝒫 at initial belief `b`. The controller itself has a fixed-size of ℓ nodes. It is improved over `k_max` iterations by following the gradient of the controller, with a step size of `α`, to maximally improve the value of the initial belief.

```
function gradient(π::ControllerPolicy, M::ControllerGradient, 𝒫::POMDP)
    𝒮, 𝒜, 𝒪, T, O, R, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R, 𝒫.γ
    X, x1, ψ, η = π.X, 1, π.ψ, π.η
    n, m, z = length(𝒮), length(𝒜), length(𝒪)
    X𝒮 = vec(collect(product(X, 𝒮)))
    T′ = [sum(ψ[x, a] * T(s, a, s′) * sum(O(a, s′, o) * η[x, a, o, x′]
          for o in 𝒪) for a in 𝒜) for (x, s) in X𝒮, (x′, s′) in X𝒮]
    R′ = [sum(ψ[x, a] * R(s, a) for a in 𝒜) for (x, s) in X𝒮]
    Z = 1.0I(length(X𝒮)) - γ * T′
    invZ = inv(Z)
    ∂Z∂ψ(hx, ha) = [x == hx ? (-γ * T(s, ha, s′)
                    * sum(O(ha, s′, o) * η[hx, ha, o, x′]
                          for o in 𝒪)) : 0.0
                    for (x, s) in X𝒮, (x′, s′) in X𝒮]
    ∂Z∂η(hx, ha, ho, hx′) = [x == hx && x′ == hx′ ? (-γ * ψ[hx, ha]
                    * T(s, ha, s′) * O(ha, s′, ho)) : 0.0
                 for (x, s) in X𝒮, (x′, s′) in X𝒮]
    ∂R′∂ψ(hx, ha) = [x == hx ? R(s, ha) : 0.0 for (x, s) in X𝒮]
    ∂R′∂η(hx, ha, ho, hx′) = [0.0 for (x, s) in X𝒮]
    ∂U′∂ψ(hx, ha) = invZ * (∂R′∂ψ(hx, ha) - ∂Z∂ψ(hx, ha) * invZ * R′)
    ∂U′∂η(hx, ha, ho, hx′) = invZ * (∂R′∂η(hx, ha, ho, hx′)
                                - ∂Z∂η(hx, ha, ho, hx′) * invZ * R′)
    return ∂U′∂ψ, ∂U′∂η
end
```

Algorithm 23.7. The `gradient` step of the controller gradient ascent. It constructs the gradients of the utility U with respect to the policy `∂U′∂ψ` and `∂U′∂η`.

23.6 *Exercises*

Exercise 23.1. List any advantages that a controller policy representation has over conditional plan and belief-based representations.

Solution: Unlike tree-based conditional plans, controllers can represent policies that can be executed indefinitely. They do not have to grow exponentially in size with the horizon.

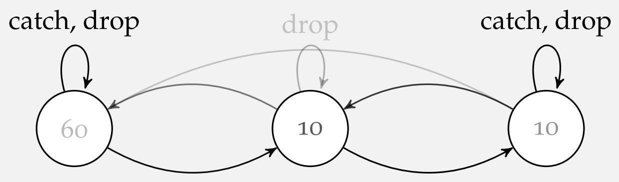
Compared to belief-based representations, the number of parameters in a controller representation tends to be far less than the number of alpha vectors for larger problems. We can also more easily optimize controllers for a fixed amount of memory.

Controllers will never divide by zero the way that belief-based policies can during execution. Belief-based methods require maintaining a belief. The discrete state filter from equation (19.7) will divide by zero if an impossible observation is made. This can happen when a noisy observation from a sensor returns an observation that the models of $T(s, a, s')$ and $O(o \mid a, s')$ does not accurately capture.

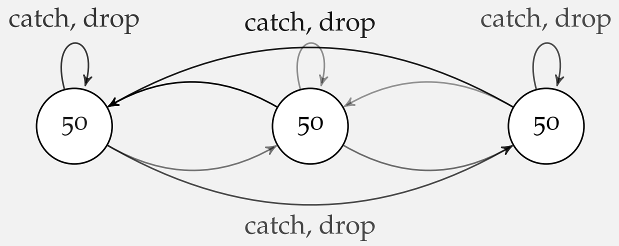
Exercise 23.2. Controller policy iteration only adds nodes with deterministic action selection functions and successor distributions. Does this mean that the resulting controller is necessarily suboptimal?

Example 23.6. An example of the controller gradient algorithm for controllers with a fixed size of $\ell = 3$. The policy is shown to refine itself over the algorithm's iterations. The agent incrementally determines how to best use its fixed number of nodes, resulting in a reasonable and interpretable policy upon convergence.

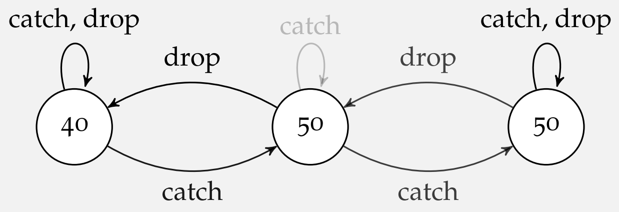
Consider the catch problem (appendix F.9) with a uniform initial belief b_1. The figures below show the utility of the policy over gradient ascent iteration applied to the catch problem with $k = 3$ nodes. The left node is x_1.



At iteration 1, the policy is essentially random, both in action selection and successor selection.



At iteration 50, the agent has determined a reasonable distance to throw the ball (50) but still has not used its three nodes to remember anything useful.



At iteration 500, the policy has constructed a reasonable plan, given its fixed three nodes of memory. It first tries throwing the ball at a distance of 40. If the child catches the ball, then it increases the range to 50. It uses the final node to remember how many times the child caught the ball (up to twice) to choose the distance.

Solution: Controller policy iteration is guaranteed to converge on an optimal policy in the limit. However, the method cannot find more compact representations of optimal controller policies that may require stochastic nodes.

Exercise 23.3. Prove that in policy iteration, any pruned node does not affect the utility.

Solution: Let x' be the new node from some iteration i, and x be a previous node from iteration $i-1$.

By construction, $\eta(x', a, o, x)$ defines all new nodes x' to only have a successor x from the previous iteration. Thus for each state s, $U^{(i)}(x', s)$ only sums over the successors $U^{(i-1)}(x, s')$ in equation (23.1). This means that the other utilities in iteration i, including a self-loop to x itself, do not affect the utility $U^{(i)}(x', s)$. Since the initial node is chosen by equation (23.3), we must ensure the utility with and without the pruned node at all beliefs is the same. A node is pruned in one of two ways.

First, x' obtains a higher utility over all states than its pruned successor x. Formally, $U^{(i)}(x, s) \leq U^{(i-1)}(x', s)$ for all s. The prune step replaces x with x', including U, ψ, and η. By construction, U has not decreased at any state s.

Second, x is identical to an existing previous node x'. Note that this means the transition $\eta(x, a, o, x') = \eta(x', a, o, x')$. This means that the utility is identical except that x is reduced by γ; in other words, $\gamma U^{(i)}(x, s) = U^{(i-1)}(x, s)$ by equation (23.1). It does not affect the final utility to prune x.

Exercise 23.4. Devise an algorithm that uses the nonlinear program algorithm to find the minimum fixed-sized controller required to obtain the optimality of a large fixed-sized controller of size ℓ. You can assume the nonlinear optimizer returns the optimal policy in this case.

Solution: The idea is to create an outer loop that increments the fixed-size of the controller, after knowing the utility of the large fixed-sized controller. First, we must compute the large fixed-sized controller's utility $U^* = \sum_s b_1(s) U(x_1, s)$ at initial node x_1 and initial belief b_1. Next, we create a loop that increments the size ℓ of the controller. At each step, we evaluate the policy and compute the utility U^ℓ. By our assumption, the controller returned produces a globally optimal utility for the fixed size ℓ. Once we arrive at a utility U^ℓ, if we see that $U^\ell = U^*$, then we stop and return the policy.

Exercise 23.5. Analyze the controller gradient ascent algorithm's gradient step. Assume that $|\mathcal{S}|$ is larger than $|\mathcal{A}|$ and $|\mathcal{O}|$. What is the most computationally expensive part of the gradient step? How might this be improved?

Solution: Computing the inverse $\mathbf{Z}^{-1} = (\mathbf{I} - \gamma \mathbf{T}_\theta)$ is the most computationally expensive part of the gradient step, as well as the entire gradient algorithm. The matrix $\mathbf{Z}$ is of size $|X \times \mathcal{S}|$. Gauss–Jordan elimination requires $O(|X \times \mathcal{S}|^3)$ operations, though the 3 in the exponent can be reduced to 2.3728639 using a state-of-the-art matrix inversion algorithm.[10] The creation of the temporary matrix T_θ also requires $O(|X \times \mathcal{S}|^2 |\mathcal{A} \times \mathcal{O}|)$ operations to

[10] F. L. Gall, "Powers of Tensors and Fast Matrix Multiplication," in *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, 2014.

support computing the inverse. All other loops and other temporary array creations require far fewer operations. This can be improved using an approximate inverse technique.