25 Sequential Problems

This chapter extends simple games to a sequential context with multiple states. A *Markov game* (*MG*) can be viewed as a Markov decision process involving multiple agents with their own reward functions.[1] In this formulation, transitions depend on the joint action and all agents seek to maximize their own reward. We generalize the response models and the Nash equilibrium solution concept from simple games to take into account the state transition model. The last part of this chapter discusses learning-based models, where the agents adapt their policies based on information from observed interactions and knowledge of the reward and transition functions.

[1] MGs, also called *stochastic games*, were originally studied in the 1950s around the same time as MDPs. L. S. Shapley, "Stochastic Games," *Proceedings of the National Academy of Sciences*, vol. 39, no. 10, pp. 1095–1100, 1953. They were introduced into the multiagent artificial intelligence community decades later. M. L. Littman, "Markov Games as a Framework for Multi-Agent Reinforcement Learning," in *International Conference on Machine Learning (ICML)*, 1994.

25.1 Markov Games

An MG (algorithm 25.1) extends a simple game to include a shared state $s \in \mathcal{S}$. The likelihood of transitioning from a state s to a state s' under a joint action $\mathbf{a}$ is given by the transition distribution $T(s' \mid s, \mathbf{a})$. Each agent i receives a reward according to its own reward function $R^i(s, \mathbf{a})$, which now also depends on the state. Example 25.1 sketches out how traffic routing can be framed as an MG.

```
struct MG
    γ  # discount factor
    ℐ  # agents
    𝒮  # state space
    𝒜  # joint action space
    T  # transition function
    R  # joint reward function
end
```

Algorithm 25.1. Data structure for an MG.

Consider commuters headed to work by car. Each car has a starting position and a destination. Each car can take any of several available roads to get to their destination, but these roads vary in the time it takes to drive them. The more cars that drive on a given road, the slower they all move.

This problem is an MG. The agents are the commuters in their cars, the states are the locations of all the cars on the roads, and the actions correspond to decisions of which road to take next. The state transition moves all car agents forward following their joint action. The negative reward is proportional to the time spent driving on a road.

Example 25.1. Traffic routing as an MG. The problem cannot be modeled using a single agent model like an MDP because we do not know the behavior of other agents, only their rewards. We can try to find equilibria or learn policies through interaction, similar to what we did for simple games.

The joint policy $\boldsymbol{\pi}$ in an MG specifies a probability distribution over joint actions, given the current state. As with MDPs, we will focus on policies that depend on the current state rather than the past history because future states and rewards are conditionally independent of the history, given the current state. In addition, we will focus on stationary policies, which do not depend on time. The probability that agent i selects action a at state s is given by $\pi^i(a \mid s)$. We will often use $\boldsymbol{\pi}(s)$ to represent a distribution over joint actions.

The utility of a joint policy $\boldsymbol{\pi}$ from the perspective of agent i can be computed using a variation of policy evaluation introduced in section 7.2 for MDPs. The reward to agent i from state s when following joint policy $\boldsymbol{\pi}$ is

$$R^i(s, \boldsymbol{\pi}(s)) = \sum_{\mathbf{a}} R^i(s, \mathbf{a}) \prod_{j \in \mathcal{I}} \pi^j(a^j \mid s) \tag{25.1}$$

The probability of transitioning from state s to s' when following $\boldsymbol{\pi}$ is

$$T(s' \mid s, \boldsymbol{\pi}(s)) = \sum_{\mathbf{a}} T(s' \mid s, \mathbf{a}) \prod_{j \in \mathcal{I}} \pi^j(a^j \mid s) \tag{25.2}$$

In an infinite-horizon discounted game, the utility for agent i from state s is

$$U^{\boldsymbol{\pi},i}(s) = R^i(s, \boldsymbol{\pi}(s)) + \gamma \sum_{s'} T(s' \mid s, \boldsymbol{\pi}(s)) U^{\boldsymbol{\pi},i}(s') \tag{25.3}$$

which can be solved exactly (algorithm 25.2).

```
struct MGPolicy
    p # dictionary mapping states to simple game policies
    MGPolicy(p::Base.Generator) = new(Dict(p))
end

(πi::MGPolicy)(s, ai) = πi.p[s](ai)
(πi::SimpleGamePolicy)(s, ai) = πi(ai)

probability(𝒫::MG, s, π, a) = prod(πj(s, aj) for (πj, aj) in zip(π, a))
reward(𝒫::MG, s, π, i) =
    sum(𝒫.R(s,a)[i]*probability(𝒫,s,π,a) for a in joint(𝒫.𝒜))
transition(𝒫::MG, s, π, s′) =
    sum(𝒫.T(s,a,s′)*probability(𝒫,s,π,a) for a in joint(𝒫.𝒜))

function policy_evaluation(𝒫::MG, π, i)
    𝒮, 𝒜, R, T, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.T, 𝒫.γ
    p(s,a) = prod(πj(s, aj) for (πj, aj) in zip(π, a))
    R′ = [sum(R(s,a)[i]*p(s,a) for a in joint(𝒜)) for s in 𝒮]
    T′ = [sum(T(s,a,s′)*p(s,a) for a in joint(𝒜)) for s in 𝒮, s′ in 𝒮]
    return (I - γ*T′)\R′
end
```

Algorithm 25.2. An MG policy is a mapping from states to simple game policies, introduced in the previous chapter. We can construct it by passing in a generator to construct the dictionary. The probability that a policy (either for an MG or a simple game) assigns to taking action `ai` from state `s` is `πi(s, ai)`. Functions are also provided for computing $R^i(s, \boldsymbol{\pi}(s))$ and $T(s' \mid s, \boldsymbol{\pi}(s))$. The policy evaluation function will compute a vector representing $U^{\boldsymbol{\pi},i}$.

25.2 Response Models

We can generalize the response models introduced in the previous chapter to MGs. Doing so requires taking into account the state transition model.

25.2.1 Best Response

A *response policy* for agent i is a policy π^i that maximizes expected utility, given the fixed policies of other agents $\boldsymbol{\pi}^{-i}$. If the policies of the other agents are fixed, then the problem reduces to an MDP. This MDP has state space $\mathcal{S}$ and action space $\mathcal{A}^i$. We can define the transition and reward functions as follows:

$$T'(s' \mid s, a^i) = T(s' \mid s, a^i, \boldsymbol{\pi}^{-i}(s)) \tag{25.4}$$

$$R'(s, a^i) = R^i(s, a^i, \boldsymbol{\pi}^{-i}(s)) \tag{25.5}$$

Because this is a best response for agent i, the MDP only uses reward R^i. Solving this MDP results in a best response policy for agent i. Algorithm 25.3 provides an implementation of this.

```
function best_response(𝒫::MG, π, i)
    𝒮, 𝒜, R, T, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.T, 𝒫.γ
    T′(s,ai,s′) = transition(𝒫, s, joint(π, SimpleGamePolicy(ai), i), s′)
    R′(s,ai) = reward(𝒫, s, joint(π, SimpleGamePolicy(ai), i), i)
    πi = solve(MDP(γ, 𝒮, 𝒜[i], T′, R′))
    return MGPolicy(s ⇒ SimpleGamePolicy(πi(s)) for s in 𝒮)
end
```

Algorithm 25.3. For an MG $\mathcal{P}$, we can compute a deterministic best response policy for agent `i`, given that the other agents are playing policies in `π`. We can solve the MDP exactly using one of the methods from chapter 7.

25.2.2 *Softmax Response*

Similar to what was done in the previous chapter, we can define a *softmax response policy*, which assigns a stochastic response to the policies of the other agents at each state. As we did in the construction of a deterministic best response policy, we solve an MDP where the agents with the fixed policies π^{-i} are folded into the environment. We then extract the action value function $Q(s, a)$ using one-step lookahead. The softmax response is

$$\pi^i(a^i \mid s) \propto \exp(\lambda Q(s, a^i)) \tag{25.6}$$

with precision parameter $\lambda \geq 0$. Algorithm 25.4 provides an implementation. This approach can be used to generate hierarchical softmax solutions (section 24.7). In fact, we can use algorithm 24.9 directly.

```
function softmax_response(𝒫::MG, π, i, λ)
    𝒮, 𝒜, R, T, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.T, 𝒫.γ
    T′(s,ai,s′) = transition(𝒫, s, joint(π, SimpleGamePolicy(ai), i), s′)
    R′(s,ai) = reward(𝒫, s, joint(π, SimpleGamePolicy(ai), i), i)
    mdp = MDP(γ, 𝒮, joint(𝒜), T′, R′)
    πi = solve(mdp)
    Q(s,a) = lookahead(mdp, πi.U, s, a)
    p(s) = SimpleGamePolicy(a ⇒ exp(λ*Q(s,a)) for a in 𝒜[i])
    return MGPolicy(s ⇒ p(s) for s in 𝒮)
end
```

Algorithm 25.4. The softmax response of agent `i` to joint policy `π` with precision parameter `λ`.

25.3 *Nash Equilibrium*

The Nash equilibrium concept can be generalized to MGs.[2] As with simple games, all agents perform a best response to one another and have no incentive to deviate. All finite MGs with a discounted infinite horizon have a Nash equilibrium.[3]

[2] Because we assume that policies are *stationary*, in that they do not vary over time, the Nash equilibria covered here are *stationary Markov perfect equilibria*.

[3] A. M. Fink, "Equilibrium in a Stochastic n-Person Game," *Journal of Science of the Hiroshima University, Series A-I*, vol. 28, no. 1, pp. 89–93, 1964.

We can find a Nash equilibrium by solving a nonlinear optimization problem similar to the one that we solved in the context of simple games. This problem minimizes the sum of the lookahead utility deviations and constrains the policies to be valid distributions:

$$\begin{aligned} \underset{\boldsymbol{\pi}, U}{\text{minimize}} \quad & \sum_{i \in \mathcal{I}} \sum_{s} \left(U^i(s) - Q^i(s, \boldsymbol{\pi}(s)) \right) \\ \text{subject to} \quad & U^i(s) \geq Q^i(s, a^i, \boldsymbol{\pi}^{-i}(s)) \text{ for all } i, s, a^i \\ & \sum_{a^i} \pi^i(a^i \mid s) = 1 \text{ for all } i, s \\ & \pi^i(a^i \mid s) \geq 0 \text{ for all } i, s, a^i \end{aligned} \tag{25.7}$$

where

$$Q^i(s, \boldsymbol{\pi}(s)) = R^i(s, \boldsymbol{\pi}(s)) + \gamma \sum_{s'} T(s' \mid s, \boldsymbol{\pi}(s)) U^i(s') \tag{25.8}$$

This nonlinear optimization problem is implemented and solved in algorithm 25.5.[4]

[4] J. A. Filar, T. A. Schultz, F. Thuijsman, and O. Vrieze, "Nonlinear Programming and Stationary Equilibria in Stochastic Games," *Mathematical Programming*, vol. 50, no. 1–3, pp. 227–237, 1991.

25.4 Fictitious Play

As we did in the context of simple games, we can take a learning-based approach to arrive at joint policies by running agents in simulation. Algorithm 25.6 generalizes the simulation loop introduced in the previous chapter to handle state transitions. The various policies run in simulation update themselves based on the state transitions and the actions taken by the various agents.

One approach for updating policies is to use a generalization of *fictitious play* (algorithm 25.7) from the previous chapter,[5] which involves maintaining a maximum-likelihood model over the policies of the other agents. The maximum likelihood model tracks the state in addition to the action being taken by each agent. We track the number of times that agent j takes action a^j in state s, storing it in table $N(j, a^j, s)$, typically initialized to 1. Then, we can compute the best response, assuming that each agent j follows the state-dependent stochastic policy:

$$\pi^j(a^j \mid s) \propto N(j, a^j, s) \tag{25.9}$$

[5] W. Uther and M. Veloso, "Adversarial Reinforcement Learning," Carnegie Mellon University, Tech. Rep. CMU-CS-03-107, 1997. M. Bowling and M. Veloso, "An Analysis of Stochastic Game Theory for Multiagent Reinforcement Learning," Carnegie Mellon University, Tech. Rep. CMU-CS-00-165, 2000.

```
function tensorform(𝒫::MG)
    ℐ, 𝒮, 𝒜, R, T = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.T
    ℐ′ = eachindex(ℐ)
    𝒮′ = eachindex(𝒮)
    𝒜′ = [eachindex(𝒜[i]) for i in ℐ]
    R′ = [R(s,a) for s in 𝒮, a in joint(𝒜)]
    T′ = [T(s,a,s′) for s in 𝒮, a in joint(𝒜), s′ in 𝒮]
    return ℐ′, 𝒮′, 𝒜′, R′, T′
end

function solve(M::NashEquilibrium, 𝒫::MG)
    ℐ, 𝒮, 𝒜, R, T = tensorform(𝒫)
    𝒮′, 𝒜′, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.γ
    model = Model(Ipopt.Optimizer)
    @variable(model, U[ℐ, 𝒮])
    @variable(model, π[i=ℐ, 𝒮, ai=𝒜[i]] ≥ 0)
    @NLobjective(model, Min,
        sum(U[i,s] - sum(prod(π[j,s,a[j]] for j in ℐ)
            * (R[s,y][i] + γ*sum(T[s,y,s′]*U[i,s′] for s′ in 𝒮))
            for (y,a) in enumerate(joint(𝒜))) for i in ℐ, s in 𝒮))
    @NLconstraint(model, [i=ℐ, s=𝒮, ai=𝒜[i]],
        U[i,s] ≥ sum(
            prod(j==i ? (a[j]==ai ? 1.0 : 0.0) : π[j,s,a[j]] for j in ℐ)
            * (R[s,y][i] + γ*sum(T[s,y,s′]*U[i,s′] for s′ in 𝒮))
            for (y,a) in enumerate(joint(𝒜))))
    @constraint(model, [i=ℐ, s=𝒮], sum(π[i,s,ai] for ai in 𝒜[i]) == 1)
    optimize!(model)
    π′ = value.(π)
    πi′(i,s) = SimpleGamePolicy(𝒜′[i][ai] ⇒ π′[i,s,ai] for ai in 𝒜[i])
    πi′(i) = MGPolicy(𝒮′[s] ⇒ πi′(i,s) for s in 𝒮)
    return [πi′(i) for i in ℐ]
end
```

Algorithm 25.5. This nonlinear program computes a Nash equilibrium for an MG $\mathcal{P}$.

```
function randstep(𝒫::MG, s, a)
    s′ = rand(SetCategorical(𝒫.𝒮, [𝒫.T(s, a, s′) for s′ in 𝒫.𝒮]))
    r = 𝒫.R(s,a)
    return s′, r
end

function simulate(𝒫::MG, π, k_max, b)
    s = rand(b)
    for k = 1:k_max
        a = Tuple(πi(s)() for πi in π)
        s′, r = randstep(𝒫, s, a)
        for πi in π
            update!(πi, s, a, s′)
        end
        s = s′
    end
    return π
end
```

Algorithm 25.6. Functions for taking a random step and running full simulations in MGs. The simulate function will simulate the joint policy `π` for `k_max` steps starting from a state randomly sampled from `b`.

After observing joint action $\mathbf{a}$ in states s, we update

$$N(j, a^j, s) \leftarrow N(j, a^j, s) + 1 \tag{25.10}$$

for each agent j.

As the distributions of the other agents' actions change, we must update the utilities. The utilities in MGs are significantly more difficult to compute than simple games because of the state dependency. As described in section 25.2.1, any assignment of fixed policies of others $\boldsymbol{\pi}^{-i}$ induces an MDP. In fictitious play, $\boldsymbol{\pi}^{-i}$ is determined by equation (25.9). Instead of solving an MDP at each update, it is common to apply the update periodically, a strategy adopted from asynchronous value iteration. An example of fictitious play is given in example 25.2.

Our policy $\pi^i(s)$ for a state s is derived from a given opponent model $\boldsymbol{\pi}^{-i}$ and computed utility U^i. We then select a best response:

$$\arg\max_a Q^i(s, a, \boldsymbol{\pi}^{-i}) \tag{25.11}$$

In the implementation here, we use the property that each state of an MG policy is a simple game policy whose reward is the corresponding Q^i.

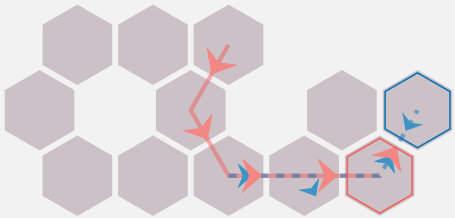
```
mutable struct MGFictitiousPlay
    𝒫  # Markov game
    i  # agent index
    Qi # state-action value estimates
    Ni # state-action counts
end

function MGFictitiousPlay(𝒫::MG, i)
    ℐ, 𝒮, 𝒜, R = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.R
    Qi = Dict((s, a) ⇒ R(s, a)[i] for s in 𝒮 for a in joint(𝒜))
    Ni = Dict((j, s, aj) ⇒ 1.0 for j in ℐ for s in 𝒮 for aj in 𝒜[j])
    return MGFictitiousPlay(𝒫, i, Qi, Ni)
end

function (πi::MGFictitiousPlay)(s)
    𝒫, i, Qi = πi.𝒫, πi.i, πi.Qi
    ℐ, 𝒮, 𝒜, T, R, γ = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.T, 𝒫.R, 𝒫.γ
    πi′(i,s) = SimpleGamePolicy(ai ⇒ πi.Ni[i,s,ai] for ai in 𝒜[i])
    πi′(i) = MGPolicy(s ⇒ πi′(i,s) for s in 𝒮)
    π = [πi′(i) for i in ℐ]
    U(s,π) = sum(πi.Qi[s,a]*probability(𝒫,s,π,a) for a in joint(𝒜))
    Q(s,π) = reward(𝒫,s,π,i) + γ*sum(transition(𝒫,s,π,s′)*U(s′,π)
                                     for s′ in 𝒮)
    Q(ai) = Q(s, joint(π, SimpleGamePolicy(ai), i))
    ai = argmax(Q, 𝒫.𝒜[πi.i])
    return SimpleGamePolicy(ai)
end

function update!(πi::MGFictitiousPlay, s, a, s′)
    𝒫, i, Qi = πi.𝒫, πi.i, πi.Qi
    ℐ, 𝒮, 𝒜, T, R, γ = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.T, 𝒫.R, 𝒫.γ
    for (j,aj) in enumerate(a)
        πi.Ni[j,s,aj] += 1
    end
    πi′(i,s) = SimpleGamePolicy(ai ⇒ πi.Ni[i,s,ai] for ai in 𝒜[i])
    πi′(i) = MGPolicy(s ⇒ πi′(i,s) for s in 𝒮)
    π = [πi′(i) for i in ℐ]
    U(π,s) = sum(πi.Qi[s,a]*probability(𝒫,s,π,a) for a in joint(𝒜))
    Q(s,a) = R(s,a)[i] + γ*sum(T(s,a,s′)*U(π,s′) for s′ in 𝒮)
    for a in joint(𝒜)
        πi.Qi[s,a] = Q(s,a)
    end
end
```

Algorithm 25.7. Fictitious play for agent `i` in an MG $\mathcal{P}$ that maintains counts `Ni` of other agent action selections over time for each state and averages them, assuming that this is their stochastic policy. It then computes a best response to this policy and performs the corresponding utility-maximizing action.

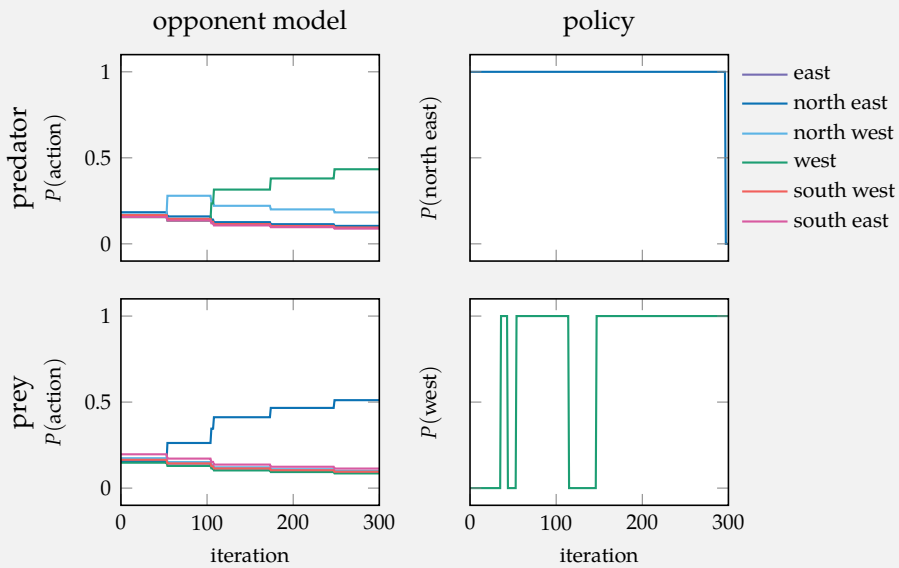
Example 25.2. Fictitious play on the predator-prey hex world problem. Stochasticity was introduced when initializing the policies to better show learning trends.

The predator-prey hex world MG (appendix F.13) has one predator (red) and one prey (blue). If the predator catches the prey, it receives a reward of 10 and the prey receives a reward of -100. Otherwise, both agents receive a -1 reward. The agents move simultaneously. We apply fictitious play with resets to the initial state every 10 steps.

We observe that the predator learns to chase the prey and the prey learns to flee. Interestingly, the predator also learns that the prey runs to the east corner and waits. The prey learns that if it waits at this corner, it can flee from the predator immediately as it jumps toward the prey. Here, the prey evades the predator by moving west when the predator moves north east.



Here is a plot of the learned opponent model of the highlighted state (both predator and prey hex locations) for both the predator and the prey:



25.5 Gradient Ascent

We can use *gradient ascent* (algorithm 25.8) to learn policies in a way similar to what was done in the previous chapter for simple games. The state must now be considered and requires learning the action value function. At each time step t, all agents perform joint actions $\mathbf{a}_t$ in a state s_t. As in gradient ascent for simple games, an agent i assumes that the agents' policies $\boldsymbol{\pi}_t^{-i}$ are the observed actions $\mathbf{a}_t^{-i}$. The gradient is

$$\frac{\partial U^{\boldsymbol{\pi}_t,i}(s_t)}{\partial \pi_t^i(a^i \mid s_t)} = \frac{\partial}{\partial \pi^i(a^i \mid s_t)} \left(\sum_{\mathbf{a}} \prod_j \pi^j(a^j \mid s_t) Q^{\boldsymbol{\pi}_t,i}(s_t, \mathbf{a}_t) \right) \tag{25.12}$$

$$= Q^{\boldsymbol{\pi}_t,i}(s_t, a^i, \mathbf{a}_t^{-i}) \tag{25.13}$$

The gradient step follows a similar pattern as in the previous chapter, except the state s is included and the expected utility estimate Q_t^i is used:

$$\pi_{t+1}^i(a^i \mid s_t) = \pi_t^i(a^i \mid s_t) + \alpha_t^i Q^i(s_t, a^i, \mathbf{a}^{-i}) \tag{25.14}$$

Again, this update may require projection to ensure that the policy π_{t+1}^i at s_t is a valid probability distribution.

As with fictitious play in the previous section, we must estimate Q_t^i. We can use Q-learning:

$$Q_{t+1}^i(s_t, \mathbf{a}_t) = Q_t^i(s_t, \mathbf{a}_t) + \alpha_t \left(R^i(s_t, \mathbf{a}_t) + \gamma \max_{a^{i\prime}} Q_t^i(s_{t+1}, a^{i\prime}, \mathbf{a}_t^{-i}) - Q_t^i(s_t, \mathbf{a}_t) \right) \tag{25.15}$$

We can use the inverse square root learning rate $\alpha_t = 1/\sqrt{t}$. Exploration is also necessary. We can use an ϵ-greedy strategy, perhaps also with $\epsilon_t = 1/\sqrt{t}$.

25.6 Nash Q-Learning

Another learning-based approach is *Nash Q-learning* (algorithm 25.9), which borrows inspiration from Q-learning (section 17.2).[6] The method maintains an estimate of the action value function, which is adapted as the agents react to each other's changing policies. In the process of updating the action value function, it computes a Nash equilibrium to model the behavior of the other agents.

[6] J. Hu and M. P. Wellman, "Nash Q-Learning for General-Sum Stochastic Games," *Journal of Machine Learning Research*, vol. 4, pp. 1039–1069, 2003.

```
mutable struct MGGradientAscent
    𝒫  # Markov game
    i  # agent index
    t  # time step
    Qi # state-action value estimates
    πi # current policy
end

function MGGradientAscent(𝒫::MG, i)
    ℐ, 𝒮, 𝒜 = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜
    Qi = Dict((s, a) ⇒ 0.0 for s in 𝒮, a in joint(𝒜))
    uniform() = Dict(s ⇒ SimpleGamePolicy(ai ⇒ 1.0 for ai in 𝒫.𝒜[i])
                     for s in 𝒮)
    return MGGradientAscent(𝒫, i, 1, Qi, uniform())
end

function (πi::MGGradientAscent)(s)
    𝒜i, t = πi.𝒫.𝒜[πi.i], πi.t
    ϵ = 1 / sqrt(t)
    πi′(ai) = ϵ/length(𝒜i) + (1-ϵ)*πi.πi[s](ai)
    return SimpleGamePolicy(ai ⇒ πi′(ai) for ai in 𝒜i)
end

function update!(πi::MGGradientAscent, s, a, s′)
    𝒫, i, t, Qi = πi.𝒫, πi.i, πi.t, πi.Qi
    ℐ, 𝒮, 𝒜i, R, γ = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜[πi.i], 𝒫.R, 𝒫.γ
    jointπ(ai) = Tuple(j == i ? ai : a[j] for j in ℐ)
    α = 1 / sqrt(t)
    Qmax = maximum(Qi[s′, jointπ(ai)] for ai in 𝒜i)
    πi.Qi[s, a] += α * (R(s, a)[i] + γ * Qmax - Qi[s, a])
    u = [Qi[s, jointπ(ai)] for ai in 𝒜i]
    π′ = [πi.πi[s](ai) for ai in 𝒜i]
    π = project_to_simplex(π′ + u / sqrt(t))
    πi.t = t + 1
    πi.πi[s] = SimpleGamePolicy(ai ⇒ p for (ai, p) in zip(𝒜i, π))
end
```

Algorithm 25.8. Gradient ascent for an agent `i` of an MG $\mathcal{P}$. The algorithm incrementally updates its distributions of actions at visited states following gradient ascent to improve the expected utility. The projection function from algorithm 23.6 is used to ensure that the resulting policy remains a valid probability distribution.

An agent following Nash Q-learning maintains an estimate of a joint action value function $\mathbf{Q}(s, \mathbf{a})$. This action value function is updated after every state transition using a Nash equilibrium computed from a simple game constructed from this value function. After a transition from s to s' following the joint action $\mathbf{a}$, we construct a simple game with the same number of agents and the same joint action space, but the reward function is equal to the estimated value of s' such that $\mathbf{R}(\mathbf{a}') = \mathbf{Q}(s', \mathbf{a}')$. The agent computes a Nash equilibrium policy $\boldsymbol{\pi}'$ over the next action $\mathbf{a}'$. Under the derived policy, the expected utility of the successor state is

$$\mathbf{U}(s') = \sum_{\mathbf{a}'} \mathbf{Q}(s', \mathbf{a}') \prod_{j \in \mathcal{I}} \pi^{j\prime}(a^{j\prime}) \tag{25.16}$$

The agent then updates its value function:

$$\mathbf{Q}(s, \mathbf{a}) \leftarrow \mathbf{Q}(s, \mathbf{a}) + \alpha\big(\mathbf{R}(s, \mathbf{a}) + \gamma \mathbf{U}(s') - \mathbf{Q}(s, \mathbf{a})\big) \tag{25.17}$$

where the learning rate α is typically a function of the state-action count $\alpha = 1/\sqrt{N(s, \mathbf{a})}$.

As with regular Q-learning, we need to adopt an exploration strategy to ensure that all states and actions are tried often enough. In algorithm 25.9, the agent follows an ϵ-greedy policy. With probability $\epsilon = 1/\sum_{\mathbf{a}}(N(s, \mathbf{a}))$, it selects an action uniformly at random. Otherwise, it will use the result from the Nash equilibrium.

25.7 Summary

- MGs are an extension of MDPs to multiple agents or an extension of simple games to sequential problems. In these problems, multiple agents compete and individually receive rewards over time.
- The Nash equilibrium can be formulated for MGs, but it must now consider all actions for all agents in all states.
- The problem of finding a Nash equilibrium can be formulated as a nonlinear optimization problem.
- We can generalize fictitious play to MGs by using a known transition function and incorporating estimates of action values.

```
mutable struct NashQLearning
    𝒫 # Markov game
    i # agent index
    Q # state-action value estimates
    N # history of actions performed
end

function NashQLearning(𝒫::MG, i)
    ℐ, 𝒮, 𝒜 = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜
    Q = Dict((j, s, a) ⇒ 0.0 for j in ℐ, s in 𝒮, a in joint(𝒜))
    N = Dict((s, a) ⇒ 1.0 for s in 𝒮, a in joint(𝒜))
    return NashQLearning(𝒫, i, Q, N)
end

function (πi::NashQLearning)(s)
    𝒫, i, Q, N = πi.𝒫, πi.i, πi.Q, πi.N
    ℐ, 𝒮, 𝒜, 𝒜i, γ = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒜[πi.i], 𝒫.γ
    M = NashEquilibrium()
    𝒢 = SimpleGame(γ, ℐ, 𝒜, a -> [Q[j, s, a] for j in ℐ])
    π = solve(M, 𝒢)
    ϵ = 1 / sum(N[s, a] for a in joint(𝒜))
    πi′(ai) = ϵ/length(𝒜i) + (1-ϵ)*π[i](ai)
    return SimpleGamePolicy(ai ⇒ πi′(ai) for ai in 𝒜i)
end

function update!(πi::NashQLearning, s, a, s′)
    𝒫, ℐ, 𝒮, 𝒜, R, γ = πi.𝒫, πi.𝒫.ℐ, πi.𝒫.𝒮, πi.𝒫.𝒜, πi.𝒫.R, πi.𝒫.γ
    i, Q, N = πi.i, πi.Q, πi.N
    M = NashEquilibrium()
    𝒢 = SimpleGame(γ, ℐ, 𝒜, a′ -> [Q[j, s′, a′] for j in ℐ])
    π = solve(M, 𝒢)
    πi.N[s, a] += 1
    α = 1 / sqrt(N[s, a])
    for j in ℐ
        πi.Q[j,s,a] += α*(R(s,a)[j] + γ*utility(𝒢,π,j) - Q[j,s,a])
    end
end
```

Algorithm 25.9. Nash Q-learning for an agent `i` in an MG $\mathcal{P}$. The algorithm performs joint-action Q-learning to learn a state-action value function for all agents. A simple game is built with `Q`, and we compute a Nash equilibrium using algorithm 24.5. The equilibrium is then used to update the value function. This implementation also uses a variable learning rate proportional to the number of times state-joint-action pairs are visited, which is stored in `N`. In addition, it uses ϵ-greedy exploration to ensure that all states and actions are explored.

- Gradient ascent approaches iteratively improve a stochastic policy, and they do not need to assume a model.
- Nash Q-learning adapts traditional Q-learning to multiagent problems and involves solving for a Nash equilibrium of a simple game constructed from models of the other players.

25.8 Exercises

Exercise 25.1. Show that MGs are extensions of both MDPs and simple games. Show this by formulating an MDP as an MG and by formulating a simple game as an MG.

Solution: MGs generalize simple games. For any simple game with $\mathcal{I}$, $\mathcal{A}$, and $\mathbf{R}$, we can construct an MG by just having a single state that self-loops. In other words, this MG has $\mathcal{S} = \{s^1\}$, $T(s^1 \mid s^1, \mathbf{a}) = 1$, and $\mathbf{R}(s^1, \mathbf{a}) = \mathbf{R}(\mathbf{a})$.

MGs generalize MDPs. For any MDP with $\mathcal{S}$, $\mathcal{A}$, T, and R, we can construct an MG by just assigning the agents to be this single agent. In other words, this MG has $\mathcal{I} = \{1\}$, $\mathcal{A}^1 = \mathcal{A}$, $T(s' \mid s, \mathbf{a}) = T(s' \mid s', a)$, and $\mathbf{R}(s, a) = R(s, a)$.

Exercise 25.2. For an agent i, given the fixed policies of other agents $\boldsymbol{\pi}^{-i}$, can there be a stochastic best response that yields a greater utility than a deterministic best response? Why do we consider stochastic policies in a Nash equilibrium?

Solution: No, if given fixed policies of other agents $\boldsymbol{\pi}^{-i}$, a deterministic best response is sufficient to obtain the highest utility. The best response can be formulated as solving an MDP, as described in section 25.2. It has been shown that deterministic policies are sufficient to provide optimal utility maximization. Hence, the same is true for a best response in an MG.

In a Nash equilibrium, a best response has to hold for all agents. Although a deterministic best response might be equal in utility to a stochastic one, an equilibrium may require stochastic responses in order to prevent other agents from wanting to deviate.

Exercise 25.3. This chapter discussed only stationary Markov policies. What other categories of policies are there?

Solution: A so-called *behavioral policy* $\pi^i(\mathbf{h}_t)$ is one that has a dependence on the complete history $\mathbf{h}_t = (s_{1:t}, \mathbf{a}_{1:t-1})$. Such policies depend on the history of play of other agents. A *nonstationary Markov policy* $\pi^i(s, t)$ is one that depends on the time step t, but not on the complete history. For example, in the predator-prey hex world domain, for the first 10 time steps, the action at a hex might be to go east, and after 10 time steps, to go west.

There can be Nash equilibria that are in the space of nonstationary, non-Markov joint policies; stationary, non-Markov joint policies; and so forth. However, it has been proven that every stationary MG has a stationary Markov Nash equilibrium.

Exercise 25.4. In MGs, fictitious play requires the utilities to be estimated. List different approaches to compute utilities, with their benefits and drawbacks.

Solution: Algorithm 25.7 performs a single backup for the visited state s and all joint actions $\mathbf{a}$. This approach has the benefit of being relatively efficient because it is a single backup. Updating all joint actions at that state results in exploring actions that were not observed. The drawback of this approach is that we may need to do this update at all states many times to obtain a suitable policy.

An alternative is only to update the visited state and the joint action that was actually taken, which results in a faster update step. The drawback is that it requires many more steps to explore the full range of joint actions.

Another alternative is to perform value iteration at all states s until convergence at every update step. Recall that the model of the opponent changes on each update. This induces a new MDP, as described for deterministic best response in section 25.2.1. Consequently, we would need to rerun value iteration after each update. The benefit of this approach is that it can result in the most informed decision at each step because the utilities Q^i consider all states over time. The drawback is that the update step is very computationally expensive.