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.\footnote{Markov games, 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.} 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.

25.1  Markov Games

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

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

Algorithm 25.1. Data structure for a Markov game.
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 a Markov game. 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 on 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.

The joint policy $\pi$ in a Markov game specifies a probability distribution over joint actions given the current state. As with Markov decision processes, 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 agent $i$ selects action $a$ at state $s$ is given by $\pi^i(a \mid s)$. We will often use $\pi(s)$ to represent a distribution over joint actions.

The utility of a joint policy $\pi$ 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 $\pi$ is

$$R^i(s, \pi(s)) = \sum_a R^i(s, a) \prod_{j \in I} \pi^j(a^j \mid s) \quad (25.1)$$

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

$$T(s' \mid s, \pi(s)) = \sum_a T(s' \mid s, a) \prod_{j \in I} \pi^j(a^j \mid s) \quad (25.2)$$

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

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

which can be solved exactly (algorithm 25.2).
25.2 Response Models

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

25.2.1 Best Response

A response policy for agent $i$ is a policy $\pi^i$ that maximizes expected utility given the fixed policies of other agents $\pi^{-i}$. 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:

$$T'(s' \mid s, a^i) = T(s' \mid s, a^i, \pi^{-i}(s)) \quad (25.4)$$
$$R'(s, a^i) = R^i(s, a^i, \pi^{-i}(s)) \quad (25.5)$$

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.
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 $\pi^{-i}$ are folded into the environment. We then extract the action value function $Q(s,a)$ using one-step lookahead. The softmax response is then

$$\pi_i(a_i | s) \propto \exp(\lambda Q(s,a_i))$$ (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 24.6). In fact, we can use algorithm 24.9 directly.

Algorithm 25.4. The softmax response of agent $i$ to joint policy $\pi$ with precision parameter $\lambda$.

25.3 Nash Equilibrium

The Nash equilibrium concept can be generalized to Markov games. As with simple games, all agents perform a best response to one another, and have no incentive to deviate. All finite Markov games with a discounted infinite horizon have a Nash equilibrium.\(^2\)

\(^2\) Because we assume policies are stationary, in that they do not vary over time, the Nash equilibria covered here are stationary Markov perfect equilibria.


© 2022 Massachusetts Institute of Technology, shared under a under a Creative Commons CC-BY-NC-ND license.

2021-03-07 23:12:00-08:00, revision 80bf4d6, comments to bugs@algorithmsbook.com
We can find a Nash equilibrium by solving a nonlinear optimization problem similar to the one 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:

\[
\begin{align*}
\text{minimize} & \quad \sum_{i \in I} \sum_{s} \left( U^i(s) - Q^i(s, \pi(s)) \right) \\
\text{subject to} & \quad U^i(s) \geq Q^i(s, a^i, \pi^{-i}(s)) \quad \text{for all } i, s, a^i \\
& \quad \sum_{a^i} \pi^i(a^i \mid s) = 1 \quad \text{for all } i, s \\
& \quad \pi^i(a^i \mid s) \geq 0 \quad \text{for all } i, s, a^i
\end{align*}
\]

where

\[
Q^i(s, \pi(s)) = R^i(s, \pi(s)) + \gamma \sum_{s'} T(s' \mid s, \pi(s)) U^i(s')
\]

This nonlinear optimization problem is implemented and solved in algorithm 25.5.4

### 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. It 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 agent \(j\) takes action \(a^j\) in state \(s\), storing it in a 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:

\[
\pi^j(a^j \mid s) \propto N^j(j, a^j, s)
\]

After observing joint action \(a\) in states \(s\), we update

\[
N(j, a^j, s) \leftarrow N(j, a^j, s) + 1
\]

for each agent \(j\).

---


```plaintext
function tensorform(𝒫::MG)
    ℐ, ℳ, ℬ, ℞ = ℐ, ℳ, ℬ, ℞
    ℐ′ = eachindex(ℐ)
    ℳ′ = eachindex(ℳ)
    ℬ′ = [eachindex(ℬ[𝑖]) for 𝑖 in ℐ]
    ℞′ = [ℛ(𝑠, ℬ) for 𝑠 in ℳ, ℬ in joint(ℬ)]
    ℬ′′ = [ℬ(𝑠, ℬ′[𝑖], ℬ′[𝑖]) for 𝑠 in ℳ, ℬ in joint(ℬ), ℬ′[𝑖] in ℬ]
    return ℐ′, ℳ′, ℬ′, ℞′
end

function solve!(𝑴::NashEquilibrium, ℐ::MG)
    ℐ, ℳ, ℬ, ℞ = tensorform(ℐ)
    ℬ′′, ℬ′′, ℞′ = ℬ′′, ℬ′′, ℞′
    γ = ℐ, ℳ, ℬ, ℞
    model = Model(Ipopt.Optimizer)
    @variable(model, 𝑈[𝑖=ℐ, ℳ=ℳ], ℬ′′[𝑖=ℐ, ℳ=ℳ] ≥ 0)
    @NLobjective(model, Min, sum(U[𝑖, ℳ] - sum(prod(𝜖[𝑗, ℳ, ℬ] for 𝑗 in ℐ) for 𝑖 in ℐ, ℳ in ℳ)
        * (ℛ[𝑖, ℳ, ℬ][𝑖] + γ * sum(𝑇[𝑖, ℳ, ℬ′′] * 𝑈[𝑖, ℳ′] for ℳ′ in ℳ) for ℳ in ℳ)
        for (𝑖, ℳ, ℬ) in enumerate(joint(ℬ))
    )
    @NLconstraint(model, [𝑖=ℐ, ℳ=ℳ, ℬ′′[𝑖]=ℬ′′[𝑖]] for ℬ′′ in ℬ′′)
    optimize!(model)
    𝜋′ = value.(𝜋)
    𝜋′(𝑖, ℳ) = SimpleGamePolicy(ℬ′′[𝑖][𝑖][𝑖] for 𝑖 in ℐ)
    𝜋′(𝑖) = MGPolicy(ℳ′[𝑖] for 𝑖 in ℐ)
    return [𝜋′(𝑖) for 𝑖 in ℐ]
end
```

Algorithm 25.5. This nonlinear program computes a Nash equilibrium for a Markov game ℐ.
function randstep(\( \mathcal{P} :: \text{MG} \), s, a)
    s' = \text{rand}(\text{SetCategorical}(\mathcal{P}.S, [\mathcal{P}.T(s, a, s') \text{ for } s' \text{ in } \mathcal{P}.S]))
    r = \mathcal{P}.R(s, a)
    return s', r
end

function simulate(\( \mathcal{P} :: \text{MG} \), \pi, k_{\text{max}}, b)
    s = \text{rand}(b)
    for k = 1:k_{\text{max}}
        a = \text{Tuple}(\pi_i(s)) \text{ for } \pi_i \text{ in } \pi
        s', r = \text{randstep}(\mathcal{P}, s, a)
        for \pi_i \text{ in } \pi
            \text{update!}(\pi_i, s, a, s')
        end
        s = s'
    end
    return \pi
end

As the distributions of the other agents' actions change, we must update the utilities. The utilities in Markov games are significantly more difficult to compute compared to simple games because of the state dependency. As described in section 25.2.1, for any assignment of fixed policies of others \( \pi^{-i} \) determined from equation (25.9) induces an MDP. 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 \( \pi^{-i} \) and computed utility \( U^i \). We then select a best response:

\[
\arg \max_a Q^i(s, a, \pi^{-i}) \quad (25.11)
\]

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

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 \( a_t \) in a state \( s_t \). As in gradient ascent for simple
mutable struct MGFictitiousPlay
    ℙ # Markov game
    i  # agent index
    Qi # state-action value estimates
    Ni # state-action counts
end

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

function (πᵢ::MGFictitiousPlay)(s)
    ℙᵢ, Qiᵢ = ℙᵢ.ℙᵢ, ℙᵢ.Qiᵢ
    ℱᵢ, ™ᵢ, ℛᵢ = ℙᵢ.ℱᵢ, ℙᵢ.™ᵢ, ℙᵢ.ℛᵢ
    πᵢ′(i, s) = SimpleGamePolicy(ai ⇒ πᵢ.Niᵢ[i,s,ai] for ai in ℛᵢ[i])
    πᵢ′(i) = MGP(i, ℙᵢ, ℡ᵢ, ℛᵢ, ℛᵢ.γ)
    π = [πᵢ′(i) for i in ℱᵢ]
    U(s, π) = sum(πᵢ.Qiᵢ[s,a]*probability(ℙᵢ,s,π,a) for a in joint(ℛᵢ))
    Q(s, π) = reward(ℙᵢ,s,π,i) + ℛᵢ*sum(transition(ℙᵢ,s,π,s′)*U(s′,π)
                        for s′ in ™ᵢ)
    Q(ai) = Q(s, joint(π, SimpleGamePolicy(ai), i))
    ai = argmax(Q, ℙᵢ.ℳᵢ[i.i])
    return SimpleGamePolicy(ai)
end

function update!(πᵢ::MGFictitiousPlay, s, a, s′)
    ℙᵢ, Qiᵢ = ℙᵢ.ℙᵢ, ℙᵢ.Qiᵢ
    ℱᵢ, ™ᵢ, ℛᵢ = ℙᵢ.ℱᵢ, ℙᵢ.™ᵢ, ℙᵢ.ℛᵢ
    for (j, aᵢ) in enumerate(aᵢ)
        πᵢ.Niᵢ[j,s,ai] += 1
    end
    πᵢ′(i, s) = SimpleGamePolicy(ai ⇒ πᵢ.Niᵢ[i,s,ai] for ai in ℛᵢ[i])
    πᵢ′(i) = MGP(i, ℙᵢ, ℡ᵢ, ℛᵢ, ℛᵢ.γ)
    π = [πᵢ′(i) for i in ℱᵢ]
    U(π, s) = sum(πᵢ.Qiᵢ[s,a]*probability(ℙᵢ,s,π,a) for a in joint(ℛᵢ))
    Q(s, a) = ℙᵢ(s, a)[i] + ℛᵢ*sum(transition(ℙᵢ,s,π,s′)*U(π,s′)
                                  for s′ in ™ᵢ)
    for a in joint(ℛᵢ)
        πᵢ.Qiᵢ[s,a] = Q(s, a)
    end
end

Algorithm 25.7. Fictitious play for agent i in an MG ℙ that maintains counts Niᵢ of other agent action selections over time for each state and averages them assuming this is their stochastic policy. It then computes a best response to this policy and performs the corresponding utility-maximizing action.
The predator-prey hex world Markov game (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 right corner and waits. Here, the prey learns that if it waits at this corner, it can flee from the predator immediately as it jumps towards the prey. In this case, it escapes the predator and runs to the other side of the map.

Below 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.
Chapter 25. Sequential Problems

Games, an agent $i$ assumes the agents' policies $\pi_t^{-i}$ are the observed actions $a_t^{-i}$. The gradient is then

$$\frac{\partial U^{\pi_t^{-i}}(s_t)}{\partial \pi_t^i(a_t^i \mid s_t)} = \frac{\partial}{\partial \pi_t^i(a_t^i \mid s_t)} \left( \sum_a \prod_j \pi_j^i(a_j^i \mid s_t) Q^{\pi_t^{-i}}(s_t, a_t^i) \right)$$

(25.12)

$$= Q^{\pi_t^{-i}}(s_t, a_t^i, a_t^{-i})$$

(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 $Q^i_t$ is used:

$$\pi_{t+1}^i(a_t^i \mid s_t) = \pi_t^i(a_t^i \mid s_t) + \alpha_t Q^i_t(s_t, a_t^i, a_t^{-i})$$

(25.14)

Again, this update may require projection to ensure a valid probability distribution.

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

$$Q_{t+1}^i(s_t, a_t) = Q_t^i(s_t, a_t) + \alpha_t \left( R^i(s_t, a_t) + \gamma \max_{a_t'} Q^i_t(s_{t+1}, a_t', a_t^{-i}) - Q_t^i(s_t, a_t) \right)$$

(25.15)

We can use the inverse square-root learning rate $\alpha_t = 1/\sqrt{t}$. Exploration is also necessary. We can use an $\epsilon$-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). 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.

An agent following Nash Q-learning maintains an estimate of a joint action value function $Q(s, a)$. 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 $a$, we construct a simple game with the same number of agents and the same joint action space, but have the reward function equal to the estimated value of $s'$ such that $R(a') = Q(s', a')$. The agent computes a Nash equilibrium policy $\pi'$ over the

---

mutable struct MGGradientAscent
    ℙ # Markov game
    i # agent index
    t # time step
    Qi # state-action value estimates
    ℙ.ℐ, ℙ.𝒮, ℙ.𝒜 # joint policy
end

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

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

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

Algorithm 25.8. Gradient ascent for an agent $i$ of an MG $ℙ$. 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.
next action $a'$. Under the derived policy, the expected utility of the successor state is:

$$U(s') = \sum_{a'} Q(s', a') \prod_{j \in I} \pi_{j'}(a_{j'})$$  \hfill (25.16)

The agent then updates its value function:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left( R(s, a) + \gamma U(s') - Q(s, a) \right)$$  \hfill (25.17)

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

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

### 25.7 Summary

- Markov games 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 reward over time.

- The Nash equilibrium can be formulated for Markov games, but 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 Markov games by using a known transition function and incorporating estimates of action values.

- Policy hill climbing does not assume a model, but instead uses a gradient ascent approach to iteratively improve a stochastic policy.

- 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.
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, 𝒫, 𝑆, 𝒜 = ℙ.ℐ, ℙ.ℙ, ℙ.𝒮, ℙ.𝒜
    𝑄 = Dict((j, s, a) => 0.0 for j in ℐ, s in 𝒫, a in joint(𝒜))
    𝑁 = Dict((s, a) => 1.0 for s in 𝒫, 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, 𝒫, 𝒫, 𝒜, 𝑅, γ = ℙ.ℐ, ℙ.ℙ, ℙ.𝒮, ℙ.𝒜, ℙ.𝑅, ℙ.γ
    𝑀 = NashEquilibrium()
    ℓ = SimpleGame(γ, ℐ, 𝒜, a → [𝑄[j, s, a] for j in ℐ])
    𝜖 = 1 / sum(𝑁[𝑠, 𝑎] for 𝑎 in joint(𝒜))
    𝜖′(𝑎𝑖) = 𝜖/length(𝐴𝑖) + (1-𝜖)*π𝑖[𝑖][𝑎𝑖]
    return SimpleGamePolicy(𝑎𝑖 ⇒ 𝜖′(𝑎𝑖) for 𝑎𝑖 in 𝒜𝑖)
end

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

Algorithm 25.9. Nash Q-learning for an agent 𝑖 in an MG 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. Additionally, it uses 𝜖-greedy exploration to ensure all states and actions are explored.
25.8 Exercises

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

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

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

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

Solution: No, if given fixed policies of other agents $\pi^{-i}$, 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 a Markov game.

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 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 $\pi^i(h_i)$ is one that has a dependence on complete history $h_i = (s_{1:t}, a_{1:t-1})$. Such policies depend on the history of play of other agents. A non-stationary 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 exist Nash equilibria that are in the space of non-stationary non-Markov joint policies, stationary non-Markov joint policies, and so forth. However, it is proven that every (stationary) Markov game has a stationary Markov Nash equilibrium.

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

Solution: The algorithm presented in this chapter performs a single backup for the visited state $s$ and all joint actions $a$. 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 to only 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.