```
function simulate(𝒫::MDP, model, π, h, s)
    for i in 1:h
        a = π(model, s)
        s′, r = 𝒫.TR(s, a)
        update!(model, s, a, r, s′)
        s = s′
    end
end
```

アルゴリズム 15.9 強化学習問題のためのシミュレーション．探索方策 π は，モデル内の情報と現在の状態 s に基づいて次の行動を生成する．マルコフ決定過程問題 $\mathcal{P}$ は真の状態遷移として取り扱われ，次の状態と報酬のサンプリングに用いられる．状態遷移と報酬はモデルの更新に用いられる．シミュレーションは時間区間 h まで実行される．

15.7 要　約

- より高い報酬を得るための状態–行動空間の探索と，既知の状態–行動に関する情報活用の間のバランスにより，探索–活用のトレードオフが決定する．
- 多腕バンディット問題は，エージェントが異なる行動をとることで確率的な報酬を受け取るような単一の状態をもつ問題である．
- ベータ分布を用いて，多腕バンディットの報酬に対する信念を維持することができる．
- ε 貪欲や探索後コミットなどの非指向性探索戦略は実装が簡単だが，貪欲ではない行動探索に過去の結果からの情報を用いない．
- ソフトマックス，分位，UCB_1，および事後サンプリング探索を含む指向性探索戦略は，過去の行動からの情報を用いて有望な行動をより効率的に探索する．
- 動的計画法を用いて，有限の時間区間のための最適な探索戦略を導出することができるが，これらの戦略を求めるための計算量が多くなる可能性がある．

15.8 演　習

15.1　各腕が 0 から 1 の間で一様に抽出される勝利確率をもつ 3 本腕バンディット問題を考える．ソフトマックス，分位，および UCB_1 の探索戦略を比較せよ．質的な問題として，ランダムに生成されたバンディット問題で，λ, α, c に対してどのような値にすれば，最も高い期待報酬を得られるか．

【解】　下図では，3 つの戦略それぞれについて，ステップごとの期待報酬を図示している．パラメータ化の効果は問題の時間区間に依存するため，異なる深さに対する結果が示されている．

ソフトマックス戦略は，現在の信念に基づいて期待報酬が高い腕を引くことを優先し，大きな λ の値をもつ問題で最も良く機能する．信頼上限探索は，そのパラメータ化に関係なくより長い時間区間でより良く機能する．信頼限界 α の値は，0 または 1 に近い値を除いて，性能に大きな影響を与えない．UCB_1 戦略は，小さな正の探索パラメータ c において最も良く機能する．c が増加すると期待報酬が減少する．これら 3 つの方策は，期待報酬を最大化するように調整することができる．

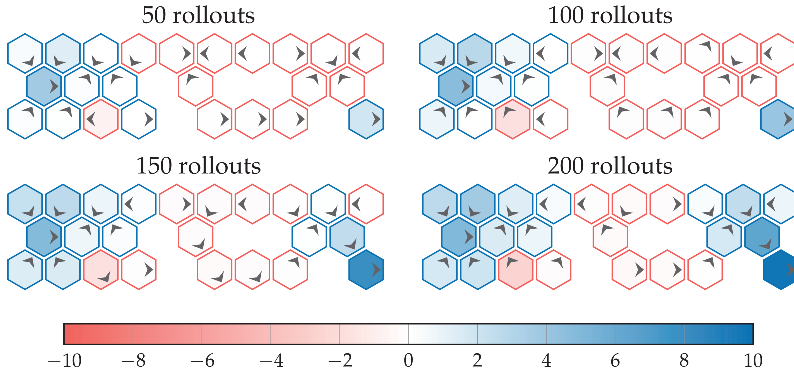


図 17.1　六角世界問題の行動価値関数を反復的に学習するために用いられる Q 学習．各状態は，Q に従ってその状態における最良行動の期待価値に応じて色分けされている．行動も同様に最良の期待価値をもつ行動である．Q 学習は $\alpha = 0.1$ および各ロールアウトにおいて 10 ステップとして実行されている．

17.3　Sarsa

Sarsa (アルゴリズム 17.3) は Q 学習の代替手段となる[6]．各ステップで Q 関数を更新するために (s, a, r, s', a') を用いることから，その名前が付けられている．Sarsa は，すべての選択可能な行動に関する最大化ではなく，次のように実際に選択した次の行動 a' を用いて Q 値を更新する．

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma Q(s',a') - Q(s,a)\right) \tag{17.11}$$

適切な探索戦略を用いると，a' は Q 学習における更新で用いられる $\arg\max_{a'} Q(s', a')$ に収束する．

[6] この方法は，以下の文献において別の名称で提案されている．G. A. Rummery and M. Niranjan, “On-Line Q-Learning Using Connectionist Systems,” Cambridge University, Tech. Rep. CUED/F-INFENG/TR 166, 1994.

```
mutable struct Sarsa
    𝒮 # state space (assumes 1:nstates)
    𝒜 # action space (assumes 1:nactions)
    γ # discount
    Q # action value function
    α # learning rate
    ℓ # most recent experience tuple (s,a,r)
end

lookahead(model::Sarsa, s, a) = model.Q[s,a]

function update!(model::Sarsa, s, a, r, s′)
    if model.ℓ != nothing
        γ, Q, α, ℓ = model.γ, model.Q, model.α, model.ℓ
        model.Q[ℓ.s,ℓ.a] += α*(ℓ.r + γ*Q[s,a] - Q[ℓ.s,ℓ.a])
    end
    model.ℓ = (s=s, a=a, r=r)
    return model
end
```

アルゴリズム 17.3　モデルフリー強化学習のための Sarsa の更新．状態–行動の価値値を含む行列 `Q` を更新する．`α` は一定の学習率であり，`ℓ` は最新の経験タプルである．Q 学習の実装と同様に，この更新関数はアルゴリズム 15.9 のシミュレータで用いることができる．

Sarsa では探索方策に従いながら，その価値を直接推定しようとするため，**オンポリシー** (on-policy) 強化学習方法の一種と分類される．対照的に，Q 学習は**オフポリシー** (off-policy) であり，探索戦略に従いながら最適方策の価値を見つけようとする．Q 学習と Sarsa は，両者とも最適な戦略に収束するが，収束の速度は適用する問題に依存する．図 17.2 に，六角世界問題で Sarsa を実行し

```
    ℓ  # most recent experience tuple (s,a,r)
end

lookahead(model::SarsaLambda, s, a) = model.Q[s,a]

function update!(model::SarsaLambda, s, a, r, s′)
    if model.ℓ != nothing
        γ, λ, Q, α, ℓ = model.γ, model.λ, model.Q, model.α, model.ℓ
        model.N[ℓ.s,ℓ.a] += 1
        δ = ℓ.r + γ*Q[s,a] - Q[ℓ.s,ℓ.a]
        for s in model.𝒮
            for a in model.𝒜
                model.Q[s,a] += α*δ*model.N[s,a]
                model.N[s,a] *= γ*λ
            end
        end
    else
        model.N[:,:] .= 0.0
    end
    model.ℓ = (s=s, a=a, r=r)
    return model
end
```

δ を Sarsa における時間差分とする．

$$\delta = r + \gamma Q(s', a') - Q(s, a) \tag{17.12}$$

行動価値関数のすべての要素が，次式を用いて更新される．

$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta N(s, a) \tag{17.13}$$

その後，次のように割引係数と指数関数的減衰パラメータを用いて訪問カウントが減衰される．

$$N(s, a) \leftarrow \gamma \lambda N(s, a) \tag{17.14}$$

特に報酬が疎な環境において適格トレースの影響が大きくなるが，報酬を得られる状況が分散されているような一般的な環境においても，このアルゴリズムによって学習を高速化することができる．

最適方策の価値を学習しようとするオフポリシーのアルゴリズム (たとえば，Q 学習) に適格トレースを適用する場合には，特に注意が必要である[9]．適格トレースは，探索方策から得られた価値を逆伝播させる．これらの不整合により，学習が不安定になる可能性がある．

[9] この問題の概説と対応方法は，以下の文献で議論される．A. Harutyunyan, M. G. Bellemare, T. Stepleton, and R. Munos, “$Q(\lambda)$ with Off-Policy Corrections,” in *International Conference on Algorithmic Learning Theory* (*ALT*), 2016.

17.5 報酬設計

報酬関数を強化することで，特に報酬が疎な問題において学習効率を向上させることができる．たとえば，単一のゴール状態への到達を目的とする場合，ゴールまでの距離に反比例する量を用いて報酬関数を補完することができる．あるいは，ゴールからどれだけ遠いかに基づいて別のペナルティを追加することもできる．たとえば，チェスゲームにおいては，ゲームの最後で勝つか負け

ペア (s,a) のサンプルを用いて，式 (17.18) の更新則を次のように近似することができる．

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(Q^*(s,a) - Q_{\boldsymbol{\theta}}(s,a))\nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s,a) \tag{17.19}$$

式 (17.19) の計算には，ここで知りたいと考えている最適方策の情報が必要であり，直接計算することはできない．その代わりとして，次のように観測された遷移と行動価値の近似からそれを推定する．

$$Q^*(s,a) \approx r + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(s',a') \tag{17.20}$$

これに基づけば，次の更新則を得る．

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(r + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(s',a') - Q_{\boldsymbol{\theta}}(s,a))\nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s,a) \tag{17.21}$$

この更新則は，アルゴリズム 17.5 で実装され，スケール化された勾配ステップ (アルゴリズム 12.2) が追加されている．これは，勾配ステップを必要以上に大きくしないようにするために用いられることがある．例 17.3 では，この更新則を線形行動価値近似とあわせて用いる方法を示している．図 17.3 は，このアルゴリズムをマウンテンカー問題に適用したものを示している．

```julia
struct GradientQLearning
    𝒜  # action space (assumes 1:nactions)
    γ  # discount
    Q  # parameterized action value function Q(θ,s,a)
    ∇Q # gradient of action value function
    θ  # action value function parameter
    α  # learning rate
end

function lookahead(model::GradientQLearning, s, a)
    return model.Q(model.θ, s,a)
end

function update!(model::GradientQLearning, s, a, r, s′)
    𝒜, γ, Q, θ, α = model.𝒜, model.γ, model.Q, model.θ, model.α
    u = maximum(Q(θ,s′,a′) for a′ in 𝒜)
    Δ = (r + γ*u - Q(θ,s,a))*model.∇Q(θ,s,a)
    θ[:] += α*scale_gradient(Δ, 1)
    return model
end
```

アルゴリズム 17.5　行動価値関数近似を伴う Q 学習における更新．新しい経験タプル `s`, `a`, `r`, `s′` を得るたびに，一定の学習率 `α` でベクトル `θ` を更新する．パラメトリック行動価値関数は `Q(θ,s,a)` によって与えられ，その勾配は `∇Q(θ,s,a)` となる．

$\gamma = 1$ の単純レギュレータ問題に線形行動価値近似を用いた Q 学習を適用する．行動価値近似は $Q_{\boldsymbol{\theta}}(s,a) = \boldsymbol{\theta}^\top \boldsymbol{\beta}(s,a)$ であり，基底関数は

$$\boldsymbol{\beta}(s,a) = [s, s^2, a, a^2, 1]$$

となる．この線形モデルでは，勾配は次式となる．

$$\nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s,a) = \boldsymbol{\beta}(s,a)$$

問題 $\mathcal{P}$ に対して，これを次のように実装することができる．

例 17.3　シミュレーションにおける行動価値関数近似を伴う Q 学習付き探索戦略の使用法．パラメータ設定は架空のものである．

```
β(s,a) = [s,s^2,a,a^2,1]
Q(θ,s,a) = dot(θ,β(s,a))
∇Q(θ,s,a) = β(s,a)
θ = [0.1,0.2,0.3,0.4,0.5] # initial parameter vector
α = 0.5 # learning rate
model = GradientQLearning(𝒫.𝒜, 𝒫.γ, Q, ∇Q, θ, α)
ε = 0.1 # probability of random action
π = EpsilonGreedyExploration(ε)
k = 20 # number of steps to simulate
s = 0.0 # initial state
simulate(𝒫, model, π, k, s)
```

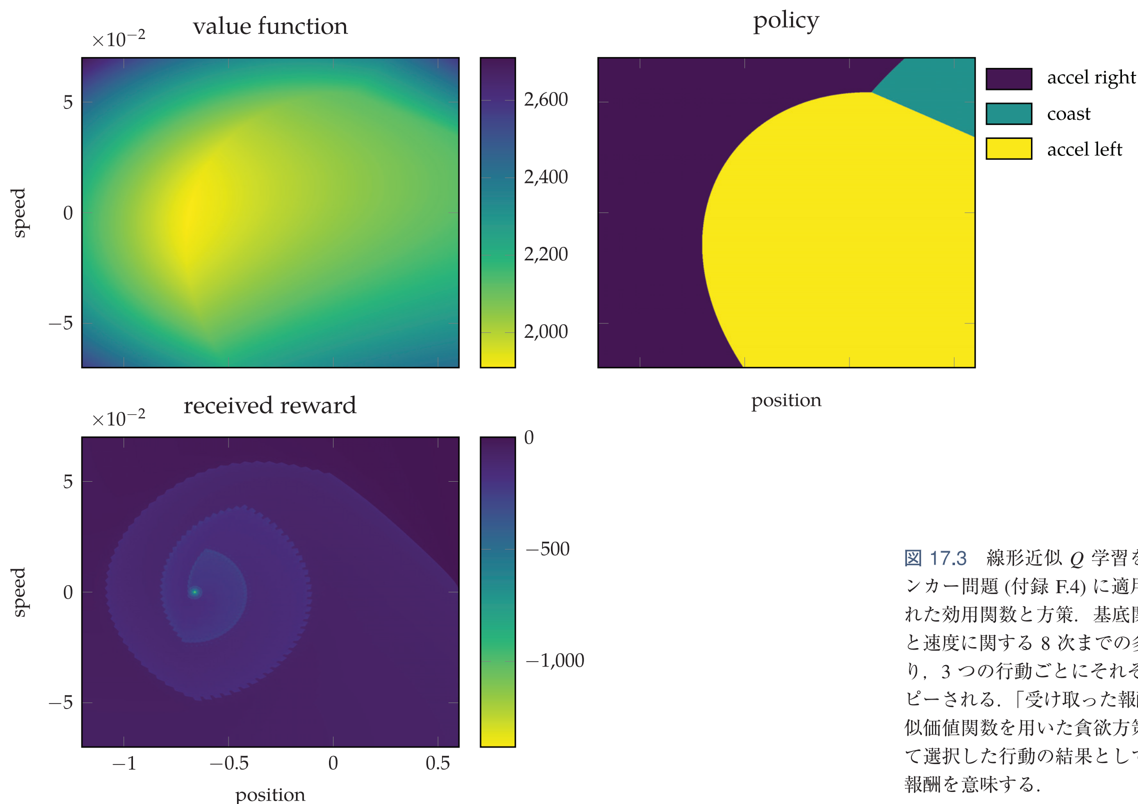


図 17.3　線形近似 Q 学習をマウンテンカー問題 (付録 F.4) に適用して得られた効用関数と方策．基底関数は位置と速度に関する 8 次までの多項式であり，3 つの行動ごとにそれぞれ 3 回コピーされる．「受け取った報酬」は，近似価値関数を用いた貪欲方策に基づいて選択した行動の結果として得られた報酬を意味する．

17.7　経 験 再 生

強化学習で大域的関数近似を用いたい場合の大きな課題として，**破壊的忘却** (catastrophic forgetting) がある．たとえば，ある特定の方策によって報酬の低い状態空間領域に推移することが最初にわかった場合，その後，その領域を避けるように方策を更新する．しかし，一定以上の時間が経過した後，状態空間のその領域を避けるべき理由を忘却し，性能の悪い方策に戻るリスクがある．

破壊的忘却は，**経験再生** (experience replay) [14] によって緩和される可能性が

[14] 経験再生は，以下の文献の研究において重要な役割を果たした．V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing Atari with Deep Reinforcement Learning,” 2013. arXiv: 1312.5602v1. この考え方は，以下の文献により先行して発表されている．L. -J. Lin, “Reinforcement Learning for Robots Using Neural Networks,” Ph.D. dissertation, Carnegie Mellon University, 1993.

ある．ここでは，固定数の直近の経験タプルが訓練の反復全体にわたって保存される．**再生メモリ** (replay memory) から一様にサンプリングされたタプルの**バッチ** (batch) が，これまでに性能が悪いと評価された戦略を避けるために用いられる[15]．更新式は，式 (17.21) から，

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \frac{1}{m_{\text{grad}}} \sum_i (r^{(i)} + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(s'^{(i)}, a') - Q_{\boldsymbol{\theta}}(s^{(i)}, a^{(i)})) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s^{(i)}, a^{(i)}) \tag{17.22}$$

に変更される．ここで $s^{(i)}, a^{(i)}, r^{(i)}, s'^{(i)}$ は，大きさ m_{grad} のランダムバッチを構成する i 番目の経験タプルである．

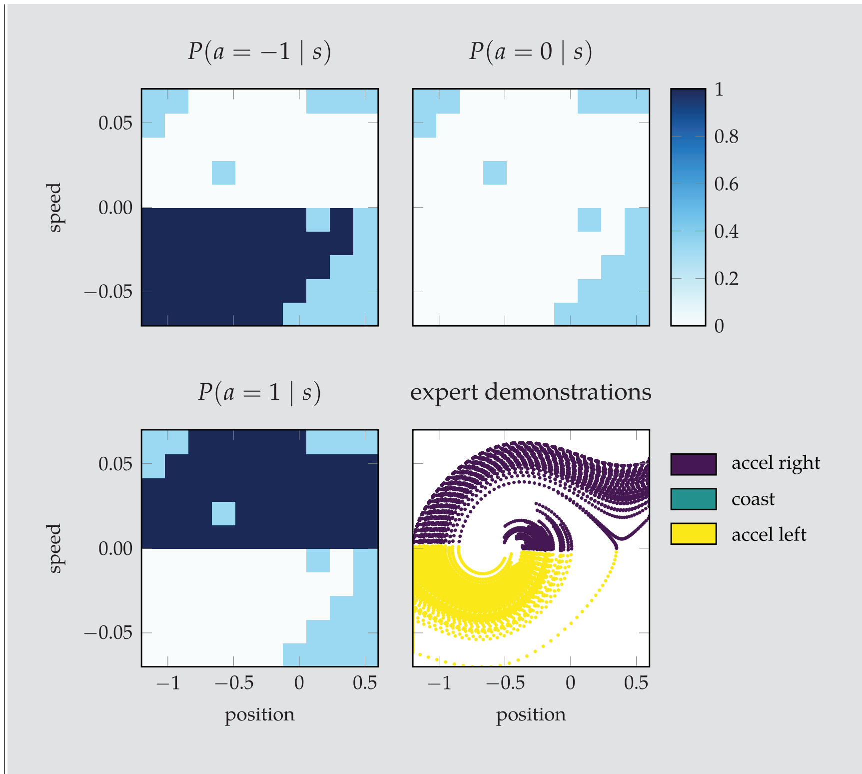
経験再生により，経験タプルを学習に複数回用いることとなり，データの効率性を向上させる．さらに，再生メモリからランダムに一様サンプリングすることで，ロールアウトから得られる相関のある連続する経験タプルを分解し，勾配推定の分散を減少させる．経験再生により，過去の方策パラメータからの情報を保持することで学習過程を安定化させることができる．

アルゴリズム 17.6 は，行動価値関数近似を用いた Q 学習に経験再生を組み込むメソッドを示している．例 17.4 は，このメソッドを単純レギュレータ問題にいかに適用するかを示している．

[15] 以下の文献では，経験を優先する経験再生が提案されている．T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized Experience Replay,” in *International Conference on Learning Representations (ICLR)*, 2016.

```
struct ReplayGradientQLearning
    𝒜      # action space (assumes 1:nactions)
    γ      # discount
    Q      # parameterized action value function Q(θ,s,a)
    ∇Q     # gradient of action value function
    θ      # action value function parameter
    α      # learning rate
    buffer # circular memory buffer
    m      # number of steps between gradient updates
    m_grad # batch size
end

function lookahead(model::ReplayGradientQLearning, s, a)
    return model.Q(model.θ, s,a)
end

function update!(model::ReplayGradientQLearning, s, a, r, s′)
    𝒜, γ, Q, θ, α = model.𝒜, model.γ, model.Q, model.θ, model.α
    buffer, m, m_grad = model.buffer, model.m, model.m_grad
    if isfull(buffer)
        U(s) = maximum(Q(θ,s,a) for a in 𝒜)
        ∇Q(s,a,r,s′) = (r + γ*U(s′) - Q(θ,s,a))*model.∇Q(θ,s,a)
        Δ = mean(∇Q(s,a,r,s′) for (s,a,r,s′) in rand(buffer, m_grad))
        θ[:] += α*scale_gradient(Δ, 1)
        for i in 1:m # discard oldest experiences
            popfirst!(buffer)
        end
    else

        push!(buffer, (s,a,r,s′))
    end
    return model
end
```

アルゴリズム 17.6　関数近似と経験再生を用いた Q 学習．更新はパラメータ化方策 `Q(θ,s,a)` と勾配 `∇Q(θ,s,a)` に依存する．パラメータベクトル `θ` と `DataStructures.jl` によって提供される循環的メモリバッファを更新する．`θ` は `m` ステップごとにバッファからの `m_grad` 個のサンプルから推定される勾配を用いて更新される．



方策がファクタに分解できる場合，ベイズネットワークを用いて状態–行動の変数上の同時分布を表すことができる．図 18.1 はその例を示している．データ $\mathcal{D}$ から構造 (5 章) とパラメータ (4 章) の両方を学習することができる．現在の状態が与えられたとき，以前に議論した推論アルゴリズムのいずれか (3 章) を用いて行動上の分布を推論することができる．

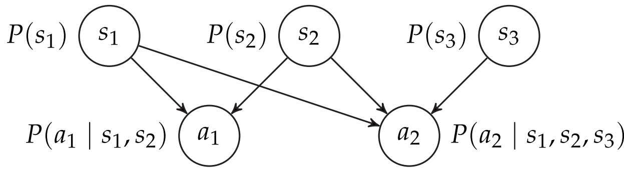


図 18.1　ベイズネットワークを用いて状態–行動変数上の同時分布を表すことができる．現在の状態変数の値が与えられたとき，行動分布を生成するために推論アルゴリズムを適用することができる．

π_θ に対して，他の表現を用いることもできる．たとえば，ニューラルネットワークを用いる場合，入力は状態変数の値に対応し，出力は行動空間上の分布のパラメータに対応する．表現が微分可能である場合 (ニューラルネットワークの場合)，勾配上昇を用いて式 (18.1) を最適化することができる．この方法はアルゴリズム 18.1 で実装される．

```
struct BehavioralCloning
    α     # step size
    k_max # number of iterations
    ∇logπ # log likelihood gradient
end
```

アルゴリズム 18.1　状態–行動タプルの集合 `D` の形式のエキスパートのデモンストレーションにより学習された，パラメータ化された確率的方策．方策のパラメータ化ベクトル `θ` は，状態が与えられた場合の行動の対数尤度を最大化することによって反復的に改善される．行動クローニングにはステップサイズ `α`，反復回数 `k_max`，および対数尤度勾配`∇logπ` が必要である．

```
function optimize(M::BehavioralCloning, D, θ)
    α, k_max, ∇logπ = M.α, M.k_max, M.∇logπ
    for k in 1:k_max
        ∇ = mean(∇logπ(θ, a, s) for (s,a) in D)
        θ += α*∇
    end
    return θ
end
```

エキスパートのデモンストレーションが最適に近いほど，結果として得られる行動クローニング方策の性能が向上する[3]．しかし，行動クローニングは**連鎖的誤差** (cascading error) によってパフォーマンス低下が引き起こされることがある．例 18.2 で議論されているように，ロールアウトにおける細かい不正確さが蓄積され，最終的には訓練データに十分に示されていない状態に遷移してしまうことによって，不適切な行動選択，最終的には無効または未確認の状況に遷移してしまうこととなる．行動クローニングはその手続きの単純さが魅力的だが，多くの問題において連鎖的誤差によって性能が低下し，特に長期の時間範囲で方策が用いられる場合には，その影響が大きくなる．

[3] U. Syed and R. E. Schapire, “A Reduction from Apprenticeship Learning to Classification,” in *Advances in Neural Information Processing Systems* (*NIPS*), 2010.

自動運転のレースカーの方策を訓練するのに行動クローニングを適用することを考える．人間のレースカードライバーがエキスパートのデモンストレーションを提供する．ドライバーはエキスパートであるため，芝生の上で滑ったり，レールに近づきすぎたりすることは絶対にない．行動クローニングで訓練されたモデルは，レールに近づいたり芝生の上で滑ったりした場合に用いるための情報が存在せず，そこから回復する方法がわからない．

例 18.2　行動クローニング固有の一般的な問題に関する簡単な例

18.2　データ集合集約

連鎖的誤差の問題に対処するための方法の 1 つとして，追加的なエキスパートによる入力を用いて，方策を修正することが考えられている．**逐次的対話型デモンストレーション** (sequential interactive demonstration) 法は，訓練された方策によって生成された状態において，エキスパートからデータを収集し，これを用いてその方策を改善するということを交互に行うものである．

逐次的対話型デモンストレーション法の一種として，**データ集合集約** (data set aggregation: DAgger) (アルゴリズム 18.2) [4] と呼ばれるものがある．これは，行動クローニングを用いて確率的方策を訓練することから始める．この方策を用いて初期状態分布 b から複数のロールアウトを実行し，それをエキスパートに提供し，各状態に対する正しい行動を提供してもらう．新しいデータは以前のデータ集合と集約され，新しい方策が訓練される．例 18.3 ではこの過程を説明している．

[4] S. Ross, G. J. Gordon, and J. A. Bagnell, “A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning,” in *International Conference on Artificial Intelligence and Statistics* (*AISTATS*), vol. 15, 2011.

```julia
struct SMILe
    𝒫     # problem with unknown reward
    bc    # Behavioral cloning struct
    k_max # number of iterations
    m     # number of rollouts per iteration
    d     # rollout depth
    b     # initial state distribution
    β     # mixing scalar (e.g., d^-3)
    πE    # expert policy
    πθ    # parameterized policy
end

function optimize(M::SMILe, θ)
    𝒫, bc, k_max, m = M.𝒫, M.bc, M.k_max, M.m
    d, b, β, πE, πθ = M.d, M.b, M.β, M.πE, M.πθ
    𝒜, T = 𝒫.𝒜, 𝒫.T
    θs = []
    π = s → πE(s)
    for k in 1:k_max
        # execute latest π to get new data set D
        D = []
        for i in 1:m
            s = rand(b)
            for j in 1:d
                push!(D, (s, πE(s)))
                a = π(s)
                s = rand(T(s, a))
            end
        end
        # train new policy classifier
        θ = optimize(bc, D, θ)
        push!(θs, θ)
        # compute a new policy mixture
        Pπ = Categorical(normalize([(1-β)^(i-1) for i in 1:k],1))
        π = s → begin
            if rand() < (1-β)^(k-1)
                return πE(s)
            else
                return rand(Categorical(πθ(θs[rand(Pπ)], s)))
            end
        end
    end
    Ps = normalize([(1-β)^(i-1) for i in 1:k_max],1)
    return Ps, θs
end
```

アルゴリズム 18.3　マルコフ決定過程 $\mathcal{P}$ に対して，エキスパートのデモンストレーションから確率的パラメータ化方策を訓練するための SMILe．新しいコンポーネント方策を小さい重みで少しずつ混合し，同時にエキスパート方策に従って行動する確率を減少させる．このメソッドは，コンポーネント方策の確率 `Ps` とパラメータ `θs` を返す．

最初はエキスパート方策である $\pi^{(1)} = \pi_E$ [6] から始める．各反復において，最新の方策 $\pi^{(k)}$ を実行して新しいデータ集合を生成し，エキスパートに正しい行動を提供するよう求める．行動クローニングは，この新しいデータ集合にのみ適用され，新しい**コンポーネント方策** (component policy) $\hat{\pi}^{(k)}$ を訓練する．このコンポーネント方策は，以前の反復からのコンポーネント方策と混合され新しい方策 $\pi^{(k+1)}$ を生成する．

コンポーネント方策を混合して $\pi^{(k+1)}$ を生成するための混合比率 β は $(0, 1)$ の範囲にある．エキスパート方策に従って行動する確率は，$(1-\beta)^k$ で，$\hat{\pi}^{(i)}$ に

[6] エキスパート方策 π_E の明示的な表現はない．π_E を評価するためには，前節で述べたようにエキスパートに対して，繰り返し問い合わせて新たな知識を提供してもらう必要がある．

組み合わせる．確率 λ_i で，方策 $\pi^{(i)}$ に従って行動選択する．アルゴリズム 18.5 は最大マージン逆強化学習の実装である．

```
function calc_weighting(M::InverseReinforcementLearning, μs)
    μE = M.μE
    k = length(μE)
    model = Model(Ipopt.Optimizer)
    @variable(model, t)
    @variable(model, ϕ[1:k] ≥ 0)
    @objective(model, Max, t)
    for μ in μs
        @constraint(model, ϕ⋅μE ≥ ϕ⋅μ + t)
    end
    @constraint(model, ϕ⋅ϕ ≤ 1)
    optimize!(model)
    return (value(t), value.(ϕ))
end

function calc_policy_mixture(M::InverseReinforcementLearning, μs)
    μE = M.μE
    k = length(μs)
    model = Model(Ipopt.Optimizer)
    @variable(model, λ[1:k] ≥ 0)
    @objective(model, Min, (μE - sum(λ[i]*μs[i] for i in 1:k))⋅
                           (μE - sum(λ[i]*μs[i] for i in 1:k)))
    @constraint(model, sum(λ) == 1)
    optimize!(model)
    return value.(λ)
end

function optimize(M::InverseReinforcementLearning, θ)
    π, ε, RL = M.π, M.ε, M.RL
    θs = [θ]
    μs = [feature_expectations(M, s→π(θ,s))]
    while true
        t, ϕ = calc_weighting(M, μs)
        if t ≤ ε
            break
        end
        copyto!(RL.ϕ, ϕ) # R(s,a) = π⋅β(s,a)
        θ = optimize(RL, π, θ)
        push!(θs, θ)
        push!(μs, feature_expectations(M, s→π(θ,s)))
    end
    λ = calc_policy_mixture(M, μs)
    return λ, θs
end
```

アルゴリズム 18.5　最大マージン逆強化学習により，特定のエキスパートのデモンストレーションの特徴期待値に整合する混合方策を計算する．`JuMP.jl` を用いて制約付き最適化問題を解く．ここでの実装では，提供された強化学習構造体が，新しい値を用いて更新することのできる重みベクトル `ϕ`をもつ必要がある．このメソッドは，コンポーネント方策の確率的な重み付け `λ` とパラメータ `θs` を返す．

18.5　エントロピー最大化逆強化学習

前節の逆強化学習では，エキスパートのデモンストレーションと同じ特徴期待値を生成してしまい，しばしば複数の方策が存在するという意味で，方策の特定が不十分であった．本節では，**エントロピー最大化逆強化学習** (maximum entropy inverse reinforcement learning) を紹介する．これは，軌跡上の分布が最

報酬関数が線形で，$R_{\boldsymbol{\phi}}(s,a) = \boldsymbol{\phi}^\top \boldsymbol{\beta}(s,a)$ であれば，前節のように $\nabla_{\boldsymbol{\phi}} R_{\boldsymbol{\phi}}(s,a)$ は単純に $\boldsymbol{\beta}(s,a)$ となる．

パラメータベクトル $\boldsymbol{\phi}$ の更新には，割引された状態訪問頻度 $b_{\gamma,\boldsymbol{\phi}}$ と，現在のパラメータベクトル下での最適方策 $\pi_{\boldsymbol{\phi}}(a \mid s)$ の両方が必要である．最適方策は強化学習を実行することによって得られる．割引された状態訪問頻度を計算するため，ロールアウトを用いるか，動的計画法を適用することができる．

動的計画法を用いて割引された状態訪問頻度を計算する場合，初期状態分布を $b^{(1)}_{\gamma\boldsymbol{\phi}} = b(s)$ として，次のように時間経過とともに反復的に更新する．

$$b^{(k+1)}_{\gamma,\boldsymbol{\phi}}(s) = \gamma \sum_a \sum_{s'} b^{(k)}_{\gamma,\boldsymbol{\phi}}(s)\pi(a \mid s) T(s' \mid s,a) \tag{18.18}$$

このようなエントロピー最大化逆強化学習が，アルゴリズム 18.6 で実装される．

```
struct MaximumEntropyIRL
    𝒫     # problem
    b     # initial state distribution
    d     # depth
    π     # parameterized policy π(θ,s)
    Pπ    # parameterized policy likelihood π(θ, a, s)
    ∇R    # reward function gradient
    RL    # reinforcement learning method
    α     # step size
    k_max # number of iterations
end

function discounted_state_visitations(M::MaximumEntropyIRL, θ)
    𝒫, b, d, Pπ = M.𝒫, M.b, M.d, M.Pπ
    𝒮, 𝒜, T, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.T, 𝒫.γ
    b_sk = zeros(length(𝒫.𝒮), d)
    b_sk[:,1] = [pdf(b, s) for s in 𝒮]
    for k in 2:d
        for (si′, s′) in enumerate(𝒮)
            b_sk[si′,k] = γ*sum(sum(b_sk[si,k-1]*Pπ(θ, a, s)*T(s, a,
     s′)
                for (si,s) in enumerate(𝒮))
            for a in 𝒜)
        end
    end
    return normalize!(vec(mean(b_sk, dims=2)),1)
end

function optimize(M::MaximumEntropyIRL, D, ϕ, θ)
    𝒫, π, Pπ, ∇R, RL, α, k_max = M.𝒫, M.π, M.Pπ, M.∇R, M.RL, M.α, M
    .k_max
    𝒮, 𝒜, γ, nD = 𝒫.𝒮, 𝒫.𝒜, 𝒫.γ, length(D)
        for k in 1:k_max
        copyto!(RL.ϕ, ϕ) # update parameters
        θ = optimize(RL, π, θ)
        b = discounted_state_visitations(M, θ)
        ∇Rτ = τ → sum(γ^(i-1)*∇R(ϕ,s,a) for (i,(s,a)) in enumerate(
    τ))
        ∇f = sum(∇Rτ(τ) for τ in D) - nD*sum(b[si]*sum(Pπ(θ,a,s)*∇R(
    ϕ,s,a)
                        for (ai,a) in enumerate(𝒜))
                for (si, s) in enumerate(𝒮))
```

アルゴリズム 18.6　エントロピー最大化逆強化学習．これは，最大エントロピー軌跡分布のもとでエキスパートのデモンストレーションの尤度を最大化する確率的方策を見つける方法である．この実装では，すべての状態に対して動的計画法を用いて予測される訪問数を計算するため，その問題が離散的であることが必要である．

18.7 要　約

- 模倣学習では，報酬関数を用いずにエキスパートのデモンストレーションから適切な行動を学習する．
- 模倣学習の一種として行動クローニングがあり，これはデータ集合に含まれる行動の条件付き尤度最大化の確率的方策を生成する．
- エキスパートに何度か相談できるようであれば，DAgger や SMILe などの反復的アプローチを適用できる．
- 逆強化学習は，エキスパートのデータから報酬関数を推論し，その後，最適方策を発見させるための従来の方法を用いる．
- マージン最大化逆強化学習は，エキスパートのデータ集合に含まれる二値の特徴量の頻度に整合する方策の発見を目的とする．
- エントロピー最大化逆強化学習は，最良の報酬パラメータの発見する問題を最大尤度推定問題として定式化し，これを勾配上昇を用いて解く．
- 生成的敵対的模倣学習は，識別器と方策を反復的に最適化する．識別器は方策による行動選択とエキスパートによる決定を区別しようとしているのに対して，方策は識別器が区別できないようにする．

18.8 演　習

18.1　エキスパートのデモンストレーションが与えられた離散問題に，行動クローニングを適用する．特徴量関数 $\boldsymbol{\beta}(s)$ を定義し，次のようにソフトマックス分布を用いて方策を表現することができる．

$$\pi(a \mid s) \propto \exp(\boldsymbol{\theta}_a^\top \boldsymbol{\beta}(s))$$

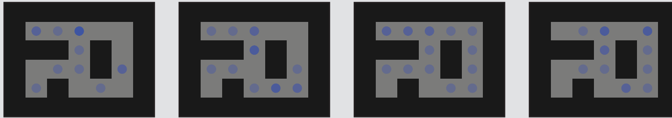
エキスパートデータから各行動に関するパラメータ $\boldsymbol{\theta}_a$ を学習することになる．各状態–行動のペアごとに 1 つのパラメータをもつ離散分布を直接推定するよりも，この方法が適切である理由を述べよ．

【解】　模倣学習では，一般的にエキスパートのデモンストレーションの比較的小さな集合に限定される．分布 $P(a \mid s)$ には $(|\mathcal{A}|-1)|\mathcal{S}|$ の独立したパラメータが必要であり，膨大な量になることがある．エキスパートのデモンストレーションは，一般的に状態空間のごく一部に対応する．$P(a \mid s)$ が提供されたデータ集合に含まれる状態に対して十分に訓練されたとしても，その結果の方策は他の状態では未訓練となる．特徴量関数を用いることで，未知の状態に対応するように一般化できる．

18.2　18.1 節では，エキスパートのデータから方策を訓練するために最尤法を用いることを示した．この方法は，訓練サンプルに割り当てられた尤度を最大化する方策のパラメータを発見しようとしている．しかし，ある正しくない行動に誤って高い確率を割り当てることが，他の正しくない行動に高い選択確率を割り当てることよりも不適切であることがある．たとえば，マウンテンカー問題でエキスパートが加速度を 1 と指示している場合に，加速度が −1 であると予測することは，加速度を 0 と予測することよりも不適切である．行動クローニングを修正することで，異なる誤分類に対して異なるペナルティを与えることが可能か．

$w_{slow} = 1.0$ $w_{slow} = 0.99$ $w_{slow} = 0.98$ $w_{slow} = 0.97$
$w_{fast} = 1.0$ $w_{fast} = 0.7$ $w_{fast} = 0.49$ $w_{fast} = 0.34$

$w_{slow} = 0.96$ $w_{slow} = 0.95$ $w_{slow} = 0.94$ $w_{slow} = 0.93$
$w_{fast} = 0.24$ $w_{fast} = 0.17$ $w_{fast} = 0.12$ $w_{fast} = 0.1$



反復は左から右へ，上から下に進む．それぞれの青い点は粒子フィルタ内の粒子を表し，グリッドのその位置に存在するという部分的な信念に対応する．

19.8 要 約

- 部分観測マルコフ決定過程 (POMDP) は状態の不確実性を含むようにマルコフ決定過程を拡張する．
- 不確実性のため，部分観測マルコフ決定過程のエージェントは自身の状態上の信念を保持する必要がある．
- 離散状態空間をもつ部分観測マルコフ決定過程に対する信念は，カテゴリカル分布を使用して表現でき，解析的に更新できる．
- 線形ガウス部分観測マルコフ決定過程の信念はガウス分布を使用して表すことができ，解析的に更新することもできる．
- 非線形連続部分観測マルコフ決定過程に対する信念はガウス分布を使用して表すこともできるが，通常は解析的に更新できない．この場合，拡張カルマンフィルタやアンセンテッドカルマンフィルタが使用される．
- 連続問題は，線形ガウスであるという仮定に基づいて，モデル化できる場合がある．
- 粒子フィルタは，状態粒子の大きな集まりを使用して，信念を近似する．

19.9 演 習

19.1 すべてのマルコフ決定過程は部分観測マルコフ決定過程として構成できるか．
【解】 できる．部分観測マルコフ決定過程の定式化は，観測分布の形式で状態の不確実性を導入することによって，マルコフ決定過程の定式化を拡張する．任意のマルコフ決定過程は，$\mathcal{O} = \mathcal{S}$ および $O(o \mid a, s') = (o = s')$ をもつ部分観測マルコフ決定過程として構成できる．

19.2 観測のない離散的部分観測マルコフ決定過程に関する信念更新とは何か．観測のない線形ガウスダイナミクスをもつ部分観測マルコフ決定過程の信念更新とは何か．

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

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

1 ステップ先読み (one-step lookahead) を使用する場合，Γ のアルファベクトルに関連付けられた行動を追跡する必要はない．Γ で表される価値関数を $\mathcal{U}^\Gamma$ と表記すると，これを用いた信念 b からの 1 ステップ先読み行動は

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

となり，ここで，

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

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

である．アルゴリズム 20.5 はこれの実装を提供している．例 20.2 は泣いている赤ちゃん問題で 1 ステップ先読みの使用を例示している．

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

function greedy(𝒫::POMDP, U, b::Vector)
    u, a = findmax(a→lookahead(𝒫, U, b, a), 𝒫.𝒜)
    return (a=a, u=u)
end

struct LookaheadAlphaVectorPolicy
    𝒫 # POMDP problem
    Γ  # alpha vectors
end

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

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

(π::LookaheadAlphaVectorPolicy)(b) = greedy(π, b).a
```

アルゴリズム 20.5　アルファベクトルの集合 Γ によって表される方策．これは最適行動と関連付けられた効用を生成するために，1 ステップ先読みを用いる．式 (20.13) は先読みを計算するために使用される．

ズム 20.6 は，もし存在するならば，δ が最大の正の値であるときの信念を決定するために，式 (20.16) を解くための実装を提供する．

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

アルゴリズム 20.6　アルファベクトル α がアルファベクトルの集合 Γ と比較して，最も改善する場合の信念ベクトル b を見つけるメソッド．そのような信念が存在しないならば，何も返されない．JuMP.jl パッケージと GLPK.jl パッケージは，それぞれ数理最適化のフレームワークと線形計画法のソルバを提供する．

アルゴリズム 20.7 は，アルゴリズム 20.6 を用いて集合 Γ 内で支配しているアルファベクトルを見つける手続きを示している．最初は，すべてのアルファベクトルが支配しているものの候補である．次に，これらの候補の 1 つを選択し，信念 b を決定するが，この場合その候補は，支配しているアルファベクトルの集合における他のすべてのアルファベクトルと比較して，値を最大限に改善するものである．その候補が改善をもたらさないならば，それを集合から取り除く．それが改善をもたらすならば，b において最大の改善をもたらす候補集合からアルファベクトルを支配集合に移す．この過程は候補がなくなるまで続く．どの信念点においても支配していないアルファベクトルと関連付けられた条件付きプランを取り除くことができる．例 20.3 は泣いている赤ちゃん問題に対する枝刈りを示している．

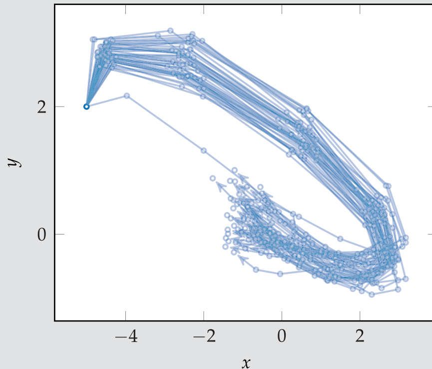
```
function find_dominating(Γ)
    n = length(Γ)
    candidates, dominating = trues(n), falses(n)
    while any(candidates)
        i = findfirst(candidates)
        b = find_maximal_belief(Γ[i], Γ[dominating])
        if b === nothing
            candidates[i] = false
        else
            k = argmax([candidates[j]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

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

アルゴリズム 20.7　支配されたアルファベクトルと関連付けられたプランを枝刈りするメソッド．find_domination 関数は集合 Γ 内のすべてを支配しているアルファベクトルを同定する．二値ベクトル candidates と dominating を使用して，どのアルファベクトルが支配集合に含める候補であるか，またどれが現在支配集合内にあるかを追跡する．

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

に従って計測する．ここで，$\boldsymbol{\varepsilon}$ はゼロ平均ガウスノイズで，共分散が $\Delta t/10\boldsymbol{I}$ である．

$\Delta t = 1$ の最適方策と信念を追跡するカルマンフィルタを用いて，10 ステップのロールアウトからの 50 の軌跡をここに示す．いずれの場合も，衛星は $\boldsymbol{s} = \boldsymbol{\mu}_b = [-5, 2, 0, 1]$ と $\boldsymbol{\Sigma}_b = [\boldsymbol{I}\ \boldsymbol{0}; \boldsymbol{0}\ 0.25\boldsymbol{I}]$ で開始された．



20.7 要　約

- 部分観測マルコフ決定過程の正確な解は，通常有限時間区間の離散部分観測マルコフ決定過程に対してのみ取得できる．
- これらの問題に対する方策は，観測に基づいて実行する行動を記述するツリーである条件付きプランとして表現できる．
- アルファベクトルは，異なる状態から始まり，特定の条件付きプランに従う場合の期待効用を含んでいる．
- アルファベクトルは部分観測マルコフ決定過程の方策の代替的表現としても機能する．
- 部分観測マルコフ決定過程の価値反復では，部分プランを反復的に計算し，準最適である部分プランを枝刈りすることによって，すべての条件付きプランを列挙する計算の負荷を回避できる．
- 2 次の報酬をもつ線形ガウス問題は，完全に観測可能な場合に関して導出された方法と非常に類似した方法を使用して正確に解くことができる．

20.8 演　習

20.1　すべての部分観測マルコフ決定過程をマルコフ決定過程として構成できるか．

【解】　できる．任意の部分観測マルコフ決定過程は，等価的に信念状態マルコフ決定過程とみなすことができ，その状態空間は部分観測マルコフ決定過程における信念の空間であり，その行動空間は部分観測マルコフ決定過程の行動空間と同じであり，そ

題ではうまく機能する．

小さな離散状態空間をもたないかもしれない問題に対して，Q マルコフ決定過程の方法を一般化できる．このような問題では，式 (21.1) の反復は実行可能ではないかもしれないが，近似的な行動価値関数 $Q(s,a)$ を得るためのこれまでの章で論じた多くの方法の 1 つを使用できる．この価値関数は，たとえばニューラルネットワーク表現を用いて，高次元の連続状態空間上で定義される．信念の点で評価された価値関数は

$$\mathcal{U}(b) = \max_a \int Q(s,a)b(s)\mathrm{d}s \tag{21.3}$$

となる．上の積分はサンプリングを通じて近似することができる．

21.2 高速情報限度

Q マルコフ決定過程と同様に，**高速情報限度** (fast informed bound) は各行動に対して 1 つのアルファベクトルを計算する．しかし高速情報限度では，観測モデルがある程度まで考慮される[5]．繰返しの計算は

$$\alpha_a^{(k+1)}(s) = R(s,a) + \gamma \sum_o \max_{a'} \sum_{s'} O(o \mid a, s') T(s' \mid s, a) \alpha_{a'}^{(k)}(s') \tag{21.4}$$

となり，各反復で，$\mathcal{O}(|\mathcal{A}|^2|\mathcal{S}|^2|\mathcal{O}|)$ の操作を必要とする．

高速情報限度は最適価値関数の上限を提供する．この上限は Q マルコフ決定過程によって提供される上限よりも緩くならないことが保証されており，より厳しくなる傾向もある．高速情報限度はアルゴリズム 21.3 に実装されており，図 21.2 では最適価値関数を計算するために使用される．

[5] Q マルコフ決定過程と高速情報限度の関係は実験結果とともに次の文献で論じられる．M. Hauskrecht, "Value-Function Approximations for Partially Observable Markov Decision Processes," *Journal of Artificial Intelligence Research*, vol. 13, pp. 33–94, 2000.

```
struct FastInformedBound
    k_max # maximum number of iterations
end

function update(𝒫::POMDP, M::FastInformedBound, Γ)
    𝒮, 𝒜, 𝒪, R, T, O, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.R, 𝒫.T, 𝒫.O, 𝒫.γ
    Γ′ = [[R(s, a) + γ*sum(maximum(sum(O(a,s′,o)*T(s,a,s′)*α′[j]
        for (j,s′) in enumerate(𝒮)) for α′ in Γ) for o in 𝒪)
        for s in 𝒮] for a in 𝒜]
    return Γ′
end

function solve(M::FastInformedBound, 𝒫::POMDP)
    Γ = [zeros(length(𝒫.𝒮)) for a in 𝒫.𝒜]
    Γ = alphavector_iteration(𝒫, M, Γ)
    return AlphaVectorPolicy(𝒫, Γ, 𝒫.𝒜)
end
```

アルゴリズム 21.3　高速情報限度．これは，離散的な状態，行動，観測空間をもつ無限時間区間部分観測マルコフ決定過程に近似的な最適方策を見つける．`k_max` は反復回数である．

21.3 高速下限

前の 2 つの節では，アルファベクトルとして表現される価値関数の上限を生成するために使用できる手法を紹介した．本節では，信念空間におけるプラン

```
function backup(𝒫::POMDP, Γ, b)
    𝒮, 𝒜, 𝒪, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.γ
    R, T, O = 𝒫.R, 𝒫.T, 𝒫.O
    Γa = []
    for a in 𝒜
        Γao = []
        for o in 𝒪
            b′ = update(b, 𝒫, a, o)
            push!(Γao, argmax(α→α⋅b′, Γ))
        end
        α = [R(s, a) + γ*sum(sum(T(s, a, s′)*O(a, s′, o)*Γao[i][j]
            for (j,s′) in enumerate(𝒮)) for (i,o) in enumerate(𝒪))
            for s in 𝒮]
        push!(Γa, α)
    end
    return argmax(α→α⋅b, Γa)
end
```

アルゴリズム 21.6　離散的な状態および行動空間をもつ部分観測マルコフ決定過程の信念をバックアップするメソッド．ここで，`Γ` はアルファベクトルのベクトル，`b` はバックアップを適用する信念ベクトルである．ベクトル信念の `Update` メソッドはアルゴリズム 19.2 で定義されている．

B の信念に対してバックアップ操作を繰り返し適用すると，収束するまで，アルファベクトルで表される価値関数の下限が次第に増加する．通常，B には初期信念から到達可能なすべての信念が含まれていないので，収束した価値関数は必ずしも最適ではない．しかしながら，B の信念が到達可能な信念空間全体にうまく分散されている限り，近似は許容できると考えられる．いずれの場合も，結果として得られる価値関数は，方策をさらに改善するために，恐らくオンラインの他のアルゴリズムとともに使用できる下限を提供することが保証されている．

点ベース価値反復はアルゴリズム 21.7 で実装されている．図 21.4 は問題例に対する複数の反復を示している．

```
struct PointBasedValueIteration
    B     # set of belief points
    k_max # maximum number of iterations
end

function update(𝒫::POMDP, M::PointBasedValueIteration, Γ)
    return [backup(𝒫, Γ, b) for b in M.B]
end

function solve(M::PointBasedValueIteration, 𝒫)
    Γ = fill(baws_lowerbound(𝒫), length(𝒫.𝒜))
    Γ = alphavector_iteration(𝒫, M, Γ)
    return LookaheadAlphaVectorPolicy(𝒫, Γ)
end
```

アルゴリズム 21.7　点ベース価値反復．これは，離散的な状態，行動および観測空間をもつ無限時間区間部分観測マルコフ決定過程の近似的な最適方策を見つける．ここで，`B` は信念のベクトル，`k_max` は反復回数である．

21.5　ランダム化点ベース価値反復

ランダム化点ベース価値反復 (randomized point-based value iteration) (アルゴリズム 21.8) は，前節の点ベース価値反復の変形版である[7]．主な違いは，B の各信念でのアルファベクトルを保持することを強制しないという事実である．

[7] M. T. J. Spaan and N. A. Vlassis, “Perseus: Randomized Point-Based Value Iteration for POMDPs,” *Journal of Artificial Intelligence Research*, vol. 24, pp. 195–220, 2005.

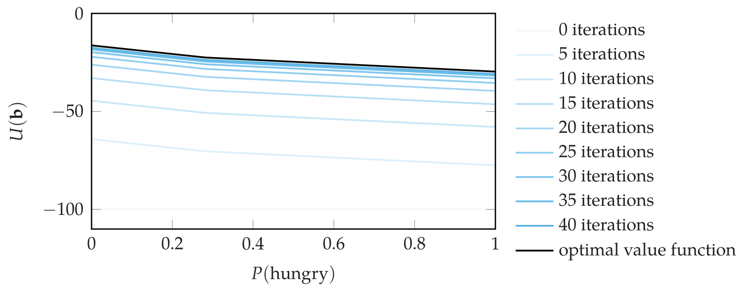


図 21.4　信念ベクトル [1/4, 3/4] および [3/4, 1/4] をもつ泣いている赤ちゃん問題に対して点ベース価値反復を用いて得られた近似価値関数．Q マルコフ決定過程や高速情報限度とは異なり，点ベース価値反復の価値関数は常に真の価値関数の下限になる．

Γ における単一のアルファベクトルでアルゴリズムを初期化し，各反復で Γ を更新し，場合によっては Γ のアルファベクトルの数を必要に応じて増減させる．更新ステップをこのように変更すると効率を向上させることができる．

```
struct RandomizedPointBasedValueIteration
    B     # set of belief points
    k_max # maximum number of iterations
end

function update(𝒫::POMDP, M::RandomizedPointBasedValueIteration, Γ)
    Γ′, B′ = [], copy(M.B)
    while !isempty(B′)
        b = rand(B′)
        α = argmax(α→α⋅b, Γ)
        α′ = backup(𝒫, Γ, b)
        if α′⋅b ≥ α⋅b
            push!(Γ′, α′)
        else
            push!(Γ′, α)
        end
        filter!(b→maximum(α⋅b for α in Γ′) <
            maximum(α⋅b for α in Γ), B′)
    end
    return Γ′
end

function solve(M::RandomizedPointBasedValueIteration, 𝒫)
    Γ = [baws_lowerbound(𝒫)]
    Γ = alphavector_iteration(𝒫, M, Γ)
    return LookaheadAlphaVectorPolicy(𝒫, Γ)
end
```

アルゴリズム 21.8　ランダム化点ベース価値反復．これは，離散的な状態，行動および観測空間をもつ無限時間区間部分観測マルコフ決定過程に対して近似的最適方策を見つける．ここで，`B` は信念ベクトル，`k_max` は反復回数である．

各更新では，アルファベクトルの集合 Γ を入力として受け取り，B における信念において，Γ によって表現される価値関数を改善するアルファベクトルの集合 Γ' を出力する．言い換えれば，すべての $\mathbf{b} \in B$ に対して，$\mathcal{U}^{\Gamma'}(\mathbf{b}) \geq \mathcal{U}^{\Gamma}(\mathbf{b})$ となる Γ' を出力する．Γ' を空集合に初期化し，B' を B に初期化することから始める．次に，B' からランダムにとってきた点 $\mathbf{b}$ を削除し，新しいアルファベクトル $\boldsymbol{\alpha}$ を得るために Γ を用いて，$\mathbf{b}$ に関する信念バックアップ (アルゴリズム 21.6) を実行する．さらに，$\mathbf{b}$ において $\Gamma \cup \{\boldsymbol{\alpha}\}$ 内で支配しているアルファベクトルを見つけて，Γ' に追加する．このアルファベクトルによって価値が改善された B' 内のすべての信念点が B' から削除される．アルゴリズムが進むにつ

```
        return V[b]
    end
    n = length(𝒫.𝒮)
    E = basis(𝒫)
    u = sum(V[E[i]] * b[i] for i in 1:n)
    for (b′, u′) in V
        if b′ ∉ E
            i = argmax([norm(b-e, 1) - norm(b′-e, 1) for e in E])
            w = [norm(b - e, 1) for e in E]
            w[i] = norm(b - b′, 1)
            w /= sum(w)
            w = [1 - wi for wi in w]
            α = [V[e] for e in E]
            α[i] = u′
            u = min(u, w⋅α)
        end
    end
    return u
end

(α::SawtoothPolicy)(b) = greedy(α, b).a
```

信念の集合 B に対して貪欲的 1 ステップ先読みを繰り返し適用して，上限の推定を厳しくすることができる．B の信念は V の信念の上位集合になるかもしれない．アルゴリズム 21.10 はこれの実装を提供する．例 21.1 では，泣いている赤ちゃん問題に対するのこぎり歯近似の複数回の反復の効果を示している．

```
struct SawtoothIteration
    V     # initial mapping from beliefs to utilities
    B     # beliefs to compute values including those in V map
    k_max # maximum number of iterations
end

function solve(M::SawtoothIteration, 𝒫::POMDP)
    E = basis(𝒫)
    π = SawtoothPolicy(𝒫, M.V)
    for k in 1:M.k_max
        V = Dict(b => (b ∈ E ? M.V[b] : greedy(π, b).u) for b in M.B)
        π = SawtoothPolicy(𝒫, V)
    end
    return π
end
```

アルゴリズム 21.10　のこぎり歯反復では，B の点で 1 ステップ先読みを繰り返し適用して，V の点での効用推定値を改善する．B の信念は V に含まれる信念の上位集合である．各反復で上限を保持するため，E に保存されている標準基底信念では更新は行われない．k_max 回の反復を実行する．

ステップサイズ 0.2 で一定間隔の信念点をもつ泣いている赤ちゃん問題の価値の上限を保持したいとする．最初の上限を得るために，高速情報限度を用いる．次に，3 ステップの反復に対するのこぎり歯反復を次のように実行できる．

```
n = length(𝒫.𝒮)
πfib = solve(FastInformedBound(1), 𝒫)
V = Dict(e => utility(πfib, e) for e in basis(𝒫))
B = [[p, 1 - p] for p in 0.0:0.2:1.0]
π = solve(SawtoothIteration(V, B, 2), 𝒫)
```

例 21.1　泣いている赤ちゃん問題に関する一定間隔の信念での上限を保持するためのものこぎり歯の能力の例証

22 オンライン信念状態計画

オンライン法は，現在の信念状態から計画を立てることで最適方策を決定する．現在の状態から到達可能な信念空間は，通常，完全な信念空間と比較して小さくなる．完全に観測可能な問題クラスの場合と同様に，多くのオンライン法は，ある時間区間までのツリーベースの探索法の一種を用いる[1]．木の深さに依存して指数関数的に計算量が増大しないように，さまざまな戦略が用いられる．オフライン法に比べて，オンライン法では実行中の決定ステップごとに多くの計算を必要とするが，高次元の問題に適用しやすいという特徴をもつ．

[1] 以下の文献で関連研究のサーベイが行われている．S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa, "Online Planning Algorithms for POMDPs," *Journal of Artificial Intelligence Research*, vol. 32, pp. 663–704, 2008.

22.1 ロールアウトに基づく先読み

アルゴリズム 9.1 では，完全に観測可能な問題におけるオンライン法としてロールアウトによる先読みが導入された．このアルゴリズムは，部分観測可能な問題に対してそのまま適用することができる．これは次の状態をランダムにサンプリングする関数を用いるが，部分観測の文脈では信念状態に対応する．この関数はアルゴリズム 21.11 でも導入されている．遷移，報酬，観測の明示的なモデルではなく，生成モデルを用いることで，高次元の状態と観測空間のある問題に対応することができる．

22.2 前方探索

アルゴリズム 9.2 の前方探索戦略を変更することなく部分観測可能な問題に適用することができる．マルコフ決定過程と部分観測マルコフ決定過程の違いは，図 22.1 に示されているように，行動と観測で分枝する 1 ステップ先の先読みにある．信念 b に基づいて行動 a を選択する価値は，深さ d まで再帰的に定義することができる．

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

ここで，$\mathcal{U}_d(b) = \max_a Q_d(b,a)$ である．$d = 0$ の場合，最大の深さに達しており，近似価値関数は，前章で議論された方法の 1 つであるか，発見的に選択されるか，または 1 回以上のロールアウトから推定される．近似価値関数 $\mathcal{U}(b)$ を用いて効用を計算する．$d > 0$ の場合，さらに深く探索を続け，別のレベルまで再帰的に進める．例 22.1 では，機械更新問題に対する Q マルコフ決定過程と前方探索を組み合わせる方法を示す．例 22.2 では，泣いている赤ちゃん問題に対する前方探索を示す．

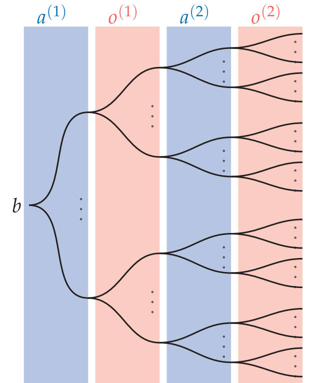


図 22.1　前方探索は，最大の期待報酬を得るための行動を選択するため，任意の有限の深さまで行動–観測–信念のグラフを探索する．この図は，深さ 2 までの探索の様子を示している．

できる．たとえば，背景知識があればルートまたはツリーの下の部分で選択可能な行動を制限することができる．観測の分枝については，一部の観測，または最も可能性の高い観測のみを考える[2]．分枝を完全に避けるために，開ループまたは 9.9.3 項で述べた後知恵最適化を現在の信念からサンプリングされた状態で適用する．

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

22.3　分枝限定法

マルコフ決定過程の文脈で紹介した**分枝限定法** (branch and bound) は，部分観測マルコフ決定過程にも拡張できる．9.4 節のアルゴリズムをそのまま用いることができる (例 22.3 参照)．これは適切な先読み実装を用いて信念を更新し，観測を考慮することに依存している．アルゴリズムの効率は，枝刈りのための上界と下界の質に依存している．

例 22.3　泣いている赤ちゃん問題への分枝限定法の適用

この例では，泣いている赤ちゃん問題に分枝限定法を適用し，深さ 5 で計算する．上界は高速情報限度から，下界は点ベース価値反復から得る．信念 [0.4,0.6] に基づいて，行動選択確率が以下のように計算される．

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

完全に観測可能な場合で行ったように上界と下界に対して，その問題背景固有のヒューリスティックを用いることができるが，代わりに前章で導入した離散状態空間に対する手法の 1 つを用いることもできる．たとえば，上界には高速情報限度を使用し，下界には点ベース価値反復を用いることができる．下界 $\underline{\mathcal{U}}$ と上界 $\overline{Q}$ が真の下界と上界である限り，分枝限定法のアルゴリズムの結果は，近似価値関数として $\underline{\mathcal{U}}$ を用いた前方探索アルゴリズムと同じ結果になる．

22.4　スパースサンプリング

前方探索はすべての観測を考慮して総和を計算するため，$|\mathcal{O}|$ に対する指数関数的な実行時間が必要である．9.5 節で導入したように，総和の計算を避けるためにサンプリングを用いることができる．各行動に対して m 個の観測を生成し，以下の計算を行う．

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

の指標として用いる．信念 b におけるギャップは上界と下界の差分 $\overline{U}(b)-\underline{U}(b)$ である．ギャップヒューリスティックを用いる探索アルゴリズムは，信念のバックアップから利益を得る可能性が高いため，ギャップを最大化する観測を選択することが一般的である．近似価値関数を用いた先読みに基づいて行動が選択される場合がある．このような行動選択は，アルゴリズム 22.4 により実装される[8]．

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

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

function (π::GapHeuristicSearch)(b)
    𝒫, k_max, d_max, δ = π.𝒫, π.k_max, π.d_max, π.δ
    Ulo = Dict{Vector{Float64}, Float64}()
    Uhi = Dict{Vector{Float64}, Float64}()
    for i in 1:k_max
        heuristic_search(π, Ulo, Uhi, b, d_max)
        if Uhi[b] - Ulo[b] < δ
            break
        end
    end
    return argmax(a→lookahead(𝒫,b′→Ulo[b′],b,a), 𝒫.𝒜)
end
```

アルゴリズム 22.4　上下界，ギャップ基準，価値関数の初期の上界と下界を用いるヒューリスティック探索の実装．特定の信念に関する価値関数の上界と下界を保持するため，ディクショナリ `Ulo` と `Uhi` を更新する．信念 `b` におけるギャップは `Uhi[b] - Ulo[b]` である．ギャップが閾値 `δ` より小さいか，最大深さ `d_max` に達するまで探索を続ける．探索には最大反復回数 `k_max` が割り当てられる．

[8] たとえば，以下の文献に示される部分観測マルコフ決定過程のためのさまざまなヒューリスティック探索のアルゴリズムが存在し，ギャップを最小限に抑えることを目的としている．S. Ross and B. Chaib-draa, "AEMS: An Anytime Online Search Algorithm for Approximate Policy Refinement in Large POMDPs," in *International Joint Conference on Artificial Intelligence* (*IJCAI*), 2007．ここでの実装は，前節でも参照されている DESPOT で用いられているものと同様である．

アルゴリズムのパフォーマンスは，ヒューリスティック探索で用いられる初期の上界と下界に強く影響を受ける．例 22.6 は，下界 $\underline{U}(b)$ に対してランダムロールアウト方策を用いている．ここでは，ロールアウトは固定された深さまでの単一の試行に基づいて行われるため，下界を生成する保証はないが，サン

22.8 要　約

- 簡単なオンライン戦略として，現在の信念から行動を生成し，近似価値関数に基づいてその期待値を推定する 1 ステップ先の先読みを実行することがある．
- 前方探索は任意の時間区間までの先読みの一般化であり，より良い方策を得ることができるが，その計算量は時間区間とともに指数関数的に増加する．
- 分枝限定法は前方探索のより効率的な方法であり，特定の経路に探索が集中しないように価値関数の上界と下界を用いる．
- スパースサンプリングは，すべての観測空間に対する繰返しの計算負荷を軽減する近似法である．
- モンテカルロツリー探索は，状態ではなく履歴に対して操作することで，部分観測マルコフ決定過程に適応することができる．
- 決定化スパースツリー探索は，決定論的に観測することを保証する特別な形式の粒子信念を用いて，探索ツリーを大幅に縮小する．
- ヒューリスティック探索では，保持される価値関数の上界と下界のギャップが大きい領域を探索するために行動–観測のペアを効率的に選択する．

22.9 演　習

22.1　$\mathcal{A} = \{a^1, a^2\}$ と信念 $\boldsymbol{b} = [0.5, 0.5]$ を考える．報酬は常に 1 とする．観測関数は $P(o^1 \mid a^1) = 0.8$ および $P(o^1 \mid a^2) = 0.4$ で与えられる．近似価値関数はアルファベクトル $\boldsymbol{\alpha} = [-3, 4]$ で与えられる．$\gamma = 0.9$ で，深さ 1 までの前方探索を使用して $\mathcal{U}(\boldsymbol{b})$ を計算せよ．下表で示される，更新後の信念を用いること．

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

【解】　式 (22.1) に従い，次のように深さ 1 での行動価値関数を計算する必要がある．

$$Q_d(\boldsymbol{b}, a) = R(\boldsymbol{b}, a) + \gamma \sum_o P(o \mid \boldsymbol{b}, a) \mathcal{U}_{d-1}(\text{Update}(\boldsymbol{b}, a, o))$$

まず，更新された信念に対する効用を計算する．

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

次に，両方の行動に対する行動価値関数を計算する．

$$Q_1(\boldsymbol{b}, a^1) = 1 + 0.9((P(o^1 \mid \boldsymbol{b}, a^1)\mathcal{U}_0(\text{Update}(\boldsymbol{b}, a^1, o^1)) + (P(o^2 \mid \boldsymbol{b}, a^1)\mathcal{U}_0(\text{Update}(\boldsymbol{b}, a^1, o^2)))$$

条件付き計画は，最初に行動 a^1 を実行し，o^1 を観測した場合は直前の行動を切り替え，o^2 を観測した場合は直前の行動を再び選択する．コントローラも同様の行動選択に従うが，コントローラノードは 5 つ少ない．さらに，コントローラは 2 つのノードで無限時間区間方策を完全に表現する (対して条件付き計画は 7 つのノードがある)．条件付き計画には，無限の深さのツリーが必要になるため，この無限の時間区間方策を表現することができない．

コントローラは条件付き計画に対していくつかの利点がある．まず，コントローラはより簡潔に表現できる．条件付き計画のノード数は深さに伴って指数関数的に増加するが，有限状態コントローラでは，必ずしも指数関数的に増加するノード数は必要であるわけではない．前章で紹介した近似法についても，大量の信念とそれに整合するアルファベクトルを保持する必要があるため，効率的であるとは限らない．コントローラは，膨大な数の到達可能な信念をごく少数のノードで考慮できるため，はるかに簡潔になる可能性がある．コントローラのもう 1 つの利点として，信念を保持する必要がないことが挙げられる．コントローラの各ノードは信念空間の部分集合に対応し，相互に排他的ではある必要はない．コントローラは，到達可能な信念空間をカバーするこれらの部分集合間を推移する．コントローラ自体が観測に基づいて新しいノードを選択し，問題領域によっては高コストになることもある信念の更新には依存しない．

コントローラ方策に従う効用は，状態空間が $X \times \mathcal{S}$ の直積のマルコフ決定過程を形成すれば計算できる．ノード x がアクティブで状態 s にいる場合の価値は，次式で計算できる．

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

式 (23.1) で与えられる連立一次方程式を解くことが，方策評価となる．もしくは，アルゴリズム 23.2 に示されるように，反復方策評価を適用することもできる．

```
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
```

アルゴリズム 23.2　有限状態コントローラ `π` の効用を計算するための反復方策評価のアルゴリズムで，反復回数 `k_max` だけ実行される．効用関数は，式 (23.1) に従って，現在のコントローラノード `x` と状態 `s` に対する単一ステップ評価を実行する．このアルゴリズムは，マルコフ決定過程に対して反復方策評価を適用したアルゴリズム 7.3 を改変したものである．

信念が既知である場合，現在の価値は次式で与えられる．

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

$U(x,s)$ を X に含まれる各ノード x に対して，1 つのアルファベクトルを定義するものと考えることができる．各アルファベクトル $\boldsymbol{\alpha}_x$ は $\alpha_x(s)=\mathcal{U}(x,s)$ により定義される．与えられたアルファベクトルに対する現在の価値は，$\mathcal{U}(x,b)=\boldsymbol{b}^\top\boldsymbol{\alpha}_x$ となる．

コントローラと初期信念が与えられた場合，次式のように価値を最大化する初期ノードを選択する．

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

23.2 方策反復

20.5 節では，条件付き計画にノードを徐々に追加して最適有限時間区間方策 (アルゴリズム 20.8) に到達するメソッドを示した．本節では，コントローラにノードを徐々に追加して無限時間区間問題を最適化する方法を示す．方策の表現は異なるが，本節で紹介する部分観測可能な問題のための方策反復は，完全観測可能な問題のための方策反復アルゴリズム (7.4 節) といくつかの類似点がある[2]．

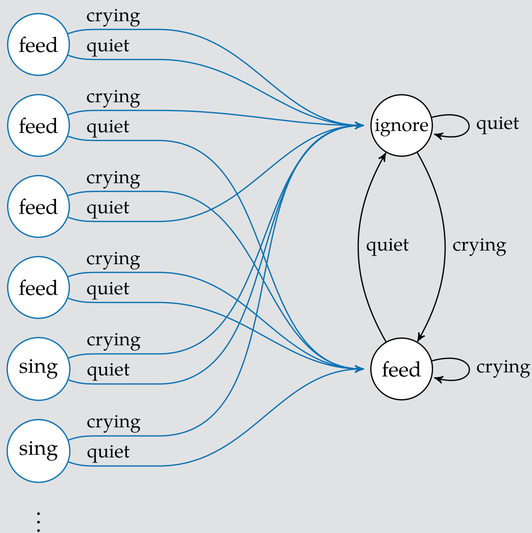
方策反復 (アルゴリズム 23.3) では，初期コントローラを任意に決定し，方策評価と方策改善を繰り返す．方策評価では，式 (23.1) を解くことで効用 $\mathcal{U}(x,s)$ を評価する．方策改善では，コントローラに新しいノードを追加する．具体的には，決定論的な行動割当て $\psi(a_i \mid x')=1$ と決定論的な後続選択分布 $\eta(x \mid x',a,o)$ の組合せごとに新しいノード x' を追加する．この過程は，反復 k のノードの集合 $X(k)$ に $|\mathcal{A}||X^{(k)}|^{|\mathcal{O}|}$ の新しいコントローラノードを追加する[3]．方策改善の手順は，例 23.3 に示される．

[2] ここでの方策反復の方法については，以下の文献を参照．E. A. Hansen, “Solving POMDPs by Searching in Policy Space,” in *Conference on Uncertainty in Artificial Intelligence* (*UAI*), 1998.

[3] すべての組合せを追加できない場合がある．**有界方策反復** (bounded policy iteration) と呼ばれる別のアルゴリズムでは，ノードを 1 つだけ追加する．P. Poupart and C. Boutilier, “Bounded Finite State Controllers,” in *Advances in Neural Information Processing Systems* (*NIPS*), 2003．アルゴリズムは，複数のノードを追加することもできる．たとえば，**モンテカルロ価値反復** (Monte Carlo value iteration) は，k 回目の反復で $O(n|\mathcal{A}||X^{(k)}|)$ の新しいノードを追加する．ここで，n はパラメータである．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)
```

アルゴリズム 23.3 反復回数を k_max，方策評価反復回数を eval_max とする部分観測マルコフ決定過程 $\mathcal{P}$ に対する方策反復．アルゴリズムは反復的方策評価 (アルゴリズム 23.2) と方策改善を適用する．枝刈りはアルゴリズム 23.4 で実装される．




```
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
```

アルゴリズム 23.4　方策反復における枝刈り．枝刈りは現在の方策 π のノード数を削減する．その際，方策評価により計算された効用 U と直前のノードのリスト prevX を入力とする．最初の手順では，直前のノードのリストにおいて，他の点に支配されているノードを改善したノードに置き換え，冗長なノードを支配されているものとしてマークする．2 番目の手順では，前のノードと同一の新しいノードをマークする．3 番目の手順では，任意の点に支配されている新しいノードをマークする．最後に，すべてのマークされたノードが削除される．

```
    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
```

方策改善はコントローラ方策の期待価値を悪化させることはない．$X^{(k)}$ のノードの価値は更新されず，それらと到達可能な後続ノードも変更されない．$X^{(k)}$ が最適なコントローラでない場合，少なくとも 1 つの新しいノードが方策改善で追加され，いくつかの状態で期待価値が改善されるため，全体のコントローラが改善される．

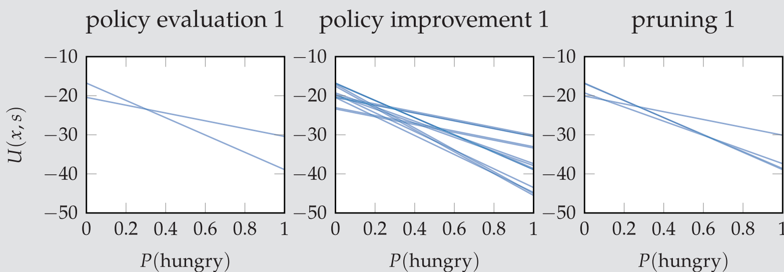
方策改善中に追加される多くのノードにおいて，方策が改善されるわけではない．方策評価の後に枝刈りを実行することにより，不要なノードが削除される．これにより，コントローラの最適価値関数を悪化させることはない．枝刈りは改善手順で発生するノードの指数関数的な増加を抑制することができる．ある場合には，枝刈りによってループが生成され，簡潔なコントローラが得られることもある．

枝刈りでは，既存のノードと同一の新しいノードが削除される．また，他のノードによって**支配されている** (dominated) 新しいノードも削除される．次式が成立するとき，ノード x は別のノード x' に支配されるという．

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

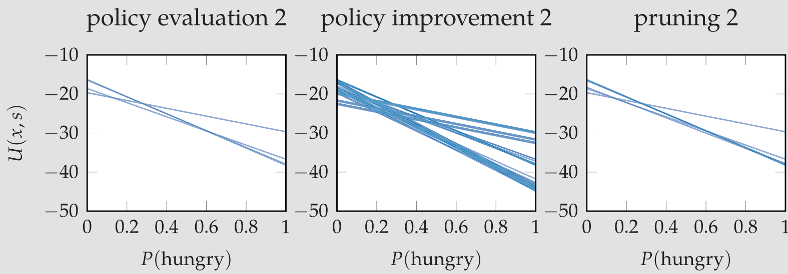
既存のノードも削除することができる．新しいノードが既存のノードを支配する場合，既存のノードをコントローラから削除する．削除されたノードへの遷移は，その代わりに支配しているノードに遷移される．この手続きは，新しいノードを削除し，支配されたノードの行動と後続リンクを新しいノードのものに更新することと等価である．例 23.4 は，泣いている赤ちゃん問題に対する評価，拡張，および枝刈りの例である．

ここでは，例 23.3 と同じ初期コントローラを用いて，方策反復の最初の反復を示す．これは，方策評価 (左) と方策改善 (中央) の 2 つの主要な手続きとオプションとなる枝刈り (右) で構成されている．



例 23.4　コントローラ方策表現での泣いている赤ちゃん問題における方策反復での，評価，改善，枝刈りステップ

方策反復の 2 回目の反復は，同じパターンに従う．



2 回目の反復の後の効用は大幅に改善され，ほぼ最適な価値になっている．枝刈りは，以前の反復に対して支配されたノードや重複ノード，および現在の反復の新しいノードを削除していることがわかる．

23.3 非線形計画

方策改善の問題は，すべてのノードにわたって ψ と η を同時に最適化する，単一の大規模な**非線形計画** (nonlinear programming) 問題として定式化できる (アルゴリズム 23.5) [4]．この定式化により，汎用的なソルバを適用できる．非線形計画法は，ベルマン期待方程式 (23.1) を満たしながら，与えられた初期信念の効用を最大化するため，コントローラの空間を直接探索する．方策評価と方策改善の手順が交互に行われることはなく，コントローラノードの数は一定である．

[4] 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.

```
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)
```

アルゴリズム 23.5 非線形計画法を用いて，初期信念を b として部分観測マルコフ決定過程 $\mathcal{P}$ の最適な固定サイズコントローラの方策を計算する．有限状態コントローラのサイズは，ノードの数 ℓ に基づいて決定される．

```
    @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
```

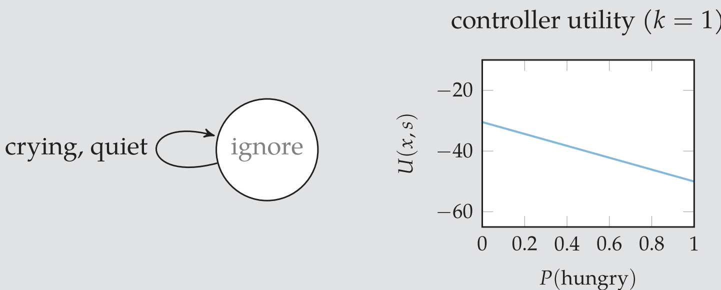
与えられた初期信念に対応する初期ノード x^1 を用いると，最適化問題は次式のように定式化される．

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

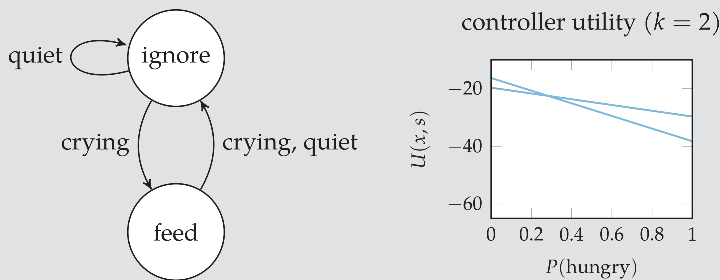
この問題は，**二次制約付き線形計画問題** (quadratically constrained linear program: QCLP) として記述することができ，専用のソルバを用いることで効果的に解くことができる[5]．例 23.5 にはこの方法が示される．

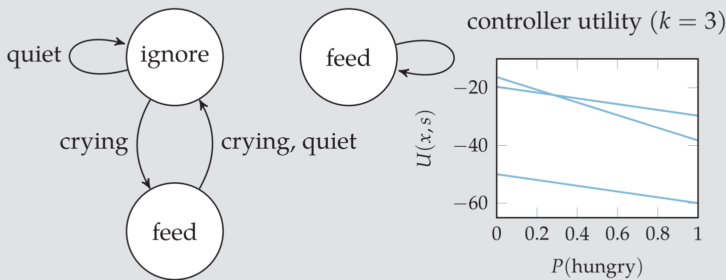
[5] 一般的な QCLP を解くことは NP 困難であるが，専用のソルバによって効率的に近似値を得られる．

ここには，非線形計画を用いて計算された $b_0 = [0.5, 0.5]$ の泣いている赤ちゃん問題の最適な固定サイズのコントローラが示されている．上部ノードを x_1 としている．



例 23.5　k の固定サイズを 1,2,3 に設定したコントローラの非線形計画のアルゴリズム．各行は，左右にそれぞれ方策と対応する効用 (アルファベクトル) が示される．確率的コントローラは円で示され，中央に最も高い確率が割り当てられた行動が示される．出力エッジは，観測に基づく後続ノードの選択が示される．ノードの行動と後続の確率の大きさは透明度により示される (不透明なものは高い確率が割り当てられ，確率が低くなればより透明になる)．





$k = 1$ のとき，最適方策は単純に永遠に無視することとなる．$k = 2$ のとき，最適方策は泣き声が観測されるまで無視し，その時点で最良行動は赤ちゃんに授乳することであり，その後無視することである．この方策は，無限時間区間をもつ泣いている赤ちゃんに対する部分観測マルコフ決定過程に非常に似ている．$k = 3$ のとき，最適方策は $k = 2$ の場合と本質的に同じである．

23.4　勾配上昇

固定サイズのコントローラ方策は勾配上昇 (付録 A.11 を参照) を用いて反復的に改善される[6]．勾配を計算することは難しい場合もあるが，これにより勾配を用いる多様な最適化技術をコントローラ最適化に応用できる．アルゴリズム 23.6 は，アルゴリズム 23.7 を用いてコントローラの勾配上昇を実装している．

[6] 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.

```
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)
```

アルゴリズム 23.6　初期信念を b とした，部分観測マルコフ決定過程 $\mathcal{P}$ に対するコントローラ勾配上昇のアルゴリズムの実装．コントローラのノードのサイズを ℓ に固定している．これは，初期信念の価値を最大化するため，コントローラの勾配に従いステップサイズ α で k_max 回の反復において改善される．

```
    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
```

```
function gradient(π::ControllerPolicy, M::ControllerGradient, 𝒫::POM
 DP)
    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]
```

アルゴリズム 23.7　コントローラ勾配上昇法の `gradient` 手順．これは，方策勾配`∂U′∂ψ`と`∂U′∂η`に対する効用 `U` の勾配を構築する．

2 エージェント単純ゲームは表で表現することができる．行はエージェント 1 の行動を表す．列はエージェント 2 の行動を表す．エージェント 1 と 2 の報酬は各セルに示される．

agent 1 \ agent 2	cooperate	defect
cooperate	$-1, -1$	$-4, 0$
defect	$0, -4$	$-3, -3$

結合方策 (joint policy) $\boldsymbol{\pi}$ はエージェントが実行する結合行動上の確率分布を示している．結合方策は個々のエージェントの方策に分解できる．エージェント i が行動 a を選択する確率が $\pi^i(a)$ で与えられる．ゲーム論では，決定論的方策は**純戦略** (pure strategy) と呼ばれ，確率的方策は**混合戦略** (mixed strategy) と呼ばれる．エージェント i の観点からの結合方策 $\boldsymbol{\pi}$ の効用は

$$\mathcal{U}^i(\boldsymbol{\pi}) = \sum_{\boldsymbol{a}\in\mathcal{A}} R^i(\boldsymbol{a}) \prod_{j\in\mathcal{I}} \pi^j(a^j) \tag{24.1}$$

である．アルゴリズム 24.2 は方策を表現し，彼らの効用を計算するルーティーンを実装している．

```
struct SimpleGamePolicy
    p # dictionary mapping actions to probabilities

    function SimpleGamePolicy(p::Base.Generator)
        return SimpleGamePolicy(Dict(p))
    end

    function SimpleGamePolicy(p::Dict)
        vs = collect(values(p))
        vs ./= sum(vs)
        return new(Dict(k ⇒ v for (k,v) in zip(keys(p), vs)))
    end

    SimpleGamePolicy(ai) = new(Dict(ai ⇒ 1.0))
end

(πi::SimpleGamePolicy)(ai) = get(πi.p, ai, 0.0)

function (πi::SimpleGamePolicy)()
    D = SetCategorical(collect(keys(πi.p)), collect(values(πi.p)))
    return rand(D)
end

joint(X) = vec(collect(product(X...)))

joint(π, πi, i) = [i == j ? πi : πj for (j, πj) in enumerate(π)]

function utility(𝒫::SimpleGame, π, i)
    𝒜, R = 𝒫.𝒜, 𝒫.R
    p(a) = prod(πj(aj) for (πj, aj) in zip(π, a))
    return sum(R(a)[i]*p(a) for a in joint(𝒜))
end
```

アルゴリズム 24.2　エージェントに関連付けられた方策は行動を確率に写像するディクショナリによって表される．方策を構成するにはさまざまな方法がある．1 つの方法は，ディクショナリのディレクトリを渡すことであり，この場合確率は正規化される．別の方法は，このディクショナリを作成するジェネレータを渡すことである．行動を渡すことで方策を構成することもでき，この場合その行動に確率 1 が割り当てられる．個別の方策 `πi` があるとき，方策が行動 `ai` に関連付ける確率を計算するために，`πi(ai)` を呼び出す．`πi()` を呼び出したら，その方策に従ってランダムな行動を返す．`𝒜` からの結合行動空間を構成するために `joint(𝒜)` を用いることができる．エージェント `i` の観点から，ゲーム `𝒫` で結合方策 `π` を実行することに関連する効用を計算するために，`utility(𝒫, π, i)` を使うことができる．

ゼロ和ゲーム (zero-sum game) はすべてのエージェントの報酬の和が 0 になる単純ゲームの 1 つの型である．ここでは，エージェントの利益は他方のエージェントの損失となる．2 エージェント $\mathcal{I} = \{1,2\}$ をもつゼロ和ゲームは正反対

$$
\begin{aligned}
&+1\frac{1}{3}\frac{1}{3}+0\frac{1}{3}\frac{1}{3}-1\frac{1}{3}\frac{1}{3}\\
&-1\frac{1}{3}\frac{1}{3}+1\frac{1}{3}\frac{1}{3}+0\frac{1}{3}\frac{1}{3}\\
=0&
\end{aligned}
$$

1 人のエージェントによるいかなる逸脱も期待利得を減少させるので，われわれはナッシュ均衡を見つけたことになる．

```
struct NashEquilibrium end

function tensorform(𝒫::SimpleGame)
    ℐ, 𝒜, R = 𝒫.ℐ, 𝒫.𝒜, 𝒫.R
    ℐ′ = eachindex(ℐ)
    𝒜′ = [eachindex(𝒜[i]) for i in ℐ]
    R′ = [R(a) for a in joint(𝒜)]
    return ℐ′, 𝒜′, R′
end

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

アルゴリズム 24.5　この非線形計画問題で，単純ゲーム $\mathcal{P}$ に対するナッシュ均衡を計算できる．

24.5 関連均衡

関連均衡 (correlated equilibrium) はエージェントが独立して行動するという仮定を緩和することによって，ナッシュ均衡の概念を一般化している．この場合の結合行動は完全な同時分布によってもたらされる．**関連結合方策** (correlated joint policy) $\pi(\boldsymbol{a})$ はすべてのエージェントの結合行動上の単一の分布である．結果として，さまざまなエージェントの行動は関連しており，方策が個々の方策 $\pi^i(a^i)$ へ分断されることを防いでいる．アルゴリズム 24.6 はそのような方策をいかに表現するかを示している．

を満たすならば，結合方策は式 (24.6) を満たす．しかし，すべての関連均衡がナッシュ均衡であるとはいえない．

関連均衡は次の線形計画問題を用いて計算できる (アルゴリズム 24.7)．

$$
\begin{aligned}
\underset{\pi}{\text{maximize}} \quad & \sum_i \sum_{\boldsymbol{a}} R^i(\boldsymbol{a})\pi(\boldsymbol{a}) \\
\text{subject to} \quad & \sum_{\boldsymbol{a}^{-i}} R^i(a^i, \boldsymbol{a}^{-i})\pi(a^i, \boldsymbol{a}^{-i}) \geq R^i(a^{i'}, \boldsymbol{a}^{-i})\pi(a^i, \boldsymbol{a}^{-i}) \text{ for all } i, a^i, a^{i'} \\
& \sum_{\boldsymbol{a}} \pi^i(\boldsymbol{a}) = 1 \\
& \pi^i(\boldsymbol{a}) \geq 0 \text{ for all } \boldsymbol{a}
\end{aligned}
\tag{24.8}
$$

線形計画問題は多項式時間で解くことができるが，結合行動空間の大きさはエージェントの数に対して指数関数的に大きくなる．制約式が関連均衡になるように強いている．しかし，目的関数はさまざまな有効な関連均衡の中から選択するために用いられる．表 24.1 は目的関数のいくつかの一般的な選択肢を提供している．

```
struct CorrelatedEquilibrium end

function solve(M::CorrelatedEquilibrium, 𝒫::SimpleGame)
    ℐ, 𝒜, R = 𝒫.ℐ, 𝒫.𝒜, 𝒫.R
    model = Model(Ipopt.Optimizer)
    @variable(model, π[joint(𝒜)] ≥ 0)
    @objective(model, Max, sum(sum(π[a]*R(a) for a in joint(𝒜))))
    @constraint(model, [i=ℐ, ai=𝒜[i], ai′=𝒜[i]],
        sum(R(a)[i]*π[a] for a in joint(𝒜) if a[i]==ai)
        ≥ sum(R(joint(a,ai′,i))[i]*π[a] for a in joint(𝒜) if a[i]==
         ai))
    @constraint(model, sum(π) == 1)
    optimize!(model)
    return JointCorrelatedPolicy(a ⇒ value(π[a]) for a in joint(𝒜))
end
```

アルゴリズム 24.7　関連均衡は単純ゲーム $\mathcal{P}$ に関して，ナッシュ均衡よりも一般的な最適性の概念であり，線形計画法を用いて計算できる．結果として得られる方策は関連しており，これはエージェントが結合行動を確率的に選択することを意味する．

名前	説明	目的関数
実用的	正味の効用の最大化	$\text{maximize}_\pi \sum_i \sum_{\boldsymbol{a}} R^i(\boldsymbol{a})\pi(\boldsymbol{a})$
平等主義	すべてのエージェントの効用の中での最小値の最大化	$\text{maximize}_\pi\, \text{minimize}_i \sum_{\boldsymbol{a}} R^i(\boldsymbol{a})\pi(\boldsymbol{a})$
金権的	すべてのエージェントの効用の中での最大値の最大化	$\text{maximize}_\pi\, \text{maximize}_i \sum_{\boldsymbol{a}} R^i(\boldsymbol{a})\pi(\boldsymbol{a})$
独裁的	エージェント i の効用の最大化	$\text{maximize}_\pi \sum_{\boldsymbol{a}} R^i(\boldsymbol{a})\pi(\boldsymbol{a})$

表 24.1　式 (24.8) の代替的な目的関数．これらを用いることで，さまざまな関連均衡が選択される．これらの記述は次の論文から採用されている．A. Greenwald and K. Hall, “Correlated Q-Learning,” in *International Conference on Machine Learning* (ICML), 2003.

24.6　繰返し最良応答

ナッシュ均衡を計算するには計算コストが高いので，代替的アプローチとして，一連の繰返しゲームで繰返し最良応答を適用することが考えられる．**繰返し最良応答** (iterated best response) (アルゴリズム 24.8) では，エージェントをランダムに循環させ，各エージェントの最適応答方策を順番に解く．この過程はナッシュ均衡に収束するかもしれないが，それは特定のクラスのゲームで保証されるだけである[9]．多くの問題で，循環が観測されることがよくある．

[9] 繰返し最良応答は，次の教科書の定理 19.12 で論じられるように，たとえば，**ポテンシャルゲーム** (potential game) として知られるクラスで収束する．N. Nisan, T. Roughgarden, É. Tardos, and V. V. Vazirani, eds., *Algorithmic Game Theory*. Cambridge University Press, 2007.

```
struct IteratedBestResponse
    k_max # number of iterations
    π     # initial policy
end

function IteratedBestResponse(𝒫::SimpleGame, k_max)
    π = [SimpleGamePolicy(ai => 1.0 for ai in 𝒜i) for 𝒜i in 𝒫.𝒜]
    return IteratedBestResponse(k_max, π)
end

function solve(M::IteratedBestResponse, 𝒫)
    π = M.π
    for k in 1:M.k_max
        π = [best_response(𝒫, π, i) for i in 𝒫.ℐ]
    end
    return π
end
```

アルゴリズム 24.8　繰返し最良応答はエージェントを循環させ，他のエージェントに対する最良応答を適用させる．アルゴリズムはある初期の方策から始め，`k_max` の繰返しの後で停止する．便利なものとして，入力として単純ゲームをとり，各エージェントに一様にランダムに行動を選択させる初期の方策を生成するコンストラクタがある．より複雑な形式のゲームの文脈を取り扱う次章において，同じ solve 関数が再使用される．

24.7　階層的ソフトマックス

行動ゲーム論 (behavioral game theory) として知られる分野は人間のエージェントをモデル化することを目的としている．人間と相互作用せざるを得ない意思決定システムを構築するときには，ナッシュ均衡を計算することが必ずしも役立つとは限らない．人々はしばしばナッシュ均衡戦略をプレイしない．最初に，ゲームに多くの異なる均衡があるとき，どの均衡を選ぶかが明確でないかもしれない．ただ 1 つの均衡があるゲームに関しても，認知の限界のためにナッシュ均衡を計算することが人には困難であるかもしれない．人間のエージェントがナッシュ均衡を計算できても，彼の相手がその計算のように行動するかどうかは疑問である．文献には多くの行動モデルがあるが[10]，1 つのアプローチとして，前節の繰返しアプローチとソフトマックスモデルを組み合わせることがある．この**階層的ソフトマックス** (hierarchical softmax) アプローチ (アルゴリズム 24.9) [11] は，$k > 0$ のレベルによってエージェントの**合理性の深さ** (depth of rationality) をモデル化している．レベル 0 のエージェントは行動を一様にランダムにプレイする．レベル 1 のエージェントは，他のプレイヤーがレベル 0 の戦略を採用すると仮定し，精度 λ のソフトマックス応答に従って行動を選択する．レベル k のエージェントは，他のプレイヤーがレベル $k-1$ をプレイしているとするソフトマックスモデルに従って行動を選択する．図 24.1 に単純ゲームに対するこのアプローチを例証している．

10) C. F. Camerer, *Behavioral Game Theory: Experiments in Strategic Interaction*. Princeton University Press, 2003.

11) このアプローチはしばしば**量子レベル *k*** (quantal-level-*k*) または**ロジットレベル *k*** (logit-level-*k*) と呼ばれる．D. O. Stahl and P. W. Wilson, “Experimental Evidence on Players’ Models of Other Players,” *Journal of Economic Behavior & Organization*, vol. 25, no. 3, pp. 309–327, 1994.

```
struct HierarchicalSoftmax
    λ # precision parameter
    k # level
    π # initial policy
end

function HierarchicalSoftmax(𝒫::SimpleGame, λ, k)
    π = [SimpleGamePolicy(ai => 1.0 for ai in 𝒜i) for 𝒜i in 𝒫.𝒜]
```

アルゴリズム 24.9　精度パラメータ λ とレベル k の階層的ソフトマックスモデル．初期設定では，すべての個々の行動に均一な確率を割り当てる初期の結合方策から開始する．

```
function FictitiousPlay(𝒫::SimpleGame, i)
    N = [Dict(aj => 1 for aj in 𝒫.𝒜[j]) for j in 𝒫.ℐ]
    πi = SimpleGamePolicy(ai => 1.0 for ai in 𝒫.𝒜[i])
    return FictitiousPlay(𝒫, i, N, πi)
end

(πi::FictitiousPlay)() = πi.πi()

(πi::FictitiousPlay)(ai) = πi.πi(ai)

function update!(πi::FictitiousPlay, a)
    N, 𝒫, ℐ, i = πi.N, πi.𝒫, πi.𝒫.ℐ, πi.i
    for (j, aj) in enumerate(a)
        N[j][aj] += 1
    end
    p(j) = SimpleGamePolicy(aj => u/sum(values(N[j])) for (aj, u) in
      N[j])
    π = [p(j) for j in ℐ]
    πi.πi = best_response(𝒫, π, i)
end
```

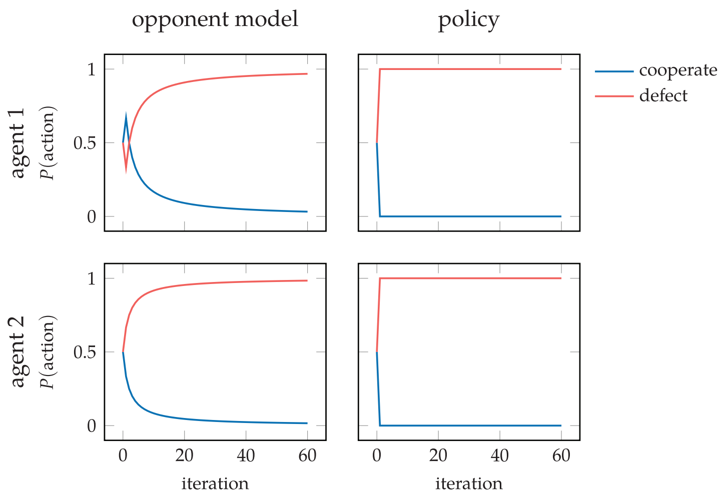


図 24.2　囚人のジレンマゲームにおいて，互いに学習し，適応する 2 つの架空プレイエージェント．最初の行には，反復を通じて，エージェント 2 に対して学習されたエージェント 1 のモデル (左) とエージェント 1 の方策 (右) を描いている．2 番目の行はエージェント 2 に対する同じものである．学習行動の変化を示すために，各エージェントの相手エージェントに対する初期の行動選択数は 1 と 10 の間の乱数が設定された．

架空プレイには，多くの変形版がある．**円滑架空プレイ** (smooth fictitious play)[16] と呼ばれる 1 つの変形版では，期待効用と，方策のエントロピーのような円滑関数を用いて，最良応答が選択される．別の変形版は**合理的学習** (rational learning) または**ベイズ学習** (Bayesian learning) と呼ばれる．合理的学習は架空プレイのモデルをベイズ事前分布として定式化された他のエージェントの行動に対する信念に拡張する．ここでは結合行動の履歴が与えられているとして，ベイズ規則が信念を更新するために使用される．従来の架空プレイは事前ディリクレ (4.2.2 項) を使用した合理的学習とみなすことができる．

[16] D. Fudenberg and D. Levine, "Consistency and Cautious Fictitious Play," *Journal of Economic Dynamics and Control*, vol. 19, no. 5–7, pp. 1065–1089, 1995.

状態遷移モデルを考慮する必要がある．

25.2.1 最良応答

エージェント i の**応答方策** (response policy) は，他のエージェントの固定された方策 $\boldsymbol{\pi}^{-i}$ が与えられたとき，期待効用を最大化する方策 π^i である．他のエージェントの方策が固定されたならば，問題はマルコフ決定過程に還元される．このマルコフ決定過程は状態空間 $\mathcal{S}$ と行動空間 $\mathcal{A}^i$ をもつ．遷移関数と報酬関数を次のように定義できる．

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

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

これはエージェント i にとって最良応答なので，マルコフ決定過程は報酬 R^i だけを用いる．このマルコフ決定過程を解けば，エージェント i にとっての最良応答方策となる．アルゴリズム 25.3 はこれの実装を与える．

```
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
```

アルゴリズム 25.3 マルコフゲーム 𝒫 に対して，他のエージェントが π の方策をプレイしているとして，エージェント i に対する決定論的な最良応答方策を計算できる．7 章のいずれかの方法を使用して，マルコフ決定過程を正確に解くことができる．

25.2.2 ソフトマックス応答

前章で行ったことと同様に，各状態で他のエージェントの方策に確率的応答を割り当てる**ソフトマックス応答方策** (softmax response policy) を定義できる．決定論的最良応答方策の構築で行ったように，固定した方策 $\boldsymbol{\pi}^{-i}$ をもつエージェントが環境に組み込まれたマルコフ決定過程を解く．次に，1 ステップ先読みを使用して行動価値関数 $Q(s,a)$ を取り出す．ソフトマックス応答は，精度パラメータ $\lambda \geq 0$ をもち，

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

となる．アルゴリズム 25.4 はこれの実装を提供している．このアプローチは階層的ソフトマックス解を生成するために使用される (24.7 節)．実際，アルゴリズム 24.9 を直接使用できる．

```
function softmax_response(𝒫::MG, π, i, λ)
    𝒫, 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])
```

アルゴリズム 25.4 結合方策 π に対するエージェント i の精度パラメータ λ をもつソフトマックス応答

```
    return MGPolicy(s ⇒ p(s) for s in 𝒮)
end
```

25.3 ナッシュ均衡

ナッシュ均衡概念はマルコフゲームへ一般化できる[2]．単純ゲームの場合と同様に，すべてのエージェントは互いに最良応答を実行し，そこから逸脱する動機がない．割引された無限時間区間をもつすべての有限マルコフゲームはナッシュ均衡をもつ[3]．

単純ゲームの文脈で解いた問題と同様な非線形最適化問題を解くことによって，ナッシュ均衡を見つけることができる．この問題では次のように，先読み効用の逸脱分の和を最小化し，方策が有効な確率分布になるように制約する．

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

ここで，

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

である．この非線形最適化問題は，アルゴリズム 25.5 で実装され，解かれる[4]．

[2] 方策は時間の経過とともに変化しないという点で**定常** (stationary) であると仮定しているので，ここで取り上げるナッシュ均衡は**定常マルコフ完全均衡** (stationary Markov perfect equilibria) である．

[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.

[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.

```
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))
```

アルゴリズム 25.5 この非線形問題が，マルコフゲーム $\mathcal{P}$ に対するナッシュ均衡を計算する．

```
            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
```

25.4 架空プレイ

単純ゲームの文脈で行ったように，シミュレーションでエージェントを動かすことで，学習ベースのアプローチを採用して結合方策に到達できる．アルゴリズム 25.6 は，状態遷移を取り扱うために前章で導入したシミュレーションループを一般化する．シミュレーションで実行されるさまざまな方策は，状態遷移やさまざまなエージェントによってとられた行動に基づいて，それら自体を更新する．

```
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
```

アルゴリズム 25.6　マルコフゲームにおいて，ランダムなステップをとり，完全なシミュレーションを実行するための関数．simulate 関数は，b からランダムにサンプリングされた状態から開始して，k_max ステップの結合方策 π をシミュレーションする．

方策を更新する 1 つのアプローチは，他のエージェントの方策上の最尤モデルを保持している，前章[5] で示した**架空プレイ** (fictitious play) (アルゴリズム 25.7) の一般化を用いることである．最尤モデルでは，各エージェントが実行する行動に加えて状態を追跡する．状態 s でエージェント j が行動 a^j をとる回数を確認し，通常 1 に初期化される表 $N(j,a^j,s)$ に格納する．各エージェント j が状態独立な確率的方策に従うことを仮定して，次のように最良応答を計算する．

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

[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.

```julia
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
```

アルゴリズム 25.7　各状態に対して他のエージェントの行動選択のカウント Ni を経時的に保持し，それらを平均する，マルコフゲーム $\mathcal{P}$ でのエージェント i の確率的方策であると仮定された架空プレイ．この方策に対する最良応答を計算し，対応する効用最大化行動を実行する．

状態 s で結合行動 $\boldsymbol{a}$ を観測した後，各エージェント j に対して次のように更新する．

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

他のエージェントの行動上の分布が変化すると，効用を更新しなければならない．マルコフゲームにおける効用は，状態に依存するため，単純ゲームのときよりも計算が大幅に困難になる．25.2.1 項で記述したように，他のエージェントの固定された方策 $\boldsymbol{\pi}^{-i}$ を割り当てることでマルコフ決定過程を誘導する．架空プレイにおいて，$\boldsymbol{\pi}^{-i}$ は式 (25.9) で決定される．更新のたびにマルコフ決定過程を解く代わりに，周期的に更新することが一般的で，非同期価値反復か

25.5 勾配上昇

前章で単純ゲームに対して行ったのと同様の方法で，方策を学習するために，**勾配上昇** (gradient ascent) (アルゴリズム 25.8) を使用できる．ここでは状態を考慮しなければならず，行動価値関数を学習する必要がある．各時間ステップ t で，すべてのエージェントは状態 s_t で結合行動 $\boldsymbol{a}_t$ をとる．単純ゲームに対する勾配上昇のように，エージェント i は，他のエージェントの方策 $\boldsymbol{\pi}_t^{-i}$ は観測された行動 $\boldsymbol{a}_t^{-i}$ であると仮定している．勾配は

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

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

である．勾配ステップは次のように，状態 s が含まれ，期待効用の推定値 Q_t^i が使用されることを除いて，前章と同様の様式に従う．

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

再びこの更新では，s_t での方策 π_{t+1}^i が有効な確率分布であることを保証するために投影が必要になるかもしれない．

```
mutable struct MGGradientAscent
    𝒫  # Markov game
    i  # agent index
    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 = 𝒜.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]
```

アルゴリズム 25.8　マルコフゲーム $\mathcal{P}$ でのエージェント i の勾配上昇．このアルゴリズムは期待効用を改善するために，勾配上昇に従って訪れた状態での行動上の分布を段階的に更新する．アルゴリズム 23.6 の射影関数は，結果として得られる方策が有効な確率分布であることを保証するために使用される．

```
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
```

アルゴリズム 25.9　マルコフゲーム $\mathcal{P}$ でのエージェント `i` のナッシュ Q 学習．このアルゴリズムは，すべてのエージェントの状態–行動価値関数を学習するために，結合行動 Q 学習を実行する．単純ゲームが `Q` で構築され，アルゴリズム 24.5 を使用してナッシュ均衡を計算する．次に，均衡を用いて価値関数を更新する．この実装では，`N` に保存された状態–結合行動のペアが実現した回数に比例する可変学習率も使用する．加えて，すべての状態と行動が探索されることを保証するために，ϵ 貪欲探索を用いる．

25.7　要　約

- マルコフゲームはマルコフ決定過程を複数エージェントに拡張したもの，あるいは単純ゲームを逐次問題に拡張したものである．これらの問題において，複数のエージェントが競合し，時間を経ながら個々に報酬を受け取る．
- ナッシュ均衡はマルコフゲームに対して定式化できるが，すべての状態におけるすべてのエージェントに関する行動を考慮しなければならない．
- ナッシュ均衡を見つけるための問題は非線形最適化問題として定式化できる．
- 既知の遷移関数を用い，行動価値の推定値を導入することによって，マルコフゲームに対して架空プレイを一般化できる．
- 勾配上昇アプローチは確率的方策を繰り返し改善しており，モデルを仮定

る．アルゴリズム 26.3 は，部分観測マルコフゲームの d ステップのナッシュ均衡を計算している．例 26.4 に示すように，このアルゴリズムは単純ゲームを構築するための可能な d ステップの結合条件付きプランをすべて列挙する．この単純ゲームのナッシュ均衡は部分観測マルコフゲームのナッシュ均衡でもある．

```
struct POMGNashEquilibrium
    b # initial belief
    d # depth of conditional plans
end

function create_conditional_plans(𝒫, d)
    ℐ, 𝒜, 𝒪 = 𝒫.ℐ, 𝒫.𝒜, 𝒫.𝒪
    Π = [[ConditionalPlan(ai) for ai in 𝒜[i]] for i in ℐ]
    for t in 1:d
        Π = expand_conditional_plans(𝒫, Π)
    end
    return Π
end

function expand_conditional_plans(𝒫, Π)
    ℐ, 𝒜, 𝒪 = 𝒜.ℐ, 𝒫.𝒜, 𝒫.𝒪
    return [[ConditionalPlan(ai, Dict(oi ⇒ πi for oi in 𝒪[i]))
        for πi in Π[i] for ai in 𝒜[i]] for i in ℐ]
end

function solve(M::POMGNashEquilibrium, 𝒫::POMG)
    ℐ, γ, b, d = 𝒫.ℐ, 𝒫.γ, M.b, M.d
    Π = create_conditional_plans(𝒫, d)
    U = Dict(π ⟹ utility(𝒫, b, π) for π in joint(Π))
    𝒢 = SimpleGame(γ, ℐ, Π, π → U[π])
    π = solve(NashEquilibrium(), 𝒢)
    return Tuple(argmax(πi.p) for πi in π)
end
```

アルゴリズム 26.3　ある深さ d までのすべての条件付きプランの単純ゲームを生成することによって，ナッシュ均衡は初期状態分布 b をもつ部分観測マルコフゲーム 𝒫 に対して計算される．アルゴリズム 24.5 を用いて，この単純ゲームのナッシュ均衡を解く．簡単化のために，最も可能性の高い結合方策を選択する．代替的に，実行開始時に結合方策をランダムに選択することもできる．

例 26.4　行動が条件付きプランに対応する単純ゲームに変換することによる，複数世話人の泣いている赤ちゃん問題に対するナッシュ均衡の計算

2 ステップの時間範囲をもつ複数世話人の泣いている赤ちゃん問題を考えよう．各エージェント i に対して，3 つの行動

$$\mathcal{A}^i = \{a_1^i, a_2^i, a_3^i\} = \{\text{ 授乳する}, \text{歌う}, \text{無視する }\}$$

と 2 つの観測

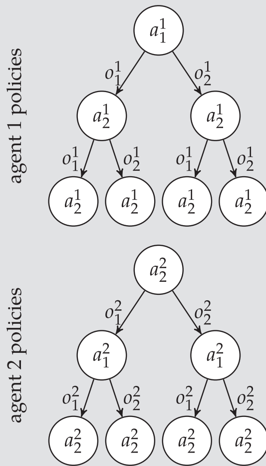
$$\mathcal{O}^i = \{o_1^i, o_2^i\} = \{\text{ 泣いている}, \text{静か }\}$$

があることを思い出そう．この部分観測マルコフゲームを単純ゲームに変換すると，次のゲームの表ができる．各世話人は完全な条件付きプランに対応する単純ゲームの行動を選択する．各エージェントの単純ゲームにおける報酬は結合方策に関連付けられた効用である．

```
struct POMGDynamicProgramming
    b # initial belief
    d # depth of conditional plans
end

function solve(M::POMGDynamicProgramming, 𝒫::POMG)
    ℐ, 𝒮, 𝒜, R, γ, b, d = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.γ, M.b, M.d
    Π = [[ConditionalPlan(ai) for ai in 𝒜[i]] for i in ℐ]
    for t in 1:d
        Π = expand_conditional_plans(𝒫, Π)
        prune_dominated!(Π, 𝒫)
    end
    𝒢 = SimpleGame(γ, ℐ, Π, π → utility(𝒫, b, π))
    π = solve(NashEquilibrium(), 𝒢)
    return Tuple(argmax(πi.p) for πi in π)
end

function prune_dominated!(Π, 𝒫::POMG)
    done = false
    while !done
        done = true
        for i in shuffle(𝒫.ℐ)
            for πi in shuffle(Π[i])
                if length(Π[i]) > 1 && is_dominated(𝒫, Π, i, πi)
                    filter!(πi′ → πi′ ≠ πi, Π[i])
                    done = false
                    break
                end
            end
        end
    end
end

function is_dominated(𝒫::POMG, Π, i, πi)
    ℐ, 𝒮 = 𝒫.ℐ, 𝒫.𝒮
    jointΠnoti = joint([Π[j] for j in ℐ if j ≠ i])
    π(πi′, πnoti) = [j==i ? πi′ : πnoti[j>i ? j-1 : j] for j in ℐ]
    Ui = Dict((πi′, πnoti, s) ⇒ evaluate_plan(𝒫, π(πi′, πnoti), s)[i]
              for πi′ in Π[i], πnoti in jointΠnoti, s in 𝒮)
    model = Model(Ipopt.Optimizer)
    @variable(model, δ)
    @variable(model, b[jointΠnoti, 𝒮] ≥ 0)
    @objective(model, Max, δ)
    @constraint(model, [πi′=Π[i]],
        sum(b[πnoti, s] * (Ui[πi′, πnoti, s] - Ui[πi, πnoti, s])
        for πnoti in jointΠnoti for s in 𝒮) ≥ δ)
    @constraint(model, sum(b) == 1)
    optimize!(model)
    return value(δ) ≥ 0
end
```

アルゴリズム 26.4　動的計画法は，初期信念 b と時間区間の深さ d が与えられたとき，部分観測マルコフゲーム 𝒫 のナッシュ均衡 π を計算する．各ステップで方策ツリーとその期待効用を繰り返し計算する．各反復での枝刈り段階で，少なくとも 1 つの他の利用可能な方策ツリーよりも期待効用が低い方策ツリーである支配された方策が取り除かれる．

枝刈りステップでは，すべての支配されている方策を削除する．エージェント i に属する方策 π^i は，π^i と少なくとも同じくらいに常に機能する別の方策 $\pi^{i'}$ が存在するならば，枝刈りされる可能性がある．計算的にはコストが高いが，この条件は線形計画問題を解くことによって確認できる．この過程は，部分観測マルコフ決定過程のコントローラノードの枝刈り (アルゴリズム 23.4) に



この場合，枝刈りステップは最良の結合方策を見つけている．このアプローチにより，アルゴリズムの次の反復で考慮する必要がある可能な結合方策の数が大幅に削減される．

26.5 要　約

- 部分観測マルコフゲームは部分観測マルコフ決定過程を複数のエージェントをもつように一般化し，マルコフゲームを部分観測可能になるように一般化する．
- 一般にエージェントは部分観測マルコフゲームにおいて信念を保持できないため，方策は通常，条件付きプランまたは有限状態コントローラの形式をとる．
- 部分観測マルコフゲームの d ステップ条件付きプランの形式におけるナッシュ均衡は，結合行動がすべての可能な部分観測マルコフゲームの結合方策で構成される単純ゲームのナッシュ均衡を見つけることによって得られる．
- 動的計画法のアプローチを用いると，探索空間を制限するために支配されたプランを枝刈りしながら，より深い条件付きプランの集合を繰り返し構築することによって，ナッシュ均衡をより効率的に計算できる．

26.6 演　習

26.1　部分観測マルコフゲームが部分観測マルコフ決定過程とマルコフゲームの両方を一般化することを示せ．

【解】 任意の部分観測マルコフ決定過程に関して，1 エージェント $\mathcal{I} = \{1\}$ の部分観測マルコフゲームを定義できる．状態 $\mathcal{S}$ は同一であり，行動 $\boldsymbol{A} = (\mathcal{A}^1)$ と観測 $\boldsymbol{O} = (\mathcal{O}^1)$ も同様である．したがって，部分観測マルコフゲームの状態遷移，観測関数，報酬は直接定義される．ナッシュ均衡の最適化では 1 エージェントをもち，部分観測マルコ

の最良応答として次のように更新される．

$$\pi^i \leftarrow \arg\max_{\pi^{i'}} \mathcal{U}^{\pi^{i'}, \boldsymbol{\pi}^{-i}}(b) \tag{27.5}$$

このとき，同じ場合は現在の方策が採用される．エージェントが自身の方策を変更しなくなったら，この過程は停止する．

このアルゴリズムは速く，収束が保証されるが，最良の結合方策を常に見つけるとは限らない．これはナッシュ均衡を見つけるための反復最良応答を当てにしているものの，ナッシュ均衡が多く存在し，それらは異なる効用をもつかもしれない．このアプローチはそれらのうちの 1 つのみを見つけることになる．

27.5　発見的探索

すべての結合方策を展開する代わりに，**発見的探索** (heuristic search) (アルゴリズム 27.4) は固定された数の方策を探索する[5]．これらの方策は反復を通して保存され，指数関数的な増大を防ぐ．発見的探索は，深さ d に達するまでの最良結合方策の展開のみを試みることによって，探索を導いていく．

[5] このアプローチは**メモリ限定動的計画法** (memory-bounded dynamic programming: MBDP) として知られている．S. Seuken and S. Zilberstein, "Memory-Bounded Dynamic Programming for Dec-POMDPs," in *International Joint Conference on Artificial Intelligence* (*IJCAI*), 2007．他にも**マルチエージェント A*** (multiagent A*: MAA*) のような発見的探索がある．D. Szer, F. Charpillet, and S. Zilberstein, "MAA*: A Heuristic Search Algorithm for Solving Decentralized POMDPs," in *Conference on Uncertainty in Artificial Intelligence* (*UAI*), 2005.

```
struct DecPOMDPHeuristicSearch
    b     # initial belief
    d     # depth of conditional plans
    π_max # number of policies
end

function solve(M::DecPOMDPHeuristicSearch, 𝒫::DecPOMDP)
    ℐ, 𝒮, 𝒜, 𝒪, T, O, R, γ = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R
     , 𝒫.γ
    b, d, π_max = M.b, M.d, M.π_max
    R′(s, a) = [R(s, a) for i in ℐ]
    𝒫′ = POMG(γ, ℐ, 𝒮, 𝒜, 𝒪, T, O, R′)
    Π = [[ConditionalPlan(ai) for ai in 𝒜[i]] for i in ℐ]
    for t in 1:d
        allΠ = expand_conditional_plans(𝒫, Π)
        Π = [[] for i in ℐ]
        for z in 1:π_max
            b′ = explore(M, 𝒫, t)
            π = argmax(π → first(utility(𝒫′, b′, π)), joint(allΠ))
            for i in ℐ
                push!(Π[i], π[i])
                filter!(πi → πi != π[i], allΠ[i])
            end
        end
    end
    return argmax(π→first(utility(𝒫′, b, π)), joint(Π))
end

function explore(M::DecPOMDPHeuristicSearch, 𝒫::DecPOMDP, t)
    ℐ, 𝒮, 𝒜, 𝒪, T, O, R, γ = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.𝒪, 𝒫.T, 𝒫.O, 𝒫.R
     , 𝒫.γ
    b = copy(M.b)
    b′ = similar(b)
    s = rand(SetCategorical(𝒮, b))
    for τ in 1:t
```

アルゴリズム 27.4　メモリ限定発見的探索は分権的部分観測マルコフ決定過程 $\mathcal{P}$ に対する条件付きプランの空間を探索するための発見的関数を用いる．`solve` 関数は深さ `d` の結合条件付きプランに対して，初期信念 `b` での価値を最大化しようとする．`explore` 関数は，ランダムな行動をとり，行動と観測をシミュレートすることによって，`t` ステップ先の未来の信念を生成する．アルゴリズムはメモリが限定され，1 エージェントごとに `π_max` の条件付きプランだけを保持する．

```
struct DecPOMDPNonlinearProgramming
    b # initial belief
    ℓ # number of nodes for each agent
end

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

function solve(M::DecPOMDPNonlinearProgramming, 𝒫::DecPOMDP)
    𝒫, γ, b = 𝒫, 𝒫.γ, M.b
    ℐ, 𝒮, 𝒜, 𝒪, R, T, O = tensorform(𝒫)
    X = [collect(1:M.ℓ) for i in ℐ]
    jointX, joint𝒜, joint𝒪 = joint(X), joint(𝒜), joint(𝒪)
    x1 = jointX[1]
    model = Model(Ipopt.Optimizer)
    @variable(model, U[jointX,𝒮])
    @variable(model, ψ[i=ℐ,X[i],𝒜[i]] ≥ 0)
    @variable(model, η[i=ℐ,X[i],𝒜[i],𝒪[i],X[i]] ≥ 0)
    @objective(model, Max, b⋅U[x1,:])
    @NLconstraint(model, [x=jointX,s=𝒮],
        U[x,s] == (sum(prod(ψ[i,x[i],a[i]] for i in ℐ)
                   *(R[s,y] + γ*sum(T[s,y,s′]*sum(O[y,s′,z]
                       *sum(prod(η[i,x[i],a[i],o[i],x′[i]] for i in ℐ)
                            *U[x′,s′] for x′ in jointX)
                   for (z, o) in enumerate(joint𝒪)) for s′ in 𝒮))
                for (y, a) in enumerate(joint𝒜))))
    @constraint(model, [i=ℐ,xi=X[i]],
                sum(ψ[i,xi,ai] for ai in 𝒜[i]) == 1)
    @constraint(model, [i=ℐ,xi=X[i],ai=𝒜[i],oi=𝒪[i]],
                sum(η[i,xi,ai,oi,xi′] for xi′ in X[i]) == 1)
    optimize!(model)
    ψ′, η′ = value.(ψ), value.(η)
    return [ControllerPolicy(𝒫, X[i],
            Dict((xi,𝒫.𝒜[i][ai]) => ψ′[i,xi,ai]
                 for xi in X[i], ai in 𝒜[i]),
            Dict((xi,𝒫.𝒜[i][ai],𝒫.𝒪[i][oi],xi′) => η′[i,xi,ai,oi,xi′]
                 for xi in X[i], ai in 𝒜[i], oi in 𝒪[i], xi′ in X[i
     ]))
        for i in ℐ]
end
```

アルゴリズム 27.5　非線形計画法は，各エージェントに対して初期信念 b とコントローラノードの数 ℓ が与えられたとき，分権的部分観測マルコフ決定過程 𝒫 に対する最適結合コントローラ方策 π を計算する．これはアルゴリズム 23.5 の非線形計画法の解法を拡張している．

各エージェントに対して固定されたノード集合 X^i，初期信念 b，初期結合ノード $\mathbf{x}_1$ が与えられたとき，最適化問題は次のようになる．

$$
\begin{aligned}
\underset{\mathcal{U},\boldsymbol{\psi},\boldsymbol{\eta}}{\text{maximize}} \quad & \sum_s b(s)\mathcal{U}(\boldsymbol{x}_1,s) \\
\text{subject to} \quad & \mathcal{U}(\boldsymbol{x},s) = \sum_{\boldsymbol{a}}\prod_i \psi^i(a^i \mid x^i)\Big(R(s,\boldsymbol{a}) + \gamma\sum_{s'} T(s' \mid s,\boldsymbol{a})\sum_{\boldsymbol{o}} O(\boldsymbol{o} \mid \boldsymbol{a},s')\sum_{\boldsymbol{x}'}\prod_i \eta^i(x^{i\prime} \mid x^i,a^i,o^i)\mathcal{U}(\boldsymbol{x}',s')\Big) \\
& \hspace{30em} \text{for all } \boldsymbol{x},s \\
& \psi^i(a^i \mid x^i) \geq 0 \text{ for all } i,x^i,a^i \\
& \sum_{a^i}\psi^i(a^i \mid x^i) = 1 \text{ for all } i,x^i \\
& \eta^i(x^{i\prime} \mid x^i,a^i,o^i) \geq 0 \text{ for all } i,x^i,a^i,o^i,x^{i\prime} \\
& \sum_{x^{i\prime}}\eta^i(x^{i\prime} \mid x^i,a^i,o^i) = 1 \text{ for all } i,x^i,a^i,o^i
\end{aligned}
\tag{27.6}
$$

27.7 要　約

- 分権的部分観測マルコフ決定過程は完全に協力的な部分観測マルコフゲームであり，共有された目標に向かって協働するエージェントのチームをモデル化し，各エージェントは局所的な情報のみを用いて個別に行動する．
- 部分観測マルコフゲームのように信念状態を決定することは実行不可能であるので，方策は通常条件付きプランまたはコントローラとして表現され，各エージェントがそれぞれの観測の連続をそれぞれの行動に写像する．
- 分権的部分観測マルコフ決定過程の多くのサブクラスが存在し，それぞれ計算量が異なる．
- 動的計画法は価値関数を繰り返し計算し，繰り返すたびに支配された方策は線形計画法を用いて枝刈りされる．
- 反復最良応答は単一のエージェントに対するそのときの最良な効用最大化応答方策を計算し，繰り返しながら結合均衡へ収束していく．
- 発見的探索は，発見的な手法に支援されながら，各反復で方策の固定された部分集合を探索する．
- 非線形計画法は固定サイズのコントローラを生成するために使用される．

27.8 演　習

27.1　結合完全観測性をもつ分権的マルコフ決定過程が，状態を知っているエージェントと異なるのはなぜか．

【解】　結合完全観測性とは，エージェントが個々の観測を共有すれば，チームは真の状態を知ることができることを意味する．これはプランニング中にオフラインで実行できる．したがって分権的マルコフ決定過程では，真の状態はプランニング中に本質的に知られている．問題は，エージェントがそれぞれの観測を共有する必要があり，それが実行中にオンラインで共有できないことである．それゆえ，プランニングには他のエージェントによる不確実な観測について推論する必要がある．

27.2　遷移，観測，報酬の独立性をもつ分権的マルコフ決定過程に対する高速アルゴリズムを提案せよ．それが正しいことを証明せよ．

【解】　ファクタ化された分権的マルコフ決定過程が 3 つの独立性の仮定をすべて満た

A 数学的概念

本付録は本書で用いられる数学的概念のいくつかを手短に概観する．

A.1 測度空間

測度空間の定義を導入する前に，集合 Ω 上の σ 代数を最初に論じる．σ 代数は次の性質を満たす Ω の部分集合の族 Σ である．

(1) $\Omega \in \Sigma$ である．

(2) $E \in \Sigma$ ならば，$\Omega \setminus E \in \Sigma$ である (**補集合に関して閉じている** (closed under complementation)) .

(3) $E_1, E_2, E_3, \ldots \in \Sigma$ ならば，$E_1 \cup E_2 \cup E_3 \cup \cdots \in \Sigma$ である (**加算和集合に関して閉じている** (closed under countable unions)) .

要素 $E \in \Sigma$ は**可測集合** (measurable set) と呼ばれる．

測度空間 (measure space) は集合 Ω，σ 代数，**測度** (measure) $\mu : \Omega \to \mathbb{R} \cup \{\infty\}$ によって定義される．μ が測度であるためには，次の性質が成立する必要がある．

(1) $E \in \Sigma$ ならば，$\mu(E) \geq 0$ である (**非負性** (nonnegativity)) .

(2) $\mu(\emptyset) = 0$ である．

(3) $E_1, E_2, E_3, \ldots \in \Sigma$ が互いに素ならば，$\mu(E_1 \cup E_2 \cup E_3 \cup \cdots) = \mu(E_1) + \mu(E_2) + \mu(E_3) + \cdots$ である (**加算加法性** (countable additivity)) .

A.2 確率空間

確率空間 (probability space) は $\mu(\Omega) = 1$ となる要件をもつ測度空間 (Ω, Σ, μ) である．確率空間の文脈では，Ω は**標本空間** (sample space)，Σ は**事象空間** (event space)，μ (またはより広く使われる P) は**確率測度** (probability measure) と呼ばれる．確率の公理[1] は $\mu(\Omega) = 1$ となる要件とともに，測度空間の非負性および可算加法性の性質に言及している．

[1] これらの公理はしばしばコルモゴロフの公理と呼ばれる．A. Kolmogorov, *Foundations of the Theory of Probability*, 2nd ed. Chelsea, 1956.

A.3 距離空間

メトリック (metric) をもつ集合は**距離空間** (metric space) と呼ばれる．メトリック d はしばしば**距離メトリック** (distance metric) と呼ばれ，次のような X の要素の対から非負の実数へ写像する関数である．すべての $x, y, z \in X$ に対して

(1) $d(x,y) = 0$ のとき，かつそのときに限り $x = y$ である (**不可識別者同一** (identity of indiscernibles)) .

(2) $d(x,y) = d(y,x)$ である (**対称性** (symmetry)) .

(3) $d(x,y) \leq d(x,z) + d(z,y)$ である (**三角不等式** (triangle inequality)) .

A.4　ノルム付きベクトル空間

ノルム付きベクトル空間 (normed vector space) は**ベクトル空間** (vector space) X と X の要素を次のような非負実数へ写像するノルム $\|\cdot\|$ からなる．すべてのスカラ α とベクトル $\boldsymbol{x}, \boldsymbol{y} \in X$ に対して

(1) $\|\boldsymbol{x}\| = 0$ のとき，かつそのときに限り $\boldsymbol{x} = \boldsymbol{0}$ である．

(2) $\|\alpha\boldsymbol{x}\| = |\alpha|\,\|\boldsymbol{x}\|$ である (**絶対値均一性** (absolutely homogeneous)) .

(3) $\|\boldsymbol{x}+\boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\|$ である (**三角不等式** (triangle inequality)) .

L_p ノルムはスカラ $p \geq 1$ でパラメータ化された一般に使用されるノルムの集合である．ベクトル $\boldsymbol{x}$ の L_p ノルムは

$$\|\boldsymbol{x}\|_p = \lim_{\rho \to p} \left(|x_1|^\rho + |x_2|^\rho + \cdots + |x_n|^\rho\right)^{\frac{1}{\rho}} \tag{A.1}$$

であり，極限は無限ノルム L_∞ を定義するために必要である．いくつかの L_p ノルムを図 A.1 に示す．

L_1: $\|\mathbf{x}\|_1 = |x_1| + |x_2| + \cdots + |x_n|$
このメトリックはしばしば**タクシーキャブノルム** (taxicab norm)と呼ばれる．

L_2: $\|\mathbf{x}\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$
このメトリックはしばしば**ユークリッドノルム** (Euclidean norm)と呼ばれる．

L_∞: $\|\mathbf{x}\|_\infty = \max(|x_1|, |x_2|, \cdots, |x_n|)$
このメトリックはしばしば**最大ノルム** (max norm)，**チェビシェフノルム**(Chebyshev norm)，**チェス盤ノルム**(chessboard norm)と呼ばれる．最後の名称は，チェスにおいてキングが2つのマス間を移動するために必要な最小の手数に由来している．

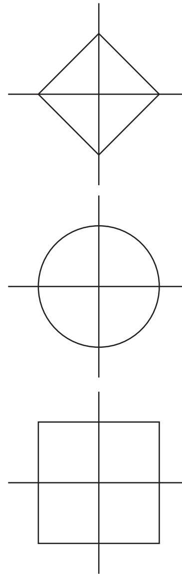


図 A.1　一般的な L_p ノルム．図には 2 次元のノルムの等高線の形を示している．このノルムのもとで，等高線上のすべての点は原点から等距離になっている．

ノルムはメトリック $d(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\|$ を定義することによってベクトル空間における距離メトリックを誘導するために用いられる．たとえば，距離を定義するために L_p ノルムを用いることができる．

A.5　正　定　性

対称行列 A は，$\boldsymbol{x}^\top A\boldsymbol{x}$ が原点を除くすべての点に対して正であるならば，**正定** (positive definite) である．言い換えれば，すべての $\boldsymbol{x} \neq \boldsymbol{0}$ に対して，$\boldsymbol{x}^\top A\boldsymbol{x} > 0$ である．対称行列 A は，$\boldsymbol{x}^\top A\boldsymbol{x}$ が常に非負ならば，**半正定** (positive semidefinite) である．言い換えれば，すべての $\boldsymbol{x}$ に対して，$\boldsymbol{x}^\top A\boldsymbol{x} \geq 0$ である．

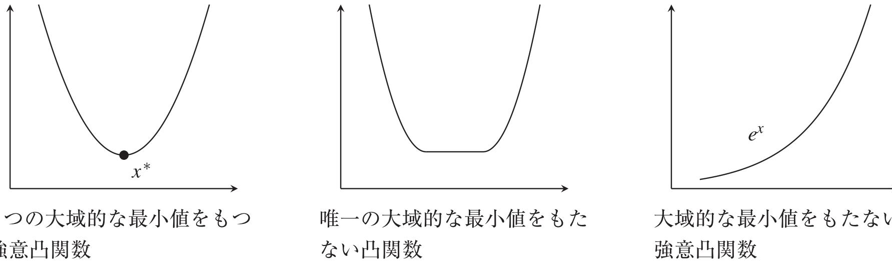


図 A.4　すべての凸関数が単一の大域的最小値をもつとは限らない．

強意凹 (strictly concave) である．

A.7　情　報　量

値 x に対して，確率 $P(x)$ を割り当てる離散分布があるならば，x を観測することの**情報量** (information content) [3] は

$$I(x) = -\log P(x) \tag{A.7}$$

で与えられる．

情報量の単位は対数の底に依存する．通常，自然対数 (底が e) を仮定し，単位は**ナチュラル** (natural) の略である**ナット** (nat) である．情報理論の文脈では，底はしばしば 2 であり，単位は**ビット** (bit) となる．この量は，メッセージ上の分布が指定された分布に従うとき，最適なメッセージのコード化に従って，値 x を送信するために必要なビット数と考えることができる．

[3] 情報理論分野の創始者である Claude Shannon に敬意を表して，情報量はシャノン情報量と呼ばれることもある．C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, no. 4, pp. 623–656, 1948.

A.8　エントロピー

エントロピー (entropy) は不確実性の情報理論的な尺度である．離散確率変数 X に関するエントロピーは次の期待情報量である．

$$H(X) = \mathbb{E}_x[I(x)] = \sum_x P(x)I(x) = -\sum_x P(x)\log P(x) \tag{A.8}$$

ここで，$P(x)$ は x に割り当てられた質量である．

$p(x)$ が x に対する密度である連続分布に関しては，**微分エントロピー** (differential entropy) (**連続エントロピー** (continuous entropy) としても知られている) は次のように定義される．

$$h(X) = \int p(x)I(x)\mathrm{d}x = -\int p(x)\log p(x)\mathrm{d}x \tag{A.9}$$

A.9　交差エントロピー

ある分布と別の分布との**交差エントロピー** (cross entropy) は期待情報量で定義できる．質量関数 $P(x)$ をもつ 1 つの離散分布と質量関数 $Q(x)$ をもつ別の分布があるとき，Q に対する P の交差エントロピーは次のように与えられる．

$$H(P,Q) = -\mathbb{E}_{x\sim P}[\log Q(x)] = -\sum_x P(x)\log Q(x) \tag{A.10}$$

密度関数 $p(x)$ と $q(x)$ をもつ連続分布に関しては

$$H(p,q) = -\int p(x) \log q(x) \mathrm{d}x \tag{A.11}$$

である．

A.10 相対エントロピー

相対エントロピー (relative entropy)，また**カルバック–ライブラーダイバージェンス** (**KL ダイバージェンス**) (Kullback–Leibler (KL) divergence) は，ある確率分布が参照分布とどの程度異なるかの尺度である[4]．$P(x)$ と $Q(x)$ が質量関数であるならば，Q から P に対する KL ダイバージェンスは対数差の期待値

$$D_{KL}(P \parallel Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)} = -\sum_x P(x) \log \frac{Q(x)}{P(x)} \tag{A.12}$$

であり，期待値には P を用いている．P の台が Q の台の部分集合であるときに限り，この量は定義される．和は，0 で割ることを避けるために，P の台上でとる．

密度関数 $p(x)$ と $q(x)$ をもつ連続分布に関しては

$$D_{KL}(p \parallel q) = \int p(x) \log \frac{p(x)}{q(x)} \mathrm{d}x = -\int p(x) \log \frac{q(x)}{p(x)} \mathrm{d}x \tag{A.13}$$

となる．同様に，この量は p の台が q の台の部分集合であるときに限り，定義される．積分は，0 で割ることを避けるために，p の台上でとる．

[4] この尺度を導入した 2 人のアメリカ人数学者 Solomon Kullback (1907–1994) と Richard A. Leibler (1914–2003) にちなんで名付けられた．S. Kullback and R. A. Leibler, “On Information and Sufficiency,” *Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951. S. Kullback, *Information Theory and Statistics*. Wiley, 1959.

A.11 勾 配 上 昇

勾配上昇 (gradient ascent) は，関数 f が微分可能な関数であるとき，f を最大化しようとする一般的なアプローチである．点 $\boldsymbol{x}$ から始め，次の更新規則を繰り返し適用する．

$$\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha \nabla f(\boldsymbol{x}) \tag{A.14}$$

ここで，$\alpha > 0$ は**ステップ係数** (step factor) と呼ばれる．この最適化アプローチの考えは，局所的最大値に到達するまで，勾配の方向にステップをとるというものである．この手法を用いて大域的最大値を見つけることは保証されない．α の値が小さければ，一般に局所的最大値に近づくためにより多くの反復が必要になる．α の値が大きければ，しばしば局所的最適値に十分近づくことなく，その周りを行ったり来たりすることになる．α が反復を通じて一定ならば，それはしばしば**学習率** (learning rate) と呼ばれる．多くのアプリケーションでは，**減衰するステップ係数** (decaying step factor) をもち，各反復で $\boldsymbol{x}$ を更新するのに加えて，

$$\alpha \leftarrow \gamma \alpha \tag{A.15}$$

に従って，α を更新する．ここで，$0 < \gamma < 1$ は**減衰係数** (decay factor) である．

A.12 テイラー展開

関数の**テイラー展開** (Taylor expansion) [5]，また**テイラー級数** (Taylor series) は本書で用いられる多くの近似に対して重要である．**微分積分学の第一基本定理** (first fundamental theorem of calculus) [6] から，

$$f(x+h) = f(x) + \int_0^h f'(x+a) \mathrm{d}a \tag{A.16}$$

[5] この概念を導入した英国の数学者 Brook Taylor (1685–1731) にちなんで名付けられた．

[6] 微分積分学の第一基本定理は関数をその導関数の積分に関連付ける．

$$f(b) - f(a) = \int_a^b f'(x) \mathrm{d}x$$

$$f(x) \approx f(a) + f'(a)(x-a) \tag{A.25}$$

2 次テイラー近似では，次のようにテイラー展開の最初の 3 項を用いる．

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2 \tag{A.26}$$

以降同様である．

多次元の場合，$\boldsymbol{a}$ についてのテイラー展開は次のように一般化される．

$$f(\boldsymbol{x}) = f(\boldsymbol{a}) + \nabla f(\boldsymbol{a})^\top(\boldsymbol{x}-\boldsymbol{a}) + \frac{1}{2}(\boldsymbol{x}-\boldsymbol{a})^\top \nabla^2 f(\boldsymbol{a})(\boldsymbol{x}-\boldsymbol{a}) + \cdots \tag{A.27}$$

最初の 2 項は $\boldsymbol{a}$ での接平面を作る．第 3 項において局所的な曲率が導入される．本書では，ここに示されている最初の 3 項のみを使用する．

A.13　モンテカルロ推定

モンテカルロ推定 (Monte Carlo estimation) を使用すると，入力 x が確率密度関数 p に従うときの関数 f の期待値を次のように評価できる．

$$\mathbb{E}_{x\sim p}[f(x)] = \int f(x)p(x)\mathrm{d}x \approx \frac{1}{n}\sum_i f(x^{(i)}) \tag{A.28}$$

ここで, $x^{(1)},\ldots,x^{(n)}$ は p から引き出されるサンプルである．推定の分散は $\mathrm{Var}_{x\sim p}[f(x)]/n$ に等しい．

A.14　重点サンプリング

重点サンプリング (importance sampling) によって，異なる分布 q から引き出されたサンプルから，$\mathbb{E}_{x\sim p}[f(x)]$ を次のように計算できる．

$$\mathbb{E}_{x\sim p}[f(x)] = \int f(x)p(x)\mathrm{d}x \tag{A.29}$$

$$= \int f(x)p(x)\frac{q(x)}{q(x)}\mathrm{d}x \tag{A.30}$$

$$= \int f(x)\frac{p(x)}{q(x)}q(x)\mathrm{d}x \tag{A.31}$$

$$= \mathbb{E}_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right] \tag{A.32}$$

上式は q から引き出されたサンプル $x^{(1)},\ldots,x^{(n)}$ を用いて次のように近似できる．

$$\mathbb{E}_{x\sim p}[f(x)] = \mathbb{E}_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right] \approx \frac{1}{n}\sum_i f(x^{(i)})\frac{p(x)^{(i)}}{q(x)^{(i)}} \tag{A.33}$$

A.15　収 縮 写 像

収縮写像 (contraction mapping) f は

$$d(f(x), f(y)) \leq \alpha d(x,y) \tag{A.34}$$

となるような距離空間上の関数に対して定義される．ここで, d は距離空間と $0 \leq \alpha < 1$ に関連する距離メトリックである．したがって，収縮写像はある集合の任意の 2 要素間の距離を縮小する．このような関数はしばしば**収縮** (contraction) または**収縮子** (contractor) と呼ばれる．

例 C.3　最初に NP 完全であると知られていた 3SAT 問題

ブール充足可能性 (Boolean satisfiability) の問題はブール式が**充足可能** (satisfiable) であるかどうかを決定することを含む．ブール式は n 個のブール変数 $x_1,\dots,x_n$ を含む論理積 ($\wedge$)，論理和 ($\vee$)，否定 ($\neg$) から構成される．リテラルは変数 x_i またはその否定 $\neg x_i$ である．3SAT 節は 3 つのリテラルまでの論理和である (たとえば，$x_3 \vee \neg x_5 \vee x_6$)．3SAT 式は次のような 3SAT 節の論理積である．

$$F(x_1,x_2,x_3,x_4) = \begin{array}{l} (\;\;x_1 \vee \;\;x_2 \vee \;\;x_3) \quad \wedge \\ (\neg x_1 \vee \neg x_2 \vee \;\;x_3) \quad \wedge \\ (\;\;x_1 \vee \neg x_2 \vee \;\;x_4) \end{array}$$

3SAT の課題は式を真にする真理値の変数への可能な割当てが存在するかどうかを決定することである．上式において，

$$F(\text{true}, \text{false}, \text{false}, \text{true}) = \text{true}$$

である．よって，この式は充足可能である．一部の 3SAT 問題では，迅速な検査で満足できる割当てを容易に見つけることができるが，一般には解くことは困難である．満足できる割当てができるかどうかを判断する 1 つの方法は，すべての変数の 2^n 個の可能な真理値を列挙することである．満足している真理値の割当てが存在するかどうかを判断することは困難であるが，真理値の割当てが満足であるかどうかの検証は線形時間で行うことができる．

C.3　空間計算量に基づくクラス

別の複雑さのクラスの集合は空間に関連しており，アルゴリズムを最後まで実行するために必要なメモリの量を指している．複雑さのクラス PSPACE には，時間を考慮せずに多項式容量の空間で解けるすべての問題の集合が含まれている．時間と空間の複雑さには根本的な違いがあり，時間は再利用できないが，空間は再利用できる．P と NP は PSPACE の部分集合であることがわかっている．PSPACE に NP ではない問題が含まれていることは，いまだ知られていないが，そうではないかと考えられている．多項式時間変換を使用すると，NP 困難と NP 完全のクラスの場合と同様に，**PSPACE 困難** (PSPACE-hard) と **PSPACE 完全** (PSPACE-complete) のクラスを定義できる．

C.4　決定可能性

決定不可能 (undecidable) な問題は有限時間内に常に解けるとは限らない．おそらく，最も有名な決定不可能な問題の 1 つは**停止問題** (halting problem) であり，十分な表現力をもつ言語[4] で書かれたプログラムを入力として受け取り，それが終了するかどうかを決定することと結び付いている．このような分析を一般に実行できるアルゴリズムは存在しないことが証明された．ある種のプログラムが終了するかどうかを正しく判断できるアルゴリズムは存在するが，任意のプログラムが終了するかどうかを判断できるアルゴリズムは存在しない．

[4] 技術的要件は，その言語が**チューリング完全** (Turing complete) であるかまたは，**計算的に普遍的** (computationally universal) であることである．これは任意のチューリングマシンをシミュレートするのに使用できることを意味する．

E 探索アルゴリズム

探索問題 (search problem) は，引き続く決定論的な遷移にわたって得られる報酬を最大化するための適切な一連の行動を見つけることに関連する．探索問題は決定論的遷移関数をもつ (第 II 部で取り扱った) マルコフ決定過程である．よく知られている探索問題には，スライディングタイルパズル，ルービックキューブ，倉庫番，目的地までの最短経路問題などがある．

E.1 探索問題

探索問題では，観測している状態 s_t に基づいて，時刻 t で行動 a_t を選択し，報酬 r_t を受け取る．**行動空間** (action space) $\mathcal{A}$ は可能な行動の集合であり，**状態空間** (state space) $\mathcal{S}$ は可能な状態の集合である．いくつかのアルゴリズムでは，これらの集合が有限であることを仮定するが，この仮定は一般には必要とされない．状態は決定論的に進展し，現在の状態ととられた行動にのみ依存する．状態 s からの有効な行動の集合を表すために，$\mathcal{A}(s)$ を用いる．有効な行動がないとき，状態は**吸収している** (absorbing) と考えられ，将来のすべての時間ステップに対して報酬はゼロになる．たとえば，ゴールの状態は典型的に吸収している．

決定論的状態遷移関数 $T(s,a)$ は後続状態 s' を与える．報酬関数 $R(s,a)$ は，状態 s から行動 a を実行したときに受け取る報酬を与える．通常探索問題には，将来の報酬にペナルティを与える割引係数 γ が含まれていない．目的は報酬の合計，すなわち**リターン** (return) を最大化する一連の行動を選択することである．アルゴリズム E.1 は探索問題を表すデータ構造体を提供する．

```
struct Search
    𝒮 # state space
    𝒜 # valid action function
    T # transition function
    R # reward function
end
```

アルゴリズム E.1　探索問題のデータ構造体

E.2 探索グラフ

有限の状態空間と行動空間をもつ探索問題は**探索グラフ** (search graph) として表すことができる．ノードは状態に対応し，エッジは状態間の遷移に対応する．ソース状態から行き先状態への各エッジには，その状態遷移をもたらす行動とソース状態からその行動をとったときの期待報酬の両方が関連付けられている．図 E.1 には，3×3 のスライディングタイルパズルに関する探索グラフの部分集合を示している．

多くのグラフ探索アルゴリズムは初期状態から探索を実行し，そこから広がってい

図 E.3 では，スライディングタイルパズル上の前方探索を実行することによって得られた探索ツリーの例を示している．深さ優先探索は無駄になる可能性があり，与えられた深さの到達可能なすべての状態に到達することになる．深さ d までの探索は，$|\mathcal{A}|$ の行動をもつ問題に対して $\mathcal{O}(|\mathcal{A}|^d)$ のノードをもつ探索ツリーを生成することになる．

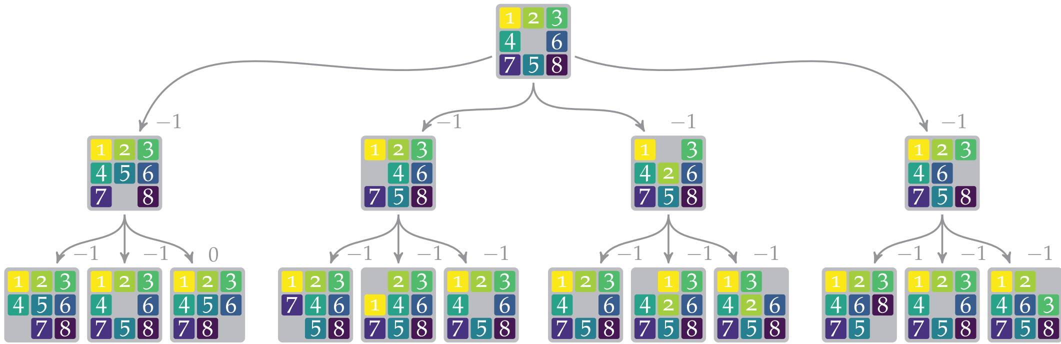


図 E.3　スライディングタイルパズルで深さ 2 まで前方探索を実行することで生じる探索ツリー．2 ステップで到達可能なすべての状態に到達し，一部の状態は複数回到達している．終端ノードへのパスが 1 つあることがわかる．そのパスのリターンは −1 であるが，他のすべてのパスのリターンは −2 である．

E.4　分枝限定法

分枝限定法 (branch and bound) (アルゴリズム E.3) として知られる一般的な手法では，期待報酬の上限と下限に関するドメイン情報を使用することで，計算を大幅に削減できる．状態 s から行動 a を実行した場合のリターンの上限は $\overline{Q}(s,a)$ である．状態 s からのリターンの下限は $\underline{U}(s)$ である．分枝限定法は深さ優先探索と同じ手順に従うが，上限に従って行動上の反復を行い，返される可能性のある最良値が以前の行動に従うことで，すでに見つけられた値よりも高い場合にのみ後続ノードに進む．分枝限定探索は例 E.1 において，前方探索と比較される．

```
function branch_and_bound(𝒫::Search, s, d, Ulo, Qhi)
    𝒜, T, R = 𝒫.𝒜(s), 𝒫.T, 𝒫.R
    if isempty(𝒜) || d ≤ 0
        return (a=nothing, u=Ulo(s))
    end
    best = (a=nothing, u=-Inf)
    for a in sort(𝒜, by=a→Qhi(s,a), rev=true)
        if Qhi(s,a) ≤ best.u
            return best # safe to prune
        end
        u = R(s,a) + branch_and_bound(𝒫,T(s,a),d-1,Ulo,Qhi).u
        if u > best.u
            best = (a=a, u=u)
        end
    end
    return best
end
```

アルゴリズム E.3　現在の状態 s から離散探索問題 $\mathcal{P}$ に対して近似的最適行動を見つけるための分枝限定探索アルゴリズム．探索は価値関数の下限 Ulo と行動価値関数の上限 Qhi を使用して深さ d まで実行される．返される名前付きタプルは最適行動 a とその有限時間区間期待価値 u で構成される．

六角世界の探索問題で分枝限定法を使用することを考えよう．探索問題の行動は決定論的な遷移を起こし，六角世界のマルコフ決定過程のようではなくて，対

例 E.1　分枝限定法が前方探索に対して得られる省力の比較．適切な上下限を使用すると，分枝限定法の効率が大幅に向上する．

```julia
function heuristic_search(𝒫::Search, s, d, Uhi, U, M)
    if haskey(M, (d,s))
        return M[(d,s)]
    end
    𝒜, T, R = 𝒫.𝒜(s), 𝒫.T, 𝒫.R
    if isempty(𝒜) || d ≤ 0
        best = (a=nothing, u=U(s))
    else
        best = (a=first(𝒜), u=-Inf)
        for a in sort(𝒜, by=a→R(s,a) + Uhi(T(s,a)), rev=true)
            if R(s,a) + Uhi(T(s,a)) ≤ best.u
                break
            end
            s′ = T(s,a)
            u = R(s,a) + heuristic_search(𝒫, s′, d-1, Uhi, U, M).u
            if u > best.u
                best = (a=a, u=u)
            end
        end
    end
    M[(d,s)] = best
    return best
end
```

アルゴリズム E.5　状態 s から開始して最大深さ d まで探索する探索問題 $\mathcal{P}$ を解くためのヒューリスティック探索アルゴリズム．ヒューリスティック Uhi は探索を導くために使用され，近似価値関数 U は終端状態で評価され，深さと状態のタプルを含むディクショナリ形式のトランスポジションテーブル M により，アルゴリズムは以前に探索された状態からの価値をキャッシュできるようになる．

行動は次のように，即時の報酬と将来のリターンのヒューリスティック推定に基づいてソートされる．

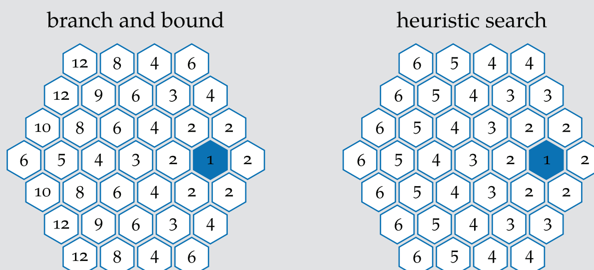
$$R(s,a)+\overline{U}(T(s,a)) \tag{E.1}$$

最適性を保証するために，ヒューリスティックが**許容可能** (admissible) であり，**整合する** (consistent) 必要がある．許容可能なヒューリスティックは真の価値関数の上限である．整合したヒューリスティックは隣接する状態への遷移によって得られる期待報酬を下回ることはない．

$$\overline{U}(s) \geq R(s,a)+\overline{U}(T(s,a)) \tag{E.2}$$

この手法は例 E.2 において，分枝限定法と比較される．

例 E.1 と同じ六角世界探索問題にヒューリスティック探索を適用できる．ヒューリスティック $\overline{U}(s)=5-\delta(s)$ を用いる．ここで，$\delta(s)$ は与えられた状態から終端報酬状態へのステップ数である．ここでは，各開始状態から分枝限定法 (左) またはヒューリスティック探索 (右) を実行したときに訪れた状態の数を示す．分枝限定法はゴール状態の近くおよび左側の状態で同じ程度に効率的であるが，ヒューリスティック探索は任意の初期状態から効率的に探索できる．



例 E.2　ヒューリスティック探索が分枝限定法の探索に対して得られる省力の比較．ヒューリスティック探索は先読みヒューリスティック価値に従って行動を自動的に順序付けする．

F.4　マウンテンカー

マウンテンカー問題 (mountain car problem)[5] では，車両は谷から出て右方向に走行しなければならない．谷の壁は非常に急であるため，不十分な速度でゴールに向けてやみくもに加速すると，車両が停止して滑り落ちてしまう．エージェントは丘に車両を上げるために，最初に左方向に加速し，戻るときに丘に登るのに十分な勢いを得ることを学ばなければならない．

状態は車両の水平方向の位置 $x \in [-1.2, 0.6]$ と速度 $v \in [-0.07, 0.07]$ である．任意の与えられた時間ステップで，車両は左に加速する $(a=-1)$ か，右に加速する $(a=1)$ か，惰性で進む $(a=0)$ かである．ターンごとに -1 の報酬を受け取り，車両が $x=0.6$ を超えた谷の右側に到達すると終了する．問題の視覚化は図 F.7 に示されている．

5)　この問題は以下の文献で導入された．A. Moore, "Efficient Memory-Based Learning for Robot Control," Ph.D. dissertation, University of Cambridge, 1990. 離散行動空間をもつよく知られた単純な形式は次の文献で与えられた．S. P. Singh and R. S. Sutton, "Reinforcement Learning with Replacing Eligibility Traces," *Machine Learning*, vol. 22, pp. 123–158, 1996.

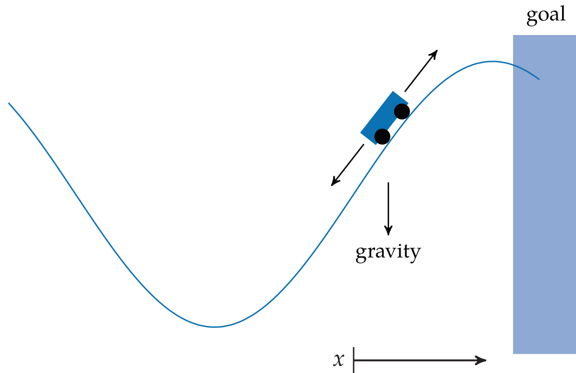


図 F.7　マウンテンカー問題では，車両が丘に登る動力を与えるために，左右の加速を交互に行う必要がある．ゴールの領域は青で示されている．

マウンテンカー問題での遷移は次のように決定論的である．

$$v' \leftarrow v + 0.001a - 0.0025\cos(3x)$$
$$x' \leftarrow x + v'$$

速度更新の重力項はパワー不足の車両を谷底に向かって押し戻すものである．遷移は状態空間の境界に止められる．

マウンテンカー問題は帰還遅延のある問題の良い例である．ゴール状態に到達するには多くの行動が必要となるため，訓練を受けていないエージェントが一貫した単位ペナルティ以外のものを受けることは困難である．最良の学習アルゴリズムは，ゴールに到達する軌跡から状態空間の残りの部分に知識を効率的に伝播させることができる．

F.5　単純レギュレータ

単純レギュレータ問題 (simple regulator problem) は単一状態をもつ単純な線形 2 次レギュレータの問題である．これは単一の実数値状態と単一の実数値行動をもつマルコフ決定過程である．遷移は，後続状態 s' がガウス分布 $\mathcal{N}(s+a, 0.1^2)$ から引き出されるような線形ガウスである．報酬は 2 次関数 $R(s,a) = -s^2$ であり，行動に依存しない．本書の例では，初期状態分布 $\mathcal{N}(0.3, 0.1^2)$ が用いられる．

7.8 節の方法を使用して最適有限時間区間方策を導き出すことはできない．この場合，$\boldsymbol{T}_s = [1]$，$\boldsymbol{T}_a = [1]$，$\boldsymbol{R}_s = [-1]$，$\boldsymbol{R}_a = [0]$ であり，w は $\mathcal{N}(0, 0.1^2)$ から引き出される．リッカチ方程式の適用では，$\boldsymbol{R}_a$ が負定であることを要求するが，これはそうでは

{ 協力, 離脱 } である．図 F.14 の表に個人の報酬が示されている．行はエージェント 1 の行動を示している．列はエージェント 2 の行動を示している．エージェント 1 と 2 の報酬は各セルに $R^1(a^1,a^2)$，$R^2(a^1,a^2)$ として示している．ゲームは一度だけ，あるいは任意の回数繰り返される．無限時間区間の場合，$\gamma = 0.9$ の割引係数を用いる．

agent 1 \ agent 2	cooperate	defect
cooperate	$-1, -1$	$-4, 0$
defect	$0, -4$	$-3, -3$

図 F.14　囚人のジレンマに対応した報酬

F.11　じゃんけん

世界中でよく行われるゲームの 1 つは**じゃんけん** (rock-paper-scissors) である．2 エージェントが，それぞれグー，パー，チョキのいずれかを選択できる．グーがチョキに勝ち，グーをプレイするエージェントは 1 の報酬を得て，チョキをプレイするエージェントは 1 のペナルティを受ける．チョキがパーに勝ち，チョキをプレイするエージェントは 1 の報酬を得て，パーをプレイするエージェントは 1 のペナルティを受ける．最後に，パーがグーに勝ち，パーをプレイするエージェントは 1 の報酬を得て，グーをプレイするエージェントは 1 のペナルティを受ける．

このゲームでは，$\mathcal{I} = \{1,2\}$ であり，$\mathcal{A} = \mathcal{A}^1 \times \mathcal{A}^2$ で，$\mathcal{A}^i = \{$ グー, パー, チョキ $\}$ である．図 F.15 には，このゲームに関する報酬が各セルに $R^1(a^1,a^2)$，$R^2(a^1,a^2)$ として示されている．ゲームは一度，または任意の回数繰り返される．無限時間区間の場合，$\gamma = 0.9$ の割引係数を用いる．

agent 1	rock	paper	scissors
rock	$0, 0$	$-1, 1$	$1, -1$
paper	$1, -1$	$0, 0$	$-1, 1$
scissors	$-1, 1$	$1, -1$	$0, 0$

図 F.15　じゃんけんゲームに対応した報酬

F.12　旅行者のジレンマ

旅行者のジレンマ (traveler's dilemma) は，航空会社が 2 人の旅行者の 2 つの同じスーツケースを紛失してしまうゲームである．航空会社は旅行者にスーツケースの価値を 2 ドルから 100 ドルの範囲で書き留めるように依頼する．両者が同じ価値を示したら，両者ともにその価値を受け取る．そうでない場合，低い価値を示した旅行者は示した価値に 2 ドルを加えた額を得て，高い価値を示した旅行者は低い価値に 2 ドル

を差し引いた額を得る．言い換えれば，報酬関数は次のようになる．

$$R_i(a_i, a_{-i}) = \begin{cases} a_i & \text{if } a_i = a_{-i} \\ a_i + 2 & \text{if } a_i < a_{-i} \\ a_{-i} - 2 & \text{otherwise} \end{cases} \quad \text{(F.11)}$$

多くの人々が 97 ドルから 100 ドルの間の数値を書く傾向にある．しかし，幾分直観に反するが，わずか 2 ドルという唯一のナッシュ均衡が存在する．

F.13　捕食者と被食者の六角世界

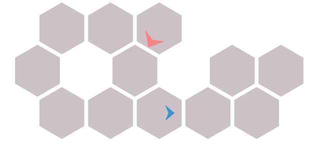


図 F.16　捕食者と被食者の六角世界の初期状態．捕食者は赤で，被食者は青である．矢印は個々のエージェントが彼らの初期セルからとる可能性のある行動を示している．

捕食者と被食者の六角世界問題 (predator–prey hex world problem) は，捕食者と被食者からなる複数のエージェントを含むように六角世界のダイナミクスを拡張している．捕食者はできるだけ早く被食者を捕まえようとし，被食者はできるだけ長く捕食者から逃げようとする．六角世界の初期状態を図 F.16 に示す．このゲームには終端状態はない．

捕食者の集合 $\mathcal{I}_{\text{pred}}$ と被食者の集合 $\mathcal{I}_{\text{prey}}$ があり，$\mathcal{I} = \mathcal{I}_{\text{pred}} \cup \mathcal{I}_{\text{pred}}$ である．状態は各エージェントの位置で，$\mathcal{S} = \mathcal{S}^1 \times \cdots \times \mathcal{S}^{|\mathcal{I}|}$ であり，各 $\mathcal{S}^i$ はすべての六角の位置に等しい．結合行動空間は $\mathcal{A} = \mathcal{A}^1 \times \cdots \times \mathcal{A}^{|\mathcal{I}|}$ であり，各 $\mathcal{A}^i$ は六角形でのすべての 6 つの移動方向からなる．

捕食者 $i \in \mathcal{I}_{\text{pred}}$ と被食者 $j \in \mathcal{I}_{\text{prey}}$ が同じ六角形を共有し，$s_i = s_j$ となるならば，被食者は貪り食われる．被食者 j はランダムに，ある六角セルに移され，子孫が世界に現れたことを表現している．そうでなければ，状態遷移は独立しており，もともとの六角世界で説明されている通りになる．

単一または複数の捕食者と単一または複数の被食者が同じセル内にいる場合，捕食者は被食者を捕まえることができる．n の捕食者と m の被食者がすべて同じセルを共有すれば，捕食者は m/n の報酬を受け取る．たとえば，2 の捕食者が 1 の被食者を一緒に捕まえた場合，それぞれが 1/2 の報酬を受け取る．3 の捕食者が一緒に 5 の被食者を捕まえた場合，それぞれが 5/3 の報酬を受け取る．捕食者が移動すれば，1 のペナルティを受ける．被食者はペナルティなしで移動できるが，捕食されると 100 のペナルティを受ける．

F.14　複数世話人の泣いている赤ちゃん

複数世話人の泣いている赤ちゃん問題 (multicaregiver crying baby problem) は泣いている赤ちゃん問題のマルチエージェントの拡張である．各世話人 $i \in \mathcal{I} = \{1, 2\}$ に対して，状態，行動，観測は次のようになる．

$$\mathcal{S} = \{\text{空腹}, \text{満腹}\} \quad \text{(F.12)}$$

$$\mathcal{A}^i = \{\text{授乳する}, \text{歌う}, \text{無視する}\} \quad \text{(F.13)}$$

$$\mathcal{O}^i = \{\text{泣いている}, \text{静か}\} \quad \text{(F.14)}$$

遷移ダイナミクスは，どちらの世話人も赤ちゃんを満足させるために授乳できることを除いて，元の泣いている赤ちゃん問題と次のように同様である．

$$T(\text{満腹} \mid \text{空腹}, (\text{授乳する}, \star)) = T(\text{満腹} \mid \text{空腹}, (\star, \text{授乳する})) = 100\% \quad \text{(F.15)}$$

ここで，$\star$ は可能なすべての別の変数の割当てを示す．それ以外の場合，行動が授乳

```
Tuple{Int64, Int64, Vector{Int64}, Float64, Float64}
julia> x[2]
0
julia> x[end]
4.6692
julia> x[4:end]
(2.5029, 4.6692)
julia> length(x)
5
julia> x = (1, 2)
(1, 2)
julia> a, b = x;
julia> a
1
julia> b
2
```

G.1.8　名前付きタプル

名前付きタプル (named tuple) はタプルのようであるが，各要素に自身の名前がある．

```
julia> x = (a=1, b=-Inf)
(a = 1, b = -Inf)
julia> x isa NamedTuple
true
julia> x.a
1
julia> a, b = x;
julia> a
1
julia> (; :a=>10)
(a = 10,)
julia> (; :a=>10, :b=>11)
(a = 10, b = 11)
julia> merge(x, (d=3, e=10)) # merge two named tuples
(a = 1, b = -Inf, d = 3, e = 10)
```

G.1.9　ディクショナリ

ディクショナリ (dictionary) はキーと値のペアのコレクションである．キーと値のペアは二重矢印演算子`=>`で示される．配列やタプルと同様に，角括弧を使用してディクショナリにインデックスを付けることができる．

```
julia> x = Dict();        # empty dictionary
julia> x[3] = 4           # associate key 3 with value 4
4
julia> x = Dict(3=>4, 5=>1) # create a dictionary with two key-value pairs
Dict{Int64, Int64} with 2 entries:
 5 => 1
 3 => 4
julia> x[5]               # return the value associated with key 5
1
julia> haskey(x, 3)       # check whether dictionary has key 3
true
julia> haskey(x, 4)       # check whether dictionary has key 4
false
```

```
 Float32
 Float64
julia> subtypes(Float64)       # Float64 does not have any subtypes
Type[]
```

オリジナルの抽象型を定義できる．

```
abstract type C end
abstract type D <: C end # D is an abstract subtype of C
struct E <: D            # E is a composite type that is a subtype of D
    a
end
```

G.1.12 パラメトリック型

Julia はパラメータをとる型である**パラメトリック型** (parametric type) をサポートしている．パラメトリック型のパラメータは中括弧内に指定され，カンマで区切られる．ディクショナリの例でパラメトリック型をすでに見ている．

```
julia> x = Dict(3=>1.4, 1=>5.9)
Dict{Int64, Float64} with 2 entries:
 3 => 1.4
 1 => 5.9
```

ディクショナリに関しては，最初のパラメータはキーの型を指定し，2 番目のパラメータは値の型を指定する．この例では，`Int64` のキーと `Float64` の値があり，型 `Dict{Int64, Float64}`のディクショナリを作成する．Julia は入力に基づいてこれらの型を推測できたが，次のように明示的に指定することもできる．

```
julia> x = Dict{Int64,Float64}(3=>1.4, 1=>5.9);
```

独自のパラメトリック型を定義することもできるが，本書ではそうする必要はない．

G.2 関　数

関数 (function) はタプルとして指定された引数を返される結果に写像する．

G.2.1 名前付き関数

名前付き関数 (named function) を定義する 1 つの方法は，次のように，キーワード `function` を用いることで，その後に関数名と引数名のタプルが続く．

```
function f(x, y)
    return x + y
end
```

さらに，割当て形式で関数をより簡潔に定義できる．

```
julia> f(x, y) = x + y;
julia> f(3, 0.1415)
3.1415
```

G.2.2 無名関数

無名関数 (anonymous function) には名前は与えられないが，名前付き変数に割り当てることはできる．無名関数を定義する 1 つの方法は，次のように矢印演算子を使用することである．