TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

BÁO CÁO TỔNG KẾT

ĐỒ ÁN 1

MÔN: CƠ SỞ TRÍ TUỆ NHÂN TẠO

DECISION MAKING

TP. HCM, ngày 02 tháng 01 năm 2022

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

BÁO CÁO TỔNG KẾT

ĐỒ ÁN 1

MÔN: CƠ SỞ TRÍ TUỆ NHÂN TẠO

DECISION MAKING

Người hướng dẫn:

- Gv. Lê Hoài Bắc
- Gv. Nguyễn Ngọc Đức

Sinh viên thực hiện:

- 19120002, Chu Chí Biên
- 19120089, Lê Ngọc Lâm
- 19120225, Lê Minh Hiển
- 19120390, Trịnh Thị Thùy
- 19120516, Nguyễn Lê Hữu Hoàng

TP. HCM, ngày 02 tháng 01 năm 2022

MỤC LỤC

THÔNG TIN THÀNH VIÊN

MSSV	Họ và tên	Email	Đóng góp
19120002	Chu Chí Biên	19120002@student.hcmus.edu.vn	100%
19120089	Lê Ngọc Lâm	19120089@student.hcmus.edu.vn	100%
19120225	Lê Minh Hiển	19120225@student.hcmus.edu.vn	100%
19120390	Trịnh Thị Thùy	19120390@student.hcmus.edu.vn	100%
19120516	Nguyễn Lê Hữu Hoàng	19120516@student.hcmus.edu.vn	100%

I. HEX WORLD

1. Phát biểu bài toán

Bài toán Hex World là một MDP đơn giản, chúng ta phải đi qua một bản đồ ô để đạt được trạng thái mục tiêu. Mỗi ô trong bản đồ ô biểu thị một trạng thái trong MDP. Chúng ta có thể di chuyển bất kỳ hướng nào trong 6 hướng. Nếu chúng ta va chạm vào đường viền bên ngoài của bản đồ sẽ không di chuyển gì cả và tốn chi phí là 1.0. Có một số ô nhất định trong Hex World là trạng thái cuối. Thực hiện bất kỳ hành động nào trong các ô này sẽ mang lại cho chúng ta phần thưởng cụ thể và sau đó đưa chúng ta đến trạng thái cuối. Không có phần thưởng nào được nhận ở trạng thái cuối. Do đó, tổng số trạng thái của Hex World là số ô + 1, vì có thêm trạng thái cuối không biểu diễn trên bản đồ. Khi đang ở một ô bất kỳ, chúng ta cần phải tìm hướng di chuyển làm sao để tránh va chạm vào đường viền và các ô chứa rewards không mong muốn.

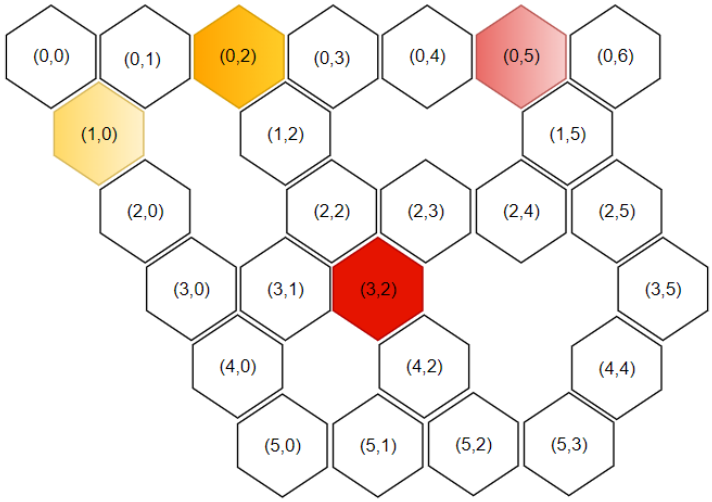
2. Thách thức

Trong bài toán Hex World, mỗi ô có tới 6 hướng di chuyển, nên khó hình dung và tìm hướng đi đúng.

3. Thực nghiệm

Cấu hình:

- DiscountFactor = 0.9
- Xác suất đi theo hướng dự định p_intended = 0.7
- Bản đồ hex world:



3.1. Mô hình hóa tính toán

Để mô hình hoá bài toán, chúng ta sử dụng các struct để lưu trữ dữ liệu

- **Struct DiscreteMDP**
  - T: lưu các bước chuyển đổi từ trạng thái này sang trạng thái khác
  - R: phần thưởng nhận được khi chuyển trạng thái
  - γ: hệ số chiết khấu
- **MDP**
  - γ: hệ số chiết khấu
  - $\mathcal{S}$: không gian trạng thái
  - $\mathcal{A}$: không gian hành động
  - T: lưu các bước chuyển đổi trạng thái
  - R: phần thưởng nhận được khi chuyển trạng thái
  - TR: chuyển đổi mẫu và thưởng
- **HexWorldMDP**
  - hexes: mảng vector lưu trữc các ô của bản đồ
  - mpd: struct DiscreteMDP
  - special_hex_rewards: lưu các vị trí của các reward

3.2. Phương pháp giải quyết

- Policy Iteration

  Policy Iteration là một cách để tính toán một policy tối ưu. Nó liên quan đến việc lặp lại giữa Policy Evaluation và cải tiến policy thông qua greedy policy. Việc lặp lại chính sách được đảm bảo hội tụ với bất kỳ chính sách ban đầu nào. Nó hội tụ trong một số lần lặp lại hữu hạn vì có rất nhiều policy và mỗi lần lặp lại sẽ cải thiện policy nếu có thể được cải thiện.

- Lý do lựa chọn phương pháp này là vì đơn giản, dễ sử dụng

3.3. Mã nguồn

- File policy.jl (nguồn sách tham khảo: Algorithms for Decision Making)

```
struct ValueFunctionPolicy
    𝒫::MDP
    U::Vector{Float64}
end

# Hàm lookahead trả về giá trị nhìn trước với mỗi state, action
```

```
function lookahead(𝒫::MDP, U::Vector, s, a)
    𝒮, T, R, γ = 𝒫.𝒮, 𝒫.T, 𝒫.R, 𝒫.γ
    return R[s,a] + γ*sum(T[s,a,s′]*U[i] for (i,s′) in
enumerate(𝒮))
end

function greedy(𝒫::MDP, U::Vector{Float64}, s)
    u, a = findmax(a -> lookahead(𝒫, U, s, a), 𝒫.𝒜)
    return (a = a, u = u)
end

(π::ValueFunctionPolicy)(s) = greedy(π.𝒫, π.U, s).a

# Hàm iterative_policy_evaluation đánh chính sách π lặp đi lặp
lại
function iterative_policy_evaluation(𝒫::MDP, π, k_max)
    𝒮, T, R, γ = 𝒫.𝒮, 𝒫.T, 𝒫.R, 𝒫.γ
    U = [0.0 for s in 𝒮]
    for k in 1:k_max
        U = [lookahead(𝒫, U, s, π(s)) for s in 𝒮]
    end
    return U
end

struct PolicyIteration
    π # initial policy
    k_max # maximum number of iterations
end

# Khởi tạo PolicyIteration
function initPolicyIteration(π, k_new)
    return PolicyIteration(π, k_new)
end

function solve(M::PolicyIteration, 𝒫::MDP, len::Int64)
    π, 𝒮 = M.π, 𝒫.𝒮

    for k = 1:M.k_max
        println(k)  #lần lặp thứ k
        U = iterative_policy_evaluation(𝒫, π, k_max)
        π′ = ValueFunctionPolicy(𝒫, U)
        for i in 1:len
            print("U$i: $(U[i])  ")              #in ra utility
            println("Direction$i: $(π(i))  ")    #in ra hướng di
chuyển
        end
        if all(π(s) == π′(s) for s in 𝒮)
            println()
            println("Loop: $k")                  #Tổng số lần lặp
            break
```

```
        end
        π = π′
    end
    return [π(s) for s in 𝒮]
end
```

- File hexWorld.jl (Tham khảo github theo link: https://github.com/algorithmsbooks/DecisionMakingProblems.jl/blob/master/src/mdp/hexworld.jl)

```
#Hàm hex_neighbors trả về các xung quanh ô đang xét
function hex_neighbors(hex::Tuple{Int,Int})
    i, j = hex
    [(i, j + 1), (i - 1, j + 1), (i - 1, j), (i, j - 1), (i + 1,
j - 1), (i + 1, j)]
end

struct HexWorldMDP
    # Problem has |hexes| + 1 states, where last state is
consuming.
    hexes::Vector{Tuple{Int,Int}}
    #discreateMDP (each cell)
    mdp::DiscreteMDP
    # The special hex rewards used to construct the MDP
    special_hex_rewards::Dict{Tuple{Int,Int},Float64}

    function HexWorldMDP(
        hexes::Vector{Tuple{Int,Int}},
        r_bump_border::Float64,
        p_intended::Float64,
        special_hex_rewards::Dict{Tuple{Int,Int},Float64},
        γ::Float64,
    )

        nS = length(hexes) + 1          #Số trạng thái
        nA = 6                          #Số hành động
        s_absorbing = nS                #Trạng thái terminal

        T = zeros(Float64, nS, nA, nS)  #init transition
(probability) [0,1]
        R = zeros(Float64, nS, nA)      #action [1,6] because we
have 6 choice to go

        p_veer = (1.0 - p_intended) / 2 # Odds of veering left
or right.

        for s in 1 : length(hexes)
            hex = hexes[s]
```

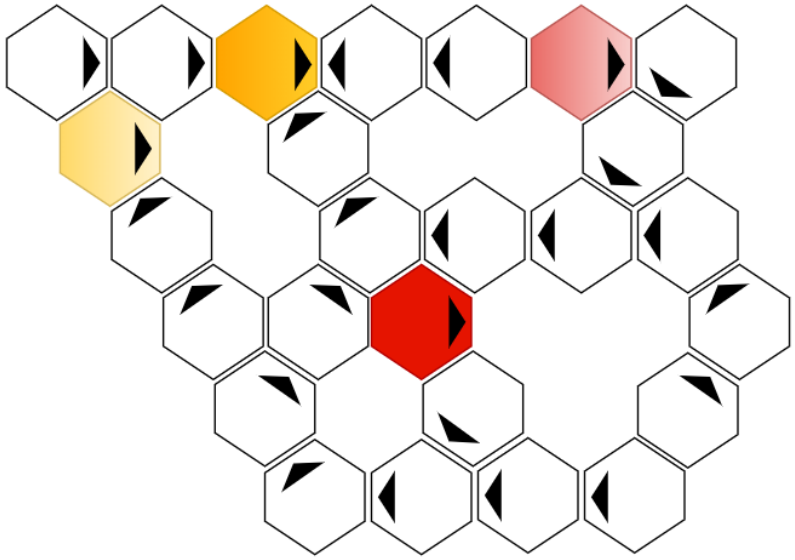
```
            if !haskey(special_hex_rewards, hex)  #kiểm tra các
states đã có key hex chưa
                # Action taken from a normal tile
                neighbors = hex_neighbors(hex)    #trả về mảng 6
vị trí xung quanh
                for (a,neigh) in enumerate(neighbors)
                    # Indended transition.
                    s′ = findfirst(h -> h == neigh, hexes)  #s'
là state kế tiếp
                                                            #fin
dFirst trả về index đầu tiên == neigh trong tập hexes
                    if s′ === nothing
                        # Off the map!
                        s′ = s
                        R[s,a] += r_bump_border*p_intended
                    end
                    T[s,a,s′] += p_intended  #tính tỷ lệ đi đến
state s'

                    # Unintended veer left.
                    a_left = mod1(a+1, nA)
                    neigh_left = neighbors[a_left]
                    s′ = findfirst(h -> h == neigh_left, hexes)
                    if s′ === nothing
                        # Off the map!
                        s′ = s
                        R[s,a] += r_bump_border*p_veer
                    end
                    T[s,a,s′] += p_veer


                    # Unintended veer right.
                    a_right = mod1(a-1, nA)
                    neigh_right = neighbors[a_right]
                    s′ = findfirst(h -> h == neigh_right, hexes)
                    if s′ === nothing
                        # Off the map!
                        s′ = s
                        R[s,a] += r_bump_border*p_veer
                    end
                    T[s,a,s′] += p_veer

                end
            else
                # Action taken from an absorbing hex
                # In absorbing hex, your action automatically
takes you to the absorbing state and you get the reward.
                for a in 1 : nA
                    T[s,a,s_absorbing] = 1.0
                    R[s,a] += special_hex_rewards[hex]
```

```
                end
            end
        end

        # Absorbing state stays where it is and gets no reward.
        for a in 1:nA
            T[s_absorbing, a, s_absorbing] = 1.0
        end

        mdp = DiscreteMDP(T, R, γ)  #Khởi tạo MDPdiscrete

        return new(hexes, mdp, special_hex_rewards)
    end
end

const HexWorldRBumpBorder = -1.0         # Chi phí rời khỏi bản
đồ
const HexWorldPIntended = 0.7            # Xác suất của đi theo
hướng dự định
const HexWorldDiscountFactor = 0.9       # Discount rate (γ)


# Hàm khởi tạo hexWorld
function initHexWorld()
    # Tạo các ô state
    hexes = [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0),
             (0, 1), (3, 1), (5, 1),
             (0, 2), (1, 2), (2, 2), (3, 2), (4, 2), (5, 2),
             (0, 3), (2, 3), (5, 3),
             (0, 4), (2, 4), (4, 4),
             (0, 5), (1, 5), (2, 5), (3, 5),
             (0, 6)]
    # Tạo các vị trí reward
    rewards = Dict{Tuple{Int,Int},Float64}(
        (1, 0) => 5.0,
        (0, 2) => 10.0,
        (3, 2) => -10.0,
        (0, 5) => -5.0,
    )
    return HexWorldMDP(hexes, HexWorldRBumpBorder,
HexWorldPIntended, rewards, HexWorldDiscountFactor)
end

# Hàm khởi tạo MDP
function initMDP(𝒫::HexWorldMDP)
    return MDP(𝒫.mdp.T, 𝒫.mdp.R, 𝒫.mdp.γ)
end
```

3.4. Phân tích

Sau khi giải bài toán với bản đồ hex world ban đầu. Tiến hành thực hiện đánh giá policy với 1.000.000 vòng lặp. Sau khi đánh ta nhận được các kết quả tốt dần lên cho đến khi đạt tối ưu. Đối với cấu hình hiện tại thì Policy Iteration giải tốt khi cho kết quả gần như đúng hoàn toàn.

Kết quả sau khi đạt tối ưu:



(Hình ảnh được vẽ sau khi đọc dữ liệu từ hàm solve in ra màn hình)

4. Tóm tắt kết quả

- **Đánh giá kết quả**
  - Thực hiện tốt khi cho ra kết quả đúng.
- **Điểm mạnh**
  - Hiểu được ý tưởng của Policy Iteration
- **Điểm yếu**
  - Chưa thông thạo ngôn ngữ Julia
  - Chưa nắm được hết tất cả các phương pháp giải bài toán trong sách tham khảo.

II. ROCK-PAPER-SCISSORS

1. Phát biểu bài toán

Rock-Paper-Scissors (hay còn gọi là Oẳn tù tì) là một trò chơi bằng tay mang tính đối nghịch giữa 2 hoặc nhiều người chơi cùng lúc khi ra một trong ba hình dạng của tay gồm Rock (đá – búa), Paper (giấy – bao), Scissors (kéo).

Trò chơi này chỉ có 3 kết quả duy nhất mang tính công bằng: Rock thắng Scissors, thua Paper; Paper thắng Rock, thua Scissors; Scissors thắng Paper, thua Rock. Trong trường hợp các người chơi ra giống nhau thì tính là hòa. Thắng sẽ cộng 1 điểm, thua bị trừ 1 điểm, hòa 0 điểm.

Ở đồ án này, nhóm sẽ nghiên cứu bài toán dựa trên thể thức truyền thống với 2 agent thi đấu đối kháng với nhau. Khi đó, Rock-Paper-Scissors trở thành trò chơi mà tổng điểm đạt được của 2 agent là 0 (zero-sum game). Nội dung trò chơi có thể được minh họa qua mô hình bên dưới.

		agent 2		
		rock	paper	scissors
agent 1	rock	0, 0	−1, 1	1, −1
	paper	1, −1	0, 0	−1, 1
	scissors	−1, 1	1, −1	0, 0

2. Thách thức

Trong thực tế, trò chơi Rock-Paper-Scissors được mọi người chơi theo kiểu lựa chọn ngẫu nhiên giữa rock, paper và scissors và phụ thuộc hoàn toàn vào sự may mắn của người chơi. Nếu chỉ sinh ra lựa chọn theo cách ngẫu nhiên thì xác suất giành chiến thắng sẽ chỉ ở mức thấp. Vì vậy, ở đồ án này, nhóm sẽ đưa ra phương pháp giúp nâng cao tỉ lệ chiến thắng của AI. Nhóm tiếp cận bài toán bằng phương pháp Fictitious Play. Ficitious Play là phương pháp đưa ra quyết định dựa trên lịch sử hành động của các agents đang chơi. Vì vậy, các thực hiện bài toán gặp những khó khăn sau đây:

- Tốn không gian lưu trữ dữ liệu vì sau mỗi lần chơi phải lưu trữ lại kết quả của lần chơi đó để làm cơ sở dữ liệu tính toán cho lần chơi sau.
- Tốn chi phí để tính toán lại xác suất của từng hành động trên cơ sở dữ liệu sau mỗi lần chơi.

3. Thực nghiệm

Ở đồ án này, nhóm sẽ nghiên cứu dựa trên hình thức thi đấu đối kháng giữa 2 agent. Vì vậy, cấu hình bài toán sẽ được thiết kế để phù hợp cho 2 người chơi. Ở mỗi lượt, 2 agent sẽ cùng lúc đưa ra một trong ba lựa chọn rock, paper, scissors. Vì vậy, joint action space của bài toán sẽ là các cặp lựa chọn như: rock – rock, scissors – rock, paper – scissors, … Tổng cộng sẽ có 9 trường hợp có thể xảy ra. Mỗi trận thắng, agent sẽ được 1 điểm, thua mất 1 điểm, hòa không có điểm.

3.1. Mô hình hóa tính toán

Rock-Paper-Scissors được cài đặt tương tự như một SimpleGame được thiết kế trong cuốn sách *Algorithm for decision making* gồm 2 agent, không gian hành động chung (joint action space) và hàm điểm thưởng (reward).

3.2. Phương pháp giải quyết

Với ý tưởng ghi nhớ lại những hành động đối thủ đã ra để có thể tính xác suất và đưa ra hành động hợp lý để đánh bại đối thủ, phương pháp Fictitious Play sẽ ghi lại những hành động đối thủ ra ở mỗi lượt chơi, tính tần suất xuất hiện của mỗi hành động để có thể dự đoán hành động có thể ra trong lượt chơi kế tiếp, cập nhật lại policy của bản thân và đưa ra hành động hợp lý để đánh bại đối thủ.

3.3. Mã nguồn

- Xây dựng struct SimpleGame và Rock-Paper-Scissors

```
struct SimpleGame
    γ  # discount factor
    𝐼  # agents
    𝒜  # joint action space
    R  # joint reward function
end

struct RockPaperScissors end

function SimpleGame(simpleGame::RockPaperScissors)
    return SimpleGame(
```

```
        0.9,
        vec(collect(1:numAgents(simpleGame))),
        [orderedActions(simpleGame, i) for i in
1:numAgents(simpleGame)],
        (a) -> jointReward(simpleGame, a)
    )
end
```

- Hàm tính toán số điểm sau 1 ván cho agent

```
function Reward(i::Int, a)
    if i == 1
        noti = 2
    else
        noti = 1
    end

    if a[i] == a[noti]
        r = 0.0
    elseif a[i] == :rock && a[noti] == :paper
        r = -1.0
    elseif a[i] == :rock && a[noti] == :scissors
        r = 1.0
    elseif a[i] == :paper && a[noti] == :rock
        r = 1.0
    elseif a[i] == :paper && a[noti] == :scissors
        r = -1.0
    elseif a[i] == :scissors && a[noti] == :rock
        r = -1.0
    elseif a[i] == :scissors && a[noti] == :paper
        r = 1.0
    end

    return r
end
```

- Xây dựng một policy cho SimpleGame

```
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
```

- Hàm đưa ra hành động hợp lý để đánh bại đối thủ

```
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

function bestResponse(𝒫::SimpleGame, π, i)
    U(ai) = Utility(𝒫, Joint(π, SimpleGamePolicy(ai), i), i)
    ai = argmax(U, 𝒫.𝒜[i])
    return SimpleGamePolicy(ai)
end
```

- Xây dựng struct FictitiousPlay

```
mutable struct FictitiousPlay
    𝒫 # simple game
    i # agent index
    N # array of action count dictionaries
    πi # current policy
end

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 = bestResponse(𝒫, π, i)
end
```

- Hàm mô phỏng 2 agent đấu với nhau

```
function Simulate(𝒫::SimpleGame, k_max, agents)
    res = [0, 0, 0]
    for k = 1:k_max
        a = [πi() for πi in agents]
```

```
        r = Reward(1, a)
        if (r == 1.0)
            res[1] += 1
        elseif (r == 0)
            res[2] += 1
        else
            res[3] += 1
        end

        update!(agents[1], a)
    end
    return (agents, res)
end
```

3.4. Phân tích

- *struct SimpleGame:* Tạo ra một Simple Game.

  Input: một struct rỗng kí hiệu game RockPaperScissors.

  Output : 1 object SimpleGame.

- *function Reward(i::Int, a):* Tính toán số điểm sau mỗi vòng đấu của mỗi agent, dựa trên luật chơi Rock-Paper-Scissors.

  Input: Số nguyên I phân biệt agent 1 với 2, vector chứa hành động của agent.

  Output: Số điểm agent đạt được sau lượt chơi hiện tại.

- *struct SimpleGamePolicy:* Tạo ra một policy cho SimpleGame.

  Input: 1 dictionary các hành động với tỉ lệ xảy ra các hành động đó hoặc một symbol chỉ một hành động duy nhất.

  Output : 1 object policy.

- *function bestResponse($\mathcal{P}$::SimpleGame, π, i):* Đưa ra policy có điểm utility cao nhất để đánh bại đối thủ với các SimpleGamePolicy trong policies. Policy này chỉ ra duy nhất một hành động đó

  Input: 1 object SimpleGame, 1 object FictitiousPlay và một số nguyên i phân biệt 2 agent với nhau.

  Output : 1 object SimpleGamePolicy có tỉ lệ chiến thắng cao nhất.

- *mutable struct FictitiousPlay:* Tạo ra 1 AI sử dụng phương pháp Fictitious Play.

  Input: 1 object SimpleGame.

Output : 1 object FictitiousPlay.

- *function update!(πi::FictitiousPlay, a):* Cập nhật lại mảng lịch sử hành động và gọi hàm best_response dựa trên lịch sử mới để cập nhật lại policy hiện tại của AI FictitiousPlay.

  Input: 1 object FictitiousPlay, 1 vector hành động của các agents

  Output: không có.

- function Simulate($\mathcal{P}$::SimpleGame, k_max, agents): Hàm mô phỏng agent AI và agent ngẫu nhiên chơi với nhau.

  Input: 1 object SimpleGame, 1 số tự nhiên k_max chỉ số lần chơi tối đa, một mảng chứa agent AI và agent ngẫu nhiên.

  Output : Touple chứa 1 mảng chứa 2 agent và kết quả thắng thua của agent AI.

4. Tóm tắt kết quả

Tỉ lệ thắng của agent AI trên 1,000,000 lần đấu với agent ngẫu nhiên được thể hiện ở bảng sau :

Tỉ lệ ra lựa chọn của agent ngẫu nhiên	Thắng	Hòa	Thua	Tỉ lệ thắng	Tỉ lệ hòa
100% Rock	999,999	0	1	99,99%	0%
100% Paper	999,999	0	1	99,99%	0%
100% Scissors	999,999	1	0	99,99%	0%
80% Rock, 20% Paper	800,883	199,116	1	80,09%	19,91%
60% Scissors, 40% Paper	399,699	600,230	71	39,97%	60,02%
20% Scissors, 30% Rock, 50% Paper	499,205	200,791	300,004	49,92%	20,07%

- **Đánh giá kết quả**
  - Thực hiện tốt khi cho ra kết quả đúng.
- **Điểm mạnh**
  - Hiểu được ý tưởng của Fictitious Play.
- **Điểm yếu**
  - Chưa thông thạo ngôn ngữ Julia
  - Chưa nắm được hết tất cả các phương pháp giải bài toán trong sách tham khảo.

III. TRAVELER'S DILEMMA

1. Phát biểu bài toán

Traveler's Dilemma là một trò chơi yêu cầu hai người chơi đưa ra 1 con số trong đoạn [2, 100]. Nếu hai hành động được đưa ra có cùng giá trị thì cả hai người chơi nhận được cùng mức thưởng có cùng giá trị với hành động được đưa ra. Ngược lại nếu khác nhau thì cả hai nhận được mức thưởng có giá trị của hành động có giá thấp hơn, đồng thời, người đã đưa ra hành động cao hơn sẽ bị phạt một khoản BONUS, người còn lại được thưởng thêm một khoản BONUS.

Ta nhận thấy để đạt được mức thưởng cao nhất thì cần phải đưa ra 1 con số lớn nhất có thể nhưng phải nhỏ hơn của đối phương. Điều này dẫn tới việc người chơi A nghĩ là người chơi B sẽ ra hành động có giá trị là P, nên người chơi A sẽ ra P-1, B cũng nghĩ là A nghĩ như vậy, nên sẽ ra hành động lớn nhất nhưng nhỏ hơn của A là P-2 và cứ thế tiếp diễn đến khi P đạt min (ở trò chơi này là 2). Vì vậy cân bằng Nash của trò chơi này là 2. Rõ ràng, với mục tiêu là không bao giờ thua thì hành động 2 là hợp lí, tuy nhiên nếu chọn 2 thì sẽ không tối ưu về mặt điểm số, và không giống với hành vi tự nhiên của con người (vì con người thường không chơi theo cân bằng Nash và khả năng tính toán có hạn chế, thậm chí nếu có tính được Nash thì cũng không chắc chắn được đối phương có tính được không, dẫn đến không tối ưu được điểm số có thể có).

2. Thách thức

Traveler's Dilemma là một bài toán có joint action space lớn với xấp xỉ 100 actions có thể sử dụng cho mỗi agent. Vì thế nếu chọn state = (action trước của agent1, action trước của agent2) thì sẽ có tổng cộng 100 * 100 trạng thái có thể xảy ra.

Nhóm tiếp cận bài toán với 2 phương pháp: Fictitious Play và Q-Learning:

- Ficitious Play là phương pháp đưa ra quyết định dựa trên lịch sử hành động của các agents đang chơi. Sau mỗi lần chơi thì fictitious play sẽ cập nhật lại policy của mình, vì vậy mà nó sẽ rất tốn kém cả về mặt không gian (khi phải lưu lại lịch sử hành động của các agents) và mặt tính toán (tính lại xác suất của mọi action có thể xảy ra sau mỗi lần ra quyết định).

- Q-Learning là phương pháp học dựa trên q-table. Với cách đặt state như đã nói ở đoạn 1 thì q-table là một bảng có kích thước (10,000x100). Dễ thấy điều này là tốn kém về mặt không gian.

3. Thực nghiệm

Cấu hình của bài Traveler's Dilemma là một trò chơi có 2 người. Ở mỗi lượt người chơi sẽ chọn 1 hành động trong đoạn [2, 100], vì vậy joint action space của trò chơi là các cặp ví dụ như (2, 2), (56, 83), (99, 100), tổng cộng 99*99 = 9801 trường hợp. Ngoài ra điểm thưởng BONUS được mặc định là 2.

3.1. Mô hình hóa tính toán

a) Fictitious Play

Đối với Fictitious play, Traveler's Dilemma được cài đặt tương tự 1 Simple game (thiết kế như trong sách Algorithms for decision making) gồm: số lượng agents, không gian hành động hợp (joint action space), và hàm reward.

Mỗi policy trong game là một dictionary chứa xác suất xảy ra của từng hành động.

b) Q-Learning

Đối với Q-Learning, Traveler's Dilemma được cài đặt như 1 MDP (Markov Decision Processes), lưu giữ lại state hiện tại của bài toán. State được cài đặt như 1 cặp hành động của ván trước đó (action của Agent1, action của Agent2).

3.2. Phương pháp giải quyết

a) Fictious Play

Với ý tưởng đối phương ra hành động gì nhiều thì khả năng cao lượt chơi tiếp theo sẽ ra tiếp hành động đó. Phương pháp Fictitious play áp dụng maximum likelihood estimate để đoán hành động tiếp theo của đối thủ là gì để ra quyết định tối ưu nhất. Ở mỗi lượt chơi agent sẽ ghi lại lịch sử tần suất xuất hiện các hành động của đối phương, qua đó cập nhật lại policy của bản thân sao cho phù hợp. Phương pháp này không đảm bảo sẽ hội tụ về cân bằng Nash.

b) Q-Learning

3.3. Mã nguồn

Phần code được chia thành các file sau đây:

- simple_game (tham khảo sách)
- fictitious_play (tham khảo sách)
- tft (tự cài đặt theo ý tưởng của paper Playing Challenging Iterated Two-Person Games Well: A Case Study on the Iterated Traveler's Dilemma)
- main (tham khảo sách)

- *struct SimpleGame*: tạo ra 1 simple game

  Input: 1 vector chứa số thự tự của các Agent, 1 vector chứa không gian hành động hợp, 1 hàm Reward.

  Output: 1 object SimpleGame.
- *struct SimpleGamePolicy*: tạo ra 1 policy cho simple game.

  Input: 1 dictionary map các hành động với xác suất xảy ra hành động đó, 1 string tên của policy.

  Output: 1 object policy.
- *function (policy_i::SimpleGamePolicy)()*: đưa ra 1 hành động dựa theo xác suất trong bảng dictionary của policy.

  Input: không có.

  Output: 1 action
- *function utility(game::SimpleGame, policies, i):* tính điểm utility của 1 policy hợp trên góc nhìn của agent số thứ i.

  Input: 1 SimpleGame, 1 vector các SimpleGamePolicy, số thứ tự của agent.

  Output: điểm utility cho policy.
- *mutable struct FictitiousPlay:* tạo ra 1 AI FictitiousPlay.

  Input: 1 object SimpleGame, số thứ tự của bản thân, 1 vector gồm các dictionary map hành động với số lần thực hiện hành động đó cho mỗi agents trong simple game, policy đang dùng hiện tại.

  Output: 1 object FictitiousPlay.
- *function best_response(game::SimpleGame, policies, i):* Tìm ra policy có điểm utility cao nhất để đấu với các SimpleGamePolicy trong policies. Policy này chỉ ra duy nhất một hành động đó.

Input: 1 object SimpleGame, 1 vector các SimpleGamePolicy, số thứ tự của bản thân agent

Output: 1 object SimpleGamePolicy được cho là tốt nhất để đấu với các policies khác

- *function update!(policy::FictitiousPlay, a):* Cập nhật lại mảng lịch sử hành động và gọi hàm best_response dựa trên lịch sử mới để cập nhật lại policy hiện tại của AI Fictitous Play.

  Input: 1 object FictitiousPlay, 1 vector hành động của các agents.

  Output: không có.

- *function (policy::FictitiousPlay)() = policy.policy():* Gọi hàm (policy_i::SimpleGamePolicy)() để đưa ra hành động dựa trên dictionary xác suất của policy.

  Input: không có.

  Output: 1 action.

- *mutable struct TFT:* Tạo 1 object thể hiện lối chơi tương tự với Tit-for-tat.

  Input: 1 object SimpleGame, 1 vector các actions mà agent có thể thực hiện, 1 object SimpleGamePolicy cho policy hiện tại, 1 số nguyên số thự tự của agent, 1 chuỗi tên của chính sách

  Output: 1 object TFT

- *function update!(policy::TFT, a):* Cập nhật lại policy hiện tại sao cho policy mới sẽ ra hành động nhỏ hơn hành động trước đó của đối thủ 1 khoảng epsilon (random giữa 1 và 2).

  Input: 1 object TFT, 1 vector hành động của các agents.

  Output: không có.

- *function play(game::SimpleGame, policies, k_max):* Cho 2 agents trong vector policies đấu với nhau k_max lượt.

  Input: 1 object SimpleGame, 1 vector các policies, số nguyên số lượt đấu tối đa.

  Output: 1 vector [dictionary map lại số lượng thắng thua, dictionary map số điểm, các policies hiện tại của agents].

- **Luồng chạy:**
  - Tạo ra một mảng các policies đơn giản:

+ random_policy (ra hành động một cách ngẫu nhiên).
+ random_96_100_policy (ra hành động ngẫu nhiên trong đoạn [96, 100])
+ simpleton2_policy (chỉ ra hành động 2)
+ simpleton63_policy (chỉ ra hành động 63)
+ simpleton99_policy (chỉ ra hành động 99)
+ simpleton100_policy (chỉ ra hành động 100)
+ tft (cập nhật lại policy dựa theo hành động trước đó của đối thủ)

– Lần lượt cho các policies trên đấu với AI FP sử dụng hàm play()

3.4. Phân tích

Với mỗi policy thì FP chỉ được tiến hành huấn luyện qua 500 trận (vì khá tốn kém về chi phí thời gian). Nhìn chung FP đã có hiệu quả khi tìm được mức ra hành động như con người (ở ngưỡng 96 97). Tuy nhiên, hiệu quả của FP thì vẫn còn không ổn định khi gặp phải một số trường hợp.

Vs	Thắng	Hòa	Thua	Policy tối ưu nhất
Random policy	16	6	478	97
Random [96, 100]	404	96	0	96
Simpleton 2	0	0	500	96
Simpleton 63	301	0	199	62
Simpleton 99	500	0	0	98
Simpleton 100	500	0	0	99
Tit-for-tat	2	9	489	80
FictitiousPlay	1	499	0	89

- Nhận thấy FP đạt hiệu quả tốt đối với các policies (random ở khoảng cao, simpleton có value cao). FP đã học được cách ra action cao nhất có thể nhưng vẫn nhỏ hơn đối phương, đúng theo ý tưởng của thuật toán.
- Khi đấu với các policies (random, titfortat) thì không đạt được hiệu quả như mong đợi. Đối với random thì có thể dễ hiểu vì policy này ra hành động một cách ngẫu nhiên, không thể đoán được, để đấu với policy này có lẽ chỉ giành được điểm khi ra hành động min (là 2), tuy nhiên bản thân phương pháp FP không chắc không có thể tìm được điểm hội tụ. Còn với tft thì có vẻ cách chơi hoặc cách cài đặt FP và hàm tính utility còn quá đơn giản để đấu với một phong cách chơi được chứng minh là hiệu quả như Tit-for-tat.
- Khi đấu với một AI FP khác thì cả 2 tìm được điểm tối ưu là 89, khi cả hai hòa đến 499 trận.
- Điều đặc biệt đáng chú ý là FP lại thua hai policies simpleton 2, và 63 (chỉ ra đúng 1 loại action). FP cho rằng policy hiệu quả nhất khi đấu với hai policies này là action 96, 97. Sau nhiều lần thử nghiệm thì nhóm nhận ra có một điểm mốc trong đoạn actions [2, 100], nếu ở dưới cái ngưỡng đó thì FP không tìm ra được cách học hiệu quả, ngưỡng đấy nằm ở mức 71, 72. Khi vượt trên ngưỡng này thì FP hoạt động như mong muốn, chỉ ra hành động nhỏ hơn 1 đơn vị sau policy simpleton. Nhóm vẫn chưa tìm được cách giải thích cho hiện tượng này.

4. Tóm tắt kết quả

- **Đánh giá kết quả**
    - Điểm mạnh:
        - Có thể tự chơi với bản thân nó để tìm ra một action có điểm số tốt.
        - Dễ cài đặt
        - Ý tưởng dễ để nắm
    - Điểm yếu:
        - Tốn bộ nhớ, vì nó phải lưu giữ lại tần suất ra mọi hành động của tất cả agents.

- Tốn thời gian, vì sau mỗi lần ra hành động thì nó lại phải cập nhật lại tần suất hành đồng, tính lại xác suất mới, để có thể quyết định được policy mới.
- k_max (số trận chơi) cần phải đủ lớn để nó học được hiệu quả
- Đấu với simpleton thì chỉ hiệu quả khi simpleton ở ngưỡng 70 trở lên. Có thể là do hàm utility hoặc cách cài đặt FP truyền thống còn quá đơn giản.

- **Điểm mạnh**
  - Hiểu được ý tưởng của FP
  - Đề bài toán dễ nắm bắt
- **Điểm yếu**
  - Julia còn mới lạ, khó nắm bắt, cộng đồng ít người hỗ trợ hơn các ngôn ngữ khác khi có thắc mắc, lỗi
  - Chưa áp dụng được các phương pháp khác, hay cải thiện FP
  - Vẫn có những trường hợp mà FP không hiệu quả mà nhóm chưa giải thích được

IV. PREDATOR-PREY HEX WORLD

1. Phát biểu bài toán

Predator-Prey Hex World là bài toán mở rộng từ Hex World ở trên, với các agent là các thú săn (predator) và các con mồi (prey). Mục tiêu của các predator là bắt được prey nhanh nhất có thể, mục tiêu của prey là chạy trốn predator lâu nhất có thể.

Tại mỗi bước, từng agent sẽ đứng yên hoặc di chuyển đến 1 trong 6 ô hex kề cạnh. Một predator sẽ ăn được một prey nếu chúng đi vào một ô tại cùng thời điểm, khi đó:

- Prey bị ăn thịt sẽ hồi sinh ngẫu ở một ô ngẫu nhiên (ô không chứa predator nào).
- Prey bị ăn thịt bị trừ 100 điểm.
- Predator sẽ được cộng số điểm được tính như sau: $20 * \frac{M}{N}$ với M, N lần lượt là số lượng predator và prey tại ô đang xét.

Ngoài ra, điểm thưởng còn có thêm:

- Nếu predator rút ngắn được khoảng cách tới prey gần nhất, sẽ được cộng 5 điểm (trừ 5 trong trường hợp ngược lại).
- Nếu prey nới rộng được khoảng cách tới predator gần nhất, sẽ được cộng 5 điểm (trừ 5 trong trường hợp ngược lại).

2. Thách thức

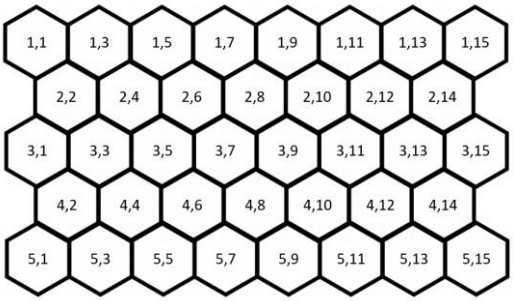
Tài liệu tham khảo thêm khá ít.

3. Thực nghiệm

3.1. Mô hình hóa tính toán

- **Hex world**:

  Là tập hợp các ô hex nằm liên tiếp tạo thành một bảng được đánh số như hình dưới. Giới hạn của chỉ số dòng và chỉ số cột được cho trước.



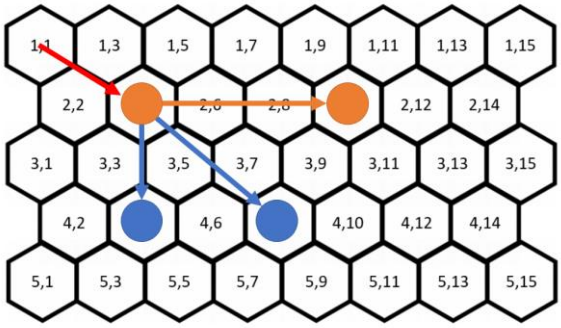
- **Agent:**

  Là tập hợp các agent là tập hợp toàn bộ các predator và prey: $I = I_{predator} \cup I_{prey}$.

- **State:**

  State của một agent được biểu diễn bởi một tập hợp các vector 2 chiều, trong đó:

  - Có $|I_{predator}|$ vector biểu diễn hướng từ agent tới các predator.
  - Có $|I_{prey}|$ vector biểu diễn hướng từ agent tới các prey.
  - Một vector biểu diễn vị trí của agent.



Ở đây, các ô hex là các lục giác đều có cạnh bằng 1 đơn vị.

Như hình trên, bài toán có 2 predator và 2 prey và các vector là state của predator (cam) tại ô (2,2). Hiển nhiên, mỗi predator cần quan tâm đến vị trí các prey, ngoài ra các predator cũng cần quan tâm đến vị trí của nhau (tránh việc tranh con mồi). Tương tự, mỗi prey quan tâm đến vị trí các predator và cả các prey khác để (tránh tập trung đông prey).

- **Action:**

  Mỗi action là một cặp số chỉ độ dời (về chỉ số hàng cột) của ô tiếp theo với ô hiện tại. Có 7 action khác nhau:

  $(0, 0), (-1, -1), (-1, 1), (1, -1), (1, 1), (0, -2), (0, 2)$

3.2. Phương pháp giải quyết

Dựa theo thuật toán Nash Q-learning.

a) Tính Q-value

Giả sử ta có một Hex World với k ô và tổng số predator, prey là $|I|$, do các agent có thể đứng trùng vị trí nên kích thước State Space cho mỗi agent là $k^{|I|}$. Từ đó có thể dẫn đến kích thước của State Space trở nên rất lớn.

Khi số lượng state lớn như vậy, việc lưu Q-value của tất cả các cặp {state, action} trong bộ nhớ là không hiệu quả. Do đó, ta dùng một neural network để ước lượng Q-value cho mỗi cặp {state, action}.

Đầu tiên, cần chuẩn hóa dữ liệu để có thể đưa vào neural network. Ta đã có state là tập hợp gồm $|I| + 1$ vector có ý nghĩa. Hiện tại, action là 2 số nguyên chỉ độ dời hàng và cột, ta cũng biến nó thành một vector.

Kết hợp $|I| + 2$ vector và tạo thành mảng một chiều X chứa $2|I| + 4$ số thực.

Cho mảng một chiều đã tạo ở trên vào neural network để ước lượng giá trị Q-value.

b) Huấn luyện mô hình

- Đầu tiên, khởi tạo hai mô hình neural network cho các predator và các prey.
- Thực hiện 30000 vòng lặp, ở mỗi vòng lặp:
  - Khởi tạo một joint state ngẫu nhiên
  - Khám phá tiếp từ joint state này trong 100 bước. Tại mỗi bước:
    + Tính state riêng của từng agent và dựa vào model tương ứng để ước lượng Q-value cho từng action. Từ đó, chọn ra action cho từng agent.
    + Khi đã có được state và action, thực hiện transition với joint state tiếp theo. Đồng thời tính được reward cho từng agent.
    + Tính joint utility theo công thức: $\boldsymbol{U}(s') = \sum_{a'} \boldsymbol{Q}(s', a') \prod_j \pi^{j'}\left(a^{j'}\right)$.
    + Với mỗi agent, dựa vào reward riêng và utility riêng để tối ưu trọng số θ của mô hình dựa theo hàm loss Mean Absolute Error:

$$l(\theta) = |Q^*(s, a) - Q_\theta(s, a)|$$

với $Q^*(s, a) = \text{reward} + \gamma * \text{utility}$ (chọn $\gamma = 0.9$)

3.3. Mã nguồn

- **Các hàm phụ trợ**
  - Khởi tạo neural network

```
# Source file: model.jl
using Flux # Thư viện deep learning

struct Model
    nn::Chain # Neural network
    opt::ADAM # Optimizer
end
```

```
function get_model(cf) # Biến cf chứa các thông số chung
    layers = []
    N = length(cf.hidden_layers)
    push!(layers, Dense(2*(cf.num_predators + cf.num_preys + 2), cf.hidden_layers[1], σ))
    for i in 1 : N - 1
        push!(layers, Dense(cf.hidden_layers[i], cf.hidden_layers[i+1], σ))
    end
    push!(layers, Dense(cf.hidden_layers[N], 1))
    nn = Chain(layers...) # Gồm các lớp Dense và activation sigmoid
    opt = ADAM(cf.η, cf.β) # Dùng Adam optimizer để tối ưu mô hình
    return Model(nn, opt)
end
```

- Tối ưu mô hình trong một bước lặp

```
# Source file: model.jl
function train_step(model, state, a, y_true) # y_true = reward + gamma * utility
    x = generate_input(state, a) # Tạo input từ state và action
    y_true = [y_true]
    parameters = params(model.nn) # Weights của model.nn

    function mae_loss(x, y_true) # Mean Absolute Error
        y_pred = model.nn(x)
        return mae(y_pred, y_true) # Dùng hàm có sẵn của thư viện Flux
    end
    loss = 0
    grads = gradient(parameters) do
        loss = mae_loss(x, y_true) # Tính loss
    end
    update!(model.opt, parameters, grads) # Thực hiện hồi quy để cập nhật weights của model
    return loss
end
```

- Tính Q-value

```
# Source file: model.jl
function Q(model, state, a)
    x = generate_input(state, a) # Tạo input từ state và action
    return Array(model.nn(x))[1] # Output của model gồm một số thực chính là Q-value
end
```

- Chọn hành động cho một agent

```
# Source file: markov_game.jl

function softmax_selection(model, state)
    a = get_all_actions()
    q_values = [Q(model, state, a[i]) for i in 1:7] # Tính Q-value của tất cả hành động với
state
    probs = softmax(q_values) # Tính tỉ lệ chọn từng hành động, dùng hàm softmax
    idx = sample(1:7, Weights(probs)) # Chọn ngẫu nhiên một hành động với xác suất như trên
    return a[idx]
end

function random_selection(model::Model, state::Matrix{Float64})
    return sample(get_all_actions()) # Chọn ngẫu nhiên một hành động
end
```

- **Huấn luyện mô hình**

(Đã lược bỏ các dòng log thông tin)

```
function train(cf::Config)
    # Khởi tạo 2 model khác nhau cho predator và prey.
    # Model có chứa một neural network để dự đoán Q-values từ cặp state-action truyền vào
```

```
    model_predator = get_model(cf)
    model_prey = get_model(cf)

    for iter in 1 : cf.num_iterations # 30000 vòng lặp
        # Mỗi vòng lặp, tạo một joint state ngẫu nhiên và đi tiếp 100 bước
        s_join = get_random_state(cf)
        for step in 1 : cf.num_steps
            # Chuyển joint state thành các state của từng agent
            s_predators = [get_inv_status(s_join, p) for p in s_join.predators]
            s_preys = [get_inv_status(s_join, p) for p in s_join.preys]

            # Mỗi agent chọn action
            a_predators = [softmax_selection(model_predator, s) for s in s_predators]
            a_preys = [softmax_selection(model_prey, s) for s in s_preys]

            # Từ joint state hiện tại và action của từng agent,
            # tính được joint state tiếp theo và các reward
            s_join_next, rw_predators, rw_preys, _, _ = forward(s, a_predators, a_preys)

            # Tính các giá trị "utility" của từng agent
            u_predators = get_utility(s_join_next, model_predator, s_join_next.predators)
            u_preys = get_utility(s_join_next, model_prey, s_join_next.preys)



            # Dựa vào reward thực tế, tối ưu hóa tham số các model
            # dựa theo công thức: Q(s, a) = (1 - α)*Q(s, a) + α*(reward + γ*utility)
            γ = 0.9
            for (s, a, rw, u) in zip(s_predators, a_predators, rw_predators, u_predators)
               train_step(model_predator, s, a, rw + γ*u) # y_true = rw + γ*u
            end
            for (s, a, rw, u) in zip(s_preys, a_preys, rw_preys, u_preys)
               train_step(model_prey, s, a, rw + γ*u) # y_true = rw + γ*u
            end

            # Chuyển đến state tiếp theo
            s_join = s_join_next
        end # Kết thúc một vòng lặp
    end # Kết thúc huấn luyện
end
```

3.4. Phân tích

Sau khi huấn luyện 2 mô hình predator và prey. Tiến hành mô phỏng để các agent đối kháng với nhau và với các agent random trong 1.000.000 vòng lặp. Nhìn chung các agent đã huấn luyện cho kết quả tốt hơn so với agent random. Tuy nhiên, hiệu quả của các agent được huấn luyện vẫn chưa cao, đặc biệt là các prey.

a) Thí nghiệm 1

Giới hạn chỉ số dòng	5
Giới hạn chỉ số cột	10
Số predator	1
Số prey	1

Trung bình số lần ăn thịt của các predator

	Trained preys	Random preys
Trained predators	74 085	97 683

Random predators	10 430	24 740

b) Thí nghiệm 2

Giới hạn chỉ số dòng	5
Giới hạn chỉ số cột	10
Số predator	3
Số prey	1

Trung bình số lần ăn thịt của các predator

	Trained preys	Random preys
Trained predators	77 830	79 258
Random predators	23 016	25 441

c) Thí nghiệm 3

Giới hạn chỉ số dòng	5
Giới hạn chỉ số cột	10
Số predator	1
Số prey	3

Trung bình số lần ăn thịt của các predator

	Trained preys	Random preys
Trained predators	209 219	241 310
Random predators	25 041	71 481

4. Tóm tắt kết quả

- **Đánh giá kết quả**
  - Vẫn thực hiện được khi kích thước State Space lớn hoặc khi số lượng agent lớn.
  - Tránh được sự rời rạc của các state. Cụ thể, hai state gần giống nhau sẽ được biểu diễn bởi hai tập vector gần giống nhau. Điều này khiến cho Q-value của chúng với cùng một action sẽ gần giống nhau.
  - Khó chuẩn hóa dữ liệu để huấn luyện neural network, cách làm hiện tại chưa hiệu quả.
  - Chưa tìm ra được cấu trúc neural network để đạt hiệu quả.
  - Đòi hỏi nhiều thời gian huấn luyện. Khi kích thước State Space càng lớn, cần càng nhiều vòng lặp để khám phá.
- **Điểm mạnh**
  - Hiểu được ý tưởng của Q-learning

- Đã có kiến thức về neural network

- **Điểm yếu**
  - Chưa quen thuộc với ngôn ngữ Julia
  - Chưa nắm bắt hết được các phương pháp

V. MULTI-CAREGIVER CRYING BABY

1. Phát biểu bài toán

Bài toán Crying Baby là một POMPD (Partially observable Markov decision) đơn giản với hai states (hungry, sated), ba actions (feed, sing, ignore) và hai observation (crying, quiet). Các hành động có tác dụng lên các trạng thái của đứa trẻ:

- Đứa trẻ chuyển sang trạng thái no khi được cho ăn.
- Làm lơ khiến em bé đang no có khả năng chuyển sang trạng thái đói. Nếu em bé đang đói thì làm lơ không làm chuyển sang trạng thái no.
- Hành động hát có tác dụng xác định thông tin tương tự như làm lơ, tuy nhiên không làm trẻ khóc khi đang no, nhưng làm tăng khả năng khóc khi trẻ đang đói.

Tại mỗi thời điểm, agent quan sát trạng thái no hay đói của đứa bé bằng quan sát đứa bé đang khóc hay im lặng. Từ đó chăm sóc đứa trẻ bằng cách chọn một trong các actions: cho đứa trẻ ăn, hát hoặc làm lơ đứa trẻ. Đứa trẻ trở nên đói theo thời gian.

Multi-caregiver Crying Baby là một bài toán được mở rộng từ bài toán Crying baby nhưng với nhiều agent, trong phần trình bày này, chúng tôi sử dụng hai agents cho phần thực nghiệm bài toán.

2. Thách thức

- Mỗi hành động tạo ra sự chuyển đổi khác nhau giữa các trạng thái, ngoài ra giữa các trạng thái có sự thay đổi về xác suất theo thời gian gây khó tiếp cận.
- Bài toán không có nhiều nguồn tài liệu tham khảo, không có nhiều giải pháp có thể sử dụng(các giải pháp hiện có thể tìm thấy: Nash Equilibrium và Dynamic Programing)

3. Thực nghiệm

- Cấu hình bài toán thực nghiệm:

- Số lượng agents: 2
- Bài toán khởi đầu với xác suất xảy ra của hai trạng thái no và đói là [0.5,0.5]
- r_hungry=-10.0: trừ 10 khi thực hiện hành động nhưng đứa trẻ vẫn đói.
- r_feed=-5.0: trừ 5 nếu cho ăn khi đứa trẻ đang no
- r_sing=-0.5: trừ 0.5 nếu hát khi đứa trẻ đang no
- p_hungry=0.1: khả năng đứa trẻ chuyển sang đói
- p_cry_hungry=0.8: khả năng quan sát thấy đứa trẻ khóc khi đói
- p_cry_sing_hungry=0.9: khi hát, quan sát thấy đứa trẻ khóc nếu đói
- p_cry_sated=0.1: khả năng đứa trẻ khóc khi no γ =0.9:

3.1. Mô hình hóa tính toán

```
struct CryingBaby
    r_hungry::Float64 = -10.0
    r_feed::Float64 = -5.0
    r_sing::Float64 = -0.5
    p_become_hungry::Float64 = 0.1
    p_cry_when_hungry::Float64 = 0.8
    p_cry_when_not_hungry::Float64 = 0.1
    p_cry_when_hungry_in_sing::Float64 = 0.9
    γ::Float64 = 0.9
end
```

Bài toán được biểu diễn bằng một POMG (Partially Observation Markov Game):

```
struct POMG
    γ  # yếu tố discount
    ℐ  # agents
    𝒮  # không gian trạng thái
    𝒜  # không gian hành động chung
    𝒪  # không gian quan sát chung
    T  # hàm chuyển đổi trạng thái
```

```
    O  # hàm quan sát chung
    R  # hàm phần thưởng chung
end
```

3.2. Phương pháp giải quyết

Một số phương pháp có thể sử dụng để giải quyết bài toán này là: Nash Equilibrium và Dynamic Programming

a. Nash Equilibrium

Cũng như các trò chơi đơn giản và trò chơi Markov, một điểm là cân bằng Nash cho POMG là khi các agent đi theo một chính sách cho phản hồi tốt nhất và không có sự thay đổi về chính sách. Nó liệt kê tất cả Conditional Plans với các bước có thể để xây dựng một Simple Game. Cân bằng Nash cho Simple Game cũng là cân bằng Nash cho POMG. Tuy nhiên, Nash cho POMG có xu hướng cục kì khó giải quyết về mặt tính toán do sự phản hồi của actions tạo ra khả năng mở rộng về chiều sâu của Conditional Plans.

Việc liệt kê tất cả Conditional Plans tạo ra

b. Dynamic Programming

Dynamic Programming bắt đầu bằng việc tạo ra tất cả các kế hoạch một bước rồi loại bỏ các kế hoạch kém tối ưu sau đó kết hợp các kế hoạch một bước, tạo ra một sự mở rộng các kế hoạch hai bước. Quá trình mở rộng và loại bỏ lặp lại cho đến khi chạm horizon.

Một kế hoạch của một agent bị loại bỏ khi tồn tại ít nhất một kế hoạch tốt hơn hoặc bằng kế hoạch đó.

Dynamic Programming thực hiện chiến lược việc mở rộng chiều sâu của Conditional plans cùng với cắt bỏ các plans không tối ưu giúp tạo ra các tiết kiệm quan trọng ngay cả ở trường hợp xấu nhất là cây Conditional plans mở rộng đầy đủ.

3.3. Mã nguồn

3.4. Phân tích

- *function MultiCaregiverCryingBaby()*

– Khởi tạo agent BabyBOMG

– Input: Không
– Ouput: BabyBOMG

```
function MultiCaregiverCryingBaby()
    BabyPOMDP = CryingBaby()
    return BabyPOMG(BabyPOMDP)
end
```

- *function transition(pomg::BabyPOMG, s, a, s')*
  - Function cho biết tại thời điểm thực hiện hành động a, trạng thái s có chuyển thành trạng thái s' hay không
  - Input: BabyPOMG, state ban đầu, action, state sau khi thực hiện action
  - Ouput: 0 nếu s chuyển thành s' sau khi thực hiện a, 1 nếu s không chuyển thành s sau khi thực hiện a

```
function transition(pomg::BabyPOMG, s, a, s')
    # Regardless, feeding makes the baby sated.
    if a[1] == FEED || a[2] == FEED
        if s' == SATED
            return 1.0
        else
            return 0.0
        end
    else
        # If neither caretaker fed, then one of two things happens.
        # First, a baby that is hungry remains hungry.
        if s == HUNGRY
            if s' == HUNGRY
                return 1.0
            else
                return 0.0
            end
        # Otherwise, it becomes hungry with a fixed probability.
        else
            probBecomeHungry = 0.5 #pomg.babyPOMDP.p_become_hungry
            if s' == SATED
                return 1.0 - probBecomeHungry
            else
                return probBecomeHungry
            end
        end
    end
end
```

- *function joint_observation(pomg::BabyPOMG, a, s', o)*

Input: BabyPOMG, hành động a, trạng thái s’ sau khi thực hiện hành động a, quan sát o

```
function joint_observation(pomg::BabyPOMG, a, s’, o)
    # If at least one caregiver sings, then both observe the result.
    if a[1] == SING || a[2] == SING
        # If the baby is hungry, then the caregivers both observe
crying/silent together.
        if s’ == HUNGRY
            if o[1] == CRYING && o[2] == CRYING
                return pomg.babyPOMDP.p_cry_when_hungry_in_sing
            elseif o[1] == QUIET && o[2] == QUIET
                return 1.0 - pomg.babyPOMDP.p_cry_when_hungry_in_sing
            else
                return 0.0
            end
        # Otherwise the baby is sated, and the baby is silent.
        else
            if o[1] == QUIET && o[2] == QUIET
                return 1.0
            else
                return 0.0
            end
        end
    # Otherwise, the caregivers fed and/or ignored the baby.
    else
        # If the baby is hungry, then there's a probability it cries.
        if s’ == HUNGRY
            if o[1] == CRYING && o[2] == CRYING
                return pomg.babyPOMDP.p_cry_when_hungry
            elseif o[1] == QUIET && o[2] == QUIET
                return 1.0 - pomg.babyPOMDP.p_cry_when_hungry
            else
                return 0.0
            end
        # Similarly when it is sated.
        else
            if o[1] == CRYING && o[2] == CRYING
                return pomg.babyPOMDP.p_cry_when_not_hungry
            elseif o[1] == QUIET && o[2] == QUIET
                return 1.0 - pomg.babyPOMDP.p_cry_when_not_hungry
            else
                return 0.0
            end
        end
    end
end
```

- *function joint_reward(pomg::BabyPOMG, s, a)*
  - Tính reward cho các agent khi thực hiện một hành động nào đó
  - Input: BabyPOMG, trạng thái s, hành động a
  - Ouput: vector chứa reward của hai agents

```
function joint_reward(pomg::BabyPOMG, s, a)
    r = [0.0, 0.0]


    # Both caregivers do not want the child to be hungry.
    if s == HUNGRY
        r += [pomg.babyPOMDP.r_hungry, pomg.babyPOMDP.r_hungry]
    end


    # One caregiver prefers to feed.
    if a[1] == FEED
        r[1] += pomg.babyPOMDP.r_feed / 2.0
    elseif a[1] == SING
        r[1] += pomg.babyPOMDP.r_sing
    end


    # One caregiver prefers to sing.
    if a[2] == FEED
        r[2] += pomg.babyPOMDP.r_feed
    elseif a[2] == SING
        r[2] += pomg.babyPOMDP.r_sing / 2.0
    end


    # Note that caregivers only experience a cost if they do something.


    return r
end
```

- *function POMG(pomg::BabyPOMG)*
  - Input: BabyPOMG
  - Ouput: POMG

```
function POMG(pomg::BabyPOMG)
    return POMG(
        pomg.babyPOMDP.γ,
        vec(collect(1:n_agents(pomg))),
```

```
        ordered_states(pomg),
        [ordered_actions(pomg, i) for i in 1:n_agents(pomg)],
        [ordered_observations(pomg, i) for i in 1:n_agents(pomg)],
        (s, a, s′) -> transition(pomg, s, a, s′),
        (a, s′, o) -> joint_observation(pomg, a, s′, o),
        (s, a) -> joint_reward(pomg, s, a)
    )
end
```

4. Tóm tắt kết quả

- **Đánh giá kết quả**
  - Thực hiện tốt khi cho ra kết quả đúng.
- **Điểm mạnh**
  - Hiểu được ý tưởng để giải bài toán.
- **Điểm yếu**
  - Chưa thông thạo ngôn ngữ Julia
  - Chưa nắm được hết tất cả các phương pháp giải bài toán trong sách tham khảo.