ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO ĐỒ ÁN THỰC HÀNH

CƠ SỞ TRÍ TUỆ NHÂN TẠO

Thành phố Hồ Chí Minh, tháng 1 năm 2022

Contents

A. GIỚI THIỆU

I – Thông tin nhóm

Thành viên:

STT	*MSSV*	*Họ và tên*
1	19120121	Nguyễn Lê Quang
2	19120219	Hà Chí Hào
3	19120384	Nguyễn Trung Thời
4	19120476	Trần Phương Đình
5	19120685	Võ Ngọc Tín

II – Đánh giá chung

Điểm mạnh:

- Nhìn chung, các bài toán đều được hoàn thiện.
- Tiếp cận được bài toán và tìm ra mô hình cùng phương pháp giải quyết phù hợp.
- Các bài toán hầu như trả về kết quả mong muốn.

Điểm yếu

- Có sự bất cập khi dịch các bài toán từ tiếng anh sang tiếng việt, một số từ dịch có thể không rõ nghĩa.
- Tiếp xúc với nhiều khái niệm mới liên quan đến lý thuyết trò chơi, gặp khó khăn trong việc cài đặt các thuật toán và các cấu trúc dữ liệu liên quan.
- Kĩ năng lập trình trong môi trường Julia còn hạn chế, dẫn đến nhiều lỗi phát sinh trong quá trình code.
- Các kết quả trả về của bài toán chưa thật sự là tối ưu nhất.

Đánh giá:

	Tiêu chí đánh giá	**Tự đánh giá**
1	Hiểu và mô tả rõ ràng các bài toán cần giải quyết	100%
2	Sử dụng các giải pháp phù hợp cho từng bài toán	100%
3	Code	90%
4	Báo cáo trình bày tốt, giải thích rõ ràng	100%

B. BÀI TOÁN

I – Hex World

1. Phát biểu bài toán

- Giới thiệu bài toán Hex World:

- Mỗi ô trong bản đồ đại diện cho 1 trạng thái và số trạng thái là hữu hạn
- Từ một trạng thái ta có thể di chuyển theo 6 hướng (hướng theo 6 cạnh của lục giác) để chuyển đổi trạng thái nếu hướng đi đó khả thi (trường hợp đụng tường hoặc không có ô bên cạnh thì không thể di chuyển đến)
- Quá trình chuyển đổi hành động là hoàn toàn ngẫu nhiên
- Tại một số ô nhất định sẽ có phần thưởng riêng đặc biệt được quy định bởi 1 giá trị (có thể âm hoặc dương).

- Bài toán Hex World

- Với tập dữ liệu đầu vào bao gồm:

+ Tập các trạng thái S (state)

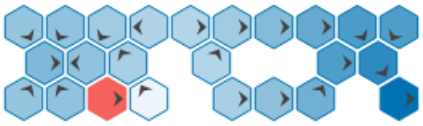
+ Tập các phần thưởng R (reward) tương ứng ở mỗi trạng thái

- Vấn đề cần giải quyết là quá trình quyết định cách mà tác nhân (agent) thực hiện các hành động ở mỗi một trạng thái để nhận được phần thưởng nhiều nhất có thể và tránh các phần thưởng âm trong bản đồ.
- Dữ liệu đầu ra: tập các hành động ứng với mỗi trạng thái

- Minh họa về bài toán

Hình 1

- Hình 1 trên là một bản đồ hex world với 24 trạng thái, 2 phần thưởng dương là 5,10 và 1 phần thưởng âm là -10.



Hình 2

- Hình 2 trên là một bản đồ hex world sau khi chạy xong thuật toán và kết quả trả về sẽ cho ta biết hướng đi của từng trạng thái trong 24 trạng thái ban đầu.

2. Thách thức

- Tập dữ liệu S, R lớn

- Tập dữ liệu S gồm các trạng thái không thể chuyển đổi nhau bằng bất cứ hành động nào (không có 2 ô nào có chung cạnh)

- Thiếu tập kiểm định

3. Thực nghiệm

3.1 MÔ HÌNH TÍNH TOÁN

a. MDP (Markov Decision Process)

Tên	*Biến*	*Mô tả*
Tập trạng thái (State)	S	Tập các trạng thái (vị trí) khả thi của agent
Tập hành động (Action)	A	Tập các hành động (hướng đi) khả thi của 1 agent
Hàm chuyển tiếp (Transition)	T(s'|s, a)	Xác suất chuyển tiếp từ trạng thái s sang trạng thái s' bởi hành động a
Hàm phần thưởng (Reward)	R(s,a)	Phần thưởng có được khi thực hiện hành động a từ trạng thái s

Hệ số chiếc khấu (discounted rate)	$\gamma \in [0, 1]$	Hệ số kiểm soát mức độ quan tâm phần thưởng Nếu $\gamma \to 0$: agent chỉ quan tâm đến phần thưởng trước mắt Nếu $\gamma \to 1$: agent quan tâm đến phần thưởng trong tương lai

b. DiscreteMDP

Tên	*Biến*	*Mô tả*
Hàm chuyển tiếp (Transition)	T(s'|s, a)	xác suất chuyển tiếp từ trạng thái s sang trạng thái s' bởi hành động a
Hàm phần thưởng (Reward)	R(s, a)	Phần thưởng có được khi thực hiện hành động a từ trạng thái s
Hệ số chiếc khấu (discounted rate)	$\gamma \in [0, 1]$	Hệ số kiểm soát mức độ quan tâm phần thưởng

c. HexworldMDP

Tên	*Biến*	*Mô tả*
Tập các tọa độ	hexes	Các vị trí mà agent có thể đứng
DiscreteMDP	mdp	Mô hình DiscreteMDP
Tập các giá trị phần thưởng	special_hex_rewards	Giá trị phần thưởng tại các ô tương ứng có trong tập, còn lại mặc định bằng 0

Sử dụng mô hình Markov Decision Process để giải quyết bài toán hexworld Với tập các Transition là một cấu trúc dữ liệu T (transition) là mảng 3 chiều còn R (rewards) là mảng 2 chiều. Giá trị của T nằm trong khoảng [0,1] còn giá trị R là 1 số thực tùy ý.

3.2 PHƯƠNG PHÁP GIẢI QUYẾT

$\pi(s)$: chính sách đưa ra quyết định hành động tại trạng thái s

U^{π} : hàm giá trị đánh giá ứng với chính sách π

a) Policy evaluation

Chúng ta sẽ đánh giá giá trị của **value function** U^{π} theo 2 cách sau:

Cách 1: Sử dụng vòng lặp

Giá trị U: Với mỗi trạng thái s, hành động a, sử dụng để tính U^{π}

Cách 1: Sử dụng vòng lặp

-Giá trị U : Với mỗi trạng thái s, hành động a

$\bigstar\ n = 1 \rightarrow U_1^{\pi} = R(s, \pi(s))$

$\bigstar n = k + 1 \rightarrow U_{k+1}^{\pi}(s) = R\big(s, \pi(s)\big) + \gamma \sum_{s'} T(s'|s, \pi(s)) U_k^{\pi}(s')$

-Lặp k_max lần để tính ra giá trị của U^{π} : kết quả trả về là 1 mảng các giá trị U^{π} ứng với mỗi hành động tại một trạng thái s tương ứng.

$$U^{\pi}(s) = R\big(s, \pi(s)\big) + \gamma \sum_{s'} T\big(s'\big|s, \pi(s)\big) U^{\pi}(s')$$

Cách 2: Sử dụng ma trận (One-step learning)

Chúng ta có thể tính toán U^{π} bằng cách không sử dụng vòng lặp nhiều lần. Mô hình tính toán được thể hiện như sau:

$$U^{\pi} = R^{\pi} + \gamma T^{\pi} U^{\pi}$$

Trong đó U^{π}, R^{π} lần lượt là hàm giá trị U và hàm phần thưởng được thể hiện bằng một mảng có |S| phần tử. Còn T^{π} là 1 ma trận gồm có |S|x|S| phần tử trong đó T_{ij}^{π} là xác suất khi chuyển từ trạng thái thứ i sang trạng thái thứ j.

Phương trình trên có thể được biến đổi như sau:

$$U^{\pi} - \gamma T^{\pi} U^{\pi} = R^{\pi}$$
$$\Leftrightarrow (1 - \gamma T^{\pi}) U^{\pi} = R^{\pi}$$
$$\Leftrightarrow U^{\pi} = (1 - \gamma T^{\pi})^{-1} R^{\pi}\ (*)$$

Từ phương trình (*) trên ta có thể tính U^{π} dựa trên T,R, γ chỉ 1 lần nhưng độ phức tạp thời gian yêu cầu lên đến O(|S|^3)

b) Policy improvement

Chúng ta sử dụng $U^{\pi}(s)$ và policy π^*để tối ưu hóa cực đại chính sách đường đi dựa theo U.

$$\pi^*(s) = \arg\max_{\pi} U^{\pi}(s)$$

c) Optimal policy

Còn một cách khác để tính chính sách đi (policy) là sử dụng action value function hay còn gọi là *Q-function* . Hàm này sẽ trả về 1 giá trị khi đang đứng tại trạng thái s mà đi theo hành động a và vẫn tiếp tục sử dụng thuật toán tham lam với Q.

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s'|s,a)U(s')$$

Từ hàm Q ở trên ta có thể tính value function U(s) như sau:

$$U(s) = \max_{a} Q(s,a)$$

Do đó, hàm policy sẽ được tính như sau:

$$\pi(s) = \arg\max_{a} Q(s,a)$$

Khi lưu trữ Q thì ta cần đến O(|S|x|A|) không gian lưu trữ thay vì O(|S|) không gian lưu trữ như U, nhưng thay vì đó chứng ta sẽ không cần trích xuất T và R để tính được policy

d) Policy Iteration

Sử dụng vòng lặp để được giá trị hội tụ của policy.Sau mỗi vòng lặp sẽ cho ra một giá trị policy mới.Kết quả trả về sẽ là một policy tương ứng cho các trạng thái trong hexworld.

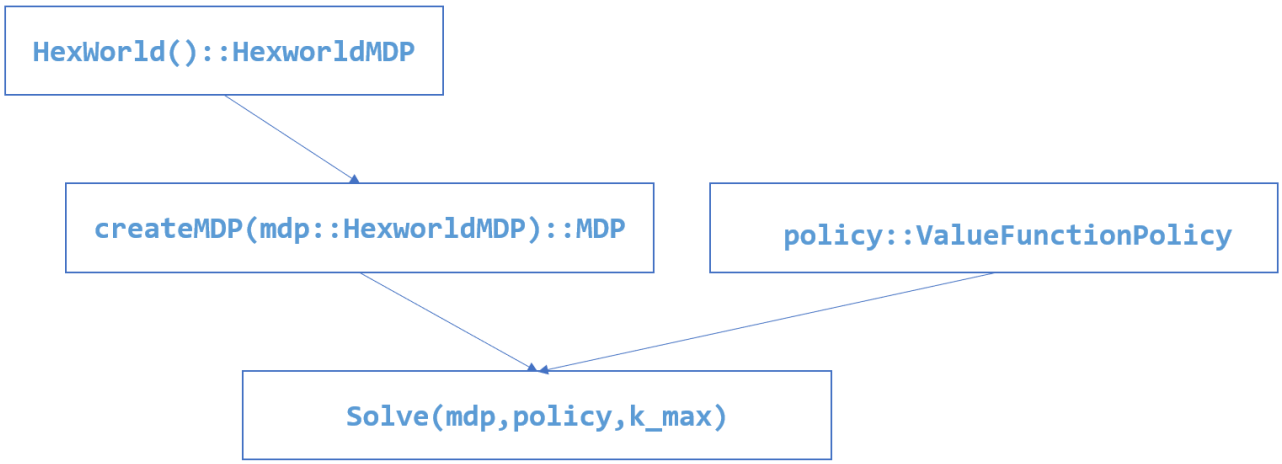
Mã giả:

```
for k = 1:M.k_max
    U = policy_evaluation(𝒫, π)        // tính giá trị value function U
    π′ = ValueFunctionPolicy(𝒫, U)     // tính chính sách
    if all(π(s) == π′(s) for s in 𝒮)   // điều kiện hội tụ
        break
    end
    π = π′
```

```
    end
    return π
```

3.3 MÔ TẢ CODE

a) Flow chính




```
HexWorld()
```

Dùng để khởi tạo ra các giá trị như : trạng thái,phần thưởng,hệ số chiếc khấu,…và trả về 1 kiểu HexWorldMDP đã được định nghĩa từ trước

```
createMDP(mdp::HexWorldMDP)
```

Dùng để ép kiểu 1 HexworldMDP về thành 1 MDP

```
policy::ValueFunctionPolicy:
```

Đây là 1 biến được khởi tạo với constructor gồm 2 tham số là MDP và U (ban đầu sẽ được khởi tạo bằng một mảng các phần tử 0), đây là 1 chính sách giúp cho thuật toán đưa ra hành động tại mỗi trạng thái (có 6 hành động tất cả). policy(s) là hàm trả về chính sách tại trạng thái s.

```
Solve(mdp::MDP,policy,k_max)
```

Đây là hàm xử lý chính để đưa ra kết quả của bài toán, kết quả trả về là 1 mảng các hành động sao cho mỗi hành động tương ứng với một trạng thái trong hexworld.

b) Hàm hỗ trợ

`lookahead(mdp,U,s,a):`

Trả về giá trị là 1 số thực được tính toán dựa trên giá trị reward,transition và U.

`greedy(mdp,U,s):`

Trả về giá trị là 1 tuple gồm 2 giá trị là u(max) và hành động tương ứng với hàm hỗ trợ là lookahead

`policy(s):`

Trả về kết quả là hành động (1 trong 6 hướng đi) tại trạng thái s với hàm hỗ trợ là greedy

`iterative_policy_evaluation(mdp,policy,k_max):`

Kết quả trả về là 1 mảng U sau khi lặp k_max lần với policy tương ứng với 2 hàm hỗ trợ là lookahead và policy(s) đã nói ở trên

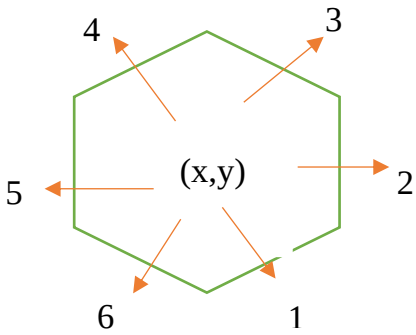
`Solve(mdp,policy,k_max)`

Kết quả trả về là 1 mảng các hành động thông qua việc gọi hàm iterative_policy_evaluation

3.4 THỰC THI

- Để chạy code, ta vào file "./src/hexworld/main.jl" để chạy. (click vào hình tam giác bên góc phải trên để chạy). Kết quả trả ra là các hành động được đánh số từ 1 đến 6 ứng với mỗi trạng thái được lưu dưới dạng tọa độ (x,y)

```
state : (0, 0) <==> action :2
state : (1, 0) <==> action :3
state : (2, 0) <==> action :1
state : (3, 0) <==> action :3
state : (0, 1) <==> action :1
state : (1, 1) <==> action :4
```



+ Các số tương ứng với các hướng đi tại trạng thái (x,y)

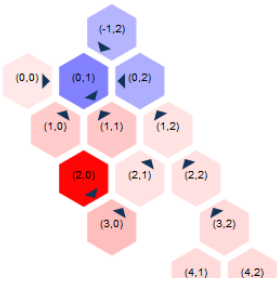
- Các hình ảnh visualize thì được lưu tại folder “src\hexworld\visualization” dưới dạng .png

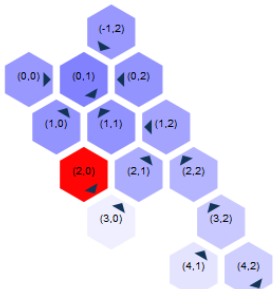
*Lưu ý: import package Luxor để vẽ hình

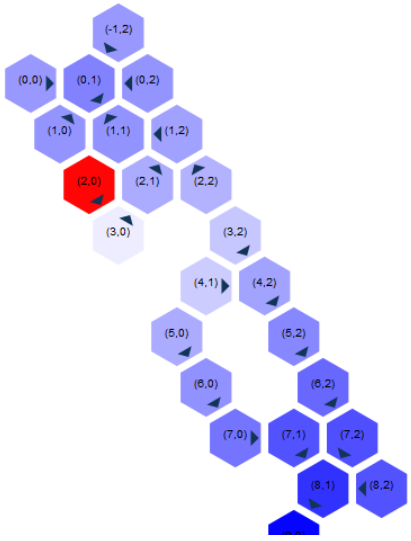
4. Kết quả

Phân tích kết quả:

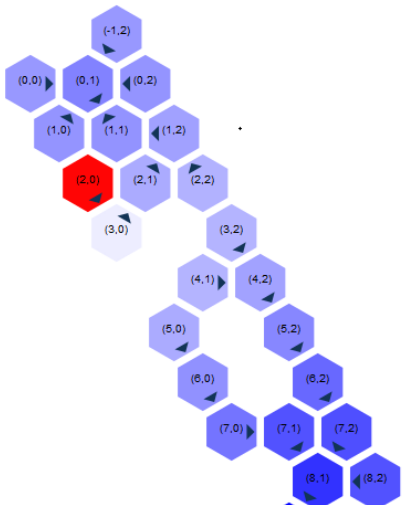
Kết quả nhận được sau 4 vòng lặp tại hàm Solve như sau:







] Iterat



Interation 3

Interation 4

Nhận xét:

- Kết quả trả về đúng với kết quả mong muốn.

- Thuật toán chạy đúng và hội tụ với số vòng lặp thấp tóm tắt kết quả

Điểm mạnh:

- Hiểu được mô hình thuật toán hexworld

- Tìm ra được policy để giải quyết bài toán

- Visualize được kết quả mô hình chạy của thuật toán

- Phân chia code thành nhiều file thuận tiện cho việc cải tiến hoặc chỉnh sửa sau này

Điểm yếu:

- Policy định hướng chưa thật sự tốt để tìm ra được phần thưởng tốt nhất.

Vd: input phần thưởng có là 5 và 10 thì các ô lân cận 5 thường sẽ tiến về 5 hơn là so với việc tìm đường đi hướng đến 10 là giá trị phần thưởng cao hơn

II – Rock-Paper-Scissors

1. Phát biểu bài toán

Bài toán cần giải quyết là trò chơi phổ biến được gọi là Rock-Paper-Scissors. Có 2 người chơi với nhau, mỗi người lựa chọn một 1 trong 3 trạng thái Rock, Paper hoặc Scissors. Nếu 2 người chơi cùng ra một trạng thái giống nhau thì 2 người chơi hòa, còn người chơi ra khác nhau sẽ nhận được thắng thua theo quy luật sau:

- Rock thắng Paper.
- Scissors thắng Paper.
- Paper thắng Rock.

Sau đó, chúng ta sẽ tính toán điểm dựa theo kết quả của các vòng đấu. Thắng một vòng đấu thì người chơi được số điểm là +1. Thua 1 vòng đấu, người chơi được số điểm là -1. Còn nếu hòa thì cả 2 người chơi đều thêm 0 điểm.

Trò chơi này gọi là zero-sum game. Bởi vì với bất kì lựa chọn nào của 2 người chơi, kết quả cảu trò chơi cũng sẽ bằng 0. Vì nếu lựa chọn khác nhau, 1 trong 2 người chơi chiến thắng thì có điểm là +1 và người còn lại sẽ có điểm -1, còn nếu giống nhau thì cả 2 đều nhận số diểm là 0. Trong các trò chơi zero-sum, có giá trị V mà ở đó người chơi 1 có thể đảm bảo rằng họ sẽ nhận được ít nhất V điểm mà không quan tâm người chơi 2 sử dụng các hành động gì. Ngược lại, người chơi 2 cũng đảm bảo có thể nhận -V điểm mà không quan tâm đến người chơi 1. Trong trò chơi Rock-Paper-Scissors, V=0, nghĩa là cả 2 người chơi đều đảm bảo rằng họ có thể đạt được 0 điểm bằng cách chơi một cách ngẫu nhiên đồng nhất của cả 3 hành động. *Lưu ý rằng*, sự ngẫu nhiên ở đây là cần thiết để đảm bảo cho một điểm số bằng 0. Vì vậy, trong bài toán này, ta sẽ thực hiện các policy (chính sách mà các người chơi đưa ra) để tính toán các chiến lược quyết định.

2. Thách thức

Khó khăn chung lớn nhất đề ra là các khái niệm về lý thuyết trò chơi thường rất phức tạp và rất khó hiểu.

3. Thực nghiệm

3.1 MÔ HÌNH TÍNH TOÁN

Mô hình Simple Games là mô hình cơ bản cho lý luận đa phương. *Có các agent* $i \in I = \{1, 2\}$ lựa chọn hành động a_i để tối đa phần thưởng r_i nhận được. Cấu trúc cụ thể như sau:

- Bài toán có 2 người chơi, là 2 agents $I = \{1, 2\}$
- discount factor $\gamma = 0.9$
- *joint action space:* $A = A_1 * A_2 * \dots * A_k$: gồm các hành động có thể có của các người chơi. Cụ thể trong trò chơi Rock-Paper-Scissors sẽ gồm 3 hành động là Rock, Paper, Scissors.
- $A = A1 \times A2$ với $Ai = \{rock, paper, scissors\}$
- *joint action:* $a = (a_1, a_2, \dots, a_k)$: Các hành động được chọn đồng thời giữa các người chơi kết hợp lại. Ví dụ: (a_1= (R, P); a_2=(C, C); …)
- *joint reward function:* $R(a) = (R_1(a), \dots, R_k(a))$: Điểm số của *joint action* a_k. Dựa theo ma trận điểm.
- *joint policy* π *:* là xác suất của các *joint action* thực hiện bởi người chơi. Xác suất agent i chọn hành động a là $\pi^i(a)$. Tiện ích của *joint policy* π từ góc nhìn của agent i là:

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

- Ma trận điểm thưởng phạt được mô tả như sau:

STATES	ROCK	PAPER	SCISSORS
ROCK	0,0	-1,1	1,-1
PAPER	1,-1	0,0	-1,1
SCISSORS	-1,1	1,-1	0,0

3.2 PHƯƠNG PHÁP GIẢI QUYẾT

a) Nash Quilibrium

Cân bằng Nash luôn tồn tại cho các trò chơi có không gian hành động hữu hạn. Cân bằng Nash là một policy chung π trong đó tất cả các agent đều tuân theo một phản ứng tốt

nhất. Nói cách khác, điểm cân bằng Nash là một policy chung, trong đó không agent nào có động cơ đơn phương chuyển đổi policy của họ. Những agent này sẽ không nhận được thêm lợi ích nếu những agent khác vẫn giữ nguyên policy của họ.

Do đó, Cân bằng Nash là giải pháp cho một trò chơi trong đó hai hoặc nhiều người chơi có một chiến lược và với việc mỗi người tham gia cân nhắc lựa chọn của đối thủ, anh ta không có động cơ, không có gì để đạt được, bằng cách chuyển đổi chiến lược của mình.

Chúng ta biết rằng, chiến lược thuần tùy của cân bằng Nash (A pure-strategy Nash quilibrium) nghĩa là người chơi luôn chọn cùng 1 trạng thái sau mỗi vòng đấu, sẽ không tồn tại trong trò chơi Kéo – Búa – Bao vì một người chơi có thể dễ dàng phản ứng lại với lựa chọn của người chơi khác. Ví dụ, người chơi 1 luôn luôn chọn Kéo thì phản hồi tốt nhất của người chơi 2 là Búa. Vì vậy có một chiến lược hỗn hợp của cân bằng Nash (A mixed-strategy Nash quilibrium) nghĩa là mỗi người chơi chọn 1 trạng thái với một tỷ lệ xác suất. Giải sử mỗi người chơi lựa chọn các trạng thái với xác suất ngẫu nhiên đồng nhất. Nghĩa là 1/3 lần chọn Búa, 1/3 lần chọn Kéo và 1/3 lần chọn Bao.

	Rock 1/3	Paper 1/3	Scissors 1/3
Rock 1/3	0,0	-1,1	1,-1
Paper 1/3 Paper	1,-1	0,0	-1,1
Scissors 1/3 Scissors	-1,1	1,-1	0,0

Ta tính toán Utility mong đợi của agent 1 (màu đỏ) là:

$$U^{i}(\pi) = 0\frac{1}{3}\frac{1}{3} - 1\frac{1}{3}\frac{1}{3} + 1\frac{1}{3}\frac{1}{3}$$

$$+1\frac{1}{3}\frac{1}{3} + 0\frac{1}{3}\frac{1}{3} - 1\frac{1}{3}\frac{1}{3}$$

$$-1\frac{1}{3}\frac{1}{3} + 1\frac{1}{3}\frac{1}{3} + 0\frac{1}{3}\frac{1}{3} = 0$$

Utility mong đợi của agent 2 cũng là 0

Bất kì thay đổi chiến thuật nào của agent sẽ làm giảm số điểm này nên đây là một cân bằng Nash của trò chơi này.

Dưới đây là một chương trình phi tuyến tính để tính toán cân bằng Nash cho trò chơi Rock-Paper-Scissors.

```
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
```

Chúng ta sẽ không bàn tới cách cài đặt này ở đây vì phương pháp này tốn chi phí cao và cài đặt phức tạp. Để tính toán một cân bằng Nash, ta sẽ dùng một cách tiếp cận khác tiết kiệm về mặt tính toán hơn là Iterated Best Response.

b) Iterated Best Response

Bởi vì tính toán cân bằng Nash có thể tốn kém về mặt tính toán, một cách tiếp cận thay thế là áp dụng lặp đi lặp lại các phản hồi tốt nhất trong một loạt các trò chơi lặp lại. Trong *Iterated best response*, chúng tôi xoay vòng ngẫu nhiên giữa các người chơi, lần lượt giải

quyết policy phản hồi tốt nhất (Best Response) của từng người chơi. Quá trình này có thể hội tụ về trạng thái cân bằng Nash.

Best Response của agent i đối với các policy của agent khác là một policy mà Utility mà họ nhận được sẽ luôn luôn tốt nhất, nghĩa là họ sẽ không có động lực để thay đổi policy của mình đối với policy của agent khác. Cách cài đặt, chúng ta đơn giản lặp đi lặp lại các hành động của agent i sau đó trả về hành động mà có Utility cao nhất.

Thuật giải:

```
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
```

c) Fictitious Play

Một cách tiếp cận thay thế trong việc tính toán các policy cho các agent khác nhau là để họ chơi với nhau trong mô phỏng và học cách phản hồi tốt nhất. Thuật toán cung cấp một triển khai của vòng lặp mô phỏng. Ở mỗi lần lặp lại, chúng tôi đánh giá các policy khác nhau để có được một *joint action* và sau đó *joint action* này được các agent sử dụng để cập nhật các policy của họ. Chúng tôi có thể sử dụng các cách khác nhau để cập nhật các policy nhằm đáp ứng các *joint action* được quan sát. Phần này tập trung vào Fictitious Play, trong đó các agent sử dụng các ước tính khả năng xảy ra tối đa của các policy mà các agent khác tuân theo. Mỗi agent tuân theo phản ứng tốt nhất của riêng mình, giả sử các agent khác hành động theo những ước tính đó.

Để tính toán ước tính khả năng xảy ra tối đa, agent i theo dõi số lần agent j thực hiện hành động a^j, lưu trữ nó trong bảng $N^i(j, a^j)$. Các số đếm này có thể được khởi tạo thành bất kỳ giá trị nào, nhưng chúng thường được khởi tạo thành 1 để tạo ra độ chắc chắn đồng nhất ban đầu. Agent i tính toán phản hồi tốt nhất của họ với giả định rằng mỗi agent j tuân theo policy ngẫu nhiên (stochastic policy)

$$\pi^j(a^j) \propto N^i(j, a^j)$$

Tại mỗi lần lặp, chúng tôi có mỗi agent hành động theo một phản hồi tốt nhất, giả sử áp dụng các policy dựa trên số ngẫu nhiên này cho các agent khác. Sau đó, chúng tôi cập nhật số lượng hành động cho các hành động được thực hiện. Fictitious Play không được đảm bảo hội tụ đến trạng thái cân bằng Nash.

Thuật giải:

```
function simulate(𝒫::SimpleGame, π, k_max)
    for k = 1:k_max
        a = [πi() for πi in π]
        for πi in π
            update!(πi, a)
        end
    end
    return π
end
```

3.3 MÔ TẢ CODE

a) SimpleGame

```
struct RockPaperScissors end

n_agents(simpleGame::RockPaperScissors) = 2
ordered_actions(simpleGame: :RockPaperScissors, i::Int) =
[:rock, :paper, :scissors]
ordered_joint_actions(simpleGame::RockPaperScissors) =
vec(collect(Iterators.product([ordered_actions(simpleGame, i) for i in
1:n_agents(simpleGame)]...)))

n_joint_actions(simpleGame::RockPaperScissors) =
length(ordered_joint_actions(simpleGame))
n_actions(simpleGame::RockPaperScissors, i::Int) =
length(ordered_actions(simpleGame, i))
```

- Struct Rock-Paper-Scissors là đối tượng đại diện cho bài toán
- Các biến hoặc hàm khởi tại cho Struct gồm:
- *n_agents:* mang số lượng của người chơi
- *ordered_actions:* trả về là các hành động của agent i, trong bài toán này thì các hành động của các agent là 3 trạng thái {rock, paper, scissors}.
- *ordered_joint_actions*: trả ra 1 vector chứa tất cả các hành động kết hợp dạng (a^i, a^j) của 2 người chơi
- *n_joints_actions*: số lượng các hành động chung của các agent trong bài toán.
- *n_actions:* số lượng hành động (lựa chọn) của agent i.

```
function reward(simpleGame::RockPaperScissors, 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

function joint_reward(simpleGame::RockPaperScissors, a)
      return [reward(simpleGame, i, a) for i in 1:n_agents(simpleGame)]
end
```

- *reward*: phần thưởng nhận được của agent i sau khi hành động a được lựa chọn.
- *joint_reward*: trả ra 1 list phần thưởng [r^1, r^2, …] với r^i là phần thưởng cho agent i.

Sau đó, ta khởi tạo bài toán đã mô hình

```
function SimpleGame(simpleGame::RockPaperScissors)
    return SimpleGame(
          0.9,
          vec(collect(1:n_agents(simpleGame))),
          [ordered_actions(simpleGame, i) for i in 1:n_agents(simpleGame)],
          (a) -> joint_reward(simpleGame, a)
    )
end
```

⇨ SimpleGame(0.9, 2, {rock, paper, scissors}, joint_reward)

b) Iterated Best Response

```
struct IteratedBestResponse
    k_max # number of iterations
```

```
    π # initial policy
end
```

Struct IteratedBestResponse gồm 2 thuộc tính: số nguyên k_max và policy hiện tại của

Agent là π. Thuật toán Iterated Best Response luân phiên từng agent và áp dụng mô hình Best Response cho từng agent, policy được giải ra sẽ gán vào policy π, thuật toán kết thúc sau k_max lần duyệt.

```
function best_response(𝒫::SimpleGame, π, i)
    U(ai) = utility(𝒫, joint(π, SimpleGamePolicy(ai), i), i)
       ai = argmax(U, 𝒫.𝒜[i])
       return SimpleGamePolicy(ai)
 end
```

Thuật toán áp dụng mô hình best_response để trả ra một policy mà là phản hồi tốt nhất đối với các policy của agent khác

- *IteratedBestResponse*: hàm khởi tạo cho thuật toán, trả ra policy có xác suất cho hành động là 1.0
- *solve:* mô tả lại thuật toán

c) Fictitious Play:

```
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]
        println(π)
    end
    return π
end
```

Mô hình Fictitious Play: Ta sẽ xây dựng 2 mô hình Fictitious Play cho 2 agent bao gồm thêm 1 thuộc tính là N: số lượng các hành động mà người chơi thực hiện (Rock, Paper hoặc Scissors)

```
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
```

Sau đó, ta thực hiện mô phỏng trò chơi để cập nhật lại *joint policy* của từng agent. *Joint policy* π ở đây là vector của các policies của 2 agent và từng cái được cập nhật

```
function simulate(𝒫::SimpleGame, π, k_max)
    for k = 1:k_max
        a = [πi() for πi in π]
        for πi in π
            update!(πi, a)
        end
    end
    return π
end
```

Ta mô phỏng đơn giản là duy trì số lượng các hành động của agent theo thời gian mỗi lần chơi, sau đó lấy trung bình của chúng để làm policy ngẫu nhiên. Sau đó, với policy như vậy, ta tính toán phản hồi tốt nhất cho policy này và tính toán Utility tốt nhất.

```
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
```

4. Kết quả

Phân tích kết quả:

a) Nash Quilibrium

Kết quả trả về là xác suất của 2 người chơi nếu ta giả sử 2 người chơi đều sử dụng chính sách ngẫu nhiên đồng nhất, thì khi đạt được cân bằng Nash, xác suất của mỗi action là 1/3

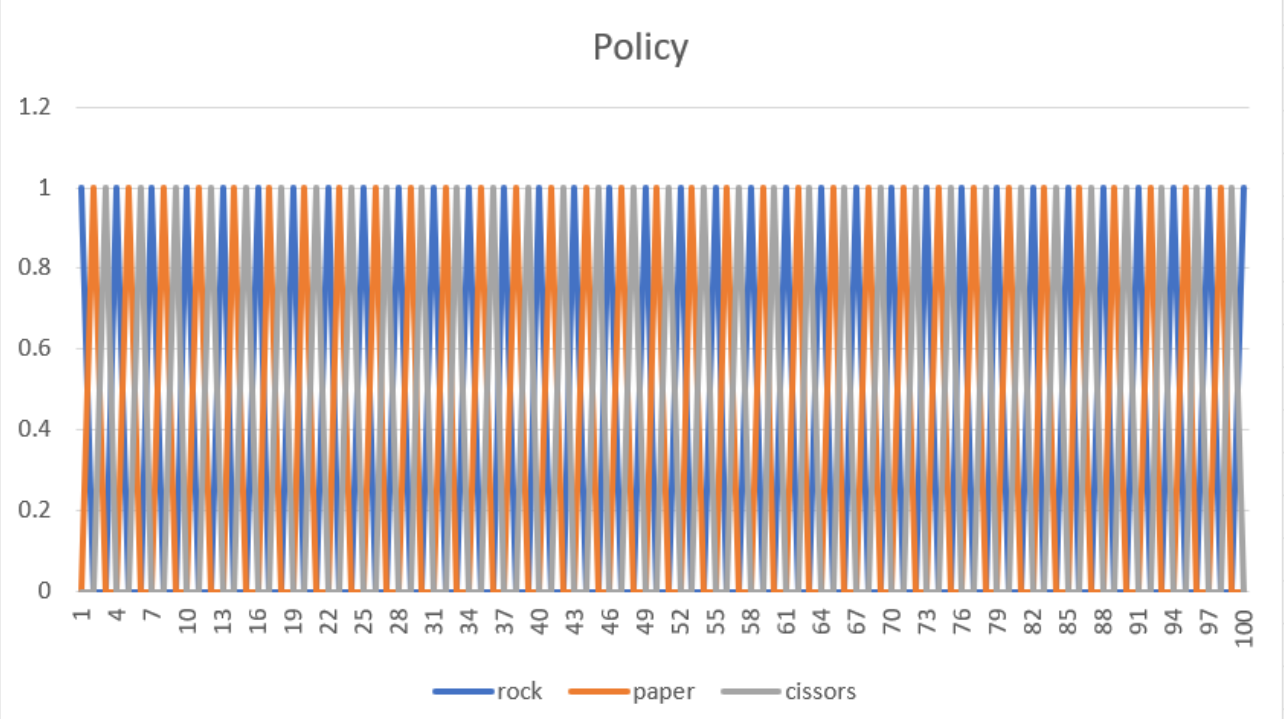
```
Agent 1: SimpleGamePolicy(Dict(:scissors => 0.3333333333333337, :rock => 0.3333333333333337, :paper => 0.3333333333333337))
Agent 2: SimpleGamePolicy(Dict(:scissors => 0.3333333333333337, :rock => 0.3333333333333337, :paper => 0.3333333333333337))
```

b) Iterated Best Response

Kết quả hàm solve: trả về các Best Response

```
SimpleGamePolicy[SimpleGamePolicy(Dict(:rock => 1.0)),
SimpleGamePolicy(Dict(:rock => 1.0))]
SimpleGamePolicy[SimpleGamePolicy(Dict(:paper => 1.0)),
SimpleGamePolicy(Dict(:paper => 1.0))]
SimpleGamePolicy[SimpleGamePolicy(Dict(:scissors => 1.0)),
SimpleGamePolicy(Dict(:scissors => 1.0))]
SimpleGamePolicy[SimpleGamePolicy(Dict(:rock => 1.0)),
SimpleGamePolicy(Dict(:rock => 1.0))]
SimpleGamePolicy[SimpleGamePolicy(Dict(:paper => 1.0)),
SimpleGamePolicy(Dict(:paper => 1.0))]
SimpleGamePolicy[SimpleGamePolicy(Dict(:scissors => 1.0)),
SimpleGamePolicy(Dict(:scissors => 1.0))]
SimpleGamePolicy[SimpleGamePolicy(Dict(:rock => 1.0)),
SimpleGamePolicy(Dict(:rock => 1.0))]
```

Thống kê lại, ta có xác suất Rock:1/3, Paper:1/3, Sccissors: 1/3. Kết quả ở trạng thái cân bằng Nash



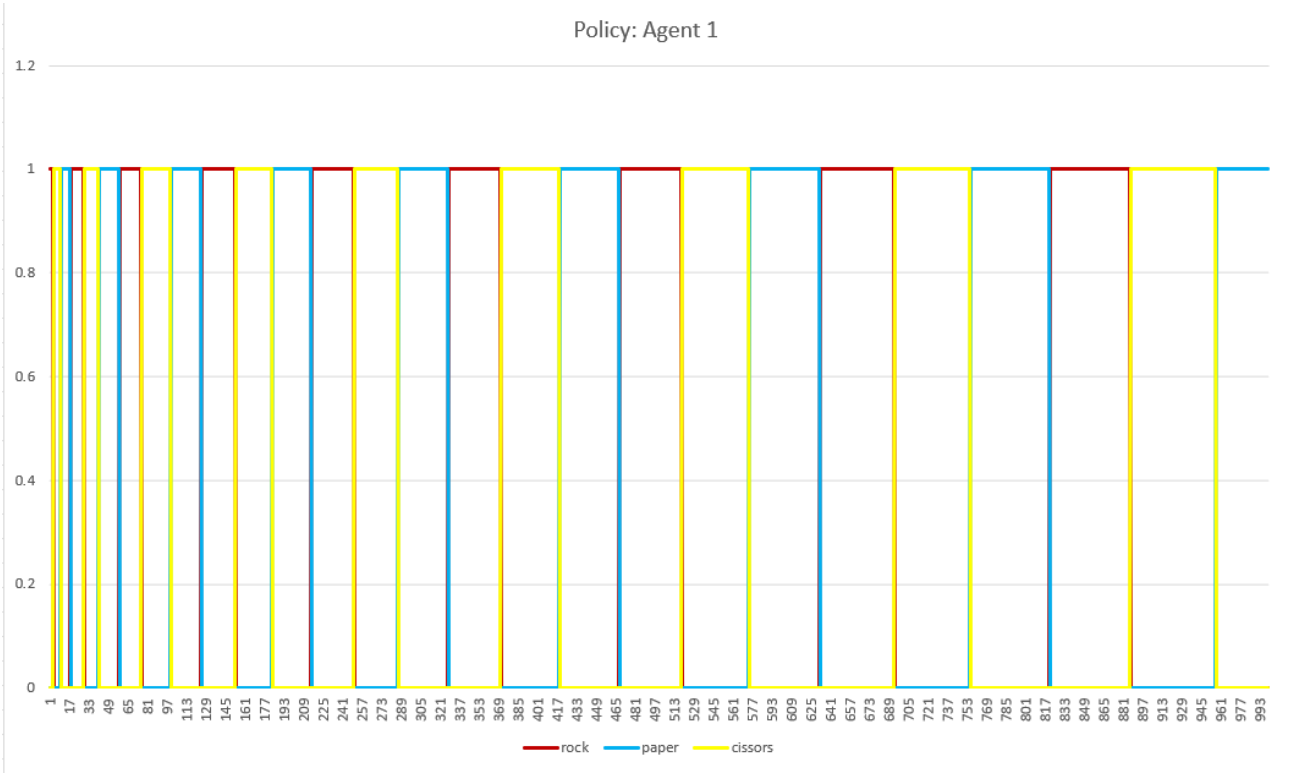
c) Fictitious Play

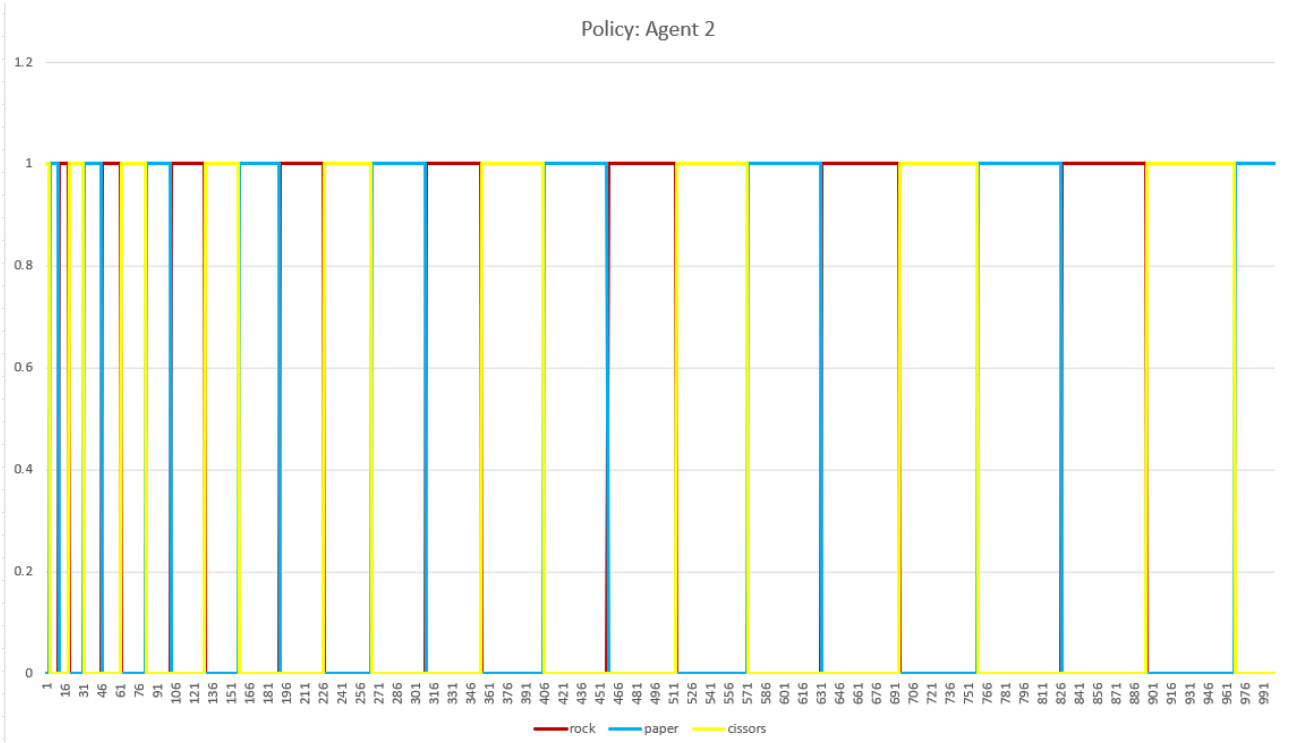
Với kết quả các policy ta nhận được sau mỗi lần update, ta đã lặp qua k_max = 300 lần và thu được kết quả thực nghiệm như sau lần lượt của 2 người chơi. Ta có policy nhận được sau mỗi lần lặp:

```
Agent 1[Dict(:scissors => 39, :rock => 30, :paper => 30), Dict(:scissors => 27, :rock => 38, :paper => 34)]SimpleGamePolicy(Dict(:paper => 1.0))
Agent 2[Dict(:scissors => 39, :rock => 30, :paper => 30), Dict(:scissors => 27, :rock => 38, :paper => 34)]SimpleGamePolicy(Dict(:rock => 1.0))
Agent 1[Dict(:scissors => 39, :rock => 30, :paper => 31), Dict(:scissors => 27, :rock => 39, :paper => 34)]SimpleGamePolicy(Dict(:paper => 1.0))
Agent 2[Dict(:scissors => 39, :rock => 30, :paper => 31), Dict(:scissors => 27, :rock => 39, :paper => 34)]SimpleGamePolicy(Dict(:rock => 1.0))
Agent 1[Dict(:scissors => 39, :rock => 30, :paper => 32), Dict(:scissors => 27, :rock => 40, :paper => 34)]SimpleGamePolicy(Dict(:paper => 1.0))
Agent 2[Dict(:scissors => 39, :rock => 30, :paper => 32), Dict(:scissors => 27, :rock => 40, :paper => 34)]SimpleGamePolicy(Dict(:rock => 1.0))
Agent 1[Dict(:scissors => 39, :rock => 30, :paper => 33), Dict(:scissors => 27, :rock => 41, :paper => 34)]SimpleGamePolicy(Dict(:paper => 1.0))
Agent 2[Dict(:scissors => 39, :rock => 30, :paper => 33), Dict(:scissors => 27, :rock => 41, :paper => 34)]SimpleGamePolicy(Dict(:rock => 1.0))
Agent 1[Dict(:scissors => 39, :rock => 30, :paper => 34), Dict(:scissors => 27, :rock => 42, :paper => 34)]SimpleGamePolicy(Dict(:paper => 1.0))
Agent 2[Dict(:scissors => 39, :rock => 30, :paper => 34), Dict(:scissors => 27, :rock => 42, :paper => 34)]SimpleGamePolicy(Dict(:rock => 1.0))
```

Ta thu được từ điển chứa số lượng của các hành động Rock, Paper và Scissors cùng với Policy của cả 2 agent.

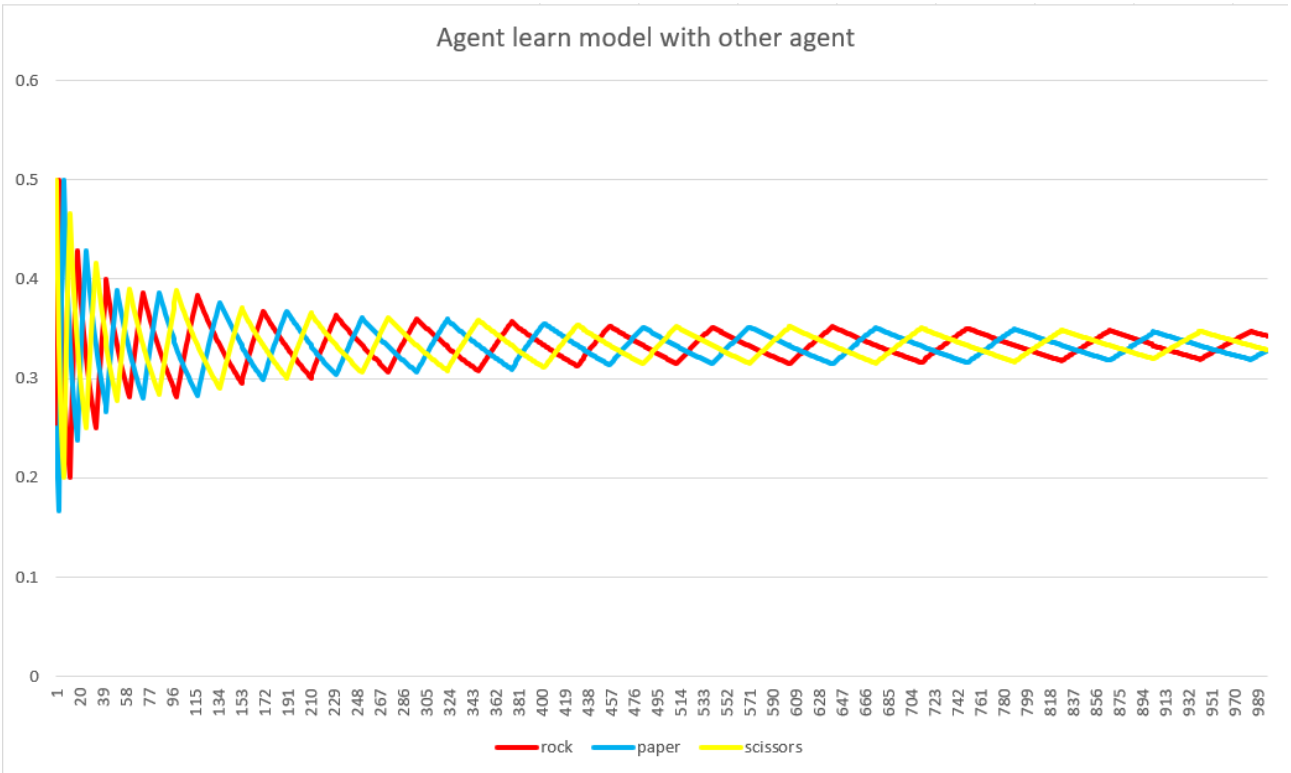
Ta vẽ biểu đồ về policy của 2 người chơi:





Ta có nhận xét rằng với số lần lặp tăng dần, tần suất mỗi agent thay đổi policy của họ cũng sẽ tăng lên. Ví dụ, ở 100 lần lặp đầu, agent có xu hướng thay đổi policy của mình trong không quá 25 vòng lặp. Nhưng ở các vòng lặp lớn hơn, tần suất thay đổi có thể lên đến 75. Như vậy, với số vòng lặp tiến đến vô cùng, người chơi sẽ có xu hướng không thay đổi policy của họ.

Ta xác suất các hành động của agent và xem xác suất thay đổi theo thời gian qua biểu đồ:



Với số vòng lặp tăng dần, xác suất của cả 3 hành động đều hội tụ dần về 1/3. Đây là xác suất của của 3 hành động trong 1 cân bằng Nash. Chúng ta đạt được kết quả như mong muốn.

Qua việc mô phỏng trò chơi giữa 2 agent trong mô hình Fictitious Play, họ sẽ càng tiếp cận dần đến với chính sách ngẫu nhiên và đạt cân bằng Nash.

⇨ Tổng kết chung, qua cả 3 mô hình mà ta phân tích, các policy đều hội tụ về cân bằng Nash, làn trạng thái mà tần suất các action đều là 1/3 như lý thuyết chúng ta đã mô tả.

Điểm mạnh:

- Áp dụng được vài mô hình và hiểu được cách tiếp cận của bài toán.
- Kết quả bài toán phản ánh được mong đợi của ta.
- Code phân chia rõ ràng, dễ đọc hiểu

Điểm yếu:

- Coding chưa tận dụng khả năng của Julia trong vẽ biểu đồ (dùng công cụ bên ngoài).

III – Traveler's Dilemma

1. Phát biểu bài toán

Bài toán "Travel's dilemma" nói về việc 2 người bị mất hành lý ở 1 sân bay và 2 hành lý đó giống y hệt nhau. Người quản lý sân bay vì muốn tránh lừa đảo nên đã đưa ra phương pháp sau: Cho 2 người viết xuống 1 tờ giấy giá tiền của hành lý trong khoảng giữa 2$ và 100$ và họ không được trao đổi với nhau. Nếu 2 người viết giá tiền bằng nhau, thì quản lý sẽ cho là 2 người đều thành thật và 2 người đều nhận được số tiền đó. Nếu 2 người viết giá tiền khác nhau, thì người viết giá tiền cao hơn sẽ nhận được số tiền bằng giá tiền thấp hơn -2, còn người viết giá tiền thấp hơn sẽ được thưởng 2$ vào giá tiền thấp hơn đó.

Ví dụ với 2 người Tuấn và Lan, ban đầu Tuấn nghĩ rằng mình sẽ ghi số lớn nhất là 100$ và sẽ nhận được 100$ nếu Lan cũng tham lam như vậy. Nhưng rồi Tuấn nghĩ là nếu mình viết 99$, thì mình sẽ nhận được 101$ và Lan nhận được 98$, nhưng sau đó Tuấn chợt nhận ra rằng Lan cũng sẽ nghĩ giống mình là cũng sẽ viết 99$. Thế là Tuấn lại viết 98$, và lại nghĩ rằng Lan nghĩ giống mình, quy trình này lặp lại cho tới khi cả 2 người viết 2$ trên giấy và cùng đi về với 2$. Kết quả này là Nash Equilibrium của bài toán, và cách suy nghĩ này được gọi là quy nạp ngược (Backward induction).

Nhưng thực tế con người sẽ không có suy nghĩ như vậy và chỉ viết trong khoảng từ 95$ tới 97$. Bài toán này là một nghịch lý trong Game Theory, con người hành động bất hợp lý (irrational) lại mang về lợi ích nhiều hơn hành động hợp lý (rational). Như John Nash từng nói "Game theory predicts that the Nash equilibrium will occur when Traveler's Dilemma is played rationally.".

Việc thưởng phạt dựa trên lựa chọn của 2 người có thể được biểu diễn bằng 1 payoff matrix như sau:

	2	3	4	...	98	99	100
2	2 2	4 0	4 0	...	4 0	4 0	4 0
3	0 4	3 3	5 1	...	5 1	5 1	5 1
4	0 4	1 5	4 4	...	6 2	6 2	6 2
...	...	...	...		...	...	...
98	0 4	1 5	2 6	...	98 98	100 96	100 96
99	0 4	1 5	2 6	...	96 100	99 99	101 97
100	0 4	1 5	2 6	...	96 100	97 101	100 100

Hình 1: The Traveler's Dilemma, Kaushik Basu, Scientific American , Vol. 296, No. 6 (JUNE 2007), pp. 90-95

Payoff matrix này có thể tóm gọn hết ý tưởng của bài toán Travel's Dilemma cho người xem, cột bên trái cùng là đại diện cho những lựa chọn của Lan, hàng trên cùng là đại diện cho những lựa chọn của Tuấn. Như ta thấy thì khi Lan chọn 100, Tuấn chọn 100 thì cả 2 người đều nhận được 100, và khi Tuấn chọn 100, Lan chọn 99 thì Tuấn nhận được 97 còn Lan lại nhận được 101, và ngược lại. Bảng cứ theo quy luật này đi lên cho tới khi gặp (2; 2) là Nash equilibrium, (2; 2) là Nash equilibrium vì nó là trạng thái cuối cùng mà người chơi sẽ nghĩ tới và sẽ không muốn chuyển qua trạng thái nào nữa.

2. Thách thức

Thách thức lớn nhất về bài Traveler's Dilemma đó chính là sự nghịch lý về hành vi hợp lí và hành vi thực tế của con người như đã có đề cập ở trên. Hành vi hợp lí theo Lý thuyết trò chơi đó chính là người chơi sẽ luôn cố gắng làm cho mình hơn đối thủ, nên sẽ cứ tiếp tục suy nghĩ tới số tiền nhỏ hơn vì nó sẽ làm cho mình nhận được tiền thưởng cao hơn đối thủ, hành vi này hội tụ về việc cả 2 người cùng chọn 2 – Nash equilibrim – vì sau đó không còn lựa chọn nào mà 2 người sẽ muốn chọn nữa. Còn hành vi thực tế của con người là sẽ không quan tâm nhiều tới người kia mà quan tâm tới số tiền mình nhận được, nên thường lựa chọn sẽ nằm dao động ở khoảng trên 95. Việc mâu thuẫn như vậy dẫn tới việc chọn policy để mô phỏng lại bài toán phải được cân nhắc kỹ cho cả 2 loại hành vi.

Còn khó khăn đối với bài toán đó chính là việc hiểu và tìm được code tham khảo cho bài toán này, vì lý thuyết trò chơi và những khái niệm liên quan là rất mới nên còn hơi lúng túng khi tra cứu trên mạng.

3. Thực nghiệm

3.1 MÔ HÌNH HOÁ TÍNH TOÁN:

a) Simple game

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

Trong simple game, những agent chọn một hành động a^i để tối ưu hoá phần thưởng r^i của họ.

Một joint action space $A = A^1 \times \cdots \times A^k$ bao gồm mọi hoán vị của những hành động A^i mà mỗi agent có thể có. Những hành động được chọn cùng một lúc qua khắp các agent có thể được gộp lại thành 1 joint action $a = (a^1, \ldots, a^k)$ từ joint action space trên.

Joint reward function $R(a) = (R^1(a), \ldots, R^k(a))$ mô tả phần thưởng nhận được từ joint action $a = (a^1, \ldots, a^k)$. Joint reward là $r = (r^1, \ldots, r^k)$.

Còn với γ (discount factor), đó chỉ là biến đại diện cho xác suất "đi tiếp" của lần lặp hiện tại để cho nó không lặp vô tận, giá trị này nằm giữa 0 và 1 nhưng không thể bằng 0 và 1.

Với bài toán traveler's dilemma, discount factor sẽ bằng 0.9, agents sẽ gồm 1 và 2, joint action space sẽ là một list chứa các hành động chọn 2…100 của mỗi agent, và joint reward function sẽ gồm 2 hàm reward giống hệt nhau của 2 agent có dạng như sau:

$$r = \begin{cases} action \; khi \; action == otherAgentAction \\ action + 2 \; khi \; action < otherAgentAction \\ otherAgentAction - 1 \; khi \; action > otherAgentAction \end{cases}$$

b) Iterated best response

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

```
end
```

c) Hierarchical softmax

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

Cụ thể của 2 mô hình tính toán Iterated best response và Hierarchical softmax sẽ được nói rõ ở phần 3.2.

3.2 PHƯƠNG PHÁP GIẢI QUYẾT

Đối với bài này thì ta sẽ chọn 2 cấu hình policy Iterated Best Response và Hierarchical Softmax, tương ứng cho 2 hành vi là hành vi hợp lí và hành vi thực tế.

3.2.1 Iterated Best Response

a) Best response

Với việc người chơi biết những gì mà những người chơi khác đang làm, một chiến thuật là một best response nếu và chỉ nếu người chơi không thể có lợi hơn khi chuyển sang chiến thuật khác. Có thể có nhiều best response.

Nếu chúng ta chỉ sử dụng những chính sách xác định (deterministic policies), thì một phản ứng mang tính xác định (deterministic best response) cho đối thủ đang 1 dùng 1 chính sách nào đó sẽ được tính một cách dễ dàng. Chúng ta chỉ cần duyệt qua hết hành động của agent thứ tự I và trả về cái mà cho mình lợi ích tốt nhất:

$$\underset{a^i \in A^i}{\operatorname{argmax}}\, U^i(a^i, \pi^{-i})$$

Định nghĩa Nash Equilibrium bằng Best response: một trò chơi đang ở trong trạng thái Nash equilibrium nếu và chỉ nếu tất cả người chơi đều đang có phản ứng tốt nhất (best response) cho việc mà những người chơi khác đang làm.

Cách tìm best response cho 1 trò chơi là cô lập 1 chiến thuật của 1 người chơi, và tìm xem phản ứng tốt nhất mà những người chơi còn lại nên có để đáp trả chiến thuật đó. Cụ thể với traveler’s dilemma:

	2	3	4	5	…
2	2; 2	4; 0	4; 0	4; 0	
3	0; 4	3; 3	5; 1	5; 1	
4	0; 4	1; 5	4; 4	6; 2*	
5	0; 4	1; 5	2; 6	5; 5	
…					

Những lựa chọn ở cột trái cùng sẽ là của player1, còn những lựa chọn ở hàng trên cùng sẽ là của player2. Ta chỉ xét những giá trị từ 2 tới 5 để dễ ví dụ.

Như bảng thì ta đã cô lập chiến thuật của player2 là chọn giá tiền 5, thì với chiến thuật của player2 như vậy, thì chiến thuật tốt nhất của player1 sẽ là chọn 4 vì nó cho ra giá tiền cao nhất cho player đó, là 6. Cho nên best response của player1 khi player2 chọn 5 là chọn 4.

	2	3	4	5	…
2	2; 2	4; 0	4; 0	4; 0	
3	0; 4	3; 3	5; 1*	5; 1	
4	0; 4	1; 5	4; 4	6; 2*	
5	0; 4	1; 5	2; 6	5; 5	
…					

Tương tự, khi cô lập chiến thuật chọn 4 của player2, thì chiến thuật tốt nhất của player1 là 3 vì nó cho ra giá tiền lớn nhất cho player1 là 5, nên best response của player1 khi player2 chọn 4 là chọn 3.

	2	3	4	5	…
2	2*; 2*	4; 0	4; 0	4; 0	
3	0; 4	3; 3	5; 1*	5; 1	
4	0; 4	1; 5	4; 4	6; 2*	
5	0; 4	1; 5	2; 6	5; 5	
…					

Giờ ta sẽ xét cùng lúc 2 chiến thuật của 2 người chơi là chọn 2. Ta sẽ cô lập lựa chọn 2 của player2 trước (cột màu đỏ), thì khi player2 chọn 2, best response của player1 sẽ là 2 vì nó cho ra giá tiền cao nhất cho player1 là 2 thay vì 0 như những lựa chọn dưới. Tương tự, khi cô lập lựa chọn 2 của player1 (hàng màu xanh), thì player2 tốt nhất nên chọn 2 luôn vì nó cũng ra giá tiền cao nhất là 2 thay vì là 0 như những lựa chọn sau.

Vì ô chiến thuật đó, cả 2 player đều có best response cho lẫn nhau, nên (2; 2) chính là Nash equilibrium cho bài toán.

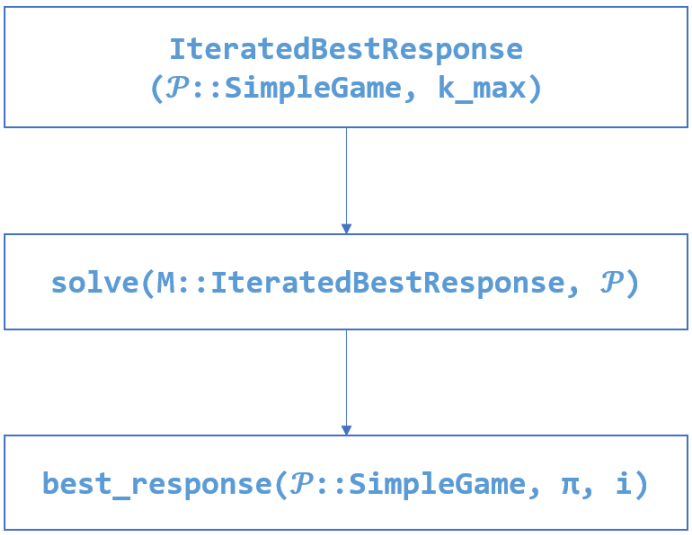
b) Iterated best response

Với phương pháp tìm Nash Equilibrium bằng Best response ta vừa mô tả ở trên, thì chi phí sẽ rất là cao đối với 1 bài toán có nhiều hành vi như vậy, cho nên, ta phải có 1 phương pháp tối ưu hơn để tính được Nash Equilibrium cho bài toán.

Iterated best response tính được Nash Equilibrium bằng cách lặp lại trò chơi 1 số lần cố định, cụ thể là k_max lần, với mỗi người chơi chọn 1 chính sách ban đầu, mỗi lần lặp như vậy 2 người chơi sẽ chọn chiến thuật dựa vào chiến thuật ở lần lặp trước, và ta sẽ tính best response cho 2 chiến thuật đó. Việc lặp lại như vậy sẽ có khả năng cao hội tụ về Nash Equilibrium. Nhưng nó chỉ hội tụ với 1 số loại trò chơi, nên việc quan sát quá trình lặp là một điều phổ biến.

c) Code

- **Flow chính:**



IteratedBestResponse: hàm tạo 1 cấu hình IteratedBestResponse (như phần mô hình hoá tính toán ở trên). Lấy vào 1 SimpleGame cũng đã có đề cập ở phần mô hình hoá tính toán, và số lần lặp k_max. Chính sách ban đầu đó chính 1 danh sách các SimpleGamePolicy trong đó mỗi SimpleGamePolicy chứa 1 dictionary chứa các cặp (action; prob) đối với mọi action trong

từng joint action trong joint action space của từng agent. Prob của các action sẽ bằng nhau và bằng 1.0/số action của joint action hiện tại.

Solve: Hàm gọi hàm best_response cho 2 agent, để 2 kết quả vào 1 list, rồi sau đó dùng lại list đó để tính best_response cho lần lặp sau, cứ làm vậy k_max lần, và sau cùng sẽ trả về list kết quả sau k_max lần lặp đó.

Best_response: tính best response cho joint policy π hiện tại khi đang là lượt của agent có thứ tự i.

- **Hàm hỗ trợ:**

```
utility(𝒫::SimpleGame, π, i)
```

Để tính toán lợi ích của việc thực hiện joint policy π trong simple game $\mathcal{P}$ khi đang là lượt của agent có thứ tự i.

```
SimpleGamePolicy(p::Dict)
```

Tham số truyền vào chính là 1 từ điển chứa các cặp (action; prob), hàm này nó chỉ có tác dụng là với mỗi cặp đó, nó tính phần trăm prob trong tổng số tất cả prob trong tất cả cặp, rồi gán lại số phần trăm vào trong prob của mỗi cặp.

```
joint(X)
```

Hàm này chỉ dùng hàm Iterators.product để phân phối mọi mảng trong X với nhau.

```
joint(π, πi, i)
```

Hàm này thay thế policy vị trí i trong joint policy π bằng policy mới πi.

3.2.2 Hierarchical Softmax

a) Softmax response

Ngoài Best response thì ta cũng có thể sử dụng Softmax response để mô hình hoá cách mà một agent sẽ chọn hành động của họ. Cách suy nghĩ của con người hoàn toàn không giống những cỗ máy tối ưu - luôn mang về lợi ích cao nhất. Cơ chế cơ bản của Softmax response nói rằng con người sẽ càng dễ mắc lỗi hơn khi những lỗi đó càng không nghiêm trọng. Chon một tham số chính xác (precision parameter) $\lambda \geq 0$, mô hình này lựa chọn hành động a^i dựa theo:

$$\pi^i(a^i) \propto \exp\left(\lambda U^i(a^i, \pi^{-1})\right)$$

Khi λ tiến về 0, agent sẽ ít nhạy cảm với sự khác biệt về lợi ích, cho nên mọi lựa chọn họ đều chọn ngẫu nhiên. Còn khi λ tiến về vô cực, chính sách này sẽ trở về deterministic best response. Chúng ta có thể xem λ như một biến mà ta có thể học từ dữ liệu, cách tiếp cận theo cách học như vậy sẽ dẫn tới việc hành vi là những dự đoán thay vì là những hành vi có căn cứ. Nhưng việc có một mô hình dự đoán hành vi như vậy sẽ rất có ích trong việc xây dựng 1 mô hình hành vi có căn cứ.

b) Hierarchical Softmax

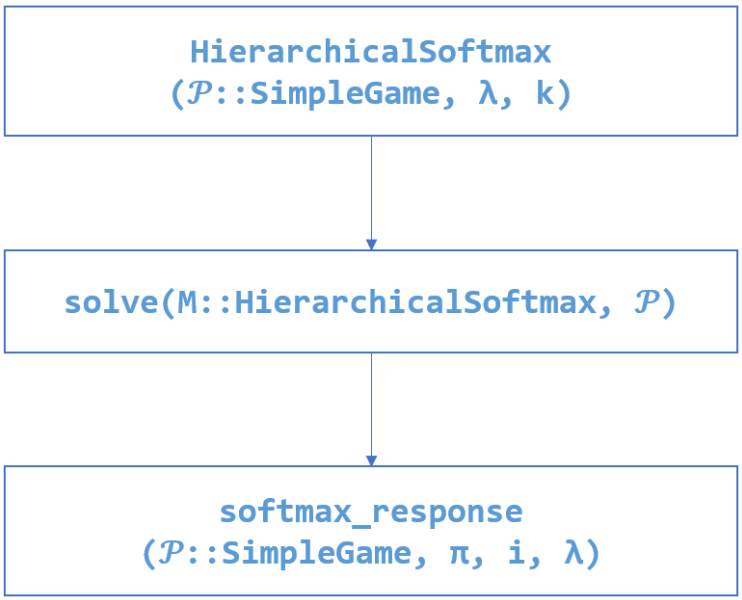
Khi ta muốn xây dựng một hệ thống thực hiện lựa chọn mà phải tương tác với con người, thì việc tính toán Nash equilibrium luôn luôn không phải là tốt nhất, vì con người thường sẽ không dùng những chiến thuật tiến tới Nash equilibrium. Vì lí do đầu tiên là trong 1 trò chơi có thể có nhiều trạng thái equilibria khác nhau nên người chơi sẽ không biết nên chọn equilibrium nào để thích ứng. Với những game chỉ có 1 equilibrium thì con người cũng sẽ không thể nào tính ra, đơn giản vì độ lớn của game và sự giới hạn của bộ não con người. Cho dù họ nghĩ họ có khả năng tính ra được Nash equilibrium thì họ lại nghĩ đối phương của họ sẽ không tính ra, cho nên việc chơi tối ưu như vậy là một điều tốn sức.

Hierarchial Softmax là một hệ thống rất tốt cho loại hệ thống nói trên, nó kết hợp phương thức tuần tự như IBR với softmax response. Phương thức Hierarchical softmax này mô hình hoá chiều sâu của sự hợp lí (depth of rationality) của một agent bằng 1 biến level k >= 0.

Một agent level 0 chọn hành động của nó 1 cách ngẫu nhiên, một agent level 1 thì sẽ giả sử đối phương dùng chiến thuật level 0 và sẽ chọn hành động theo một softmax response với độ chính xác λ (không đổi trong suốt quá trình chạy tuần tự) để đáp lại chiến thuật level 0 đó. Một cách tổng quát thì một agent level k sẽ chọn hành động theo softmax response khi đối phương của họ là level k-1.

c) Code

- **Flow chính:**



`HierarchicalSoftmax(𝒫::SimpleGame, λ, k):`

Hàm tạo 1 cấu hình HierarchicalSoftmax (cấu trúc như phần mô hình hoá tính toán ở trên), lấy vào 1 SimpleGame, tham số chính xác, và số level k mà mình sẽ lặp (số lần lặp).

`solve(M::HierarchicalSoftmax, 𝒫):`

Hàm dường như có các bước thực hiện như IBR, vì cả 2 phương pháp này đều là phương pháp tuần tự, chỉ là thay vì gọi best_response thì sẽ gọi softmax_response cho mỗi joint policy mới mỗi lần lặp.

`softmax_response(𝒫::SimpleGame, π, i, λ):`

Tính softmax response cho joint policy π hiện tại khi đang là lượt của agent có thứ tự I, với độ chính xác λ.

- **Hàm hỗ trợ: sử dụng lại những hàm hỗ trợ được sử dụng ở IBR.**

4. Kết quả

Phân tích kết quả:

a) Iterated Best Response

Kết quả thử nghiệm của bài toán sẽ là giá trị trả về của hàm solve, là một joint policy, như trên đã nói, là 1 list các SimpleGamePolicy, mỗi SimpleGamePolicy chứa biến p chính

là từ điển chứa các cặp (action; prob). Sau khi chạy xong thuật toán với k_max = 100, giá trị trả về là 1 list chứa 2 SimpleGamePolicy, giá trị của biến p của mỗi SimpleGamePolicy đều là một từ điển có 1 phần tử có key là 2 và value là 1.0, tức là action = 2, prob = 1.0 (khi print ra có dạng Dict(2 => 1.0) Dict(2 => 1.0)), và đó chính là Nash equilibrium mà ta đã dự đoán.

Ý nghĩa: Iterated Best Response là một phương pháp hiệu quả, tối ưu và đáng tin cậy để tính được Nash equilibrium và đồng thời dễ cài đặt.

b) Hierarchical Softmax

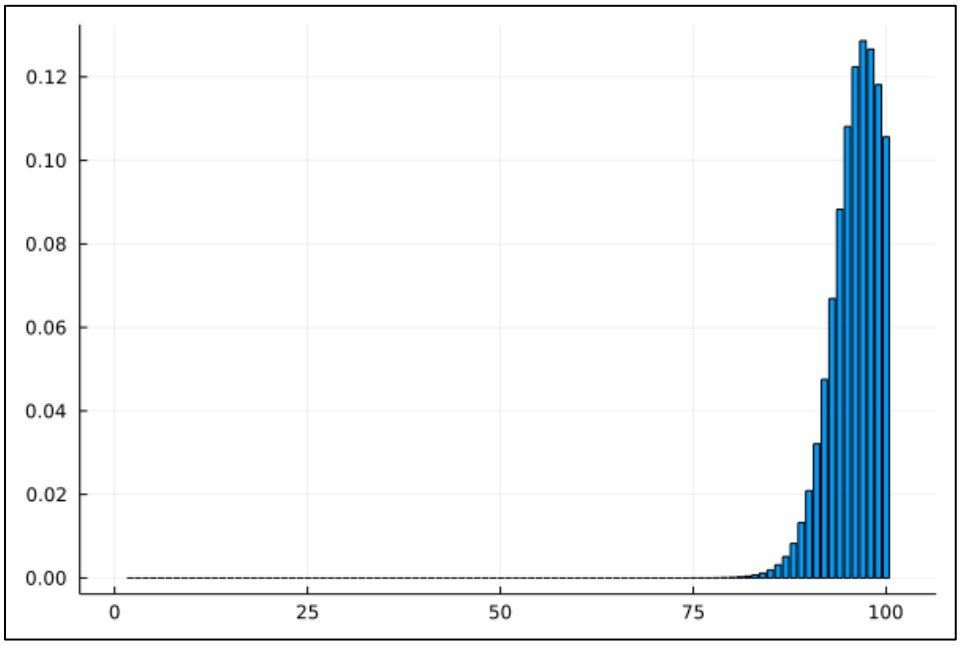
Tham số chính xác đó được ta chọn là 0.5 vì kết quả nó đưa ra giống với mong đợi của ta nhất (là lựa chọn tập trung ở 95 trở lên).

Kết quả thử nghiệm của bài toán cũng sẽ là giá trị trả về của hàm solve, cũng là một joint policy, là 1 list các SimpleGamePolicy, mỗi SimpleGamePolicy chứa biến p chính là từ điển chứa các cặp (action; prob). Sau khi chạy xong thuật toán với λ = 0.5, k = 10, giá trị trả về là 1 list chứa 2 SimpleGamePolicy tương đồng nhau, giá trị của biến p của mỗi SimpleGamePolicy đều là một từ điển có 99 phần tử, khi print ra các phần tử của từ điển đó của SimpleGamePolicy thứ nhất, ta có:

2 : 1.8290008908057377e-21
3 : 3.015512672800902e-21
4 : 4.971739885712644e-21
5 : 8.197013301962656e-21
6 : 1.3514590187157728e-20
7 : 2.2281792306362172e-20
8 : 3.673646492482178e-20
9 : 6.056819113188285e-20
10 : 9.986006504696612e-20
11 : 1.6464141333643143e-19
12 : 2.7144780020590605e-19
13 : 4.475417620842363e-19
14 : 7.378716226748965e-19
15 : 1.216544639350121e-18
16 : 2.005743023652760 3e-18
17 : 3.3069111866546966e-18
18 : 5.4521748137538e-18
19 : 8.9891165870114e-18
20 : 1.4820547721809034e-17
21 : 2.4434952272372878e-17
22 : 4.028642556000359e-17
23 : 6.642108674125526e-17
24 : 1.0950985853332579e-16
25 : 1.8055123311525616e-16
26 : 2.976786584882597e-16
27 : 4.907891360830721e-16
28 : 8.091744880886994e-16
29 : 1.3341031902197235e-15
30 : 2.199564307024146e-15
31 : 3.626468459263477e-15
32 : 5.9790356863107305e-15
33 : 9.857763314295432e-15
34 : 1.625270405780575e-14
35 : 2.6796178886497197e-14
36 : 4.4179430103649365e-14
37 : 7.283956613928514e-14
38 : 1.2009214204237882e-13
39 : 1.9799846902913018e-13
40 : 3.264442874541648e-13
41 : 5.382156404236299e-13
42 : 8.873675745882943e-13
43 : 1.4630217951488644e-12
44 : 2.412115153147744e-12
45 : 3.976905560339961e-12
46 : 6.556808788808934 5e-12
47 : 1.0810350117780378e-11
48 : 1.7823254182241037e-11
49 : 2.938557828156433e-11
50 : 4.8448627959768935e-11
51 : 7.987828344027057e-11
52 : 1.3169702493900404e-10
53 : 2.1713168620704838e-10
54 : 3.5798962932666043e-10
55 : 5.90225115838168e-10

56 : 9.731167010191914e-10
57 : 1.6043981985034359e-9
58 : 2.645205422028757e-9
59 : 4.361206405209267e-9
60 : 7.1904136589165336e-9
61 : 1.1854987653010073e-8
62 : 1.9545569514813027e-8
63 : 3.222519405253193e-8
64 : 5.313035703034358e-8
65 : 8.759713383619207e-8
66 : 1.4442321453408088e-7
67 : 2.3811350815434896e-7
68 : 3.925824859816133e-7
69 : 6.472582259639021e-7
70 : 1.0671460421185993e-6
71 : 1.7594199562340443e-6
72 : 2.9007756483383676e-6
73 : 4.782523058983758e-6
74 : 7.884918504918905e-6
75 : 1.2999682239840421e-5
76 : 2.1431899609680266e-5
77 : 3.533263849193083e-5
78 : 5.824663284775086e-5
79 : 9.601333218232245e-5
80 : 0.00015824724725372054
81 : 0.0002607644335682796
82 : 0.00042954464934018646
83 : 0.0007071600191436851
84 : 0.0011630947678960047
85 : 0.0019100130901855202
86 : 0.003128607447763954
87 : 0.005103447339925789
88 : 0.00826920041120103
89 : 0.013256029860947326
90 : 0.02089691869915554
91 : 0.03211305066565022
92 : 0.04755170974250347
93 : 0.06692074604726286
94 : 0.0882950984900803
95 : 0.10811653132611773
96 : 0.12242106735293848
97 : 0.12865448100069946
98 : 0.1266419725577256
99 : 0.11813095698074508
100 : 0.10562729389624304

Khi vẽ ra biểu đồ:



Ý nghĩa: Ta thấy rằng Hierarchical Softmax đã dự đoán đúng hành động thực tế của con người, là tập trung chọn những giá tiền cao và hầu như không chọn những giá tiền thấp – những lựa chọn gần Nash equilibrium. Suy ra Hierarchical là một trong những phương pháp hiệu quả và rất tốt cho mô phỏng sự "bất hợp lí" của hành vi con người, và đồng thời dễ cài đặt.

Điểm mạnh:

- Khả năng tự học cao, sớm lĩnh hội được cú pháp của Julia.
- Hiểu được ý tưởng của bài toán Traveler's Dilemma khá nhanh, tìm được nhiều bài báo khoa học liên quan.
- Tìm ra và phân loại được policy nào cho hành vi nào.
- Phân code ra thành nhiều file để tiện quản lý.

Điểm yếu:

- Bước đầu gặp nhiều khó khăn trong việc tìm được mã nguồn để tham khảo.
- Mã nguồn tham khảo, dù biết cú pháp Julia nhưng có những hàm thuộc những thư viện của Julia thì phải tìm hiểu thêm, cộng với việc tìm hiểu cách hệ thống Pkg hoạt động.

IV – Predator-Prey Hex World

1. Phát biểu bài toán

- Predator prey hex world là bài toán mà trong đó sử dụng hex world dynamics để chứa nhiều agents bao gồm "kẻ săn mồi" (predator) và "con mồi" (prey). Khi đó, predator cố gắng để bắt được các con mồi nhanh nhất có thể và con mồi phải chạy khỏi các predator càng lâu càng tốt.

- Một hoặc nhiều predator có thể bắt được nhiều con mồi cùng lúc nếu điều đó xảy ra trong một hex cell và phần thưởng sẽ được chia đều nhau cho mỗi predator.

2. Thách thức

- Gặp nhiều khó khăn khi code vì thiếu nguồn tài liệu để tham khảo về cách thực thi và luồng chạy của bài toán này.
- Việc mô phỏng các predator và prey được thực hiện dựa trên mô hình học máy phức tạp, dẫn đến gặp nhiều khó khăn trong lúc cài đặt và thể hiện kết quả.

3. Thực nghiệm

3.1 MÔ HÌNH TÍNH TOÁN

a) Markov Game

Markov Game là một phiên bản mở rộng hơn của Simple Game với một trạng thái chia sẻ $s \in S$. Khả năng chuyển đổi từ trạng thái s đến trạng thái s' dưới một joint action a được cung cấp bởi hàm phân phối chuyển tiếp $T(s' \mid s, \mathbf{a})$. Mỗi agent i nhận được một phần thưởng dựa trên một hàm phần thưởng riêng biệt với mỗi agent $Ri(s, \mathbf{a})$, cũng phụ thuộc vào trạng thái s. Cấu trúc của một Markov Game được minh họa với đoạn mã nguồn sau:

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

```
end
```

b) Predator Prey Hex World

- Predator Prey Hex World có trạng thái khởi tạo như hình bên dưới với mũi tên màu đỏ là predator và mũi tên màu xanh là prey. Hướng của các mũi tên chỉ ra các hành động tiềm năng được thực hiện bởi chủ thể của nó. Predator Prey Hex World không có trạng thái kết thúc.

- Ta có một tập các predator $\mathcal{I}_{pred}$ và một tập các prey $\mathcal{I}_{prey}$, với $\mathcal{I} = \mathcal{I}_{pred} \cup \mathcal{I}_{prey}$. Các trạng thái bao gồm các vị trí của mỗi agent: $S = S^1 \times ... \times S^{|\mathcal{I}|}$, với mỗi S^i là một vị trí hex tương ứng. Không gian joint action là $A = A^1 \times ... \times A^{|\mathcal{I}|}$, với mỗi A^i chứa tất cả các hướng chuyển động trong hex.
- Nếu một predator $i \in \mathcal{I}_{pred}$ và $j \in \mathcal{I}_{prey}$ ở cùng một hex với $s_i = s_j$ thì prey sẽ bị predator ăn và sau đó prey j sẽ được chuyển đến một hex ngẫu nhiên, được thể hiện như là những thế hệ sau của prey đã bị ăn thịt. Mặt khác, các trạng thái chuyển đổi là độc lập và được mô tả trong Original Hex World (tham khảo bài Hex World)
- Một hoặc nhiều predator có thể bắt một hoặc nhiều prey nếu điều đó cùng xảy ra trong một hex. Giả sử ta có *n* predator và *m* prey ở cùng một ô thì, mỗi predator nhận được phần thưởng là *m / n*. Ví dụ: nếu 5 predator bắt được 3 prey, thì phần thưởng cho mỗi predator sẽ là 3/5 prey.
- Khi một predator hoặc prey di chuyển, nó sẽ bị -1 điểm. Predator sẽ được 10 điểm nếu bắt được prey và prey sẽ bị trừ 100 điểm.

3.2 PHƯƠNG PHÁP GIẢI QUYẾT

a) Markov Game's Policy

- Joint policy π trong Markov Game định nghĩa một khả năng phân phối xác suất dựa trên các joint action cho trạng thái hiện tại. Cùng với MDPs, ta sẽ tập trung trên các policy phụ thuộc vào các trạng thái hiện tại hơn các trạng thái quá khứ vì những trạng thái ở tương lai và các phần thưởng đều có điều kiện độc lập với lịch sử so với trạng thái hiện tại
- Ta cũng sẽ chú ý đến các stationary policy (không phụ thuộc vào thời gian). Khả năng agent i chọn một action a ở trạng thái s được biểu diễn bởi $\pi^i(a \mid s)$. Hàm $\pi(s)$ thường được dùng để biểu diễn sự phân bố trên những joint action.
- Lợi ích của một joint policy π từ quan điểm của agent i có thể được tính toán bằng cách sử dụng một biến thể của policy evaluation. Phần thưởng cho agent i từ trạng thái s khi tuân theo joint policy π là:

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

- Xác suất chuyển trạng thái từ s sang s' khi tuân theo π là:

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

- Trong một infinite-horizon discounted game, lợi ích dành cho agent i từ trạng thái s là:

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

- Đoạn code bên dưới mô tả một cách cài đặt của Markov Game's Policy:

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

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

```
(πi::SimpleGamePolicy)(s, ai) = πi(ai)

probability(𝒫::MG, s, π, a) = prod(πj(s, aj) for (πj, aj) in zip(π, a))

# Phần thưởng cho agent i từ trạng thái s khi tuân theo joint policy π
reward(𝒫::MG, s, π, i) =
  sum(𝒫.R(s,a)[i]*probability(𝒫,s,π,a) for a in joint(𝒫.𝒜))

# Xác suất chuyển trạng thái từ s sang s’ khi tuân theo π
transition(𝒫::MG, s, π, s′) =
  sum(𝒫.T(s,a,s′)*probability(𝒫,s,π,a) for a in joint(𝒫.𝒜))

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

b) Response Model

- Ta có thể tổng quát hóa các Response Model thành các Markov Game, với yêu cầu phải tính toán đến các mô hình chuyển đổi trạng thái

i. Best Response

- Một response policy cho agent *i* là một policy π^i mà tối đa hóa được lợi ích mong muốn và đưa ra các fixed policy cho những agent π^{-i} khác. Nếu các policy của những agent khác đều cố định, thì vấn đề trên trở thành một MDP. MDP này có không gian trạng thái S và không gian hành động A^i. Ta có thể định nghĩa các hàm chuyển đổi và hàm phần thưởng như sau:

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

- Bởi vì đây là best response cho agent *i* nên MDP chỉ dùng reward R^i. Việc giải MDP này cho kết quả là một best response policy cho agent *i*. Đoạn code bên dưới mô tả cách cài đặt best response:

ii. Softmax Response

- Ta có thể định nghĩa một softmax response policy bằng cách gán một stochastic response cho các policy của những agent khác tại mỗi trạng thái. Tương tự việc xây dựng nên một deterministic best response policy, ta giải một MDP mà trong đó các agent với các fixed policy π^{-i} được xếp vào môi trường. Tiếp theo, ta trích xuất hàm giá trị hành động $Q(s, a)$ bằng cách sử dụng one-step lookahead. Ta có softmax response với tham số precision $\lambda \geq 0$ như sau

```
function best_response(𝒫::MG, π, i)
    𝒮, 𝒜, R, T, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.T, 𝒫.γ

    # Hàm chuyển đổi trạng thái
    T'(s,ai,s') = transition(𝒫, s, joint(π, SimpleGamePolicy(ai), i), s')

    # Hàm phần thưởng
    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
```

$$\pi^i(a^i \mid s) \propto \exp(\lambda Q(s, a^i))$$

- Đoạn code bên dưới mô tả cách cài đặt Softmax Response:

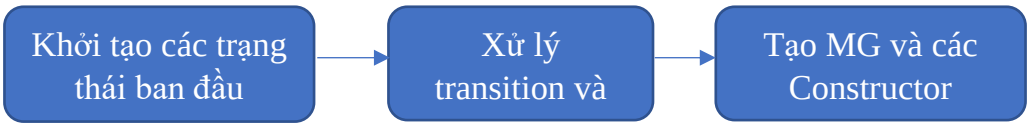
```
function softmax_response(𝒫::MG, π, i, λ)
    𝒮, 𝒜, R, T, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.T, 𝒫.γ
    T'(s,ai,s') = transition(𝒫, s, joint(π, SimpleGamePolicy(ai), i), s')
    R'(s,ai) = reward(𝒫, s, joint(π, SimpleGamePolicy(ai), i), i)
    mdp = MDP(γ, 𝒮, joint(𝒜), T', R')
    πi = solve(mdp)
    Q(s,a) = lookahead(mdp, πi.U, s, a)
    p(s) = SimpleGamePolicy(a => exp(λ*Q(s,a)) for a in 𝒜[i])
```

```
    return MGPolicy(s => p(s) for s in 𝒮)
  end
```

c) *Predator Prey Hex World (cài đặt thuật toán và kiểm thử)*

Cài đặt Predator Prey Hex World

- Work flow chính của chương trình:



- Khởi tạo các trạng thái ban đầu của trò chơi:

```
# Khởi tạo số agent là 2
n_agents(mg::PredatorPreyHexWorldMG) = 2
```

- Ta có một tập các predator $\mathcal{I}_{pred}$ và một tập các prey $\mathcal{I}_{prey}$, với $\mathcal{I} = \mathcal{I}_{pred} \cup \mathcal{I}_{prey}$.
- Cài đặt các trạng thái các vị trí của mỗi agent: $S = S^1 \times ... \times S^{|\mathcal{I}|}$

```
# Đánh số thứ tự của các state
ordered_states(mg::PredatorPreyHexWorldMG, i::Int) =
vec(collect(1:length(mg.hexes)))

# Gán mỗi state với một agent tương ứng
ordered_states(mg::PredatorPreyHexWorldMG) =
vec(collect(Iterators.product([ordered_states(mg, i) for i in
1:n_agents(mg)]...)))
```

- Cài đặt không gian joint action là $A = A^1 \times ... \times A^{|\mathcal{I}|}$

```
# Đánh số thứ tự cho các action
ordered_actions(mg::PredatorPreyHexWorldMG, i::Int)
vec(collect(1:n_actions(mg.hexWorldDiscreteMDP)))

# Đánh số thứ tự cho các joint action
ordered_joint_actions(mg::PredatorPreyHexWorldMG) =
vec(collect(Iterators.product([ordered_actions(mg, i) for i in
1:n_agents(mg)]...)))
```

- Cài đặt hàm xử lý khi một predator $i \in \mathcal{I}_{pred}$ và $j \in \mathcal{I}_{prey}$ ở cùng một hex với $s_i = s_j$:

```
function transition(mg::PredatorPreyHexWorldMG, s, a, s’)
    # Khi một prey bị bắt, một prey mới sẽ được tạo ra ở một vị trí ngẫu
nhiên và predator sẽ vẫn đứng yên.
    if s[1] == s[2]
        prob = Float64(s’[1] == s[1]) / length(mg.hexes)
    else
        # Ngược lại, predator và prey sẽ tuân theo theo HexWorldMDP.
        prob = mg.hexWorldDiscreteMDP.T[s[1], a[1], s’[1]]
mg.hexWorldDiscreteMDP.T[s[2], a[2], s’[2]]
    end
    return prob
end
```

- Cài đặt hàm reward: khi một predator hoặc prey di chuyển, nó sẽ bị -1 điểm. Predator sẽ được 10 điểm nếu bắt được prey và prey sẽ bị trừ 100 điểm.

```
function reward(mg::PredatorPreyHexWorldMG, i::Int, s, a)
    r = 0.0

    if i == 1
        # Predator bị -1 điểm khi di chuyển và được 10 điểm khi bắt được con
mồi
        if s[1] == s[2]
            return 10.0
        else
            return -1.0
        end
    elseif i == 2
        # Prey bị -1 điểm khi di chuyển và -100 điểm khi bị bắt
        if s[1] == s[2]
            r = -100.0
        else
            r = -1.0
        end
    end

    return r
end
```

- Cài đặt các hàm khởi tạo cho MG và Predator Prey Hex World:

```
# Khởi tạo một MG theo các tham số của một PredatorPreyHexWorld
function MG(mg::PredatorPreyHexWorldMG)
    return MG(
        mg.hexWorldDiscreteMDP.γ,
        vec(collect(1:n_agents(mg))),
        ordered_states(mg),
        [ordered_actions(mg, i) for i in 1:n_agents(mg)],
        (s, a, s′) -> transition(mg, s, a, s′),
        (s, a) -> joint_reward(mg, s, a)
    )
end

# Khởi tạo một PredatorPreyHexWorld theo MG
function PredatorPreyHexWorldMG(hexes::Vector{Tuple{Int,Int}},
                                r_bump_border::Float64,
                                p_intended::Float64,
                                γ::Float64)
    hexWorld = HexWorldMDP(hexes,
                           r_bump_border,
                           p_intended,
                           Dict{Tuple{Int64,Int64},Float64}(),
                           γ)
    mdp = hexWorld.mdp
    return PredatorPreyHexWorldMG(hexes, mdp)
end
```

Kiểm thử

Ta có chương trình kiểm thử thuật toán đã cài đặt như sau:

```
@testset "predator_prey.jl" begin
    m = PredatorPreyHexWorld()
    hexes = m.hexes
    @test p.n_agents(m) == 2
    @test length(p.ordered_states(m, rand(1:2))) == length(hexes) &&
length(p.ordered_states(m)) == length(hexes)^2
    @test length(p.ordered_actions(m, rand(1:2))) == 6 &&
length(p.ordered_joint_actions(m)) == 36
    @test p.n_actions(m, rand(1:2)) == 6 && p.n_joint_actions(m) == 36

    @test 0.0 <= p.transition(m, rand(p.ordered_states(m)),
rand(p.ordered_joint_actions(m)), rand(p.ordered_states(m))) <= 1.0
```

```
    @test -1.0 <= p.reward(m, rand(1:2), rand(p.ordered_states(m)),
rand(p.ordered_joint_actions(m))) <= 10.0
    @test [-1.0, -1.0] <= p.joint_reward(m, rand(p.ordered_states(m)),
rand(p.ordered_joint_actions(m))) <= [10.0, 10.0]
    mg = p.MG(m)
end
```

4. Kết quả

Phân tích kết quả:

Test Summary:	Pass	Total
predator_prey.jl	7	7

Chương trình đã pass tất cả các test case mong muốn.

Điểm mạnh

- Các kết quả trả về gần như mong đợi.
- Tiếp cận được lý thuyết trò chơi Predator Prey Hex World và các phương pháp giải.

Điểm yếu

- Tiếp xúc với nhiều khái niệm mới liên quan đến lý thuyết trò chơi, gặp khó khăn trong việc cài đặt các thuật toán và các cấu trúc dữ liệu liên quan.

V – Multi-Caregiver Crying Baby

1. Phát biểu bài toán

1.1 SƠ LƯỢC VỀ BÀI TOÁN CRYING BABY

Bài toán "The crying baby" là một dạng POMDP đơn giản với 2 states, 3 actions và 2 observations. Mục đích của bài toán là làm thế nào để chăm sóc một em bé tốt, bằng cách là lựa chọn thời gian nào thích hợp để cho ăn (feed), hát (sing) hay có thể ngó lơ (ignore) em bé. 2 state gồm: hungry và stated. 3 action gồm: feed, sing và ignore. 2 observations gồm: crying và quiet.

TRANSITIONS: Từ một state s, thực hiện một action a => tỷ lệ chuyển tiếp sang state s'

Hàm transision có các thỏa mãn sau:

Hungry + feed => stated 100%

Hungry + sing => hungry 100%

Hungry + ignore => hungry 100%

Stated + feed => stated 100%

Stated + sing => hungry 10%

Stated + ignore => hungry 10%

OBSERVATIONS: Từ một state s, thực hiện một action a=> tỷ lệ khả năng xuất hiện observation o tương ứng

Mối liên hệ giữa trạng thái, action và trạng thái quan sát được

Hungry + feed => cry 80%

Hungry + sing => cry 90%

Hungry + ignore => cry 90%

Stated + feed => cry 10%

Stated + sing => cry 0%

Stated + ignore => cry 10%

Lưu ý: cry + quiet = 100% =>

Cry = 10% ⇔ quiet = 90%

REWARD FUNCTION: Khi thực hiện một action a, hoặc ứng với một observation o sẽ có phần thưởng cho người giữ trẻ

Hungry độc lập với các actions khác => -10, Feed => -5, Sing => -0.5

1.2 BÀI TOÁN MULTICARE GIVER CRYING BABY

Bài toán này thuộc POMGs - Partially Observable Markov Games – có thể được xem là phần mở rộng của POMDPs với nhiều agent nên đây là bài toán mở rộng của Crying Baby ở phần 1. Chúng ta có 2 người giữ trẻ (caregiver). Tương tự như bài toán Crying baby, có 2 state, 3 action và 2 observation. Mỗi action của caregiver đều tốn chi phí. Nếu cả hai người cùng thực hiện chung 1 action thì chi phí sẽ giảm một nửa. (Ví dụ, nếu cả 2 cùng cho bé ăn thì reward -2.5 thay vì -5).

Người giữ trẻ không hoàn toàn quan sát tình trạng của em bé, mà dựa vào sự ồn ào (đứa trẻ khóc). Cả hai người đều quan sát như nhau. Vì do cơ cấu phần thưởng, có sự đánh đổi giữa việc giúp đỡ lẫn nhau và việc tham lam lựa chọn một action ít tốn kém hơn. **Mục đích cả 2 caregiver là làm sao để có được reward lớn nhất.**

Cụ thể, mỗi caregiver I trong tập I = {1, 2} có states, actions và observations tương tự phía trên.

States	HUNGRY, STATED
Actions	FEED, SING, IGNORE
Observations	CRYING, QUIET

TRANSITION

Tương tự như bài toán crying baby, ngoại trừ mỗi người caregiver lựa chọn action của riêng mình

State	Action of caregiver 1	Action of caregiver 2	Transition state	Probability
HUNGRY	FEED	*	SATED	100%

*: tất cả action gôm FEED/ SING/ IGNORE

Có thể đổi chỗ action của caregiver 1 và 2 mà không ảnh hưởng kết quả

Nếu action không phải feed

State	Action of caregiver 1	Action of caregiver 2	Transition state	Probability
HUNGRY	SING/ IGNORE	SING/ IGNORE	HUNGRY	100%
STATED	SING/ IGNORE	SING/ IGNORE	STATED	50%

OBSERVATIONS

Các observations tương tự như bài toán crying baby, tuy nhiên cả 2 caregiver sẽ quan sát các observation của baby là giống nhau nhưng không nhất thiết phải lựa chọn action giống người còn lại

State	Action of caregiver 1	Action of caregiver 2	Observation	Probability
HUNGRY	SING	*	CRYING	90%
HUNGRY	SING	*	QUIET	10%
STATED	SING	*	CRYING	0%

*: tất cả các action

Có thể đổi chỗ action của caregiver 1 và 2 mà không ảnh hưởng kết quả

Nếu action không phải SING, thì observations như sau:

State	Action of caregiver 1	Action of caregiver 2	Observation	Probability
HUNGRY	*	*	CRYING	90%
HUNGRY	*	*	QUIET	10%
STATED	SING	*	CRYING	0%
STATED	*	*	QUIET	100%

REWARD

Người thứ nhất sở trường là feed, người thứ hai sở trường là sing.

State/Action	Reward of caregiver 1	Reward of caregiver 2
FEED	-2.5	-5.0
SING	-0.5	-0.25
HUNGRY	-10	-10

2. Thách thức

- Việc lựa chọn action để hành động không thể phụ thuộc vào niềm tin (beliefs) của caregiver như trong các bài toán POMDPs nên chỉ có thể xây dựng conditional plan trong d bước (độ sâu d) từ các action.
- Bài toán không có điểm dừng cụ thể. Việc dừng lại thuật toán phụ thuộc vào giá trị d (độ sâu của conditional plan, tức các bước trong conditional plan mà chúng ta đặt ra). Điều này đặt thêm một vấn đề là nên chọn d như thế nào là tốt nhất?
- Việc tính toán bằng phương pháp Nash quilibrium là tốn kém vì có nhiều policy phụ thuộc trùng lắp lẫn nhau. Và việc cải thiến nó bằng dynamic programming đôi lúc cũng chưa chắc có thể đạt hiệu quả như mong đợi. Nếu rơi vào trường hợp xấu nhất thì chi phí cũng tương tự như việc mở rộng lại toàn bộ conditional plan.

- Vừa cài đặt các phương thức liên quan tới POMG, vừa cài đặt các phương thức liên quan tới Simple Game, dẫn đến code dài dòng, nhiều phương thức trùng tên nhau dễ nhầm lẫn.

3. Thực nghiệm

3.1 MÔ HÌNH TÍNH TOÁN

POMG là một dạng mở rộng của MGs có partial observability và mở rộng của POMDPs với nhiều agent. Có các agent $i \in I = \{1,2\}$, mỗi agent I chọn một action $a_i \in A_i$ = {feed, sing, ignore}, dựa trên phần quan sát được observation o_i. Transition từ state s tới state s' dựa trên action a T (s'| s,a). Joint reward dựa trên reward thực hiện action theo state R^i =(s,a). Mỗi agent cố gắng thực hiện sao cho đạt được reward cho họ là cao nhất.

Sau khi các agent thực hiện joint action a, thì sẽ nhận được joint observation o = (o^1 , ….. ,o^k) ∈ space O = O^1 X …. X O^k. Mỗi agent thì nhận được một observation o^i của riêng họ từ joint observation. Theo như bài toán thì mỗi agent đều sẽ suy luận về suy nghĩ của người khác và chọn những action cho riêng mình. Kết quả trả về của bài toán sẽ là một chuỗi các action, probability của mỗi agent.

Vì không giống như POMDP có thể mô hình hóa các beliefs, chúng ta sẽ xây dựng policy tập trung vào các policy không yêu cầu belief của agent để quyết định action. Sử dụng tree-base conditional plan để biểu diễn bài toán.

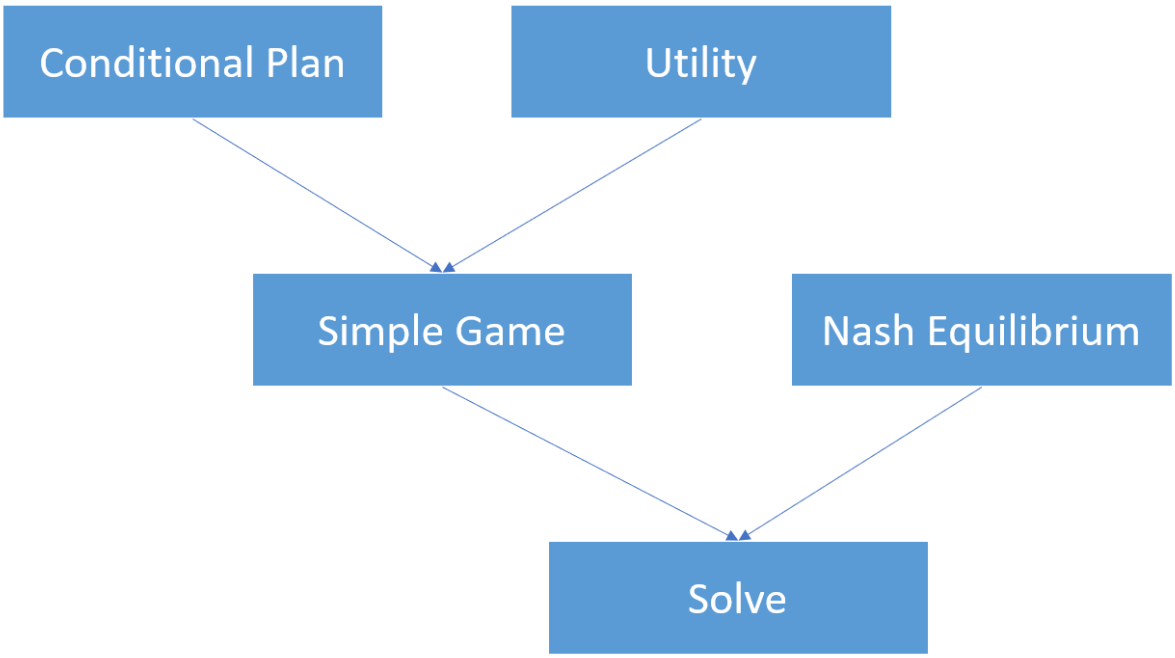
Cấu trúc POMG gồm:

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

3.2 PHƯƠNG PHÁP GIẢI QUYẾT

Ý tưởng tổng quan:

Bởi vì MDP có khả năng mở rộng một phần quan sát (partial observability) nên một Makov Game cũng có thể mở rộng thành Partially Observation Makov Game (POMG) – cũng chính là bài toán chúng ta cần giải quyết. Để giải quyết bài toán này, chúng ta có thể chuyển bài toán từ dạng POMG sang Simple game và sử dụng Nash Equilibrium.



Với mỗi agent, các joint action trong simple game tương tự như joint conditional plan trong POMGs, các reward ứng với mỗi action sẽ tương tự utility của joint conditional plan trong POMG. Từ conditional plan, kết hợp với utility, chuyển POMG về dạng Simple game.

Theo như bài toán, sẽ có 2 agent $I \in I = \{1,2\}$ với 3 action $A_i = \{a_i^1, a_i^2, a_i^3\}$ = {feed, sing, ignore} và 2 observation $O_i = \{ o_i^1, o_i^2\}$ = {crying, quiet}.

Hướng giải quyết bài toán theo các bước như sau:

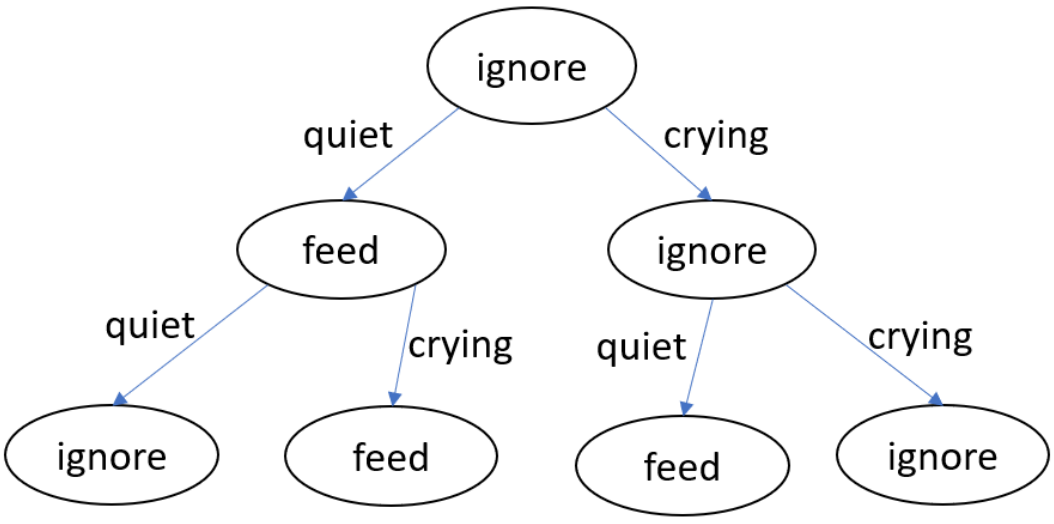
a) Xây dựng conditional plan

Với mỗi agent, chúng ta xây dựng conditional plan dựa trên action {feed, sing, ignore}. Biểu diễn conditional plan bằng tree với mỗi node là 1 action, nhánh theo sau node là các observation có thể quan sát được sau khi thực hiệc action đó.

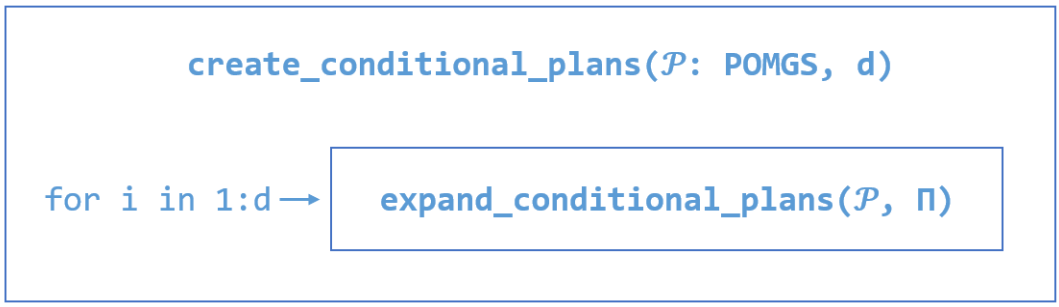
```
##code Khai báo cấu trúc 1 conditional plan
struct ConditionalPlan
    a # action được chọn là root
    subplans # dictionary để ánh xạ từ các observation thành các subplan
    end
    ConditionalPlan(a) = ConditionalPlan(a, Dict())
    (π::ConditionalPlan)() = π.a
    (π::ConditionalPlan)(o) = π.subplans[o]
```

Mỗi agent có một conditional plan, với root là một action được khởi tạo. Quá trình sẽ diễn ra như sau: từ một action root, dựa trên các observation có thể quan sát được, tiếp tục đi theo nhánh tương ứng và lựa chọn action tiếp theo là node của nhánh đó, cho tới khi đi hết tree.

Hình minh họa conditional plan tree của 1 agent có độ cao 2



Work flow



`create_conditional_plans(𝒫: POMGS, d)`

Hàm tạo conditional plan, ứng với chiều cao của Conditional Plan tree đã định trước, mở rộng dần condition plan.

`expand_conditional_plans(𝒫, Π)`

Hàm mở rộng conditional plan, Các tham số lấy từ bài toán POMG.

b) Xây dựng utility

Đầu tiên, chúng ta cần tính hàm đệ quy joint utility U^{π} theo công thức:

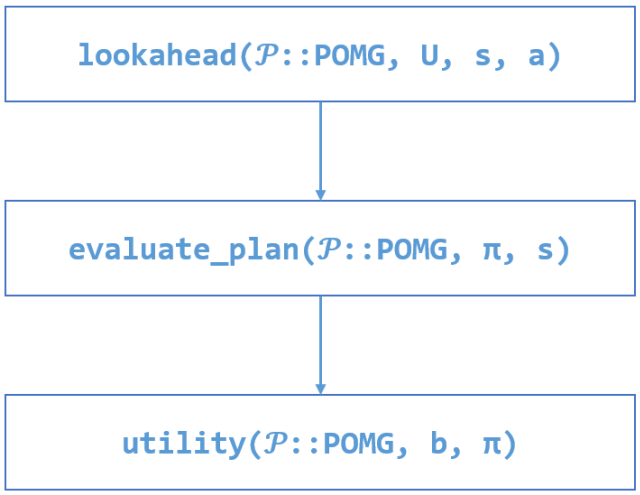
$$\mathbf{U}^{\boldsymbol{\pi}}(s) = \mathbf{R}(s, \boldsymbol{\pi}()) + \gamma \left[\sum_{s'} T(s' \mid s, \boldsymbol{\pi}()) \sum_{\mathbf{o}} O(\mathbf{o} \mid \boldsymbol{\pi}(), s') \mathbf{U}^{\boldsymbol{\pi}(\mathbf{o})}(s') \right]$$

Với:

$\pi()$: vector của các action tại gốc liên kết với π

$\pi(o)$: vector của các subplan liên kết với các agent khác nhau quan sát các thành phần của chúng trong joint observation

Work flow



`lookahead(𝒫::POMG, U, s, a):`

tính giá trị biểu thức:

$$\cdot \left[\sum_{s'} T(s' \mid s, \boldsymbol{\pi}()) \sum_{\mathbf{o}} O(\mathbf{o} \mid \boldsymbol{\pi}(), s') \mathbf{U}^{\boldsymbol{\pi}(\mathbf{o})}(s') \right]$$

`valuate_plan(𝒫::POMG, π, s):`

thực hiện tính U theo công thức:

$$\mathbf{U}^{\boldsymbol{\pi}}(s) = \mathbf{R}(s, \boldsymbol{\pi}()) + \gamma \left[\sum_{s'} T(s' \mid s, \boldsymbol{\pi}()) \sum_{\mathbf{o}} O(\mathbf{o} \mid \boldsymbol{\pi}(), s') \mathbf{U}^{\boldsymbol{\pi}(\mathbf{o})}(s') \right]$$

`utility(𝒫::POMG, b, π):`

thực hiện tính Utility, kết hợp với policy π từ phân phối trạng thái ban đầu b tính theo công thức

$$\mathbf{U}^{\boldsymbol{\pi}}(b) = \sum_{s} b(s) \mathbf{U}^{\boldsymbol{\pi}}(s)$$

c) Từ conditional plan và utility chuyển về dạng simple game và giải quyết bài toán theo nash equilibrium

Nash equilibirum của POMG là khi tất cả các agent thực hiện các action theo policy phản hồi tốt nhất tới họ (mỗi agent đều tuân theo policy). Thuật toán thực hiện d bước (đây cũng chính là độ cao của conditional plan tree).

Tuy nhiên, để có thể sử dụng các phương thức của Simple Game, cần cài đặt cấu trúc Simple Game và Policy của nó.

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

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
```

Tiến hành cài đặt Nash Equilibrium của Simple Game

```
# Thuật toán NashEquilibrium của Simple game
function solveNE(M::NashEquilibrium, 𝒫::SimpleGame)
    ℐ, 𝒜, R = tensorform(𝒫)
    model = Model(Ipopt.Optimizer)
```

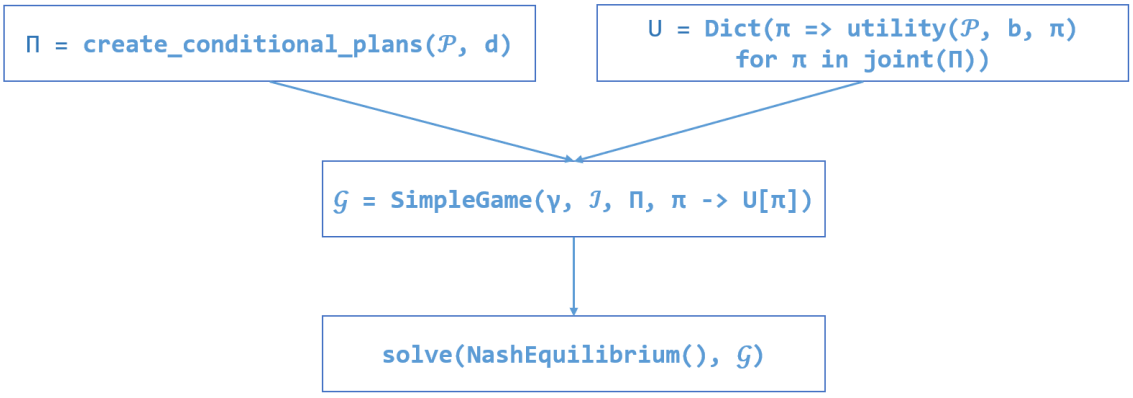
```
    @variable(model, U[ℐ])
    @variable(model, π[i=ℐ, 𝒜[i]] ≥ 0)
    @NLobjective(model, Min,
        sum(U[i] - sum(prod(π[j,a[j]] for j in ℐ) * R[y][i]
            for (y,a) in enumerate(joint(𝒜))) for i in ℐ))
    @NLconstraint(model, [i=ℐ, ai=𝒜[i]],
        U[i] ≥ sum(
            prod(j==i ? (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
```

Sau đó tiến hành giải quyết bài toán bằng cách kết hợp Conditional plan ở phần 1, Utility ở phần 2, chuyển đổi thành Simple game và sử dụng Nash Equilibrium của Simple game để giải quyết bài toán.

```
struct POMGNashEquilibrium
    b # initial belief
    d # Chiều sâu của  conditional plans
    end
```

```
solve(M::POMGNashEquilibrium, 𝒫::POMG):
```

Tạo conditional plan cho từng agent (ở bài toán này có 2 agent), tính utility. Từ conditional plan và utility, chuyển về Simple game và giải quyết bài toán theo NashEquilibrium của simple game.



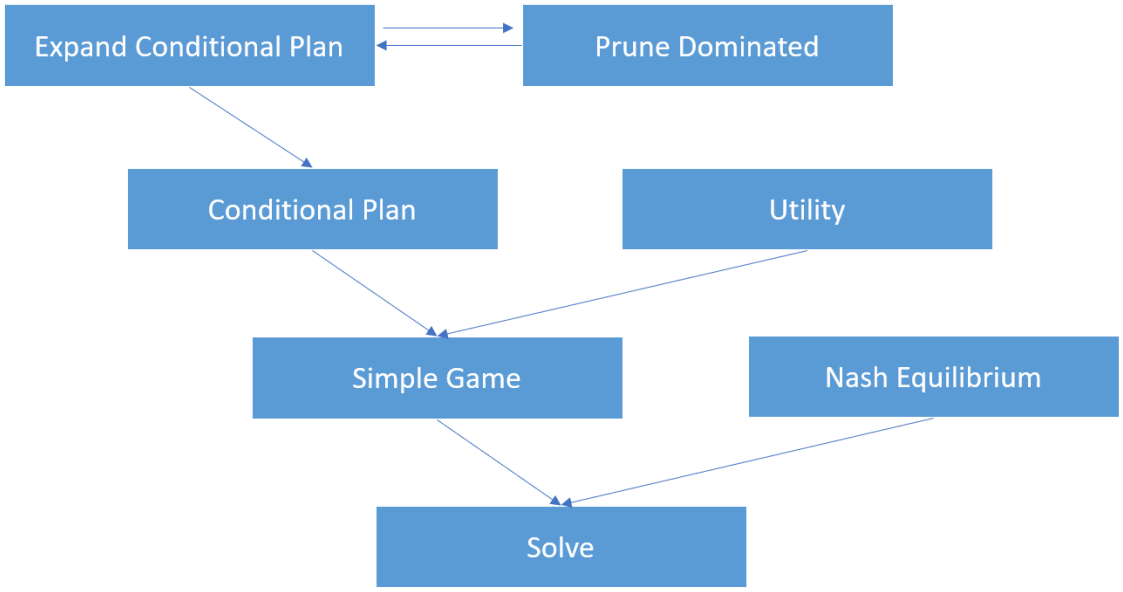
d) Tối ưu hóa: Dynamic Programming

Nếu chúng ta giải quyết bài toán theo hướng Nash equilibrium thông thường thì sẽ tốn kém về mặt tính toán. Bởi vì để xây dựng một conditional plan ứng với tất cả các action có thể, ở một số độ sâu của tree thì các action đều thỏa mãn conditional plan, dẫn đến sự dư thừa. Chúng ta sẽ điều chỉnh việc lặp các giá trị theo hướng tiếp cận POMDPs, lặp lại giữa việc mở rộng độ sâu của conditional plan và lược bỏ những plan dưới mức tối ưu.

Nhược điểm: trường hợp xấu nhất thì độ phức tạp bằng với việc mở rộng toàn bộ policy tree (conditional plan tree).

Ưu điểm: cách tiếp cận theo hướng tăng dần độ cao có thể tiết kiệm chi phí đáng kể.

Ý tưởng: Đầu tiên chúng ta xây dựng một conditional plan với độ cao là 1 (chỉ có 1 bước). Tiến hành tối ưu plan này bằng cách loại bỏ những plan mà bị chi phối bởi plan khác. Sau đó, mở rộng ra conditional plan có độ cao là 2 (có 2 bước). Lặp lại việc tối ưu này cho tới khi đạt được độ cao mong muốn. Sau khi tạo xong conditional plan thì thực hiện như cũ.



Thực hiện:

Để có thể tối ưu các plan bằng cách loại bỏ những plan bị chi phối. Ví dụ policy π^i thuộc về agent I có thể bị loại bỏ nếu tồn tại một policy $\pi^{i'}$ sao cho luôn hoạt động giống như

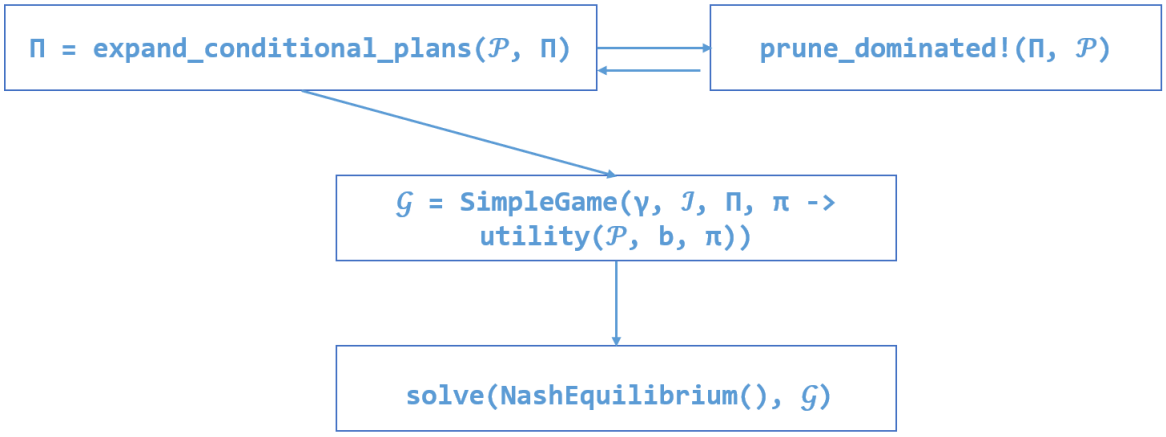
policy π^i. Kiểm tra phụ thuộc: policy π^i phụ thuộc vào policy $\pi^{i,}$ nếu không tồn tại b (π^{-i} , s) giữa các joint policy π^{-i} và state s thỏa mãn:

$$\sum_{\pi^{-i}} \sum_{s} b(\pi^{-i}, s) U^{\pi^{i\prime}, \pi^{-i}, i}(s) \geq \sum_{\pi^{-i}} \sum_{s} b(\pi^{-i}, s) U^{\pi^{i}, \pi^{-i}, i}(s)$$

$$\Leftrightarrow \quad \sum_{\pi^{-i}} \sum_{s} b(\pi^{-i}, s) \left[U^{\pi^{i\prime}, \pi^{-i}, i}(s) - U^{\pi^{i}, \pi^{-i}, i}(s) \right] \geq 0 \qquad (*)$$

b là joint distribution thông qua policy của agent còn lại và state.

```
#Cấu trúc của POMG theo Dynamic Programming
struct POMGDynamicProgramming
    b # initial belief
    d # depth of conditional plans
    end
```




```
is_dominated(𝒫::POMG, Π, i, πi):
```

Kiểm tra phụ thuộc policy π_i có phụ thuộc vào policy khác hay không theo công thức:

$$\sum_{\pi^{-i}} \sum_{s} b(\pi^{-i}, s) \left[U^{\pi^{i\prime}, \pi^{-i}, i}(s) - U^{\pi^{i}, \pi^{-i}, i}(s) \right] \geq 0$$

```
prune_dominated!(Π, 𝒫::POMG):
```

Hàm loại bỏ những policy có phụ thuộc.

```
solve(M::POMGDynamicProgramming, 𝒫::POMG):
```

Hàm Solution mới bằng cách vừa xây dựng Condition plan, vừa tối ưu Condition plan, sau đó chuyển về Simple game tương tự như phần phía trên.

4. Kết quả

Phân tích kết quả

Lần lượt chạy thử phương pháp Nash Equilibrium thông thường và khi sử dụng Dynamic Programming. Với belief khởi tạo random, độ cao của conditional plan là 2, các giá trị thuộc tính được quy định như sau:

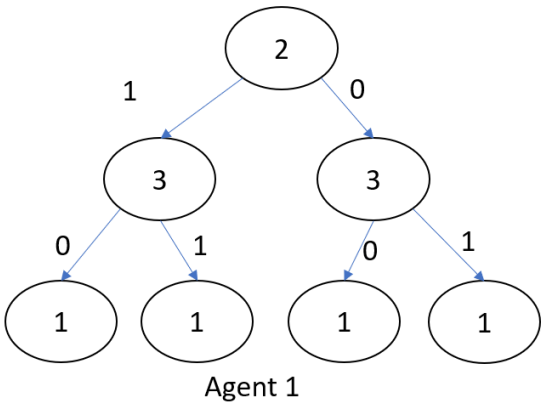
SATED = 1 HUNGRY = 2	FEED = 1 IGNORE = 2 SING = 3	CRYING = true QUIET = false

Kết quả trả về của hàm solve Nash Equilibrium:

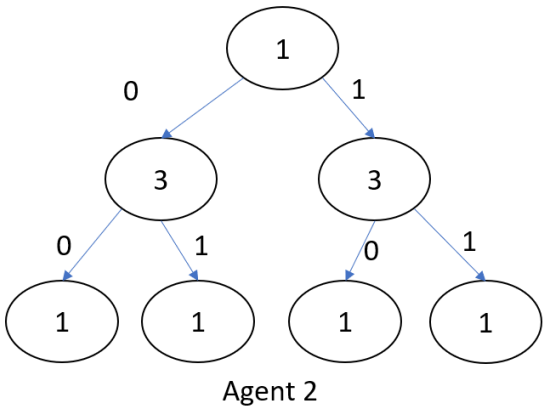
```
(ConditionalPlan(2, Dict{Bool, ConditionalPlan}(0 => ConditionalPlan(3,
Dict{Bool, ConditionalPlan}(0 => ConditionalPlan(1, Dict{Any, Any}()), 1 =>
ConditionalPlan(1, Dict{Any, Any}()))), 1 => ConditionalPlan(3, Dict{Bool,
ConditionalPlan}(0 => ConditionalPlan(1, Dict{Any, Any}()), 1 =>
ConditionalPlan(1, Dict{Any, Any}())))))),
ConditionalPlan(1, Dict{Bool, ConditionalPlan}(0 => ConditionalPlan(3,
Dict{Bool, ConditionalPlan}(0 => ConditionalPlan(1, Dict{Any, Any}()), 1 =>
ConditionalPlan(1, Dict{Any, Any}()))), 1 => ConditionalPlan(3, Dict{Bool,
ConditionalPlan}(0 => ConditionalPlan(1, Dict{Any, Any}()), 1 =>
ConditionalPlan(1, Dict{Any, Any}()))))))))
```

Kết quả trả về 1 tuple gồm 2 plan của 2 agent thỏa mãn yêu cầu.

Từ kết quả trên, minh họa qua tree plan của 2 agent như sau:

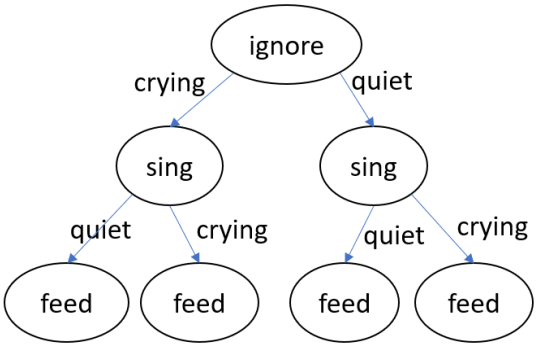


Agent 1

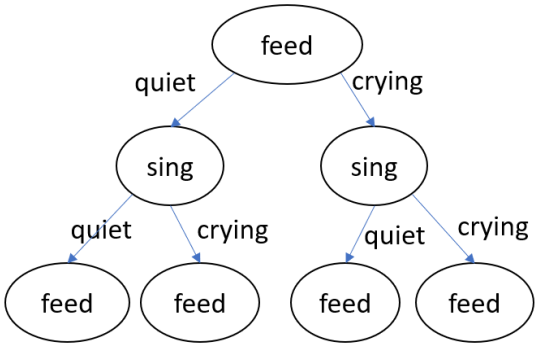


Agent 2

Tương ứng với:



Agent 1



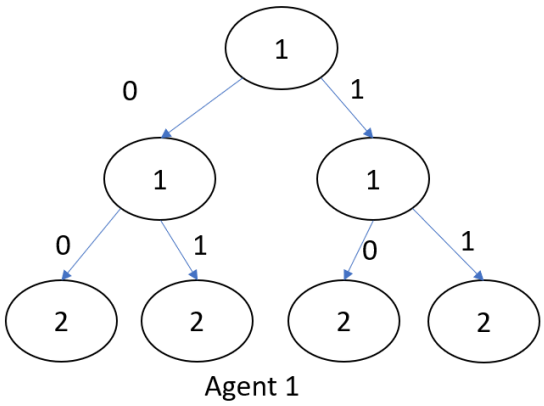
Agent 2

Vậy chuỗi hành động của agent 1: ignore => sing => feed

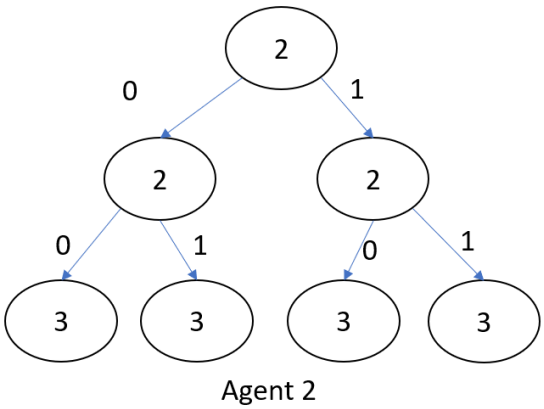
Agent 2: feed => sing => feed

Kết quả trả về của hàm solve của Nash Equilibrium kết hợp Dynamic Programming:

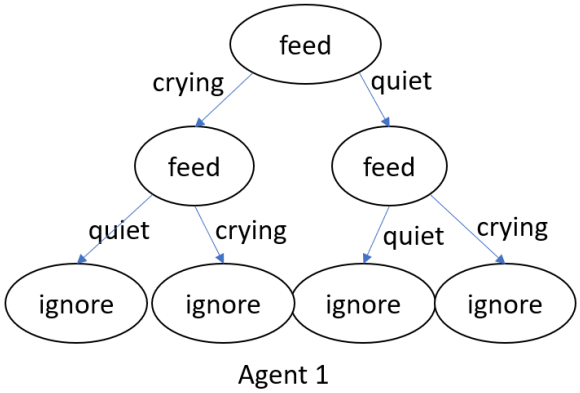
```
(ConditionalPlan(1, Dict{Bool, ConditionalPlan}(0 => ConditionalPlan(1,
Dict{Bool, ConditionalPlan}(0 => ConditionalPlan(2, Dict{Any, Any}()), 1 =>
ConditionalPlan(2, Dict{Any, Any}()))), 1 => ConditionalPlan(1, Dict{Bool,
ConditionalPlan}(0 => ConditionalPlan(2, Dict{Any, Any}()), 1 =>
ConditionalPlan(2, Dict{Any, Any}()))))), ConditionalPlan(2, Dict{Bool,
ConditionalPlan}(0 => ConditionalPlan(2, Dict{Bool, ConditionalPlan}(0 =>
ConditionalPlan(3, Dict{Any, Any}()), 1 => ConditionalPlan(3, Dict{Any,
Any}()))), 1 => ConditionalPlan(2, Dict{Bool, ConditionalPlan}(0 =>
ConditionalPlan(3, Dict{Any, Any}()), 1 => ConditionalPlan(3, Dict{Any,
Any}())))))))
```



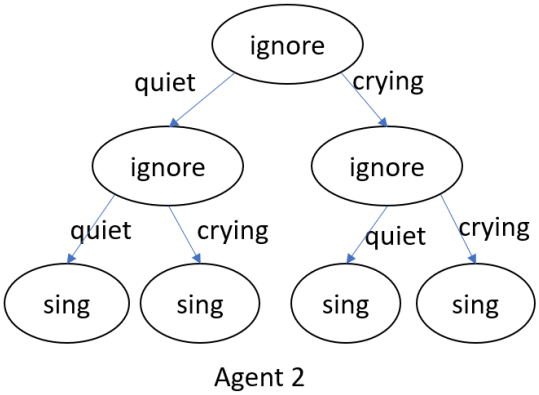
Agent 1



Agent 2



Agent 1



Agent 2

Vậy chuỗi hành động của agent 1: feed => feed => ignore

Agent 2: ignore => ignore => sing

Điểm mạnh:

- Hiểu được mô hình tính toán của POMG và sử dụng Nash Equilibrium của Simple Game để giải quyết bài toán.
- Tối ưu hóa cách giải ban đầu nhờ Dynamic Programming.
- Tìm được kết quả là chuỗi các action mà agent cần thực hiện.

Điểm yếu

- Chưa tính toán cụ thể reward mà agent sẽ nhận được khi thực hiện chuỗi action đó.
- Mã nguồn nhiều, chưa thật sự tối giản, dễ gây nhầm lẫn giữa các hàm Solve.