BÁO CÁO ĐỒ ÁN THỰC HÀNH
Môn: Cơ sở trí tuệ nhân tạo

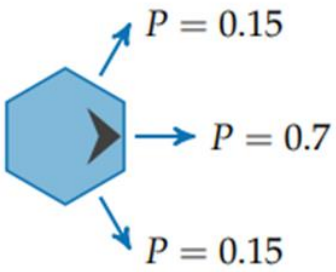
1. Thông tin nhóm

Họ và tên	MSSV
Vũ Văn Đô	19120007
Huỳnh Ngô Trung Trực	19120040
Lê Phạm Lan Anh	19120447
Nguyễn Đại Nghĩa	19120735

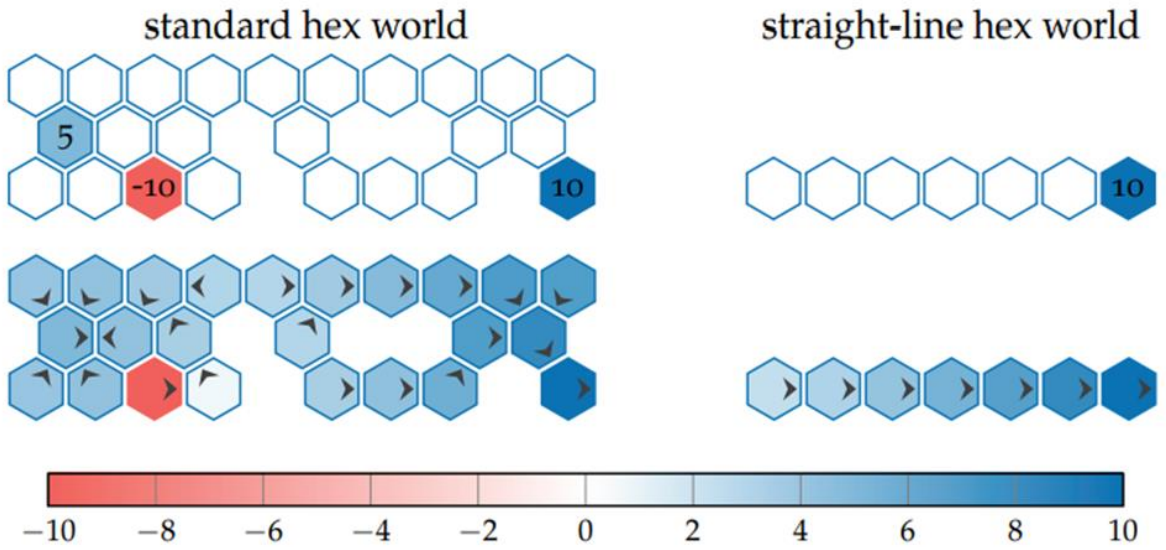
2. Phát biểu bài toán

2.1. Hex world

"Hex world" là một quy trình quyết định Markov (MDP - Markov decision process) đơn giản, trong đó 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. Ta có thể di chuyển theo bất kỳ hướng nào trong số 6 hướng. Kết quả của những hành động này là ngẫu nhiên. Như trong hình dưới, khi ta thực hiện hành động di chuyển 1 bước theo hướng xác định, xác suất ta di chuyển đúng hướng là 0.7, và xác suất ta di chuyển theo 1 trong 2 hướng lân cận là 0.3 (mỗi hướng có xác suất 0.15). Nếu ta va chạm vào đường viền bên ngoài của lưới, thì ta sẽ không thay đổi trạng thái, nhưng chi phí cho hành động này là 1.0.



Một số ô nhất định trong bài toán hex world ô chứa 1 số nguyên là key. Thực hiện bất kỳ hành động nào trong các ô này sẽ mang lại cho ta phần thưởng cụ thể và sau đó chuyển ta đến trạng thái cuối. Không có phần thưởng nào được nhận ở trạng thái cuối. Do đó, tổng số trạng thái trong bài toán hex world là số ô cộng với 1 (trạng thái đầu cuối). Hình dưới đây cho thấy một chính sách tối ưu cho hai cấu hình bài toán hex world ví dụ. Nhóm em gọi ví dụ đầu tiên là standard hex world và ví dụ thứ 2 là straight-line hex world.



Các trạng thái của standard hex world sẽ được nhóm em tự thiết kế lại để phù hợp với mục đích trình bày và sẽ sử dụng trong các phần tiếp theo. Straight-line hex world được sử dụng để minh họa cách phần thưởng được truyền từ trạng thái mang phần thưởng duy nhất của nó ở ô ngoài cùng bên phải.

2.2. Rock-Paper-Scissors

Rock-Paper-Scissors (RPS) là trò chơi với sự tham gia của nhiều người chơi. Với mỗi lượt chơi, mỗi người sẽ chọn một trong ba lựa chọn là kéo (scissors), búa (rock) hoặc bao (paper). Biết rằng kéo thắng bao, bao thắng búa và búa thắng kéo.

2.3. Traveler's Dilemma

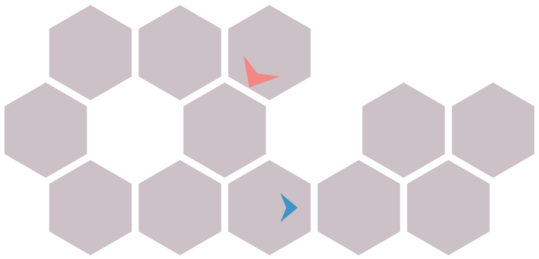
Thế tiến thoái lưỡng nan của lữ khách được đưa ra vào năm 1994 bởi nhà kinh tế Kaushik Basu, đưa ra một kịch bản trong đó một hãng hàng không làm mất hai chiếc vali giống nhau của hai lữ khách. Hãng hàng không sẵn sàng đền bù với yêu cầu hai lữ khách hãy viết riêng ước tính của họ về giá trị trong khoảng từ 2 đến 100 đô la mà không trao đổi với người kia. Số tiền đền bù mỗi người nhận được được quy định như sau:

- Nếu cả hai lữ khách đều viết cùng một số, mỗi người đều sẽ nhận được số tiền đúng với số đã viết.
- Nếu họ viết các số khác nhau, giá trị chung của mỗi chiếc vali sẽ được coi là số thấp hơn. Người đưa ra giá trị thấp hơn sẽ nhận thêm 2 đô la so với con số này, ngược lại người đưa ra số cao sẽ bị trừ đi 2 đô la.

Với mô tả như trên, mỗi lữ khách phải suy nghĩ và đưa ra một con số sao cho tối đa hóa được số tiền mình có thể nhận được trong khi không biết được con số mà người còn lại đưa ra.

2.4. Predator-Prey Hex World

"Predator-prey hex world" là một bài toán mở rộng của bài toán hex world và bao gồm nhiều tác nhân bao gồm động vật ăn thịt (predators) và con mồi (prey). Một kẻ săn mồi cố gắng bắt con mồi càng nhanh càng tốt, và con mồi cố gắng thoát khỏi kẻ săn mồi càng lâu càng tốt. Trạng thái ban đầu của hex world được thể hiện trong hình bên dưới. Không có trạng thái đầu cuối trong trò chơi này.



Có một tập hợp những kẻ săn mồi $\mathcal{I}_{pred}$ và một tập hợp những con mồi $\mathcal{I}_{prey}$, với $\mathcal{I} = \mathcal{I}_{pred} \cup \mathcal{I}_{prey}$. Các trạng thái chứa các vị trí của mỗi tác nhân: $\mathcal{S} = \mathcal{S}^1 \times \cdots \times \mathcal{S}^{|\mathcal{I}|}$, với mỗi $\mathcal{S}^i$ bằng tất cả các vị trí ô. Không gian hoạt động chung là $\mathcal{A} = \mathcal{A}^1 \times \cdots \times \mathcal{A}^{|\mathcal{I}|}$, trong đó mỗi $\mathcal{A}^i$ bao gồm tất cả sáu hướng chuyển động.

Nếu một kẻ săn mồi $i \in \mathcal{I}_{pred}$ và con mồi $j \in \mathcal{I}_{prey}$ ở cùng một ô với $s_i = s_j$, thì con mồi sẽ bị ăn thịt. Con mồi j sau đó được vận chuyển đến một ô hình lục giác ngẫu nhiên, đại diện cho con cái của nó xuất hiện trên thế giới. Mặt khác, các chuyển đổi trạng thái là độc lập và được mô tả trong hex world gốc.

Một hoặc nhiều kẻ săn mồi có thể bắt một hoặc nhiều con mồi nếu tất cả chúng tình cờ ở trong cùng một ô. Nếu n kẻ săn mồi và m con mồi cùng dùng chung một ô thì những kẻ săn mồi nhận được phần thưởng là m / n. Ví dụ, nếu hai kẻ săn mồi cùng bắt một con mồi, chúng sẽ nhận được 1/2 phần thưởng. Nếu ba kẻ săn mồi cùng bắt được năm con mồi, mỗi con sẽ nhận được phần thưởng là 5/3. Những kẻ săn mồi di chuyển nhận được hình phạt đơn vị. Con mồi có thể di chuyển mà không bị phạt, nhưng chúng bị phạt 100 vì ăn thịt.

2.5. Multi-Caregiver Crying Baby

Bài toán Crying Baby là một bài toán POMDP đơn giản với hai states, ba actions, và hai observations. Mục tiêu của chúng ta sẽ là chăm sóc và không để cho em bé khóc, để làm được điều này, ở mỗi bước chúng ta sẽ chọn một trong ba hành động: cho em bé ăn, hát cho em bé nghe và bỏ qua em bé. Sau một khoảng thời gian em bé sẽ trở nên đói và khóc, ngoài ra thì em bé cũng có thể khóc vì vài lý do khác, vậy nên khi em bé khóc chúng ta phải lựa chọn một hành động hợp lý để chăm sóc em bé.

Multi-Caregiver Crying Baby là một phiên bản mở rộng của bài toán trên, điểm khác biệt ở đây là chúng ta sẽ có nhiều người chăm sóc em bé hơn. Mỗi hành động của người chăm sóc sẽ nhận một lượng penalty nhất định, được tính như sau:

Cả hai người chăm sóc đều muốn giúp em bé khi em bé đói, trường hợp này họ sẽ nhận một lượng penalty như nhau là -10. Tuy nhiên người thứ nhất thích cho ăn hơn và người thứ hai lại thích hát hơn. Đối với hành động cho ăn, vì người thứ nhất thích cho ăn hơn nên chỉ nhận thêm -2.5 penalty, trong khi người thứ hai sẽ phải nhận thêm -5.0 penalty. Đối với hành động hát ru, người thứ nhất sẽ chịu thêm -0.5 penalty, trong khi người thứ hai sẽ nhận thêm -0.25 penalty.

Vì bà chủ chỉ có một phần quà duy nhất, nên người nào muốn nhận được phần thưởng phải có ít điểm penalty hơn.

3. Thách thức

3.1. Hex world

Nhóm em nhận thấy các hành động không có kết quả tất định mà tồn tại yếu tố ngẫu nhiên, gây khó khăn trong tính toán, là thách thức của bài toán vì khó áp dụng các phương pháp tìm kiếm thông thường như BFS, DFS.

Bài toán MDP có không gian trạng thái lớn gây khó khăn trong quá trình xây dựng và giải quyết bài toán.

Khó đánh giá thuật giải một cách hiệu quả.

3.2. Rock-Paper-Scissors

Các người chơi đều muốn giành chiến thắng nên sẽ phải cân nhắc để đưa ra sự lựa chọn tốt nhất. Tuy nhiên, đây là trò chơi mang tính đối kháng, sự lựa chọn của người này sẽ ảnh hưởng đến sự lựa chọn của người còn lại. Nên việc đề ra chiến lược hợp lý và nắm bắt chiến lược của đối phương cũng là một thách thức cho mỗi người chơi

3.3. Traveler's Dilemma

Mỗi lữ khách đều muốn tối đa hóa số tiền cá nhân mình nhận được. Tuy nhiên đây là một trò chơi mang tính đối kháng, sự lựa chọn của người này gây ảnh hưởng đến sự lựa chọn của người còn lại. Cụ thể, tùy vào lựa chọn của người kia mà con số tối ưu của người này sẽ bị thay đổi. Dễ thấy giả sử biết được số tiền đề xuất của một người đưa ra là x thì số tiền tối ưu mà người còn lại nên đưa ra là max(2, x-1). Tuy nhiên không có cách nào để có thể biết được lựa chọn của đối phương, theo mô tả của bài toán.

3.4. Predator-Prey Hex World

Predator-Prey Hex World là bài toán tìm kiếm đối kháng có nhiều agent đưa ra quyết định đồng thời và độc lập, vì vậy khi một agent đưa quyết định, luôn tồn tại yếu tố không tất định.

Bài toán Markov game có không gian trạng thái lớn gây khó khăn trong quá trình xây dựng và giải quyết bài toán.

Khó đánh giá thuật giải một cách hiệu quả.

3.5 Multi-Caregiver Crying Baby

Vì em bé còn quá nhỏ và không thể nói cho chúng ta biết lý do em bé khóc, thế nên việc lựa chọn những hành động như nào để có thể nhận ít penalty nhất là một thách thức.Thêm nữa, nếu em bé đói và cả hai người đều không cho ăn kịp thời, thì cả hai sẽ phải chịu lượng điểm phạt rất lớn.

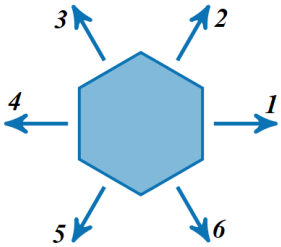
4. Thực nghiệm

4.1. Hex world

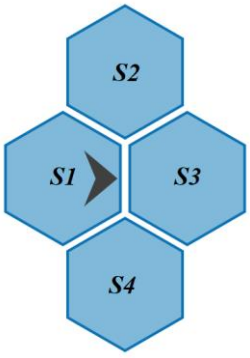
4.1.1. Mô hình hoá bài toán

Bài toán hex world là một MDP đơn giản, vì vậy nhóm em mô hình hoá bài toán bằng các thuộc tính như trong MDP, cụ thể như sau:

- Discount factor, kí hiệu $\gamma \in [0, 1]$: thể hiện mức độ "cận thị" của mô hình. Với giá trị càng gần 0, chiến lược đưa ra nhằm giúp tối đa điểm thưởng ở các lượt gần tiếp theo, giá trị càng gần 1 thì chiến lực đưa ra nhằm giúp tối đa điểm thưởng ở tầm xa hơn. Ở bài này, nhóm em sử dụng $\gamma = 0.9$.
- State space, kí hiệu là $\mathcal{S}$: không gian trạng thái của mô hình, gồm các ô trong bản đồ và một trạng thái cuối.
- Action space, kí hiệu là $\mathcal{A}$: không gian hành động của mô hình, gồm 6 hướng di chuyển ứng với mỗi trạng thái. Ở bài này, chúng em quy định hành động 1 là đi sang phải, các hành động tiếp theo được quy định số theo chiều kim đồng hồ, như hình dưới đây.



- Transition function, kí hiệu là T(s, a, s'): hàm thể hiện xác suất đến được trạng thái s' từ một trạng thái s khi thực hiện hành động a. Dựa vào bài toán, ta thấy có 3 trường hợp:
    - Đối với trạng thái tại ô không có key, xác suất ta di chuyển đúng hướng là 0.7, và xác suất ta di chuyển theo 1 trong 2 hướng lân cận là 0.3 (mỗi hướng có xác suất 0.15). Xét một ví dụ như hình bên dưới, ta có $T(S1, A1, S2) = 0.15$, $T(S1, A1, S3) = 0.7$, $T(S1, A1, S4) = 0.15$, điều này có ý nghĩa xác suất từ tráng thái S1, thực hiện hành động A1 sẽ được kết quả trạng thái S2 là 15%, tương tự với S3, S4 lần lượt là 70%, 15%.



    - Đối với trạng thái tại ô chứa key, khi thực hiện bất kì hành động nào đều chắc chắn (xác suất 100%) dẫn đến trạng thái cuối.
    - Đối với trạng thái cuối, mọi hành động đều chắc chắn (xác suất 100%) không thay đổi trạng thái.

- Reward function, kí hiệu là R(s,s'): hàm thể hiện các phần thưởng (có thể âm hoặc dương) nhận được khi thay đổi sang một trạng thái mới. Ở bài toán này, ta nhận thấy có 2 loại phần thưởng:
    - Đối với trạng thái ô không chứa key, khi thực hiện hành động dẫn tới việc đi ra ngoài biên (khi đó trạng thái sẽ không thay đổi), chi phí cho điều này là 1. Nói cách khác, reward khi thực hiện hành động mà không thay đổi trạng thái (trừ trạng thái cuối) là -1.
    - Đối với trạng thái ô chứa key, khi thay đổi trạng thái thành trạng thái cuối, phần thưởng (reward) cho hành động này bằng key của ô đó.
- Sample transition and reward, kí hiệu là TR: gồm các transition và các reward mẫu. Ở bài toán này, không có các transition và reward nên trong phần thực nghiệm của nhóm em, biến TR sẽ không chứa giá trị.

4.1.2. Phương pháp giải quyết

Sau khi mô hình hoá bài toán thành một MDP, nhóm em sử dụng hai phương pháp sẽ được trình bày sau đây.

4.1.2.1. Phương pháp cơ bản

Nhóm em sử dụng phương pháp Iterative để giải quyết bài toán. Đây là một phương pháp dễ cài đặt, không gian lưu trữ ít. Mặc dù phương pháp này còn hạn chế về độ phức tạp thời gian chạy. Tuy nhiên với một môi trường không quá lớn (không quá 100 trạng thái) thì thuật toán thực hiện với thời gian không đáng kể. Đó là những lí do nhóm em sử dụng phương pháp này để giải quyết bài toán đặt ra.

Ở phương pháp này, nhóm em sử dụng các biến quan trọng sau:

- MDP: chứa các tham số của bài toán, đã được trình bày ở phần mô hình hoá.
- π: policy.
- k_{max}: số vòng lặp tối đa.
- U: hàm giá trị đánh giá tạm thời.

Trong đó "π" bao gồm:

- U_π: hàm giá trị đánh giá chính sách ứng với policy π
- $\pi(s)$: hành động khi ở trạng thái s được quyết định bởi policy π

$$\pi(s) = \max_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s'}^{\mathcal{S}} T(s,a,s').U_{n-1}(s') \right)$$

Trong đó $r(s,a)$ là giá trị phần thưởng kỳ vọng khi thực hiện hành động a tại trạng thái s.

Ta có lập công thức tính hàm đánh giá U dựa vào policy π là công thức truy hồi sau

$$\begin{cases} U_0(s) = 0 \\ U_k(s) = r^*\big(s,\pi(s)\big) + \gamma \sum_{s'}^{\mathcal{S}} T(s,\pi(s),s').U_{k-1}(s') \quad ,\forall\, k = 1 \to k_{max} \end{cases}$$

Trong đó $r^*(s,a)$ là giá trị phần thưởng ngẫu nhiên khi thực hiện hành động a tại trạng thái s.

Từ các công thức đã được lập ở trên, ta tiến hành giải như sau:

- Tạo policy π_0 với hàm đánh giá $U_{\pi_0}(s) = 0,\ \forall s \in \mathcal{S}$.
- Với $\forall\, k = 1 \to k_{max}$, ta tạo hàm U dựa vào π_{k-1} và tạo π_k từ U vừa tính được.
- Nếu $\exists k$ thoải $\pi_k(s) = \pi_{k-1}(s),\ \forall s \in \mathcal{S}$, thì π_k là policy tốt nhất tìm được (kết quả hội tụ tại k). Ngược lại, $\pi_{k_{max}}$ là policy tốt nhất tìm được.

4.1.2.2. Phương pháp cải tiến

Ở phương pháp trên, ta nhận thấy ở công thức truy hồi để tính hàm đánh giá U dựa vào policy π tồn tại một yếu tố ngẫu nhiên ở hàm $r^*(s,a)$. Điều này dẫn đến kết quả hội tụ chậm, nên không tiết kiệm được tài nguyên chạy (cả về mặt thời gian và không gian).

Để tối ưu hoá thời gian chạy không cần thiết, đồng thời vẫn không làm suy giảm tính chính xác, nhóm em đã cải tiến bằng cách thay thế hàm $r^*(s,a)$ bằng hàm $r(s,a)$. Nói cách khác, nhóm em thay thế yếu tố ngẫu nhiên bằng một yêu tố tất định. Điều này làm mất tính ngẫu nhiên vốn có của MDP, tuy nhiên sẽ loại bỏ được nhiều trường hợp sai lệnh so với kết quả (vì yếu tố ngẫu nhiên) làm thuật toán chạy không tốt và thậm chí không giải được.

4.1.3. Code

Lưu ý: *Để thực thi thuật toán, mở terminal tại thư mục code hex world và chạy câu lệnh*

```
julia main.jl 1
```

để chạy chương trình theo phương pháp cơ bản.
Hoặc chạy câu lệnh

```
julia main.jl 2
```

để chạy chương trình theo phương pháp cải tiến.

Ở phần này, chúng em sẽ trình bày tổng quan về phần cài đặt thuật toán bằng ngôn ngữ lập trình julia. Nhóm em đã cài đặt đầy đủ phần mô hình hoá bài toán và phương pháp giải quyết, ngoài ra nhóm cài đặt phần visualization để dễ dàng quan sát đánh giá kết quả đạt được.

Các struct được cài đặt gồm các struct sau.

- Struct MDP chứa các thuộc tính định nghĩa một Markov decision process.

MDP	

`γ::Float64`
`𝒮::Array{Int}`
`𝒜::Array{Int}`
`T::Array{Float64, 3}`
`R::Array{Float64, 2}`
`TR`
	Discount factor
	State space
	Action space
	Transition function
	Reward function
	Sample transition and reward

- Struct DiscreteMDP chứa tóm gọn các thuộc tính có trong MDP.

DiscreteMDP	

`T::Array{Float64, 3}`
`R::Array{Float64, 2}`
`γ::Float64`
	Transition function
	Reward function
	Discount factor

- Struct HexWorldMDP chứa các thuộc tính để định nghĩa một hex world và DiscreteMDP của nó.

HexWorldMDP	

`hexes::Vector{Tuple{Int,Int}}`
`dMDP::DiscreteMDP`
`special_hex_rewards::Dict{`
`Tuple{Int,Int},Float64}`
	Mảng toạ độ các ô trong bản đồ
	DiscreteMDP tương ứng
	Các giá trị key của bản đồ

- Struct Policy chứa thông tin của policy và MDP cần giải quyết.

Policy	
`mdp::MDP` `U::Vector{Float64}`	MDP cần giải quyết Hàm giá trị đánh giá policy tương ứng
`Policy(s)::Int`	Hàm trả về hành động ứng với trạng thái s được tính dựa trên policy tương ứng

Các hàm chính được cài đặt gồm các hàm sau.
(Các đoạn code được minh hoạ dưới đây được lược bỏ một số dòng lệnh không cần thiết để phù hợp với nội dung bài báo cáo)

- `HexWorld()::HexWorldMDP`

Là hàm khởi tạo một HexWorldMDP từ các thông tin của bài toán hex world. Hàm này cho phép thay đổi map bằng cách thay đổi biến `map1`.

```
function HexWorld()
    hexes, rewards = map1
    return HexWorldMDP(hexes, HexWorldRBumpBorder,
HexWorldPIntended, rewards, HexWorldDiscountFactor)
end
```

- `Solve(::Policy, ::Int, ::Vector{Tuple{Int, Int}}, ::Int)`
  `::HexWorldMDP`

Là hàm giải quyết bài toán. Hàm này nhận vào các thông tin của bài toán chứa trong tham số `policy`, tham số `k_max` số lần lặp tối đa, tham số `hexes` phục vụ việc visualize và tham số `method` thể hiện phương pháp làm (`method==1` nếu sử dụng phương pháp 1, và `method==2` nếu sử dụng phương pháp 2)

```
function Solve(
    policy::Policy,
    k_max::Int,
    hexes::Vector{Tuple{Int, Int}},
    method::Int
)
    mdp = policy.mdp
    𝒮 = mdp.𝒮
    for k = 1 : k_max
        U = iterative_policy_evaluation(mdp, policy, k_max,
method::Int)
        policy’ = Policy(mdp, U)

        if all(policy(s) == policy’(s) for s in 𝒮)
```

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

- `function iterative_policy_evaluation(::Policy, ::Int, ::Int) :: Vector{Float64}`

Là hàm tính U dựa trên policy tương ứng. Hàm này nhận vào các thông tin của bài toán chứa trong tham số `policy`, tham số `k_max` số lần lặp tối đa và tham số `method` thể hiện phương pháp làm (`method==1` nếu sử dụng phương pháp 1, và `method==2` nếu sử dụng phương pháp 2)

```
function iterative_policy_evaluation(policy::Policy, k_max::Int,
method::Int)
    mdp = policy.mdp
    𝒮 = mdp.𝒮
    U = zeros(Float64, length(𝒮))
    for k in 1 : k_max
        U = [lookahead(mdp, U, s, policy(s), method == 2) for s in
𝒮]
    end
    return U
end
```

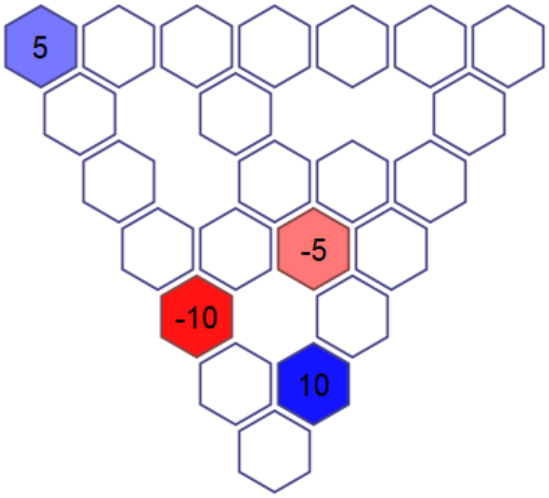
- `function lookahead(::MDP, ::Vector{Float64}, ::Int64, ::Int64, ::Bool) :: Int`

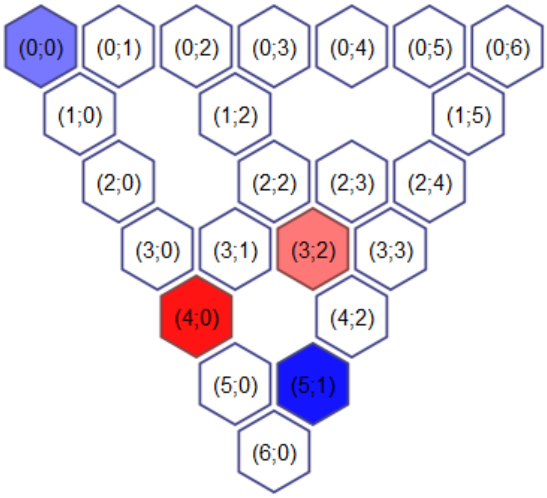
Là hàm tính giá trị tại trạng thái `s` khi thực hiện hành động `a` dựa trên policy tương ứng. Hàm này nhận vào các thông tin của bài toán chứa trong tham số `mdp`, tham số `k_max` số lần lặp tối đa và tham số `is_deterministic` thể hiện phương pháp tính giá trị `r`.

```
function lookahead(mdp::MDP, U::Vector{Float64}, s::Int64,
a::Int64, is_deterministic::Bool)
    𝒮, T, R, γ = mdp.𝒮, mdp.T, mdp.R, mdp.γ
    r = rand_with_prob(R[s, :], T[s, a, :])
    if is_deterministic
        r = sum(T[s, a, s′] * R[s, s′] for s′ in 𝒮)
    end
    return r + γ * sum(T[s, a, s′] * U[s′] for s′ in 𝒮)
end
```

4.1.4. Phân tích

Để phân tích thuật toán, nhóm em đã thiết kế một số bản đồ. Dưới đây là một bản đồ mẫu với hình bên trái là bản đồ cùng các key của bản đồ đó, hình bên phải là bản đồ cùng toạ độ của nó.





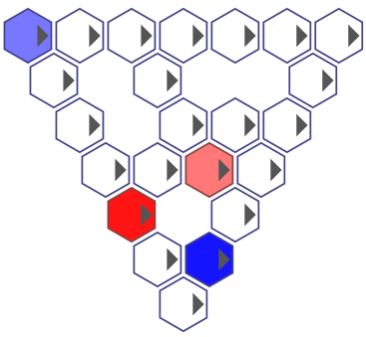
Để so sánh 2 phương pháp, ta chạy thử ví dụ trên mỗi phương pháp 5 lần với k_max = 1000, ta được kết quả sau.

```
D:\Subject\nam3_hk1\AI\code\Hexworld>julia main.jl 1

Num of loops: 16

D:\Subject\nam3_hk1\AI\code\Hexworld>julia main.jl 1

Num of loops: 34

D:\Subject\nam3_hk1\AI\code\Hexworld>julia main.jl 1

Num of loops: 19

D:\Subject\nam3_hk1\AI\code\Hexworld>julia main.jl 1

Num of loops: 26

D:\Subject\nam3_hk1\AI\code\Hexworld>julia main.jl 1

Num of loops: 12
```

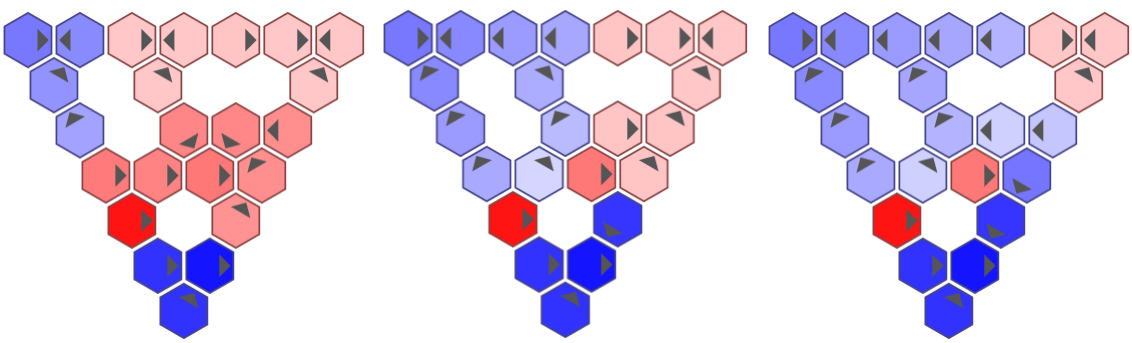
```
D:\Subject\nam3_hk1\AI\code\Hexworld>julia main.jl 2

Num of loops: 6

D:\Subject\nam3_hk1\AI\code\Hexworld>julia main.jl 2

Num of loops: 6

D:\Subject\nam3_hk1\AI\code\Hexworld>julia main.jl 2

Num of loops: 6

D:\Subject\nam3_hk1\AI\code\Hexworld>julia main.jl 2

Num of loops: 6

D:\Subject\nam3_hk1\AI\code\Hexworld>julia main.jl 2

Num of loops: 6
```

Dễ dự đoán được các lần chạy của phương pháp 2 có kết quả không thay đổi, vì không có yếu tố ngẫu nhiên trong thuật toán. Đối với phương pháp 1, ta thấy được sự chênh lệch lớn giữa các lần chạy, ta có thể kết luận rằng một yếu tố ngẫu nhiên nhỏ có thể dẫn tới bài toán có tính bất ổn rất lớn. Các đánh giá sau này được mặc định là sửa dụng phương pháp 2 nếu không giải thích gì thêm.

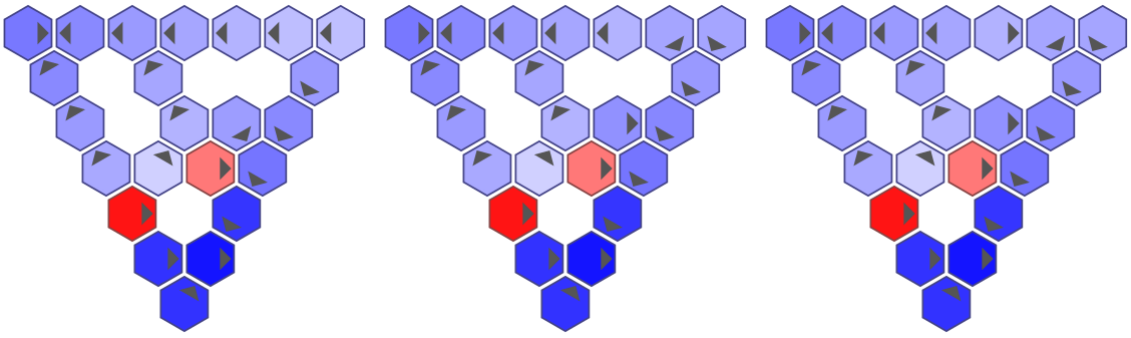
Để hình dung được quá trình tính toán policy qua các lần lặp, nhóm em đã hình ảnh hoá lại các giá trị đánh giá qua từng vòng lặp như hình dưới đây.



Before the loop



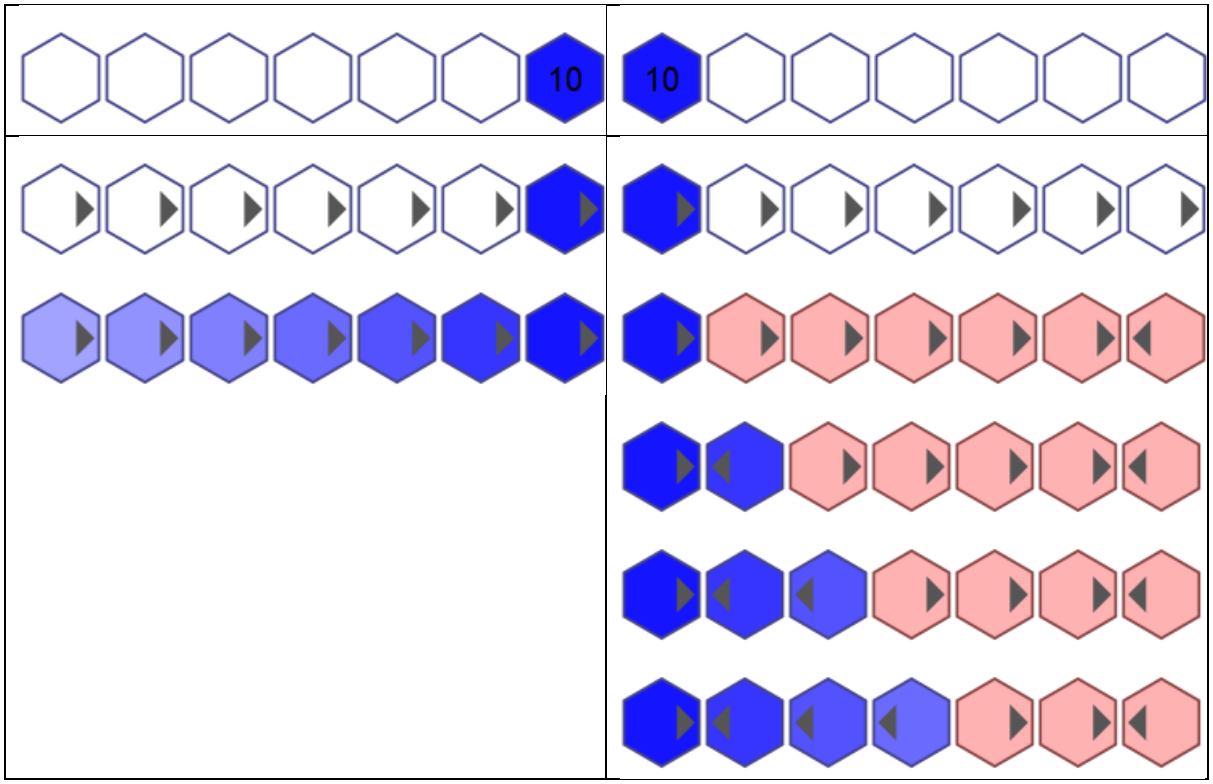
Loop1 Loop2 Loop3

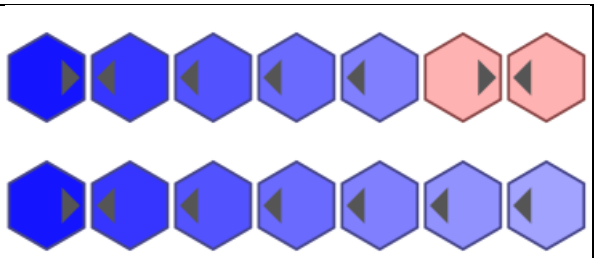


Loop4 Loop5 Loop6

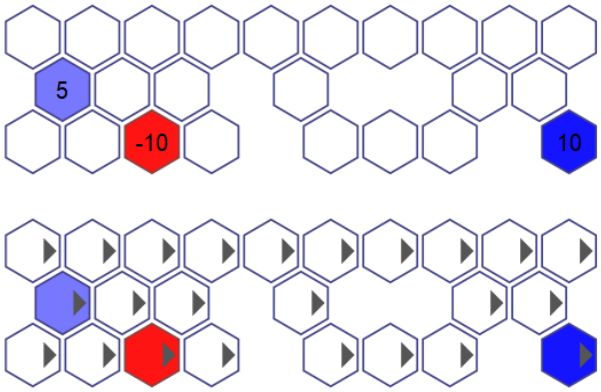
Dựa vào kết quả trên, ta thấy rằng đây là một policy hiệu quả bởi nó nhanh chóng đưa ra hành động tốt nhất dựa vào những giá trị đánh giá có trước.

Tuy nhiên thuật toán này còn một số điểm yếu như việc đánh giá phụ thuộc vào hành động ban đầu của khác trạng thái. Để dễ hình dung hơn, ta thử nghiệm bài toán với straight-line hex world đã đề cập ở đề bài trong 2 trường hợp sau.



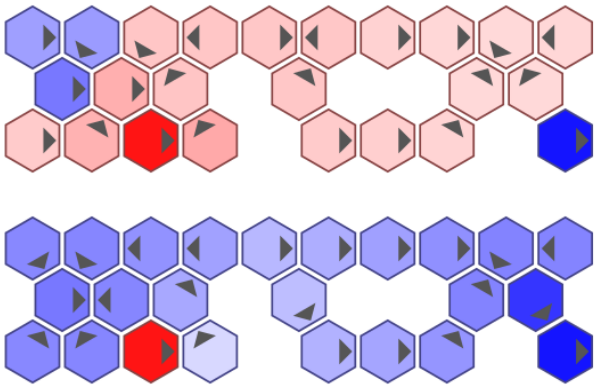
	

Ta thấy mặc dù 2 bản đồ chỉ khác nhau về hướng nhưng số vòng lặp để giải quyết bài toán có sự chênh lệch. Tuy nhiên đây không phải vấn đề quá lớn vì nó chỉ xảy ra ở các trường hợp đặc biệt. Nhóm em đề xuất một cách khắc phục bằng cách cài đặt các hành động ban đầu không đi về cùng một hướng mà là các hướng ngẫu nhiên.



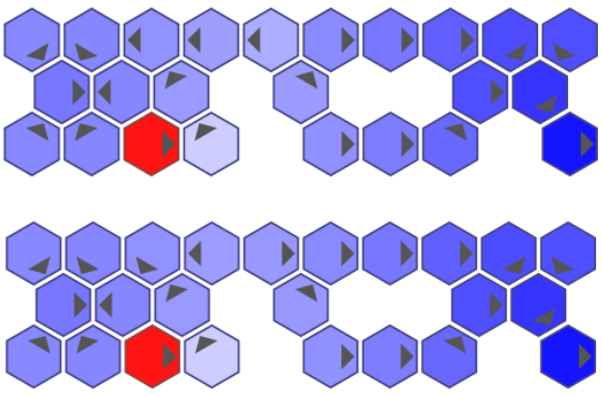
Map and key

Before the loops



Loop1

Loop2



Loop3

Loop4

Ta thấy thuật toán này thực hiện tốt ở nhiều trường hợp. Tuy nhiên, nhóm em nhận thấy phương pháp có điểm hạn chế là trong các trường hợp bản đồ quá lớn, khi đó cần thực hiện các phương pháp cải tiến thời gian chạy để phù hợp hơn với bài toán.

4.2. Rock-Paper-Scissors

Trong thực nghiệm này, ta xét trò chơi Rock-Paper-Scissors với quy mô là 2 người chơi. Người chiến thắng sẽ được thưởng 1 điểm và kẻ thua cuộc sẽ bị phạt -1 điểm. Nếu cả 2 đều đưa ra cùng sự lựa chọn thì hòa nhau, mỗi người không được thưởng, cũng không bị phạt. Trò chơi sẽ được thực hiện k_max lần, với k_max = {1000, 1000000}.

4.2.1. Mô hình hóa tính toán

RPS là trò chơi thuộc loại Simple game (trò chơi đơn giản). Simple game là một mô hình cơ bản cho những trò chơi suy luận nhiều người tham gia. Trò chơi được biểu diễn theo 4 thuộc tính: discount factor, agents, joint action space và joint reward function.

- Discount factor γ ($0 \leq \gamma < 1$): là một khái niệm trong kinh tế học để xây dựng bài toán sao cho điểm thưởng thời điểm hiện tại giá trị hơn điểm thưởng trong tương lai. Trong cụ thể trò chơi này với trường hợp số lần chơi tiến tới vô cùng thì $\gamma = 0.9$
- Agents $\mathcal{I}$: là danh sách người tham gia. Trong thực nghiệm này là 2 người, đánh số lần lượt là 1 và 2: $\mathcal{I} = \{1, 2\}$
- Joint action space $\mathcal{A}$: là không gian bao gồm tất cả các hành động có thể có cho mỗi người chơi: $\mathcal{A} = \mathcal{A}^1 \times \mathcal{A}^2$ với $\mathcal{A}^i = \{\text{rock, paper, scissors}\}$, $i = \{1, 2\}$
- Joint reward function R: là hàm trả về điểm tương ứng với mỗi hành động: $R^1(a^1, a^2)$, $R^2(a^1, a^2)$

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

4.2.2. Phương pháp giải quyết

Trong thử nghiệm này, nhóm em dùng thuật toán Fictitious Play để giải quyết bài toán. Khi cho 2 người tham gia chơi trong một vòng lặp vô hạn lần, mỗi người chơi phải đề ra những chiến lược để đưa ra sự lựa chọn tốt nhất cho hiện tại. Họ cần học và cập nhật chiến lược của mình dựa trên các lần chơi trước đó của đối phương. Fictitious Play là một hướng tiếp cận bằng cách người chơi sẽ ước tính lựa chọn có khả năng xảy ra cao nhất của đối thủ. Mỗi người chơi sẽ tuân theo để đưa ra câu trả lời tốt nhất dựa trên những ước tính này.

Để tính toán khả năng xảy ra, người chơi sẽ ghi nhớ số lần dùng hành động a^j của người chơi j và lưu vào bảng N(j, a^j). Số lượng này có thể được khởi tạo tùy ý, trong bài toán RPS nhóm em khởi tạo nó là 1. Người chơi sẽ xem xét hành động được đối thủ sử dụng nhiều nhất và chọn ra hành động có thể thắng hành động đó.

Nhóm em quyết định thử nghiệm với Fictitious Play vì:

- Dominant Strategy Equilibrium (Cân bằng áp đảo) không thể giải quyết bài toán này vì không tồn tại sự lựa chọn áp đảo do sự lựa chọn có tốt hay không phụ thuộc vào sự lựa chọn của đối phương.
- Nhóm em không chọn Nash Equilibrium do đây là một trò chơi lặp lại nhiều lần và Nash Equilibrium khá tốn kém trong việc tính toán.
- Fictitious Play mô phỏng tương tự như việc não người sẽ nhớ các lựa chọn nhiều nhất mà đối phương đưa ra để điều chỉnh chiến lược của mình.

4.2.3. Code

Đầu tiên, ta có một lớp SimpleGame gồm các thuộc tính như định nghĩa [2]

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

Sau đây là lớp RockPaperScissors và hàm trả về các giá trị các thuộc tính $\mathcal{I}$, $\mathcal{A}$, $\mathbf{R}$ của RockPaperScissors tương ứng với SimpleGame. [2]

```
# Đối tượng trò chơi Rock Paper Scissors
struct RockPaperScissors end
# Số người tham gia
n_agents(simpleGame::RockPaperScissors) = 2
# Tập các hành động mà mỗi người có thể lựa chọn
ordered_actions(simpleGame::RockPaperScissors, i::Int) = [:rock, :paper, :scissors]
# Bảng các hành động các cả hai người tham gia
ordered_joint_actions(simpleGame::RockPaperScissors) =
vec(collect(Iterators.product([ordered_actions(simpleGame, i) for i in
1:n_agents(simpleGame)]...)))
# Độ dài của ordered_actions và ordered_joint_actions
n_joint_actions(simpleGame::RockPaperScissors) =
length(ordered_joint_actions(simpleGame))
n_actions(simpleGame::RockPaperScissors, i::Int) = length(ordered_actions(simpleGame, i))
# Điểm thưởng 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
# Bảng điểm cho tất cả các trường hợp
function joint_reward(simpleGame::RockPaperScissors, a)
    return [reward(simpleGame, i, a) for i in 1:n_agents(simpleGame)]
end
```

Đây là hàm tạo ra một SimpleGame cho trò chơi RockPaperScissors với các thuộc tính tương ứng trong phần Mô hình hóa tính toán. [2]

```
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
```

Tiếp theo ta định nghĩa lớp FictitiousPlay gồm 4 thuộc tính. [1]

```
mutable struct FictitiousPlay
    𝒫 # đối tượng SimpleGame của trò RPS
    i # thứ tự người chơi
    N # mảng các từ điển lưu số lượng xuất hiện của từng hành động của người chơi
    πi # chiến lược hiện tại
end
```

Đây là hàm tạo một FictitiousPlay dựa vào một SimpleGame $\mathcal{P}$ và người chơi i. N sẽ được tạo mặc định là 1 cho tất cả các hành động. πi sẽ được tạo mặc định là một SimpleGamePolicy với tham số truyền vào là một Dict gồm các keys là các hành động và values có giá trị bằng nhau và bằng 1. [1]

```
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
```

SimpleGamePolicy có thuộc tính p là một từ điển gồm keys là các hành động và values là xác suất tương ứng với hành động đó. Ngoài ra, nó còn có các hàm tạo theo tham số Dict được chuẩn hóa sao cho giá trị của p thỏa: $0 \leq p.value \leq 1$ [1]

```
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
```

Ta có hàm **main** là hàm được thực thi đầu tiên của chương trình. Trong hàm, ta khởi tạo một trò chơi RPS là $\mathcal{P}$ và cách chơi theo Fictitious Play cho 2 người chơi là $\pi 1$ và $\pi 2$. Sau đó, ta gọi hàm **simulate($\mathcal{P}$, π, 1000000)** để mô phỏng quá trình chơi trò RPS theo cách chơi Fictitious Play của 2 người chơi sau 1000000 lần.

```
function main()
    𝒫 = SimpleGame(RockPaperScissors())
    π1 = FictitiousPlay(𝒫, 1)
    π2 = FictitiousPlay(𝒫, 2)
    π = [π1, π2]
    simulate(𝒫, π, 1000000)
end
```

Hàm **simulate** sẽ thực hiện simple game $\mathcal{P}$, với cách chơi π và số lần chơi k_max. Mỗi lần chơi k, 2 người chơi sẽ chọn ra chiến lược chứa trong a và sau đó cập nhật chiến lược bằng cách gọi hàm **update!**. Cuối cùng là lưu kết quả bằng hàm **write_score!** và **write_prop!** và thực hiện **visualization** để vẽ đồ thị. [1]

```
function simulate(𝒫::SimpleGame, π, k_max)
    # mảng chứa số điểm của 2 người chơi
    score = [0, 0]
    # mảng chứa số điểm của 2 người chơi theo các lần chơi
    agent = [[], []]
    # mảng chứa xác suất các hành động đã chọn của 2 người chơi theo các lần chơi
    prop = [[[], [], []], [[], [], []]]
    # mảng chứa các lựa chọn của 2 người chơi theo các lần chơi
    policy = [[[], [], []], [[], [], []]]
    # duyệt k_max vòng
    for k = 1:k_max
        # a là mảng sự lựa chọn của 2 người chơi
        a = [πi() for πi in π]
        # cập nhật chiến lược của từng người chơi
        for πi in π
            update!(πi, a)
        end
        # lưu kết quả vào agent, prop, policy để vẽ visualization
        write_score!(π, score, agent, policy)
        write_prop!(π, prop)
    end
    # vẽ visualization
    visualization(k_max, agent, prop, policy)
```

```
    return π
end
```

Hàm **update!** sẽ tiến hành cập nhật số lượng hành động cho mỗi người chơi bằng cách cộng thêm 1 vào hành động vừa thực hiện. Sau đó, tạo ra π gồm 2 SimpleGamePolicy chứa xác suất xuất hiện của các hành động của 2 người chơi. Cuối cùng là đưa ra lựa chọn tốt nhất bằng cách gọi hàm **best_response**. [1]

```
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
```

Best response (câu trả lời tốt nhất) cho một người chơi là chiến lược mà không có điều gì làm họ thay đổi chiến lược của mình dựa trên một tập các chiến lược cố định của đối thủ. Hàm **best_response** sẽ tính toán **utility** cho joint policy π của người chơi **i**. Sau đó chọn ra hành động có trung bình lớn nhất để trả về bằng hàm **argmax**. [1]

```
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
```

Utility (lợi ích) của một joint policy π từ góc nhìn của người chơi **i** được tính theo công thức:

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

Công thức trên được tính trong hàm **utility(𝒫::SimpleGame, π, i)** [1]

```
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
```

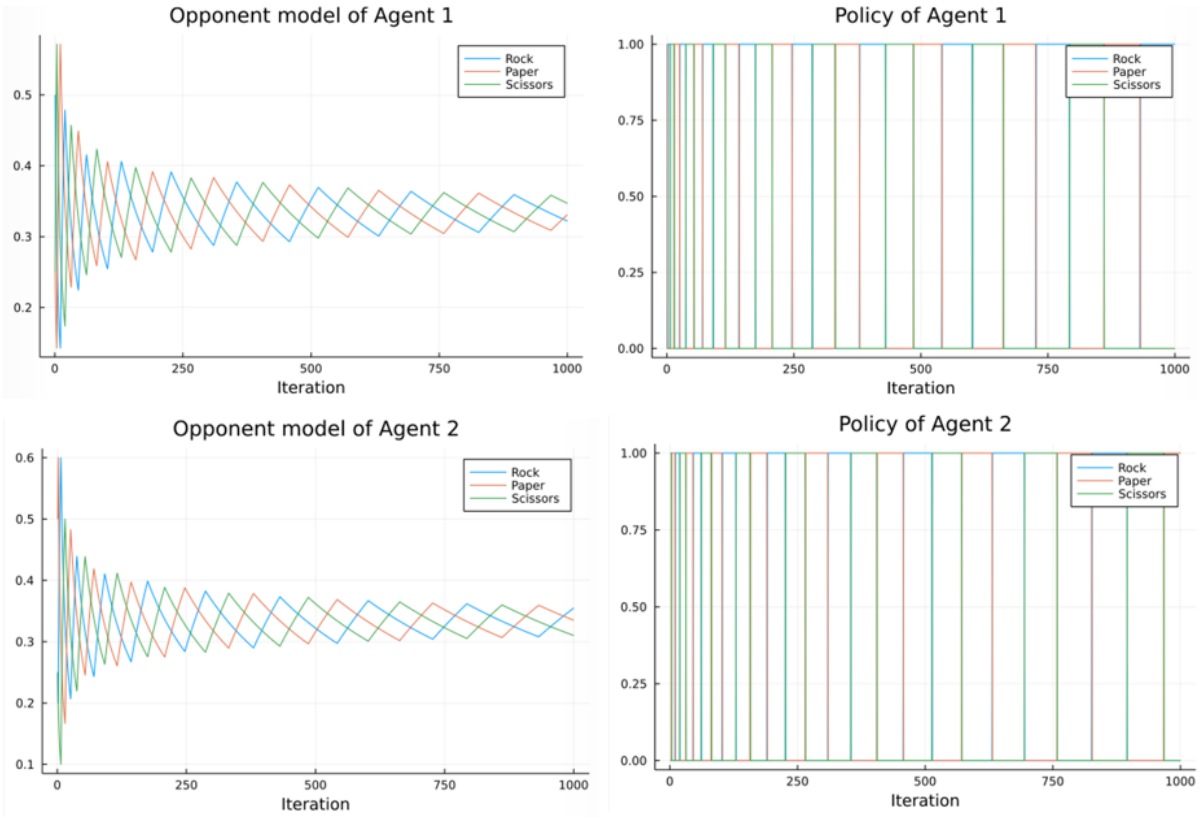
4.2.4. Phân tích

Có 3 loại đồ thị được xét:

- Opponent model: là đồ thị biểu diễn xác suất chọn Rock, Paper, Scissors của đối thủ
- Policy: là đồ thị biểu diễn sự lựa chọn của người chơi
- Scores of two agents: là đồ thị biểu diễn điểm của hai người chơi

Opponent model và Policy

- Với k_max = 1000



Ví dụ xét lần chơi thứ 750:

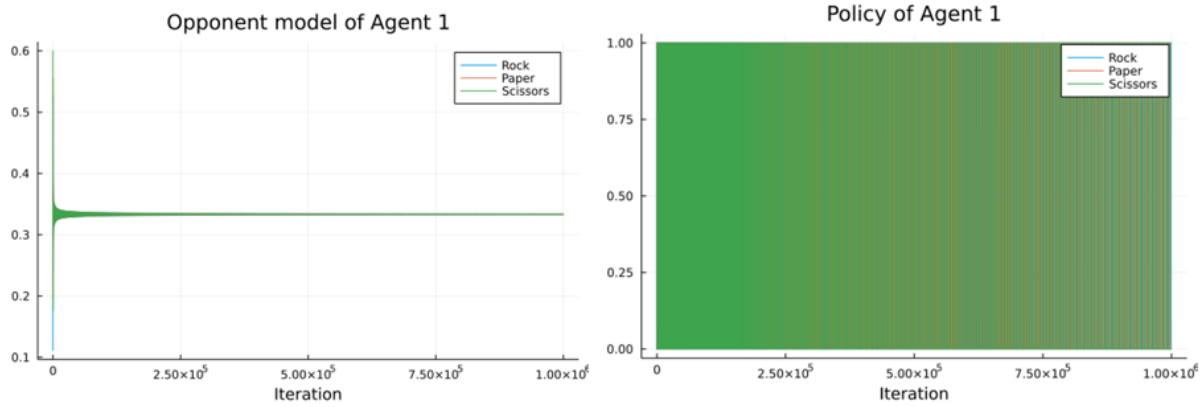
+ Người chơi 1 nhận thấy đối thủ có xu hướng chọn Scissors nên đã chọn Rock

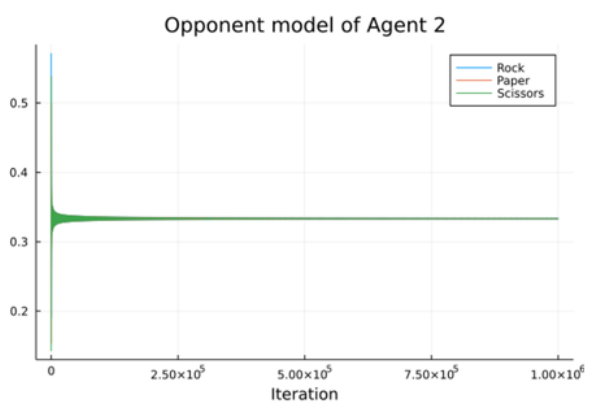
+ Người chơi 2 nhận thấy đối thủ có xu hướng chọn Paper nên đã chọn Scissors

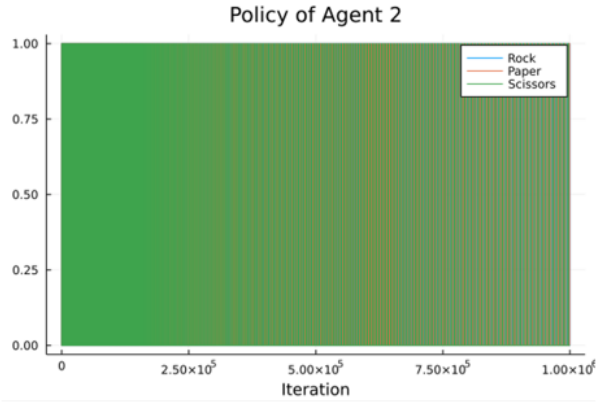
Trong đồ thị Opponent Model, ta thấy sự chênh lệch giữa các lựa chọn Rock-Paper-Scissors ngày càng giảm và có xu hướng hội tụ về chiến lược ngẫu nhiên Nash Equilibrium.

Trong đồ thị Policy, ta thấy sự thay đổi lựa chọn ngày càng thưa dần, tức người chơi có xu hướng giữ một sự lựa chọn lâu hơn.

- Với k_max = 1000000



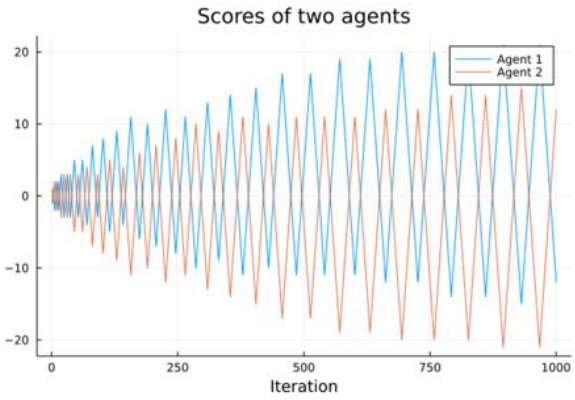


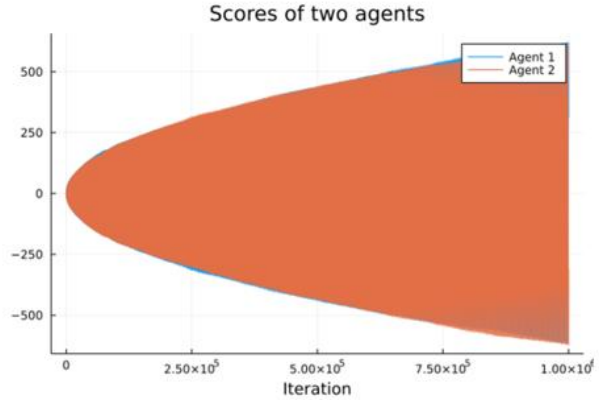


Trong đồ thị Opponent Model, phân phối xác suất xuất hiện của Rock-Paper-Scissors có xu hướng hội tụ về $\frac{1}{3}$.

Trong đồ thị Policy, các đường thẳng dọc dày đặc chồng chéo lên nhau khiến ta chỉ thấy màu xanh lá lúc ban đầu, sau đó thưa dần và xuất hiện thêm màu đỏ và xanh lam ở những lần lặp sau.

Scores of two agents





Ngược lại, trong đồ thị Scores of two agents, ta thấy đồ thị có xu hướng phân kỳ. Sự chênh lệch điểm số của 2 người chơi ngày càng tăng.

Vậy ta thấy càng chơi nhiều lần, phân phối xác suất các hành động sẽ hội tụ về $\frac{1}{3}$ và người chơi sẽ có xu hướng giữ sự lựa chọn lâu hơn.

4.3. Traveler's Dilemma

4.3.1. Mô hình hóa tính toán

Traveler's dilemma là một trò chơi thuộc loại Simple Game. Simple Game là một mô hình cơ bản cho những trò chơi suy luận nhiều người tham gia. Mỗi người chơi sẽ lựa chọn một hành động để tối đã hóa phần thưởng của bản thân. Mỗi Simple Game được biểu diễn theo 4 thuộc tính:

- Discount factor, kí hiệu γ ($0 \leq \gamma < 1$): thể hiện mức độ "cận thị" trong chiến thuật của người chơi. Với giá trị càng gần 0, chiến lược đưa ra nhằm giúp tối đa điểm thưởng ở các lượt gần tiếp theo, giá trị càng gần 1 thì chiến lực đưa ra nhằm giúp tối đa điểm thưởng ở tầm xa hơn.

- Agents, kí hiệu $\mathcal{I}$: danh sách người tham gia trong trò chơi. Trong Traveler's Dilemma I={1,2}
- Joint action space, kí hiệu A: Không gian của tất cả các hoán vị hành động của tất cả người chơi. Trong Traveler's Dilemma A=A^1×A^2 với A^i={x|x thuộc N,2≤x≤99},i={1,2}
- Joint reward funtion, kí hiệu R: Là hàm trả về phần thưởng của mỗi người chơi tương ứng với mỗi hành động kết hợp a thuộc không gian A.

Xem mỗi hành động kết hợp a là một cặp tọa độ (a1, a2), thì phần thưởng r = (r1, r2) là cặp giá trị ghi trong ô đó với màu tương ứng.
Ví dụ:

R((100, 100)) = (100, 100)
R((99, 100)) = (101, 97)
…

	100	99	98	97	⋯	3	2
100	100, 100	97, 101	96, 100	95, 99	⋯	1, 5	0, 4
99	101, 97	99, 99	96, 100	95, 99	⋯	1, 5	0, 4
98	100, 96	100, 96	98, 98	95, 99	⋯	1, 5	0, 4
97	99, 95	99, 95	99, 95	97, 97	⋯	1, 5	0, 4
⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮
3	5, 1	5, 1	5, 1	5, 1	⋯	3, 3	0, 4
2	4, 0	4, 0	4, 0	4, 0	⋯	4, 0	2, 2

4.3.2. Phương pháp giải quyết

Như đã trình bày trong phần thách thức của bài toán, nếu biết được người còn lại đưa ra giá trị x thì giá trị tối ưu của người này đưa ra sẽ là max(2, x-1), tức ngoài trường hợp biên (x=2) thì người nào cũng muốn đưa ra con số nhỏ hơn của người kia đúng một đơn vị. Cả hai bên đều biết được chiến thuật này, nhưng lại đều không biết trước được quyết định của người còn lại. Do đó dẫn đến một số nhận xét như sau:

- Tồn tại điểm cân bằng Nash (Nash Equilibrium): Vì joint action space của trò chơi này là hữu hạn (kích thước 98x98) nên chắc chắn tồn tại điểm cân bằng Nash. Trạng thái cân bằng Nash là một chính sách chung trong đó không bên nào có động cơ đơn phươngchuyển đổi chính sách của mình. Ta có thể lập luận để tìm điểm này một cách dễ dàng như sau:
  So sánh phương án x = 100 và x = 99 từ góc nhìn của người chơi 1:
  - ❖ Nếu người chơi 2 đưa ra số tiền y thuộc [2, 98] thì việc chọn 100 hay 99 đều cho kết quả số tiền nhận về như nhau là y-2
  - ❖ Nếu người chơi 2 đưa ra số tiền y = 99: chọn 100 sẽ nhận được 97, trong khi chọn 99 sẽ nhận được 99

- ❖ Nếu người chơi 2 đưa ra số tiền y = 100: chọn 100 sẽ nhận được 100, trong khi chọn 99 sẽ nhận được 101

Từ 3 so sánh trên, rõ ràng đơn phương so sánh thì phương án chọn 99 luôn tốt hơn phương án 100 bất kể số tiền người kia đưa ra là bao nhiêu.

Lập luận tương tự, bài toán dẫn đến việc đưa ra quyết định lựa chọn x = 2, y = 2. Ở trạng thái này, không có người chơi nào có động cơ thay đổi quyết định của mình nữa. Quá trình suy luận trên được mô tả theo suy nghĩ của con người. Để có thể chuyển đổi và giải quyết trên các ngôn ngữ của máy tính, em lựa chọn phương án sử dụng mô hình Iterated Best Response để giải quyết bài toán. Mô hình này thực hiện lặp đi lặp lại các phản hồi tốt nhất qua một loạt các trò chơi. Trong mỗi phản hồi tốt nhất sẽ thực hiện xoay vòng giữa các người chơi để giải quyết chiến lược tốt nhất. Người ta chứng minh được khi số lượng vòng lặp đủ lớn, kết quả sẽ hội tụ về cân bằng Nash.

- Một cách tiếp cận khác đó là dựa trên sự kết hợp của Softmax Response và quá trình lặp qua các vòng lặp, gọi là Hierachical Softmax. Với Softmax Response, ta cho rằng các tác nhân thực hiện hành vi trong bài toán sẽ không hoàn toàn thực hiện hành vi tối ưu của mình như trên phân tích ở cân bằng Nash mà thường sẽ mắc lỗi trong quá trình này nếu những lỗi này không gây tốn kém quá nhiều. Tham số cho việc đưa ra quyết định của mô hình Softmax Response là λ với

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

giá trị λ càng gần về 0 thì quyết định càng mang tính ngẫu nhiên, và càng tiến về vô cùng thì kết quả hội tụ về gần với best response. Một điều quan trọng là λ có thể được học từ dữ liệu và cho phép dự đoán hành vi [1].

4.3.3. Code

Khai báo lớp SimpleGame gồm 4 thuộc tính cơ bản đã nêu trên: discount factor, agents, joint action space, joint reward function [1]

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

Khai báo lớp Travelers là một "lớp kế thừa" của SimpleGame [1]

```
struct Travelers end
```

Tạo các hàm phụ trợ để xác định các thuộc tính của đối tượng Travelers, tuân theo SimpleGame [2]

```
# agents
n_agents(simpleGame::Travelers) = 2

# joint action space
ordered_actions(simpleGame::Travelers, i::Int) = 2:100
```

```
ordered_joint_actions(simpleGame::Travelers) =
vec(collect(Iterators.product([ordered_actions(simpleGame, i) for i in
1:n_agents(simpleGame)]...)))
n_joint_actions(simpleGame::Travelers) = length(ordered_joint_actions(simpleGame))
n_actions(simpleGame::Travelers, i::Int) = length(ordered_actions(simpleGame, i))

# joint reward funtion
function reward(simpleGame::Travelers, i::Int, a)
    if i == 1
        noti = 2
    else
        noti = 1
    end
    if a[i] == a[noti]
        r = a[i]
    elseif a[i] < a[noti]
        r = a[i] + 2
    else
        r = a[noti] - 1
    end
    return r
end

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

Constructor cho đối tượng lớp Travelers [1]

```
function SimpleGame(simpleGame::Travelers)
    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
```

SimpleGamePolicy [1]

```
struct SimpleGamePolicy
    p # dictionary mapping actions to probabilities
    function SimpleGamePolicy(p::Base.Generator)
        return SimpleGamePolicy(Dict(p))
    end
    function SimpleGamePolicy(p::Dict)
        vs = collect(values(p))
        vs ./= sum(vs)
        return new(Dict(k => v for (k,v) in zip(keys(p), vs)))
    end
```

```
    SimpleGamePolicy(ai) = new(Dict(ai => 1.0))
end

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

function (πi::SimpleGamePolicy)()
    D = SetCategorical(collect(keys(πi.p)), collect(values(πi.p)))
    return rand(D)
end
joint(X) = vec(collect(Iterators.product(X...)))
joint(π, πi, i) = [i == j ? πi : πj for (j, πj) in enumerate(π)]

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

Mô hình đầu tiên được lựa chọn để giải quyết bài toán là Iterated Best Response, sử dụng mô hình cơ sở là Best Response [1]

```
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
```

Khai báo cấu trúc và định nghĩa Iterated Best Response cho một SimpleGame [1]

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

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

Triển khai hàm giải quyết bài toán SimpleGame bằng mô hình Iterated Best Response, dựa trên mô hình cơ sở là Best Response [1]

```
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
```

Sau khi khai báo và triển khai mã nguồn đầy đủ như trên, ta tiến hành quá trình chạy thử nghiệm và quan sát kết quả. Ở đây nhóm em thực hiện quan sát quyết định của các agents khi cho số vòng lặp thay đổi từ 1 đến 150.

```
𝒫 = SimpleGame(Travelers())
for i in 1:150
    print(string("k = "), i, ": ")
    println(solve(IteratedBestResponse(𝒫, i), 𝒫))
end
```

Mô hình thứ hai được lựa chọn là Hierrachical Softmax.

```
# Hierarchical Softmax
function softmax_response(𝒫::SimpleGame, π, i, λ)
    𝒜i = 𝒫.𝒜[i]
    U(ai) = utility(𝒫, joint(π, SimpleGamePolicy(ai), i), i)
    return SimpleGamePolicy(ai => exp(λ*U(ai)) for ai in 𝒜i)
    end

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

Khai báo cấu trúc và định nghĩa Hierachical Softmax cho một SimpleGame [1]

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

Triển khai hàm giải quyết bài toán SimpleGame bằng mô hình Hierachical Softmax, dựa trên mô hình cơ sở là Softmax Response [1]

```
function solve(M::HierarchicalSoftmax, 𝒫)
    π = M.π
    for k in 1:M.k
        π = [softmax_response(𝒫, π, i, M.λ) for i in 𝒫.ℐ]
    end
    return π
end
```

Chạy thử nghiệm với các giá trị k = 0,1,2,3,4 và λ = 0.1, 0.3, 0.5, kết hợp với trực quan hóa kết quả bằng biểu đồ

```
function visualization(λ, k)
    𝒫 = SimpleGame(Travelers())
    M = HierarchicalSoftmax(𝒫, λ, k)
    dict = solve(M, 𝒫)[1].p
    a = []
    b = []
    for i = 2:100
        push!(a, i)
        push!(b, dict[i])
    end
    im = plot(bar(x=a, y=b), Layout(title = string("k=",k, ", λ=", λ)))
```

```
    filename = string("k=",k, "_λ=", λ,".png")
    filename = joinpath(@__DIR__, filename)
    savefig(im, filename)
end

for k = 0:4
    for λ = [0.1, 0.3, 0.5]
        visualization(λ, k)
    end
end
```

4.3.4. Phân tích

a/ Mô hình Iterated Best Response

Tiến hành thực hiện thí nghiệm với số vòng lặp thay đổi từ 1 → 150, đáp ứng của agents giảm dần, đúng 1 đơn vị sau mỗi vòng lặp. Kết quả này phù hợp với việc thay đổi quyết định khi so sánh phương án x và phương án x-1. Qua mỗi vòng lặp mỗi agent nhận thấy đưa ra phương án x-1 là luôn tốt hơn phương án x.

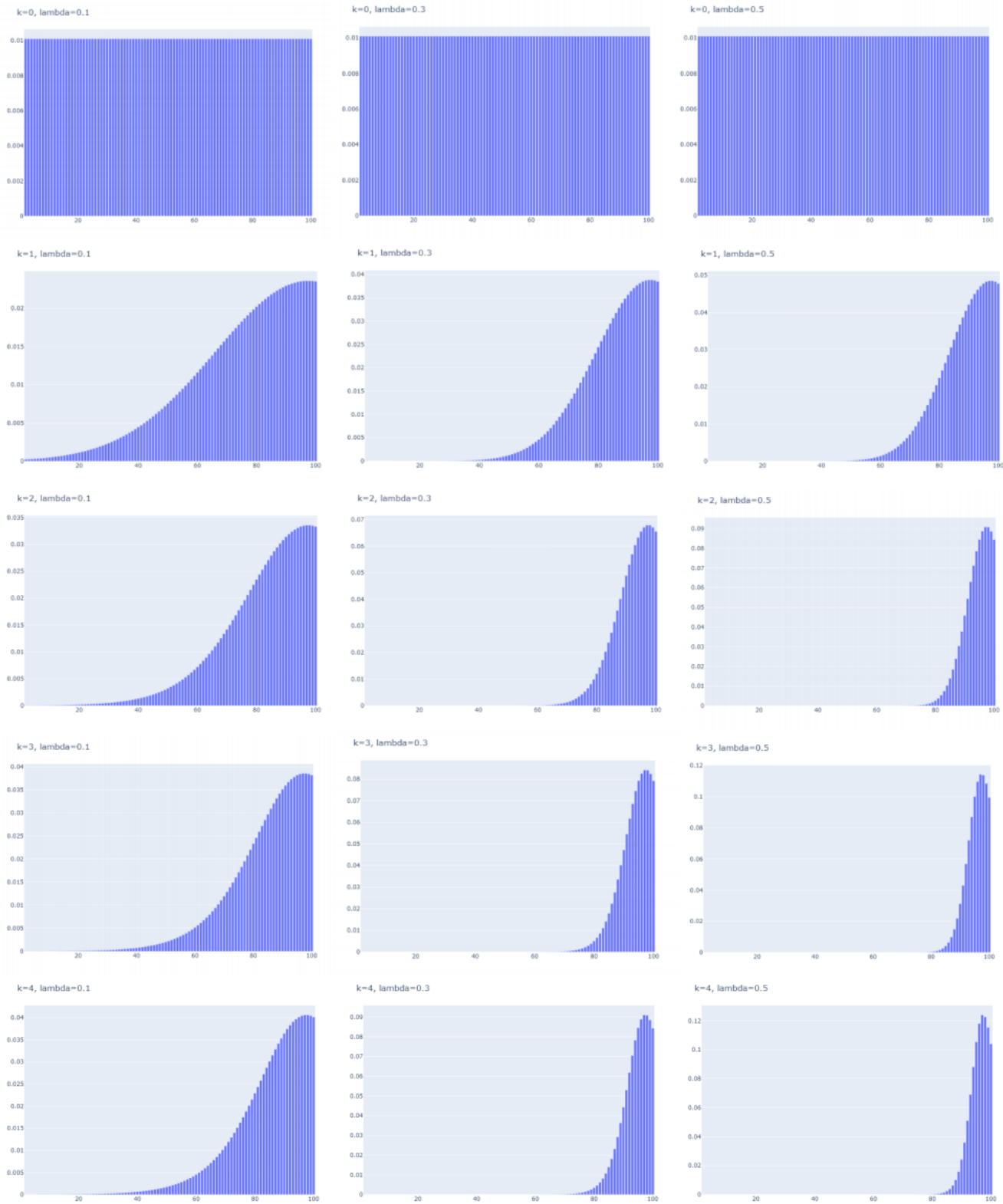
Ở các vòng lặp cuối, kết quả không thay đổi, giữ ở giá trị là 2, chứng tỏ việc lặp lại best_response đã hội tụ tại cân bằng Nash một cách tự động: cân bằng ở joint action a = (2,2) đem lại reward là r = (2,2). Kết quả này phù hợp với định nghĩa của cân bằng Nash, rằng các tác nhân không còn động cơ để đơn phương thay đổi quyết định của mình nữa.

Với kết quả trên, có thể thấy được ý nghĩa là việc lặp lại các vòng lặp sau khiến cho tác nhân thay đổi chiến thuật của mình thông qua việc so sánh với các quyết định trước đó. Điểm cân bằng Nash thực sự tồn tại đối với joint action space hữu hạn. Tuy nhiên điểm cân bằng này là kết quả của quá trình thay đổi chiến lược lặp đi lặp lại, theo phương diện so sánh kết cục một cách đơn phương, khiến cho mỗi tác nhân có xu hướng kéo mức tiền nhận được xuống thấp cho đến khi cân bằng và không còn động cơ thay đổi, chứ không đem lại một kết cục tốt mà lẽ ra hai bên có thể "thỏa thuận" đạt được là r = (100, 100).

b/ Mô hình Hierachical Softmax

Tiến hành thực hiện thí nghiệm với các bộ giá trị (k, λ) với k = 0, 1, 2, 3, 4 và λ = 0.1, 0.3, 0.5 và quan sát xác suất đưa ra các phương án của người chơi.

Biểu đồ thống kê kết quả của Hierachical Softmax cho bài toán Traveler's Dilemma



- Với k = 0, việc đưa ra quyết định của các agents gần như là ngẫu nhiên, do đó biểu đồ xác suất có giá trị khá tương đồng giữa tất cả các phương án thuộc [2, 100]
- Với k ≥ 1, việc đưa ra quyết định của một agent sẽ dựa vào quyết định của agent kia ở level trước đó (k - 1). Kết quả cho thấy xác suất lựa chọn sẽ lệch hẳn về phía khoảng giá trị gần 100, cụ thể xác suất cao nhất nằm ở giá trị 97.
  Ở mỗi level của k, khi λ có xu hướng tăng thì phân phối xác suất có xu hướng lệch phải rõ hơn, tức giảm nhanh xác suất các lựa chọn ở khu vực < 50 và tăng xác suất các lựa chọn ≥ 50, nhất là khu vực ≥ 80

Kết quả của thử nghiệm trên mô hình Hierachical Softmax này cho thấy việc thực tế các agents không chú trọng hoặc không biết đến điểm cân bằng Nash có thể dẫn tới những kết cục khác của bài toán. Ngoài ra, việc tồn tại nhiều điểm cân bằng Nash cũng như chi phí tính toán điểm cân bằng này khiến cho việc đưa ra quyết định theo điểm cân bằng này không thực sự hiệu quả. Hơn nữa, thực tế, thậm chí khi một agent tính toán được điểm cân bằng này thì cũng không chắc đối phương sẽ lựa chọn nó. Kết quả là khi thực hiện mô hình Hierachical Softmax Response, qua các level cao (k tăng), việc lựa chọn của cá nhân đã đưa tới một kết cục xấp xỉ (97,97), khả quan hơn nhiều so với cân bằng Nash khi đứng ở góc nhìn thứ 3.

4.4. Predator-Prey Hex World

4.4.1. Mô hình hóa tính toán

4.4.2. Phương pháp giải quyết

Trong thử nghiệm này, nhóm em dùng thuật toán Fictitious Play để giải quyết bài toán. Khi cho 2 người tham gia chơi trong một lượt chơi có số lượng bước đi là k_max, sau mỗi bước đi, mỗi người chơi phải dựa vào trạng thái hiện tại và đếm số lần tích lũy các hành động của đối phương để để đưa ra sự lựa chọn tốt nhất cho hiện tại. Người chơi cần học và cập nhật chiến lược của mình dựa trên các bước đi trước đó của đối phương.

Fictitious Play là một hướng tiếp cận bằng cách người chơi sẽ ước tính lựa chọn có khả năng xảy ra cao nhất của đối thủ. Mỗi người chơi sẽ tuân theo để tính toán được phân phối xác suất cho các hành động ở trạng thái hiện tại, từ đó tính toán được lợi ích và chi phí của việc chuyển đổi trạng thái và đưa ra lựa chọn hành động là câu trả lời tốt nhất dựa trên những ước tính này. Việc quyết định thử nghiệm với Fictitious Play cho bài toán này được cho là phù hợp nhất vì Fictitious Play mô phỏng tương tự như việc não người sẽ có xu hướng ghi nhớ các lựa chọn nhiều nhất mà đối phương đưa ra để điều chỉnh chiến lược của mình.

4.4.3. Code

Lớp MGFictitiousPlay gồm có các thuộc tính $\mathcal{P}$ là một Markov game, i là số thứ tự của người chơi, Qi là các ước tính giá trị của một hành động từ một trạng thái và Ni là các số lượng thực hiện một hành động từ một trạng thái. [1]

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

Đây là hàm tạo ra MGFictitiousPlay với tham số đầu vào là một MG và số thứ tự người chơi i. Qi được khởi tạo là phần thưởng của từng hành động tương ứng với từng trạng thái. Ni được khởi tạo là 1 với tất cả các hành động của từng trạng thái. [1]

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

Hàm train_fictitious_play dùng để thực hiện trò chơi Predator-Prey Hex World k_max bước đi. Đầu tiên là khởi tạo một Markov Game $\mathcal{P}$ và π là mảng hai MGFictitiousPlay predator và prey. Sau đó, gọi hàm simulate($\mathcal{P}$, π, k_max) để mô phỏng trò chơi.

```
function
train_fictitious_play(PPHW::DecisionMakingProblems.PredatorPreyHexWorldMG, k_max)
    𝒫 = MG(PPHW)
    π = [MGFictitiousPlay(𝒫, i) for i in 𝒫.ℐ]
    simulate(𝒫, π, k_max)
    π = [MGFPtoMGPolicy(𝒫, πi) for πi in π]
    return π
end
```

Hàm simulate sẽ thực hiện trò chơi $\mathcal{P}$ là một Markov game, theo cách chơi π được truyền vào là mảng các MGFictitiousPlay của các người chơi và sẽ di chuyển k_max bước. Đầu tiên, hàm sẽ khởi tạo trạng thái ngẫu nhiên s cho các người chơi. Với mỗi lần di chuyển, hàm sẽ gọi (πi::MGFictitiousPlay)(s) để tìm ra hành động tốt nhất tại trạng thái s của mỗi người chơi và lưu vào a. Sau đó dùng hàm randstep để trả về s′, r lần lượt là trạng thái mới và phần thưởng tương ứng sau khi hành động a. Tiếp đến, hàm sẽ thực hiện cập nhật chiến lược cho lần chơi kế tiếp bằng cách gọi hàm update!. Cuối cùng là gán trạng thái hiện tại là s′. [1]

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

(πi::MGFictitiousPlay)(s) được gọi trong simulate để chọn ra hành động tốt nhất tại trạng thái s.

Đầu tiên, hàm sẽ tính joint policy π là phân phối xác suất của các hành động tại trạng thái hiện tại. Sau đó, hàm tiến hành tính utility của người chơi tại trạng thái s theo công thức sau:

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

$R^i(s,\pi(s))$ là phần thưởng dành cho người chơi i tại trạng thái s theo joint policy π và được tính theo công thức:

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

$T(s'|s,\pi(s))$ là xác suất để chuyển từ trạng thái s sang s' khi áp dụng joint policy π và được tính theo công thức:

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

Cuối cùng, hàm sẽ trả về hành động ai là sự lựa chọn tốt nhất bằng hàm argmax. [1]

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

Hàm update! có nhiệm vụ cập nhật chiến lược cho MGFictitiousPlay khi người chơi thực hiện hành động trong a để chuyển từ trạng thái s sang s'. Đầu tiên, hàm tiến hành cập nhật số lượng lần thực hiện hành động trong a tại trạng thái s. Tiếp đến, hàm sẽ tính joint policy π là phân phối xác suất của các hành động tại trạng thái hiện tại. Sau đó, hàm tiến hành tính utility của người chơi tại trạng thái s. Cuối cùng, hàm tiến hành cập nhật giá trị Qi cho MGFictitiousPlay πi. [1]

```
function update!(πi::MGFictitiousPlay, s, a, s′)
    𝒫, i, Qi = πi.𝒫, πi.i, πi.Qi
    ℐ, 𝒮, 𝒜, T, R, γ = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.T, 𝒫.R, 𝒫.γ
    for (j,aj) in enumerate(a)
        πi.Ni[j,s,aj] += 1
    end
```

```
    πi'(i,s) = SimpleGamePolicy(ai => πi.Ni[i,s,ai] for ai in 𝒜[i])
    πi'(i) = MGPolicy(s => πi'(i,s) for s in 𝒮)
    π = [πi'(i) for i in ℐ]
    U(π,s) = sum(πi.Qi[s,a]*probability(𝒫,s,π,a) for a in joint(𝒜))
    Q(s,a) = R(s,a)[i] + γ*sum(T(s,a,s')*U(π,s') for s' in 𝒮)
    for a in joint(𝒜)
        πi.Qi[s,a] = Q(s,a)
    end
end
```

4.4.4 Phân tích

Tiến hành thử nghiệm với số vòng lặp k_max = 2000 thu được kết quả cho đầy đủ trạng thái , được trình bày tóm tắt trong các hình bên dưới.

Quy ước đánh số các vị trí trên bản đồ và hướng di chuyển

Hành động của predator và prey cho các trạng thái

	1	2	3	4	5	6	7	8	9	10	11	12
1	E E	E SE	E SW	SW SE	E SE	E NE	SW E	SW E	E E	E E	E NE	E NE
2	W SW	SE SW	E SW	W SE	SE SW	SE NE	SE E	W SW	SE W	SE E	SE NE	SE NE
3	W SW	W W	SW SE	W SE	SW SW	SW E	SE E	SW NW	SW W	SW E	SW E	SW NE
4	NE E	NE SE	NE E	NE E	SE SE	W E	W E	SE E	SE E	SE E	E E	W NE
5	NW SW	NW W	NW SW	NW W	SE W	SE E	SE E	SW NW	SW W	SE E	SE NE	SE NE
6	W NW	SW W	SW W	SW NE	SW NW	SW W	E E	SW W	SW W	SW W	SW W	SW NE
7	W NW	SW W	E E	SW SE	SW NE	W SW	W W	SW SW	SW W	SW W	SW W	SW W
8	NW E	E E	E E	NW NE	E NE	E E	E E	E E	E E	E E	E E	E NE
9	W NW	NE W	NE W	W NE	NE NW	E E	E E	W NW	E NW	E E	E NE	E NE
10	W W	NW W	NW W	W NW	NW NW	E E	E E	W NW	W W	NW NE	E NE	E NE
11	W W	W W	W W	W NW	W NW	NE E	E E	W NW	W W	W W	W NW	E NE
12	W NW	W W	W W	W NW	W NW	W E	NE SW	W W	W W	W W	W W	W W

Từ kết quả trên, ta có một số nhận xét về hành động của các agents như sau:

- Với predator agent: kết quả cho thấy predator đưa ra hầu hết các bước di chuyển có hướng đi theo xu hướng đuổi theo prey
- Với prey agent: các bước di chuyển ở từng trạng thái có xu hướng thoát khỏi predator. Trong một số trạng thái đặc biệt, ví dụ prey đang ở trong góc, prey còn có thể đưa ra quyết định đứng yên để chờ đợi các bước di chuyển tiếp theo của predator, hoặc chờ predator di chuyển lại ô gần sát để thực hiện nhảy tới đúng ô đó ở lượt tiếp theo.

Trong kết quả quan sát được bên trên, ngoài việc prey đợi predator tới gần để thực hiện chạy thoát vào đúng ô hiện tại của predator thì predator cũng "học" được điều này. Ví dụ ở trạng thái (6, 7), predator tính được xác suất cao rằng prey sẽ di chuyển vào ô 6 của mình trong lượt tiếp theo, nên hành động đưa ra của predator là đứng yên tại ô 6 này và đã bắt được prey.

Từ việc quan sát sự hội tụ của hành động của các agent và hướng di chuyển mô tả bên trên, mô hình Fictitious Play cho ra kết quả phù hợp với phản ứng của các đối tượng trong thực tế.

4.5 Multi-Caregiver Crying Baby

4.5.1 Mô hình hóa tính toán

Multi-Caregiver Crying Baby là bài toán thuộc loại Partially Observable Markov Game (POMG). Những người chơi cùng tham gia và quan sát để đưa ra cho mình những lựa chọn phù hợp để tối ưu được kết quả của bản thân.

Mỗi bài toán thuộc thể loại POMG sẽ được miêu tả bằng 8 thuộc tính sau:

- Discount factor, ký hiệu là γ ($0 \leq \gamma < 1$): thể hiện mức độ "cận thị" trong chiến thuật của người chơi. Với giá trị càng gần 0, chiến lược đưa ra nhằm giúp tối đa điểm thưởng ở các lượt gần tiếp theo, giá trị càng gần 1 thì chiến lực đưa ra nhằm giúp tối đa điểm thưởng ở tầm xa hơn.
- Agents, ký hiệu là $\mathcal{I}$: danh sách người chơi tham gia. Trong bài toán này danh sách người chơi sẽ là [1, 2].
- State Space, ký hiệu là $\mathcal{S}$: đây là danh sách các trạng thái của đối tượng cần quan sát. Trong bài toán này đó là trạng thái của em bé, gồm 2 trạng thái: ["HUNGRY", "SATED"].
- Joint action space, ký hiệu A: không gian hành động của tất cả người chơi. Trong trường bài toán này A = [["FEED", "SING", "IGNORE"], ["FEED", "SING", "IGNORE"]]
- Joint observation space, ký hiệu $\mathcal{O}$: không gian quan sát của những người chơi. Trong bài toán này $\mathcal{O}$ = [["CRYING", "QUIET"], ["CRYING", "QUIET"]]
- Transition function, ký hiệu là T: đây là hàm chuyển đổi hành động. Trong bài toán này chúng ta sẽ có một số hàm chuyển đổi hành động như sau:

  `T(sated | hungry,(feed, ⋆)) = T(sated | hungry,(⋆, feed)) = 100%`

  Khi em bé đói, nếu một trong hai người chơi cho ăn, thì tỷ lệ em bé no sẽ là 100%

  `T(hungry | hungry,(⋆, ⋆)) = 100%.`

Khi em bé đói nếu hai người chơi thực hiện một thao tác bất kỳ khác thao tác cho ăn thì em bé sẽ tiếp tục đói với tỷ lệ là 100%.

$$T(sated \mid sated, (\star, \star)) = 50\%$$

Khi em bé đói nếu hai người chơi thực hiện một thao tác bất kỳ khác thao tác cho ăn thì em bé sẽ có 50% tỷ lệ trở nên hết đói.

- Joint observation function, ký hiệu O: hàm quan sát chung của người chơi. Trong bài toán này hàm O được biểu diễn như sau:

$$O((cry, cry) \mid (sing, \star), hungry) = O((cry, cry) \mid (\star, sing), hungry) = 90\%$$

Khi em bé đang đói nếu một trong hai người chọn hành động là hát và người còn lại chọn hành động khác cho ăn thì có đến 90% sau đó em bé sẽ khóc.

$$O((quiet, quiet) \mid (sing, \star), hungry) = O((quiet, quiet) \mid (\star, sing), hungry) = 10\%$$

Khi em bé đang đói nếu một trong hai người chọn hành động là hát và người còn lại chọn hành động khác cho ăn thì chỉ có 10% em bé sẽ im lặng.

$$O((cry, cry) \mid (sing, \star), sated) = O((cry, cry) \mid (\star, sing), sated) = 0\%$$

Khi bé đang no, một trong hai người chọn hát và người còn lại chọn hành động khác cho ăn thì tỷ lệ em bé khóc sau đó là 0%.

$$O((cry, cry) \mid (\star, \star), hungry) = O((cry, cry) \mid (\star, \star), hungry) = 90\%$$

Khi em bé đang đói nếu cả hai người đều không thực hiện hành động cho ăn thì có đến 90% là em bé sẽ khóc.

$$O((quiet, quiet) \mid (\star, \star), hungry) = O((quiet, quiet) \mid (\star, \star), hungry) = 10\%$$

Khi em bé đang đói chỉ có 10% là em bé không khóc nếu như cả hai người chăm sóc đều không cho em bé ăn.

$$O((cry, cry) \mid (\star, \star), sated) = O((cry, cry) \mid (\star, \star), sated) = 0\%$$

Khi em bé đang no và cả hai người đều thực hiện hành động khác cho ăn thì có 0% tỷ lệ em bé sẽ khóc.

$$O((quiet, quiet) \mid (\star, \star), sated) = O((quiet, quiet) \mid (\star, \star), sated) = 100\%$$

Khi em bé đang no nếu thì tỷ lệ em bé im lặng khi cả hay thực hiện hành động khác cho ăn là 100%.

- Joint reward function, ký hiệu R: đây là hàm phần thưởng mà người chơi nhận được sau mỗi lượt. Chi tiết hàm này đã được chúng em đề cập ở mục phát biểu bài toán.

4.5.2 Phương pháp giải quyết

Tồn tại điểm cân bằng Nash (Nash Equilibrium): Vì joint action space của trò chơi này là hữu hạn nên chắc chắn tồn tại điểm cân bằng Nash. Trạng thái cân bằng Nash là một chính

sách chung trong đó không bên nào có động cơ đơn phương chuyển đổi chính sách của mình. Ta có thể lập luận để tìm điểm này một cách dễ dàng như sau.

Nếu ta giải chỉ đơn thuần giải quyết bài toán bằng cách sinh ra cây trạng thái hoàn chỉnh, thì để đi được đến điểm cân bằng, chúng ta sẽ mất rất nhiều thời gian và chi phí. Vậy nên ta sẽ sử dụng phương pháp Dynamic Programming để có thể giải bài toán này.

Ở thuật toán Dynamic Programming, khi sinh ra mỗi tầng mới của cây trạng thái, ta sẽ sử dụng công thức để kiểm tra và lược bỏ những nhánh bị "nuốt trọn" bởi một nhánh khác. Ta policy $\boldsymbol{\pi}^i$bị thống trị bởi $\boldsymbol{\pi}^{i\prime}$ nếu không tồn tại b($\boldsymbol{\pi}^{-i}$,s) giữa các policy chung khác $\boldsymbol{\pi}^{-i}$ và trạng thái s:

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

4.5.3 Code

Đầu tiên chúng ta cần định nghĩa một số struct cần thiết như: POMG, POMGDynamicProgramming, ConditionalPlan, SimpleGame, SimpleGamePolicy. Tiếp theo chúng ta sẽ định nghĩa các thành phần cần thiết của struct POMG là struct quan trọng nhất để định nghĩa toàn bộ bài toán.Sau khi đã định nghĩa đủ các struct và hàm cần thiết, tiếp theo chúng ta sẽ đi đến bước định nghĩa các hàm solve để giải quyết bài toán và một số hàm khác liên quan như is_dominated, prune_dominated! để xác định các policy cần được lược bỏ.

Chi tiết chức năng các hàm sẽ được comment rõ trong file code đính kèm.

4.5.4 Phân tích kết quả

Vì bài toán có yếu tố ngẫu nhiên ở phần chuyển đổi trạng thái của em bé, nên nhóm chúng em đã quyết định chạy thử bài toán nhiều lần với số lượng tầng của cây trạng thái lần lượt là 2 và 3.

Trong cả hai giá trị của d, đều tồn tại giá trị cân bằng nash. Sau khi chạy nhiều lần ở các trường hợp d, chúng em nhận thấy được chiến thuật tối ưu nhất để dành chiến thắng là sử dụng hành động bỏ qua ở tất cả các lượt, để chờ người còn lại thực hiện hành động khác bỏ qua và chịu nhiều điểm penalty hơn.

5. Tóm tắt kết quả

5.1. Hex world

Hex world là một MDP đơn giản, nhóm em đã sử dụng 2 phương pháp đơn giản để giải quyết bài toán để đưa ra được các đánh giá tốt về các thuật toán này trong việc xử lý các bài toán MDP.

Sau quá trình thực nghiệm, nhóm chúng em nhận thấy mô hình bài toán hội tụ nhanh và chính xác trong các trường hợp không gian trạng thái nhỏ. Các phương pháp được sử dụng trong bài báo cáo này phù hợp hơn cho việc hiểu bài toán và là tiền đề cho các phương pháp khác chứ không phù hợp để giải quyết các bài toán lớn, khiaáy cần các thuật toán cải tiến hơn.

5.2. Rock-Paper-Scissors

Rock-Paper-Scissors là trò chơi đơn giản có tổng bằng không (zero-sum game) với sự tham gia của nhiều người chơi. Fictitious Play là một thuật toán hiệu quả để áp dụng vào trò chơi Rock-Paper-Scissors. Thời gian chạy khá nhanh với ngôn ngữ Julia. Ta thấy model sẽ có xu hướng hội tụ về chiến lược ngẫu nhiên Nash Equilibrium qua k_max lần chạy. Và khi k_max tiến tới vô cùng thì model sẽ hội tụ về $\frac{1}{3}$. Trong khi đó, điểm số của 2 người chơi lại phân kỳ.

5.3. Traveler's Dilemma

Traveler's dilemma là một trò chơi đa tác nhân với tổng khác không, thuộc lớp SimpleGame. Mục tiêu của mỗi tác nhân là tìm chiến thuật để đưa quyết định nhằm tối đa số tiền mình có thể nhận.
Với không gian hành động kết hợp hữu hạn, trò chơi tồn tại điểm cân bằng Nash là a = (2,2) với r = (2,2), điều này đã thể hiện qua kết quả hội tụ ở thực nghiệm chạy mô hình Iterated Best Response. Kết quả này là phù hợp với chiến thuật thay đổi lựa chọn qua mỗi vòng lặp: đưa ra mức giá thấp đi 1 đơn vị để gia tăng lợi ích cá nhân, nhưng như đã dự đoán về mặt lý thuyết, đây là một kết cục rất không tốt. $101 mới là giá trị cao nhất có thể nhận được, kết cục (100, 100) chưa đưa tới 101, vì động cơ thay đổi chiến thuật mà mỗi tác nhân có xu hướng kéo kết cục chung về mức thấp.
Với cách tiếp cận khác, mô phỏng hành vi của con người, không quá chú trọng trong việc tính toán điểm tối ưu, sử dụng mô hình Softmax Response, kết quả phân phối xác suất thực nghiệm chỉ ra rằng các tác nhân có xu hướng chọn các giá trị cao, xung quanh mức 97 (cao nhất). Kết quả này phù hợp với kết cục chung đứng từ góc nhìn thứ 3 và mô phỏng hợp lý hành vi chấp nhận sai sót nhỏ trong quá trình tối ưu.

5.4. Predator-Prey Hex world

Predator-Prey Hex world là một trò chơi đa tác nhân thuộc lớp Markov Game. Các tác nhân tham gia trò chơi với hàm số phần thưởng khác nhau và mục tiêu đều muốn tối đa hóa lợi ích nhận được. Trong trò chơi, các agents sẽ đưa ra chiến lược dựa trên trạng thái hiện tại chứ không phụ thuộc vào các hành động và trạng thái ở quá khứ [1].

Để giải quyết bài toán, nhóm em đã tiếp cận theo cách sử dụng tổng quát mô hình Fictitious Play, theo đó, agents sẽ đưa ra quyết định bằng cách tính toán phản hồi tốt nhất dựa vào hành động có khả năng xảy ra cao nhất của agent khác (maximum-likelihood model). Kết quả chạy thực nghiệm chương trình cho thấy kẻ đi săn có xu hướng di chuyển theo con đường ngắn nhất để tới vị trí con mồi, còn con mồi có xu hướng chạy về góc và chờ đợi kẻ đi săn di chuyển lại gần sát và sau đó đổi hướng. Kết quả này phù hợp với các hành động quan sát được trong thực tế.

5.5 Multi-Caregiver Crying Baby

Multi-Caregiver Crying Baby là một phiên bản mở rộng của bài toán Crying baby. Đây là một bài toán mang tính đối kháng, mỗi người chơi tham gia đều phải tìm ra cho mình một chiến thuật thật hợp lý để có thể giành cho mình một kết quả có lợi. Với không gian hành động hữu hạn, trò chơi tồn tại điểm cân bằng Nash (điểm cân bằng nash là khác nhau tùy vào số lượt chơi). Sau nhiều lần thực nghiệm, chúng em nhận ra được hành động bỏ qua là một chiến thuật tốt để có thể chiến thắng trò chơi. Vì khi đó nếu người chơi còn lại không thực hiện hành động cho ăn và chịu thêm điểm penalty thì cả hai sẽ phải chịu một lượng điểm trừ lớn vì đã để em

bé đói. Điều này vô hình chung làm lệch đi mục đích ban đầu của bài toán là chăm sóc em bé thật tốt.

6. Tự đánh giá

Qua quá trình tự tìm hiểu và thực hiện các yêu cầu được giao, nhóm em có những phần tự đánh giá như sau:

- Đã hiểu cơ bản về phát biểu của các bài toán đưa ra, các thành phần trong bài toán và xác định được yêu cầu cần giải quyết cho từng bài toán
- Đã tìm hiểu được về các dạng bài toán đưa ra quyết định như Simple Game, Markov Game,… và các thuộc tính chung của từng dạng bài toán.
- Tìm hiểu và nắm được các ý tưởng trong mô hình hóa các bài toán.
- Nhóm em đã lên kế hoạch và phân bổ thời gian đọc tài liệu, tìm thêm các nguồn kiến thức trên internet để cùng trao đổi và hoàn thành các yêu cầu, hiểu được

Bên cạnh những nội dung đã hoàn thành, trong quá trình thực hiện đồ án, một số khó khăn mà nhóm gặp phải là:

- Học và thao tác trên ngôn ngữ mới: Julia
- Các mô hình để giải quyết cho từng loại bài toán khá nhiều và có nhiều khái niệm, ý tưởng mới, cần nhiều thời gian tìm hiểu

7. Tài liệu tham khảo

[1] Mykel J. Kochenderfer, Tim A. Wheeler, Kyle H. Wray. *Algorithms for Decision Making*, Massachusetts Institute of Technology, 2022

[2] DecisionMakingProblems.jl Repository on Github
https://github.com/algorithmsbooks/DecisionMakingProblems.jl.git