# I - Phát biểu bài toán: mô tả các bài toán cần giải quyết.

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 Kéo, Búa hoặc Bao. 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:

* Búa thắng Kéo.
* Kéo thắng Bao.
* Bao thắng Búa.

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. Cụ thể, ta có ma trận phần thưởng của trò Kéo - Búa - Bao 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 |

# II - Thách thức: khó khăn, thách thức cụ thể đối với từng bài toán.

# III - Thực nghiệm:

## **MÔ HÌNH HÓA TÍNH TOÁN (CẤU TRÚC DỮ LIỆU/ PHƯƠNG PHÁP THỂ HIỆN)**

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

Graphical user interface, text

Description automatically generated

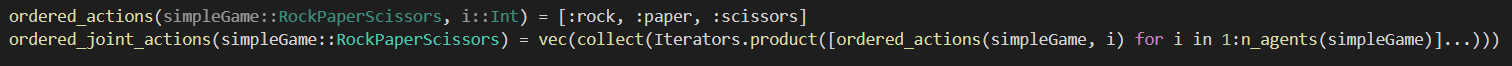
* *agents:* Số lượng người chơi là 2

Text

Description automatically generated

* *joint action space:* A = A1 \* A2 \* … \* Ak : gồm các hành động có thể có của các người chơi. Cụ thể trong trò chơi Kéo – Búa – Bao sẽ gồm 3 hành động là Rock, Paper, Scissors.

A = R \* P \* S



* *joint action:* a = (a1, a2,…, ak) : Các hành động được chọn đồng thời giữa các người chơi kết hợp lại. (a1= (R, P); a2=(C, C); …)
* *joint reward function:* R(a) = (R­1(a), . . . , Rk(a)) : Điểm số của *joint action ak*. Dựa theo ma trận điểm.

Text

Description automatically generated

* *joint policy :* là xác suất của các *joint action* thực hiện bởi người chơi. Xác suất agent i chọn hành động a là i(a). Tiện ích của *joint policy* từ góc nhìn của agent i là:

Text

Description automatically generated

Graphical user interface

Description automatically generated with low confidence

A policy associated with an agent is represented by a dictionary that maps actions to probabilities. There are different ways to construct a policy. One way is to pass in a dictionary directory, in which case the probabilitiesare normalized. Another way is to pass in a generator that creates this dictionary. We can also construct a policy by passing in an action, in which case it assigns probability 1 to that action. If we have an individual policy πi, we can call πi(ai) to compute the probability the policy associates with action ai. If we call πi(), then it will return a random action according to that policy. We can use joint(𝒜) to construct the joint action space from 𝒜. We can use utility(𝒫, π, i) to compute the utility associated with executing joint policy π in the game 𝒫 from the perspective of agent i.

## **PHƯƠNG PHÁP GIẢI QUYẾT (THUẬT TOÁN)**

### ***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.

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**

**Paper 1/3**

**Scissors 1/3**

**Rock 1/3**

|  |  |  |
| --- | --- | --- |
| 0,0 | -1,1 | 1,-1 |
| 1,-1 | 0,0 | -1,1 |
| -1,1 | 1,-1 | 0,0 |

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

Ui() =

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.

Text

Description automatically generated

Kết quả trả về là một cân bằng Nash của trò chơi



Tuy nhiên, chúng ta sẽ không bàn tới cách cài đặt này ở đây vì phương pháp này tốn chi phí cao

Để tính toán một cân bằng Nash, ta sẽ dùng một cách tiếp cận khác tiết kiêm về mặt tính toán hơn là Iterated Best Response.

### ***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.

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

Cấu trúc của *Iterated best response*

struct IteratedBestResponse

    k\_max # số lượng vòng lặp

    π # policy ban đầu

end

Thuận toán bắt đầu với một vài policy ban đầu và dừng lại sau k\_max lần lặp:

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

Để thuật tiện, chúng ta lấy policy đầu vào là mỗi agent sẽ chọn hành động Kéo, Búa hoặc Bao một cách ngẫu nhiên đồng nhất.

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

Kết quả:

Begin solving Iterated Best Response...

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

Done

### ***Hierachical Softmax***

Một lĩnh vực được gọi là lý thuyết trò chơi hành vi nhằm mục đích mô hình hóa các hành vi của con người. Khi xây dựng các hệ thống ra quyết định phải tương tác với con người, việc tính toán cân bằng Nash không phải lúc nào cũng hữu ích. Con người thường không chơi chiến lược cân bằng Nash. Trước hết, có thể không rõ nên áp dụng điểm cân bằng nào nếu có nhiều điểm cân bằng khác nhau trong trò chơi Rock-Paper-Scissors. Ngay cả khi có thể tính toán cân bằng Nash, họ có thể nghi ngờ rằng đối thủ của họ có thể thực hiện phép tính đó hay không ?

Có nhiều mô hình hành vi khác nhau, nhưng một cách tiếp cận là kết hợp cách tiếp cận lặp lại từ phần trước với một mô hình softmax. Phương pháp tiếp cận softmax phân cấp này mô hình hóa độ sâu hợp lý của agent theo cấp k ≥ 0. Agent cấp 0 thực hiện các hành động của nó một cách ngẫu nhiên đồng nhất. Agent cấp 1 giả định những agent khác áp dụng chiến lược cấp 0 và chọn các hành động theo phản ứng softmax với độ chính xác λ.

Tóm lại, một người chơi cấp k chọn các hành động theo mô hình softmax của những người chơi khác đang chơi cấp độ (k - 1).

Cấu trúc của thuật toán: mô hình softmax phân cấp với tham số chính xác λ và cấp k. Theo mặc định, nó bắt đầu với một policy chung ban đầu là xác suất đồng nhất cho tất cả các hành động riêng lẻ.

struct HierarchicalSoftmax

    λ # precision parameter

    k # level

    π # initial policy

end

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

    π = [SimpleGamePolicy(ai => 1.0 for ai in 𝒜i) for 𝒜i in 𝒫.𝒜]

    return HierarchicalSoftmax(λ, k, π)

end

Ở hàm solve, ta áp dụng softmax\_response cho các agent theo phản hồi các agent trước đó.

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

    println(π)

    return π

end

function softmax\_response(𝒫::SimpleGame, π, i, λ)

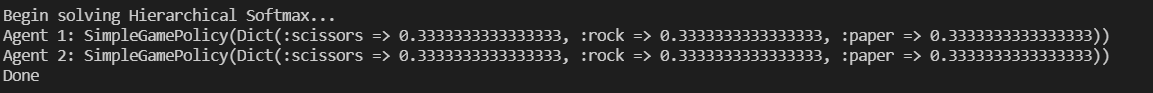
    𝒜i = 𝒫.𝒜[i]

    U(ai) = utility(𝒫, joint(π, SimpleGamePolicy(ai), i), i)

    return SimpleGamePolicy(ai => exp(λ \* U(ai)) for ai in 𝒜i)

end

Kết quả ta thu được hội tụ về một cân bằng Nash



### ***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 aj, lưu trữ nó trong bảng Ni(j, aj). 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)

Text

Description automatically generated

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.

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* 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

o Nêu rõ cấu hình các bài toán thực nghiệm. o Với mỗi thử nghiệm:

 Mô hình hóa tính toán: cấu trúc dữ liệu hoặc phương pháp thể hiện các bài

toán.

 Phương pháp giải quyết: phương pháp, thuật toán được sử dụng cho việc giải

quyết bài toán, lý do lựa chọn phương pháp.

 Code: hướng dẫn và mô tả về đoạn code mà bạn đã viết (hay sử dụng). Lưu ý:

cần ghi rõ nguồn (nếu bạn sao chép đoạn code ở đâu đó).

 Phân tích: kết quả thử nghiệm và ý nghĩa.

# IV - Tóm tắt kết quả:

o Dựa trên các tiêu chí đánh giá kết quả của bạn.

o Điểm mạnh và điểm yếu trong đồ án này của bạn.

***Hierarchical Softmax:***

Chúng ta có thể tìm hiểu các tham số k và λ của mô hình hành vi này từ dữ liệu. Nếu chúng ta có một tập hợp các *joint action* được thực hiện bởi các người chơi khác nhau, chúng ta có thể tính toán khả năng liên quan cho một k và λ nhất định. Sau đó, chúng ta có thể sử dụng một thuật toán tối ưu hóa để cố gắng tìm k và λ tối đa hóa khả năng xảy ra. Việc tối ưu hóa này thường không thể được thực hiện bằng phân tích, nhưng chúng tôi có thể sử dụng các phương pháp số để thực hiện tối ưu hóa này. Ngoài ra, chúng ta có thể sử dụng phương pháp tiếp cận Bayes để học tham số