# 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 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.

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

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

Trong trò chơi zero-sum này, các mô hình ta áp dụng sẽ hội tụ về cân bằng Nash nên cần đmả bảo mô hình đủ tốt.

# 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:

* Bài toán có 2 người chơi, là 2 agents = {1, 2}
* discount factor
* *joint action space:* A = A1 \* A2 \* … \* Ak : 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 × A2 với A*i* = {rock, paper, scissors}

* *joint action:* a = (a1, a2,…, ak) : Các hành động được chọn đồng thời giữa các người chơi kết hợp lại. Ví dụ: (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.
* *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

* 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 |

## **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. 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.

Calendar

Description automatically generated

**Paper 1/3**

**Scissors 1/3**

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 và cài đặt phức tạp

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.

Thuật giải:

Text

Description automatically generated

Kết quả: 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))]

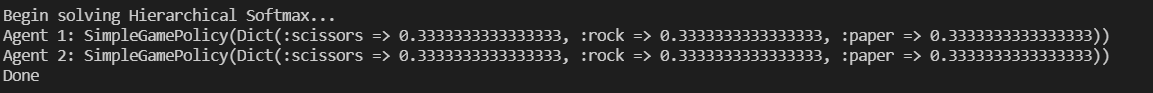
SimpleGamePolicy[SimpleGamePolicy(Dict(:scissors => 1.0)), SimpleGamePolicy(Dict(:scissors => 1.0))]

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

A picture containing text, stationary, pencil

Description automatically generated



### ***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.

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

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 biểu đồ về policy của 2 người chơi:

Chart

Description automatically generated

Chart

Description automatically generatedTa 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. Do đó, 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

## **MÔ TẢ CODE**

#### **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 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.

Text

Description automatically generated

* *reward*: phần thưởng nhận được của agent i sau khi hành động a được lựa chọn.
* *joint\_reward*: trả ra 1 list phần thưởng [r1, r2, …] với 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)

#### **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

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

* *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

#### **Fictitious Play:**

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

# IV - Tóm tắt kết quả:

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

* Mô tả bài toán: 9/10
* Mô hình hóa bài toán: 9/10
* Coding: 7/10
* Báo cáo: 8/10

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

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

Điểm yếu:

* Coding chưa tận dụng khả năng của Julia trong vẽ biểu đồ