Báo cáo của các bạn nên được trình bày với các nội dung như sau:

# Thông tin nhóm: MSSV, họ và tên các thành viên trong nhóm (theo thứ tự MSSV).

19120685 – Võ Ngọc Tín

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

## **Sơ lược về bài toán crying baby**

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

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

Hàm transision có các thỏa mãn sau:

Hungry + feed => stated 100%

Hungry + sing => hungry 100%

Hungry + ignore => hungry 100%

Stated + feed => stated 100%

Stated + sing => hungry 10%

Stated + ignore => hungry 10%

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

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

Hungry + feed => cry 80%

Hungry + sing => cry 90%

Hungry + ignore => cry 90%

Stated + feed => cry 10%

Stated + sing => cry 0%

Stated + ignore => cry 10%

Lưu ý: cry + quiet = 100% =>

Cry = 10% ⬄ quiet = 90%

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

Hungry độc lập với các actions khác => -10

Feed => -5

Sing => -0.5

## **Bài toán multicare giver crying baby**

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

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

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

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

**TRANSITION**

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

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

\*: tất cả action gôm FEED/ SING/ IGNORE

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

Nếu action không phải feed

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

**OBSERVATIONS**

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

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

\*: tất cả các action

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

Nếu action không phải sing, thì observations như sau:

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

**REWARD**

Người thứ nhất sở trường là feed:

Người thứ hai sở trường là sing:

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

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

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

# **III - Thực nghiệm:**

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

POMG là một dạng mở rộng của MGs có partial observability và mở rộng của POMDPs với nhiều agent. Có các agent i I = {1,2}, mỗi agent I chọn một action ai Ai = {feed, sing, ignore}, dựa trên phần quan sát được observation oi. Transition từ state s tới state s’ dựa trên action a T (s’| s,a). Joint reward dựa trên reward thực hiện action theo state Ri =(s,a). Mỗi agent cố gắng thực hiện sao cho đạt được reward cho họ là cao nhất. Sau khi các agent thực hiện joint action a, thì sẽ nhận được joint observation o = (o1 , ….. ,ok) space O = O1 X …. X Ok. Mỗi agent thì nhận được một observation oi của riêng họ từ joint observation. Theo như bài toán thì mỗi agent đều sẽ suy luận về suy nghĩ của người khác và chọn những action cho riêng mình.

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

**Cấu trúc POMG gồm:**

struct POMG

γ # discount factor

ℐ # agents

𝒮 # state space

𝒜 # joint action space

𝒪 # joint observation space

T # transition function

O # joint observation function

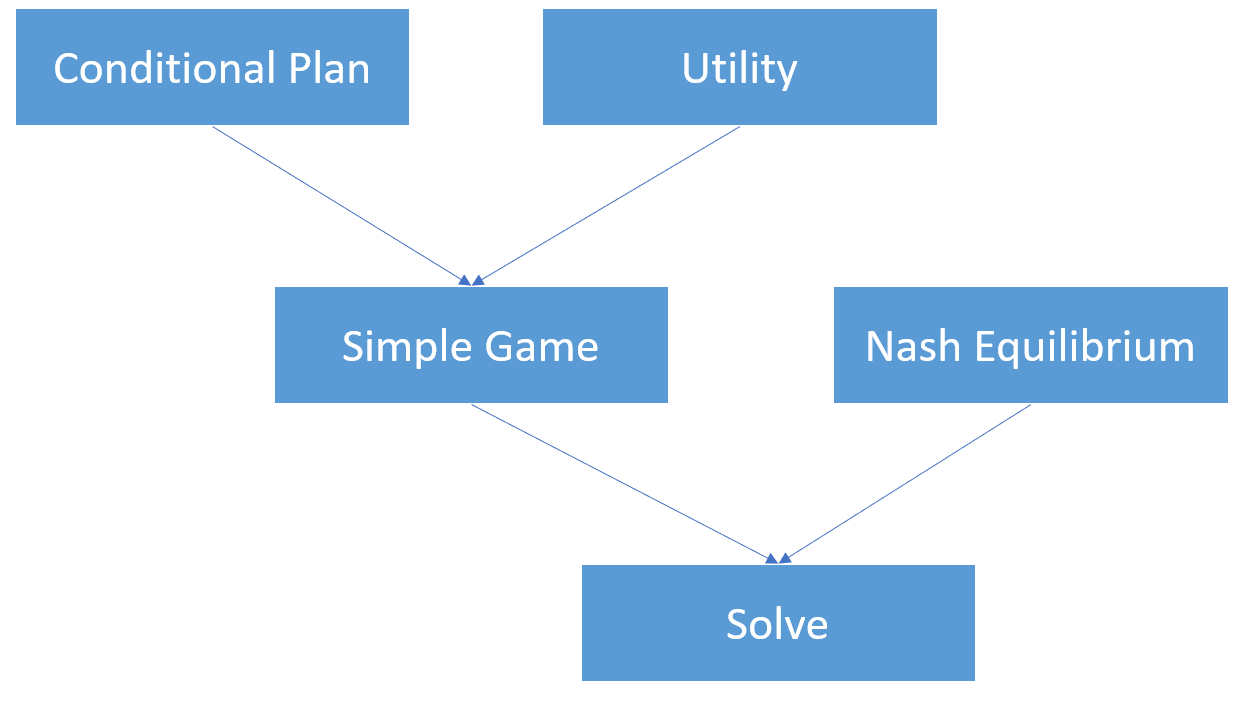
R # joint reward function

end

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

***Ý tưởng tổng quan:***

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



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

Theo như bài toán, sẽ có 2 agent I I = {1,2} với 3 action Ai = {ai1, ai2,ai3} = {feed, sing, ignore} và 2 observation Oi = { oi1, oi2} = {crying, quiet}.

Hướng giải quyết bài toán theo các bước như sau:

### **XÂY DỰNG CONDITIONAL PLAN**

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

*##code Khai báo cấu trúc 1 conditional plan*

struct ConditionalPlan

a # action được chọn là root

subplans # dictionary để ánh xạ từ các observation thành các subplan

end

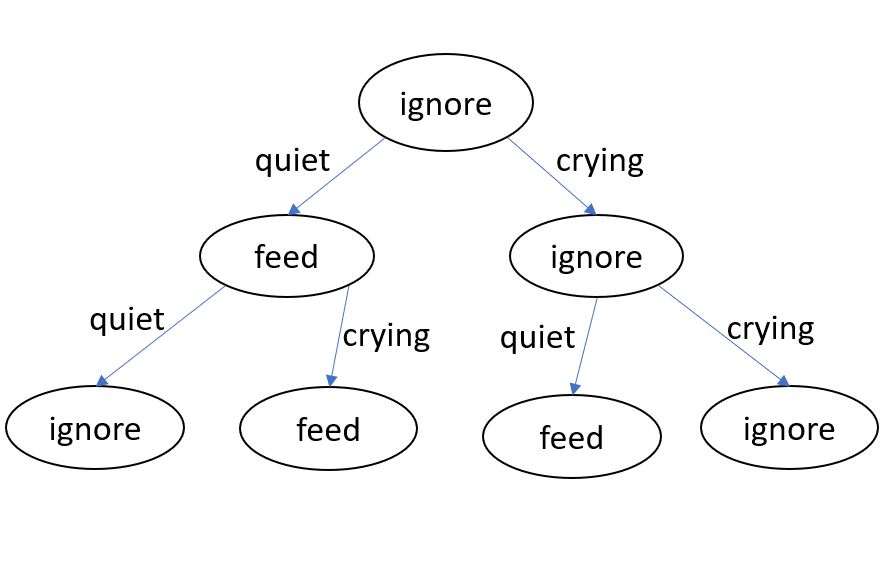
ConditionalPlan(a) = ConditionalPlan(a, Dict())

(π::ConditionalPlan)() = π.a

(π::ConditionalPlan)(o) = π.subplans[o]

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

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



# code Tạo conditional plan bằng cách tạo và mở rộng dần theo độ cao d của cây

function create\_conditional\_plans(𝒫: POMGS, d)

# ℐ là các agents

# 𝒜 là các actions

# 𝒪 là các observations

# Các tham số lấy từ bài toán POMG

ℐ, 𝒜, 𝒪 = 𝒫.ℐ, 𝒫.𝒜, 𝒫.𝒪

# Tạo Conditional Plan cho từng action rồi đến từng agent

Π = [ [ConditionalPlan(ai) for ai in 𝒜[i]] for i in ℐ]

# Ứng với chiều cao của Conditional Plan tree đã định trước, mở rộng ra condition plan

for t in 1:d

Π = expand\_conditional\_plans(𝒫, Π)

end

return Π

end

# Hàm mở rộng conditional plan

function expand\_conditional\_plans(𝒫, Π)

# Các tham số lấy từ bài toán POMG

ℐ, 𝒜, 𝒪 = 𝒫.ℐ, 𝒫.𝒜, 𝒫.𝒪

# Trong từng observation -> từng πi => tạo thành 1 dictionary

# Trong từng action, ứng với một Π[i], mỗi πi trong đó sẽ tạo thành 1 dictionary phía trên, tạo ra Conditional plan

#Tạo conditional plan cho mỗi agent

return [ [ConditionalPlan(ai, Dict(oi => πi for oi in 𝒪[i] )) for πi in Π[i] for ai in 𝒜[i] ] for i in ℐ]

end

### **XÂY DỰNG UTILITY**

Đầu tiên, chúng ta cần tính hàm đệ quy joint utility U𝜋 theo công thức:

A screenshot of a computer

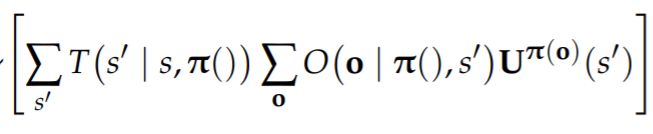
Description automatically generated with low confidence

Với:

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

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

# Hàm lookaheah thực hiện tính phần trong dấu ngoặc



function lookahead(𝒫::POMG, U, s, a)

𝒮, 𝒪, T, O, R, γ = 𝒫.𝒮, joint(𝒫.𝒪), 𝒫.T, 𝒫.O, 𝒫.R, 𝒫.γ

u′ = sum(T(s,a,s′)\*sum(O(a,s′,o)\*U(o,s′) for o in 𝒪) for s′ in 𝒮)

return R(s,a) + γ\*u′

end

# Thực hiện tính function utility theo công thức:

A screenshot of a computer

Description automatically generated with low confidence

function evaluate\_plan(𝒫::POMG, π, s)

a = Tuple(πi() for πi in π)

U(o,s′) = evaluate\_plan(𝒫, [πi(oi) for (πi, oi) in zip(π,o)], s′)

return isempty(first(π).subplans) ? 𝒫.R(s,a) : lookahead(𝒫, U, s, a)

end

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

A picture containing text, watch

Description automatically generated

function utility(𝒫::POMG, b, π)

u = [evaluate\_plan(𝒫, π, s) for s in 𝒫.𝒮]

return sum(bs \* us for (bs, us) in zip(b, u))

end

### **TỪ CONDITIONAL PLAN VÀ UTILITY CHUYỂN VỀ DẠNG SIMPLE GAME VÀ GIẢI QUYẾT BÀI TOÁN THEO NASH EQUILIBRIUM**

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

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

struct SimpleGame

γ # discount factor

ℐ # agents

𝒜 # joint action space

R # joint reward function

end

struct SimpleGamePolicy

p # dictionary mapping actions to probabilities

function SimpleGamePolicy(p::Base.Generator)

return SimpleGamePolicy(Dict(p))

end

function SimpleGamePolicy(p::Dict)

vs = collect(values(p))

vs ./= sum(vs)

return new(Dict(k => v for (k,v) in zip(keys(p), vs)))

end

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

end

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

function (πi::SimpleGamePolicy)()

D = SetCategorical(collect(keys(πi.p)), collect(values(πi.p)))

return rand(D)

end

Tiến hành cài đặt Nash Equilibrium của Simple Game

# Thuật toán NashEquilibrium của Simple game

function solveNE(M::NashEquilibrium, 𝒫::SimpleGame)

ℐ, 𝒜, R = tensorform(𝒫)

model = Model(Ipopt.Optimizer)

@variable(model, U[ℐ])

@variable(model, π[i=ℐ, 𝒜[i]] ≥ 0)

@NLobjective(model, Min,

sum(U[i] - sum(prod(π[j,a[j]] for j in ℐ) \* R[y][i]

for (y,a) in enumerate(joint(𝒜))) for i in ℐ))

@NLconstraint(model, [i=ℐ, ai=𝒜[i]],

U[i] ≥ sum(

prod(j==i ? (a[j]==ai ? 1.0 : 0.0) : π[j,a[j]] for j in ℐ)

\* R[y][i] for (y,a) in enumerate(joint(𝒜))))

@constraint(model, [i=ℐ], sum(π[i,ai] for ai in 𝒜[i]) == 1)

optimize!(model)

πi′(i) = SimpleGamePolicy(𝒫.𝒜[i][ai] => value(π[i,ai]) for ai in 𝒜[i])

return [πi′(i) for i in ℐ]

end

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

struct POMGNashEquilibrium

b # initial belief

d # Chiều sâu của conditional plans

end

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

# Các agent ℐ, tỷ suất chiết khấu γ, belief khởi tạo b và chiều cao của conditional plan d

ℐ, γ, b, d = 𝒫.ℐ, 𝒫.γ, M.b, M.d

# Tạo conditional plan cho từng agent (ở bài toán này có 2 agent)

Π = create\_conditional\_plans(𝒫, d)

# Tinh utility

U = Dict(π => utility(𝒫, b, π) for π in joint(Π))

# Từ conditional plan và utility, chuyển về Simple game

𝒢 = SimpleGame(γ, ℐ, Π, π -> U[π])

# Giải quyết bài toán theo NashEquilibrium của simple game

π = solve(NashEquilibrium(), 𝒢)

return Tuple(argmax(πi.p) for πi in π)

end

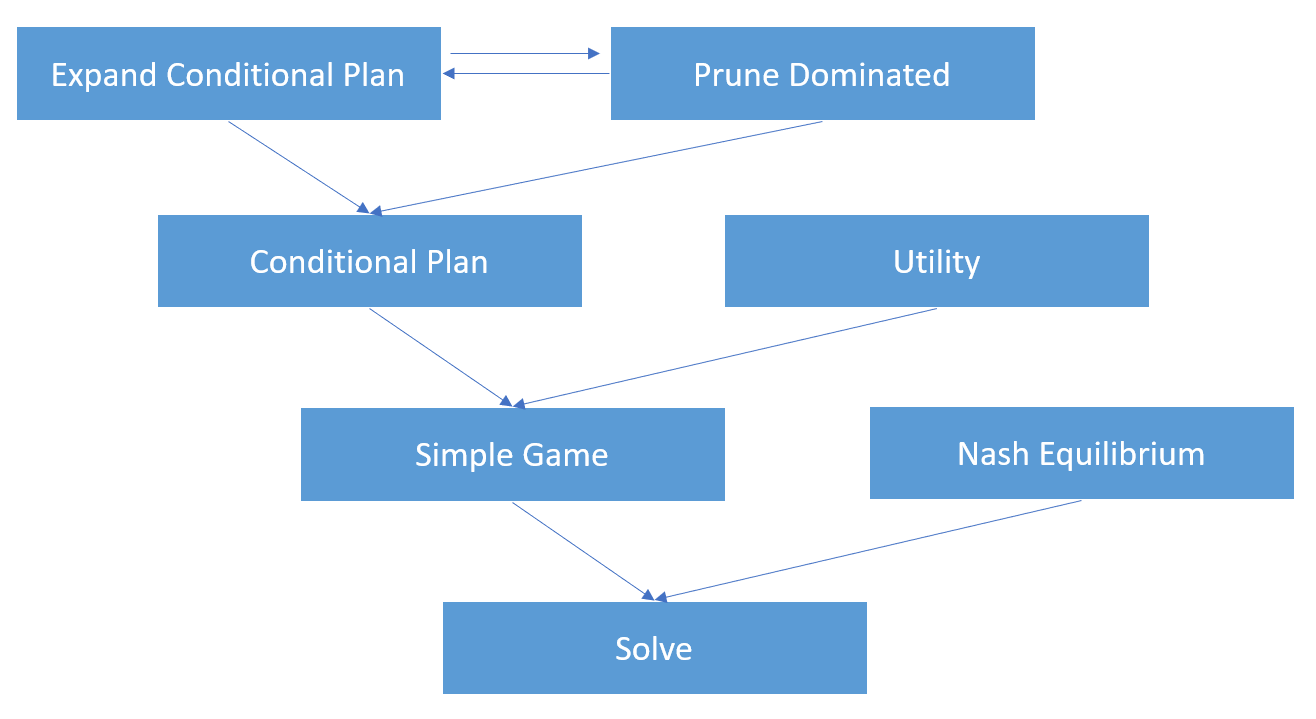
### **Tối ưu hóa: Dynamic Programming**

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

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

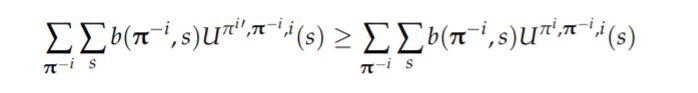
Ưu điểm: cách tiếp cận theo hướng tăng dần độ cao có thể tiết kiệm chi phí đáng kể.

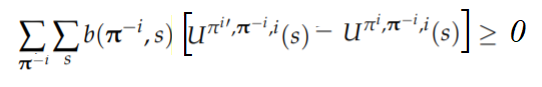
Ý tưởng: Đầu tiên chúng ta xây dựng một conditional plan với độ cao là 1 (chỉ có 1 bước). Tiến hành tối ưu plan này bằng cách loại bỏ những plan mà bị chi phối bởi plan khác. Sau đó, mở rộng ra conditional plan có độ cao là 2 (có 2 bước). Lặp lại việc tối ưu này cho tới khi đạt được độ cao mong muốn.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



Thực hiện:

Để có thể tối ưu các plan bằng cách loại bỏ những plan bị chi phối. Ví dụ policy 𝜋i thuộc về agent I có thể bị loại bỏ nếu tồn tại một policy 𝜋i’ sao cho luôn hoạt động giống như policy 𝜋i. Kiểm tra phụ thuộc: policy 𝜋i phụ thuộc vào policy 𝜋i’ nếu không tồn tại b ( 𝜋-i, s) giữa các joint policy 𝜋-i và state s thỏa mãn:



⬄ (\*)

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

#Cấu trúc của POMG theo Dynamic Programming

struct POMGDynamicProgramming

b # initial belief

d # depth of conditional plans

end

# Kiểm tra phụ thuộc policy 𝜋\_i có phụ thuộc vào policy khác hay không theo công thức phía trên Text

Description automatically generated

function is\_dominated(𝒫::POMG, Π, i, πi)

# Lấy agent và state từ POMG

ℐ, 𝒮 = 𝒫.ℐ, 𝒫.𝒮

jointΠnoti = joint([Π[j] for j in ℐ if j ≠ i])

π(πi′, πnoti) = [j==i ? πi′ : πnoti[j>i ? j-1 : j] for j in ℐ]

Ui = Dict((πi′, πnoti, s) => evaluate\_plan(𝒫, π(πi′, πnoti), s)[i]

for πi′ in Π[i], πnoti in jointΠnoti, s in 𝒮)

model = Model(Ipopt.Optimizer)

@variable(model, δ)

@variable(model, b[jointΠnoti, 𝒮] ≥ 0)

@objective(model, Max, δ)

@constraint(model, [πi′=Π[i]],

sum(b[πnoti, s] \* (Ui[πi′, πnoti, s] - Ui[πi, πnoti, s])

for πnoti in jointΠnoti for s in 𝒮) ≥ δ)

@constraint(model, sum(b) == 1)

optimize!(model)

return value(δ) ≥ 0

end

# Hàm loại bỏ những policy có phụ thuộc

function prune\_dominated!(Π, 𝒫::POMG)

done = false

while !done

done = true

for i in shuffle(𝒫.ℐ)

for πi in shuffle(Π[i])

# Nếu có policy 𝜋\_i nào thỏa mãn hàm is\_dominated và length thì sẽ loại bỏ khỏi Π[i]

if length(Π[i]) > 1 && is\_dominated(𝒫, Π, i, πi)

filter!(πi′ -> πi′ ≠ πi, Π[i])

done = false

break

end

end

end

end

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

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

ℐ, 𝒮, 𝒜, R, γ, b, d = 𝒫.ℐ, 𝒫.𝒮, 𝒫.𝒜, 𝒫.R, 𝒫.γ, M.b, M.d

Π = [[ConditionalPlan(ai) for ai in 𝒜[i]] for i in ℐ]

# Xét theo độ sâu conditional plan d, loại bỏ các policy phụ thuộc

for t in 1:d

Π = expand\_conditional\_plans(𝒫, Π)

prune\_dominated!(Π, 𝒫)

end

# Chuyển về dạng Simple Game và giải quyết bài toán theo NashQeuilibirum trình bày phía trên

𝒢 = SimpleGame(γ, ℐ, Π, π -> utility(𝒫, b, π))

π = solve(NashEquilibrium(), 𝒢)

return Tuple(argmax(πi.p) for πi in π)

end

## **PHÂN TÍCH KẾT QUẢ VÀ Ý NGHĨA**

* Khó khăn: khó tính toán

Vì không giống như POMDP có thể mô hình hóa cấc beliefs, chúng ta sẽ xây dựng policy tập trung vào các policy không yêu cầu belief của agent để quyết định action

* Thuận lợi: 1 nash euilibrium của simple game có thể áp dụng trực tiếp như Nash equilibrium của POMGs

# 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.