**MSSV**: 19120553  
**Họ và tên**: Chung Hoàng Tuấn Kiệt

**Multi Caregivers crying baby**

1. **Phát biểu:**

Giả sử nhiều *agent* cùng quan sát 1 *baby* (cùng 1 *state*). Bài toán sử dụng struct *POMG* – được xem là phần mở rộng từ markov game trong việc quan sát một phần hoặc mở rộng từ cấu trúc POMDP. Tất cả các *agent* đều nắm được trạng thái hiện tại của hệ thống nhưng không cần tất cả các agent phải quan sát một cách toàn diện. Mục tiêu của bài toán: các *agent* cùng quan sát *state* của baby và đưa ra các *action* phù hợp dựa trên các kết quả nhận được từ các *action* trước sao cho đạt được hiệu quả cao nhất theo các xác xuất đã được quy ước từ trước.

1. **Thách thức**:

Cách tiếp cận từ struct *POMDP* không thể sử dụng trong bài toán này do các *individual* *agent* không quan sát. Do *actioin* của các *agent* phụ thuộc vào các *agent* khác. Suy ra phân phối xác suất so với các hành động chung đòi hỏi mỗi *agent* phải suy luận về các *agent* khác. Suy ra một phân phối so với các quan sát khác cũng phức tạp vì các quan sát phụ thuộc vào hành động của các *agent* khác. Do khó mô hình hóa *belief* *state* một cách rõ ràng trong *POMG*, chúng tôi sẽ tập trung vào các *policy* không yêu cầu *belief* để xác định một *action*. Chúng ta có thể sử dụng biểu diễn *tree* *base* *conditional* *plan* và biểu diễn the *graph*-*based* *controller*. Như trong trò chơi Markov, mỗi *agent* trong *POMG* hoạt động theo chính sách *πi*, hoặc tương đương, các tác nhân cùng hoạt động theo policy chung *π = (π1,.., Πk).*

1. **Cấu hình:**

Sử dụng struct *POMG* trong bài toán này, chúng ta có 2 *caregiver* cho *baby*. Giống như trong struct *POMDP*, baby có 2 trạng thái là *hungry* hoặc *sated*. Mỗi *caregiver* có thể *feed*, *ignore* hoặc *sing*. Nếu cả 2 có cùng 1 *action* thì *cost* sẽ giảm 1 nửa. Ví dụ, nếu cả 2 *caregiver* *feed* the *baby*, sau đó *reward* chỉ là −2.5 thay vì −5. Tuy nhiên, *caregivers* không quan sát *baby* một cách hoàn toàn. Thay vào đó, họ dựa vào những quan sát về tiếng khóc của *baby*, cả hai đều nhận được cùng một quan sát. Như một hệ quả của cơ cấu phần thưởng, có sự đánh đổi giữa việc giúp đỡ lẫn nhau và tham lam chọn một hành động ít tốn kém hơn.

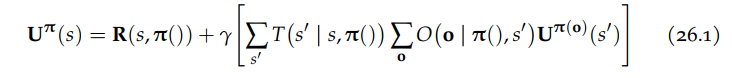
Trong bài toán này, chúng ta sẽ sử dụng *conditional* *plan* làm *policy* cho bài toán.

*Conditional* *plan* là một *tree* mà các *action* liên kết với các *nodes* quan sát với các cạnh. Mỗi *agent* có *plan* riêng. Sau khi quan sát, mỗi agent đi xuống các nodes bên dưới của cây, chọn cạnh liên kết với *observation* của họ. Quá trình thực hiện các *action* và lựa chọn các cạnh dựa trên các quan sát tiếp tục cho đến khi đạt đến phần cuối của cây. Hình ảnh bên dưới mô phỏng *condition* *plan* cho 2 *agent* với *depth* = 2.

Diagram

Description automatically generated

Hàm *joint\_ultility* là hàm Reward cho plan được đánh giá bằng công thức sau:



* Trong đó
  + R: là hàm *joint\_reward* (Tính cost của các agent khi chọn action)
  + T: là hàm *transition* (Tính xác xuất các action làm thay đổi state của baby)
  + O: là hàm *joint\_observation* (Tính xác xuất đứa trẻ khóc dựa vào state hiện tại và action)
  + là hàm đánh giá plan (*evaluate\_plan*) theo *observation* (quan sát được) và state

**A picture containing text, watch

Description automatically generated**

1. **Mô hình hóa và giải quyết:**
   1. **Khởi tạo bài toán:**

|  |
| --- |
| struct **POMG**  **γ**  **ℐ**  **𝒮**  **𝒜**  **𝒪**  **T**  **O**  **R**  end |

Trong đó: **γ** là discount factor có giá trị là 0.9  
 **ℐ** là danh sách các agent  
 **𝒮** là các state của baby (*hungry* hoặc *sated*)  
 **𝒜** là các action (*feed, ignore* hoặc *sing)* **𝒪** là các quan sát (*crying* hoặc *quite)*  **T** là hàm transition (được khởi tạo với hàm *joint\_transition)*  **O** là hàm quan sát (được khởi tạo với hàm *joint\_observation)* **R** là hàm reward (được khởi tạo với hàm *joint\_reward)*

Các xác suất sử dụng trong bài toán

|  |
| --- |
| ***T****(sated | hungry, feed) = 100%*  ***T****(hungry | hungry, sing) = 100%*  ***T****(hungry | hungry, ignore) = 100%*  ***T****(sated | sated, feed) = 100%*  ***T****(hungry | sated, sing) = 10%*  ***T****(hungry | sated, ignore) = 10%*  ***R****(hungry) = -10*  ***R****(feed) = -5*  ***R****(sing) = -0.5*  ***O****(cry | feed, hungry) = 80%*  ***O****(cry | sing, hungry) = 90%*  ***O****(cry | ignore, hungry) = 80%*  ***O****(cry | feed, sated) = 10%*  ***O****(cry | sing, sated) = 0%*  ***O****(cry | ignore, sated) = 10%*  ***beliefState*** *=* [0.5, 0.5] |

Hàm *trainsition* – Xét sự thay đổi *state* của các *agent*

* + Nếu action của một trong các agent là *FEED* và state lúc sau là *SATED* thì return 1, nếu khác thì *return* 0.
  + Nếu khác, sẽ rơi vào 2 trường hợp:
    - Nếu cả *state* hiện tại và lúc sau đều là *HUNGRY* thì return 1, nếu khác return 0
    - Nếu state lúc đầu không phải *HUNGRY* thì return 0.5

Hàm *joint\_reward* – Tính *reward* cho các *agent*:

* + Chú ý: Các *agent* phải thực hiện một action thì mới nhận được reward
  + Đặt 1 biến r là *reward* cho 2 agent
  + Nếu state hiện tại là *HUNGRY* thì các agent sẽ nhận *reward* là *R(hungry)*
  + Nếu action của agent 1 là *FEED* thì sẽ nhận reward là *R(feed) / 2,* nếu action là *SING* thì nhận reward *R(sing)*
  + Nếu *action* của agent 2 là *FEED* thì sẽ nhận reward là *R(feed),* nếu action là *SING* thì sẽ nhận *R(sing) / 2*

Hàm *joint\_observation –* Tính xác xuất Crying (hàm này sẽ trả về kết quả dựa theo các biến xác suất đã được khởi tạo phía trên)

|  |
| --- |
| *struct ConditionalPlan*  *a*  *subplans*  *end* |

Với *a* là *action* ở *root* và *subplan* là *mapping* những *observation* từ *agent*.

* 1. **Giải quyết bài toán bằng thuật giải Nash Equilibrium:**

**Lý do:** Thuật giải dễ cài đặt và dễ hiểu

*Policy* để đánh giá bài toán là *ConditionalPlan* với các hàm đã cài đặt theo công thức trên.

Điểm cân bằng *Nash* cho *POMG* là khi tất cả các agent hành động theo *policy* phản hồi tốt nhất cho nhau, sao cho không có *agent* nào có động cơ làm sai lệch *policy* của họ. Các điểm cân bằng Nash cho *POMG* có xu hướng cực kỳ khó giải quyết về mặt tính toán. Thuật toán này tính toán cân bằng *Nash d-step* cho một *POMG*. Nó liệt kê tất cả các kế hoạch điều kiện chung bước *d* có thể có của nó để xây dựng một trò chơi đơn giản. Điểm cân bằng *Nash* cho *SimpleGame* này cũng là điểm cân bằng Nash cho *POMG*. *SimpleGame* có các *agent* tương tự như *POMG*. Có một hành động chung trong *SimpleGame* cho mọi *ConditionalPlan* chung trong *POMG*. *Reward* nhận được cho mỗi *action* tương đương với các tiện ích theo *ConditionalPlan* chung trong *POMG*. Một cân bằng *Nash* của *SimpleGame* được xây dựng này có thể được áp dụng trực tiếp như một cân bằng *Nash* của *POMG*.

Diagram

Description automatically generated

Hàm *solve* của thuật giải :

* + Khởi tạo *plan* bằng hàm *create\_conditional\_plan*
  + Tạo *simpleGame* với *Reward* function trùng với công thứ trên
  + Hàm *solve SimpleGame:* sẽ khởi tạo, add các *parameter* và *optimize* *model* bằng thư viện của hệ thống. Sau đó sẽ khởi tạo *SimpleGamePolicy* và trả về kết quả.
  + Trả về kết quả là có kiểu dữ liệu là *Tuple*.
  1. **Giải quyết bài toán bằng thuật giải Dynamic Programming:**

**Lý do:** Thuật giải hiệu quả, là phần mở rộng từ thuật toán *Nash Equilibrium* nhưng có áp dụng quy hoạch động nên tiết kiệm được rất nhiều thời gian.

*Policy* để đánh giá bài toán là *ConditionalPlan* với các hàm đã cài đặt theo công thức trên.

Phương pháp sừ dụng *Nash* để tính toán yêu cầu chi phí tính toán khá lớn do phải duyệt qua tất cả các trường hợp có thể xảy ra của *plan*. Chúng ta có thể chuyển đổi phương pháp trên bằng cách lượt bỏ bớt các *sub optioinal* của *subplan* và xem xét *plan* hiện tại. Mặc dù độ phức tạp tính toán trong trường hợp xấu nhất cũng giống như độ phức tạp của việc mở rộng toàn bộ các cây *ConditionalPlan*, nhưng cách tiếp cận gia tăng này có thể dẫn đến tiết kiệm đáng kể.

Nó bắt đầu bằng cách xây dựng tất cả các *one-step plan*. Lược bỏ bất kỳ *plan* nào bị chi phối bởi một *plan* khác, và sau đó chúng tôi mở rộng tất cả các kết hợp của *one-step plan* để tạo ra *two-steps plan*. Quy trình xen kẽ giữa mở rộng và cắt tỉa này được lặp lại cho đến khi đạt được đích đến mong muốn. Bước cắt tỉa sẽ loại bỏ tất cả các *policy* chi phối. Một *policy* *πi* thuộc về một *agent* thứ *i* có thể bị loại bỏ nếu tồn tại một *policy* *πi* ′ khác luôn hoạt động ít nhất cũng như *πi*. Mặc dù tốn kém về mặt tính toán, điều kiện này có thể được kiểm tra bằng cách giải một chương trình tuyến tính. Quá trình này liên quan đến việc lược bớt nút *controller* trong *POMDPs* .Sẽ rất khó tính về mặt tính toán nếu giải một chương trình tuyến tính riêng biệt cho mọi sự kết hợp có thể có của các *policy* của *agent* khác. Thay vào đó, chúng ta có thể thực hiện một cách tiếp cận hiệu quả hơn nhiều mà sẽ không bao giờ cắt bỏ một chính sách tối ưu mà có thể bị lược bớt. Một chính sách *πi* bị chi phối bởi *πi* ′ nếu không tồn tại *b(π − i, s)* giữa các chính sách chung khác và các trạng thái *s* sao cho:

A picture containing shape

Description automatically generated

Ở đây, *b* là sự phân phối chung so với các pollicy của các *agent* khác và của *state*. Không khả thi khi tính toán một *belief state*, nhưng phương trình trên kiểm tra không gian của *belief* đối với sự chi phối của *policy* cá nhân. Chúng ta có thể xây dựng một chương trình tuyến tính duy nhất để kiểm tra phương trình trên nếu chương trình tuyến tính khả thi, thì điều đó có nghĩa là *πi* không bị chi phối bởi bất kỳ *πi ′* nào khác.

Text, letter

Description automatically generated

Bước lược bỏ loại bỏ các *policy* bị chi phối bằng cách chọn ngẫu nhiên một *agent* I và kiểm tra xem có sự chi phối của từng *policy* của nó hay không. Quá trình này lặp lại cho đến khi vượt qua tất cả các *agent* không tìm thấy bất kỳ *policy* chi phối nào.

Hàm *solve* sẽ giải quyết bài toán hiện tại. Các bước thực hiện:

* + Khởi tạo *ConditionalPlan* cho các agent
  + Duyệt qua d lần (d là độ sâu của plan hiện tại) để:
    - Mở rộng *ConditionalPlan.*
    - Lọc lại và lượt bỏ các optioinal của subplan.
  + Tạo *SimpleGame* từ ConditionalPlan của các agent, discount factor (có giá trị là 0.9) và hàm *Reward* là hàm *ultility* được triển khai từ công thức trên**.**
  + Gọi hàm *solve (Solve simpleGame)* để solve *SimpleGame* vừa khởi tạo.
  + Lưu và trả kết quả vào Tuple.

Các hàm được sử dụng trong hàm *solve*:

* + Hàm *prune!:* kiểm tra và gọi hàm filter để loại bỏ các sub optional của subplan. Điều chỉnh giá trị ngay trên Plan hiện tại.
  + Hàm *expandConditionalPlan:* tạo array ConditionalPlan từ Action Space, Observation Space và Plan hiện tại.
  + Hàm *solve SimpleGame:* sẽ khởi tạo, add các parameter và optimize model bằng thư viện của hệ thống. Sau đó sẽ khởi tạo *SimpleGamePolicy* và trả về kết quả.

1. **Đánh giá các thuật giải:**
   1. **Đánh giá chung**

Tiêu chí đánh giá: Thời gian giải quyết bài toán

|  |  |  |
| --- | --- | --- |
| **Depth** | **Nash Equilibrium** | **Dynamic Programming** |
| 2 | 14.1 | 25.5 |
| 3 | 45.2 | 28.2 |
| 4 | 2405 | 33.6 |
| 5 | Chưa xác định | 43.1 |

Bảng đánh giá trên được thực hiện (đơn vị thời gian: giây (s)) với

1000 lần với *depth* = 2  
 400 lần với *depth* = 3  
 3 lần với *depth* = 4  
 3 lần với *depth* = 5

Do thời gian thực hiện với depth bằng 4 và 5 trên thuật toán *Nash* tốn chi phí thời gian lớn nên chỉ có thể thực hiện lấy trung bình trên 3 lần đo.

Đối với thuật toán Nash Equilibrium, khi depth = 5 thì cần chi phí thời gian rất lớn để giải quyết bài toán. Trang thái **Chưa xác định** trên do thí nghiệm được chạy hơn 1 ngày và vẫn chưa đạt được kết quả.

* 1. **Nash Equilibrium:**

Điểm mạnh:   
 Dễ cài đặt và dễ hiểu  
 Thu được kết quả là duy nhất khi thực hiện qua nhiều vòng lặp  
Điểm yếu:  
 Do không loại bỏ bớt các *sub*-*optional* ở sub-plan nên dẫn đến thuật toán tốn chi phí tính toán quá cao. Do đó, không khả thi khi giải quyết bài toán với độ sâu của *ConditionalPlan* .

* 1. **Dynamic Programming:**

Điểm mạnh:  
 Được mở rộng từ thuật toán *Nash* *Equilibrium* và có loại bỏ các *sub*-*optional* của *sub*-*plan* nên tiết kiệm chi phí tính toán cao  
 Số *ConditionalPlan* thu được tăng theo độ sâu của bài toán. Với:  
 1000 vòng lặp và *depth* = 2 thu được 8 *plan* khác nhau  
 400 vòng lặp với *depth* = 3 thu được 21 *plan* khác nhau  
 Số *ConditionalPlan* tạo ra đa dạng. Có thể tối ưu bài toán  
Điểm yếu:  
 Khó cài đặt hơn *Nash* *Equilibrium*

1. **Tóm tắt đồ án:**

Điểm mạnh:

Giải quyết bài toán theo nhiều cách và có thống kê số liệu từ đó đưa ra được lựa chọn cho người đọc trong việc chọn lựa thuật toán để cài đặt giải quyết bài toán.

Có trực quan hóa kết quả qua hàm *DrawTree*, từ đó có thể dễ dàng phân tích kết quả.

Có chú thích code và giải thích ý nghĩa các tham số trong bài toán cũng như các hàm khởi tạo, từ đó người dùng có thể dễ thực hiện trên các bài toán tương tự.  
Điểm yếu:

Chưa đánh giá thuật toán trước khi tiến hành thực nghiệm, dẫn đến tốn chi phí thời gian lớn cho thuật toán *Nash* *Equilibrium*.  
 Chưa tạo được đánh giá trực quan khác cho bài toán, ngoài việc thống kê kết quả và trực quan hóa kết quả.

1. **Chú thích code:**

Trong file *code* có hàm (*DrawTree)* vẽ ra kết quả là *ConditionalPlan* dưới dạng cây sử dụng thư viện D3Tree của julia. Sau khi thực thi có thể chạy đoạn code trên để thấy kết quả trực quan hơn.

Hàm thống kê kết quả vào file excel có thể bỏ qua do phải tốn thời gian để nhiều để thống kê lại kết quả. Các *plan* phổ biến trong quá trình chạy thuật toán cũng đã được thống kê vào file excel. Có thể copy đoạn kết quả đó để khởi tạo biến và sử dụng hàm *DrawTree* bằng cách truyền kết quả đó vào biến . Song đó có thể nhìn thấy các plan phổ biến một các trực quan hơn. (Cây ở trạng thái **interactive** nên phải click vào để xem chi tiết từng subplan.

1. **Các nguồn source code tham khảo:**

Các source code khởi tạo cho struct *POMG*

[Code init cho POMG](https://github.com/algorithmsbooks/DecisionMakingProblems.jl/tree/49b8440c5b3bb1397005d8393241c37c81ed6e1c/src/pomg)

Các source code thực thi thuật giải *Nash Equilibrium* và *Dynamic Programming*

*Sách Algorithms for decision making*