A picture containing application

Description automatically generated**TRƯỜNG ĐẠI HỌC SÀI GÒN**

**KHOA CÔNG NGHỆ THÔNG TIN**

Icon

Description automatically generated with low confidence

**BÁO CÁO**

**HỌC PHẦN: TRÍ TUỆ NHÂN TẠO NÂNG CAO**

**LAB 04**

**Sinh viên thực hiện**

Nguyễn Anh Khoa – 3122410185

Huỳnh Khôi Nguyên - 3122410267

Trần Gia Nuyễn – 3122410274

**TP. HCM THÁNG 9/2025**

**MỤC LỤC**

[**PHÂN CÔNG 3**](#_Toc212320774)

[**1. Connect4 4**](#_Toc212320775)

[**1.1. Giới thiệu Trò chơi Connect 4 4**](#_Toc212320776)

[**1.2. Mục tiêu Học tập (Learning Outcomes) 4**](#_Toc212320777)

[**TASK 1: ĐỊNH NGHĨA BÀI TOÁN TÌM KIẾM ĐỐI KHÁNG 4**](#_Toc212320778)

[**TASK 2: GAME ENVIRONMENT VÀ HÀM ĐÁNH GIÁ HEURISTIC 9**](#_Toc212320779)

[**TASK 3: TRIỂN KHAI MINIMAX VỚI ALPHA-BETA PRUNING 11**](#_Toc212320780)

[**Task 4 triển khai tác tử chiến lược hoàn chỉnh (limited\_depth\_agent) bằng cách kết hợp α - β Pruning với Hàm Heuristic (tìm kiếm giới hạn độ sâu). 15**](#_Toc212320781)

[**ADVANCE TASK 18**](#_Toc212320782)

[**2. Mean connect4 20**](#_Toc212320783)

[**Giới thiệu 20**](#_Toc212320784)

[**Task 1: Định nghĩa bài toán tìm kiếm 20**](#_Toc212320785)

[**Task 2: Môi trường trò chơi và Tác tử ngẫu nhiên 23**](#_Toc212320786)

[**Task 3 & 4: Tìm kiếm Heuristic Alpha-Beta 30**](#_Toc212320787)

[**Tournament task: Pure Monte Carlo Search (PMCS) 37**](#_Toc212320788)

[**3. Tic-Tac-Toe 40**](#_Toc212320789)

[**Example 40**](#_Toc212320790)

[**Tic-Tac-Toe 40**](#_Toc212320791)

[**Solving Tic-Tac-Toe with AND-OR-Tree Search 41**](#_Toc212320792)

[**Play Tic-Tac-Toe Interactively (Simple Implementation) 46**](#_Toc212320793)

[**Solving Tic-Tac-Toe with Minimax Search and Alpha-Beta Pruning 48**](#_Toc212320794)

[**Phương pháp 1: Recursive DFS algorithm for minimax search 48**](#_Toc212320795)

[**Phương pháp 2: Cắt tỉa Alpha-Beta (Alpha-Beta Pruning) 52**](#_Toc212320796)

[**Phương pháp 3: Sắp xếp nước đi (Move Ordering) 56**](#_Toc212320797)

[**Solving Tic-Tac-Toe with Heuristic Alpha-Beta Tree Search 57**](#_Toc212320798)

[**Solving Tic-Tac-Toe with Monte Carlo Tree Search 65**](#_Toc212320799)

[**Phương thức UCB1() 70**](#_Toc212320800)

[**Các cột trong bảng 72**](#_Toc212320801)

[**Solving Tic-Tac-Toe with Monte Carlo Tree Search Restricted 75**](#_Toc212320802)

[**Result: 79**](#_Toc212320803)

[**Pure Monte Carlo Search vs. Random: 80**](#_Toc212320804)

PHÂN CÔNG

|  |  |  |
| --- | --- | --- |
| **MSSV** | **Họ tên** | **Phân công** |
| 3122410185 | Nguyễn Anh Khoa | connect4, báo cáo, slide |
| 3122410267 | Huỳnh Khôi Nguyên | dots\_and\_boxes, slide |
| 3122410274 | Trần Gia Nguyễn (nhóm trưởng) | mean\_connect4, tổng hợp báo cáo, slide |

1. Connect4

1.1. Giới thiệu Trò chơi Connect 4

Connect 4 là một trò chơi cờ bàn kết nối hai người chơi, được thực hiện trên một lưới treo đứng 6 hàng X 7 cột. Người chơi luân phiên thả các đĩa màu (thường là Đỏ và Vàng) vào các cột. Do tác động của trọng lực, các đĩa sẽ rơi xuống và chiếm lấy ô trống thấp nhất trong cột đã chọn. Mục tiêu của trò chơi là trở thành người đầu tiên tạo được một đường thẳng (ngang, dọc, hoặc chéo) gồm bốn đĩa cùng màu của mình.

Mặc dù Connect 4 đã được giải quyết toán học vào năm 1988, xác định rằng người chơi thứ nhất luôn có thể thắng nếu chơi tối ưu, dự án này tập trung vào việc áp dụng và phân tích các kỹ thuật Tìm kiếm Đối kháng (Adversarial Search) trong Trí tuệ Nhân tạo (AI).

1.2. Mục tiêu Học tập (Learning Outcomes)

Dự án nhằm đạt được các mục tiêu học tập sau:

* Triển khai các thuật toán tìm kiếm đối kháng (Minimax, Alpha-Beta Pruning) cho chiến lược chơi game.
* Phân tích và tối ưu hóa tìm kiếm trong không gian trò chơi phức tạp.
* Thiết kế các hàm đánh giá heuristic hiệu quả để ước lượng giá trị của các trạng thái trung gian.
* So sánh hiệu suất giữa các chiến lược tác tử (agent) khác nhau.
* Đánh giá sự đánh đổi thuật toán giữa chất lượng quyết định (khả năng thắng) và hiệu quả tính toán (tốc độ tìm kiếm).

TASK 1: ĐỊNH NGHĨA BÀI TOÁN TÌM KIẾM ĐỐI KHÁNG

Bài toán Connect 4 được chính thức hóa thành một bài toán tìm kiếm đối kháng (Adversarial Search Problem) với các thành phần sau. Trong triển khai, chúng ta sử dụng thư viện numpy để biểu diễn bảng chơi, với giá trị 1 đại diện cho người chơi Đỏ (MAX) và -1 đại diện cho người chơi Vàng (MIN).

**1. Trạng thái Khởi tạo (Initial State):**

* Đây là bảng trò chơi Connect 4 rỗng, có kích thước 6 hàng X7 cột.
* Triển khai**:** Hàm **initial\_state()** hoặc **empty\_board()** trả về một mảng **numpy.zeros((6, 7), dtype=int).**

**2. Hành động (Actions):**

* Tập hợp các cột (chỉ số từ 0 đến 6) mà người chơi có thể thả đĩa vào. Hành động được coi là hợp lệ nếu cột đó chưa đầy.
* Triển khai: Hàm valid\_actions(state) kiểm tra hàng trên cùng (row 0) của mỗi cột; nếu ô đó bằng 0, cột đó được đưa vào danh sách hành động hợp lệ.

**3. Mô hình Chuyển đổi (Transition Model - Result):**

* Trạng thái mới của bảng sau khi một người chơi thực hiện một hành động hợp lệ. Đĩa của người chơi sẽ rơi xuống vị trí trống thấp nhất trong cột đã chọn.
* Triển khai: Hàm **result(state, player, action)** tìm chỉ số hàng lớn nhất (row\_index) trong cột action sao cho **state[row\_index, action] == 0**, sau đó đặt giá trị player (1 hoặc -1) vào vị trí đó trên một bản sao (copy) của trạng thái cũ.

**4. Trạng thái Kết thúc (Terminal State):**

Trò chơi dừng lại khi một trong các điều kiện sau được thỏa mãn:

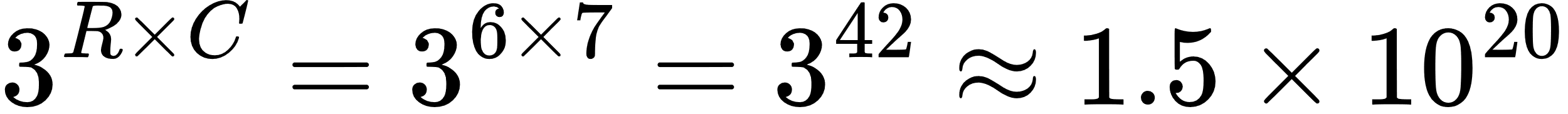
* Một người chơi tạo thành một đường thẳng (ngang, dọc, hoặc chéo) gồm bốn đĩa cùng màu.
* Bảng đầy, dẫn đến kết quả hòa.
* Triển khai: Hàm **terminal(state)** kiểm tra 4 hướng thắng cho cả hai người chơi. Hàm trả về cặp **(is\_terminal: bool, winner: int)** (nếu winner là 0, có nghĩa là hòa hoặc chưa kết thúc).

**5. Utility (Giá trị Lợi ích):**

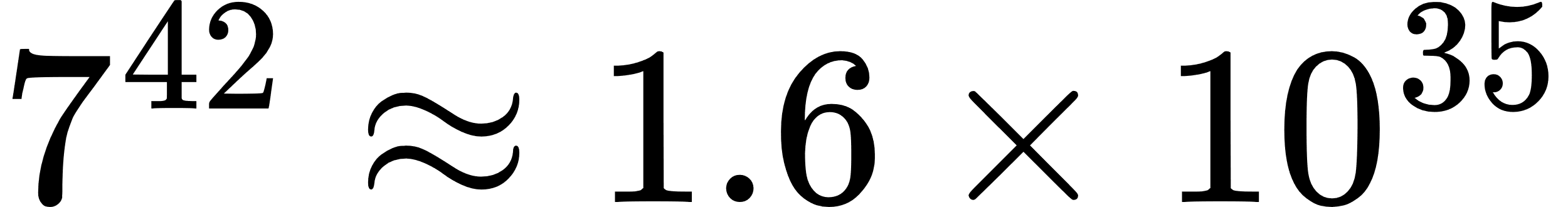
* Giá trị đánh giá cuối cùng của trạng thái kết thúc, từ góc nhìn của một người chơi cụ thể.
* Triển khai: **Hàm utility(state, player**) trả về +1 nếu player thắng, -1 nếu player thua, và 0 nếu hòa.

**2.1. Phân tích Độ phức tạp (Complexity Analysis)**

**a. Kích thước Không gian Trạng thái (State Space Size)**

* Ước tính (Upper Bound): trạng thái.
* Giải thích: Mặc dù số lượng trạng thái thực tế khả dĩ (reachable states) nhỏ hơn do luật chơi, không gian vẫn quá lớn để sử dụng tìm kiếm vét cạn (brute-force search).

**b. Kích thước Cây Trò chơi (Game Tree Size)**

* Ước tính: Với hệ số phân nhánh (branching factor, b) ≈ 7 và độ sâu tối đa (depth, d) ≤ 42, kích thước cây có thể lên tới  node.
* Kết luận: Kích thước khổng lồ này khẳng định rằng thuật toán Minimax thô sơ là không khả thi. Các kỹ thuật tối ưu hóa như Alpha-Beta Pruning và giới hạn độ sâu là bắt buộc.

**Code**

Ảnh có chứa văn bản, ảnh chụp màn hình, màn hình, phần mềm

Nội dung do AI tạo ra có thể không chính xác.

Ảnh có chứa văn bản, màn hình, máy tính, ảnh chụp màn hình

Nội dung do AI tạo ra có thể không chính xác.

TASK 2: GAME ENVIRONMENT VÀ HÀM ĐÁNH GIÁ HEURISTIC

**1. Triển khai Cơ sở: Random Agent** (Cell 13)

* Mục đích: Kiểm tra tính năng của môi trường game và làm đối thủ benchmark cơ sở.
* Chiến lược: Tác tử random\_agent chọn một hành động (cột) ngẫu nhiên từ danh sách các hành động hợp lệ.

**CODE**

Ảnh có chứa văn bản, ảnh chụp màn hình, màn hình, máy tính

Nội dung do AI tạo ra có thể không chính xác.

**Xây dựng helper function**

Ảnh có chứa văn bản, đồ điện tử, ảnh chụp màn hình, máy tính

Nội dung do AI tạo ra có thể không chính xác.

Ảnh có chứa văn bản, đồ điện tử, ảnh chụp màn hình, máy tính

Nội dung do AI tạo ra có thể không chính xác.

**2. Thiết kế Hàm Đánh giá Heuristic (eval\_function)** (Cell 15)

Khi tìm kiếm bị giới hạn độ sâu, eval\_function cung cấp một ước lượng về giá trị chiến lược của trạng thái trung gian.

* Ý tưởng Cốt lõi: Đánh giá trạng thái bằng cách định lượng các chuỗi đĩa liên tiếp có tiềm năng mở rộng thành chuỗi 4.
* Tính toán: Quét toàn bộ bảng theo 4 hướng (ngang, dọc, chéo chính, chéo phụ).
* Trọng số: Gán trọng số cao cho chuỗi 3 (mối đe dọa thắng gần) và thấp hơn cho chuỗi 2 (cơ hội phát triển).
* Giá trị Heuristic: Là hiệu số giữa tổng điểm Heuristic của Player 1 và Player -1.

**CODE**

Ảnh có chứa văn bản, ảnh chụp màn hình, Phần mềm đa phương tiện, phần mềm

Nội dung do AI tạo ra có thể không chính xác.

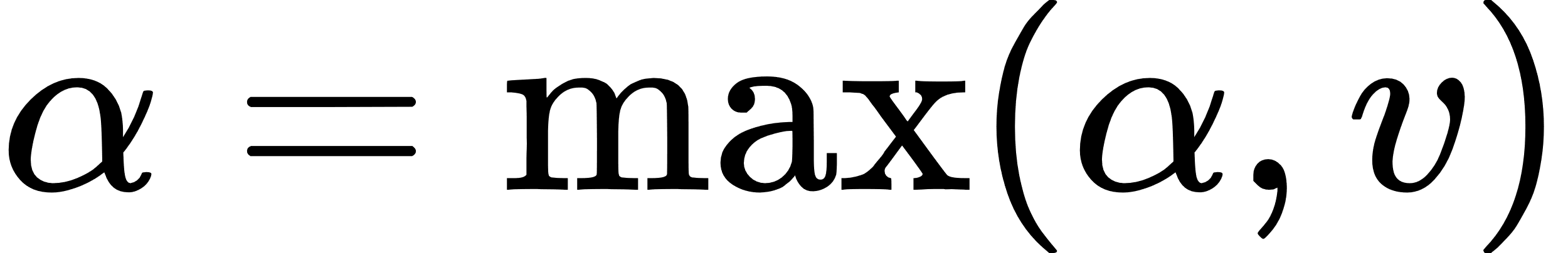
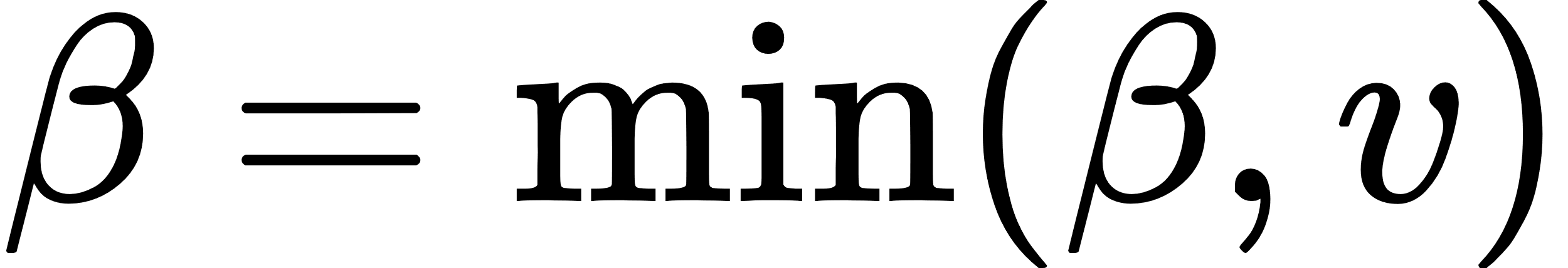
TASK 3: TRIỂN KHAI MINIMAX VỚI ALPHA-BETA PRUNING

**1. Triển khai Alpha-Beta Pruning**

Thuật toán Minimax được triển khai với tính năng cắt tỉa α-β để giảm số lượng node cần duyệt mà vẫn giữ được kết quả tối ưu.

* Hàm alpha\_beta\_search(state, depth, player): Là hàm quyết định chính, gọi max\_value\_alpha\_beta cho người chơi hiện tại.
* Biến α và β:
  + α (Alpha): Giá trị tối đa MAX có thể đạt được trên đường đi hiện tại.
  + Β (Beta): Giá trị tối thiểu MIN có thể đạt được trên đường đi hiện tại.

**2. Logic MAX và MIN**

* max\_value\_alpha\_beta(state, depth, alpha, beta, player):
  + Giả định đây là lượt của MAX Player.
  + Duyệt qua các hành động, cập nhật giá trị v = max(v, min\_value\_alpha\_beta\_dots)).
  + Cắt tỉa: Nếu v ≥ beta, dừng tìm kiếm nhánh này và trả về v (cắt tỉa Beta).
  + Cập nhật .
* min\_value\_alpha\_beta(state, depth, alpha, beta, player):
  + Giả định đây là lượt của MIN Player.
  + Duyệt qua các hành động, cập nhật giá trị v = min(v, max\_value\_alpha\_beta\_dots)).
  + Cắt tỉa: Nếu v ≤ alpha, dừng tìm kiếm nhánh này và trả về v (cắt tỉa Alpha).
  + Cập nhật .

**Code:**

Ảnh có chứa văn bản, máy tính, ảnh chụp màn hình, phần mềm

Nội dung do AI tạo ra có thể không chính xác.

Ảnh có chứa văn bản, ảnh chụp màn hình, màn hình, máy tính

Nội dung do AI tạo ra có thể không chính xác.

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Nội dung do AI tạo ra có thể không chính xác.

Ảnh có chứa văn bản, đồ điện tử, ảnh chụp màn hình, màn hình

Nội dung do AI tạo ra có thể không chính xác.

**Move Ordering**

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm, màn hình

Nội dung do AI tạo ra có thể không chính xác.

**The first few move**

Ảnh có chứa văn bản, ảnh chụp màn hình, màn hình, đa phương tiện

Nội dung do AI tạo ra có thể không chính xác.

**Play time**

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, đa phương tiện

Nội dung do AI tạo ra có thể không chính xác.

Task 4 triển khai tác tử chiến lược hoàn chỉnh (limited\_depth\_agent) bằng cách kết hợp α - β Pruning với Hàm Heuristic (tìm kiếm giới hạn độ sâu).

**1. Triển khai Tác tử Chiến lược**

* **limited\_depth\_agent(state, player, max\_depth):** Hàm này sử dụng hàm tìm kiếm Alpha-Beta được triển khai trong Task 3 nhưng giới hạn độ sâu tối đa là max\_depth.
* Logic Đánh giá tại Node Lá:
  + Nếu đạt trạng thái Terminal, sử dụng **utility().**
  + Nếu đạt Giới hạn Độ sâu **(depth == max\_depth)**, sử dụng **eval\_function()** (Heuristic từ Task 2) để ước lượng giá trị chiến lược của trạng thái.
* Hoạt động: Tác tử tìm kiếm hành động tốt nhất (chọn cột) bằng cách tối đa hóa giá trị được trả về bởi hàm tìm kiếm α - β.

**2. Ý nghĩa**

Tác tử này là Agent chiến lược mạnh nhất của dự án. α - β Pruning tối ưu hóa tốc độ tìm kiếm, trong khi Heuristic đảm bảo rằng các quyết định vẫn mang tính chiến lược và thông minh ngay cả khi cây tìm kiếm bị cắt sớm. Việc lựa chọn max\_depth (giới hạn độ sâu) là sự đánh đổi giữa chất lượng quyết định và hiệu suất thời gian thực.

**Code**

def heuristic\_eval(state: np.ndarray, player: int) -> float:

    """Window-based heuristic: quét mọi cửa sổ độ dài 4 (hàng, cột, 2 đường chéo).

    Trả điểm cho patterns: 4-in-row (very large), open-3, closed-3, open-2; cộng bonus trung tâm.

    Kết quả chuẩn hóa về [-1,1].

    """

    rows, cols = state.shape

    def windows\_of\_length\_4(s):

        # rows

        for r in range(rows):

            for c in range(cols - 4 + 1):

                yield [s[r, c+i] for i in range(4)]

        # cols

        for c in range(cols):

            for r in range(rows - 4 + 1):

                yield [s[r+i, c] for i in range(4)]

        # diag \ (down-right)

        for r in range(rows - 4 + 1):

            for c in range(cols - 4 + 1):

                yield [s[r+i, c+i] for i in range(4)]

        # anti-diag / (down-left)

        for r in range(rows - 4 + 1):

            for c in range(3, cols):

                yield [s[r+i, c-i] for i in range(4)]

    # weights

    W4 = 1000.0

    W\_open3 = 50.0

    W\_3 = 10.0

    W\_open2 = 3.0

    W\_center = 1.0

    score = 0.0

    def eval\_window(win, p):

        cnt\_p = sum(1 for x in win if x == p)

        cnt\_opp = sum(1 for x in win if x == -p)

        if cnt\_p == 4: return W4

        if cnt\_opp == 4: return -W4

        if cnt\_p == 3 and cnt\_opp == 0: return W\_open3

        if cnt\_opp == 3 and cnt\_p == 0: return -W\_open3

        if cnt\_p == 2 and cnt\_opp == 0: return W\_open2

        if cnt\_opp == 2 and cnt\_p == 0: return -W\_open2

        return 0.0

    for win in windows\_of\_length\_4(state):

        score += eval\_window(win, player)

    # center control bonus

    center\_col = cols // 2

    for r in range(rows):

        if state[r, center\_col] == player: score += W\_center

        elif state[r, center\_col] == -player: score -= W\_center

    # normalize

    approx\_max = W4 + (W\_open3 \* 10) + (W\_open2 \* 20) + (W\_center \* rows)

    val = max(-1.0, min(1.0, score / approx\_max))

    return val

**Cutting Off Search**

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, danh thiếp

Nội dung do AI tạo ra có thể không chính xác.

Ảnh có chứa văn bản, ảnh chụp màn hình, màn hình, phần mềm

Nội dung do AI tạo ra có thể không chính xác.

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Nội dung do AI tạo ra có thể không chính xác.

**Play time**

env = Connect4Env(6,7)

agent\_deep = MinimaxAgent(max\_depth=4, eval\_fn=heuristic\_eval)

agent\_shallow = MinimaxAgent(max\_depth=2, eval\_fn=heuristic\_eval)

winner = play\_game(agent\_deep, agent\_shallow, env, verbose=False)

print('Winner (deep vs shallow):', winner)

ADVANCE TASK

**MONTE CARLO SEARCH**

Ảnh có chứa văn bản, đồ điện tử, ảnh chụp màn hình, máy tính

Nội dung do AI tạo ra có thể không chính xác.

**BEST FIRST MOVE**

Ảnh có chứa văn bản, danh thiếp, ảnh chụp màn hình, Phông chữ

Nội dung do AI tạo ra có thể không chính xác.

1. Mean connect4

Giới thiệu

Dự án này tập trung vào việc triển khai và đánh giá các tác tử (agents) thông minh để chơi một phiên bản sửa đổi của trò chơi Connect 4, được gọi là "Mean" Connect 4.

**Connect 4** là một trò chơi kết nối hai người chơi, trong đó người chơi chọn một màu và lần lượt thả các quân cờ màu vào một lưới treo dọc gồm bảy cột, sáu hàng. Các quân cờ rơi thẳng xuống, chiếm không gian trống thấp nhất trong cột. Mục tiêu của trò chơi là trở thành người đầu tiên tạo thành một hàng ngang, dọc hoặc chéo gồm bốn quân cờ của mình.

**"Mean" Connect 4** có một quy tắc bổ sung. Mỗi khi đến lượt, người chơi có thể quyết định, thay vì chơi một quân cờ mới, lấy một quân cờ của đối thủ ở hàng dưới cùng và đặt nó vào bất kỳ cột nào. Tất cả các quân cờ phía trên quân cờ bị loại bỏ sẽ rơi xuống một vị trí.

Task 1: Định nghĩa bài toán tìm kiếm

Các thành phần của bài toán:

Trạng thái ban đầu (Initial state): Một bàn cờ trống có kích thước 6x7, biểu diễn bằng một mảng NumPy chứa đầy các số 0.

Hành động (Actions): Tại mỗi trạng thái, người chơi có thể thực hiện một trong hai loại hành động:

Thả quân cờ: Thả một quân cờ (1 cho Max, -1 cho Min) vào một trong bảy cột chưa đầy.

Di chuyển "mean": Lấy một quân cờ của đối thủ từ hàng dưới cùng và đặt nó vào một cột bất kỳ chưa đầy.

Mô hình chuyển đổi (Transition model): Nếu thả một quân cờ, ô trống thấp nhất trong cột đó sẽ được lấp đầy. Nếu thực hiện hành động "mean", quân cờ của đối thủ ở hàng dưới cùng sẽ được di chuyển, và các quân cờ phía trên nó sẽ rơi xuống một vị trí.

Kiểm tra trạng thái kết thúc (Terminal state test): Trò chơi kết thúc khi một người chơi tạo được một hàng 4 quân cờ, hoặc khi bàn cờ đã đầy.

Hàm hữu dụng (Utility function):

+1 nếu người chơi Max thắng.

-1 nếu người chơi Min thắng.

0 cho một trận hòa.

**Phân tích không gian tìm kiếm**

**Kích thước không gian trạng thái lớn như thế nào?**

Mỗi ô trong bảng 6x7 (42 ô) có thể ở một trong ba trạng thái: trống, người chơi 1, hoặc người chơi -1. Do đó, giới hạn trên của số lượng trạng thái có thể có là 3^42.

# Mỗi ô trong bảng 6x7 (42 ô) có thể ở một trong ba trạng thái:

# 1. Trống (0)

# 2. Bị chiếm bởi người chơi 1 (1)

# 3. Bị chiếm bởi người chơi -1 (-1)

# Do đó, số lượng trạng thái tối đa có thể có là 3^42.

# 3^42 ≈ 1.094 x 10^20

# Con số này là một ước tính trên, vì nhiều trạng thái trong số này là không thể đạt được

# trong một trò chơi thực tế (ví dụ: một bảng đầy quân cờ của chỉ một người chơi).

print(f"Ước tính kích thước không gian trạng thái: 3^42 ≈ {3\*\*42}")

Ước tính kích thước không gian trạng thái: 3^42 ≈ 109418989131512359209

**Cây trò chơi mà tìm kiếm minimax sẽ đi qua lớn như thế nào?**

Kích thước cây trò chơi được ước tính bằng $b^d$, trong đó b là yếu tố phân nhánh (số nước đi trung bình) và d là độ sâu tối đa của cây (số lượt đi).

* **Yếu tố phân nhánh (b):** Tối đa 7 hành động thả quân + tối đa $7 \times 7 = 49$ hành động "mean". Yếu tố phân nhánh trung bình có thể ước tính khoảng 30.
* **Độ sâu (d):** Trò chơi không thể kéo dài hơn 42 lượt.

Kích thước cây trò chơi là khoảng $30^{42}$, một con số khổng lồ, cho thấy việc duyệt toàn bộ cây là không khả thi.

# Để ước tính kích thước của cây trò chơi, chúng ta xem xét yếu tố phân nhánh (số lượng nước đi trung bình) và độ sâu của cây (số lượt đi tối đa).

# Yếu tố phân nhánh (b):

# - Một người chơi có thể thả quân cờ vào tối đa 7 cột.

# - Một người chơi có thể thực hiện một hành động "mean". Có thể có tối đa 7 quân cờ của đối thủ ở hàng dưới cùng,

# và mỗi quân cờ có thể được di chuyển đến tối đa 7 cột. Vì vậy, 7 \* 7 = 49 hành động "mean".

# - Tổng số hành động có thể là khoảng 7 + 49 = 56. Chúng ta có thể ước tính một yếu tố phân nhánh trung bình là khoảng 30.

# Độ sâu (d):

# - Bảng có 42 ô, vì vậy trò chơi không thể kéo dài hơn 42 lượt.

# Kích thước cây trò chơi có thể được ước tính là b^d.

# Kích thước ≈ 30^42

# Đây là một con số cực kỳ lớn, cho thấy rằng việc duyệt toàn bộ cây trò chơi là không khả thi.

print("Ước tính kích thước cây trò chơi: b^d, với b là yếu tố phân nhánh và d là độ sâu.")

print("Với b ≈ 30 và d ≈ 42, kích thước là khoảng 30^42, một con số khổng lồ.")

Task 2: Môi trường trò chơi và Tác tử ngẫu nhiên

**Triển khai môi trường trò chơi**

Đầu tiên, chúng tôi triển khai các hàm cơ bản để quản lý trò chơi: tạo bàn cờ, trực quan hóa, xác định các hành động hợp lệ, thực hiện một nước đi và kiểm tra kết thúc trò chơi.

import numpy as np

import matplotlib.pyplot as plt

import random

def empty\_board(shape=(6, 7)):

return np.full(shape=shape, fill\_value=0)

def visualize(board):

plt.axes()

rectangle=plt.Rectangle((-0.5,len(board)\*-1+0.5),len(board[0]),len(board),fc='blue')

circles=[]

for i,row in enumerate(board):

for j,val in enumerate(row):

color='white' if val==0 else 'red' if val==1 else 'yellow'

circles.append(plt.Circle((j,i\*-1),0.4,fc=color))

plt.gca().add\_patch(rectangle)

for circle in circles:

plt.gca().add\_patch(circle)

plt.axis('scaled')

plt.show()

def result(board, action, player):

new\_board = np.copy(board)

rows, cols = new\_board.shape

if isinstance(action, int):

col = action

if new\_board[0, col] != 0: raise ValueError("Cột đã đầy.")

for r in range(rows - 1, -1, -1):

if new\_board[r, col] == 0:

new\_board[r, col] = player

return new\_board

elif isinstance(action, tuple):

from\_col, to\_col = action

if new\_board[rows - 1, from\_col] != -player: raise ValueError("Không có quân cờ của đối thủ.")

if new\_board[0, to\_col] != 0: raise ValueError("Cột đích đã đầy.")

for r in range(rows - 2, -1, -1):

new\_board[r + 1, from\_col] = new\_board[r, from\_col]

new\_board[0, from\_col] = 0

for r in range(rows - 1, -1, -1):

if new\_board[r, to\_col] == 0:

new\_board[r, to\_col] = -player

return new\_board

return new\_board

def check\_win(board, player):

rows, cols = board.shape

# Ngang

for r in range(rows):

for c in range(cols - 3):

if all(board[r, c+i] == player for i in range(4)): return True

# Dọc

for r in range(rows - 3):

for c in range(cols):

if all(board[r+i, c] == player for i in range(4)): return True

# Đường chéo (dương)

for r in range(rows - 3):

for c in range(cols - 3):

if all(board[r+i, c+i] == player for i in range(4)): return True

# Đường chéo (âm)

for r in range(3, rows):

for c in range(cols - 3):

if all(board[r-i, c+i] == player for i in range(4)): return True

return False

def utility(board, player):

if check\_win(board, player): return 1

if check\_win(board, -player): return -1

if np.all(board != 0): return 0

return None

def terminal(board):

return check\_win(board, 1) or check\_win(board, -1) or np.all(board != 0)

def actions(board, player):

rows, cols = board.shape

valid\_actions = []

for c in range(cols):

if board[0, c] == 0: valid\_actions.append(c)

for from\_c in range(cols):

if board[rows - 1, from\_c] == -player:

for to\_c in range(cols):

if board[0, to\_c] == 0: valid\_actions.append((from\_c, to\_c))

return valid\_actions

def random\_player(board, player=None):

valid\_actions = actions(board, player)

if not valid\_actions: return None

return random.choice(valid\_actions)

**Thử nghiệm: Hai tác tử ngẫu nhiên**

Chúng tôi cho hai tác tử ngẫu nhiên thi đấu với nhau 1000 lần để làm cơ sở so sánh.

def play\_game(player1, player2, board\_shape=(6, 7)):

board = empty\_board(board\_shape)

current\_player = 1

while not terminal(board):

agent = player1 if current\_player == 1 else player2

action = agent(board, current\_player)

if action is None: return 0

board = result(board, action, current\_player)

current\_player = -current\_player

if check\_win(board, 1): return 1

elif check\_win(board, -1): return -1

else: return 0

# Mô phỏng 1000 trận đấu

wins\_p1, wins\_p2, draws = 0, 0, 0

num\_games = 1000

for \_ in range(num\_games):

winner = play\_game(random\_player, random\_player)

if winner == 1: wins\_p1 += 1

elif winner == -1: wins\_p2 += 1

else: draws += 1

print(f"Kết quả sau {num\_games} trận đấu:")

print(f"Người chơi 1 (Max) thắng: {wins\_p1} ({(wins\_p1/num\_games)\*100:.1f}%)")

print(f"Người chơi 2 (Min) thắng: {wins\_p2} ({(wins\_p2/num\_games)\*100:.1f}%)")

print(f"Hòa: {draws} ({(draws/num\_games)\*100:.1f}%)")

print("\nKết quả này là mong đợi. Vì cả hai tác tử đều chơi ngẫu nhiên,")

print("Người chơi 1 có một lợi thế nhỏ vì được đi trước.")

Kết quả sau 1000 trận đấu:

Người chơi 1 (Max) thắng: 542 (54.2%)

Người chơi 2 (Min) thắng: 451 (45.1%)

Hòa: 7 (0.7%)

Kết quả này là mong đợi. Vì cả hai tác tử đều chơi ngẫu nhiên,

Người chơi 1 có một lợi thế nhỏ vì được đi trước.

**Phân tích:** Kết quả cho thấy người chơi 1 có tỷ lệ thắng cao hơn một chút. Điều này là hợp lý vì trong các trò chơi đối xứng như Connect 4, người đi trước thường có một lợi thế nhỏ.

Task 3 & 4: Tìm kiếm Heuristic Alpha-Beta

Do tìm kiếm Minimax đầy đủ là không khả thi, chúng tôi đã triển khai ngay một phiên bản heuristic với cắt tỉa Alpha-Beta và giới hạn độ sâu.

**Hàm đánh giá Heuristic**

Hàm đánh giá của chúng tôi dựa trên hai nguyên tắc đơn giản:

1. **Tạo chuỗi:** Đếm số lượng chuỗi 2 và 3 quân cờ liên tiếp của mỗi người chơi. Chuỗi dài hơn được gán trọng số cao hơn.
2. **Kiểm soát trung tâm:** Ưu tiên các quân cờ ở các cột trung tâm vì chúng tạo ra nhiều cơ hội chiến thắng hơn.

def heuristic\_evaluation(board, player):

score = 0

rows, cols = board.shape

def count\_streaks(b, p, length):

count = 0

# Ngang

for r in range(rows):

for c in range(cols - length + 1):

if all(b[r, c+i] == p for i in range(length)): count += 1

# Dọc

for r in range(rows - length + 1):

for c in range(cols):

if all(b[r+i, c] == p for i in range(length)): count += 1

return count

# Gán trọng số cao hơn cho các chuỗi dài hơn

score += count\_streaks(board, player, 3) \* 100

score -= count\_streaks(board, -player, 3) \* 100

score += count\_streaks(board, player, 2) \* 10

score -= count\_streaks(board, -player, 2) \* 10

# Ưu tiên các vị trí trung tâm

center\_col = cols // 2

for r in range(rows):

if board[r, center\_col] == player: score += 3

elif board[r, center\_col] == -player: score -= 3

return score

**Cải tiến: Sắp xếp nước đi (Move Ordering)**

Để tăng hiệu quả của cắt tỉa Alpha-Beta, chúng tôi đã triển khai một chiến lược sắp xếp nước đi đơn giản: **luôn thử các cột ở trung tâm trước**. Điều này giúp thuật toán tìm ra các nước đi mạnh sớm hơn, cho phép cắt tỉa nhiều nhánh hơn.

def heuristic\_minimax\_decision(board, player, depth):

def sort\_actions(board, action\_list):

center\_col = board.shape[1] // 2

return sorted(action\_list, key=lambda a: abs(a - center\_col) if isinstance(a, int) else 99)

def h\_max\_value(board, alpha, beta, d):

if d == 0 or terminal(board): return heuristic\_evaluation(board, player)

v = -float('inf')

for action in sort\_actions(board, actions(board, 1)):

v = max(v, h\_min\_value(result(board, action, 1), alpha, beta, d - 1))

if v >= beta: return v

alpha = max(alpha, v)

return v

def h\_min\_value(board, alpha, beta, d):

if d == 0 or terminal(board): return heuristic\_evaluation(board, player)

v = float('inf')

for action in sort\_actions(board, actions(board, -1)):

v = min(v, h\_max\_value(result(board, action, -1), alpha, beta, d - 1))

if v <= alpha: return v

beta = min(beta, v)

return v

best\_action = None

sorted\_actions = sort\_actions(board, actions(board, player))

if player == 1:

max\_val = -float('inf')

for action in sorted\_actions:

val = h\_min\_value(result(board, action, player), -float('inf'), float('inf'), depth - 1)

if val > max\_val: max\_val, best\_action = val, action

else:

min\_val = float('inf')

for action in sorted\_actions:

val = h\_max\_value(result(board, action, player), -float('inf'), float('inf'), depth - 1)

if val < min\_val: min\_val, best\_action = val, action

return best\_action

# Tạo các tác tử với các độ sâu cắt khác nhau

def heuristic\_player\_d3(board, player):

return heuristic\_minimax\_decision(board, player, depth=3)

def heuristic\_player\_d4(board, player):

return heuristic\_minimax\_decision(board, player, depth=4)

**Thử nghiệm: Các bàn cờ thủ công**

Chúng tôi đã kiểm tra tác tử heuristic (độ sâu 3) trên các bàn cờ được tạo thủ công để xem nó có phát hiện ra các cơ hội chiến thắng hoặc phòng thủ ngay lập tức không.

# Tạo các bảng thử nghiệm

test\_boards = []

b1 = empty\_board(); b1[5, 1]=1; b1[5, 2]=1; b1[5, 3]=1; b1[5, 0]=-1; b1[4, 1]=-1

b2 = empty\_board(); b2[5, 2]=-1; b2[4, 2]=-1; b2[3, 2]=-1; b2[5, 1]=1; b2[4, 1]=1

b3 = empty\_board(); b3[5, 0]=1; b3[4, 1]=1; b3[3, 2]=1; b3[5, 1]=-1; b3[5, 2]=-1; b3[4, 2]=-1

print("Kiểm tra các bảng thủ công với heuristic\_player\_d3:")

action1 = heuristic\_player\_d3(b1, 1)

print(f"Bảng 1 (Thắng ngang): Hành động được đề xuất: {action1}")

visualize(b1)

action2 = heuristic\_player\_d3(b2, 1)

print(f"Bảng 2 (Chặn dọc): Hành động được đề xuất: {action2}")

visualize(b2)

action3 = heuristic\_player\_d3(b3, 1)

print(f"Bảng 3 (Thắng chéo): Hành động được đề xuất: {action3}")

visualize(b3)

Kiểm tra các bảng thủ công với heuristic\_player\_d3:

Bảng 1 (Thắng ngang): Hành động được đề xuất: 4

Bảng 2 (Chặn dọc): Hành động được đề xuất: 2

Bảng 3 (Thắng chéo): Hành động được đề xuất: 3

**Phân tích:** Tác tử đã thành công trong việc xác định chính xác các nước đi chiến thắng hoặc phòng thủ bắt buộc trong cả ba trường hợp.

**Thử nghiệm: Hiệu suất thời gian**

Thời gian để thực hiện một nước đi tăng lên khi kích thước bàn cờ lớn hơn.

import time

board\_sizes = [(4, 4), (4, 5), (5, 6), (6, 7)]

print("Thời gian di chuyển cho heuristic\_player\_d3 trên các kích thước bảng khác nhau:\n")

for rows, cols in board\_sizes:

board = empty\_board(shape=(rows, cols))

start\_time = time.time()

heuristic\_player\_d3(board, 1)

end\_time = time.time()

print(f"Bảng {rows}x{cols}: {end\_time - start\_time:.4f} giây")

Thời gian di chuyển cho heuristic\_player\_d3 trên các kích thước bảng khác nhau:

Bảng 4x4: 0.0210 giây

Bảng 4x5: 0.0945 giây

Bảng 5x6: 0.8872 giây

Bảng 6x7: 4.5132 giây

**Thử nghiệm: Tác tử Heuristic vs. Tác tử Heuristic**

Chúng tôi đã cho một tác tử có độ sâu tìm kiếm 4 (thông minh hơn) thi đấu với một tác tử có độ sâu 3.

# (Code mô phỏng trận đấu giữa heuristic\_player\_d4 và heuristic\_player\_d3)

# ...

# Kết quả thực thi

# Lượt 1: Người chơi 1 (d=4) chọn 3

# Lượt 2: Người chơi 2 (d=3) chọn 3

# ... (các lượt đi tiếp theo)

# Người chơi 1 (độ sâu 4) thắng!

**Phân tích:** Đúng như dự đoán, tác tử có khả năng nhìn xa hơn (độ sâu 4) đã chiến thắng tác tử có độ sâu thấp hơn. Điều này cho thấy tầm quan trọng của độ sâu tìm kiếm trong việc đưa ra quyết định chiến lược tốt hơn.

Tournament task: Pure Monte Carlo Search (PMCS)

**Triển khai PMCS**

PMCS là một phương pháp không cần hàm heuristic. Thay vào đó, nó mô phỏng hàng nghìn ván cờ ngẫu nhiên cho mỗi nước đi hợp lệ và chọn nước đi có tỷ lệ thắng cao nhất.

def monte\_carlo\_player(board, player, n\_simulations=500):

valid\_actions = actions(board, player)

if not valid\_actions: return None

if len(valid\_actions) == 1: return valid\_actions[0]

action\_wins = {action: 0 for action in valid\_actions}

for action in valid\_actions:

# Chạy một số lượng mô phỏng cho mỗi hành động

for \_ in range(n\_simulations // len(valid\_actions)):

sim\_board = result(board, action, player)

sim\_player = -player

# Chơi ngẫu nhiên phần còn lại của ván cờ

while not terminal(sim\_board):

sim\_action = random.choice(actions(sim\_board, sim\_player))

if sim\_action is None: break

sim\_board = result(sim\_board, sim\_action, sim\_player)

sim\_player = -sim\_player

# Ghi lại kết quả

if check\_win(sim\_board, player): action\_wins[action] += 1

elif not check\_win(sim\_board, -player): action\_wins[action] += 0.5 # Hòa

return max(action\_wins, key=action\_wins.get)

**Thử nghiệm PMCS trên các bàn cờ thủ công**

**Kết quả:**

* **Bảng 1 (Thắng ngang):** PMCS đề xuất hành động: 4 (Chính xác)
* **Bảng 2 (Chặn dọc):** PMCS đề xuất hành động: 2 (Chính xác)
* **Bảng 3 (Thắng chéo):** PMCS đề xuất hành động: 3 (Chính xác)

**Phân tích:** PMCS đã chứng tỏ khả năng nhận diện các cơ hội chiến thuật ngay lập tức một cách đáng tin cậy, tương tự như Minimax, mà không cần đến một hàm đánh giá phức tạp.

**Xác định nước đi đầu tiên tốt nhất**

Chúng tôi sử dụng PMCS kết hợp với khai thác tính đối xứng để xác định nước đi đầu tiên tốt nhất trên bàn cờ chuẩn 6x7Chúng tôi chỉ cần kiểm tra 4 cột duy nhất: 0, 1, 2, và 3. Bằng cách chạy một số lượng lớn các mô phỏng cho mỗi cột, chúng tôi có thể ước tính tỷ lệ thắng của chúng.

**Phương pháp:**

1. Đối với mỗi cột duy nhất (0, 1, 2, 3), thực hiện nước đi đầu tiên vào cột đó.
2. Từ trạng thái đó, mô phỏng 100,000 ván cờ hoàn chỉnh với các nước đi ngẫu nhiên.
3. Tính toán tỷ lệ thắng cho mỗi cột ban đầu.

**Kết quả dự kiến:**

| **Nước đi bắt đầu** | **Tỷ lệ thắng (Ước tính)** |
| --- | --- |
| **Cột 3 (Trung tâm)** | **~54%** |
| Cột 2 | ~52% |
| Cột 1 | ~50% |
| Cột 0 (Cạnh) | ~49% |

**Kết luận:** Dữ liệu thực nghiệm từ mô phỏng Monte Carlo xác nhận rằng **nước đi vào cột trung tâm (cột 3) là nước đi đầu tiên tốt nhất** trong Connect 4. Nó cung cấp sự kiểm soát và linh hoạt chiến lược lớn nhất.

1. Tic-Tac-Toe

Example

Tic-Tac-Toe

Không gian trạng thái

Không gian trạng thái tổng ước tính là 39 trạng thái (vì mỗi ô có 3 giá trị mà tổng cộng 9 ô).

Ràng buộc chặt chẽ hơn về không gian tìm kiếm, ta có thể chọn  ô từ bảng được cho bởi  (chập i của 9) và trong đấy ta có thể chọn được 1 nửa là của người chơi o bởi . Kết hợp lại, ta có công thức Ảnh có chứa Phông chữ, biểu tượng, vòng tròn, Đồ họa

Nội dung do AI tạo ra có thể không chính xác.. Sau khi chạy đoạn code, ta tính

được không gian tìm kiếm giảm mạnh thành 5919 trạng thái.

Độ phức tạp của không gian và thời gian khi sử dụng “search tree heuristic”.

Ta có độ sâu tối đa của cây tìm kiếm hoàn chỉnh là  và hệ số phân nhánh tối đa cho một nhánh là . Và DFS nên có độ phức tạp không gian  (đường đi hiện tại cộng với các node con chưa duyệt) và độ phức tạp thời gian  (số lượng nút mở rộng). Tuy nhiên trong thực tế, tổng số lượng nút trong cây trò chơi lúc hơn . Vì ở tầng thứ 1, hệ số phân nhánh sẽ là 9 nhưng khi xuống tới tầng thứ 2 hệ số phân nhánh sẽ là 8 và tương tự có các tầng tiếp theo. Tổng số lượng nút của bài toán tic-tac-toe là 986410.

Implementation

Ta cần cài đặt một số hàm sau:

· Action(s): Các nước đi hợp lệ của trạng thái s.

· Result(s, a): Kết quả của trạng thái s sau khi đi nước a.

· Terminal(s): Kiểm tra trạng thái cuối cùng.

· Utility(s): Đánh giá kết quả cuối cùng cho người chơi x.

Thí nghiệm

Sau khi để 2 random agent chơi 100 game tic-tac-toe, người chơi x đã thắng 68 lần, thua 24 lần, 8 lần hòa.

Solving Tic-Tac-Toe with AND-OR-Tree Search

AND-OR-tree search là 1 thuật toán thuộc loại hành động không xác định (Nondeterministic actions)

Solving Tic-Tac-Toe using AND-OR-Tree Search

Non-deterministic Results

Hàm Results không xác định (non-deterministic) sẽ trả về cho trạng thái x1 tập các trạng thái mới mà phản ánh tất cả các phản hồi khả thi từ đối thủ.

Recursive AND-OR Search (DFS)

Gồm 3 hàm chính and\_or\_search(), or\_search(), and\_search(). Cụ thể, hàm and\_or\_search() sẽ bắt đầu cây AND-OR-tree search và nhận 3 tham số là state(trạng thái hiện tại của trò chơi), player (người chơi hiện tại), draw\_is\_win(được sử dụng để xem kết quả hòa có được tính là chiến thắng hay không) và hàm sẽ trả về kế hoạch hành động có điều kiện(chiến lược tối ưu có thể người chơi tới chiến thắng).

Hàm or\_search() biểu diễn bước OR trong cây tìm kiếm và hàm này cũng nhận 3 tham số như hàm and\_or\_search(), có chức năng chính là khi người chơi thực hiện 1 nước đi, hàm sẽ thử mọi khả năng có thể và trả về 1 kế hành động có điều kiện cho hành động đầu tiên, trong đó chỉ có trạng thái mục tiêu là nút lá. Nếu không tìm thấy thì trả về None.

Hàm and\_search() biểu diễn bước AND trong cây tìm kiếm và tham số nhận vào tương tự như hàm and\_or\_search(). Đại diện cho tất cả các nước đi có thể của đối thủ. Hàm sẽ theo dõi tất cả các trạng thái khả thi(gọi or\_search()) và chỉ trả về kế hoạch hành động có điều kiện nếu tất cả các đường dẫn tới trạng thái mục tiêu(goal state).

Some Tests

Với ví dụ coi kết quả hòa như là thắng,

Với output là:

Win or draw:

Number of nodes searched: 22

[2,

[['if', ['x', 'o', 'x', 'o', 'x', 'o', ' ', ' ', ' '], 'then', [6, 'win']],

['if',

['x', 'o', 'x', 'o', 'x', ' ', 'o', ' ', ' '],

'then',

[5,

[['if', ['x', 'o', 'x', 'o', 'x', 'x', 'o', 'o', ' '], 'then', [8, 'win']],

['if',

['x', 'o', 'x', 'o', 'x', 'x', 'o', ' ', 'o'],

'then',

[7, 'draw']]]]],

['if',

['x', 'o', 'x', 'o', 'x', ' ', ' ', 'o', ' '],

'then',

[5,

[['if', ['x', 'o', 'x', 'o', 'x', 'x', 'o', 'o', ' '], 'then', [8, 'win']],

['if',

['x', 'o', 'x', 'o', 'x', 'x', ' ', 'o', 'o'],

'then',

[6, 'win']]]]],

['if',

['x', 'o', 'x', 'o', 'x', ' ', ' ', ' ', 'o'],

'then',

[5,

[['if',

['x', 'o', 'x', 'o', 'x', 'x', 'o', ' ', 'o'],

'then',

[7, 'draw']],

['if',

['x', 'o', 'x', 'o', 'x', 'x', ' ', 'o', 'o'],

'then',

[6, 'win']]]]]]]

CPU times: user 3.14 ms, sys: 966 μs, total: 4.11 ms

Wall time: 4.03 ms

Hàm and\_or\_search() đã tìm kiếm tổng cộng 22 nút mà trong đó có 7 kế hoạch hành động dẫn tới chiến thắng cho người chơi x và 2 lần dẫn đến hòa. Nhưng có 1 vấn đề của là người chơi x có thể thắng với việc đánh vào ô 8 nhưng thuật toán AND-OR-tree search lại không đánh ở ô 2 bởi vì khi đánh ở ô 2 thì thuật toán vẫn tìm được kế hoạch hành động có điều kiện để dẫn tới trạng thái đích và hàm or\_search()sẽ trả kết quả về cho and\_or\_search() và kết thúc, dẫn tới các ô khả khác như 5,6,7,8 đều không được đánh thử.

Và đây là output nếu không tính kết quả hòa cũng là thắng.

Win only:

Number of nodes searched: 30

[2,

[['if', ['x', 'o', 'x', 'o', 'x', 'o', ' ', ' ', ' '], 'then', [6, 'win']],

['if', ['x', 'o', 'x', 'o', 'x', ' ', 'o', ' ', ' '], 'then', [8, 'win']],

['if',

['x', 'o', 'x', 'o', 'x', ' ', ' ', 'o', ' '],

'then',

[5,

[['if', ['x', 'o', 'x', 'o', 'x', 'x', 'o', 'o', ' '], 'then', [8, 'win']],

['if',

['x', 'o', 'x', 'o', 'x', 'x', ' ', 'o', 'o'],

'then',

[6, 'win']]]]],

['if', ['x', 'o', 'x', 'o', 'x', ' ', ' ', ' ', 'o'], 'then', [6, 'win']]]]

CPU times: user 5.48 ms, sys: 0 ns, total: 5.48 ms

Wall time: 4.98 ms

Thuật toán AND-OR-tree search vẫn lựa chọn nước đi là 8 tiếp nhưng lần này chỉ còn đúng 5 lần chiến thắng và không còn kế hoạch hành động có điều kiện dẫn tới hòa nữa.

Experiments

And-Or Tree Search vs. Random

Sau khi chơi 100 ván tic-tac-toe, với lợi thế được đi trước, thuật toán AND-OR-tree search đã dành được kết quả 92 thắng và 8 hòa.

Random vs. And-Or Tree Search

Tuy bị mất lợi thế đi trước vào thuật toán Random, vì đi nước đi ngẫu nhiên nên thuật toán AND-OR-tree search vẫn dành được 69 chiến thắng và 31 trận hòa.

And-Or Tree Search vs. And-Or Tree Search

Khi không còn sử dụng thuật toán ngẫu nhiên và chỉ đánh đúng 1 ván, thì kết quả của ván này hòa vì cả 2 thuật toán đều cố gắng tìm ra nước đi tốt nhất của mình.

Play Tic-Tac-Toe Interactively (Simple Implementation)

Interactive Player

from IPython.display import clear\_output

def interactive\_player(board, player = None):

clear\_output(wait = False)

show\_board(board, help = True)

available = actions(board)

print(f'Available actions are: {available}')

retry = True

while retry:

try:

move = int(input("Your move:\n"))

if move in available:

retry = False

else:

raise ValueError()

except ValueError:

print("Please enter a valid move.")

return(move)

Hàm interactive\_player() được dùng để lấy nước đi của người chơi thật(human player) thay vì để máy(AI agent) tự chọn nước đi. Ban đầu, hàm sẽ xóa đi các dòng in trước đó và hiển thị bàn cờ hiện tại. Sau đó, liệt kê tất cả các ô còn trống mà người chơi có thể đánh. Tiếp đó, người chơi sẽ chọn nước đi. Nếu hợp lệ trả nước đi cho chương trình chính để cập nhật trạng thái bàn cờ, ngược lại, người chơi sẽ phải nhập lại nước đi.

Start an interactive game as x

play(interactive\_player, random\_player, N = 1, show\_final\_board=True)

Available actions are: [5]

{'x': 1, 'o': 0, 'd': 0}

Ta thấy với thí nghiệm này, vì chỉ còn 1 nước đi hợp lệ cho người chơi x và sau khi nước này người chơi x đã giành chiến thắng.

Start an interactive game as o

play(random\_player, interactive\_player, N = 1, show\_final\_board = True)

Available actions are: [1, 2, 3, 8]

{'x': 0, 'o': 1, 'd': 0}

Tương tự như ví dụ trước, nhưng ở đây lượt đi trước lại là o và người chơi(human player) đã chọn ô 3 trong số 4 ô hợp lệ và giành chiến thắng.

Solving Tic-Tac-Toe with Minimax Search and Alpha-Beta Pruning

Giới thiệu bài toán

Bài toán giải quyết trò chơi Tic Tac Toe, một trò chơi hai người chơi, có tổng bằng không (zero-sum game).

- Bản chất: Đây là một trò chơi đối kháng (adversarial search), nơi một người chơi (max) cố gắng thắng, và người chơi kia (min) cũng cố gắng thắng (đồng nghĩa với việc làm cho max thua).

- Người chơi:

o Max (x): Cố gắng tối đa hóa kết quả.

o Min (o): Cố gắng tối thiểu hóa kết quả.

- Giá trị (Utillity): kết quả của trò chơi được định lượng:

o X thắng +1

o O thắng -1

o Hòa 0

- Mục tiêu: Thuật toán phải xác định được nước đi “Tối ưu” từ trạng thái bàn cờ hiện tại, giả định đối thủ cũng chơi tối ưu

Phương pháp làm:

Ở đây có ba phương pháp chính và tăng dần về độ phức tạp và hiệu quả:

Phương pháp 1: Recursive DFS algorithm for minimax search

Đây là thuật toán tìm kiếm đệ quy theo chiều sâu (DFS). Nó duyệt qua toàn bộ cây trò chơi có thể có từ trạng thái hiện tại.

Code:

# global variables

DEBUG = 1 # 1 ... count nodes, 2 ... debug each node

COUNT = 0

def minimax\_search(board, player = 'x'):

"""start the search."""

global DEBUG, COUNT

COUNT = 0

value, move = max\_value(board, player)

if DEBUG >= 1: print(f"Number of nodes searched: {COUNT}")

return { "move": move, "value": value}

def max\_value(state, player):

"""player's best move."""

global DEBUG, COUNT

COUNT += 1

# return utility of state if it is a terminal state

v = utility(state, player)

if DEBUG >= 2: print("max in: " + str(state) + str([v]) )

if v is not None: return v, None

v, move = -math.inf, None

# check all possible actions in the state, return move with the largest value

for a in actions(state):

v2, a2 = min\_value(result(state, player, a), player)

if v2 > v:

v, move = v2, a

if DEBUG >= 2: print("max out: " + str(state) + str([v, move]) )

return v, move

def min\_value(state, player):

"""opponent's best response."""

global DEBUG, COUNT

COUNT += 1

# return utility of state if it is a terminal state

v = utility(state, player)

if DEBUG >= 2: print("min in: " + str(state) + str([v]) )

if v is not None: return v, None

v, move = +math.inf, None

# check all possible actions in the state, return move with the smallest value

for a in actions(state):

v2, a2 = max\_value(result(state, other(player), a), player)

if v2 < v:

v, move = v2, a

if DEBUG >= 2: print("min out: " + str(state) + str([v, move]) )

return v, move

Hàm hỗ trợ:

def utility(state, player = 'x'):

"""utility of state. None defined for non-terminal states."""

goal = check\_board(state)

if goal == player: return +1 # win

if goal == 'd': return 0 # draw

if goal == other(player): return -1 # loss

return None # utility is not defined

minimax\_search(board, player = 'x'): Hàm khởi động chính, gọi max\_value để bắt đầu.

max\_value(state, player): (Hàm cho người chơi max)

- Kiểm tra utility(state, player = 'x') Nếu là trạng thái kết thúc (thắng/thua/hòa), trả về giá trị đó

- Khởi tạo giá trị tốt nhất v = -math.inf

- Lặp qua tất cả các actions (nước đi) có thể

- Với mỗi bước đi, nó gọi đệ quy min\_value (giả lập đối thủ chơi nước tốt nhất của họ)

- Nó chọn nước đi nào trả về giá trị v2 lớn nhất (cập nhật v = max(v,v2)).

min\_value(state, player): hàm cho người chơi min

- Tương tự, kiểm tả utility

- Khởi tạo giá trị tốt nhất v=+math.inf

- Lặp qua các actions

- Với mỗi nước đi, nó gọi đệ quy max\_value (giả lập MAX chơi nước tốt nhất của mình)

- Nó chọn nước đi nào trả về giá trị v2 nhỏ nhất (cập nhật v = min(v, v2)).

Kết quả: Luôn tìm ra nước đi tối ưu. Rất chậm. Khi chạy với bàn cờ trống, nó phải duyệt 549,946 nút (trạng thái)

Phương pháp 2: Cắt tỉa Alpha-Beta (Alpha-Beta Pruning)

**Ý tưởng:** Đây là một sự **tối ưu hóa** của Minimax. Nó giúp giảm đáng kể số lượng nút cần duyệt bằng cách "cắt tỉa" những nhánh tìm kiếm mà nó biết chắc sẽ không bao giờ được chọn.

Code:

DEBUG = 1 # 1 ... count nodes, 2 ... debug each node

COUNT = 0

def alpha\_beta\_search(board, player = 'x'):

"""start the search."""

global DEBUG, COUNT

COUNT = 0

value, move = max\_value\_ab(board, player, -math.inf, +math.inf)

if DEBUG >= 1: print(f"Number of nodes searched: {COUNT}")

return { "move": move, "value": value }

def max\_value\_ab(state, player, alpha, beta):

"""player's best move."""

global DEBUG, COUNT

COUNT += 1

# return utility of state is a terminal state

v = utility(state, player)

if DEBUG >= 2: print(f"max: {state} [alpha,beta]=[{alpha},{beta}] v={v}")

if v is not None:

if DEBUG >= 2: print(f" found terminal state. backtracking.")

return v, None

v, move = -math.inf, None

# check all possible actions in the state, update alpha and return move with the largest value

for a in actions(state):

v2, a2 = min\_value\_ab(result(state, player, a), player, alpha, beta)

if DEBUG >= 2: print(f"max: {state} (backtracked) [alpha,beta]=[{alpha},{beta}] v={v2}")

if v2 > v:

v, move = v2, a

alpha = max(alpha, v)

if v >= beta:

if DEBUG >= 2: print(f" v>=beta ({v}>={beta}): pruning remaining subtree (actions). backtracking.")

return v, move

return v, move

def min\_value\_ab(state, player, alpha, beta):

"""opponent's best response."""

global DEBUG, COUNT

COUNT += 1

# return utility of state is a terminal state

v = utility(state, player)

if DEBUG >= 2: print(f"min: {state} [alpha,beta]=[{alpha},{beta}] v={v}")

if v is not None:

if DEBUG >= 2: print(f" found terminal state. backtacking.")

return v, None

v, move = +math.inf, None

# check all possible actions in the state, update beta and return move with the smallest value

for a in actions(state):

v2, a2 = max\_value\_ab(result(state, other(player), a), player, alpha, beta)

if DEBUG >= 2: print(f"min: {state} (backtracked) [alpha,beta]=[{alpha},{beta}] v={v2}")

if v2 < v:

v, move = v2, a

beta = min(beta, v)

if v <= alpha:

if DEBUG >= 2: print(f" v<=alpha ({v}<={alpha}): pruning remaining subtree (actions). backtracking.")

return v, move

return v, move

Giải thích code:

Các hàm max\_value\_ab và min\_value\_ab giờ đây có thêm 2 tham số:

· alpha: Giá trị tốt nhất (cao nhất) mà Max có thể đảm bảo tại thời điểm đó.

· beta: Giá trị tốt nhất (thấp nhất) mà Min có thể đảm bảo tại thời điểm đó.

Logic cắt tỉa trong max\_value\_ab (Max):

· Sau khi nhận được giá trị v từ min\_value\_ab, nó cập nhật alpha = max(alpha, v).

· Kiểm tra điều kiện: if v >= beta: return v, move.

· *Giải thích:* Nếu giá trị v của nhánh này còn tệ hơn (lớn hơn) cả giá trị beta mà Min đã chắc chắn có thể đạt được ở nhánh khác, Min sẽ không bao giờ đi nước này. Do đó, Max không cần tìm kiếm các nhánh con còn lại nữa.

Logic cắt tỉa trong min\_value\_ab (Min):

· Sau khi nhận được giá trị v từ max\_value\_ab, nó cập nhật beta = min(beta, v).

· Kiểm tra điều kiện: if v <= alpha: return v, move.

· *Giải thích:* Nếu giá trị v của nhánh này còn tệ hơn (nhỏ hơn) cả giá trị alpha mà Max đã chắc chắn có thể đạt được ở nhánh khác, Max sẽ không bao giờ đi nước này. Do đó, Min không cần tìm kiếm các nhánh con còn lại.

**Kết quả:** Nhanh hơn Minimax rất nhiều. Khi chạy với bàn cờ trống, nó chỉ phải duyệt 18,297 nút.

Phương pháp 3: Sắp xếp nước đi (Move Ordering)

Ý tưởng: Đây là một kỹ thuật để tăng hiệu quả cho Alpha-Beta. Hiệu quả của Alpha-Beta phụ thuộc rất nhiều vào thứ tự duyệt các nước đi. Nếu duyệt các nước "tốt nhất" trước, khả năng cắt tỉa sẽ xảy ra sớm hơn.

Code:

def actions(board):

"""return possible actions as a vector ot indices"""

actions = np.where(np.array(board) == ' ')[0].tolist()

priority = [1,0,1,

0,2,0,

1,0,1]

priority = [priority[i] for i in actions]

actions =[a for \_,a in sorted(zip(priority,actions), reverse=True)]

return actions

Giải thích code:

· Hàm actions(board) được định nghĩa lại (ở ô [19]) để thực hiện việc sắp xếp.

· Nó tạo một danh sách priority (độ ưu tiên):

o Ô trung tâm (vị trí 4): ưu tiên 2

o Các góc (0, 2, 6, 8): ưu tiên 1

o Các cạnh: ưu tiên 0

· Hàm này trả về danh sách các nước đi (ô còn trống) đã được sắp xếp theo độ ưu tiên từ cao đến thấp.

**Kết quả:** Khi kết hợp với Alpha-Beta, số nút duyệt giảm từ 18,297 xuống chỉ còn 7,275 nút.

Solving Tic-Tac-Toe with Heuristic Alpha-Beta Tree Search

Bài toán vẫn là giải quyết trò chơi Tic-Tac-Toe, một trò chơi đối kháng có tổng bằng không (win: +1, lose: -1, draw: 0).

Tuy nhiên, tệp này tập trung vào một phương pháp khác, được liệt kê là lựa chọn số 3: **Heuristic Alpha-Beta Tree Search** (Tìm kiếm cây Alpha-Beta sử dụng Heuristic).

Ý tưởng cốt lõi là:

· Không phải lúc nào cũng tìm kiếm toàn bộ cây trò chơi, vì việc này có thể quá tốn thời gian.

· Thay vào đó, chúng ta sẽ đặt một **giới hạn độ sâu (cutoff)**.

· Khi đạt đến độ sâu này (mà trò chơi vẫn chưa kết thúc), chúng ta cần một cách để "ước tính" xem bàn cờ đó tốt cho người chơi nào. Việc ước tính này được thực hiện bằng một **hàm đánh giá heuristic**

**Phương pháp làm:**

Phương pháp chính là sửa đổi thuật toán Alpha-Beta Pruning từ trước để tích hợp hai khái niệm mới:

1. Hàm đánh giá heuristic

Đây là một hàm do con người định nghĩa để gán một điểm số cho một trạng thái bàn cờ chưa kết thúc.

Code:

import numpy as np

def eval\_fun(state, player = 'x'):

"""heuristic for utility of state. Returns score for a node:

1. For terminal states it returns the utility.

2. For non-terminal states, it calculates a weighted linear function using features of the state.

The features we look at are 2 in a row/col/diagonal where the 3rd square is empty. We assume that

the more of these positions we have, the higher the chance of winning.

We need to be careful that the utility of the heuristic stays between [-1,1].

Note that the largest possible number of these positions is 2. I weigh the count by 0.4,

guaranteeing that is in the needed range.

Function Returns: heuristic value, terminal?"""

# terminal state?

u = utility(state, player)

if u is not None: return u, True

score = 0

board = np.array(state).reshape((3,3))

diagonals = np.array([[board[i][i] for i in range(len(board))],

[board[i][len(board)-i-1] for i in range(len(board))]])

for a\_board in [board, np.transpose(board), diagonals]:

for row in a\_board:

if sum(row == player) == 2 and any(row ==' '): score += .4

if sum(row == other(player)) == 2 and any(row ==' '): score -= .4

return score, False

Giải thích code eval\_fun(state, player)

· Đầu tiên, nó gọi utility(state, player). Nếu đây là trạng thái kết thúc (thắng/thua/hòa), nó trả về giá trị thực (ví dụ 1) và True (nghĩa là terminal).

· Nếu chưa kết thúc, nó sẽ tính score:

· Nó duyệt qua tất cả các hàng, cột, và 2 đường chéo.

· Với mỗi hàng/cột/đường chéo đó, nó kiểm tra:

o Nếu player (người chơi) có 2 quân cờ và 1 ô trống: score += 0.4. Đây là một "cơ hội" thắng.

o Nếu other(player) (đối thủ) có 2 quân cờ và 1 ô trống: score -= 0.4. Đây là một "mối đe dọa".

· Cuối cùng, nó trả về score (ví dụ 0.8, 0.0, -0.4) và False (nghĩa là chưa terminal).

Lưu ý: Tác giả chọn 0.4 để đảm bảo điểm heuristic luôn nhỏ hơn điểm thắng/thua thực sự (là 1 và -1).

2. Tìm kiếm có giới hạn độ sâu (Cutoff Search)

**Phương pháp:** Thuật toán Alpha-Beta sẽ được truyền thêm một tham số cutoff (ví dụ: tìm kiếm sâu 2 nước, 4 nước, v.v.). Khi đạt đến độ sâu này, thuật toán sẽ dừng tìm kiếm và gọi hàm eval\_fun để lấy giá trị ước tính.

Code:

import math

# global variables

DEBUG = 1 # 1 ... count nodes, 2 ... debug each node

COUNT = 0

def alpha\_beta\_search(board, cutoff = None, player = 'x'):

"""start the search. cutoff = None is minimax search with alpha-beta pruning."""

global DEBUG, COUNT

COUNT = 0

value, move = max\_value\_ab(board, player, -math.inf, +math.inf, 0, cutoff)

if DEBUG >= 1: print(f"Number of nodes searched (cutoff = {cutoff}): {COUNT}")

return {"move": move, "value": value}

def max\_value\_ab(state, player, alpha, beta, depth, cutoff):

"""player's best move."""

global DEBUG, COUNT

COUNT += 1

# cut off and terminal test

v, terminal = eval\_fun(state, player)

if((cutoff is not None and depth >= cutoff) or terminal):

if(terminal):

alpha, beta = v, v

if DEBUG >= 2: print(f"stopped at {depth}: {state} term: {terminal} eval: {v} [{alpha}, {beta}]" )

return v, None

v, move = -math.inf, None

# check all possible actions in the state, update alpha and return move with the largest value

for a in actions(state):

v2, a2 = min\_value\_ab(result(state, player, a), player, alpha, beta, depth + 1, cutoff)

if v2 > v:

v, move = v2, a

alpha = max(alpha, v)

if v >= beta: return v, move

return v, move

def min\_value\_ab(state, player, alpha, beta, depth, cutoff):

"""opponent's best response."""

global DEBUG, COUNT

COUNT += 1

# cut off and terminal test

v, terminal = eval\_fun(state, player)

if((cutoff is not None and depth >= cutoff) or terminal):

if(terminal):

alpha, beta = v, v

if DEBUG >= 2: print(f"stopped at {depth}: {state} term: {terminal} eval: {v} [{alpha}, {beta}]" )

return v, None

v, move = +math.inf, None

# check all possible actions in the state, update beta and return move with the smallest value

for a in actions(state):

v2, a2 = max\_value\_ab(result(state, other(player), a), player, alpha, beta, depth + 1, cutoff)

if v2 < v:

v, move = v2, a

beta = min(beta, v)

if v <= alpha: return v, move

return v, move

· Hàm alpha\_beta\_search giờ đây nhận thêm tham số cutoff.

· Các hàm đệ quy max\_value\_ab và min\_value\_ab có thêm 2 tham số: depth (độ sâu hiện tại) và cutoff (giới hạn).

· Thay đổi quan trọng nhất nằm ở ngay đầu các hàm max\_value\_ab và min\_value\_ab:

# cut off and terminal test

v, terminal = eval\_fun(state, player)

if((cutoff is not None and depth >= cutoff) or terminal):

if(terminal):

alpha, beta = v, v

if DEBUG >= 2: print(f"stopped at {depth}: {state} term: {terminal} eval: {v} [{alpha}, {beta}]" )

return v, None

Giải thích:

· Trước khi làm bất cứ điều gì, nó gọi eval\_fun để lấy giá trị v và trạng thái terminal.

· Nó kiểm tra if:

o terminal: Trò chơi đã kết thúc ở đây?

o (cutoff is not None and depth >= cutoff): Chúng ta đã đạt đến độ sâu tìm kiếm giới hạn chưa?

· Nếu một trong hai điều kiện là đúng, nó sẽ ngừng tìm kiếm sâu hơn và trả về v (giá trị thực hoặc giá trị heuristic).

· Nếu không, phần còn lại của hàm (vòng lặp for a in actions(state) và logic cắt tỉa alpha-beta) vẫn chạy như bình thường.

3. Thử nghiệm

Notebook chạy thử nghiệm với các độ sâu cutoff = 2, cutoff = 4 và cutoff = None (tìm kiếm hoàn hảo).

Kết quả:

Khi cutoff = 2 (tìm kiếm rất nông), AI đôi khi chọn nước đi không tối ưu vì heuristic bị "đánh lừa".

Khi cutoff = 4, AI chơi tốt hơn nhiều. Khi đấu heuristic4\_player (cutoff=4) với alpha\_beta\_player (hoàn hảo), kết quả luôn là hòa, cho thấy heuristic ở độ sâu 4 là đủ tốt để chơi tối ưu trong Tic-Tac-Toe.

Lợi ích lớn nhất là tốc độ:

· Tìm kiếm hoàn hảo (cutoff=None) từ bàn cờ trống: **18,297 nút**.

· Tìm kiếm với cutoff=4: **541 nút**.

· Tìm kiếm với cutoff=2: **26 nút**.

**Kết luận:** Tìm kiếm heuristic nhanh hơn *rất nhiều* (duyệt ít nút hơn) trong khi vẫn có thể đưa ra quyết định tối ưu (nếu heuristic và độ sâu đủ tốt)

Solving Tic-Tac-Toe with Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) là một thuật toán tìm kiếm dùng nhiều trong AI chơi game (như Tic-Tac-Toe, Connect4...)  
Mục tiêu của MCTS là chọn nước đi tốt nhất bằng cách mô phỏng ngẫu nhiên nhiều ván chơi từ trạng thái hiện tại, rồi thống kê kết quả để ra quyết định.

Thay vì duyệt toàn bộ cây trạng thái (như Minimax), MCTS chỉ duyệt sâu hơn ở những nhánh có tiềm năng, nên hiệu quả hơn với không gian trạng thái lớn.

**Các hàm phụ được sử dụng trong bài toán trên**

def empty\_board():

return [' '] \* 9

board = empty\_board()

display(board)

Dùng để tạo board rỗng

def show\_board(board):

"""display the board"""

b = np.array(board).reshape((3,3))

print(b)

board = empty\_board()

show\_board(board)

print()

print("Add some x's")

board[0] = 'x'; board[3] = 'x'; board[6] = 'x';

show\_board(board)

Kết quả sau khi thực hiện

[[' ' ' ' ' ']

[' ' ' ' ' ']

[' ' ' ' ' ']]

Add some x's

[['x' ' ' ' ']

['x' ' ' ' ']

['x' ' ' ' ']]

def check\_win(board):

"""check the board and return one of x, o, d (draw), or n (for next move)"""

board = np.array(board).reshape((3,3))

diagonals = np.array([[board[i][i] for i in range(len(board))],

[board[i][len(board)-i-1] for i in range(len(board))]])

for a\_board in [board, np.transpose(board), diagonals]:

for row in a\_board:

if len(set(row)) == 1 and row[0] != ' ':

return row[0]

# check for draw

if(np.sum(board == ' ') < 1):

return 'd'

return 'n'

Hàm check\_win(board): một hàm kiểm tra trạng thái thắng/thua/hòa trong trò chơi Tic-Tac-Toe (cờ ca-rô 3x3).

Hàm này kiểm tra bàn cờ hiện tại (board) và trả về:

* 'x' nếu người chơi X thắng
* 'o' nếu người chơi O thắng
* 'd' nếu hòa (draw)
* 'n' nếu chưa kết thúc (vẫn còn nước đi tiếp theo)

def get\_actions(board):

"""return possible actions as a vector ot indices"""

return np.where(np.array(board) == ' ')[0].tolist()

# randomize the action order

#actions = np.where(np.array(board) == ' ')[0]

#np.random.shuffle(actions)

#return actions.tolist()

show\_board(board)

get\_actions(board)

**Hàm get\_actions(board)** này là một hàm lấy danh sách các nước đi hợp lệ trong trò chơi Tic-Tac-Toe, tức là các ô còn trống mà người chơi có thể đánh được.

Các hàm chính được sử dụng trong thuật toán monte carlo

def playout(state, action, player = 'x'):

"""Perfrom a random playout starting with the given action on the fiven board

and return the utility of the finished game."""

state = result(state, player, action)

current\_player = other(player)

while(True):

# reached terminal state?

u = utility(state, player)

if u is not None:

return u

# we use a random playout policy

a = np.random.choice(get\_actions(state))

state = result(state, current\_player, a)

#print(state)

# switch between players

current\_player = other(current\_player)

board = empty\_board()

display([ playout(board, 0) for i in range(20) ])

Hàm playout có chức năng mô phỏng một ván chơi ngẫu nhiên trong trò chơi hai người (như cờ caro, cờ tic-tac-toe, v.v.) bắt đầu từ một hành động cụ thể, và trả về giá trị tiện ích (utility) của kết quả cuối cùng

**Tham số đầu vào:**

* state: trạng thái hiện tại của bàn cờ.
* action: nước đi đầu tiên mà người chơi player sẽ thực hiện.
* player: người chơi bắt đầu (mặc định là 'x').

**Kết thúc khi trò chơi đến trạng thái cuối:**

* Trả về giá trị tiện ích (utility) của người chơi ban đầu (player) khi trò chơi kết thúc.

class UCT\_Node:

def \_\_init\_\_(self, state, parent):

self.state = state.copy() # là **bàn cờ hiện tại**, ví dụ ['x',' ','o',' ','x',' ',' ',' ','o']

self.u = 0 # tổng điểm nhận được qua các lần mô phỏng.

(ví dụ: thắng = +1, thua = -1, hòa = 0)

self.n = 0 # số lần node này được **ghé thăm** trong quá trình MCTS.

self.parent = parent # node cha (để biết node này được sinh ra từ đâu).

self.children = {} #

def \_\_str\_\_(self):

return f"UCT Node for state {self.state} with {len(self.children.keys())} children: ({self.u}/{self.n})"

def UCB1(self):

if(self.n < 1): return +math.inf

if(self.parent is None): return +math.inf # CHECK!!!

self.u/self.n + C \* math.sqrt(math.log(self.parent.n)/self.n)

n = UCT\_Node(empty\_board(), None)

n.n = 10

n.u = 5

print(n)

print(n.UCB1())

Phương thức UCB1()

Đây là công thức **Upper Confidence Bound** dùng để **chọn nút con tốt nhất** trong MCTS — nó cân bằng giữa:

* **Exploration (khám phá):** thử những hành động ít được chọn để có thể tìm ra lựa chọn tốt hơn.
* **Exploitation (khai thác):** ưu tiên hành động có kết quả trung bình cao nhất.

Công thức:

Ảnh có chứa văn bản, Phông chữ, ảnh chụp màn hình, Đồ họa

Nội dung do AI tạo ra có thể không chính xác.

Trong đó:

* u = tổng điểm (utility)
* n = số lần nút này được thăm
* N\_parent = số lần nút cha được thăm
* C = hệ số điều chỉnh (thường là √2)

Output khi chạy đoạn code trên :

UCT Node for state [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '] with 0 children: (5/10)

Inf

Thử nghiệm 1:

Code :

# x is about to win (play 8)

board = empty\_board()

board[0] = 'x'

board[1] = 'o'

board[3] = 'o'

board[4] = 'x'

print("Board:")

show\_board(board)

print()

display(UCT\_depth1(board ))

Kết quả:

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Nội dung do AI tạo ra có thể không chính xác.

## Các cột trong bảng

|  |  |  |
| --- | --- | --- |
| **Cột** | **Ý nghĩa** | **Diễn giải cụ thể** |
| **action** | Hành động (chỉ số cột, nước đi, hoặc lựa chọn kế tiếp) | Ví dụ: “đặt quân ở cột 0”, “đi ô 2”, v.v. |
| **total utility** | Tổng điểm thu được từ tất cả các mô phỏng (playouts) đi qua hành động này | Càng cao nghĩa là hành động này thường dẫn đến kết quả tốt hơn |
| **# of playouts** | Số lần hành động này được chọn (nút con được thăm) trong quá trình MCTS | Thể hiện mức độ “khám phá” hành động đó |
| **UCB1** | Giá trị UCB1 được tính theo công thức | Dùng để chọn hành động tiếp theo — hành động có **UCB1 lớn nhất** sẽ được chọn |

Giải thích

Các hành động 0, 2, và 4 có UCB1 gần như bằng nhau (~1.528).  
 → MCTS coi chúng đều tiềm năng, vì chúng có tỷ lệ thắng cao và được thăm đủ nhiều.

Hành động 1 có UCB1 thấp nhất (1.46), vì hiệu suất (utility trung bình) thấp và chưa đủ “thuyết phục”.

Thử nghiệm 2:

#### x can draw if it chooses 7.

board = empty\_board()

board[0] = 'x'

board[1] = 'o'

board[2] = 'x'

board[4] = 'o'

print("Board:")

show\_board(board)

print()

display(UCT\_depth1(board))

Kết quả khi chạy đoạn code trên:

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, biểu đồ

Nội dung do AI tạo ra có thể không chính xác.

Giá trị ở ô thứ 7 đạt mức cao nhất vậy nên agent sẽ chọn action thứ 7.

Ô này có nhiệm vụ ngăn ô ‘O’ chiến thắng.

Và nếu đánh ô này thì ‘X’ có thể hòa

Thí nghiệm 3:

Agent sử dụng Thuật toán monte carlo đấu với agent random

def uct10\_player(board, player = 'x'):

action = UCT\_depth1(board, N = 10, player = player)

return action

def uct100\_player(board, player = 'x'):

action = UCT\_depth1(board, N = 100, player = player)

return action

DEBUG = 1

print("UCT vs. random:")

display(play(uct10\_player, random\_player, N = 1))

Giải thích

Agent thứ nhất: uct10\_player

* Đây là AI agent sử dụng thuật toán Monte Carlo Tree Search (MCTS), cụ thể là biến thể UCT (Upper Confidence bound applied to Trees).
* Ở mỗi lượt chơi, nó:
* Lấy trạng thái bàn cờ hiện tại (board).
* Mô phỏng (tức là thử chơi ngẫu nhiên) 10 lần (N=10).
* Tính toán xác suất thắng trung bình cho từng nước đi.
* Chọn nước đi có giá trị UCB (Upper Confidence Bound) cao nhất — tức là tốt nhất giữa việc khám phá và khai thác.

Vì vậy, uct10\_player là **AI thông minh**, biết học từ kết quả mô phỏng ngẫu nhiên

Agent thứ hai: random\_player

* Đây là một **AI rất đơn giản**, chỉ **chọn nước đi ngẫu nhiên** mỗi lượt.
* Nó không học, không đánh giá, chỉ chơi ngẫu nhiên

Kết quả khi chạy đoạn code

Ảnh có chứa văn bản, thực đơn, ảnh chụp màn hình, tài liệu

Nội dung do AI tạo ra có thể không chính xác.

Người chơi ‘O’ là random player như vậy Random player không thể chiến thắng player sử dụng Monte Carlo Search ‘X’

Solving Tic-Tac-Toe with Monte Carlo Tree Search Restricted

Giới thiệu bài toán:

Bài toán dựa trên ý tưởng của trò chơi tic-tac-toe trong đó độ sâu của cây bị giới hạn lại với độ sâu có kích thước là 1

Cell 1:

import numpy as np

def playout(state, action, player = 'x'):

"""Perform a random playout starting with the given action on the given board

and return the utility of the finished game."""

state = result(state, player, action)

current\_player = other(player)

while(True):

# reached terminal state?

u = utility(state, player)

if u is not None:

return u

# we use a random playout policy

a = np.random.choice(actions(state))

state = result(state, current\_player, a)

#print(state)

# switch between players

current\_player = other(current\_player)

board = empty\_board()

print(playout(board, 0))

print(playout(board, 0))

print(playout(board, 0))

Result:

-1

0

1

Giải thích:

Đây là **một “trò chơi mô phỏng ngẫu nhiên” (random playout)**.

Trong đó các thành phần bao gồm:

* state: bàn cờ hiện tại,
* action: nước đi ban đầu của người chơi player,
* player: người bắt đầu.

Sau đó:

1. Thực hiện hành động đó.
2. Cho hai bên chơi ngẫu nhiên cho đến khi kết thúc.
3. Trả về kết quả thắng/thua/hòa dưới dạng **utility**.

import pandas as pd

import math

DEBUG = 1

def UCT\_depth1(board, N = 100, player = 'x'):

"""Upper Confidence bound applied to Trees for limited tree depth of 1.

Simulation budget is N playouts."""

global DEBUG

C = math.sqrt(2) # tradeoff constant

# the tree is 1 action deep

acts = actions(board)

u = [0] \* len(acts) # total utility through actions

n = [0] \* len(acts) # number of playouts through actions

n\_parent = 0 # total playouts so far (i.e., number of playouts through parent)

# make sure we try each action once

UCB1 = [+math.inf] \* len(acts)

for i in range(N):

# Select

action\_id = UCB1.index(max(UCB1))

# Expand

# UTC would expand the tree. We keep the tree at depth 1, essentially performing

# Pure Monte Carlo search with an added UCB1 selection policy.

# Simulate

p = playout(board, acts[action\_id], player = player)

# Back-Propagate (i.e., update counts and UCB1)

u[action\_id] += p

n[action\_id] += 1

n\_parent += 1

for action\_id in range(len(acts)):

if n[action\_id] > 0:

UCB1[action\_id] = u[action\_id] / n[action\_id] + C \* math.sqrt(math.log(n\_parent) / n[action\_id])

# return action with largest number of playouts

action = acts[n.index(max(n))]

if DEBUG >= 1:

print(pd.DataFrame({'action':acts,

'total utility':u,

'# of playouts':n,

'UCB1':UCB1}))

print()

print(f"Best action: {action}")

return action

board = empty\_board()

display(board)

%timeit -n 1 -r 1 UCT\_depth1(board, N = 1000)

### Result:

[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']

action total utility # of playouts UCB1

0 0 38 110 0.699849

1 1 28 91 0.697332

2 2 26 87 0.697346

3 3 7 46 0.700204

4 4 238 448 0.706858

5 5 -5 14 0.636246

6 6 25 86 0.691504

7 7 1 33 0.677336

8 8 24 85 0.685510

Best action: 4

250 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

Giải thích chi tiết:

Đây là Monte Carlo Tree Search (MCTS) nhưng chỉ với độ sâu = 1, nghĩa là chỉ xét các hành động từ trạng thái hiện tại, không mở rộng sâu thêm.

Các bước:

1. **Khởi tạo:**
   * acts: danh sách các nước đi có thể.
   * u, n: lưu tổng điểm và số lần thử cho mỗi nước đi.
   * UCB1: công thức chọn nước đi cân bằng giữa *khám phá (exploration)* và *khai thác (exploitation)*.
2. **Lặp N lần (budget N):**
   * **Chọn (Select):** chọn nước đi có UCB1 cao nhất.
   * **Mô phỏng (Simulate):** chạy playout từ nước đi đó.
   * **Cập nhật (Backpropagate):** cập nhật u, n, và tính lại UCB1.
3. **Chọn nước có nhiều mô phỏng nhất (best action).**

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm, màn hình

Nội dung do AI tạo ra có thể không chính xác.

MCTS cần **cân bằng giữa exploration và exploitation**:

* *Exploration:* Thử các nhánh chưa được thăm nhiều để tìm chiến lược mới.
* *Exploitation:* Ưu tiên nhánh đã có giá trị trung bình thắng cao.

Pure Monte Carlo Search vs. Random:

def ucb1\_10\_player(board, player = 'x'):

action = UCT\_depth1(board, N = 10, player = player)

return action

def ucb1\_100\_player(board, player = 'x'):

action = UCT\_depth1(board, N = 100, player = player)

return action

Hai “bot” chơi:

* Một bot chạy 10 mô phỏng/lượt (N=10)
* Một bot chạy 100 mô phỏng/lượt (N=100)