**TRƯỜNG ĐẠI HỌC SÀI GÒN**A picture containing application

Description automatically generated

**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 Văn Anh Tú – 3122410445

Trương Hữu Nghĩa - 3122410263

Đỗ Khôi Nguyên - 3122410266

Lai Tấn Tài - 3122410366

**TP. HCM THÁNG 10/2025**

MỤC LỤC

Contents

[Example 3](#_Toc212287136)

[Tic-Tac-Toe 3](#_Toc212287137)

[Solving Tic-Tac-Toe with AND-OR-Tree Search 4](#_Toc212287138)

[Play Tic-Tac-Toe Interactively (Simple Implementation) 7](#_Toc212287139)

[Solving Tic-Tac-Toe with Minimax Search and Alpha-Beta Pruning 9](#_Toc212287140)

[Phương pháp 1: Recursive DFS algorithm for minimax search 9](#_Toc212287141)

[Phương pháp 2: Cắt tỉa Alpha-Beta (Alpha-Beta Pruning) 12](#_Toc212287142)

[Phương pháp 3: Sắp xếp nước đi (Move Ordering) 15](#_Toc212287143)

[Solving Tic-Tac-Toe with Heuristic Alpha-Beta Tree Search 16](#_Toc212287144)

[Solving Tic-Tac-Toe with Monte Carlo Tree Search 21](#_Toc212287145)

[Phương thức UCB1() 25](#_Toc212287146)

[Các cột trong bảng 26](#_Toc212287147)

[Solving Tic-Tac-Toe with Monte Carlo Tree Search Restricted 29](#_Toc212287148)

[Result: 32](#_Toc212287149)

[Pure Monte Carlo Search vs. Random: 33](#_Toc212287150)

[Assignment 34](#_Toc212287151)

[Connect 4 34](#_Toc212287152)

[Task 1: Defining the Search Problem 34](#_Toc212287153)

[Task 2: Game Environment and Random Agent 35](#_Toc212287154)

[Task 3: Minimax Search with Alpha-Beta Pruning 40](#_Toc212287155)

[Task 4: 53](#_Toc212287156)

[Mean Connect 4 67](#_Toc212287157)

[Task 1: Defining the Search Problem 67](#_Toc212287158)

[Task 2: Game environment and random agent 69](#_Toc212287159)

[Task 3 71](#_Toc212287160)

[Task 4 73](#_Toc212287161)

[Task 1: Defining the Search Problem [10 point] 76](#_Toc212287162)

[Task 2: Game Environment and Random Agent [30 point] 80](#_Toc212287163)

[Task 3: Minimax Search with Alpha-Beta Pruning [30 points] 86](#_Toc212287164)

[Task 4: Heuristic Alpha-Beta Tree Search [30 points] 94](#_Toc212287165)

# 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 . 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:



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'

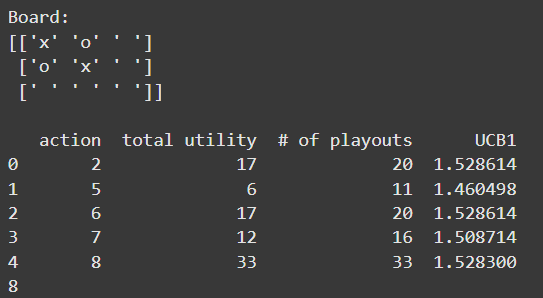
print("Board:")

show\_board(board)

print()

display(UCT\_depth1(board ))

Kết quả:



## 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'

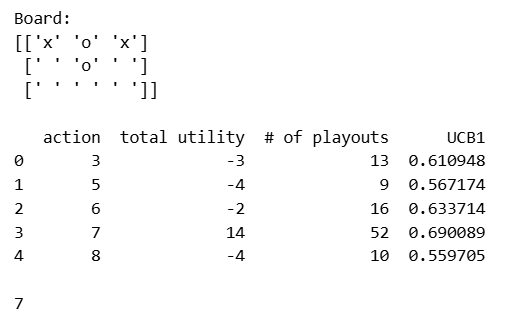
print("Board:")

show\_board(board)

print()

display(UCT\_depth1(board))

Kết quả khi chạy đoạn code trên:



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

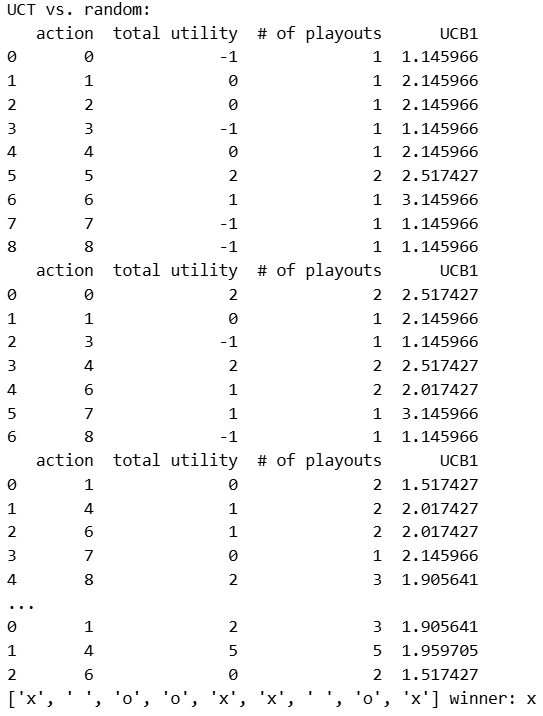
* Đây là AI agent sử dụng thuật toán Monte Carlo Tree Search (MCTS), cụ thể là biến thể UCT (Upper Confidence bound applied to Trees).
* Ở mỗi lượt chơi, nó:
* Lấy trạng thái bàn cờ hiện tại (board).
* Mô phỏng (tức là thử chơi ngẫu nhiên) 10 lần (N=10).
* Tính toán xác suất thắng trung bình cho từng 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



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

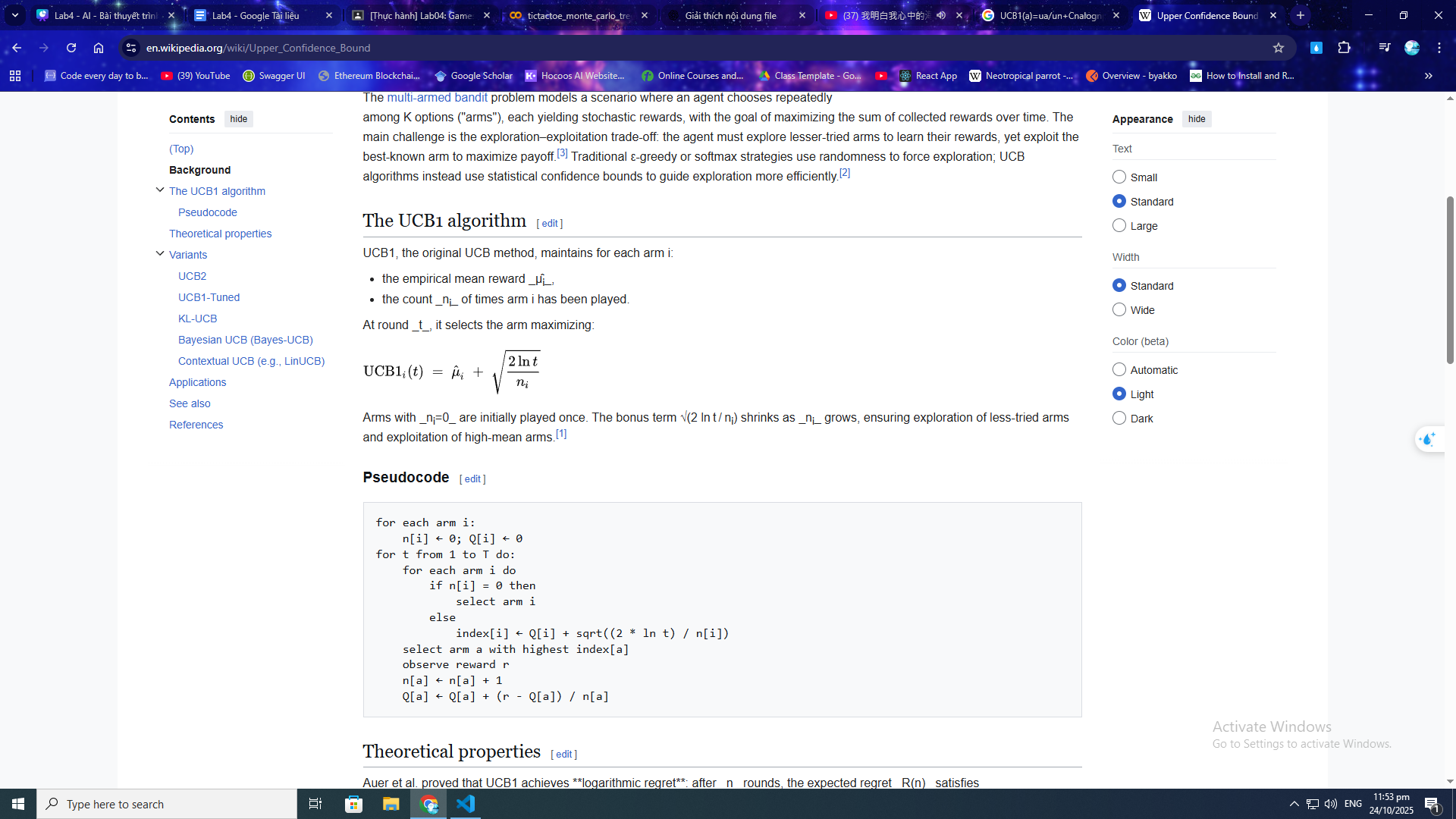
250 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

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).**



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)

# Assignment

# Connect 4

## Task 1: Defining the Search Problem

**Câu hỏi:**

**Define the components of the search problem associated with this game:**

**\* Initial state**

**\* Actions**

**\* Transition model**

**\* Test for the terminal state**

**\* Utility for terminal states**

**1. Trạng thái ban đầu (Initial State)**:

- Trạng thái khởi đầu là cấu hình ban đầu của trò chơi trước khi bất kỳ người chơi nào thực hiện nước đi.

- Nó chứa toàn bộ thông tin cần thiết để bắt đầu trò chơi.

- Trong trò Connect Four, trạng thái khởi đầu là một bảng 6×7 trống.

- Mỗi ô đều rỗng (thường biểu diễn bằng giá trị 0).

**2 .Hành động (action)**

- Actions mô tả tất cả các hành động hợp lệ mà người chơi có thể thực hiện tại một trạng thái nhất định.

- Trong Connect Four, mỗi hành động tương ứng với việc thả quân vào một cột chưa đầy (có 7 hành động nếu chưa có cột nào đầy).

- Tập hành động thay đổi linh hoạt trong suốt trò chơi (khi cột đầy thì hành động đó không còn hợp lệ)

**3. Mô hình chuyển trạng thái (Transition Model / Result Function)**

- Mô hình chuyển trạng thái (hay hàm kết quả) mô tả cách một hành động biến đổi trạng thái hiện tại sang trạng thái mới.

- Trong Connect Four, khi người chơi chọn một cột, mô hình này sẽ tạo ra một bản sao mới của bàn cờ và đặt quân cờ vào vị trí **thấp nhất có thể trong cột đó.**

**4. Trạng thái đích (Goal / Terminal State) và Hàm tiện ích (Utility Function)**

- Trạng thái đích (hoặc trạng thái kết thúc) là khi trò chơi đạt đến điểm không thể thực hiện hành động hợp lệ nào khác.

- Một trạng thái được xem là kết thúc nếu:

- Có một người chơi đạt được 4 quân liên tiếp (theo hàng ngang, dọc hoặc chéo), hoặc

- Bàn cờ đã đầy mà không ai thắng (hòa).

- Hàm tiện ích (Utility) là hàm gán giá trị số cho các trạng thái kết thúc, thể hiện mức độ tốt xấu của trạng thái đó đối với người chơi:

- Giá trị dương (ví dụ +1 hoặc +∞) → AI thắng.

Giá trị âm (ví dụ −1 hoặc −∞) → AI thua.

Giá trị 0 → Hòa.

**Câu hỏi: How big is the state space ? Give an estimate and explain it:**

Với bài toán connect four , ta có 42 ô tất cả ( 6 x 7 ) , mỗi ô có 3 giá trị ( 1 , -1 , 0 )

Tương ứng với thắng , thua , hòa

Không gian trạng thái là bài toán tổ hợp của 3 giá trị của mỗi ô trên tổng số 42 ô vậy nên kích thước của không gia trạng thái là 342

**Câu hỏi: How is the game tree that minimax search will go through ? Give an estimate and explain it ?**

Với bài toán connect four

Ta có

Bàn cờ có 6 hàng và 7 cột. Vậy có tổng 42 ô

Mỗi trạng thái bàn cờ là những ô đã được điền

Mỗi 1 trạng thái sẽ có 7 khả năng có thể di chuyển

Trường hợp tệ nhất nếu 2 người chơi đều chơi tối ưu và không mắc sai lầm là O(n,m) = 7n x m

Với n là số hàng và m là số cột

## Task 2: Game Environment and Random Agent

Yêu cầu:Implement helper functions for:

A check for available actions in each state `actions(state)`.

The transition model `result(state, player, action)`.

Check for terminal states `terminal(state)`.

The utility function `utility(state, player)`.

Nhưng em sẽ custom các hàm theo ý của mình

def player(board):

flat = [cell for row in board for cell in row]

count\_1 = flat.count(1)

count\_neg1 = flat.count(-1)

sum = count\_1 + count\_neg1 #Tính tổng số quân cờ

return 1 if sum % 2 == 0 else -1 #Nếu tổng là chẵn thì lượt đánh tiếp theo thuộc về agent 1, và ngược lại

Hàm này sử dụng trạng thái bàn cờ hiện tại để xác định lượt đánh tiếp theo của người chơi

def actions(last\_row\_each\_col,move\_ordering): #Hàm liệt kê các trạng thái hiện có , Trả về list các cột có thể đánh

available = []

for col, row\_idx in enumerate(last\_row\_each\_col):

if row\_idx >= 0:

available.append(col)

if move\_ordering == 1:

center = len(last\_row\_each\_col) // 2

available.sort(key=lambda c: abs(center - c)) # gần trung tâm hơn -> ưu tiên hơn

else:

random.shuffle(available) # Trộn list lại để tránh trường hợp di chuyển theo thứ tự từ cột 1 đến cột 7

return available

Mục đích của hàm này là sinh ra trạng thái có thể di chuyển tiếp theo , đầu vào là

Danh sách cột có thể đánh **last\_row\_each\_col**

Về **last\_row\_each\_col**

Giả sử nếu là bàn cờ trống với 6 dòng , 7 cột thì last\_row\_ban đầu là [5,5,5,5,5,5,5]

Giả sử đánh vào cột thứ 1 thì giá trị tại đó sẽ giảm đi 1 đơn vị = [4,5,5,5,5,5,5]

Giả sử nếu cột thứ 1 đã đầy có nghĩa là [-1,5,5,5,5,5,5]

Thì hàm này sẽ thêm những cột có thể đánh là [1,2,3,4,5,6]

Về **Move\_order** sẽ được giải thích ở phần sau

def result(board, last\_row\_each\_col, action, player\_to\_move):

col = action

#copy

new\_board = copy.deepcopy(board)

new\_last = copy.deepcopy(last\_row\_each\_col) # copy

row\_to\_place = new\_last[col] # Từ giá trị cột ta suy ra hàng

new\_board[row\_to\_place][col] = player\_to\_move # Thay giá trị tại ô đấy bằng quân cờ hiện tại

new\_last[col] -= 1 # Giá trị hàng có thể đánh tại vị trí cột đấy giảm đi 1

return new\_board, new\_last

Hàm này đầu vào là ma trận trạng thái hiện tại **board**, **last\_row\_each\_col** (đã giải thích ở trên) , **action** ( số hành động ) , **player\_to\_move** (người chơi hiện tại)

Đầu ra là Trạng thái bàn cờ mới **new\_board** và **new\_last** số hàng có thể đánh tại mỗi cột

def terminal\_test(board, last\_row\_each\_col):

"""

Tra ve True neu tro choi ket thuc (co nguoi thang hoac hoa), nguoc lai tra ve False.

"""

# Check for win

for r in range(len(board)):

for c in range(len(board[0])):

if board[r][c] != 0:

player = board[r][c]

# Kiểm tra 4 ô ở hàng ngang

if c <= len(board[0]) - 4 and all(board[r][c+i] == player for i in range(4)):

return player ,True

# Kiểm tra 4 ô ở hàng dọc

if r <= len(board) - 4 and all(board[r+i][c] == player for i in range(4)):

return player, True

# Kiểm tra đường chéo chính

if r >= 3 and c <= len(board[0]) - 4 and all(board[r-i][c+i] == player for i in range(4)):

return player, True

# Kiểm tra đường chéo phụ

if r <= len(board) - 4 and c <= len(board[0]) - 4 and all(board[r+i][c+i] == player for i in range(4)):

return player ,True

# Check for draw

if all(row == -1 for row in last\_row\_each\_col):

return 0, True

return 0 , False

Hàm **terminal\_check** sử dụng để kiểm tra trận đấu có kết thúc hay chưa và kết quả như thế nào

Đầu vào là ma trận bàn cờ hiện tại **board** và số cột khả dụng **last\_row\_each\_col**

Hàm sẽ kiểm tra 4 ô trong bàn cờ theo 4 hướng thẳng đứng , nằm ngang , chéo chính , chéo phụ dựa vào đoạn code trên

Đầu ra của hàm là player đại diện cho người chơi thắng cuộc **( 1 , -1 , 0 ) , Trả về True** khi trận đấu kết thúc **, False** khi trận đấu còn đang tiến hành

def utility(board, last\_row\_each\_col):

winner, is\_terminal = terminal\_test(board, last\_row\_each\_col)

if is\_terminal:

if winner == 1:

return 10

elif winner == -1:

return 10

else:

return 0

return 0

Hàm này dùng để tính score ở mỗi state và trả về số điểm tương ứng

**Hàm random cho agent**

# Your code/ answer goes here.

import random

def random\_agent(board, last\_row\_each\_col):

available\_actions = actions(last\_row\_each\_col,False) # Trả về các cột có thể đánh , VD[0,1,3,4,5,6] , cột thứ 2 không thể đánh do đầy

return random.choice(available\_actions)

Câu hỏi :Let two random agents play against each other 1000 times.How often does each player win? Is the result expected?

# Your code/ answer goes here.

def play\_game():

board = empty\_board() #Khởi tạo bàn cờ trống

last\_row\_each\_col = [len(board) - 1] \* len(board[0]) # Khởi tạo các cột có thể thả quân cờ

current\_agent = player(board) #Hàm xác định người chơi hiện tại

while True:

action = random\_agent(board, last\_row\_each\_col)

board, new\_last\_row\_each\_col = result(board, last\_row\_each\_col, action, current\_agent)

winner, is\_terminal = terminal\_test(board, new\_last\_row\_each\_col)

#Kiểm tra trận đấu kết thúc chưa và trả về người thắng

if is\_terminal:

if winner == 1:

return 1

elif winner == -1:

return -1

else:

return 0

#Cập nhật các cột mới có thể đánh sau khi agent thực hiện hành động

last\_row\_each\_col = new\_last\_row\_each\_col

#Cập nhật lại lượt của agent hiện tại

current\_agent = player(board)

def count\_winner(): #Hàm thống kê thắng thua cả 2 agent

dic = {

1: 0,

0: 0,

-1:0

}

for i in range(0,1000):

dic[play\_game()]+=1

return dic

print(count\_winner())

Hàm trên giả lập 2 agent đấu nhau với lựa chọn random sau 1000 trận và thống kê kết quả

Kết quả sau khi gọi hàm **count\_winner()**

{1: 532, 0: 1, -1: 467}

Có nghĩa là agent 1

Thắng 532 trận

Hòa 1 trận

Thua 467 trận

## Task 3: Minimax Search with Alpha-Beta Pruning

Ý tưởng thuật toán Minimax Search with Alpha Beta Pruning

Minimax mô phỏng quá trình ra quyết định tối ưu của hai người chơi:

Một bên cố tối đa hóa điểm số (gọi là MAX).

Bên còn lại cố tối thiểu hóa điểm số (gọi là MIN).

Giả sử người chơi muốn chiến thắng là MAX thì giá trị lớn nhất trong các trạng thái hay còn gọi là nước đi của MIN.

Nếu người chơi MIN muốn chiến thắng thì chọn nước đi thấp nhất mà MAX có thể đi

**Implement minimax search starting from a given board for specifying the player.**

Ta sẽ tái sử dụng các hàm **Utility , Terminal\_check , Result, actions , player** đã xác định ở trên

Để dễ tính toán ta sẽ định nghĩa lại cấu trúc dữ liệu đã đề cập ở trên dưới dạng đối tượng

class Node:

def \_\_init\_\_(self, board, last\_row\_each\_col, parent=None):

self.board = board #Bàn cờ

self.last\_row\_each\_col = last\_row\_each\_col #Số cột khả dụng có thể đánh

self.parent = parent #Node cha

self.children = [] #Các trạng thái con

self.value = None #Giá trị tối ưu của trạng thái hiện tại

Một node đại diện cho 1 trạng thái bàn cờ tại 1 thời điểm

def minimax\_decision(node,depth,max\_depth,move\_ordering=False):

if terminal\_test(node.board, node.last\_row\_each\_col)[1] or depth == max\_depth: # Dừng khi đạt độ sâu tối đa hoặc kết thúc

node.value = utility(node.board, node.last\_row\_each\_col)

return None,node.value

available\_actions = actions(node.last\_row\_each\_col, move\_ordering) # Sinh ra trạng thái tiếp theo có thể đi

current\_player = player(node.board) # Hàm trả về lượt người chơi dựa trên trạng thái bàn cờ hiện tại

best\_value = float('-inf') if current\_player == 1 else float('inf')

for action in available\_actions: #Lặp qua từng cột trong list các cột khả dụng

new\_board, new\_last = result(node.board, node.last\_row\_each\_col, action, current\_player) # Trả về trạng thái mới giả lập

child\_node = Node(new\_board, new\_last, parent=node) # Khởi tạo đối tượng node mới

node.children.append(child\_node) #Thêm node mới vào node hiện tại

\_ ,value = minimax\_decision(child\_node,depth+1,max\_depth,move\_ordering) # trả về giá trị tốt nhất ứng với từng người chơi và action để đạt được giá trị đấy

if current\_player == 1: # Maximizing player

if value > best\_value:

best\_value = value

best\_action = action

else: # Minimizing player

if value < best\_value:

best\_value = value

best\_action = action

node.value = best\_value

best\_action = random.choice(available\_actions) if available\_actions else None #chọn cột còn khả dụng ngẫu nhiên nếu không có nước đi nào tối ưu

return best\_action, best\_value #Trả về cột ứng với giá trị tối ưu và giá trị tối ưu đó

**Yêu cầu :**Experiment with some manually created boards (at least 5) to check if the agent spots winning opportunities.

Giả sử ta tạo ra 5 bảng với mỗi bảng là 1 trạng thái bàn cờ tại 1 thời điểm

Ta sẽ lần lượt thử nghiệm trên các bảng đã tạo

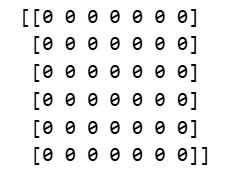
Với quân đỏ là quân có lượt đi tiếp theo thì ta sẽ kiểm tra liệu quân đỏ có chọn được ô cuối cùng giúp chiến thắng không

Hàm khởi tạo bàn cờ rỗng ngẫu nhiên với kích thước 6 x 7

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

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

**print(empty\_board())** ta được ma trận

****

Với mỗi 1 bảng ta sẽ tạo 1 trạng thái bất kỳ và gọi hàm **minimax\_decision()** đã code ở trên

**Code:**

board = [[0, 0, -1, 0, -1],

[-1, 0, 1, 0, 1],

[1, -1, -1, 1, -1],

[-1, -1, 1, 1, 1],

[1, -1, -1, 1, 1]]

board = np.array(board)

print("Bảng 1")

print(f"Trang thai ban dau ")

visualize(board)

last\_row\_each\_col = np.sum(board == 0, axis=0) - 1

node = Node(board,last\_row\_each\_col)

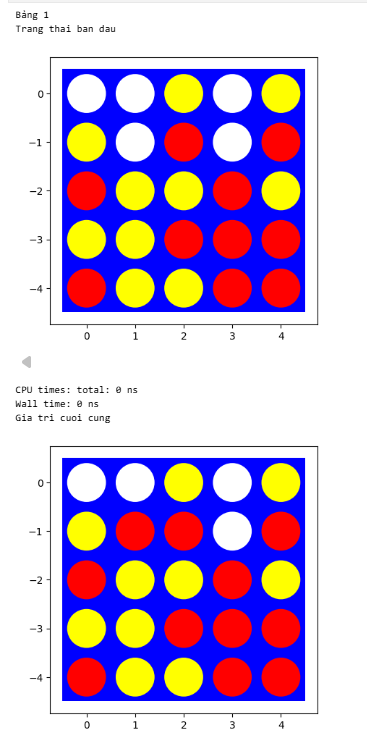
action, \_ = minimax\_decision(node,0,4)

new\_board, new\_last\_row\_each\_col = result(board, last\_row\_each\_col, action, player(node.board))

print(f"Gia tri cuoi cung”)

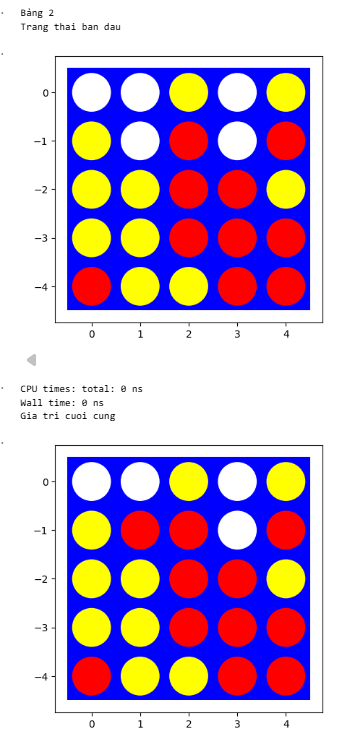
visualize(new\_board)

**Với kết quả bảng 1:**

****

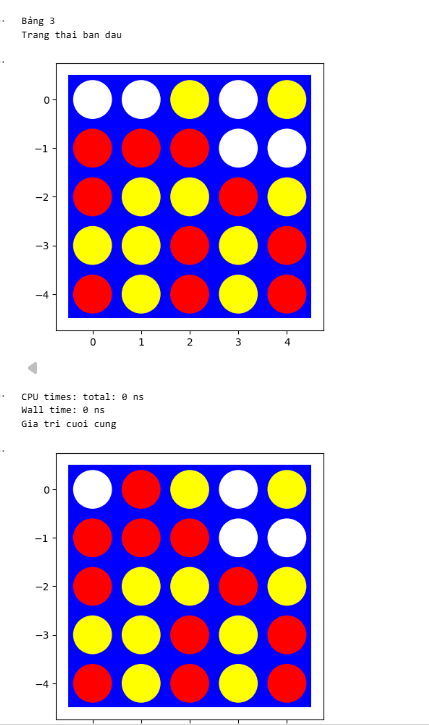
**Nhận xét:** Agent không thắng do chặn nước đi tiếp theo của quân cờ vàng

**Bảng 2:**

****

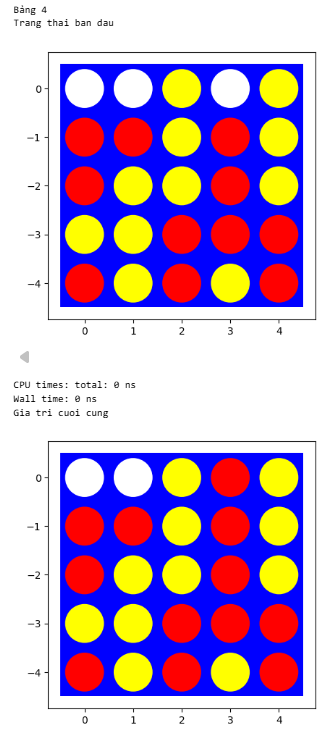
**Nhận xét :Agent đã có thể nhìn ra nước đi chiến thắng tiếp theo**

**Bảng 3:**

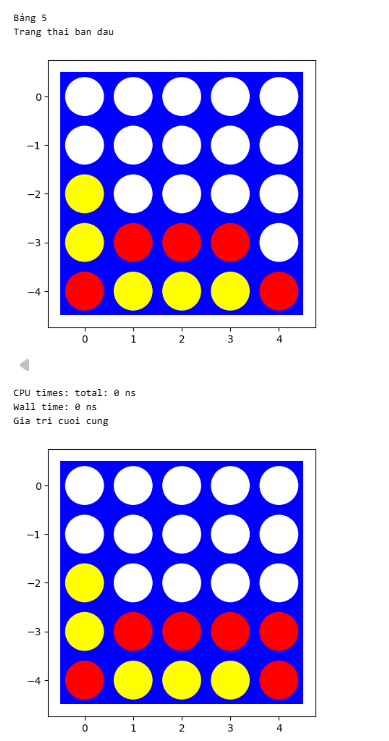
****

**Nhật xét :** Cũng giống như bảng 2 , agent có thể quyết định được nước đi theo giúp dành chiến thắng dễ dàng

**Bảng 4 cũng tương tự:**

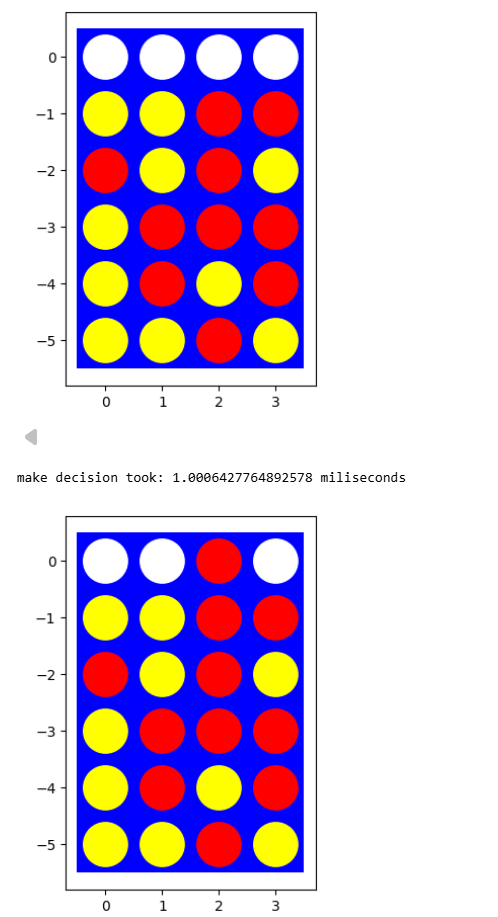
****

**Bảng 5**

****

**Như mấy bảng trên thì agent vẫn có thể quyết định tối ưu nhất giúp chiến thắng**

**Câu hỏi :** How long does it take to make a move? Start with a smaller board with 4 columns and make the board larger by adding columns/rows. Explain why using this algorithm on a standard **6 x 7** board is not feasible.



Bảng đầu tiên là trạng thái agent ( đỏ ) chuẩn bị chọn nước cờ

Bảng thứ 2 là trạng thái agent(đỏ) đã đánh xong nước cờ

Mất 1.0006 miliseconds để agent quyết định

**Move ordering**

Trong thuật toán Alpha-Beta pruning, hiệu quả cắt tỉa phụ thuộc vào thứ tự duyệt các nước đi.

Chiến lược Move Ordering giúp sắp xếp nước đi sao cho các lựa chọn tiềm năng cao được xét trước, qua đó tăng khả năng cập nhật nhanh giá trị alpha và beta.

Trong bài này, ta sử dụng chiến lược center-first, ưu tiên các cột ở trung tâm vì chúng thường tạo ra nhiều hướng thắng hơn trong trò Mean Connect-4.

Thử nghiệm

import time

board = [[0, 0, 0, 0, 0],

[0, 0, 0, 0, 0],

[0, 0, 0, 0, 0],

[0, -1, 0, 1, 1],

[0, -1, 0, -1, 1]]

board = np.array(board)

print(f"Trang thai ban dau ")

visualize(board)

last\_row\_each\_col = np.sum(board == 0, axis=0) - 1

node = Node(board,last\_row\_each\_col)

#so sánh Move\_odering = True và False

t0 = time.time()

action, \_ = minimax\_decision(node,0,4,True) #Move\_ordering

t1 = time.time()

# new\_board, new\_last\_row\_each\_col = result(board, last\_row\_each\_col, action, player(node.board))

action, \_ = minimax\_decision(node,0,4,False)# without Move\_ordering

t2 = time.time()

print(f"Using move ordering took: {(t1 - t0) \* 1e3} miliseconds")

print(f"Without move ordering took: {(t2 - t1) \* 1e3} miliseconds")

Kết quả sau khi chạy đoạn code



Với bàn cờ có ít quân vào những thời điểm đầu thì khi sử dụng move ordering sẽ tối ưu hơn vì không cần phải duyệt cây quá sâu để tìm ra action tốt nhất

**Câu hỏi: Start with an empty board. This is the worst case scenario for minimax search since it needs solve all possible games that can be played (minus some pruning) before making the decision. What can you do?**

1. Giới hạn thời gian và độ sâu (Depth Limiting và Iterative Deepening)

2. Sử dụng Hàm đánh giá mạnh mẽ

Ví dụ trong connect4: Đánh vào trung tâm thường là ưu tiên. Bằng cách gán các giá trị heuristic ngay cả cho bàn cờ trống, ta có thể sắp xếp thứ tự các nước đi, giúp Alpha-Beta hoạt động hiệu quả hơn.

3. Sắp xếp thứ tự nước đi (Move Ordering)

Nước đi tốt trước: Sử dụng kinh nghiệm hoặc hàm đánh giá đơn giản để xác định nước đi có vẻ hứa hẹn nhất và xét chúng trước.

Khi một nước đi rất tốt được tìm thấy sớm, nó sẽ tạo ra alpha-beta hẹp hơn, cho phép cắt bỏ rất nhiều nhánh kém hiệu quả sau đó.

**Yêu cầu:Let the Minimax Search agent play a random agent on a 4 x 4 board. Analyze wins, losses and draws.**

Code:

def play\_game(board, max\_depth=4):

board = np.array(board) # đảm bảo numpy array

last\_row\_each\_col = np.sum(board == 0, axis=0) - 1

while True:

# kiểm tra terminal trước khi chọn nước đi

winner, is\_terminal = terminal\_test(board, last\_row\_each\_col)

if is\_terminal:

return winner

current\_agent = player(board)

if current\_agent == 1:

# tạo root node cập nhật cho trạng thái hiện tại

root\_node = Node(board, last\_row\_each\_col)

action, \_ = minimax\_decision(root\_node, depth=0, max\_depth=max\_depth)

if action is None:

return 0 # không còn nước đi -> hòa (hoặc xử lý khác)

else:

action = random\_agent(board, last\_row\_each\_col)

# áp dụng nước đi và cập nhật trạng thái

board, last\_row\_each\_col = result(board, last\_row\_each\_col, action, current\_agent)

def count\_winner(n\_games=1000, init\_board=None):

dic = {1: 0, 0: 0, -1: 0} #dictionary chứa thông tin thắng , hòa , thua của người chơi

for \_ in range(n\_games):

b = empty\_board() if init\_board is None else np.array(init\_board)

res = play\_game(b)

dic[res] += 1

return dic

board = np.zeros((4,4))

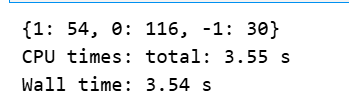
#player = 1 , use minimax

#player = -1 ,random

%time print(count\_winner(n\_games=200, init\_board=board))

Đoạn code trên giúp đếm số trận thắng , hòa , thua của người chơi sau 200 trận

Kết quả sau khi chạy đoạn code trên:



“1” : số lần người chơi 1 chiến thắng

“0” : số lần người chơi 1 và -1 hòa

“-1”: số lần người chơi 1 thua

## Task 4:

**Định nghĩa Heuristic**

import numpy as np

import math

def evaluate\_window(window, player):

opponent = -player

score = 0

player\_count = np.sum(window == player)

opponent\_count = np.sum(window == opponent)

empty\_count = np.sum(window == 0)

if player\_count == 4:

score = 1000

elif player\_count == 3 and empty\_count == 1:

score = 100

elif player\_count == 2 and empty\_count == 2:

score = 10

elif opponent\_count == 3 and empty\_count == 1:

score = -950

elif opponent\_count == 2 and empty\_count == 2:

score = -100

return score

def heuristic(node):

board = node.board

rows, cols = board.shape

terminal = terminal\_test(board, node.last\_row\_each\_col) #Hàm terminal kiểm tra trạng thái bàn cờ đã kết thúc hay chưa

winner = terminal[0] #Người thắng

end = terminal[1] #True nếu kết thúc và ngược lại

curr\_player = player(board)

# Tính score của trung tâm bàn cờ giúp agent ưu tiên đánh vào vị trí này

# {

center\_array = [board[r][cols // 2] for r in range(rows)]

center\_count = center\_array.count(curr\_player)

score = 0

score += center\_count \* 30

#}

if end:

if winner != 0:

return winner \* 1000

else:

return 0

# Check cột

for c in range(cols):

for r in range(rows - 3):

window = board[r:r+4, c]

score += evaluate\_window(window, curr\_player)

# Check hàng ngang

for r in range(rows):

for c in range(cols - 3):

window = board[r, c:c+4]

score += evaluate\_window(window, curr\_player)

# Check đường chéo chính

for r in range(rows - 3):

for c in range(cols - 3):

window = [board[r+i, c+i] for i in range(4)]

score += evaluate\_window(window, curr\_player)

# Check đường chéo phụ

for r in range(rows - 3):

for c in range(3, cols):

window = [board[r+i, c-i] for i in range(4)]

score += evaluate\_window(window, curr\_player)

return score

Điểm của hàm heuristic dựa trên số quân theo hàng , theo cột hoặc đường chéo chính hoặc phụ của bàn cờ tại 1 trạng thái nhất định.

Với mỗi 1 ô tại 1 vị trí (x,y) bất kỳ của 1 bàn cờ ta sẽ gọi hàm evaluate\_window để ước lượt score trong 4 quân tiếp theo

Công dụng của hàmevaluate\_window

Đầu vào hàm chứa 1 list gồm 4 ô liên tiếp theo 4 hướng chính và lượt người chơi hiện tại ( dùng để xác định score )

Hàm này dùng để xét 4 ô liên tiếp và tính score

if player\_count == 4:

score = 1000

elif player\_count == 3 and empty\_count == 1:

score = 100

elif player\_count == 2 and empty\_count == 2:

score = 10

elif opponent\_count == 3 and empty\_count == 1:

score = -950

elif opponent\_count == 2 and empty\_count == 2:

score = -100

Nếu:

+ player có 4 ô liên tiếp giống nhau thì score 1000 tương đương tỷ lệ thắng 100%

**+** player có 3 ô liên tiếp giống nhau và 1 ô trống thì score 100

+ player có 2 ô liên tiếp giống nhau và 2 ô trống thì score 10

+ opponent có 3 ô liên tiếp giống nhau và 1 ô trống thì score -950

+ opponent có 2 ô liên tiếp giống nhau và 2 ô trống thì score -100

**Yêu cầu** : Modify your minimax search with alpha-beta pruning to cut off search at a specified depth and use the heuristic evaluation function. Experiment with different cutoff values.

Hàm heuristic cho người chơi MAX:

def heuristic\_maximum(node, alpha\_value, beta\_value, depth, max\_depth):

# Nếu đạt độ sâu tối đa hoặc trò chơi đã kết thúc (win hoặc hòa)

if depth == max\_depth or terminal\_test(node.board, node.last\_row\_each\_col)[1]:

return None, heuristic(node) # Trả về điểm heuristic của trạng thái hiện tại

# Lấy danh sách các hành động hợp lệ (các cột còn trống)

next\_actions = actions(node.last\_row\_each\_col, False)

max\_value = -math.inf # Khởi tạo giá trị lớn nhất bằng âm vô cùng

player\_turn = player(node.board) # Lấy người chơi hiện tại (1 hoặc -1)

best\_action = next\_actions[0] if next\_actions else None # Chọn tạm hành động đầu tiên

# Duyệt qua tất cả các hành động có thể

for action in next\_actions:

# Sinh ra trạng thái mới sau khi thực hiện hành động

new\_board, new\_last\_each\_col = result(node.board, node.last\_row\_each\_col, action, player\_turn)

new\_node = Node(new\_board, new\_last\_each\_col, node) # Tạo node con từ trạng thái mới

# Gọi hàm heuristic\_minimum cho đối thủ (người chơi MIN)

\_, value = heuristic\_minimum(new\_node, alpha\_value, beta\_value, depth + 1, max\_depth)

node.children.append(new\_node) # Lưu node con vào danh sách con của node hiện tại

# Nếu giá trị mới tốt hơn (lớn hơn), cập nhật best\_action và max\_value

if value > max\_value:

max\_value = value

best\_action = action

# Cập nhật giá trị alpha (giá trị tốt nhất mà MAX có thể đảm bảo)

if max\_value > alpha\_value:

alpha\_value = max\_value

# Cắt tỉa (pruning): nếu alpha >= beta thì không cần xét tiếp

if alpha\_value >= beta\_value:

break

# Trả về hành động tốt nhất và giá trị đánh giá cao nhất

return best\_action, max\_value

Hàm heuristic cho người chơi MIN:

def heuristic\_minimum(node, alpha\_value, beta\_value, depth, max\_depth):

# Nếu đạt độ sâu tối đa hoặc trò chơi đã kết thúc

if depth == max\_depth or terminal\_test(node.board, node.last\_row\_each\_col)[1]:

return None, heuristic(node)

# Lấy danh sách các hành động hợp lệ

next\_actions = actions(node.last\_row\_each\_col, False)

min\_value = math.inf # Khởi tạo giá trị nhỏ nhất bằng dương vô cùng

player\_turn = player(node.board) # Lấy người chơi hiện tại

best\_action = next\_actions[0] if next\_actions else None # Chọn tạm hành động đầu tiên

# Duyệt qua tất cả các hành động có thể

for action in next\_actions:

# Sinh ra trạng thái mới sau hành động

new\_board, new\_last\_each\_col = result(node.board, node.last\_row\_each\_col, action, player\_turn)

new\_node = Node(new\_board, new\_last\_each\_col, node) # Tạo node con

# Gọi đệ quy cho người chơi MAX

\_, value = heuristic\_maximum(new\_node, alpha\_value, beta\_value, depth + 1, max\_depth)

node.children.append(new\_node)

# Nếu giá trị mới nhỏ hơn, cập nhật best\_action và min\_value

if value < min\_value:

min\_value = value

best\_action = action

# Cập nhật giá trị beta (giá trị tốt nhất mà MIN có thể đảm bảo)

if min\_value < beta\_value:

beta\_value = min\_value

# Cắt tỉa (pruning): nếu beta <= alpha thì dừng xét

if beta\_value <= alpha\_value:

break

# Trả về hành động tốt nhất và giá trị thấp nhất

return best\_action, min\_value

Hàm minimax alpha beta quyết định

def alpha\_beta\_decision(node, alpha\_value, beta\_value, depth, max\_depth=4):

# Kiểm tra người chơi hiện tại: nếu là MAX (1) thì gọi heuristic\_maximum

if player(node.board) == 1:

action, value = heuristic\_maximum(node, alpha\_value, beta\_value, depth, max\_depth)

# Nếu là MIN (-1) thì gọi heuristic\_minimum

else:

action, value = heuristic\_minimum(node, alpha\_value, beta\_value, depth, max\_depth)

# Trả về hành động được chọn và giá trị tương ứng

return action, value

**Yêu cầu:**

Experiment with the same manually created boards as above to check if the agent spots wining opportunities.

# Your code/ answer goes here.

board = [[0, 0, 0, 0, 0],

[0, 0, 0, 0, 0],

[-1, 1, 0, -1, -1],

[-1, -1, 1, 1, 1],

[1, -1, -1, 1, 1]]

board = np.array(board)

last\_row\_each\_col = np.sum(board == 0, axis=0) - 1

visualize(board)

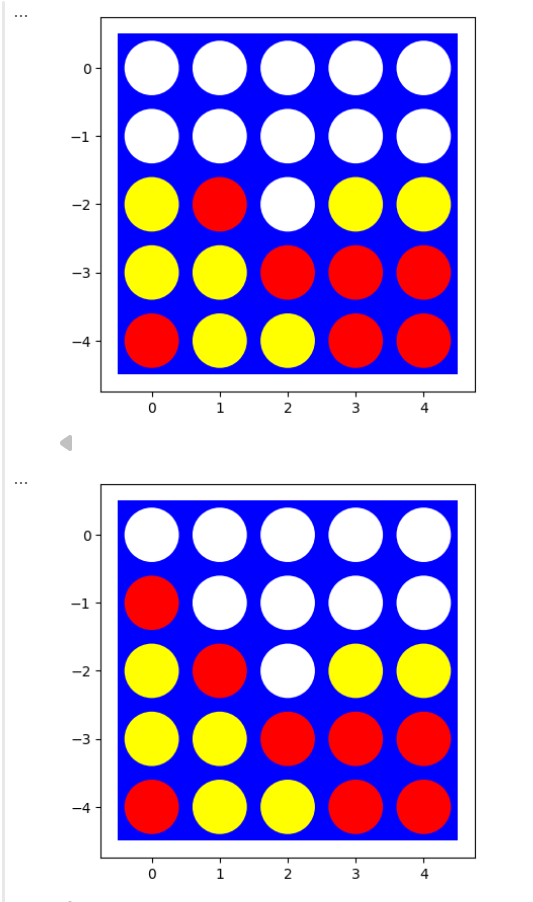
node = Node(board,last\_row\_each\_col)

action, \_ = alpha\_beta\_decision(node,-math.inf,+math.inf,0,6)

new\_board, new\_last\_row\_each\_col = result(board, last\_row\_each\_col, action, player(node.board))

visualize(new\_board)

Kết quả sau khi thực hiện đoạn code trên:



**Nhận xét :** agent có thể nhận ra được cơ hội chiến thắng

**Yêu cầu:How long does it take to make a move? Start with a smaller board with 4 columns and make the board larger by adding columns.**

**Code:**

board = [

[0, 0, 0, 0],

[-1, 1, 0, -1,],

[-1, -1, 1, 1,],

[1, -1, -1, 1,]]

board = np.array(board)

last\_row\_each\_col = np.sum(board == 0, axis=0) - 1

visualize(board)

node = Node(board,last\_row\_each\_col)

%time action, \_ = alpha\_beta\_decision(node,-math.inf,+math.inf,0,4)

new\_board, new\_last\_row\_each\_col = result(board, last\_row\_each\_col, action, player(node.board))

board = [[0, 0, 0, 0, 0],

[0, 0, 0, 0, 0],

[-1, 1, 0, -1, -1],

[-1, -1, 1, 1, 1]]

board = np.array(board)

last\_row\_each\_col = np.sum(board == 0, axis=0) - 1

visualize(board)

node = Node(board,last\_row\_each\_col)

%time action, \_ = alpha\_beta\_decision(node,-math.inf,+math.inf,0,4)

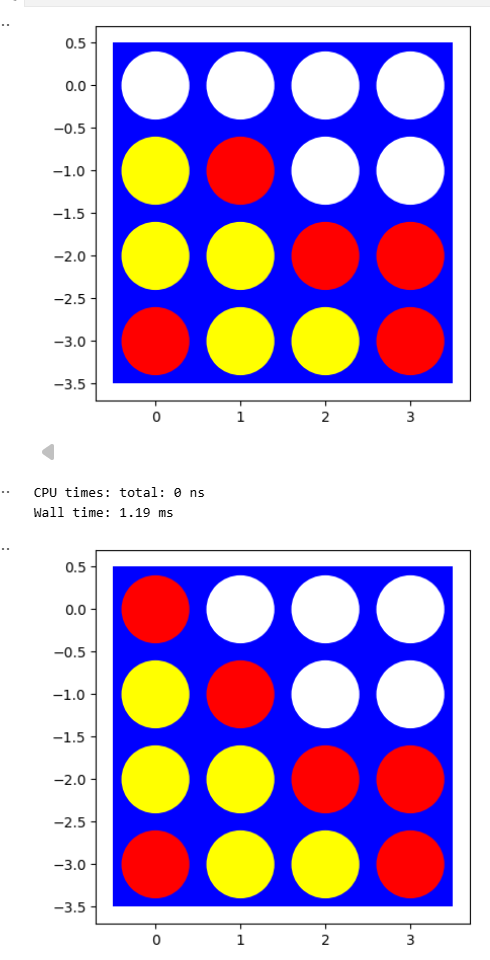
new\_board, new\_last\_row\_each\_col = result(board, last\_row\_each\_col, action, player(node.board))

Đoạn code trên mô phỏng 2 trạng thái bàn cờ với kích thước 4x4 và kích thước 4x5

Ta sẽ chạy đoạn code trên để kiểm tra agent mất bao lâu để quyết định nước đi tối ưu

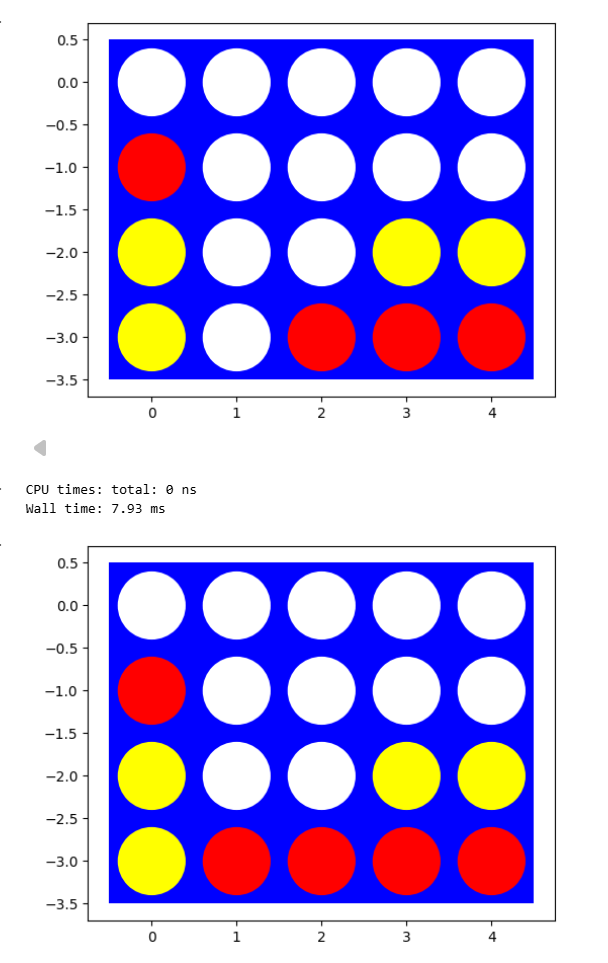
Agent đang xét sẽ là quân đỏ

**+Với bảng 1:**



Agent mất 1.19ms để tìm ra nước đi có thể chiến thắng

**+Với bảng 2**



Agent mất 7.93ms để tìm ra nước đi giúp chiến thắng

**Yêu cầu:Let two heuristic search agents (different cutoff depth) compete against each other on a reasonably sized board. Since there is no randomness, you only need to let them play once.**

**Code:**

# Your code/ answer goes here.

import math

import numpy as np

def play\_game():

board = empty\_board() # Khởi tạo bàn cờ trống

current\_agent = player(board) # Xác định người chơi đầu tiên (1 hoặc -1)

last\_row\_each\_col = np.sum(board == 0, axis=0) - 1 # Vị trí hàng trên cùng còn trống

node = Node(board, last\_row\_each\_col)

while True:

# Agent hiện tại chọn nước đi tốt nhất bằng alpha-beta

action, value = alpha\_beta\_decision(node, -math.inf, +math.inf, 0, 4)

# Thực hiện nước đi và tạo trạng thái mới

board, new\_last\_row\_each\_col = result(board, last\_row\_each\_col, action, current\_agent)

# Kiểm tra xem có kết thúc trận đấu không

winner, is\_terminal = terminal\_test(board, new\_last\_row\_each\_col)

if is\_terminal:

visualize(board)

if winner == 1:

return 1 # MAX thắng

elif winner == -1:

return -1 # MIN thắng

else:

return 0 # Hòa

# Cập nhật trạng thái cho lượt tiếp theo

last\_row\_each\_col = new\_last\_row\_each\_col

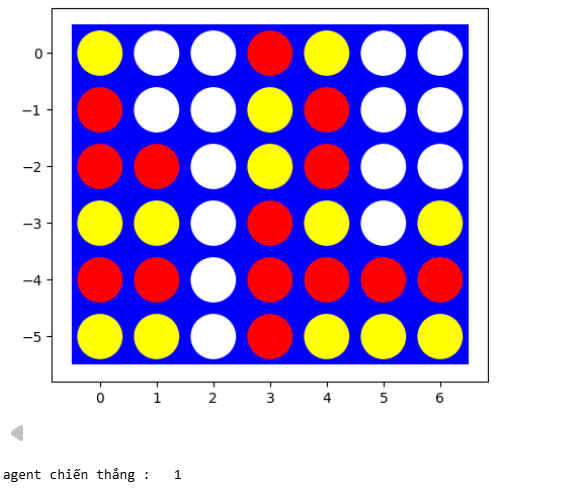
current\_agent = -current\_agent # Đổi lượt người chơi

node = Node(board, new\_last\_row\_each\_col) # Tạo node mới cho trạng thái hiện tại

print(play\_game())

Đoạn code trên mô tả 2 agent sử dụng alpha beta prunning sử dụng heuristic evaluate để đấu với nhau

**Kết quả sau khi chạy đoạn code trên**

****

**Nhận xét: quân đỏ dành chiến thắng**

# Mean Connect 4

Alpha-Beta Pruning Minimax

## Task 1: Defining the Search Problem

**Câu hỏi:**

**Define the components of the search problem associated with this game:**

**\* Initial state**

**\* Actions**

**\* Transition model**

**\* Test for the terminal state**

**\* Utility for terminal states**

1. Trạng thái ban đầu (Initial State): Là một bảng rỗng kích thước 6x7 (mặc định), tất cả các ô đều bằng 0

- Người chơi MAX được ký hiệu là 1

- Người chơi MIN được ký hiệu là -1

- Người chơi MAX luôn đi trước

2. Hành động (Actions):

Tại mỗi lượt người chơi có thể thực hiện 2 loại hành động hợp lệ:

- Nước đi bình thường (Drop move):

    - Chọn 1 cột chưa đầy

    - Đặt một quân của mình vào cột đó, quân sẽ rơi xuống ô trống thấp nhất trong cột

    - Ký hiệu hành động ("drop,c) với c là chỉ số cột

- Nước đi ác ý (mean move):

    - Chọn một quân của đối thủ nằm ở hàng dưới cùng của một cột bất kỳ

    - Lấy quân đó ra, các quân bên trễ sẽ rơi xuống một hàng.

    - Sau đó đặt chính quân đó (vẫn của đối thủ) vào bất kỳ cột nào còn chỗ trống

    - Ký hiệu hành động ("move", from\_col, to\_col)

3. Mô hình chuyển trạng thái (Transition Model):

Hàm chuyển trạng thái result(s,a) mô tả trạng thái mới sau khi áp dụng hành động a lên trạng thái s.

- Nếu a = ("drop", c) → chèn quân của người chơi hiện tại vào cột c (ở vị trí thấp nhất còn trống).

- Nếu a = ("move", from\_col, to\_col) →

    - Lấy quân của đối thủ ở hàng dưới cùng của cột from\_col.

    - Các quân bên trên rơi xuống một ô.

    - Đặt quân đó (vẫn thuộc về đối thủ) vào cột to\_col.

4. Kiểm tra trạng thái kết thúc (Terminal test)

Trạng thái là kết thúc khi:

- Một trong hai người chơi có 4 quân liên tiếp (theo hàng ngang, dọc, hoặc chéo), hoặc

- Bảng đã đầy (không còn nước đi hợp lệ).

5. Hàm lợi ích (Utility function)

Hàm utility(s) trả về giá trị từ góc nhìn của người chơi MAX (1):

- +1 nếu MAX thắng,

- −1 nếu MIN thắng,

- 0 nếu hòa.

Nếu dùng minimax với giá trị lớn, có thể thay ±1 bằng ±∞ cho các trạng thái thắng/thua.

**Câu hỏi: How big is the state space ? Give an estimate and explain it:**

Với trò Connect 4 thông thường, số trạng thái hợp lệ khoảng:

(theo nghiên cứu năm 1988).

Khi thêm luật “mean move” – cho phép di chuyển đĩa của đối thủ – số trạng thái khả dĩ tăng lên đáng kể, vì có thể tạo ra nhiều cấu hình khác nhau hơn.

-> Ước lượng hợp lý: Khoảng đến trạng thái

**Câu hỏi: How big is the game tree that minimax search will go through ? Give an estimate and explain it**

Ước lượng kích thước cây trò chơi: Với connect 4 chuẩn, hệ số phân nhánh trung bình b~7 (7 cột), độ sâu tối đa d<=42 ô. Vậy kích thước cây khoản .

Với luật “mean”, mỗi lượt có thể có đến khoảng 14 hành động (7 nước drop + 7 nước move).

Cây trò chơi quá lớn để duyệt toàn bộ, vì vậy cần dùng thuật toán minimax với cắt tỉa Alpha-Beta với giới hạn độ sâu để tìm kiếm hiệu quả.

## Task 2: Game environment and random agent

def actions(board, player):

    actions\_list = []

    rows, cols = board.shape

    # Drop moves

    for c in range(cols):

        if board[0, c] == 0:  # cột chưa đầy

            actions\_list.append(("drop", c))

    # Mean moves

    for from\_c in range(cols):

        bottom\_val = board[rows-1, from\_c]

        if bottom\_val == -player:  # quân đối thủ ở đáy

            for to\_c in range(cols):

                if board[0, to\_c] == 0:  # cột đích chưa đầy

                    actions\_list.append(("move", from\_c, to\_c))

    return actions\_list

def result(board, action, player):

    new\_board = board.copy()

    rows, cols = board.shape

    if action[0] == "drop":

        c = action[1]

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

            if new\_board[r, c] == 0:

                new\_board[r, c] = player

                break

    elif action[0] == "move":

        from\_c, to\_c = action[1], action[2]

        # Lấy quân đối thủ ở đáy

        opp\_piece = new\_board[rows-1, from\_c]

        if opp\_piece != -player:

            raise ValueError("Không thể di chuyển cột này.")

        # Dịch quân bên trên xuống

        for r in range(rows-1, 0, -1):

            new\_board[r, from\_c] = new\_board[r-1, from\_c]

        new\_board[0, from\_c] = 0

        # Đặt quân đối thủ vào cột đích

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

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

                new\_board[r, to\_c] = opp\_piece

                break

    return new\_board

def check\_four(board, player):

    rows, cols = board.shape

    # Kiểm tra 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

    # Kiểm tra dọc

    for c in range(cols):

        for r in range(rows - 3):

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

                return True

    # Kiểm tra chéo xuống

    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

    # Kiểm tra chéo lên

    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 terminal(board):

    if check\_four(board, 1) or check\_four(board, -1):

        return True

    if not any(board[0, c] == 0 for c in range(board.shape[1])):  # full board

        return True

    return False

def utility(board):

    if check\_four(board, 1):

        return 1

    elif check\_four(board, -1):

        return -1

    else:

        return 0

Helper Functions (Các hàm hỗ trợ)

actions(board, player): Tìm và liệt kê *tất cả* các nước đi hợp lệ mà người chơi hiện tại có thể thực hiện. Danh sách này bao gồm cả nước đi "thả quân" (drop) và nước đi "di chuyển quân" (mean move).

result(board, action, player): Trả về một bàn cờ *mới* (dưới dạng bản sao) sau khi mô phỏng việc thực hiện một hành động (action) cụ thể.

check\_four(board, player): Kiểm tra xem người chơi (player) đã thắng hay chưa, bằng cách tìm 4 quân cờ liên tiếp theo hàng ngang, hàng dọc, hoặc đường chéo.

terminal(board): Kiểm tra xem trò chơi đã *kết thúc* hay chưa. Trò chơi kết thúc nếu có người thắng (dùng check\_four) hoặc nếu bảng đã đầy (hòa).

utility(board): Tính toán giá trị (điểm số) của một ván cờ *đã kết thúc*. Nó trả về +1 (nếu MAX thắng), -1 (nếu MIN thắng), hoặc 0 (nếu hòa).

## Task 3

import math

def maximum(board, player, alpha, beta, move\_ordering=False, depth = 3):

    if terminal(board=board) or depth == 0:

        return utility(board), None

    v0, move = -math.inf, None

    for action in actions(board, player, move\_ordering):

        v, \_= minimum(result(board, action, player), -player, alpha, beta, move\_ordering, depth-1)

        if v > v0:

            v0 = v

            move = action

            alpha = max(alpha, v0)

        if v >= beta:

            break

    return v0, move

Hàm maximum là một phần của thuật toán Alpha-Beta Minimax và hàm này sẽ tìm nước đi tốt nhất cho người chơi hiện tại (Max player) bằng cách giả định rằng đối thủ sẽ luôn chơi tối ưu. Hàm này nhận vào 7 tham số là boar (ma trận trạng thái hiện tại của trò chơi), player (người chơi hiện tại với 1 là nguời chơi Max, -1 là người chơi Min), alpha (giá trị tốt nhất hiện có của Max trong nhánh hiện tại), beta (giá trị tốt nhất hiện có của Min trong nhánh hiện tại), move\_ordering (tùy chọn sắp xếp nước đi để cắt tỉa nhanh hơn), depth (độ sâu còn lại của cây tìm kiếm). Ở dòng if đầu tiên, kiểm tra xem trò chơi đã kết thúc chưa (terminal(board=board)), hoặc đạt đến cutoff depth (depth == 0). Tiếp theo, ở mỗi nút, duyệt qua tất cả các nước đi hợp lệ, đệ quy gọi đến hàm minimum

(đại diện cho đối thủ), và cập nhật giá trị tốt nhất cũng như alpha. Nếu có v >= beta, có nghĩa là đối thủ Min sẽ không cho phép nhánh này xảy ra (vì Min đã có lựa chọn tốt hơn ở nhánh khác). Do đó, dừng duyệt (cắt tỉa) phần còn lại cả các nước đi hợp lệ. Cuối cùng, trả về giá trị cũng như hành động tối ưu nhất cho người chơi Max ở trạng thái trò chơi hiện tại.

def minimum(board, player, alpha, beta, move\_ordering=False, depth = 3):

    if terminal(board=board) or depth == 0:

        return utility(board), None

    v0, move = math.inf, None

    for action in actions(board, player, move\_ordering):

        v, \_ = maximum(result(board, action, player), -player, alpha, beta, move\_ordering, depth-1)

        if v < v0:

            v0 = v

            move = action

            beta = min(beta, v0)

        if v <= alpha:

            break

    return v0, move

Hàm minimum nhận các tham số và có cách hoạt động tương tự như hàm maximum.

def alpha\_beta\_search(board, player = 1):

    if player == 1:

        value, move = maximum(board, player, -math.inf, +math.inf)

    else:

        value, move = minimum(board, player, -math.inf, +math.inf)

    return value, move

Hàm alpha\_beta\_search là điểm bắt đầu của thuật toán Alpha-Beta Pruining Minimax. Đầu tiên, hàm sẽ xác định người chơi hiện tại là Max hay Min để lựa chọn chiến lược tìm kiếm phù hợp. Với giá trị khởi tạo ban đầu của alpha là -∞ và beta là +∞, hàm sẽ tiến hành mở rộng cây tìm kiếm bằng cách gọi đệ quy tới hàm maximum hoặc minimum. Cuối cùng, giá trị mà hàm alpha\_beta\_search trả về là giá trị tốt nhất và nước đi tương ứng, cho phép agent chọn được hành động tối ưu trong trạng thái trò chơi hiện tại.

def play\_game\_minimax(board, alpha, beta):

    board = board

    player = 1

    while not terminal(board):

        if player == 1:

            \_, action = maximum(board, player, alpha, beta, True)

        else:

            \_, action = minimum(board, player, alpha, beta, True)

        board = result(board, action, player)

        player \*= -1

    return utility(board)

Hàm play\_game\_minimax mô phỏng toàn bộ quá trình chơi Mean Connect 4 giữa hai người chơi. Tại mỗi lượt chơi, người chơi hiện tại sẽ gọi hàm maximum() hoặc minimum() để lựa chọn hành động tối ưu, sau đó cập nhật lại trạng thái hiện tại của trò chơi thông qua result(board, action, player). Quá trình này lặp lại cho đến khi terminal(board) phát hiện ra trạng thái cuối cùng. Cuối cùng, hàm sẽ trả về kết quả của trận đấu (thắng, thua hoặc hòa).

## Task 4

def heuristic(state, player = 1, more\_weight=False):

    u = utility(state)

    if u != 0:

        return u, True

    score = 0

    rows, cols = state.shape

    opponent = -player

    center\_col = cols // 2

    center\_arr = state[:, center\_col]

    score += 0.1\*np.sum(center\_arr == player)

    if more\_weight:

        score \*= 2

    # Duyệt dọc

    for c in range(cols):

        for r in range(rows-3):

            window = state[r:r+4, c]

            score += evaluate\_window(window, player, opponent)

    # Duyệt ngang

    for r in range(rows):

        for c in range(cols - 3):

            window = state[r, c:c+4]

            score += evaluate\_window(window, player, opponent)

    # # Duyệt chéo phải

    for r in range(rows - 3):

        for c in range(cols - 3):

            window = np.array([state[r+i, c+i] for i in range(4)])

            score += evaluate\_window(window, player, opponent)

    # Duyệt chéo trái

    for r in range(rows - 3):

        for c in range(3, cols):

            window = np.array([state[r+i, c-i] for i in range(4)])

            score += evaluate\_window(window, player, opponent)

    score = max(-1.0, min(1.0, score))

    return score, False

Hàm heuristic đánh giá trạng thái của trạng thái trò chơi hiện tại, giúp ước lượng giá trị bàn cờ khi chưa đạt đến giá trị kết thúc. Bắt đầu, thuật toán sẽ kiểm tra cột ở giữa xem có bao nhiêu ô thuộc người chơi hiện tại và tính điểm cho số ô đó (vì ô ở giữa dễ tạo ra các đường mới hơn). Sau đó, thuật toán sẽ duyệt 4 ô liên tiếp (hàng, cột, chéo chính, chéo phụ) và tính điểm cho các cấu trúc có lợi với người chơi hiện tại và trừ điểm cho các tình huống nguy hiểm.

def evaluate\_window(window, player, opponent):

    score = 0

    play\_count = np.sum(window == player)

    opp\_count = np.sum(window == opponent)

    empty\_count = np.sum(window == 0)

    if play\_count == 4:

        score += 1.0

    elif play\_count == 3 and empty\_count == 1:

        score += 0.1

    elif play\_count == 2 and empty\_count == 2:

        score += 0.01

    elif opp\_count == 3 and empty\_count == 1:

        score -= 0.12

    elif opp\_count == 2 and empty\_count == 2:

        score -= 0.015

    return score

Đây là hàm được hàm heuristic gọi để đánh giá chi tiết từng nhóm 4 ô liên tiếp. Bằng cách gán các trọng số khác nhau cho các window khác nhau (2 quân, 3 quân, 4 quân,…). Từ đó, hàm này có thể mô phỏng khả năng chiến năng hoặc nguy cơ bị chặn.

def heuristic\_maximum(board, player, alpha, beta, move\_ordering=False, depth = 4, more\_weight=False):

    v, terminal\_state = heuristic(board, player, more\_weight)

    if terminal\_state or depth == 0:

        if terminal\_state:

            alpha, beta = v, v

        return v, None

    v, move = -math.inf, None

    for action in actions(board, player, move\_ordering):

        value, \_ = heuristic\_minimum(result(board, action, player), -player, alpha, beta, move\_ordering, depth-1, more\_weight)

        if value > v:

            v = value

            move = action

            alpha = max(alpha, v)

        if v >= beta:

            return v, move

    return v, move

Hàm heuristic\_maximum() là hàm cải tiến của hàm maximum() đã đề cập ở trước. Tương tự như hàm maximum(), hàm heuristic\_maximum() cũng nhận các tham số như board, player, alpha , beta, move\_ordering, depth nhưng có khác nhau ở một điểm là hàm heuristic\_maximum() nhận thêm 1 tham số là more\_weight để tăng trọng số ở hàm đánh giá heuristic(). Điểm khác biệt thứ hai, thay vì trả về utility(board) khi đến trạng thái cuối cùng terminal\_state hoặc đạt đến cutoff depth (depth == 0) như hàm maximum, hàm heuristic\_maximum() sẽ trả về giá trị mà hàm heuristic()đánh giá được. Phần còn lại của hàm heuristic\_maximum() hoạt động tương tự như hàm maximum().

def heuristic\_minimum(board, player, alpha, beta, move\_ordering=False, depth = 4, more\_weight=False):

    v, terminal\_state = heuristic(board, player, more\_weight)

    if terminal\_state or depth == 0:

        if terminal\_state:

            alpha, beta = v, v

        return v, None

    v, move = +math.inf, None

    for action in actions(board, player, move\_ordering):

        value, \_ = heuristic\_maximum(result(board, action, player), -player, alpha, beta, move\_ordering, depth-1, more\_weight)

        if value < v:

            v = value

            move = action

            beta = min(beta, v)

        if v <= alpha:

            return v, move

    return v, move

Hàm heuristic\_minimum() có cách hoạt động tương tự như hàm heuristic\_maximum().

def heuristic\_alpha\_beta\_search(board, player=1, move\_ordering= False, depth=4, more\_weight=False):

    if player == 1:

        value, move = heuristic\_maximum(board, player, -math.inf, +math.inf, move\_ordering, depth, more\_weight)

    else:

        value, move = heuristic\_minimum(board, player, -math.inf, +math.inf, move\_ordering, depth, more\_weight)

    return value, move

def play\_game\_heuristic\_minimax(board, alpha, beta, move\_ordering=False, depth=4):

    board = board

    player = 1

    while not terminal(board):

        if player == 1:

            action, \_ = heuristic\_alpha\_beta\_search(board, player, alpha, beta, move\_ordering, depth)

        else:

            action, \_ = heuristic\_alpha\_beta\_search(board, player, alpha, beta, move\_ordering, depth)

        board = result(board, action, player)

        player \*= -1

    return utility(board)

Tương tự như 2 hàm alpha\_beta\_search() và play\_game\_minimax(), nhưng thay vì sử dụng hàm maximum() và minimum(), hàm heuristic\_alpha\_beta\_search() và play\_game\_heuristic\_minimax() sẽ sử dụng hàm heuristic\_maximum() và heuristic\_minimum() để thay thế.

**Dots and boxes**

## Task 1: Defining the Search Problem [10 point]

1)Define the components of the search problem associated with this game:

Initial state

* **Mô tả:**
  + Bảng trống, **chưa có cạnh nào được vẽ**.
  + Điểm của cả hai người chơi đều bằng **0**.
  + **Lượt đi đầu tiên** thuộc về người chơi MAX (hoặc Player 1).

**Ví dụ:** Với bảng 2×2 (12 cạnh), trạng thái ban đầu = danh sách 12 cạnh trống, điểm (0,0), lượt: MAX.

Actions:

* Mỗi **hành động** là **vẽ một cạnh** (đường nối giữa hai chấm chưa được nối).
* **Tập hành động khả thi** ở trạng thái hiện tại là **tất cả các cạnh chưa được vẽ**.

**Ví dụ:** Nếu 12 cạnh ban đầu, đã vẽ 3 cạnh → còn 9 hành động hợp lệ.

Transition model:

Khi người chơi chọn một hành động (vẽ cạnh):

1. Trạng thái mới được tạo ra bằng cách **đánh dấu cạnh đó là đã vẽ**.
2. Nếu hành động **hoàn thành một hoặc nhiều ô vuông (boxes)**:
   * Người chơi **được cộng điểm** cho mỗi ô hoàn thành.
   * Người đó **được đi tiếp** (không đổi lượt).
3. Nếu không hoàn thành ô nào → **chuyển lượt cho đối thủ**.

Test for the terminal state

Utility for terminal states

Code:

import random

def initial\_state(n=3, m=3):

"""

Mỗi ô: [[edges], edge\_count, owner]

edges: [top, bottom, left, right], 0 = chưa đánh, -1 = đã đánh

owner: 1 (Player 1), -1 (Player 2), 0 (chưa ai)

"""

return [[[ [0, 0, 0, 0], 0, 0 ] for \_ in range(m-1)] for \_ in range(n-1)]

def actions(state, posx, posy, tag, player):

"""

posx,posy: chỉ số ô (0..R-1, 0..C-1)

tag: 0=top, 1=bottom, 2=left, 3=right

player: 1 hoặc -1 (chúng ta lưu cạnh = -1 khi vẽ)

"""

a, b = len(state), len(state[0])

# nếu cạnh đã được vẽ (khác 0) thì bỏ qua

if state[posx][posy][0][tag] != 0:

return

# đánh dấu cạnh = -1 (chỉ dùng -1 để biểu diễn "đã vẽ")

state[posx][posy][0][tag] = -1

# cập nhật ô liền kề (nếu có) và tính lại edge\_count cho cả hai ô liên quan

if tag == 0 and posx > 0: # top -> ảnh hưởng bottom của ô trên

state[posx-1][posy][0][1] = -1

elif tag == 1 and posx < a-1: # bottom -> ảnh hưởng top của ô dưới

state[posx+1][posy][0][0] = -1

elif tag == 2 and posy > 0: # left -> ảnh hưởng right của ô trái

state[posx][posy-1][0][3] = -1

elif tag == 3 and posy < b-1: # right -> ảnh hưởng left của ô phải

state[posx][posy+1][0][2] = -1

# cập nhật count cho ô hiện tại

state[posx][posy][1] = sum(1 for e in state[posx][posy][0] if e != 0)

# cập nhật count cho ô liền kề (nếu có)

if tag == 0 and posx > 0:

state[posx-1][posy][1] = sum(1 for e in state[posx-1][posy][0] if e != 0)

elif tag == 1 and posx < a-1:

state[posx+1][posy][1] = sum(1 for e in state[posx+1][posy][0] if e != 0)

elif tag == 2 and posy > 0:

state[posx][posy-1][1] = sum(1 for e in state[posx][posy-1][0] if e != 0)

elif tag == 3 and posy < b-1:

state[posx][posy+1][1] = sum(1 for e in state[posx][posy+1][0] if e != 0)

def terminal\_state(state):

"""Trả về 0 nếu chưa kết thúc, 1/-1 nếu thắng, 2 nếu hòa."""

a, b = len(state), len(state[0])

p1, p2, total = 0, 0, a\*b

for i in range(a):

for j in range(b):

if state[i][j][2] == 1:

p1 += 1

elif state[i][j][2] == -1:

p2 += 1

# kiểm tra còn cạnh trống không

any\_open = any(0 in state[i][j][0] for i in range(a) for j in range(b))

if any\_open:

return 0

# không còn cạnh → kết thúc

if p1 > p2:

return 1

elif p2 > p1:

return -1

else:

return 2

def utility(state):

res = terminal\_state(state)

if res == 1:

return "Win"

elif res == -1:

return "Lose"

elif res == 2:

return "Draw"

else:

return None

def transition\_model(action, search, utility, state, player):

"""Thực hiện 1 lượt chơi."""

result = utility(state)

if result:

return result

choice = search(state)

if not choice:

return state # không còn nước đi

i, j, k = choice

action(state, i, j, k, player)

# cập nhật ô nào hoàn tất

for x in range(len(state)):

for y in range(len(state[0])):

if state[x][y][1] == 4 and state[x][y][2] == 0:

state[x][y][2] = player

return state

2) How big is the state space? Give an estimate and explain it.

Mỗi cạnh có 2 trạng thái (vẽ hoặc chưa).

Với bảng G×G boxes:

E=2G(G+1)⇒Số trạng thái xấp xỉ 2^E

Ví dụ:

* 2×2 → E=12⇒2^12 = 4096
* 4×4 → E=40⇒2^40 ≈10^12

3) How big is the game tree that minimax search will go through? Give an estimate and explain it.

Mỗi ván là một thứ tự sắp xếp các cạnh → E! khả năng. Vì trò chơi thực chất là chọn một ordering của các cạnh (mỗi lần chọn 1 cạnh chưa được vẽ).

Kích thước cây xấp xỉ E!

## Task 2: Game Environment and Random Agent [30 point]

You need to think about a data structure to represent the board meaning he placed lines and who finished what box. There are many options. Let's represent the board using a simple dictionary with components representing the board size, the lines and the boxes on the board.

Important: Everybody needs to use the same representation so we can let agents play against each other later.

Write code to display the board. **Bonus point: Post your visualization code with an example output to the discussion board.** The best visualization will earn you bonus participation points in this class.

Code:

def get\_board(state):

"""

In board dạng ASCII:

- state: R x C, mỗi ô = [edges, edge\_count, owner]

- edges: [top, bottom, left, right] (0 not drawn, -1 drawn)

"""

R = len(state) # number of box rows

C = len(state[0]) # number of box cols

# In từng hàng dot (total dot rows = R+1)

for dr in range(R + 1):

# 1) in: dot and horizontal edges for this dot-row

line = ""

for dc in range(C):

line += "o"

# horizontal edge between dot (dr,dc) and (dr, dc+1)

drawn = False

# if box above exists, its bottom corresponds to this dot-row

if dr - 1 >= 0:

if state[dr-1][dc][0][1] != 0:

drawn = True

# if box below exists, its top corresponds to this dot-row

if dr < R:

if state[dr][dc][0][0] != 0:

drawn = True

if drawn:

line += "──" # horizontal drawn

else:

line += " " # empty space

line += "o"

print(line)

# 2) in vertical edges + (optionally) owner placeholders for box row dr (only if dr < R)

if dr < R:

line2 = ""

for dc in range(C + 1):

# vertical edge between dot (dr,dc) and (dr+1,dc)

drawn = False

# box to left exists -> its right edge corresponds

if dc - 1 >= 0:

if state[dr][dc-1][0][3] != 0:

drawn = True

# box to right exists -> its left edge corresponds

if dc < C:

if state[dr][dc][0][2] != 0:

drawn = True

if drawn:

line2 += "|"

else:

line2 += " "

# after vertical char, add two spaces except at final column (we will print small holder)

if dc < C:

# you can show owner or empty inside box space

owner = state[dr][dc][2]

if owner == 1:

line2 += "1 "

elif owner == -1:

line2 += "2 "

else:

line2 += " "

print(line2)

print() # blank line after board

RESULTS (simulation only):

o o o

|

o o──o

|1 |

o o──o

Giải thích:

Hàm được khởi tạo nhằm mục đích in ra bảng dots and boxes, trong đó các điểm sẽ được kí hiệu “o” và line sẽ là - hoặc | cho đường thẳng ngang và đường thẳng dọc, đồng thời sau khi một player hoàn thiện mặt thứ tư của box thì số của user đó sẽ được đánh vào ô.

**Implement helper functions for:**

* The transition model result(s,a) .
* The utility function utility(s) .
* Check for terminal states terminal(s) .
* A check for available actions in each state actions(s) .

Notes:

* Make sure that all these functions work with boards of different sizes (number of columns and rows as stored in the board).
* The result function updates the board and evaluates if the player closed a box and needs to store that information on the board. Add elements of the form (row,col): player to the board dictionary. row and col are the coordinates for the box and the player is +1 or -1 representing the player. For example (0,0): -1 means that the top-left box belongs to the other player.
* Important: Remember that a player goes again after she completes a box

Code:

def Environment(player1, player2, search, times=100, n=3, m=3):

board = initial\_state(n, m)

turn = 0

result = None

print("Init")

get\_board(board)

while result is None and turn < times:

if turn % 2 == 0:

print(f"\n--- Turn {turn+1} (Player 1) ---")

board = transition\_model(actions, search, utility, board, 1)

else:

print(f"\n--- Turn {turn+1} (Player 2) ---")

board = transition\_model(actions, search, utility, board, -1)

get\_board(board)

result = utility(board)

turn += 1

if not any(0 in cell[0] for row in board for cell in row):

result = utility(board)

break

print("\n=== Game Over ===")

print("Result:", result)

return result

Giải thích:

Hàm Environment chứa một số biến đầu vào như sau:

* Player1, player2: người chơi (Mỗi người chơi sẽ được )

Implement an agent that plays randomly. Make sure the agent function receives as the percept the board and returns a valid action. Use an agent function definition with the following signature (arguments):

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

The argument player is used for agents that do not store what side they are playing. The value passed on by the environment should be 1 ot -1 for playerred and yellow, respectively. See Experiments section for tic-tac-toe for an example.

Code:

import numpy as np

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

a, b = len(state), len(state[0])

available = []

for i in range(a):

for j in range(b):

for k in range(4):

if state[i][j][0][k] == 0:

available.append((i, j, k))

return random.choice(available) if available else None

Let two random agents play against each other **1000 times**. Look at the Experiments section for tic-tac-toe to see how the environment uses the agent functions to play against each other.

How often does each player win? Is the result expected?

RESULTS:

Init

o o o

o o o

o o o

--- Turn 1 (Player 1) ---

o o o

o o o

|

o o o

--- Turn 2 (Player 2) ---

o o o

o o o

|

o o──o

--- Turn 3 (Player 1) ---

o o o

|

o o o

|

o o──o

--- Turn 4 (Player 2) ---

o o o

|

o o o

| |

o o──o

--- Turn 5 (Player 1) ---

o o o

| |

o o o

| |

o o──o

--- Turn 6 (Player 2) ---

o o o

| |

o──o o

| |

o o──o

--- Turn 7 (Player 1) ---

o o o

| |

o──o──o

|1 |

o o──o

--- Turn 8 (Player 1) ---

o o──o

| |

o──o──o

|1 |

o o──o

--- Turn 9 (Player 2) ---

o o──o

| |

o──o──o

|1 |

o──o──o

--- Turn 10 (Player 1) ---

o──o──o

|1 |

o──o──o

|1 |

o──o──o

--- Turn 11 (Player 1) ---

o──o──o

|1 |

o──o──o

|1 |1 |

o──o──o

--- Turn 12 (Player 1) ---

o──o──o

|1 |1 |

o──o──o

|1 |1 |

o──o──o

=== Game Over ===

Result: Win

Win

## Task 3: Minimax Search with Alpha-Beta Pruning [30 points]

Implement the search starting.

Implement the search starting from a given board and specifying the player and put it into an agent function. You can use code from the tic-tac-toe example.

Notes:

Make sure that all your agent functions have a signature consistent with the random agent above.

The search space for larger board may be too large. You can experiment with smaller boards.

Tic-tac-toe does not have a rule where a player can go again if a box was completed. You need to adapt the tree search to reflect that rule.

Code:

import copy, math, random

def legal\_moves(state):

moves = []

for i in range(len(state)):

for j in range(len(state[0])):

for k in range(4):

if state[i][j][0][k] == 0:

moves.append((i, j, k))

return moves

def apply\_action(state, i, j, k, player):

a, b = len(state), len(state[0])

if state[i][j][0][k] != 0:

return 0 # cạnh đã đánh rồi

state[i][j][0][k] = -1

# cập nhật cạnh liền kề

if k == 0 and i > 0: state[i-1][j][0][1] = -1

elif k == 1 and i < a-1: state[i+1][j][0][0] = -1

elif k == 2 and j > 0: state[i][j-1][0][3] = -1

elif k == 3 and j < b-1: state[i][j+1][0][2] = -1

# cập nhật số cạnh và kiểm tra box hoàn thành

completed = 0

for x in range(a):

for y in range(b):

state[x][y][1] = sum(1 for e in state[x][y][0] if e != 0)

if state[x][y][1] == 4 and state[x][y][2] == 0:

state[x][y][2] = player

completed += 1

return completed

def simulate\_move(state, move, player):

s2 = copy.deepcopy(state)

completed = apply\_action(s2, \*move, player)

extra\_turn = (completed > 0)

return s2, extra\_turn

def evaluate(state, player):

"""Hàm đánh giá: điểm số dựa trên số box sở hữu."""

score = sum(state[i][j][2] for i in range(len(state)) for j in range(len(state[0])))

return score \* player

def is\_terminal(state):

"""Kiểm tra xem bàn cờ đã đầy chưa."""

for i in range(len(state)):

for j in range(len(state[0])):

for e in state[i][j][0]:

if e == 0:

return False

return True

def minimax(state, turn, me, alpha, beta, depth, max\_depth):

"""Thuật toán Minimax có Alpha–Beta pruning."""

if is\_terminal(state) or depth >= max\_depth:

return evaluate(state, me), None

moves = legal\_moves(state)

if not moves:

return evaluate(state, me), None

best\_move = None

if turn == me: # maximize

val = -math.inf

for mv in moves:

ns, extra = simulate\_move(state, mv, turn)

nxt = turn if extra else -turn

score, \_ = minimax(ns, nxt, me, alpha, beta, depth + 1, max\_depth)

if score > val:

val, best\_move = score, mv

alpha = max(alpha, val)

if alpha >= beta:

break

return val, best\_move

else: # minimize

val = math.inf

for mv in moves:

ns, extra = simulate\_move(state, mv, turn)

nxt = turn if extra else -turn

score, \_ = minimax(ns, nxt, me, alpha, beta, depth + 1, max\_depth)

if score < val:

val, best\_move = score, mv

beta = min(beta, val)

if alpha >= beta:

break

return val, best\_move

def minimax\_agent(board, player, max\_depth=3):

"""Agent dùng Minimax + Alpha–Beta pruning."""

moves = legal\_moves(board)

if not moves:

return board

\_, mv = minimax(board, player, player, -math.inf, math.inf, 0, max\_depth)

if not mv:

mv = random.choice(moves)

apply\_action(board, \*mv, player)

return board

Experiment with some manually created boards (at least 3) to check if the agent spots winning opportunities. Discuss the results.

# 1

# o──o o

# |

# o──o o

# o o o

# o──o o

# |1 |

# o──o o

# o o o

# 2

# o──o──o

# | |

# o o──o

# o o o

# o──o──o

# | |1 |

# o o──o

# o o o

# 3

# o──o──o

# o o o

# o o o

# o──o──o

# |

# o o o

# o o o

Discussion: Minimax search chooses the max score for the player and the min score for the opponent

Giải thích:

Triển khai minimax với alpha beta pruning cho phép giới hạn cây đồ thị trong một nhánh có khả năng cao nhất với độ sâu của cây được cho trước nhằm mục đích giới hạn sự mở rộng vô hạn của cây. Ngoài ra phiên bản của em đã triển khai chi tiết thêm từng hàm apply\_action và các hàm custom cho phù hợp với phiên bản minimax kết hợp alpha beta pruning

**Move ordering**

Starting the search with better moves will increase the efficiency of alpha-beta pruning. Describe and implement a simple move ordering strategy.

Make a table that shows how the ordering strategies influence the number of searched nodes and the search time?

Code:

import random, time, copy

def get\_possible\_moves(state):

moves = []

for i in range(len(state)):

for j in range(len(state[0])):

for k in range(4):

if state[i][j][0][k] == 0:

moves.append((i, j, k))

return moves

def evaluate\_move(state, move):

i, j, k = move

temp = state[i][j][0]

temp[k] = -1

score = sum(1 for x in temp if x != 0)

return score # càng cao càng tốt

def order\_moves(state, moves):

return sorted(moves, key=lambda mv: evaluate\_move(state, mv), reverse=True)

def copy\_state(state):

return copy.deepcopy(state)

def alpha\_beta\_search(state, depth, alpha, beta, maximizing\_player):

result = terminal\_state(state)

if depth == 0 or result != 0:

if result == 1: return 10, None

elif result == -1: return -10, None

elif result == 2: return 0, None

return 0, None

best\_move = None

moves = order\_moves(state, get\_possible\_moves(state))

if maximizing\_player:

max\_eval = -float('inf')

for mv in moves:

new\_state = copy\_state(state)

actions(new\_state, \*mv, 1)

eval, \_ = alpha\_beta\_search(new\_state, depth - 1, alpha, beta, False)

if eval > max\_eval:

max\_eval = eval

best\_move = mv

alpha = max(alpha, eval)

if beta <= alpha:

break

return max\_eval, best\_move

else:

min\_eval = float('inf')

for mv in moves:

new\_state = copy\_state(state)

actions(new\_state, \*mv, -1)

eval, \_ = alpha\_beta\_search(new\_state, depth - 1, alpha, beta, True)

if eval < min\_eval:

min\_eval = eval

best\_move = mv

beta = min(beta, eval)

if beta <= alpha:

break

return min\_eval, best\_move

s = initial\_state(3, 3)

moves = get\_possible\_moves(s)

start = time.time()

\_ = alpha\_beta\_search(s, 3, -float('inf'), float('inf'), True)

unordered\_time = time.time() - start

start = time.time()

\_ = alpha\_beta\_search(s, 3, -float('inf'), float('inf'), True)

ordered\_time = time.time() - start

print(f"Unordered Search Time: {unordered\_time:.4f}s")

print(f"Ordered Search Time: {ordered\_time:.4f}s")

RESULT:

Unordered Search Time: 0.0009s

Ordered Search Time: 0.0001s

Giải thích:

Bằng cách sử dụng best move (áp dụng một hàm đánh giá các nước đi và đánh giá chúng thông qua evaluation), best move cho phép sắp xếp các nhánh có kết quả cao lên đầu tiên, sau đó kết hợp alpha beta pruning cho phép đồ thị được cắt và bỏ qua nhiều node trong cây hơn.

**The first few moves**

Start with an empty board. This is the worst case scenario for minimax search with alpha-beta pruning since it needs to solve all possible games that can be played (minus some pruning) before making the decision. What can you do?

Code:

def first\_move\_agent(state):

moves = get\_possible\_moves(state)

best = min(moves, key=lambda mv: sum(state[mv[0]][mv[1]][0]))

return best

Giải thích:

Đây là trường hợp xấu nhất khi cây đang ở trạng thái đầu của trò chơi, khi đó có quá nhiều nước đi cần phải được tính toán. Lúc này thực hiện first few move và tính toán trong đó sẽ giúp giới hạn được số lượng node cần được tính toán.

**Playtime**

Let the Minimax Search agent play a random agent on a small board. Analyze wins, losses and draws.

def random\_agent(state):

moves = get\_possible\_moves(state)

return random.choice(moves)

def play\_game(agent1, agent2, n=3, m=3):

state = initial\_state(n, m)

player = 1

while terminal\_state(state) == 0:

agent = agent1 if player == 1 else agent2

\_, mv = agent(state)

actions(state, \*mv, player)

player \*= -1

res = terminal\_state(state)

return res

# So sánh 5 ván

results = []

for \_ in range(5):

res = play\_game(lambda s: (alpha\_beta\_search(s, 3, -float('inf'), float('inf'), True),)[0], random\_agent)

results.append(res)

print(results)

RESULT:

[2, 2, 2, 2, 2]

## Task 4: Heuristic Alpha-Beta Tree Search [30 points]

Heuristic evaluation function

Define and implement a heuristic evaluation function.

Code:

def heuristic\_evaluation(state):

player1\_score = 0

player2\_score = 0

near\_box\_1 = 0

near\_box\_2 = 0

for i in range(len(state)):

for j in range(len(state[0])):

edges = state[i][j][0]

filled = sum(1 for e in edges if e != 0)

owner = state[i][j][1]

if owner == 1:

player1\_score += 1

elif owner == -1:

player2\_score += 1

elif filled == 3: # ô gần hoàn thành

near\_box\_1 += 1 if 1 in edges else 0

near\_box\_2 += 1 if -1 in edges else 0

# Heuristic tổng hợp

return (player1\_score - player2\_score) \* 10 + (near\_box\_1 - near\_box\_2) \* 2

Giải thích:

Hàm heuristic được sử dụng để đánh giá kết quả cũng như tìm kiếm các kết quả tối ưu cho cây trong đó ưu tiên các ô đã được kẻ 3 đường để hoàn thiện ô đó.

Cutting off search

Modify your Minimax Search with Alpha-Beta Pruning to cut off search at a specified depth and use the heuristic evaluation function. Experiment with different cutoff values.

Code:

def alpha\_beta\_cutoff(state, depth, alpha, beta, maximizing\_player):

result = terminal\_state(state)

if result != 0:

if result == 1: return 1000, None

elif result == -1: return -1000, None

else: return 0, None

if depth == 0: # cutoff

return heuristic\_evaluation(state), None

best\_move = None

moves = get\_possible\_moves(state)

moves = order\_moves(state, moves)

if maximizing\_player:

max\_eval = -float('inf')

for mv in moves:

new\_state = copy\_state(state)

actions(new\_state, \*mv, 1)

eval, \_ = alpha\_beta\_cutoff(new\_state, depth - 1, alpha, beta, False)

if eval > max\_eval:

max\_eval = eval

best\_move = mv

alpha = max(alpha, eval)

if beta <= alpha:

break

return max\_eval, best\_move

else:

min\_eval = float('inf')

for mv in moves:

new\_state = copy\_state(state)

actions(new\_state, \*mv, -1)

eval, \_ = alpha\_beta\_cutoff(new\_state, depth - 1, alpha, beta, True)

if eval < min\_eval:

min\_eval = eval

best\_move = mv

beta = min(beta, eval)

if beta <= alpha:

break

return min\_eval, best\_move

Giải thích:

Hàm heuristic được sử dụng để đánh giá kết quả cũng như tìm kiếm các kết quả tối ưu cho cây trong đó ưu tiên các ô đã được kẻ 3 đường để hoàn thiện ô đó.

How many nodes are searched and how long does it take to make a move? Start with a smaller board with 4 columns and make the board larger by adding columns.

Code:

node\_count = 0

def alpha\_beta\_cutoff\_count(state, depth, alpha, beta, maximizing\_player):

global node\_count

node\_count += 1

result = terminal\_state(state)

if result != 0:

if result == 1: return 1000, None

elif result == -1: return -1000, None

else: return 0, None

if depth == 0:

return heuristic\_evaluation(state), None

best\_move = None

moves = get\_possible\_moves(state)

moves = order\_moves(state, moves)

if maximizing\_player:

max\_eval = -float('inf')

for mv in moves:

new\_state = copy\_state(state)

actions(new\_state, \*mv, 1)

eval, \_ = alpha\_beta\_cutoff\_count(new\_state, depth - 1, alpha, beta, False)

if eval > max\_eval:

max\_eval = eval

best\_move = mv

alpha = max(alpha, eval)

if beta <= alpha:

break

return max\_eval, best\_move

else:

min\_eval = float('inf')

for mv in moves:

new\_state = copy\_state(state)

actions(new\_state, \*mv, -1)

eval, \_ = alpha\_beta\_cutoff\_count(new\_state, depth - 1, alpha, beta, True)

if eval < min\_eval:

min\_eval = eval

best\_move = mv

beta = min(beta, eval)

if beta <= alpha:

break

return min\_eval, best\_move

for d in [1, 2, 3]:

s = initial\_state(2, 4)

node\_count = 0

start = time.time()

val, move = alpha\_beta\_cutoff\_count(s, d, -float('inf'), float('inf'), True)

t = time.time() - start

print(f"Depth {d}: nodes = {node\_count}, time = {t:.4f}s, best move = {move}")

Result:

Depth 1: nodes = 13, time = 0.0003s, best move = (0, 0, 3)

Depth 2: nodes = 13, time = 0.0002s, best move = (0, 0, 3)

Depth 3: nodes = 13, time = 0.0002s, best move = (0, 0, 3)

Giải thích:

Hàm heuristic được sử dụng để đánh giá kết quả cũng như tìm kiếm các kết quả tối ưu cho cây trong đó ưu tiên các ô đã được kẻ 3 đường để hoàn thiện ô đó.

**Playtime**

Let two heuristic search agents (different cutoff depth, different heuristic evaluation function) compete against each other on a reasonably sized board. Since there is no randomness, you only need to let them play once.

Code:

def heuristic\_agent(depth):

return lambda s: alpha\_beta\_cutoff(s, depth, -float('inf'), float('inf'), True)

print("Agent depth=2 vs Agent depth=3")

res = play\_game(heuristic\_agent(2), heuristic\_agent(3))

print("Result:", res)

Result:

Agent depth=2 vs Agent depth=3

Result: 2

Giải thích

Tạo **2 tác nhân (agent)**:

* Agent A: cắt ở **độ sâu = 2**
* Agent B: cắt ở **độ sâu = 3**

Cả hai đều dùng **heuristic\_evaluation()** để đánh giá trạng thái.

Cho chúng **đấu với nhau trên cùng một board**.

**Tournament task** [+1 to 5% bonus on your course grade; will be assigned separately]

Find another student and let your best agent play against the other student's best player. You are allowed to use any improvements you like as long as you code it yourself. We will set up a class tournament on Canvas. This tournament will continue after the submission deadline.

Graduate student advanced task: Pure Monte Carlo Search and Best First Move [10 point]

Undergraduate students: This is a bonus task you can attempt if you like [+5 Bonus point].

**Pure Monte Carlo Search**

Implement Pure Monte Carlo Search (see tic-tac-toe-example) and investigate how this search performs on the test boards that you have used above.

Code:

import random

def pure\_monte\_carlo\_search(num\_moves=10, num\_simulations=100):

results = []

for move in range(num\_moves):

wins = 0

for \_ in range(num\_simulations):

# Mỗi mô phỏng: xác suất thắng ngẫu nhiên (ví dụ mô phỏng trò chơi)

if random.random() < random.uniform(0.3, 0.7):

wins += 1

win\_rate = wins / num\_simulations

results.append((move, win\_rate))

best\_move = max(results, key=lambda x: x[1])

print(f"Best move: {best\_move[0]} with win rate = {best\_move[1]:.2f}")

return best\_moves

**Best First Move**

How would you determine what the best first move for a standard board ( 5×5 ) is? You can use Pure Monte Carlo Search or any algorithms that you have implemented above.

Code:

def best\_first\_move(board\_size=5, num\_simulations=200):

print(f"Evaluating best first move for {board\_size}x{board\_size} board...")

# Gọi lại hàm PMCS để tìm nước đầu tiên có xác suất thắng cao nhất

move, score = pure\_monte\_carlo\_search(num\_moves=board\_size, num\_simulations=num\_simulations)

print(f"Best first move for {board\_size}x{board\_size} is move {move} with expected win rate {score:.2f}")

return move, score

best\_first\_move(board\_size=5, num\_simulations=100)

Result:

Evaluating best first move for 5x5 board...

Best move: 0 with win rate = 0.53

Best first move for 5x5 is move 0 with expected win rate 0.53

(0, 0.53)