A picture containing application

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

**KHOA CÔNG NGHỆ THÔNG TIN**

Icon

Description automatically generated with low confidence

**BÁO CÁO**

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

**LAB 04**

**Sinh viên thực hiện**

Nguyễn Văn Anh Tú – 3122410445

Trương Hữu Nghĩa - 3122410263

Đỗ Khôi Nguyên - 3122410266

Lai Tấn Tài - 3122410366

**TP. HCM THÁNG 09/2025**

MỤC LỤC

Contents

[Example 3](#_Toc211962841)

[Tic-Tac-Toe 3](#_Toc211962842)

[Solving Tic-Tac-Toe with AND-OR-Tree Search 3](#_Toc211962843)

[Play Tic-Tac-Toe Interactively (Simple Implementation) 3](#_Toc211962844)

[Solving Tic-Tac-Toe with Minimax Search and Alpha-Beta Pruning 3](#_Toc211962845)

[Solving Tic-Tac-Toe with Heuristic Alpha-Beta Tree Search 3](#_Toc211962846)

[Solving Tic-Tac-Toe with Monte Carlo Tree Search 3](#_Toc211962847)

[Assginment 3](#_Toc211962848)

[Mean Connect 4 3](#_Toc211962849)

Example

Tic-Tac-Toe

…

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

…

Play Tic-Tac-Toe Interactively (Simple Implementation)

…

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:
  + Max (x): Cố gắng tối đa hóa kết quả.
  + Min (o): Cố gắng tối thiểu hóa kết quả.
* Giá trị (Utillity): kết quả của trò chơi được định lượng:
  + X thắng +1
  + O thắng -1
  + 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

board = empty\_board()

show\_board(board)

actions(board)

Giải thích code:

* Hàm actions(board) được định ngh ĩa lại (ở ô [19]) để thực hiện việc sắp xếp.
* Nó tạo một danh sách priority (độ ưu tiên):
  + Ô trung tâm (vị trí 4): ưu tiên 2
  + Các góc (0, 2, 6, 8): ưu tiên 1
  + 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:
  + Nếu player (người chơi) có 2 quân cờ và 1 ô trống: score += 0.4. Đây là một "cơ hội" thắng.
  + 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).

1. 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:
  + terminal: Trò chơi đã kết thúc ở đây?
  + (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.

1. 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

…

Assginment

Mean Connect 4

Alpha-Beta Pruining 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 chiển 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 enviroment 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 cac 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.