Mean Connect 4

Alpha-Beta Pruining Minimax

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.