Bài tập Thực hành 3

Thông tin sinh viên

- MSSV: 22850034
- Họ và tên: Cao Hoài Việt
- Email: viet.ch2612@gmail.com

Yêu cầu

1. Trình bày lại theo cách hiểu của mình:

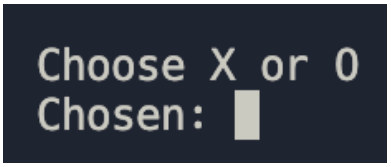
Diễn giải ý nghĩa của các hàm và các biến

1. Định nghĩa giá trị số nguyên `HUMAN` (tức người chơi) bằng `-1` và giá trị số nguyên `COMP` (máy) bằng `1`.

2. Định nghĩa ma trận 2 chiều `board` với kích thước 3x3, với các phần tử ban đầu được khởi tạo bằng 0.
    a. Sau khi được sửa lại, kích thước của board sẽ có thể tuỳ thích dựa vào biến `BOARD_SIZE`
    b. Biến này chỉ dùng để tính độ sâu tối đa, không dùng để khởi tạo mảng do đề bài có yêu cầu nhập trạng thái bắt đầu là tuỳ biến.
3. `evaluate(state)` : Hàm này dùng để đánh giá trạng thái của bàn cờ hiện tại và trả về kết quả +1 nếu máy tính thắng, -1 nếu người chơi thắng, 0 nếu hòa.
4. `wins(state, player)` : Hàm này kiểm tra xem một người chơi cụ thể có thắng hay không bằng cách kiểm tra trên ba hàng, ba cột hoặc hai đường chéo của bàn cờ. Nếu người chơi thắng, hàm trả về True, ngược lại trả về False.
5. `game_over(state)` : Hàm này kiểm tra xem trò chơi đã kết thúc chưa, nếu một trong hai người chơi đã thắng hoặc bàn cờ đã được điền đầy, hàm trả về True, ngược lại trả về False.
6. `empty_cells(state)` : Hàm này trả về danh sách các ô trống còn lại trên bàn cờ.
7. `valid_move(x, y)` : Hàm này kiểm tra xem nước đi của người chơi có hợp lệ hay không bằng cách kiểm tra xem ô được chọn có phải là ô trống không.
8. `set_move(x, y, player)` : Hàm này đặt nước đi của người chơi lên bàn cờ nếu nước đi đó hợp lệ.
9. `minimax(state, depth, player)` : Hàm này thực hiện thuật toán Minimax để tìm nước đi tốt nhất cho máy tính. Hàm này sử dụng đệ quy để duyệt qua tất cả các trạng thái có thể của bàn cờ và trả về nước đi tốt nhất cho máy tính.
10. `clean()` : Hàm này xóa màn hình console trước khi in hình bàn cờ mới lên.
11. `render(state, c_choice, h_choice)` : Hàm này in bàn cờ lên console.

Diễn giải các bước thực hiện trong hàm `main()` khi thực thi:

1. Cho người chơi chọn cờ của mình, `X` hoặc `O`, nếu không phải thì sẽ nhập lại. Nếu người chơi chọn `X` thì máy sẽ là `O` và ngược lại.



2. Cho người chơi chọn đi trước hay không, nếu nhập `Y` tức sẽ đi trước và `N` nếu muốn để máy đi trước.

3. Chương trình sẽ chạy vòng lặp `while` tới khi nào không còn chỗ trống và game chưa kết thúc (Một trong 2 thắng).
    a. Trong vòng lặp thì sẽ luân phiên lượt đi giữa người chơi và máy.
4. Khi vòng lặp `while` kết thúc thì sẽ kiểm tra kết quả trò chơi: Người chơi thắng, máy thắng hay hoà để hiển thị lên màn hình console.

2. Dựa trên code có sẵn để cài đặt chạy trên máy của mình.

Đã hoàn thành

3. Điều chỉnh code để chấp nhận input bất kỳ (trạng thái ban đầu bất kỳ). Ví dụ, đã có sẵn một số bước đã thực hiện.

Để điều chỉnh code chấp nhận input bất kỳ, em đã thêm 1 biến tên là pre_input_board tức trạng thái có sẵn các nước đi. Ví dụ input dưới dây đã có sẵn các nước đi X và O

```
pre_input_board = [
    ['X', 'O', 'O', 'X'],
    [0, 0, 0, 'O'],
    [0, 0, 0, 'X'],
    [0, 0, 0, 'O']
]
```

```
# If the human selected X
board = [
    [-1, 1, 1, -1],
    [0, 0, 0, 1],
    [0, 0, 0, -1],
    [0, 0, 0, 1]
]
```

Sau đó, em viết thêm hàm `convert_board` để biến đổi các nước đi có sẵn thành dạng `-1`, `1` tuỳ theo lựa chọn của người chơi. Như này biến board có thể tiếp tục thực hiện mà không gặp vấn đề.

```
def convert_board(board, h_choice, c_choice):
    print(h_choice)
    print(c_choice)
    rows = len(board)
    cols = len(board[0])

    for i in range(rows):
        for j in range(cols):
            if board[i][j] == h_choice:
                board[i][j] = HUMAN
            elif board[i][j] == c_choice:
                board[i][j] = COMP

    return board
```

Tất nhiên chúng ta vẫn có thể bắt đầu bằng một board trống

```
# 4x4
pre_input_board = [
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]
]
```

4. Thực hiện trên input có giới hạn lớn hơn như 10x10 hay 20x20 với luật chơi có thể thay đổi như 5 ô liên tiếp.

4.1 Luật chơi có thể thay đổi

Ở đây em sẽ định nghĩa thêm một biến mới là `NUM_TO_WIN` để xác định người chơi cần có bao nhiêu X hoặc O liên tiếp để chiến thắng.

Từ biến này, em sẽ truyền vào hàm `wins()` để kiểm tra trạng thái win. Các hoạt động là sẽ sử dụng vòng lặp for kiểm tra từng dòng, từng cột và từng hàng chéo thay vì định nghĩa sẵn các trạng thái win của bài code mẫu.

```
def wins(state, player, num_to_win=NUM_TO_WIN):
    """
    This function tests if a specific player wins on a board of any size.
    :param state: the state of the current board as a 2D list
    :param player: a human or a computer
    :param num_to_win: the number of consecutive marks needed to win
    :return: True if the player wins
    """
    rows = len(state)
    cols = len(state[0])

    # Check rows
    for i in range(rows):
        for j in range(cols - num_to_win + 1):
            if all(state[i][j+k] == player for k in range(num_to_win)):
                return True

    # Check columns
    for i in range(rows - num_to_win + 1):
        for j in range(cols):
            if all(state[i+k][j] == player for k in range(num_to_win)):
                return True

    # Check diagonals
    for i in range(rows - num_to_win + 1):
        for j in range(cols - num_to_win + 1):
            if all(state[i+k][j+k] == player for k in range(num_to_win)):
                return True

    for i in range(num_to_win-1, rows):
        for j in range(cols - num_to_win + 1):
            if all(state[i-k][j+k] == player for k in range(num_to_win)):
                return True

    return False
```

4.2 Thực hiện trên input có giới hạn lớn hơn như 10x10 hay 20x20

Để có thể chạy được trên bàn cờ có giới hạn 3x3 thì chúng ta không thể sử dụng thuật toán minimax được. Vì nếu trên bàn cờ 4x4 thì số lượng các nước đi có thể xảy ra lên tới 16! = 20.922.789.888.000 (20 triệu tỉ) nên việc tính toán hết là hoàn toàn không thể. Nên chúng ta sẽ phải áp dụng tỉa nhánh alpha-beta và tìm kiếm ở độ sâu chấp nhận được.

Để giới hạn độ sâu tìm kiếm, ta sử dụng biến `MAX_DEPTH` để giới hạn. Trong bài này, em sử dụng độ sâu tối đa là `9`

Sau đó, sử dụng biến trên làm input cho hàm tỉa nhánh alpha_beta. Khi khởi tạo, alpha sẽ bằng $-\infty$ và beta sẽ bằng $+\infty$

```
alpha = -infinity
beta = +infinity
```

Dưới đây là hàm tỉa nhánh

```
def alpha_beta_pruning(state, depth, player, alpha, beta):
    """
    AI function that choice the best move
    :param state: current state of the board
    :param depth: node index in the tree (0 <= depth <= BOARD_SIZE ** 2),
    but never nine in this case (see iaturn() function)
    :param player: an human or a computer
    :param alpha: the alpha value for alpha-beta pruning
    :param beta: the beta value for alpha-beta pruning
    :return: a list with [the best row, best col, best score]
    """
    if player == COMP:
        best = [-1, -1, -infinity]
    else:
        best = [-1, -1, +infinity]

    if depth == MAX_DEPTH:
        depth = MAX_DEPTH

    if depth == 0 or game_over(state):
        score = evaluate(state)
        return [-1, -1, score]

    for cell in empty_cells(state):
        x, y = cell[0], cell[1]
        state[x][y] = player
        score = alpha_beta_pruning(state, depth - 1, -player, alpha, beta)
        state[x][y] = 0
        score[0], score[1] = x, y

        if player == COMP:
            if score[2] > best[2]:
                best = score  # max value
            alpha = max(alpha, score[2])
            if beta <= alpha:
                break
        else:
            if score[2] < best[2]:
                best = score  # min value
            beta = min(beta, score[2])
            if beta <= alpha:
```

```
                break

    return best
```

Lưu ý khi chạy chương trình

Thuật toán của em vẫn chưa thể chạy được trên bảng 4x4 với hệ số depth ≥ 10.

Cách chạy thuật toán:

```
python alpha_beta_prening.py
# hoặc
python3 alpha_beta_prening.py
```