**BÁO CÁO DỰ ÁN: TÌM KIẾM ĐỐI KHÁNG - CHƠI CONNECT 4 (ADVERSARIAL SEARCH: PLAYING CONNECT 4)**

**I. GIỚI THIỆU VÀ MỤC TIÊU DỰ ÁN**

**1.1. Giới thiệu Trò chơi Connect 4**

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

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

**1.2. Mục tiêu Học tập (Learning Outcomes)**

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

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

**II. TASK 1: ĐỊNH NGHĨA BÀI TOÁN TÌM KIẾM ĐỐI KHÁNG**

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

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

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

**2. Hành động (Actions):**

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

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

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

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

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

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

**5. Utility (Giá trị Lợi ích):**

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

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

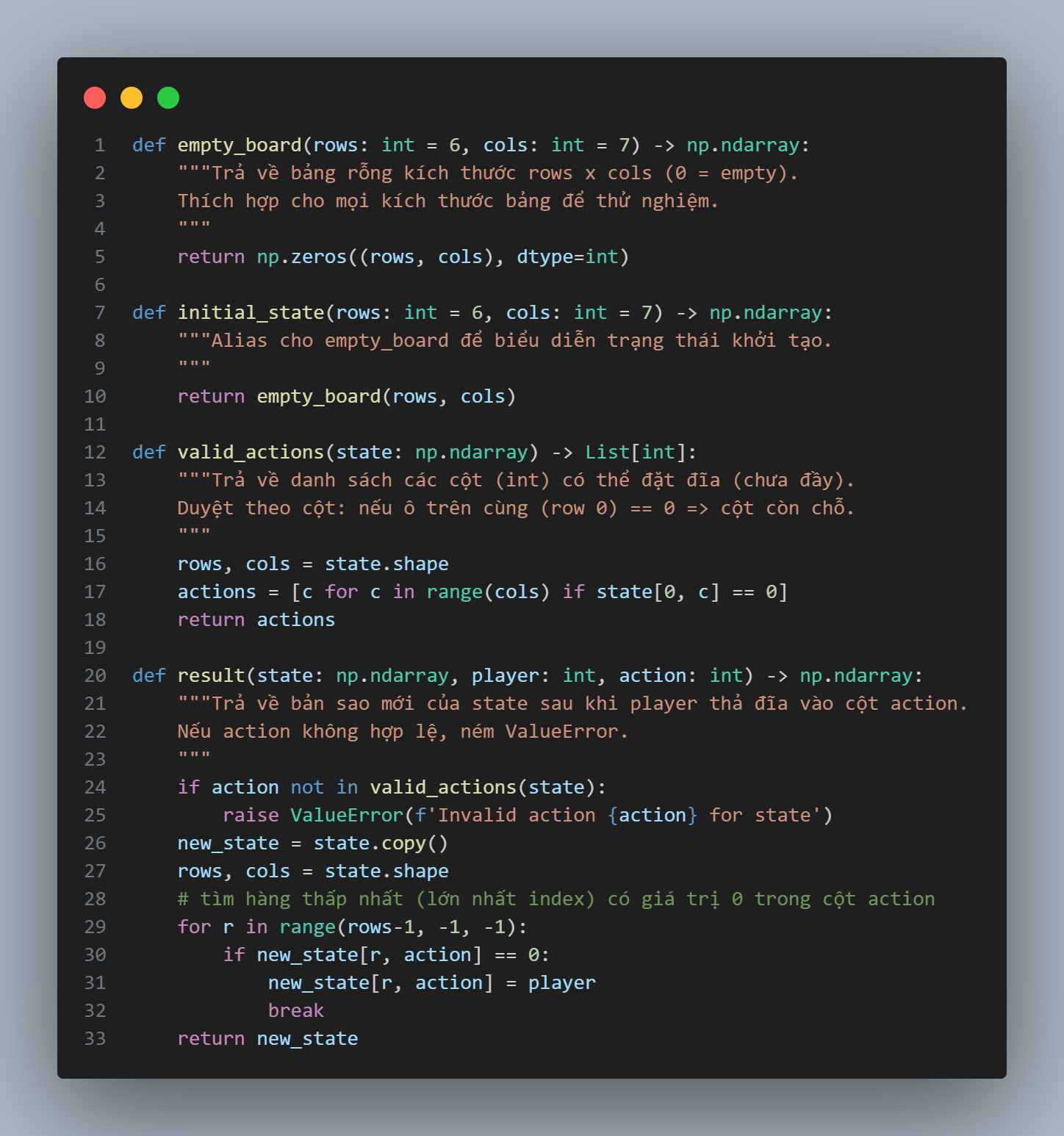
**a. Kích thước Không gian Trạng thái (State Space Size)**

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

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

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

**Code**





**III. TASK 2: GAME ENVIRONMENT VÀ HÀM ĐÁNH GIÁ HEURISTIC**

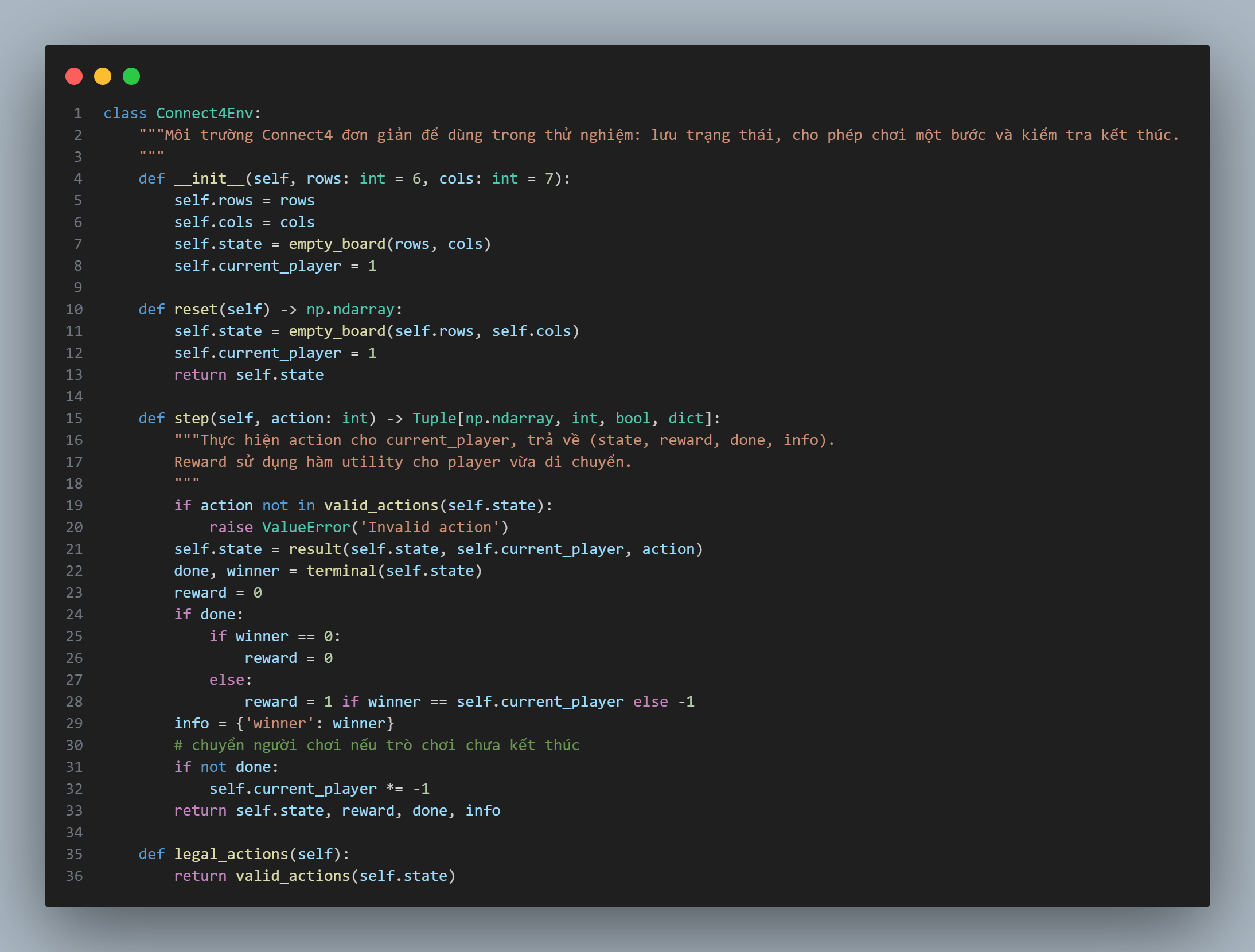
**1. Triển khai Cơ sở: Random Agent** (Cell 13)

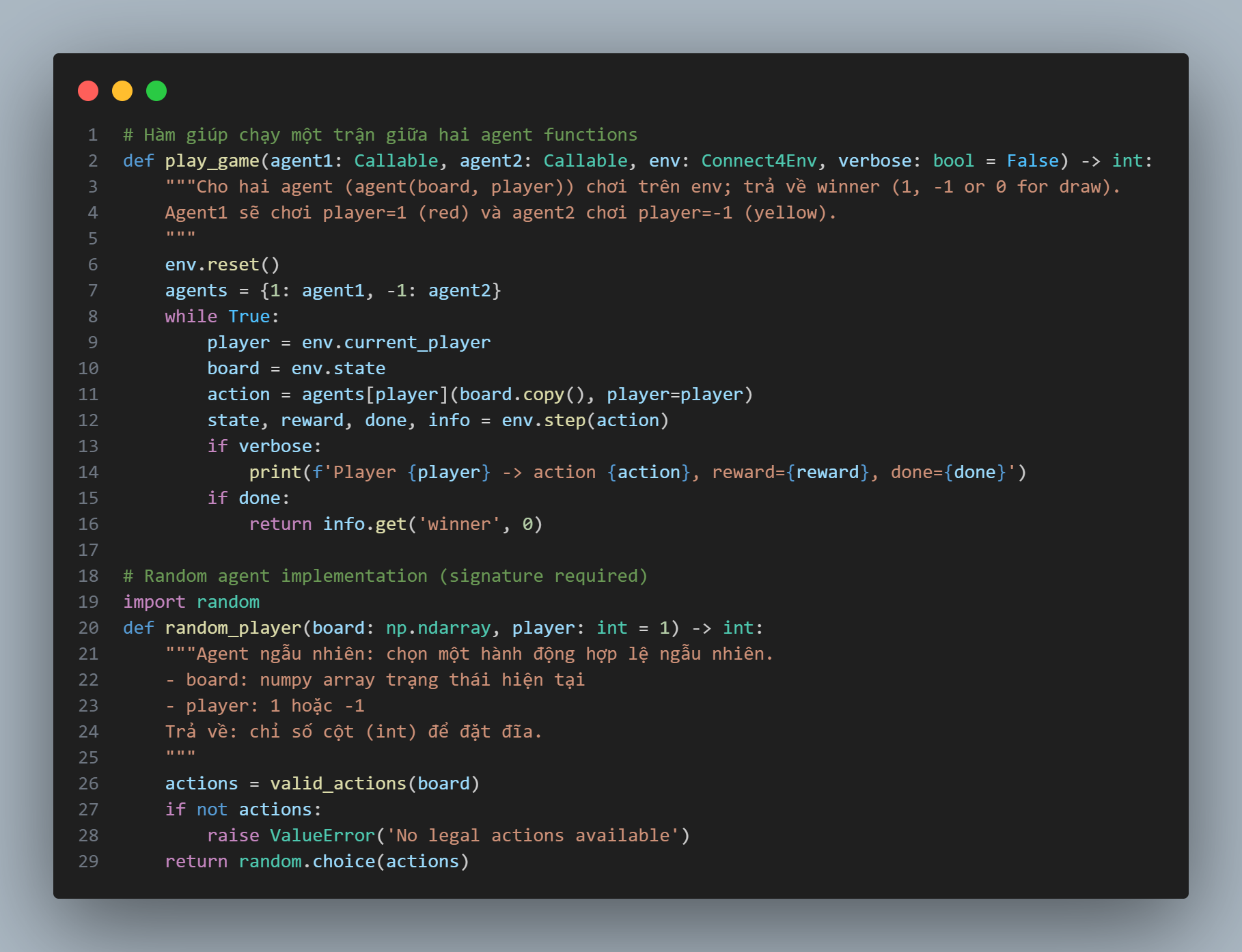
* Mục đích: Kiểm tra tính năng của môi trường game và làm đối thủ benchmark cơ sở.
* Chiến lược: Tác tử random\_agent chọn một hành động (cột) ngẫu nhiên từ danh sách các hành động hợp lệ.

**CODE**



**Xây dựng helper function**



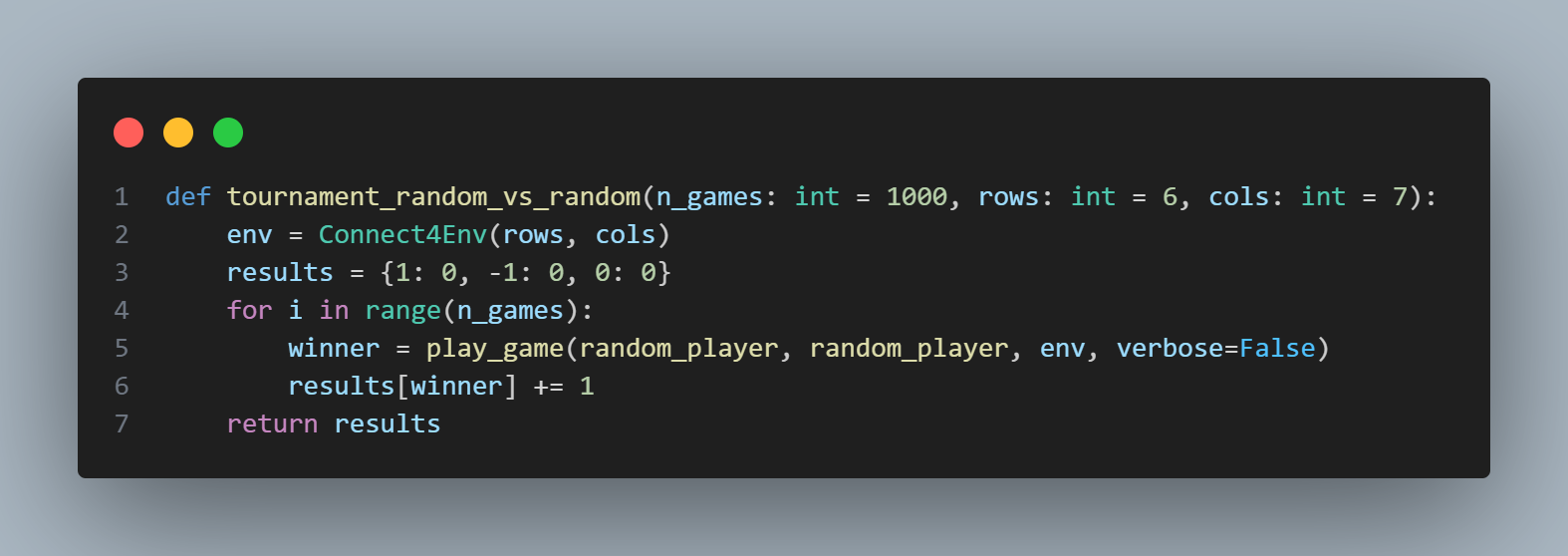


**2. Thiết kế Hàm Đánh giá Heuristic (eval\_function)** (Cell 15)

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

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

**CODE**



**IV. TASK 3: TRIỂN KHAI MINIMAX VỚI ALPHA-BETA PRUNING**

**1. Triển khai Alpha-Beta Pruning**

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

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

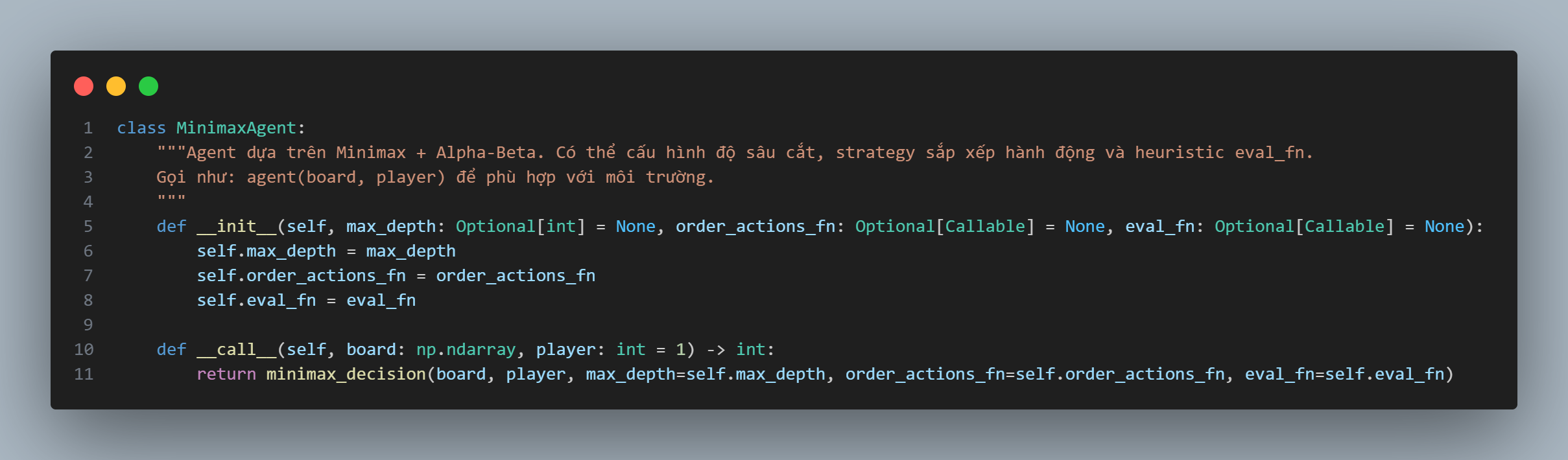
**2. Logic MAX và MIN**

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

**Code:**





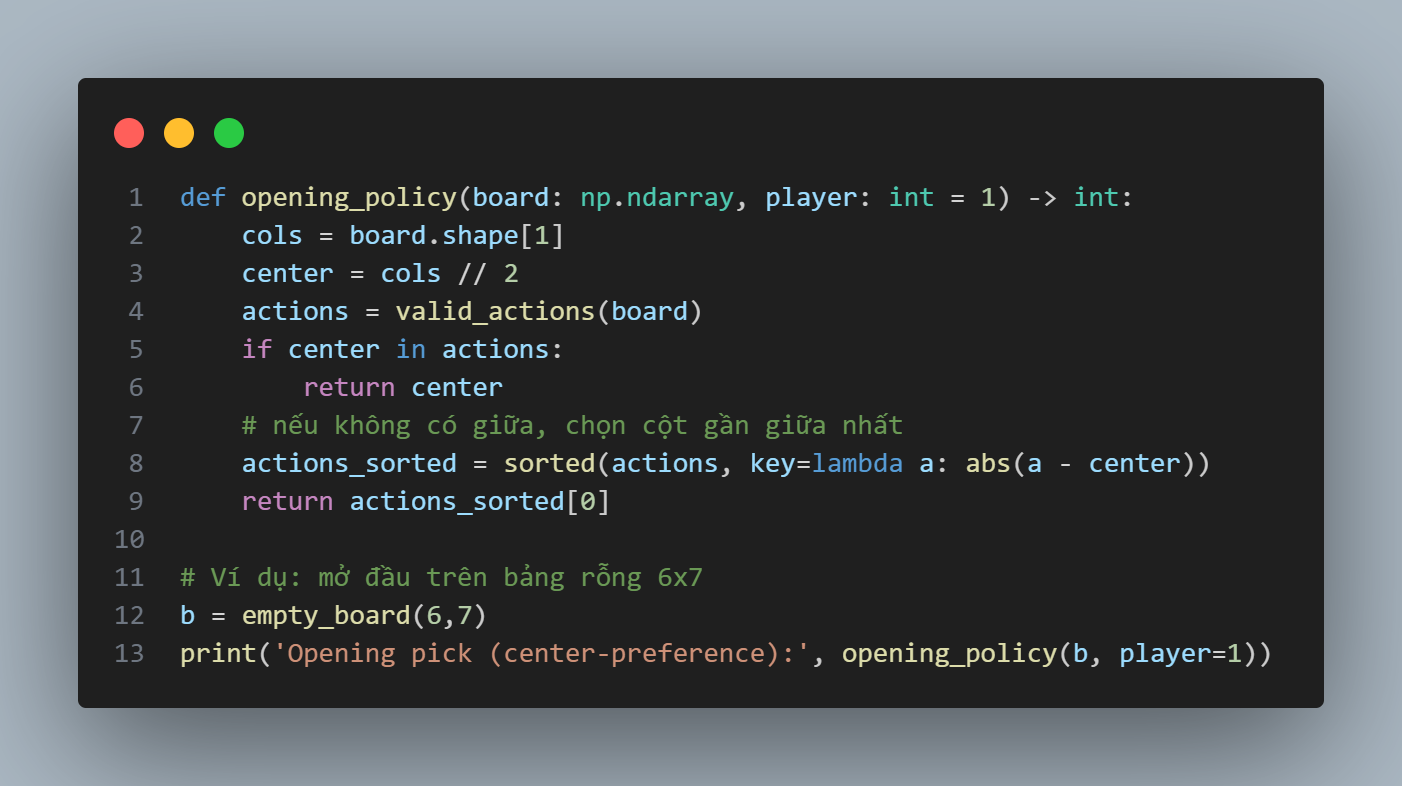




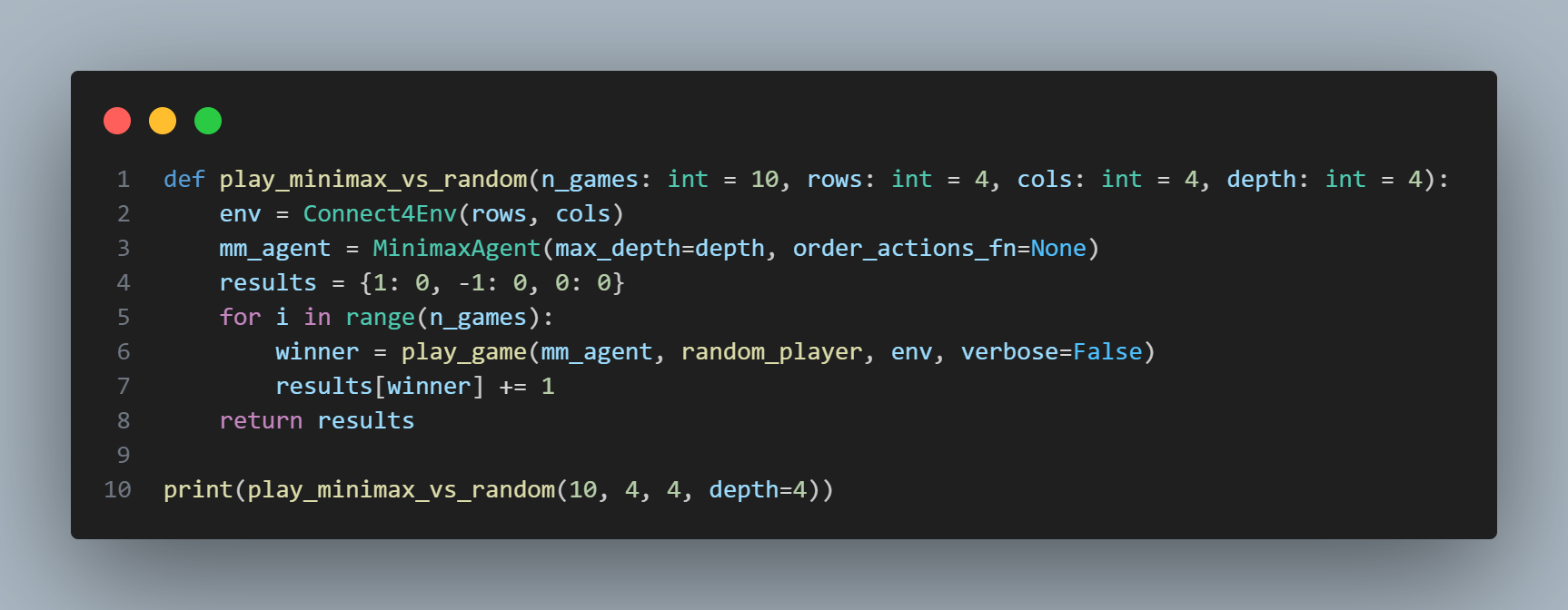
**Move Ordering**



**The first few move**



**Play time**



V. TASK 4: HEURISTIC ALPHA-BETA TREE SEARCH (GIỚI HẠN ĐỘ SÂU)

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

**1. Triển khai Tác tử Chiến lược**

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

**2. Ý nghĩa**

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

**Code**

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

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

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

    Kết quả chuẩn hóa về [-1,1].

    """

    rows, cols = state.shape

    def windows\_of\_length\_4(s):

        # rows

        for r in range(rows):

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

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

        # cols

        for c in range(cols):

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

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

        # diag \ (down-right)

        for r in range(rows - 4 + 1):

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

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

        # anti-diag / (down-left)

        for r in range(rows - 4 + 1):

            for c in range(3, cols):

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

    # weights

    W4 = 1000.0

    W\_open3 = 50.0

    W\_3 = 10.0

    W\_open2 = 3.0

    W\_center = 1.0

    score = 0.0

    def eval\_window(win, p):

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

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

        if cnt\_p == 4: return W4

        if cnt\_opp == 4: return -W4

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

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

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

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

        return 0.0

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

        score += eval\_window(win, player)

    # center control bonus

    center\_col = cols // 2

    for r in range(rows):

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

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

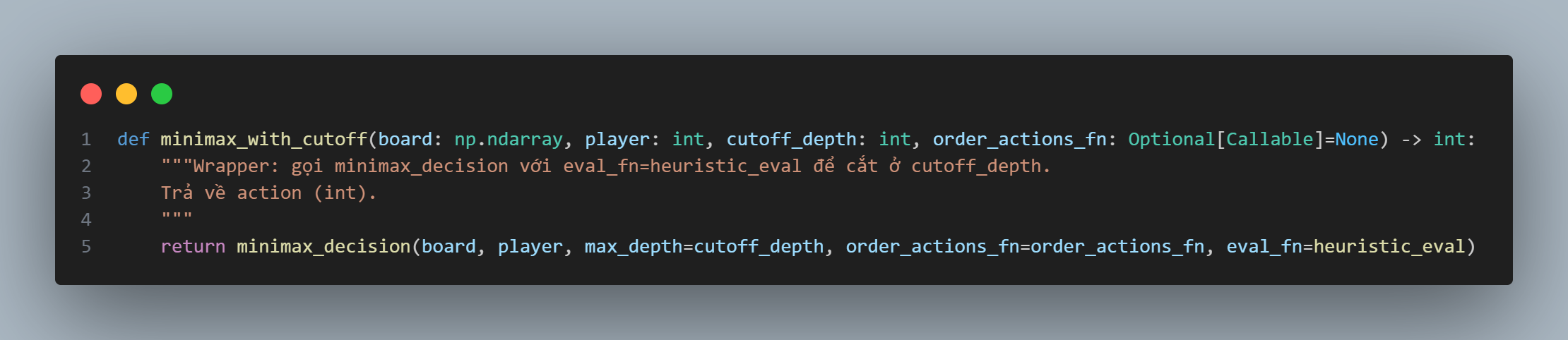
    # normalize

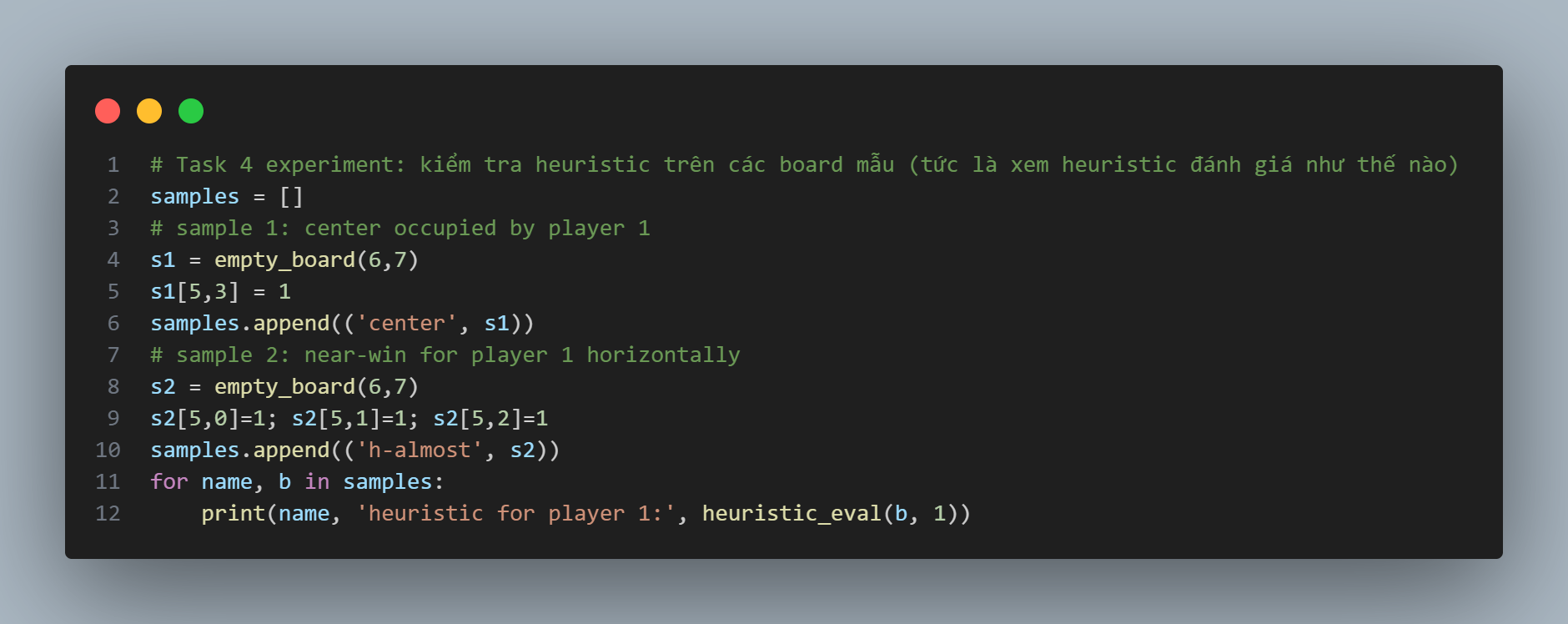
    approx\_max = W4 + (W\_open3 \* 10) + (W\_open2 \* 20) + (W\_center \* rows)

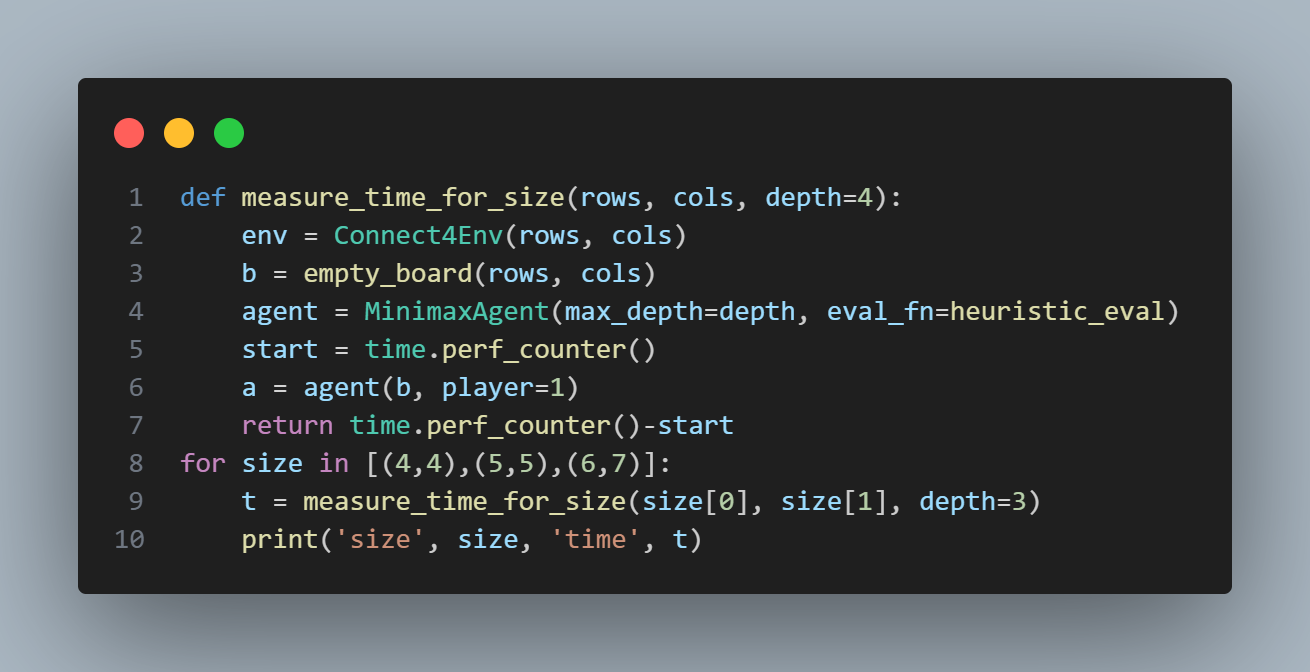
    val = max(-1.0, min(1.0, score / approx\_max))

    return val

**Cutting Off Search**







**Play time**

env = Connect4Env(6,7)

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

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

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

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

**ADVANCE TASK**

**MONTE CARLO SEARCH**



**BEST FIRST MOVE**

