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): $3^{R \times C} = 3^{6 \times 7} = 3^{42} \approx 1.5 \times 10^{20}$trạng 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 $7^{42} \approx 1.6 \times 10^{35}$ 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

```python
def empty_board(rows: int = 6, cols: int = 7) -> np.ndarray:
    """Trả về bảng rỗng kích thước rows x cols (0 = empty).
    Thích hợp cho mọi kích thước bảng để thử nghiệm.
    """
    return np.zeros((rows, cols), dtype=int)

def initial_state(rows: int = 6, cols: int = 7) -> np.ndarray:
    """Alias cho empty_board để biểu diễn trạng thái khởi tạo.
    """
    return empty_board(rows, cols)

def valid_actions(state: np.ndarray) -> List[int]:
    """Trả về danh sách các cột (int) có thể đặt đĩa (chưa đầy).
    Duyệt theo cột: nếu ô trên cùng (row 0) == 0 => cột còn chỗ.
    """
    rows, cols = state.shape
    actions = [c for c in range(cols) if state[0, c] == 0]
    return actions

def result(state: np.ndarray, player: int, action: int) -> np.ndarray:
    """Trả về bản sao mới của state sau khi player thả đĩa vào cột action.
    Nếu action không hợp lệ, ném ValueError.
    """
    if action not in valid_actions(state):
        raise ValueError(f'Invalid action {action} for state')
    new_state = state.copy()
    rows, cols = state.shape
    # tìm hàng thấp nhất (lớn nhất index) có giá trị 0 trong cột action
    for r in range(rows-1, -1, -1):
        if new_state[r, action] == 0:
            new_state[r, action] = player
            break
    return new_state
```

```python
def _check_four_in_a_row(arr: List[int], player: int) -> bool:
    """Helper: kiểm tra chuỗi con độ dài 4 của cùng màu trong list 1D.
    """
    count = 0
    for v in arr:
        if v == player:
            count += 1
            if count >= 4:
                return True
        else:
            count = 0
    return False

def terminal(state: np.ndarray) -> Tuple[bool, int]:
    """Trả về (is_terminal, winner).
    Nếu trò chơi đã kết thúc trả về (True, winner) với winner ∈ {1, -1, 0}.
    Nếu chưa kết thúc trả về (False, 0).
    """
    winner = 0  # 0 if no winner yet, 1 or -1 indicate player who won
    rows, cols = state.shape
    # kiểm tra hàng ngang
    for r in range(rows):
        if _check_four_in_a_row(list(state[r, :]), 1):
            return True, 1
        if _check_four_in_a_row(list(state[r, :]), -1):
            return True, -1
    # kiểm tra cột dọc
    for c in range(cols):
        col_vals = [state[r, c] for r in range(rows)]
        if _check_four_in_a_row(col_vals, 1):
            return True, 1
        if _check_four_in_a_row(col_vals, -1):
            return True, -1
    # kiểm tra đường chéo (
    # các đường chéo có thể được trích xuất bằng cách cố định (r0,c0) và đi xuống phải
    for r0 in range(rows):
        for c0 in range(cols):
            diag = []
            r, c = r0, c0
            while r < rows and c < cols:
                diag.append(state[r, c])
                r += 1
                c += 1
            if _check_four_in_a_row(diag, 1):
                return True, 1
            if _check_four_in_a_row(diag, -1):
                return True, -1
    # kiểm tra đường chéo (/)
    for r0 in range(rows):
        for c0 in range(cols):
            diag = []
            r, c = r0, c0
            while r < rows and c >= 0:
                diag.append(state[r, c])
                r += 1
                c -= 1
            if _check_four_in_a_row(diag, 1):
                return True, 1
            if _check_four_in_a_row(diag, -1):
                return True, -1
    # nếu bảng đầy thì kết thúc hòa
    if not any(state[0, c] == 0 for c in range(cols)):
        return True, 0
    return False, 0

def utility(state: np.ndarray, player: int) -> int:
    """Trả về utility từ quan điểm của player: +1 nếu player thắng, -1 nếu thua, 0 khác.
    """
    is_term, winner = terminal(state)
    if not is_term:
        return 0
    if winner == 0:
        return 0
    return 1 if winner == player else -1
```

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

```python
def visualize(board):
    plt.axes()
    rectangle=plt.Rectangle((-0.5,len(board)*-1+0.5),len(board[0]),len(board),fc='blue')
    circles=[]
    for i,row in enumerate(board):
        for j,val in enumerate(row):
            color='white' if val==0 else 'red' if val==1 else 'yellow'
            circles.append(plt.Circle((j,i*-1),0.4,fc=color))

    plt.gca().add_patch(rectangle)
    for circle in circles:
        plt.gca().add_patch(circle)

    plt.axis('scaled')
    plt.show()

board = [[0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 1, 0, 0, 0],
         [0, 0, 0, 1, 0, 0, 0],
         [0,-1,-1, 1,-1, 0, 0]]
visualize(board)
```

Xây dựng helper function

```python
class Connect4Env:
    """Môi trường Connect4 đơn giản để dùng trong thử nghiệm: lưu trạng thái, cho phép chơi một bước và kiểm tra kết thúc.
    """
    def __init__(self, rows: int = 6, cols: int = 7):
        self.rows = rows
        self.cols = cols
        self.state = empty_board(rows, cols)
        self.current_player = 1

    def reset(self) -> np.ndarray:
        self.state = empty_board(self.rows, self.cols)
        self.current_player = 1
        return self.state

    def step(self, action: int) -> Tuple[np.ndarray, int, bool, dict]:
        """Thực hiện action cho current_player, trả về (state, reward, done, info).
        Reward sử dụng hàm utility cho player vừa đi chuyển.
        """
        if action not in valid_actions(self.state):
            raise ValueError('Invalid action')
        self.state = result(self.state, self.current_player, action)
        done, winner = terminal(self.state)
        reward = 0
        if done:
            if winner == 0:
                reward = 0
            else:
                reward = 1 if winner == self.current_player else -1
        info = {'winner': winner}
        # chuyển người chơi nếu trò chơi chưa kết thúc
        if not done:
            self.current_player *= -1
        return self.state, reward, done, info

    def legal_actions(self):
        return valid_actions(self.state)
```

```python
# Hàm giúp chạy một trận giữa hai agent functions
def play_game(agent1: Callable, agent2: Callable, env: Connect4Env, verbose: bool = False) -> int:
    """Cho hai agent (agent(board, player)) chơi trên env; trả về winner (1, -1 or 0 for draw).
    Agent1 sẽ chơi player=1 (red) và agent2 chơi player=-1 (yellow).
    """
    env.reset()
    agents = {1: agent1, -1: agent2}
    while True:
        player = env.current_player
        board = env.state
        action = agents[player](board.copy(), player=player)
        state, reward, done, info = env.step(action)
        if verbose:
            print(f'Player {player} -> action {action}, reward={reward}, done={done}')
        if done:
            return info.get('winner', 0)

# Random agent implementation (signature required)
import random
def random_player(board: np.ndarray, player: int = 1) -> int:
    """Agent ngẫu nhiên: chọn một hành động hợp lệ ngẫu nhiên.
    - board: numpy array trạng thái hiện tại
    - player: 1 hoặc -1
    Trả về: chỉ số cột (int) để đặt đĩa.
    """
    actions = valid_actions(board)
    if not actions:
        raise ValueError('No legal actions available')
    return random.choice(actions)
```

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

```python
def tournament_random_vs_random(n_games: int = 1000, rows: int = 6, cols: int = 7):
    env = Connect4Env(rows, cols)
    results = {1: 0, -1: 0, 0: 0}
    for i in range(n_games):
        winner = play_game(random_player, random_player, env, verbose=False)
        results[winner] += 1
    return results
```

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.
  - B (Beta): Giá trị tối thiểu MIN có thể đạt được trên đường đi hiện tại.

2. Logic MAX và MIN

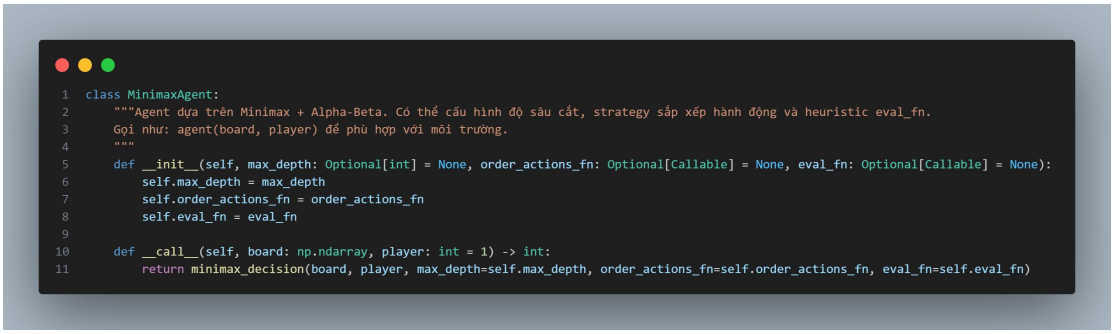
- max_value_alpha_beta(state, depth, alpha, beta, player):
  - Giả định đây là lượt của MAX Player.
  - Duyệt qua các hành động, cập nhật giá trị v = max(v, min_value_alpha_beta_dots)).
  - Cắt tỉa: Nếu v ≥ beta, dừng tìm kiếm nhánh này và trả về v (cắt tỉa Beta).
  - Cập nhật $\alpha = \max(\alpha, v)$.
- 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 $\beta = \min(\beta, v)$.

Code:

```python
def minimax_decision(board: np.ndarray, player: int, max_depth: Optional[int] = None, order_actions_fn: Optional[Callable] = None, eval_fn: Optional[Callable] = None) -> int:
    """Trả về hành động tốt nhất cho player bằng Minimax + Alpha-Beta.
    - board: numpy array trạng thái hiện tại
    - player: người đang đi (1 hoặc -1)
    - max_depth: nếu không None thì cắt ở độ sâu đó (depth tính theo số lượt từ root, root depth=0)
    - order_actions_fn: hàm tùy chọn để sắp xếp hành động nhằm cải thiện pruning
    - eval_fn: hàm heuristic(state, player) được gọi khi cắt (cutoff)
    """
    nodes = 0
    def max_value(state, alpha, beta, depth):
        nonlocal nodes
        nodes += 1
        is_term, winner = terminal(state)
        if is_term:
            return utility(state, player)
        if max_depth is not None and depth >= max_depth:
            return eval_fn(state, player) if eval_fn is not None else utility(state, player)
        v = -float('inf')
        actions = valid_actions(state)
        if order_actions_fn is not None:
            actions = order_actions_fn(state, player, actions)
        for a in actions:
            v = max(v, min_value(result(state, player, a), alpha, beta, depth+1))
            alpha = max(alpha, v)
            if alpha >= beta:
                break
        return v
```

```python
def min_value(state, alpha, beta, depth):
        nonlocal nodes
        nodes += 1
        is_term, winner = terminal(state)
        if is_term:
            return utility(state, player)
        if max_depth is not None and depth >= max_depth:
            return eval_fn(state, -player) if eval_fn is not None else utility(state, player)
        v = float('inf')
        actions = valid_actions(state)
        if order_actions_fn is not None:
            actions = order_actions_fn(state, -player, actions)
        for a in actions:
            v = min(v, max_value(result(state, -player, a), alpha, beta, depth+1))
            beta = min(beta, v)
            if alpha >= beta:
                break
        return v

    best_action = None
    best_val = -float('inf')
    start = time.perf_counter()
    actions = valid_actions(board)
    if order_actions_fn is not None:
        actions = order_actions_fn(board, player, actions)
    for a in actions:
        v = min_value(result(board, player, a), -float('inf'), float('inf'), 1)
        if v > best_val:
            best_val = v
            best_action = a
    elapsed = time.perf_counter() - start
    return best_action
```



```python
class MinimaxAgent:
    """Agent dựa trên Minimax + Alpha-Beta. Có thể cấu hình độ sâu cắt, strategy sắp xếp hành động và heuristic eval_fn.
    Gọi như: agent(board, player) để phù hợp với môi trường.
    """
    def __init__(self, max_depth: Optional[int] = None, order_actions_fn: Optional[Callable] = None, eval_fn: Optional[Callable] = None):
        self.max_depth = max_depth
        self.order_actions_fn = order_actions_fn
        self.eval_fn = eval_fn

    def __call__(self, board: np.ndarray, player: int = 1) -> int:
        return minimax_decision(board, player, max_depth=self.max_depth, order_actions_fn=self.order_actions_fn, eval_fn=self.eval_fn)
```

```python
def center_first_order(state, player, actions):
    # ưu tiên các cột gần giữa bảng (thường tốt cho Connect4)
    cols = state.shape[1]
    center = (cols - 1) / 2.0
    return sorted(actions, key=lambda a: abs(a - center))

def time_agent_move(agent_fn, board, player=1):
    start = time.perf_counter()
    a = agent_fn(board, player=player)
    elapsed = time.perf_counter() - start
    return a, elapsed

# Tạo board phức tạp hơn để thử nghiệm thời gian
test_board = empty_board(6,7)
# làm đầy một vài cột để tăng branching
for c in [0,1,2,4,5]:
    for r in range(3):
        test_board[5-r, c] = 1 if r % 2 == 0 else -1

# Agent không dùng ordering
agent_plain = MinimaxAgent(max_depth=4, order_actions_fn=None)
a1, t1 = time_agent_move(agent_plain, test_board, player=1)
# Agent dùng ordering trung tâm
agent_ordered = MinimaxAgent(max_depth=4, order_actions_fn=center_first_order)
a2, t2 = time_agent_move(agent_ordered, test_board, player=1)
print('plain:', a1, 'time:', t1)
print('ordered:', a2, 'time:', t2)
```

Move Ordering

```python
def measure_ordering(board, player, depths=(2,3,4)):
    results = []
    for d in depths:
        a_plain, t_plain = None, None
        a_ord, t_ord = None, None
        agent_plain = MinimaxAgent(max_depth=d, order_actions_fn=None)
        agent_ord = MinimaxAgent(max_depth=d, order_actions_fn=center_first_order)
        start = time.perf_counter(); a_plain = agent_plain(board, player=player); t_plain = time.perf_counter()-start
        start = time.perf_counter(); a_ord = agent_ord(board, player=player); t_ord = time.perf_counter()-start
        results.append((d, t_plain, t_ord))
    return results

# Thử trên một board thử nghiệm (đã tạo ở ô trước là test_board)
res = measure_ordering(test_board, player=1, depths=(2,3))
print('depth, time_plain, time_ordered')
for row in res:
    print(row)
```

The first few move

```python
def opening_policy(board: np.ndarray, player: int = 1) -> int:
    cols = board.shape[1]
    center = cols // 2
    actions = valid_actions(board)
    if center in actions:
        return center
    # nếu không có giữa, chọn cột gần giữa nhất
    actions_sorted = sorted(actions, key=lambda a: abs(a - center))
    return actions_sorted[0]

# Ví dụ: mở đầu trên bảng rỗng 6x7
b = empty_board(6,7)
print('Opening pick (center-preference):', opening_policy(b, player=1))
```

Play time

```python
def play_minimax_vs_random(n_games: int = 10, rows: int = 4, cols: int = 4, depth: int = 4):
    env = Connect4Env(rows, cols)
    mm_agent = MinimaxAgent(max_depth=depth, order_actions_fn=None)
    results = {1: 0, -1: 0, 0: 0}
    for i in range(n_games):
        winner = play_game(mm_agent, random_player, env, verbose=False)
        results[winner] += 1
    return results

print(play_minimax_vs_random(10, 4, 4, depth=4))
```

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

```python
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

```python
def minimax_with_cutoff(board: np.ndarray, player: int, cutoff_depth: int, order_actions_fn: Optional[Callable]=None) -> int:
    """Wrapper: gọi minimax_decision với eval_fn=heuristic_eval để cắt ở cutoff_depth.
    Trả về action (int).
    """
    return minimax_decision(board, player, max_depth=cutoff_depth, order_actions_fn=order_actions_fn, eval_fn=heuristic_eval)
```

```python
# Task 4 experiment: kiểm tra heuristic trên các board mẫu (tức là xem heuristic đánh giá như thế nào)
samples = []
# sample 1: center occupied by player 1
s1 = empty_board(6,7)
s1[5,3] = 1
samples.append(('center', s1))
# sample 2: near-win for player 1 horizontally
s2 = empty_board(6,7)
s2[5,0]=1; s2[5,1]=1; s2[5,2]=1
samples.append(('h-almost', s2))
for name, b in samples:
    print(name, 'heuristic for player 1:', heuristic_eval(b, 1))
```

```python
def measure_time_for_size(rows, cols, depth=4):
    env = Connect4Env(rows, cols)
    b = empty_board(rows, cols)
    agent = MinimaxAgent(max_depth=depth, eval_fn=heuristic_eval)
    start = time.perf_counter()
    a = agent(b, player=1)
    return time.perf_counter()-start
for size in [(4,4),(5,5),(6,7)]:
    t = measure_time_for_size(size[0], size[1], depth=3)
    print('size', size, 'time', t)
```

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

```python
def rollout_random(b, player):
    """Thực hiện một rollout ngẫu nhiên từ trạng thái b; trả về winner."""
    env = Connect4Env(b.shape[0], b.shape[1])
    env.state = b.copy()
    env.current_player = player
    while True:
        actions = env.legal_actions()
        if not actions:
            # no legal actions -> draw
            return 0
        a = random.choice(actions)
        state, reward, done, info = env.step(a)
        if done:
            return info.get('winner', 0)
    # fallback (shouldn't reach)
    return 0

def pure_monte_carlo(b, player, n_trials=200):
    # trả về action có tỉ lệ thắng cao nhất khi thực hiện n_trials rollout cho mỗi action
    actions = valid_actions(b)
    if not actions:
        return None
    scores = {a: 0 for a in actions}
    for a in actions:
        for t in range(n_trials):
            newb = result(b, player, a)
            winner = rollout_random(newb, -player)
            if winner == player:
                scores[a] += 1
    # pick argmax (break ties randomly)
    max_score = max(scores.values())
    best_actions = [a for a, s in scores.items() if s == max_score]
    return random.choice(best_actions)

# quick test on a small board:
b = empty_board(4,4)
print('PureMC pick:', pure_monte_carlo(b, 1, n_trials=50))
```

BEST FIRST MOVE

```python
b = empty_board(6,7)
best_move = pure_monte_carlo(b, 1, n_trials=200)
print('Best first move by Pure Monte Carlo (approx):', best_move)
```