ĐẠI HỌC QUỐC GIA, THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

Nhập môn Trí tuệ Nhân tạo

BÁO CÁO BTL2

Đề tài: Game playing agent

GVHD: Vương Bá Thịnh

SV thực hiện: Phan Đình Tuấn Anh 2210118
Hồ Anh Dũng 2310543
Nguyễn Trọng Tài 2212995
Nguyễn Thiện Minh 2312097

TP Hồ Chí Minh, Tháng 5 Năm 2025

Mục lục

Danh sách thành viên & Mức độ hoàn thành

No.	Họ & tên	MSSV	Phân công	% Hoàn thành
1	Phan Đình Tuấn Anh	210118	- Hiện thực logic game	100%
2	Hồ Anh Dũng	2310543	- Hiện thực logic của Minimax Agent	100%
3	Nguyễn Trọng Tài	2212995	- Hiện thực GUI	100%
4	Nguyễn Thiện Minh	2312097	- Hiện thực logic của Minimax Agent	100%

1 Giới thiệu

1.1 Đặt vấn đề

Trí tuệ Nhân tạo (Artificial Intelligence - AI) trong lĩnh vực trò chơi (Game Playing) luôn là một chủ đề nghiên cứu hấp dẫn, với Cờ Vua là một trong những thử thách kinh điển. Độ phức tạp và không gian trạng thái khổng lồ ($\approx 10^{120}$) của Cờ Vua đòi hỏi các thuật toán tìm kiếm thông minh và hàm lượng giá hiệu quả để xây dựng một agent có khả năng chơi tốt. Bài tập lớn này tập trung vào việc triển khai một game playing agent cho Cờ Vua, sử dụng các thuật toán tìm kiếm đối kháng cổ điển làm nền tảng để xây dựng một agent đủ tốt để có thể thắng được random agent.

1.2 Mục tiêu

Bài tập lớn được thực hiện nhằm đạt được các mục tiêu chính sau:

1. **Hiện thực Engine Cờ Vua Đúng luật:** Đảm bảo agent tuân thủ đầy đủ các quy tắc di chuyển, các nước đi đặc biệt (nhập thành, bắt tốt qua đường, phong cấp) và các điều kiện kết thúc ván cờ (chiếu hết, hòa cờ).
2. **Năng lực Thi đấu Cơ bản:** Chứng minh khả năng chiến thắng tuyệt đối (10/10 ván) trước một agent chơi ngẫu nhiên (random agent).
3. **Phân cấp Độ khó:** Cho phép điều chỉnh cấp độ chơi của agent, chủ yếu thông qua việc thay đổi độ sâu tìm kiếm (search depth).
4. **Tập trung vào Thuật toán Tìm kiếm:** Toàn bộ agent được xây dựng dựa trên các thuật toán Searching truyền thống và heuristics, không sử dụng Machine Learning.

1.3 Phạm vi Báo cáo

Báo cáo này tập trung vào các khía cạnh kỹ thuật chính của bài tập lớn, bao gồm:

- Kiến trúc hệ thống và vai trò của các module.
- Chi tiết hiện thực engine Cờ Vua (`game_state.py`): biểu diễn trạng thái, quản lý luật chơi, sinh nước đi.
- Chi tiết hiện thực Minimax Agent (`minimax_ai.py`): thuật toán Minimax, cắt tỉa Alpha-Beta, các kỹ thuật tối ưu hóa tìm kiếm (Iterative Deepening, Transposition Table, Quiescence Search, Move Ordering), và hàm lượng giá (evaluation function) dựa trên heuristics.
- Kết quả thực nghiệm và đánh giá hiệu năng.

Phần giao diện người dùng (GUI) và luồng điều khiển chính được mô tả tổng quan.

2 Thiết kế và Hiện thực Chi tiết

2.1 Kiến trúc tổng thể

Các thành phần chính của chương trình và luồng tương tác dữ liệu được thể hiện trong Hình 1.

- `game_state.py`: Đây là "trái tim"của engine Cờ Vua. Module này chịu trách nhiệm hoàn toàn về việc biểu diễn trạng thái hiện tại của ván cờ (vị trí các quân cờ, lượt đi), thực thi luật chơi, sinh tất cả các nước đi hợp lệ cho người chơi hiện tại, và cập nhật trạng thái bàn cờ sau mỗi nước đi. Nó không chứa bất kỳ logic AI nào mà chỉ cung cấp một giao diện (API) cho các module khác tương tác với ván cờ.
- `minimax_ai.py`: Module này chứa toàn bộ trí tuệ của agent chơi cờ. Nó nhận trạng thái hiện tại từ `game_state.py`, sử dụng thuật toán Minimax với các kỹ thuật tối ưu hóa để tìm kiếm nước đi tốt nhất, và trả về nước đi đó cho module điều khiển. Hàm lượng giá (evaluation function) cũng được định nghĩa trong module này.
- `chess_visualizer.py`: Module này sử dụng thư viện Pygame để tạo giao diện người dùng đồ họa (GUI). Nó nhận thông tin trạng thái từ `game_state.py` để vẽ bàn cờ, quân cờ, hiển thị thông tin phụ trợ (lịch sử nước đi, quân bị bắt), và nhận input từ người dùng để thực hiện nước đi.
- `main.py`: Module điều khiển chính của ứng dụng. Nó khởi tạo các đối tượng cần thiết (GameState, ChessVisualizer, ChessAI), quản lý menu lựa chọn chế độ chơi và độ khó, xử lý sự kiện từ người dùng, và gọi các hàm tương ứng từ các module khác.

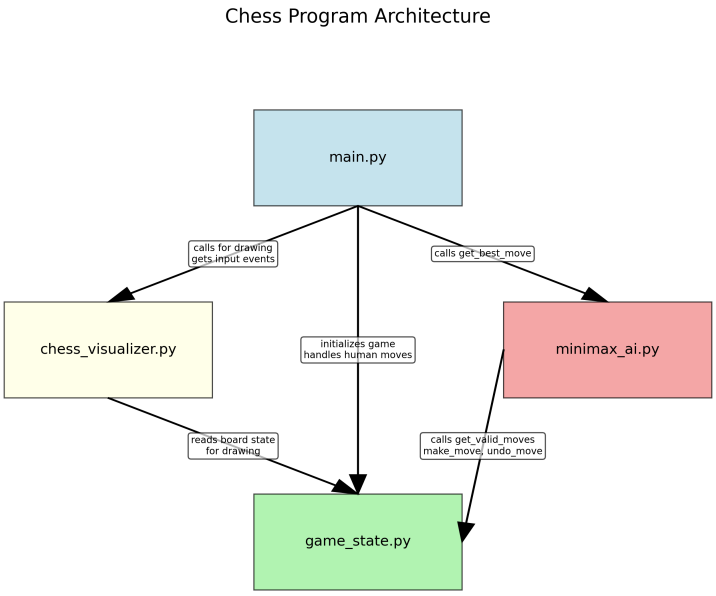
Luồng dữ liệu cơ bản khi đến lượt AI đi: `main.py` yêu cầu `minimax_ai.py` tìm nước đi dựa trên trạng thái hiện tại từ `game_state.py`. Sau khi AI trả về nước đi, `main.py` cập nhật trạng thái trong `game_state.py` và yêu cầu `chess_visualizer.py` vẽ lại bàn cờ.

2.2 Biểu diễn Trạng thái và Luật chơi (`game_state.py`)

2.2.1 Biểu diễn bàn cờ và quân cờ

Bàn cờ được biểu diễn bằng một danh sách hai chiều (list of lists) có kích thước 8x8, đặt tên là `self.board`. Mỗi phần tử trong danh sách này đại diện cho một ô trên bàn cờ và chứa một chuỗi ký tự hai chữ cái để định danh quân cờ hoặc ô trống:

- `"--"` : Biểu thị một ô trống.
- `"w"+ <ký tự quân>`: Biểu thị một quân cờ Trắng. Ví dụ: `"wp"` cho Tốt trắng (White Pawn), `"wN"` cho Mã trắng (White Knight - sử dụng 'N' để tránh trùng với King 'K'), `"wB"` cho Tượng trắng (White Bishop), `"wR"` cho Xe trắng (White Rook), `"wQ"` cho Hậu trắng (White Queen), `"wK"` cho Vua trắng (White King).



Hình 1: Sơ đồ kiến trúc tổng thể và luồng tương tác chính của chương trình.

- `"b"+ <ký tự quân>`: Biểu thị một quân cờ Đen, tương tự như quân trắng (ví dụ: `"bp"`, `"bN"`,...).

Lựa chọn biểu diễn này dựa trên sự đơn giản trong việc triển khai và debug. Mặc dù các phương pháp biểu diễn khác như bitboards có thể mang lại hiệu suất cao hơn cho các phép toán trên bàn cờ, biểu diễn bằng danh sách chuỗi đủ trực quan và hiệu quả cho mục tiêu của project này. Việc truy cập và cập nhật trạng thái của một ô cờ cụ thể (ví dụ, `self.board[row][col]`) rất dễ dàng và nhanh chóng.

2.2.2 Sinh và Kiểm tra nước đi hợp lệ

Quá trình xác định các nước đi hợp lệ cho người chơi hiện tại là một trong những chức năng phức tạp nhất của engine. Nó được thực hiện trong hàm `get_valid_moves()` và bao gồm hai giai đoạn chính:

1. **Sinh tất cả nước đi tiềm năng (Pseudo-legal Moves Generation):** Đầu tiên, hệ thống duyệt qua toàn bộ bàn cờ. Nếu một ô chứa quân cờ của người chơi có lượt đi, hàm sinh nước đi tương ứng với loại quân đó sẽ được gọi (ví dụ, `get_pawn_moves(row, col, moves)`, `get_rook_moves(row, col, moves)`,...). Các hàm này tạo ra một tập hợp tất cả các nước đi mà quân cờ đó có thể thực hiện *về mặt lý thuyết* (dựa trên quy tắc di chuyển cơ bản của nó như Tốt đi thẳng, Mã đi chữ L, Tượng đi chéo,...), không quan tâm đến việc nước đi đó có đặt Vua của mình vào thế bị chiếu hay không. Tập hợp này tạm gọi là "pseudo-legal".

2. **Kiểm tra tính hợp lệ (Legality Check):** Sau khi có tập hợp các nước đi pseudo-legal, hệ thống thực hiện kiểm tra tính hợp lệ thực sự cho từng nước đi. Đối với mỗi nước đi M trong tập hợp:
   - Thực hiện mô phỏng nước đi M trên một bản sao tạm thời của bàn cờ (hoặc thực hiện trực tiếp rồi hoàn tác - undo).
   - Sau khi M được thực hiện, kiểm tra xem Vua của người chơi vừa đi có bị đối phương chiếu hay không bằng cách gọi hàm `is_in_check()`. Hàm `is_in_check()` sẽ tạm thời chuyển lượt đi cho đối phương và sinh tất cả các nước đi pseudo-legal của họ để xem có nước đi nào tấn công vào vị trí Vua của người chơi hiện tại không.
   - Nếu Vua không bị chiếu sau khi thực hiện M, thì M được coi là một nước đi hợp lệ (legal move) và được thêm vào danh sách kết quả cuối cùng. Ngược lại, M bị loại bỏ.

   Quá trình này đảm bảo rằng agent sẽ không bao giờ thực hiện một nước đi khiến Vua của chính mình rơi vào thế bị chiếu. Đồng thời, nếu sau khi lọc mà danh sách nước đi hợp lệ rỗng, hàm `get_valid_moves()` cũng sẽ xác định xem người chơi hiện tại đang ở trạng thái Chiếu bí (checkmate - nếu Vua đang bị chiếu) hay Hòa cờ do hết nước đi (stalemate - nếu Vua không bị chiếu). Các cờ trạng thái `self.checkmate` và `self.stalemate` được cập nhật tương ứng.

2.2.3 Các nước đi đặc biệt

Việc triển khai chính xác các nước đi đặc biệt là yếu tố quan trọng để đảm bảo tính toàn vẹn của luật chơi Cờ Vua:

- **Nhập thành (Castling):** Quyền nhập thành của mỗi bên (vua và các xe tương ứng) được theo dõi trong đối tượng `self.castle_rights` (thuộc lớp `CastleRights`). Hàm `get_castle_moves(row, col, moves)` được gọi riêng cho Vua để sinh các nước nhập thành. Nó kiểm tra các điều kiện sau:
  1. Vua và Xe tham gia nhập thành chưa từng di chuyển.
  2. Vua hiện tại không bị chiếu.
  3. Tất cả các ô nằm giữa Vua và Xe phải trống.
  4. Các ô mà Vua sẽ đi qua hoặc đáp xuống không bị quân đối phương tấn công.

  Nếu tất cả các điều kiện được thỏa mãn, nước đi nhập thành (di chuyển Vua 2 ô và Xe tương ứng) sẽ được thêm vào danh sách các nước đi.

- **Bắt Tốt qua đường (En Passant):** Trạng thái về ô mà tại đó có thể thực hiện bắt Tốt qua đường được lưu trong biến `self.en_passant_possible` (dưới dạng một tuple `(row, col)`). Biến này được cập nhật mỗi khi một Tốt thực hiện nước đi đầu

tiên 2 ô. Hàm `get_pawn_moves()` khi sinh các nước đi bắt quân cho Tốt sẽ kiểm tra xem ô đích chéo có trùng với `self.en_passant_possible` hay không. Nếu có, một nước đi bắt Tốt qua đường (được đánh dấu bằng cờ `is_en_passant`) sẽ được tạo ra.

- **Phong cấp (Pawn Promotion):** Khi một Tốt di chuyển đến hàng cuối cùng của đối phương (hàng 8 cho Trắng, hàng 1 cho Đen), nước đi đó sẽ được đánh dấu bằng cờ `is_pawn_promotion`. Trong implementation hiện tại của project, Tốt luôn được phong cấp thành Hậu (Queen) để đơn giản hóa. Việc cho phép người chơi (hoặc AI) chọn loại quân để phong cấp là một cải tiến có thể thực hiện trong tương lai.

2.2.4 Quản lý trạng thái và Lịch sử

Ngoài việc biểu diễn bàn cờ, lớp `GameState` còn chịu trách nhiệm quản lý một loạt các thông tin trạng thái quan trọng khác để đảm bảo ván cờ diễn ra đúng luật và hỗ trợ các tính năng như hoàn tác nước đi (undo move) và phát hiện các điều kiện hòa cờ phức tạp:

- `self.white_to_move`: Biến boolean cho biết lượt đi hiện tại thuộc về Trắng (True) hay Đen (False).
- `self.white_king_location` và `self.black_king_location`: Lưu trữ tọa độ (hàng, cột) hiện tại của Vua Trắng và Vua Đen. Thông tin này rất quan trọng cho việc kiểm tra chiếu và nhập thành.
- `self.move_log`: Một danh sách lưu trữ tất cả các đối tượng `Move` đã được thực hiện trong ván cờ. Điều này cần thiết cho việc hiển thị lịch sử, hoàn tác nước đi, và kiểm tra lặp lại thế cờ.
- `self.castle_rights_log` và `self.en_passant_log`: Các danh sách này lưu trữ lịch sử thay đổi của quyền nhập thành và trạng thái ô en passant sau mỗi nước đi. Chúng rất quan trọng để khôi phục chính xác trạng thái khi hoàn tác nước đi.
- **Phát hiện Lặp lại Ba lần (Threefold Repetition):** Để phát hiện điều kiện hòa cờ này, một dictionary `self.position_history` được sử dụng. Key của dictionary này là một chuỗi đại diện duy nhất cho một trạng thái bàn cờ cụ thể, bao gồm vị trí các quân cờ, lượt đi hiện tại, quyền nhập thành còn lại, và ô en passant khả dĩ. Value tương ứng với mỗi key là số lần trạng thái đó đã xuất hiện. Nếu một trạng thái xuất hiện từ 3 lần trở lên, cờ `self.stalemate` (dùng chung cho các loại hòa cờ) sẽ được bật.
- **Phát hiện Thiếu quân (Insufficient Material):** Luật hòa cờ này được kiểm tra bằng hàm `is_insufficient_material()`. Hàm này đếm số lượng và loại quân còn lại trên bàn cờ (không tính Vua) để xác định các trường hợp hòa cờ rõ ràng như Vua đấu Vua, Vua và Mã đấu Vua, Vua và Tượng đấu Vua, hoặc Vua và Tượng đấu Vua và Tượng (khi các Tượng cùng màu ô).

Việc quản lý cẩn thận các thành phần trạng thái này đảm bảo engine có thể xử lý chính xác mọi tình huống trong một ván Cờ Vua.

2.3 Minimax Agent (minimax_ai.py)

Module minimax_ai.py là nơi chứa "bộ não"của agent, triển khai thuật toán tìm kiếm và logic đánh giá thế cờ.

2.3.1 Thuật toán cốt lõi: Minimax và Cắt tỉa Alpha-Beta (Alpha-Beta Pruning)

Nền tảng của agent là thuật toán **Minimax**, một phương pháp tìm kiếm đệ quy tiêu chuẩn cho các trò chơi đối kháng hai người chơi, có tổng bằng không (zero-sum game) và thông tin hoàn hảo (perfect information) như Cờ Vua. Thuật toán này hoạt động bằng cách xây dựng một cây trò chơi (game tree), trong đó mỗi node đại diện cho một trạng thái bàn cờ và các cạnh đại diện cho các nước đi. Minimax khám phá cây này đến một độ sâu nhất định (depth-limited search), giả định rằng cả hai người chơi đều sẽ chọn nước đi tối ưu tại mỗi bước. Người chơi hiện tại (Max player, trong trường hợp này là agent AI) cố gắng tối đa hóa điểm số thu được từ hàm lượng giá (evaluation function) tại các node lá, trong khi đối thủ (Min player) được giả định sẽ cố gắng tối thiểu hóa điểm số đó. Giá trị từ các node lá được truyền ngược lên cây để xác định nước đi tốt nhất tại node gốc.
Tuy nhiên, việc duyệt toàn bộ cây trò chơi Minimax, ngay cả ở độ sâu vừa phải, cũng trở nên không khả thi do sự bùng nổ tổ hợp (combinatorial explosion) của số lượng thế cờ. Để giải quyết vấn đề này, kỹ thuật tối ưu hóa **Cắt tỉa Alpha-Beta (Alpha-Beta Pruning)** được tích hợp. Alpha-Beta Pruning là một cải tiến của Minimax giúp giảm đáng kể số lượng node cần phải duyệt mà không làm thay đổi kết quả cuối cùng. Kỹ thuật này duy trì hai giá trị, *alpha* (điểm số tốt nhất mà Max player có thể đảm bảo đạt được tại hoặc phía trên node hiện tại trên đường đi từ gốc) và *beta* (điểm số tốt nhất mà Min player có thể đảm bảo đạt được tại hoặc phía trên node hiện tại).

- Tại một node Max, nếu giá trị hiện tại của một nhánh con $\geq$ beta, agent có thể dừng duyệt các nhánh con còn lại của node Max đó (gọi là beta cut-off), vì Min player sẽ không bao giờ cho phép Max player đi vào nhánh này (do Min đã có lựa chọn khác tốt hơn cho mình, mang lại giá trị $\leq$ beta).

- Tương tự, tại một node Min, nếu giá trị hiện tại của một nhánh con $\leq$ alpha, agent có thể dừng duyệt các nhánh con còn lại của node Min đó (gọi là alpha cut-off), vì Max player sẽ không bao giờ cho phép Min player đi vào nhánh này.

Bằng cách cắt tỉa các nhánh "vô ích"này, Alpha-Beta Pruning có thể giảm đáng kể không gian tìm kiếm, cho phép agent tìm kiếm sâu hơn trong cùng một khoảng thời gian. Pseudocode 1 mô tả logic cốt lõi của thuật toán Alpha-Beta Pruning, bao gồm cả việc tích hợp với các kỹ thuật tối ưu hóa khác như Transposition Table và Quiescence Search.

```
function alphabeta(node, depth, alpha, beta, maximizingPlayer):
    # 1. Transposition Table Lookup
    tt_entry = TT_lookup(node.hash) # Tra cuu TT
    if tt_entry.is_valid AND tt_entry.depth >= depth:
        if tt_entry.flag == EXACT: return tt_entry.score
```

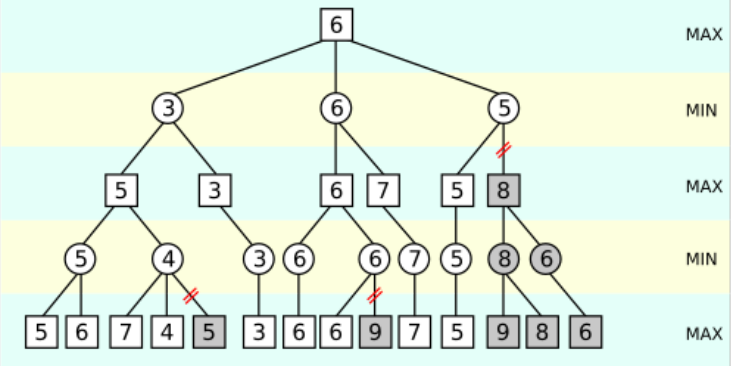
```
        if tt_entry.flag == LOWERBOUND: alpha = max(alpha, tt_entry.score)
        if tt_entry.flag == UPPERBOUND: beta = min(beta, tt_entry.score)
        if alpha >= beta: return tt_entry.score # Cat tia dua tren TT

    # 2. Base Case: Depth Limit or Terminal Node
    if depth == 0:
        return quiescence_search(node, alpha, beta) # Goi Quiescence Search
    if node is terminal_node: # Checkmate or Stalemate
        return evaluate_terminal(node) # Luong gia trang thai ket thuc

    # 3. Generate and Order Moves
    # Su dung nuoc di tot nhat tu TT (tt_entry.best_move) de uu tien sap xep
    ordered_moves = generate_ordered_moves(node, tt_entry.best_move)

    if maximizingPlayer:
        maxEval = MIN_VALUE # Gia tri toi thieu ban dau
        original_alpha = alpha # Luu alpha ban dau de xac dinh TT flag
        best_move_found_at_node = null

        for each move in ordered_moves:
            child = make_move(node, move) # Thuc hien nuoc di
            eval = alphabeta(child, depth - 1, alpha, beta, False) # De quy
            undo_move(node, move) # Hoan tac nuoc di

            if eval > maxEval:
                maxEval = eval
                best_move_found_at_node = move
            alpha = max(alpha, eval) # Cap nhat alpha

            if beta <= alpha: # Dieu kien Beta cut-off
                break # Dung duyet cac nuoc di con lai cua node nay

        # 4. Store result in Transposition Table
        flag_to_store = UPPERBOUND # Gia su ban dau
        if maxEval > original_alpha AND maxEval < beta:
```

```
    ↪ flag_to_store = EXACT
41          if maxEval >= beta: flag_to_store = LOWERBOUND
42          TT_store(node.hash, maxEval, depth, flag_to_store,
    ↪ best_move_found_at_node)
43          return maxEval
44      else: # minimizingPlayer
45          minEval = MAX_VALUE # Gia tri toi da ban dau
46          original_beta = beta # Luu beta ban dau
47          best_move_found_at_node = null
48
49          for each move in ordered_moves:
50              child = make_move(node, move) # Thuc hien nuoc di
51              eval = alphabeta(child, depth - 1, alpha, beta, True)
    ↪ # De quy
52              undo_move(node, move) # Hoan tac nuoc di
53
54              if eval < minEval:
55                  minEval = eval
56                  best_move_found_at_node = move
57              beta = min(beta, eval) # Cap nhat beta
58
59              if beta <= alpha: # Dieu kien Alpha cut-off
60                  break # Dung duyet cac nuoc di con lai cua node
    ↪ nay
61
62          # 4. Store result in Transposition Table
63          flag_to_store = LOWERBOUND # Gia su ban dau
64          if minEval > alpha AND minEval < original_beta:
    ↪ flag_to_store = EXACT
65          if minEval <= alpha: flag_to_store = UPPERBOUND
66          TT_store(node.hash, minEval, depth, flag_to_store,
    ↪ best_move_found_at_node)
67          return minEval
68
```

Listing 1: Pseudocode cho Alpha-Beta Search (Report).

2.3.2 Các Kỹ thuật Tối ưu hóa Tìm kiếm (Search Optimization Techniques)

Hiệu suất của thuật toán Alpha-Beta Pruning, mặc dù đã là một cải tiến lớn so với Minimax thuần túy, vẫn có thể được nâng cao đáng kể thông qua việc áp dụng các kỹ thuật tối ưu hóa tìm kiếm chuyên biệt. Trong lĩnh vực Chess programming, các kỹ thuật này đóng vai trò then chốt để agent có thể đạt được độ sâu tìm kiếm (search depth) đủ lớn trong một khoảng thời gian hợp lý, từ đó đưa ra các quyết định chiến thuật và chiến lược chính xác hơn. Các kỹ thuật này không thay đổi logic cốt lõi của Alpha-Beta mà tập trung vào việc



Hình 2: Minh họa nguyên lý cắt tỉa Alpha-Beta. Nguồn: Wikipedia.

giảm số lượng node trên cây trò chơi cần duyệt, quản lý thời gian tìm kiếm một cách hiệu quả hơn, và cải thiện độ chính xác của việc đánh giá trạng thái tại các node lá. Trong project này, các phương pháp tối ưu hóa chính đã được nghiên cứu và triển khai bao gồm:

2.3.2.1 Tìm kiếm Sâu dần (Iterative Deepening Search - IDS): Trong các trò chơi phức tạp với không gian trạng thái lớn như Cờ Vua, việc xác định trước một độ sâu tìm kiếm (search depth) cố định tối ưu cho mọi tình huống là rất khó khăn. Nếu độ sâu quá nông, agent có thể bỏ lỡ các đòn phối hợp hoặc mối đe dọa quan trọng. Ngược lại, nếu độ sâu quá lớn, thời gian tìm kiếm có thể trở nên không chấp nhận được. Kỹ thuật Tìm kiếm Sâu dần (Iterative Deepening Search - IDS) giải quyết vấn đề này bằng cách thực hiện một chuỗi các lượt tìm kiếm có giới hạn độ sâu (depth-limited search), bắt đầu từ độ sâu 1 và tăng dần lên (ví dụ: depth=1, sau đó depth=2, depth=3,...) cho đến khi đạt độ sâu tối đa mong muốn hoặc hết thời gian cho phép [1].

- **Khả năng "Anytime"(Anytime Algorithm):** Một ưu điểm lớn của IDS là nó có thể bị ngắt bất cứ lúc nào (ví dụ, do hết thời gian) và vẫn có thể trả về nước đi tốt nhất (best move) đã tìm được từ lượt tìm kiếm ở độ sâu hoàn chỉnh gần nhất. Điều này cực kỳ quan trọng trong các hệ thống game thực tế, nơi thường có giới hạn thời gian nghiêm ngặt cho mỗi nước đi.

- **Cải thiện Sắp xếp Nước đi (Move Ordering):** Thông tin thu được từ các lượt tìm kiếm ở độ sâu thấp hơn, đặc biệt là nước đi thuộc Biến thể Chính (Principal Variation - PV, là chuỗi các nước đi được cho là tốt nhất từ node gốc đến một node lá) và các nước đi đã gây ra cắt tỉa (cutoffs), có thể được sử dụng để sắp xếp thứ tự ưu tiên các nước đi sẽ được xem xét ở các lượt tìm kiếm sâu hơn. Việc thử các nước đi "có vẻ tốt"trước tiên giúp thuật toán Alpha-Beta Pruning tìm thấy điểm cắt tỉa sớm hơn và hiệu quả hơn, qua đó giảm đáng kể tổng số node cần duyệt. Các nghiên cứu và thực nghiệm trong lĩnh vực Chess programming cho thấy, ngay cả khi mục tiêu là tìm kiếm đến một độ sâu cố định, việc sử dụng IDS thường nhanh hơn so với tìm kiếm trực tiếp đến độ sâu đó chính nhờ vào việc cải thiện move ordering một cách động wiki$_{i}terative, Dom_{W}ilson$**Quản lý Thời gian (Time Management):**$IDSchophpphnbthigianmtcch$

Trong implementation của agent này, hàm `get_best_move()` trong module `minimax_ai.py` là nơi quản lý vòng lặp Iterative Deepening. Hàm này gọi hàm `alphabeta_root()` lặp đi lặp lại với tham số `depth` tăng dần. Nước đi tốt nhất tổng thể ('best_move_overall') được cập nhật sau mỗi lần hoàn thành một độ sâu tìm kiếm. Mặc dù implementation hiện tại chưa khai thác triệt để thông tin từ PV của lượt tìm kiếm trước để thực hiện sắp xếp nước đi một cách chi tiết cho lượt tìm kiếm sau (ví dụ, lưu trữ toàn bộ PV và các nước đi phản bác - refutation moves), việc có được 'best_move_overall' từ độ sâu $d-1$ cũng cung cấp một gợi ý ban đầu quan trọng cho việc bắt đầu tìm kiếm ở độ sâu d. Pseudocode 2 minh họa logic này.

```
function iterative_deepening(root_node, time_limit_per_move):
    best_move_found_overall = null
    data_from_previous_iteration = null // Luu thong tin nhu PV, scores

    for current_search_depth = 1 to MAX_POSSIBLE_DEPTH:
        iteration_start_time = get_current_time()

        // Goi ham tim kiem goc voi do sau hien tai
        // Truyen data_from_previous_iteration de ho tro sap xep nuoc di
        current_best_move_this_depth, current_score, info_this_search =
            alphabeta_root_search(root_node, current_search_depth, data_from_previous_iteration)

        // Kiem tra gioi han thoi gian SAU khi hoan thanh mot do sau (hoac trong luc tim kiem)
        if time_limit_exceeded(iteration_start_time, time_limit_per_move):
            // Neu timeout xay ra trong luc tim kiem o do sau current_search_depth,
            // current_best_move_this_depth co the khong dang tin cay.
            // Trong truong hop do, nen dung best_move_found_overall tu do sau HOAN THANH truoc do.
            break // Thoat khoi vong lap IDS

        // Neu tim kiem cho do sau hien tai hoan thanh trong thoi gian:
        best_move_found_overall = current_best_move_this_depth
        data_from_previous_iteration = info_this_search // Cap nhat thong tin cho lan lap sau

        // Tuy chon: Kiem tra diem chieu bi de dung som
```

```
        if is_checkmate_score(current_score):
            break

    return best_move_found_overall // Tra ve nuoc di tot nhat tu
↪ do sau cuoi cung da hoan thanh

```

Listing 2: Pseudocode cho Tìm kiếm Sâu dần (Iterative Deepening Search).

2.3.2.2 Bảng Băm (Transposition Table - TT): Trong Cờ Vua, rất nhiều thế cờ (positions) có thể đạt được thông qua các chuỗi nước đi khác nhau (gọi là các phép hoán vị - transpositions). Việc tính toán lại giá trị cho các thế cờ giống hệt nhau là một sự lãng phí tài nguyên tính toán. Bảng Băm (Transposition Table - TT) là một cấu trúc dữ liệu, thường được triển khai dưới dạng một hash table, được sử dụng để lưu trữ (cache) kết quả đánh giá của các trạng thái đã được duyệt qua trong quá trình tìm kiếm [2].

- **Lưu trữ (Store):** Khi một node trong cây tìm kiếm được đánh giá xong (tức là giá trị minimax của nó đã được xác định hoặc một cận (bound) đã được tìm thấy), thông tin về node đó sẽ được lưu vào TT. Mỗi entry trong TT thường bao gồm:
  - *Khóa băm (Hash Key):* Một giá trị hash (thường là Zobrist hash trong các engine hiện đại) đại diện gần như duy nhất cho trạng thái bàn cờ (bao gồm vị trí quân cờ, lượt đi, quyền nhập thành, ô en passant tiềm năng).
  - *Điểm số lượng giá (Score):* Giá trị minimax của node, hoặc một cận của giá trị đó.
  - *Độ sâu (Depth):* Độ sâu tìm kiếm còn lại tại node đó khi nó được lưu. Thông tin này quan trọng để xác định xem entry có đủ tốt để sử dụng hay không.
  - *Cờ (Flag):* Cho biết loại điểm số được lưu: EXACT (điểm số là chính xác, node đã được duyệt hoàn toàn và giá trị nằm trong khoảng [alpha, beta] ban đầu), LOWERBOUND (điểm số $\geq$ giá trị thực, thường xảy ra do beta cut-off, giá trị trả về là beta), hoặc UPPERBOUND (điểm số $\leq$ giá trị thực, thường xảy ra do alpha cut-off, giá trị trả về là alpha).
  - *Nước đi tốt nhất (Best Move / Hash Move):* Nước đi tốt nhất được tìm thấy từ trạng thái này, nếu có.
- **Tra cứu (Probe):** Trước khi bắt đầu duyệt sâu một node, agent tính Zobrist hash (hoặc key tương đương) của trạng thái hiện tại và thực hiện tra cứu trong TT. Nếu tìm thấy một entry khớp (thường là Zobrist key giống nhau) và độ sâu đã lưu trong entry đó đủ lớn (thường là $\geq$ độ sâu tìm kiếm hiện tại cần thực hiện):
  - Nếu flag là EXACT, agent có thể trả về ngay score đã lưu mà không cần duyệt tiếp nhánh con đó.

- Nếu flag là LOWERBOUND, score đã lưu có thể được dùng để cập nhật (tăng) giá trị alpha hiện tại của cửa sổ tìm kiếm. Nếu alpha mới $\geq$ beta, có thể xảy ra beta cut-off.
- Nếu flag là UPPERBOUND, score đã lưu có thể được dùng để cập nhật (giảm) giá trị beta hiện tại. Nếu alpha $\geq$ beta mới, có thể xảy ra alpha cut-off.

- **Cải thiện Sắp xếp Nước đi:** Ngay cả khi entry trong TT không đủ sâu để gây cắt tỉa trực tiếp, nước đi tốt nhất (hash move) được lưu trong entry đó vẫn là một ứng cử viên rất mạnh để thử đầu tiên khi duyệt các nước đi con. Việc này thường dẫn đến việc tìm thấy nhánh tốt nhất (hoặc nhánh gây cắt tỉa) sớm hơn, đóng góp đáng kể vào hiệu quả của Alpha-Beta Pruning [3]. Trong các engine mạnh, phần lớn các beta-cutoff đến từ việc thử hash move đầu tiên.

Trong project này, `self.transposition_table` được triển khai dưới dạng một Python dictionary đơn giản. Khóa băm được tạo bởi hàm `_get_position_key()`, là một chuỗi ký tự đại diện cho toàn bộ trạng thái trò chơi (bàn cờ, lượt đi, quyền nhập thành, ô en passant). Các thông tin về score, depth, flag (được định nghĩa là các hằng số `TT_EXACT`, `TT_LOWERBOUND`, `TT_UPPERBOUND`), và best move được lưu trữ trong một tuple hoặc một đối tượng nhỏ. Kết quả thực nghiệm trình bày ở Mục 3.4 cho thấy số TT hits (số lần tra cứu thành công và sử dụng được thông tin từ TT) tăng đáng kể ở độ sâu tìm kiếm 4 và 5. Điều này minh chứng cho hiệu quả của kỹ thuật này trong việc giảm thiểu tính toán lặp lại, đặc biệt khi không gian tìm kiếm trở nên rộng lớn hơn ở các độ sâu lớn.

2.3.2.3 Tìm kiếm Tĩnh lặng (Quiescence Search - QS): Một trong những vấn đề cố hữu của các thuật toán tìm kiếm có độ sâu giới hạn là "hiệu ứng đường chân trời"(horizon effect) [4]. Hiện tượng này xảy ra khi việc dừng tìm kiếm một cách đột ngột tại một độ sâu cố định (ví dụ, depth=0) có thể khiến agent bỏ lỡ các diễn biến chiến thuật quan trọng như một chuỗi bắt quân liên tiếp, một nước phong cấp, hoặc một nước chiếu hết ngay sau đó. Điều này dẫn đến việc hàm lượng giá được áp dụng trên một trạng thái "động"(volatile) hoặc "không ổn định"như vậy, cho kết quả đánh giá thiếu chính xác và dẫn đến các quyết định sai lầm. Ví dụ, agent có thể thấy mình hơn một Tốt sau một nước bắt quân, nhưng không nhìn thấy rằng đối phương có thể bắt lại quân vừa bắt Tốt đó ngay nước sau, làm thay đổi hoàn toàn cục diện. Quiescence Search (QS) là một kỹ thuật được thiết kế để giải quyết vấn đề này [5].

- **Nguyên lý hoạt động:** Khi tìm kiếm Alpha-Beta chính đạt đến độ sâu 0 (tức là node lá theo tiêu chí độ sâu ban đầu), thay vì gọi hàm lượng giá `evaluate_board()` ngay lập tức, agent sẽ kích hoạt một lượt tìm kiếm phụ, gọi là Quiescence Search. Trong lượt tìm kiếm này, agent không duyệt tất cả các nước đi hợp lệ mà chỉ xem xét và duyệt đệ quy một tập hợp giới hạn các nước đi được coi là "ồn ào"(noisy moves) – thường là các nước bắt quân (captures), phong cấp (promotions), và đôi khi là các nước chiếu (checks) hoặc các nước thoát chiếu (check evasions).

- **Mục tiêu:** Mục tiêu của QS là tiếp tục tìm kiếm các chuỗi nước đi ồn ào này cho đến khi đạt được một trạng thái "tĩnh lặng"(quiescent state). Trạng thái tĩnh lặng là trạng thái mà tại đó không còn nước đi ồn ào nào có khả năng làm thay đổi đáng kể điểm số của thế cờ trong lượt đi ngay sau đó. Chỉ tại các trạng thái tĩnh lặng này, hàm lượng giá chính mới được áp dụng.

- **Cải thiện độ chính xác đánh giá:** Bằng cách "nhìn"qua các chuỗi chiến thuật ngắn hạn, QS đảm bảo rằng điểm số lượng giá phản ánh một cách chính xác hơn giá trị thực của vị trí, thay vì bị đánh lừa bởi một lợi thế hoặc bất lợi vật chất tạm thời chưa được giải quyết hoàn toàn.

- **Pruning trong QS:** Vì QS có thể mở rộng cây tìm kiếm khá sâu nếu có nhiều lựa chọn bắt quân liên tiếp, việc kiểm soát độ sâu của nó là cần thiết. Các kỹ thuật pruning như Đánh giá Trao đổi Tĩnh (Static Exchange Evaluation - SEE), giúp xác định xem một chuỗi bắt quân có thực sự lợi về vật chất hay không trước khi duyệt, hoặc Delta Pruning (bỏ qua các nước bắt quân mà giá trị quân bị bắt không đủ lớn để có khả năng cải thiện cận alpha của người chơi hiện tại) thường được áp dụng để hạn chế sự bùng nổ của QS.

Trong implementation của project này, hàm `quiescence_search()` được gọi tại các node lá (depth=0) của tìm kiếm Alpha-Beta chính. Nó bắt đầu bằng việc tính toán điểm số "stand-pat"(điểm số của trạng thái hiện tại nếu không thực hiện thêm nước đi ồn ào nào). Sau đó, nó chỉ sinh và duyệt các nước bắt quân (captures) và phong cấp (promotions). Quá trình này tiếp tục một cách đệ quy, sử dụng cùng cửa sổ alpha-beta của tìm kiếm chính, nhưng thường với một giới hạn độ sâu bổ sung (ví dụ, chỉ tìm kiếm thêm 2-3 nước bắt quân) để tránh tìm kiếm vô hạn trong các tình huống phức tạp. Hiện tại, implementation này chưa tích hợp đầy đủ SEE hoặc Delta Pruning trong Quiescence Search, đây là một điểm có thể được cải thiện trong các phiên bản nâng cấp. Tuy nhiên, ngay cả với việc chỉ xét captures và promotions, QS đã góp phần đáng kể vào việc cải thiện tính ổn định và sức mạnh chiến thuật của agent. Điều này được phản ánh qua số lượng lớn QNodes (nút được duyệt trong Quiescence Search) được ghi nhận trong kết quả thực nghiệm (Mục 3.4). Pseudocode 3 mô tả logic hoạt động của QS.

```
function quiescence_search(node, alpha, beta, q_depth_limit): //
    ↪ q_depth_limit de tranh tim kiem vo han
    // Tuy chon: Tra cuu TT cho QS
    // tt_entry = TT_lookup_quiescence(node.hash) ...

    stand_pat_score = evaluate(node) // Luong gia trang thai "tinh
    ↪  lang" hien tai

    if stand_pat_score >= beta:
        return beta // Cat tia (Fail-high)
    if node.is_maximizing_player:
        alpha = max(alpha, stand_pat_score)
```

```
    // else if node.is_minimizing_player and stand_pat_score < alpha: // Dieu kien fail-low
    // return alpha (cat tia)

    if q_depth_limit <= 0: // Dat gioi han do sau cua QS
        return stand_pat_score

    // Sinh cac nuoc di "on ao" (captures, promotions) va sap xep chung (vi du: MVV-LVA)
    noisy_moves = generate_ordered_noisy_moves(node)

    for each move in noisy_moves:
        // Tuy chon: Ap dung cac ky thuat pruning nhu SEE > 0 hoac Delta Pruning o day
        // if simple_static_exchange_evaluation(move) < SAFETY_MARGIN AND not move.is_promotion:
        // continue // Bo qua nuoc bat quan ro rang la bat loi

        child = make_move(node, move)
        score = quiescence_search(child, alpha, beta, q_depth_limit - 1) // Goi de quy QS
        undo_move(node, move) // Hoan tac

        if node.is_maximizing_player:
            alpha = max(alpha, score)
            if alpha >= beta:
                // Tuy chon: Luu vao TT (voi co LOWERBOUND)
                return beta // Cat tia
        else: // Minimizing player
            beta = min(beta, score)
            if alpha >= beta:
                // Tuy chon: Luu vao TT (voi co UPPERBOUND)
                return alpha // Cat tia

    // Tra ve gia tri tot nhat tim duoc trong pham vi alpha-beta
    return alpha if node.is_maximizing_player else beta

```

Listing 3: Pseudocode cho Tìm kiếm Tĩnh lặng (Quiescence Search).

2.3.2.4 Sắp xếp Nước đi (Move Ordering) và MVV-LVA: Hiệu suất của thuật toán Alpha-Beta Pruning phụ thuộc rất lớn vào thứ tự mà các nước đi (children nodes) được duyệt tại mỗi node trong cây tìm kiếm [6]. Nếu các nước đi "tốt nhất"(những nước đi có khả năng dẫn đến cắt tỉa hoặc có giá trị minimax cao/thấp nhất cho người chơi hiện tại) được xem xét trước, thì số lượng node cần duyệt có thể giảm đi một cách đáng kể. Do

đó, việc áp dụng các heuristics (phương pháp phỏng đoán) để sắp xếp nước đi một cách thông minh là cực kỳ quan trọng.

- **Các nguồn ưu tiên trong Sắp xếp Nước đi:** Trong một engine cờ vua hiện đại, việc sắp xếp nước đi thường tuân theo một thứ tự ưu tiên phức tạp, kết hợp nhiều nguồn thông tin khác nhau. Các nguồn thông tin này giúp định hướng tìm kiếm vào những nhánh có khả năng cao chứa nước đi tốt nhất hoặc gây ra cắt tỉa sớm. Một số nguồn ưu tiên phổ biến bao gồm:

  1. *Nước đi từ Bảng Băm (Hash Move):* Nếu Transposition Table trả về một nước đi tốt nhất đã được tìm thấy trước đó cho trạng thái hiện tại (thường từ một lượt tìm kiếm ở độ sâu tương đương hoặc lớn hơn), nước đi này có xác suất cao là nước đi mạnh nhất và nên được thử đầu tiên.
  2. *Nước đi bắt quân tốt (Good/Winning Captures):* Các nước đi bắt quân mà được đánh giá sơ bộ là có lợi về vật chất. Thường được sắp xếp bằng các heuristic như MVV-LVA hoặc được lọc qua Đánh giá Trao đổi Tĩnh (Static Exchange Evaluation - SEE) để loại bỏ những nước bắt quân rõ ràng là bất lợi.
  3. *Nước đi phong cấp (Promotions):* Đặc biệt là phong cấp Tốt thành Hậu, vì nó làm thay đổi cán cân vật chất một cách đáng kể.
  4. *Nước đi "Sát thủ"(Killer Moves):* Là các nước đi không bắt quân nhưng đã gây ra beta cut-off ở cùng một độ sâu trong các nhánh anh em (sibling nodes) đã được duyệt trước đó trong cây tìm kiếm. Ý tưởng là nếu một nước đi đã tốt ở một nhánh tương tự, nó cũng có thể tốt ở nhánh hiện tại. Thường mỗi độ sâu sẽ lưu trữ một hoặc hai killer moves.
  5. *Heuristic Lịch sử (History Heuristic):* Gán điểm cho các cặp (quân cờ, ô đến) dựa trên tần suất chúng đã gây ra cắt tỉa hoặc cải thiện đáng kể giá trị alpha trong quá khứ của quá trình tìm kiếm hiện tại. Các nước đi "thành công"trong lịch sử được ưu tiên hơn.
  6. *Nước đi bắt quân không tốt (Bad/Losing Captures):* Các nước đi bắt quân nhưng bị SEE đánh giá là bất lợi, thường được xem xét sau cùng trong số các nước bắt quân.
  7. *Các nước đi còn lại (Quiet Moves):* Các nước đi không bắt quân, không phong cấp khác, thường được sắp xếp dựa trên các heuristic vị trí hoặc giá trị tĩnh của quân di chuyển.

- **MVV-LVA (Most Valuable Victim - Least Valuable Attacker):** Đây là một heuristic đơn giản nhưng rất hiệu quả để sắp xếp ưu tiên giữa các nước đi bắt quân [7]. Nguyên tắc cơ bản là ưu tiên các nước đi bắt được quân cờ có giá trị cao nhất của đối phương (Most Valuable Victim - Nạn nhân Giá trị nhất) bằng quân cờ có giá trị thấp nhất của mình (Least Valuable Attacker - Kẻ tấn công Giá trị thấp nhất). Ví dụ, Tốt bắt Hậu (Pawn captures Queen) sẽ được đánh giá cao hơn Mã bắt Hậu (Knight captures Queen), và cả hai đều được đánh giá cao hơn Hậu bắt Tốt (Queen

captures Pawn). Điều này giúp agent nhanh chóng khám phá các trao đổi vật chất có lợi và có khả năng tìm thấy các nước đi chiến thuật mạnh sớm hơn.

Trong implementation của project này, hàm `order_moves()` thực hiện việc sắp xếp nước đi. Mặc dù chưa triển khai đầy đủ tất cả các nguồn ưu tiên phức tạp như Killer Moves hay History Heuristic, nó tập trung vào các yếu tố quan trọng sau: Ưu tiên hàng đầu là các nước bắt quân, và điểm số của chúng được tính dựa trên một biến thể của MVV-LVA: `Score = (Giá trị quân bị bắt * 10) - Giá trị quân bắt + Bonus cố định lớn`. Nước đi phong cấp Tốt (thành Hậu) cũng được cộng điểm thưởng cao để được ưu tiên. Các nước đi sau đó được sắp xếp giảm dần theo điểm số này trước khi đưa vào vòng lặp tìm kiếm Alpha-Beta. Việc áp dụng MVV-LVA và ưu tiên promotions đã giúp agent tập trung vào các biến thể chiến thuật có khả năng thay đổi cục diện nhanh chóng, từ đó tăng cường hiệu quả của quá trình tìm kiếm. Pseudocode 4 minh họa logic cơ bản của việc tính điểm và sắp xếp nước đi.

```
function calculate_move_score_for_ordering(move,
    ↪ hash_move_candidate):
    score = 0 // Diem mac dinh cho cac nuoc di thong thuong

    // 1. Uu tien cao nhat cho Hash Move (neu co va hop le)
    if move == hash_move_candidate:
        score = 20000 // Diem rat cao de dam bao duoc thu dau tien
        return score

    // 2. Uu tien cho nuoc Phong cap (Promotion), dac biet la
    ↪ phong Hau
    if move.is_promotion:
        if move.promoted_piece_type == QUEEN:
            score = 15000 + get_piece_value(QUEEN) // Vi du: 15000
    ↪  + 900
        else: // Phong cap thanh cac quan khac it gia tri hon
            score = 14000 + get_piece_value(move.
    ↪ promoted_piece_type)
        return score // Phong cap thuong rat manh, xet ngay sau
    ↪ hash move

    // 3. Uu tien cho nuoc Bat quan (Captures), sap xep bang MVV-
    ↪ LVA
    if move.is_capture:
        victim_value = get_piece_value(move.captured_piece)
        attacker_value = get_piece_value(move.moving_piece)
        // MVV-LVA: Quan gia tri cao bi bat boi quan gia tri thap
    ↪ se co diem cao
        // Nhan victim_value voi he so lon hon de phan biet ro ret
        score = 10000 + (victim_value * 10) - attacker_value
```

```
        // Vi du: Pawn x Queen (wp, bq) -> 10000 + (900*10) - 100 = 18900
        // Queen x Pawn (wq, bp) -> 10000 + (100*10) - 900 = 10100
        return score

    // 4. Tuy chon: Uu tien cho Killer Moves (neu co)
    // if is_killer_move(move, current_depth):
    //     score = 8000 + killer_move_priority_score // Killer moves thuong co do uu tien cao

    // 5. Tuy chon: Uu tien dua tren History Heuristic (neu co)
    // score += get_history_heuristic_score(move.from_square, move.to_square, move.piece_type)
    // Vi du: history_score co the tu 0 den 7000

    // Cac nuoc di khong bat quan, khong phong cap khac se co diem thap hon (dua tren history hoac mac dinh)
    return score

// Ham sap xep nuoc di chinh
function generate_ordered_moves(current_node_state, best_move_from_tt):
    all_currently_legal_moves = generate_all_legal_moves(current_node_state)
    scored_moves_list = []

    for each_legal_move in all_currently_legal_moves:
        # Pass best_move_from_tt to prioritize it if it's among legal_moves
        move_ordering_score = calculate_move_score_for_ordering(each_legal_move, best_move_from_tt)
        scored_moves_list.append( (move_ordering_score, each_legal_move) )

    // Sap xep danh sach cac nuoc di giam dan theo diem so
    sort scored_moves_list in descending order based on score

    // Tra ve danh sach cac nuoc di da duoc sap xep
    return [move_object for score, move_object in scored_moves_list]

```

Listing 4: Pseudocode Sắp xếp Nước đi với MVV-LVA và Ưu tiên Hash Move (Report).

2.3.3 Hàm Lượng giá và Heuristics (`evaluate_board()`)

Hàm lượng giá (`evaluate_board()`) là một thành phần cực kỳ quan trọng của Minimax Agent, vì nó cung cấp một giá trị định lượng (score) cho một trạng thái bàn cờ tĩnh (static position). Điểm số này được thuật toán tìm kiếm sử dụng để so sánh và lựa chọn giữa các nhánh khác nhau của cây trò chơi. Một hàm lượng giá tốt cần phải nắm bắt được các yếu tố chiến lược và chiến thuật quan trọng của Cờ Vua, đồng thời phải đủ nhanh để không làm chậm đáng kể quá trình tìm kiếm. Điểm số được tính từ góc độ của người chơi Trắng (điểm dương lợi cho Trắng, điểm âm lợi cho Đen). Các thành phần chính (heuristics) của hàm lượng giá trong project này bao gồm:

2.3.3.1 Giá trị quân cờ (Material Balance): Đây là yếu tố cơ bản và thường là quan trọng nhất trong hầu hết các hàm lượng giá Cờ Vua. Mỗi loại quân cờ được gán một giá trị điểm số tương đối dựa trên sức mạnh trung bình của nó:

- Tốt (Pawn): 100 điểm
- Mã (Knight): 320 điểm
- Tượng (Bishop): 330 điểm (thường được đánh giá nhỉnh hơn Mã một chút do khả năng kiểm soát đường chéo dài)
- Xe (Rook): 500 điểm
- Hậu (Queen): 900 điểm
- Vua (King): Giá trị của Vua thường được coi là vô hạn hoặc một số rất lớn (ví dụ, 20000 điểm trong implementation này) để thể hiện rằng việc mất Vua (chiếu hết) đồng nghĩa với thua cuộc.

Điểm cân bằng vật chất (Material Balance) được tính bằng cách lấy tổng giá trị quân cờ của Trắng trừ đi tổng giá trị quân cờ của Đen. `Score = WhiteMaterial - BlackMaterial`.

2.3.3.2 Bảng Vị trí Quân cờ (Piece-Square Tables - PST): Ngoài giá trị vật chất, vị trí của quân cờ trên bàn cũng ảnh hưởng lớn đến sức mạnh của nó. Piece-Square Tables (PST) là các bảng 64 phần tử (tương ứng 64 ô cờ), định nghĩa một giá trị thưởng hoặc phạt (bonus/penalty) cho từng loại quân (riêng cho quân Trắng và quân Đen) khi nó đứng trên một ô cụ thể [8, 9]. Các giá trị này thường được thiết kế để mã hóa các nguyên tắc chiến lược cơ bản về vị trí quân cờ [10]:

- Khuyến khích đặt Mã (Knights) và Tượng (Bishops) ở các vị trí trung tâm, nơi chúng có tầm hoạt động và kiểm soát tốt hơn. Phạt điểm nếu chúng ở các vị trí rìa bàn cờ.
- Khuyến khích Tốt (Pawns) tiến lên, đặc biệt là các Tốt trung tâm, để kiểm soát không gian và chuẩn bị cho việc phong cấp.

- Đối với Vua (King), vai trò vị trí thay đổi theo giai đoạn ván cờ. Trong giai đoạn khai cuộc và trung cuộc (middlegame), Vua cần được đặt ở vị trí an toàn, thường là sau một hàng Tốt bảo vệ (sau khi nhập thành). Do đó, PST cho Vua ở giai đoạn này sẽ thưởng điểm cho các vị trí an toàn. Ngược lại, trong giai đoạn tàn cuộc (endgame), khi số lượng quân cờ trên bàn đã giảm nhiều, Vua trở thành một quân cờ tấn công và phòng thủ tích cực, nên PST sẽ khuyến khích Vua di chuyển vào trung tâm.

Implementation này sử dụng các bảng PST riêng biệt cho từng loại quân (Pawn, Knight, Bishop, Rook, Queen, King). Đối với Vua, có hai bảng riêng biệt (`KING_MIDDLE_TABLE`, `KING_END_TABLE`) và việc chọn bảng nào được quyết định dựa trên giai đoạn hiện tại của ván cờ (ước lượng thông qua tổng số quân còn lại trên bàn). Các bảng PST cho quân Đen là phiên bản đối xứng (mirrored) của bảng cho quân Trắng. Tổng điểm từ PST của tất cả các quân cờ trên bàn được cộng dồn vào điểm số lượng giá tổng thể. Bảng 1 là một ví dụ về PST cho quân Mã trắng.

Bảng 1: Bảng vị trí quân Mã Trắng (Knight Table) - Ví dụ

-50	-40	-30	-30	-30	-30	-40	-50
-40	-20	0	0	0	0	-20	-40
-30	0	10	15	15	10	0	-30
-30	5	15	20	20	15	5	-30
-30	0	15	20	20	15	0	-30
-30	5	10	15	15	10	5	-30
-40	-20	0	5	5	0	-20	-40
-50	-40	-30	-30	-30	-30	-40	-50

2.3.3.3 Cấu trúc Tốt (Pawn Structure): Cấu trúc của các quân Tốt có ảnh hưởng sâu sắc đến cục diện chiến lược của ván cờ. Một cấu trúc Tốt tốt có thể tạo ra các tiền đồn (outposts) cho các quân khác, kiểm soát các ô quan trọng, và bảo vệ Vua. Ngược lại, các điểm yếu trong cấu trúc Tốt có thể bị đối phương khai thác. Hàm lượng giá xem xét các yếu tố sau:

- *Tốt chồng (Doubled Pawns):* Hai (hoặc nhiều hơn) Tốt của cùng một bên đứng trên cùng một cột. Các Tốt này thường kém linh hoạt và khó bảo vệ lẫn nhau. Agent bị phạt một số điểm nhất định cho mỗi cặp Tốt chồng (`DOUBLED_PAWN_PENALTY`).

- *Tốt cô lập (Isolated Pawns):* Một Tốt không có Tốt đồng minh trên các cột liền kề (trái và phải). Tốt cô lập không thể được bảo vệ bởi Tốt khác và ô phía trước nó thường trở thành một điểm yếu. Agent bị phạt điểm cho mỗi Tốt cô lập (`ISOLATED_PAWN_PENALTY`).

- *Tốt thông (Passed Pawns):* Một Tốt không còn Tốt đối phương nào cản đường trên cùng cột và các cột liền kề trên đường tiến tới hàng phong cấp. Tốt thông là một vũ khí chiến lược cực kỳ mạnh mẽ, đặc biệt trong tàn cuộc. Agent được thưởng điểm đáng kể cho mỗi Tốt thông, với điểm thưởng tăng lên khi Tốt càng tiến gần đến hàng cuối cùng (dựa trên mảng `PASSED_PAWN_BONUS`).

Việc phát hiện các đặc điểm này được thực hiện bằng cách phân tích vị trí của các Tốt trên bàn cờ so với các Tốt khác (cả bạn và thù).

2.3.3.4 An toàn Vua (King Safety): Đảm bảo an toàn cho Vua là một yếu tố tối quan trọng, đặc biệt trong giai đoạn khai cuộc và trung cuộc khi bàn cờ còn nhiều quân có khả năng tấn công. Một Vua bị lộ (exposed) có thể dễ dàng trở thành mục tiêu của các đòn phối hợp chiến thuật.

- *Lá chắn Tốt (Pawn Shield):* Implementation hiện tại tập trung vào việc thưởng điểm cho các Tốt đứng ở các vị trí che chắn phía trước Vua, đặc biệt là sau khi Vua đã nhập thành. Ví dụ, nếu Vua Trắng nhập thành cánh Vua (kingside), các Tốt ở f2, g2, h2 (hoặc f3, g3, h3 nếu đã tiến) đóng vai trò quan trọng. Agent được cộng một số điểm nhỏ (`KING_SAFETY_PAWN_SHIELD_BONUS`) cho mỗi Tốt tham gia vào lá chắn này.
- *Các yếu tố khác (chưa triển khai chi tiết):* Các đánh giá King Safety phức tạp hơn có thể bao gồm việc phạt điểm nếu có các cột mở (open files) hoặc đường chéo mở (open diagonals) hướng về Vua, hoặc khi có nhiều quân mạnh của đối phương đang nhắm vào khu vực quanh Vua. Những yếu tố này chưa được triển khai chi tiết trong project này nhưng là hướng cải tiến quan trọng.

2.3.3.5 Kiểm soát Trung tâm (Center Control): Kiểm soát các ô trung tâm của bàn cờ (thường là d4, e4, d5, e5 và các ô lân cận) mang lại lợi thế chiến lược lớn vì các quân cờ đặt ở trung tâm thường có tầm hoạt động và khả năng di chuyển tốt hơn đến các khu vực khác của bàn cờ. Agent được thưởng điểm (`CENTER_CONTROL_BONUS`) cho việc các quân cờ (đặc biệt là Tốt, Mã, Tượng) chiếm giữ hoặc tấn công các ô trung tâm này.

2.3.3.6 Tổng hợp điểm số lượng giá: Điểm lượng giá cuối cùng của một trạng thái bàn cờ được tính bằng tổng có trọng số của tất cả các thành phần heuristic đã phân tích ở trên:

$$\text{Score} = \text{Material} + w_1 \times \text{PST} + w_2 \times \text{PawnStructure} + w_3 \times \text{KingSafety} + w_4 \times \text{CenterControl}$$

Các trọng số w_i được điều chỉnh thông qua quá trình thử nghiệm và quan sát để cân bằng tầm quan trọng tương đối của mỗi yếu tố, đảm bảo agent đưa ra các quyết định cân bằng giữa vật chất và vị trí.

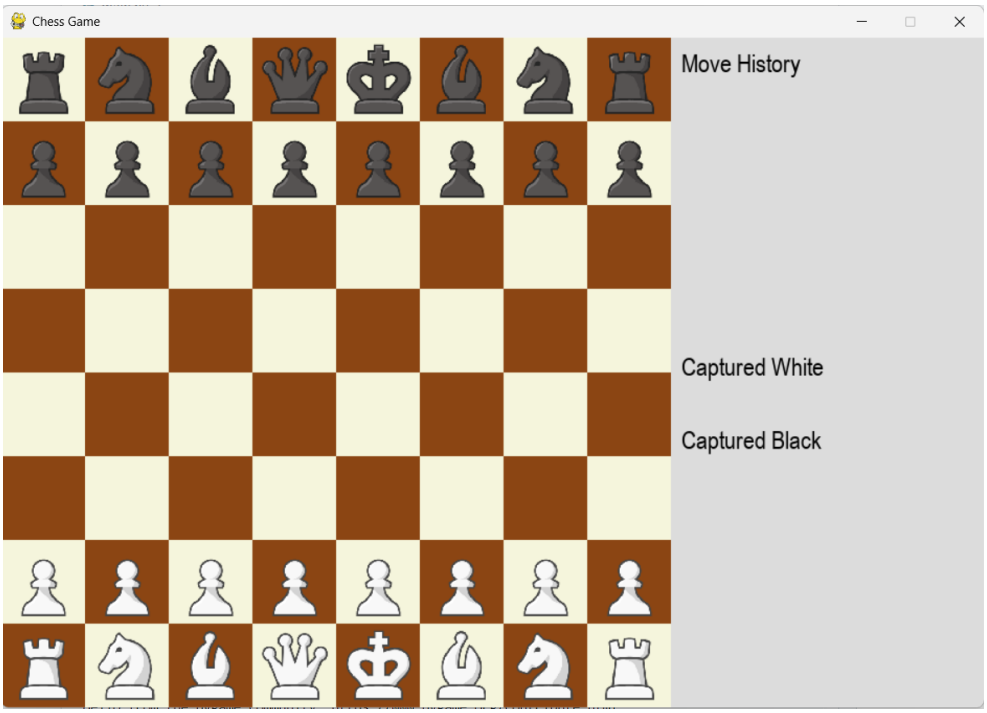
2.4 Giao diện và Luồng chính (`chess_visualizer.py`, `main.py`)

Mặc dù không phải là trọng tâm chính của project về mặt thuật toán AI, việc xây dựng một giao diện người dùng (Graphical User Interface - GUI) và một luồng điều khiển chính là cần thiết để người dùng có thể tương tác với agent và để kiểm thử hệ thống.

- **Giao diện Người dùng (`chess_visualizer.py`):** Module này sử dụng thư viện **Pygame** để tạo ra một giao diện đồ họa trực quan. Các chức năng chính bao gồm:

- Vẽ bàn cờ 8x8 với các ô màu xen kẽ.
- Tải và hiển thị hình ảnh của các quân cờ trên các ô tương ứng.
- Xử lý input từ chuột của người dùng: cho phép người chơi chọn một quân cờ (bằng cách click vào ô chứa quân đó) và sau đó chọn ô đích để thực hiện nước đi.
- Hiển thị các nước đi hợp lệ cho quân cờ được chọn (ví dụ, bằng cách tô sáng các ô đích khả thi).
- Hiển thị một bảng điều khiển phụ (panel) chứa thông tin như lịch sử các nước đi đã thực hiện, danh sách các quân cờ đã bị bắt bởi mỗi bên.
- Hiển thị các thông báo trạng thái của ván cờ, chẳng hạn như chiếu (check), chiếu hết (checkmate), hòa cờ (stalemate/draw), và người chiến thắng.
- Thực hiện hiệu ứng hình ảnh đơn giản cho việc di chuyển quân cờ (animation) để tạo cảm giác mượt mà hơn.

Hình 3 là một ảnh chụp màn hình minh họa giao diện chính của chương trình khi một ván cờ đang diễn ra.



Hình 3: Giao diện người dùng chính của chương trình, hiển thị bàn cờ, quân cờ và các thông tin phụ trợ.

- **Luồng chính và Quản lý Trò chơi (`main.py`):** Module này đóng vai trò điều phối toàn bộ ứng dụng:

- Khởi tạo các đối tượng cần thiết: một instance của `GameState` để quản lý logic cờ, một instance của `ChessVisualizer` để xử lý hiển thị, và một instance của `ChessAI` nếu chế độ chơi có AI.
- Hiển thị menu chính cho phép người dùng lựa chọn các chế độ chơi khác nhau: Người vs Người (Player vs Player), Người vs AI (Player vs AI), AI vs Random Agent.
- Cho phép người dùng chọn độ khó cho AI (Easy, Medium, Hard), tương ứng với việc thiết lập độ sâu tìm kiếm khác nhau cho Minimax Agent.
- Quản lý vòng lặp trò chơi chính (main game loop): Trong mỗi vòng lặp, xử lý các sự kiện từ Pygame (như click chuột, đóng cửa sổ), cập nhật trạng thái trò chơi dựa trên input của người dùng hoặc nước đi của AI, và gọi các hàm của `ChessVisualizer` để vẽ lại giao diện.
- Khi đến lượt AI, `main.py` sẽ gọi phương thức `get_best_move()` của đối tượng AI, truyền vào trạng thái hiện tại của `GameState`. Sau khi nhận được nước đi từ AI, nó sẽ cập nhật `GameState` và hiển thị nước đi đó.

Sự tách biệt giữa logic game (GameState), logic AI (MinimaxAI), và hiển thị (ChessVisualizer) giúp cho việc phát triển, gỡ lỗi và mở rộng từng thành phần trở nên dễ dàng hơn.

3 Kết quả và Đánh giá

3.1 Kiểm tra tính đúng luật

Quá trình kiểm thử thủ công và tự động xác nhận rằng agent luôn thực hiện các nước đi hợp lệ theo đầy đủ các quy tắc của Cờ Vua. Các tình huống phức tạp như nhập thành khi Vua đang bị chiếu, nhập thành qua ô bị tấn công, bắt tốt qua đường không hợp lệ, và các trường hợp tự đặt Vua vào thế bị chiếu đều được xử lý chính xác, agent không thực hiện các nước đi này. Các điều kiện kết thúc ván cờ như chiếu hết và hòa cờ (stalemate, threefold repetition, insufficient material) cũng được phát hiện đúng. (Đáp ứng Yêu cầu 1).

3.2 Thi đấu với Agent Ngẫu nhiên

Để đánh giá khả năng chiến thắng một đối thủ yếu theo Yêu cầu 2, một loạt 10 ván đấu tự động đã được tiến hành giữa Minimax Agent (cấu hình ở độ sâu tìm kiếm 2 - mức "Easy") và một agent đối thủ chỉ chọn các nước đi hợp lệ một cách hoàn toàn ngẫu nhiên. Kết quả của các thử nghiệm này như sau:

- **Tỷ lệ thắng của AI:** Minimax Agent đã chiến thắng tuyệt đối trong **10/10** ván đấu, đạt tỷ lệ 100%.
- **Thời gian trung bình để thắng mỗi ván:** Khoảng 15.66 giây.

- **Số ply (nửa nước đi) trung bình mỗi ván thắng:** Khoảng 31.1 ply, tương đương với khoảng 15-16 nước đi đầy đủ của cả hai bên.

Kết quả 100% chiến thắng này rõ ràng khẳng định agent đã đáp ứng được Yêu cầu 2 của đề bài. Việc agent ở độ sâu 2 có thể nhanh chóng đạt được thế chiếu hết (thường trong vòng chưa đến 20 nước) cho thấy sự vượt trội rõ ràng của việc áp dụng thuật toán tìm kiếm có định hướng và hàm lượng giá cơ bản so với chiến lược hoàn toàn ngẫu nhiên. Agent AI có khả năng nhận diện các cơ hội chiến thuật đơn giản và khai thác các sai lầm hiển nhiên của đối thủ ngẫu nhiên.

3.3 Phân cấp Độ khó

Việc thay đổi tham số độ sâu tìm kiếm (search depth) cho Minimax Agent (Depth 2 cho "Easy", Depth 3 cho "Medium", Depth 4 cho "Hard") đã tạo ra sự khác biệt rõ rệt về năng lực và phong cách chơi của agent:

- **Depth 2 (Easy):** Agent ở độ sâu này thường chỉ có khả năng "nhìn"trước một vài nước đi đơn lẻ. Các quyết định chủ yếu dựa trên các mối đe dọa hoặc cơ hội bắt quân tức thời (ví dụ, thấy một quân không được bảo vệ thì bắt, hoặc tránh một nước chiếu trực tiếp). Agent dễ mắc phải các bẫy chiến thuật đơn giản (ví dụ, bẫy hai nước) và thường không có kế hoạch dài hạn.

- **Depth 3 (Medium):** Với khả năng nhìn trước sâu hơn một chút, agent bắt đầu thể hiện khả năng nhận diện các đòn phối hợp ngắn (2-3 nước). Nó chơi ổn định hơn, tránh được các lỗi sơ đẳng và các bẫy đơn giản mà Depth 2 mắc phải. Agent bắt đầu có những ý đồ chiến thuật cơ bản hơn, ví dụ như phát triển quân một cách hợp lý hơn hoặc cố gắng kiểm soát trung tâm.

- **Depth 4 (Hard):** Ở độ sâu này, agent thể hiện khả năng tính toán sâu hơn đáng kể. Nó có thể nhận diện và thực hiện các chuỗi chiến thuật phức tạp hơn, phòng thủ chắc chắn hơn trước các mối đe dọa tiềm tàng, và có khả năng thiết lập các đòn tấn công có chiều sâu hơn. Các quyết định của agent ở Depth 4 trông "thông minh"và có tính toán hơn hẳn so với các độ sâu thấp hơn.

Sự khác biệt này xác nhận rằng việc điều chỉnh độ sâu tìm kiếm là một phương pháp hiệu quả và trực tiếp để phân cấp độ khó cho một game playing agent dựa trên thuật toán Minimax, đáp ứng Yêu cầu 3 của đề bài.

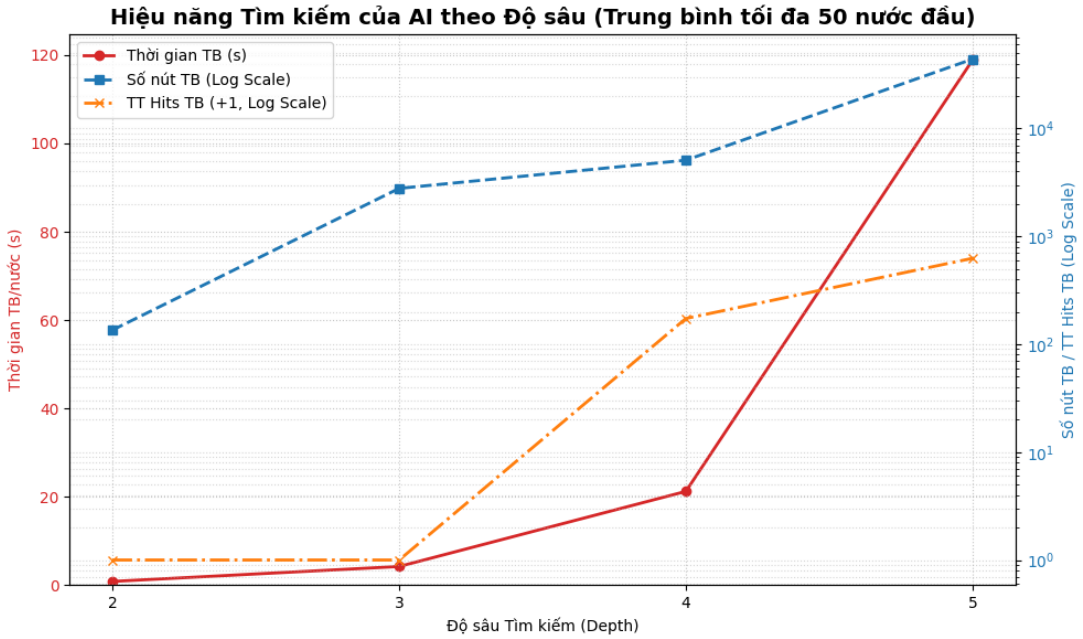
3.4 Hiệu năng Tìm kiếm

Để đánh giá hiệu năng của thuật toán tìm kiếm và các kỹ thuật tối ưu hóa đã triển khai, các thử nghiệm chi tiết đã được thực hiện. Agent AI được cho thi đấu với agent ngẫu nhiên ở các độ sâu tìm kiếm khác nhau (từ 2 đến 5). Các chỉ số hiệu năng trung bình, bao gồm thời gian tìm kiếm cho mỗi nước đi, tổng số nút (nodes) đã duyệt trong cây tìm kiếm, số

nút được duyệt trong giai đoạn Quiescence Search (QNodes), và số lần tra cứu thành công từ Transposition Table (TT Hits), được ghi lại qua tối đa **50 nước đi đầu tiên của AI** trong mỗi ván cờ (hoặc cho đến khi ván cờ kết thúc sớm). Việc đánh giá trên một số lượng nước đi lớn hơn này (so với 10 nước ban đầu) nhằm mục đích cung cấp một cái nhìn tổng quan và ổn định hơn về hiệu năng, mặc dù kết quả vẫn chủ yếu phản ánh hành vi của agent trong giai đoạn khai cuộc và đầu trung cuộc. Kết quả chi tiết được trình bày trong Bảng 2 và được minh họa trực quan trong Hình 4.

Bảng 2: So sánh hiệu năng tìm kiếm trung bình theo độ sâu (Tối đa 50 nước)

Depth	Time (s)	Nodes (k)	QNodes (k)	TT Hits	Moves Eval
2	0.818	0.14	0.42	0.0	10
3	4.201	2.79	3.27	0.0	15
4	21.207	5.08	10.71	172.3	11
5	118.872	44.00	61.45	625.2	9



Hình 4: Biểu đồ hiệu năng tìm kiếm theo độ sâu (Thời gian và Số nút).

Phân tích kết quả:

- **Thời gian và Số nút duyệt:** Các kết quả thực nghiệm cho thấy rõ ràng rằng chi phí tính toán, bao gồm cả thời gian xử lý trung bình cho mỗi nước đi và tổng số nút (nodes) đã duyệt trong cây tìm kiếm, đều tăng trưởng một cách mạnh mẽ (gần như theo hàm mũ) khi độ sâu tìm kiếm (search depth) tăng lên. Thời gian trung bình tăng từ dưới 1 giây ở Độ sâu 2 lên đến gần 2 phút ở Độ sâu 5. Tương tự, số nút

duyệt tăng từ vài trăm lên đến hàng chục nghìn. Điều này hoàn toàn phù hợp với lý thuyết về độ phức tạp của cây trò chơi Cờ Vua và nhấn mạnh giới hạn thực tế của việc tìm kiếm sâu với các tài nguyên phần cứng thông thường.

- **Hiệu quả của Transposition Table (TT):** Số lần tra cứu thành công từ Bảng Băm (TT Hits) gần như bằng không ở Độ sâu 2 và 3. Tuy nhiên, chỉ số này bắt đầu tăng lên đáng kể ở Độ sâu 4 (trung bình khoảng 172 hits) và tiếp tục tăng mạnh ở Độ sâu 5 (trung bình khoảng 625 hits), mặc dù số lượng nước đi thực tế được đánh giá ở các độ sâu này có thể ít hơn do ván cờ kết thúc sớm. Xu hướng này củng cố kết luận rằng TT là một kỹ thuật tối ưu hóa quan trọng, đặc biệt hiệu quả cho tìm kiếm sâu. Khi cây tìm kiếm mở rộng, xác suất gặp lại các thế cờ đã duyệt (transpositions) qua các chuỗi nước đi khác nhau tăng lên, và TT giúp agent tránh phải tính toán lại, tiết kiệm thời gian và tài nguyên.
- **Vai trò của Quiescence Search (QS):** Số lượng nút được duyệt trong giai đoạn Quiescence Search (QNodes) chiếm một tỷ trọng lớn và cũng tăng nhanh theo độ sâu tìm kiếm. Ví dụ, ở Độ sâu 5, số QNodes (trung bình 61.45k) còn lớn hơn cả số Nodes thông thường (44k). Điều này nhấn mạnh tầm quan trọng của QS trong việc phân tích sâu các chuỗi bắt quân và các biến thể chiến thuật tiềm ẩn tại các node lá của cây tìm kiếm chính, giúp hàm lượng giá được áp dụng trên các thế cờ ổn định hơn.
- **Sự cần thiết của Iterative Deepening (ID) và Giới hạn Thời gian:** Các thử nghiệm ở Độ sâu 5 cho thấy thời gian tìm kiếm trung bình (119 giây) rất gần với giới hạn thời gian cho mỗi nước đi được đặt ra trong quá trình đánh giá (180 giây). Phân tích log chi tiết khi chạy thử nghiệm cũng cho thấy nhiều lần tìm kiếm ở Độ sâu 5 (và cả một số ở Độ sâu 4) đã bị ngắt giữa chừng do timeout. Điều này chứng minh vai trò không thể thiếu của Iterative Deepening trong việc quản lý thời gian thực tế, đảm bảo agent có thể đưa ra được một nước đi hợp lý trong khoảng thời gian cho phép, ngay cả khi không thể hoàn thành tìm kiếm đến độ sâu tối đa mong muốn.
- **Ảnh hưởng của việc kết thúc sớm các ván đấu:** Việc các ván đấu ở độ sâu cao hơn (Depth 4 và 5) thường kết thúc với ít nước đi được đánh giá hơn (tương ứng 11 và 9 nước) cho thấy rằng AI ở các độ sâu này có khả năng tận dụng sai lầm của đối thủ ngẫu nhiên để dẫn đến chiếu hết một cách nhanh chóng hơn.

Nhìn chung, các kỹ thuật tối ưu hóa đã cho phép agent đạt được độ sâu tìm kiếm đáng kể và thể hiện hành vi chơi cờ có tính toán, cho thấy hiệu quả của thuật toán Alpha-Beta pruning và các phương pháp hỗ trợ đã triển khai.

3.5 Thi đấu kiểm chứng

Để kiểm chứng khả năng thi đấu của agent trong một môi trường thực tế hơn, agent với cấu hình (Depth: 4, time-limit: 60s) đã được cho thi đấu với chess bot The Pickpocket trên

nền tảng chess.com có mức ELO ước tính khoảng 1850.
Trong ván đấu thử nghiệm, agent đã thể hiện khả năng cạnh tranh và giành chiến thắng trong một số tình huống phức tạp. Video sau ghi lại quá trình agent giành chiến thắng trước chess bot:
https://www.youtube.com/watch?v=huqCbpy91o4
Kết quả này cho thấy agent có tiềm năng nhất định khi đối đầu với đối thủ mạnh hơn agent ngẫu nhiên.

3.6 Tuân thủ Yêu cầu Thuật toán

Toàn bộ logic của agent, từ việc sinh nước đi, tìm kiếm, đến đánh giá thế cờ, đều được xây dựng hoàn toàn dựa trên các thuật toán tìm kiếm cổ điển (Minimax, Alpha-Beta) và các phương pháp heuristics được thiết kế thủ công. Không có bất kỳ thành phần nào của agent sử dụng các mô hình hay kỹ thuật thuộc lĩnh vực Học máy (Machine Learning), tuân thủ đúng Yêu cầu 4 của đề bài.

4 Kết luận

4.1 Tóm tắt kết quả

Nhóm chúng em đã thành công trong việc thiết kế và hiện thực một engine Cờ Vua cơ bản cùng với một Minimax Agent có khả năng thi đấu. Agent đã chứng minh khả năng hoạt động đúng luật, chiến thắng thuyết phục trước đối thủ ngẫu nhiên, và thể hiện các cấp độ chơi khác nhau thông qua việc điều chỉnh độ sâu tìm kiếm. Việc áp dụng các kỹ thuật tối ưu hóa quan trọng như Cắt tỉa Alpha-Beta (Alpha-Beta Pruning), Tìm kiếm Sâu dần (Iterative Deepening Search), Bảng Băm (Transposition Table), Tìm kiếm Tĩnh lặng (Quiescence Search), và Sắp xếp Nước đi với MVV-LVA đã góp phần đáng kể vào hiệu năng và sức mạnh của agent. Các mục tiêu chính của Bài tập lớn đã được hoàn thành, cung cấp một nền tảng vững chắc cho việc tìm hiểu sâu hơn về AI trong game.

4.2 Hạn chế

Mặc dù đã đạt được các mục tiêu đề ra, nhóm em vẫn thấy có một số hạn chế như sau:

- **Hàm lượng giá (Evaluation Function):** Các heuristics được sử dụng, mặc dù bao gồm các yếu tố cơ bản như vật chất, vị trí quân (PST), cấu trúc Tốt, an toàn Vua và kiểm soát trung tâm, vẫn còn tương đối đơn giản. Nhiều khía cạnh tinh tế hơn của Cờ Vua chưa được đánh giá chi tiết, ví dụ như tính linh động của các quân cờ (piece mobility), sự phối hợp giữa các quân nặng (ví dụ: Xe trên cùng một cột mở, Tượng và Hậu phối hợp tấn công), các điểm yếu cấu trúc Tốt phức tạp hơn (ví dụ: backward pawns, pawn islands), hoặc đánh giá King Safety một cách toàn diện hơn (ví dụ: phạt điểm khi Vua bị tấn công trực tiếp bởi nhiều quân đối phương, hoặc khi có các cột/đường chéo mở nguy hiểm hướng về Vua).

- **Tối ưu hóa tìm kiếm chưa đầy đủ:** Mặc dù các kỹ thuật tối ưu hóa cơ bản đã được triển khai, vẫn còn thiếu các phương pháp tiên tiến hơn. Ví dụ, trong Sắp xếp Nước đi, việc triển khai Killer Heuristic và History Heuristic có thể cải thiện đáng kể hiệu quả cắt tỉa. Trong Quiescence Search, việc tích hợp Static Exchange Evaluation (SEE) hoặc Delta Pruning sẽ giúp kiểm soát tốt hơn sự mở rộng của cây tìm kiếm.
- **Thiếu kiến thức chuyên sâu về các giai đoạn ván cờ:** Agent hiện tại không sử dụng Sách Khai cuộc (Opening Book) cho giai đoạn đầu ván cờ, cũng như không có Bảng dữ liệu Tàn cuộc (Endgame Tablebases) cho các thế cờ có ít quân. Điều này có nghĩa là agent phải tự "suy nghĩ"từ đầu cho mọi nước đi, kể cả những nước đi khai cuộc đã được lý thuyết hóa rõ ràng, và có thể chơi không tối ưu trong các tình huống tàn cuộc phức tạp so với các engine được trang bị các kiến thức này.
- **Khả năng mở rộng và hiệu suất của Transposition Table:** Việc sử dụng Python dictionary cho Transposition Table là đơn giản để triển khai nhưng có thể không phải là giải pháp hiệu suất cao nhất cho các engine cờ vua đỉnh cao do overhead của Python. Các chiến lược thay thế (replacement strategies) cho TT khi bảng đầy cũng chưa được xem xét kỹ lưỡng.

4.3 Hướng phát triển

Dựa trên các hạn chế đã nêu, nhóm em đề xuất một số hướng phát triển tiềm năng để cải thiện và nâng cao agent trong tương lai:

- **Nâng cấp Toàn diện Hàm lượng giá:**
  - Bổ sung đánh giá tính linh động (mobility) của các quân cờ.
  - Thêm các heuristic đánh giá sự phối hợp giữa các quân (ví dụ: connected rooks, bishop pair).
  - Phát triển các thành phần đánh giá King Safety và Pawn Structure chi tiết và tinh vi hơn.
  - Nghiên cứu và áp dụng các hàm lượng giá đã được công bố từ các engine mạnh (ví dụ, các thành phần trong PeSTO's Evaluation Function [11]).
- **Tích hợp Kiến thức Chuyên sâu về Cờ Vua:**
  - Xây dựng hoặc tích hợp một Sách Khai cuộc (Opening Book) để agent có thể thực hiện các nước đi đầu tiên một cách nhanh chóng và theo lý thuyết.
  - Nghiên cứu khả năng sử dụng Bảng dữ liệu Tàn cuộc (Endgame Tablebases) cho các trường hợp tàn cuộc có số lượng quân giới hạn (ví dụ, 3-5 quân) để đảm bảo các nước đi tối ưu tuyệt đối.
- **Cải thiện các Kỹ thuật Tối ưu hóa Tìm kiếm:**

  - Triển khai các kỹ thuật Sắp xếp Nước đi nâng cao hơn như Killer Heuristic và History Heuristic.
  - Tích hợp Static Exchange Evaluation (SEE) vào cả Quiescence Search và quá trình sắp xếp nước đi bắt quân để lọc bỏ các trao đổi bất lợi sớm hơn.
  - Nghiên cứu các kỹ thuật pruning khác như Null Move Pruning, Futility Pruning.

- **Cải thiện Giao diện Người dùng và Tính năng Phân tích:** Nâng cấp GUI với các tính năng như phân tích nước đi, hiển thị cây tìm kiếm (nếu có thể), hoặc cho phép người dùng thiết lập các thế cờ cụ thể để thử nghiệm.
- **Tối ưu hóa hiệu suất ở mức thấp hơn:** Nếu mục tiêu là hiệu suất cao hơn nữa, có thể xem xét việc chuyển một số phần tính toán cốt lõi sang các ngôn ngữ biên dịch như C++ hoặc Rust, hoặc sử dụng các thư viện Python tối ưu hóa như NumPy cho các phép toán trên mảng.

Tài liệu tham khảo

[1] *Iterative Deepening*. Chess Programming Wiki. 2025-05-05. **october** 2019. URL: https://www.chessprogramming.org/Iterative_Deepening.

[2] *Transposition Table*. Chess Programming Wiki. 2025-05-05. **july** 2024. URL: https://www.chessprogramming.org/Transposition_Table.

[3] *Stockfish*. Stockfish GitHub Wiki. 2025-05-05. **april** 2025. URL: https://github.com/official-stockfish/Stockfish/wiki/Understanding-Stockfish.

[4] *Horizon effect*. Wikipedia. 2025-05-05. **april** 2025. URL: https://en.wikipedia.org/wiki/Horizon_effect.

[5] *Quiescence Search*. Chess Programming Wiki. 2025-05-05. **december** 2024. URL: https://www.chessprogramming.org/Quiescence_Search.

[6] *Move Ordering*. Chess Programming Wiki. 2025-05-05. **march** 2022. URL: https://www.chessprogramming.org/Move_Ordering.

[7] *MVV-LVA*. Chess Programming Wiki. 2025-05-05. **november** 2019. URL: https://www.chessprogramming.org/MVV-LVA.

[8] *Piece-Square Tables*. Chess Programming Wiki. 2025-05-05. **march** 2022. URL: https://www.chessprogramming.org/Piece-Square_Tables.

[9] Marcel Vanthoor. *Piece-Square Tables*. Rustic Chess. 2025-04-30. **april** 2025. URL: https://www.rustic-chess.org/evaluation/psqt.html.

[10] Adam Berent. *Piece Square Table*. Adam Berent Software & Hobbies. 2025-04-30. **march** 2019. URL: https://adamberent.com/piece-square-table/.

[11] *PeSTO's Evaluation Function*. Chess Programming Wiki. 2025-04-21. **july** 2021. URL: https://www.chessprogramming.org/PeSTO%27s_Evaluation_Function.

A Phụ lục A: Link Video Báo cáo

Link video báo cáo cho Bài tập lớn này được lưu trữ và có thể truy cập tại link Youtube sau: https://www.youtube.com/watch?v=vu7UjjQ77S0

B Phụ lục B: Source code

Source code hoàn chỉnh cho Bài tập lớn này được lưu trữ và có thể truy cập tại link GitHub sau:
https://github.com/minhnguyenrun/Pychess
Kho lưu trữ bao gồm toàn bộ mã nguồn Python, file README hướng dẫn cài đặt và sử dụng.