# BÁO CÁO TIỂU LUẬN

## Đề tài: Xây dựng và Phân tích Chi tiết Engine Cờ vua sử dụng Thuật toán Negamax và Các Kỹ thuật Tối ưu Hóa Nâng cao

Sinh viên: Nguyễn Đình Phú-Nguyễn Ngọc Bình An

Lớp: 2526I\_AIT2002\_3 – Cơ sở trí tuệ nhân tạo

**MỤC LỤC**

[**CHƯƠNG 1: MỞ ĐẦU 3**](#_Toc212369672)

[**1.1. Giới thiệu đề tài 3**](#_Toc212369673)

[**1.2. Mục tiêu của dự án 3**](#_Toc212369674)

[**1.3. Cấu trúc báo cáo 4**](#_Toc212369675)

[**CHƯƠNG 2: KIẾN TRÚC VÀ LOGIC HOẠT ĐỘNG 4**](#_Toc212369676)

[**2.1. Sơ đồ kiến trúc hệ thống 4**](#_Toc212369677)

[**2.2. Luồng hoạt động chi tiết 5**](#_Toc212369678)

[**2.3. Các thư viện và công nghệ sử dụng 6**](#_Toc212369679)

[**CHƯƠNG 3: PHÂN TÍCH CHI TIẾT CÁC HÀM VÀ MODULE 6**](#_Toc212369680)

[**3.1. Module Điều khiển và Tìm kiếm (search.py) 6**](#_Toc212369681)

[**3.1.1. Hàm điều khiển chính: find\_best\_move() 6**](#_Toc212369682)

[**3.1.2. Hàm tìm kiếm gốc: search\_root() 7**](#_Toc212369683)

[**3.1.3. Hàm tìm kiếm đệ quy chính: negamax() 8**](#_Toc212369684)

[**3.1.4. Hàm tìm kiếm yên tĩnh: quiescence\_search() 9**](#_Toc212369685)

[**3.2. Module Lượng giá (evaluation.py) 10**](#_Toc212369686)

[**3.2.1. Hàm lượng giá chính: evaluate\_board() 10**](#_Toc212369687)

[**3.2.2. Các hàm lượng giá thành phần 11**](#_Toc212369688)

[**CHƯƠNG 4: PHÂN TÍCH ĐỘ PHỨC TẠP VÀ HIỆU SUẤT 12**](#_Toc212369689)

[**4.1. Độ phức tạp thuật toán (Time Complexity) 12**](#_Toc212369690)

[**4.2. Độ phức tạp không gian (Space Complexity) 12**](#_Toc212369691)

[**4.3. Các chỉ số đo lường hiệu suất 13**](#_Toc212369692)

[**4.3.1. Số nút trên giây (Nodes Per Second - NPS) 13**](#_Toc212369693)

[**4.3.2. Hiệu quả cắt tỉa và Bảng băm 13**](#_Toc212369694)

[**CHƯƠNG 5: MÔI TRƯỜNG CÀI ĐẶT VÀ HƯỚNG DẪN SỬ DỤNG 13**](#_Toc212369695)

[**5.1. Yêu cầu hệ thống 13**](#_Toc212369696)

[**5.2. Cài đặt thư viện 14**](#_Toc212369697)

[**5.3. Sách khai cuộc (Opening Book) 14**](#_Toc212369699)

[**CHƯƠNG 6: HƯỚNG PHÁT TRIỂN VÀ CẢI TIỆN 14**](#_Toc212369700)

[**6.1. Cải tiến thuật toán tìm kiếm 14**](#_Toc212369701)

[**6.2. Cải tiến hàm lượng giá 14**](#_Toc212369702)

[**6.3. Cải tiến kiến trúc và hiệu năng 15**](#_Toc212369703)

[**CHƯƠNG 7: KẾT LUẬN 15**](#_Toc212369704)

[**7.1. Tổng kết kết quả 15**](#_Toc212369705)

[**7.2. Bài học kinh nghiệm 15**](#_Toc212369706)

[**TÀI LIỆU THAM KHẢO 16**](#_Toc212369707)

# CHƯƠNG 1: MỞ ĐẦU

## 1.1. Giới thiệu đề tài

Cờ vua, với sự phức tạp về chiến lược và chiến thuật, từ lâu đã trở thành một "bài toán tiêu chuẩn" (benchmark problem) trong lĩnh vực Trí tuệ Nhân tạo. Việc phát triển một chương trình máy tính có khả năng chơi cờ không chỉ là một bài tập lập trình mà còn là một cuộc thám hiểm sâu vào các thuật toán tìm kiếm, kỹ thuật tối ưu hóa và phương pháp biểu diễn tri thức. Dự án này trình bày quá trình thiết kế, triển khai và phân tích một engine cờ vua hoàn chỉnh bằng ngôn ngữ Python. Nền tảng của chương trình là thuật toán Negamax (một biến thể của Minimax) được tăng cường sức mạnh bởi một bộ sưu tập các kỹ thuật tối ưu hóa hiện đại như Alpha-Beta, Bảng băm, Sắp xếp nước đi, và nhiều heuristic khác. Báo cáo này sẽ phân tích chi tiết từng thành phần, từ logic tìm kiếm đến hàm lượng giá, nhằm cung cấp một cái nhìn toàn diện về cách một engine cờ vua hiện đại hoạt động.

## 1.2. Mục tiêu của dự án

* Xây dựng một engine cờ vua có khả năng: Triển khai một chương trình có khả năng nhận một thế cờ, phân tích và đưa ra nước đi tối ưu dựa trên thuật toán tìm kiếm.
* Triển khai thuật toán tìm kiếm cốt lõi: Xây dựng thuật toán Negamax với Alpha-Beta làm xương sống cho quá trình ra quyết định.
* Tích hợp các kỹ thuật tối ưu hóa hiệu suất: Bổ sung các kỹ thuật như Tìm kiếm sâu dần (Iterative Deepening), Bảng băm (Transposition Table), Sắp xếp nước đi (Move Ordering), Tìm kiếm tĩnh (Quiescence Search), Cắt tỉa nước đi rỗng (Null Move Pruning), và các Heuristic (Killer, History) để tăng tốc độ và độ sâu tìm kiếm.
* Thiết kế hàm lượng giá toàn diện: Xây dựng một hàm lượng giá (evaluation function) phức tạp, có khả năng đánh giá các khía cạnh đa dạng của một thế cờ, bao gồm vật chất, vị trí quân, cấu trúc Tốt, an toàn Vua, và sự kiểm soát.
* Phân tích và đánh giá chi tiết: Phân tích sâu về chức năng của từng hàm, độ phức tạp thuật toán, hiệu suất thực tế và đề xuất các hướng cải tiến trong tương lai.

## 1.3. Cấu trúc báo cáo

Báo cáo được tổ chức thành 7 chương, bắt đầu từ tổng quan đến chi tiết. Chương 1 và 2 giới thiệu và mô tả kiến trúc tổng thể. Chương 3 là phần trọng tâm, phân tích sâu từng hàm và module. Chương 4 đánh giá hiệu suất và độ phức tạp. Chương 5 cung cấp hướng dẫn cài đặt. Chương 6 thảo luận về các hướng phát triển. Chương 7 đưa ra kết luận cuối cùng.

# CHƯƠNG 2: KIẾN TRÚC VÀ LOGIC HOẠT ĐỘNG

## 2.1. Sơ đồ kiến trúc hệ thống

Engine được xây dựng theo kiến trúc module hóa, giúp dễ dàng quản lý, gỡ lỗi và mở rộng. Các thành phần chính tương tác với nhau như sau:

1. Module Điều khiển Chính (search.py - find\_best\_move): Đây là điểm vào chính của engine. Nó điều phối toàn bộ quá trình tìm kiếm, quản lý thời gian, khởi tạo các cấu trúc dữ liệu và triển khai vòng lặp Tìm kiếm sâu dần.
2. Module Tìm kiếm (search.py - negamax, search\_root, quiescence\_search): Là trái tim của engine, chịu trách nhiệm duyệt cây trò chơi. Nó triển khai thuật toán Negamax và các kỹ thuật tối ưu hóa liên quan trực tiếp đến việc cắt tỉa các nhánh tìm kiếm.
3. Module Lượng giá (evaluation.py - evaluate\_board): Là bộ não chiến lược, cung cấp khả năng đánh giá "tĩnh" một thế cờ. Module này không tìm kiếm mà chỉ tính toán một điểm số dựa trên các đặc điểm của bàn cờ.
4. Module Hằng số và Dữ liệu (constant.py): Chứa tất cả các giá trị, hằng số, và các bảng dữ liệu tĩnh như Bảng vị trí quân (PST), giá trị quân cờ, điểm thưởng/phạt. Việc tách riêng module này giúp dễ dàng tinh chỉnh (tuning) engine.
5. Thư viện Lõi Cờ vua (python-chess): Cung cấp nền tảng cơ bản để quản lý trạng thái bàn cờ, sinh nước đi hợp lệ, thực hiện/hoàn tác nước đi, và các tiện ích như mã hóa Zobrist.

## 2.2. Luồng hoạt động chi tiết

Khi nhận được yêu cầu tìm nước đi, luồng xử lý của engine diễn ra như sau:

1. Khởi tạo: Hàm find\_best\_move được gọi.
   * Tra sách khai cuộc: Engine thử tìm thế cờ hiện tại trong sách khai cuộc Polyglot. Nếu tìm thấy, nó sẽ ngay lập tức trả về nước đi từ sách mà không cần tìm kiếm.
   * Thiết lập lại: Nếu không có nước đi trong sách, engine sẽ xóa Bảng băm và khởi tạo lại các cấu trúc dữ liệu như Killer Moves và History Heuristic để chuẩn bị cho một lượt tìm kiếm mới.
2. Vòng lặp Tìm kiếm sâu dần (Iterative Deepening):
   * Engine bắt đầu một vòng lặp for depth in range(1, max\_depth + 1).
   * Ở mỗi độ sâu depth, nó gọi hàm search\_root(depth).
3. Tìm kiếm ở Cấp độ Gốc (search\_root):
   * Lấy danh sách tất cả các nước đi hợp lệ.
   * Nó sử dụng hàm order\_moves để sắp xếp các nước đi này, ưu tiên nước đi tốt nhất từ lần lặp trước (PV-move).
   * Nó duyệt qua từng nước đi, thực hiện nước đi đó trên bàn cờ, và gọi hàm negamax(depth - 1) để đánh giá kết quả.
   * Sau mỗi lần gọi, nó hoàn tác nước đi và cập nhật nước đi tốt nhất tìm được ở độ sâu hiện tại.
4. Tìm kiếm Đệ quy (negamax):
   * Đây là hàm đệ quy thực hiện tìm kiếm theo chiều sâu.
   * Khi được gọi, nó kiểm tra các điều kiện dừng (chiếu hết, hết độ sâu), tra cứu Bảng băm, áp dụng các kỹ thuật cắt tỉa (Null Move), sắp xếp nước đi, và sau đó gọi đệ quy chính nó cho các nước đi con.
   * Khi đạt độ sâu bằng 0, nó chuyển quyền kiểm soát cho quiescence\_search.
5. Tìm kiếm Yên tĩnh (quiescence\_search):
   * Hàm này chỉ xem xét các nước đi "ồn ào" (ăn quân, phong cấp) để tránh "hiệu ứng đường chân trời".
   * Nó tiếp tục tìm kiếm cho đến khi thế cờ trở nên ổn định, sau đó gọi evaluate\_board.
6. Lượng giá Tĩnh (evaluate\_board):
   * Hàm này tính toán một điểm số duy nhất cho thế cờ hiện tại dựa trên các yếu tố chiến lược và trả về cho chuỗi gọi đệ quy.
7. Hoàn tất:
   * Vòng lặp trong find\_best\_move tiếp tục cho đến khi hết thời gian hoặc đạt độ sâu tối đa.
   * Engine in thông tin tìm kiếm theo chuẩn UCI sau mỗi lần hoàn thành một độ sâu.
   * Cuối cùng, nó trả về nước đi tốt nhất tổng thể đã tìm được.

## 2.3. Các thư viện và công nghệ sử dụng

* Ngôn ngữ lập trình : Python3.
* Thư viện chính: python-chess. Đây là thư viện cốt lõi, cung cấp các lớp và phương thức để:
  + Biểu diễn bàn cờ (chess.Board).
  + Sinh và xác thực nước đi (board.legal\_moves, board.is\_capture).
  + Thực hiện và hoàn tác nước đi (board.push(), board.pop()).
  + Kiểm tra trạng thái ván cờ (board.is\_checkmate(), board.is\_stalemate()).
  + Cung cấp mã băm Zobrist (chess.polyglot.zobrist\_hash), một thành phần thiết yếu cho Bảng băm.

# CHƯƠNG 3: PHÂN TÍCH CHI TIẾT CÁC HÀM VÀ MODULE

Đây là chương quan trọng nhất, đi sâu vào từng thành phần của engine.

## 3.1. Module Điều khiển và Tìm kiếm (search.py)

Module này chứa logic chính điều khiển quá trình tìm kiếm nước đi.

### 3.1.1. Hàm điều khiển chính: find\_best\_move()

#### Mục đích

Là hàm "giao diện" cấp cao nhất, quản lý toàn bộ quá trình tìm kiếm từ đầu đến cuối, bao gồm quản lý thời gian, vòng lặp sâu dần, và khởi tạo/dọn dẹp tài nguyên.

#### Logic hoạt động chi tiết

1. Kiểm tra Sách khai cuộc: Sử dụng chess.polyglot.MemoryMappedReader để đọc tệp sách .bin. Nếu thế cờ hiện tại có trong sách, hàm trả về nước đi tương ứng và kết thúc ngay lập tức. Đây là một cách tối ưu hóa cực kỳ hiệu quả cho giai đoạn đầu ván cờ.
2. Khởi tạo Môi trường Tìm kiếm: Đặt lại các biến toàn cục về trạng thái ban đầu:
   * position\_count = 0: Reset bộ đếm số nút đã duyệt.
   * killer\_moves = [[None, None] for \_ in range(MAX\_DEPTH)]: Xóa tất cả các nước đi "sát thủ" đã lưu từ lần tìm kiếm trước.
   * history\_heuristic = ...: Reset bảng điểm lịch sử về 0.
   * transposition\_table.clear(): Xóa toàn bộ Bảng băm. Điều này rất quan trọng vì các mục trong bảng chỉ hợp lệ cho một lượt tìm kiếm duy nhất.
3. Vòng lặp Tìm kiếm Sâu dần (Iterative Deepening):
   * Bắt đầu một vòng lặp từ depth = 1 đến max\_depth.
   * Trong mỗi vòng lặp, nó gọi search\_root(gamestate, depth, best\_move\_overall). Tham số best\_move\_overall là nước đi tốt nhất từ độ sâu depth - 1, được dùng để ưu tiên tìm kiếm ở độ sâu hiện tại.
   * Sau khi search\_root trả về, nó cập nhật best\_move\_overall với nước đi mới.
   * Tính toán và in ra thông tin theo chuẩn UCI (Universal Chess Interface), bao gồm độ sâu, điểm số, thời gian, số nút, NPS (nodes per second), và chuỗi nước đi chính (PV).
4. Quản lý Heuristic và Thời gian:
   * Sau mỗi 5 độ sâu, gọi age\_history\_heuristic() để giảm giá trị của bảng lịch sử, tránh tràn số và giữ cho thông tin luôn mới.
   * Kiểm tra nếu thời gian tìm kiếm đã vượt quá time\_limit\_seconds. Nếu có, vòng lặp sẽ bị ngắt.
   * Kiểm tra nếu tìm thấy một thế chiếu hết bắt buộc (forced mate). Nếu có, vòng lặp cũng sẽ kết thúc sớm.
5. Trả về kết quả: Trả về best\_move\_overall, là nước đi tốt nhất tìm được từ độ sâu cuối cùng mà engine đã hoàn thành.

### 3.1.2. Hàm tìm kiếm gốc: search\_root()

#### Mục đích

Chuyên xử lý việc tìm kiếm ở cấp độ đầu tiên (root) của cây trò chơi. Nó khác với negamax vì không cần quản lý cửa sổ alpha-beta theo cách tương tự và chịu trách nhiệm khởi tạo các cuộc gọi đệ quy đầu tiên.

#### Logic hoạt động chi tiết

1. Khởi tạo alpha = -infinity và beta = +infinity.
2. Lấy danh sách các nước đi hợp lệ.
3. Ưu tiên PV-Move: Nếu có một pv\_move (nước đi tốt nhất từ lần lặp trước) được truyền vào, nó sẽ được di chuyển lên đầu danh sách để được duyệt đầu tiên. Đây là một tối ưu hóa quan trọng cho Iterative Deepening.
4. Sắp xếp nước đi: Gọi order\_moves cho các nước đi còn lại để sắp xếp chúng theo thứ tự ưu tiên.
5. Vòng lặp duyệt nước đi:
   * Với mỗi nước đi trong danh sách đã sắp xếp:
     1. Thực hiện nước đi: gamestate.make\_move(move).
     2. Gọi hàm tìm kiếm chính: score = -negamax(gamestate, depth - 1, -beta, -alpha, 1). Lưu ý cách cửa sổ alpha-beta được đảo ngược.
     3. Hoàn tác nước đi: gamestate.unmake\_move().
     4. Cập nhật điểm số tốt nhất: Nếu score mới tốt hơn best\_score, cập nhật best\_score và best\_move.
     5. Cập nhật alpha: alpha = max(alpha, score).
6. Trả về một tuple chứa nước đi tốt nhất và điểm số tương ứng.

### 3.1.3. Hàm tìm kiếm đệ quy chính: negamax()

#### Mục đích

Đây là hàm cốt lõi, thực hiện thuật toán tìm kiếm Negamax với cắt tỉa Alpha-Beta và tất cả các kỹ thuật tối ưu hóa phức tạp khác.

#### Logic hoạt động chi tiết

1. Kiểm tra điều kiện dừng đệ quy:
   * Nếu ván cờ kết thúc (chiếu hết/hòa), trả về điểm số tương ứng (-MATE\_VALUE hoặc 0).
   * Nếu depth <= 0, trả về kết quả từ quiescence\_search(alpha, beta).
   * Nếu có sự lặp lại thế cờ hoặc vi phạm luật 50 nước, trả về 0 (hòa).
2. Tra cứu Bảng băm (Transposition Table Lookup):
   * Tính Zobrist hash cho thế cờ hiện tại.
   * Nếu tìm thấy một mục trong transposition\_table có độ sâu lớn hơn hoặc bằng độ sâu hiện tại:
     + Nếu cờ là TT\_EXACT, trả về điểm số đã lưu ngay lập tức.
     + Nếu cờ là TT\_LOWERBOUND, cập nhật alpha.
     + Nếu cờ là TT\_UPPERBOUND, cập nhật beta.
     + Nếu alpha >= beta sau khi cập nhật, trả về điểm số đã lưu (gây ra cắt tỉa).
3. Cắt tỉa nước đi rỗng (Null Move Pruning):
   * Kiểm tra các điều kiện: độ sâu đủ lớn, không bị chiếu, và còn quân mạnh.
   * Nếu thỏa mãn, engine thực hiện một "nước đi rỗng" (bỏ qua lượt) và gọi đệ quy negamax với độ sâu giảm đi.
   * Nếu điểm số trả về từ cuộc gọi này lớn hơn hoặc bằng beta, điều đó có nghĩa là thế cờ hiện tại rất mạnh, và có thể cắt tỉa nhánh này ngay lập tức bằng cách trả về beta.
4. Duyệt nước đi con:
   * Gọi order\_moves để sắp xếp các nước đi hợp lệ.
   * Vào vòng lặp duyệt từng nước đi:
     + Thực hiện nước đi.
     + Gọi đệ quy score = -negamax(depth - 1, -beta, -alpha, ...).
     + Hoàn tác nước đi.
     + Cập nhật best\_score và best\_move.
     + Cập nhật alpha.
     + Cắt tỉa Alpha-Beta: Nếu alpha >= beta, đây là điểm cắt tỉa.
       - Nếu nước đi gây cắt tỉa không phải là nước ăn quân, nó sẽ được lưu vào Killer Moves và History Heuristic.
       - Thoát khỏi vòng lặp ngay lập tức.
5. Lưu vào Bảng băm (Transposition Table Store):
   * Xác định cờ (TT\_EXACT, TT\_LOWERBOUND, hoặc TT\_UPPERBOUND) dựa trên mối quan hệ giữa best\_score và cửa sổ alpha-beta ban đầu.
   * Tạo một TTEntry mới và lưu vào transposition\_table với Zobrist hash làm khóa.
6. Trả về best\_score.

### 3.1.4. Hàm tìm kiếm yên tĩnh: quiescence\_search()

#### Mục đích

Giải quyết "hiệu ứng đường chân trời" (horizon effect) bằng cách đảm bảo engine không dừng tìm kiếm ở một thế cờ không ổn định (ví dụ, giữa một loạt nước ăn quân). Nó mở rộng tìm kiếm chỉ với các nước đi "ồn ào" (tactical moves).

#### Logic hoạt động chi tiết

1. Tăng bộ đếm position\_count.
2. Đánh giá "Stand Pat": Gọi evaluate\_board() để lấy điểm số của thế cờ hiện tại, giả sử không thực hiện thêm nước đi nào. Đây là một cận dưới cho điểm số.
3. Nếu điểm "stand pat" đã đủ để gây cắt tỉa beta (stand\_pat >= beta), trả về beta.
4. Cập nhật alpha với điểm "stand pat".
5. Tạo danh sách nước đi "ồn ào": Chỉ bao gồm các nước ăn quân (generate\_legal\_captures) và các nước phong cấp không ăn quân.
6. Sắp xếp nước đi: Gọi order\_moves để sắp xếp các nước đi này, ưu tiên các nước ăn quân có giá trị cao (ví dụ: Tốt ăn Hậu).
7. Vòng lặp duyệt nước đi:
   * Thực hiện nước đi.
   * Gọi đệ quy score = -quiescence\_search(-beta, -alpha, ...).
   * Hoàn tác nước đi.
   * Cập nhật alpha và kiểm tra cắt tỉa beta như trong negamax.
8. Trả về alpha, là điểm số tốt nhất sau khi thế cờ đã được ổn định.

## 3.2. Module Lượng giá (evaluation.py)

Module này chứa "trí tuệ" cờ vua của engine, số hóa các trạng thái của bàn cờ

### 3.2.1. Hàm lượng giá chính: evaluate\_board()

#### Mục đích

Tính toán một điểm số tĩnh cho một thế cờ, tổng hợp từ nhiều yếu tố khác nhau. Điểm dương lợi cho Trắng, điểm âm lợi cho Đen.

#### Logic hoạt động chi tiết

1. Kiểm tra các trạng thái kết thúc đơn giản: Nếu là chiếu hết, hòa, hoặc không đủ quân, trả về các giá trị tương ứng.
2. Tính toán Giai đoạn Ván cờ (Game Phase):
   * Tính tổng giá trị "phase" của tất cả các quân cờ trên bàn cờ (trừ Vua). Mỗi quân có một giá trị phase riêng (ví dụ: Hậu=4, Xe=2).
   * Giá trị current\_phase\_score xác định ván cờ đang ở giai đoạn giữa trung cuộc và tàn cuộc.
3. Tính toán Điểm Vật chất và Vị trí (PST):
   * Duyệt qua từng loại quân (Tốt, Mã, Tượng,...).
   * Với mỗi quân của Trắng, cộng vào tổng điểm trung cuộc (mg\_total) và tàn cuộc (eg\_total) giá trị vật chất và giá trị vị trí từ bảng PST tương ứng.
   * Với mỗi quân của Đen, trừ đi các giá trị tương tự. Bảng PST cho quân Đen được lấy bằng cách lật ngược bảng của quân Trắng (chess.square\_mirror).
4. Tính toán các Yếu tố Phụ:
   * Với mỗi bên (Trắng và Đen), gọi các hàm con để tính điểm thưởng/phạt cho:
     + get\_pawn\_structure(): Cấu trúc Tốt.
     + get\_sub\_piece\_bonus(): Các quân phụ (Xe, Tượng, Mã).
     + get\_king\_safety(): An toàn Vua.
     + evaluate\_attacks(): Tấn công và kiểm soát.
   * Cộng/trừ các điểm số này vào mg\_total và eg\_total.
5. Nội suy Điểm số Cuối cùng:
   * Sử dụng hàm phase\_score\_calculator để nội suy tuyến tính giữa mg\_total và eg\_total dựa trên current\_phase\_score.
   * Công thức: (mg\*phase + eg\*(total-phase))/total.
6. Điều chỉnh theo Lượt đi: Trả về điểm số cuối cùng. Nếu đến lượt Trắng đi, trả về final\_score. Nếu đến lượt Đen đi, trả về -final\_score.

### 3.2.2. Các hàm lượng giá thành phần

#### get\_pawn\_structure()

Tổng hợp điểm từ các hàm con đánh giá Tốt:

* get\_doubled\_pawns\_penalty(): Phạt điểm cho các Tốt đứng trên cùng một cột.
* get\_isolated\_pawns\_penalty(): Phạt điểm cho Tốt không có Tốt đồng minh ở các cột liền kề.
* get\_connected\_pawns\_bonus(): Thưởng điểm cho các Tốt được bảo vệ bởi Tốt khác.
* get\_passed\_pawn\_bonus(): Thưởng điểm rất lớn cho Tốt không bị Tốt đối phương cản đường, giá trị thưởng tăng theo hạng của Tốt.
* get\_backward\_pawn\_penalty(): Phạt điểm cho Tốt yếu, không thể tiến lên và dễ bị tấn công.

#### get\_sub\_piece\_bonus()

Tổng hợp điểm từ các hàm con đánh giá quân phụ:

* get\_rook\_bonus(): Thưởng điểm cho Xe trên cột mở/nửa mở và Xe ở hàng thứ 7 (hoặc thứ 2 của đối phương).
* get\_double\_bishop\_bonus(): Thưởng một lượng điểm đáng kể nếu một bên sở hữu cả hai Tượng.
* get\_knight\_outpost\_bonus(): Thưởng điểm cho Mã đứng ở vị trí tiền đồn mạnh (được Tốt bảo vệ và không thể bị Tốt đối phương tấn công).

#### get\_king\_safety()

Tổng hợp điểm từ các hàm con đánh giá Vua:

* pawn\_shield\_penalty(): Phạt điểm nếu các Tốt che chắn cho Vua đã nhập thành bị thiếu hoặc di chuyển sai vị trí.
* king\_attack\_zone\_penalty(): Tính toán mức độ nguy hiểm bằng cách đếm số quân đối phương và các đòn tấn công của chúng vào các ô xung quanh Vua.
* king\_activity\_bonus(): Thưởng điểm cho Vua hoạt động tích cực ở trung tâm trong giai đoạn tàn cuộc.
* king\_attack\_bonus(): Thưởng điểm nếu các quân của mình đang trực tiếp tấn công Vua đối phương.
* king\_additional(): Các yếu tố khác như thưởng cho việc đã nhập thành, phạt nếu Vua di chuyển mà chưa nhập thành.

#### evaluate\_attacks()

Đánh giá các yếu tố liên quan đến tấn công và kiểm soát:

* Tấn công quân đối phương: Thưởng một lượng điểm nhỏ cho mỗi quân đối phương đang bị tấn công.
* Tấn công Vua đối phương: Thưởng điểm cho mỗi quân đang chiếu hoặc tấn công Vua đối phương.
* Kiểm soát trung tâm: Thưởng điểm cho việc chiếm giữ các ô trung tâm (d4, e4, d5, e5).

# CHƯƠNG 4: PHÂN TÍCH ĐỘ PHỨC TẠP VÀ HIỆU SUẤT

## 4.1. Độ phức tạp thuật toán (Time Complexity)

Độ phức tạp của thuật toán tìm kiếm trong cờ vua được đo bằng số nút (thế cờ) phải duyệt. Nó phụ thuộc vào hệ số nhánh (b - branching factor, số nước đi trung bình) và độ sâu tìm kiếm (d).

* Trường hợp xấu nhất (Minimax thuần túy): O(b^d). Với b ≈ 35, độ phức tạp này tăng theo hàm mũ và không khả thi trong thực tế.
* Trường hợp tốt nhất (Alpha-Beta với sắp xếp nước đi hoàn hảo): O(b^(d/2)). Đây là giới hạn lý thuyết khi thuật toán luôn duyệt nước đi tốt nhất đầu tiên, cho phép cắt tỉa tối đa.
* Thực tế: Engine này, nhờ vào các kỹ thuật như Sắp xếp nước đi (TT move, MVV-LVA, Killers, History), Null Move Pruning và Bảng băm, tiệm cận rất gần với trường hợp tốt nhất. Các kỹ thuật này giúp giảm đáng kể "hệ số nhánh hiệu quả", cho phép engine đạt đến độ sâu lớn hơn nhiều so với một thuật toán Alpha-Beta đơn giản.

## 4.2. Độ phức tạp không gian (Space Complexity)

Bộ nhớ mà engine sử dụng chủ yếu đến từ các thành phần sau:

* Bảng băm (Transposition Table): O(N), với N là số lượng mục được lưu. Đây là thành phần tốn nhiều bộ nhớ nhất. Kích thước của bảng là một sự đánh đổi trực tiếp giữa bộ nhớ và hiệu suất (bảng lớn hơn -> tỷ lệ trúng cao hơn -> tìm kiếm nhanh hơn).
* Ngăn xếp Đệ quy: O(d). Bộ nhớ cần thiết cho các lệnh gọi hàm đệ quy, tỷ lệ thuận với độ sâu tìm kiếm tối đa.
* Killer Moves và History Heuristic: O(d) và O(1) tương ứng. Lượng bộ nhớ này tương đối nhỏ và không đáng kể so với Bảng băm.

## 4.3. Các chỉ số đo lường hiệu suất

### 4.3.1. Số nút trên giây (Nodes Per Second - NPS)

NPS là chỉ số đo lường "tốc độ thô" của engine, được tính bằng position\_count / thời\_gian\_tìm\_kiếm. Mặc dù NPS cao cho thấy engine duyệt cây nhanh, nó không phải lúc nào cũng tương quan với sức cờ. Một engine "thông minh" hơn có thể có NPS thấp hơn nhưng cắt tỉa hiệu quả hơn, chỉ duyệt các nhánh quan trọng và đạt được kết quả tốt hơn. Do được viết bằng Python, NPS của engine này sẽ thấp hơn đáng kể so với các engine viết bằng C++ hoặc Rust, nhưng các nguyên tắc thuật toán vẫn được thể hiện rõ.

### 4.3.2. Hiệu quả cắt tỉa và Bảng băm

Hiệu quả thực sự của engine nằm ở khả năng cắt tỉa. Điều này có thể được đánh giá qua:

* Tỷ lệ trúng Bảng băm (TT Hit Rate): Tỷ lệ phần trăm các thế cờ được tìm thấy trong Bảng băm. Tỷ lệ cao cho thấy engine đang tái sử dụng hiệu quả các tính toán trước đó.
* Hiệu quả Sắp xếp nước đi: Có thể đo bằng cách theo dõi vị trí của nước đi gây ra cắt tỉa beta trong danh sách đã sắp xếp. Nếu cắt tỉa thường xuyên xảy ra ở nước đi đầu tiên hoặc thứ hai, điều đó cho thấy hệ thống sắp xếp (đặc biệt là TT move và Killer moves) đang hoạt động rất tốt.

# CHƯƠNG 5: MÔI TRƯỜNG CÀI ĐẶT VÀ HƯỚNG DẪSN SỬ DỤNG

## 5.1. Yêu cầu hệ thống

* Hệ điều hành: Windows, macOS, hoặc Linux.
* Trình thông dịch: Python 3.7 trở lên.
* Bộ nhớ RAM: Tối thiểu 4GB. Khuyến nghị 16GB trở lên để Bảng băm có thể lưu trữ nhiều mục hơn, cải thiện hiệu suất.

## 5.2. Cài đặt thư viện

Engine yêu cầu thư viện python-chess. Cài đặt bằng trình quản lý gói pip:

pip install python-chess

## 5.3. Sách khai cuộc (Opening Book)

Engine có khả năng sử dụng sách khai cuộc định dạng Polyglot (.bin). Người dùng cần:

1. Tải một tệp sách khai cuộc, ví dụ như CerebellumLight.bin.
2. Cập nhật đường dẫn tuyệt đối đến tệp này trong hàm find\_best\_move của tệp search.py.

# CHƯƠNG 6: HƯỚNG PHÁT TRIỂN VÀ CẢI TIỆN

## 6.1. Cải tiến thuật toán tìm kiếm

* Principal Variation Search (PVS): Một biến thể của Alpha-Beta, tối ưu hóa việc tìm kiếm bằng cách giả định nước đi đầu tiên là tốt nhất và tìm kiếm nó với cửa sổ đầy đủ, trong khi các nước đi sau được kiểm tra nhanh bằng "cửa sổ rỗng" (null window).
* Late Move Reductions (LMR): Một kỹ thuật heuristic giúp giảm độ sâu tìm kiếm cho các nước đi được coi là "kém hứa hẹn" (những nước đi ở cuối danh sách đã sắp xếp), giúp tiết kiệm thời gian tìm kiếm.
* Aspiration Windows: Bắt đầu tìm kiếm với một cửa sổ alpha-beta hẹp xung quanh điểm số của lần lặp trước. Nếu kết quả nằm ngoài cửa sổ này (fail-high hoặc fail-low), thực hiện tìm kiếm lại với cửa sổ rộng hơn.

## 6.2. Cải tiến hàm lượng giá

* Tinh chỉnh Tự động (Automatic Tuning): Các hằng số trong constant.py hiện được chọn thủ công. Có thể sử dụng các phương pháp như Texel's Tuning để tự động tối ưu hóa hàng trăm tham số này bằng cách cho engine chơi hàng nghìn ván cờ với chính nó và điều chỉnh trọng số dựa trên kết quả.
* Bổ sung các khái niệm chiến lược: Thêm các yếu tố đánh giá phức tạp hơn như sự cơ động của quân (mobility), các quân bị ghim (pins), các mối đe dọa (threats), và không gian kiểm soát.
* Sử dụng Mạng nơ-ron: Thay thế hoặc bổ sung hàm lượng giá thủ công bằng một mạng nơ-ron nhỏ (NNUE - Efficiently Updatable Neural Network) như trong Stockfish để có một hàm lượng giá mạnh mẽ và linh hoạt hơn.

## 6.3. Cải tiến kiến trúc và hiệu năng

* Chuyển sang Ngôn ngữ Biên dịch: Viết lại các phần cốt lõi (tìm kiếm, lượng giá) bằng C++ hoặc Rust sẽ mang lại sự gia tăng hiệu suất (NPS) lên đến hàng chục hoặc hàng trăm lần, cho phép engine tìm kiếm sâu hơn đáng kể trong cùng một khoảng thời gian.
* Sử dụng Bitboards: Thay thế các cấu trúc dữ liệu của python-chess bằng cách tự triển khai biểu diễn bàn cờ bằng bitboard (số nguyên 64-bit). Điều này cho phép thực hiện các phép toán sinh nước đi và kiểm tra tấn công cực kỳ nhanh chóng thông qua các thao tác bitwise.
* Sử dụng Cpython hoặc Numba kết hợp với Bitboards để cho rất hiệu suất ngang ngửa C-LEVEL

# CHƯƠNG 7: KẾT LUẬN

## 7.1. Tổng kết kết quả

Dự án đã thành công trong việc xây dựng một engine cờ vua hoàn chỉnh, không chỉ triển khai các thuật toán tìm kiếm kinh điển mà còn tích hợp một hệ thống phức tạp các kỹ thuật tối ưu hóa hiện đại. Engine đã chứng tỏ khả năng phân tích sâu sắc các thế cờ thông qua sự kết hợp giữa một hàm tìm kiếm hiệu quả và một hàm lượng giá đa diện. Mặc dù được triển khai bằng Python, một ngôn ngữ thông dịch, các nguyên tắc và kỹ thuật được áp dụng là nền tảng cho hầu hết các engine cờ vua hàng đầu thế giới.

## 7.2. Bài học kinh nghiệm

* Sức mạnh của sự kết hợp: Một engine cờ vua mạnh không đến từ một thuật toán duy nhất, mà từ sự tương tác hài hòa của hàng chục kỹ thuật và heuristic nhỏ. Cắt tỉa Alpha-Beta là nền tảng, nhưng sức mạnh thực sự đến từ Sắp xếp nước đi, Bảng băm, và các kỹ thuật cắt tỉa khác.
* Hàm lượng giá là Trái tim Chiến lược: "Tầm nhìn" của engine được quyết định hoàn toàn bởi hàm lượng giá. Một hàm lượng giá tốt có thể hướng dẫn thuật toán tìm kiếm đến các nhánh hứa hẹn, ngay cả ở độ sâu thấp.
* Tối ưu hóa là không ngừng: Luôn có không gian để cải thiện, từ việc tinh chỉnh một hằng số nhỏ trong hàm lượng giá đến việc tái cấu trúc toàn bộ thuật toán tìm kiếm. Lập trình cờ vua là một quá trình lặp đi lặp lại của việc thử nghiệm, đo lường và cải tiến.

Dự án này không chỉ là một sản phẩm phần mềm mà còn là một hành trình học hỏi sâu sắc về Trí tuệ Nhân tạo, thuật toán và nghệ thuật giải quyết vấn đề phức tạp.

# TÀI LIỆU THAM KHẢO

1. Chess Programming Wiki (CPW): Nguồn tài liệu toàn diện và uy tín nhất về mọi khía cạnh của lập trình cờ vua. (chessprogramming.org)
2. Tài liệu thư viện python-chess: Hướng dẫn chi tiết về cách sử dụng thư viện để quản lý trạng thái cờ vua. (python-chess.readthedocs.io)
3. Mã nguồn Stockfish: Nghiên cứu mã nguồn của engine cờ vua mã nguồn mở mạnh nhất thế giới để học hỏi các kỹ thuật tiên tiến. (github.com/official-stockfish/Stockfish)