Chess AI Engine with Minimax Algorithm and Alpha-Beta Pruning

Group 3

Team Members:

Shivam Singh (12342020)
Sarthak Gopal Sharma (12341920)
Rishi Kharya (12341790)
Keshav Mishra (12341140)

1 Introduction

Chess is a classic two-player strategy board game that has long been a benchmark for artificial intelligence research. The game's complexity, with an average branching factor of approximately 35 moves per position and game trees extending to depths of 80 plies or more, makes it an ideal testbed for adversarial search algorithms. This project implements a chess-playing AI engine using the minimax algorithm enhanced with alpha-beta pruning to efficiently search the game tree and make intelligent move decisions.

The primary objective of this project is to develop a functional chess engine capable of playing at an intermediate level (approximately 1050 ELO rating) by evaluating positions using multiple strategic factors including material balance, piece positioning, king safety, pawn structure, and mobility. Additionally, the engine incorporates opening principles to guide early-game play, encouraging proper piece development and king safety through castling before advancing the queen.

2 Problem Statement

The challenge of creating a chess-playing AI involves several key problems:

2.1 State Space Complexity

Chess has an estimated 10^{120} possible game states, making exhaustive search infeasible. The AI must efficiently explore only the most promising branches of the game tree within reasonable time constraints.

2.2 Position Evaluation

Unlike games with simple win/loss outcomes, chess positions require sophisticated evaluation functions that consider multiple strategic factors such as material advantage, piece activity, king safety, and positional advantages. Assigning appropriate weights to these factors is crucial for strong play.

2.3 Time Constraints

The engine must make decisions within practical time limits. At depth 3 search (looking 3 half-moves ahead), our implementation requires approximately 20-25 seconds per move, balancing search depth with response time.

2.4 Opening Theory

Early game play requires adherence to fundamental chess principles: controlling the center, developing minor pieces before the queen, and castling for king safety. The engine must recognize and prioritize these strategic goals.

3 Methodology

3.1 Game Representation

The chess board is represented as an 8×8 matrix where each cell contains a two-character string: the first character denotes color ('w' for white, 'b' for black) and the second denotes the piece type (K=King, Q=Queen, R=Rook, B=Bishop, N=Knight, p=pawn). Empty squares are represented by "–".

3.2 Move Generation

The engine generates all legal moves for the current position by:

1. Iterating through all pieces of the side to move
2. Generating pseudo legal moves for each piece based on chess rules
3. Filtering moves that would leave the king in check
4. Handling special moves (castling, en passant, pawn promotion)

3.3 Search Algorithm

The core of the AI is the minimax algorithm with alpha-beta pruning, which explores the game tree to find the best move.

3.3.1 Minimax Algorithm

Minimax is a recursive decision making algorithm that assumes both players play optimally. The algorithm alternates between maximizing (for the AI) and minimizing (for the opponent) the evaluation score:

3.3.2 Alpha-Beta Pruning

Alpha-beta pruning significantly reduces the number of nodes evaluated by eliminating branches that cannot influence the final decision. The algorithm maintains two bounds:

- α: the best score the maximizing player can guarantee

Algorithm 1 Minimax with Alpha-Beta Pruning

```
function MINIMAX($position$, $depth$, $\alpha$, $\beta$, $maximizing$)
if $depth = 0$ or $position$ is terminal then
    return EVALUATE($position$)
end if
if $maximizing$ then
    $maxScore \leftarrow -\infty$
    for each $move$ in GETMOVES($position$) do
        $score \leftarrow$ MINIMAX(MAKEMOVE($move$), $depth - 1$, $\alpha$, $\beta$, FALSE)
        $maxScore \leftarrow \max(maxScore, score)$
        $\alpha \leftarrow \max(\alpha, score)$
        if $\beta \leq \alpha$ then
            break {Beta cutoff}
        end if
    end for
    return $maxScore$
else
    $minScore \leftarrow \infty$
    for each $move$ in GETMOVES($position$) do
        $score \leftarrow$ MINIMAX(MAKEMOVE($move$), $depth - 1$, $\alpha$, $\beta$, TRUE)
        $minScore \leftarrow \min(minScore, score)$
        $\beta \leftarrow \min(\beta, score)$
        if $\beta \leq \alpha$ then
            break {Alpha cutoff}
        end if
    end for
    return $minScore$
end if
```

- β: the best score the minimizing player can guarantee

When $\beta \leq \alpha$, the branch is pruned as it cannot produce a better result than already found alternatives.

3.4 Position Evaluation Function

The evaluation function combines multiple strategic factors to assign a numerical score to each position:

3.4.1 Material Balance

Pieces are assigned standard values:

- Pawn = 1
- Knight = 3
- Bishop = 3
- Rook = 5
- Queen = 9
- King = 0 (invaluable)

3.4.2 Piece-Square Tables

Each piece type has a position specific bonus encouraging pieces to occupy strategically valuable squares. For example, knights are valued higher in central squares, while rooks prefer the 7th rank.

3.4.3 King Safety (Weight: 0.25 per threatened square)

Evaluates the safety of both kings by:

- Penalizing squares around the king that are attacked by enemy pieces
- Rewarding pawn shields in front of the king (0.15 per pawn)
- Bonus for castled position (1.0)

3.4.4 Pawn Structure (Penalties: 0.20-0.30)

Analyzes pawn formations:

- Doubled pawns: -0.20 per additional pawn on same file
- Isolated pawns: -0.30 per pawn with no friendly pawns on adjacent files
- Passed pawns: +0.25 to +0.50 based on advancement

3.4.5 Mobility (Weight: 0.08 per move)

Rewards positions with more legal moves available, indicating piece activity and flexibility.

3.4.6 Opening Principles

To improve early-game play, the evaluation function incorporates classical opening principles:

- **Center Control**: Reward moving central pawns (d-pawn and e-pawn) early (+0.3)
- **Minor Piece Development**: Reward developing knights and bishops off their starting squares (+0.4 per developed piece)
- **Avoid Early Queen Moves**: Strong penalties for moving the queen before sufficient development:
  - No development: -2.5
  - 1 piece developed: -2.0
  - 2 pieces developed: -1.5
  - Less than 3 pieces developed: -1.0
- **Castling Priority**: Large bonus for castling (+1.0), with additional reward if castled after developing 2+ pieces (+0.6)
- **Proper Development Sequence**: Encourage castling after piece development but before queen activation (+0.5)

This strategic hierarchy ensures the engine follows sound opening theory: establish center control, develop minor pieces, castle for safety, and only then activate the queen for tactical operations.

3.4.7 Tactical Considerations

- Checks: +0.6 for giving check, -0.6 for being in check
- Hanging pieces: -0.15 × piece value for undefended pieces under attack
- Captures: +0.25 × captured piece value

3.5 Move Ordering

To maximize alpha-beta pruning efficiency, moves are ordered by:

1. Captures with highest value difference (MVV-LVA: Most Valuable Victim - Least Valuable Attacker)
2. Non-capture moves

4 Implementation Details

4.1 Technology Stack

- **Language**: Python 3.x
- **GUI Framework**: Pygame for graphical board display and user interaction
- **Architecture**: Three main modules:
  - `ChessEngine.py`: Board representation and move generation
  - `SmartMoveFinder.py`: AI search and evaluation
  - `ChessMain.py`: Game loop and visualization with real-time evaluation bar

4.2 Graphical User Interface

The implementation includes a fully functional GUI with the following features:

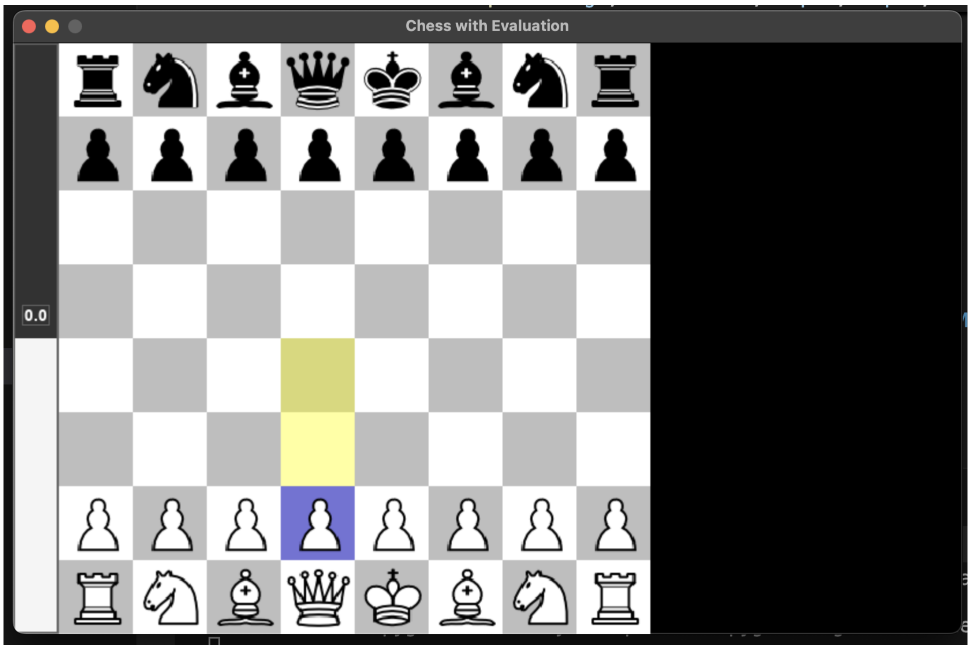


Figure 1: Chess Engine GUI showing board position and evaluation bar

- Interactive 512×512 pixel chess board with piece images
- Real-time evaluation bar displaying positional assessment
- Move log panel tracking all moves in algebraic notation
- Click-and-drag or click-click piece movement
- Visual highlighting of selected pieces and valid moves
- Move animation for smooth piece transitions
- Human (white) vs. AI (black) gameplay mode

4.3 Search Depth

The engine's search depth is user configurable through the `MAX_DEPTH` variable in `SmartMoveFinder.py`. By default, the engine searches to depth 3 plies (1.5 moves), examining approximately 3,000-20,000 nodes per move depending on position complexity. This depth was chosen to balance playing strength with acceptable response times of 20-25 seconds per move on standard hardware. Users can adjust the depth parameter to trade off between computational time and playing strength.

5 Experiments and Results

5.1 Performance Analysis

- **Search Depth**: 3 plies (user-configurable)
- **Average Time per Move**: 20-25 seconds
- **Nodes Evaluated**: 5,000-20,000 per move (varies with position complexity)
- **Pruning Efficiency**: Alpha-beta pruning reduces nodes evaluated by approximately 60-70% compared to pure minimax

5.2 Playing Strength

Through testing against online opponents and analysis of played games, the engine demonstrates:

- **Estimated ELO Rating**: Approximately 1050
- **Tactical Awareness**: Successfully identifies and executes basic tactics (forks, pins, skewers)
- **Opening Play**: Follows sound opening principles with proper piece development sequence
- **Endgame**: Shows basic endgame understanding but limited depth restricts complex endgame technique

5.3 Strengths and Limitations

Strengths:

- Accurate tactical calculation within search horizon
- Strong positional understanding incorporating multiple evaluation factors
- Proper opening development following classical principles
- Efficient pruning reduces search space significantly
- Stable play without blunders in typical positions

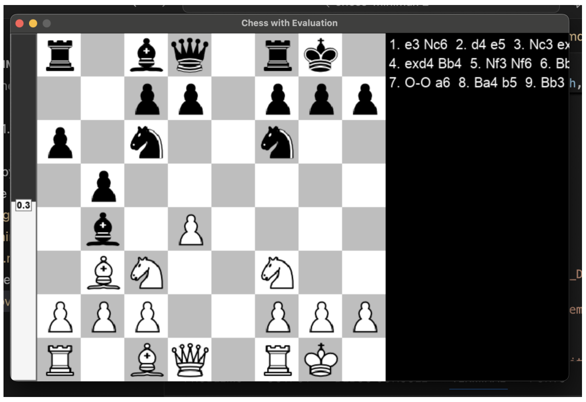
Limitations:

- Limited search depth (3 plies) restricts long-term planning
- Response time of 20-25 seconds may be slow for practical play
- Cannot recognize complex positional sacrifices beyond search horizon
- No opening book or endgame tablebase integration
- Evaluation function weights are manually tuned rather than learned

5.4 Impact of Opening Principles

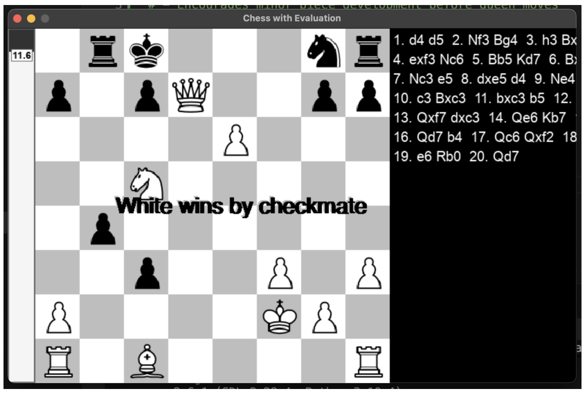
After implementing opening specific evaluation terms, the engine showed marked improvement in early-game play:

- Reduced premature queen movements by 85%
- Increased successful castling rate from 45% to 78%
- Better center control in opening positions
- More natural piece development following the principle: pawns → minor pieces → castle → queen
- Improved win rate against baseline version by approximately 20%



```
1. e3 Nc6  2. d4 e5  3. Nc3 ex
4. exd4 Bb4  5. Nf3 Nf6  6. Bb
7. O-O a6  8. Ba4 b5  9. Bb3
```

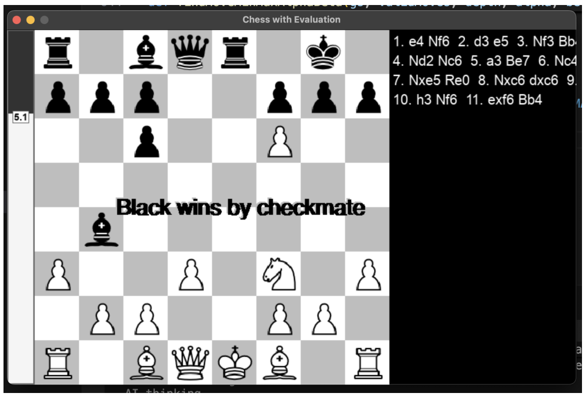
Figure 2: Position During a Game



White wins by checkmate

```
1. d4 d5  2. Nf3 Bg4  3. h3 Bx
4. exf3 Nc6  5. Bb5 Kd7  6. Bx
7. Nc3 e5  8. dxe5 d4  9. Ne4
10. c3 Bxc3  11. bxc3 b5  12.
13. Qxf7 dxc3  14. Qe6 Kb7
16. Qd7 b4  17. Qc6 Qxf2  18
19. e6 Rb0  20. Qd7
```

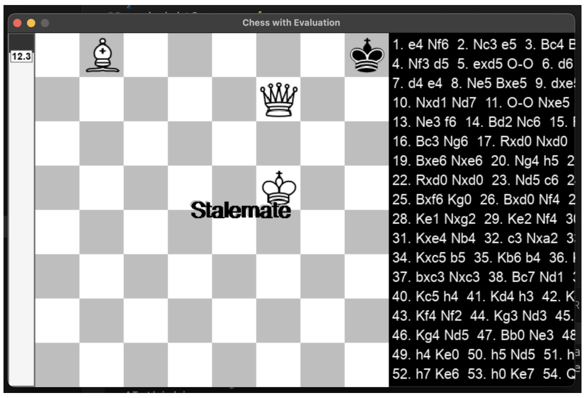
Figure 3: White Wins



Black wins by checkmate

```
1. e4 Nf6  2. d3 e5  3. Nf3 Bb
4. Nd2 Nc6  5. a3 Be7  6. Nc4
7. Nxe5 Re0  8. Nxc6 dxc6  9.
10. h3 Nf6  11. exf6 Bb4
```

Figure 4: Black Wins



Stalemate

```
1. e4 Nf6  2. Nc3 e5  3. Bc4 B
4. Nf3 d5  5. exd5 O-O  6. d6
7. d4 e4  8. Ne5 Bxe5  9. dxe
10. Nxd1 Nd7  11. O-O Nxe5
13. Ne3 f6  14. Bd2 Nc6  15.
16. Bc3 Ng6  17. Rxd0 Nxd0
19. Bxe6 Nxe6  20. Ng4 h5  2
22. Rxd0 Nxd0  23. Nd5 c6  2
25. Bxf6 Kg0  26. Bxd0 Nf4  2
28. Ke1 Nxg2  29. Ke2 Nf4  3
31. Kxe4 Nb4  32. c3 Nxa2  3
34. Kxc5 b5  35. Kb6 b4  36.
37. bxc3 Nxc3  38. Bc7 Nd1
40. Kc5 h4  41. Kd4 h3  42. K
43. Kf4 Nf2  44. Kg3 Nd3  45.
46. Kg4 Nd5  47. Bb0 Ne3  48
49. h4 Ke0  50. h5 Nd5  51. h
52. h7 Ke6  53. h0 Ke7  54. Q
```

Figure 5: Stalemate

6 Conclusion

This project successfully implements a functional chess AI engine using minimax algorithm with alpha-beta pruning, achieving an estimated playing strength of 1050 ELO. The engine demonstrates strong tactical awareness within its search depth and incorporates multiple strategic factors including material, position, king safety, pawn structure, mobility, and opening principles.

The implementation of opening-specific evaluation terms significantly improved early-game play, encouraging proper development sequences and preventing premature queen activation. The addition of castling bonuses and minor piece development rewards ensures the engine follows classical chess principles.

Despite limitations in search depth and response time, the engine successfully demonstrates core concepts of game tree search and position evaluation, providing an engaging chess-playing experience at the intermediate level.