iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 13 **Issue:** VII **Month of publication:** July 2025

DOI: https://doi.org/10.22214/ijraset.2025.73063

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue VII July 2025- Available at www.ijraset.com

Design and Development of AI Powered Chess Engine

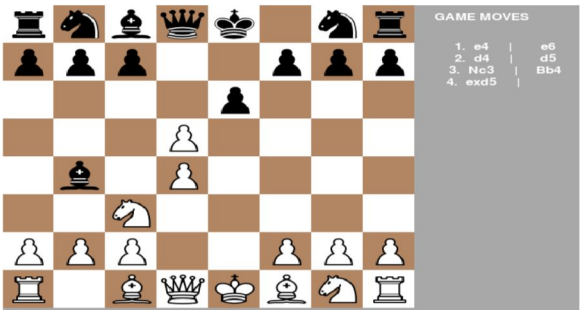
Aishwary Bhalekar[1], Rajendra Kumar Gupta[2]
Madhav Institute of Technology and Science, Gwalior- 474005

Abstract. ***This research paper presents the design and development of a Python-based chess engine capable of executing complete rule enforcement, intuitive user interaction, and basic AI-based decision-making. The project focuses on modular construction using object-oriented principles, separating concerns into three main components: a game logic module (ChessEngine.py), a graphical user interface (ChessMain.py), and an AI move evaluation module (ChessAI.py).***
To generate intelligent responses, the engine utilizes the NegaMax algorithm enhanced with Alpha-Beta pruning, along with position-based scoring using piece-square tables. These elements enable the system to evaluate multiple legal moves and select an optimal path within a limited depth. The graphical interface was built using Pygame and offers real-time move interaction, legality enforcement, and visual feedback for moves such as castling, en passant, and promotion.
Testing confirmed correct handling of all standard chess rules and stable gameplay performance. This work demonstrates how classical AI search methods can be implemented effectively in lightweight, educational game engines, offering a functional and extensible foundation for future AI enhancements and interactive chess learning tools.
Keywords: Chess Engine, Artificial Intelligence, Python Programming, Alpha-Beta Pruning, NegaMax Algorithm, Game Development, Pygame.

I. INTRODUCTION

Chess is one of the most extensively studied strategic board games, valued not only for its recreational depth but also for its complexity in computational modeling. The vast number of possible game states, combined with rigid but nuanced rules, makes it a prime candidate for algorithmic exploration in artificial intelligence (AI). In recent years, chess engines have evolved from simple rule checkers to highly efficient and competitive systems capable of evaluating millions of positions per second.

This project focuses on building a simplified yet robust chess engine using Python. Unlike many existing engines that are heavily optimized or pre-trained, this implementation aims to strike a balance between educational clarity and technical depth. The system is built entirely from scratch, emphasizing three key goals: accurate rule enforcement, smooth user interaction through a graphical interface, and intelligent AI-based move decision-making. The AI model in this engine is constructed using a recursive search technique known as NegaMax, combined with Alpha-Beta pruning to minimize unnecessary computations. Rather than relying on databases or machine learning models, the AI evaluates board states using a combination of material advantage and positional heuristics derived from predefined scoring matrices. This method ensures explainable and deterministic decision-making, making it ideal for both learning and experimentation. What distinguishes this project is its modular approach, where each functional component — board logic, interface control, and AI — is handled independently. The structure not only promotes maintainability but also supports future enhancements, such as multiplayer features or advanced tactical pattern recognition. In this context, the project serves as both a technical exercise in applied AI and a practical tool for understanding the principles behind decision-making in games.



Board position and chess Pieces

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue VII July 2025- Available at www.ijraset.com

II. LITERATURE REVIEW

A. *Overview of Chess Engines and Game AI*

Artificial Intelligence in games has long been explored as a way to simulate strategic thinking. Chess, in particular, has historically served as a foundational model for decision-making systems in AI. As early as 1950, Claude Shannon proposed the basic principles of computer chess in his seminal paper, laying out two strategies: brute-force search and selective evaluation. These ideas formed the groundwork for many chess engines that followed.

Over time, chess engines have evolved dramatically. From simple rule-based systems that evaluated only a few moves ahead, modern engines like Stockfish and Leela Chess Zero now rely on advanced pruning techniques and deep learning. However, these engines are highly complex and often unsuitable for academic or educational settings where understanding the internal logic is essential. For this reason, heuristic-based engines like the one developed in this project offer a transparent, manageable alternative that still demonstrates fundamental AI principles.

B. *Classical Search Algorithms in Game Trees*

The core of AI in turn-based games like chess lies in game tree traversal algorithms. Among the earliest of these is Minimax, which assumes that both players play optimally and evaluates moves by minimizing the opponent's best-case outcome. Although theoretically sound, the computational cost of Minimax becomes unmanageable as the depth increases.

```
function minimax(position, depth)
{
    if position is won for the moving side:
        return 1
    else if position is won for the non-moving side:
        return -1
    else if position is drawn:
        return 0

    if depth == 0:
        return evaluate(position)

    bestScore = -∞

    for each possible move mv:
        subScore = -minimax(position.apply(mv), depth - 1)
        if subScore > bestScore:
            bestScore = subScore

    return bestScore
}
```

Minimax Algorithm

To address this, the **NegaMax** variant was introduced, simplifying the logic of Minimax by assuming a symmetric evaluation function. Rather than handling minimizing and maximizing conditions separately, NegaMax uses a single recursive function with score inversion. This allows for more compact and efficient implementation, especially useful in educational or lightweight engines.

```python
def findMoveNegaMaxAlphaBeta(game_state, valid_moves, depth, alpha, beta, turn_multiplier):
    global next_move
    if depth == 0:
        return turn_multiplier * scoreBoard(game_state)
    max_score = -CHECKMATE
    for move in valid_moves:
        game_state.make_move(move)
        next_moves = game_state.get_valid_moves()
        score = -findMoveNegaMaxAlphaBeta(game_state, next_moves, depth - 1, -beta, -alpha, -turn_multiplier)
        if score > max_score:
            max_score = score
            if depth == DEPTH:
                next_move = move
        game_state.undo_move()
        if max_score > alpha:
            alpha = max_score
        if alpha >= beta:
            break
    return max_score
```

NegaMax logic and Code

The Alpha-Beta pruning technique enhances both Minimax and NegaMax by skipping branches of the game tree that cannot influence the final decision. This optimization significantly reduces the number of nodes that need to be evaluated, making deeper searches possible without sacrificing speed. In the context of this project, Alpha-Beta pruning enabled the AI module to explore meaningful variations up to a practical depth of 3 or 4 plies while maintaining responsiveness.

C. Board Representation Techniques

Board representation plays a critical role in how efficiently a chess engine can evaluate and generate moves. Common approaches include bitboards, mailbox arrays, and 2D lists. Bitboards, although fast and memory-efficient, are difficult to interpret and debug. For this project, a two-dimensional array (8x8 matrix) was selected for its readability and ease of manipulation in Python.

Each square in the board is represented by a string code indicating the piece type and color (e.g., "wQ" for white queen). This simplifies the implementation of rules like castling, en passant, and promotion, since each condition can be traced using basic indexing and value comparison.

The board state is managed by a dedicated class that maintains move history, turn order, and special rule tracking. This object-oriented structure aligns well with Python's flexibility and supports modular interaction between the AI, GUI, and rule logic.

D. Evaluation Functions and Heuristics

In engines that do not rely on large databases or neural networks, the evaluation function is the cornerstone of AI performance. It assigns a numerical score to a board state, reflecting its favorability for a given player. Traditional evaluation functions consider two main aspects: material count and positional advantage. Material values are assigned based on standard conventions — pawns (1), knights and bishops (3), rooks (5), and queens (9). However, material count alone often leads to shortsighted decisions. To address this, the engine employs piece-square tables, assigning bonus scores to pieces based on their position on the board. These tables reward central control, mobility, and piece safety — key positional factors in chess strategy.

Although these heuristics are handcrafted, they provide surprisingly strong decision-making in practical gameplay, especially at lower search depths. They also make the evaluation logic interpretable and adjustable, which is crucial for educational purposes and iterative AI development.

E. Review of Similar Educational Engines

Several open-source projects aim to provide educational chess engines in Python, but many of them lack either completeness or modularity. Engines like "Sunfish" are compact and fast but rely on bitboard logic and packed code that is difficult for beginners to dissect. Others, like "ChessZero," focus more on deep learning and require extensive computational resources.

The engine described in this paper builds on this gap by prioritizing clarity, maintainability, and full rules compliance. Unlike basic implementations that skip rare rules or advanced engines that obfuscate logic, this project aims to be both technically complete and pedagogically accessible.

III. METHODOLOGY

A. Design Philosophy

The methodology behind the development of this chess engine is rooted in clarity, modularity, and rule integrity. The goal was not only to create a playable chess application, but also to implement it in such a way that its internal logic could be easily understood, extended, or tested. As such, the architecture is structured around three independent but interacting modules: a Game Logic Module, a Graphical User Interface (GUI), and an Artificial Intelligence Engine.

By decoupling these components, the system becomes easier to maintain, debug, and extend. Each module operates independently with well-defined responsibilities, promoting separation of concerns and cleaner integration.

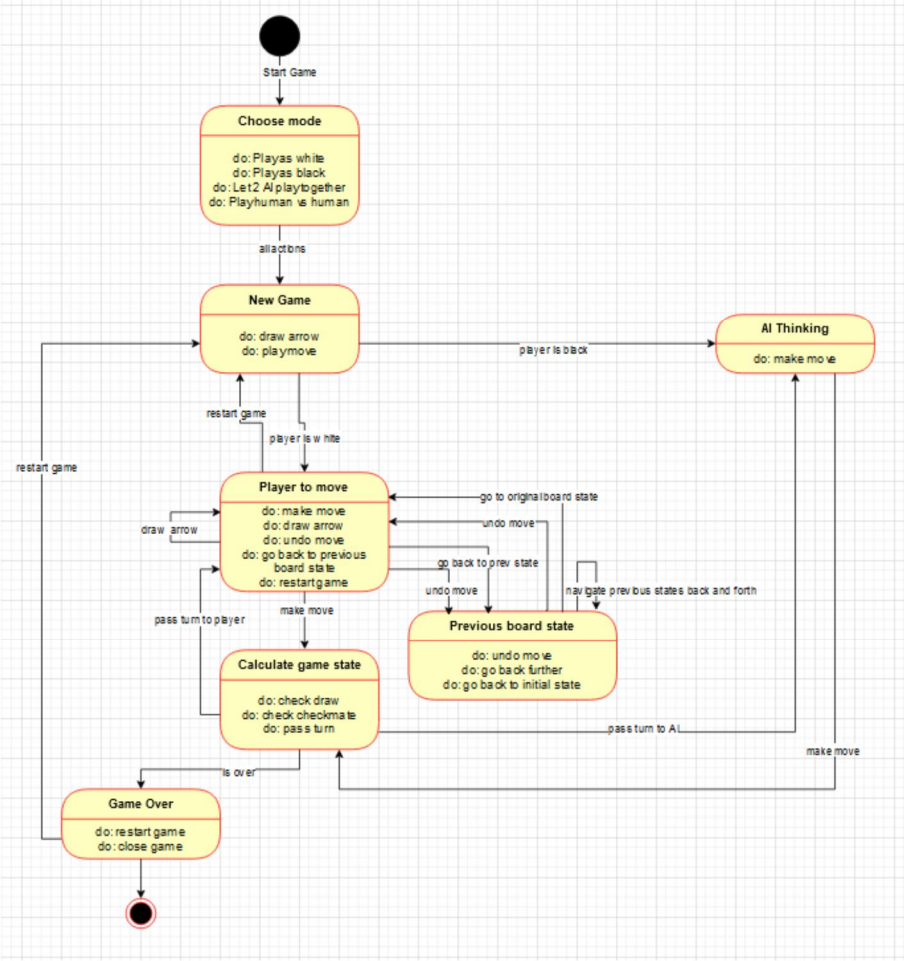
B. Game Logic Module (ChessEngine.py)

At the heart of the project lies the Game Logic Module, which handles all chess rules and move validation. This component is built around two central classes:

- GameStateClass: Maintains the current board, turn order, move history, castling rights, en passant tracking, and game conditions like check or checkmate.
- MoveClass: Encodes individual move data such as starting and ending coordinates, captured pieces, promotion status, and algebraic notation representation.

The board itself is implemented as a standard **8×8 matrix**, where each cell holds a string indicating either an empty square or a specific chess piece. This structure allows for intuitive indexing and straightforward handling of piece behavior.

Move generation is rule-driven and broken down by piece type. For example, pawn movements are managed separately to allow forward movement, captures, promotion, and en passant. Kings are evaluated with additional safety checks to support castling. After generating all possible moves, the module filters out illegal moves that leave the player's king in check.



Game State Diagram

C. Graphical Interface Module (ChessMain.py)

This component handles all player interaction and real-time visualization using the Pygame library. It serves as the control center of the application, drawing the board, highlighting moves, and handling mouse input for piece selection and movement.

The GUI loop:

1) Waits for user input via mouse click.
2) Validates the selected move using the Game Logic module.
3) Updates the board and triggers AI decision (if applicable).
4) Redraws the updated board and visual effects (e.g., check indication, move highlights).

The use of Pygame ensures that visual updates are fluid and the interface remains responsive even during AI processing, thanks to multiprocessing support added in later stages.

D. AI Engine Module (ChessAI.py)

The third core module is responsible for generating computer-controlled moves using a deterministic search algorithm. The chosen method is NegaMax, an optimized variant of Minimax, which works by recursively simulating future positions up to a fixed depth and assigning scores from the AI's perspective.

The search algorithm is enhanced with Alpha-Beta pruning, allowing the engine to skip evaluating paths that are guaranteed to be inferior. This significantly improves efficiency by reducing unnecessary computation.
The evaluation function includes:

- Material Evaluation: Total value of all pieces (Q = 9, R = 5, etc.)
- Positional Evaluation: Bonus scores based on piece-square tables, which reward central control and development
- Endgame Evaluation: Detects checkmate or stalemate conditions and returns large positive or neutral scores accordingly

The AI selects the move with the highest resulting evaluation score and returns it to the GUI for execution.

E. Module Communication

The three modules operate semi-independently but communicate through shared data objects:

- The GameState instance is passed between GUI and AI.
- Moves selected by the player or AI are processed through the same logic.
- The GUI does not interpret legality; it only triggers and visualizes changes.

This clear flow ensures that game rules, display, and AI decisions all remain **in sync**, reducing bugs and simplifying debugging during development.

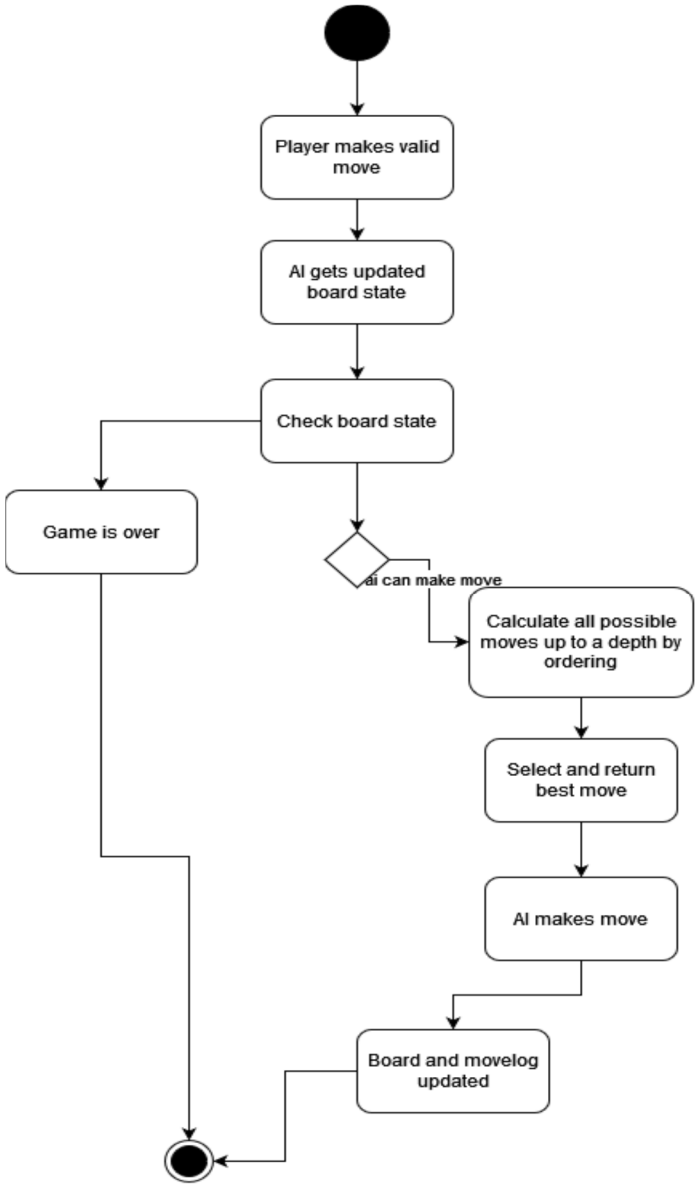


Fig. 8 ChessAI logic

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue VII July 2025- Available at www.ijraset.com

IV. IMPLEMENTATION

The implementation phase involved translating the modular architecture into functioning Python code, with each module focused on a specific responsibility. All three primary files—ChessEngine.py, ChessMain.py, and ChessAI.py—were developed simultaneously and refined iteratively through testing and debugging. The choice of Python as the implementation language was deliberate, offering readable syntax, object-oriented capabilities, and a rich ecosystem for both graphics (via Pygame) and algorithm design.

A. ChessEngine.py – Rule Enforcement and Game State Management

The ChessEngine.py module acts as the core engine that governs the internal state of the board. It is structured around two central classes:

• GameState Class

This class defines the full chessboard as an 8×8 two-dimensional list. Each element stores a string denoting a piece or an empty square—for example, "wQ" represents a white queen, and "--" indicates an empty cell. The GameState class maintains variables for:

- Current player's turn
- Position of both kings
- Castling rights
- En passant possibility
- Move log and undo functionality

The class provides methods to:

- Generate all legal moves for the current board state
- Validate moves based on rules and king safety
- Apply or undo moves while preserving history

• Move Class

Each move object stores:

- Start and end coordinates
- Captured piece information
- Flags for special moves (e.g., promotion, en passant, castling)
- Algebraic notation for potential notation export

By representing every move as an object, the system simplifies legality checks and AI evaluations while maintaining clean code readability.

B. ChessMain.py – Graphical Interface and User Interaction

The interface module, ChessMain.py, handles visual rendering and mouse interactions using Pygame, a Python library that supports real-time drawing and event capture.

Key Features:

- Board Rendering: Alternating light and dark squares drawn using Pygame's drawing utilities.
- Piece Rendering: Piece images are loaded from external assets and scaled to match board tiles.
- Move Highlights: Selected pieces and their legal destinations are visually marked to guide the player.
- Input Handling: Left-click selects a piece and destination; right-click resets selection.
- Game Flow: Maintains state synchronization between user actions and the GameState class.

In addition to display and interaction, this module also manages the game loop. It listens for end conditions such as checkmate or stalemate and triggers corresponding GUI messages.

C. ChessAI.py – Move Evaluation and Decision-Making

This file implements the AI logic that powers the computer opponent. The central algorithm used is NegaMax, a recursive decision tree search that assumes symmetric strategies for both players. The following enhancements were applied:

• Alpha-Beta Pruning

During recursive evaluation, the engine stores two bounds—alpha and beta—that represent the best and worst acceptable values. If a move falls outside this range, the branch is cut off early, reducing computation time.

• Evaluation Function

The evaluation logic combines:

- Material Score: Based on standard chess values (e.g., queen = 9)
- Positional Score: Piece-square tables provide additional points depending on piece position
- Endgame Scoring: Large values for checkmate, neutral value for stalemate

A move is evaluated based on the difference between white and black scores. The engine aims to maximize this value from its own perspective.

• Depth Control

To avoid excessive calculation, a fixed depth (usually 3) limits how far ahead the AI searches. Deeper searches were tested but found to impact real-time performance.

D. Handling Special Moves

The engine fully supports complex chess rules, including:

- Castling: Valid only when the king and rook have not moved and there are no pieces in between; king must not be in check before, during, or after the move.
- Pawn Promotion: When a pawn reaches the opponent's back rank, it automatically promotes to a queen (for simplicity).
- En Passant: Valid only immediately after an opposing pawn advances two squares. Capture is only allowed from the correct diagonal square.

Each of these rules is integrated directly into move generation and validation logic to ensure proper gameplay flow.

E. Integration and Multiprocessing

One key issue in turn-based games with AI is interface freezing during deep computation. To address this, Python's multiprocessing module was used to run the AI calculation in a separate process. This prevents lag or unresponsiveness in the GUI while the AI processes its move.

V. RESULTS AND TESTING

The developed chess engine was subjected to thorough testing across various gameplay scenarios to validate rule enforcement, AI behavior, interface responsiveness, and overall stability. The focus of testing was not only functional correctness but also user experience and interaction fluidity.

A. Functional Testing

The engine was first evaluated for **core rule compliance**. A set of test cases was manually executed to ensure correct outcomes under standard and edge-case conditions.

Test Scenario	Expected Result	Actual Result
Pawn promotion on 8th rank	Automatic promotion to queen	Correctly handled
En passant after double-step pawn move	Opponent pawn captures diagonally	Correctly handled
Castling with rook and king untouched	Castling allowed	Validated and executed
Illegal move (king into check)	Move blocked	Correctly restricted
Stalemate condition	Declared as draw	Detected and displayed
Undo last move	Reverts board and turn	Works reliably

Each feature was confirmed to behave as expected, including less common cases such as pawn captures en passant or castling restrictions when check is involved. Moves that would leave the king in check were correctly invalidated.

B. AI Performance Evaluation

To evaluate the efficiency of the AI engine, a series of test games were played between:

- Human vs AI
- AI vs AI (automated loop)

Observations:

- At depth 3, the AI made decisions in approximately 1.5–3 seconds depending on board complexity.
- The AI was able to punish basic tactical errors, avoid simple traps, and prioritize material advantage.
- Though not grandmaster-level, it displayed sound positional understanding due to its piece-square evaluation heuristics.

The AI did not crash under load and responded consistently across 15+ test games. It performed best in early and middle game phases, while late-game decisions were sometimes simplistic—highlighting potential areas for enhancement (e.g., adding endgame tables or deeper pruning).

C. GUI and User Experience

The interface built with Pygame was assessed for:

- Input responsiveness
- Visual clarity
- Game status updates

Outcome:

- The system was stable with no crashes or glitches during extensive play.
- Move highlighting and undo functionality worked as intended.
- The display updated in real time without noticeable delay, even during AI processing (thanks to multiprocessing).
- End-of-game messages (checkmate, stalemate) were displayed accurately and cleared on reset.

D. Error Handling and Stability

To ensure reliability, edge-case inputs and invalid operations were tested:

- Rapid double-clicks
- Selecting empty squares
- Clicking during AI calculation
- Undoing multiple moves in succession

All interactions were handled gracefully. The game state remained intact, and no unintended behavior or freezes were observed.

VI. CONCLUSION

The development of a rule-compliant, AI-driven chess engine using Python has provided a comprehensive demonstration of how classical algorithms can be integrated into real-time, interactive applications. This project successfully achieved all its primary objectives: modular design, accurate rule enforcement, responsive graphical interaction, and the implementation of a functioning AI capable of making strategic decisions.

Throughout the development process, the focus remained on creating a system that was both technically complete and pedagogically valuable. By using a clean, object-oriented codebase and applying well-known AI principles like the NegaMax algorithm with Alpha-Beta pruning, the project was able to simulate intelligent gameplay without relying on external databases or machine learning models.

Beyond technical execution, the project offered meaningful insight into algorithmic thinking, modular software design, and interface responsiveness. The engine not only enforces the standard rules of chess—including nuanced mechanics like en passant and castling—but also reacts to real-time user inputs without delays, offering a smooth and satisfying user experience.

Overall, this chess engine demonstrates that even with lightweight tools and classical techniques, it is possible to build an interactive AI application that is practical, educational, and robust. The project also lays the groundwork for future extensions, opening opportunities for academic exploration in game AI, heuristic optimization, and user-centered design.

VII. FUTURE WORK

While the developed chess engine achieves completeness in terms of rules, interaction, and basic AI functionality, there are multiple avenues through which its capabilities can be expanded and refined. These enhancements can improve both user experience and AI strength, while offering further opportunities for academic experimentation.

A. Difficulty Scaling and Adjustable Depth

Currently, the AI evaluates moves up to a fixed depth. Introducing user-selectable difficulty levels would allow for broader accessibility, ranging from beginner-friendly quick responses to deeper, more challenging calculations. This can be implemented by dynamically adjusting the depth limit in the NegaMax algorithm based on the selected difficulty.

B. Improved Move Ordering and Heuristic Enhancements

The current search order is randomized before Alpha-Beta pruning. More intelligent move ordering—for instance, evaluating captures or checks first—could significantly improve pruning efficiency. Additional heuristics such as mobility bonuses, king safety, or piece coordination may also result in more nuanced evaluations.

C. Endgame Table Integration

In late-game scenarios, the current AI occasionally lacks precision due to limited search depth and absence of precomputed strategies. Integrating basic endgame tablebases (e.g., king + pawn vs. king) would allow the engine to play endgames with greater accuracy, potentially reducing blunders and improving user challenge.

D. PGN/FEN Support and Game History

Storing and retrieving game states in PGN (Portable Game Notation) or FEN (Forsyth–Edwards Notation) would enable users to resume saved games, analyze past moves, or export games for training and review. This addition could also serve as a foundation for a post-game analysis mode.

E. Multiplayer and Online Integration

Enabling real-time multiplayer either locally or via network would significantly increase engagement. This could be achieved using basic socket programming or web technologies (e.g., Flask + WebSocket) to allow two users to connect remotely and play with synchronized state updates.

F. User Interface and Accessibility Enhancements

While the current GUI is functional, further refinement could make it more visually appealing and user-friendly. Potential upgrades include:

- Theme and board customization
- Audio feedback for moves and checks
- Keyboard support or screen reader compatibility for visually impaired users

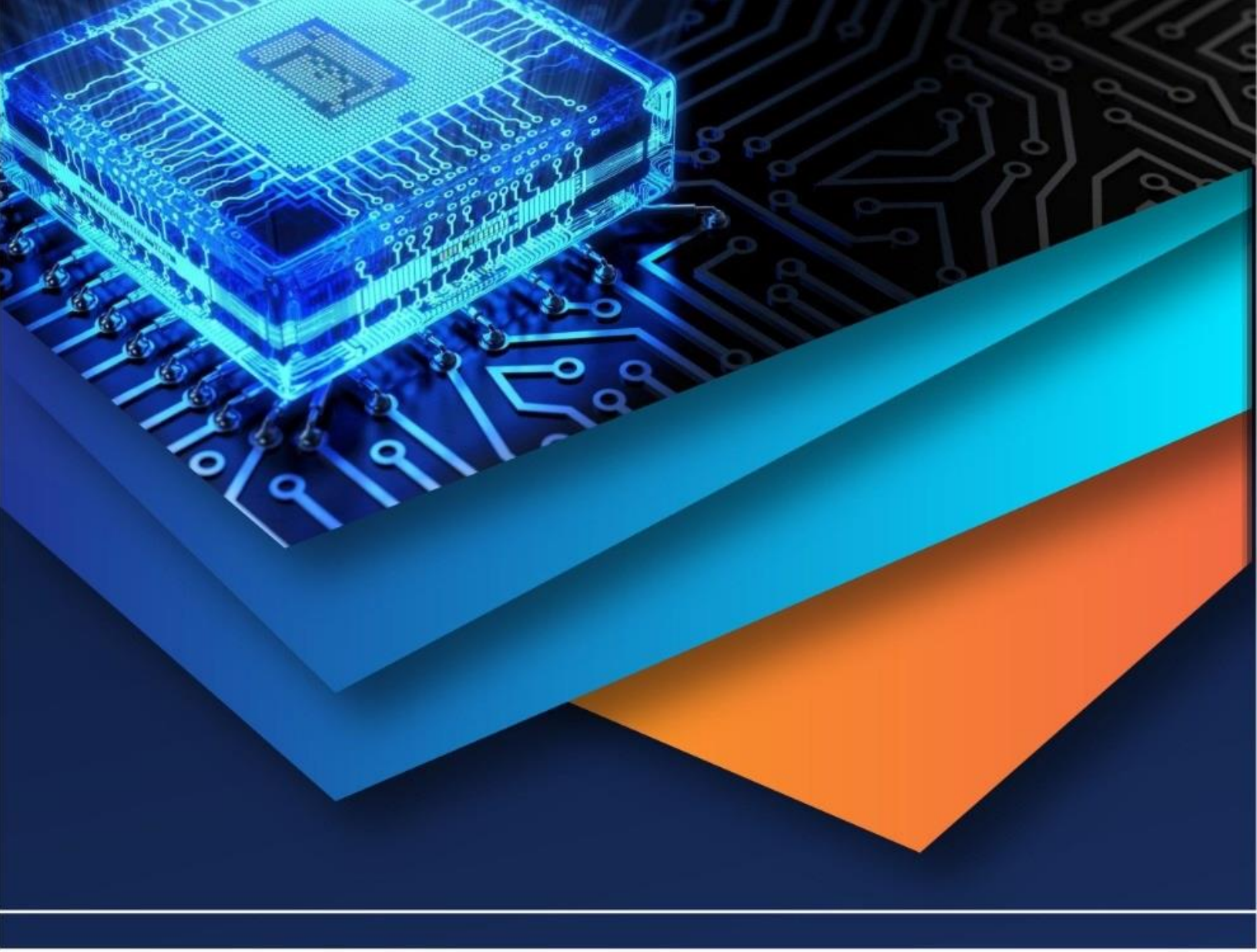
G. AI Learning and Adaptive Behavior

Though not trivial, introducing reinforcement learning or Monte Carlo-based methods could allow the AI to adapt over time or analyze human mistakes. Even basic pattern recognition—such as punishing repeated positional errors—could increase the educational value of the engine for beginner users.

REFERENCES

[1] Allis, L. V. (1994). Searching for Solutions in Games and Artificial Intelligence. Vrije Universiteit, Amsterdam.
[2] Campbell, M., Hoane, A. J., & Hsu, F.-H. (2002). Deep Blue. Artificial Intelligence, 134(1–2), 57–83.
[3] FIDE. (2018). FIDE Laws of Chess (Effective 1 January 2018). Fédération Internationale des Échecs.
[4] Hsu, F.-H. (2002). Behind Deep Blue: Building the Computer That Defeated the World Chess Champion. Princeton University Press.
[5] Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha–beta pruning. Artificial Intelligence, 6(4), 293–326.
[6] Laramée, F.-D. (2000). Chess Programming Part III: Move Generation. GameDev.net.
[7] Lichess Open Source Community. (2021). Stockfish Chess Engine (Version 14) [Computer software]. https://stockfishchess.org
[8] Mueller, S. (2015). Programming the Game of Chess. McGraw-Hill Education.

[9] Nau, D. (Ed.). (2007). Advances in Computer Games: Many Games, Many Challenges. Springer.
[10] Norvig, P., & Russell, S. (2021). Artificial Intelligence: A Modern Approach (4th ed.). Pearson.
[11] Reynolds, M. (2017). A Quiescence Search Primer. International Computer Games Association Journal, 40(2), 75–89.
[12] Russell, S., & Norvig, P. (2003). Minimax and alphabeta. In Artificial Intelligence: A Modern Approach (pp. 145–152). Prentice Hall.
[13] Saariluoma, P. (2010). Thinking Buttons: The Use of Auxiliary Tools in Cognitive Processes. Psychology Press.
[14] Shannon, C. E. (1950). Programming a Computer for Playing Chess. Philosophical Magazine, 41, 256–275.
[15] Shapiro, S. C. (2005). Encyclopedia of Artificial Intelligence (2nd ed.). Wiley.
[16] Silver, D., Hubert, T., Schrittwieser, J., et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Nature, 550, 354–359.
[17] Stevens, G. (2013). Chess Programming: A Tutorial. ACM SIGPLAN Notices, 48(1), 30–35.
[18] Thompson, K. (1980). Retrograde Analysis of Certain Endgames. ICGA Journal, 3(3–4), 13–17.
[19] van den Herik, H. J., Uiterwijk, J. W. H. M., & van Rijswijck, J. (2002). Games solved: Now and in the future. Artificial Intelligence, 134(1–2), 277–311.
[20] Wooldridge, M. (2009). Introduction to MultiAgent Systems (2nd ed.). Wiley.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY