

|  |
| --- |
| **RAJALAKSHMI INSTITUTE OF TECHNOLOGY** |
| (An Autonomous Institution, Affiliated to Anna University, Chennai) |

**DEPARTMENT OF CSE (ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)**

**ACADEMIC YEAR 2025 - 2026**

**SEMESTER III**

**ARTIFICIAL INTELLIGENCE LABORATORY**

**MINI PROJECT REPORT**

|  |  |
| --- | --- |
| **REGISTER NUMBER** | 2117240030007 |
| **NAME** | Anto Jenishia A |
| **PROJECT TITLE** | Chess AI using Minimax Algorithm and Alpha–Beta Pruning |
| **DATE OF SUBMISSION** | 29/10/2025 |
| **FACULTY IN-CHARGE** | **Mrs. M. Divya** |

**Signature of Faculty In-charge**

**INTRODUCTION**

Artificial Intelligence (AI) is a field of computer science focused on creating systems that can think, learn, and act intelligently. AI systems mimic human cognitive processes such as reasoning, problem-solving, and decision-making. These capabilities are achieved through intelligent agents that perceive their environment and act to achieve specific goals. Game-playing has always been a benchmark for testing AI strategies. Among various games, chess represents a complex decision-making environment requiring strategic reasoning and planning. AI in chess involves searching through possible moves, predicting opponent responses, and selecting the most optimal move. Building a chess-playing AI demonstrates how intelligent agents can operate in adversarial conditions where each move impacts the opponent’s future choices. It helps understand core AI topics like problem-solving agents, heuristic search, and adversarial game trees. This project aims to design and implement a Python-based Chess AI that uses the Minimax algorithm with Alpha–Beta pruning to make intelligent, strategic moves against a human player.

**PROBLEM STATEMENT**

Develop an intelligent chess-playing agent that can analyze possible board configurations, predict the opponent’s responses, and choose the best move using Minimax search and Alpha–Beta pruning techniques, while efficiently managing computational resources.

**GOAL**

**Expected Result:**  
 A functional Chess AI that plays logically and strategically against a human player using heuristic evaluation functions.

**Possibilities:**

* Explore more complex AI approaches like Monte Carlo Tree Search (MCTS) or Neural Networks for improved performance.
* Introduce adaptive difficulty levels and self-learning capabilities.

**THEORETICAL BACKGROUND**

**Chess Game:**  
 Chess is a two-player, zero-sum, deterministic game, making it ideal for applying adversarial search algorithms. Each player aims to maximize their advantage while minimizing the opponent’s advantage.

**Algorithm:**  
 The Minimax algorithm is a recursive search algorithm used in decision-making and game theory. It simulates all possible moves for both players and assumes the opponent plays optimally.  
Alpha–Beta pruning improves Minimax by eliminating branches that don’t influence the final decision, thus reducing computation time without affecting accuracy.

**Literature Survey:**

* *Deep Blue (IBM, 1997)* used Minimax and heuristic evaluation to defeat the world chess champion Garry Kasparov.
* *AlphaZero (DeepMind, 2018)* applied neural networks with Monte Carlo search for self-learning chess play.
* *GeeksforGeeks (2020)* and *Sebastian Lague’s tutorials* provide clear explanations of Minimax and pruning for practical implementation.

**Justification for Choosing the Algorithm:**  
 Minimax with Alpha–Beta pruning provides a balance between performance and simplicity, ideal for a mini project setting. It effectively models decision-making in games while being computationally manageable in Python.

**ALGORITHM EXPLANATION WITH EXAMPLE**

**Minimax Algorithm Steps:**

1. Generate all possible legal moves.
2. Simulate each move recursively.
3. Alternate between maximizing (AI) and minimizing (human) turns.
4. Evaluate each terminal state using a heuristic function.
5. Choose the move with the best evaluation score.
6. Apply Alpha–Beta pruning to ignore irrelevant branches.

**Example:**  
 If the AI can move its knight to capture a pawn (+3) or make a neutral move (0), while predicting the opponent may counterattack (-5), it compares both outcomes recursively and selects the move leading to the best net advantage

**IMPLEMENTATION AND CODE**

Language Used: Python  
Libraries: python-chess, tkinter, random, time

**SOURCE CODE:**

import tkinter as tk

import chess

import math

import random

# ------------------------------

# EVALUATION FUNCTION

# ------------------------------

def evaluate(board):

    """Evaluate board using material, mobility, and center control."""

    piece\_values = {chess.PAWN: 1, chess.KNIGHT: 3.2, chess.BISHOP: 3.3, chess.ROOK: 5, chess.QUEEN: 9, chess.KING: 100}

    score = 0

    # --- Material ---

    for piece\_type in piece\_values:

        score += len(board.pieces(piece\_type, chess.WHITE)) \* piece\_values[piece\_type]

        score -= len(board.pieces(piece\_type, chess.BLACK)) \* piece\_values[piece\_type]

    # --- Mobility ---

    white\_mobility = len(list(board.legal\_moves)) if board.turn == chess.WHITE else 0

    board.turn = chess.BLACK

    black\_mobility = len(list(board.legal\_moves))

    board.turn = not board.turn  # restore

    score += 0.1 \* (white\_mobility - black\_mobility)

    # --- Center Control ---

    center\_squares = [chess.D4, chess.D5, chess.E4, chess.E5]

    for sq in center\_squares:

        piece = board.piece\_at(sq)

        if piece:

            if piece.color == chess.WHITE:

                score += 0.3

            else:

                score -= 0.3

    # --- Reward giving check ---

    if board.is\_check():

        score += 0.5

    return score

# ------------------------------

# MINIMAX WITH ALPHA-BETA

# ------------------------------

def minimax(board, depth, alpha, beta, maximizing):

    # Handle terminal states smartly

    if board.is\_game\_over():

        if board.is\_checkmate():

            # If it's white to move and checkmate, that means black won

            return (-9999 if board.turn == chess.WHITE else 9999), None

        elif board.is\_stalemate():

            return -200, None  # discourage stalemates

        else:

            return 0, None  # draws, repetition, insufficient material

    if depth == 0:

        return evaluate(board), None

    legal\_moves = list(board.legal\_moves)

    if not legal\_moves:

        return evaluate(board), None

    best\_move = None

    if maximizing:

        max\_eval = -math.inf

        for move in legal\_moves:

            board.push(move)

            eval\_score, \_ = minimax(board, depth - 1, alpha, beta, False)

            board.pop()

            if eval\_score > max\_eval:

                max\_eval = eval\_score

                best\_move = move

            alpha = max(alpha, eval\_score)

            if beta <= alpha:

                break

        if best\_move is None:

            best\_move = random.choice(legal\_moves)

        return max\_eval, best\_move

    else:

        min\_eval = math.inf

        for move in legal\_moves:

            board.push(move)

            eval\_score, \_ = minimax(board, depth - 1, alpha, beta, True)

            board.pop()

            if eval\_score < min\_eval:

                min\_eval = eval\_score

                best\_move = move

            beta = min(beta, eval\_score)

            if beta <= alpha:

                break

        if best\_move is None:

            best\_move = random.choice(legal\_moves)

        return min\_eval, best\_move

# ------------------------------

# CHESS GUI CLASS

# ------------------------------

class ChessGUI:

    def \_\_init\_\_(self, root):

        self.root = root

        self.board = chess.Board()

        self.square\_size = 64

        self.selected\_square = None

        # --- GUI Setup ---

        self.canvas = tk.Canvas(root, width=8\*self.square\_size, height=8\*self.square\_size)

        self.canvas.pack()

        self.status = tk.Label(root, text="AI (White) is making the first move...", font=("Arial", 14))

        self.status.pack()

        self.symbols = {

            'r': '♜', 'n': '♞', 'b': '♝', 'q': '♛', 'k': '♚', 'p': '♟',

            'R': '♖', 'N': '♘', 'B': '♗', 'Q': '♕', 'K': '♔', 'P': '♙'

        }

        self.canvas.bind("<Button-1>", self.on\_click)

        self.draw\_board()

        # AI starts first

        self.root.after(800, self.ai\_move)

    # --------------------------

    # DRAW BOARD

    # --------------------------

    def draw\_board(self):

        self.canvas.delete("all")

        colors = ["#EEEED2", "#769656"]

        for r in range(8):

            for c in range(8):

                color = colors[(r + c) % 2]

                if self.selected\_square == chess.square(c, 7 - r):

                    color = "#BACA44"  # highlight selection

                self.canvas.create\_rectangle(

                    c \* self.square\_size, r \* self.square\_size,

                    (c + 1) \* self.square\_size, (r + 1) \* self.square\_size,

                    fill=color, outline=""

                )

                piece = self.board.piece\_at(chess.square(c, 7 - r))

                if piece:

                    self.canvas.create\_text(

                        c \* self.square\_size + self.square\_size // 2,

                        r \* self.square\_size + self.square\_size // 2,

                        text=self.symbols[piece.symbol()],

                        font=("Arial", 36) )

        self.root.update()

    # --------------------------

    # PLAYER MOVE (BLACK)

    # --------------------------

    def on\_click(self, event):

        if self.board.is\_game\_over():

            self.status.config(text="Game Over!")

            return

        if self.board.turn != chess.BLACK:

            return  # not your turn

        row = event.y // self.square\_size

        col = event.x // self.square\_size

        square = chess.square(col, 7 - row)

        if self.selected\_square is None:

            piece = self.board.piece\_at(square)

            if piece and piece.color == chess.BLACK:

                self.selected\_square = square

        else:

            move = chess.Move(self.selected\_square, square)

            if move in self.board.legal\_moves:

                self.board.push(move)

                self.selected\_square = None

                self.draw\_board()

                if self.board.is\_checkmate():

                    self.status.config(text="You won! (Checkmate)")

                    return

                elif self.board.is\_stalemate():

                    self.status.config(text="Draw (Stalemate)")

                    return

                self.status.config(text="AI thinking...")

                self.root.after(500, self.ai\_move)

            else:

                self.selected\_square = None

        self.draw\_board()

    # --------------------------

    # AI MOVE (WHITE)

    # --------------------------

    def ai\_move(self):

        if self.board.is\_game\_over():

            self.status.config(text="Game Over!")

            return

        self.status.config(text="AI thinking...")

        self.root.update()

        \_, best\_move = minimax(self.board, 3, -math.inf, math.inf, True)

        if best\_move:

            self.board.push(best\_move)

        self.draw\_board()

        if self.board.is\_checkmate():

            self.status.config(text="AI wins by checkmate!")

        elif self.board.is\_stalemate():

            self.status.config(text="Draw (Stalemate)")

        else:

            self.status.config(text="Your move (Black ♟)")

# ------------------------------

# MAIN FUNCTION

# ------------------------------

def main():

    root = tk.Tk()

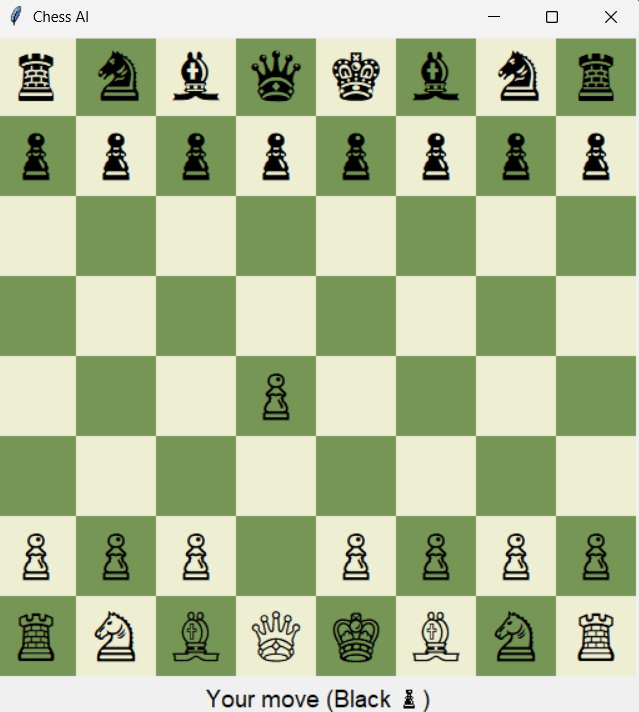
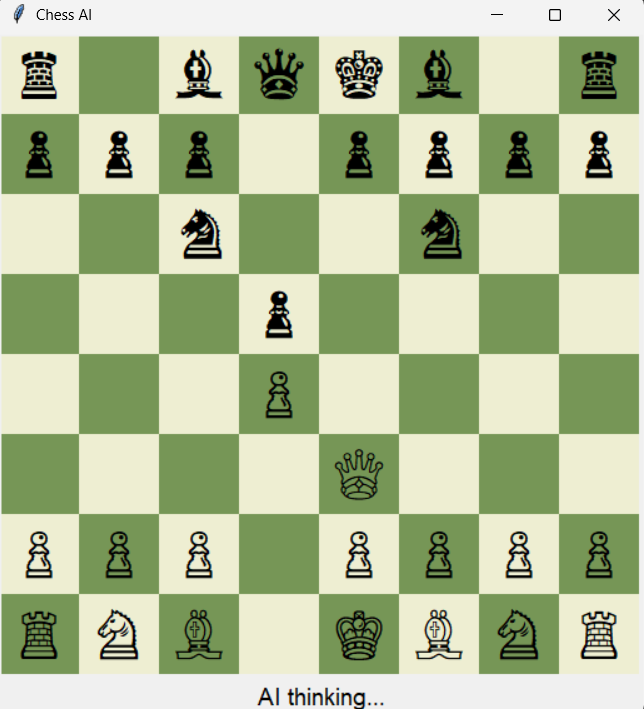
    root.title("Chess AI")

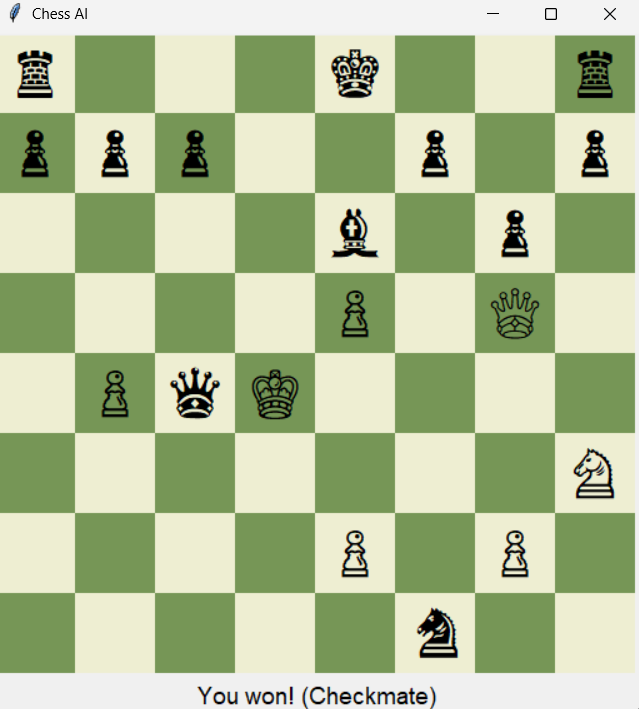
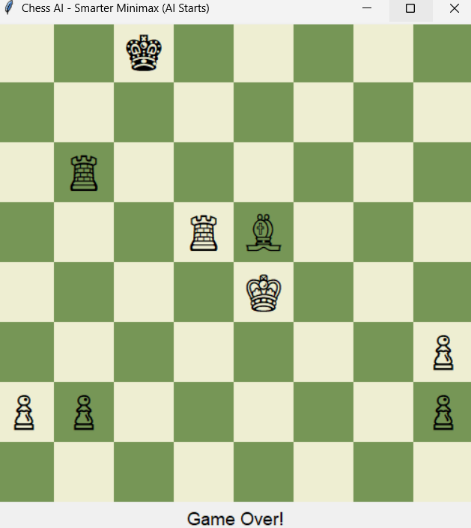
    app = ChessGUI(root)

    root.mainloop()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**OUTPUT**

****

Explanation:  
 The GUI displays a functional chessboard. After each human move, the AI calculates possible responses, applies Minimax search with pruning, evaluates outcomes, and makes its move. The AI successfully avoids illegal moves, handles checkmate/stalemate conditions, and responds strategically**.**

**RESULTS AND FUTURE ENHANCEMENT**

Results:

* A fully functional chess AI capable of intelligent gameplay.
* Demonstrates search-based decision-making, heuristic evaluation, and rational agent behaviour.
* Handles terminal conditions such as checkmate and stalemate correctly.

Future Enhancements:

* Add adaptive difficulty by adjusting search depth.
* Integrate machine learning for improved evaluation.
* Implement Monte Carlo Tree Search or Neural Network evaluation models.
* Improve GUI with move highlighting and hint system.

|  |  |
| --- | --- |
| **Git Hub Link of the project and report** | **https://github.com/AntoJenishia/Chess-AI/blob/main/chess\_ai.py** |

**REFERENCES**

* Russell, S., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach (4th Ed.)*. Pearson Education.
* python-chess Documentation: https://python-chess.readthedocs.io/
* GeeksforGeeks. (2020). *Minimax Algorithm in Game Theory (Part 1 & 2)*.
* Sebastian Lague. (2018). *Game AI – Minimax and Alpha–Beta Pruning* [YouTube Series].
* Wikipedia Contributors. (2024). *Alpha–Beta Pruning.* Retrieved from https://en.wikipedia.org/wiki/Alpha–beta\_pruning