

**National University of Sciences and Technology (NUST)**

School of Electrical Engineering & Computer Science

***Course Name:*** *Data Structures and Algorithms (DSA)*

**End Semester Project Report**

*“Chess Game Puzzle Solver”*

**Submitted By:**

|  |  |
| --- | --- |
| **Team Member Names** | **CMS IDs** |
| Muhammad Hamdan Ishfaq | 455881 |
| Muhammad Muntazar | 470861 |
| Abdul Mateen | 457052 |

**Github Link:** <https://github.com/Overproness/chess-puzzle-solver-with-minimax-and-alpha-beta/>

**Table of Contents:**

**Introduction...................................................................................................................................3 Literature Review..........................................................................................................................3 Methadology..................................................................................................................................3 Components of the system.........................................................................................................3** *Chessboard Setup and User Interaction***..........................................................................................3** *Mini Max algorithm with alpha beta pruning***..................................................................................4** *Heuristic for board Evaluation***........................................................................................................6** *User Interface***...................................................................................................................................7 Result and Analysis......................................................................................................................8 Conclusion.....................................................................................................................................8 References.....................................................................................................................................8**

## **Introduction**

Chess puzzles are a popular tool for honing strategic thinking and improving chess-playing skills. The purpose of this project is to develop a chess puzzle solver that leverages artificial intelligence techniques, specifically the Minimax algorithm with Alpha-Beta pruning. The solver allows users to set up chess puzzles, and the AI determines the optimal moves to solve them. The application includes a graphical user interface (GUI) for intuitive interaction, where users can move or remove pieces on the board and request the AI to solve the puzzle for either side.

This report discusses the design, implementation, and performance of the Chess Puzzle Solver, justifying the methodologies and heuristics employed.

## Literature Review

In the realm of artificial intelligence applied to board games, numerous advancements have been made to enhance the efficiency and decision-making capabilities of algorithms. The Minimax algorithm, a foundational strategy in game theory, is widely used for its ability to simulate possible outcomes by evaluating all potential moves and countermoves. Alpha-Beta pruning, a refinement of the Minimax algorithm, reduces computational complexity by eliminating branches that cannot influence the final decision, thus enabling deeper searches. Despite these optimizations, the computational demands of exhaustive evaluations in chess puzzles remain a challenge. This project draws inspiration from existing chess engines and AI implementations, aiming to balance performance and computational efficiency in solving chess puzzles.

## Methodology

The methodology for this project is centered around implementing the Minimax algorithm enhanced with Alpha-Beta pruning to optimize the search for the best moves. A custom GUI built using Pygame allows users to interact with the chessboard.

## Components of the System

### 1. Chessboard Setup and User Interaction

The chessboard is initialized in its standard setup, allowing users to move or remove pieces freely. This flexibility supports puzzle creation.

**Code Implementation:**

class ChessPuzzleSolver:

    def \_\_init\_\_(self):

        self.board = chess.Board()

        self.selected\_square = None

    def handle\_click(self, pos, button):

        clicked\_square = get\_square\_from\_mouse(pos)

        if clicked\_square is not None:

            if button == 1:  # Left-click: Move pieces

                if self.selected\_square is None:

                    piece = self.board.piece\_at(clicked\_square)

                    if piece:

                        self.selected\_square = clicked\_square

                else:

                    if clicked\_square != self.selected\_square:

                        self.board.set\_piece\_at(

                            clicked\_square, self.board.piece\_at(self.selected\_square)

                        )

                        self.board.remove\_piece\_at(self.selected\_square)

                    self.selected\_square = None

            elif button == 3:  # Right-click: Remove pieces

                self.board.remove\_piece\_at(clicked\_square)

**Justification:**

* The use of the chess.Board class from the python-chess library ensures accurate handling of chess rules while allowing flexibility for manual adjustments.
* Allowing unrestricted moves during setup makes it easy to create custom puzzles.

### 2. Minimax Algorithm with Alpha-Beta Pruning

The Minimax algorithm is used to simulate all possible moves and counter-moves, evaluating board states to find the optimal move. The make\_best\_move() function acts as the interface between the Minimax evaluation and the actual gameplay. It identifies the best move to make based on the Minimax results and executes it on the board.

**Code Implementation:**

def make\_best\_move(board, depth):

    best\_move = None

    best\_value = float("-inf") if board.turn == chess.WHITE else float("inf")

    for move in board.legal\_moves:

        board.push(move)

        move\_value = minimax(board, depth - 1, float("-inf"), float("inf"), not board.turn)

        board.pop()

        if (board.turn == chess.WHITE and move\_value > best\_value) or (board.turn == chess.BLACK and move\_value < best\_value):

            best\_value = move\_value

            best\_move = move

    if best\_move:

        board.push(best\_move)

    return best\_move

**Role in Integration:**

* The make\_best\_move() function simplifies the interaction between the AI logic and the chessboard. It directly applies the calculated optimal move, ensuring seamless gameplay.

**Justification for Approach:**

* This approach ensures that the Minimax results are immediately actionable. By incorporating the depth parameter, it balances computational efficiency and decision quality.
* Alternatives like storing moves in a separate data structure were avoided to reduce overhead and complexity in gameplay integration. Alpha-Beta pruning optimizes this process by eliminating branches that cannot influence the final decision.

**Code Implementation:**

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

    if depth == 0 or board.is\_game\_over():

        return evaluate\_board(board)

    if maximizing:

        max\_eval = float("-inf")

        for move in board.legal\_moves:

            board.push(move)

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

            board.pop()

            max\_eval = max(max\_eval, eval)

            alpha = max(alpha, eval)

            if beta <= alpha:

                break

        return max\_eval

    else:

        min\_eval = float("inf")

        for move in board.legal\_moves:

            board.push(move)

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

            board.pop()

            min\_eval = min(min\_eval, eval)

            beta = min(beta, eval)

            if beta <= alpha:

                break

        return min\_eval

**Justification:**

* Alpha-Beta pruning reduces computational complexity, allowing deeper searches in reasonable time by eliminating branches that cannot influence the final decision. It was most effective in this project when solving puzzles where certain moves led to immediate material gains or threats to high-value pieces. In these cases, pruning skipped the evaluation of irrelevant branches, saving computational resources and enabling a greater search depth. For example, positions with forced captures or checks benefited the most, as the algorithm quickly identified optimal responses without delving into moves that would clearly be inferior based on preliminary evaluations. This ensured efficiency without sacrificing decision quality.
* The algorithm ensures optimal move selection by evaluating all possible outcomes up to a specified depth.

### 3. Heuristics for Board Evaluation

The evaluation function assigns a value to each board state based on the material advantage. The heuristic values used for each piece are as follows: 1 for Pawn, 3 for Knight and Bishop, 5 for Rook, 9 for Queen, and 0 for King. These values were chosen because they align with standard chess valuation, emphasizing material balance as a fundamental aspect of position evaluation.

Alternative heuristics, such as positional factors like piece mobility or control of key squares, were not incorporated to maintain simplicity and computational efficiency. Incorporating such factors would require more complex logic and computational resources, which could detract from the primary focus on solving puzzles effectively and within a reasonable timeframe.

**Code Implementation:**

def evaluate\_board(board):

    return sum(piece\_value(piece) for piece in board.piece\_map().values())

def piece\_value(piece):

    values = {chess.PAWN: 1, chess.KNIGHT: 3, chess.BISHOP: 3, chess.ROOK: 5, chess.QUEEN: 9, chess.KING: 0}

    return values[piece.piece\_type] \* (1 if piece.color == chess.WHITE else -1)

**Justification:**

* The chosen values reflect standard chess heuristics, prioritizing material advantage.
* Alternative heuristics like positional bonuses or king safety were excluded after careful consideration. While positional bonuses could reward control of key squares and mobility, their calculation would introduce significant complexity and computational overhead. King safety heuristics were also omitted because puzzles often focus on immediate tactics rather than long-term strategic positioning. The exclusion of these heuristics ensured the solver remained efficient and focused on solving puzzles effectively within a reasonable timeframe, emphasizing material evaluation as the cornerstone of its heuristic design.

### 4. User Interface

The GUI is built using Pygame, displaying the chessboard, pieces, and control buttons for solving the puzzle. The GUI enhances user experience by providing clear visual feedback and interactive controls, such as highlighting the selected piece and the target square during moves. This design supports intuitive puzzle setup and solving, making it easier for users to focus on the puzzle logic without struggling with the interface. Additionally, the graphical representation of the board proved invaluable for debugging, as it allowed developers to visually verify the state of the board at every step of execution. User feedback highlighted the simplicity and responsiveness of the GUI as key features that improved engagement and usability.

**Code Implementation:**

def draw\_board(board, selected\_square=None):

    for rank in range(8):

        for file in range(8):

            rect = pygame.Rect(file \* CELL\_SIZE, rank \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE)

            color = WHITE if (rank + file) % 2 == 0 else BLACK

            pygame.draw.rect(screen, color, rect)

            # Highlight selected square

            if selected\_square == (file, rank):

                pygame.draw.rect(screen, (200, 200, 0), rect)

            # Draw the piece

            square = chess.square(file, 7 - rank)

            piece = board.piece\_at(square)

            image\_key = piece\_to\_image\_key(piece)

            if image\_key:

                screen.blit(piece\_images[image\_key], rect.topleft)

**Justification:**

* Pygame provides an intuitive interface for creating interactive elements like buttons and highlights.
* The clear display facilitates user interaction and debugging.

## Results and Analysis

The implementation of the Minimax algorithm with Alpha-Beta pruning yielded mixed results. While the pruning technique significantly reduced the number of evaluated nodes, the algorithm's performance was hindered by its depth limitation. Set to a depth of 5, the algorithm struggles with puzzles requiring longer planning horizons. For instance, puzzles solvable in 2-3 optimal moves often require 10-15 moves due to the limited foresight of the search depth. This inefficiency arises because the algorithm evaluates each position independently within the constraints of the specified depth, without effectively prioritizing imminent solutions.

Additionally, the simplicity of the evaluation function, which emphasizes material advantage, may overlook nuanced positional considerations. These factors, combined with the computational demands of exhaustive searches, highlight the trade-offs between depth, efficiency, and heuristic design. Optimizing these aspects through advanced heuristics or dynamic depth adjustments could significantly enhance performance.

## Conclusion

The Chess Puzzle Solver successfully integrates AI with a user-friendly GUI, allowing users to create and solve chess puzzles. The use of Minimax with Alpha-Beta pruning ensures efficient move calculation, and the chosen heuristics strike a balance between simplicity and effectiveness. Future improvements could include advanced evaluation metrics and support for complex puzzles.

## References

1. python-chess Library Documentation: <https://python-chess.readthedocs.io/>
2. Pygame Library Documentation: <https://www.pygame.org/docs/>
3. Understanding of Alpha beta pruning: https://youtu.be/dEs\_kbvu\_0s?si=xME0n5W-AwX5LLqG
4. Understanding of Min Max Algorithm: https://youtu.be/Ntu8nNBL28o?si=oxEEElUB5x3HJRed
5. Basic Chess Moves Understanding: https://youtu.be/8IlJ3v8I4Z8?si=UXeF4ER1fqSOF3NN
6. Assets:  
   *Pawn: https://images.app.goo.gl/e2qe6AbTJ8pqwwFY6  
   Rook: https://images.app.goo.gl/JhVHgPwABoCBN9ja9  
   Knight: https://images.app.goo.gl/CWRA64A69r8Vnd236  
   King: https://images.app.goo.gl/zQGNGaLSWMSkcKge6  
   Queen: https://images.app.goo.gl/nagbHMMw4VSupEgn7  
   Bishop: https://images.app.goo.gl/SKaV6AG4qYSgLGi68*