Artificial Intelligence (AI) Integration in Khmer Chess (Ouk Chaktrang) Game Development

Taing Heangleng[1], Luy Mithona[2], Sek Socheat[3]
[1]Master of Science in Information Technology (MsIT), Norton University, Phnom Penh, Cambodia
[2]Department of Computer Studies (DCS), Norton University, Phnom Penh, Cambodia
[3]Department of Computer Studies (DCS), Norton University, Phnom Penh, Cambodia

E-mail: taing.steven@gmail.com , mithonaluy@norton-u.com, socheatsek@norton-u.com

Corresponding Author: taing.steven@gmail.com

Received: 27-02-2024**; Accepted:** 22-06-2024**; Published:** 01-07-2024

Abstract: This paper focuses on developing an AI-based Khmer Chess (Ouk Chaktrang) game using the Python language, without relying on external libraries or frameworks. The project aims to create a Python package that can be used to develop various platforms, providing an accessible and user-friendly game that highlights the unique aspects of Khmer chess, while also serving as a foundation for future AI development in this area. The core of the AI implementation employs the Minimax algorithm combined with alpha-beta pruning to enable the AI to make strategic decisions by evaluating possible moves. Key strengths of this project include a functional game engine with features like move generation and check detection using an efficient reverse-check method. The development uses a 2D array for board representation and implements a piece scoring system. The study also demonstrates its versatility by being implemented in both console and GUI versions. However, key limitations and areas for improvement are identified. The analysis shows a significant increase in computational time as the AI's search depth increases, particularly at higher depths. The AI's move selection is currently random from the list of best moves, leading to unpredictable decisions. Moreover, the AI does not employ advanced techniques such as iterative deepening, selective search, or machine learning due to a lack of training data. In conclusion, this project serves as a foundational benchmark for future development in Khmer Chess AI and aims to promote Cambodian cultural heritage by showcasing the traditional game of Ouk Chaktrang. By addressing the limitations and implementing the proposed enhancements, the project has the potential to become an even more robust and engaging tool for preserving this culturally significant game.

Keywords: Artificial Intelligence (AI), Minimax Algorithm, Alpha–Beta Pruning, Khmer Chess, Ouk, Chaktrang (អុកខ្មែរ ឬ ល្បែងចត្រង្គ)

1. Introduction

1.1. Background of Proposed Study

Across cultures and centuries, games have served as playgrounds for wit, strategy, and competition. They test our minds, ignite our passions, and forge bonds between players. Among all games, chess is arguably the most well-known, played by countless people around the world in various forms and rulesets. In Cambodia, the ancient echoes of Angkor Wat resonate with the click of wooden pieces on a checkered board as the timeless game of Khmer chess, known as Ouk or Chaktrang, unfolds.

Chaktrang, a captivating and complex variant of traditional board games, has enthralled players in Cambodia for centuries. Its unique board setup, diverse pieces, and intricate rules offer a challenging test of tactical skill and strategic foresight. Due to its popularity, many programmers have transformed this ancient game into digital platforms, yet most of these versions still require two players. Currently, fewer people know how to play this game due to its complexity and the lack of available partners.

Copyright © 2024 IJAIMS International Journal of Artificial Intelligence and Mathematical Sciences, Volume3, Issue 01

This is why an Artificial Intelligence (AI) based player is essential—it provides individuals with the opportunity to learn and practice, while also promoting this traditional game. Additionally, this project will serve as a practical example, demonstrating the usage of AI in game development.

In the development of an intelligent Chaktrang game requires the integration of sophisticated artificial intelligence techniques that enhance both the game experience and user interaction. Central to this research are algorithms that facilitate effective decision-making strategies, allowing the AI to simulate an opponent capable of challenging even seasoned players. A prevalent approach is the Minimax algorithm, which operates on the principle of minimizing the possible loss in the worst scenario. By employing this algorithm in combination with alpha-beta pruning, the AI can effectively reduce the number of nodes evaluated in the game tree, accelerating decision-making processes without sacrificing strategic depth.

1.2. Objective

The aim of this study is to:

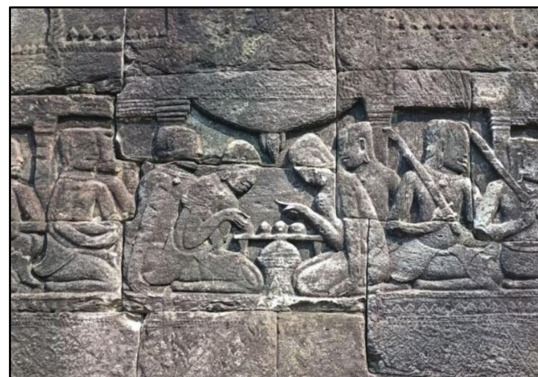
- **Develop an Engine and AI-Based Khmer Chess Game**: Create an intuitive and beginner-friendly AI chess game specifically for Ouk Chaktrang using Python language, without relying on any external frameworks or libraries. This approach will make the game accessible to novices and highlight the unique elements of Khmer chess.
- **Establish a Development Baseline**: Serve as a foundational benchmark for future developments in Khmer chess AI, providing a robust starting point for continued research and enhancement of AI-driven Khmer chess applications.
- **Inspire AI Integration**: Encourage and motivate developers to explore AI integration in their applications, demonstrating the potential of AI technology in traditional and culturally significant games.
- **Promote Cambodian Culture**: Celebrate and promote Cambodia's rich cultural heritage by spotlighting the traditional game of Ouk Chaktrang, fostering greater appreciation and engagement both locally and internationally.

2. Literature Review

2.1. Khmer Chess (Ouk Chaktrang)

Khmer Chess, also referred to Ouk Chaktrang, is a traditional Cambodian variant of chess with a rich history spanning centuries. Originating during the Angkor period around 800 AD, this game is evidenced by sculptures found on ancient Khmer temples [1]. It shares similarities with Western chess but features distinct rules and unique pieces that set it apart.

Statue at Angkor Wat Temple Constructed during the reign of King Suryavarman II (1112 - 1152)



Statue at Bayon Temple Constructed during the reign of King Jayavarman VII (1181 - 1201)

Figure 1: Statue of Chaktrang playing on Cambodian temples

Similar to international or Western chess, Khmer Chess requires two players who compete on an 8x8 board, totaling 64 squares. Each player commands 16 pieces, divided into black and white sets [2]. These pieces have their designated starting positions as Figure 2.

Copyright © 2024 IJAIMS International Journal of Artificial Intelligence and Mathematical Sciences, Volume3, Issue 01

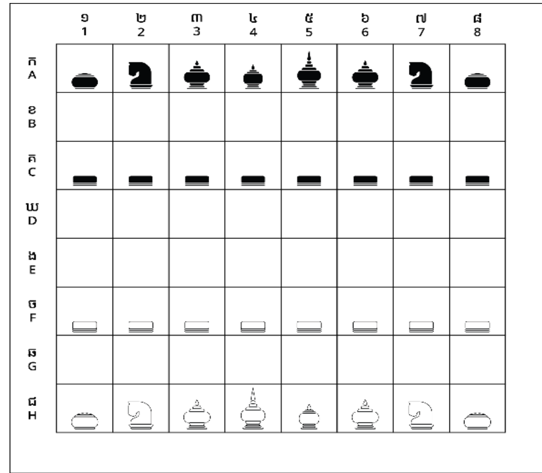


Figure 2: Starting position of Chaktrang

In addition, to win the game, players must capture the opponent's King by taking turns moving their pieces in unique ways, following specific rules as Figure 3.

Assigned Name: **Pawn**
Khmer Name: **Trey (ត្រី)**
Rule: Move and capture like a pawn in chess but cannot move two steps on the first move; therefore, there is no en passant. A pawn that reaches 6th rank is promoted to Neang (Trey Bork)

Assigned Name: **Bishop**
Khmer Name: **Koul (គោល)**
Rule: Move one step in any diagonal direction or one step forward

Assigned Name: **Knight**
Khmer Name: **Ses (សេះ)**
Rule: Moves like a knight in chess, two steps horizontally or vertically and then one step perpendicular to that direction

Assigned Name: **Rook**
Khmer Name: **Tuuk (ទូក)**
Rule: Move like a rook in chess, any numbers of steps vertically or horizontally

Normal Move | **First Move (Special Rule)**

Assigned Name: **Queen**
Khmer Name: **Neang (នាង ឬ ឆ្មើង)**
Rule: Move one step in any diagonal direction and can jump at 2nd square straight ahead at her first move.

Normal Move | **First Move (Special Rule)**

Assigned Name: **King**
Khmer Name: **Khon (ស្តេច ខុន ឬ អង្គ)**
Rule: Move one step in any directions. During its first move, the king can jump like a knight to the second row. This move cannot capture and cannot be performed while in check. The game ends when the king is checkmated, stalemated, or capture

Note: ● only move one step, ✕ only capture, ⊗ move and capture in one step, ◀▶▲▼ move and capture in multiple step

Figure 3: Chaktrang game's rules and pieces movement

To prevent the games from going on forever, there are rules that will declare the game a draw if it does not end in a certain number of moves, similar to Chess's 50-move rule. The maximum number of moves allowed will be called the limit. During the endgame, one of the players will most likely have some advantage and be the one trying to win, chasing the other's king into checkmate, while the other player tries to escape with a draw. For the purpose of this rule document the player with advantage will be called the chasing player, while the one with disadvantage will be called the escaping player. The chasing player has to win before the count reaches the limit. Otherwise, the game is declared a draw. The numbers of limit sets based on following conditions. When a player has three or fewer pieces, they may start counting from 1, with a limit of 64, it is calling Board's Honor Counting. The chasing player can declare a draw anytime. If the counting player checkmates without stopping the count, the game is a draw. Another count calls Piece's Honor Counting, when no unpromoted pawns are left and one player has only the king, counting starts from the number of pieces plus one. The limit varies based on the chasing player's material advantage (e.g., two rooks = 8 moves, one rook = 16 moves, are two bishops = 22, two knights = 32, one bishop = 44, one knight = 64, and queen and promoted pawns = 64). The count doesn't restart even if pieces are captured [3].

Minimax Algorithm

The Minimax algorithm, formulated by John von Neumann in 1928, is a foundational concept in game-playing AI, designed to maximize one's own gain while minimizing the opponent. It is used in two-player zero-sum games with perfect information to select optimal moves by a depth-first enumeration of the game tree [4]. In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score as possible while the minimizer tries to do the opposite and get the lowest score as possible. Every board state has a value associated with it. In a given state if the maximizer has upper hand, then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state, then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

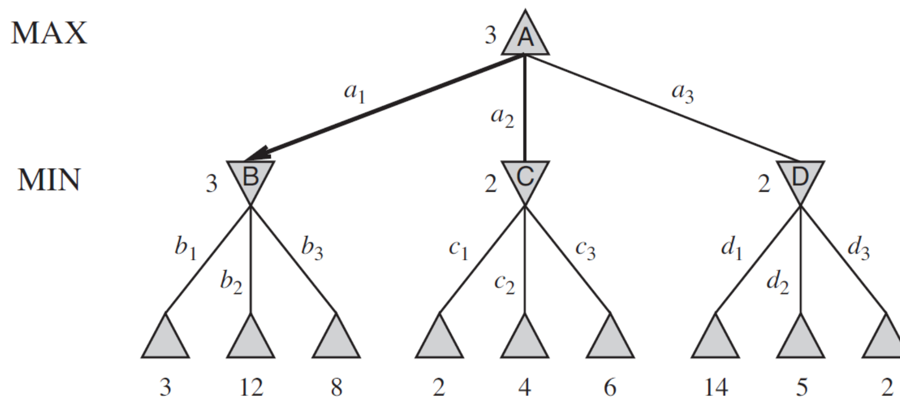


Figure 4: Minimax of a hypothetical search space. Leaf nodes show heuristic values

Figure 4 is a simulation of MiniMax search in two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a1, because it leads to the state with the highest minimax value, and MIN's best reply is b1, because it leads to the state with the lowest minimax value [5].

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value

(* Initial call *)
minimax(origin, depth, TRUE)
```

Figure 5: The pseudo-code for the depth-limited minimax algorithm [6]

Copyright © 2024 IJAIMS International Journal of Artificial Intelligence and Mathematical Sciences, Volume3, Issue 01

2.2. Alpha–Beta Pruning

The fundamental disadvantage of the minimax search is that the number of game states it has to examine is exponential in the depth of the tree. To improve search efficiency, a technique called alpha-beta pruning can be used. Alpha-beta pruning can reduce the number of nodes that are evaluated. It returns the same move as minimax, but prunes away branches that cannot influence the final decision. Alpha-beta pruning gets its name from two parameters that describe bounds on the backed-up values that appear along the path: α represents the value of the best choice for MAX found so far, while β represents the value of the best choice for MIN found so far [5]. Here's how alpha-beta pruning works:

- Alpha-beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively.
- The algorithm maintains an alpha value (α) which is the value of the best choice found so far for MAX and a beta value (β) which is the value of the best choice found so far for MIN.
- Pruning occurs when the algorithm determines that a node's value cannot influence the final decision because it is worse than the current α or β value.

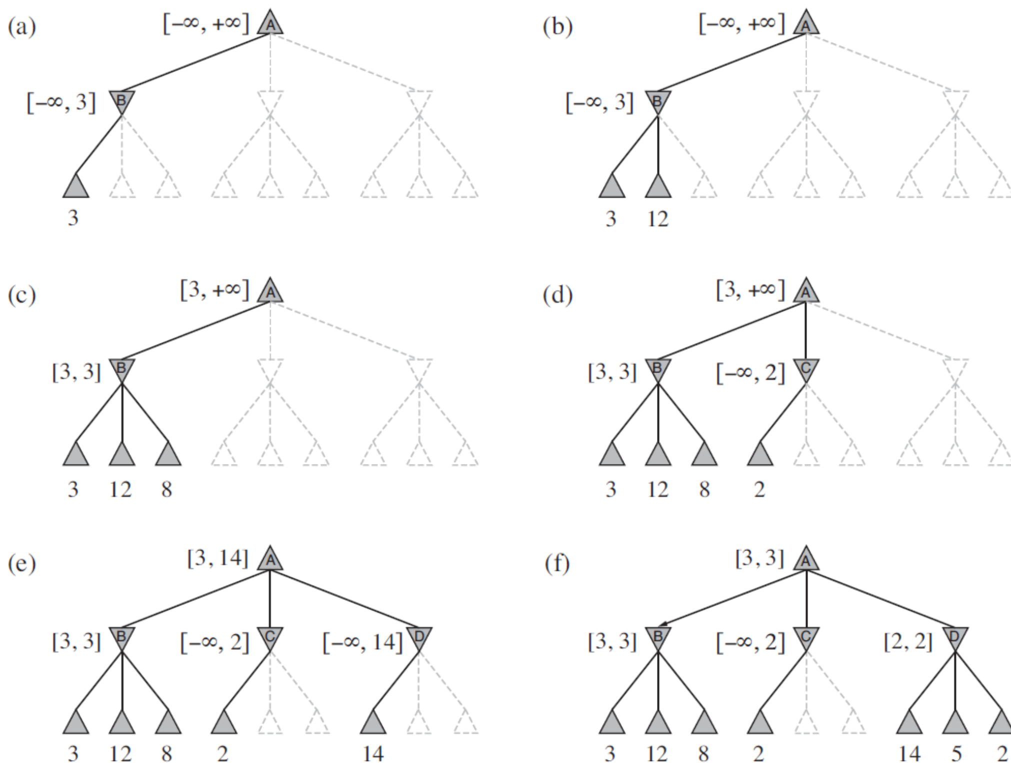


Figure 6: A left-to-right alpha-beta prune of the hypothetical minimax search space

Figure 6 stages in the calculation of the optimal decision for the game tree in Figure 4. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of at most 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is at least 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth at most 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3 [5].

Copyright © 2024 IJAIMS International Journal of Artificial Intelligence and Mathematical Sciences, Volume3, Issue 01

```
function alphabeta(node, depth, alpha, beta, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node

    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, alpha, beta, FALSE))
            alpha := max(alpha, value)  // Update alpha
            if beta <= alpha then
                break  // Prune branch
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, alpha, beta, TRUE))
            beta := min(beta, value)  // Update beta
            if beta <= alpha then
                break  // Prune branch
        return value

(* Initial call *)
alphabeta(origin, depth, −∞, +∞, TRUE)
```

Figure 7: The pseudo-code for depth limited minimax with alpha–beta pruning [6]

3. Methodology and Development

3.1. Software Development Method

The development of this game utilizes the Waterfall model. This structured approach significantly enhances the development of the chess game by establishing clear stages, ensuring clarity of requirements, and promoting thorough testing phases [7]. The project begins with creating the Chaktrang chess engine package, which is written in Python. There are two main modules to implement:

- **Person vs Person (PvP) Game Engine**: This module includes the game state, board representation, move generation, and a fully functional game for two players.
- **AI Chess Game**: This module focuses on game state evaluation, quiescence search, and the AI opponent.

The chosen algorithm for AI discission-making is the Minimax algorithm, a classic method that provides better insight for beginners. The goal is to develop a useful and enjoyable game, not an unbeatable chess master. Moreover, machine learning techniques like reinforcement learning require a large amount of data to train the model, which is currently unavailable.

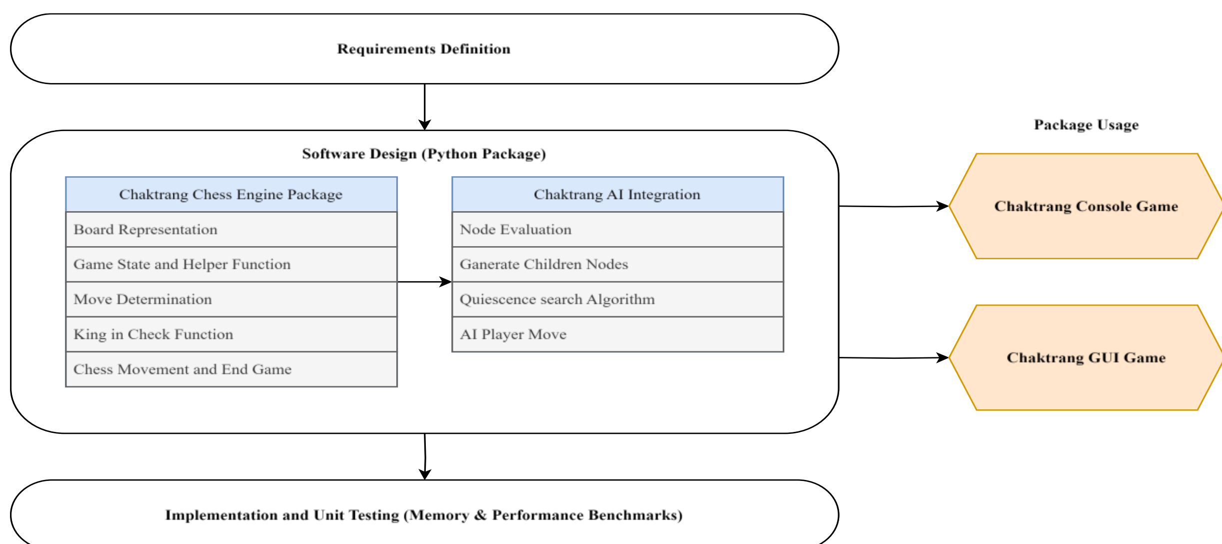


Figure 8: Software Development Process of Khmer Chess (Ouk Chaktrang) Game

Copyright © 2024 IJAIMS International Journal of Artificial Intelligence and Mathematical Sciences, Volume3, Issue 01

3.2. Person vs Person (PvP) Game Engine

To generate legal moves and track game changes in a chess program, a board representation is essential. There are various techniques for implementing the board, such as object arrays, 2D arrays, and bitboards. In this project, we use a 2D array for its simplicity and ease of calculating chess movements. Each piece and space are characterized by a single string, making it piece-centric. A dash ("-") represents a blank space, while to distinguish between the two sides, lowercase and uppercase letters are used: “P, p” for Pawn, “B, b” for Bishop, “N, n” for Knight, “R, r” for Rook, “Q, q” for Queen, and “K, k” for King. In addition, there are few more special pieces: "T, t" for Promoted Pawn, as in Cambodian chess, where pawns are flipped upside down when they reach enemy territory and move like a queen; and to easily manage the first state of the Queen and King, two more characters are used: "�q, ꝗ" for the Queen's first move and "Ḱ, ḱ" for the King's first move.

Furthermore, the core function of this game lies in its game state. The __init__ method is responsible for setting up the initial state, including the board configuration, player turn, game status (such as game over or checkmate), king in check status, and the move log. Additionally, there are three helper functions:

- **switchPlayer()**: This function is responsible for switching the turn between players.
- **checkPlayer(piece)**: This function helps determine which player a given piece belongs to by accepting a single string as a parameter.
- **kingLocation(board, player_turn)**: This function determines the location of the king on the board for the current player. It is particularly useful for implementing the inCheck function and other functions related to the king.

Please refer to Figure 9 for the code snippet.

```
def __init__(self):
    self.board = [
        ["R", "N", "B", "Ꝗ", "Ḱ", "B", "N", "R"],
        ["-", "-", "-", "-", "-", "-", "-", "-"],
        ["P", "P", "P", "P", "P", "P", "P", "P"],
        ["-", "-", "-", "-", "-", "-", "-", "-"],
        ["-", "-", "-", "-", "-", "-", "-", "-"],
        ["p", "p", "p", "p", "p", "p", "p", "p"],
        ["-", "-", "-", "-", "-", "-", "-", "-"],
        ["r", "n", "b", "ḱ", "ꝗ", "b", "n", "r"]
    ]
    self.player_turn = "Min" # "Min" | "Max"
    self.status = None  # gameover | checkmate | draw
    self.in_check = None  # K | k
    self.pin_pieces = []
    self.move_log = []

def switchPlayer(self):
    self.player_turn = "Max" if self.player_turn == "Min" else "Min"

def checkPlayer(self, piece):
    return "Min" if piece.islower() else "Max"

def kingLocation(self, board, player_turn):
    king_first_move = "ḱ" if player_turn == "Min" else "Ḱ"
    king = "k" if player_turn == "Min" else "K"
    king_row, king_col = -1, -1
    for row in range(8):
        for col in range(8):
            piece = board[row][col]
            if piece == king or piece == king_first_move:
                king_row, king_col = row, col
                break
    return king_row, king_col
```

Figure 9: Game State Initialization and Helper Functions

The critical part of this project is determining valid movements for each piece, which involves understanding the game rules and finding an effective way to generate valid moves. The moveDetermination(row, col, board, player_turn) method is key to this process, as it identifies possible moves for a piece on the board at a given location (row, col). It accepted parameters of row and col: The coordinates of the piece on the board, board: The current state of the game board, and player_turn: The player whose turn it is ("Min" or "Max"). the logic of this function is to retrieves the piece located then check for ally’s piece. Additionally, the direction variable is used to determine the move direction based on player turn, this is important for piece that move in only one direction like pawn and bishop. After that, matching the piece to letter that identify a piece type and generate all the move from the rule by increment row and column with preset coordinates, then filter out invalid moves (i.e., moves outside the board boundaries or moves blocked by pieces of the same player). Finally, the method returns the list of valid moves for the given piece, or None if no valid moves are found.

Copyright © 2024 IJAIMS International Journal of Artificial Intelligence and Mathematical Sciences, Volume3, Issue 01

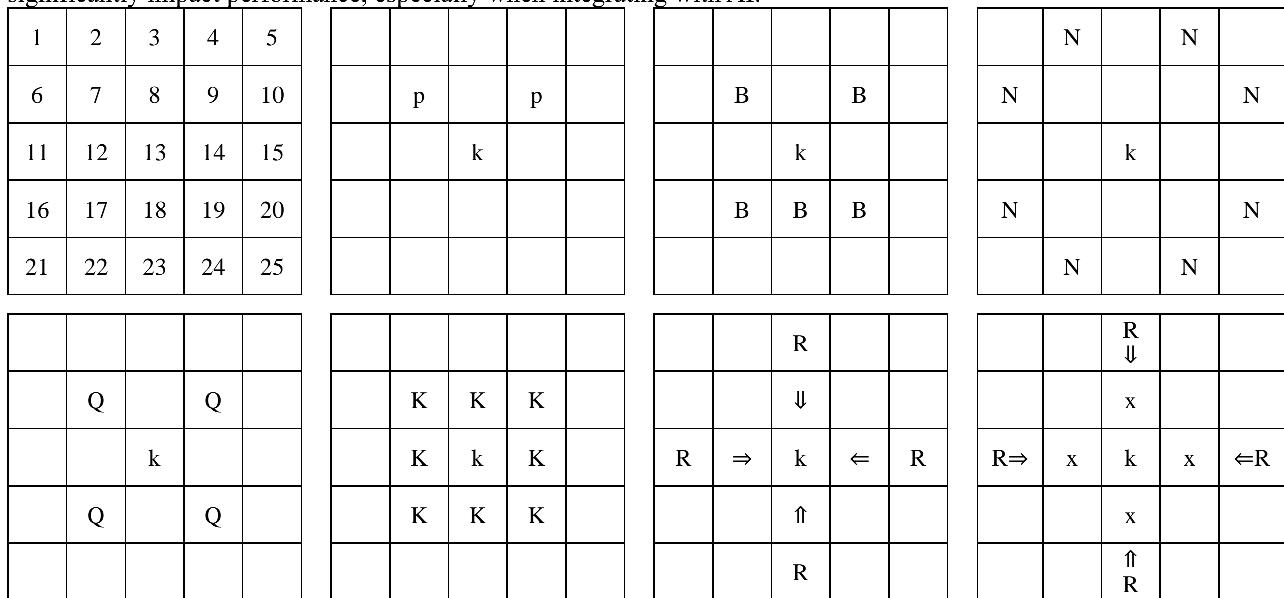
```
def moveDetermination(self, row, col, board, player_turn):
        piece = board[row][col]
        if piece == "-" or self.checkPlayer(piece) != player_turn:
            return None
        direction = -1 if piece.islower() else 1

        match piece:
            case "P" | "p":
                possible_moves = [(row + direction, col), (row + direction, col - 1), (row + direction, col + 1)]
                moves = list(filter(lambda move: (
                            (0 <= move[0] < 8 and 0 <= move[1] < 8)
                            and (
                                (move == possible_moves[0] and board[move[0]][move[1]] == "-")
                                or (move == possible_moves[1]  and board[move[0]][move[1]] != "-"
                                    and self.checkPlayer(board[move[0]][move[1]]) != player_turn)
                                or (move == possible_moves[2] and board[move[0]][move[1]] != "-"
                                    and self.checkPlayer(board[move[0]][move[1]]) != player_turn)
                            )
                        ),
                        possible_moves,
                    )
                )
                return moves
            case "B" | "b":
                possible_moves = [(row + direction, col),(row + 1, col - 1),(row + 1, col + 1),(row - 1, col - 1),(row - 1, col + 1)]
                moves = list(filter(lambda move: (
                            (0 <= move[0] < 8 and 0 <= move[1] < 8)
                            and (board[move[0]][move[1]] == "-" or self.checkPlayer(board[move[0]][move[1]]) != player_turn)
                        ),
                        possible_moves,
                    ))
                return moves
            case "N" | "n":
                possible_moves = [(row + 2, col + 1), (row + 2, col - 1), (row - 2, col + 1), (row - 2, col - 1),
                    (row + 1, col + 2), (row + 1, col - 2), (row - 1, col + 2), (row - 1, col - 2)]
                moves = list(filter(lambda move: (
                            (0 <= move[0] < 8 and 0 <= move[1] < 8)
                            and (board[move[0]][move[1]] == "-" or self.checkPlayer(board[move[0]][move[1]]) != player_turn)
                        ),
                        possible_moves,
                    )
                )
                return moves
            case "R" | "r":
                directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
                moves = []
                for direction in directions:
                    for i in range(1, 8):
                        move = (row + direction[0] * i, col + direction[1] * i)
                        if 0 <= move[0] < 8 and 0 <= move[1] < 8:
                            if board[move[0]][move[1]] == "-":
                                moves.append(move)
                            elif (
                                self.checkPlayer(board[move[0]][move[1]]) != player_turn
                            ):
                                moves.append(move)
                                break
                            else:
                                break
                return moves
            case "Q" | "q" | "T" | "t":
                possible_moves = [(row + 1, col + 1), (row + 1, col - 1), (row - 1, col + 1), (row - 1, col - 1)]
                moves = list(filter(lambda move: (
                            (0 <= move[0] < 8 and 0 <= move[1] < 8)
                            and (board[move[0]][move[1]] == "-" or self.checkPlayer(board[move[0]][move[1]]) != player_turn)
                        ),
                        possible_moves,
                    )
                )
                return moves
            case "K" | "k":
                possible_moves = [(row + 1, col + 1), (row + 1, col - 1), (row - 1, col + 1), (row - 1, col - 1),
                    (row, col + 1), (row, col - 1), (row + 1, col), (row - 1, col)]
                moves = list(filter(lambda move: (
                            (0 <= move[0] < 8 and 0 <= move[1] < 8)
                            and (board[move[0]][move[1]] == "-" or self.checkPlayer(board[move[0]][move[1]]) != player_turn)
                        ),
                        possible_moves,
                    )
                )
                return moves
            case "Q" | "q":
                possible_moves = [(row + direction, col + 1), (row + direction, col - 1), (row + (direction * 2), col)]
```


Copyright © 2024 IJAIMS International Journal of Artificial Intelligence and Mathematical Sciences, Volume3, Issue 01

```
                moves = list(filter(lambda move: (
                            (0 <= move[0] < 8 and 0 <= move[1] < 8)
                            and (board[move[0]][move[1]] == "-" or self.checkPlayer(board[move[0]][move[1]]) != player_turn)
                        ),
                        possible_moves,
                    )
                )
                return moves
            case "Ḱ" | "ḱ":
                possible_moves = [(row + 1, col + 1), (row + 1, col - 1), (row + direction, col), (row + direction, col + 1),
                    (row + direction, col - 1), (row + direction, col + 2), (row + direction, col - 2), (row + (direction * 2), col + 1),
                    (row + (direction * 2), col - 1)]
                moves = list(filter(lambda move: (
                            (0 <= move[0] < 8 and 0 <= move[1] < 8)
                            and (board[move[0]][move[1]] == "-" or self.checkPlayer(board[move[0]][move[1]]) != player_turn)
                        ),
                        possible_moves,
                    )
                )
                return moves
            case _:
                return None
```

Figure 10: Move Determination Function

In a game of chess, the king is in check when it's under direct threat of capture by an opponent's piece. This means the opponent could capture the king on their next move if no action is taken to prevent it. To implement this functionality, one might initially consider checking all the opponent's pieces, their possible moves, and whether those moves threaten the king. Simultaneously, it would require checking all allied pieces to see if any valid moves could counter the threat, repeating the first step for each move. However, this approach is computationally intensive, leading to thousands of iterations, which can significantly impact performance, especially when integrating with AI.



Nº	Square Nº	Movement	Threat Pieces to Min King (k)	Threat Pieces to Max King (K)
1	2	-2, -1	N	n
2	4	-2, +1	N	n
3	6	-1, -2	N	n
4	7	-1, -1	P, B, Q, T, K	b, q, t, k
5	8	-1, 0	R, K	r, b, k
6	9	-1, +1	P, B, Q, T, K	b, q, t, k
7	10	-1, +2	N	n
8	12	0, -1	R, K	r, k
9	14	0, +1	R, K	r, k
10	16	+1, -2	N	n

Copyright © 2024 IJAIMS International Journal of Artificial Intelligence and Mathematical Sciences, Volume3, Issue 01

11	17	+1, -1	B, Q, T, K	p, b, q, t, k
12	18	+1, 0	R, B, K	r, k
13	19	+1, +1	B, Q, T, K	p, b, q, t, k
14	20	+1, +2	N	n
15	22	+2, -1	N	n
16	24	+2, +1	N	n

Figure 11: Enemy's Piece Type and Threat Positions

A more efficient solution involves a reverse-check from the king's location. Instead of evaluating every opponent's move, the algorithm should focus on the king's immediate surroundings and potential attacking vectors. By analyzing the positions directly around the king and tracing back to possible attacker positions, the function can more effectively determine if the king is in check, thereby enhancing performance. Figure 11 illustrates the potential threat routes for the king from its surrounding position. The figure highlights that, when considering the movement constraints and possible attack vectors of the opponent's pieces, only 16 immediate squares around the king need to be examined to determine if the king is in check. If rook movements are factored in, the complexity slightly increases, encompassing fewer than 50 moves. This method significantly reduces the computational load compared to evaluating all opponent moves, ensuring a more efficient and streamlined approach to determining the king's safety. The visualization in Figure 9 clearly emphasizes this focused checking strategy, showcasing how targeted analysis can yield optimal results with far less computational effort.

```
king_attacker_movements = {
            "Min": [{"position": (2, -1), "attackers": "N"}, {"position": (2, 1), "attackers": "N"},
                {"position": (1, -2), "attackers": "N"}, {"position": (1, -1), "attackers": "BQTK"},
                {"position": (1, 0), "attackers": "RBK"}, {"position": (1, 1), "attackers": "BQTK"},
                {"position": (1, 2), "attackers": "N"}, {"position": (0, -1), "attackers": "Rk"},
                {"position": (0, 1), "attackers": "RK"}, {"position": (-1, -2), "attackers": "N"},
                {"position": (-1, -1), "attackers": "PBQTK"}, {"position": (-1, 0), "attackers": "RK"},
                {"position": (-1, 1), "attackers": "PBQTK"}, {"position": (-1, 2), "attackers": "N"},
                {"position": (-2, -1), "attackers": "N"}, {"position": (-2, 1), "attackers": "N"}],
            "Max": [{"position": (2, -1), "attackers": "n"}, {"position": (2, 1), "attackers": "n"},
                {"position": (1, -2), "attackers": "n"}, {"position": (1, -1), "attackers": "pbqtk"},
                {"position": (1, 0), "attackers": "rk"}, {"position": (1, 1), "attackers": "pbqtk"},
                {"position": (1, 2), "attackers": "n"}, {"position": (0, -1), "attackers": "rk"},
                {"position": (0, 1), "attackers": "rk"}, {"position": (-1, -2), "attackers": "n"},
                {"position": (-1, -1), "attackers": "bqtk"}, {"position": (-1, 0), "attackers": "rbk"},
                {"position": (-1, 1), "attackers": "bqtk"}, {"position": (-1, 2), "attackers": "n"},
                {"position": (-2, -1), "attackers": "n"},{"position": (-2, 1), "attackers": "n"}]
        }
```

```
def kingInCheck(self, board, player_turn):
        king_row, king_col = self.kingLocation(board, player_turn)
        possible_attack = self.king_attacker_movements[player_turn]
        for move in possible_attack:
            pos = move["position"]
            attackers = move["attackers"]
            if (
                0 <= king_row + pos[0] < 8 and 0 <= king_col + pos[1] < 8
                and board[king_row + pos[0]][king_col + pos[1]] in attackers
            ):
                return True
        rook_directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        rook = "R" if player_turn == "Min" else "r"
        for direction in rook_directions:
            ally_piece = 0
            enemy_piece = 0
            for i in range(1, 7):
                m = (king_row + direction[0] * i, king_col + direction[1] * i)
                if 0 <= m[0] < 8 and 0 <= m[1] < 8:
                    if (board[m[0]][m[1]] != "-" and self.checkPlayer(board[m[0]][m[1]]) == player_turn):
                        ally_piece += 1
                    elif (
                        board[m[0]][m[1]] != "-" and self.checkPlayer(board[m[0]][m[1]]) != player_turn
                        and board[m[0]][m[1]] != rook
                    ):
                        enemy_piece += 1
                    elif board[m[0]][m[1]] == rook and enemy_piece == 0 and ally_piece + len(rook) <= 2:
                        return True
        return False
```

Figure 12: Checking King in Check Function

Copyright © 2024 IJAIMS International Journal of Artificial Intelligence and Mathematical Sciences, Volume3, Issue 01

A digital game of chess becomes engaging and functional when players can move the pieces and visually observe the changes on the board. To achieve this, implementing the chess movement function is crucial. The package is designed for versatility, allowing it to be integrated into various platforms such as console games, GUIs, or web applications.

In this context, there are two functions to facilitate moves: one using array indices (moveByIndex) and another using algebraic notation (moveByNotation). The moveByIndex function first verifies if the game status is "gameover." If so, it returns False, disallowing any further moves. Otherwise, it retrieves the piece being moved and the piece being captured, if any, from the board. It then checks if the move is valid based on the current board state and the player's turn. If the move is valid, it logs the move in move_log and updates the board, handling any piece promotions accordingly.Before updating the board state, specific conditions are checked: if a Pawn ("P") reaches the 6th row from its starting edge, it is promoted to "T." Similar promotions occur for Queens and Kings on their first move. If a king is captured, the game status is set to "gameover," and the function returns False. Otherwise, the origin position is cleared ("-"), and the destination is updated with the moved piece. Finally, if the move is valid and no king is captured, the function switches the player's turn and returns True. The moveByNotation function accepts algebraic notation as input, converts it to index positions, and calls the moveByIndex function to execute the move. To achieve this, dictionaries mapping ranks to rows and files to columns are created. The notation system follows the format {origin row notation}{origin column notation}{destination row notation}{destination column notation}, i.e., "f4e4."

```
ranks_to_rows = {"a": 0, "b": 1, "c": 2, "d": 3, "e": 4, "f": 5, "g": 6, "h": 7}
rows_to_ranks = {v: k for k, v in ranks_to_rows.items()}
files_to_cols = {"1": 0, "2": 1, "3": 2, "4": 3, "5": 4, "6": 5, "7": 6, "8": 7}
cols_to_files = {v: k for k, v in files_to_cols.items()}
```

```
    def moveByIndex(self, origin_row, origin_col, destination_row, destination_col):
        if self.status == "gameover":
            return False
        piece_moved = self.board[origin_row][origin_col]
        piece_captured = self.board[destination_row][destination_col]

        valid_move = self.moveDetermination(
            origin_row, origin_col, self.board, self.player_turn
        )
        if valid_move:
            self.move_log.append(
                f"{piece_moved}{self.rows_to_ranks[origin_row]}{self.cols_to_files[origin_col]}{piece_captured}
                 {self.rows_to_ranks[destination_row]}{self.cols_to_files[destination_col]}"
            )
            if (destination_row, destination_col) in valid_move:
                if piece_moved == "p" and destination_row == 2:
                    piece_moved = "t"
                elif piece_moved == "P" and destination_row == 5:
                    piece_moved = "T"
                elif piece_moved == "Ɋ":
                    piece_moved = "Q"
                elif piece_moved == "ɋ":
                    piece_moved = "q"
                elif piece_moved == "Ḱ":
                    piece_moved = "K"
                elif piece_moved == "ḱ":
                    piece_moved = "k"
                self.board[destination_row][destination_col] = piece_moved
                self.board[origin_row][origin_col] = "-"
                if (
                    piece_captured == "K"
                    or piece_captured == "k"
                    or piece_captured == "Ḱ"
                    or piece_captured == "ḱ"
                ):
                    self.status = "gameover"
                    return False
                self.switchPlayer()
                return True
        return False

    def moveByNotation(self, notation):
        origin_row = self.ranks_to_rows[notation[0]]
        origin_col = self.files_to_cols[notation[1]]
        destination_row = self.ranks_to_rows[notation[2]]
        destination_col = self.files_to_cols[notation[3]]
        return self.moveByIndex(origin_row, origin_col, destination_row, destination_col)
```

Figure 13: Make Move Functions

Copyright © 2024 IJAIMS International Journal of Artificial Intelligence and Mathematical Sciences, Volume3, Issue 01

3.3. Artificial Intelligence (AI) Integration

The integration of Artificial Intelligence (AI) in chess has revolutionized this ancient game, bringing an unprecedented level of strategy and analysis to both casual players and grandmasters alike. In this module, the AI class is designed for a game-playing AI with functionalities like move evaluation and selection. There are three main methods: _evaluation, _childrenNode, and _minimax.

- **_evaluation() method**: This method calculates and returns the evaluation score of the board by summing up the scores of all pieces according to their assigned values in piece_score. The score assigned to each piece is based on how many moves it can make and its importance. For optimized calculation, the scores of each piece type are kept as small as possible, with the king having the highest score to ensure its significance.
- **_childrenNode() method**: This method generates all possible moves (child nodes) for the given player on the current board. Each move results in a new board state, which is added to a list of nodes. This method is used in the _minimax method for generating new states in deeper nodes and for the _evaluation method to calculate the score in order to find the best choice.
- **_minimax() method**: This method recursively evaluates the best move for the given player up to a certain depth. It evaluates board states and returns the best score and moves for the "Max" or "Min" player. Alpha represents the best score for the maximizer (initially $-\infty$), and beta represents the best score for the minimizer (initially $+\infty$). The algorithm prunes branches where it can no longer improve the score for the current player.

With the help of these methods, the aiMove method is created. It uses the _minimax method to find the best move for the AI up to a given depth. The AI picks a move randomly from the list of best moves and executes it. This AI evaluates all possible moves up to a specific depth, considering both the current player's and the opponent's moves, to decide the most advantageous move. It's a common approach in game-playing AI, particularly in chess-like games.

```
import random

class AI:
    def __init__(self, game):
        self.game = game
        self.piece_score = {
            "P": 1, "N": 8, "B": 5, "R": 9, "T": 4, "Q": 4, "Q": 4, "Ḱ": 100, "K": 100,
            "p": -1, "n": -8, "b": -5, "r": -9, "t": -4, "q": -4, "q": -4, "ḱ": -100, "k": -100,
        }

    def _enemy(self, player):
        return "Max" if player == "Min" else "Min"

    def _evaluation(self, board):
        score = 0
        for r in range(8):
            for c in range(8):
                score += self.piece_score.get(board[r][c], 0)
        return score

    def _childrenNode(self, board, player):
        nodes = []
        for c in range(8):
            for r in range(8):
                if board[r][c] != "-" and self.game.checkPlayer(board[r][c]) == player:
                    for move in self.game.moveDetermination(r, c, board, player):
                        movements = (r, c, move[0], move[1])
                        new_board = [row[:] for row in board]
                        moved_piece = new_board[r][c]
                        new_board[move[0]][move[1]] = moved_piece
                        new_board[r][c] = "-"
                        nodes.append({"movement": movements, "node": new_board})
        return nodes

    def _minimax(self, board, depth, alpha, beta, player):
        if depth == 0:
            return {"score": self._evaluation(board), "moves": None}

        nodes = self._childrenNode(board, player)

        if player == "Max":
            best_score = -float("inf")
            best_move = []
            for node in nodes:
                score = self._minimax(
                    node["node"], depth - 1, alpha, beta, self._enemy(player)
                )["score"]
                if score > best_score:
                    best_score = score
```


Copyright © 2024 IJAIMS International Journal of Artificial Intelligence and Mathematical Sciences, Volume3, Issue 01

```
                best_move = [node["movement"]]  # Reset to a single move
            elif best_score == score:
                best_move.append(node["movement"])  # Add to the list of best moves
            alpha = max(alpha, score)
            if alpha >= beta:
                break  # Prune subtree
        return {"score": best_score, "moves": best_move}
    else:
        best_score = float("inf")
        best_move = []
        for node in nodes:
            score = self._minimax(
                node["node"], depth - 1, alpha, beta, self._enemy(player)
            )["score"]
            if score < best_score:
                best_score = score
                best_move = [node["movement"]]
            elif best_score == score:
                best_move.append(node["movement"])
            beta = min(beta, score)
            if alpha <= beta:
                break  # Prune subtree
        return {"score": best_score, "moves": best_move}

def aiMove(self, depth, player):
    alpha = -float("inf") if player == "Max" else float("inf")
    beta = float("inf") if player == "Max" else -float("inf")
    best_move = self._minimax(self.game.board, depth, alpha, beta, player)
    if best_move["moves"]:
        move = random.choice(best_move["moves"])
        return self.game.moveByIndex(move[0], move[1], move[2], move[3])
```

Figure 14: AI Class with MiniMax Algorithm and Alpha–Beta Pruning

4. Results And Discussion

4.1. Implementation

The development of Khmer chess AI Python package is successfully tested it by implementing two types of games: a console game and a GUI game.

For the console game, the implementation resulted in a functional game where the board is displayed using Unicode symbols. Players make their moves by inputting chess notation, and the game shows the move log after every turn. If the king is in check, the player is alerted using the print() function. This console game serves as an example for implementing a chatbot game with text-based or emoji interactions.

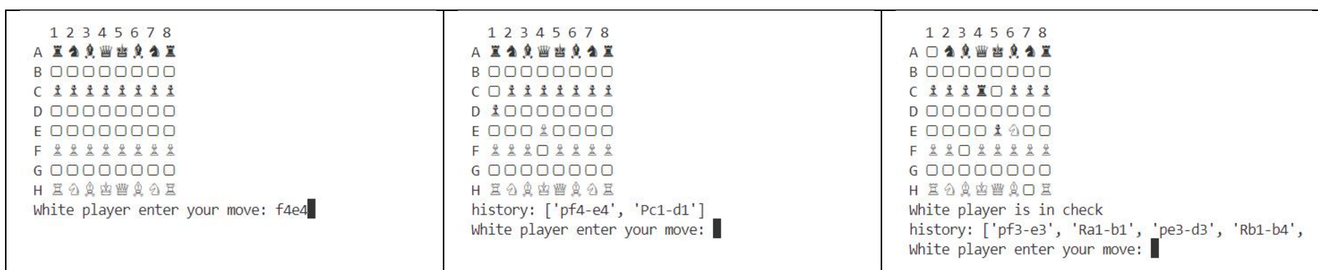


Figure 15: Khmer Chess (Chaktrang) Console Game

For the GUI game, the implementation was successful using the PyGame library. This version enhances the gameplay experience by providing a visual representation of the chessboard and pieces. It highlights possible moves and indicates when the king is in check. The graphical interface offers a more immersive and interactive experience for players, making it an excellent choice for those looking for a richer engagement with the game.

Copyright © 2024 IJAIMS International Journal of Artificial Intelligence and Mathematical Sciences, Volume3, Issue 01

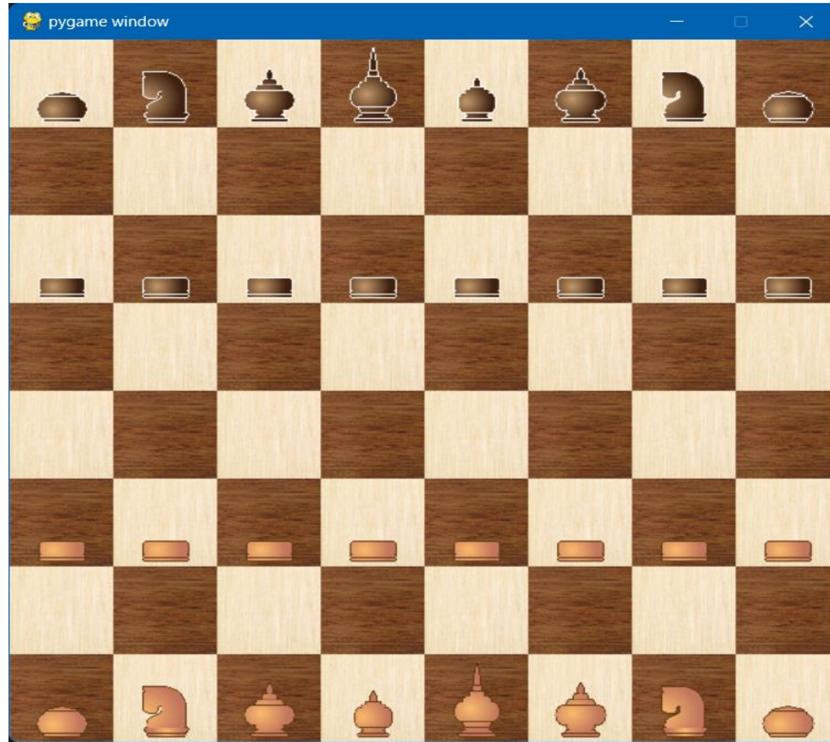
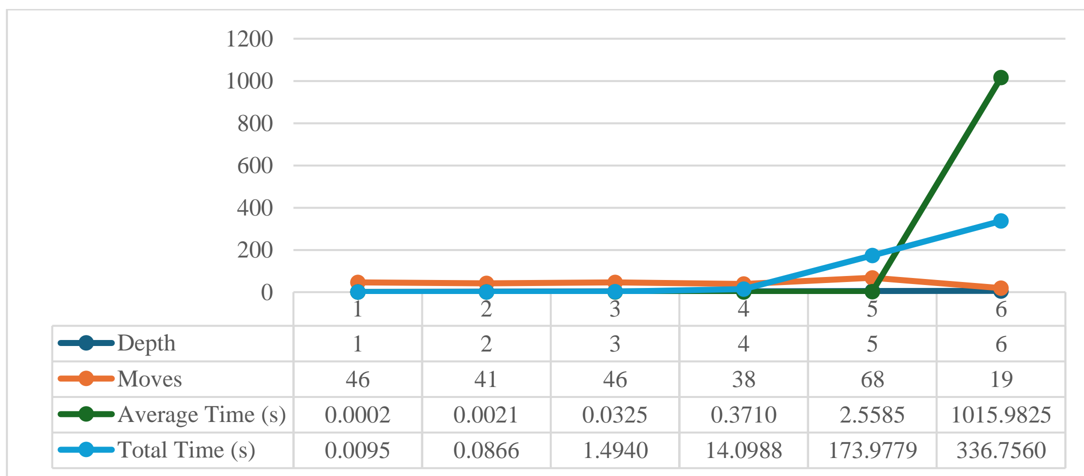


Figure 16: Khmer Chess (Chaktrang) GUI's Game

4.2. Analysis



Depth	1	2	3	4	5	6
Moves	46	41	46	38	68	19
Average Time (s)	0.0002	0.0021	0.0325	0.3710	2.5585	1015.9825
Total Time (s)	0.0095	0.0866	1.4940	14.0988	173.9779	336.7560

Figure 17: Performance Benchmark

The performance benchmark analysis of the chess AI shows a significant increase in computational time as the search depth increases. At Depth 1, the AI evaluates 46 moves with an average time of 0.0002 seconds per move, totaling 0.0095 seconds. As the depth increases to 2 and 3, the average time per move grows to 0.0021 and 0.0325 seconds, respectively, resulting in total times of 0.0866 and 1.4940 seconds. At Depth 4, the average time per move jumps to 0.3710 seconds, culminating in 14.0988 seconds for 38 moves. By Depth 5, each move takes an average of 2.5585 seconds. However, at Depth 6, the time per move skyrockets to 1015.9825 seconds, greatly affecting the user experience and resource utilization. Depth 3 emerges as the best choice for Chess AI, balancing strategic depth and computational feasibility. This serves as a starting point for Khmer chess (Chaktrang) AI, where techniques like iterative deepening and selective search can help maintain this balance, allowing the AI to achieve strategic depth without compromising computational efficiency.

Copyright © 2024 IJAIMS International Journal of Artificial Intelligence and Mathematical Sciences, Volume3, Issue 01

4.3. Strength and Weakness

The AI Khmer chess (Ouk Chaktrang) project, demonstrates both strengths and weaknesses in its current state. The project successfully implements core game functionalities and AI decision-making, but it also faces challenges related to computational intensity and strategic depth.

Strengths:

- The project features a functional Khmer chess AI that employs the Minimax algorithm with alpha-beta pruning for strategic move selection. This is a significant strength, enabling the AI to simulate an opponent that can challenge human players.
- The game is implemented in both a console version and a GUI version. This versatility highlights the potential of the game engine for deployment on different platforms.
- A 2D array is used for board representation, which makes calculations of piece movements straightforward.
- The project incorporates essential chess game functionalities, including move generation, player turn switching, and check detection. The check detection function uses an efficient reverse-check approach that focuses on the king's immediate surroundings, reducing computational load.
- A structured Waterfall model was used for software development which ensures clarity and facilitates thorough testing phases.
- The project serves as a foundational benchmark for future development in Khmer Chess AI. It also aims to promote the Cambodian cultural heritage by showcasing the traditional game of Ouk Chaktrang.

Weakness:

- Computational time increases significantly with search depth. At Depth 6, the average time per move is extremely high (1015.9825 seconds), making the game impractical.
- The AI’s move selection is currently random from the list of best moves, which means it can sometimes make unpredictable decisions.
- The AI does not implement techniques like iterative deepening or selective search, which could help improve performance and strategic depth.
- The project does not utilize machine learning techniques like reinforcement learning, due to a lack of training data.

5. Conclusion

The Khmer chess (Ouk Chaktrang) AI project provides a functional platform for playing the traditional Cambodian game, demonstrating both the successful implementation of core game mechanics and areas for future enhancement. While the AI effectively uses the Minimax algorithm with alpha-beta pruning, yet, certain limitations affect its performance and strategic depth. The project's strengths include a functional AI, versatile implementation, efficient board representation, and core game functionalities like move generation, player turn switching, and check detection. The check detection uses an efficient reverse-check approach, reducing computational load. The project also serves as a foundational benchmark for future developments in Khmer Chess AI, promoting Cambodian culture. However, the project also has weaknesses including significant computational time increase with search depth, random move selection from the best moves, lack of advanced AI techniques like iterative deepening or selective search, and not using machine learning techniques due to lack of data.

For the future improvement, several enhancements can be made.

- **Performance Optimization**: The project can be improved by using the Python dataclass decorator and Numpy arrays to optimize the code, which will improve efficiency. Additionally, techniques like iterative deepening and selective search can be implemented to enhance the strategic depth of the AI without drastically increasing computational time.
- **Resource Optimization**: Optimizing resource use is crucial, especially for higher search depths where computational time drastically increases. By enhancing the algorithms and using the most efficient data structures, the game can be made more resource-friendly.
- **Enhanced Decision-Making**: The AI's decision-making process can be improved by incorporating piece-square tables (PST). These tables assign values to pieces based on their position on the board, allowing the AI to make more informed moves that prioritize strategic positioning.
- **Pinning Pieces**: Implementing the ability for the AI to recognize when a piece is pinned when the king is in check can help to develop stronger strategic play. This would allow the AI to better calculate future move options.
- **Checkmate and Stalemate**: The game logic needs to be expanded to fully handle checkmate and stalemate scenarios to provide a complete game experience.

Copyright © 2024 IJAIMS International Journal of Artificial Intelligence and Mathematical Sciences, Volume3, Issue 01

These enhancements, alongside addressing the existing limitations, will lead to a more robust and strategically capable Khmer chess AI. The current project provides a strong foundation for future developments, and with these improvements, it has the potential to become an even more engaging and valuable tool for promoting and preserving the traditional game of Ouk Chaktrang.

References

[1] “មិនធម្មតា! គ្រាន់តែលេងអុកសោះ ទទួលបានអត្ថប្រយោជន៍ច្រើនដល់ម្ល៉ឹង,” អង្គភាពសារព័ត៌មាន SBM, https://sbm.news/articles/6219e6e5ed15c72f60a00731 (Accessed 04 November 2024).

[2] “Ok Chaktrong,” IntoCambodia.org. https://intocambodia.org/content/ok-chaktrong (Accessed 04 November 2024).

[3] “Ouk Chaktrang,” pychess.org, https://www.pychess.org/variants/cambodian (Accessed 16 January 2025)

[4] Luger, George F., “Artificial intelligence: structures and strategies for complex problem solving 6th Edition,” University of New Mexico: Pearson Education, Inc., 2019.

[5] Stuart J. Russell and Peter Norvig, “Artificial Intelligence: A Modern Approach 3rd Edition,” Pearson Education, Inc., 2010.

[6] Fouad Sabry, “Minimax: Fundamentals and Applications,” One Billion Knowledgeable, 2023

[7] Sommerville, Ian, “Software engineering 9th Edition,” Addison-Wesley: Pearson Education, Inc., 2011

Copyright © 2024 IJAIMS International Journal of Artificial Intelligence and Mathematical Sciences, Volume3, Issue 01