Intelligent Systems Techniques Coursework Report

Candidate No.285324

August 19, 2024

Contents

1 Introduction

This report provides a comprehensive analysis of the development and implementation of an interactive Chinese Chess game, focusing on key aspects such as gameplay mechanics, search algorithms, move validation, and challenges encountered during the development process. The project aimed to create an engaging and intelligent Chinese Chess game where a human player competes against a computer AI. The AI employs the Minimax algorithm with Alpha-Beta pruning to make strategic decisions, while the game interface ensures an immersive and user-friendly experience.

2 Gameplay

2.1 Interactive Chinese Chess Gameplay

- Criteria Met: Satisfied
- Explanation: The game offers an engaging interactive Chinese Chess experience where a human player competes against a computer AI. Moves are input using player's mouse click (e.g., click a cannon and click a position in the board, if its valid, cannon will move to that position), which the system accurately processes to determine new positions of the pieces. The main game loop manages the game state, user inputs, and AI responses, ensuring a smooth and interactive gameplay experience.

2.2 Board Representation

- Criteria Met: Satisfied
- Explanation: The board is visually represented on the screen using Pygame, displaying all pieces in their correct positions on a 10x9 grid, matching the traditional Chinese Chess layout. The graphical interface pleasing visual representation. This visual setup is initialized at the start and updates dynamically as the game progresses, accurately reflecting the game state at all times. The board also accurately reflects the unique elements of Chinese Chess, such as the "river" in the middle and the "palaces" where the kings and advisors are confined, horses can move by 1x2 or 2x1, but invalid if there is a piece on the point in between. A Chariot can move and capture either vertically or horizontally by any number of blocks, and it can capture the opponent's pieces in this way too, and other pieces can only move under rules. Additionally, the four corners of the window display information about the current turn, the AI player's difficulty level, the AI's actions, and the round state (as shown in figures 1 and 2).

2.3 Interface Update

- Criteria Met: Satisfied
- Explanation: The board's visual representation updates dynamically as moves are made by both the human player and the AI. The game interface promptly refreshes after each move, ensuring the human player sees the latest game state. This includes updates for both user and AI moves (as shown in graph 2).

2.4 Help Facility

- Criteria Met: Partially Satisfied
- Explanation: The game includes a help facility that highlights available moves for the current player. However, the feature does not currently employ AI for optimal move suggestions, which would be a more advanced implementation (as shown in figure 2). Additionally, when one side is in check, game will popup a window to warn the gamer (as shown in figure 4).

2.5 Adjustable AI levels

- Criteria Met: Satisfied

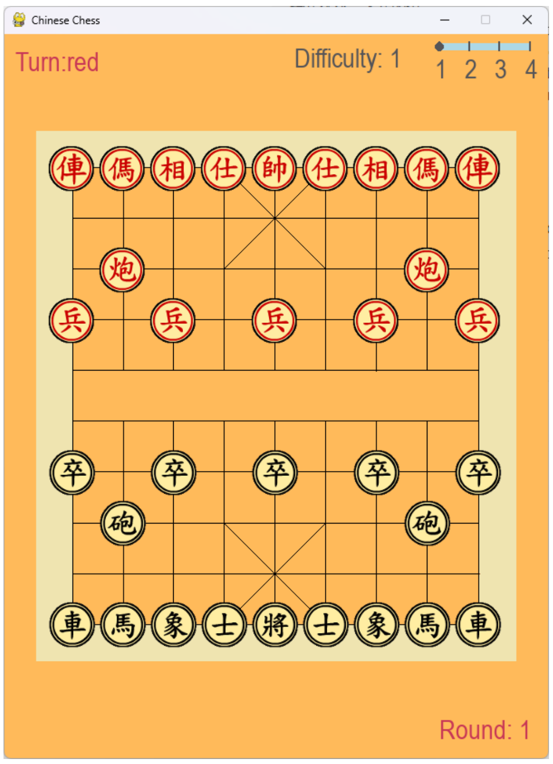


Figure 1: Initially Board Representation

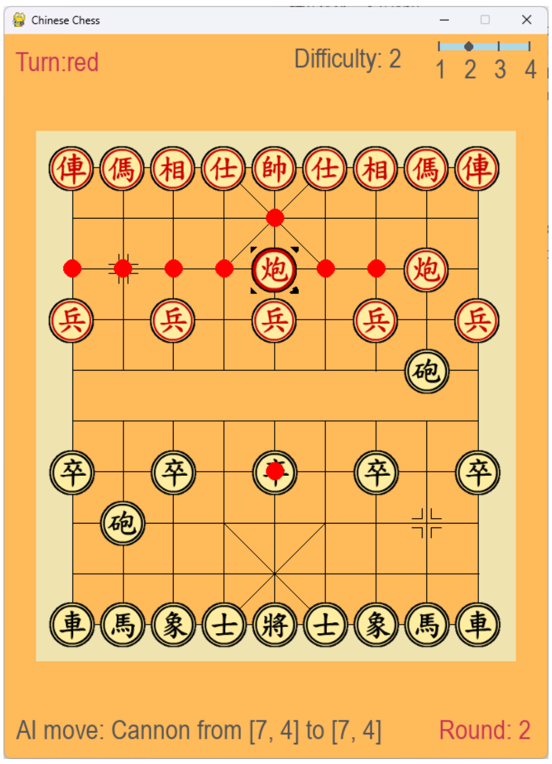


Figure 2: Interface Update

- Explanation: The game allows human player to adjust the AI difficulty level even in the middle of a game. The human player can change the AI's difficulty level at any time, providing flexibility that caters to both novice and experienced players by modifying the AI's cleverness. (as shown in figure 2).

3 Search Algorithm

3.1 State Representation

- Criteria Met: Satisfied
- Explanation: The game state is represented efficiently using a combination of a Numpy 2D array for the board and a dictionary for piece positions. This allows for quick access and manipulation of game pieces during move generation and evaluation. The `MinimaxEngine` class initializes with a copy of the current game state, ensuring that the AI can simulate moves without altering the actual game board. This setup ensures that the state representation is both efficient and flexible, allowing for accurate move simulations and reversals.

3.2 Successor Function

- Criteria Met: Satisfied
- Explanation: The successor function generates all possible moves for the AI by evaluating the legal moves for each piece. The `get_free_positions` method returns a dictionary mapping each piece to its list of legal moves. This method is utilized during the Minimax algorithm to explore potential future game states. By iterating over each piece and its possible moves, the successor function ensures that the AI considers all viable options, leading to a comprehensive evaluation of the game state.

3.3 Minimax Evaluation and Alpha-Beta Pruning

- Criteria Met: Satisfied
- Explanation: The AI employs the Minimax algorithm with Alpha-Beta pruning to optimize decision-making. The `minimax` method recursively evaluates game states to a specified depth,

alternating between maximizing and minimizing players. Alpha-Beta pruning is used to eliminate branches that do not need to be explored, significantly reducing the computational usage. The algorithm tracks the best move evaluations, updating alpha and beta values to prune unnecessary branches effectively. This approach helps the AI make informed decisions several steps ahead and choose the move that maximizes its score.

3.4 Use of Heuristics

- Criteria Met: Satisfied
- Explanation: Heuristics are used to evaluate the desirability of game states. The `static_evaluation` method calculates a score based on various factors such as piece value, board control, and the safety of the king. Additional heuristic elements include distance-based scoring for attacking pieces and capturing potential. The `count_team_score` method further refines the evaluation by considering the positions and potential moves of all pieces, ensuring a comprehensive assessment of the game state. This use of heuristics allows the AI to make more nuanced and strategic decisions. More detailed, this methods take 4 marking criteria, in all situation, AI is max player. Firstly, if current state of the board is human player win the game, than score is negative infinite, if AI player win the game, score is positive infinite. Secondly, if current state of the board is AI's king in check, minus 500 score, if human player's king in check, add 500 score, which is a huge score, because if put enemy's king in check is a good . Thirdly, get all pieces in the board and count their score by 3 approaches which are calculate the `attack_pieces`' (Horse, Chariot, Cannon, Pawn) attacking score by calculate the Manhattan distance to the enemy's king, calculate the additional score based on their valid moves which can capturing the piece, put enemy's king in check, self score, and the score of capturing the piece which make self king in check is 200. Add self side pieces' score and minus other side pieces' score.

4 Validation of Moves

4.1 No Invalid Moves by AI

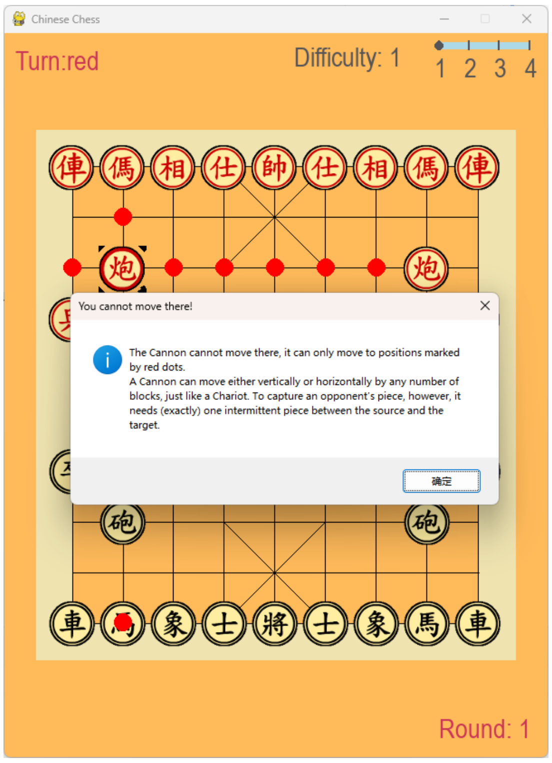
- Criteria Met: Satisfied
- Explanation: The AI is designed to generate only valid moves for each piece, ensuring adherence to the rules of Chinese Chess. Each piece class (e.g., King, Advisor, Elephant) includes an `is_valid_move` method that checks if a move is valid based on the current game state and piece-specific movement rules. The `MinimaxEngine` employs these methods to verify the validity of each potential move before making a decision, Furthermore, the `MinimaxEngine` can only analyze free positions from each piece's valid moves. As a result, the AI does not make any invalid moves, maintaining the integrity of the game.

4.2 User Move Validity Check

- Criteria Met: Satisfied
- Explanation: The game ensures that all user moves are checked for validity before they are executed. Each piece class includes methods to determine its valid moves, only valid moves will highlight to the player, and player can only move based on these highlight moves (as shown in figure 2).

4.3 Rejection of Invalid User Moves

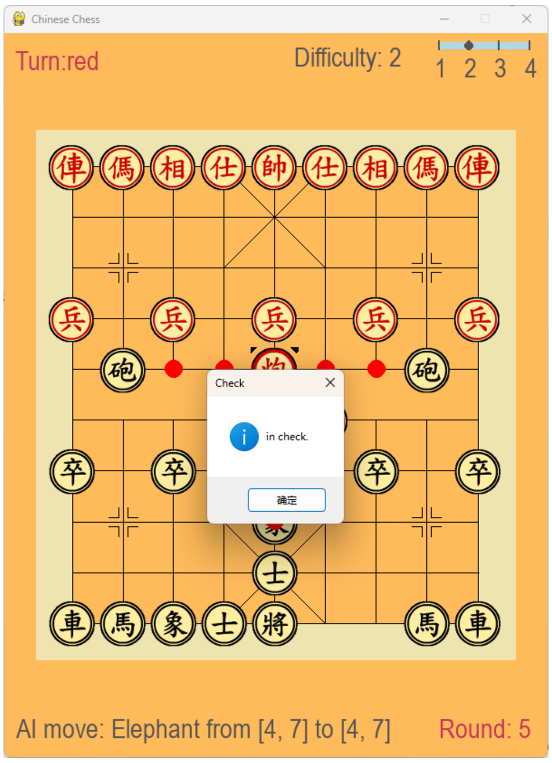
- Criteria Met: Satisfied
- Explanation: Invalid user moves are promptly rejected by the game, with explanations provided to the player. When a user attempts an invalid move, the game highlights the issue and provides feedback, helping the player understand why the move is not allowed. This feedback mechanism is implemented within the `is_valid_move` methods of each piece class, which return False if a move does not comply with the rules. The interface then informs the player of the invalid move, maintaining clarity and enhancing the learning experience (as shown in figure 3).



You cannot move there!

The Cannon cannot move there, it can only move to positions marked by red dots.
A Cannon can move either vertically or horizontally by any number of blocks, just like a Chariot. To capture an opponent's piece, however, it needs (exactly) one intermittent piece between the source and the target.

Figure 3: Initially Board Representation



Check

in check.

AI move: Elephant from [4, 7] to [4, 7]

Figure 4: In Check

5 Challenges

5.1 User Move Validation

In the development, validating player moves and providing hints for valid moves against the complex rules of Chinese Chess was challenging due to the variety of pieces and their unique movement patterns. For example, the cannon can move vertically and horizontally in normal situations, but for capturing an opponent's piece, it requires exactly one intermittent piece between the source and the target. To solve this problem, the `get_legit_moves(piece)` function was implemented to consider the entire board state and update it in real-time.

The game maintains a Numpy 2D array to store the board state and updates it dynamically with each move. This approach allows the `get_legit_moves` function to easily determine the correct moves for each piece. Additionally, each piece class has its own `is_valid_move()` function, which checks whether a move is valid based on the specific rules for that piece. This modular design ensures that all possible moves are accurately validated according to the rules of Chinese Chess.

5.2 Implementing the Minimax Algorithm with Alpha-Beta Pruning

The minimax algorithm with Alpha-Beta Pruning itself is well established, but the biggest challenge came from applying the algorithm to this project.

5.2.1 Minimax Algorithm Implement Challenge 1

The first challenge was implementing a static evaluation function. The static evaluation function needed to assess the current state of the game with a reasonable heuristic. Without a good heuristic, the AI would not be able to make informed decisions. The evaluation function had to consider various factors such as piece value, board control, and specific rules of Chinese Chess. My solution was to use heuristics to evaluate the desirability of game states. The `static_evaluation` method calculates a score based on various factors, such as piece value, board control, and the safety of the king. Additional heuristic elements include distance-based scoring for attacking pieces and capturing potential. The `count_team_score` method further refines the evaluation by considering the positions and potential moves of all pieces, ensuring a comprehensive assessment of the game state. This use of heuristics allows the AI to make more nuanced and strategic decisions. More detailed approaches are introduced before. By refining the heuristic through extensive testing and adjustments, the static evaluation function provided a robust basis for the AI's decision-making.

5.2.2 Minimax Algorithm Implement Challenge 2

The second major challenge was implementing the simulation of board moves and their reversals. This required perfecting the user move validation mentioned earlier because any incorrect move could lead to inaccurate game states and faulty AI decisions. If a move is not accurately simulated, it could result in illegal positions, invalid evaluations, and ultimately a broken game logic. In terms of reversing moves, it was crucial to return every piece to its original position perfectly without affecting the overall game state. Additionally, if the game was won, lost, or drawn during the simulation, the game state needed to be reverted accordingly.

5.3 Efficient State Representation

Representing the game state efficiently was critical for the performance of the AI, especially during move generation and evaluation. The game combines a Numpy 2D array for the board with a dictionary for piece positions to achieve an efficient state representation. This setup allows direct access and manipulation of the game state, facilitating quick lookups and updates. The Numpy array provides a fast, structured way to manage the board grid, while the dictionary efficiently maps piece positions. This dual representation ensures that the AI can swiftly generate and evaluate moves, enhancing overall performance.

5.4 Implementing a User-Friendly Help Game

Developing a user-friendly help game that could provide solid suggestions and information of current board state without overwhelming the player was challenging. Initially, a basic version of the game was no help, only experienced player who know well about this game can get good experience. After that, the game have a help information at four corner of the window, and a highlight piece with highlighted valid moves, if player make a in-valid move, the game will popup a hint about this piece's rule. Future plans include integrating the AI to suggest optimal moves, leveraging the Minimax evaluation to provide intelligent hints.