Overview：

Firstly, the project uses PyGame to complete the front-end modelling, including the board, piece hint text and console buttons, as shown below:

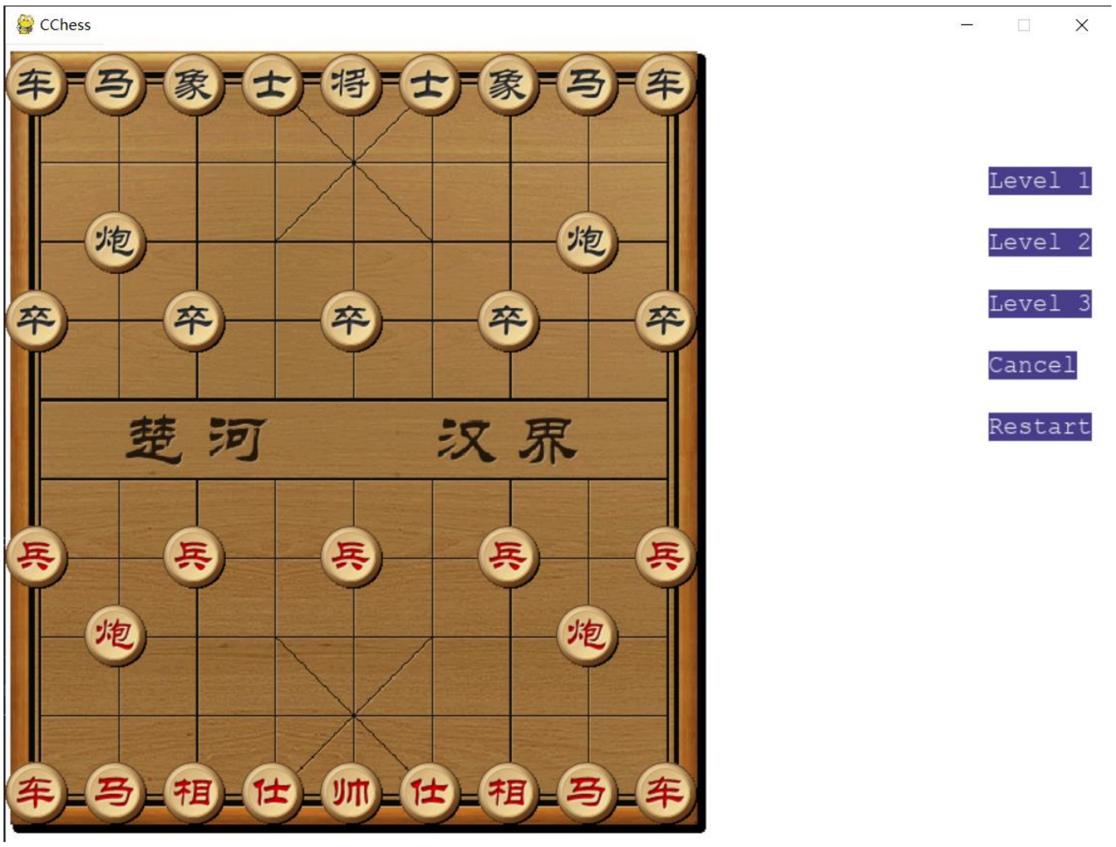


Figure 1 CChess Game Interface

In the decision-making process of the program, MinMax algorithm was used, and alpha-beta pruning was used as an optimization tool, the relevant functions are stored in

For the evaluation of the position, the chess position-value table from Eleeye was used. The Chinese chess move rules were also applied.

Gameplay：

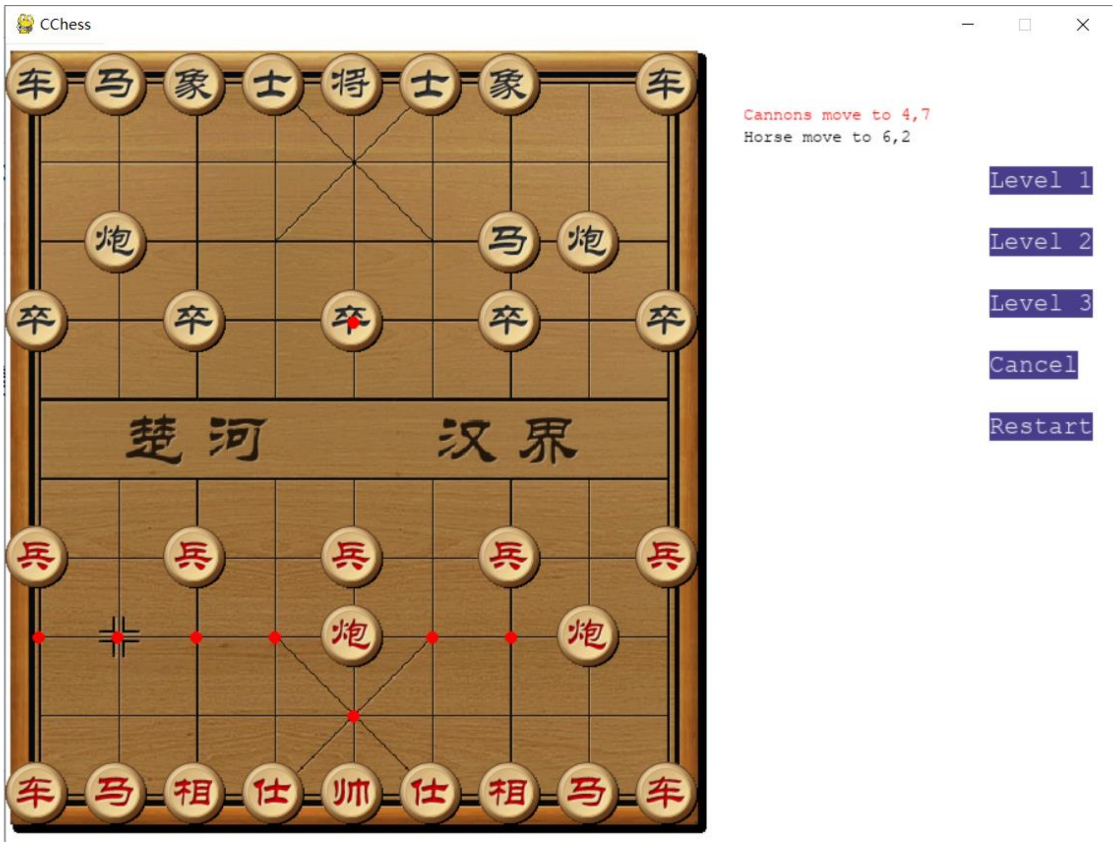
In this section, there're 5 marking criteria, with 3 fully satisfied criterion and 2 partly satisfied criterion .

Interactive Chinese Chess gameplay: Satisfied

PyGame is used as the front-end framework in this project. In the game flow, the project is controlled by listening to the player's mouse events, allowing the player to tap the pieces first and then tap the destination to complete the move.

Board representation shown on screen: Satisfied

The board and pieces are displayed on the left side of the window, and each move is aligned to the board grid.



```
Cannons move to 4,7
Horse move to 6,2
```

Figure 2 Pieces Move and Align

The interface properly updates the display after completed moves: Satisfied

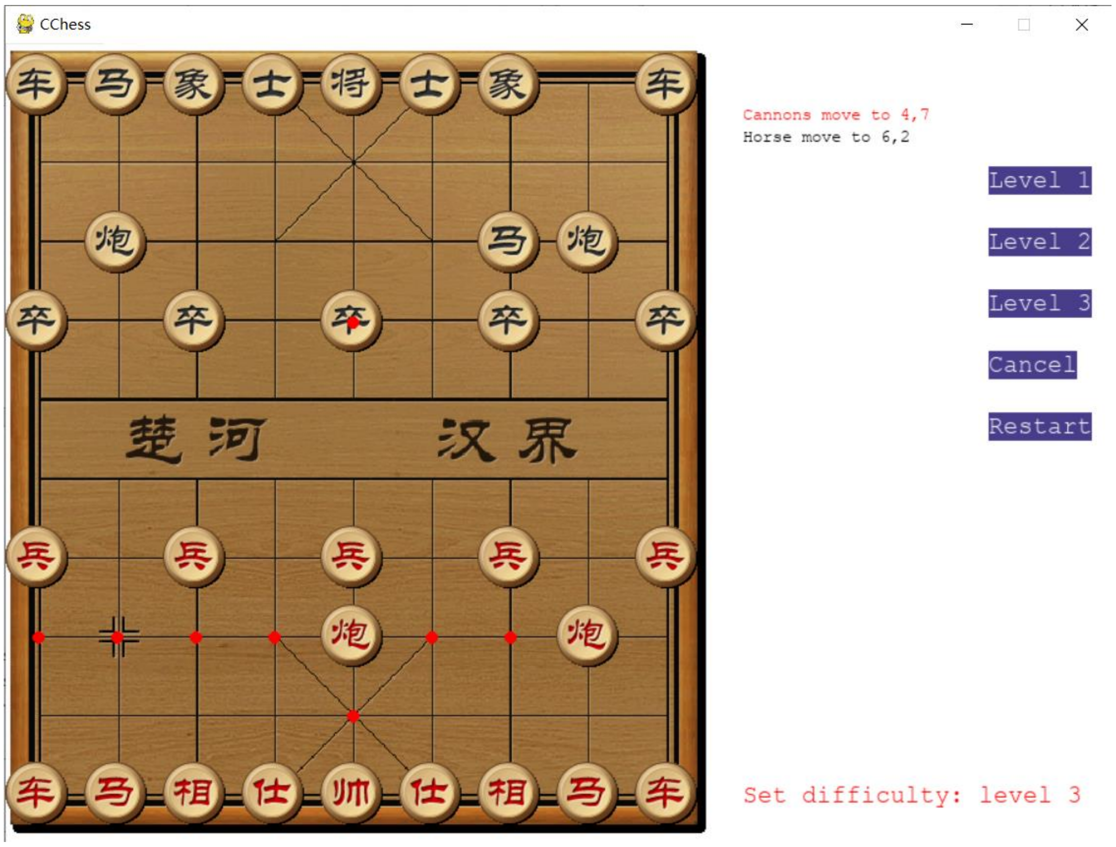
Use lists to record and update piece positions in real time, rendering them on the board in real time.

A help facility that provides hints about available moves: Part Satisfied

Red dots are used to indicate how far a piece can travel, but they do not provide a first-pick probability or win probability, and you need to click again on a piece after it has travelled to ensure that the content is updated smoothly.

Different levels of AI cleverness: Part Satisfied

The adjustment of difficulty is accomplished by changing the maximum depth of the search tree, but when the difficulty is level 3(search depth is 6), the search efficiency decreases substantially, which lengthens the ai reaction time.



```
Cannons move to 4,7
Horse move to 6,2
```

```
Set difficulty: level 3
```

Figure 3 Change Difficulty

Search algorithm:

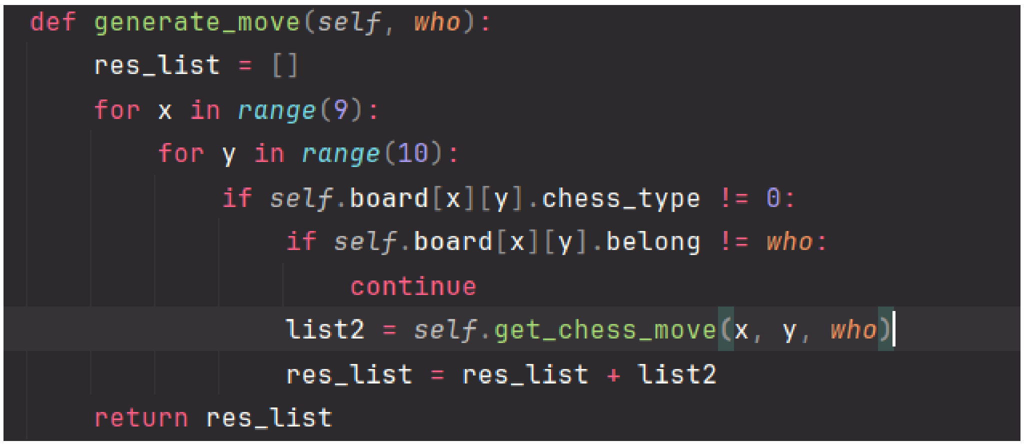
In this section, there're 4 marking criteria, with 4 fully satisfied criterion.

Appropriate state representation: Satisfied

Use lists to record the state of the field, including piece ownership, relationship to neighbouring pieces, piece survival, and piece position.

Successor function generates AI moves. Satisfied

Iterate over each feasible piece in the board to obtain its reachable space, and do not reevaluate this move if it was ever recorded in the history information, or perform the MinMax algorithm if it does not occur



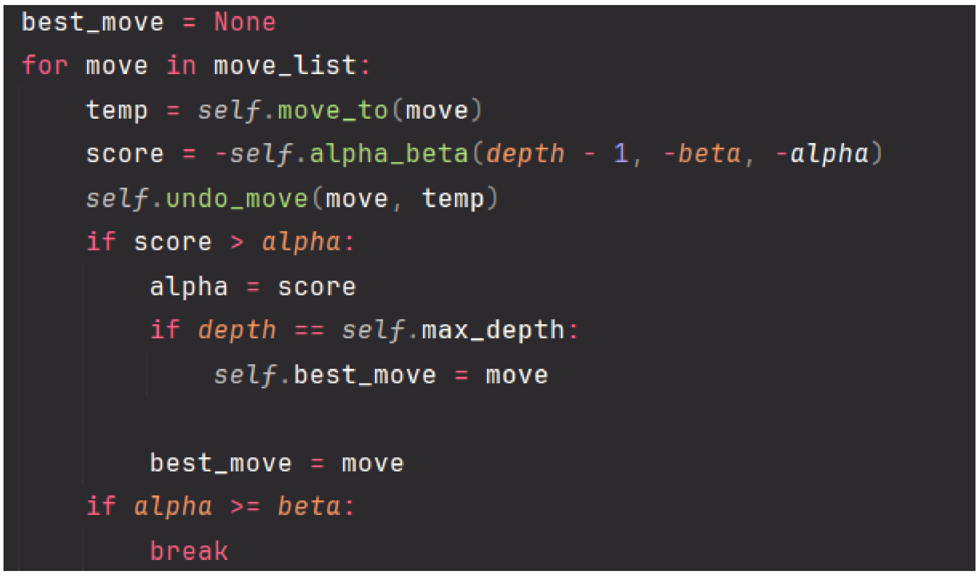
```python
def generate_move(self, who):
    res_list = []
    for x in range(9):
        for y in range(10):
            if self.board[x][y].chess_type != 0:
                if self.board[x][y].belong != who:
                    continue
                list2 = self.get_chess_move(x, y, who)
                res_list = res_list + list2
    return res_list
```

Figure 4 Move Generation

Minimax evaluation and Alpha-Beta pruning. Satisfied

Create a search tree, specifying the player as maximize player and ai as minimize player. Recursively search through the tree. When searching the computer step, taking the opposite number for the score and swapping the values of alpha and beta.

Perform alpha-beta pruning of the search tree.

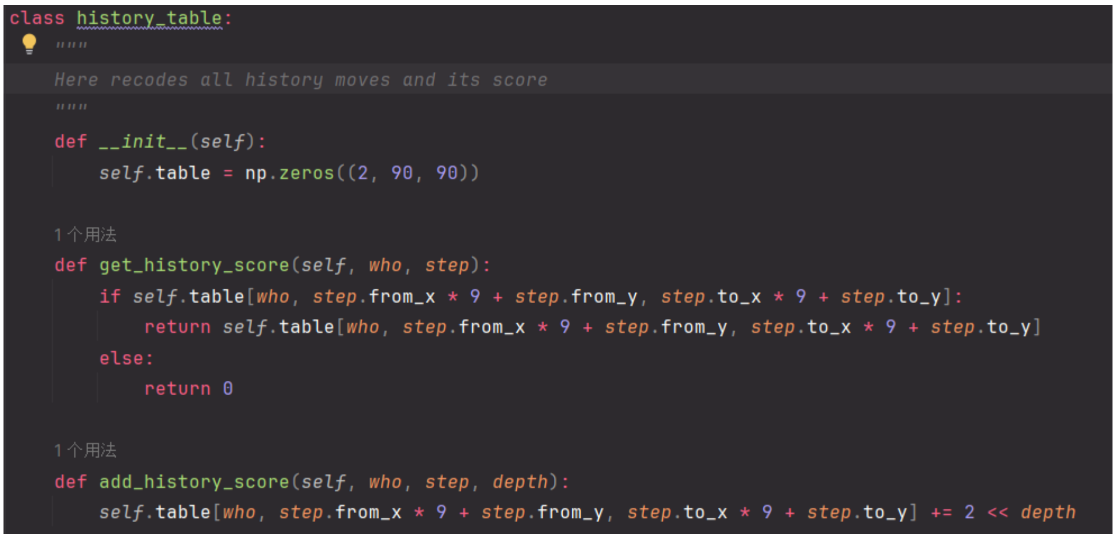


```python
best_move = None
for move in move_list:
    temp = self.move_to(move)
    score = -self.alpha_beta(depth - 1, -beta, -alpha)
    self.undo_move(move, temp)
    if score > alpha:
        alpha = score
        if depth == self.max_depth:
            self.best_move = move

        best_move = move
    if alpha >= beta:
        break
```

Figure 5 Alpha-Beta pruning

Appropriate use of heuristics. Satisfied

Use two arrays to store all the searched history scores for the player and the AI respectively, and update the history if there are any updates at the time of use.



```
class history_table:
    """
    Here recodes all history moves and its score
    """
    def __init__(self):
        self.table = np.zeros((2, 90, 90))

    1个用法
    def get_history_score(self, who, step):
        if self.table[who, step.from_x * 9 + step.from_y, step.to_x * 9 + step.to_y]:
            return self.table[who, step.from_x * 9 + step.from_y, step.to_x * 9 + step.to_y]
        else:
            return 0

    1个用法
    def add_history_score(self, who, step, depth):
        self.table[who, step.from_x * 9 + step.from_y, step.to_x * 9 + step.to_y] += 2 << depth
```

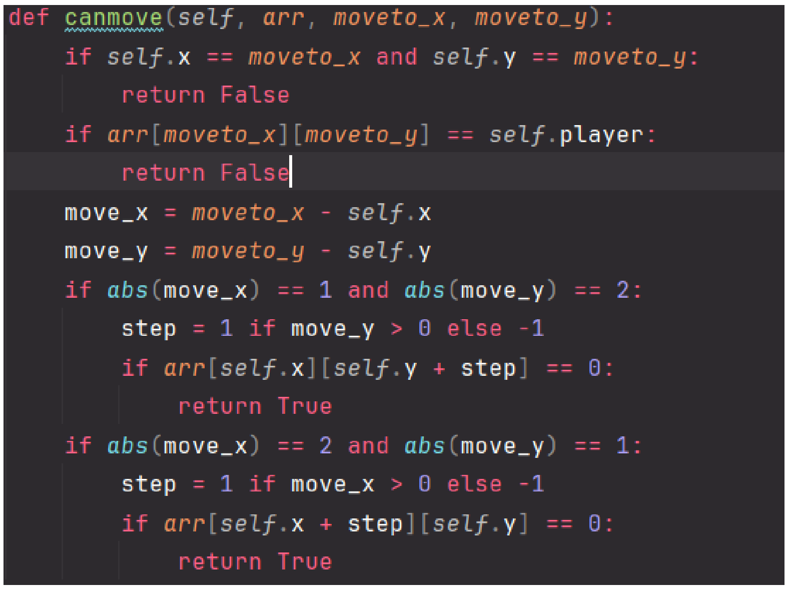
Figure 6 History Heuristics

Validation of move

In this section, there're 3 marking criteria, with 2 fully satisfied criterion and 1 partly satisfied.

No invalid moves by the AI: Satisfied

The player's moves are checked according to the type of piece, and each piece class inherits the func canmove() from the base class Pieces, which checks the range of the piece's moves.

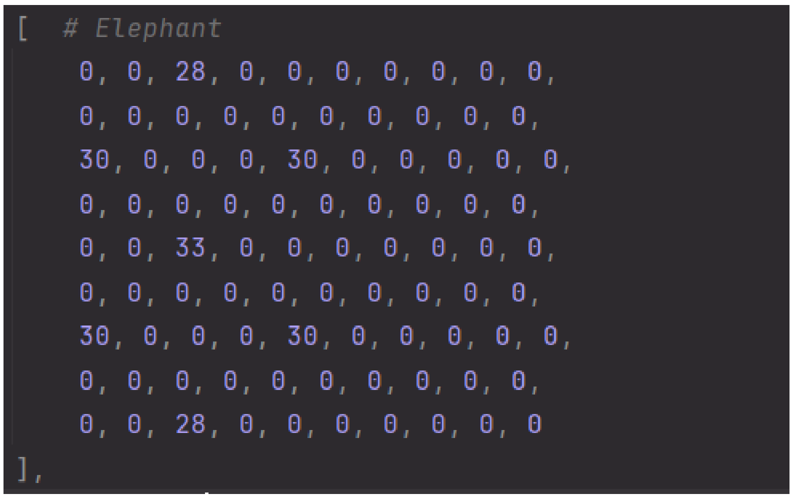


```
def canmove(self, arr, moveto_x, moveto_y):
    if self.x == moveto_x and self.y == moveto_y:
        return False
    if arr[moveto_x][moveto_y] == self.player:
        return False
    move_x = moveto_x - self.x
    move_y = moveto_y - self.y
    if abs(move_x) == 1 and abs(move_y) == 2:
        step = 1 if move_y > 0 else -1
        if arr[self.x][self.y + step] == 0:
            return True
    if abs(move_x) == 2 and abs(move_y) == 1:
        step = 1 if move_x > 0 else -1
        if arr[self.x + step][self.y] == 0:
            return True
```

Figure 7 Func canmove (take horse as an example)

User moves checked for validity: Satisfied

The above checks are also performed when AI walks, and in addition, unreachable positions are scored as 0.

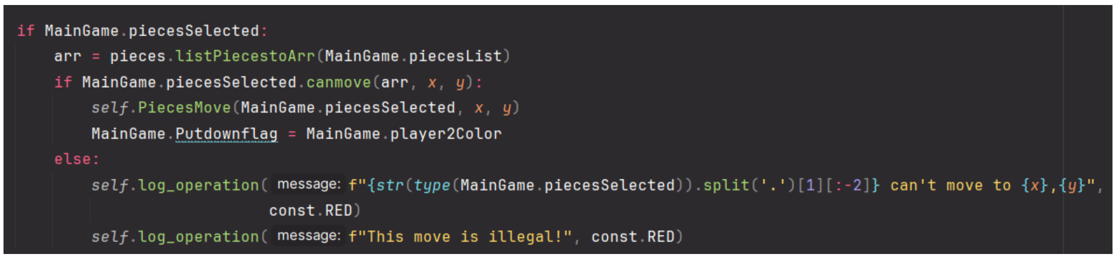


```
[   # Elephant
     0, 0, 28, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     30, 0, 0, 0, 30, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 33, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     30, 0, 0, 0, 30, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 28, 0, 0, 0, 0, 0, 0, 0
],
```

Figure 8 Elephant Postion-Value Table as an Example

Rejection of invalid user moves, with explanation given: Part Satisfied

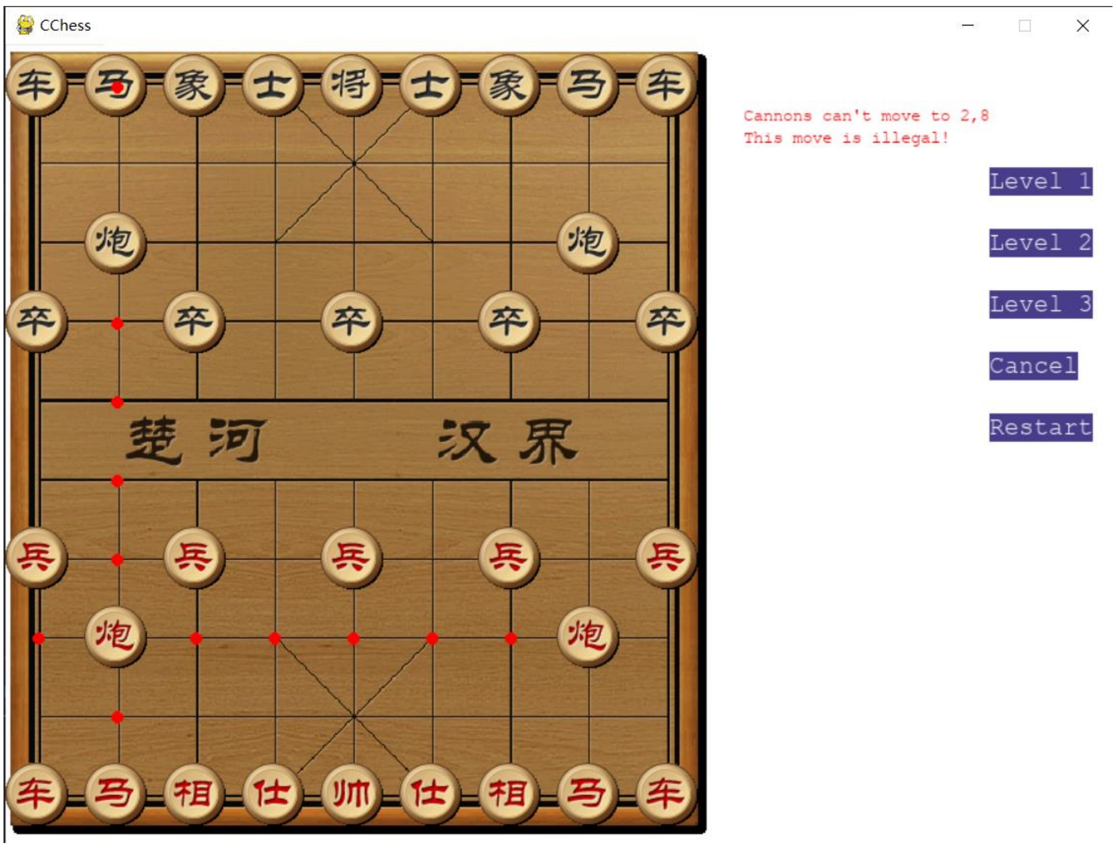
Dynamically detects the reach of the pieces during the move, with special constraints on the straight-line range of cannons and chariot, and diagonal range of horses and elephants.



```
if MainGame.piecesSelected:
    arr = pieces.listPiecestoArr(MainGame.piecesList)
    if MainGame.piecesSelected.canmove(arr, x, y):
        self.PiecesMove(MainGame.piecesSelected, x, y)
        MainGame.Putdownflag = MainGame.player2Color
    else:
        self.log_operation( message: f"{str(type(MainGame.piecesSelected)).split('.')[1][:-2]} can't move to {x},{y}",
                          const.RED)
        self.log_operation( message: f"This move is illegal!", const.RED)
```

Figure 9 Move Check

This function currently only indicates whether a move is correct or not, but cannot show which rule has been violated (blocked or unreachable).



```
Cannons can't move to 2,8
This move is illegal!
```

Figure 10 Detect Illegal Movement

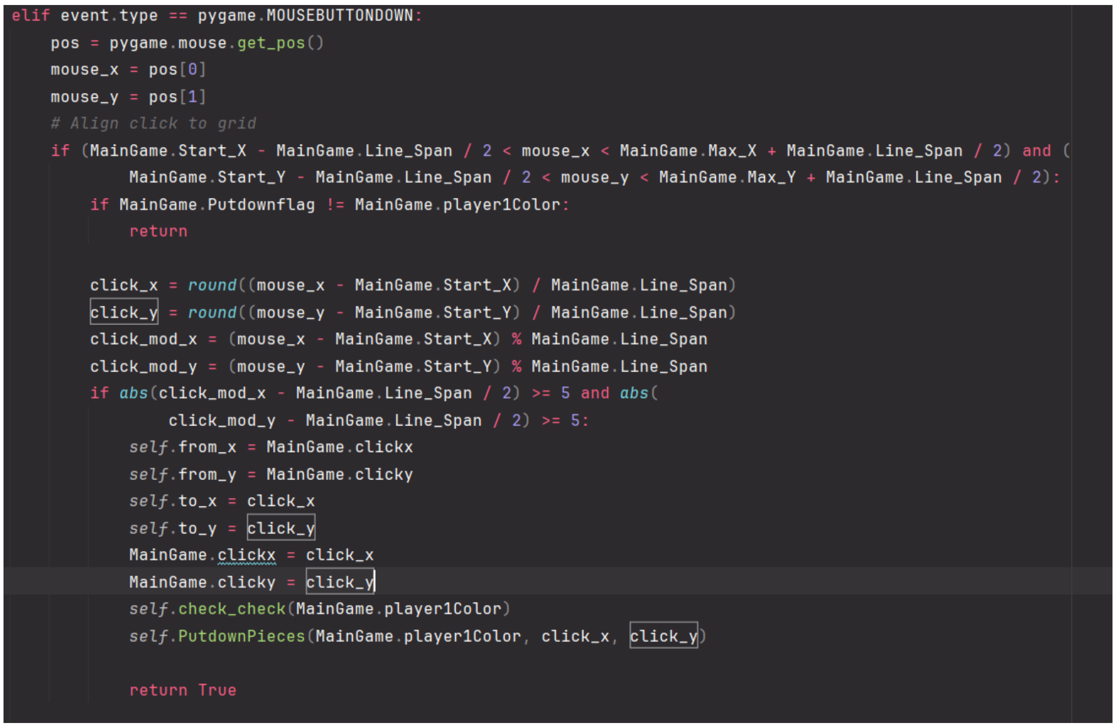
Problems Encountered:

The problems encountered in the project can be divided into two categories, one is the problems in the process control process and the other is the stability and existence problems in the structure of the algorithm

Correct Rendering of Pieces and Click Position Determination:

This is a process control problem, in the process of the project, first of all, there is the problem that the pieces can not be correctly aligned with the grid, the means of solving this problem include two parts, one is to adjust the background image correctly, so that it is scaled correctly in the direction of the length and width of the image, and the other is to measure the size of the board grid and adjust the size of the pieces to fit each other.

Another problem is the pointing of the pieces, after positioning the pieces to the grid in the previous step, use PyGame to listen to the coordinates of the mouse clicks, and take an integer grid of the clicked position, approximating it as a click on the piece.

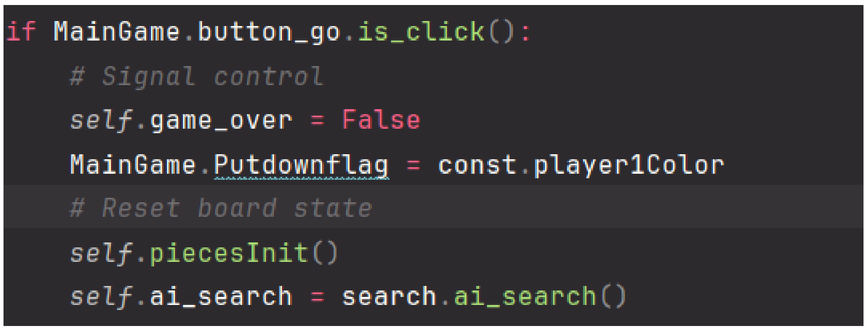


```python
elif event.type == pygame.MOUSEBUTTONDOWN:
    pos = pygame.mouse.get_pos()
    mouse_x = pos[0]
    mouse_y = pos[1]
    # Align click to grid
    if (MainGame.Start_X - MainGame.Line_Span / 2 < mouse_x < MainGame.Max_X + MainGame.Line_Span / 2) and (
            MainGame.Start_Y - MainGame.Line_Span / 2 < mouse_y < MainGame.Max_Y + MainGame.Line_Span / 2):
        if MainGame.Putdownflag != MainGame.player1Color:
            return

        click_x = round((mouse_x - MainGame.Start_X) / MainGame.Line_Span)
        click_y = round((mouse_y - MainGame.Start_Y) / MainGame.Line_Span)
        click_mod_x = (mouse_x - MainGame.Start_X) % MainGame.Line_Span
        click_mod_y = (mouse_y - MainGame.Start_Y) % MainGame.Line_Span
        if abs(click_mod_x - MainGame.Line_Span / 2) >= 5 and abs(
                click_mod_y - MainGame.Line_Span / 2) >= 5:
            self.from_x = MainGame.clickx
            self.from_y = MainGame.clicky
            self.to_x = click_x
            self.to_y = click_y
            MainGame.clickx = click_x
            MainGame.clicky = click_y
            self.check_check(MainGame.player1Color)
            self.PutdownPieces(MainGame.player1Color, click_x, click_y)

            return True
```

Figure 11 Processing Click

Repentance and Game Reset:

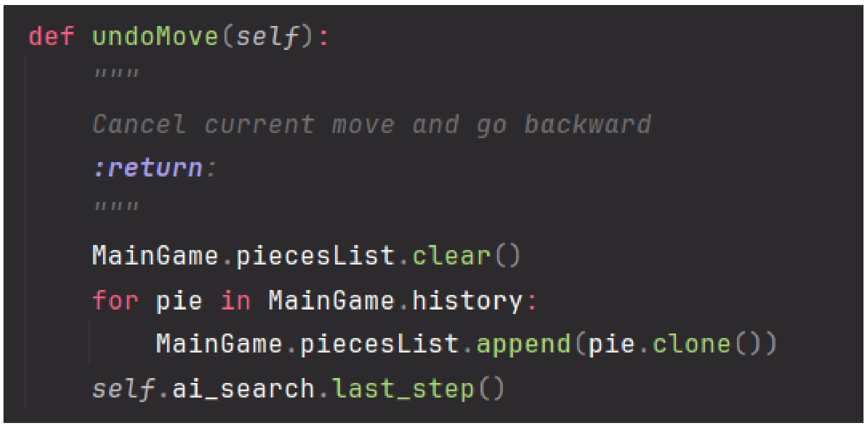
After determining the winner and the loser, it is up to the player to decide whether to restart the game or not, before which the game flow needs to be disconnected in order to facilitate the settlement, but the mid-game reset game does not need to be disconnected from the game flow, in order to solve this problem, I introduced Putdownflag as a semaphore, which is a signal that solves the task of distinguishing between the player and the computer at the same time.



```python
if MainGame.button_go.is_click():
    # Signal control
    self.game_over = False
    MainGame.Putdownflag = const.player1Color
    # Reset board state
    self.piecesInit()
    self.ai_search = search.ai_search()
```

Figure 12 Game Reset

In order to realize the function of repentance, a variation history has been created in the game, which is used to store information about the history of the position. When you regret a move, the board state and the history search went back to last step. However, there is still a problem with this feature, mainly in the fact that you can't repent after selecting a piece, which causes a conflict with the MainGame.piecesSelected semaphore.

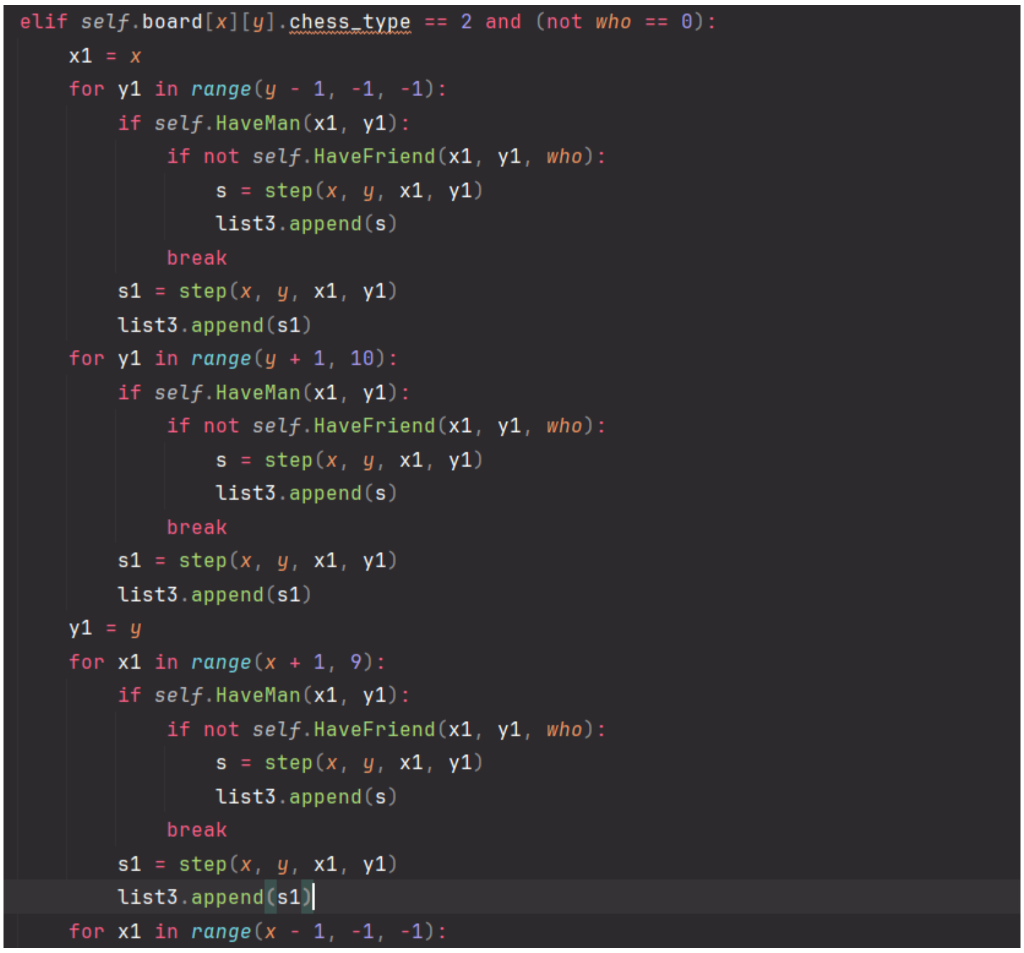


```python
def undoMove(self):
    """

    Cancel current move and go backward
    :return:
    """
    MainGame.piecesList.clear()
    for pie in MainGame.history:
        MainGame.piecesList.append(pie.clone())
    self.ai_search.last_step()
```

Figure 13 Repentance

Moves Checked for validity:

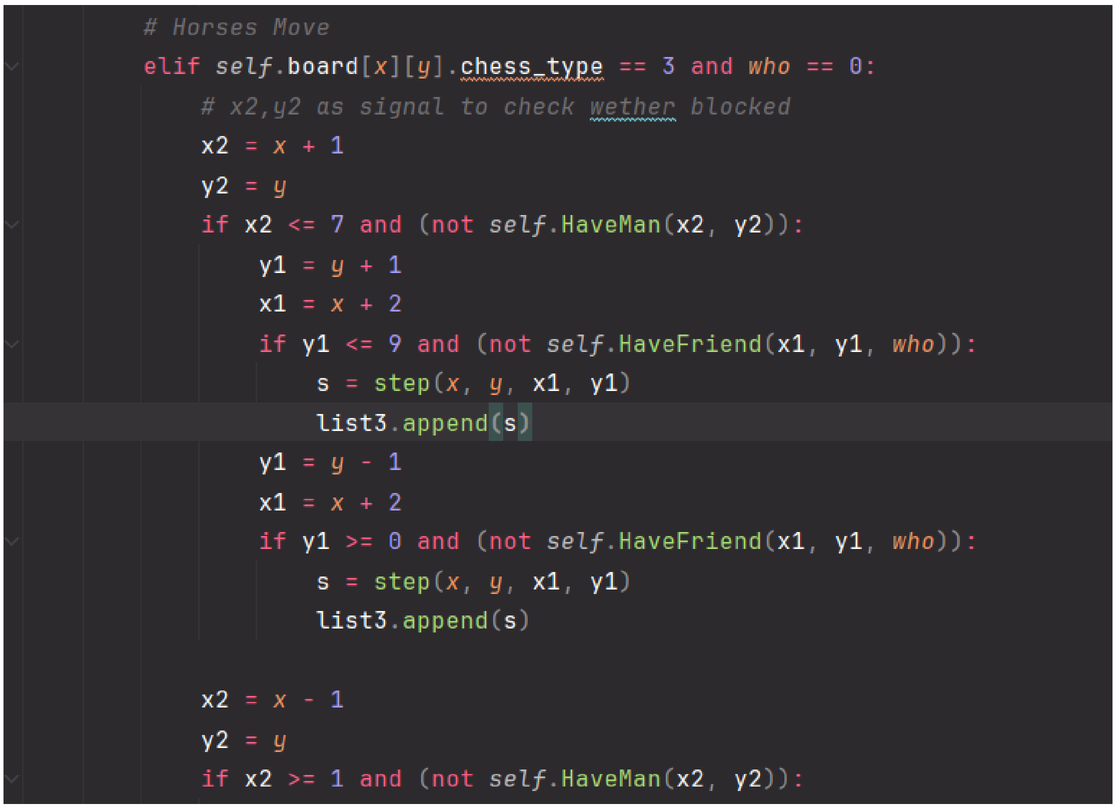
The game involves the judgement of the legality of two types of walkers, one in the linear direction and the other in the diagonal direction; for the linear direction, traverse all positions in that axis upwards until blocked.



```python
elif self.board[x][y].chess_type == 2 and (not who == 0):
    x1 = x
    for y1 in range(y - 1, -1, -1):
        if self.HaveMan(x1, y1):
            if not self.HaveFriend(x1, y1, who):
                s = step(x, y, x1, y1)
                list3.append(s)
            break
        s1 = step(x, y, x1, y1)
        list3.append(s1)
    for y1 in range(y + 1, 10):
        if self.HaveMan(x1, y1):
            if not self.HaveFriend(x1, y1, who):
                s = step(x, y, x1, y1)
                list3.append(s)
            break
        s1 = step(x, y, x1, y1)
        list3.append(s1)
    y1 = y
    for x1 in range(x + 1, 9):
        if self.HaveMan(x1, y1):
            if not self.HaveFriend(x1, y1, who):
                s = step(x, y, x1, y1)
                list3.append(s)
            break
        s1 = step(x, y, x1, y1)
        list3.append(s1)
    for x1 in range(x - 1, -1, -1):
```

Figure 14 Part of c mov detection

For those in the diagonal direction, iterate through the feasible directions and then judge the diagonal positions in the direction.



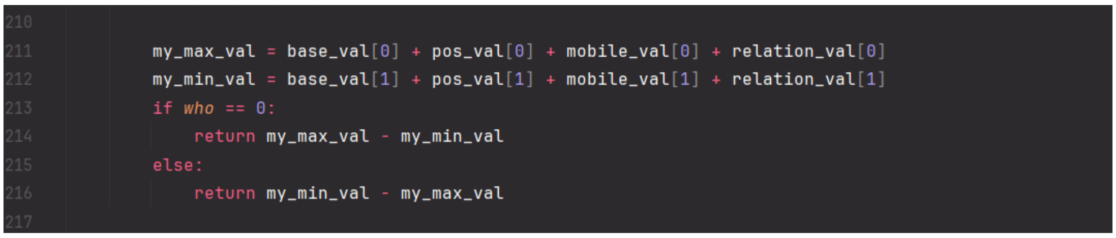
```
# Horses Move
elif self.board[x][y].chess_type == 3 and who == 0:
    # x2,y2 as signal to check wether blocked
    x2 = x + 1
    y2 = y
    if x2 <= 7 and (not self.HaveMan(x2, y2)):
        y1 = y + 1
        x1 = x + 2
        if y1 <= 9 and (not self.HaveFriend(x1, y1, who)):
            s = step(x, y, x1, y1)
            list3.append(s)
        y1 = y - 1
        x1 = x + 2
        if y1 >= 0 and (not self.HaveFriend(x1, y1, who)):
            s = step(x, y, x1, y1)
            list3.append(s)

    x2 = x - 1
    y2 = y
    if x2 >= 1 and (not self.HaveMan(x2, y2)):
```

Figure 15 Part of Horses Move Detection

For pieces with restricted range of movement, check their x-y coordinates, which must not exceed the specified range.

Board Evaluation:

The difficulty in this part is how to consider the value of the pieces. In the initial version, only the positional value of the pieces was considered, and the AI player's strategy appeared to be aggressive and irrational (too much loss after many encounters), so in the final decision, a composite evaluation criterion was used to consider the attack and defense of the pieces in a balanced way.



```
my_max_val = base_val[0] + pos_val[0] + mobile_val[0] + relation_val[0]
my_min_val = base_val[1] + pos_val[1] + mobile_val[1] + relation_val[1]
if who == 0:
    return my_max_val - my_min_val
else:
    return my_min_val - my_max_val
```

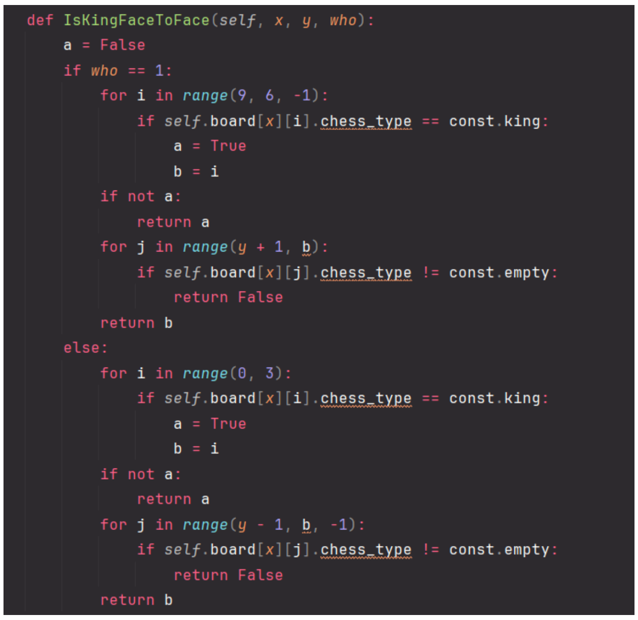
Figure 16 Calculation of Scores

Under this system, the value of a piece consists of four parts, its own value, the value of the position it occupies, the value of the piece it protects and the value of the piece it attacks.

Detecting Check and Victory or Defeat:

For an ordinary general, it is possible to check whether the king's position is within the opponent's reach by traversing the list of pieces, but this means it is difficult to detect the presence of a king face-to-face.

So I decided to write this into the board's detection process.



```python
def IsKingFaceToFace(self, x, y, who):
    a = False
    if who == 1:
        for i in range(9, 6, -1):
            if self.board[x][i].chess_type == const.king:
                a = True
                b = i
        if not a:
            return a
        for j in range(y + 1, b):
            if self.board[x][j].chess_type != const.empty:
                return False
        return b
    else:
        for i in range(0, 3):
            if self.board[x][i].chess_type == const.king:
                a = True
                b = i
        if not a:
            return a
        for j in range(y - 1, b, -1):
            if self.board[x][j].chess_type != const.empty:
                return False
        return b
```

Figure 17 King Face-to-Face detect