CS4386 Assignment 1 (Semester B, 2021-2022)
Xian Jia Le, Ben
56214537
(PYTHON)

Analysis:

Since the game in the assignment is similar to the board game called "Tic-Tac-Toe ", the **minimax approach** will be suitable to be implemented as AI in this game. Although they are similar, their score counting is different. In this game, when a player marks a symbol in the game, the score will be counted based on the current board state and the sequence. If there are two same board states but with different sequences, their score will also be different. We can not save the transposition table by using a hash table that we can implement in Tic-Tac-Toe before. Therefore, **Zobrist hashing** is not suitable in this scenario.

As a result, I decided to implement the AI with the **Mini-Max algorithm**.

Solution:

Main Algorithm:

1. Find the opponent's move by comparing the actual board with its own memorized board.
2. Markdown the opponent's sequence in its own sequence board.
    a. As such, AI can keep track of the sequence of the whole game.
3. Based on the AI's own sequence board, calculate its opponent score.
4. Get the possible move based on the position of the opponent's last action.
    a. In the game, the number of possibles I set to 8.
        i. It is used to increase performance and there is no need to search a lot of empty cells which will behave like human players.
5. Get the suggested move from the Minimax approach.
    a. In the game, the minimax approach is equipped with a-b pruning in order to reduce the time complexity
    b. The evaluation formula is **AI's new score + evaluation value- opponent score** if it is in the maximum node. If it is in minimum node, the formula is the **AI's previous score + evaluation value - opponent predicted score**

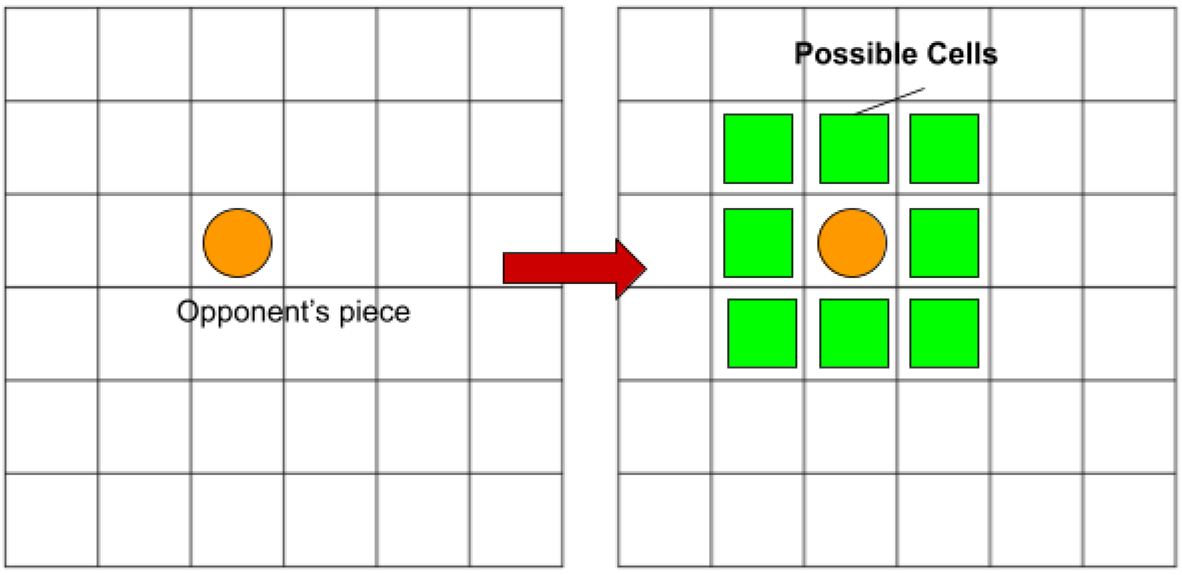
6. Markdown own sequence in the sequence board.
7. Return the suggested move from the minimax approach.

“get_move” code:

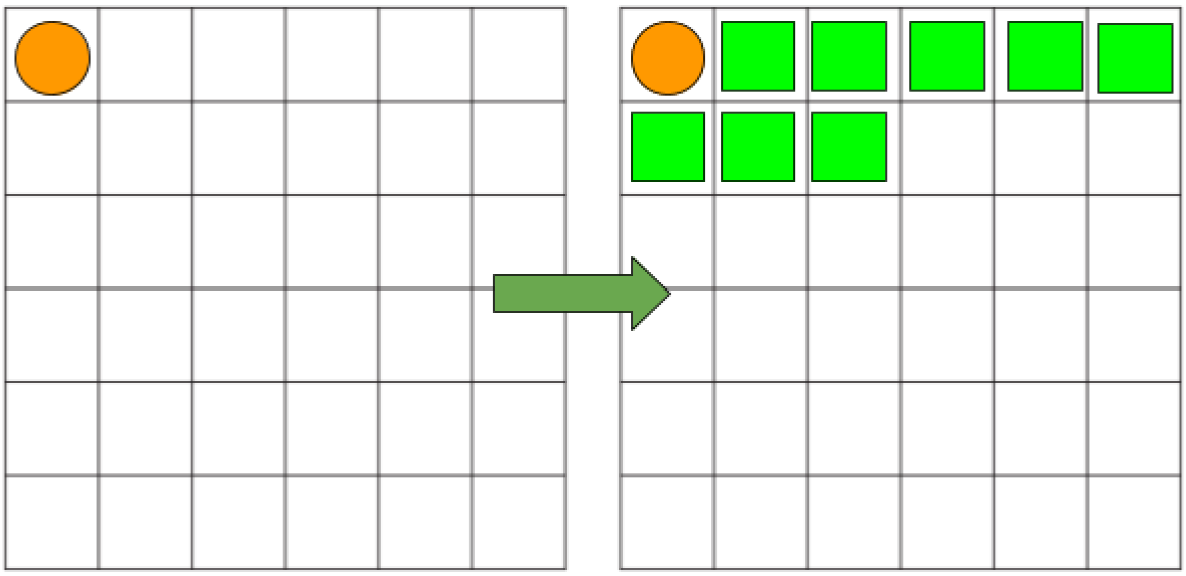
```
 def get_move(self, state, player):
        # find opponent move from state
        different = self.find_different_of_two_Grid(
            state, self.virtualSequenceGrid)  # get opponent cell
        if different != NULL:  # re-organise move sequence
            self.lastSquence = self.lastSquence+1
            self.virtualSequenceGrid[different[0]
                                     ][different[1]] = self.lastSquence
            # calc opponent score
            self.virtualOpponentScore = self.virtualOpponentScore + \
                alignement(state, different[0], different[1])

        print(player, self.score, " vs ", "Opponent ",
self.virtualOpponentScore)
        # get possible move
        games=self.get_possible_cells(different,state,1)
        # perform mini max
        val, move = self.find_move_mini_Max(state.copy(), games,
player, self.score, self.virtualOpponentScore, True, -1000, 1000)
        random_move = move

        # mark the sequence in grid virtual grid
        self.lastSquence = self.lastSquence+1
        self.virtualSequenceGrid[random_move[0]
                                 ][random_move[1]] = self.lastSquence
        return random_move
```

Get possible move

1. Get the neighbour empty cells from the position of the opponent's latest move



2. If the number of neighbour cells is less than (8+extra), find other empty cells in linear search.



“get_possible_cells” Code:

```
    def get_possible_cells(self, searchPos, state,numberOfExtra):
        cells = []
        if searchPos!=NULL:
            # top-left
            if (searchPos[0]-1>=0 and searchPos[1]-1>=0) and
state[searchPos[0]-1][searchPos[1]-1] is None:
                cells.append([searchPos[0]-1, searchPos[1]-1])
            # top
            if (searchPos[1]-1>=0) and
state[searchPos[0]][searchPos[1]-1] is None:
                cells.append([searchPos[0], searchPos[1]-1])
            # top-right
            if (searchPos[0]+1<6 and searchPos[1]-1>=0) and
state[searchPos[0]+1][searchPos[1]-1] is None:
                cells.append([searchPos[0]+1, searchPos[1]-1])
            # left
            if (searchPos[0]-1>=0) and
state[searchPos[0]-1][searchPos[1]] is None:
                cells.append([searchPos[0]-1, searchPos[1]])
            # right
            if (searchPos[0]+1<6) and
state[searchPos[0]+1][searchPos[1]] is None:
                cells.append([searchPos[0]+1, searchPos[1]])
            # bottom-left
            if (searchPos[0]-1>=0 and searchPos[1]+1<6) and
state[searchPos[0]-1][searchPos[1]+1] is None:
                cells.append([searchPos[0]-1, searchPos[1]+1])
            # bottom
            if (searchPos[1]+1<6) and
state[searchPos[0]][searchPos[1]+1] is None:
                cells.append([searchPos[0], searchPos[1]+1])
            # bottom-right
            if (searchPos[0]+1<6 and searchPos[1]+1<6) and
state[searchPos[0]+1][searchPos[1]+1] is None:
                cells.append([searchPos[0]+1, searchPos[1]+1])

        if len(cells)< 8+numberOfExtra:
            for x, row in enumerate(state):
                for y, cell in enumerate(row):
                    if [x,y] not in cells:
                        if cell is None:
                            cells.append([x,y])
                    if len(cells)>=8+numberOfExtra:
                        return cells

        return cells
```

Mini-Max with AB pruning

1. Try each possible move
    a. compute simulated score

b. get the evaluation value and suggested the move by trying the next possible action
c. get the maximized value between alpha and evaluation value
d. If alpha is larger than beta, then prune the node

2. return the result

Code:

```
    def find_move_mini_Max(self, currentBoardState, possibleMove,
player, prevScore, prevOppScore, isMax, alpha, beta):
        # print(len(possibleMove))
        if(len(possibleMove) == 0):
            return 0, NULL
        maxVal = -1000
        minVal = 1000

        for i, move in enumerate(possibleMove):
            stateCopy = currentBoardState.copy()
            possibleMoveCopy = possibleMove.copy()
            possibleMoveCopy.pop(i)
            if(isMax == True):
                update(stateCopy, move[0], move[1], player)
                curScore = alignement(stateCopy, move[0],
move[1])+prevScore
                evalVal, evalMove =
self.find_move_mini_Max(stateCopy.copy(
                ), possibleMoveCopy, player, curScore, prevOppScore,
not(isMax), alpha, beta)
                eval = curScore+evalVal-prevOppScore
                alpha = max(alpha, eval)
                if(eval > maxVal):
                    maxVal = eval
                    moveSelected = move
                if(beta <= alpha):
                    return maxVal, moveSelected

            else:
                update(stateCopy, move[0], move[1], "O"if player ==
"X"else"X")
                curOppScore = alignement(
                    stateCopy, move[0], move[1])+prevOppScore
                evalVal, evalMove =
self.find_move_mini_Max(stateCopy.copy(
                ), possibleMoveCopy, player, prevScore, curOppScore,
not(isMax), alpha, beta)
                eval = prevScore+evalVal-curOppScore
                beta = min(beta, eval)
                if(eval < minVal):
                    minVal = eval
                    moveSelected = move
                if(beta <= alpha):
                    return minVal, moveSelected
        # find the score of two player separately, current self score
is Known, opponent score is unknown
        if(isMax == True):
            return maxVal, moveSelected
        else:
            return minVal, moveSelected
```