REPORTE FINAL BOT – “NACHO”

Luis Ángel Reyes Frausto

Samuel Iván Sánchez Salazar

Introducción.

Para el desarrollo de este bot decidimos implementar el algoritmo minimax ya que es ideal para este tipo de juegos y ya teníamos conocimiento de él por haber sido nuestro tema de exposición. Las distintas implementaciones del bot pueden ser consultadas en el github: https://github.com/Tankel/Tic-Tac-Chec-Bot, donde la versión definitiva de la que estaremos hablando es la de playerNacho.py

Evolución del bot utilizando minimax y otras estrategias.

Antes de implementar el minimax, consideramos que algo crucial del bot, sería el orden y posición en que pondríamos las primeras piezas, así que en los primeros 4 movimientos tratamos de alinear las piezas en horizontalente en la fila de abajo y si esto no era posible la poníamos de manera random. Haciendo esto notamos que le llegaba a ganar hasta el 70% de las partidas al bot random.

Al implementar el minimax, el bot llegaba a ganar hasta el 90% de las partidas al random, al implementar como heursitica claramente el si alguno ganaba (+16 si yo ganaba, -16 si el oponente ganaba), y si ninguno se cumplía calculamos: número máximo de piezas alineadas – número máximo de piezas alineadas del oponente, de tal manera que beneficia al que tenga más.

Después notamos que, si el random por pura suerte empezaba la partida y trataba también de alinear en los primeros 4 movimientos, nuestro bot no hacía nada por bloquearlo, al parecer es efectivo para tratar de alinear las 4 nuestras pero no al evitar que el oponente alinee las otras 4. Es ahí que notamos que la función minimax no funciona en cuanto a profundidad, siendo esta “1”, funciona muy rápido y correctamente, pero al aumentarla no presenta ningún cambio e incluso llega a perder más partidas. Este error no pudimos resolverlo, pero aun así la profundidad de 1 resulta muy efectiva.

Por este problema decidimos implementar otro método llamado blockOpponent el cual es llamado cuando el oponente ya tiene 3 piezas alineadas y yo no, el cual primero coloca una de mis piezas (si es que no tiene en el tablero) en la celda faltante del oponente, sino checa mis piezas en el tablero y se posiciona en cualquier lugar de línea (comiendo una pieza enemiga o poniéndose en la celda vacía).

Otro problema común que presentamos comúnmente es que después de todos estos intentos de movimientos, se me retornaba un tablero vacío (ya que no existía un mejor movimiento dada la heurística), por lo que, si pasa esto, el bot hará un movimiento random.

Finalmente experimentamos un poco con el orden en que ponemos las piezas iniciales (alineadas horizontalmente) peleando unas alineaciones contra otras, donde no había mucha variación, pero nos quedamos con el siguiente orden: Bishop, Pawn, Rook, Knight.

Al final de todo este recorrido, nuestro bot fue capaz de ganarle aproximadamente el 98% de las partidas al bot random.

Documentación.

Atributos**.**

La clase cuenta en general con los siguiente atributos inicializados y distribuidos en las clases __init__ y setColor:

- **name**: El nombre del jugador.
- **pawnDirection**: La dirección de movimiento del peón del jugador (-1 para ir hacia adelante, 1 para ir hacia atrás).
- **currentTurn**: El turno actual del jugador.
- **piecesOnBoard**: Una lista que indica qué piezas del jugador están en el tablero.
- **enemyPiecesOnBoard**: Una lista que indica qué piezas del oponente están en el tablero.
- **piecesCode**: Una lista que contiene los códigos de valor de las piezas del jugador de tal manera que al pasarle una pieza del jugador como índice, devuelve su verdadero color.

Librerías: copy, random, time.

Funciones.

La clase TTCPlayer del bot random nos resultó muy útil usándolo de referencia, por lo que los métodos setColor, updatePawnDirection, sameSign, isInsideBoard, wasPieceMovement, getPawnValidMovements, getBishopValidMovements, getKnightValidMovements, getRookValidMovements y getValidMovements son exactamente los mismos.

- **__init__(self, name**): inicializa los atributos de la clase.

```
    def __init__(self, name):
        self.name = name
        self.pawnDirection = -1
        self.currentTurn = -1

        self.piecesOnBoard = [0] * 5
        self.enemyPiecesOnBoard = [0] * 5
```

- **setColor(self, piecesColor):** establece el color de las piezas del jugador (-1 para negro, 1 para blanco).

```python
def setColor(self, piecesColor):
    self.piecesCode = [0, 1, 2, 3, 4]
    self.piecesCode = [x*piecesColor for x in self.piecesCode]
    self.piecesColor = piecesColor
```

- **__updatePawnDirection(self, board):** actualiza la dirección del peón del jugador en función de su posición en el tablero.

```python
def __updatePawnDirection(self, board):
    # If the pawn is in the limit of the board, it should reverse
    if self.piecesColor in board[0]:
        self.pawnDirection = 1

    # If the pawn is in the start of the board, it should go forward
    if self.piecesColor in board[3]:
        self.pawnDirection = -1
```

- **__sameSign(self, a, b):** verifica si dos piezas tienen el mismo color.

```python
def __sameSign(self, a, b):
    return ((a < 0 and b < 0) or (a > 0 and b > 0))
```

- **__isInsideBoard(self, row, col):** verifica si una posición está dentro del tablero.

```python
def __isInsideBoard(self, row, col):
    return (row >= 0 and row < 4 and col >= 0 and col < 4)
```

- **__wasPieceMovement(self, oldBoard, newBoard):** verifica si un movimiento fue un movimiento válido o una captura.

```python
def __wasPieceMovement(self, oldBoard, newBoard):
    changedSquares = []

    for i in range(4):
        for j in range(4):
            if oldBoard[i][j] != newBoard[i][j]:
                changedSquares.append((i, j))

    if len(changedSquares) != 2:
        return False, False

    def areChangesFromCapture(row1, col1, row2, col2):
        return (newBoard[row1][col1] == 0
                and oldBoard[row2][col2] != 0
                and newBoard[row2][col2] == oldBoard[row1][col1])

    def areChangesFromMovement(row1, col1, row2, col2):
        return (newBoard[row1][col1] == 0
                and newBoard[row2][col2] == oldBoard[row1][col1])

    wasMovement = (areChangesFromMovement(changedSquares[0][0], changedSquares[0][1], changedSquares[1][0],
                   or areChangesFromMovement(changedSquares[1][0], changedSquares[1][1], changedSquares[0][0

    wasCapture = (areChangesFromCapture(changedSquares[0][0], changedSquares[0][1], changedSquares[1][0], ch
                  or areChangesFromCapture(changedSquares[1][0], changedSquares[1][1], changedSquares[0][0],

    return (wasMovement, wasCapture)
```

- **__getPawnValidMovements(self, position, board, pawnDirection):** da los movimientos válidos para un peón en una posición dada.

```
def __getPawnValidMovements(self, position, board, pawnDirection):
    validMovements = []

    row = position[0]
    col = position[1]

    # Move 1 to the front
    newRow = row + pawnDirection
    if self.__isInsideBoard(newRow, col) and board[newRow][col] == 0:
        validMovements.append((newRow, col))

    # Attack to the left
    newCol = col - 1
    if self.__isInsideBoard(newRow, newCol) and board[newRow][newCol] != 0 and not self.__sameSign(board
    [newRow][newCol], board[row][col]):
        validMovements.append((newRow, newCol))

    # Attack to the right
    newCol = col + 1
    if self.__isInsideBoard(newRow, newCol) and board[newRow][newCol] != 0 and not self.__sameSign(board
    [newRow][newCol], board[row][col]):
        validMovements.append((newRow, newCol))

    return validMovements
```

- **__getBishopValidMovements(self, position, board):** da los movimientos válidos para un alfil en una posición dada.

```
def __getBishopValidMovements(self, position, board):
    validMovements = []

    row = position[0]
    col = position[1]

    # To check whether I already encountered a piece in this diagonal or not
    # 0 -> Up-Left Diagonal
    # 1 -> Up-Right Diagonal
    # 2 -> Down-Left Diagonal
    # 3 -> Down-Right Diagonal
    diagEncounteredPiece = [False] * 4

    # Describe the direction of the movement for the bishop in the same
    # order as described above
    movDirection = [[-1, -1],
                    [-1, 1],
                    [1, -1],
                    [1, 1]]

    # A bishop can move at most 3 squares
    for i in range(1, 4):
        # Check 4 directions of movement
        for j in range(4):
            newCol = col + i * movDirection[j][0]
            newRow = row + i * movDirection[j][1]

            # If I haven't found a piece yet in this direction and its inside the board
            if not diagEncounteredPiece[j] and self.__isInsideBoard(newRow, newCol):
                # If the proposed square its occupied
                if board[newRow][newCol] != 0:
                    # If the piece that occupies the square its from the opponent, then its a valid movement
                    if not self.__sameSign(board[row][col], board[newRow][newCol]):
                        validMovements.append((newRow, newCol))
                    diagEncounteredPiece[j] = True
                else:  # If not, just append the movement
                    validMovements.append((newRow, newCol))

    return validMovements
```

- **__getKnightValidMovements(self, position, board):** da los movimientos válidos para un caballo en una posición dada.

```python
def __getKnightValidMovements(self, position, board):
    validMovements = []

    row = position[0]
    col = position[1]

    # Describe the movements of the knight
    movements = [[-2, 1],
                 [-1, 2],
                 [1, 2],
                 [2, 1],
                 [2, -1],
                 [1, -2],
                 [-1, -2],
                 [-2, -1]]

    # Loop through all possible movements
    for move in movements:
        newRow = row + move[0]
        newCol = col + move[1]

        # For the knight we just need to check if the new square is valid and it is not occupied by a piece of the same color
        if self.__isInsideBoard(newRow, newCol) and not self.__sameSign(board[row][col], board[newRow][newCol]):
            validMovements.append((newRow, newCol))

    return validMovements
```

- **__getRookValidMovements(self, position, board):** da los movimientos válidos para una torre en una posición dada.

```python
def __getRookValidMovements(self, position, board):
    validMovements = []

    row = position[0]
    col = position[1]

    # Checks whether or not I have found a piece in this direction
    # 0 - Up
    # 1 - Right
    # 2 - Down
    # 3 - Left
    dirPieceEncountered = [False] * 4

    # Describe the direction of movement for the rook
    # The order is the same as described above
    movDirection = [[-1, 0],
                    [0, 1],
                    [1, 0],
                    [0, -1]]

    # The rook can move maximum 3 squares
    for i in range(1, 4):
        # Loop through all possible movements
        for j in range(4):
            newRow = row + i * movDirection[j][0]
            newCol = col + i * movDirection[j][1]

            if not dirPieceEncountered[j] and self.__isInsideBoard(newRow, newCol):
                if board[newRow][newCol] != 0:
                    if not self.__sameSign(board[newRow][newCol], board[row][col]):
                        validMovements.append((newRow, newCol))
                    dirPieceEncountered[j] = True
                else:
                    validMovements.append((newRow, newCol))

    return validMovements
```

- **__getValidMovements(self, pieceCode, position, board):** da los movimientos válidos para una pieza en una posición dada.

```
def __getValidMovements(self, pieceCode, position, board):
    if abs(pieceCode) == 1:
        return self.__getPawnValidMovements(position, board, self.pawnDirection)
    elif abs(pieceCode) == 2:
        return self.__getBishopValidMovements(position, board)
    elif abs(pieceCode) == 3:
        return self.__getKnightValidMovements(position, board)
    elif abs(pieceCode) == 4:
        return self.__getRookValidMovements(position, board)
    else:
        print("Piece ", pieceCode, " not recognized")
        return []
```

- **__updatePiecesOnBoard(self, board):** actualiza las lista de piezas de ambos jugadores de acuerdo al tablero.

```
def __updatePiecesOnBoard(self, board):
    self.piecesOnBoard = [0] * 5

    for i in range(len(board)):
        for j in range(len(board[0])):
            if self.__sameSign(board[i][j], self.piecesColor):
                self.piecesOnBoard[abs(board[i][j])] = 1
            elif self.__sameSign(board[i][j], -self.piecesColor):
                self.enemyPiecesOnBoard[abs(board[i][j])] = 1
```

- **__moveRandomPiece(self, board):** mueve una pieza aleatoria del jugador en el tablero.

```
def __moveRandomPiece(self, board):
    piece = 0

    while (self.piecesOnBoard[piece] != 1):
        piece = random.randint(1, 4)

    pieceCode = self.piecesCode[piece]
    row = -1
    col = -1

    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == pieceCode:
                row = i
                col = j

                i = len(board)
                break

    validMovements = self.__getValidMovements(pieceCode, (row, col), board)
    if len(validMovements) == 0:
        return board

    newRow, newCol = validMovements[random.randint(
        0, len(validMovements)-1)]

    board[row][col] = 0
    board[newRow][newCol] = pieceCode

    return board
```

- **__putRandomPiece(self, board):**
  Primero bloquea al oponente poniendo una pieza en su casilla faltante si está a punto de ganar.

```
def __putRandomPiece(self, board):
    piece = -1

    oppAlignedValue, _, _, oppMissingPositions, oppAllignedPositions = self.__maxAlignedValue(
        board, -self.piecesColor)

    myAlignedValue, _, myMissingPieces, myMissingPositions, _ = self.__maxAlignedValue(
        board, self.piecesColor)

    # Check the missing pieces in order 1,2,3,4 or -1,-2,-3,-4
    # cause i think ist better to block with Knight that with Rook
    if self.__sameSign(myMissingPieces[0], 1):
        myMissingPieces.sort()
    else:
        myMissingPieces.sort(reverse=True)

    # block if the opponent is about to allign in the first moves
    # not calling the blakcOpponent method because it eats pieces
    if oppAlignedValue == 3 and myAlignedValue != 3:
        for i in oppMissingPositions:
            # print(i)
            x, y = i
            for k in myMissingPieces:
                if (board[x][y] == 0 and self.piecesOnBoard[abs(k)] == 0):
                    board[x][y] = k
                    self.__updatePiecesOnBoard(board)
                    return board
```

```
    if self.availableCaptures > 0:
        for i in oppAllignedPositions:
            x, y = i
            for a in range(4):
                for b in range(4):
                    if self.__sameSign(board[a][b], self.piecesColor) and board[a][b] != 0 and board[a]
                    [b] not in oppMissingPositions:
                        validMoves = self.__getValidMovements(
                            board[a][b], (a, b), board)
                        if i in validMoves:
                            board[x][y] = board[a][b]
                            board[a][b] = 0
                            self.__updatePiecesOnBoard(board)
                            return board
```

En los primeros 4 turnos coloca las piezas conforme a la estrategia descrita.

```
    # Trying to allign in the bottom in order 2,1,4,3 just because it worked better than others orders
    than i tried
    if self.currentTurn == 0:
        for i in range(4):
            if (board[3][i] == 0):
                board[3][i] = self.piecesCode[2]
                return board
    elif self.currentTurn == 1 and self.piecesOnBoard[1] == 0:
        for i in range(4):
            if (board[3][i] == 0):
                board[3][i] = self.piecesCode[1]
                return board
    elif self.currentTurn == 2 and self.piecesOnBoard[4] == 0:
        for i in range(4):
            if (board[3][i] == 0):
                board[3][i] = self.piecesCode[4]
                return board
    elif self.currentTurn == 3:
        for i in range(4):
            if (board[3][i] == 0 and self.piecesOnBoard[3] == 0):
                board[3][i] = self.piecesCode[3]
                return board
        if self.piecesOnBoard[3] == 0:
            for k in myMissingPositions:
                horseMoves = self.__getKnightValidMovements(k, board)
                for h in horseMoves:
                    a, b = h
                    if board[a][b] == 0:
                        board[a][b] = self.piecesCode[3]
                        return board
```

En los demás turnos pone una pieza en la línea que tiene más piezas.

```
        # put in the line that i have the most
        for i in myMissingPositions:
            # print(i)
            x, y = i
            for k in myMissingPieces:
                if (board[x][y] == 0 and self.piecesOnBoard[abs(k)] == 0):
                    board[x][y] = k
                    self.__updatePiecesOnBoard(board)
                    return board
```

Y si nada de esto funciona, pone una pieza en una posición aleatoria disponible.

```
        # if nothing else worked put it randomly
        while (piece == -1 or self.piecesOnBoard[piece] != 0):
            piece = random.randint(1, 4)

        newRow = random.randint(0, 3)
        newCol = random.randint(0, 3)

        while (board[newRow][newCol] != 0):
            newRow = random.randint(0, 3)
            newCol = random.randint(0, 3)

        board[newRow][newCol] = piece * self.piecesColor

        # If a new pawn is put on the board, we should reset its direction.
        if piece == 1:
            self.pawnDirection = -1
        return board
```

- **__getBestMove(self, board, depth, isMaximizingPlayer):** implementación del algoritmo minimax que hace el mejor movimiento según la heurística de __evaluateBoard.

```
    def __getBestMove(self, board, depth, isMaximizingPlayer):
        bestMove = None
        bestScore = float('-inf') if isMaximizingPlayer else float('inf')
        hasChanges = False  # Flag to track changes

        if depth == 0 or self.__checkVictory(board, 1) or self.__checkVictory(board, -1):
            return board, self.__evaluateBoard(board)

        for i in range(len(board)):
            for j in range(len(board[0])):
                localHasChanges = False  # Flag to track changes for each iteration
                # Check for empty cells putting the pieces not in board
                if board[i][j] == 0:
                    for pieceCode in range(1, 5):
                        piecesOnBoard = self.piecesOnBoard if isMaximizingPlayer else self.enemyPiecesOnBoard
                        piecesCode = self.piecesCode if isMaximizingPlayer else []
                        if piecesOnBoard[pieceCode] == 0 and pieceCode in piecesCode:
                            newBoard = [row[:] for row in board]
                            newPiecesOnBoard = copy.deepcopy(
                                self.piecesOnBoard)
                            newEnemyPiecesOnBoard = copy.deepcopy(
                                self.enemyPiecesOnBoard)

                            if isMaximizingPlayer:
                                newBoard[i][j] = pieceCode
                                newPiecesOnBoard[pieceCode] = 1
                            else:
                                newBoard[i][j] = -pieceCode
                                newEnemyPiecesOnBoard[pieceCode] = 1

                            if depth > 1:
                                _, score = self.__getBestMove(
                                    newBoard, depth - 1, not isMaximizingPlayer)
                            else:
                                score = self.__evaluateBoard(newBoard)

                            if (isMaximizingPlayer and score > bestScore) or (not isMaximizingPlayer and score < bestScore):
                                bestScore = score
                                bestMove = [row[:] for row in newBoard]
                                localHasChanges = True  # Changes occurred for this move
                # check for my pieces all the movements
```

```
            # Check for my pieces all the movements
            elif board[i][j] in self.piecesCode:
                piecesOnBoard = self.piecesOnBoard if isMaximizingPlayer else self.enemyPiecesOnBoard
                if piecesOnBoard[board[i][j]] == 1:
                    validMovements = self.__getValidMovements(
                        board[i][j], (i, j), board)
                    for move in validMovements:
                        newBoard = [row[:] for row in board]
                        newPiecesOnBoard = copy.deepcopy(
                            self.piecesOnBoard)
                        newEnemyPiecesOnBoard = copy.deepcopy(
                            self.enemyPiecesOnBoard)

                        if isMaximizingPlayer:
                            newPiecesOnBoard[board[i][j]] = 0
                        else:
                            newEnemyPiecesOnBoard[board[i][j]] = 0

                        newBoard[i][j] = 0
                        newRow, newCol = move
                        newBoard[newRow][newCol] = board[i][j]

                        if depth > 1:
                            _, score = self.__getBestMove(
                                newBoard, depth - 1, not isMaximizingPlayer)
                        else:
                            score = self.__evaluateBoard(newBoard)

                        if score is not None and ((isMaximizingPlayer and score > bestScore) or (not isMaximizingPlayer and score <
                        bestScore)):
                            bestScore = score
                            bestMove = [row[:] for row in newBoard]
                            localHasChanges = True  # Changes occurred for this move

            hasChanges |= localHasChanges  # Update the overall flag based on this iteration
```

```
        # If no changes occurred, return an error value
        if not hasChanges:
            return None, None

        return bestMove, bestScore
```

- **__blockOpponent(self, board, myMissingPieces, oppMissingPositions, oppAlignedPositions):** evita que el oponente gane poniendo una de las piezas en la casilla faltante del oponente o comiendo una de sus piezas.

```
    def __blockOpponent(self, board, myMissingPieces, oppMissingPositions, oppAlignedPositions):
        # Place my pieces in the opponent's missing positions
        for i in oppMissingPositions:
            x, y = i
            for k in myMissingPieces:
                if board[x][y] == 0 and self.piecesOnBoard[abs(k)] == 0:
                    board[x][y] = k
                    self.__updatePawnDirection(board)
                    return board

        # Move one of my pieces in the opponent's line
        for i in range(len(board)):
            for j in range(len(board[0])):
                if board[i][j] in self.piecesCode and board[i][j] != 0:
                    validMovements = self.__getValidMovements(
                        board[i][j], (i, j), board)
                    for oppCoor in oppMissingPositions.union(oppAlignedPositions):
                        if oppCoor in validMovements and oppCoor != (i, j):
                            x, y = oppCoor
                            board[x][y] = board[i][j]
                            board[i][j] = 0
                            self.__updatePawnDirection(board)
                            return board

        return None
```

- **__checkVictory(self, board, piecesColor):** verifica si el jugador dado ha ganado.

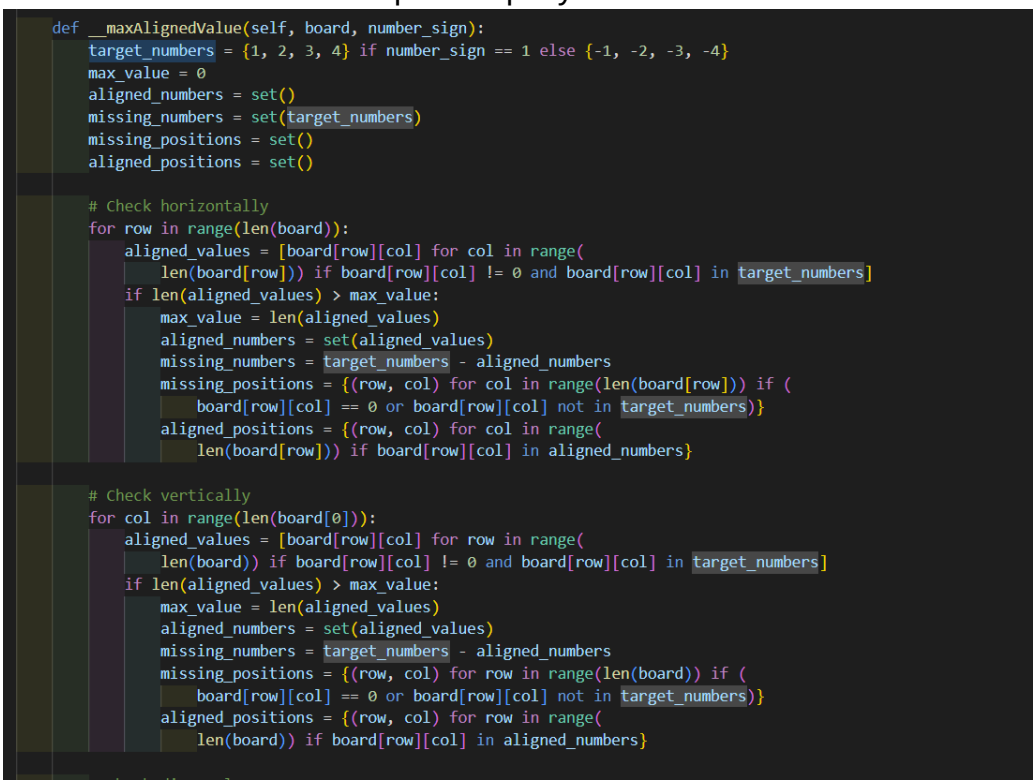
```python
def __checkVictory(self, board, piecesColor):
    target_numbers = {1, 2, 3, 4} if piecesColor == 1 else {-1, -2, -3, -4}

    # Check horizontally
    for row in board:
        if all(piece in target_numbers for piece in row):
            return True

    # Check vertically
    for col in range(len(board[0])):
        column_values = [board[row][col] for row in range(len(board))]
        if all(piece in target_numbers for piece in column_values):
            return True

    # Check diagonals
    diagonal_values = [board[i][i] for i in range(len(board))]
    if all(piece in target_numbers for piece in diagonal_values):
        return True

    reverse_diagonal_values = [
        board[i][len(board) - 1 - i] for i in range(len(board))]
    if all(piece in target_numbers for piece in reverse_diagonal_values):
        return True

    return False
```

- **__evaluateBoard(self, board):** heurística que evalúa el tablero considerando si alguno de los jugadores ha ganado y el número máximo de piezas que tienen alineadas.

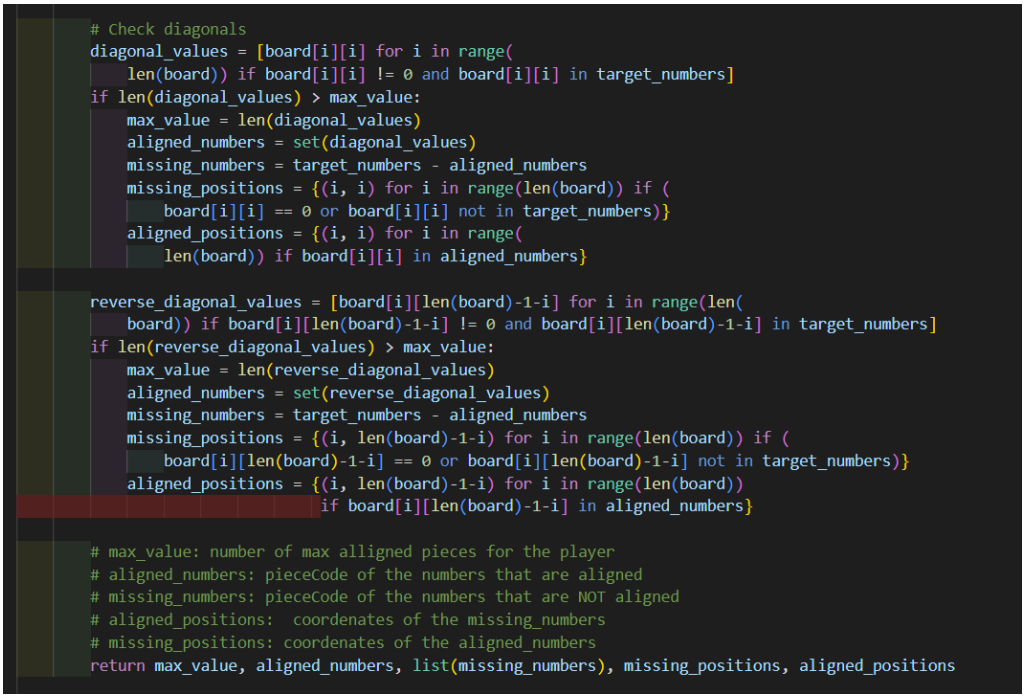
```python
def __evaluateBoard(self, board):
    # Evaluation function for the minimax algorithm
    # It gives a value to a given board state
    # Positive values are good for my bot, negative for the opponent

    # If i have won
    if self.__checkVictory(board, self.piecesColor):
        return 16
    # If the opponent have won
    elif self.__checkVictory(board, -self.piecesColor):
        return -16

    myAlignedValue, _, _, _, _ = self.__maxAlignedValue(
        board, self.piecesColor)

    oppAlignedValue, _, _, _, _ = self.__maxAlignedValue(
        board, -self.piecesColor)

    # Return who has more pieces allgined
    return myAlignedValue - oppAlignedValue
```

- **__maxAlignedValue(self, board, piecesColor):** función que obtiene:
  - Número máximo de piezas alineadas del jugador.
  - Lista de piezas que están alineadas.

- Lista de piezas que NO están alineadas.
- Coordenadas de las celdas faltantes para alinear.
- Coordenadas de las piezas que ya están alineadas.



```python
def __maxAlignedValue(self, board, number_sign):
    target_numbers = {1, 2, 3, 4} if number_sign == 1 else {-1, -2, -3, -4}
    max_value = 0
    aligned_numbers = set()
    missing_numbers = set(target_numbers)
    missing_positions = set()
    aligned_positions = set()

    # Check horizontally
    for row in range(len(board)):
        aligned_values = [board[row][col] for col in range(
            len(board[row])) if board[row][col] != 0 and board[row][col] in target_numbers]
        if len(aligned_values) > max_value:
            max_value = len(aligned_values)
            aligned_numbers = set(aligned_values)
            missing_numbers = target_numbers - aligned_numbers
            missing_positions = {(row, col) for col in range(len(board[row])) if (
                board[row][col] == 0 or board[row][col] not in target_numbers)}
            aligned_positions = {(row, col) for col in range(
                len(board[row])) if board[row][col] in aligned_numbers}

    # Check vertically
    for col in range(len(board[0])):
        aligned_values = [board[row][col] for row in range(
            len(board)) if board[row][col] != 0 and board[row][col] in target_numbers]
        if len(aligned_values) > max_value:
            max_value = len(aligned_values)
            aligned_numbers = set(aligned_values)
            missing_numbers = target_numbers - aligned_numbers
            missing_positions = {(row, col) for row in range(len(board)) if (
                board[row][col] == 0 or board[row][col] not in target_numbers)}
            aligned_positions = {(row, col) for row in range(
                len(board)) if board[row][col] in aligned_numbers}
```



```python
    # Check diagonals
    diagonal_values = [board[i][i] for i in range(
        len(board)) if board[i][i] != 0 and board[i][i] in target_numbers]
    if len(diagonal_values) > max_value:
        max_value = len(diagonal_values)
        aligned_numbers = set(diagonal_values)
        missing_numbers = target_numbers - aligned_numbers
        missing_positions = {(i, i) for i in range(len(board)) if (
            board[i][i] == 0 or board[i][i] not in target_numbers)}
        aligned_positions = {(i, i) for i in range(
            len(board)) if board[i][i] in aligned_numbers}

    reverse_diagonal_values = [board[i][len(board)-1-i] for i in range(len(
        board)) if board[i][len(board)-1-i] != 0 and board[i][len(board)-1-i] in target_numbers]
    if len(reverse_diagonal_values) > max_value:
        max_value = len(reverse_diagonal_values)
        aligned_numbers = set(reverse_diagonal_values)
        missing_numbers = target_numbers - aligned_numbers
        missing_positions = {(i, len(board)-1-i) for i in range(len(board)) if (
            board[i][len(board)-1-i] == 0 or board[i][len(board)-1-i] not in target_numbers)}
        aligned_positions = {(i, len(board)-1-i) for i in range(len(board))
                             if board[i][len(board)-1-i] in aligned_numbers}

    # max_value: number of max alligned pieces for the player
    # aligned_numbers: pieceCode of the numbers that are aligned
    # missing_numbers: pieceCode of the numbers that are NOT aligned
    # aligned_positions:  coordenates of the missing_numbers
    # missing_positions: coordenates of the aligned_numbers
    return max_value, aligned_numbers, list(missing_numbers), missing_positions, aligned_positions
```

- **play(self, board):** Método principal que representa la estrategia general del bot. Considera el tablero actual, el número de capturas disponibles y el número

máximo de turnos permitidos. El método devuelve el tablero actualizado después de que el bot haya realizado su movimiento.

```python
def play(self, board):
    start = time.time()
    self.currentTurn += 1
    self.__updatePiecesOnBoard(board)

    originalBoard = [row[:] for row in board]

    oppAlignedValue, _, _, oppMissingPositions, oppAlignedPositions = self.__maxAlignedValue(
        board, -self.piecesColor)

    myAlignedValue, _, myMissingPieces, _, _ = self.__maxAlignedValue(
        board, self.piecesColor)

    for n in range(1000):
        # put the first 4 pieces in semi-random order trying to allign
        # when there are pieces not in board, put those in the line with most piece
        if self.currentTurn < 3 or sum(self.piecesOnBoard) < 4:
            newBoard = self.__putRandomPiece(board)
        # when the opponent is about to allign and im not, block him or eat one of his pieces
        elif oppAlignedValue == 3 and myAlignedValue != 3 and self.availableCaptures > 0:
            newBoard = self.__blockOpponent(
                board, myMissingPieces, oppMissingPositions, oppAlignedPositions)
        # ehrn all the pieces are in board
        else:
            if n > 0:  # If nothing else has workes make a random move
                newBoard = self.__moveRandomPiece(board)
            # if its the first attempt calculate the bestMove with minimax
            else:
                newBoard, _ = self.__getBestMove(
                    board, 1, self.piecesColor)
```

```python
        # Check if the move was a capture
        _, wasCapture = self.__wasPieceMovement(originalBoard, newBoard)
        if wasCapture:
            if self.availableCaptures > 0:
                self.availableCaptures -= 1
            else:
                board = [row[:] for row in originalBoard]
                continue
        # If it was a valid move break the loop
        if newBoard != originalBoard:
            break

    self.__updatePawnDirection(newBoard)
    print("Time taken:", time.time() - start)

    # To print thr board
    # for row in newBoard:
    #     print(row)

    return newBoard
```

- **reset(self):** método que reinicia los atributos necesarios para iniciar otra partida.

```
def reset(self):
    self.pawnDirection = -1
    self.piecesOnBoard = [0] * 5
    self.enemyPiecesOnBoard = [0] * 5
    self.currentTurn = -1
    self.availableCaptures = 5
```

Conclusión.

El desarrollar este bot fue muy interesante y frustrante a la vez ya que experimentamos muchas ideas que no funcionaron, las cuales se pueden ver en las otras 5 distintas clases player que implementamos.
Consideramos que algo que podemos mejorar, es sin duda nuestra función minimax, ya que como habíamos comentado no logramos implementarla del todo correcto en cuanto el tema de la profundidad para que pudiera visualizar más jugadas a futuro (quizás por las copias de las matrices y arreglos).
Pero al final logramos implementar otras estrategias para que compensar un poco la falla del minimax y para empezar bien el juego (que quizás no son tan óptimas y nos llevan a consultar demasiadas veces el tablero una y otra vez), pero hemos de decir que el bot funciona bastante decente y logra su cometido de ganar la mayoría de las partidas.