Ising-Model-Based Algorithm for Othello Strategy

Jiun-Cheng Jiang and Wen-Ning Wan

Department of Physics National Tsing Hua University, Taiwan

(Dated: December 29, 2022)

Abstract

I. INTRODUCTION

A. Othello

1. Rule

Othello is a two-player strategy board game played on an 8×8 board. Each player has pieces, which are black and white. The game starts with four pieces in the middle of the board, two black and two white. The players take turns placing pieces on the board with their assigned color facing up.

A piece can only be placed on an empty square if it can flip at least one of the opponent's pieces. To flip the opponent's pieces, the new piece must be placed so that an straight (horizontal, vertical, or diagonal) line connects it with at least one of the opponent's pieces. All of the opponent's pieces between the new piece and the connected piece are then flipped to the new piece's color. The object of the game is to have the majority of pieces turned to display your color when the last playable empty square is filled.

The game ends when neither player has a legal move. The winner is the player with the majority of pieces on the board. If both players have the same number of pieces, the game is a draw.

2. Strategy

The strategy of Othello is made around the concept of mobility and stability. Mobility is the number of legal moves a player can make. Stability is the number of pieces that cannot be flipped by the opponent.

a. Grab corner Since there is no way to flip the corner pieces, the player who has the corner pieces has a great advantage. Therefore, the corner has the highest stability and has the highest priority to be grabbed.

Additionally, pieces next to the corner piece are also cannot be flipped by the opponent. Therefore, these pieces has the second highest stability.

b. X square The X square is the square surrounding the corner. The player should avoid placing pieces on the X square since it is the most vulnerable square. The opponent can have the chance to occupy the corner if the player places a piece on the X square. Therefore the X square has the lowest stability.

c. X edge The X edge is the edge surrounding the X square. Keeping X edge may make the opponent places pieces on the X square, which help the player to occupy the corner. Therefore, the X edge has high tactical value but low stability.

d. Danger zone The danger zone is the next rows in from the four edge rows. We should avoid placing pieces on the danger zone since it would help the opponent occupy the edge which has high stability. Therefore, the danger zone has low stability.

e. Mobility Mobility is the number of moves the player is currently able to make, which has significant weight in the opening of game, but diminishes to zero towards the endgame.

3. $\mathbb{Z}2$ Gauge Theory

Othello can be viewed as discretization of $\mathbb{Z}2$ gauge, since there are some common characteristics.

Gauge invariant states of $\mathbb{Z}2$ gauge theory is equivalent to the state of Ising model on the board. The flipping on a board corresponds to acting the unit strength magnetic flux on the board. Note that flipping twice makes it back to white due to $\mathbb{Z}2$. [1]

B. Ising Model

The Ising model has played an important role in the evolution of ideas in statistical physics and quantum field theory.

In order to define Ising model, consider a two dimensional square lattice, on each site or node of the lattice we have an atom or a magnet of spin s_i. In the Ising model, spins have two possible values, up or down which we map to the numerical values $+1$ or -1. The Hamiltonian of the system is given by

$$\hat{H} = -J \sum_{\langle i,j \rangle} s_i s_j - \mu \sum_i s_i \tag{1}$$

The first term is the spin-spin interaction and for J > 0 the system is ferromagnetic. The minimum energy E_0 is obtained for the ground state, which is the unique state in which all spins point in the direction of μ. The minimum energy E_0 is obtained for the ground state, which is the unique state in which all spins point in the direction of B. This is equal to

$$E_0 = -(2J + \mu)N \tag{2}$$

where N is the number of spins in the system.

The partition function is then given by

$$Z = \sum_{\{s_i\}} e^{-\beta H_{s_i}} \tag{3}$$

where $\{s_i\} \equiv \{s_1, s_2, \ldots, s_N\}$ is a spin configuration of the system.

C. Ising-Model-Based Algorithm for Othello Strategy

1. *Concepts*

Since the Othello can be viewed as discretization of $\mathbb{Z}2$ gauge, we can use the Ising model to describe the configuration of Othello game.

2. *Ansatz of the Hamiltonian of Ising Model for Othello*

We make an ansatz of the Hamiltonian of Ising model for Othello, which is

$$\hat{H} = -\sum_{i,j} J(i,j) s_i s_j - \mu(t) M \tag{4}$$

where the first term determined the stability while the second term determined the mobility, and $J(i,j)$ is the interaction between i-piece and j-piece which is

$$J(i,j) = \sum_p r_t s_p^{(ij)} \tag{5}$$

$s_p^{(ij)}$ is the pieces between i-piece and j-piece. r_t is the relevant weight the tactical value of each piece while the middle-16-pieces is set to 1. From Sec. I A 2, we can obtain the following relation: $r_{corner} \geq r_{fixed} \geq r_{Xedge} \geq r_{edge} \geq r_{dangerzone} \geq r_{Xsquare}$. The equal sign

only happens in endgame and at the time $r_t = 1$. M is the number of moves the player is currently able to make, and $\mu(t)$ is a time dependent coefficient used to weight the first term and second term, which should be a decay function.

3. Evolution Method

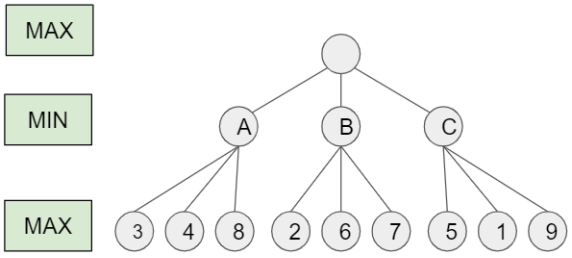
Every time the player makes a move, there would be a state transition. We can view it as that each move imposes a ladder operator on the state and it will lead to the energy change. So we can just calculate the energy change instead of calculating the total energy each time.

As a result, for each move what we need to consider is the interaction between placing piece and placed pieces and the the change of mobility. And then all we have to do is to find the move which make it to the lowest energy, which is the most stable, in the end.

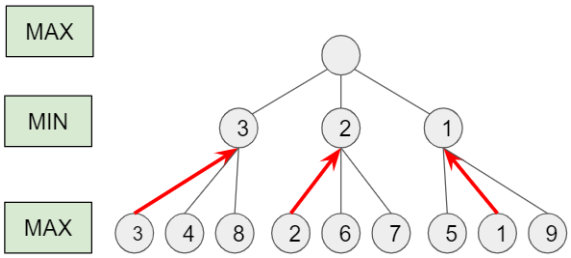
D. Game Algorithm

1. Minimax Algorithm

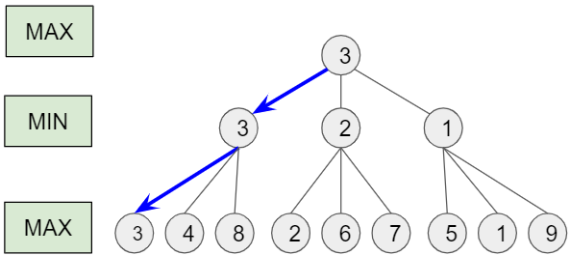
Minimax Algorithm will be the method we use to find the ideal move. To understand Minimax Search, we come across two players. One named Max, and the other named Min. Max eager to get make the with higher score, while Min looks for lower score. This makes them competing one another. We make the score change into a tree diagram, so that we can clearly see how Max and Min would make their choice.



Every layer indicates the situation one is facing, and the nodes under indicates the move choice they have and the outcome they will get. Take Figure as example, Max has A, B, and C conditions to choose, and each condition then leads to a total of nine choices for Min. As previously mentioned, Min will always pick the minimum result, so we return a value to the node with the minimum value of its successors. Min' s choice is demonstrated under.

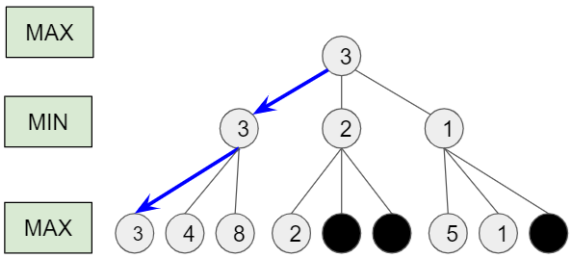


The main idea of this algorithm is to predict our opponent' s next move. If Max picks the A condition, Min gets 3, 4, 8. Obviously, Min will pick 3 for the minimum result, and same for the other conditions. After the values are passed to the upper layer, Max faces 3, 2, 1, and 3 will be its final choice.



Now we are clear with the strategy for both side, and the moves are predictable. However, it is predictable only in known layers. If we want to go further depth, it will take up time and memory space. Therefore, we will need the aid of alpha-beta pruning.

2. Alpha-beta Pruning



By pruning, we make the tree diagram smaller. Some nodes in B condition are filled black is because if we look at a layer from left to right, we come across A condition branch, and then the B condition. We meet 2, and since 2 is smaller than 3 from the A condition, we dismiss the rest of the nodes because B condition will no longer be an option for Max. This is how we save time and memory space. In the former models, we will test with a depth of five, and gradually increase the depth to an acceptable quantity.

E. Parameters

After the algorithm had been planned, we have to decide the value of each node. Basically, it will be the Hamilton of the condition. Looking into the formula, we still miss the J and mu. Since the quantity of J and mu will evaluate will the game goes on. For example, mobility is important in the beginning of the game, and less important in the end of the game. We might use a time dependent function, such as exponential, or sigmoid to do the regression of each parameter. This part remains unclear.

F. Dataset

We will be using the record of 2013 World Othello Championship, which we find it online.[2] It contains 50 fully played game with approximate 60 moves. We will be using them to train and adjust our model.

II. CODE

A. Othello

Appendix A: Code: Othello

1. Dictionary of Pieces' Position

```
'''
Defines the dictionary of the pieaces positions
'''


pos_sheet= {'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4, 'f': 5, 'g': 6, 'h': 7, '
    1': 0, '2': 1, '3': 2, '4': 3, '5': 4, '6': 5, '7': 6, '8': 7}
row_inverse_sheet = {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f', 6: 'g',
    7: 'h'}
col_inverse_sheet = {0: '1', 1: '2', 2: '3', 3: '4', 4: '5', 5: '6', 6: '7',
    7: '8'}
```

2. Board

```
'''
Board class for the game.
'''
import numpy as np
from .dict import pos_sheet, row_inverse_sheet, col_inverse_sheet



class Board(object):
    '''The board class is used to represent the board of the game.'''

    def __init__(self):
        '''
        The constructor of the board class.

        By default: board properties include:
            - board: a 8x8 numpy array.
                - 1 for white.
                - -1 for black.
                - 0 for empty.
        '''
        self._board = np.zeros((8, 8), dtype=int)
        self._board[3, 3] = 1
        self._board[4, 4] = 1
        self._board[3, 4] = -1
        self._board[4, 3] = -1

    @staticmethod
    def valid_pos_list(pos):
        if (type(pos) == list):
```

```
            pass
        elif (type(pos) == str):
            if (len(pos) != 2):
                return False
            if (pos[0] not in pos_sheet or pos[1] not in pos_sheet):
                return False
            pos = [pos_sheet[pos[0]], pos_sheet[pos[1]]]
        else:
            return False
        return pos

    @staticmethod
    def pos_name(pos):
        '''Returns a list of postion names in the form of strings.'''
        name = []
        for p in pos:
            name.append(
                row_inverse_sheet[p[0]] + col_inverse_sheet[p[1]])
        return name

    def get_board(self):
        '''Returns the board.'''
        return self._board

    def set_board(self, board):
        '''Sets the board.'''
        for i, j in np.ndindex(board.shape):
            self._board[i, j] = board[i, j]

    def __getpiece__(self, pos):
```

```
        '''Returns the color piece at a given position.'''
        return self._board[pos[0], pos[1]]

    def __setpiece__(self, pos, color):
        '''
        Sets the piece at a given position.
        '''
        self._board[pos[0], pos[1]] = color

    def _check_valid_move_white(self, i, j):
        '''Checks if a move is valid for white.'''
        if (self.__getpiece__([i, j]) != 0):
            return False
        for m in [-1, 0, 1]:
            for n in [-1, 0, 1]:
                if (m == 0 and n == 0):
                    continue
                if (i+m < 0 or i+m > 7 or j+n < 0 or j+n > 7):
                    continue
                if (self.__getpiece__([i+m, j+n]) == -1):
                    for k in range(2, 8):
                        if (i+k*m < 0 or i+k*m > 7 or j+k*n < 0 or j+k*n > 7)
                            :
                            break
                        if (self.__getpiece__([i+k*m, j+k*n]) == 0):
                            break
                        if (self.__getpiece__([i+k*m, j+k*n]) == 1):
                            return True
        return False
```

```
    def valid_move_white(self):
        '''Returns a list of valid moves for white.'''
        valid_moves = []
        for i in range(8):
            for j in range(8):
                if (self._check_valid_move_white(i, j)):
                    valid_moves.append([i, j])
        return valid_moves


    def _check_valid_move_black(self, i, j):
        '''Checks if a move is valid for black.'''
        if (self.__getpiece__([i, j]) != 0):
            return False
        for m in [-1, 0, 1]:
            for n in [-1, 0, 1]:
                if (m == 0 and n == 0):
                    continue
                if (i+m < 0 or i+m > 7 or j+n < 0 or j+n > 7):
                    continue
                if (self.__getpiece__([i+m, j+n]) == 1):
                    for k in range(2, 8):
                        if (i+k*m < 0 or i+k*m > 7 or j+k*n < 0 or j+k*n > 7)
                            :
                            break
                        if (self.__getpiece__([i+k*m, j+k*n]) == 0):
                            break
                        if (self.__getpiece__([i+k*m, j+k*n]) == -1):
                            return True
        return False
```

```
    def valid_move_black(self):
        '''Returns a list of valid moves for black.'''
        valid_moves = []
        for i in range(8):
            for j in range(8):
                if (self._check_valid_move_black(i, j)):
                    valid_moves.append([i, j])
        return valid_moves

    def _flip_white(self, i, j):
        '''Flips the pieces for white.'''
        for m in [-1, 0, 1]:
            for n in [-1, 0, 1]:
                if (m == 0 and n == 0):
                    continue
                if (i+m < 0 or i+m > 7 or j+n < 0 or j+n > 7):
                    continue
                if (self._board[i+m, j+n] == -1):
                    for k in range(2, 8):
                        if (i+k*m < 0 or i+k*m > 7 or j+k*n < 0 or j+k*n > 7)
                            :
                            break
                        if (self.__getpiece__([i+k*m, j+k*n]) == 0):
                            break
                        if (self.__getpiece__([i+k*m, j+k*n]) == 1):
                            for l in range(1, k):
                                self.__setpiece__([i+l*m, j+l*n], 1)
                            break

    def place_white(self, pos):
```

```
        '''Places a white piece on the board.'''
        self.__setpiece__([pos[0], pos[1]], 1)
        self._flip_white(pos[0], pos[1])

    def _flip_black(self, i, j):
        '''Flips the pieces for black.'''
        for m in [-1, 0, 1]:
            for n in [-1, 0, 1]:
                if (m == 0 and n == 0):
                    continue
                if (i+m < 0 or i+m > 7 or j+n < 0 or j+n > 7):
                    continue
                if (self._board[i+m, j+n] == 1):
                    for k in range(2, 8):
                        if (i+k*m < 0 or i+k*m > 7 or j+k*n < 0 or j+k*n > 7)
                            :
                            break
                        if (self.__getpiece__([i+k*m, j+k*n]) == 0):
                            break
                        if (self.__getpiece__([i+k*m, j+k*n]) == -1):
                            for l in range(1, k):
                                self.__setpiece__([i+l*m, j+l*n], -1)
                            break

    def place_black(self, pos):
        '''Places a black piece on the board.'''
        self.__setpiece__([pos[0], pos[1]], -1)
        self._flip_black(pos[0], pos[1])

    def pseudo_flip_white(self, pos: list):
```

```
        '''
        Counting how many flips can make for white.

        Returns a list of positions that would be flipped.
        '''
        i = pos[0]
        j = pos[1]
        can_place = []
        for m in [-1, 0, 1]:
            for n in [-1, 0, 1]:
                if (m == 0 and n == 0):
                    continue
                if (i+m < 0 or i+m > 7 or j+n < 0 or j+n > 7):
                    continue
                if (self._board[i+m, j+n] == -1):
                    for k in range(2, 8):
                        if (i+k*m < 0 or i+k*m > 7 or j+k*n < 0 or j+k*n > 7)
                            :
                            break
                        if (self.__getpiece__([i+k*m, j+k*n]) == 0):
                            break
                        if (self.__getpiece__([i+k*m, j+k*n]) == 1):
                            for l in range(1, k):
                                can_place.append([i+l*m, j+l*n])
                            break
        return can_place

    def pseudo_flip_black(self, pos):
        '''
        Counting how many flips can make for black.
```

```
        Returns a list of positions that would be flipped.
        '''
        i = pos[0]
        j = pos[1]
        can_place = []
        for m in [-1, 0, 1]:
            for n in [-1, 0, 1]:
                if (m == 0 and n == 0):
                    continue
                if (i+m < 0 or i+m > 7 or j+n < 0 or j+n > 7):
                    continue
                if (self._board[i+m, j+n] == 1):
                    for k in range(2, 8):
                        if (i+k*m < 0 or i+k*m > 7 or j+k*n < 0 or j+k*n > 7)
                            :
                            break
                        if (self.__getpiece__([i+k*m, j+k*n]) == 0):
                            break
                        if (self.__getpiece__([i+k*m, j+k*n]) == -1):
                            for l in range(1, k):
                                can_place.append([i+l*m, j+l*n])
                            break
        return can_place

    def print_board(self):
        '''Prints the board.'''
        labelsx = ['1', '2', '3', '4', '5', '6', '7', '8']
        labelsy = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
```

```
        print('  ', end='')
        for i in range(8):
            print(labelsx[i], end=' ')
        print()
        for i in range(8):
            print(labelsy[i], end=' ')
            for j in range(8):
                if (self.__getpiece__([i, j]) == -1):
                    print(' ', end=' ')
                elif (self.__getpiece__([i, j]) == 1):
                    print(' ', end=' ')
                else:
                    print('.', end=' ')
            print()
        print()

    def __repr__(self):
        return self.print_board()
```

3. Game

```
'''
The game class is used to represent the game process of othello.
'''
import numpy as np
from .board import Board


class game(Board):
    '''
```

```
    The game class is used to represent the game of othello.

    The game class inherits the board class.
    '''

    def __init__(self):
        '''
        The constructor of the game class.

        By default: game properties include:
            - board: a 8x8 numpy array.
                - 1 for white.
                - -1 for black.
                - 0 for empty.
            - turn: -1 for black and 1 for white.
            - end: False for not ended and True for ended.

        '''
        super().__init__()
        self._turn = -1
        self.end = False

    @property
    def turn(self):
        return self._turn

    @turn.setter
    def turn(self, turn):
        self._turn = turn
```

```
    def check_next_turn(self):
        '''Checks the next turn and whether the game is end.'''
        if self.valid_move_black() == [] and self.valid_move_white() == []:
            self.end = True
            return
        elif self.turn == -1 and self.valid_move_white() == []:
            return
        elif self.turn == 1 and self.valid_move_black() == []:
            return
        else:
            self.turn *= -1
            return


    def to_place(self, pos):
        '''Places a piece at a given valid position.'''
        pos = self.valid_pos_list(pos)
        if (pos == False):
            return False
        if self.turn == -1:
            valid = self.valid_move_black()
            if (pos in valid):
                self.place_black(pos)
                self.check_next_turn()
                return True
            else:
                return False
        elif self.turn == 1:
            valid = self.valid_move_white()
            if (pos in valid):
                self.place_white(pos)
```

```
                self.check_next_turn()
                return True
            else:
                return False

    def count_black(self):
        '''Returns the number of black pieces.'''
        return np.count_nonzero(self.get_board() == -1)

    def count_white(self):
        '''Returns the number of white pieces.'''
        return np.count_nonzero(self.get_board() == 1)

    def get_lead(self):
        '''Returns side of the lead.'''
        if self.count_black() > self.count_white():
            return -1
        elif self.count_black() < self.count_white():
            return 1
        else:
            return 0

    def winner(self):
        '''Returns the winner.'''
        if self.end == True:
            if self.count_black() > self.count_white():
                return 'Black'
            elif self.count_black() < self.count_white():
                return 'White'
            else:
```

```
                return 'Draw'
        else:
            return False
```

4. Playing Mode

```
import numpy as np
from .util.game import game



def pvp():
    match = game()
    match.print_board()
    while match.end == False:
        if match.turn == -1:
            print('Black\'s turn')
            print('Valid moves for black:', str(
                match.pos_name(match.valid_move_black())).upper())
        else:
            print('White\'s turn')
            print('Valid moves for white:', str(
                match.pos_name(match.valid_move_white())).upper())
        pos = input(
            'Please enter the position you want to place a piece: ').lower().
               strip()
        if match.to_place(pos):
            match.print_board()
        else:
            print('Invalid position. Please try again.')
    print('Game over. The winner is', match.winner())
```

```
def load_from_txt(fn, output=False):
    color, row, col = np.loadtxt(fn, dtype=int, unpack=True)
    for i in range(len(color)):
        if color[i] == 0:
            color[i] = -1
    match = game()
    round = 0
    if output:
        match.print_board()
    board = [match.get_board().copy()]
    while match.end == False:
        if round == len(color):
            print('Check file', fn, 'for invalid number of moves (Game does
                not end).')
            return
        if match.turn == color[round]:
            if match.to_place([row[round], col[round]]):
                if output:
                    match.print_board()
                board.append(match.get_board().copy())
                round += 1
            else:
                print('Check file', fn, 'for invalid move at round =.',round
                    +1)
                return
        else:
            print('Check file', fn, 'for invalid turn at round =.',round+1)
            return
```

```
    print('Game over. The winner is', match.winner())
    return board


def GreedyBot(strategy=1):
    '''
    GreedyBot plays against the player.
    strategy = 1: GreedyBot plays the move that flips the most pieces.
    strategy = 2: GreedyBot plays the move that gives the least actions to
       the player.
    '''
    match = game()
    match.print_board()
    while match.end == False:
        if match.turn == -1:
            print('Your turn')
            print('Your valid moves:', str(
                match.pos_name(match.valid_move_black())).upper())
            pos = input(
                'Please enter the position you want to place a piece: ').
                   lower().strip()
        else:
            if strategy == 1:
                valid_moves = match.valid_move_white()
                number_of_flips = []
                for i in valid_moves:
                    number_of_flips.append(len(match.pseudo_flip_white(i)))
                pos = valid_moves[number_of_flips.index(max(number_of_flips))
                   ]
            elif strategy == 2:
                valid_moves = match.valid_move_white()
```

```
                number_of_actions = []
                for i in valid_moves:
                    tmp_match = game()
                    tmp_match.set_board(match.get_board())
                    tmp_match.turn = 1
                    tmp_match.to_place(i)
                    number_of_actions.append(len(tmp_match.valid_move_black()
                        ))
                pos = valid_moves[number_of_actions.index(min(
                    number_of_actions))]
            else:
                print('Invalid strategy.')
                return
        if match.to_place(pos):
            match.print_board()
        else:
            print('Invalid position. Please try again.')
    print('Game over. The winner is', match.winner())
```

Appendix B: Code: Algorithm

---

[1] K. Hashimoto, N. Iizuka, and S. Sugishita, Phys. Rev. D **96**, 126001 (2017).

[2] K.-P. Chan, 2013 othello world cup game records.