Laboratory Report

Name：Xiaoyi Zhang

Number：JL21010001

Reversi

Problem Description

Reversi, also known as apple chess, flip chess, is a classic strategic game. Generally, the two sides of the chess piece are black and white, so it is called ' black and white chess '. Because when playing chess, the other party 's chess pieces are turned into their own chess pieces, so it is also called ' flipping chess '. The red and green chess pieces on both sides become ' apple chess '. It uses an 8 * 8 board, with two black and white players playing chess in turn, and finally winning in many ways.
Rules :
(1) Black side first, both sides alternate playing chess.
(2) One-step legitimate chess steps include : drop a new piece in a space, and reverse one or more pieces of the opponent.
(3) Between the newly fallen pieces and the existing pieces of the same color on the chessboard, all the pieces that the other side is caught in must be reversed. Can be horizontal clamp, vertical clamp, oblique clamp. Clamp position must be all opponent ' s pieces, can not have spaces.
(4) One-step chess can be turned in several directions (horizontal, vertical, diagonal). Any piece caught must be turned over, and the player has no right to choose not to turn a piece.
(5) Unless you flip at least one of the opponent 's pieces, otherwise you can not fall. If one side does not have a legitimate move, that is, no matter where he goes, he cannot flip at least one of his opponent 's pieces, then he can only abstain from this round, and his opponent continues to fall until he has a legitimate move to play.
(6) If a party has at least one legal step to go, he must fall, and shall not abstain.
(7) The game continues until the board is filled or both sides have no legal moves.

Test Requirement

- Implement miniAlphaGo for Reversi using Monte Carlo tree search algorithm.
- Using Python language.
- The algorithm part needs to be implemented by itself. Do not use existing packages, tools, or interfaces

Experimental Environment

python 3.10 pycharm 2021.3.2

Summary Design

The design idea of this experiment is Monte Carlo method. MCTS is a ' statistical simulation method '. In the 1940s, to build nuclear weapons, Feng. Neumann et al.invented the algorithm. It is named after Monte Carlo, suggesting that it is based on probability. Assuming that we want to calculate the area of an irregular shape, we only need to randomly throw a point in a rectangle containing this irregular shape. For each point, N + 1, if this point is in an irregular shape, W + 1. The probability of falling into irregular graphics is W / N. When enough points are thrown, we can

think : Irregular graphic area = Rectangular area * W / N. The UCT algorithm is an improvement of the Monte Carlo method. It is easier to obtain the optimal solution than the Monte Carlo method. Its basic structure is the same as the Monte Carlo method. It is mainly divided into four steps : selection, expansion, simulation and back propagation. The value function is defined as :

$$value = child.reward/child.visits + c\sqrt{\frac{2Ln(node.visits)}{child.visits}}$$

Probably can be divided into four steps. Selection, Expansion, Simulation, Back Propagation. In the beginning, the search tree has only one node, which is where we need to make decisions. Each node in the search tree contains basic information : the situation represented, the number of visits, the cumulative score, and the current coordinate position.

(1) selection

In the selection stage, it is necessary to select the most urgent node N that needs to be expanded from the root node, that is, the situation R to make the decision, and the situation R is the first node to be checked in each iteration. For the situation being checked, there may be three possibilities : first, all the feasible actions of the node have been expanded ; second, the feasible actions of the node have not been expanded ; third, the node game has ended.
For these three possibilities : if all feasible actions have been extended, then the UCB formula will be used to calculate the UCB value of all child nodes of the node, and find the child node with the largest value to continue checking. Iterate down repeatedly. If the examined situation still has child nodes that have not been expanded, then this node is considered the target node N of this iteration, and find action A where N has not been expanded. Perform step (2). If the node being checked is a node that the game has ended. Then execute step (4) directly from this node.
The number of visits to each checked node will increase at this stage. After repeated iterations, we will find a node at the bottom of the search tree to continue the next steps.

(2) extension

At the end of the selection phase, a node N that is most urgently expanded and an action A that has not yet been expanded are found. Create a new node Nn as a new child node of N in the search tree. The situation of Nn is the situation of node N after executing action A.
(3) simulation

In order to get Nn an initial score. Starting from Nn, let the game proceed randomly until a game ending is obtained, which will be used as the initial score of Nn. Victory / failure is generally used as a score, only 1 or 0.
(4) back propagation

After the simulation of Nn ends, its parent node N and all nodes on the path from the root node to N will add their own cumulative scores based on the results of this simulation. If a game ending is found directly in the selection of [1], update the score based on that ending. Each iteration will expand the search tree. As the number of iterations increases, the size of the search tree also increases. When a certain number of iterations or time is over, select the best child node under the root node as the result of this decision.

Detail Design

Game

First determine whether the end, if the game is not over, there is a process to switch players during the game, and to obtain the coordinates of the player at this time

```
1 def put_stones(self, event):  # Place a chess piece
```

```
        # Is the game over
        if self.validBoard == False:
            self.validBoard = True
            self.board = rvs.getInitialBoard()
            self.isPayerTurn = True

            for numid in self.step:
                self.delete(numid)
            self.step = []
            self.refresh()
            return

        # Computer rounds
        if not (self.isPayerTurn):
            return
        # Player rounds
        x = self.canvasx(event.x)
        y = self.canvasy(event.y)
        # Obtain coordinates
        i = int(x / self.cell_size)
        j = int(y / self.cell_size)
        if self.board[i][j] != 0 or rvs.updateBoard(self.board,
rvs.PLAYER_NUM, i, j, checkonly=True) == 0:
            return

        rvs.updateBoard(self.board, rvs.PLAYER_NUM, i, j)
        self.refresh()
        isPayerTurn = False
        self.after(100, self.AI_move)
```

Count the time spent in one step, according to the current board, determine whether the board is terminated, if the current player does not have a legitimate chess position, then switch players ; if another player does not have a legitimate position, the game is stopped.

```
def AI_move(self):
        while True:
            player_possibility = len(rvs.possible_positions(self.board,
rvs.PLAYER_NUM))
            mcts_possibility = len(rvs.possible_positions(self.board,
rvs.COMPUTER_NUM))
            if mcts_possibility == 0:
                break
            start= time.time()
            stone_pos = rvs.mctsNextPosition(self.board)
            end =time.time()
            one_time=end-start
            print("Computer position:", stone_pos)
            print("Step time:",format(one_time, '.4f'),"s")
            total.append(one_time)
            rvs.updateBoard(self.board, rvs.COMPUTER_NUM, stone_pos[0],
stone_pos[1])
            self.refresh()

            player_possibility = len(rvs.possible_positions(self.board,
rvs.PLAYER_NUM))
```

```
            mcts_possibility = len(rvs.possible_positions(self.board,
rvs.COMPUTER_NUM))

            if mcts_possibility == 0 or player_possibility > 0:
                break

        if player_possibility == 0 and mcts_possibility == 0:
            self.showResult()
            self.validBoard = False

        self.isPayerTurn = True
```

Print winner, black chess win, white chess win, draw three possible

```
def showResult(self):
        player_stone = rvs.countTile(self.board, rvs.PLAYER_NUM)
        mcts_stone = rvs.countTile(self.board, rvs.COMPUTER_NUM)

        if player_stone > mcts_stone:
            tkinter.messagebox.showinfo('游戏结束', "你获胜了")

        elif player_stone == mcts_stone:
            tkinter.messagebox.showinfo('游戏结束', "平局")

        else:
            tkinter.messagebox.showinfo('游戏结束', "你失败了")
        print(sum(total))

```

renew

```
def refresh(self):
        for i in range(rvs.BOARD_SIZE):
            for j in range(rvs.BOARD_SIZE):
                x0 = i * self.cell_size + self.margin
                y0 = j * self.cell_size + self.margin

                if self.board[i][j] == 0:
                    continue
                if self.board[i][j] == rvs.PLAYER_NUM:
                    bcolor = "#000000"
                if self.board[i][j] == rvs.COMPUTER_NUM:
                    bcolor = "#ffffff"
                self.create_oval(x0 + 2, y0 + 2, x0 + self.cell_size - 2, y0
+ self.cell_size - 2, fill=bcolor,
                                         width=0)
```

MCTS algorithm

Firstly, the chessboard is initialized, the rows and columns are initialized to 0, and the middle four positions of the initial setting are crossed to put the black and white chess pieces.

```
def getInitialBoard():
    board = {}
    # Initialize the chessboard coordinates to 0
    for i in range(0, BOARD_SIZE):
```

```
        board[i] = {}

        for j in range(0, BOARD_SIZE):
            board[i][j] = 0
    # Place the initial crossed pieces in the center of the board
    board[BOARD_SIZE / 2 - 1][BOARD_SIZE / 2 - 1] = COMPUTER_NUM
    board[BOARD_SIZE / 2][BOARD_SIZE / 2] = COMPUTER_NUM

    board[BOARD_SIZE / 2 - 1][BOARD_SIZE / 2] = PLAYER_NUM
    board[BOARD_SIZE / 2][BOARD_SIZE / 2 - 1] = PLAYER_NUM

    return board
```

Loop traversal from left to right from top to bottom to get the number of returned pieces

```
def countTile(board, tile):
    stones = 0
    for i in range(0, BOARD_SIZE):
        for j in range(0, BOARD_SIZE):
            if board[i][j] == tile:
                stones += 1

    return stones
```

If there is a chess piece in this position, jump out of the loop, there is no chess piece in this position, and then judge whether the walk is legal. If it is legal, return the coordinate of the chess piece, add the position of the possible lower part after the array positions.

```
def possible_positions(board, tile):
    positions = []
    for i in range(0, BOARD_SIZE):
        for j in range(0, BOARD_SIZE):
            if board[i][j] != 0:
                continue
            if updateBoard(board, tile, i, j, checkonly=True) > 0:
                positions.append((i, j))
    return positions
```

Whether it is legal to go, if legal return need to flip the list of pieces. Temporarily put the tile to the position of the board array. If it is currently a player 's fall, judge the computer pieces that need to change ; if the current is the computer, then judge the player pieces that need to change. For the pieces to be flipped : traverse the pieces in 8 directions around x, y, and the pieces that are satisfied in the chessboard and the currently traversed pieces are each other 's pieces, then continue to search for the next position in this direction, and then judge whether it is out of bounds, out of bounds, stop (this step is mainly used to judge the position of the pieces on the boundary), change the search direction, go all the way to the out of bounds or not. The position of the pieces, out of bounds, there is no piece to flip. It is your own chess piece, all the chess pieces in the middle must be flipped to determine whether you have searched for your own chess piece. If so, the coordinates are regressed back to the starting point along the direction, and the coordinates along the way are recorded. The position, that is, where the inversion is needed, removes the chess pieces temporarily placed in front of it, that is, restores the chess board, and there is no chess piece to be flipped, then the move is illegal.

```
def updateBoard(board, tile, i, j, checkonly=False):
```

```
    # This location already has pieces or bounds, returning False
    reversed_stone = 0

    # temporarily place tile in the specified location
    board[i][j] = tile
    if tile == 2:
        change = 1
    else:
        change = 2

    # Chess to be flipped
    need_turn = []
    for xdirection, ydirection in direction:
        x, y = i, j
        x += xdirection
        y += ydirection
        if isOnBoard(x, y) and board[x][y] == change:
            x += xdirection
            y += ydirection
            if not isOnBoard(x, y):
                continue
            # Go straight to the out of bounds or not the position of the other chess piece.
            while board[x][y] == change:
                x += xdirection
                y += ydirection
                if not isOnBoard(x, y):
                    break
            # Out of bounds, there is no piece to flip
            if not isOnBoard(x, y):
                continue
            # Is your own chess piece, all the pieces in the middle to flip
            if board[x][y] == tile:
                while True:
                    x -= xdirection
                    y -= ydirection
                    # Back to the starting point is over
                    if x == i and y == j:
                        break
                    # Pieces to be flipped
                    need_turn.append([x, y])
    # Remove the previously temporarily placed pieces, that is, restore the board
    board[i][j] = 0  # restore the empty space
    # There is no chess piece to be flipped, then the move is illegal. The rules of flipping chess.
    for x, y in need_turn:
        if not (checkonly):
            board[i][j] = tile
            board[x][y] = tile  # Flip the pieces
        reversed_stone += 1
    return reversed_stone

```

The UCB value is calculated first, and the node _ tuple that needs to be passed in is a tuple with four elements. The number of rewards is the number of wins nplayout is the number of simulated matches, cval is a constant

```
def ucb1(node_tuple, t, cval):
        name, nplayout, reward, childrens = node_tuple

        if nplayout == 0:
            nplayout = 0.00000000001

        if t == 0:
            t = 1
        #reward is the number of wins nplayout is the number of simulated
matches cval is a constant
        return (reward / nplayout) + cval * math.sqrt(2 * math.log(t) /
nplayout)
```

Calculate the number of players chess pieces and computer chess pieces, computer chess pieces will return true. Then check whether you can play chess in this position : if you go to a place where there is no place to play (that is, the play is in an illegal state at this time), first judge whether the computer can still play, if not, then change to the player to continue to play, if the computer can continue to play, the computer continues to play, down to both sides can not play, evaluate the number of chess pieces, one of the two sides can continue to play, continue to cycle judgment. Randomly place a chess piece, switch rounds, and record the depth because the search depth is set to prevent over-search.

```
def find_playout(tep_board, tile, depth=0):
        def eval_board(tep_board):
            player_tile = countTile(tep_board, PLAYER_NUM)
            computer_tile = countTile(tep_board, COMPUTER_NUM)
            if computer_tile > player_tile:
                return True
            return False
        if depth > 32:
            return eval_board(tep_board)
        turn_positions = possible_positions(tep_board, tile)

        # See if you can play chess in this position
        if len(turn_positions) == 0:
            if tile == COMPUTER_NUM:
                neg_turn = PLAYER_NUM
            else:
                neg_turn = COMPUTER_NUM

            neg_turn_positions = possible_positions(tep_board, neg_turn)

            if len(neg_turn_positions) == 0:
                return eval_board(tep_board)
            else:
                tile = neg_turn
                turn_positions = neg_turn_positions

        # Place a piece randomly
        temp = turn_positions[random.randrange(0, len(turn_positions))]
        updateBoard(tep_board, tile, temp[0], temp[1])

        # Conversion rounds
        if tile == COMPUTER_NUM:
            tile = PLAYER_NUM
        else:
```

```
            tile = COMPUTER_NUM

        return find_playout(tep_board, tile, depth=depth + 1)
```

Search for the best path. The four parameters of parent, t _ playout, reward, t _ childrens are assigned to each child node, and these four parameters are used to calculate the UCB value. Realize the maximum and minimum search, the computer selects the maximum value, and the player selects the minimum value. The computer searches for the largest UCB value for itself and stores the largest UCB value for the current child node layer, giving the player the smallest UCB value. Randomly play chess, expand.

```
def find_path(root, total_playout):
        current_path = []
        child = root
        parent_playout = total_playout
        isMCTSTurn = True

        while True:
            if len(child) == 0:
                break
            maxidxlist = [0]
            cidx = 0
            if isMCTSTurn:
                maxval = -1
            else:
                maxval = 2

            for n_tuple in child:
                parent, t_playout, reward, t_childrens = n_tuple

                #Maximize minimum search, computer selects maximum, player selects minimum
                if isMCTSTurn:
                    cval = ucb1(n_tuple, parent_playout, 0.1)

                    if cval >= maxval:
                        if cval == maxval:
                            maxidxlist.append(cidx)
                        else:
                            maxidxlist = [cidx]
                            maxval = cval
                else:
                    cval = ucb1(n_tuple, parent_playout, -0.1)

                    if cval <= maxval:
                        if cval == maxval:
                            maxidxlist.append(cidx)
                        else:
                            maxidxlist = [cidx]
                            maxval = cval

                cidx += 1

            # Randomly play chess, expand
            maxidx = maxidxlist[random.randrange(0, len(maxidxlist))]
            parent, t_playout, reward, t_childrens = child[maxidx]
```

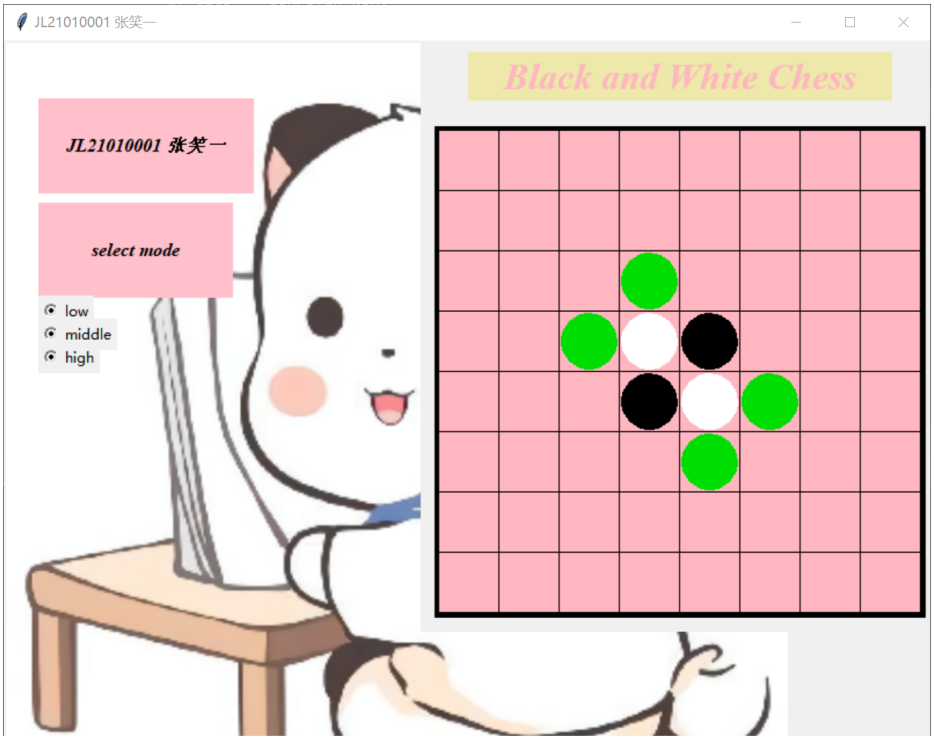
```
            current_path.append(parent)
            parent_playout = t_playout
            child = t_childrens
            isMCTSTurn = not (isMCTSTurn)

        return current_path

```

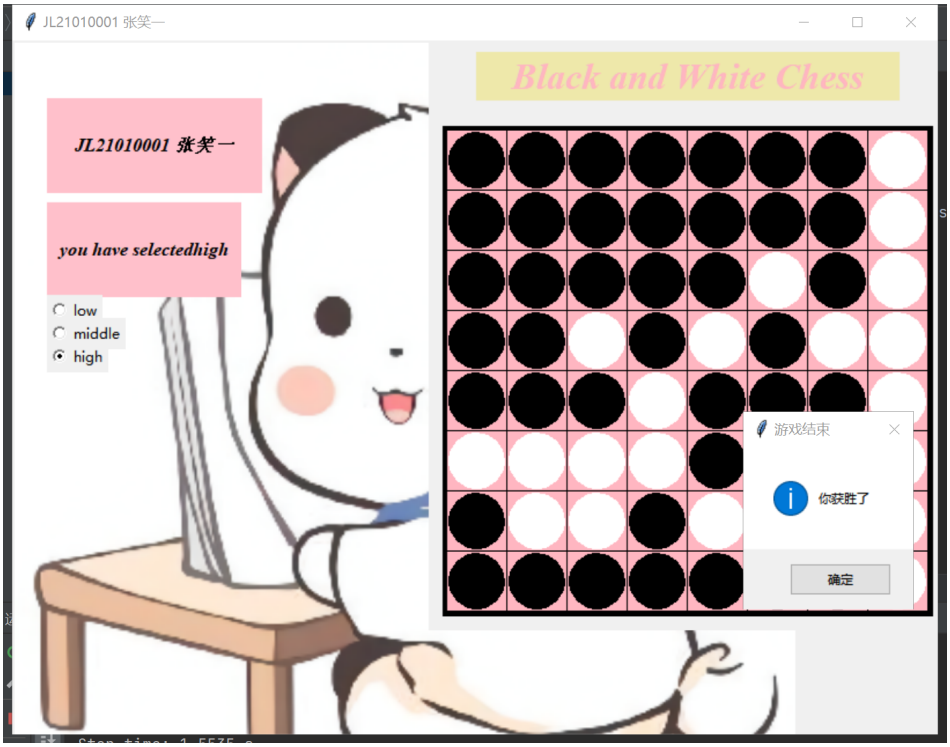
Debugging Analysis and Test Results

Running the code, the following occurs :



On the left side of the figure is ' select mode ', select the desired difficulty, you can start the game. Black represents the player, white defaults to the computer, and green refers to the legal place to go.

Here is the final chess set, if you win will show that you won, otherwise for the computer to win.


JL21010001 张笑一
Black and White Chess
JL21010001 张笑一
you have selectedhigh
low
middle
high
游戏结束
你获胜了
确定