به نام خدا

الگوریتم mcts:

در کدم از دو نوع از این الگوریتم استفاده کردم یکی ورژن صفر و یکی ورژن ucb که با اولی همه تست ها پاس شد و با دومی تست‌های اول و چهارم پاس نشد.

کلاس GameState :

```
class GameState:
    def __init__(self, board, player):
        self.board = board
        self.player = player
        self.val = 0
        self.vis = 0
        self.parent = None
        self.children = []
    def __hash__(self):
        return hash(str(self.board))
    def __eq__(self, other):
        if other is None:
            return False
        for i in range(3):
            for j in range(3):
                if self.board[i][j] != other.board[i][j]:
                    return False
        return True

    def get_uct(self):
        if self.vis == 0:
            return float('inf')
        return self.val/self.vis + C *
(math.log(self.parent.vis)/self.vis)**0.5
```

مقادیر ذخیره شده:

Board که همان استیت ماست.

Player یعنی اینکه آخرین مهره را چه کسی گذاشته که وارد این وضعیت شدیم.

Val که همان utility است. وقتی ببرد یکی زیاد، ببازد یکی کم و مساوی شود هیچ اتفاقی نمیفتد و فقط مقدار vis زیاد میشود.

مقدار vis که همان تعداد آزمایشات است.

Parent و children هم که والد و فرزندان آن استیت هستند.

تابع get_uct که مقدار ucb را برای آن استیت محاسبه میکند.

تابع findBestMove(board): (این تابع از ورژن صفر mcts استفاده میکند)

```
root_state = GameState(board, 'X')
    gameStateList = []
    makeChildren(gameStateList, root_state)
    num_of_experiments =
num_of_iterations//len(root_state.children)
    for child in root_state.children:
        for i in range(num_of_experiments):
            simulate(child)
    next_move = get_best2(root_state.children)
    return get_difference(next_move, root_state)
```

ابتدا از استیت اولیه یک آبجکت GameState درست می‌کنیم، سپس فرزندان این استیت اولیه را با استفاده از تابع makeChildren می‌سازیم، سپس تعداد آزمایش‌هایی که برای هرکدام از فرزندها باید انجام دهیم را بدست می‌آوریم که در ورژن صفر به همه فرزندان تعداد مساوی آزمایش می‌رسد. سپس برای هر کدام از فرزندان به تعداد num_of_experiments تابع simulate را صدا میزنیم که این تابع یک بار با شروع از آن فرزند آزمایش انجام میدهد.

در آخر تابع get_best2 رو صدا میزنیم که بین این فرزندان، آن فرزندی که بیشترین آزمایش موفق را دارد برگردانده میشود.

تابع makeChildren :

```
ch_indexes = get_empty_house(root_state.board)
    for idx in ch_indexes:
        childBoard = copy.deepcopy(root_state.board)
        turn = 'O' if root_state.player=='X' else 'X'
        childBoard[idx//3][idx % 3] = turn
        gs = GameState(childBoard, turn)
        gs.parent = root_state
        gameStateList.append(gs)
        root_state.children.append(gs)
```

در این تابع ابتدا خانه‌های خالی را به کمک تابع get_empty_house بدست می‌آوریم تا استیت های فرزند را بوجود آوریم، سپس همه استیت های فرزند را بوجود آورده و در children مربوط به root_state اضافه میکنیم.

تابع simulate :

```
currentState = gameState
    while(isMovesLeft(currentState.board)):
        if(checkWin(currentState.board)):
            winner = checkWhoWin(currentState.board)
            backpropagate(gameState, winner)
            return
        else:
            turn = 'O' if currentState.player == 'X' else 'X'
            empty_spots = get_empty_house(currentState.board)
            idx = random.choice(empty_spots)
            nextBoard = copy.deepcopy(currentState.board)
            nextBoard[idx // 3][idx % 3] = turn
            currentState = GameState(nextBoard, turn)
    backpropagate(gameState, 'T')
```

در این تابع با شروع از gameState که حالتی است که بازی را از آنجا میخواهیم شروع کنیم، ابتدا به کمک تابع isMovesLeft چک میکنیم که بازی تمام نشده باشد، سپس چک میکنیم که اگر بازی برنده داشته ، بازی تمام شود و در غیر اینصورت به بازی ادامه دهد بدین صورت که نوبت عوض شد و شخص دیگر یک خانه را به صورت تصادفی انتخاب کند، وقتی بازی تمام شد ، تابع backpropagate را صدا میزنیم تا مقادیر val و vis عوض شوند.

تابع get_best2 :

این تابع از بین این فرزندان، آن فرزندی که بیشترین مقدار val را داراست ، انتخاب میکند.

```
def get_best2(gameStateList):
    return max(gameStateList, key=lambda x: x.val)
```

تابع backpropagate :

```
def backpropagate(gameState, winner):
    while gameState != None:
        gameState.vis += 1
        if winner == 'O':
            gameState.val += 1
        elif winner == 'X':
            gameState.val -= 1
        gameState = gameState.parent
```

این تابع از استیتی که از آن شروع به بازی کردیم تا root_state عقب گرد میرود و مقادیر val و vis را آپدیت میکند.

تابع findBestMove2 : (بر اساس ucb کار می‌کند)

```
def findBestMove2(board):
    root_state = GameState(board, 'X')
    gameStateList = []
    makeChildren(gameStateList, root_state)
    for i in range(num_of_iterations):
        best = selection_process(gameStateList, root_state)
        simulate(best)
    next_move = get_best_uct_based(root_state)
    return get_difference(next_move, root_state)
```

در این تابع ابتدا از استیت اولیه یک آبجکت استیت می‌سازیم و سپس به کمک تابع makeChildren فرزندان این استیت اولیه را در root_state.children و gameStateList ذخیره میکنیم. سپس به تعداد num_of_iterations که همان تعداد کل آزمایشات است عملیات selection_process اجرا میکنیم تا استیتی که میخواهیم آزمایش کنیم را بدست آوریم، سپس به کمک تابع simulate یک آزمایش انجام میدهیم. در آخر بین فرزندان root_state بهترین را به عنوان اکشن بعدی برمی‌گردانیم، به کمک تابع get_difference تفاوت بین استیت بعدی و استیت قبلی را پیدا میکنیم و مختصات آن خانه را برمیگردانیم.

تابع selection_process :

```
def selection_process(gameStateList, root_state):
    if len(root_state.children) == 0:
        makeChildren(gameStateList, root_state)
    if len(root_state.children) == 0:
        return root_state
    best = get_best_uct_based(root_state)
    if best.vis == 0:
        return best
    return selection_process(gameStateList, best)
```

در این تابع از root_state شروع میکنیم و بهترین را از بین فرزندان آن انتخاب میکنیم، ابتدا اگر root_state فرزندانش ساخته نشده بودند، سعی میکنیم آن ها را بسازیم ، اگر بعد از این کار باز هم فرزندی نداشت یعنی این یک استیت پایانی است، پس همان را برمیگردانیم، در غیر اینصورت تابع get_best_uct_based را صدا زده و بین فرزندان بهترین را انتخاب میکند، اگر یکی از این فرزندان مقدار vis صفر داشته باشد، مقدار ucb بی نهایت دارد و قطعا انتخاب میشود و همان برگردانده میشود. در غیر اینصورت عملیات selection_process را به صورت بازگشتی این بار در سطح بعدی اجرا میکنیم.

تابع get_best_uct_based :

```
def get_best_uct_based(gameState):
    return max(gameState.children, key=lambda x: x.get_uct())
```

از بین فرزندان استیت ورودی ، آن را که بیشترین مقدار ucb را دارد، انتخاب میکنیم.

برخی توابع کمکی دیگر:

تابع checkWhoWin :

```
def checkWhoWin(board):
    for row in range(3):
        if (board[row][0] == board[row][1] and board[row][1] ==
board[row][2] and not board[row][0] == '_'):
            return board[row][0]
    for col in range(3):
        if (board[0][col] == board[1][col] and board[1][col] ==
board[2][col] and not board[0][col] == '_'):
            return board[0][col]
```

```
    if (board[0][0] == board[1][1] and board[1][1] ==
board[2][2] and not board[0][0] == '_'):
        return board[0][0]

    if (board[0][2] == board[1][1] and board[1][1] ==
board[2][0] and not board[0][2] == '_'):
        return board[0][2]

    return False
```

در صورتی که یکی برنده شده بود، بگوید که کدام بازیکن برنده شده است.

تابع get_empty_house :

```
def get_empty_house(board):
    empty_spots = [i*3+j for i in range(3)
                   for j in range(3) if board[i][j] == "_"]
    return empty_spots
```

خانه هایی را که خالی هستند را برمیگرداند که این در پیدا کردن فرزندان یک استیت کمک میکند.