程序报告

学号：2212046　　姓名：王昱

一、问题重述

黑白棋问题：

黑白棋（Reversi），也叫苹果棋，翻转棋，是一个经典的策略性游戏，一般棋子双面为黑白两色，故称“黑白棋”。因为行棋之时将对方棋子翻转，则变为己方棋子，故又称“翻转棋”。具体规则不详细赘述。

实验要求：

- 使用 「蒙特卡洛树搜索算法」实现 miniAlphaGo for Reversi
- 使用 Python 语言
- 算法部分需要自己实现，不要使用现成的包、工具或者接口

对问题的理解：

题目要求使用蒙特卡洛树搜索算法实现黑白棋，具体棋盘类(Board)以及游戏类(Game)实验平台已经给出。实验要求的是自己编写程序来实现一个 AIplayer，本质上是通过多次迭代，用蒙特卡洛树搜索算法来寻找 AIplayer 对下一步的最优解。

二、设计思想

2.1 蒙特卡罗树搜索算法：

2.1.1 概念： 蒙特卡洛树搜索是一种基于概率的启发式搜索算法，通过模拟随机选择走子步骤以评估当前节点的潜在价值，从而辅助决策制定。同时，其也是一种强化学习算法，被广泛应用于复杂决策问题的求解中。它通过随机模拟和搜索树的建立来寻找最优策略。

2.1.2 算法流程：

一共有四个阶段，分别是选择、扩展、模拟和反向传播。

- 选择：在选择阶段，根据 UCB（Upper Confidence Bound）算法选择当前节点的子节点，以权衡探索和利用
- 扩展：如果叶子节点不是终止节点，那么就扩展这个节点，生成它的子节点
- 模拟：对子节点进行模拟，直到达到终止节点
- 反向传播：根据模拟的结果，更新当前节点以及它的所有祖先节点的统计信息

==========

2.2 算法设计

2.2.1 实现节点（Node）类

对于节点类，每个节点维护了两个值，分别是节点的访问次数和节点的价值。访问次数表示这个节点被访问的次数，价值表示这个节点的平均收益。在选择子节点的时候，通常会使用 UCB 公式来进行选择，公式如下：

$$UCB = \frac{Q}{N} + C \times \sqrt{\frac{\ln P}{N}}$$

Node 类用于表示每一个蒙特卡洛树的节点，其中定义如下属性：

```
def __init__(self, now_board, parent=None, action=None, color=""):
        self.visits = 0  # 访问次数
        self.reward = 0.0  # 期望值
        self.now_board = now_board  # 棋盘状态
        self.children = []  # 孩子节点
        self.parent = parent  # 父节点
        self.action = action  # 对应动作
        self.color = color# 该节点玩家颜色
```

（各个属性的意义已经在注释中给出）

定义如下函数：

获取当前节点的 ucb 值，其中未访问的节点初始化未无穷大

```
    def get_ucb(self, ucb_param):
```

向树中添加子节点，因为我们需要不断扩展树

```
    def add_child(self, child_now_board, action, color):
```

判断节点是否完全扩展

```
    def full_expanded(self):
```

==========

2.22 实现 AIplayer

第一步：初始化智能体，定义一些需要用到的函数

初始化最大迭代次数，参数 C 以及 AI 方的颜色

```
    def __init__(self, color):
        self.max_times = 150  # 最大迭代次数
        self.ucb_param = 1  # ucb 的参数 C
        self.color = color
```

判断游戏是否结束，通过判断双方合法落子的个数是否为 0，若都为 0 则说明游戏已经结束

```
    def game_over(self, board):
```

根据棋盘当前状态获取最佳落子位置

```
    def get_move(self, board):
```

第二步：分别实现选择、扩展、模拟和反向传播四个阶段

1.选择

```
    def select(self, node):
```

从该节点向下选择以便于扩展的节点，策略为 UCB，一共有三种情况：

（1）如果是叶子节点，即还没有扩展过的节点，那么直接返回进行扩展。

（2）如果是节点已经完全扩展，那么递归选择一个 UCB 值最大的节点进行扩展。

（2）如果是节点还没有完全扩展，那么优先选择一个访问次数为 0 的孩子，从左开始遍历。

2.扩展

```
    def extend(self, node):
```

扩展函数实现对没有完全扩展和为扩展的节点进行扩展，在树中添加新的节点。

该函数的步骤大概如下：

（1）检查节点是否被访问过：如果传入的节点 node 是新的或者尚未被模拟过，就不需要扩展，直接返回该节点以进行模拟。

（2）确定扩展的颜色：如果节点已经被访问过，需要扩展新的子节点。首先确定新节点的颜色，如果当前节点的颜色是‘X’（黑棋），那么新节点的颜色应该是‘O’（白棋），反之

亦然。
（3）获取合法动作并扩展子节点。
（4）返回第一个子节点：如果当前节点成功添加了子节点，函数返回子节点列表中的第一个节点。如果没有可行的动作（即子节点列表为空），则返回当前节点。
3.模拟
def stimulate(self, node):
（1）模拟起始：函数接收一个节点 node 作为模拟的起始点。使用 deepcopy 来创建当前棋盘状态的副本，以便在模拟过程中不影响原始棋盘。
（2）获取当前颜色 color 的所有合法动作：
如果有合法动作，随机选择一个执行，并更新棋盘。
如果当前颜色没有合法动作，切换到对手颜色，并尝试获取对手的合法动作。
（3）评估模拟结果：使用 board.get_winner() 方法来评估模拟结束时的胜负关系和获胜方的领先棋子数。
（4）评分策略：
考虑胜负关系和获胜的子数，定义获胜积 10 分，每多赢一个棋子多 1 分

- 如果是平局（winner 为 2），则奖励（reward）为 0 分。
- 如果黑棋赢（winner 为 0），则奖励为基础的 10 分加上领先的棋子数 diff。
- 如果白棋赢（winner 为 1），则奖励为基础的 10 分加上领先的棋子数 diff 的负值。

（5）返回奖励：函数返回计算出的奖励值 reward，这个值将用于后续的反向传播步骤，以更新搜索树中节点的统计信息。
4.反向传播
def backup(self, node, reward)：
通过循环从当前节点反向传播更新父节点。
==========
第三步：实现搜索核心框架
def uct(self, max_times, root):
迭代 max_times 次，经过多次测试，我选择了 150 次。
这部分首先进行 150 次模拟，在所有模拟完成后，代码会遍历根节点的所有子节点，找到具有最大 UCB 值的节点作为最佳的下一步落子位置，并返回该节点的 action。这样我们就可以得出该算法下最优的下一步。

三、代码内容（main.py）

```
import math
import random
import sys
from copy import deepcopy
from game import Game
class Node:
    def __init__(self, now_board, parent=None, action=None, color=""):
        self.visits = 0    # 访问次数
        self.reward = 0.0    # 期望值
        self.now_board = now_board    # 棋盘状态
        self.children = []    # 孩子节点
        self.parent = parent    # 父节点
        self.action = action    # 对应动作
```

```
            self.color = color    # 该节点玩家颜色
        def get_ucb(self, ucb_param):
            if self.visits == 0:
                return sys.maxsize    # 未访问的节点 ucb 为无穷大
            # UCB 公式
            explore = math.sqrt(2.0 * math.log(self.parent.visits) / float(self.visits))
            now_ucb = self.reward / self.visits + ucb_param * explore
            return now_ucb
        def add_child(self, child_now_board, action, color):
            child_node = Node(child_now_board, parent=self, action=action, color=color)
            self.children.append(child_node)
        # 判断是否完全扩展
        def full_expanded(self):
            if len(self.children) == 0:
                return False
            for kid in self.children:
                if kid.visits == 0:
                    return False
            return True
class AIPlayer:
    """
    AI 玩家
    """

    def __init__(self, color):
        """
        玩家初始化
        :param color: 下棋方，'X' - 黑棋，'O' - 白棋
        """
        self.max_times = 150    # 最大迭代次数
        self.ucb_param = 1    # ucb 的参数 C

        self.color = color
    def uct(self, max_times, root):
        for i in range(max_times):    # 最多模拟 max 次
            selected_node = self.select(root)
            leaf_node = self.extend(selected_node)
            reward = self.stimulate(leaf_node)
            self.backup(leaf_node, reward)

        max_node = None    # 搜索完成，然后找出最适合的下一步
        max_ucb = -sys.maxsize
        for child in root.children:
            child_ucb = child.get_ucb(self.ucb_param)
```

```
            if max_ucb < child_ucb:
                max_ucb = child_ucb
                max_node = child    # max_node 指向 ucb 最大的孩子
        return max_node.action
    def select(self, node):
        # print(len(node.children))
        if len(node.children) == 0:    # 叶子，需要扩展
            return node
        if node.full_expanded():    # 完全扩展,递归选择 ucb 最大的孩子
            max_node = None
            max_ucb = -sys.maxsize
            for child in node.children:
                child_ucb = child.get_ucb(self.ucb_param)
                if max_ucb < child_ucb:
                    max_ucb = child_ucb
                    max_node = child    # max_node 指向 ucb 最大的孩子
            return self.select(max_node)
        else:
            for kid in node.children:    # 从左开始遍历
                if kid.visits == 0:
                    return kid
    def extend(self, node):
        if node.visits == 0:    # 自身还没有被访问过，不扩展，直接模拟
            return node
        else:
            if node.color == 'X':
                new_color = 'O'
            else:
                new_color = 'X'
            for action in list(node.now_board.get_legal_actions(node.color)):
                new_board = deepcopy(node.now_board)
                new_board._move(action, node.color)
                node.add_child(new_board, action=action, color=new_color)
            if len(node.children) == 0:
                return node
            return node.children[0]
    def stimulate(self, node):
        board = deepcopy(node.now_board)
        color = node.color
        count = 0
        while (not self.game_over(board)) and count < 50:    # 游戏没有结束，就模拟下棋
            action_list = list(node.now_board.get_legal_actions(color))
            if not len(action_list) == 0:    # 可以下，就随机下棋
                action = random.choice(action_list)
```

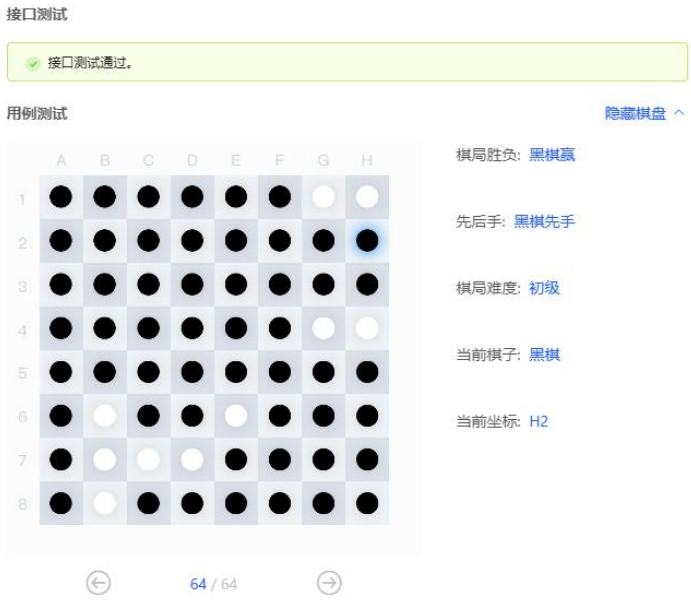
```
                    board._move(action, color)
                    if color == 'X':
                        color = 'O'
                    else:
                        color = 'X'
                else:
                    if color == 'X':
                        color = 'O'
                    else:
                        color = 'X'
                    action_list = list(node.now_board.get_legal_actions(color))
                    action = random.choice(action_list)
                    board._move(action, color)
                    if color == 'X':
                        color = 'O'
                    else:
                        color = 'X'
                count = count + 1
        winner, diff = board.get_winner()
        if winner == 2:
            reward = 0
        elif winner == 0:
            reward = 10 + diff
        else:
            reward = -(10 + diff)
        if self.color == 'X':
            reward = - reward
        return reward
    def backup(self, node, reward):
        while node is not None:
            node.visits += 1
            if node.color == self.color:
                node.reward += reward
            else:
                node.reward -= reward
            node = node.parent
        return 0
    def game_over(self, board):
        b_list = list(board.get_legal_actions('X'))
        w_list = list(board.get_legal_actions('O'))
        is_over = (len(b_list) == 0 and len(w_list) == 0)    # 返回值  True/False
        return is_over
    def get_move(self, board):
        if self.color == 'X':
```

```
                player_name = '黑棋'
            else:
                player_name = '白棋'
            print("请等一会，对方 {}-{} 正在思考中...".format(player_name, self.color))

            root = Node(now_board=deepcopy(board), color=self.color)

            action = self.uct(self.max_times, root)

            return action
# 导入黑白棋文件
from game import Game

# 人类玩家黑棋初始化
black_player =  HumanPlayer("X")

# AI 玩家 白棋初始化
white_player = AIPlayer("O")

# 游戏初始化，第一个玩家是黑棋，第二个玩家是白棋
game = Game(black_player, white_player)

# 开始下棋
game.run()
```

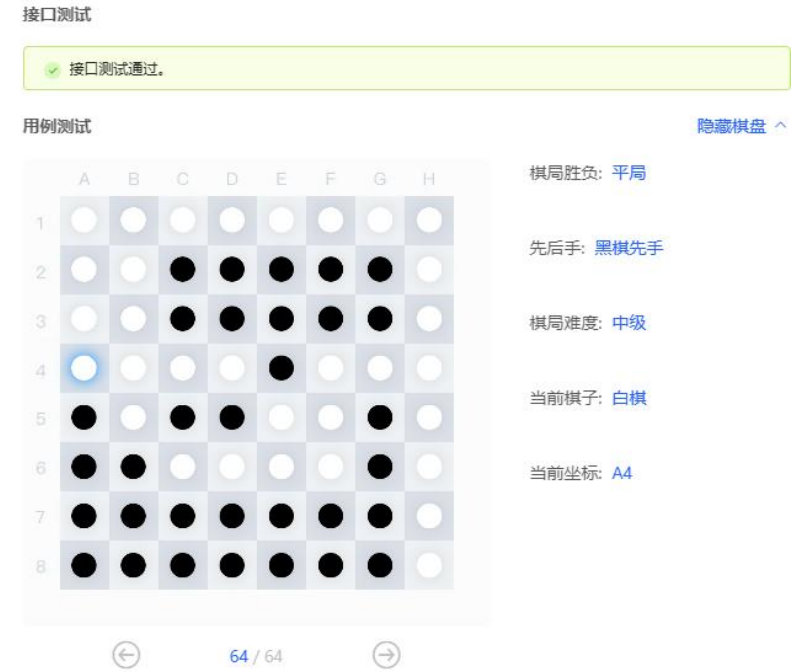
四、实验结果

以下是代码分别对弈初级，中级，高级 AI 的结果：

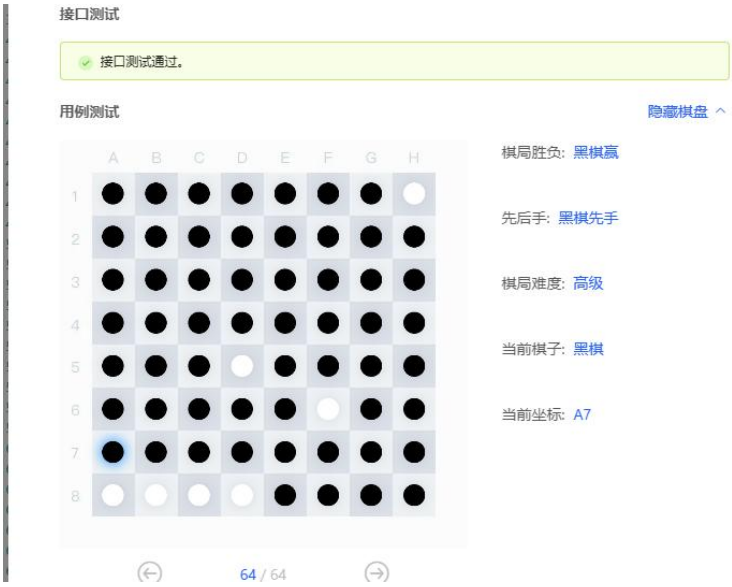
初级：



中级：



高级：



五、实验感想与总结

1.通过这次实验，我对蒙特卡洛树算法在棋类游戏中的应用有了更深入的理解。实验过程中，我深刻体会到算法优化的重要性。每一次模拟都是对策略的一次考验，而算法的调整则直接影响到模拟的效果。

2.最大的挑战是如何平衡探索和利用。过多的探索可能会导致我们忽视了已知的有效策略，而过度的利用又可能使我们错过新的、更优的策略。在实验中不断调整参数，寻找最佳平衡点，需要进行深入的思考。在本题中，由于黑白棋子的个数也是影响结果的重要因素，所以把棋子的个数也纳入得分标准。

3.这次实验是一次宝贵的学习经历，它不仅增强了我的编程技能，还激发了我对人工智能领域的热情。