人工智能导论实验报告

学号：2313211 姓名：王众

一、待解决的问题

> 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” (Reversi) 。
> 棋子双面为红、绿色的称为“苹果棋”。它使用 8x8 的棋盘，由两人执黑子和白子轮流下棋，最后子多方为胜方。
> 随着网络的普及，黑白棋作为一种最适合在网上玩的棋类游戏正在逐渐流行起来。
> 中国主要的黑白棋游戏站点有 Yahoo 游戏、中国游戏网、联众游戏等。
>
> 游戏规则：
>
> 1.黑方先行，双方交替下棋。
>
> 2.一步合法的棋步包括：
>
> - 在一个空格新落下一个棋子，并且翻转对手一个或多个棋子；
> - 新落下的棋子必须落在可夹住对方棋子的位置上，对方被夹住的所有棋子都要翻转过来，可以是横着夹，竖着夹，或是斜着夹。
> - 夹住的位置上必须全部是对手的棋子，不能有空格；
> - 一步棋可以在数个（横向，纵向，对角线）方向上翻棋，任何被夹住的棋子都必须被翻转过来，棋手无权选择不去翻某个棋子。
>
> 3.如果一方没有合法棋步，也就是说不管他下到哪里，都不能至少翻转对手的一个棋子，那他这一轮只能弃权，而由他的对手继续落子直到他有合法棋步可下。
>
> 4.如果一方至少有一步合法棋步可下，他就必须落子，不得弃权。
>
> 5.棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。
>
> 6.如果某一方落子时间超过 1 分钟 或者 连续落子 3 次不合法，则判该方失败。
>
> 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。

二、算法设计

2.1蒙特卡洛树算法的原理

蒙特卡洛树搜索（简称 MCTS）是 Rémi Coulom 在 2006 年在它的围棋人机对战引擎 「Crazy Stone」中首次发明并使用的的 ，并且取得了很好的效果。在黑白棋中主要有以下四个重要的步骤：分别是选择、扩展、模拟、反向传播。

- 选择： 从根节点 R 开始，递归选择子节点，直至到达叶节点或到达具有还未被扩展过的子节点的节点L。具体来说，通常用UCB1(Upper Confidence Bound，上限置信区间)选择最具“潜力”的后续节点

- 扩展：如果 L 不是一个终止节点，则随机创建其后的一个未被访问节点，选择该节点作为后续子节点C。
- 模拟：从节点 C出发，对游戏进行模拟，直到博弈游戏结束。
- 反向传播：用模拟所得结果来回溯更新导致这个结果的每个节点中获胜次数和访问次数。

2.2 代码实现

2.2.1 节点的初始化定义

首先，要建立蒙特卡洛树就需要先建立树的节点，所以我们需要先定义`Node`类。

```
class Node:
    #蒙特卡洛树搜索中的节点
    def __init__(self, board, parent, action, color, is_terminal=False):
        #节点初始化
        self.board = board #:param board: 棋盘状态
        self.parent = parent#:param parent: 父节点
        self.action = action# :param action: 到达此节点的动作
        self.color = color  # 表示轮到哪种颜色下棋
        self.children = []
        self.visits = 0  # 访问次数
        self.value = 0.0  # 节点价值
        self.is_fully_expanded = False#当所有合法动作都已创建子节点时，将节点标记为已完全扩展
        self.is_terminal = is_terminal#:param is_terminal: 是否为终端节点
```

我们在`Node`节点中首先需要定义棋盘当前的状态（一个记录位置的二维数组）。然后是关于节点本身的父节点的记录，因为不是二叉树所以需要有孩子的记录、处于那一种颜色的棋的那一层，访问次数与每个节点的价值，并记录是否为叶节点。

2.2.2 AI玩家类的初始化

```
def __init__(self, color):
    self.color = color#当前AI棋子的颜色
    self.opponent_color = 'O' if color == 'X' else 'X'#对手的颜色
    self.time_limit = 5  #最大的循环时间定义为5秒
    self.C = 1.0 #UCT参数设置为1即可完成任务无需动态调整
    self.min_visits = 100  #增加早停机制，根节点访问100次即可提前返回提高效率
    self.board_cache = {}
```

关于`board_cache`的定义：在蒙特卡洛树搜索(MCTS)的模拟阶段，同一个棋盘状态可能会被多次评估。通过缓存已计算过的状态，避免重复计算，节省时间当算法在不同路径上遇到相同的棋盘状态时，可以直接从缓存中获取评估结果，而不需要再次进行随机模拟。进一步提高效率。

2.2.3 选择

```
def select(self, node):
    while not node.is_terminal:
        if not node.is_fully_expanded:
            return node
        node = self.get_best_child(node, self.C)
        return node
```

如果不是叶节点，并且在这个行为之后还能找到未被找过的下一个节点，那么就计算他们之中UCB的最大值。如果为叶节点或者没有另外的下一步了，则直接返回当前节点。

```
def get_best_child(self, node, exploration_weight):
    def ucb_score(child):
        # 避免除以零
        if child.visits == 0:
            return float('inf')
        # UCB公式
        exploitation = child.value / child.visits
        exploration = exploration_weight * math.sqrt(2 * math.log(node.visits) /
child.visits)
        return exploitation + exploration
    # 如果当前节点颜色与AI相同，选择最大UCB值；否则选择最小UCB值
    if node.color == self.color:
        return max(node.children, key=ucb_score)
    else:
        return min(node.children, key=lambda child: -ucb_score(child))
```

关于`get_best_child`函数，首先需要保证计算公式中的分母不能为0，然后我们使用UCB公式计算得分并比较

如果是我方取最大的节点，对方的取最小的节点。

2.2.4 扩展

在扩展阶段我们首先需要获取下一步可能的所有行为，然后将已经探查过的节点去掉。如果所有节点均已经被探查，那么将该节点标记为已完全扩展。如果并不是完全扩展的节点，那么使用随机数随机选择一个未被探查过的节点，并更新棋盘的状态。下一步检查对方现在是否有能走的节点，如果没有那么再检查自己还不能走，不能走则游戏结束。能走则创建节点并继续下棋。创建节点后检查如果新节点所有节点已经被探查完了则标记为完全扩展。最后返回新创建的节点。

```
def expand(self, node):
    # 获取所有可能的动作
    tried_actions = [child.action for child in node.children]
    legal_actions = list(node.board.get_legal_actions(node.color))
    # 过滤掉已经尝试过的动作
    untried_actions = [action for action in legal_actions if action not in
tried_actions]
    if not untried_actions:
        node.is_fully_expanded = True
        return self.get_best_child(node, self.c)
    # 随机选择一个未尝试的动作
    action = random.choice(untried_actions)
    # 创建新的棋盘状态
    new_board = deepcopy(node.board)
    new_board._move(action, node.color)
    # 下一个玩家的颜色
    next_color = 'O' if node.color == 'X' else 'X'
    # 检查下一个玩家是否有合法动作
    if not list(new_board.get_legal_actions(next_color)):
        # 如果下一个玩家没有合法动作，可能需要跳过该玩家的回合
        # 检查当前玩家是否还有其他合法动作
        if not list(new_board.get_legal_actions(node.color)):
```

```
            # 游戏结束
            is_terminal = True
        else:
            # 当前玩家继续
            next_color = node.color
            is_terminal = False
        else:
            is_terminal = False
            # 创建并返回新节点
            child = Node(
                board=new_board,
                parent=node,
                action=action,
                color=next_color,
                is_terminal=is_terminal
            )
            node.children.append(child)
            # 如果所有可能的动作都已尝试，将节点标记为完全扩展
            if len(node.children) == len(legal_actions):
                node.is_fully_expanded = True

                return child
```

2.2.5 模拟

模拟的主要原则是：优先选择角落位置，没有角落位置则会随机选择位置并且避开靠近角落的危险位置。

```
def simulate(self, node):
    # 使用棋盘状态的哈希值作为缓存键
    board_state = str(node.board._board)
    if board_state in self.board_cache:
        return self.board_cache[board_state]
    # 创建一个临时棋盘副本
    board_copy = deepcopy(node.board)
    current_color = node.color
    # 减少最大模拟深度，加速模拟过程
    max_depth = 20
    # 快速模拟算法 - 使用简单的随机策略
    depth = 0
    while depth < max_depth:
        # 检查是否有合法动作
        legal_actions = list(board_copy.get_legal_actions(current_color))
        if not legal_actions:
            # 切换玩家
            current_color = 'O' if current_color == 'X' else 'X'
            legal_actions = list(board_copy.get_legal_actions(current_color))
            if not legal_actions:
                # 游戏结束
                break
                # 优先选择角落位置，提高模拟质量
                corner_positions = ['A1', 'A8', 'H1', 'H8']
                action = None
                for corner in corner_positions:
                    if corner in legal_actions:
```

```
                        action = corner
                        break
                        # 如果没有角落位置，随机选择
                        if not action:
                            action = random.choice(legal_actions)
                            board_copy._move(action, current_color)
                            # 切换玩家
                            current_color = 'O' if current_color == 'X' else 'X'
                            depth += 1
                            # 计算模拟结果
                            winner, diff = board_copy.get_winner()
                            # 确定奖励值
                            if winner == 2:  # 平局
                                reward = 0.5
                            elif (winner == 0 and self.color == 'X') or (winner
== 1 and self.color == 'O'):
                                # AI赢了
                                reward = 1.0
                            else:
                                # AI输了
                                reward = 0.0
                                # 缓存结果
                                self.board_cache[board_state] = reward
                                return reward
```

2.2.6 反向传播

如果颜色相同，则使用奖励值，如果不同则使用1-奖励值

```
def backpropagate(self, node, reward):
    while node:
        node.visits += 1
        # 如果当前节点的父节点的颜色与AI相同，则使用奖励值；否则使用(1-奖励值)
        if node.parent and node.parent.color == self.color:
            node.value += reward
        else:
            node.value += 1 - reward
            node = node.parent
```

2.2.7 蒙特卡洛树的封装综合实现

在此函数中进行循环，依次进行选择、扩展、模拟和反向传播

```
def monte_carlo_tree_search(self, root):
    start_time = time.time()
    iterations = 0
    # 在时间限制内尽可能多地进行搜索迭代
    while time.time() - start_time < self.time_limit:
        # 添加早停条件
        if root.visits > self.min_visits and iterations > 100:
            break
            # 选择
            leaf = self.select(root)
            # 扩展
            if not leaf.is_terminal and not leaf.is_fully_expanded:
```

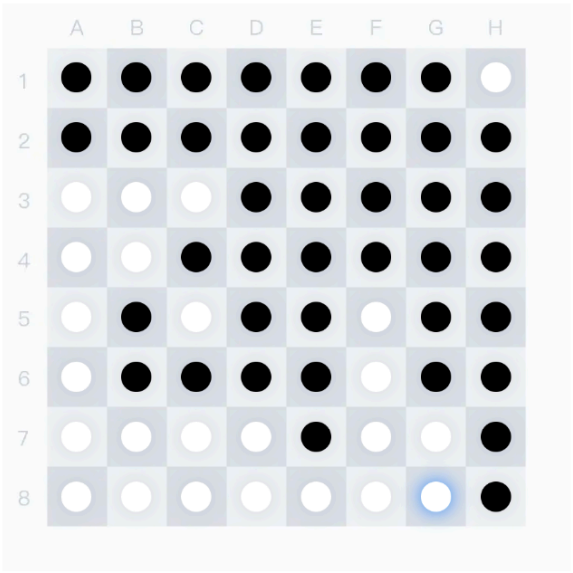
```
                child = self.expand(leaf)
                # 模拟
                reward = self.simulate(child)
                # 反向传播
                self.backpropagate(child, reward)
            else:
                # 如果叶子节点是终端节点或已完全扩展，直接进行模拟
                reward = self.simulate(leaf)
                # 反向传播
                self.backpropagate(leaf, reward)
                iterations += 1
                # 选择访问次数最多的子节点
                best_child = self.get_best_child(root, 0)  # 设置探索参数为0，纯粹基于访问次数
                return best_child.action
```

2.3 平台测试

我们依次选择初级中级高级三个难度进行测试:

初级：黑棋先手（领先16子）

测试详情 隐藏棋盘



棋局胜负：黑棋赢

先后手：黑棋先手

棋局难度：初级

当前棋子：白棋

当前坐标：G8

64 / 64

确定

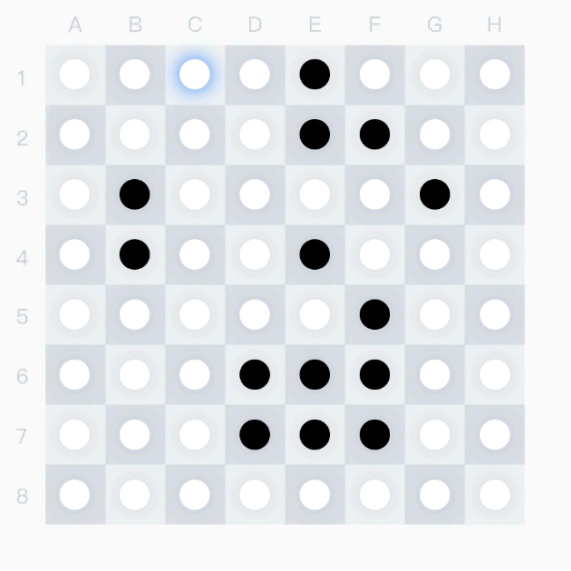
中级：（白棋后手，领先36子）

测试详情 展示棋盘

测试点	状态	时长	结果
对手对弈	✓	39s	白棋获胜, 领先棋子数：36

确定

测试详情 隐藏棋盘



棋局胜负：白棋赢

先后手：黑棋先手

棋局难度：中级

当前棋子：白棋

当前坐标：C1

64 / 64

确定

高级：（黑棋后手，领先28子）

系统测试

main.py

接口测试

接口测试通过。

用例测试 展示棋盘

测试点	状态	时长	结果
对手对弈	✓	30s	黑棋获胜, 领先棋子数：28

提交结果

系统测试

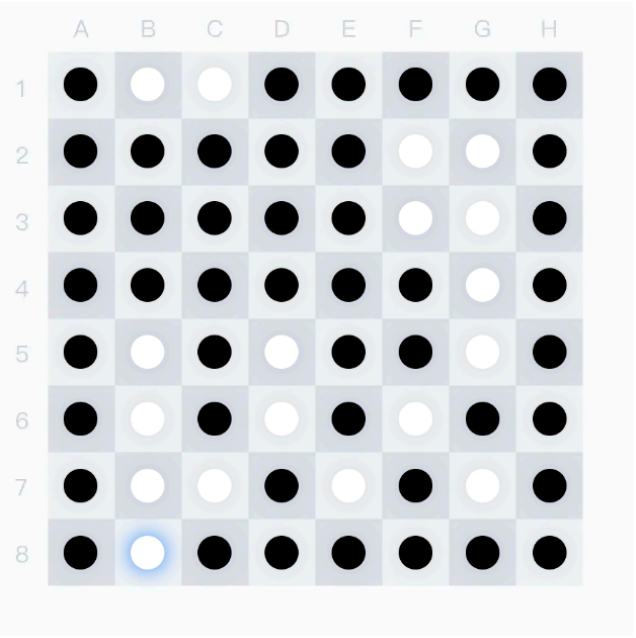
main.py

接口测试

接口测试通过。

用例测试

隐藏棋盘



棋局胜负: 黑棋赢

先后手: 白棋后手

棋局难度: 高级

当前棋子: 白棋

当前坐标: B8

64 / 64

提交结果