程序报告

学号：2211044　　姓名：陆皓喆

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” (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。
>
> **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：
>
> - 在一个空格处落下一个棋子，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。

> **对问题的理解**
>
> 实验平台已经给出了`game.py`和`board.py`的源代码，所以我们理论上只需要实现`AIplayer`模块的代码就可以了。

二、设计思想

2.1 蒙特卡洛树算法原理

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**
  从根节点开始，根据UCB的大小选择一个UCB最大的子结点，直到到达叶子节点或具有还未被扩展的子节点。
- **扩展**
  若选择的节点不是终止节点，则随机扩展它的一个为扩展过的后继节点。
- **模拟**
  从上一环节要被扩展的节点出发，模拟扩展搜索树。在本实验中，采用随机模拟的方法，即从该节点出发，随机走子，直到走子结束。
- **反向传播**
  根据上一环节的模拟结果回溯更新模拟路径上的奖励均值和被访问次数。

2.2 设计算法来实现对弈

我们只考虑蒙特卡洛树的实现算法，其他类我们在此处不作考虑。

2.2.1 选择

```
    def search(self):
        if len(self.root.actions) == 1:
            return self.root.actions[0]

        return self.search_by_montecarlo_tree()
```

这一部分，首先判定是否是第一步操作，如果不是的话我们就执行`search_by_montecarlo_tree`函数，跳转到下方

```
    def search_by_montecarlo_tree(self):
        try:
            func_timeout(timeout=3, func=self.build_montecarlo_tree)
        except FunctionTimedOut:
            pass

        return self.root.get_best_reward_child().preAction
```

该函数实现了对于节点最佳的扩展路径的选择，经过调参，我们发现在每次搜索时长为3秒左右时，效果最佳，所以我们限制了timeout为3秒，返回对于奖励值最大的节点的预测。

```
    def get_value(self):
        if self.visit_count == 0:
            return
        for color in ['X', 'O']:
            self.value[color] = self.reward[color] / self.visit_count + \
                                Node.coefficient * math.sqrt(
                math.log(self.parent.visit_count)*2 / self.visit_count)
```

这一块内容实现了对于奖励值的计算，使用了课上说到过的公式。

2.2.2 扩展

```
    def expand(self, node: Node):#扩展
        if len(node.actions) == 0:
            board = deepcopy(node.board)
            color = 'X' if node.color == 'O' else 'O'
            child = Node(board=board, color=color, parent=node,
pre_action="none", root_color=self.color)
            node.add_child(child)
            return node.best_child
        for action in node.actions:
            board = deepcopy(node.board)
            board._move(action=action, color=node.color)
            color = 'X' if node.color == 'O' else 'O'
            child = Node(board=board, color=color, parent=node,
pre_action=action, root_color=self.color)
            node.add_child(child=child)
        return node.best_child
```

扩展函数实现对未完全扩展 / 叶节点的扩展。

扩展函数对 node 讨论以下情况：

（1）node 对应的棋局没有可以走子的位置了：返回node的父节点
（2）node 还有未扩展过的走子位置（即还存在可走子位置没被放进children里）

随机选择一个未扩展过的子节点，并把该子节点的扩展结果放进 node 的 children 里。返回新扩展的子节点。

2.2.3 模拟

```
    def simulation(self, node: Node):#模拟
        board = deepcopy(node.board)
        color = node.color
        while not self.game_over(board=board):
            actions = list(board.get_legal_actions(color=color))
            if len(actions) != 0:
                board._move(random.choice(actions), color)
            color = 'X' if color == 'O' else 'O'
        winner, difference = board.get_winner()
        return winner, difference
```

这一部分完成了蒙特卡洛树的模拟操作，对于下面的操作与对方的操作进行模拟，从而找出最优解。

2.2.4 反向传播

```
    def back_propagation(self, node: Node, winner, difference):#反向传播
        while node is not None:
            node.visit_count += 1
            if winner == 0:
                node.reward['O'] -= difference
                node.reward['X'] += difference
            elif winner == 1:
                node.reward['X'] -= difference
            elif winner == 2:
                pass
            if node is not self.root:
                node.parent.visit_count += 1
                for child in node.parent.children:
                    child.get_value()
                node.parent.visit_count -= 1
            node = node.parent
```

这一部分，实现的是蒙特卡洛树的反向传播，根据winner的不同，分别完成各节点的奖励值的更新，并将各root的访问次数做更新。

2.2.5 节点类

```
class Node:#定义节点类
    coefficient = 2


    def __init__(self, board, color, root_color, parent=None, pre_action=None):
        self.board = board
        self.color = color.upper()
        self.root_color = root_color
        self.parent = parent
        self.children = []
        self.best_child = None
        self.get_best_child()
        self.preAction = pre_action
        self.actions = list(self.board.get_legal_actions(color=color))
        self.isOver = self.game_over()
        self.reward = {'X': 0, 'O': 0}
        self.visit_count = 0
        self.value = {'X': 1e5, 'O': 1e5}
        self.isLeaf = True
        self.best_reward_child = None
        self.get_best_reward_child()


    def game_over(self):
```

```
        black_list = list(self.board.get_legal_actions('X'))
        white_list = list(self.board.get_legal_actions('O'))
        game_is_over = len(black_list) == 0 and len(white_list) == 0
        return game_is_over

    def get_value(self):
        if self.visit_count == 0:
            return
        for color in ['X', 'O']:
            self.value[color] = self.reward[color] / self.visit_count + \
                                Node.coefficient * math.sqrt(
                math.log(self.parent.visit_count)*2 / self.visit_count)

    def add_child(self, child):
        self.children.append(child)
        self.get_best_child()
        self.get_best_reward_child()
        self.isLeaf = False

    def get_best_child(self):
        if len(self.children) == 0:
            self.best_child = None
        else:
            sorted_children = sorted(self.children, key=lambda child:
child.value[self.color], reverse=True)
            self.best_child = sorted_children[0]
        return self.best_child

    def get_best_reward_child(self):
        if len(self.children) == 0:
            best_reward_child = None
        else:
            sorted_children = sorted(self.children, key=lambda child:
child.reward[

 self.color] / child.visit_count if child.visit_count > 0 else -1e5,
                                     reverse=True)
            best_reward_child = sorted_children[0]
        self.best_reward_child=best_reward_child
        return self.best_reward_child
```

这一部分是我们实验中所需要的节点。我们在root类中就已经将一些对应的函数写在里面了，在后续的四个核心步骤中，我们只需要调用对应的函数即可，不需要在外部再写函数了。

三、代码内容

我们设计了蒙特卡洛树算法来实现该部分的内容。

```
class Node:#定义节点类
    coefficient = 2

    def __init__(self, board, color, root_color, parent=None, pre_action=None):
        self.board = board
```

```
        self.color = color.upper()
        self.root_color = root_color
        self.parent = parent
        self.children = []
        self.best_child = None
        self.get_best_child()
        self.preAction = pre_action
        self.actions = list(self.board.get_legal_actions(color=color))
        self.isOver = self.game_over()
        self.reward = {'X': 0, 'O': 0}
        self.visit_count = 0
        self.value = {'X': 1e5, 'O': 1e5}
        self.isLeaf = True
        self.best_reward_child = None
        self.get_best_reward_child()


    def game_over(self):
        black_list = list(self.board.get_legal_actions('X'))
        white_list = list(self.board.get_legal_actions('O'))
        game_is_over = len(black_list) == 0 and len(white_list) == 0
        return game_is_over

    def get_value(self):
        if self.visit_count == 0:
            return
        for color in ['X', 'O']:
            self.value[color] = self.reward[color] / self.visit_count + \
                                Node.coefficient * math.sqrt(
                math.log(self.parent.visit_count)*2 / self.visit_count)

    def add_child(self, child):
        self.children.append(child)
        self.get_best_child()
        self.get_best_reward_child()
        self.isLeaf = False

    def get_best_child(self):
        if len(self.children) == 0:
            self.best_child = None
        else:
            sorted_children = sorted(self.children, key=lambda child:
child.value[self.color], reverse=True)
            self.best_child = sorted_children[0]
        return self.best_child

    def get_best_reward_child(self):
        if len(self.children) == 0:
            best_reward_child = None
        else:
            sorted_children = sorted(self.children, key=lambda child:
child.reward[

 self.color] / child.visit_count if child.visit_count > 0 else -1e5,
                                     reverse=True)
            best_reward_child = sorted_children[0]
```

```
        self.best_reward_child=best_reward_child
        return self.best_reward_child

from copy import deepcopy
import csv
import torch
from func_timeout import func_timeout, FunctionTimedOut
import math
import os.path
import random

class MonteCarlo_Search:#主体部分，执行蒙特卡洛树搜索
    def __init__(self, board, color):
        self.root = Node(board=deepcopy(board), color=color, root_color=color)
        self.color = color
        self.experience = {"state": [], "reward": [], "color": []}
        self.max_experience = 10000000000
        self.trans = {"X": 1, "O": -1, ".": 0}
        self.learning_rate = 0.3
        self.epsilon = 0.3
        self.gamma = 0.999


    def get_experience(self):
        queue = []
        for child in self.root.children:
            queue.append(child)
        while len(queue) > 0:
            if len(self.experience) == self.max_experience:
                break
            if not queue[0].isLeaf:
                self.add_experiences(queue[0])
                for child in queue[0].children:
                    queue.append(child)
            queue.pop(0)

    def add_experiences(self, node: Node):

        if len(self.experience["reward"]) == self.max_experience:
            return

        experience = self.get_state(node)
        self.experience["state"].append(experience)
        reward = node.reward["X" if node.color == "O" else "O"] /
node.visit_count
        self.experience["reward"].append(reward)
        self.experience["color"].append(node.color)

    def get_state(self, node):
        new_statement=node.board._board
        return new_statement

    def search(self):
        if len(self.root.actions) == 1:
            return self.root.actions[0]
```

```
        return self.search_by_montecarlo_tree()

    def search_by_montecarlo_tree(self):
        try:
            func_timeout(timeout=3, func=self.build_montecarlo_tree)
        except FunctionTimedOut:
            pass

        return self.root.get_best_reward_child().preAction

    def build_montecarlo_tree(self):
        while 1==1:
            current_node = self.select()
            if current_node.isOver:
                winner, difference = current_node.board.get_winner()
            else:
                if current_node.visit_count:
                    current_node = self.expand(current_node)
                winner, difference = self.simulation(current_node)
            self.back_propagation(node=current_node, winner=winner,
difference=difference)

    def select(self):#选择
        current_node = self.root
        while not current_node.isLeaf:
            if random.random() > self.epsilon:
                current_node = current_node.get_best_child()
            else:
                current_node = random.choice(current_node.children)
            self.epsilon *= self.gamma
        return current_node

    def simulation(self, node: Node):#模拟
        board = deepcopy(node.board)
        color = node.color
        while not self.game_over(board=board):
            actions = list(board.get_legal_actions(color=color))
            if len(actions) != 0:
                board._move(random.choice(actions), color)
            color = 'X' if color == 'O' else 'O'
        winner, difference = board.get_winner()
        return winner, difference


    def expand(self, node: Node):#扩展
        if len(node.actions) == 0:
            board = deepcopy(node.board)
            color = 'X' if node.color == 'O' else 'O'
            child = Node(board=board, color=color, parent=node,
pre_action="none", root_color=self.color)
            node.add_child(child)
            return node.best_child
        for action in node.actions:
            board = deepcopy(node.board)
            board._move(action=action, color=node.color)
            color = 'X' if node.color == 'O' else 'O'
```

```
            child = Node(board=board, color=color, parent=node,
pre_action=action, root_color=self.color)
            node.add_child(child=child)
        return node.best_child

    def back_propagation(self, node: Node, winner, difference):#反向传播
        while node is not None:
            node.visit_count += 1
            if winner == 0:
                node.reward['O'] -= difference
                node.reward['X'] += difference
            elif winner == 1:
                node.reward['X'] -= difference
            elif winner == 2:
                pass
            if node is not self.root:
                node.parent.visit_count += 1
                for child in node.parent.children:
                    child.get_value()
                node.parent.visit_count -= 1
            node = node.parent

    def game_over(self, board):#判定游戏是否结束
        black_list = list(board.get_legal_actions('X'))
        white_list = list(board.get_legal_actions('O'))
        game_is_over = len(black_list) == 0 and len(white_list) == 0
        return game_is_over


class AIPlayer:
    def __init__(self, color: str):
        self.color = color.upper()
        self.comments = "请稍后，{}正在思考".format("黑棋(X)" if self.color == 'X'
else "白棋(O)")

    def get_move(self, board):
        print(self.comments)
        action = MonteCarlo_Search(board, self.color).search()#执行蒙特卡洛树搜索，并
返回对应的action
        return action
```

四、实验结果

在线测试

在写完程序后，我和自己写的AI下了几盘棋，都被AI虐了（不是我菜，是我的AI强bushi），下面附上下棋的截图。

```
=====开始游戏!=====

  A B C D E F G H
1 . . . . . . . .
2 . . . . . . . .
3 . . . . . . . .
4 . . . O X . . .
5 . . . X O . . .
6 . . . . . . . .
7 . . . . . . . .
8 . . . . . . . .
统计棋局: 棋子总数 / 每一步耗时 / 总时间
黑    棋: 2 / 0 / 0
白    棋: 2 / 0 / 0

请'黑棋-X'方输入一个合法的坐标(e.g. 'D3', 若不想进行，请务必输入'Q'结束游戏。):  D3
  A B C D E F G H
1 . . . . . . . .
2 . . . . . . . .
3 . . . X . . . .
4 . . . X X . . .
5 . . . X O . . .
6 . . . . . . . .
7 . . . . . . . .
8 . . . . . . . .
统计棋局: 棋子总数 / 每一步耗时 / 总时间
黑    棋: 4 / 2 / 2
白    棋: 1 / 0 / 0
```

平台测试

由于蒙特卡洛树模拟十分耗费时间，所以在此我只测试了两个样例，即高级难度黑棋先手、高级难度白棋后手，两个样例均花费了两分钟左右的时间，最后也是获得了胜利。

测试点1：高级难度 黑棋先手

接口测试

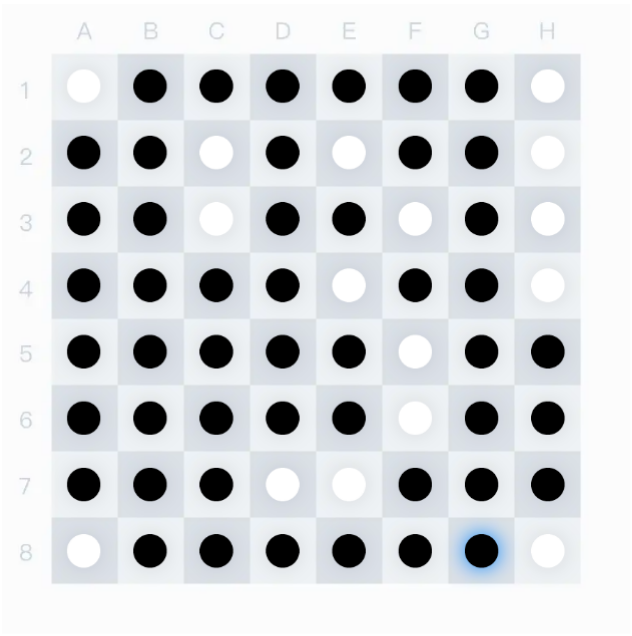
接口测试通过。

用例测试 展示棋盘

测试点	状态	时长	结果
对手对弈	✓	131s	黑棋获胜, 领先棋子数: 32

提交结果

用例测试 隐藏棋盘



棋局胜负: 黑棋赢

先后手: 黑棋先手

棋局难度: 高级

当前棋子: 黑棋

当前坐标: G8

64 / 64

提交结果

可以看出，我们领先了对方32个子，获胜！

测试点2：高级难度 白棋后手

接口测试

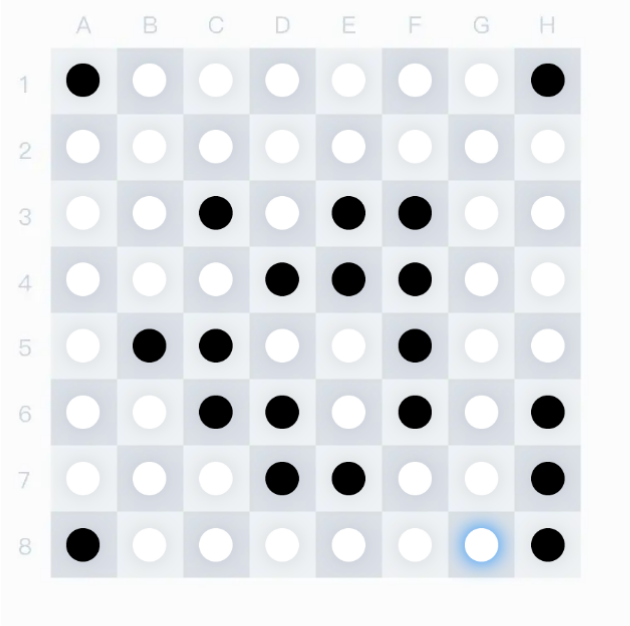
接口测试通过。

用例测试 展示棋盘

测试点	状态	时长	结果
对手对弈	✓	103s	白棋获胜, 领先棋子数：24

提交结果

用例测试 隐藏棋盘



棋局胜负: 白棋赢

先后手: 白棋后手

棋局难度: 高级

当前棋子: 白棋

当前坐标: G8

64 / 64

提交结果

可以看出，我们领先了24个棋子，获胜！

五、总结

1. 本实验的关键在于实现蒙特卡洛树的四个过程：选择、扩展、模拟、反向传播。
2. 问题难点在于存在终局节点（没有可以走子的位置），要正确讨论这种情况下选择、扩展、模拟、反向传播的实现过程，否则会干扰到实现步骤。
3. 难点还在于如何设计`get_best_reward_child`函数，由于本题中，黑白棋子的个数更能体现最终结果，并且在模拟阶段受走棋次数限制，不一定能够完整走完棋局，所以在设计`get_best_reward_child`和反向传播的时候，重点考察每轮棋局的黑白棋子数。
4. 当我设置每轮时长为60秒的时候，训练效果反而不如每轮3秒来的好，可能是在反向传播与模拟的过程中，出现了一些问题。