程序报告

一、问题重述

（简单描述对问题的理解，从问题中抓住主干，必填）

==========

题目规则的重点如下：

1. 落子的限制：合法的落子即为规则前三点所阐述的落子

2. 棋局的结束条件：双方都无子可下或者棋盘填满，其实棋盘填满这一情况已经被包括在了双方都无子可下的情况内

3. 棋局的胜负条件：棋局结束时棋盘上棋子的数量、超时或者连续三次落子不合法

除了阐述性的规则，题目还提供了 ***board.py*** 和 ***game.py*** 两个必需的文件，后者没有太多值得研究的价值，前者的重要方法如下：

1. ***get_winner()***：该方法并不判定棋局结束与否，仅仅是统计棋盘上黑子与白子的数量，返回胜方以及领先棋子数

2. ***_move()***：该方法实现落子，会改变棋盘对象，这意味着在真正落子前需要保存棋盘对象的棋局

3. ***get_legal_actions()***：该方法获取所有合法的落子位置，需要注意该方法返回的是一个生成器而不是简单的列表，使用返回值之前需要类型转换

二、设计思想

（所采用的方法，有无对方法加以改进，该方法有哪些优化方向（参数调整，框架调整，或者指出方法的局限性和常见问题），伪代码，理论结果验证等… **思考题，非必填**）

==========

实验采用的设计即为教材中的蒙特卡洛搜索。在实验过程中我思考过三种可能的优化方向：

1. 游戏终局分数 reward 的设计

尝试过利用 ***get_winner()***返回的领先棋子数来设计，以及教材中给出的将 reward 设计为±1 的两种做法。

2. UCB1 算法中超参数的设计

查阅过一些资料，最后出于测试时间长以及对 UCB1 算法不甚了解，修改超参数若干次后并没有看出明显差异，没有深入探究。

3. 模拟过程的设计

查阅过资料，最后选择通过教材提到的随机策略来进行模拟。

三、代码内容

（能体现解题思路的主要代码，有多个文件或模块可用多个"====="隔开，必填）

==========

主要内容为实现蒙特卡洛树对象，将蒙特卡洛树的节点以及蒙特卡洛树分别封装为一个对象：

```
#!/usr/bin/Anaconda3/python
# -*- coding: utf-8 -*-

from cmath import pi
```

```
from copy import deepcopy
from math import sqrt, log
import random

from func_timeout import func_timeout, FunctionTimedOut

C = 1 / sqrt(2)
MAX = 1
MIN = 0

class Node:
    def __init__(self, parent, board, color, action, pick):
        # 父节点
        self.parent = parent
        # 子节点
        self.children = []
        # 总收益分数
        self.reward = 0.0
        # 被访问次数
        self.visited = 0
        # 从子节点中选择最大还是最小
        self.pick = pick
        self.board = deepcopy(board)
        self.color = color
        self.action = action
        self.unvisitedActions = list(board.get_legal_actions(color))
        self.bound = 0

    def calculate(self, coe=C):
        self.bound = self.reward / self.visited + coe * \
            sqrt(2 * log(self.parent.visited)/self.visited)


class MCT:
    def __init__(self, color):
        self.rootcolor = color

    def search(self, board):
        # only one action can be taken
        actions = list(board.get_legal_actions(self.rootcolor))
        if(len(actions) == 1):
            return actions[0]

        # create root node
```

```
        root = Node(None, deepcopy(board), self.rootcolor, None, MAX)
        # loop to search
        try:
            func_timeout(55, self.loop, args=[root])
        except FunctionTimedOut:
            pass

        return self.pickbest(root, 0).action

    def loop(self, root):
        while True:
            node = self.select(root)
            winner, reward = self.simulate(node.board, node.color)
            self.backprop(node, reward)

    # return a node to be selected
    def select(self, node):
        while len(list(node.board.get_legal_actions(node.color))):
            if(len(node.unvisitedActions)):
                # randomly pick one
                return self.expand(node)
            else:
                # pick the one with the best bound
                node = self.pickbest(node, C)
        return node

    # return a node to be expanded
    def expand(self, node):
        # randomly pick one action
        action = random.choice(node.unvisitedActions)
        node.unvisitedActions.remove(action)
        board = deepcopy(node.board)
        board._move(action, node.color)
        child = Node(node, board, 'X' if node.color == 'O' else 'O', action,
1-node.pick)
        node.children.append(child)

        return child

    def simulate(self, board, color):
        def gameon(board):
            return       len(list(board.get_legal_actions('X')))       or
len(list(board.get_legal_actions('X')))
```

```
        # 随机进行一场游戏
        newboard = deepcopy(board)
        while gameon(newboard):
            actions = list(newboard.get_legal_actions(color))
            # randomly make an action
            if(len(actions) == 0):
                action = None
            else:
                action = random.choice(actions)
            # move
            if(action is None):
                pass
            else:
                newboard._move(action, color)
            # move in turn
            color = 'X' if color == 'O' else 'O'
        # decide the winner
        winner, reward = newboard.get_winner()
        # reward /= 16
        # reward /= 8
        if  (winner==0  and  self.rootcolor=='X')  or  (winner==1  and
self.rootcolor=='O'):
            # wins
            # reward = reward
            reward = 1
        elif winner==2:
            # reward = reward
            reward = 0
        else:
            # reward = -reward
            reward = -1
        return winner, reward

    def backprop(self, node, reward):
        # 向上更新收益
        while node is not None:
            node.visited += 1
            if node.pick == MAX:
                node.reward -= reward
            else:
                node.reward += reward
            node = node.parent

    def pickbest(self, node, coe):
```

```
        # calculate the bound of each child
        for child in node.children:
            child.calculate(coe=coe)
        # sorted
        sortedlist = sorted(
            node.children, key=lambda child: child.bound, reverse=True)
        # always pick the one with the highest bound
        return sortedlist[0]


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.color = color

    def get_move(self, board):
        """
        根据当前棋盘状态获取最佳落子位置
        :param board: 棋盘
        :return: action 最佳落子位置, e.g. 'A1'
        """
        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))

        #    -----------------  请  实  现  你  的  算  法  代  码
--------------------------------------

        mct = MCT(self.color)
        action = mct.search(board)
        #
------------------------------------------------------------------------
-
```

```
        return action
```

四、实验结果

（实验结果，必填）

==========

初中高级的测试都通过了，其中高级测试了三次，结果为二胜一平：

1. 黑子先手

2. 黑子先手

3. 白子后手

中级和初级也经过了多次测试，结果皆为胜多于负。

五、总结

（自评分析（是否达到目标预期，可能改进的方向，实现过程中遇到的困难，从哪些方面可以提升性能，模型的超参数和框架搜索是否合理等），**思考题，非必填**）

==========

实验过程中，因为一开始对公式的不理解，我将 reward 设计为：

```
winner, reward = newboard.get_winner()
reward /= 64
```

即试图降低随机策略（经验）的影响（一开始认为这种随机不可靠）。经过测试发现，这种策略只能战胜初级的对手，连中级的对手都无法战胜。经过仔细阅读教材以及搜索资料，对蒙特卡洛树倾向于选择访问次数多、表现优秀的节点的性质有了更深入的了解，于是又尝试了提高经验的影响为：

```
reward /= 16
```

或

```
reward /= 4
```

测试表明，这样可以战胜中级的对手了，但是似乎比较难战胜高级对手。最后我又将其调整为教材上的策略，随机棋局获胜则将 reward 设为 1，失败设为-1，平局设为 0，使得其对高级对手的表现又更好了。我将其理解为：由于最后决定采取何种 action 时进行了一次 reward 除以访问次数的除法，如果采用我一开始的做法，实际是降低了领先棋子数较低的走法的影响，从而影响了整体的表现。

除了探索 reward 的设计，我还思考了可以如何改进模拟过程，我没有思考出一个更好的策略。查阅了一些资料，发现 AlphaGo 在模拟这一步并不采用随机策略，而且利用一个神经网络直接进行打分。

在测试中我还发现，思考时间的长短确实可以影响算法的表现。但是因为测试时间长以及熟练程度的问题，我没有对超参数的设置进行更多探索。