程序报告

一、问题重述

用蒙特卡洛树算法编写黑白棋AIplayer

规则如下：

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。

二、设计思路

蒙特卡洛树一般在以下场景中使用：

智能体对环境没有完整的认识，事先并不知道每种行动的优劣，只有他们尝试过了之后才能确定。

程序的思路大致为：

对于当前节点，判断其是否为叶节点。

- 不是
  找到当前节点value最大的子节点，再次判断
- 是
  判断该节点的被访问次数是否是0。
    - 是
      计算其value并且迭代累加回去。
    - 不是
      枚举当前节点所有可能的动作并添加到树中。
      令第一个新节点成为当前节点。
      计算其value值并向上累加。

在本程序中，分别编写了node类和mcts类来实现此算法。

具体逻辑在代码注释中有所表现。

三、代码内容

mctsNode.py

```python
from math import log, sqrt


class Node:
    def __init__(self, board, parent, color, action):
        """
        构造

        :param board:    当前棋盘状态
        :param parent:   父节点
        :param color:    执子方
        :param action:   来到此node的action
        :return:
        """
        self.parent = parent
        self.board = board
        self.color = color
        self.prevAction = action
        self.children = []  # 子节点列表
        self.visit_times = 0  # 被访问次数
        self.unVisitActions = list(board.get_legal_actions(color))  # 目前的棋盘状态下合法的下法

        self.isOver = self.gameOver(board)  # 是否双方都没有合法操作

        if (self.isOver == 0) and (len(self.unVisitActions) == 0):  # 我方无合法操作，但是对方有合法操作
            self.unVisitActions.append("noway")

        self.reward = {'X': 0, 'O': 0}
        self.bestVal = {'X': 0, 'O': 0}

    @staticmethod
    def gameOver(board):
        """
        如果双方都没有合法落子了，游戏结束

        :param board:    当前棋盘状态
        :return:         true/false 是否游戏无法继续
        """
        l1 = list(board.get_legal_actions('X'))
        l2 = list(board.get_legal_actions('O'))
        return len(l1) == 0 and len(l2) == 0

    def calcBestVal(self, balance, color):
        """
        如果双方都没有合法落子了，游戏结束

        :param balance:    参数
        :param color:      当前执子
        :return:
        """
        if self.visit_times == 0:
```

```
            print("-------------------------")
            print("oops!visit_times==0!")
            self.board.display()
            print("-------------------------")
        # 公式
        c1 = self.reward[color] / self.visit_times
        c2 = balance * sqrt(2 * log(self.parent.visit_times) /
self.visit_times)
        self.bestVal[color] = c1 + c2
```

mcts.py

```
from mctsNode import Node
from copy import deepcopy
from func_timeout import FunctionTimedOut, func_timeout
import random
import math


class MonteCarlo:
    # uct方法的实现
    # return: action(string)
    def search(self, board, color):
        """
        构造

        :param board:    当前棋盘状态
        :param color:    执子方
        """

        # actions是当前所有的合法落子
        actions = list(board.get_legal_actions(color))
        # 特殊情况：只有一种落子，那么直接返回这一种。
        if len(actions) == 1:
            return list(actions)[0]

        # 创建根节点
        newBoard = deepcopy(board)
        root = Node(newBoard, None, color, None)

        # 考虑时间限制
        try:
            # 测试程序规定每一步在60s以内
            func_timeout(59, self.whileFunc, args=[root])
        except FunctionTimedOut:
            pass

        # 返回能够得到bestValue的那个action
        return self.best_child(root, math.sqrt(2), color).prevAction

    def whileFunc(self, root):
        """
        构造

        :param root:    根节点
        """
        while True:
```

```
            # mcts four steps
            # selection,expansion
            expand_node = self.tree_policy(root)
            # simulation
            reward = self.default_policy(expand_node.board,
expand_node.color)
            # Backpropagation
            self.backup(expand_node, reward)

    @staticmethod
    def expand(node):
        """
        输入一个节点，在该节点上拓展一个新的节点，使用random方法执行Action，返回新增的
节点

        :param node:    待拓展的节点
        :return:        拓展出来的节点
        """

        # 在有效且没被访问过的落子中随机选择一个，并从列表中删除此落子
        action = random.choice(node.unVisitActions)
        node.unVisitActions.remove(action)

        # 执行action，得到新的board
        newBoard = deepcopy(node.board)
        # 如果还有落子机会
        if action != "noway":
            ###########################################################
            #此处因为pycharm疯狂报错说_move是私有的函数所以都改为了move#
            ###########################################################
            newBoard.move(action, node.color)
        else:
            pass

        newColor = 'X' if node.color == 'O' else 'O'
        newNode = Node(newBoard, node, newColor, action)
        node.children.append(newNode)

        return newNode

    @staticmethod
    def best_child(node, balance, color):
        # 对每个子节点调用一次计算bestValue
        for child in node.children:
            child.calcBestVal(balance, color)

        # 对子节点按照bestValue降序排序
        sortedChildren = sorted(node.children, key=lambda x:
x.bestVal[color], reverse=True)

        # 返回bestValue最大的元素
        return sortedChildren[0]

    def tree_policy(self, node):
        """
        根据exploration/exploitation算法返回最好的需要expand的节点
        注意如果节点是叶子结点（棋局结束）直接返回。

```

```
        :param node:     当前需要开始搜索的节点（例如根节点）
        :return:         还有未展开的节点，那么返回这个展开的节点；都已经被展开了，返回value最好的child
        """
        retNode = node
        # 如果棋局还没有结束
        while not retNode.isOver:
            if len(retNode.unVisitActions) > 0:
                # 还有未展开的节点，那么返回这个展开的节点
                return self.expand(retNode)
            else:
                # 都已经被展开了，返回value最好的child
                retNode = self.best_child(retNode, math.sqrt(2), retNode.color)

        return retNode

    @staticmethod
    def default_policy(board, color):
        """
        蒙特卡罗树搜索的Simulation阶段
        输入一个需要expand的节点，随机操作后创建新的节点，返回新增节点的reward。
        注意输入的节点应该不是子节点，而且是有未执行的Action可以expend的。

        基本策略是随机选择Action。

        :param board:    当前棋盘
        :param color:    当前执子
        :return:         赢家和赢了多少
        """
        newBoard = deepcopy(board)
        newColor = color

        def gameOver(board1):
            l1 = list(board1.get_legal_actions('X'))
            l2 = list(board1.get_legal_actions('O'))
            return len(l1) == 0 and len(l2) == 0

        while not gameOver(newBoard):
            actions = list(newBoard.get_legal_actions(newColor))
            if len(actions) == 0:
                action = None
            else:
                action = random.choice(actions)

            if action is None:
                pass
            else:
                newBoard.move(action, newColor)

            newColor = 'X' if newColor == 'O' else 'O'

        # 0黑 1白 2平局，diff是二者相差的棋子数目
        winner, diff = newBoard.get_winner()
        diff /= 64
        return winner, diff

    @staticmethod
```

```
    def backup(node, reward):
        """
        回溯，将reward加回去

        :param node:         当前节点
        :param reward:
        :return:
        """

        newNode = node
        # 节点不为none时（根节点的parent是none）
        while newNode is not None:
            # 被访问次数增加
            newNode.visit_times += 1

            if reward[0] == 0:
                newNode.reward['X'] += reward[1]
                newNode.reward['O'] -= reward[1]
            elif reward[0] == 1:
                newNode.reward['X'] -= reward[1]
                newNode.reward['O'] += reward[1]
            elif reward[0] == 2:
                pass

            newNode = newNode.parent
```

main.py

```
from mcts import MonteCarlo


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----------------------------------
----

        # 计算在当前棋盘我方的最好策略。
        mcts = MonteCarlo()
        action = mcts.search(board, self.color)
        # ------------------------------------------------------------------
------

        return action
```

四、实验结果

与RandomPlayer对打

试运行（给AI设限5秒内给出结果）

```
from game import Game
from player import HumanPlayer, RandomPlayer
from AIPlayer import AIPlayer

# 黑棋初始化
black_player1 = AIPlayer("X")
black_player2 = HumanPlayer("X")
black_player3 = RandomPlayer("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_player3, white_player)

# 开始下棋
game.run()
```

结果:

```
  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

请等一会，对方 黑棋-X 正在思考中...
  A B C D E F G H
1 . . . . . . . .
```

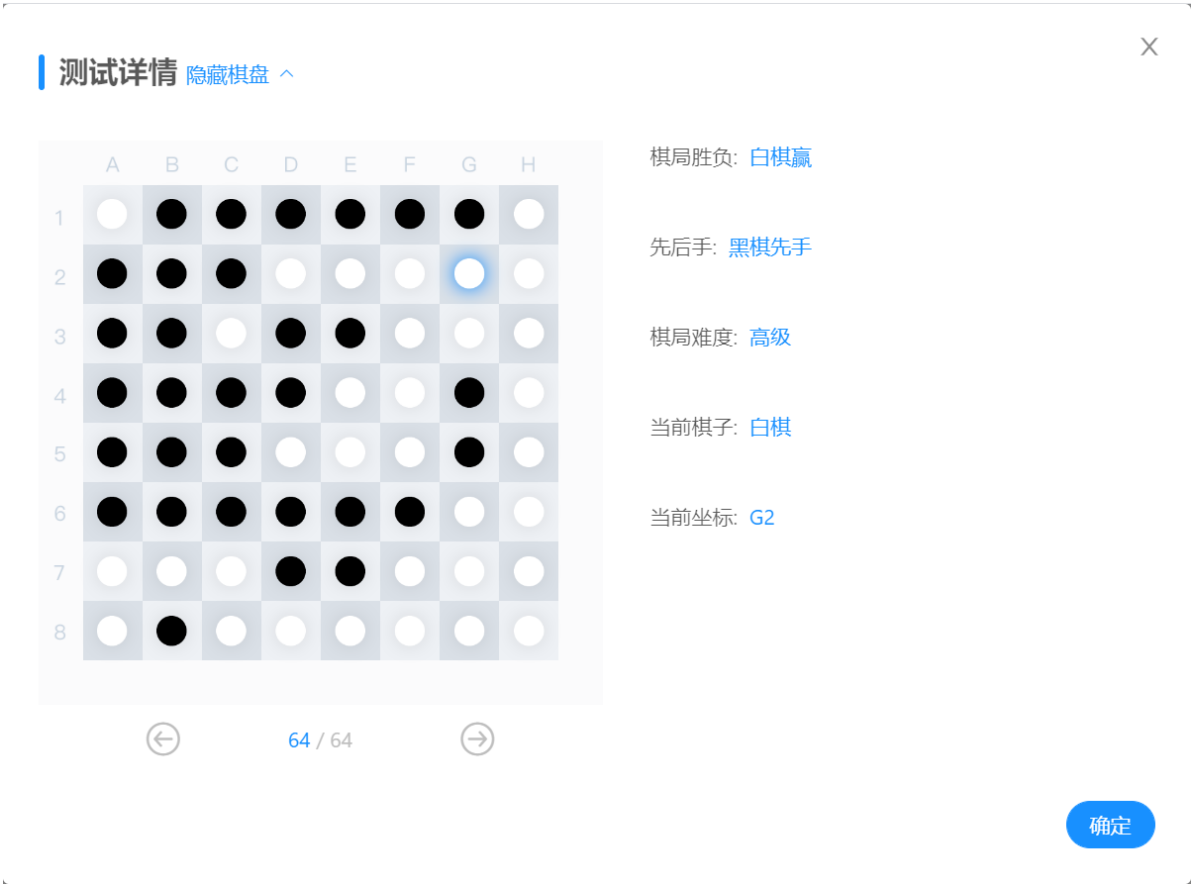
```
2 . . . . . .
3 . . . X . . . .
4 . . . X X . . .
5 . . . X O . . .
6 . . . . . .
7 . . . . . .
8 . . . . . .
统计棋局: 棋子总数 / 每一步耗时 / 总时间
黑    棋: 4 / 0 / 0
白    棋: 1 / 0 / 0

请等一会，对方 白棋-O 正在思考中...
  A B C D E F G H
1 . . . . . .
2 . . . . . .
3 . . . X . . . .
4 . . . X X . . .
5 . . O O O . . .
6 . . . . . .
7 . . . . . .
8 . . . . . .
统计棋局: 棋子总数 / 每一步耗时 / 总时间
黑    棋: 3 / 0 / 0
白    棋: 3 / 5 / 5

请等一会，对方 黑棋-X 正在思考中...
  A B C D E F G H
1 . . . . . .
2 . . . . . .
3 . . . X . . . .
4 . . . X X . . .
5 . . O X O . . .
6 . . X . . . . .
7 . . . . . .
8 . . . . . .
统计棋局: 棋子总数 / 每一步耗时 / 总时间
黑    棋: 5 / 0 / 0
白    棋: 2 / 5 / 5

..............
..............
..............

请等一会，对方 白棋-O 正在思考中...
  A B C D E F G H
1 X O O O O O O O
2 X O O O O O O O
3 X O O O O O O O
4 X X O O O O O O
5 X X O X X X X X
6 X O X X O X X X
7 O X X X X X . .
8 O X X O O X X X
统计棋局: 棋子总数 / 每一步耗时 / 总时间
黑    棋: 28 / 0 / 0
白    棋: 34 / 5 / 145

请等一会，对方 白棋-O 正在思考中...
```

```
  A B C D E F G H
1 X O O O O O O O
2 X O O O O O O O
3 X O O O O O O O
4 X X O O O O O O
5 X X O X O X O X
6 X O X X O O O X
7 O O O O O O O .
8 O X X O O X X X
统计棋局: 棋子总数 / 每一步耗时 / 总时间
黑    棋: 19 / 0 / 0
白    棋: 44 / 5 / 150

请等一会，对方 黑棋-X 正在思考中...
  A B C D E F G H
1 X O O O O O O O
2 X O O O O O O O
3 X O O O O O O O
4 X X O O O O O O
5 X X O X O X O X
6 X O X X O O X X
7 O O O O O O O X
8 O X X O O X X X
统计棋局: 棋子总数 / 每一步耗时 / 总时间
黑    棋: 21 / 0 / 0
白    棋: 43 / 5 / 150


=====游戏结束!=====

  A B C D E F G H
1 X O O O O O O O
2 X O O O O O O O
3 X O O O O O O O
4 X X O O O O O O
5 X X O X O X O X
6 X O X X O O X X
7 O O O O O O O X
8 O X X O O X X X
统计棋局: 棋子总数 / 每一步耗时 / 总时间
黑    棋: 21 / 0 / 0
白    棋: 43 / 5 / 150

白棋获胜!

Process finished with exit code 0
```

与mo平台的高级对打

（给AI设限59秒）

结果：



五、总结

是否达到目标预期：是

可能改进的方向：提高算法性能，不知为何

实验过程中遇到的困难：对于pycharm的不熟悉，对于蒙特卡洛树算法的理解不够深入

收获：由于pycharm无微不至甚至奇奇怪怪的报错了解了更多关于python语言的规范知识。