黑白棋程序报告

学号：2111460　　姓名：张洋

一、问题重述

用蒙特卡洛树算法实现 miniAlphaGo for Reversi，黑白棋的游戏规则如下：

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、设计思想

蒙特卡洛树搜索是一种基于随机模拟和搜索的算法，常用于解决复杂的决策问题。在黑白棋 AI player 的实现中，蒙特卡洛树搜索算法可以用来预测每个可能走步的胜率，并选择最有可能获胜的走步。

下面是利用蒙特卡洛树实现黑白棋 AI player 的程序编写的设计思路：

1. 初始化蒙特卡洛树

首先需要初始化蒙特卡洛树。可以将根节点设置为当前棋盘状态，并添加一个子节点表示下一步棋步。

2. 扩展蒙特卡洛树

对于当前节点，需要根据当前棋盘状态生成所有可能的合法行动，并将这些行动作为子节点添加到当前节点。

3. 选择蒙特卡洛树节点

在蒙特卡洛树的搜索过程中，需要选择节点进行扩展。使用 UCB1 算法，该算法会计算每个节点的上限置信区间值，然后选择具有最大值的节点进行扩展。

4. 模拟蒙特卡洛树节点

对于选定的节点，需要进行随机模拟以获取预测的胜率。可以使用随机策略或启发式策略进行模拟。例如，可以随机生成棋盘状态，并将随机状态作为下一步棋步进行模拟。

5. 回溯蒙特卡洛树节点

在模拟结束后，需要回溯节点并更新胜率和访问次数。需要将模拟结果反向传播到根节

点，更新每个节点的访问次数和胜率信息。

6. 重复搜索过程

一次蒙特卡洛树搜索结束后，可以选择具有最大访问次数的子节点进行下一步棋步。

7. 结束搜索

可以设置搜索次数或时间限制来控制搜索过程的结束，本题中设置时间不超过 60 秒。

总体来说，利用蒙特卡洛树搜索算法实现黑白棋 AI player 需要实现蒙特卡洛树的初始化、扩展、选择、模拟和回溯等操作。

三、代码内容

```
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
```

```
from copy import deepcopy
from func_timeout import FunctionTimedOut, func_timeout
import random
import math
from board import Board

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":
                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
```

```
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':
```

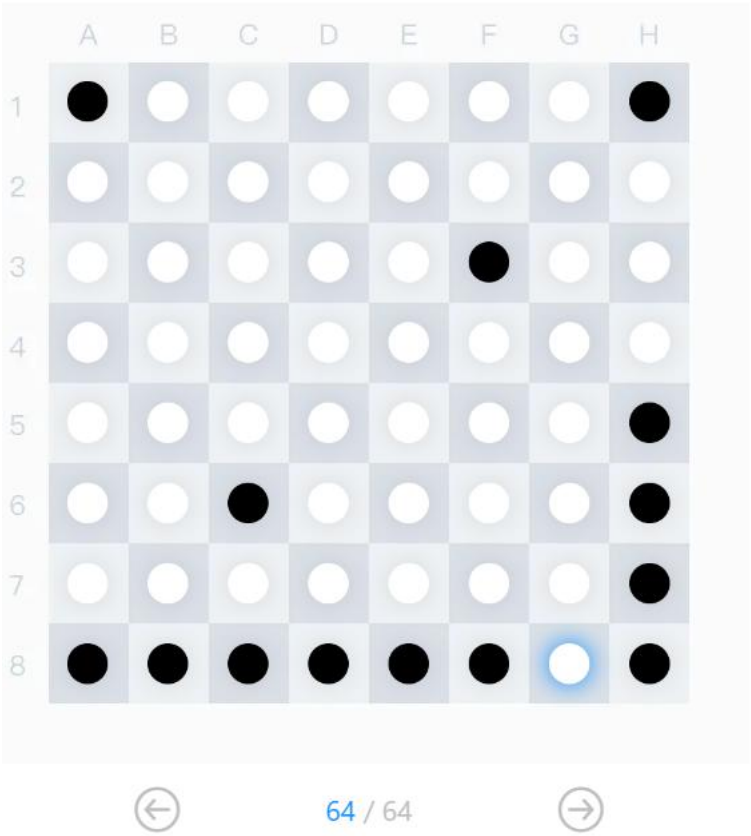
```
            player_name = '黑棋'
        else:
            player_name = '白棋'
        print("请等一会，对方 {}-{} 正在思考中...".format(player_name,
self.color))

        # -----------------请实现你的算法代码
--------------------------------------
        mcts = MonteCarlo()
        action = mcts.search(board, self.color)
        #
------------------------------------------------------------------------
-

        return action
```

四、实验结果

与 mo 平台中的高级等级对打：胜利



五、总结

本次实验已经达到了预期目标，通过了测试用例，打败了平台上三种等级的对手。

可能的改进方向：

1. 改进启发式策略：当前的启发式策略是基于规则的，可以考虑使用更加智能的机器学习算法，例如深度学习，来提高搜索效率和准确性。

2. 使用更高级的搜索算法：蒙特卡洛树搜索算法是一种基于随机化的搜索算法，可以考虑使用更加高级的搜索算法。

3. 使用多种搜索算法的组合：可以考虑使用多种搜索算法的组合来实现黑白棋 AI player，例如蒙特卡洛树搜索算法和 Alpha-Beta 剪枝算法的组合，以达到更好的效果。

4. 加入先验知识：可以考虑将领域专家的先验知识加入到搜索算法中，例如将棋谱中的开局变化和策略加入到搜索算法中，从而提高搜索效率和准确性。

5. 优化代码和算法：可以通过优化代码和算法来提高搜索效率和速度，例如使用更加高效的数据结构、并行计算等。

在实现黑白棋 AI player 的过程中，我学到了许多关于蒙特卡洛树搜索算法的知识和技巧。以下是我个人的心得体会和总结：

1. 参数设置非常重要

在实现蒙特卡洛树搜索算法时，设置参数非常重要。例如，搜索次数、UCB1 算法的权重等，这些参数会直接影响到搜索算法的效果。因此，需要进行多次实验，尝试不同的参数组合，以找到最优参数。

2. 启发式策略对搜索效果有很大的影响

启发式策略可以提高搜索效率。在黑白棋 AI player 中，一些基于规则的启发式策略可以减少搜索空间，使搜索算法更快地找到最优解。例如，可以考虑避免棋盘上出现孤立的子，或者优先考虑占据角落等策略。

3. 代码结构和设计非常重要

实现蒙特卡洛树搜索算法的代码结构和设计非常重要。如果代码结构混乱，会导致代码难以维护和扩展。因此，需要考虑如何分解算法的各个步骤，以及如何将各个步骤组合起来实现完整的搜索算法。

4. 并行计算可以提高搜索速度

蒙特卡洛树搜索算法的计算量非常大，可以考虑使用并行计算来加速搜索速度。例如，可以使用多线程或 GPU 进行并行计算，以提高算法的效率和速度。

实现黑白棋 AI player 是一项有趣和具有挑战性的任务。通过实现蒙特卡洛树搜索算法，我学到了很多关于搜索算法和人工智能的知识和技巧。这些经验和技巧也可以应用到其他领域，例如棋类游戏的 AI 设计、自然语言处理、图像处理等。