黑白棋：程序报告

1. 实验内容

1.1 实验背景

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。

1.2 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。

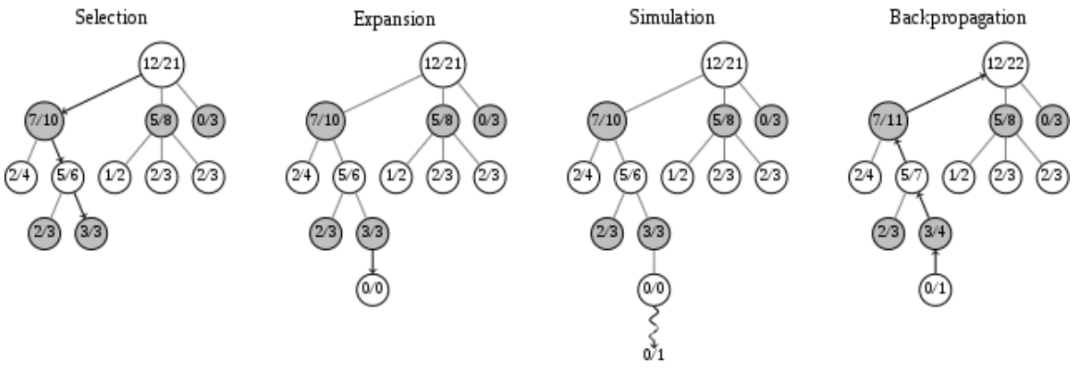
2. 实验设计

2.1 数据结构设计

本次实验使用的数据结构为树，具体而言即构造实验中要求的**蒙特卡洛树**。

在蒙特卡洛树中，每一个树形节点的初始化记录如下：

```
self.father = None #需要记录父亲节点
self.children = [] #需要记录子节点
self.Q = 0 #记录累计得分
self.N = 0 #记录访问的总次数
self.action = action #记录下棋动作
self.board = board #记录此时的棋盘状况
self.color = color #记录棋子对应的颜色
self.action_list = list(board.get_legal_actions(color))#当前可以落子的位置
```



需要记录维护树形结构的父亲与儿子，记录下蒙特卡洛树需要记录的累计得分和总访问次数，另外需要记录下当前棋盘的对应状况，以便后续使用

2.2 算法设计

本次实验使用蒙特卡洛树算法，下面分析算法的步骤

2.2.1 Selection

选择部分的算法思路为：

- 从根节点出发，若当前节点非叶子节点，则判断当前节点是否已拓展完所有可行步骤
- 若拓展完全，则**随机**选择当前UCB值中最优的一个节点，进一步拓展
- 若未拓展完全，则对当前节点进行一次拓展，并将拓展的点作为 **选择点** 返回

```
def select(self):
    node = self.root
    while not node.Is_Leaf():
        #若当前节点非叶
        if not node.Fully():
            return self.expand(node)
        #拓展一个点，返回
        else:
            node = self.bestChild(node)
            #随机选择一个ucb值最好的点，继续做
    return node
```

2.2.2 Expansion

拓展部分的算法思路如下：

- 根据父亲节点的棋盘，随机选择能够行动的一个落子，进行落子
- 构造新节点，将新棋盘，该节点落子值和颜色值记录下。维护树形结构
- 返回该点

```
def expand(self, node):
    action = node.Action_once()
    tmp_board = copy.deepcopy(node.board)
    #要进行深拷贝，不能影响棋盘
    tmp_board._move(action, node.color)
    #对当前棋盘进行这一次action，得到新的棋盘
    if node.color == 'O':
        next_color = 'X'
    elif node.color == 'X':
        next_color = 'O'
    child = TreeNode(action, tmp_board, next_color)
    node.addChild(child)
    #建立新节点后，添加当父亲节点的孩子中
    return child
```

2.2.3 simulate

模拟部分的算法思路如下：

- 对本次要进行模拟的点，进行棋盘拷贝和颜色确定，这是要进行模拟的游戏局面
- 若游戏结束，即双方都无法落子，则返回 **棋子差分数**
- 若游戏可以进行，则随机落子，交换颜色

```
def simulate(self, node):
    #进行模拟，随机落子，查看结果
    tmp_board = copy.deepcopy(node.board)
    #为了不影响棋盘同样深拷贝
    tmp_color = node.color
    while True:
        action_list = list(tmp_board.get_legal_actions(tmp_color))
        if len(action_list) == 0:
            #如果自己不能落子
            if tmp_color == 'O':
                oppo_color = 'X'
            else:
                oppo_color = 'O'
            if len(list(tmp_board.get_legal_actions(oppo_color)))==0:
                # 对手也无法落子,结束
                winner, score = tmp_board.get_winner()
                # 0-黑棋赢,1-白旗赢,2-表示平局
                if winner == 0:
                    return score
                elif winner == 1:
                    return -score
                elif winner == 2:
                    return 0
            else:
                tmp_color = oppo_color
        else:
            action = random.choice(action_list)
            tmp_board._move(action, tmp_color)
            if tmp_color == 'O':
                tmp_color = 'X'
            else:
                tmp_color = 'O'
```

2.2.4 回溯

回溯部分的代码思路如下：

- 根据当前节点，一路回溯到root节点
- 对于回溯需要处理的节点，将Q值进行reward处理，即加上模拟出的分数

```
def backPropagate(self, node, reward):
    while node is not None:
        node.N += 1
        if node.color == 'O':
            node.Q += reward
        else:
            node.Q -= reward
        #递归，去找父亲继续更新
        node = node.father
```

2.3 主函数设计

主函数设计中，主要使用了python提供的func_set_timeout函数，该函数能够控制代码的运行时间，并严格控制在1分钟之内完成当前的一步

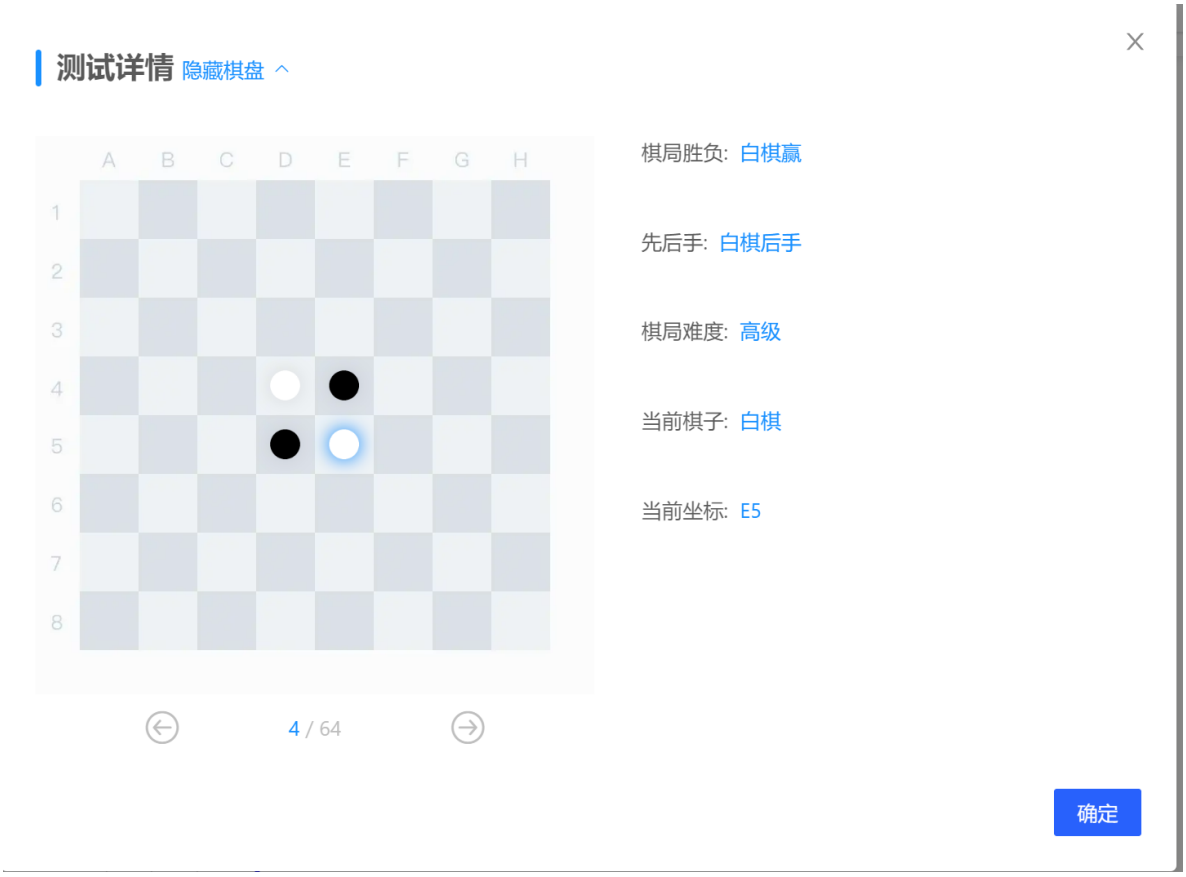
```
def search(self):
    try:
        self._search()
    except FunctionTimedOut:
        pass
    return self.bestChild(self.root).getAction()

@func_set_timeout(58) # 总共运行时间
def _search(self):
    while True:
        select_node = self.select()
        reward = self.simulate(select_node)
        self.backPropagate(select_node, reward)
```

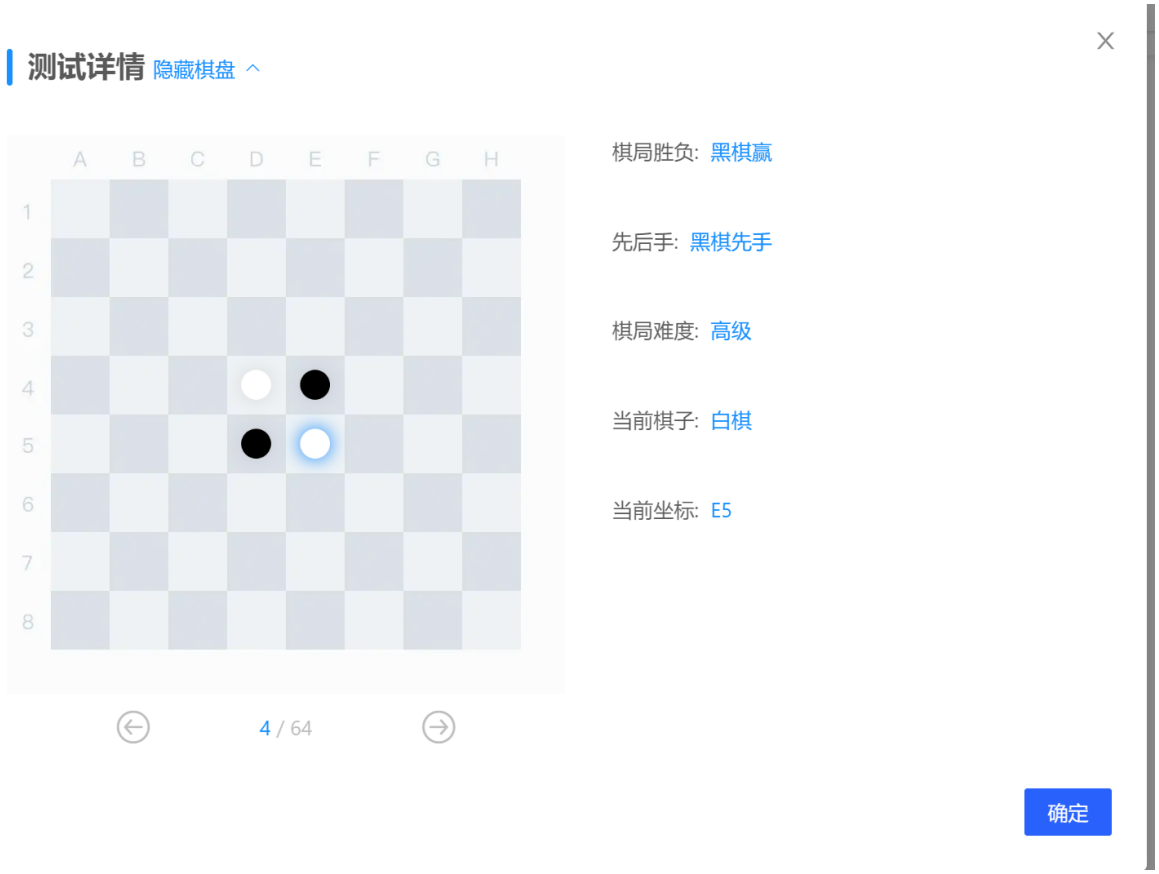
3. 实验结果

下面提供实验结果的正确性验证：

选择白棋，高级难度，胜



选择黑棋，高级难度，胜



4. 思考与心得

本次实验主要实现了蒙特卡洛树算法来实现黑白棋的操作，在之前我还尝试了一次alpha-beta剪枝的算法来实现本次实验，结果显示，如果使用alpha-beta剪枝来实现黑棋操作，将会得到一个十分迅速（大概每一步只需要10s）左右，便可以得到一个领先40+子的结果，然而在白棋上的效果又会弱很多。因为我试验的时候只要是基于论文中已经被研究出的棋盘权值，因此这些参数的作用更加不言而喻。而蒙特卡洛树中更加注重随机性，这使得我们调节的参数相对较少，更多只需要对c值进行调整便可以。