程序报告

学号： 3180105438　　姓名：贺情怡

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）

==========

利用给出的 board.py，使用蒙特卡洛树搜索算法来完成黑白棋 AI。
AI 需要完成的功能：

1. **在当前棋盘状态下，选择一个合法且在算法上最优的落子位置，作为返回值**
2. **搜索及决策时间不超过一分钟，若无合法位置则返回 None**
3. **在游戏结束（即双方均无合法落子位置）时，尽量最大化己方与对方的棋子数量差**

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**）

==========

采用了蒙特卡洛树搜索算法。
该算法利用上限置信区间算法的思想对搜索过程加以优化，其中有超参数 C 可以影响搜索的效果。

$$score = x_{child} + C \cdot \sqrt{\frac{\log(N_{parent})}{N_{child}}}$$

对于一个子节点，其奖励期望的上界可以写作　　其根号下的常数 2 被合并到 C 里面了。可以看出，C 越大，程序越倾向于探索访问次数偏小的节点。

算法伪代码：

```
While (time_is_enough):
    For (node = root; all son_node.visited in node.sons; node = choson_son)
        Choson_son = the son with max UCB of node
    # Selection，从根往下，选择一个儿子没有被完全访问过的节点

    Expand_Candidate = x if ((x in node.sons) and (not x.visited))
    Node_to_expand = random.choice(Expand_Candidate)
    # Expansion，随机选择一个没有被访问过的儿子节点

    Leaf = node
    For (node = Node_to_sxpand; node has son; node = random.choice(node.sons))
```

```
            Leaf = node
        # Simulation，随机选择儿子节点直到叶子节点

        For (node = Leaf; node != root; node = node.father)
            Update(node)
        # Back Propagation，更新访问过的信息以及胜负/奖励分数信息
```

在搜索过程中，每一次“采样”都有四个步骤：选择，扩展，模拟和反向传播：

1. 其中选择主要受到 UCB 函数中 C 值的影响
2. 扩展完全随机
3. 模拟时由于黑白棋合法落子位置与当前局面的相关性非常大，没有找到随机以外的合适方式进行落子（基于当前局面的贪心甚至不如随机算法）
4. 反向传播时更新的收益分数也是可以人为影响算法效益的部分。由于 board.py 提供了分数差的信息，可以用分数差的相关函数作为奖励收益（此处采用了分数差*k，k 为人为规定的系数）

但是根据 UCB 的 score 函数组成，其实能发现 k 如果只是作为乘上去的系数，本质上就是 C，不过添加一个 k 可以方便调整也更直观而已。

最后就是搜索次数可以对搜索效果产生影响了，由于给出了一分钟的落子时限，虽然在测试时使用的 25s 采样时间效果已经不错了，但在提交的时候应该还是会顶着时间上限吧（笑）

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）

==========

UCB1：

```
    def UCB1(self, color, board_in):
        """
        :param color: 当前节点对应的颜色
        :param board_in: 当前棋盘状态
        :return : 根据采样结果，对 AI 方最有利的落子位置
        """
        board = deepcopy(board_in)
        score_act = None
        act = None
        action_list = list(board.get_legal_actions(color))

        rec_sum = 0       # 记录这个节点的总分（用来算 select 的子节点）

        for action in action_list:
            play_board = deepcopy(board)
            play_board._move(action, color)
            tmp_key = tuple(np.ravel(play_board._board))
```

```
            # 计算该 action 后对应的棋盘的 key 值
            if self.rec.get((color, tmp_key)):
                # 访问过则继续计算总分
                rec_sum += self.rec.get((color, tmp_key))

        for action in action_list:
            play_board = deepcopy(board)
            play_board._move(action, color)
            tmp_key = tuple(np.ravel(play_board._board))
            score_tmp = (self.scr.get((color, tmp_key)) / self.rec.get((color,
tmp_key)) +
                self.C * math.sqrt(
                    math.log(rec_sum) / self.rec.get((color, tmp_key))
                ))
            # 计算键值 以及积分
            if score_act == None:
                score_act, act = (score_tmp, action)
            else:
                if score_act < score_tmp:
                    score_act, act = (score_tmp, action)
            # 更新积分最高的子节点
        return act
```

==========

选择：

```
    def Select(self, board):
        """
        :param board: 输入需要被搜索的棋盘
        :return: color 是 select 到最后的那个节点已经落子的棋子颜色, act 是上一个
落子的位置, tmpkey 是这个棋盘的状态
        """
        color = self.color
        while(True):
            # 一直 select 直到有一个节点没有完全被扩展
            action_list = list(board.get_legal_actions(color))
            if len(action_list) == 0:
                return None, None, None

            all_explored = True # 这个节点的子节点是否全部访问过
            non_vis_son = []    # 记录没有访问过的儿子节点

            rec_sum = 0       # 记录这个节点的总分（用来算 select 的子节点）
```

```
            for action in action_list:
                play_board = deepcopy(board)
                play_board._move(action, color)
                tmp_key = tuple(np.ravel(play_board._board))
                # 计算该 action 后对应的棋盘的 key 值
                if not self.rec.get((color, tmp_key)):
                    # 没有访问过则记录 该子节点 以及更新节点未访问信息
                    all_explored = False
                    non_vis_son.append((action, tmp_key))
                else:
                    # 访问过则继续计算总分
                    rec_sum += self.rec.get((color, tmp_key))

            if all_explored:
                # 如果全部访问过，则在该节点中选择分数最高的儿子
                act = self.UCB1(color, board)
            else:
                # 有未访问节点，则随机返回一个未访问节点，作为 extend 的对象
                act, tmp_key = (random.choice(non_vis_son))
                board._move(act, color)
                return (color, act, tmp_key)

            # 到这里的时候应该是要 select 下一个节点了
            board._move(act, color)
            tmp_key = tuple(np.ravel(board._board))
            # 落子，更新新棋盘的 key 值
            self.vis.add((color, tmp_key))
            # 记录路径上的节点信息

            color = "X" if color == "O" else "O"
            # 切换颜色
```

==========

扩展：

```
    def Expand(self, board, color, act, tmpkey):
        """
        :param board: 当前要扩展的棋盘
        :param color: 当前已经落子的棋子颜色
        :param act: 当前已经落子的位置
        :param tmpkey: 当前棋盘状态
        :return: 返回乘上系数后得到的分差
        """
        game_state, scr_diff = self.Simulate(board, color)
```

```
        self.rec[(color, tmpkey)] = 1
        # 记录该节点下的访问次数+1
        if (game_state == 0 and self.color == "O") or (game_state == 1 and
self.color == "X"):
            scr_diff = - scr_diff
            # 把 scr_diff 改成（AI-对方）的分差，可以为负
        scr_diff *= 0.4
        # 加一个系数
        if color == self.color:
            # 如果当前决策节点的颜色是 AI 的颜色，则加上分差，否则减去分差
            self.scr[(color, tmpkey)] = scr_diff
        else:
            self.scr[(color, tmpkey)] = - scr_diff
        return scr_diff
```

==========

模拟：

```
    def Simulate(self, board, player):
        """
        用随机来模拟下棋过程
        :param board: 当前棋盘状态
        :param player: 当前刚完成落子的玩家
        :return: (winner, 分数差)，其中 winner 是 0 黑棋， 1 白棋， 2 平局
        """
        while(True):
            player = "X" if player == "O" else "O"
            # 切换执棋方
            legal_actions = list(board.get_legal_actions(player))
            if len(legal_actions) == 0:
                if self.game_over(board):
                    return board.get_winner()
                    # 0 黑棋， 1 白棋， 2 平局
                    # 后面还有个分数差的参数
                    break
                else:
                    continue

            if len(legal_actions) == 0:
                action = None
            else:
                action = random.choice(legal_actions)
            # 用随机落子来模拟
            if action is None:
```

```
                continue
            else:
                board._move(action, player)
                if self.game_over(board):
                    return board.get_winner()
```

==========

Back Propagation：

```
    def BackPropagate(self, scr_diff):
        """
        :param scr_diff: 乘上系数的 AI 与对手的分数差
        """
        for (color, key) in self.vis:
            self.rec[(color, key)] += 1
            if color == self.color:
            # 如果当前决策节点的颜色是 AI 的颜色，则加上分差，否则减去分差
                self.scr[(color, key)] += scr_diff
            else:
                self.scr[(color, key)] -= scr_diff
```

==========

UCTS 的主要部分：

```
    def MCTS_choice(self, board_input):
        """
        :param board_input: 输入当前棋盘
        :return: 返回落子坐标
        树的状态节点用 rec 和 scr 两个 dict 来存储，存下了（当前落子方，棋盘状态）：（访问次数，合计分数）的状态
        """
        starttime = datetime.datetime.now()

        count = 0
        while True:
            count += 1
            currenttime = datetime.datetime.now()

            if (currenttime - starttime).seconds > 3 or count > 1000:
                break

            board = deepcopy(board_input)
            color = "X" if self.color == "O" else "O"
            # color 是对方的颜色
```

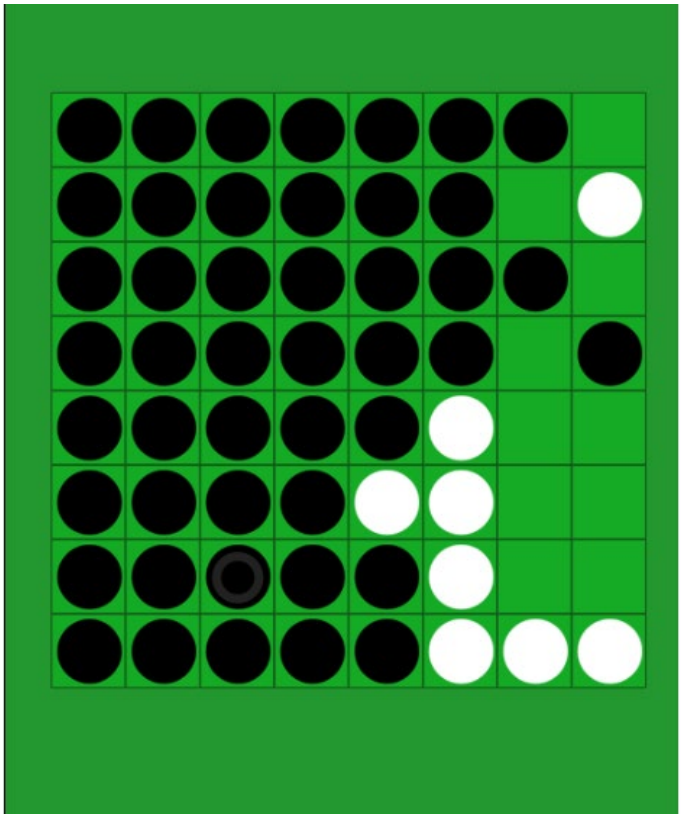
```
            self.vis = set()
            # 记录树上搜索过的路径，方便更新

            color, act, tmpkey = self.Select(board)
            # color 是 select 到最后的那个节点已经落子的棋子颜色
            # act 是上一个落子的位置
            # tmpkey 是这个棋盘的状态

            if color == None:
                # 如果没有可以落子的地方，进入下一轮尝试
                continue
            scr_diff = self.Expand(board, color, act, tmpkey)
            # Expand 得到当前扩展节点的分数，并用于 bp
            self.BackPropagate(scr_diff)

        print(count)
        return self.UCB1(self.color, board_input)
```

四、实验结果

（实验结果，必填）

==========

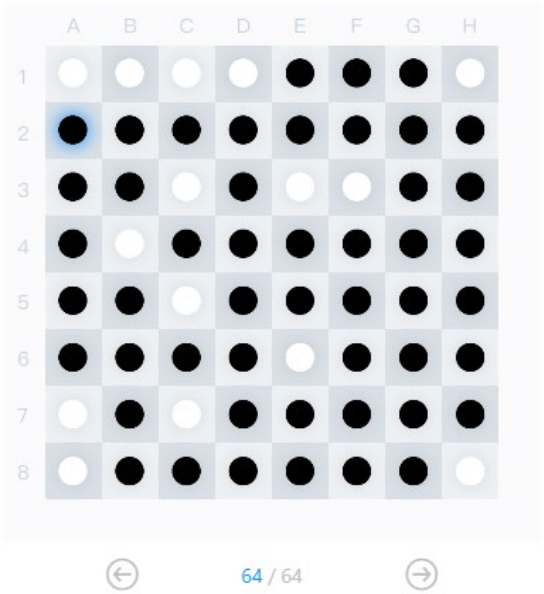
下图为与网站（http://www.7k8k.com/h5/3906_swf.html）的黑白棋 AI 对战结果，白子为网站 AI，黑子为蒙特卡洛树搜索结果：



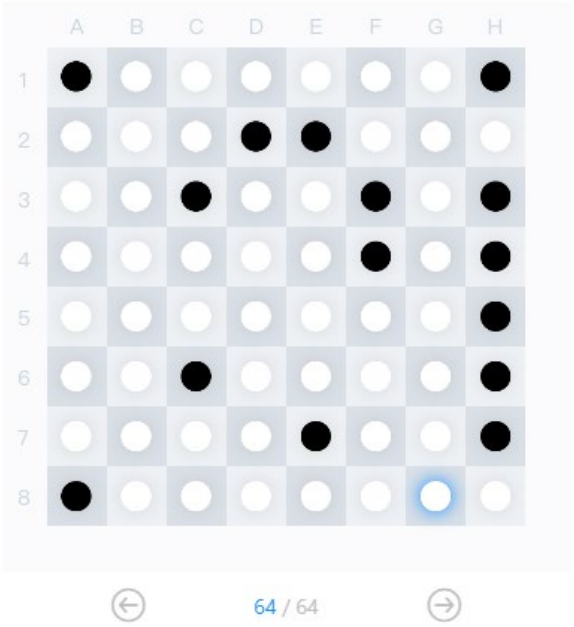
==========

下面是采样时间 25 秒，超参数 C 为 1.4，k 为 0.4 的 AI 与测试平台上提供的棋局对战的结果，可以看出效果很好且胜负情况较为稳定：

测试详情 隐藏棋盘

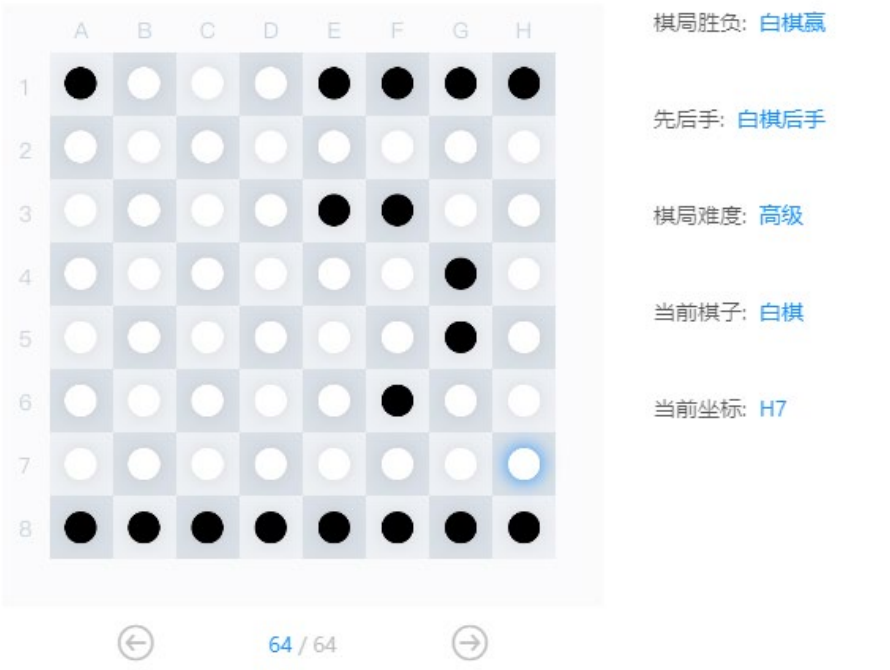


测试详情 隐藏棋盘



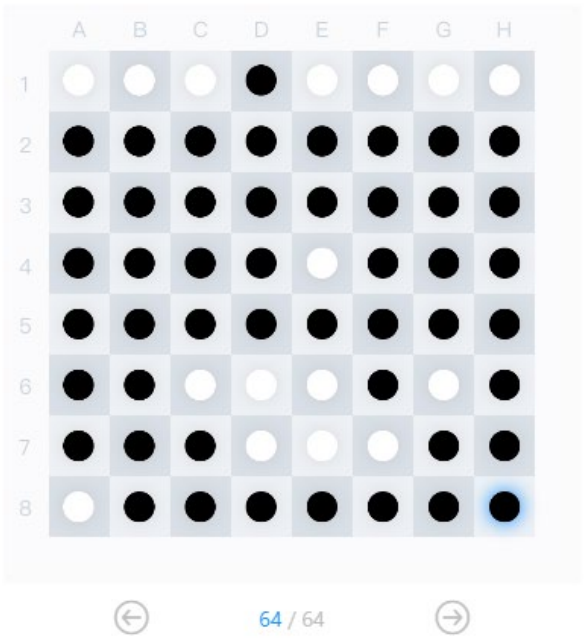
==========

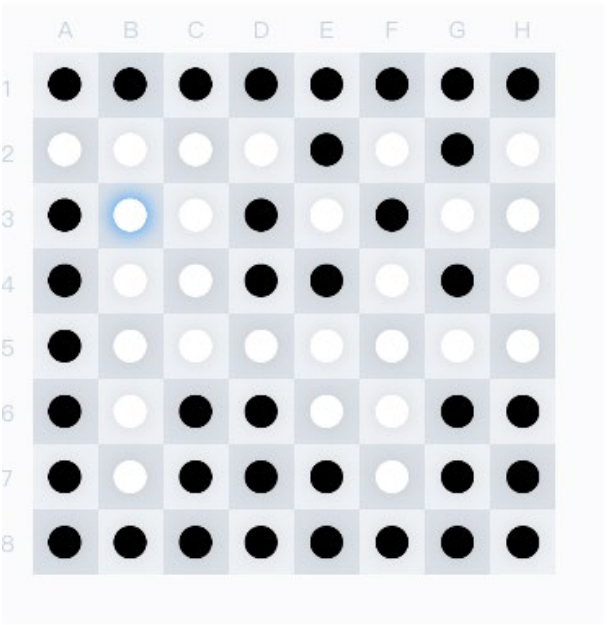
K=0.4，采样时间 55 秒的结果：



==========

k=0.2，运行时间 55 秒则不那么稳定：





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**）

==========

因为很少使用 python 写长代码，这次试验挑战还挺大的（尤其是需要记录当前棋局状态作为树的节点，操作了很久才成功把棋盘 hash 到词典里），从 vim 切换到 vscode 之后，debug 倒是方便了很多。

参考了这篇蒙特卡洛树解决五子棋的 blog（https://www.cnblogs.com/xmwd/p/python_game_based_on_MCTS_and_UCT_RAVE.html），它的树节点是（color，position），并没有记录棋局状态，只是记录了落子位置，想必会有一定误差，但对于五子棋的大棋盘来说应该更加合理。
这份代码也没有把采样的四个步骤分开，虽然可读性较差，但他这个写法也非常巧妙（虽然对于未探索完但探索了一部分的节点的处理有点小 bug），直接找出整条从根到叶子的树链，模拟的随机可以当成 select 的特例，而 extend 只需要对第一个随机选择的节点特殊处理即可。

也参考了这个黑白棋教程的 report 部分（https://github.com/MolinDeng/Othello-MCTS），主要是对照和自己的理解是否有出入。

在最后的测试阶段中，我感受到人工选取的超参数对 AI 效果的影响有多大。
在选取 k=0.4 之前还用了很久 k=0.2 的算法，但哪怕运行时间提升到 55 秒，也很明显感受到它的稳定性较差，受随机下棋影响很大，猜测是采样结果分数的影响过小，相当于 C 的取值太大，导致其一直无法收敛到较优的解上。