程序报告

学号：2112514　　姓名：辛浩然

1　问题重述

黑白棋，也称翻转棋，是一种二人对弈棋类游戏。游戏使用 8x8 的棋盘，其中初始状态中心的四个格子摆放黑白两种棋子各两个，其他格子为空。黑方先行，双方轮流落子，每次必须翻转对手至少一个棋子的颜色，然后在空格上落下自己的棋子。当棋盘填满或者双方都无法落子时游戏结束，得到黑色棋子数目多的一方胜利。

该问题的本质在于如何制定一种策略找到最优落子。传统的方法是使用极大极小算法，但是由于黑白棋中落子的选择非常多，很难穷举所有情况，因此使用蒙特卡洛树搜素算法是一个较好的选择。

蒙特卡洛树搜索是一种人工智能搜索算法，其主要思想是通过模拟随机落子并统计胜率，来评估每个落子位置的优劣，并在此基础上进行搜索，直至找到最优解。在黑白棋中，蒙特卡洛树搜索可以通过模拟多次随机落子并统计胜率来评估每个落子位置的价值，然后选择具有最高价值的落子位置进行下一步操作。由于黑白棋的状态空间非常庞大，使用传统的极小极大搜索算法进行解决非常困难，而蒙特卡洛树搜索在应对大规模状态空间方面有很好的效果，因此被广泛应用于黑白棋等博弈问题的解决。

2　设计思想

2.1　UCB1 算法

UCB1 算法是一种用于多臂赌博机问题的算法，其中多个赌博机的每个臂都有一个未知的概率分布，我们的目标是找到具有最高期望奖励的臂。

UCB1 算法的思想是在每个时刻都选择未被选择的臂中具有最高置信上限的臂来选择。该算法使用置信上限作为度量，该度量考虑了该臂已经被选择的次数以及每次选择该臂所获得的奖励的平均值。每次选择臂时，我们计算每个臂的置信上限，然后选择置信上限最高的臂。具体而言，对于第 t 轮，我们选择臂 i_t，其中

$$i_t = \text{argmax}_i \left\{ \bar{X}_i + \sqrt{\frac{2 \ln t}{N_i}} \right\}$$

其中，$\bar{X}_i$ 是第 i 个臂在前 $t-1$ 轮中的平均奖励，N_i 是第 i 个臂在前 $t-1$ 轮中被选择的次数。$\sqrt{\frac{2 \ln t}{N_i}}$ 是置信上限项，它度量了我们对第 i 个臂的奖励分布的不确定性。当 N_i 越大时，置信上限项越小，我们更有可能选择其他未被选择的臂。因此，UCB1 算法能够在探索与利用之间取得平衡，并在较短时间内找到最佳的臂。

2.2 蒙特卡洛树搜索

蒙特卡洛搜索是一种以模拟随机走子过程的方式搜索棋盘状态空间的算法，通常应用于棋类游戏和其他游戏领域。它通过对当前状态的多次模拟，从而得出每个棋子落子位置的胜率或得分，选择得分最高的位置作为下一步的决策。

蒙特卡洛搜索一般包含以下过程：

1. 选择（Selection）：从根节点 R 开始，递归选择子节点，直至到达叶节点或到达具有还未被扩展过的子节点的节点 L；
2. 扩展（Expansion）：如果 L 不是一个终止节点，则随机创建其后的一个未被访问节点，选择该节点作为后续子节点 C；
3. 模拟（Simulation）：从节点 C 出发，对游戏进行模拟，直到博弈游戏结束；
4. 反向传播（Backpropagation）：用模拟所得结果来回溯更新导致这个结果的每个节点中获胜次数和访问次数。

在每次选择节点时，蒙特卡洛搜索会采用一些探索和利用的策略，如 UCB1 算法，来在探索未知节点和利用已知节点之间做出权衡。经过多次模拟和反向传播，蒙特卡洛搜索可以得到最优的落子位置，用于指导游戏的决策。

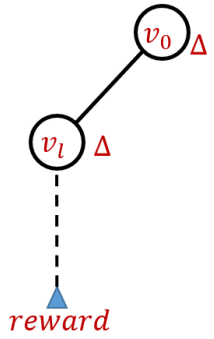
2.3 伪代码

function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

function BESTCHILD(v, c)
  return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$



S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \rightarrow S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

3 代码内容

3.1 代码

```python
class Node:

    def __init__(self, parent, state):  # 节点初始化
        self.visits = 0  # 每个节点被访问次数
        self.score = 0  # 每个节点得分
        self.state = state  # 棋子的左边
        self.unvisited = []  # 未扩展的子节点
```

```
        self.next_nodes = []  # 子节点
        self.parent = parent  # 父节点

    def calUCB1(self, c=1):  # UCB1计算
        if self.visits == 0 or self.parent.visits == 0:
            return 0
        return self.score / self.visits + c * (math.sqrt(2 * math.log(
    self.parent.visits / self.visits)))

class AIPlayer:

    def __init__(self, color):
        # 玩家初始化:'X' - 黑棋, 'O' - 白棋
        self.color = color
        self.select_color = color  # 选择过程所用棋子的颜色
        self.sim_color = ""  # 模拟过程所用棋子的颜色
        self.root = Node(None, (-1, -1))  # 根节点初始化

    def changeColor(self, cur_color):
        # 交换下棋
        return "X" if cur_color == 'O' else 'O'

    def game_over(self, board):
        # 判断游戏是否结束, 双方都无棋子可下游戏结束
        return len(list(board.get_legal_actions('X'))) == 0 and len(
    list(board.get_legal_actions('O'))) == 0

    def Select(self, board):
        # 选择过程
        cur_node = self.root
        while not self.game_over(board):
            if cur_node.unvisited:
                #如果有未拓展过的节点, 则随机扩展一个节点
                action = cur_node.unvisited.pop( random.randrange(len(
    cur_node.unvisited)))
                # 随机选取一个未扩展的子结点
                cur_node.next_nodes.append(Node(cur_node, action))
                # 加入到当前节点的子节点的子结点列表中
                board._move(action, self.select_color)
                # 在棋盘上移动
                self.select_color = self.changeColor(self.select_color)
                # 更换颜色
```

```
                cur_node.next_nodes[-1].unvisited = list(
                    board.get_legal_actions(self.select_color))
                # 初始化新节点的未扩展子结点列表
                return cur_node.next_nodes[-1]
                # 返回新节点
            else:
                best_node = max(cur_node.next_nodes,key=lambda x: x.
calUCB1(),default=None)
                if best_node == None:
                    # 如果当前节点没有任何子节点，创建一个虚拟节点并继
续搜索
                    cur_node.next_nodes.append(Node(cur_node, (-1, -1))
)
                    best_node = cur_node.next_nodes[0]
                    cur_node.next_nodes[0].unvisited = list(
                        board.get_legal_actions(
                            self.changeColor(self.select_color)))
                    # 虚拟节点的 unvisited 列表中包含了所有可能的动作
                    # 更换颜色继续搜索节点
                elif best_node.state != (-1, -1):
                    # 如果选择的不是占位节点，说明有棋可下
                    board._move(best_node.state, self.select_color)
                self.select_color = self.changeColor(self.select_color)
                cur_node = best_node  # 将当前节点设为选择的子结点
        return cur_node

    def Simulate(self, board):
        # 模拟过程
        while not self.game_over(board):
            action_list = list(board.get_legal_actions(self.sim_color))
            if action_list:
                # 有合法落子位置，下棋
                action = random.choice(action_list)
                board._move(action, self.sim_color)
            self.sim_color = self.changeColor(self.sim_color)
            # 即使当前玩家没有合法落子位置，也需要交换颜色
        winner, win_score = board.get_winner()
        # 获得获胜方和获胜子数
        if winner == 0:
            return "X", win_score
        elif winner == 1:
            return "O", win_score
```

```
        else:
            return "-", win_score

    def Backpropagation(self, node, winner, win_score):
        # 反向传播过程
        while node != None:
            if self.select_color == winner:
                node.score -= win_score
            # 如果胜利者与选择过程所用棋子颜色相同，过程中所有节点减去赢的得分
            elif self.changeColor(self.select_color) == winner:
                node.score += win_score
            # 如果胜利者与选择过程所用棋子颜色不同，过程中所有节点加上赢的得分
            node.visits += 1  # 该节点访问次数加1
            self.select_color = self.changeColor(self.select_color)
            node = node.parent  # 移动到父节点进行下一次迭代

    def MCTSmain(self, board):
        start_time = datetime.datetime.now()
        self.root = Node(None, (-1, -1))  # 创建根节点
        self.root.unvisited = list(board.get_legal_actions(self.color))  #初始根节点未访问节点列表
        while ((datetime.datetime.now() - start_time).seconds < 60):  # 时间限制在1分钟内
            new_board = copy.deepcopy(board)
            self.select_color = copy.deepcopy(self.color)
            start_node = self.Select(new_board)  #选择
            self.sim_color = copy.deepcopy(self.select_color)  #赋值模拟选手颜色
            winner, win_score = self.Simulate(new_board)  #模拟走棋
            self.Backpropagation(start_node, winner, win_score)  #反向传播
            # 判断是否有多次重复访问的点
            flag = False
            for n in self.root.next_nodes:
                if n.visits > 1000:
                    flag = True
                    break
            if flag:  # 如果存在多次访问的节点，则停止搜索
                break
        if (self.root.next_nodes):
```

```
123             # 选择UCB1值最大的节点
124             tmp = max(self.root.next_nodes,key=lambda x: x.calUCB1(),
    default=None)
125         return tmp.state
126 
127     def get_move(self, board):
128         # 根据当前棋盘状态获取最佳落子位置
129         if self.color == 'X':
130             player_name = '黑棋'
131         else:
132             player_name = '白棋'
133         print("请等一会，对方 {}-{} 正在思考中...".format(player_name,
    self.color))
134         action = self.MCTSmain(board)
135         return action
```

3.2 主要思路

3.2.1 Node 类

Node 类用于表示搜索树中的节点。其中，每个节点有如下属性：

- visits: 节点被访问的次数
- score: 节点的得分
- state: 节点对应的状态，即棋子的坐标
- unvisited: 未扩展的子节点
- next_nodes: 子节点列表
- parent: 父节点

在节点初始化时，会给上述属性赋初值。其中，visits 和 score 初始化为 0，state、unvisited 和 parent 分别根据传入的参数进行赋值，next_nodes 为空列表。

类中 calUCB1 是计算 UCB1 值的方法。该方法中，先判断当前节点和其父节点是否都已经被访问过，若未被访问过则直接返回 0。否则，根据节点的 score 和 visits 以及父节点的 visits 计算 UCB1 值，返回结果。

3.2.2 AIPlayer 类

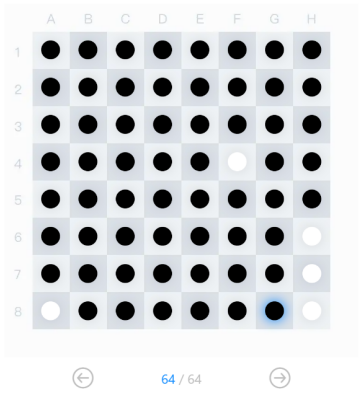
- 在初始化时，根据传入的棋子颜色初始化 AIPlayer 实例，并创建根节点；
- MCTSmain 方法用于整个 MCTS 过程，该方法通过迭代选择、扩展、模拟和反向传播过程，最终得出最佳的下棋位置。其中时间限制为 1 分钟。
- Select 方法用于选择过程，从根节点 R 开始，递归选择子节点，直至到达叶节点或到达具有还未被扩展过的子节点的节点 L。在游戏结束前，如果当前节点有未拓展的子节点，则随机拓展一个节点；如果当前节点没有未拓展的子节点，则使用 UCB1 算法来计算每个子节点的分值，并选择分值最高的子节点进行拓展。如果当前节点没有子节点，创建一个占位虚拟节点并继续搜索。
- Simulate 方法用于模拟过程，该方法采用随机落子的方式模拟游戏进程；

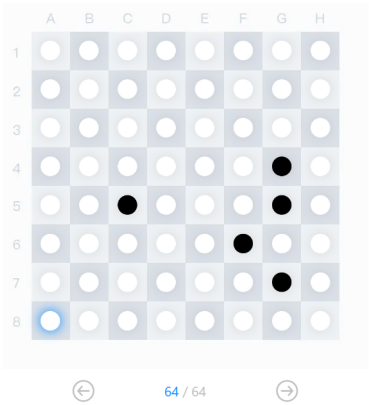
- Backpropagation 方法用于反向传播过程，该方法从叶子结点开始，将访问次数加 1，并根据游戏结果更新节点的分数；

4 实验结果

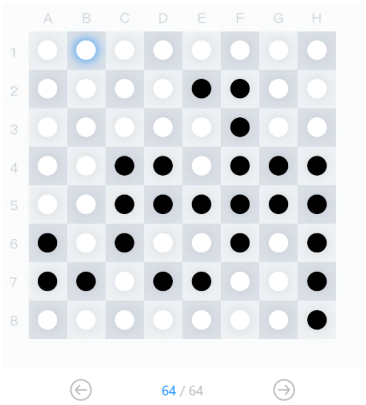
棋局对弈结果：

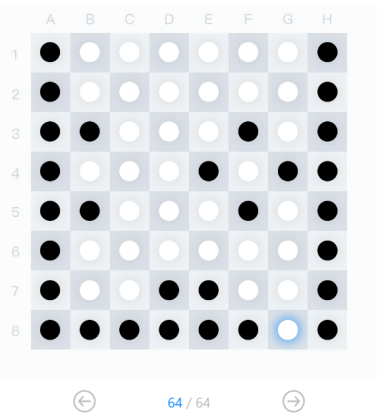
初级：



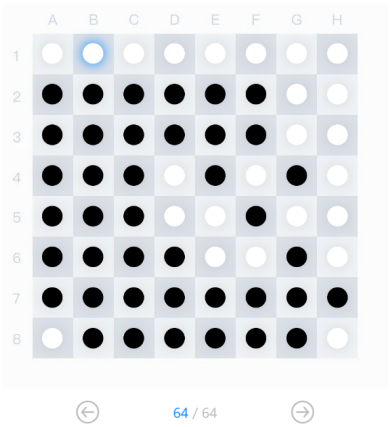


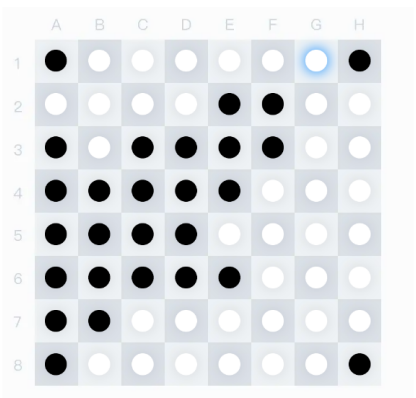
中级：





高级：





5 总结

该实现基本达到了预期的目标。仍存在一些可以优化的方向，比如：使用并行化算法来加速搜索；使用动态调整策略来平衡搜索和模拟的时间成本等。