实验报告

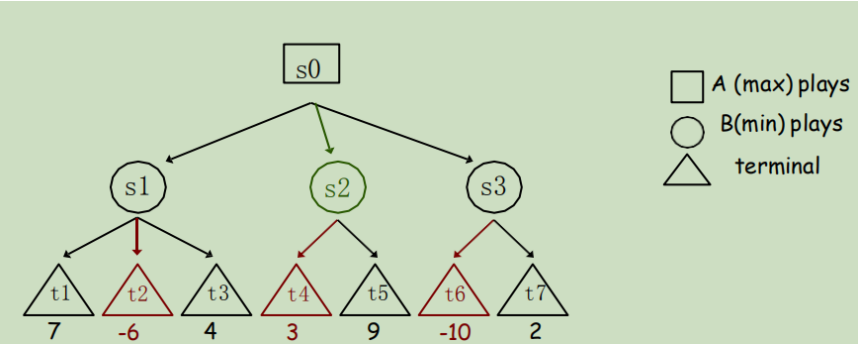
学院：数据科学与计算机学院　　专业：计算机科学与技术　　科目：人工智能
学号：17341213　　姓名：郑康泽

博弈树搜索

一． 算法原理

1. 博弈树

在博弈树中，有两个玩家，其中先手的玩家称为 MAX，后手的玩家称为 MIN，两个玩家交替行动，直至达到某个终止状态。博弈树的每一层中的每个节点代表其中一个玩家的当前状态，每个分支代表一个玩家做出的动作，在博弈树的叶节点可以映射到一个实数，该实数代表对先手玩家 MAX 的效益，该数字越大表明对玩家 MAX 越有利，而对玩家 MIN 越不利。所以在博弈树搜索中，MAX 玩家会尽量走到效益大的叶节点，而 MIN 玩家会尽量走到效益小甚至是负效益的叶节点。例子如下（最下面的数字是对于玩家 MAX 的收益）：

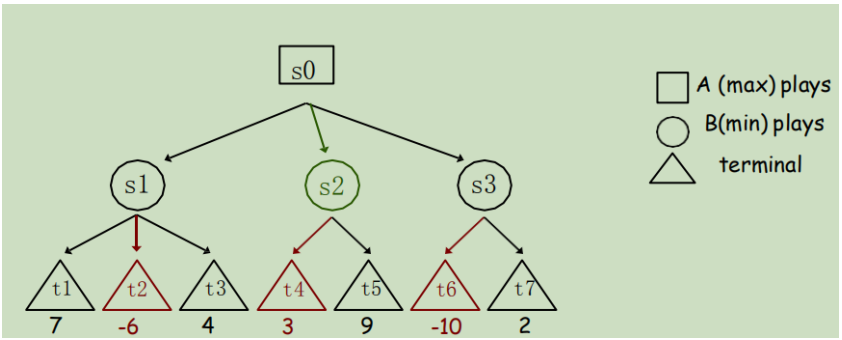


2. MiniMax 策略

假设对方总是能做出最优的行动，即己方总是能做出最小化对方获得的收益的行动，通过最小化对方的策略，可以最大化己方的利益。该策略的步骤如下：

1） 构建完整的博弈树（每个叶子节点都表示终止状态）
　A） 根节点表示起始状态，边表示可能的行动
　B） 每个叶子节点（终止状态）都标记了对应的效益值
2） 返向传播效益$U(n)$
　A） 每个叶子节点t的$U(t)$值都是预定义好的
　B） 假如节点n是一个 MIN 节点，则$U(n) = \min\{U(c): c是n的子节点\}$
　C） 假如节点n是一个 MAX 节点，则$U(n) = \max\{U(c): c是n的子节点\}$

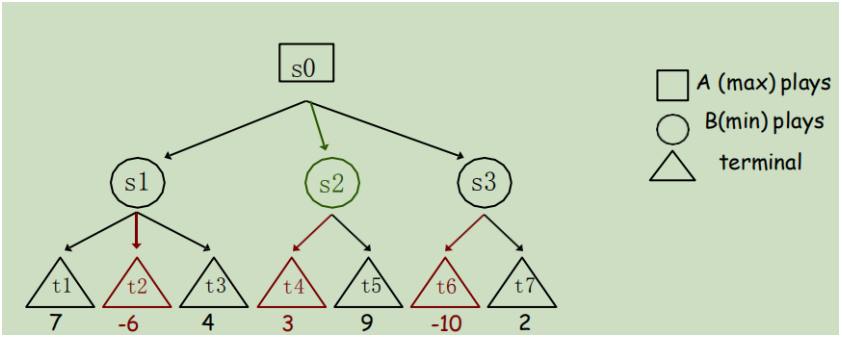
再来看看这棵博弈树：



对于$s1$这个 MIN 节点，它会选择具有最小效益值的子节点，即它返回给父节点$s0$的效益值为-6，同理$s2$、$s3$返回的效益值为 3、-10。那么再来看看根结点$s0$，它是一个 MAX 节点，那么他会选择具有最大效益值的子节点，所以它会选择$s2$子节点。所以如果 MIN 玩家和 MAX 玩家在上面这个博弈树上使用 MiniMax 策略运行搜索的话，那么他们所走的路径应该是$s0 \rightarrow s2 \rightarrow t4$，最后显然是 MAX 玩家收益了。

3. Alpha Beta 剪枝

还是上面那个例子：



可以看出，其实搜索完$t6$之后就不用搜索$t7$了，即$s3 \rightarrow t7$这条分支可以被剪枝掉，因为遍历完$s2$后，$s0$就可以确定至少可以获得 3 收益值，而$s3$搜完子节点$t6$就可以确定知道至多返回-10 收益值，那么$s0$肯定不会选择$s3$子节点，那么$s3$也不用搜索子节点$t7$了。所以在博弈树搜索中，可以利用剪枝，降低时间复杂度，以缩短搜索时间，这是非常有必要的，因为在树结构中，随着层数的增加，深搜的时间复杂度是以指数倍增加的。对于博弈树搜索中，因为有两种玩家身份，所以对于不同身份的玩家，就有两种不同的剪枝，具体如下：

1) 对 MAX 节点的剪枝$(\alpha - cuts)$：
   A) 在 MAX 节点n
   B) 设β是n被遍历过的兄弟节点中的最低值
   C) 设α是n被遍历过的子节点中的最高值
   D) 当$\alpha \geq \beta$时，可以停止遍历n子节点
2) 对 MIN 节点的剪枝$(\beta - cuts)$：
   A) 在 Min 节点n
   B) 设α是n被遍历过的兄弟节点中的最高值
   C) 设β是n被遍历过的子节点中的最低值
   D) 当$\alpha \geq \beta$时，可以停止遍历n子节点

二. 伪代码

1. MiniMax 策略

```
DFMiniMax(n, Player)
input: n is the current state
       Player is MAX or MIN
return: utility of state n

if n is TERMINAL:
      return V(n)
```

```
childlist = n.Successors(Player)
 if Player == MAX:
        return minimum of DFMiniMax(c, MIN) over c ∈ childlist
 else:
        return maximum of DFMiniMax(c, MAX) over c ∈ childlist
```

2. Alpha Beta 剪枝

```
AlphaBeta(n, Player, alpha, beta)
input: n is the current state
        Player is MAX or MIN
        alpha is from ancestor
        beta is from ancestor
return: utility of state n

if n is TERMINAL:
        return V(n)

cur_alpha = -inf
cur_beta = inf
childlist = n.successors(Player)
if Player == MAX:
        for c in childlist:
                cur_alpha = max(cur_alpha, AlphaBeta(c, MIN, alpha, beta))
                if cur_alpha ≥ beta:
                        break
                alpha = max(alpha, cur_alpha)
        return cur_alpha
else:
        for c in childlist:
                cur_beta = min(cur_beta, AlphaBeta(c, MAX, alpha, beta))
                if alpha ≥ cur_beta:
                        break
                beta = min(beta, cur_beta)
        return cur_beta
```

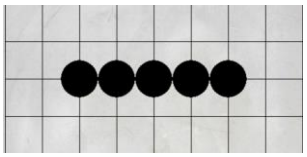
三. 评价函数及搜索策略

1. 评价函数

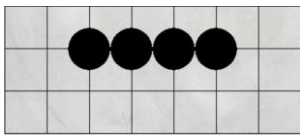
参考了网上许多份实现五子棋 AI 的代码，大家的想法都是一致的，都是从棋型出发，去评估黑子和白子的得分，将所有棋型的得分加在一起就是总得分，所以 AI 的智能程度取决于所判断的棋型以及所获得的分数的科学性。当然还有许多可以优化的地方，比如如果两种可以得分的棋型如果交叉在一起，得分应该更高，例如两个活三的棋型交叉在一起，这时候应该有更高的加分，因为此时已经可以判定胜利了。因为我将人作为 MAX 玩家，将 AI 作为 MIN 玩家，所以我的评估函数是：人的得分-AI 的得分，后来在跟它

玩的时候发现它有时候喜欢去攻击，而不是去防守。通过与同学的讨论，我得到了解决方案，就对 AI 的得分乘以一个 0 到 1 之间系数，弱化它对得分的欲望，强化它对防守的欲望，因为防守可以使评估函数的值更容易变小。主要思路就是这样，但棋型的设置以及分数的设置都十分具有技巧性，设置的好，根本玩不过 AI。

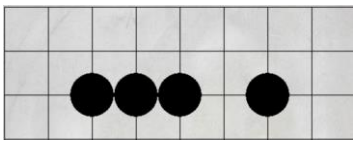
1) 得分最高的棋型当然就是连五了，这都已经赢了，可以将连五的棋型对应的得分设为最大值；

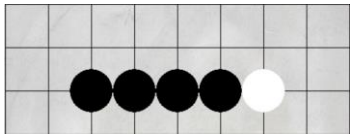


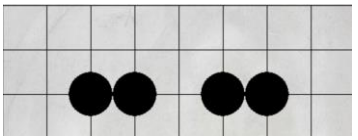
2) 活四的得分应该跟连五一样，因为活四一出现就已经决定胜负了；



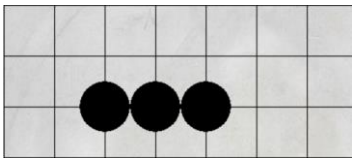
3) 冲四的得分应该低于活四，应该冲四是可以防住的；

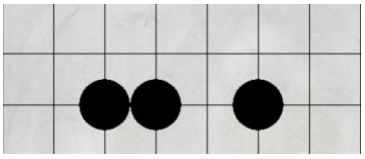




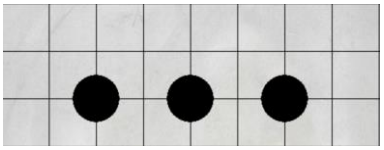


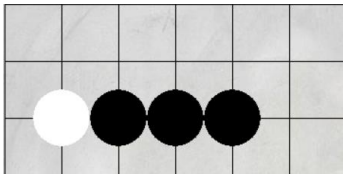
4) 活三也是极具有威胁的，如果不防，就会形成活四，那就防不住，所以得分也较高；

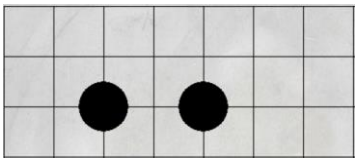


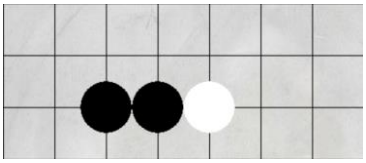


5) 相对来讲，眠三、活二以及眠二的威胁就较少，在一定情况下，可防可不防；









2. 搜索策略

每次选择下子的位置附近肯定是有子，不可能一开始下的时候就在边角的地方下，这样肯定是输的。所以每次深搜，选定的位置附近一定是有子的，没有子的位置直接跳过，不搜这个位置，这样也可以减少许多时间去搜索。那么如何定义附近呢？我一开始设置四个方向上距离为一的位置上没有子，则定义该位置为孤立的，那么就不去搜它，后来尝试设置在四个方向上、距离在二以内的位置没有子才算孤立，发现这样的效果更好，尽管可能会花费更多的时间去搜索，毕竟搜索的范围变大了。至于搜索深度，我设置的是三层，因为三层的搜索时间是可以接受的，四层的搜索时间无法接受，还有一个方面是三层的搜索我已经完全打不过了。

四． 代码解释

1. 设计一个 AI 类

1） 初始化函数：初始化 AI 记录信息以及初始化棋局信息

```
class UndefeatedAI:
    def __init__(self, man_first):
        '''
        :param man_first: True -> 人先手, False -> 人后手
        '''
        self.size = N                                   # 棋盘大小
        self.man_first = man_first                      # 是否人先手
        self.max_depth = 2                              # 搜索深度为 self.max_depth + 1
        self.all_chess = set()                          # 棋盘上所有位置
        self.chess_list = [[0] * N for _ in range(N)]   # 记录棋盘上下的子的情况
        self.shape2score = [((0, 1, 1, 0, 0), 50),      # 各个形状及其得分
                            ((0, 0, 1, 1, 0), 50),
                            ((1, 1, 0, 1, 0), 200),
                            ((0, 0, 1, 1, 1), 500),
                            ((1, 1, 1, 0, 0), 500),
                            ((0, 1, 1, 1, 0), 5000),
                            ((0, 1, 0, 1, 1, 0), 5000),
                            ((0, 1, 1, 0, 1, 0), 5000),
                            ((1, 1, 1, 0, 1), 5000),
                            ((1, 1, 0, 1, 1), 5000),
                            ((1, 0, 1, 1, 1), 5000),
                            ((1, 1, 1, 1, 0), 5000),
```

```
                    ((0, 1, 1, 1, 1), 5000),
                    ((0, 1, 1, 1, 1, 0), 50000),
                    ((1, 1, 1, 1, 1), 99999999)]

    # 初始化棋局
    if not man_first:                           # AI先手
        self.ai_chess = {(5, 5), (5, 6)}
        self.man_chess = {(5, 4), (6, 5)}
    else:
        self.ai_chess = {(5, 4), (6, 5)}
        self.man_chess = {(5, 5), (5, 6)}

    for i in range(self.size):
        for j in range(self.size):
            self.all_chess.add((i, j))
```

```
    # 1 代表先手即黑色子，2 代表后手即白子
    for each in self.ai_chess:
        if man_first:
            self.chess_list[each[0]][each[1]] = 2
        else:
            self.chess_list[each[0]][each[1]] = 1

    for each in self.man_chess:
        if man_first:
            self.chess_list[each[0]][each[1]] = 1
        else:
            self.chess_list[each[0]][each[1]] = 2
```

2) 打印棋盘：通过命令行的输出当前棋局情况（已弃用）

```
def print_board(self):
    '''
    ###弃用###
    输出当前棋局情况
    '''
    board = [[' ' for j in range(self.size)] for i in range(self.size)]

    if self.man_first:
        for pos in self.man_chess:
            board[pos[0]][pos[1]] = 'x'
        for pos in self.ai_chess:
            board[pos[0]][pos[1]] = 'o'
    else:
        for pos in self.man_chess:
            board[pos[0]][pos[1]] = 'o'
        for pos in self.ai_chess:
            board[pos[0]][pos[1]] = 'x'
```

```
    for i in range(self.size):
        for j in range(self.size):
            print('----', end='')
        print('-')
        for j in range(self.size):
            print('| {} '.format(board[i][j]), end='')
        print('|', i)
    for j in range(self.size):
        print('----', end='')
    print('-')
    for j in range(self.size):
        print('  {} '.format(j), end='')
    print()
    print()
```

3) 检查是否已分出胜负：检查所下子的四个方向

```
    def check(self, is_man, pos):
        '''
        :param is_man: True -> 人, False -> AI
        :param pos: 人或AI下的子的位置
        :return: True -> 获得胜利, False -> 没获得胜利
        '''
        # 获取该方的下的子
        if is_man:
            chess_list = self.man_chess
        else:
            chess_list = self.ai_chess

        # 检查四条直线的方向
        dirs = ((1, 0), (0, 1), (1, 1), (1, -1))
        for dir in dirs:
            chess_num1 = 0
            chess_num2 = 0
            # 一条直线的一个方向
```

```
            # 一条直线的一个方向
            _pos = pos
            while ((_pos[0] - dir[0]), (_pos[1] - dir[1])) in chess_list:
                chess_num1 += 1
                _pos = (_pos[0] - dir[0]), (_pos[1] - dir[1])
            # 一条直线的另一个方向
            _pos = pos
            while ((_pos[0] + dir[0]), (_pos[1] + dir[1])) in chess_list:
                chess_num2 += 1
                _pos = (_pos[0] + dir[0]), (_pos[1] + dir[1])
            # 一条直线上有连续的4个子（除了当前下的）即为胜利
            if chess_num1 + chess_num2 >= 4:
                return True
        return False
```

4) 获取当前棋盘上的空余位置：

```
    def get_empty_pos(self):
        '''
        获取当前的空位置
        '''
        return self.all_chess - self.ai_chess - self.man_chess
```

5) 检查该位置是否属于孤立点：通过检查该位置四个方向上距离在 2 以内的位置是否有子，一共是 16 个位置，方便之后的搜索

```
    def solitary(self, pos):
        '''
        :param pos: 在棋盘上的位置
        :return: True -> 孤立（直线距离2以内没子）, False -> 不孤立
        '''
        # 16个邻居
        dirs = ((-1, -1), (-1, 0), (0, -1), (1, 1), (1, 0), (0, 1), (1, -1), (-1, 1),
                (-2, -2), (-2, 0), (0, -2), (2, 2), (2, 0), (0, 2), (2, -2), (-2, 2))
        for dir in dirs:
            neighbor = (pos[0] + dir[0], pos[1] + dir[1])
            if (0 <= neighbor[0] < self.size and 0 <= neighbor[1] < self.size and
                (neighbor in self.ai_chess or neighbor in self.man_chess)):
                return False
        return True
```

6) 计算一个位置在一个方向上的最大得分：每次取该位置该方向上相邻的 5 个或 6 个位置的信息，来计算在该方向上得分，最后取最大值作为代表该位置在该方向上的最大得分

```python
def cal_one_pos(self, is_man, pos, dir, chess_dir_score):
    '''
    :param is_man: True -> 人, False -> AI
    :param pos: 人或AI下的子的位置
    :param dir: 要算该方向上的得分
    :param chess_dir_score: 形式为(mpos, dir, score)
    :return: 人或AI在pos下子并在dir方向的得分
    '''
    # 获得我方下的子和对方下的子
    if is_man:
        my_chess = self.man_chess
        opponent_chess = self.ai_chess
    else:
        my_chess = self.ai_chess
        opponent_chess = self.man_chess
```

```python
    # 该位置以及该方向是否已经计算过分数，若是直接返回0，避免重复加分
    for mpos, _dir, _ in chess_dir_score:
        for _pos in mpos:
            if _pos == pos and _dir == dir:
                return 0

    pos_dir_max_score = [None, None, 0]     # 定义该位置以及该方向的最大得分情况(mpos, dir, score)
    plus = 0                                # 得分形状重合可以加分

    # 起始点为(pos[0] + i * dir[0], pos[1] + i * dir[1])
    for i in range(-5, 1):
        shape = []
        # 每次取6个子
        for j in range(6):
            px = pos[0] + (i + j) * dir[0]
            py = pos[1] + (i + j) * dir[1]
            if 0 <= px < self.size and 0 <= py < self.size:
                if (px, py) in my_chess:
                    shape.append(1)
                elif (px, py) in opponent_chess:
                    shape.append(2)
```

```python
                else:
                    shape.append(0)
            else:
                continue

        if len(shape) < 5:          # 不足以凑够5个子
            continue
        elif len(shape) == 5:       # 不足以凑够6个子
            shape1 = tuple(shape)
            shape2 = None
        else:                       # 能凑够6个子
            shape1 = tuple(shape[:5])
            shape2 = tuple(shape)
```

```python
        for _shape, _score in self.shape2score:
            if _shape == shape1 or _shape == shape2:        # 形状是否可以得分
                if _score > pos_dir_max_score[2]:           # 更新该方向的最大得分情况
                    mpos = []                               # 能够得分的形状中包括的子的位置
                    for j in range(5):
                        mpos.append((pos[0] + (i + j) * dir[0], pos[1] + (i + j) * dir[1]))
                    pos_dir_max_score = [tuple(mpos), dir, _score]

    if pos_dir_max_score[2] != 0:
        # 得分形状相交可以加分
        for mpos1, _, score in chess_dir_score:
            mpos1 = set(mpos1)
            mpos2 = set(pos_dir_max_score[0])
            if mpos1.intersection(mpos2) != set() and pos_dir_max_score[2] > 10 and score > 10:
                plus += (score + pos_dir_max_score[2]) // 2
        # 记录在该位置下子在该方向上能获取的最大得分
        chess_dir_score.append(pos_dir_max_score)

    # 返回得分情况
    return pos_dir_max_score[2] + plus
```

7） 评估函数：计算人下的每个子在每个方向上的得分和S_1，同理计算AI下的每个子在

每个方向上的得分和S_2，最后当前棋局的得分为$S_1 - S_2 * ratio$, 其中$0 < ratio < 1$。乘上这个比例是为了使得AI更注重于防守，而不是去进攻

```
def estimate(self):
    '''
    :return: 当前棋局得分为 人的得分 - Ai的得分 * ratio (0<ratio<1), ratio的意义在于让AI更注重防守
    '''
    # 每个子在每个方向上最大得分和
    ai_score = 0
    man_score = 0
    # 记录每个子在每个方向上的最大得分
    ai_chess_dir_score = []
    man_chess_dir_score = []
    # 四个方向
    dirs = ((1, 1), (1, 0), (0, 1), (1, -1))
    for pos in self.ai_chess:
        for dir in dirs:
            ai_score += self.cal_one_pos(False, pos, dir, ai_chess_dir_score)
    for pos in self.man_chess:
        for dir in dirs:
            man_score += self.cal_one_pos(True, pos, dir, man_chess_dir_score)

    return man_score - ai_score * 0.3
```

8） 博弈树搜索：利用$Alpha\ Beta$剪枝降低时间复杂度

```
def dfs_minimax(self, is_man, opponent_act, cur_depth, alpha, beta):
    '''
    :param is_man: True -> 人 极大节点, False -> AI 极小节点
    :param opponent_act: 对方上一步下的子的位置
    :param cur_depth: 当前搜索的深度: self.max_depth - cur_depth + 1
    :param alpha: 极大祖先节点的alpha
    :param beta: 极小祖先接地那的beta
    :return: 对于极大节点，返回最大得分以及最佳的下子位置，对于极小节点，返回最小得分以及最佳的下子位置
    '''
    # 如果当前棋局已经分出胜负或者深度已到，则停止深搜
    if self.check(not is_man, opponent_act) or cur_depth == 0:
        return self.estimate(), None
```

```
    # 获得空位置
    possible_pos = self.get_empty_pos()
    # 最佳的下子位置
    best_act = None
    # 当前极大节点的alpha或者当前极小节点的beta
    cur_alpha = float('-inf')
    cur_beta = float('inf')

    for each_pos in possible_pos:
        if self.solitary(each_pos):         # 无邻居，不考虑
            continue

        if is_man:
            self.man_chess.add(each_pos)
        else:
            self.ai_chess.add(each_pos)

        value, _ = self.dfs_minimax(not is_man, each_pos, cur_depth-1, alpha, beta)     # 下这一步获得的最有利的分数
```

```
        if is_man:
            self.man_chess.remove(each_pos)
        else:
            self.ai_chess.remove(each_pos)

        # 极大节点
        if is_man:
            # 更新当前alpha
            if cur_alpha < value:
                cur_alpha = value
                best_act = each_pos
            if cur_alpha > alpha:
                alpha = cur_alpha
            # 剪枝
            if alpha >= beta:
                return alpha, best_act
```

```
            # 极小节点
            else:
                # 更新当前beta
                if cur_beta > value:
                    cur_beta = value
                    best_act = each_pos
                if cur_beta < beta:
                    beta = cur_beta
                # 剪枝
                if alpha >= beta:
                    return beta, best_act

        if is_man:
            return cur_alpha, best_act
        else:
            return cur_beta, best_act
```

9) 手动输入坐标下棋：检查输入的合法性（已弃用）

```
    def get_input(self):
        '''
        ###弃用####
        :return: 返回人输入要下子的坐标，需要检查
        '''
        while True:
            # 要求输入的格式为 'x y'
            try:
                x, y = input('输入要下的位置，格式为：行 列（从0开始算）\n').split()
            except ValueError:
                print('输入错误，请重新输入\n')
            else:
                # 要求输入是数字
                try:
                    x = int(x)
                    y = int(y)
                except ValueError:
                    print('输入非数字，请重新输入\n')
```

```
            else:
                # 要求输入的坐标合法
                if (0 <= x < self.size and 0 <= y < self.size and (x, y) not in self.man_chess
                        and (x, y) not in self.ai_chess):
                    return x, y
                else:
                    print('越界或该位置上已有子')
```

10) 运行游戏：由于利用了已弃用的两个函数*print_board*和*get_input*，所以该函数也已弃用：

```
    def play(self):
        '''
        ###弃用####
        '''
        # 画出初始棋局
        self.print_board()

        # 人先手则获取要下的位置
        if self.man_first:
            (x, y) = self.get_input()
            self.man_chess.add((x, y))
            self.print_board()
        # AI先手则将初始棋局的后手的一个子作为人下的子
        else:
            (x, y) = (5, 4)
```

```
        while True:
            # AI获取最有利的下子位置
            _, pos = self.dfs_minimax(False, (x, y), self.max_depth, float('-inf'), float('inf'))
            print('ai下的位置: ', pos)
            self.ai_chess.add(pos)
            self.print_board()
            # 检查是否胜负分明
            if self.check(False, pos):
                print('你输了')
                return

            x, y = self.get_input()
            self.man_chess.add((x, y))
            self.print_board()
            # 检查是否胜负分明
            if self.check(True, (x, y)):
                print('你赢了')
                return
```

11) 获取 AI 下子的位置：输入是人下子的位置

```
def get_ai_action(self, man_act):
    '''
    :param man_act: 人下子的位置
    :return: AI下子的位置
    '''
    # 通过搜索获得最有利的下子位置
    value, pos = self.dfs_minimax(False, man_act, self.max_depth, float('-inf'), float('inf'))
    # 更新
    self.ai_chess.add(pos)
    if self.man_first:
        self.chess_list[pos[0]][pos[1]] = 2
    else:
        self.chess_list[pos[0]][pos[1]] = 1
    return pos
```

12) 输入人下子的位置，更新 AI 记录的信息：

```
def input_man_act(self, pos):
    '''
    :param pos: 人下子的位置
    '''
    # 更新
    self.man_chess.add(pos)
    if self.man_first:
        self.chess_list[pos[0]][pos[1]] = 1
    else:
        self.chess_list[pos[0]][pos[1]] = 2
```

13) 获取当前棋局信息：黑子和白子的位置

```
def get_chess_list(self):
    '''
    :return: 当前棋局的下子情况
    '''
    return self.chess_list
```

2. 通过 Pygame 进行 UI 设计

1) 画棋盘：画网格及棋子，对于 AI 最近一步棋的颜色进行特殊处理，方便辨认

```
def show_board(screen, bg_img, chess_list, last_step=(None, None)):
    '''
    :param screen: pygame.surface
    :param bg_img: 背景图
    :param chess_list: 当前棋局的下子情况
    :param last_step: (a, b) denotes (is man or not, last action)
    '''
    # 贴背景
    screen.blit(bg_img, (0, 0))
    # 画线
    for i in range(N):
        pygame.draw.line(screen, BLACK, (Space, Space + i * Cell_Size), (Grid_Size - Space, Space + i * Cell_Size))
        pygame.draw.line(screen, BLACK, (Space + i * Cell_Size, Space), (Space + i * Cell_Size, Grid_Size - Space))
```

```
    # 画棋
    for i in range(N):
        for j in range(N):
            if chess_list[i][j] == 0 or (i, j) == last_step:
                continue
            pos = (Space + Cell_Size * j, Space + Cell_Size * i)
            if chess_list[i][j] == 1:
                pygame.draw.circle(screen, BLACK, pos, Cell_Size // 2)
            else:
                pygame.draw.circle(screen, White, pos, Cell_Size // 2)
```

```
    # AI上一步棋特别标志 灰色标志
    if last_step[0] is not None and last_step[1] is not None:
        pos = (Space + Cell_Size * last_step[1][1], Space + Cell_Size * last_step[1][0])
        if not last_step[0]:
            # if chess_list[last_step[1][0]][last_step[1][1]] == 1:
            #     pygame.draw.circle(screen, (100, 100, 100), pos, Cell_Size // 2)
            # else:
            pygame.draw.circle(screen, (200, 200, 200), pos, Cell_Size // 2)
        else:
            if chess_list[last_step[1][0]][last_step[1][1]] == 1:
                pygame.draw.circle(screen, BLACK, pos, Cell_Size // 2)
            else:
                pygame.draw.circle(screen, White, pos, Cell_Size // 2)
```

2) 显示信息：在 UI 上显示一些必要的信息，比如谁赢了

```
def display_text(screen, text, pos, size, color, font_name):
    '''
    :param screen: pygame.surface
    :param text: 要显示的文字
    :param pos: 显示文字的中心位置
    :param size: 文字大小
    :param color: 文字颜色
    :param font_name: 文字字体
    '''
    font = pygame.font.Font(font_name, size)
    tx_surf = font.render(text, True, color)
    tx_rect = tx_surf.get_rect()
    tx_rect.midtop = pos
    screen.blit(tx_surf, tx_rect)
```

3. 主函数

```
if __name__ == '__main__':
    # 初始化pygame
    pygame.init()
    pygame.display.set_caption('五子棋')
    screen = pygame.display.set_mode((Grid_Size, Grid_Size))
    bg_img = pygame.image.load('background.jpg').convert()
    font_name = pygame.font.get_default_font()
```

```
    # 显示提示信息
    show_board(screen, bg_img, [[0] * 11 for _ in range(11)])
    display_text(screen, 'Press \'Y\' to be on the offensive...',
                 (Grid_Size // 2, Grid_Size // 2), 30, (0, 0, 255), font_name)
    display_text(screen, 'Press others to be on the defensive...',
                 (Grid_Size // 2, Grid_Size // 2 + 30), 30, (0, 0, 255), font_name)
    pygame.display.flip()
```

```
# 按Y表示人先手，否则人后手
man_first = False
have_chosen = False
while not have_chosen:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.QUIT
        elif event.type == pygame.KEYUP:
            if event.key == pygame.K_y:
                man_first = 1
            have_chosen = True

game = UndefeatedAI(man_first)          # 实例化一个AI
```

```
# AI先手则将初始棋局的后手的一个子作为人下的子
action = None
if not man_first:
    action = game.get_ai_action((5, 4))
chess_list = game.get_chess_list()
show_board(screen, bg_img, chess_list, (False, action))
pygame.display.flip()
```

```
# 交互开始
end = False
win_or_lose = None
while not end:
    # 通过UI交互获取人要下子的位置
    have_chosen = False
    pos = None
    while not have_chosen:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.QUIT
            elif event.type == pygame.MOUSEBUTTONDOWN:
                pos = (round((event.pos[1] - Space) / Cell_Size), round((event.pos[0] - Space) / Cell_Size))
                # print(pos)
                # 检查合法性
                if not (0 <= pos[0] < N) or not (0 <= pos[1] < N) or not (pos in game.get_empty_pos()):
                    break
```

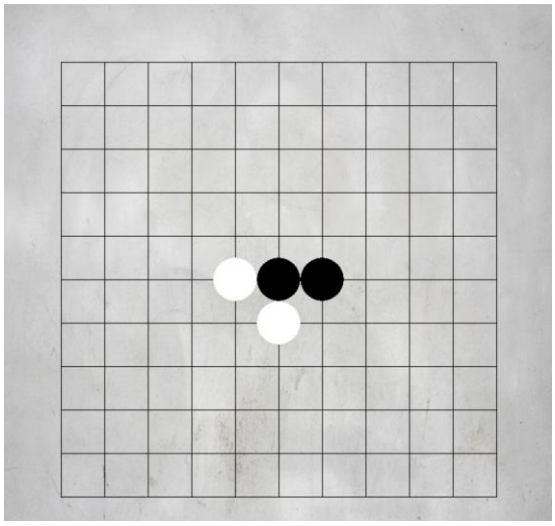
```
                # 更新棋局
                game.input_man_act(pos)
                show_board(screen, bg_img, chess_list, (True, pos))
                pygame.display.flip()
                have_chosen = True
                # 检查是否胜负分明
                if game.check(True, pos):
                    win_or_lose = True
                    end = True
```

```
    if not end:
        # AI下子并更新棋局
        action = game.get_ai_action(pos)
        show_board(screen, bg_img, chess_list, (False, action))
        pygame.display.flip()
        # 检查是否胜负分明
        if game.check(False, action):
            win_or_lose = False
            end = True

# 显示谁赢了
if win_or_lose:
    display_text(screen, 'You win!', (Grid_Size // 2, Grid_Size // 2), 80, (255, 0, 0), font_name)
else:
    display_text(screen, 'You lose!', (Grid_Size // 2, Grid_Size // 2), 80, (255, 0, 0), font_name)
display_text(screen, 'Press any key to quit...', (Grid_Size // 2, Grid_Size // 2 + 80), 20, (0, 0, 255), font_name)
pygame.display.flip()
```

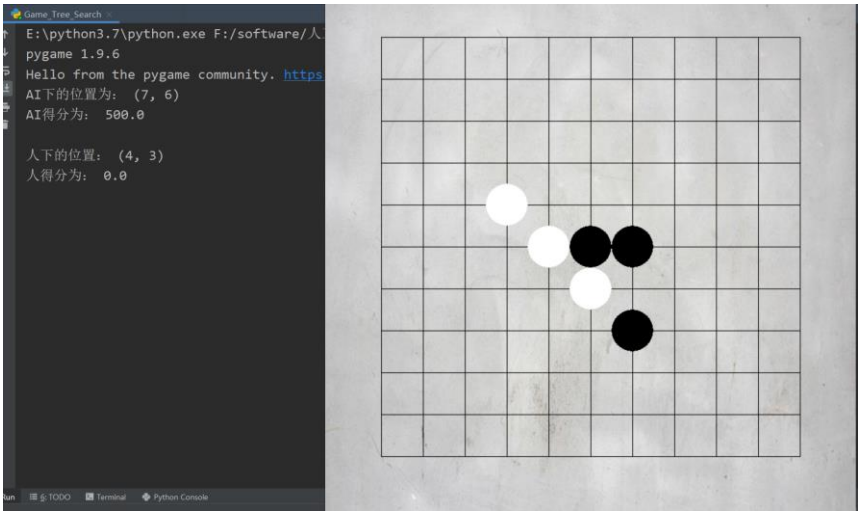
五． 实验结果展示

1. 初始界面

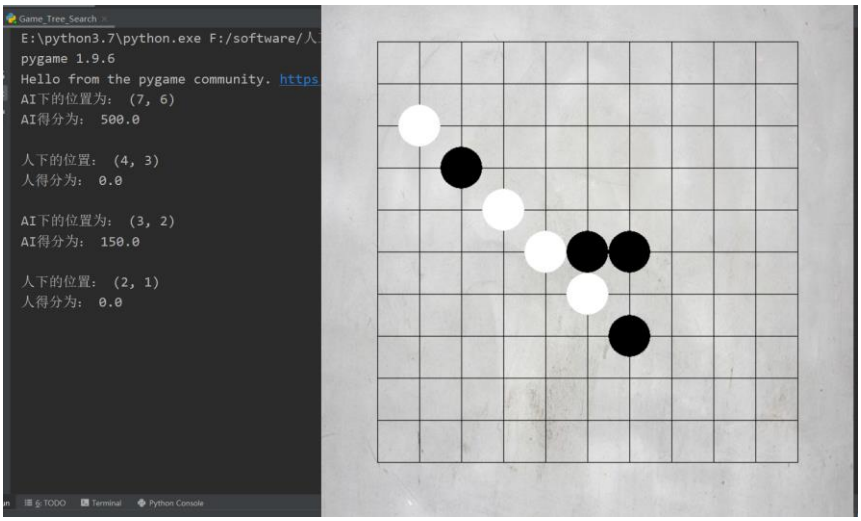


2. **AI 先手（AI 为黑子，人为白子）**

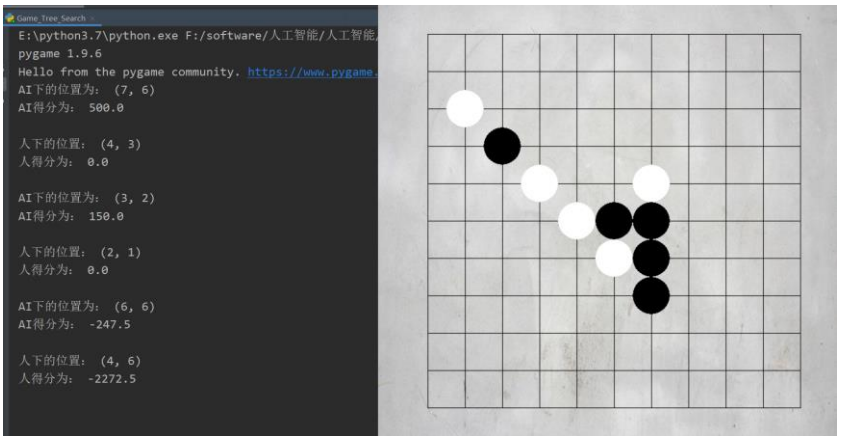
1) 第一回合：



2) 第二回合：

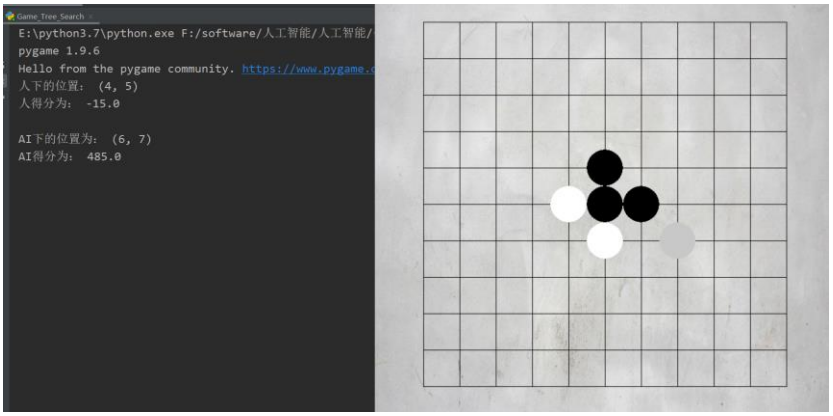


3) 第三回合：

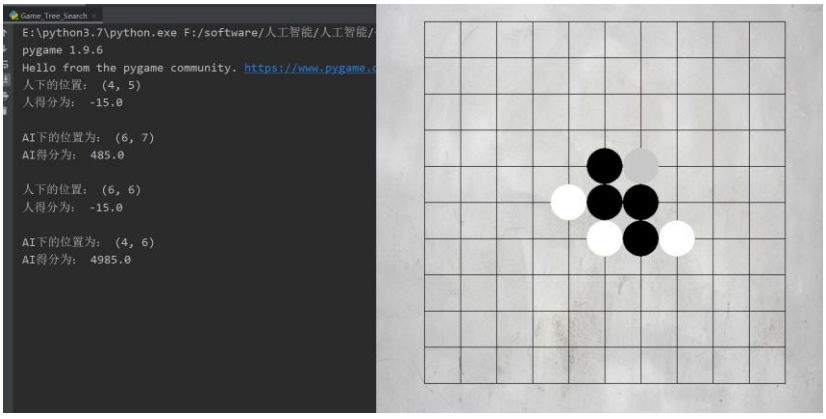


3. 人先手（人为黑子，AI 为白子或灰子）

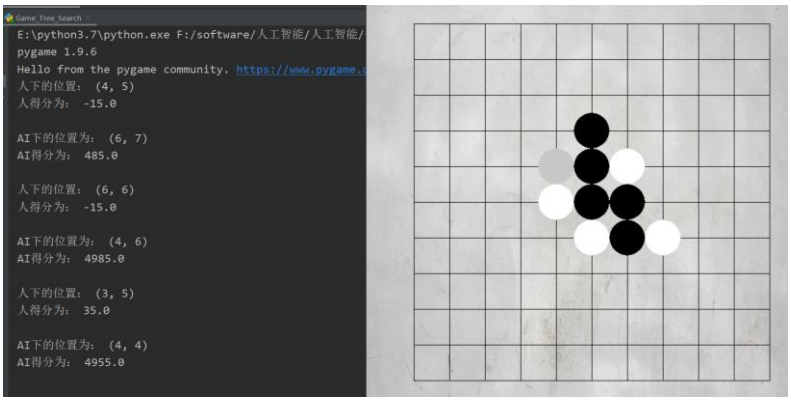
1） 第一回合：



2） 第二回合：



3） 第三回合：



4. 人先手（人为黑子，AI 为白子或灰子）