中山大学计算机学院本科生实验报告

(2020 秋季学期)

课程名称：人工智能 **任课老师：**饶洋辉

年级 + 班级	18 级计算机	年级 (方向)	计算机科学
学号	18340236	姓名	朱煜
Email	zhuy85@mail2.sysu.edu.cn	完成日期	2020.12.09

实验八 博弈树搜索

1 实验原理

1.1 博弈树

博弈树本质上是一种类似于树型的数据结构，如图1，主要作用是把博弈问题转换成为对生成的博弈树的搜索问题，进而为求解博弈问题。对于可以在有限步数中分出胜负的双方博弈问题，通过对叶子节点各方优势的评估，将结果逐层向上返回，最终就会得到一系列的分支选择，也就是想要求得的解。

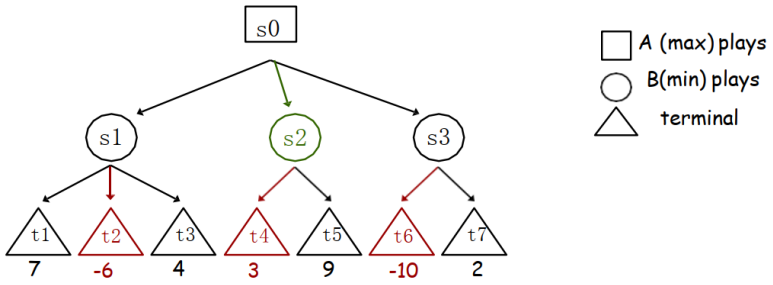


图 1: 博弈树

以实验中实现的五子棋博弈为例，五子棋由于属于双人完备信息的博弈，需要对弈双方轮流落子，最上面的根节点，就是当前局面，作为博弈树的初始节点，博弈树的叶子节点，代表的是结束的棋局最终局面，该局面可以是分出胜负的局面，也可以是在一定限制下到达的极限局面（如平局）。在博弈树中，每个节点代表各个局面，连接节点的线就相当于落子，每一层代表的玩家根据评估来改变局势。

1.2 MiniMax

极大极小算法（MiniMax 算法）是一种找到失败的最大可能性中的最小值的算法，也就是基于对方的最优选择来最小化对手的收益，通常通过递归来进行实现。在图1中，B(MIN 玩家) 在 s_1, s_2, s_3 会分别选择 t_2, t_4, t_6 来最小化收益，而 A(MAX 玩家) 在 $s0$ 会选择 s_2 来最大化收益。

以实验中实现的五子棋博弈为例，就是在有限的搜索深度之内选择最小化对手收益的行动来行棋。五子棋博弈过程化成一颗博弈树，首先将博弈的双方表示为 MAX 和 MIN，实验中以 MAX 代表玩家，MIN 代表 AI。然后通过评价方法 G (P) 对当前棋局 P 的形式进行优劣评估，若当前棋局对玩家有利，G(P) 为正; 若当前局势对敌方有利，G (P) 为负。当玩家走步时，需要选对己方有利的局面去走，也就是选择 G (P) 大的节点走。敌方行动时，正好相反。

1.3 Alpha-beta 剪枝

Alpha-Beta 剪枝用于解决极大极小算法中的数据冗余问题。该算法利用深度优先遍历的剪枝原理，使得不必将博弈树完全展开以提高搜索效率。

Alpha-Beta 剪枝涉及到 Alpha 和 Beta 两个参数，并将这两个参数作为极大极小算法参数值，搜索开始时，Alpha 表示 MAX 收益最低的局面，初始值设为负无穷，而 Beta 的值则表示 MIN 收益最低的值，初始值设为正无穷。在深度优先遍历的过程中，在 MAX 节点中，如果 Alpha 的值小于节点的估值，则就调整 Alpha; 在 MIN 节点中，如果 Beta 大于节点值，则调整 Beta 逐步递减。

当出现 MAX 节点如果 Alpha 值大于任何祖先 MIN 节点的 Beta 值，就进行 alpha 剪枝；MIN 节点如果 Beta 值小于任何祖先 Max 节点的 Alpha 值，就进行 Beta 剪枝。剪枝过程如图2。

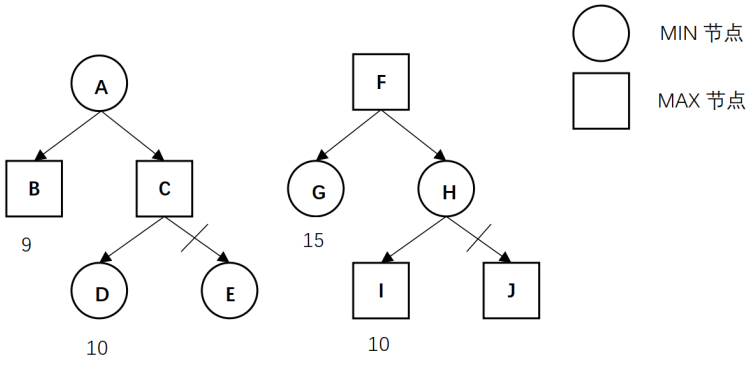


图 2: Alpha-beta 剪枝

探索 B 节点后，此时 A 的 β 值为 9，而之后探索 D 节点，可得到 C 节点的 α 值大于父节点 A 的 β 值，发生 Alpha 剪枝，此时 E 节点被剪枝掉。同理，探索 G 节点后，此时 F 的 α 值为 15，而之后探索 I 节点，可得到 H 节点的 β 值小于父节点 F 的 α 值，发生 Beta 剪枝，此时 J 节点被剪枝掉。算法伪代码如下

Algorithm 1 AlphaBeta 剪枝

Input: 决策树

输出: 决策树上各点的收益

```
function AlphaBeta(n, Player, alpha, beta)
    if n is TERMINAL then
        return V(n)
    end if
    ChildList = n.Successors(Player)
    if Player == MAX then
        for c in ChildList do
            alpha = max(alpha, AlphaBeta(c, MIN, alpha, beta))
            if beta <= alpha then
                break
                return alpha
            end if
        end forreturn PathToGetPoint(Point)
    else
        for c in ChildList do
            alpha = min(beta, AlphaBeta(c, MAX, alpha, beta))
            if beta <= alpha then
                break
                return beta
            end if
        end for
    end if
end function
```

2 实验过程

本次实验实现五子棋博弈，设计好正确的评估函数后生成根据博弈树结构生成五子棋的博弈树，再使用 MiniMax 策略寻找最优的落子点，在搜索过程中采用 Alpha-beta 剪枝。

2.1 关键代码

本次实验采用 python 实现五子棋的 UI 与具体人机博弈下 AI 的决策，关键代码分为决策树搜索，评价函数与界面实现三部分。

2.1.1 决策树搜索

实验中定义了 GameTree 类用以封装需要的方法，保存了搜索用到的变量，类的初始化函数如下

```
    def __init__(self, first):
        self.first = first                  # 谁先手
        self.column = 11                    # 棋盘列数
        self.row = 11                       # 棋盘行数
        self.grid_width = 40                # UI 界面格子的大小
        self.chessboard = self._init_chessboard()       # 初始化棋盘
        self.AI_pieces = [(5,5),(6,5)]if first == 'AI' else [(5,6),(4,5)]
        ↪  # 根据先后手初始化 AI 的棋子
        self.player_pieces = [(5,5),(6,5)]if not first == 'AI' else
        ↪  [(5,6),(4,5)]      # 初始化玩家的棋子
        # 打印初始化的棋子
        for point in self.AI_pieces:
            self._print_piece(point,'white')
        for point in self.player_pieces:
            self._print_piece(point, 'black')
        self.pos_in_chessboard = [  (i,j)  for i in range(self.column) for
        ↪  j in range(self.row)]    # 初始化棋盘的所有位置
        # 进攻棋形的相应得分，评价函数用
        self.shape_score = [(50, (0, 1, 1, 0, 0)),
                (50, (0, 0, 1, 1, 0)),
                (200, (1, 1, 0, 1, 0)),
                (500, (0, 0, 1, 1, 1)),
                (500, (1, 1, 1, 0, 0)),
                (5000, (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)),
                (5000000, (0, 1, 1, 1, 1, 0)),
                (99999999, (1, 1, 1, 1, 1))]
        # 开始游戏
        self._play()
```

根据算法，由于 MIN 与 MAX 点的独立性，可分别用两个函数处理了 MIN 节点和 MAX 节点两种情况，经查阅资料，可使用负值最大算法既处理 MIN 节点也处理 MAX 节点。在每个都会选取最大的分数，然而返回到上一层节点时，给出的是分数的相反数，因此在 MAX 节点仍选择的是最大值，而在 MIN 节点选择的便是分数的相反数中的最大值，则对应原分数的最小值，在这个过程中同样可使用 alpha-beta 剪枝，负值最大算法，主要是代码量上的减少，时间与空间上的效率并没有提升。实现代码如下，根据返回的 best_pos，AI 可获得最优的落子点，本次实验采用的搜索深度为 3。

```
    def _alphabeta(self, AI_or_Player, alpha, beta, depth):
        # 叶节点或达到搜索深度，直接返回
        if depth == 0 or self._win(self.AI_pieces) or
        ↪  self._win(self.player_pieces):
            return -self._eva() if AI_or_Player == 'AI' else
            ↪  self._eva(),(-1, -1)
        # 初始化最优落子点
        best_pos = (-1, -1)
        # 获得所有可落子的位置
        blank_pos_list =
        ↪  list(set(self.pos_in_chessboard).difference(set(self.AI_pieces+self.player_piece
        for blank_pos in blank_pos_list:
            # 如果该空白位置周围没有棋子，则不作考虑
            if not self._has_neightnor(blank_pos):
                continue
            # 进入下一层，负值最大算法
            if AI_or_Player == 'AI':
                self.AI_pieces.append(blank_pos)
```

```
                value, _ = self._alphabeta('Player', -beta, -alpha, depth -
                ↪  1)
            else:
                self.player_pieces.append(blank_pos)
                value, _ = self._alphabeta('AI', -beta, -alpha, depth - 1)
            value = - value
            if AI_or_Player == 'AI':
                self.AI_pieces.remove(blank_pos)
            else:
                self.player_pieces.remove(blank_pos)
            if value > alpha:
                if depth == 3:
                    best_pos = blank_pos
                # alpha-beta 剪枝
                if value >= beta:
                    return beta, best_pos
                alpha = value
        return alpha, best_pos
```

2.1.2 评价函数

对于实现的五子棋博弈，需要了解一些五子棋常见的进攻棋形，根据进攻的棋形我们可以判断当前棋盘局势的优劣，以此来作为我们的评估函数，评价函数设计参考了 CSDN 上的博客。五子棋最常见的基本棋型有以下几种：连五，活四，冲四，活三，眠三，活二，眠二，其具体棋形如类初始化中 shape_score 所示，列表中每个元组中的第一项表示该棋形的得分，第二项表示具体棋形，其中 1 表示棋子，0 表示空白位置。

```
        self.shape_score = [(50, (0, 1, 1, 0, 0)),
                (50, (0, 0, 1, 1, 0)),
                (200, (1, 1, 0, 1, 0)),
                (500, (0, 0, 1, 1, 1)),
                (500, (1, 1, 1, 0, 0)),
                (5000, (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)),
                (5000000, (0, 1, 1, 1, 1, 0)),
                (99999999, (1, 1, 1, 1, 1))]
```

根据以上棋形，我们可以计算五子棋博弈双方的在棋形上的总得分。而玩家作为极大节点，AI 是极小节点，选择玩家的得分减去 AI 的得分作为当前五子棋局势的分数，在这里可以通过调整双方分数的占比来达到选择 AI 趋于进攻还是防守。对于博弈的双方，计算已下的棋在四个方向上是否构成 shape_score 中的形状，实现代码如下：

```
    def _eva(self):
        # 玩家的得分棋阵
        player_score_list = []
        player_score = 0
        # 每个棋子在四个方向根据棋形计算
        for piece in self.player_pieces:
            x = piece[0]
            y = piece[1]
            player_score += self._cal_score(x, y, 0, 1, self.AI_pieces,
            ↪  self.player_pieces, player_score_list)
            player_score += self._cal_score(x, y, 1, 0, self.AI_pieces,
            ↪  self.player_pieces, player_score_list)
            player_score += self._cal_score(x, y, 1, 1, self.AI_pieces,
            ↪  self.player_pieces, player_score_list)
            player_score += self._cal_score(x, y, -1, 1, self.AI_pieces,
            ↪  self.player_pieces, player_score_list)
        # AI 的得分棋阵
        AI_score_list = []
        AI_score = 0
        # 每个棋子在四个方向根据棋形计算
        for piece in self.AI_pieces:
            x = piece[0]
            y = piece[1]
```

```
            AI_score += self._cal_score(x, y, 0, 1, self.player_pieces, self.AI_pieces, AI_score_list)
            AI_score += self._cal_score(x, y, 1, 0, self.player_pieces, self.AI_pieces, AI_score_list)
            AI_score += self._cal_score(x, y, 1, 1, self.player_pieces, self.AI_pieces, AI_score_list)
            AI_score += self._cal_score(x, y, -1, 1, self.player_pieces, self.AI_pieces, AI_score_list)
        # 最后棋盘的评估
        return player_score - AI_score
```

其中 _cal_score 函数用于计算得分，实现代码如下，函数解释见注释。

```
    def _cal_score(self, x, y, dx, dy, enemy_list, my_list, my_score_list):
        add_score = 0
        # 在已确定的方向选择分数最大的形状
        max_score_shape = (0, None)
        # 在同个方向已计算过的则跳过 防止重复计算
        for score_piece in my_score_list:
            for pt in score_piece[1]:
                if (x, y) == pt and (dx, dy) == score_piece[2]:
                    return 0

        # 在该方向上得到当前局势的棋形
        for offset in range(-5, 1):
            pos = []
            for i in range(0, 6):
                pos_now = (x + (i + offset) * dx, y + (i + offset) * dy)
                if pos_now in enemy_list:
                    pos.append(2)
                elif pos_now in my_list:
                    pos.append(1)
                else:
                    pos.append(0)
            # 当前棋形
            shape_5 = (pos[0], pos[1], pos[2], pos[3], pos[4])
```

```
24              shape_6 = (pos[0], pos[1], pos[2], pos[3], pos[4], pos[5])
25
26              for (score, shape) in self.shape_score:
27                  if shape_5 == shape or shape_6 == shape:
28                      # 判断是否为当前最大得分的棋形
29                      if score > max_score_shape[0]:
30                          # 保存最大得分的（得分，（各棋子位置），方向）
31                          max_score_shape = (score, ((x + (0 + offset) * dx,
                            ↪  y + (0 + offset) * dy),
32                                                     (x + (1 + offset) * dx,
                                                       ↪  y + (1 + offset) *
                                                       ↪  dy),
33                                                     (x + (2 + offset) * dx,
                                                       ↪  y + (2 + offset) *
                                                       ↪  dy),
34                                                     (x + (3 + offset) * dx,
                                                       ↪  y + (3 + offset) *
                                                       ↪  dy),
35                                                     (x + (4 + offset) * dx,
                                                       ↪  y + (4 + offset) *
                                                       ↪  dy)),
36                                                 (dx, dy))
37
38          if max_score_shape[1] is not None:
39              # 已统计的进攻棋形
40              for score_piece in my_score_list:
41                  # 当前进攻棋形
42                  for pt1 in score_piece[1]:
43                      for pt2 in max_score_shape[1]:
44                          # 出现相同位置的不同方向出现进攻棋形
45                          if pt1 == pt2 and max_score_shape[0] > 10 and
                            ↪  score_piece[0] > 10:
46                              # 分数增加 鼓励同个棋子凑多个棋形
47                              add_score += score_piece[0] +
                                ↪  max_score_shape[0]
48              # 增加新进攻棋形
```

```
49              my_score_list.append(max_score_shape)
50          # 最终得分
51          return add_score + max_score_shape[0]
```

2.1.3 界面实现

本次实现的五子棋使用了 Python 的 graphics 库，通过简单地点与点之间画线绘制基础的棋盘，实现较为简单，代码如下

```
1       def _init_chessboard(self):
2           win = GraphWin(" 五子棋", self.grid_width * (self.column+2),
            ↪  self.grid_width * (self.row+2))
3           win.setBackground("yellow")
4           point_on_line = 3*self.grid_width/2
5           # 画线
6           while point_on_line < self.grid_width * (self.column+1):
7               l = Line(Point(point_on_line, 3*self.grid_width/2),
                ↪  Point(point_on_line, self.grid_width *
                ↪  (self.column-1)+3*self.grid_width/2))
8               l.draw(win)
9               point_on_line = point_on_line + self.grid_width
10          point_on_line = 3*self.grid_width/2
11          while point_on_line < self.grid_width * (self.row+1):
12              l = Line(Point(3*self.grid_width/2, point_on_line),
                ↪  Point(self.grid_width * (self.row-1)+3*self.grid_width/2,
                ↪  point_on_line))
13              l.draw(win)
14              point_on_line = point_on_line + self.grid_width
15          return win
```

关于棋子的绘制与得分的绘制函数见源码。

3 实验分析

运行代码，开始五子棋博弈。选择先后手界面如图3，为了方便演示，本次博弈选择玩家先手。无论选择哪种先后手，规定 AI 持白子，玩家持黑子。选择后正式游戏界面如图4，

游戏中为 11*11 的线条，而左上角实时显示当前棋局的得分，根据评价函数可知，正表示玩家优势，负表示 AI 优势。

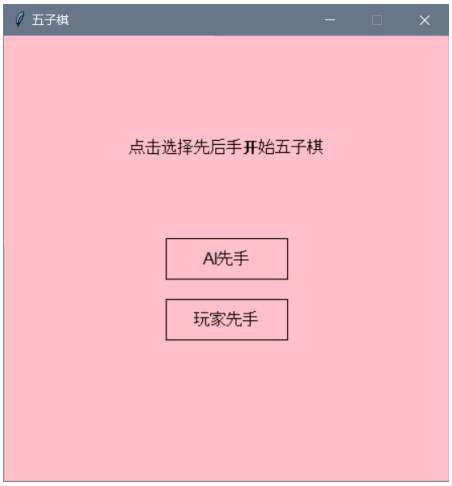


图 3: 先后手界面

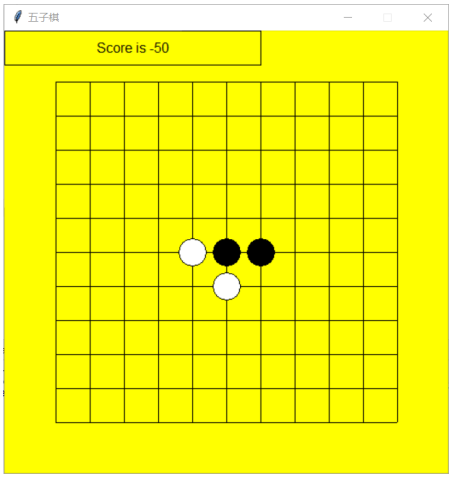
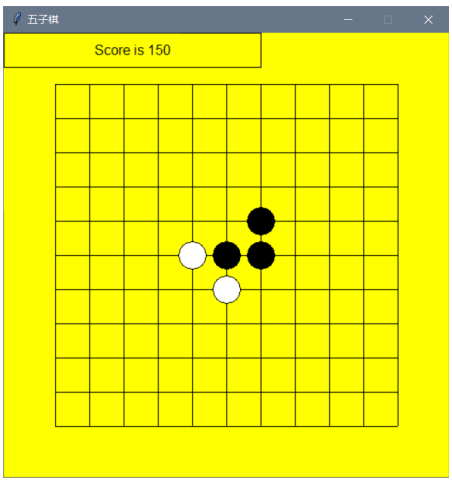


图 4: 初始化的棋盘

第一回合的博弈如图5，我选择了右上方的黑子位置凑出两个活二，而 AI 选择左下角的白子位置，将我一个活二变成眠二的同时，凑出了三个活二，当前棋局得分一下子来到了-400。



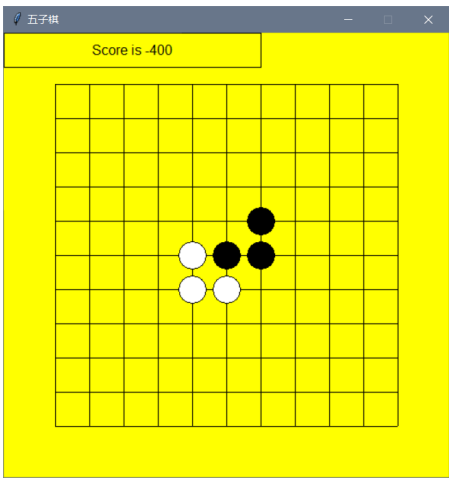


图 5: 第一回合

第二回合的博弈如图6，我选择了右下方的黑子位置凑出一个活三，再加上凑出的活二，将局势得分一下子拉到了 9900. 而 AI 选择右下角的白子位置，将我一个活二变成眠三的同时，凑出了一个活三，当前棋局得分又掰到了-9000。

第三回合的博弈如图7，我为了防守被迫选择了左上的黑子位置消除它活三的威胁，将

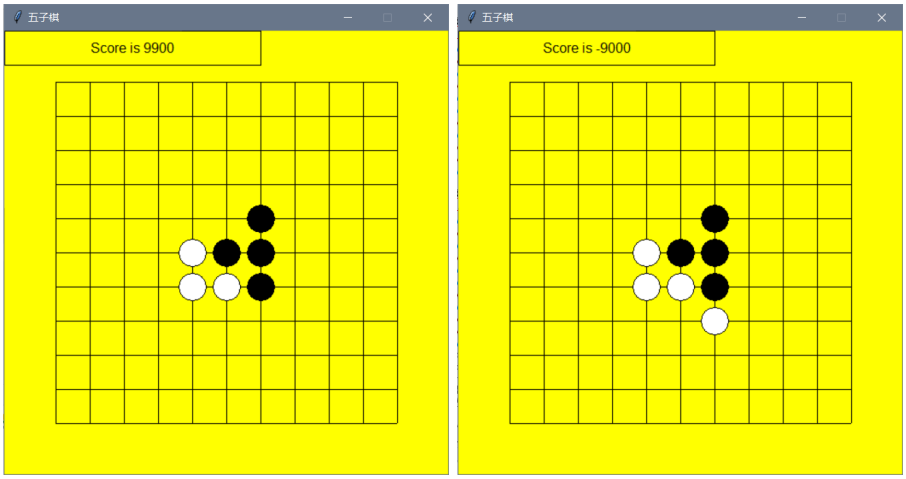


图 6: 第二回合

局势拉回到平局。而 AI 不讲武德选择左上角的白子位置，凑出一个新的活三进行进攻，同时还将我的活二变成眠二。

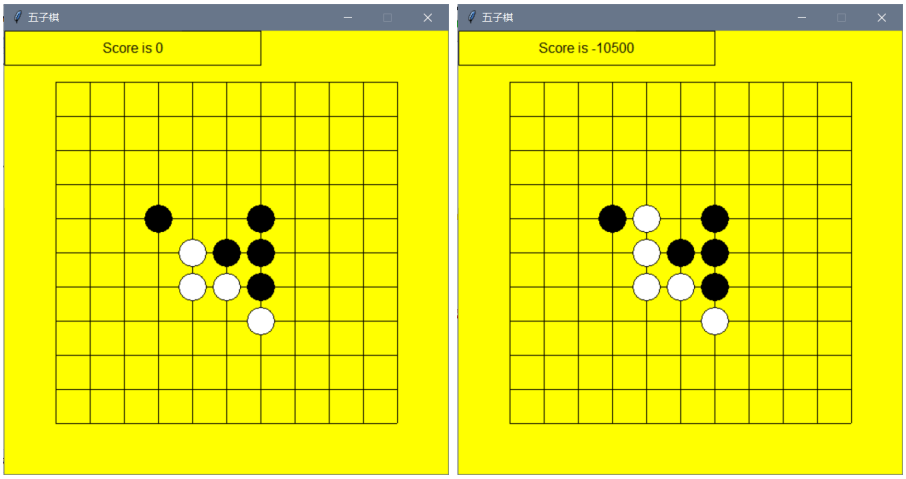


图 7: 第三回合

第四回合的博弈如图8，我选择右上的黑子位置直接来个冲四希望 AI 能没搜索到这个位置，但 AI 很快就堵住了我的冲四，这也说明了设置的 AI 在进攻比重较大时，也会着重于防守而不是无脑进攻。

经过以上四个回合的博弈，可见根据博弈树实现的五子棋博弈展现出了较好的效果。对于我一个不怎么会玩五子棋的人来说，稍有不慎，就会疏漏一些需要防守的位置导致失败。从得分上看，在以上对局中 AI 占据的优势也较大，展现了十分有效地进攻性。当然，随着评价函数中参数的调整，AI 的表现也不同，趋于防守的 AI 获胜欲望低，但玩家同样也难

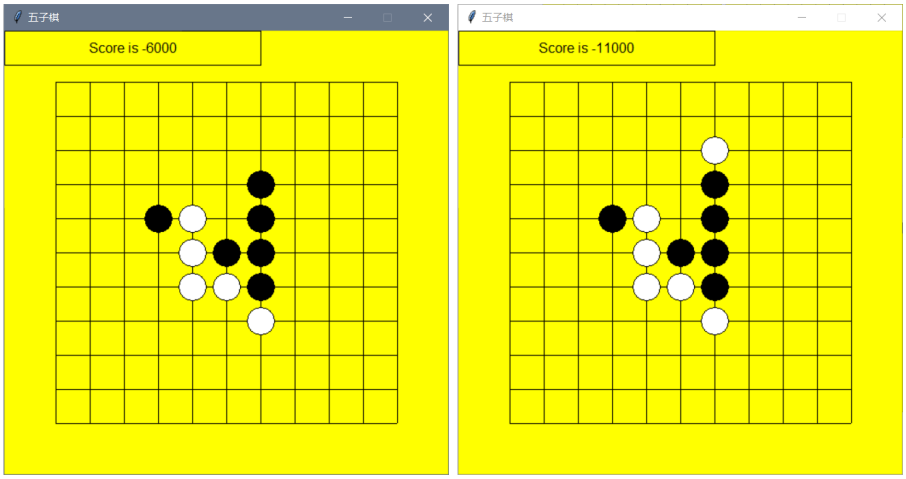


图 8: 第四回合

以获胜；趋于攻击的 AI 获胜欲望高，但也会因为想要构建进攻棋形而忽略了防守。

4 实验思考

- 本次实验使用博弈树进行五子棋博弈，虽然采用了剪枝的方式减少了搜索的深度，但由于棋盘上可下的位置较多，搜索分支指数级增长。在棋子数增加时，即使限制了搜索深度，还是会出现搜索时间过长的情况，这也是博弈树的缺点之一。
- 本次实验使用博弈树进行五子棋博弈。其中评价函数是博弈树是否能正确表示局势的关键。而这涉及了五子棋的相关知识，对于不深入了解五子棋的我来说，最初设计的评价函数根据连子的个数与得分判断，忽略了一些棋形，因此效果并不好。在查找了网上资料后，采用了最终的评价函数。