中山大學
SUN YAT-SEN UNIVERSITY

《人工智能实验》

实验报告

（实验十）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	17 计算机科学与技术
学 生 姓 名 ：	仲逊
学　　号	16327143
时　　间 ：	2019　年　12　月　5　日

实验10 ：博弈树搜索

一、 算法原理

在一个回合制双方进行的博弈过程中，当两方的利益完全对立时，说明双方的利益总是此消彼长的。$MinMax$搜索就是这样一种策略，它假设对方能总是做出最优的行动，那么一方只需要通过最小化对方的收益即可最大化己方的收益。

在这个博弈过程中，两方轮流做决策，每次都有不同的决策分支，如此就形成了一棵博弈树，在这颗树中，奇数层和偶数层分别代表了两方的决策选项，根节点作为想要最大化收益的一方（Max）节点，它要做的就是最小化下一层节点（Min节点）的收益。

$MinMax$搜索是一种 DFS 的方法，需要遍历所有的决策分支，但是由于博弈树的规模增长是指数级的，当状态空间大到一定程度时，我们就无法使用$MinMax$搜索高效地得到最大收益的策略，因此需要对博弈树进行剪枝。$\alpha-\beta$剪枝是一种经典的剪枝策略，它通过记录α和β值来剪掉博弈树中的无用分支，减少搜索次数。例如当前节点是Max节点，且已经搜索过他的左子树，得到了一个最大的收益，这时在搜索他的右子树时，由于右子树的根节点是Min节点，那么只要它的一个分值收益小于它的父亲Max节点的当前最大收益时，其余分支就可以被剪除，无需搜索。具体可以通过下面的伪代码来理解。

二、 伪代码

1. NegaMax 搜索算法（本质上是将 MinMax 极大极小两个节点的搜索函数合二为一）

---

function $NegaMax(node, depth)$ **return** $score$:

  if $node$为终止节点 **or** $depth = 0$ **then return** $evalution(node)$;

  $best\_score = -INFINITY$;

  $childlist = Successor(node)$;

  for each $child$ **in** $childlist$ **do**:

    $Make\_Next\_Move(child)$; /*假设先走这一步*/

    $score = -NegaMax(child, depth - 1)$; /*递归计算子节点的分数*/

    /*由于子节点的 Max or Min 与自己相反，故使用负值来作为 score*/

$Un\_make\_Next\_Move(child)$; /*还原走的状态*/

$best\_score = Max\{score, best\_score\}$;

return $best\_score$;

---

2. 基于 NegaMax 搜索的 α－β 剪枝

---

function $Alpha\_Beta(node, depth, \alpha, \beta)$ **return** $score$:

$if\ node$为终止节点 $or\ depth = 0$ **then return** $evalution(node)$;

$childlist = Successor(node)$;

$close\_list \leftarrow$ 已经探索过的节点集合，初始为空集;

for each $child$ **in** $childlist$ **do**:

$Make\_Next\_Move(child)$;

$score = -Alpha\_Beta(child, depth - 1, -\alpha, -\beta)$; /*递归计算子节点的分数*/

/*由于子节点的 Max or Min 与自己相反，故使用负值来作为 score*/

$Un\_make\_Next\_Move(child)$; /*还原走的状态*/

$\alpha = Max\{score, \alpha\}$;

if $\alpha \geq \beta$ **then return** β; /* $\alpha - \beta$剪枝*/

return α;

三、 评价函数与搜索策略

1. 评价函数

评价函数作用是评估一个特定状态的棋盘下两名玩家分别的得分情况，我们首先应当分析的是五子棋的常见棋型和其对应的分数，越接近五子连线或者不可阻止的四子连线的情况赢的可能性就越高，相应的，其分值就应当设定的越高，而对于可以阻止的四子连子或者不可阻止的三子连线，其分值就应当下降，以此类推，我们就可以枚举出比较有价值的棋型并赋予每种棋型一个分值。

- 棋型得分设置：

  有价值的棋型枚举（O 和 X 分别代表一种棋子，_ 则代表没有落子的空棋盘）：

  ① 不可阻止的二连子：_ _OO_ 、 _OO_ _

  ② 即将成为四连子(可阻止)：O _OO_ 、_ _OOO 、_ OOO_ 、OOO_ _

  ③ 即将成为四连子(不可阻止)：_ _OOO_ 、_OOO_ _ 、_O _OO_、_OO_O_

④ 即将成为五连子(可阻止)：OOOO_ 、O _OOO 、OO_OO 、OOO_O 、_OOOO

⑤ 即将成为五连子(不可阻止)：_OOOO_

⑥ 五连子：OOOOO

上述的棋型分值由小到大，我给棋型设置的分数表如下所示：

（1 表示该处有棋子，0 表示该处可以落子）

```
        self.shape_score = {
            (0, 1, 1, 0, 0) : 10,
            (0, 0, 1, 1, 0) : 10,
            (1, 1, 0, 1, 0) : 100,
            (0, 0, 1, 1, 1) : 100,
            (1, 1, 1, 0, 0) : 100,
            (0, 1, 1, 1, 0) : 100,
            (1, 1, 1, 0, 1) : 1000,
            (1, 1, 0, 1, 1) : 1000,
            (1, 0, 1, 1, 1) : 1000,
            (1, 1, 1, 1, 0) : 1000,
            (0, 1, 1, 1, 1) : 1000,
            (1, 1, 1, 1, 1) : 10000,
            (0, 1, 0, 1, 1, 0) : 1000,
            (0, 0, 1, 1, 1, 0) : 1000,
            (0, 1, 1, 1, 0, 0) : 1000,
            (0, 1, 1, 0, 1, 0) : 1000,
            (0, 1, 1, 1, 1, 0) : 5000,
        }
```

■ 判定棋局得分

在整个棋盘上搜索棋型的方法是对于每个 player 的每个落子点，分别向纵、横、正斜、反斜四个方向（对应的方向向量为（0,1）（1,0）（1,1）（1，-1））搜索是否有符合上述棋型的形状，在一个方向上只考虑分数最高的棋型，以此来计算整个棋盘上一个 player 的落子得分。在计算出每个 player 的得分后，己方的得分可以表示为$score_{me} - weight \times score_{enemy}$，$weight$是一个可以调节的权重，为 1 时就是简单的我方得分－敌方得分。

2. **搜索策略**

搜索深度深度为 3 的带 Alpha-Beta 剪枝的 NegaMax 搜索，详见下面的代码分析部分。

四、 代码截图

核心代码展示：

■ 基于 NegaMax 的 α－β 剪枝：

```
    def alpha_beta(self, turn, depth, alpha, beta):
        """NegaMax 搜索 alpha_beta 剪枝
        Args:
            turn bool 轮到谁走
            depth 搜索深度
            alpha beta alpha-beta 剪枝的两个值
        Returns:
            当前状态棋盘的评估分数
        """
        # 到达搜索深度or 游戏结束返回当前得分
        if depth == 0 or self.win(self.player1) or self.win(self.player2):
            return self.evaluation(turn)

        # 获取子状态节点，即当前可以落子的位置(棋盘上为空的地方)
        next_accessible_pos = self.get_children()
        # 遍历每一个子状态节点
        for next_pos in next_accessible_pos:
            # 忽略所有周围没有相邻棋子的点
            if not self.has_neightnor(next_pos):
                continue
            # 假设将子落在next_pos
            if turn:
                self.player1.append(next_pos)
            else:
                self.player2.append(next_pos)
            self.two_player.append(next_pos)

            # 递归获取这个在这个位置落子的分数
            value = - self.alpha_beta(not turn, depth - 1, -beta, -alpha)

            # 还原状态
            if turn:
                self.player1.remove(next_pos)
            else:
                self.player2.remove(next_pos)
            self.two_player.remove(next_pos)

            # alpha-beta 剪枝
            if value > alpha:
                alpha = value
                # 当前节点需要落的下一个子的坐标
                if depth == self.DEPTH:
                    self.next_point[0] = next_pos[0]
                    self.next_point[1] = next_pos[1]
```

```
                # 剪枝
                if alpha >= beta:
                    return beta

        return alpha
```

- 评估函数

```
    def evaluation(self, turn):
        """评估函数
        Args:
            turn Boolean 轮到谁
        Returns:
            当前棋盘的分数
        """
        # 分别计算我方和敌方的棋盘得分
        score1 = self.compute_player_score(self.player1, self.player2)
        score2 = self.compute_player_score(self.player2, self.player1)
        # 我方棋盘得分 - weight×地方得分 此处 weight = 0.5
        return score1 - 0.5*score2 if turn else score2 - 0.5*score1
```

- 计算一个玩家的得分

```
    def compute_player_score(self, me, enemy):
        """计算一个玩家的得分
        Args:
            me, enemy 坐标列表
        Returns:
            该玩家的得分
        """
        # 存储已经计算过的得分和形状,避免重复计算
        used_score_shape = set()
        score = 0
        # 计算我方每个棋子四个方向上的得分之和作为最后的score
        for point in me:
            x, y = point[0], point[1]
            # 计算横，竖，正斜，反斜四个方向的分数之和
            for (dir_x, dir_y) in [(0,1),(1,0),(1,1),(1,-1)]:
                score += self.compute_dir_score(
                                    x, y, dir_x, dir_y, me, enemy, used_score_shape)
        return score
```

- 计算一个棋子一个方向上的得分

```
    def compute_dir_score(self, x, y, dir_x, dir_y, me, enemy, used_score_shape):
        """计算一个棋子给定方向上的得分
```

```
        Args:
            x, y 棋子坐标
            dir_x, dir_y 方向 分为横向,纵向,正斜,反斜四个方向,用方向向量表示为
                        (0,1)(1,0)(1,1)(1,-1)
            me, enemy 坐标列表
            used_score_shape 已经计算过的得分和形状 set
        Returns:
            该棋子指定方向上的得分
        """
        # 判断该棋子在这个方向上是否已有得分
        for sc_points_dir in used_score_shape:
            _, points, dir = sc_points_dir[0], sc_points_dir[1], sc_points_dir[2]

            for point in points:
                # 棋子坐标相同且得分方向相同，则忽略
                if (x, y) == (point[0], point[1]) and (
                 dir_x, dir_y) == (dir[0], dir[1]):
                    return 0

        # 该方向上得分的最大值和形状，格式为(分数,(五个棋子的坐标),方向)
        max_sc_points_dir = (0, None, None)
        # 在落子点周围按照指定方向查找棋子形状
        for offset in range(-5, 1):
            shape = []
            # 我方落子为1，敌方落子为-1，棋盘为空的位置用0表示
            for i in range(0, 6):
                if (x + (i + offset) * dir_x, y + (i + offset) * dir_y) in enemy:
                    shape.append(-1)
                elif (x + (i + offset) * dir_x, y + (i + offset) * dir_y) in me:
                    shape.append(1)
                else:
                    shape.append(0)
            # shape5 和 shape6 分别为该棋子该方向上五个棋子的棋型和六个棋子的棋型
            # 因为得分棋型中分为五子型和六子型
            shape5 = (shape[0], shape[1], shape[2], shape[3], shape[4])
            shape6 = (shape[0], shape[1], shape[2], shape[3], shape[4], shape[5])
            # 判断是否有能够得分的棋型 shape_score 是棋型和对应得分的dict
            for shape in self.shape_score:
                score = self.shape_score[shape]
                # 如果有得分棋型，则取该方向上得分最高的棋型
                if shape5 == shape or shape6 == shape:
                    if score > max_sc_points_dir[0]:
                        # max_sc_points_dir 格式为(分数,(五个棋子的坐标),方向)
                        max_sc_points_dir = (
```

```
                        score, (
                            (x + (0+offset) * dir_x, y + (0+offset) * dir_y),
                            (x + (1+offset) * dir_x, y + (1+offset) * dir_y),
                            (x + (2+offset) * dir_x, y + (2+offset) * dir_y),
                            (x + (3+offset) * dir_x, y + (3+offset) * dir_y),
                            (x + (4+offset) * dir_x, y + (4+offset) * dir_y)
                        ),
                        (dir_x, dir_y)
                    )

        # 将该得分棋型记为已计算过得分
        if max_sc_points_dir[1] is not None:
            used_score_shape.add(max_sc_points_dir)
        # 返回最大分数
        return max_sc_points_dir[0]
```

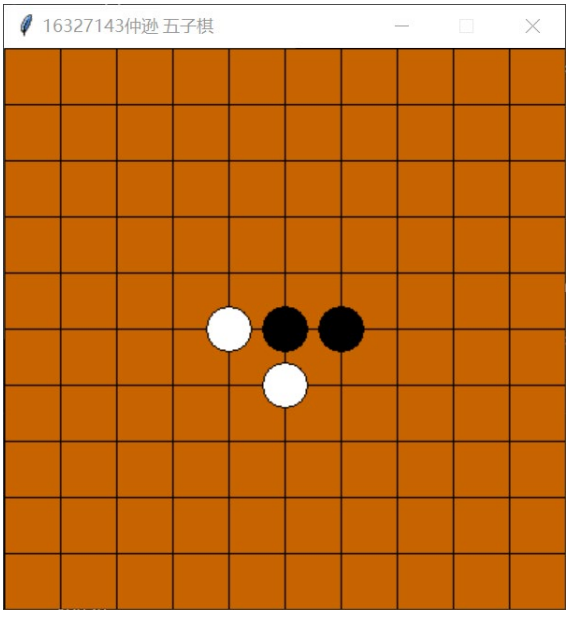
五、 实验结果展示

进入界面，选择是否先手：

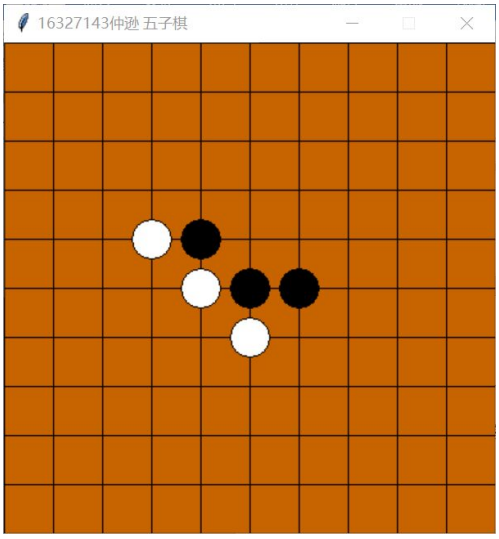
```
PS C:\Users\acer> python D:\AI_Lab\lab10\Gomoku.py
Do you want to first_hand?[y/n]:
```

■ 选择是

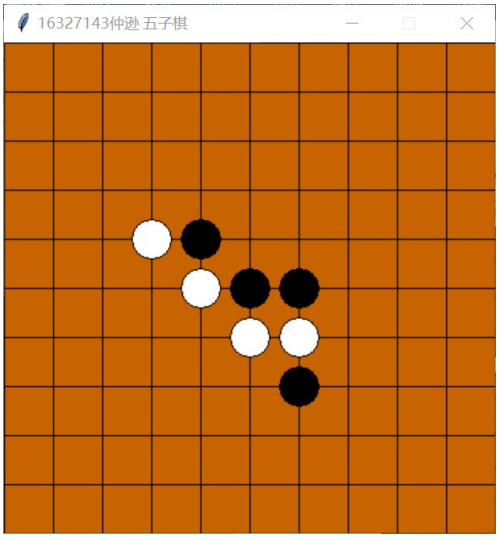
初始状态：



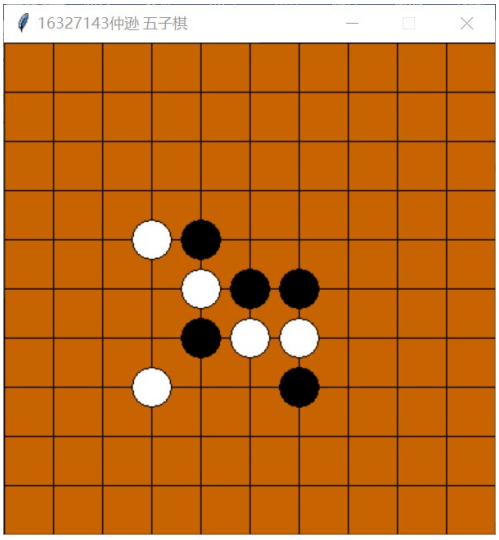
己方先手，先落黑子



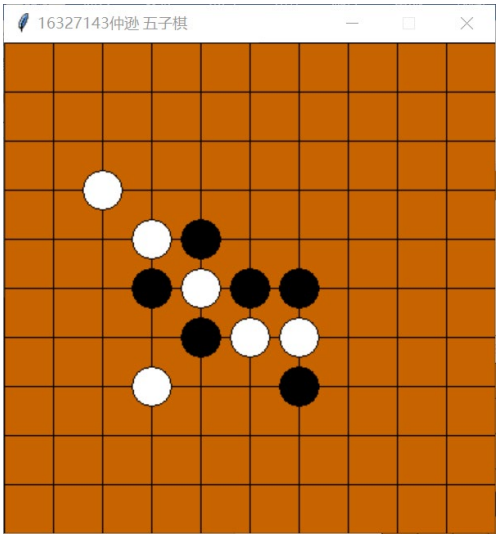
落子 2



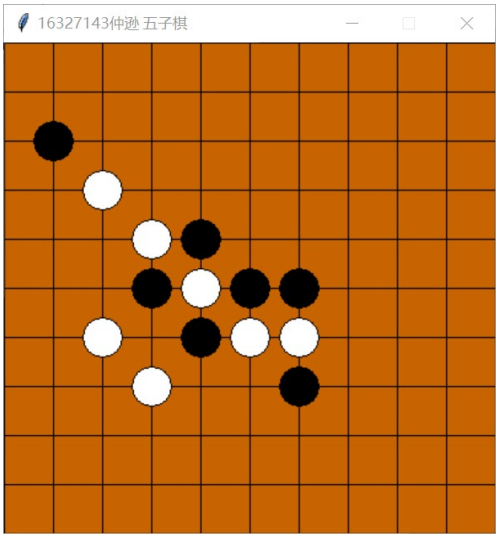
落子 3



落子 4



落子 5

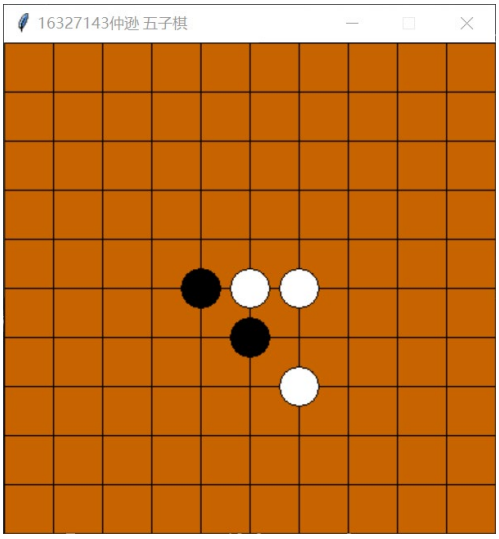


每步得分：

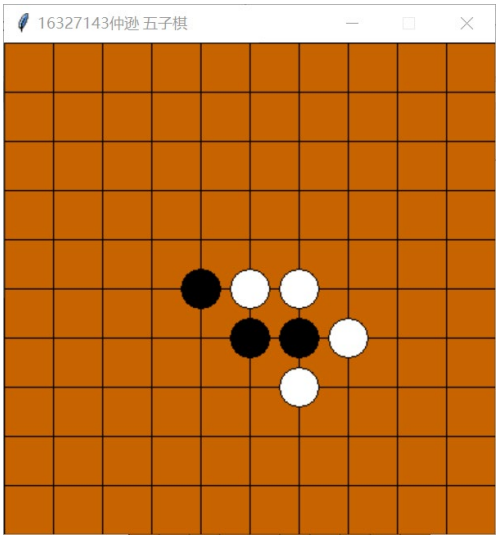
```
PS C:\Users\acer> python D:\AI_Lab\lab10\Gomoku.py
Do you want to first_hand?[y/n]:y
得分：  450.0
得分：  400.0
得分：  300.0
得分：  -0.0
得分：  -150.0
```

■ **选择否，电脑先手：**

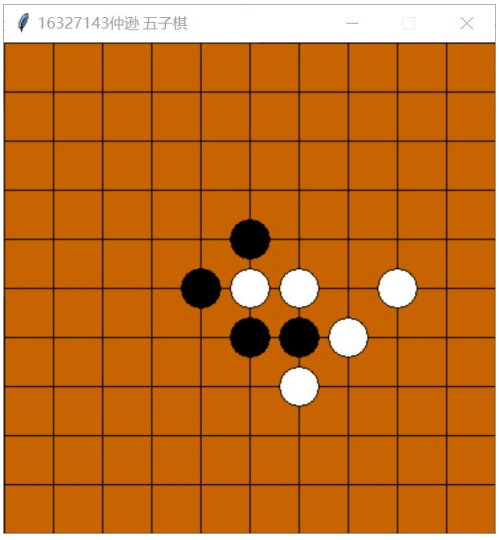
初始状态，电脑白子先手：



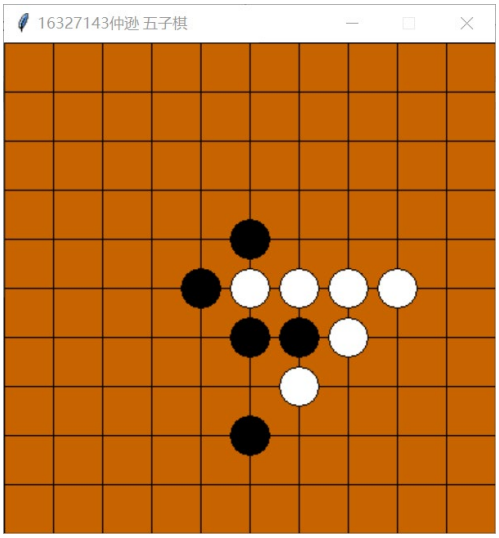
落子 1：



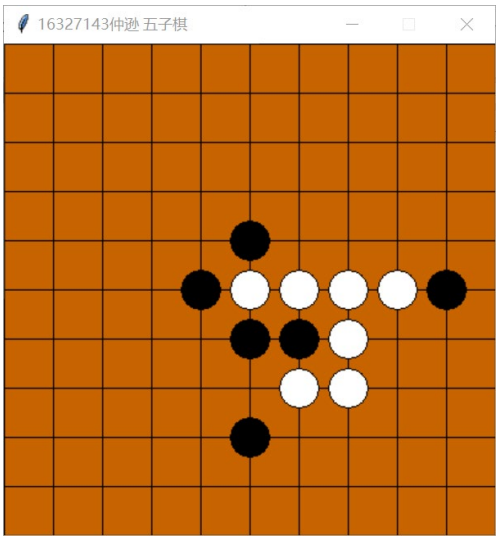
落子 2：



落子 3：



落子 4：



每步得分：

```
PS C:\Users\acer> python D:\AI_Lab\lab10\Gomoku.py
Do you want to first_hand?[y/n]:n
得分：  50.0
得分：  375.0
得分：  575.0
得分：  575.0
得分：  750.0
```