博弈树搜索实验报告

18340040 冯大纬

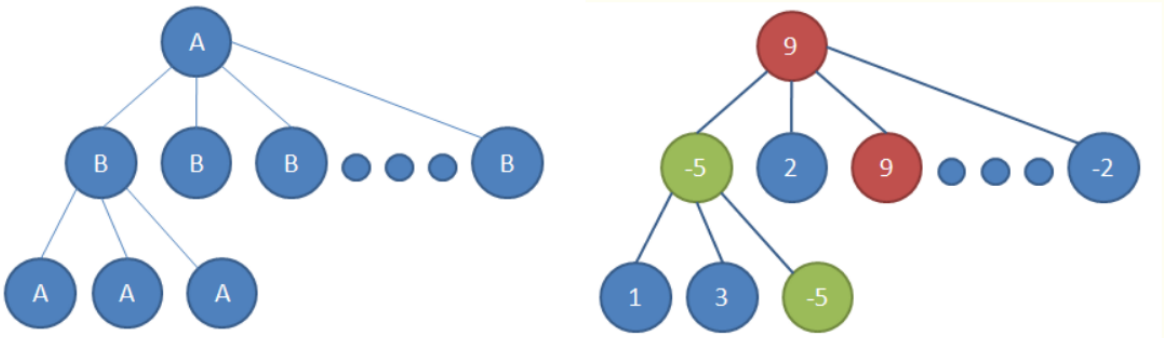
1. 算法原理

1.1 MinMax搜索

极大极小值（MinMax）搜索适用于二人零和博弈问题。两个玩家在博弈树上逐层交替行动，两人的利益相互对立，对抗搜索。玩家AB均采用最优策略，玩家A作为MAX玩家希望使得分最大化，玩家B作为MIN玩家则希望得分最小化。

在搜索树中，表示A走棋的节点即为极大节点，表示B走棋的节点为极小节点。

如下图：A为极大节点，B为极小节点。称A为极大节点，是因为A会选择局面评分最大的一个走棋方法，称B为极小节点，是因为B会选择局面评分最小的一个走法，这里的局面评分都是相对于A来说的。这样做就是假设A和B都会选择在有限的搜索深度内，得到的最好的走棋方法。



图源 cnblog

1.2 AlphaBeta剪枝

MinMax搜索必须检查的游戏状态随着博弈的进行呈指数级增长，需要通过高效的剪枝进行优化。一种常用的剪枝是alphabeta剪枝，即剪掉不可能影响决策的分支，尽可能消除部分决策树。
MAX玩家的估价函数为alpha值，MIN玩家的估价函数为beta值。根据MinMax搜索的定义，这两个值按照以下方式更新：

$$\text{alpha} = max_{son}(beta_{son}) \qquad (1)$$

$$\text{beta} = min_{son}(alpha_{son}) \qquad (2)$$

如果MAX玩家在某个节点上的alpha值大于等于它在博弈树上所有祖先的beta值的最小值那么显然该节点的alpha值对祖先的beta值不会产生贡献。继续对该节点进行搜索只会导致这个节点的alpha值变得更大，更大的alpha值并不会减小祖先的beta值，因此该节点无需继续搜索。对于MIN节点同理，如果该节点的beta值小于等于所有祖先节点的alpha值的最小值，则该节点无需继续搜索。

1.3 估值函数

本次实验我设计的估值函数基于模式匹配思想。

首先根据（x,y）坐标的横向，纵向以及两条斜向各向两边扩展4个位置，获得一个4×9的矩阵，这个矩阵的每一行代表了一个方向上的连续9枚棋子，以（x,y）为中心。

然后定义三种转移矩阵，分别为

$$
\begin{aligned}
A &= [1, 1] \\
B &= [5, 8, 5] \\
C &= [10, 15, 15, 10]
\end{aligned} \tag{1}
$$

用这三个转移矩阵和采样出来的4×9矩阵做一维卷积，由于己方下棋为1，对方为-1，空位为0，所以如果有连续的2、3、4子，就会和ABC三个矩阵的的内积获得很高的分数，而对方的同样模式则会获得负值。

基于这个矩阵以及终局位置，比如双三，活四等位置再加成分数，作为估值函数。

2. 优化点

在五子棋棋局上，通常情况下可以认为中间位置具有更大的优势，而越靠近边缘则优势越小，所以在搜索过程中可以先搜寻优势更大的位置，这样后续优势小的位置更有概率被剪枝。

在本次实验中，我选择将最外层空位设置为0优先级，每向里一层，优先级递增，也就是让搜索函数从最里面向外搜。

	0	1	2	3	4	5	6	7	8	9	10
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	0.0
2	0.0	3.0	6.0	6.0	6.0	6.0	6.0	6.0	6.0	3.0	0.0
3	0.0	3.0	6.0	9.0	9.0	9.0	9.0	9.0	6.0	3.0	0.0
4	0.0	3.0	6.0	9.0	12.0	12.0	12.0	9.0	6.0	3.0	0.0
5	0.0	3.0	6.0	9.0	12.0	15.0	12.0	9.0	6.0	3.0	0.0
6	0.0	3.0	6.0	9.0	12.0	12.0	12.0	9.0	6.0	3.0	0.0
7	0.0	3.0	6.0	9.0	9.0	9.0	9.0	9.0	6.0	3.0	0.0
8	0.0	3.0	6.0	6.0	6.0	6.0	6.0	6.0	6.0	3.0	0.0
9	0.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	0.0
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

优化过后，对比同样让AI起手，搜索4步的搜索时间

优化	搜索时间
无位置优化	455.93 s
位置优化	10.41 s

由于优势比较大的位置先搜索，所以后序优势较小的很多位置都会由于剪枝而很快结束，所以速度提升了4000%.

3. 伪代码与流程图

3.1 伪代码

3.1.1 MinMax搜索

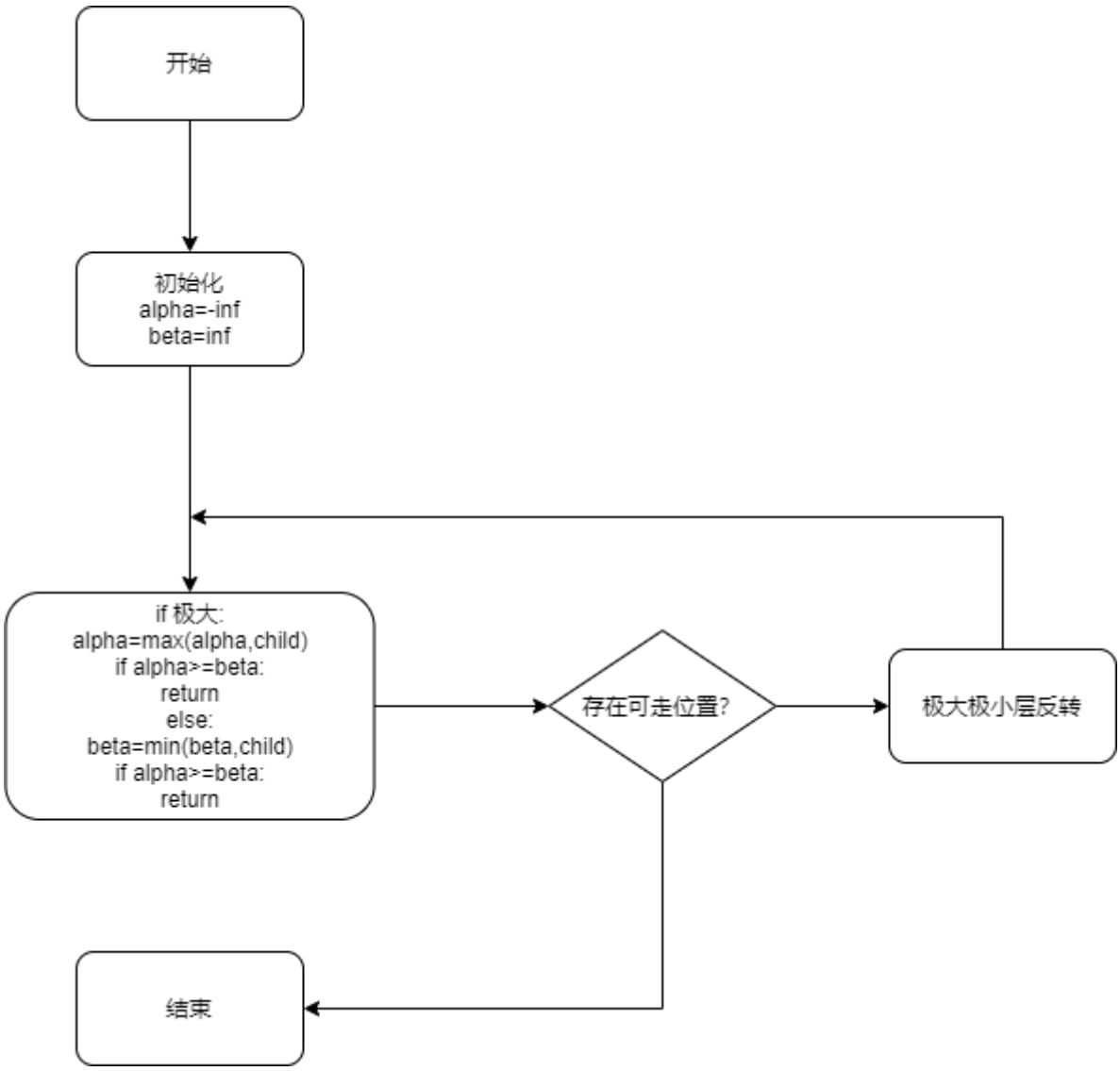
```
//node记录当前player，depth记录搜索深度
function minimax(node, depth)
    // 如果能得到确定的结果或者深度为零，使用评估函数返回局面得分
    if node is a terminal node or depth = 0
```

```
        return the heuristic value of node
    // 如果轮到对手走棋，是极小节点，选择一个得分最小的走法
    if the adversary is to play at node
        let α := +∞
        for each child of node
            α := min(α, minimax(child, depth-1))
    // 如果轮到我们走棋，是极大节点，选择一个得分最大的走法
    else {we are to play at node}
        let α := -∞
        foreach child of node
            α := max(α, minimax(child, depth-1))
    return α;
```

3.1.2 AlphaBeta搜索

```
function alphabeta(node, depth, α, β, Player)
    //达到最深搜索深度或胜负已分
    if  depth = 0 or node is a terminal node
        return the heuristic value of node
    if  Player = MaxPlayer // 极大节点
        for each child of node // 子节点是极小节点
            α := max(α, alphabeta(child, depth-1, α, β, not(Player) ))
            if β ≤ α
            // 该极大节点的值>=α>=β，该极大节点后面的搜索到的值肯定会大于β，因此不会被其上层的极小节点所选用了。对于根节点，β为正无穷
                break //beta剪枝
        return α
    else // 极小节点
        for each child of node //子节点是极大节点
            β := min(β, alphabeta(child, depth-1, α, β, not(Player) )) // 极小节点
            if β ≤ α // 该极大节点的值<=β<=α，该极小节点后面的搜索到的值肯定会小于α，因此不会被其上层的极大节点所选用了。对于根节点，α为负无穷
                break //alpha剪枝
        return β
```

3.2 流程图



4. 关键代码

```
def alpha_beta(self, _map, _x, _y, _type, d, alpha, beta):
    if d == self.depth:  # 到达深度，返回估值
        return self.__score(_map, _x, _y, -_type)
    if _type == -1:  # 极小节点
        _map[_x, _y] = 1
        self.padMapNeg[_x + 4, _y + 4] = 1  # 假定落子
        for (_i, _j, _) in self.searchSeq:
            if int(_map[_i, _j]) == 0:
                beta = min(beta, self.alpha_beta(_map.copy(), _i, _j, _type=1,
d=d + 1, alpha=alpha, beta=beta))
                if beta <= alpha:  # beta剪枝
                    self.padMapNeg[_x + 4, _y + 4] = 0  # 回溯落子
                    return beta
        self.padMapNeg[_x + 4, _y + 4] = 0  # 回溯落子
        return beta
    elif _type == 1:  # 极大节点
        _map[_x, _y] = -1
        self.padMap[_x + 4, _y + 4] = -1  # 假定落子
        for (_i, _j, _) in self.searchSeq:
            if int(_map[_i, _j]) == 0:
                alpha = max(alpha,
                            self.alpha_beta(_map.copy(), _i, _j, _type=-1, d=d +
1, alpha=alpha, beta=beta))
```

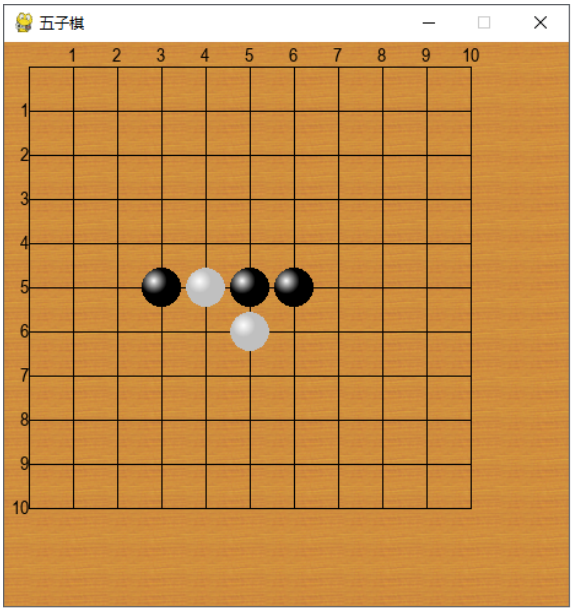
```
                if alpha >= beta:  # alpha剪枝
                    self.padMap[_x + 4, _y + 4] = 0  # 回溯落子
                    return alpha
            self.padMap[_x + 4, _y + 4] = 0  # 回溯落子
            return alpha
```

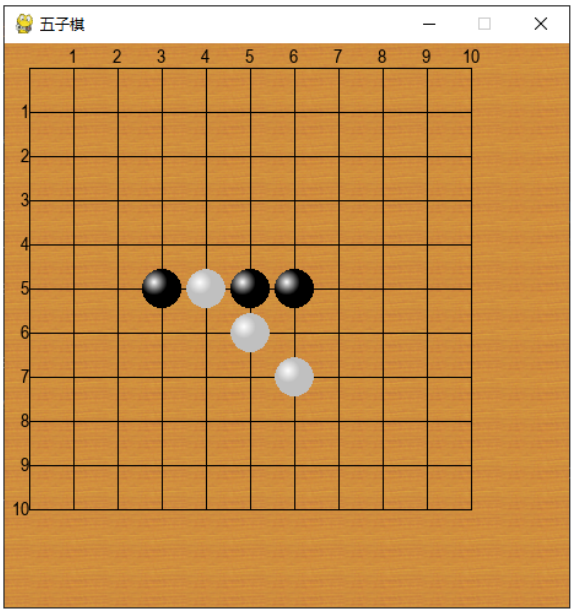
5. 实验结果分析

人机对弈

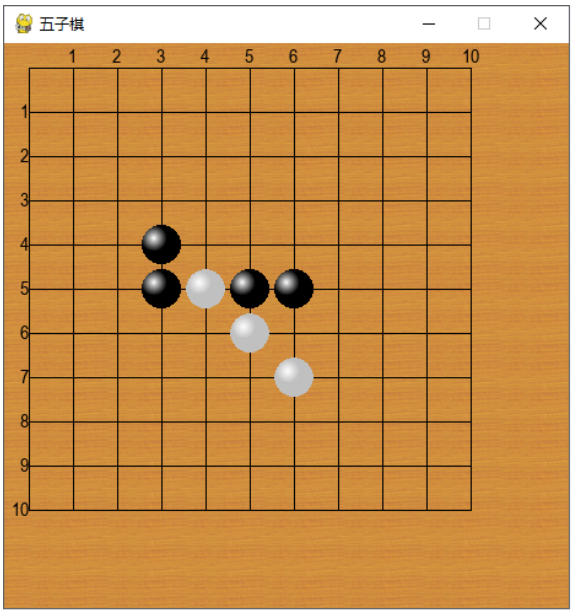
电脑先手，玩家后手

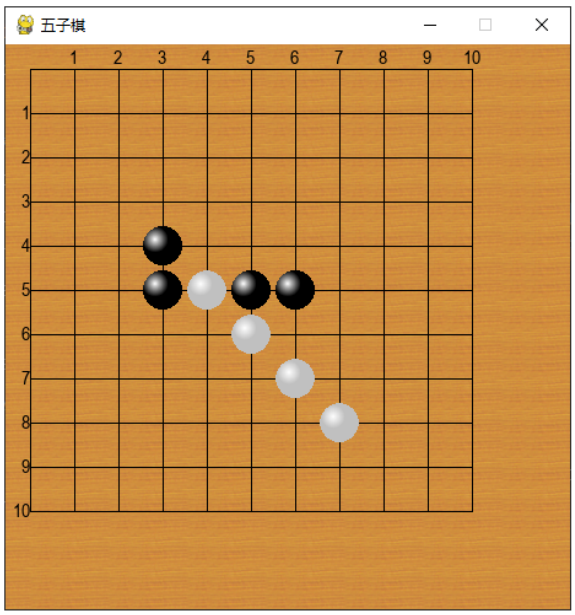
电脑落子(5,3)，玩家落子(7,6)，为自己连出三子



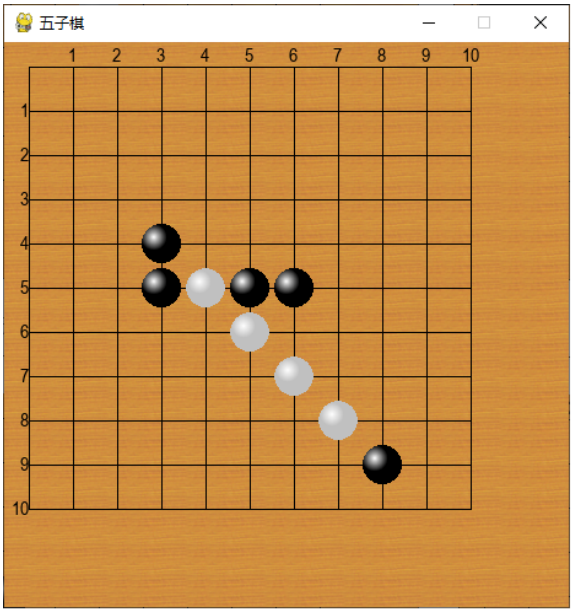


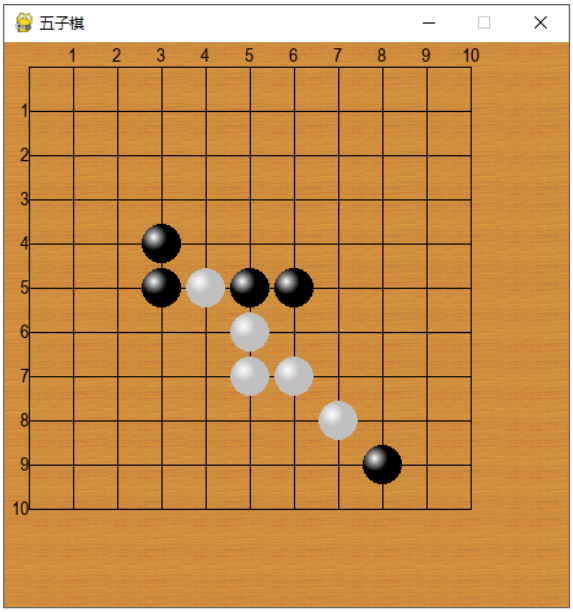
电脑落子(4,3)堵截玩家，玩家落子(8,7)试图蒙骗电脑



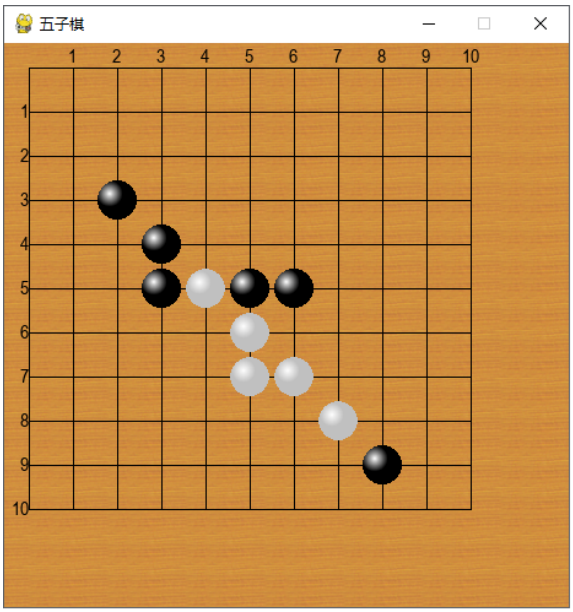


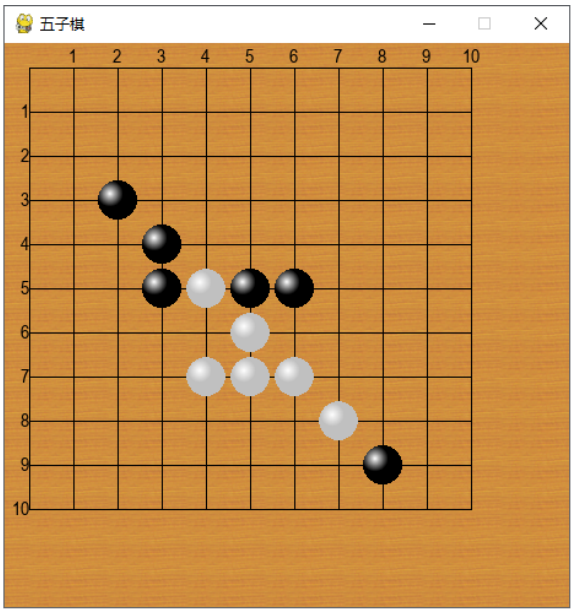
电脑识破玩家，落子(9,8)堵死玩家，玩家只好落子(7,5)改换路线



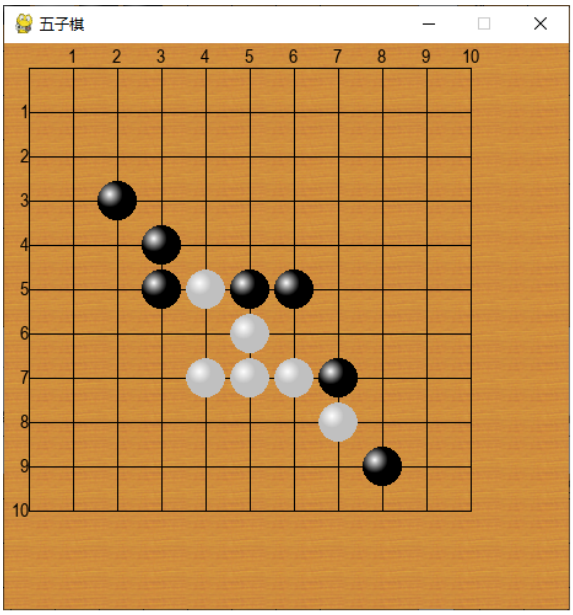


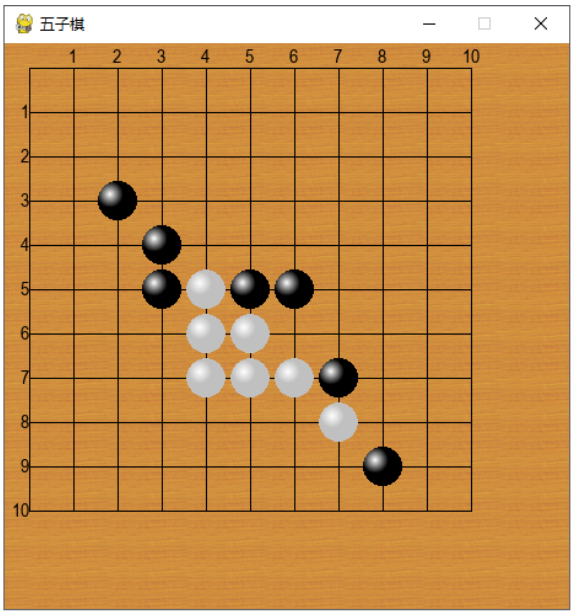
电脑认为玩家两子连在一起目前没有威胁，开始乱下(3,2)，玩家落子(7,4)连出3子



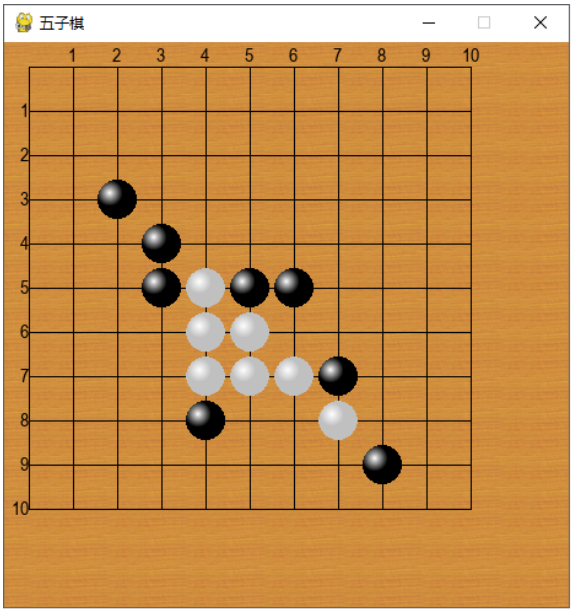


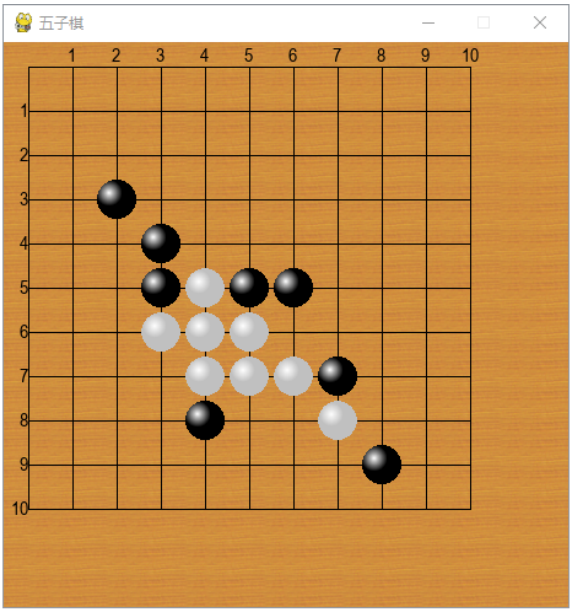
电脑开始堵截玩家的3子，落子(7,7)，玩家落子(6,4)



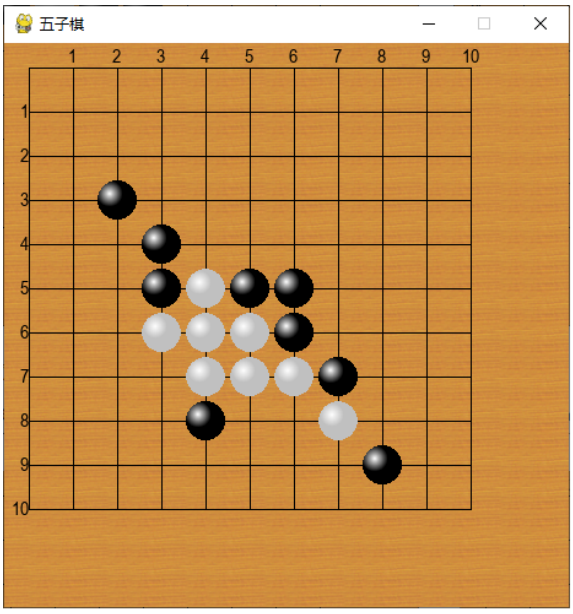


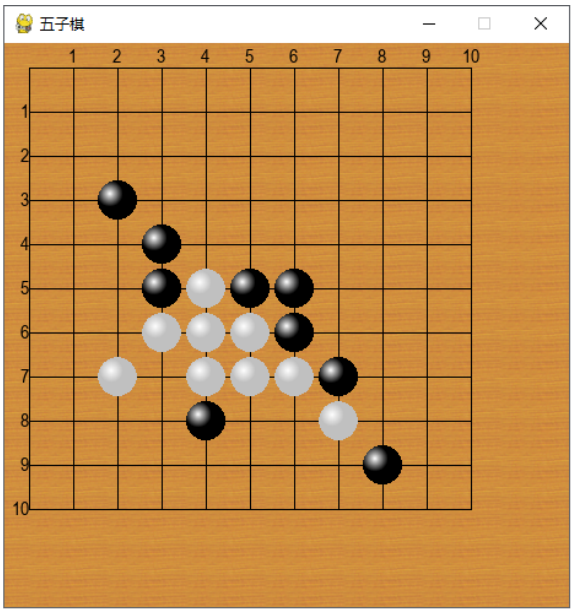
电脑落子(8,4)，终结玩家试图双三的操作，玩家只好改换思路，落子(6,3)，开始布局



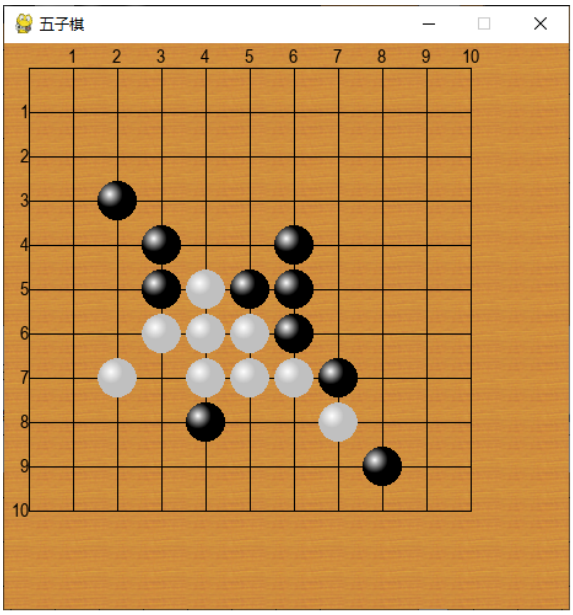


电脑只察觉到玩家试图连出4子，于是落子(7,7)堵截玩家，玩家直接落子(7,2)形成绝杀局面





电脑搜索无果，发现自己必死，开始抽风乱下，玩家一步(7,3)终结棋局，击败电脑



总结

经过本次对弈，我发现这个程序虽然堵截玩家的3子，4子操作非常及时，但如果玩家布局比较深远，则机器无法搜索到有效信息，所以会被玩家击败。