人工智能lab9 实验报告

学号：

姓名：TRY

专业：计算机科学与技术

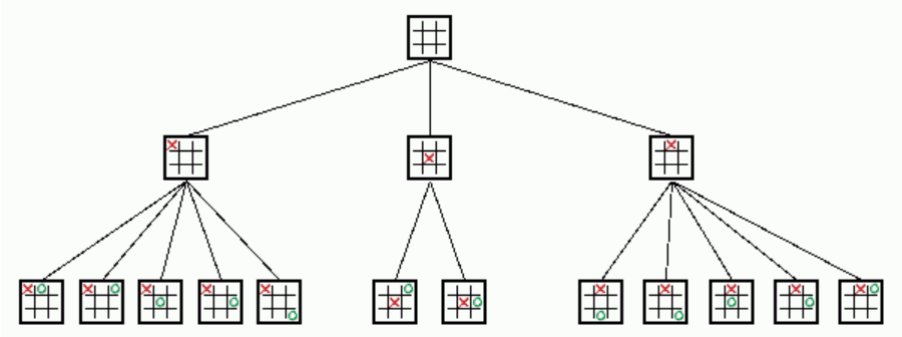
时间：2020/12/8

一、算法原理

本次实验是用博弈树+$MiniMax$搜索+ alpha-beta剪枝实现N*N的五子棋游戏。其中，博弈双方是玩家和电脑，N在实验中取11。实际上，要实现可以智能下五子棋的AI， 就是通过遍历所有可能性，求出该种可能在一定深度下的博弈结果，选择最优的可能，即博弈树搜索。

1.1 博弈树搜索

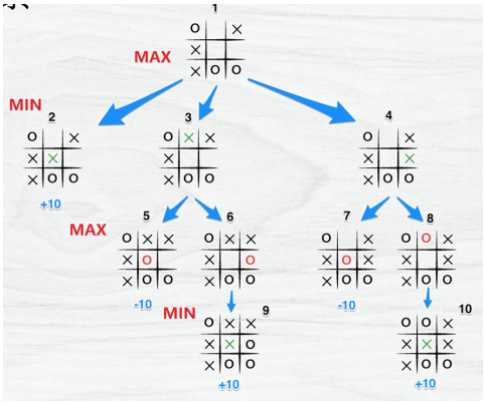
"**博弈**"表示相互采取最优策略斗争。比如说下五子棋，双方轮流扩展节点，就是相互博弈（如下图）。而博弈树就是用来表示博弈的过程。其中，内部节点和叶节点表示问题的状态，扩展节点表示一个行动，两个player的行动逐层交替出现，并用评价函数来对当前节点的优劣进行评分。博弈树搜索的目的是找出对双方都是最优的子节点的值。



1.2 $MiniMax$搜索

对于一个棋局，可以通过$MiniMax$**搜索**（极大极小值搜索）来评估当前的分数，判断优劣。player A和player B的行动逐层交替，两者利益相互对立，即假设A要使分数更大，则B要使分数更小；A和B均采取最优策略。例如实验中，对于AI来说要使得得分越小越好。若要判断落子在哪里最好，就是要计算落子在某一个点之后，当前局面的得分，然后取得分最小的那个地方落子，这就是Min（极小值）搜索。相应地，玩家是要使得得分越大越好，因此就是Max（极大值）搜索。

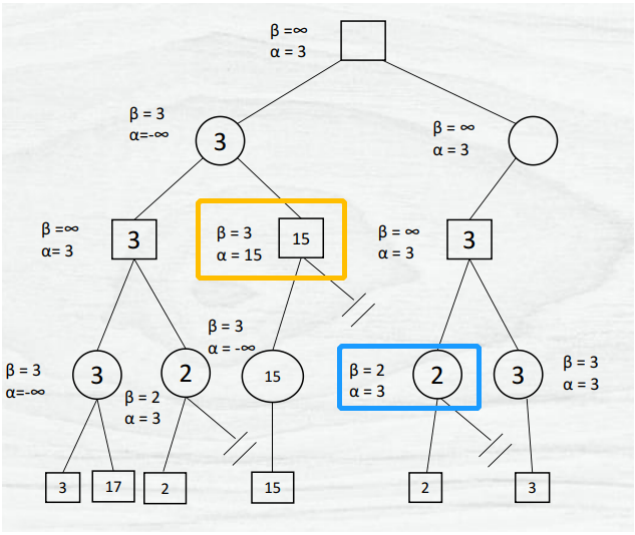
并且，需要**规定搜索深度**（例如为3），即需考虑落子在某个点后，3步后得分最大。而不是只考虑1步（不长远）。



1.3 $\alpha-\beta$剪枝搜索

然而，如果通过纯暴力搜索博弈树以寻找最佳策略，必须检查的游戏状态的数目随着博弈的进行呈指数增长，效率十分低下。因此，引入$alpha-beta$剪枝，剪掉不可能影响决策的分支，尽可能地消除部分搜索树。

例如，对于下图中黄色框里的15极大值节点，由于此时$\beta<\alpha$，故对于其父节点来说，已经不会再选取黄色框子节点的最大值作为自己节点的值了，因此黄色框节点的右子树已没有探索的必要了，可以发生α**剪枝。**同理，对于蓝色框中的2极小值节点，由于此时$\beta<\alpha$，故对于其父节点来说，已经不会再选取蓝色框子节点的最小值作为自己节点的值了，因此蓝色框节点的右子树也没有探索的必要了，可以发生β**剪枝。**



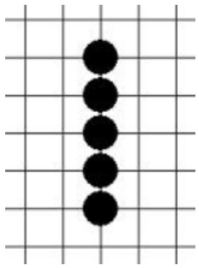
1.4 五子棋棋型和评价函数

然而，要将$MiniMax$搜索和alpha-beta剪枝运用在本次实验中，需要先了解五子棋的基本棋型，再制定具体的评价函数。

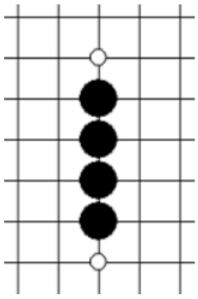
1.4.1 五子棋棋型介绍

最常见的基本棋型有：连五，活四，冲四，活三，眠三，活二，眠二。

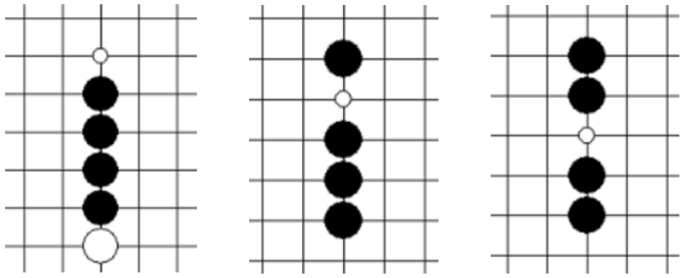
1. **连五：** 顾名思义，五颗同色棋子连在一起，不需要多讲。



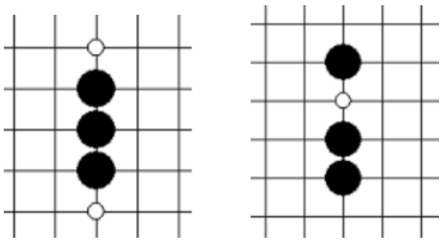
2. **活四：**有两个连五点（即有两个点可以形成五），图中白点即为连五点。活四出现的时候，如果对方单纯过来防守的话，是已经无法阻止自己连五了。



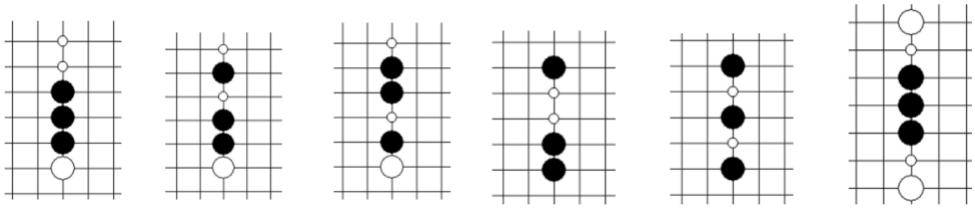
3. **冲四**：有一个连五点，如下面三图，均为冲四棋型。图中白点为连五点。相对比活四来说，冲四的威胁性就小了很多，因为这个时候，对方只要跟着防守在那个唯一的连五点上，冲四就没法形成连五。



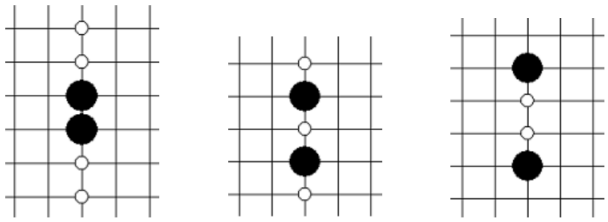
4. **活三：**可以形成活四的三，如下图，代表两种最基本的活三棋型。图中白点为活四点。活三棋型是我们进攻中最常见的一种，因为活三之后，如果对方不以理会，将可以下一手将活三变成活四，而我们知道活四是已经无法单纯防守住了。所以，当我们面对活三的时候，需要非常谨慎对待。在自己没有更好的进攻手段的情况下，需要对其进行防守，以防止其形成可怕的活四棋型。



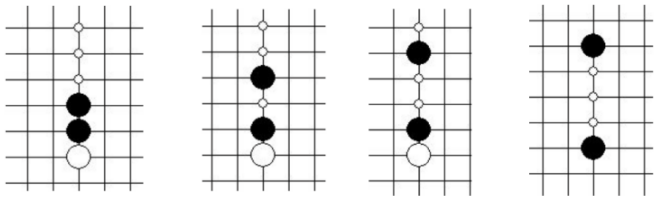
5. **眠三**：只能够形成冲四的三，如下各图，分别代表最基础的六种眠三形状。图中白点代表冲四点。眠三的棋型与活三的棋型相比，危险系数下降不少，因为眠三棋型即使不去防守，下一手它也只能形成冲四，而对于单纯的冲四棋型，我们知道，是可以防守住的。



6. **活二**：能够形成活三的二，如下图，是三种基本的活二棋型。图中白点为活三点。活二棋型看起来似乎很无害，因为他下一手棋才能形成活三，等形成活三，我们再防守也不迟。



7. **眠二**：能够形成眠三的二。图中四个为最基本的眠二棋型，白点为眠三点。



1.4.2 评价函数设计

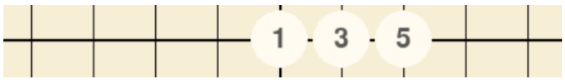
由上面的介绍可知，有7种有效的棋型，我们可以创建黑棋和白棋两个数组，记录棋盘上黑棋和白棋分别形成的所有棋型的个数，然后按照一定的规则进行评分。

在实验中，$MiniMax$的实现是：对整个棋盘进行遍历， 对于每一个白棋或黑棋，以它为中心，记录符合的棋型个数。

具体实现方法如下：

1. 遍历棋盘上的每个点，如果是黑棋或白棋，则对这个点所在四个方向形成的四条线分别进行评估。四个方向即水平，竖直，两个斜线。
2. 对于具体的一条线，如下图，已选取点为中心，取该方向上前面四个点，后面四个点，组成一个长度为9的数组。

然后找下和中心点相连的同色棋子有几个，比如下图，相连的白色棋子有3个，根据相连棋子的个数再分别进行判断，最后得出这行属于上面说的哪一种棋型。 在评估白棋1的时候，白棋3和5已经被判断过，所以要标记下，下次遍历到这个方向的白棋3和5，需要跳过，避免重复统计棋型。



3. 根据棋盘上黑棋和白棋的棋型统计信息，按照一定规则进行评分。 在下棋过程中，我们更趋向于组成上述7种棋型，而这些形状中，我们更希望能组成活三，活四来赢得比赛，针对不同的棋形对应着当前棋局上的不同得分，可以模仿棋形编写一个评分表：

```
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)),
```

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

有了评分的函数后，就可以编写一个对当前棋局计算得分的函数。可以分别计算玩家得分和AI的得分，因为玩家是极大节点，AI是极小节点，因此我们最后返回一个玩家得分减去AI得分的值。

1.5 界面UI设计

本次实验的UI设计参考了网上的代码：https://www.cnblogs.com/qiaozhoulin/p/4546884.html，并且需要安装`graphics`模块作为设计。

二、伪代码

- $MiniMax$的深度优先搜索策略伪代码如下：

```
Function DFMiniMax:
Input: n, Player
Output: V(n)

if n is TERMINAL then
    return V(n)
ChildList = n.Successors(Player)
if Player == MIN then
    return minimum of DFMiniMax(c,MAX) over c in ChildList
else:
    return maximum of DFMiniMax(c,MIN) over c in ChildList
end if
```

- alpha-beta剪枝的伪代码：

```
Function AlphaBeta:
Input:n,Player,alpha,beta
Output: alpha or beta

If n is TERMINAL then
    retrun V(n)
end if
n.Successprd(Player)
If Player == MAX then
    for c in ChildList:
        alpha = max(alpha,AlphaBeta(c,Min,alpha,beta))
        if beta<=alpha
            break return alpha
        end if
    end for
Else If Player == MIN
    for c in ChildList:
        if beta<=alpha then
            break return beta
        end if
    end for
end if
```

三、核心代码截图

1. 评价函数evaluation：计算当前棋局的得分

- 对整个棋盘进行遍历， 对于每一个白棋或黑棋，以它为中心，记录符合的棋型及其得分。
- 返回值是`player_score - AI_score * 0.1`，原因是：统计棋局的方法是统计当前棋局的得分，让AI得分相对较小一些能让AI更趋于去防守，否则有可能会出现AI以攻为守的情况，但如果下一步是人下，人就取胜了。所以要乘上一个系数，这里乘了0.1。

```
def evaluation():
    # 算玩家自己的得分
    score_all_arr_player = []  # 得分形状的位置 用于计算如果有相交 得分翻倍
    player_score = 0
    for pt in Man_pos:
        m = pt[0]   # 横坐标
        n = pt[1]   # 纵坐标
        # 计算四个方向的总得分
        player_score += cal_score(m, n, 0, 1, AI_pos, Man_pos, score_all_arr_player) # 水平左右方向
        player_score += cal_score(m, n, 1, 0, AI_pos, Man_pos, score_all_arr_player)   # 竖直上下方向
        player_score += cal_score(m, n, 1, 1, AI_pos, Man_pos, score_all_arr_player)    # 左下->右上方向
        player_score += cal_score(m, n, -1, 1, AI_pos, Man_pos, score_all_arr_player)   # 左上->右下方向

    #  算ai的得分， 并减去
    score_all_arr_ai = []
    AI_score = 0
    for pt in AI_pos:
        m = pt[0]
        n = pt[1]
        AI_score += cal_score(m, n, 0, 1, Man_pos, AI_pos, score_all_arr_ai)
        AI_score += cal_score(m, n, 1, 0, Man_pos, AI_pos, score_all_arr_ai)
        AI_score += cal_score(m, n, 1, 1, Man_pos, AI_pos, score_all_arr_ai)
        AI_score += cal_score(m, n, -1, 1, Man_pos, AI_pos, score_all_arr_ai)

    return player_score - AI_score * 0.1    # ! ! !
```

2. 计算当前步数得分函数：cal_socre

- 定义变量`max_score_shape`：

```
# 格式：max_score, 5个位置, 方向(delta_x, delta_y)。同样适用与score_all_arr
max_score_shape = (0, None)
```

- 首先，需要遍历每个落子点的4个方向，收集4个方向上前后11个位置的棋型。

```
for offset in range(-5, 1):
   pos = []
   for i in range(0, 6):
      if (m + (i + offset) * x_direct, n + (i + offset) * y_direct) in
enemy_list:
          pos.append(2)  # 敌人标记为2
      elif (m + (i + offset) * x_direct, n + (i + offset) * y_direct)
in my_list:
          pos.append(1)  # 自己标记为1
      else:
          pos.append(0)  # 空标记为0
   tmp_shap5 = (pos[0], pos[1], pos[2], pos[3], pos[4])    # 取5个点
   tmp_shap6 = (pos[0], pos[1], pos[2], pos[3], pos[4], pos[5])    # 取
6个点
```

- 同一方向上可能有多个棋型，而我们只留下分数最高的，并保存下来，防止重复计算。

```
for (score, shape) in shape_score:  # 评估分数
    if tmp_shap5 == shape or tmp_shap6 == shape:    # tmp_shape和评估份数
中的相匹配
        if score > max_score_shape[0]:
            max_score_shape = (score, ((m + (0 + offset) * x_direct, n +
(0 + offset) * y_direct), (m + (1 + offset) * x_direct, n + (1 + offset)
* y_direct), (m + (2 + offset) * x_direct, n + (2 + offset) * y_direct),
(m + (3 + offset) * x_direct, n + (3 + offset) * y_direct), (m + (4 +
offset) * x_direct, n + (4 + offset) * y_direct)), (x_direct, y_direct))
```

- 最后，考虑2个活三形成的威胁，得分翻倍处理：

```
# 计算两个形状相交， 如两个3活相交，得分增加（一个子的除外）
if max_score_shape[1] is not None:
    for item in score_all_arr:  # 查看别的方向上的得分形状
        for pt1 in item[1]:
            for pt2 in max_score_shape[1]:  # 如果存在两个得分形状有点重合
                if pt1 == pt2 and max_score_shape[0] > 10 and item[0] >
10:
                    add_score += item[0] + max_score_shape[0]  # 将重合的
形状得分翻倍
    score_all_arr.append(max_score_shape)   # 由于传的是引用，故可以成立
```

3. $MiniMax$函数：玩家落子

- 由于是通过递归来实现剪枝，因此在一开头，需要判断当前的棋局是否是有一方获胜，或者是否达到了搜索深度，如果是的话就不继续进行这个节点的搜索，立刻回溯，并返回当前棋局的评分。
- 在每次循环遍历位置时，需要判断该位置周围有没有棋子，如果没有则证明不值得下（不会下在空旷区域），因此跳过。
- 通过回溯的思想遍历棋盘的位置，并计算子节点的`Min`值，作为`alpha-temp`，并与`alpha`比较更新。
- 注意：这里呈现的是玩家走的时候的策略$MiniMax$（玩家为极大值节点），AI走的策略为$MiniMin$（极小值节点）整体思路和前者相同，就是相反求值，因此忽略。

```
# 极大节点 人走
```

```
def MiniMax(last_step, depth, max_alpha, min_beta):
    # 判断是不是终止状态
    if game_win(last_step, True) or depth == 0:
        return evaluation(), (-1, -1)
    alpha = float('-inf')  # 设定alpha为无穷小
    pos = (-1, -1)
    # 获得可以走的位置tuple
    available_pos = get_available_pos()
    for pos1 in available_pos:   # 回溯的思想
        # 忽略周围没有棋子的位置（不值得下）
        if neighbour(pos1) is False:
            continue
        Man_pos.add(pos1)
        All_pos.add(pos1)
        alpha_temp, _ = MiniMin(pos1, depth - 1, max_alpha, min_beta)
# 查看子节点：depth-1!
        # 更新当前节点的alpha值
        if alpha < alpha_temp:
            alpha = alpha_temp
            pos = pos1
        Man_pos.remove(pos1)
        All_pos.remove(pos1)

        # 判断要不要alpha剪枝
        if alpha > min_beta:
            return alpha, pos
        # 如果要继续搜索，查看是否需要更新当前最大的alpha值（用来传给子节点用，可以剪枝）
        if alpha >= max_alpha:
            max_alpha = alpha
    return alpha, pos
```

四、实验结果

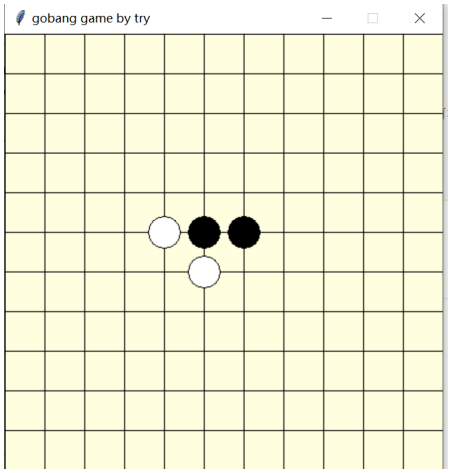
4.1 样例1：深度为3+人先手

4.1.1 初始状态：选择先手

- 控制台输入：1，表示人先手

```
请选择玩家先手1 还是电脑先手0   请输入：1
人先手
```

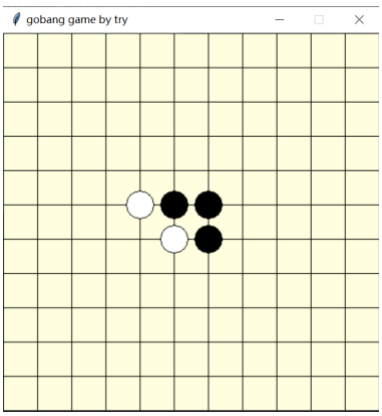
- 初始棋盘： 从一个特定的棋局开始，黑子表示人，白子表示机器

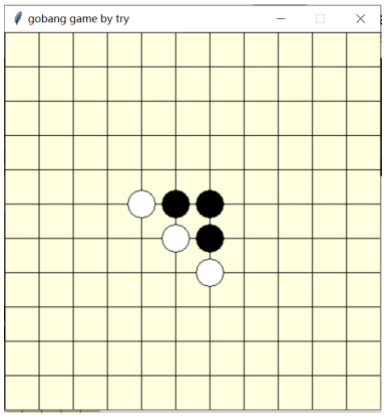


4.1.2 第一回合~第五回合

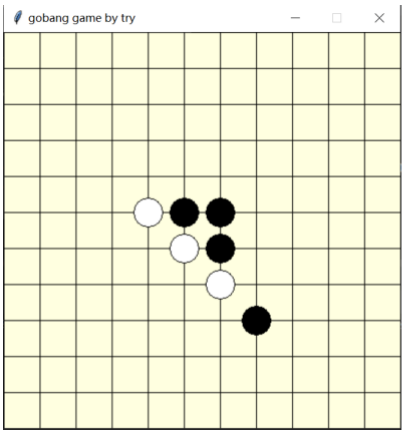
以下展示连续的5回合。

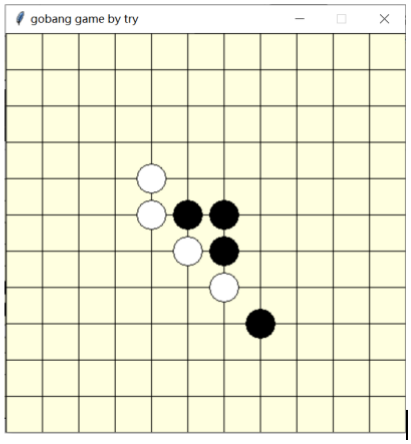
- 第一回合：用户：电脑得分 = 195:− 450



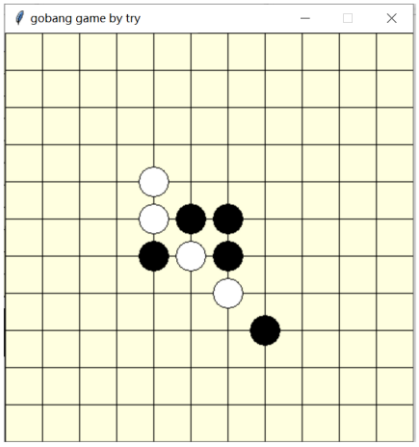


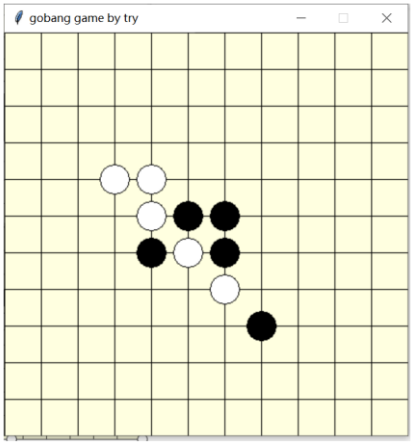
- 第二回合：用户：电脑得分 = 0 : −110



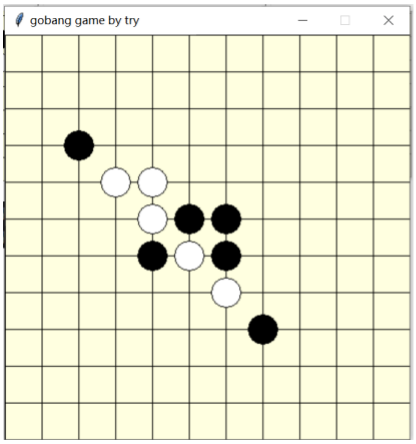


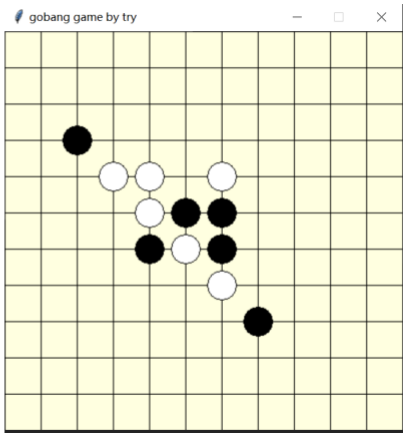
- 第三回合：用户：电脑得分 = 0 : −960



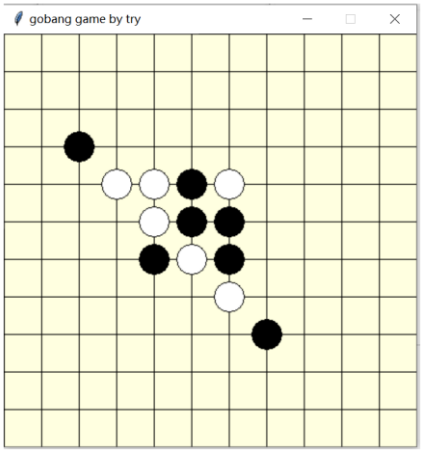


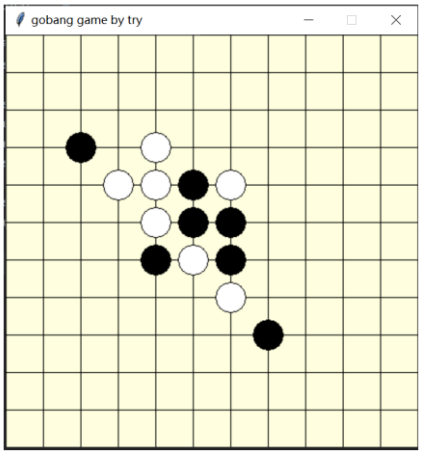
- 第四回合：用户：电脑得分 = $45:-500$





- 第五回合：用户：电脑得分 = $50:-110$





- 五回合的得分截图：

```
第1回合用户落子得分为：  195.0
第1回合电脑落子得分为：  -450.0
第2回合用户落子得分为：  0.0
第2回合电脑落子得分为：  -110.0
第3回合用户落子得分为：  0.0
第3回合电脑落子得分为：  -960.0
第4回合用户落子得分为：  45.0
第4回合电脑落子得分为：  -500.0
第5回合用户落子得分为：  50.0
第5回合电脑落子得分为：  -110.0
```

  - 注：由于这五步我下的策略不大好，所以分数不大，且电脑比较“智能”，因此每次电脑落子之后都会负数较大。

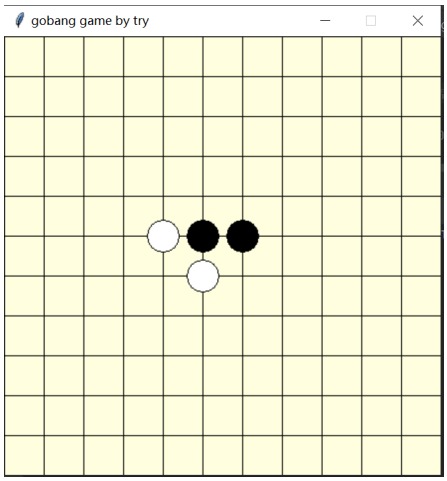
4.2 样例2：深度为2+人先手

4.2.1 初始状态：选择先手

- 控制台输入：1，表示人先手

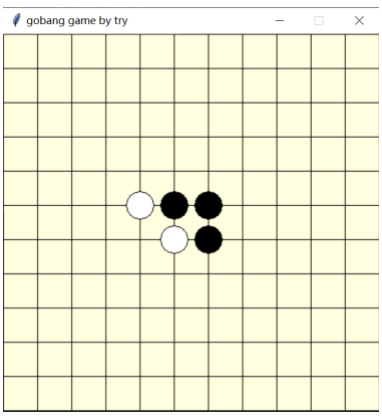
```
请选择玩家先手1 还是电脑先手0  请输入：1
人先手
```

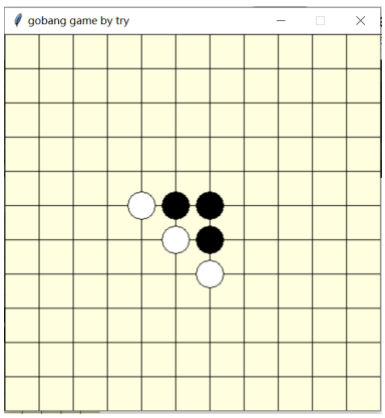
- 初始棋盘： 从一个特定的棋局开始，黑子表示人，白子表示机器



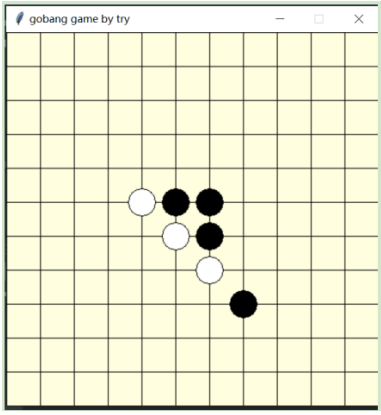
4.2.2 第一回合~第五回合

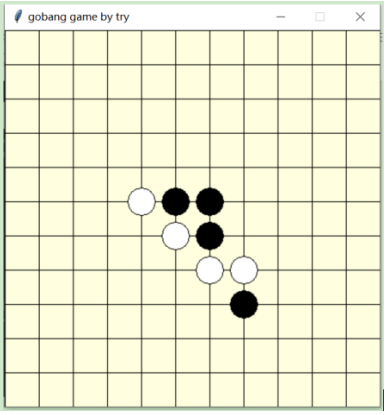
- 第一回合：用户：电脑得分 = 195 : −450



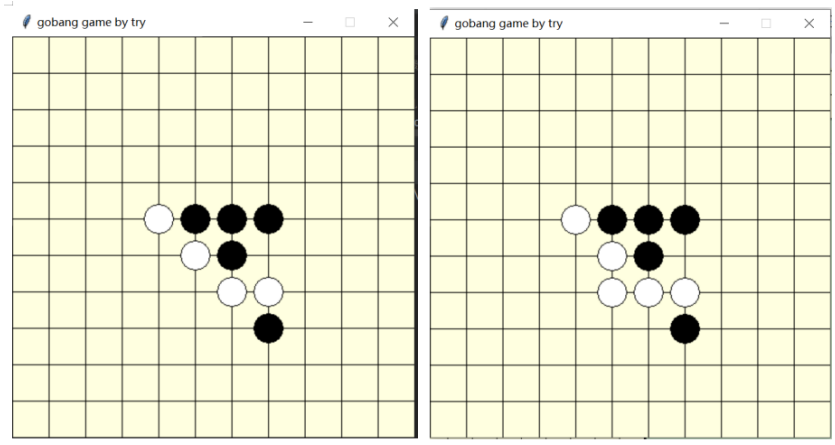


- 第二回合：用户：电脑得分 = 0 : −110

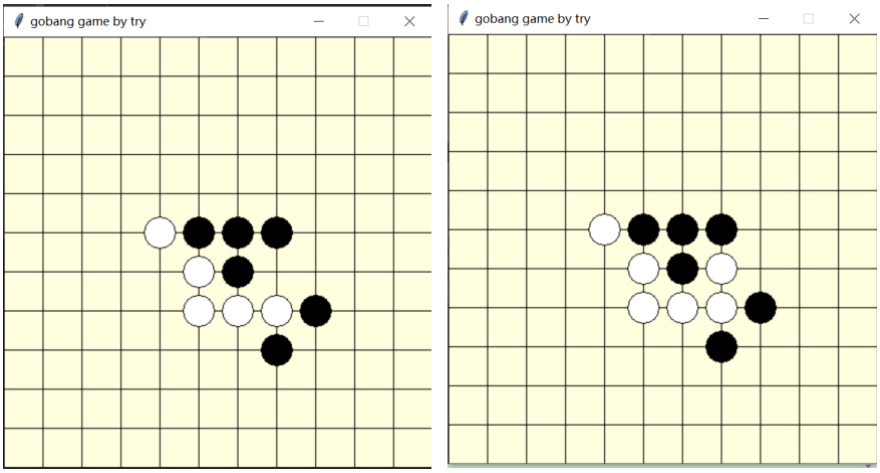




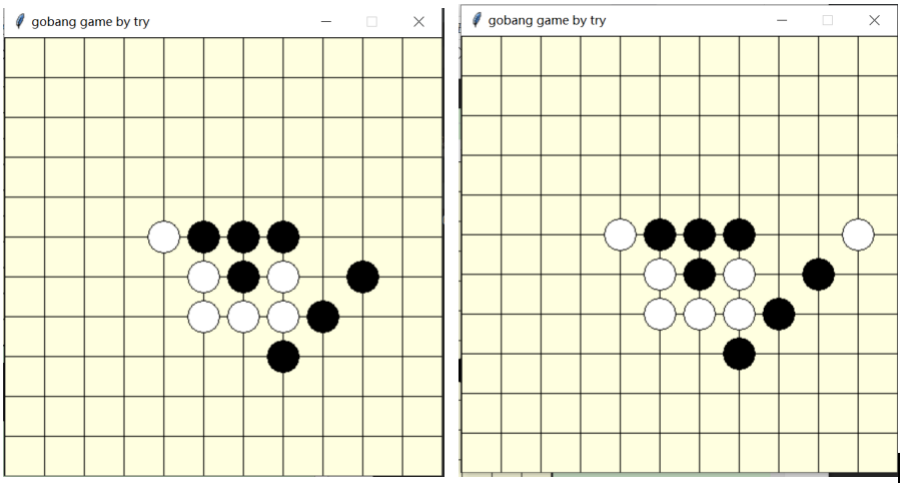
- 第三回合：用户：电脑得分 = 990 : −600



- 第四回合：用户：电脑得分 = 350 : 235



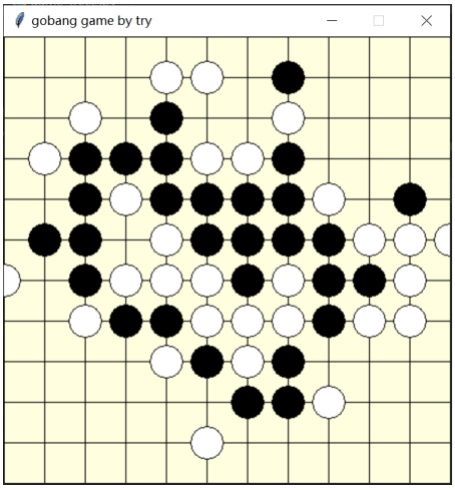
- 第五回合：用户：电脑得分 = 5185 : 685



- 五回合的得分截图：

```
第1回合用户落子得分为：  195.0
第1回合电脑落子得分为：  -450.0
第2回合用户落子得分为：  0.0
第2回合电脑落子得分为：  -110.0
第3回合用户落子得分为：  990.0
第3回合电脑落子得分为：  -600.0
第4回合用户落子得分为：  350.0
第4回合电脑落子得分为：  235.0
第5回合用户落子得分为：  5185.0
第5回合电脑落子得分为：  685.0
```

- 之后一直与电脑博弈，到最后第27回合之后停止，原因在于棋局已尽（能下的地方很少了），很难分出胜负了：




```
第21回合用户落子得分为：  5350.0
第21回合电脑落子得分为：  2300.0
第22回合用户落子得分为：  12350.0
第22回合电脑落子得分为：  3345.0
第23回合用户落子得分为：  12345.0
第23回合电脑落子得分为：  1845.0
第24回合用户落子得分为：  3095.0
第24回合电脑落子得分为：  880.0
第25回合用户落子得分为：  15680.0
第25回合电脑落子得分为：  530.0
第26回合用户落子得分为：  10430.0
第26回合电脑落子得分为：  1430.0
第27回合用户落子得分为：  10430.0
第27回合电脑落子得分为：  -70.0
```

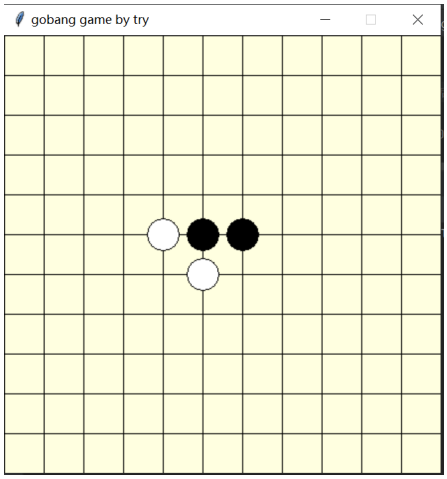
4.3 样例3：深度为2+电脑先手

4.3.1 初始状态：选择先手

- 控制台输入：0，表示电脑先手

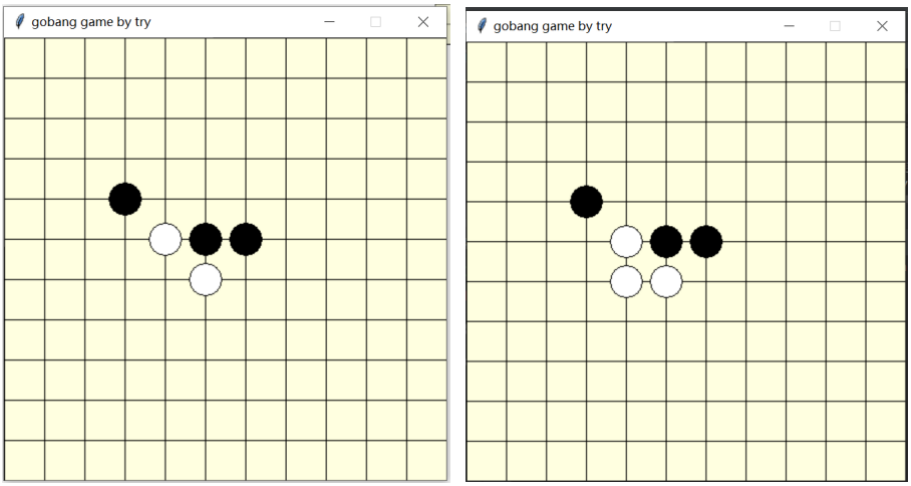
```
请选择玩家先手1 还是电脑先手0   请输入：0
电脑先手
```

- 初始棋盘： 从一个特定的棋局开始，黑子表示电脑，白子表示玩家

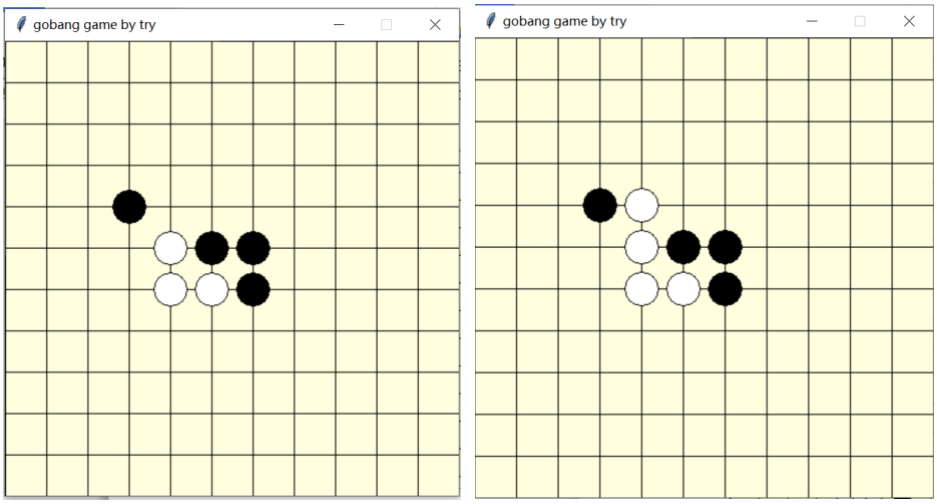


4.3.2 第一回合~第五回合

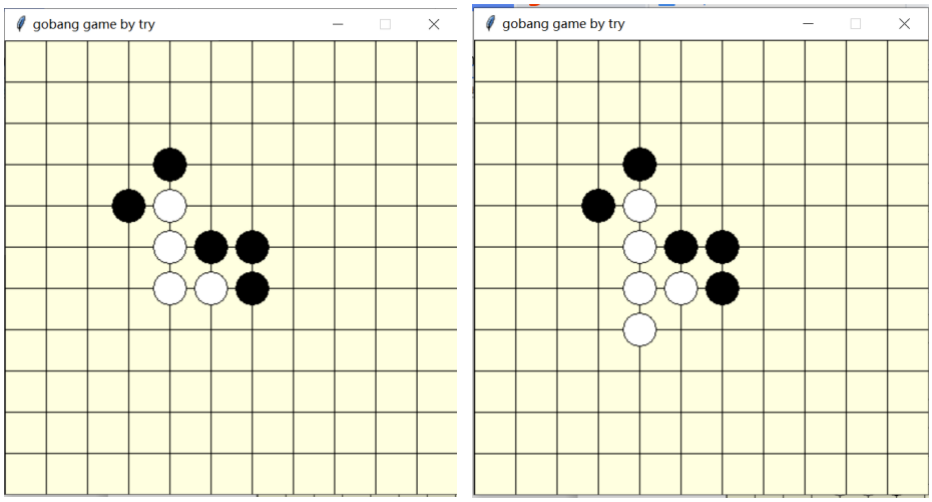
- 第一回合：电脑：用户得分 = 0 : 200



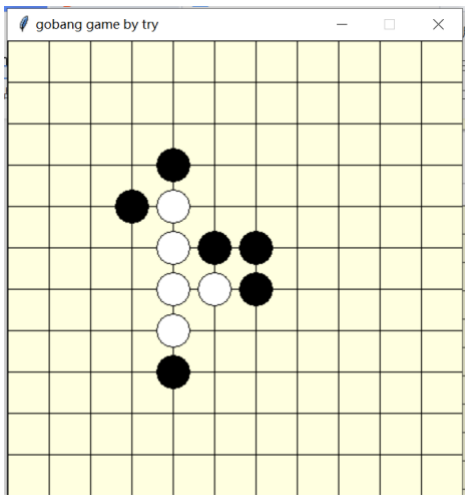
- 第二回合：电脑：用户得分 = 30 : 4995

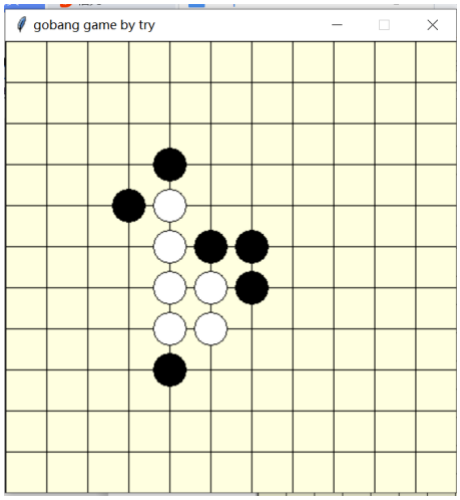


- 第三回合：电脑：用户得分 = 490 : 4990

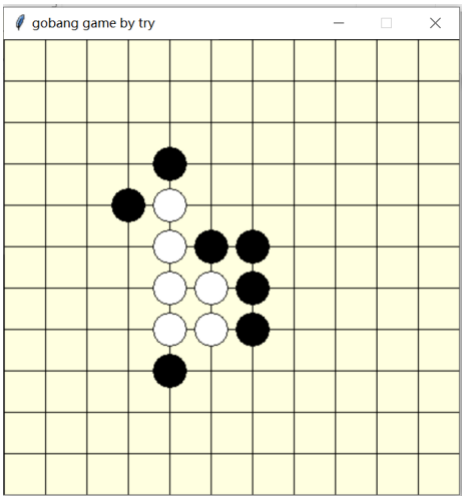


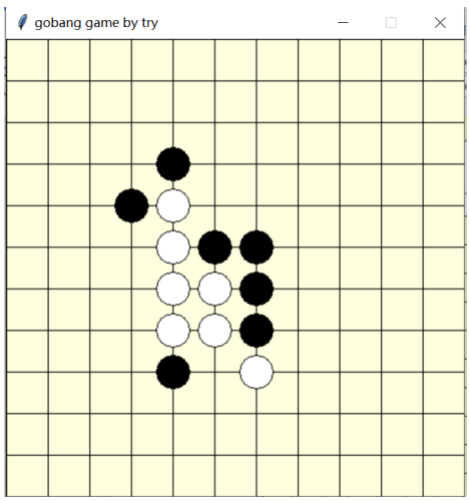
- 第四回合：电脑：用户得分 = -10 : 190





- 第五回合：电脑：用户得分 = $-455:4945$

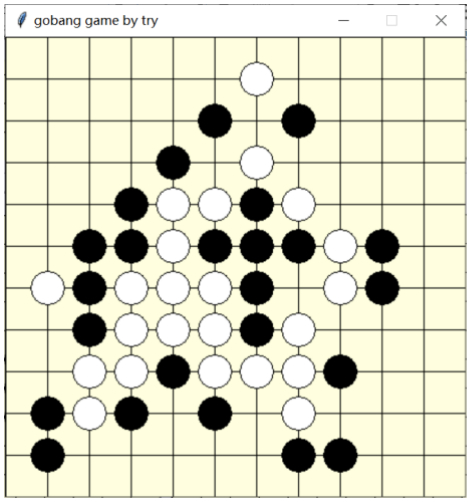




- 五回合的得分截图：

```
第1回合电脑落子得分为：  0.0
第1回合用户落子得分为：  200.0
第2回合电脑落子得分为：  30.0
第2回合用户落子得分为：  4995.0
第3回合电脑落子得分为：  490.0
第3回合用户落子得分为：  4990.0
第4回合电脑落子得分为：  -10.0
第4回合用户落子得分为：  190.0
第5回合电脑落子得分为：  -455.0
第5回合用户落子得分为：  4945.0
```

- 之后一直与电脑博弈，到最后第22回合之后停止，原因在于棋局已尽（能下的地方很少了），很难分出胜负了：




```
第16回合电脑落子得分为：  -210.0
第16回合用户落子得分为：  290.0
第17回合电脑落子得分为：  -230.0
第17回合用户落子得分为：  290.0
第18回合电脑落子得分为：  -260.0
第18回合用户落子得分为：  240.0
第19回合电脑落子得分为：  -960.0
第19回合用户落子得分为：  90.0
第20回合电脑落子得分为：  -170.0
第20回合用户落子得分为：  390.0
第21回合电脑落子得分为：  -560.0
第21回合用户落子得分为：  1040.0
第22回合电脑落子得分为：  -15.0
```

4.4 结果分析与比较

以下呈现上面三个样例的得分比较：（用户得分：电脑得分）

用户得分：电脑得分	深度为3+人先手	深度为2+人先手	深度为2+电脑先手
第一回合	195：-450	195：-450	200：0
第二回合	0：-110	50：-110	4995：30
第三回合	0：-960	990：-600	4990：490
第四回合	45：-500	350：235	190： -10
第五回合	50：-110	5185：685	4945：-455

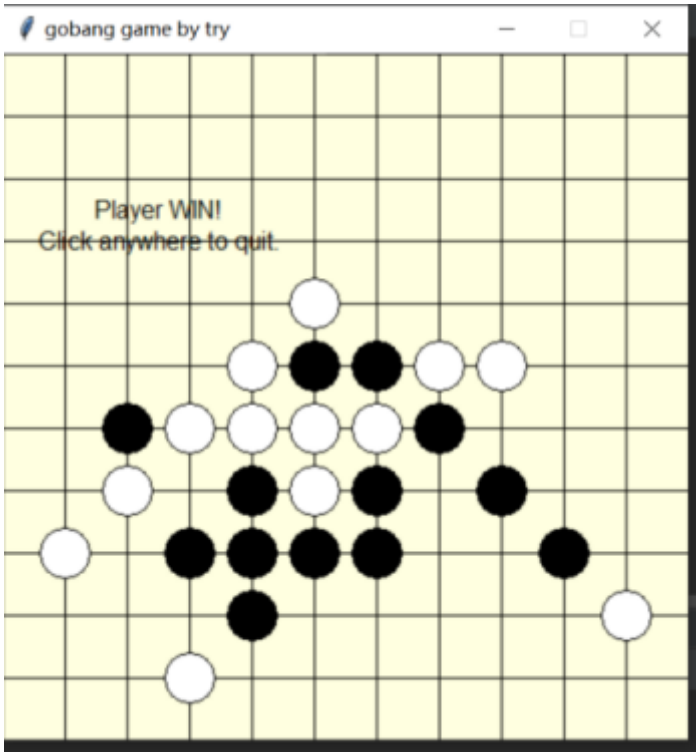
结果分析：

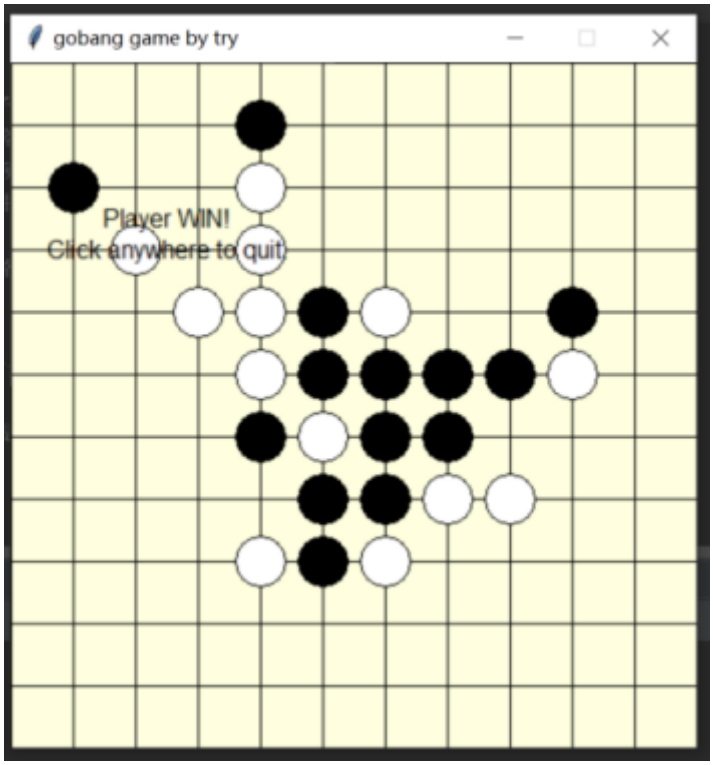
- **不同深度：**从第1、2列的数据可以明显看出，随着AI思考的深度的增加，玩家的优势大幅减少，且经常在电脑AI落棋之后处于很大劣势，十分被动。而且从棋局可以看出，两种情况在第二回合的时候电脑的落子位置就不同；当深度为2时，AI较为保守，经常会对“活二”、“眠二”、“眠三”等局势进行防守，导致局面常常会形成僵局，玩家难以突破重围落子，在深度为3时，AI思考的较多，较为冒险，会反转局面形成攻势。因此可看出深度对模型的影响很大。
- **先手选择**：从第2、3列的数据可以看出，在深度为2时，当选择电脑先手时，人的优势较小，电脑优势较大，因此可以看出先手选择对结果的影响十分大。特别是在深度为3时，电脑先手的赢面非常大。

4.5 结果输出

- 另附两张我在深度为3时战胜AI的截图：*（非常艰难的赢得了比赛）*

另附上两次我战胜电脑的截图：


gobang game by try
Player WIN!
Click anywhere to quit.


gobang game by try
Player WIN!
Click anywhere to quit.