一. 实验概述

(1) 实验目的

熟悉和掌握博弈树的启发式搜索过程、α-β剪枝算法和评价函数，并利用α-β剪枝算法开发一个五子棋人机博弈游戏。

(2) 实验内容

1.以五子棋人机博弈问题为例，实现α-β剪枝算法的求解程序（编程语言不限），要求设计适合五子棋博弈的评估函数。

2.要求初始界面显示 15*15 的空白棋盘，电脑执白棋，人执黑棋，界面置有重新开始、悔棋等操作。

3.设计五子棋程序的数据结构，具有评估棋势、选择落子、判断胜负等功能。

4.撰写实验报告，提交源代码（进行注释）、实验报告、汇报 PPT。

二. 实验方案设计

(1) 总体设计思路与总体架构

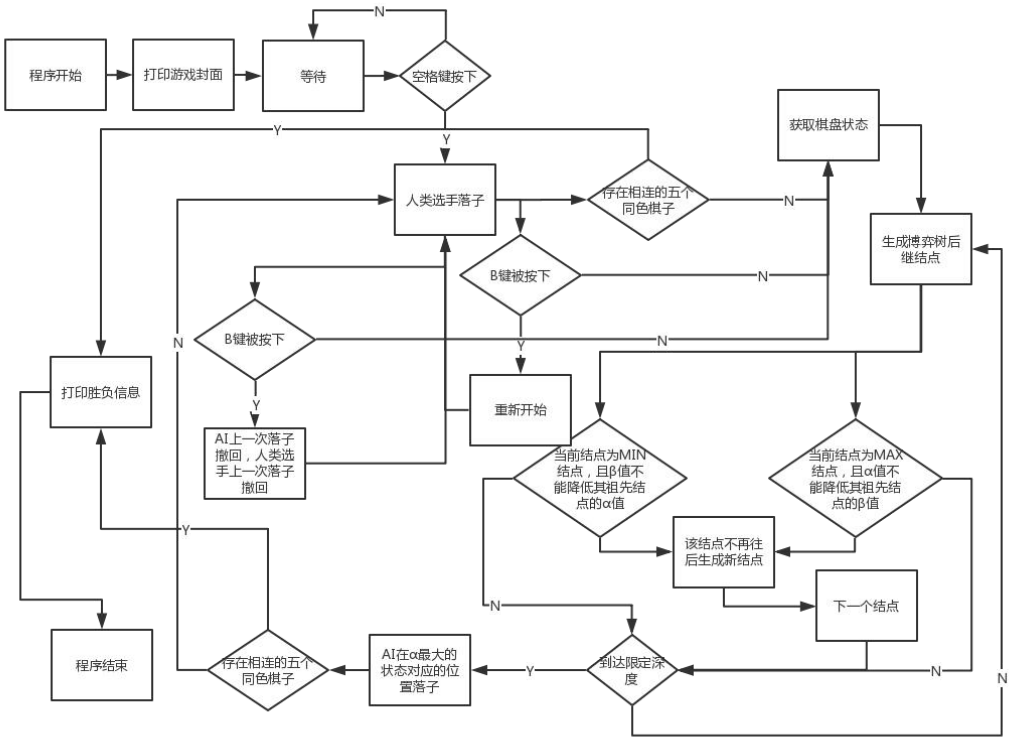
总体设计框架分为三大部分：

①人机交互界面部分，通过图形用户界面显示棋盘的状态、落子的次序，打印操作提示文本信息，在游戏开始时显示游戏封面，在游戏结束时给出胜负信息等。

②搜索部分，AI 接收到代表棋盘状态的矩阵信息后使用深度受限的启发式搜索，尝试不同位置的落子，并根据这些位置的评估函数选择最优落子位置。

③游戏控制部分，通过设置事件监测来获取键盘和鼠标的操作信息，通过键盘按下的键位来做出开始/悔棋/重新开始等控制行为，通过判断离鼠标点击的位置最近的网格来获取玩家的落子位置。并在每次 AI/玩家落子后判断是否产生输赢。

流程图如下：



(2) 核心算法及基本原理：

带 α - β 剪枝的极小极大搜索：

轮到 AI 落子时，在获取当前棋盘状态后，AI 尝试所有位置的落子方案，生成后继状态的结点。采用极小极大搜索形成博弈树，对于所有的 MAX 结点，其 α 值为其所有直接后继结点的 β 值的最大值，意义是 AI 在玩家足够聪明而使得其能够最小化 AI 的得分的情况下所能得到的最大得分，对于所有的 MIN 结点，其 β 值为其所有直接后继结点的 α 值的最小值，意义是玩家在 AI 足够聪明而使得其能够最大化自己的得分的情况下所能限制 AI 得到的最小得分。所有叶子结点都有效用函数评价出的数值，用来衡量 AI 的得分。

任何 MAX 结点 X 的 alpha 值如果不能降低其祖先结点的 beta 值时，对 X 以下的分枝可以停止搜索。任何 MIN 结点 Y 的 beta 值如果不能升高其祖先结点的 alpha 值时，对 Y 以下的分枝可以停止搜索。从而大大减少了搜索状态数。

由于是人机对弈游戏，考虑到玩家的游戏感受，AI 的决策时间不能过长，因此极小极大搜索不能搜索到叶子结点，而采用深度受限的启发式搜索，对所有结点设置评估函数来代替效用函数评价 AI 的得分，在较短时间内做出具有较大得分期望的决策。

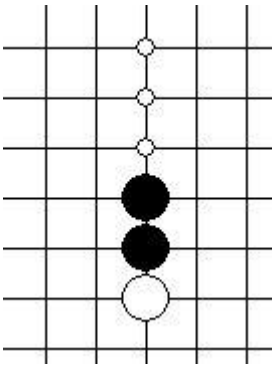
评估函数的设计：

找出与落子位置相连的本方棋子数量，根据形成的棋型来评价方案地期望得分。

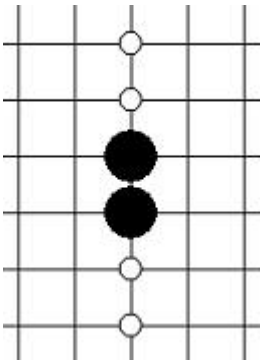
不同棋型的评估值如下：

①单子，5 分。即周围没有相连的本方棋子。

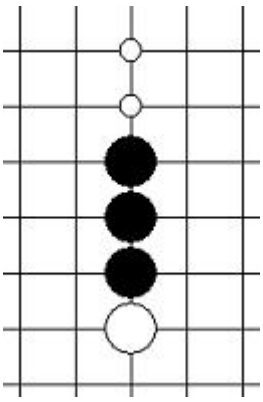
②死二，10 分，即如下图所示，一端被封死的二连子棋型。



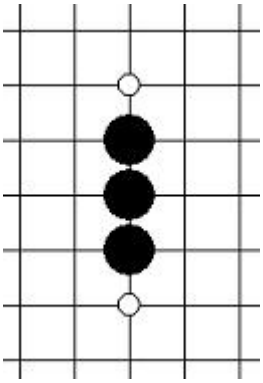
③活二，20 分，即如下图所示，没有被封死端的二连子棋型。



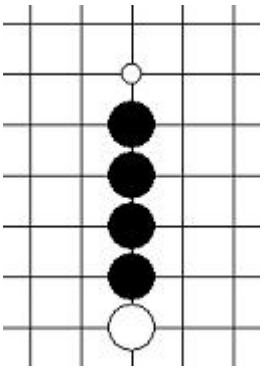
④死三，20 分，即如下图所示，一端被封死的三连子棋型。



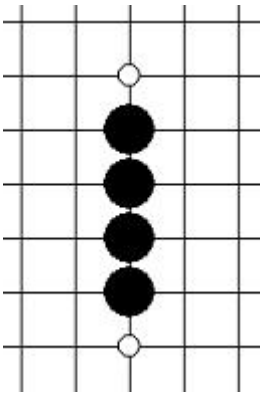
⑤活三，45 分，即如下图所示，没有被封死端的三连子棋型。



⑥死四，60 分，即如下图所示，一端被封死的四连子棋型。



⑦活四，120 分，即如下图所示，没有被封死端的四连子棋型。



⑧成五，300 分，即五连子棋型。

结点的评估函数为该结点落子位置在横、竖、两个斜向方向上所能形成的四种棋型的评估值。为了避免 AI 陷入固定的模式，以 0.8 的概率选择最大的两个评估值之和作为返回的评估值，0.15 的概率选择最大的三个评估值之和，0.05 的概率选择所有的四个评估值之和。

⑶ 模块设计：本实验的具体模块设计

①主循环模块：

通过计数器记录每一步轮到哪一方落子，并且在轮到玩家落子时设置键盘和鼠标监测来探知玩家的操作(落子、悔棋等)。在 AI/玩家落子后将新的棋盘局面呈现在屏幕上。在游戏开始时调用封面显示，在游戏结束时调用结束显示。

②评估模块：

对每个结点，通过对落子位置在横、竖、两个斜向方向上所能形成的四种棋型的检测，将其中最大的两个评估值相加作为评估函数，评价 AI 的得分期望。同时，有一定几率将其中最大的三/四个评估值相加作为评估函数来评价 AI 的得分期望，实现对同一局面可以作出不同的决策来避免被玩家使用特定棋路连续击败。

③搜索模块：

使用上文的深度受限的带 α - β 剪枝的极小极大搜索确定 AI 的最佳落子位置。

④记录模块

在每一次 AI/玩家落子后，更新对局步数，并将该步所下棋子的位置、颜色送入栈中记录，同时更新反映棋盘状态的矩阵。悔棋操作可以通过将栈顶两步落子位置弹出并将棋盘对应位置更新为无棋子实现。

⑤显示模块

显示游戏封面、背景、结束对话框等。同时在游戏过程中以高频率刷新棋盘状态的显示，在无操作时肉眼无法察觉到画面闪烁，而在悔棋/落子等操作时可以明显看出棋盘变化。

(4) 其他创新内容或优化算法

①落子位置优化：

事实上，由于极小极大搜索深度受限，得到的结果只能是眼下 AI 得分期望最大的落子选择，并不存在战略性的长远眼光。从这一点来看，对于那些离已落的棋子很远的棋盘位置显然是不符合 AI 得分期望最大这一要求的，如果每次落子都搜索整个棋盘，则会造成时间上的大量浪费。

因此可以对尝试的落子位置的范围进行缩小，对于每个结点，选择落子位置只在它对应的棋盘状态中存在的棋子的外切矩形的附近(宽度 1~2 格)和内部的范围之中。对于离外切矩形很远的位置，落子后所形成的棋型大部分都是单子，因此外切矩形附近和内部的落子位置评估很容易大于这些单子。通过将落子范围限制在已有棋子的外切矩形附近和内部，明显缩短了 AI 的决策时间。

同时存在一种特殊情况，棋盘的正中央很空。这时候优先下到棋盘的正中央，因为那里更不容易出界，从而存在更多的连子机会。

②搜索方式实现创新：

在实现深度受限的带 α - β 剪枝的极小极大搜索时，可以采用如下的递归实现的方式：

```python
# 负值极大算法搜索 alpha + beta剪枝
def negamax(is_ai, depth, alpha, beta):
    # 游戏是否结束 | | 探索的递归深度是否到边界
    if game_win(list1) or game_win(list2) or depth == 0:
        return evaluation(is_ai)
    blank_list = list(set(list_all).difference(set(list3)))
    order(blank_list)   # 搜索顺序排序  提高剪枝效率
    # 遍历每一个候选步
    for next_step in blank_list:
        global search_count
        search_count += 1
        # 如果要评估的位置没有相邻的子，  则不去评估  减少计算
        if not has_neightnor(next_step):
            continue
        if is_ai:
            list1.append(next_step)
        else:
            list2.append(next_step)
        list3.append(next_step)
        value = -negamax(not is_ai, depth - 1, -beta, -alpha)
        if is_ai:
            list1.remove(next_step)
        else:
            list2.remove(next_step)
        list3.remove(next_step)
        if value > alpha:
            print(str(value) + "alpha:" + str(alpha) + "beta:" + str(beta))
            print(list3)
            if depth == DEPTH:
                next_point[0] = next_step[0]
                next_point[1] = next_step[1]
            # alpha + beta剪枝点
            if value >= beta:
                global cut_count
                cut_count += 1
                return beta
            alpha = value
    return alpha
```

但是实际运行时，搜索耗时较长，玩家落子后 AI 需要的思考时间较长，对弈体验不佳。可以在限制深度较浅时将递归实现的方式改为循环嵌套的方式，如下：

```python
def ai_go():
    global min_x, max_x, min_y, max_y, color_flag, matrix
    time_start = time.time()
    evaluate_matrix = [[0 for i in range(SIZE + 2)] for j in range(SIZE + 2)]  # 结点估值矩阵
    if step != 0:
        if step == 1:
            #第一步下的位置
            if matrix[(SIZE + 1) // 2][(SIZE + 1) // 2] == 0:
                rx, ry = (SIZE + 1) // 2,(SIZE + 1) // 2
            else:
                if matrix[(SIZE + 1) // 2][(SIZE + 1) // 2] != 0 and matrix[(SIZE + 1) // 2 + 1][(SIZE + 1) // 2 + 1] == 0:
                    rx, ry = (SIZE + 1) // 2 + 1,(SIZE + 1) // 2 + 1
        else:
            min_tx1, min_ty1, max_tx1, max_ty1 = range_legal(min_x, min_y, max_x, max_y)
            evaluate_matrix = [[0 for i in range(SIZE + 2)] for j in range(SIZE + 2)]  # 第一层的估值矩阵
            Max = -100000
            rx, ry = 0, 0

            for i in range(min_tx1, max_tx1 + 1):
                for j in range(min_ty1, max_ty1 + 1):
                    cut_flag = 0  # 剪枝标记
                    evaluate_matrix2 = [[0 for i in range(SIZE + 2)] for j in range(SIZE + 2)]

                    if matrix[i][j] == 0:
                        matrix[i][j] = color_flag
                        min_tx2, min_ty2, max_tx2, max_ty2 = range_legal(min_tx1, min_ty1, max_tx1, max_ty1)
                        [list_h, list_v, list_s, list_b] = get_list(i, j, color_flag)
                        eva1 = evaluate(list_h, list_v, list_s, list_b)

                        for ii in range(min_tx2, max_tx2 + 1):
                            for jj in range(min_ty2, max_ty2 + 1):

                                if matrix[ii][jj] == 0:
                                    matrix[ii][jj] = -color_flag
                                    [list_h, list_v, list_s, list_b] = get_list(ii, jj, -color_flag)
                                    eva2 = -evaluate(list_h, list_v, list_s, list_b)

                                    evaluate_matrix2[ii][jj] = eva2 + eva1
                                    matrix[ii][jj] = 0
                                    # 剪枝
                                    if evaluate_matrix2[ii][jj] < Max:
                                        evaluate_matrix[i][j] = evaluate_matrix2[ii][jj]
                                        cut_flag = 1
                                        break
                            if cut_flag:
                                break

                        if cut_flag == 0:
                            Min = 100000
                            for ii in range(min_tx2, max_tx2 + 1):
                                for jj in range(min_ty2, max_ty2 + 1):
                                        if evaluate_matrix2[ii][jj] < Min and matrix[ii][jj] == 0:
                                            Min = evaluate_matrix2[ii][jj]

                            evaluate_matrix[i][j] = Min

                            if Max < Min:
                                Max = Min
                                rx, ry = i, j

                        matrix[i][j] = 0

        time_end = time.time()
        print("Time cost:", round(time_end - time_start, 4), "s")
        add_chess(rx, ry, color_flag)
```

从而大大节省了 AI 思考时间，同时 AI 的决策能力并没有出现明显下降，依然能够保持对玩家的较高水平。

③引入随机化因素避免 AI 被同一棋路击败

由于当 AI 的评估函数确定时，对于同一局面一定会得出相同的落子位置，从而存在某种棋路，使得玩家只要使用这种棋路 AI 就必败。因此在评估函数中引入随机化因素，除了选取四个方向上的棋型中分数最高的两个相加以外，还有一定几率选取最大的三个/四个相加。从而会在一些劣于严格取分数最高的两个相加评估出的最优落子位置的位置上落子，避免被玩家使用同一棋路连续击败。

三. 实验过程

⑴ 环境说明：

本实验使用 Windows 10 操作系统，使用 Python 语言在 Spyder(python3.7)平台上开发，核心使用库是 pygame、os、time、sys、 tkinter.messagebox 等。

⑵ 源代码文件清单，主要函数清单

main.py：程序源文件，实现了上述所有模块和算法。

def draw_background(surf)：绘制棋盘的网格线和棋盘的“星”

def xy_range(x, y)：刷新棋盘已占有棋子的外切矩形范围

def evaluate(list_h, list_v, list_s, list_b)：评估一个结点的分值，h、v、s、b 四个 list 分别对应得该结点在水平、竖直、左斜、反斜方向的四种棋型

def range_legal (_min_x, _min_y, _max_x, _max_y)：判断棋子位置是否超出棋盘

def ai_go()：使用深度受限的带 α - β 剪枝的极小极大搜索来决定 AI 落子的位置

def add_chess(x, y, color)：将新落到棋盘上的棋子的位置、颜色等信息入栈

def back_chess()：悔棋操作，将栈顶的两个棋子(即最近的 AI 和玩家的各一次落子)弹出

def draw_text(surf, text, size, x, y, color)：在游戏界面中打印出提示文字信息

def draw_movements(surf)：根据棋盘状态的矩阵重新显示游戏界面的棋盘状态，刷新频率高

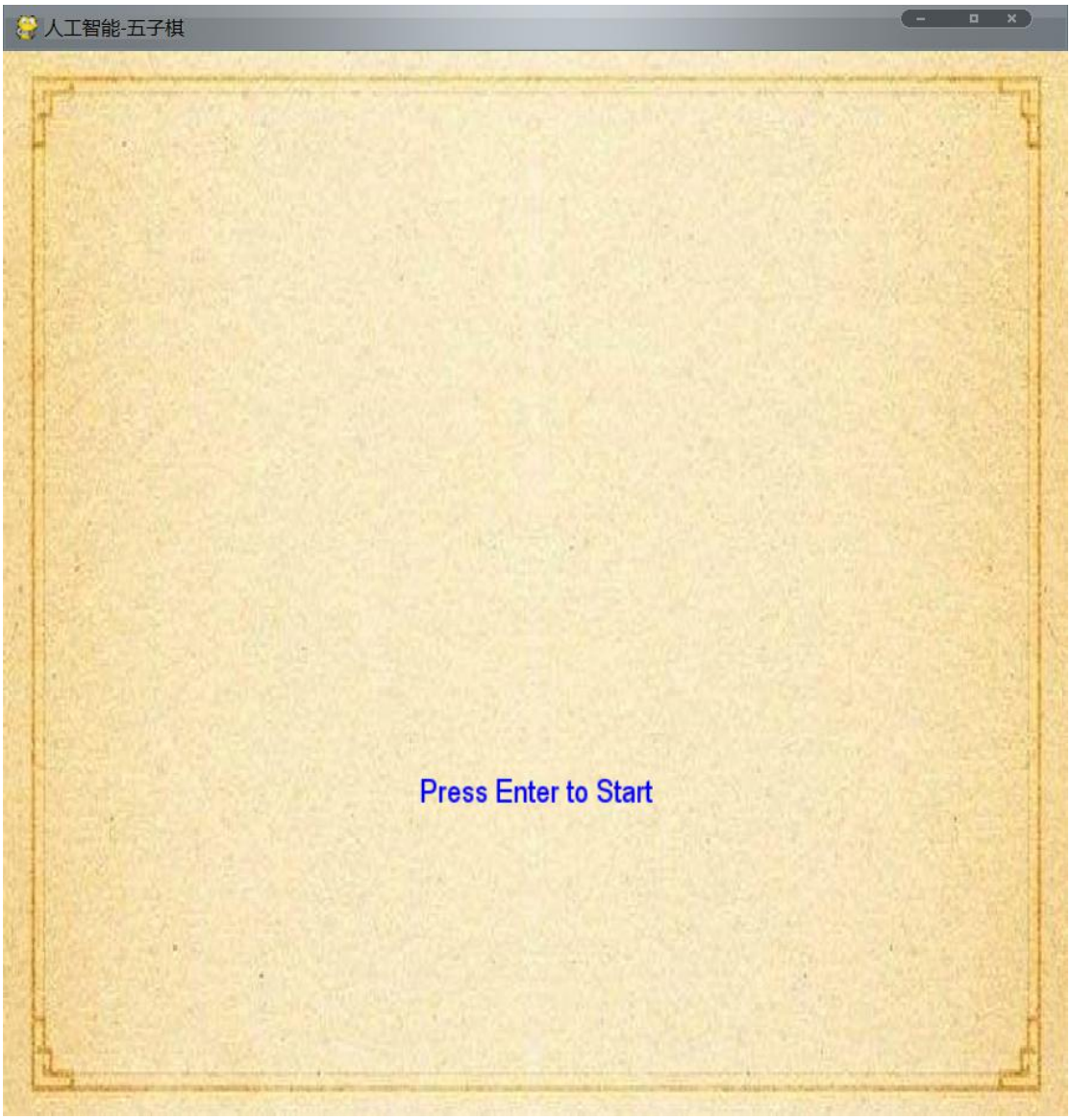
def player_go(pos)：通过鼠标点击的位置确定玩家的落子位置

def game_judge ()：判断是否存在五连子，游戏是否可以结束

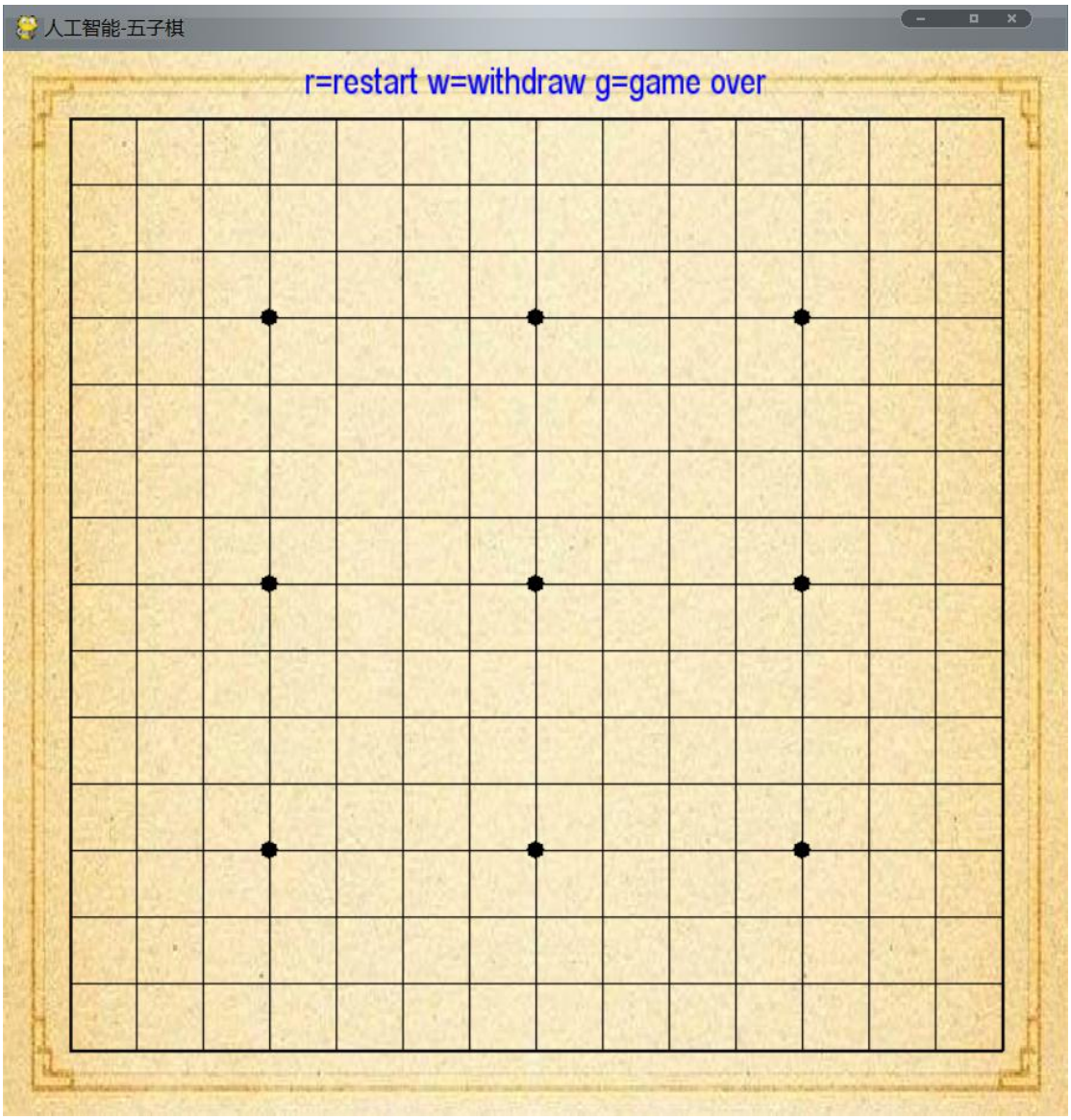
def init_UI (surf)：打印游戏封面，在游戏开始/重新开始时初始化棋盘状态和清空记录栈

⑶ 实验结果展示

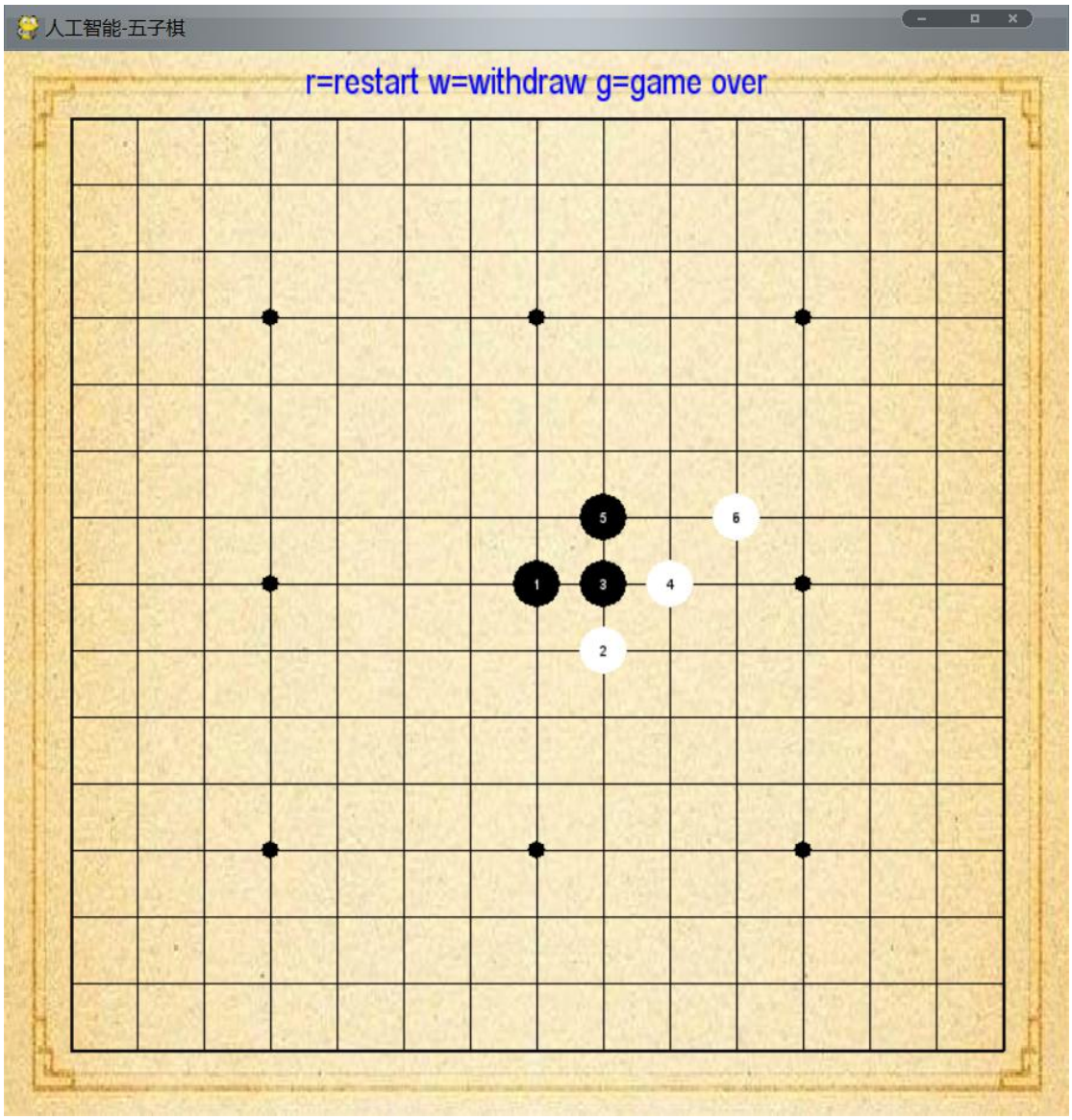
游戏开始封面：



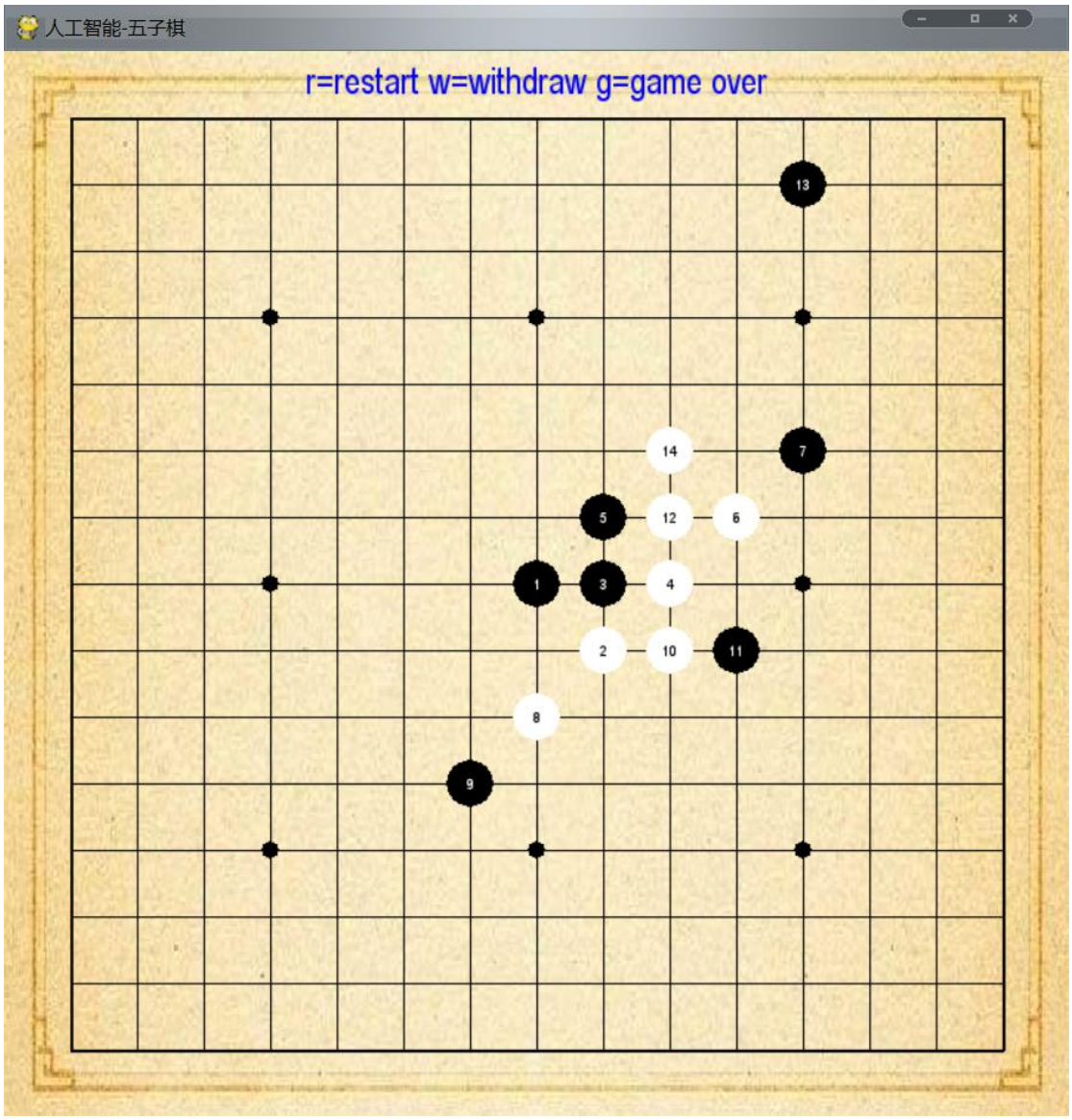
按下回车键后游戏正式开始(15*15 的棋盘正上方有按 R 键重新开始，按 W 键悔棋，按 G 键退出游戏的操作提示)：



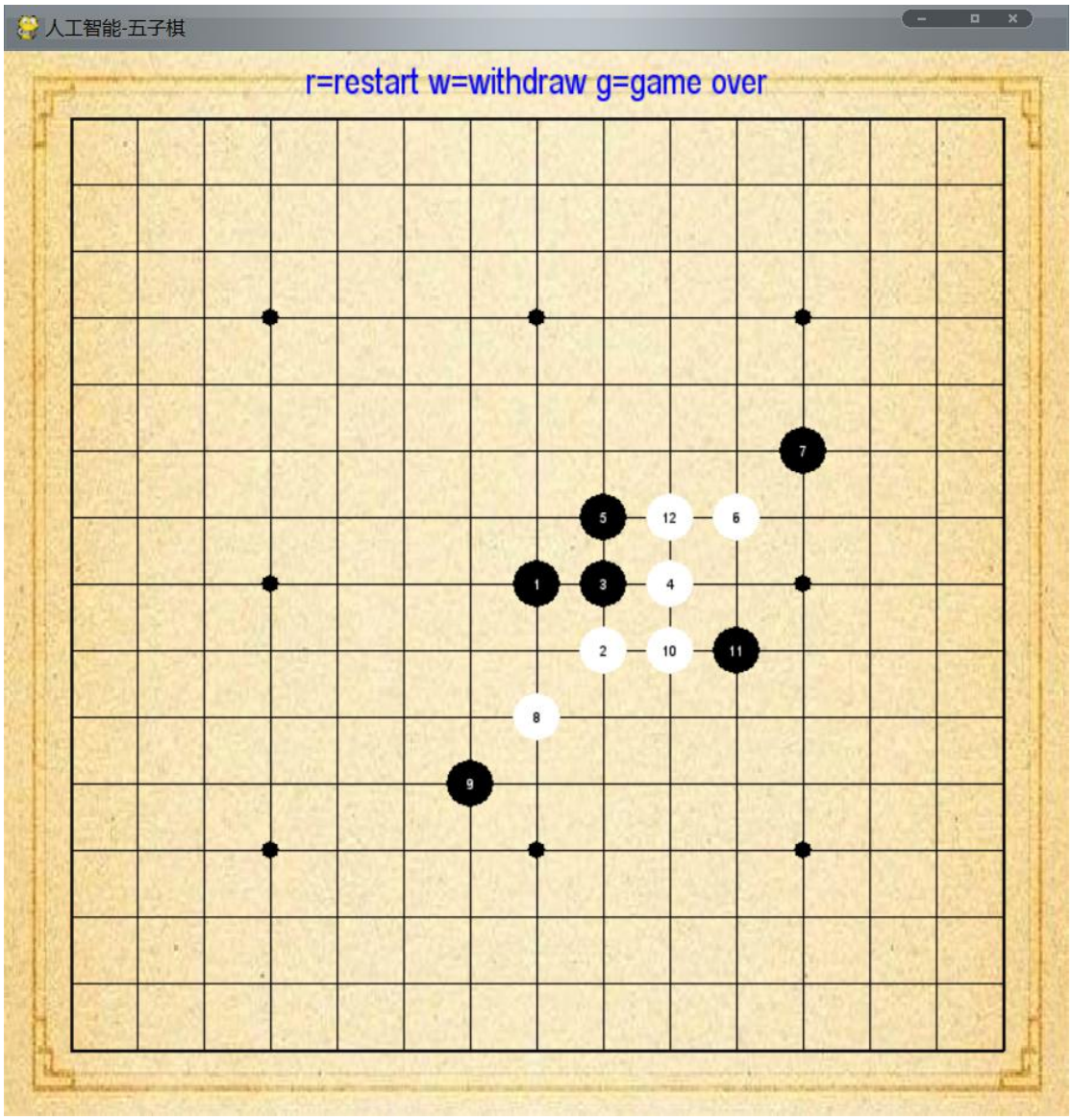
棋子上的数字代表了该位置是第几步下的：



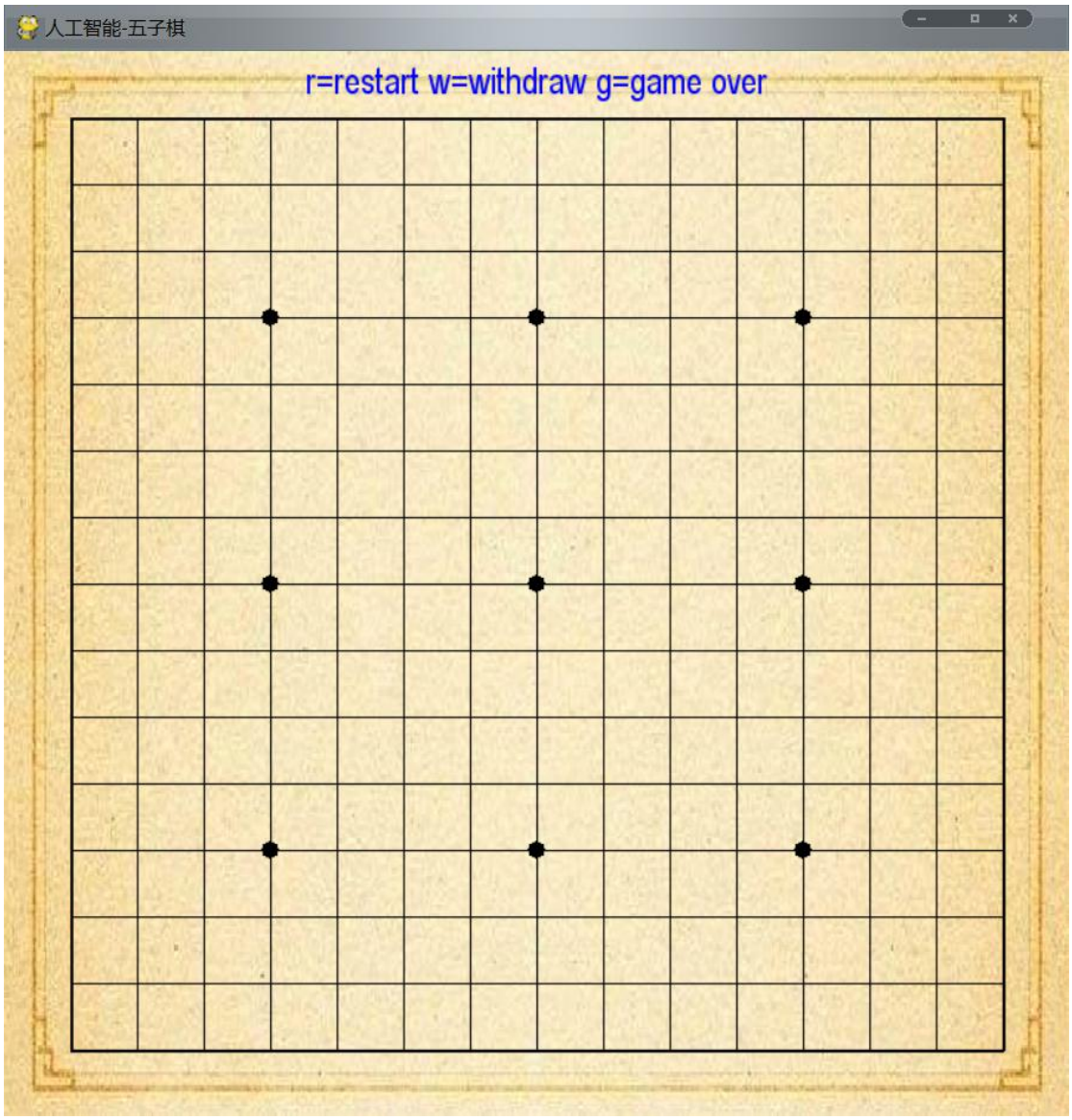
下错了，需要悔棋：


人工智能-五子棋
r=restart w=withdraw g=game over
13
14
7
5
12
6
1
3
4
2
10
11
8
9

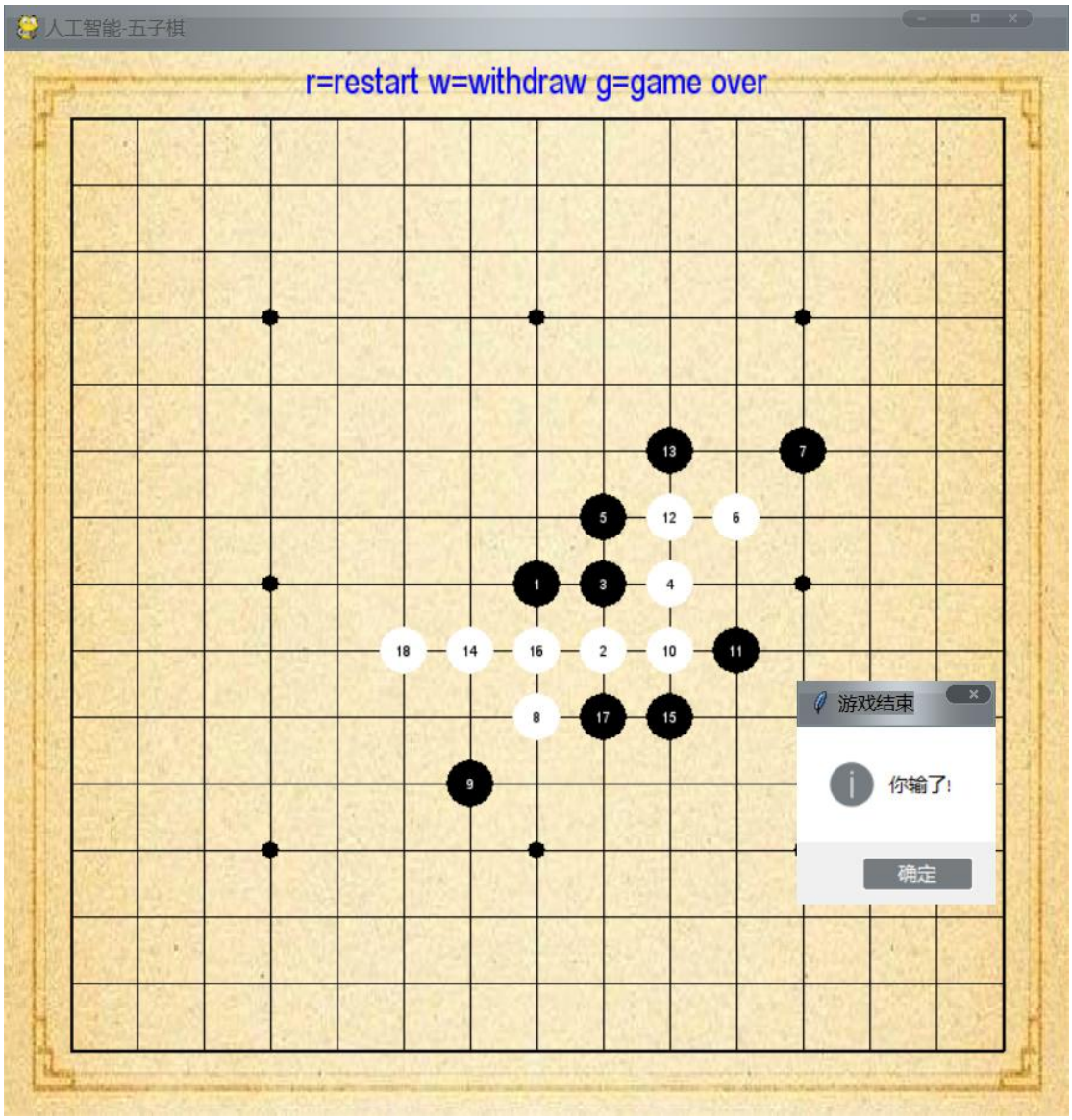
按 W 键悔棋：



按 R 键可以重新开始：



胜负已分的提示：



同时在控制台有游戏的选项设置以及每次 AI 落子所用的决策时间：

```
Console 1/A
Time cost: 0.0958 s
An exception has occurred, use %tb to see the full traceback.

SystemExit


In [4]:

In [4]: runfile('C:/Users/dell/Desktop/2.py', wdir='C:/Users/dell/Desktop')
Time cost: 0.0 s
Time cost: 0.1017 s
Time cost: 0.058 s
Time cost: 0.0748 s
Time cost: 0.0598 s
```

实验结论：

使用深度受限的带α-β剪枝的极小极大搜索设计出的五子棋 AI 具有较高程度的和人类玩家对弈的水平和能力，并且搜索用时短，决策速度较快。在引入了随机化评估函数选取机制后可以应对同一局面做出不同决策，避免被玩家某些棋路连续击败，更加具有人类思维的特点。

四. 总结

(1) 实验中存在的问题及解决方案

问题 1：原始程序效率低下
解决方案：原始程序搜索整个棋局范围，但是通过分析发现，如果下的棋子距离对手过远不会是一步好棋。于是我们将 AI 落子范围限定在已有棋子的外切矩形周边和内部，这样大大减少了搜索的范围。同时结合自身下棋的经验，将 AI 的第一步落在棋局的中央或者是中央附近，这样整体应该是有利的。

问题 2：悔棋时已经放上的棋子无法撤销显示
解决方案：通过高频率刷新游戏界面，在棋盘无变化时维持用户视觉感受，在棋盘变化时只需要修改棋盘状态矩阵就能显示出新的局面。

问题 3：当限制搜索深度时，使用递归方式实现带α-β剪枝的极小极大搜索较慢，玩家落子后 AI 思考时间较长
解决方案：在确定了限制深度时改用循环嵌套的方式替代递归实现，虽然不易修改限制深度，但换来了更快的 AI 思考时间，且较浅的搜索深度已经足以对抗人脑的决策，实现人机博弈。

问题 4：AI 出招套路比较固定，针对同一个局面的，AI 的落子点相同。如果对弈的人反复尝试，可能比较容易找出战胜 AI 的方式，而且之后利用这个套路可以一直战胜 AI。
解决方案：修改 AI 对于棋局的评估函数，相同棋局下，AI 有一定的概率采取其他的评估函数。这种概率比较小，类似于突变。这样 AI 大体上是按照同一种的评估方法，以此保证整体评估方式的一致性,小概率采取其他评估方式，使得棋招更具有多变性。

(2) 后续改进方向

①关于五子棋的棋型还存在更多复杂的种类(如眠三等)，通过引入更多的棋型，可以进

一步提高 AI 决策时的最优程度

②加大搜索深度，使 AI 变得更有长远眼光，并且能够识破玩家的一些埋 伏性棋路，能够进一步提升人机对弈的难度

③可以使用深度学习的方法训练 AI，这样 AI 可以通过不断的训练，获得很好的效果。甚至可以通过用深度学习训练的 AI 来与剪枝算法的 AI 进行对抗，帮助我们调整剪枝算法的评估函数，各种棋局类型的估分。

④可以通过多线程方式 MTD(F)或者 PVS，这样可以加快 AI 搜索的速度，加大搜索的深度，获得更加好的决策效果。