中山大学计算机学院

人工智能

本科生实验报告

（2023 学年春季学期）

课程名称：Artificial Intelligence

教学班级	**刘永梅老师班级**	专业（方向）	**信息与计算科学**
学号	**22336313**	姓名	**郑鸿鑫**

一、 实验题目

利用 Alpha-Beta 剪枝算法实现五子棋残局闯关

二、 实验内容

1. 算法原理

Alpha-Beta 剪枝算法的基本原理是“剪掉”那些明显不会对最终决策产生影响的分支。具体来说，算法使用两个参数 α（alpha）和 β（beta）来记录到目前为止找到的最佳移动的评分。α 代表最大化玩家（通常是 AI）的最佳选择的最低评分，而 β 代表最小化玩家（对手）的最佳选择的最高评分。

在搜索树的每一层，Alpha-Beta 算法执行以下步骤：

1.初始化：在搜索开始时，α 设置为负无穷大（对于最大化玩家），β 设置为正无穷大（对于最小化玩家）。

递归搜索：算法递归地搜索博弈树的每一层。在每一层，算法首先为最大化玩家找到可能的最佳移动，并更新 α 值。然后，为最小化玩家找到可能的最佳移动，并更新 β 值。

2.剪枝：在递归过程中，如果在某一层的搜索中，α 值大于或等于 β 值，那么算法可以停止在这一层搜索剩余的分支。这是因为即使这些分支被完全搜索，它们的评分也不会比当前已知的最佳移动更好。这意味着搜索可以跳过这些分支，从而节省时间和计算资源。

3.回溯：当搜索回溯到上一层时，算法将当前层的最佳评分传递给上一层，作为上一层搜索的 α 或 β 值的一部分。

4.选择最佳移动：当搜索完成时，α 和 β 中的较大值（对于最大化玩家）或较小值（对于最小化玩家）将代表当前局面的最佳移动。

Alpha-Beta 剪枝算法之所以有效，是因为它避免了对那些不可能改善当前已知最佳选择的分支进行搜索。通过这种方式，算法可以更快地找到最佳移动，特

别是在搜索深度较大时，效率提升尤为明显。然而，需要注意的是，Alpha-Beta 剪枝并不改变搜索的完备性，它只是加速了搜索过程。

2. 伪代码

Alpha-Beta 剪枝函数伪代码：

```
Procedure AlphaBetaSearch(board, isblack, alpha, beta, depth, max_depth, matrix)
    // 检查是否达到搜索深度或棋盘已满
    If depth = max_depth Or is_board_full(board) Then
        Return (-1, -1), evaluate(board), matrix
    EndIf

    // 初始化最佳落子位置和评分
    best_move, best_score <- 初始化值

    // 遍历所有可能的后继状态
    For each (x, y, next_board) In get_successors(board, 1 if isblack else 0) Do
        // 递归搜索下一个状态
        _, _, move_score, _ <- AlphaBetaSearch(next_board, not isblack, alpha,
beta, depth + 1, max_depth, matrix)

        // 更新 alpha 和 beta 以及最佳落子位置
        If isblack And move_score > best_score Then
            alpha <- max(alpha, move_score)
            best_move, best_score <- (x, y), move_score
        Else If not isblack And move_score < best_score Then
            beta <- min(beta, move_score)
            best_move, best_score <- (x, y), move_score
        EndIf

        // 剪枝
        If isblack And alpha >= beta Then Break
        If not isblack And beta <= alpha Then Break
    EndFor

    // 更新矩阵评分
    matrix[best_move[0]][best_move[1]] <- best_score

    // 返回最佳落子位置和评分
    Return best_move, best_score, matrix
EndProcedure
```

评估函数伪代码：

```
Procedure is_board_full(board)
    For each row In board Do
        If EMPTY Is In row Then
            Return False
        EndIf
    EndFor
    Return True
EndProcedure
Function get_value(coordinate)
    x, y <- coordinate
    Return value[x][y]
EndFunction
Procedure generate_pattern(color)
    // 定义棋型模式和对应的得分
    p1 <- (-1, color, color, -1)
    // ... 其他棋型定义省略，遵循同样的模式
    p15 <- (color, color, color, color, color)
    // 收集所有棋型及其得分到字典中
    pattern <- {p1: (50, 30), p2: (50, 30), ..., p15: (9999999, 9999999)}
    Return pattern
EndProcedure
Function evaluate(board)
    color <- 1
    other <- 0
    pattern_c <- generate_pattern(color)
    pattern_o <- generate_pattern(other)
    Score <- 0
    judge <- 0
    // 计算己方棋型得分
    For each (pattern, score) In pattern_c Do
        If score[0] >= 3000 Then
            judge <- judge + 1
        EndIf
        Score <- Score + count_pattern(board, pattern) * score[0]
    EndFor
    // 计算对方棋型得分并从己方得分中扣除
    For each (pattern, score) In pattern_o Do
        If score[1] >= 3000 Then
            judge <- judge - 1
        EndIf
        Score <- Score - count_pattern(board, pattern) * score[1]
    EndFor
    // 位置权重评分
```

```
    For m From 0 To 14 Do
        For n From 0 To 14 Do
            If board[m][n] = color Then
                Score <- Score + get_value((m, n))
            Else If board[m][n] = other Then
                Score <- Score - get_value((m, n))
            EndIf
        EndFor
    EndFo
    // 检查特殊胜利条件
    If abs(judge) >= 2 Then
        Return judge * 20000 + Score
    EndIf
    Return Score
EndFunction
```

3. 关键代码展示（带注释）

Alpha-Beta 剪枝函数：

```python
def AlphaBetaSearch(board, isblack, alpha, beta, depth, max_depth,matrix):
    # 如果达到搜索深度上限或者棋盘已满，则返回评估值
    if depth == max_depth or is_board_full(board):
        return -1, -1, evaluate(board),matrix
    # 初始化最佳落子位置和评分
    best_move = (-1, -1)
    best_score = float('-inf') if isblack else float('inf')
    # 生成所有可能的后继状态
    for x, y, next_board in get_successors(board, 1 if isblack else 0):
        # 递归调用 AlphaBetaSearch
        _, _, move_score,_ = AlphaBetaSearch(next_board, not isblack, alpha,
beta, depth + 1, max_depth,matrix)
        # 如果当前评分更好，更新最佳落子位置和评分
        if isblack and move_score > best_score:
            best_score = move_score
            best_move = (x, y)
            alpha = max(alpha, best_score)
            # matrix[x][y] = best_score
            if alpha >= beta:  # 剪枝条件
                break
        elif not isblack and move_score < best_score:
            best_score = move_score
            best_move = (x, y)
            beta = min(beta, best_score)
            # matrix[x][y] = best_score
            if beta <= alpha:  # 剪枝条件
                break
    matrix[best_move[0]][best_move[1]] = best_score
    return best_move[0], best_move[1], best_score,matrix
```

辅助函数 1（判断棋盘是否下满）：

```python
def is_board_full(board):
    for row in board:
        if EMPTY in row:
```

```
            return False
    return True
```

辅助函数 2（给棋盘上每个点分配权重，并以此为搜索优先级）：

```
value = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
         [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
         [0, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 0],
         [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 1, 0],
         [0, 1, 2, 3, 4, 4, 4, 4, 4, 4, 4, 3, 2, 1, 0],
         [0, 1, 2, 3, 4, 5, 5, 5, 5, 5, 4, 3, 2, 1, 0],
         [0, 1, 2, 3, 4, 5, 6, 6, 6, 5, 4, 3, 2, 1, 0],
         [0, 1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1, 0],
         [0, 1, 2, 3, 4, 5, 6, 6, 6, 5, 4, 3, 2, 1, 0],
         [0, 1, 2, 3, 4, 5, 5, 5, 5, 5, 4, 3, 2, 1, 0],
         [0, 1, 2, 3, 4, 4, 4, 4, 4, 4, 4, 3, 2, 1, 0],
         [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 1, 0],
         [0, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 0],
         [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
#获取位置权值
def get_value(coordinate):
    x, y = coordinate[0], coordinate[1]
    return value[x][y]
```

辅助函数 3（生成某一颜色的特殊棋型）：

```
def generate_pattern(color):
    p1 = (-1,color,color,-1)
    p2 = (-1,color,-1,color,-1)
    p3 = (color,color,-1,color,-1)
    p4 = (color,color,color,-1)
    p5 = (-1,color,color,color)
    p6 = (-1,color,color,color,-1)
    p7 = (-1,color,-1,color,color,-1)
    p8 = (-1,color,color,-1,color,-1)
    p9 = (color,color,color,-1,color,color)
    p10 = (color,color,-1,color,color)
    p11 = (color,-1,color,color,color)
    p12 = (color,color,color,color,-1)
    p13 = (-1,color,color,color,color)
    p14 = (-1,color,color,color,color,-1)
    p15 = (color,color,color,color,color)
    #字典的键为某种特殊棋型，值为一个元组，元组第一个元素为黑棋对应得分，第二个元素为白棋对
应得分
    pattern = {p1:(50,30),p2:(50,30),p3:(200,100),p4:(500,300),p5:(500,300),

p6:(5000,3000),p7:(5000,3000),p8:(5000,3000),p9:(6000,4000),p10:(6000,4000),

p11:(6000,4000),p12:(6000,4000),p13:(6000,4000),p14:(100000,80000),p15:(9999
999,9999999)}
    return pattern
```

辅助函数 4（评价函数，用于评估盘面对黑棋的分数）：

```
def evaluate(board):
    color = 1
    other = 0
    # 对黑白棋各生成一套 pattern
    pattern_c = generate_pattern(color)
    pattern_o = generate_pattern(other)
    Score = 0
    judge = 0 #用于判定特殊胜利条件
    #匹配黑棋各种特殊棋型并加分
    for pattern,score in pattern_c.items():
```

```
        if score[0] >= 3000:
            judge += 1
        Score += count_pattern(board,pattern) * score[0]
    #匹配白棋的各种特殊棋型并减分
    for pattern,score in pattern_o.items():
        if score[1] >= 3000:
            judge -= 1
        Score -= count_pattern(board,pattern) * score[1]
    # 位置权重评分
    for m in range(15):
        for n in range(15):
            if board[m][n] == color:
                Score += get_value((m,n))
            elif board[m][n] == other:
                Score -= get_value((m,n))
    if abs(judge) >= 2:
        return judge * 20000 + Score
    return Score
```

对给出的匹配 pattern 辅助函数中，只对 get_successors()函数进行了修改，代码如下：

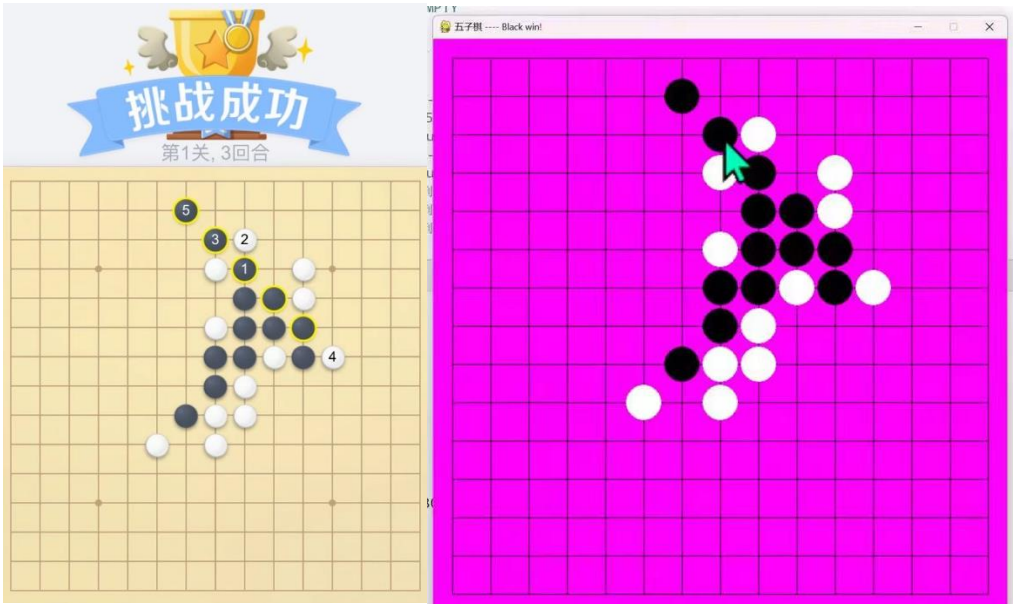
```
def get_successors(board, color, priority=get_value, EMPTY=-1):
    #以 get_value 作为搜索的优先级，实现从中间到四周的搜索顺序
    # 注意：生成器返回的所有 next_board 是同一个 list!
    from copy import deepcopy
    next_board = deepcopy(board)
    idx_list = [(x, y) for x in range(15) for y in range(15)]
    idx_list.sort(key=priority, reverse=True)
    for x, y in idx_list:
        if board[x][y] == EMPTY:
            next_board[x][y] = color
            yield (x, y, next_board) # 生成器
            next_board[x][y] = EMPTY
```

三、 实验结果及分析

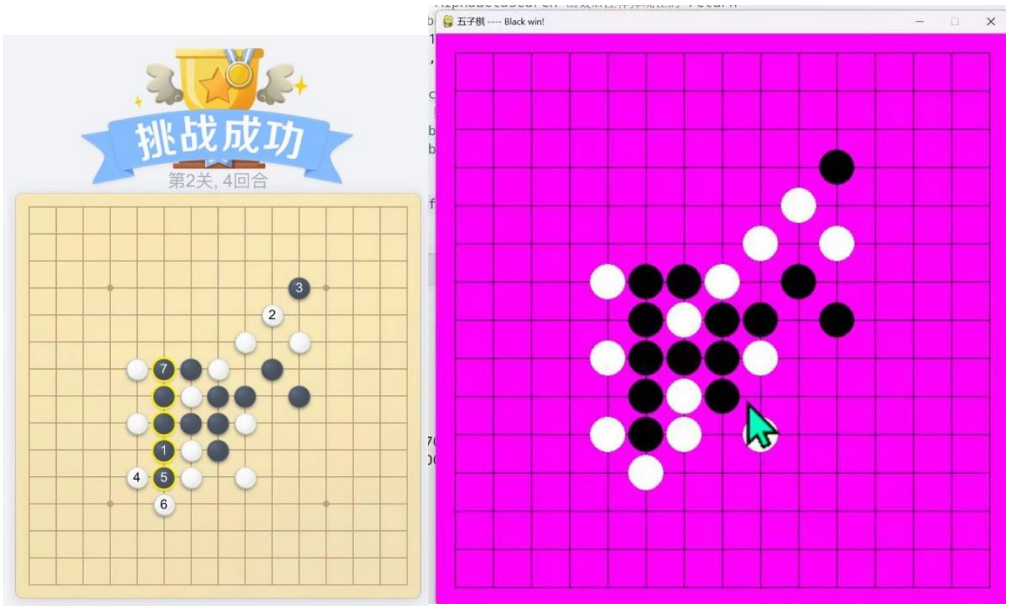
1. 实验结果展示示例

展示样例较多，详见 Result 文件夹。

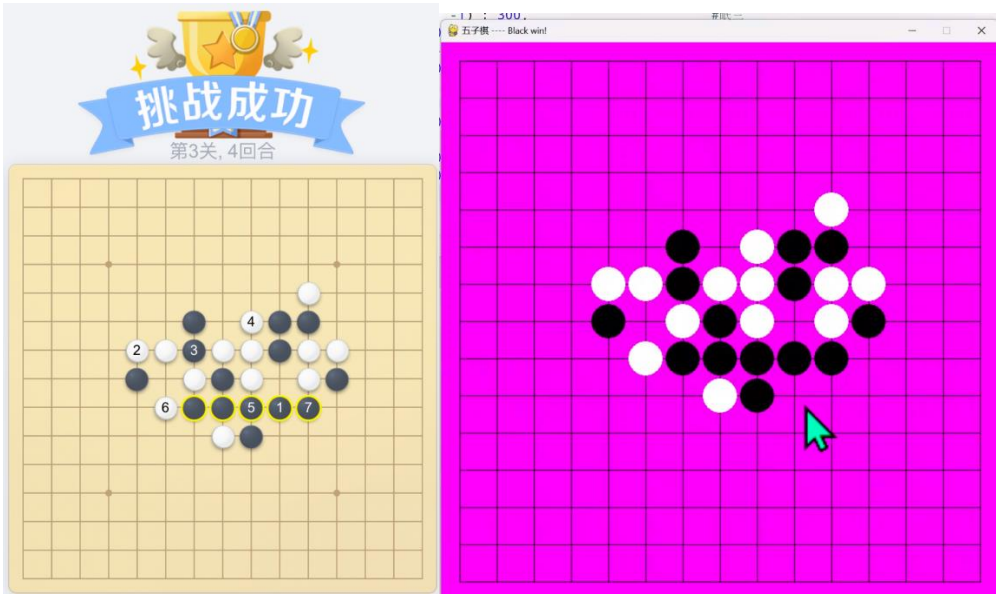
第一关顺利通过结果：



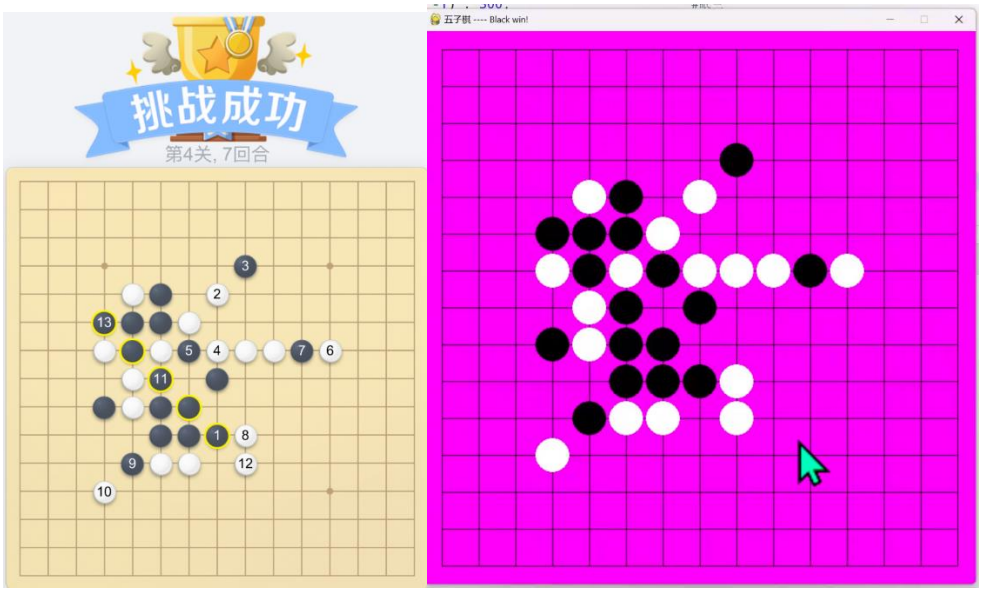
第二关顺利通过结果：



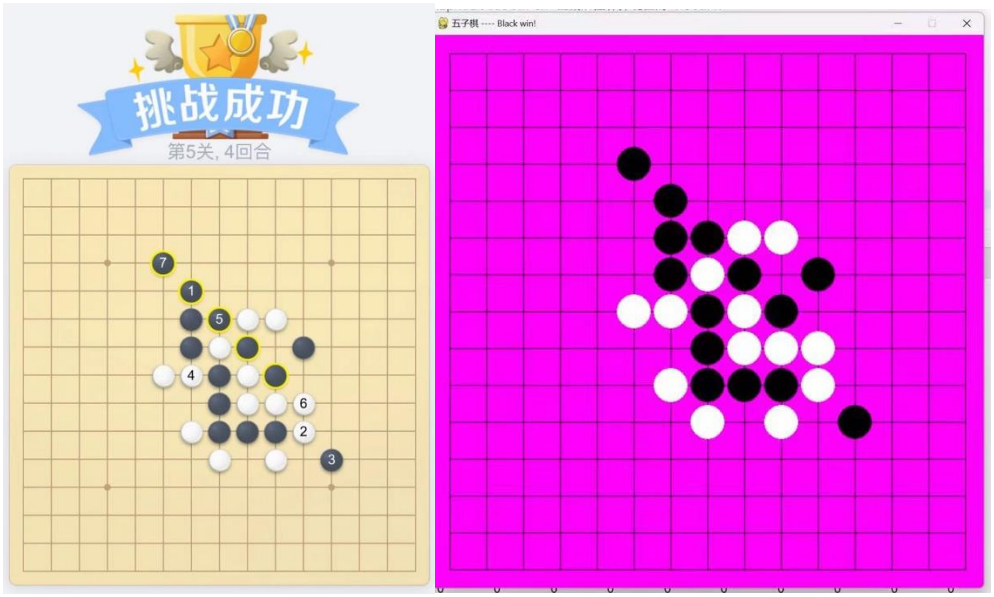
第三关顺利通过结果：



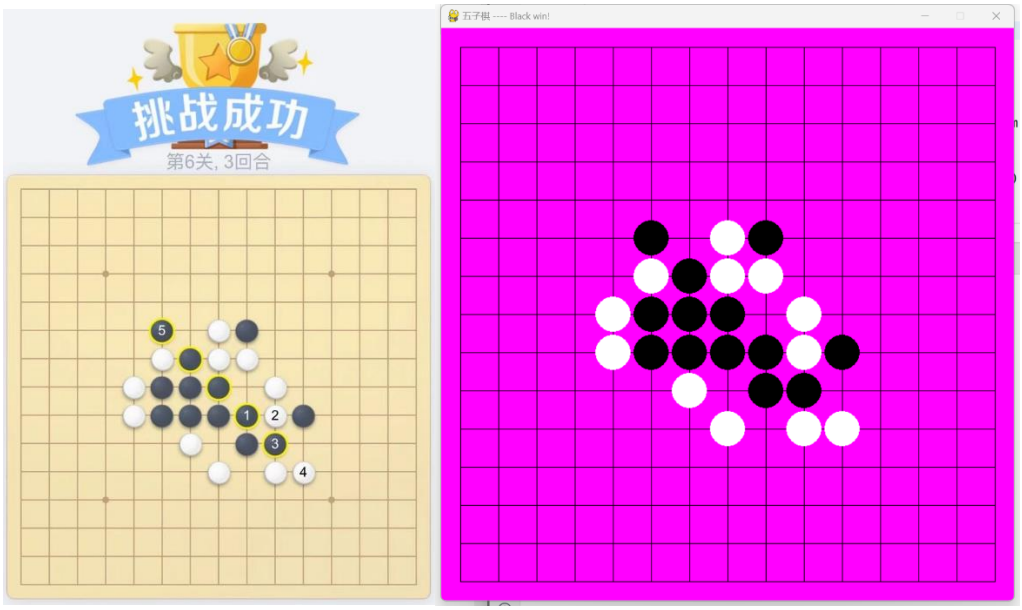
第四关顺利通过结果：



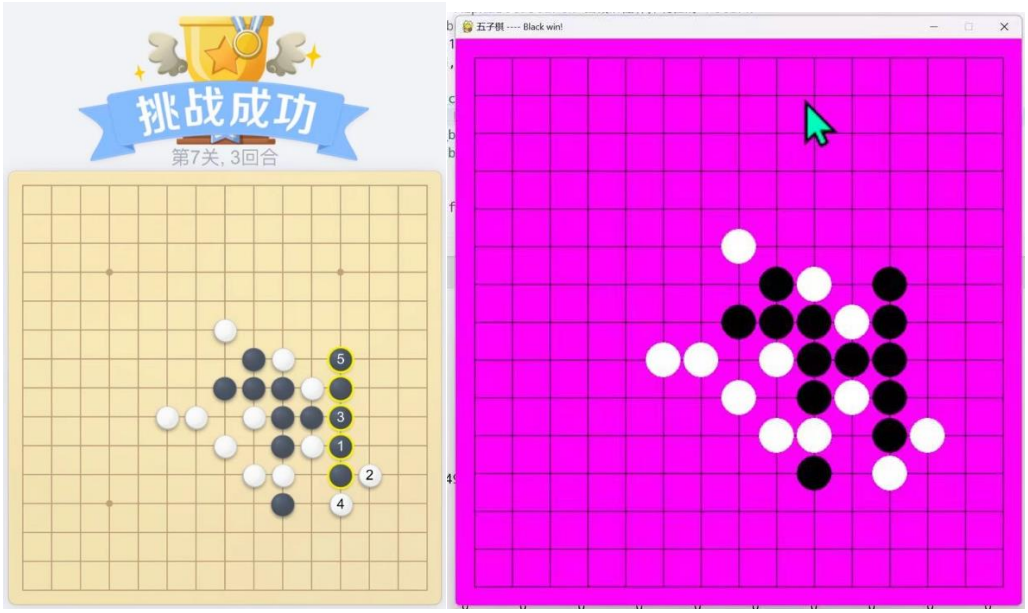
第五关顺利通过结果：



第六关顺利通过结果：

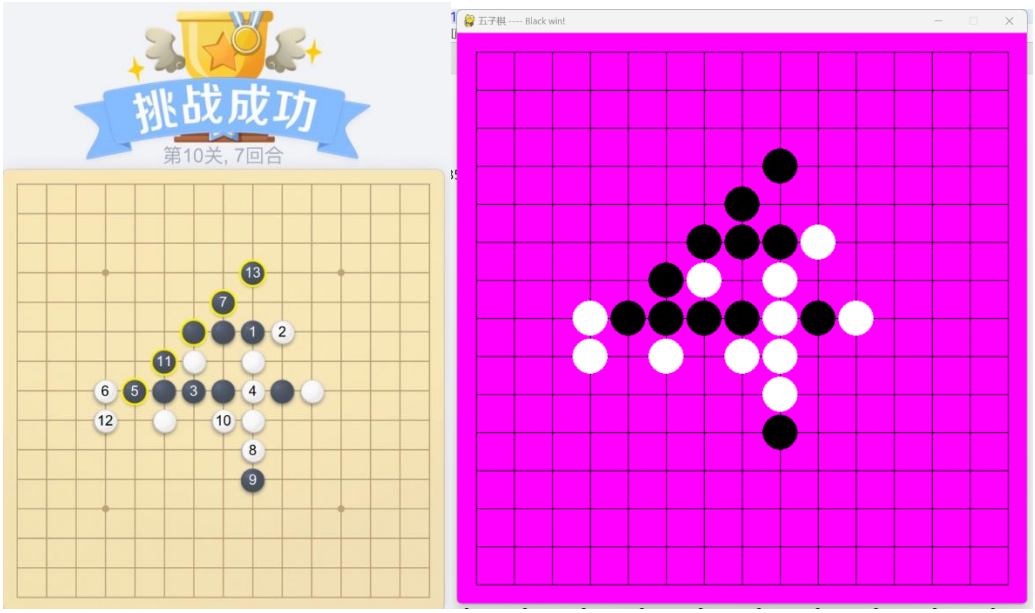


第七关顺利通过结果：

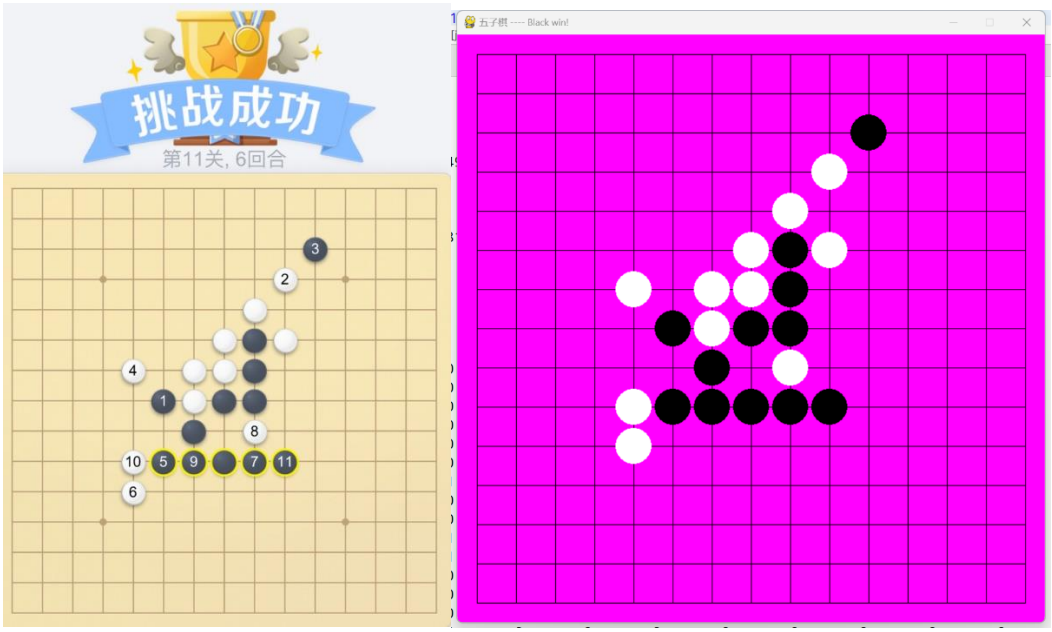


第八第九关通关失败

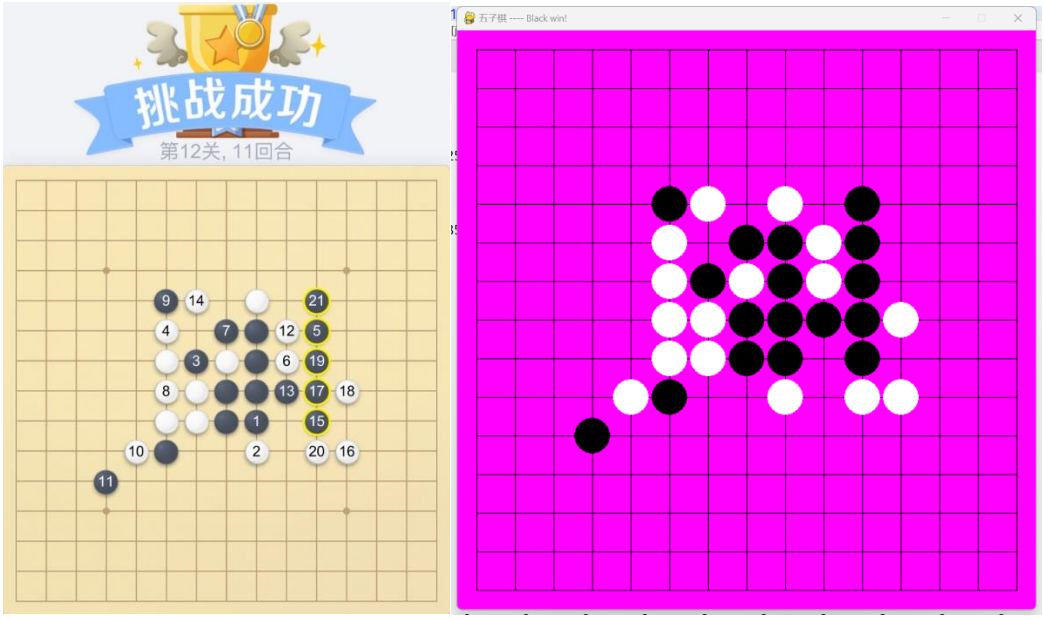
第十关顺利通过结果：



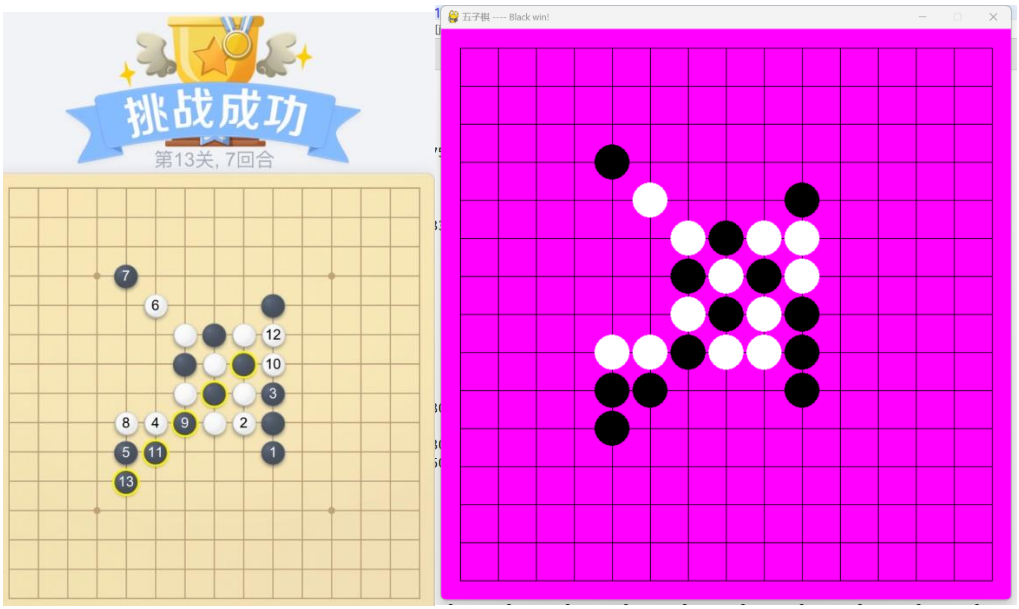
第十一关顺利通过结果：



第十二关顺利通过结果：

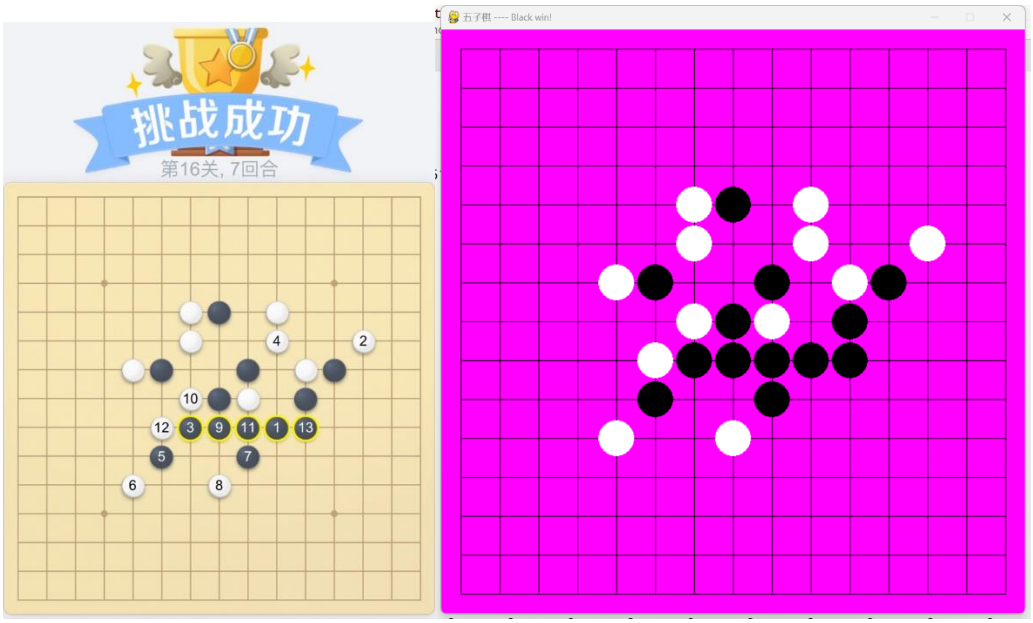


第十三关顺利通过结果：

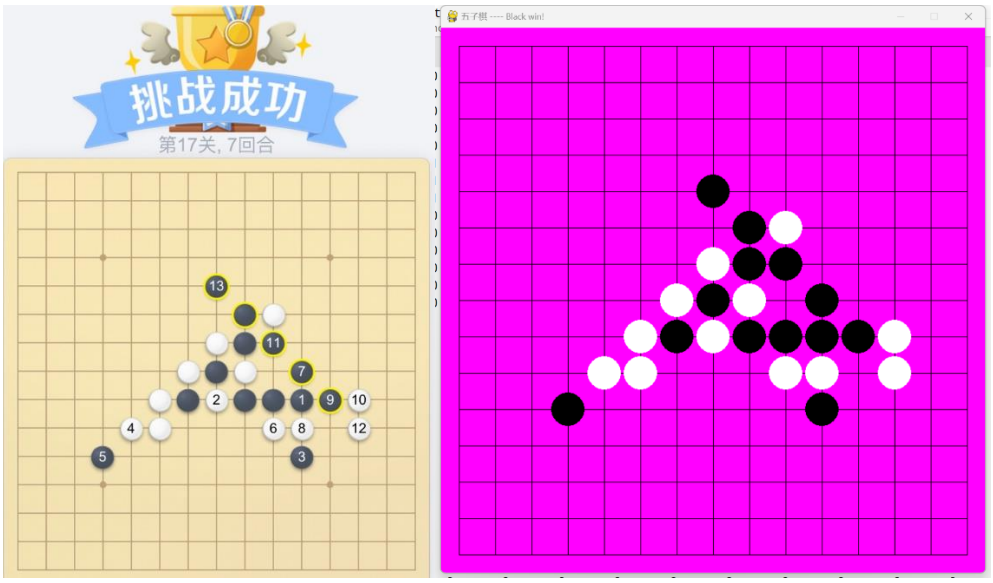


第十四关，十五通关失败

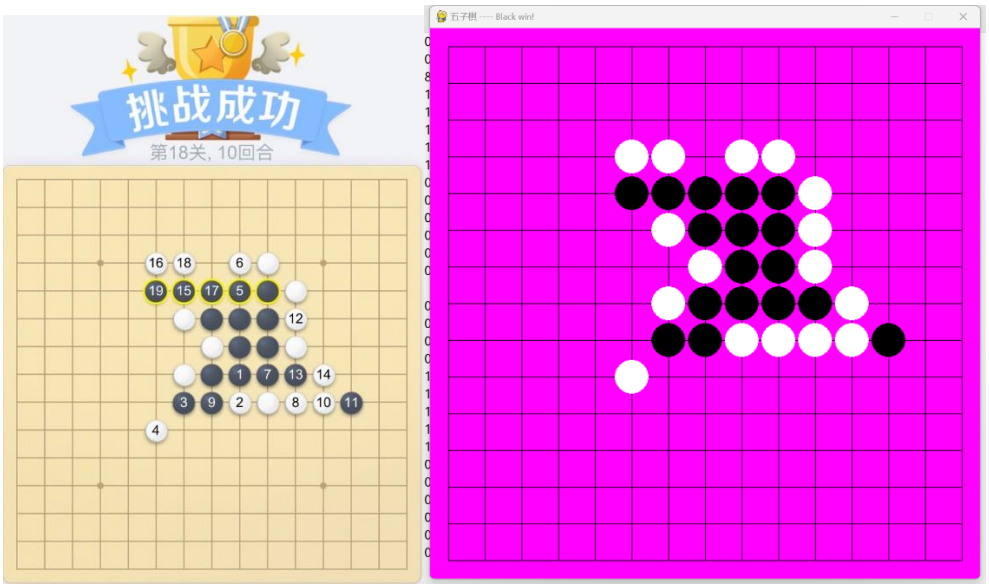
第十六关顺利通过结果：



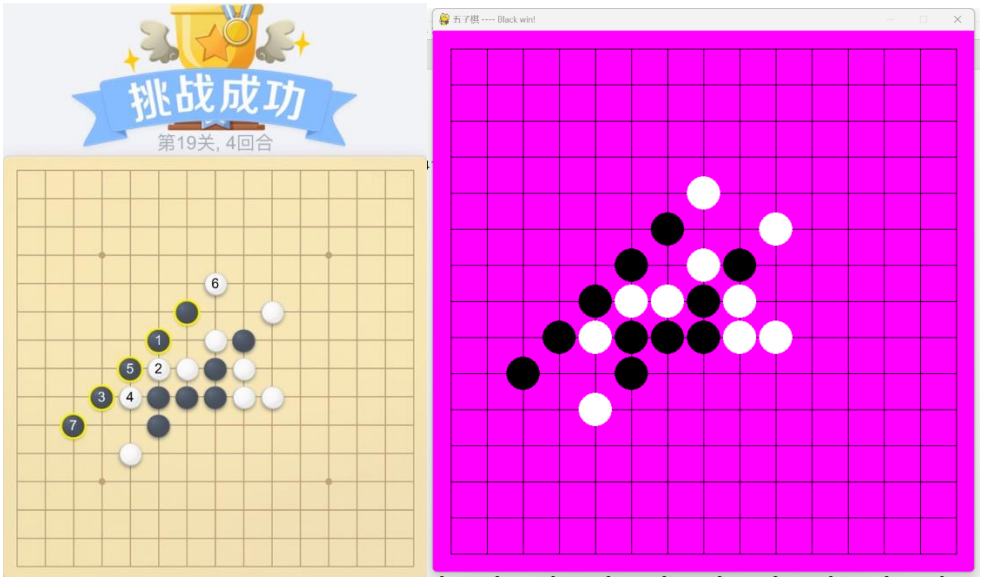
第十七关顺利通过结果：



第十八关顺利通过结果：



第十九关顺利通过结果：

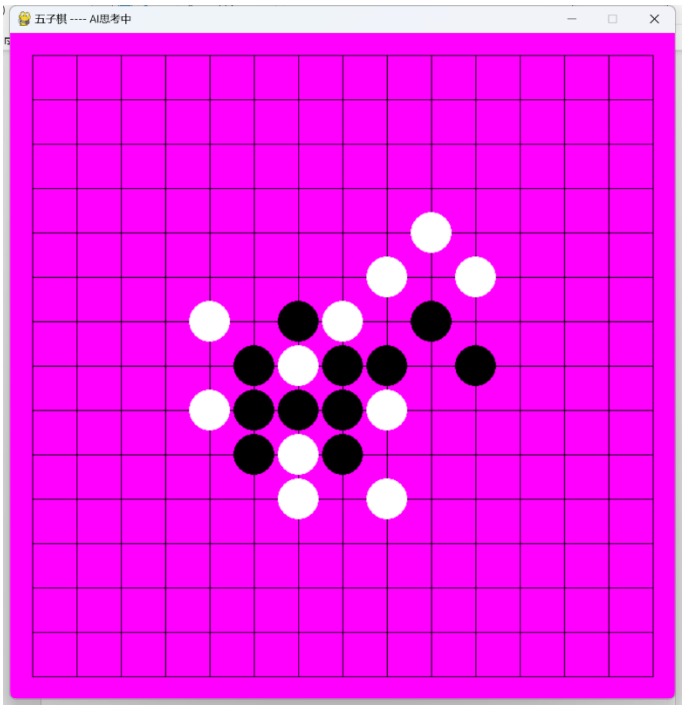


第二十关通关失败

2. 剪枝步骤分析：

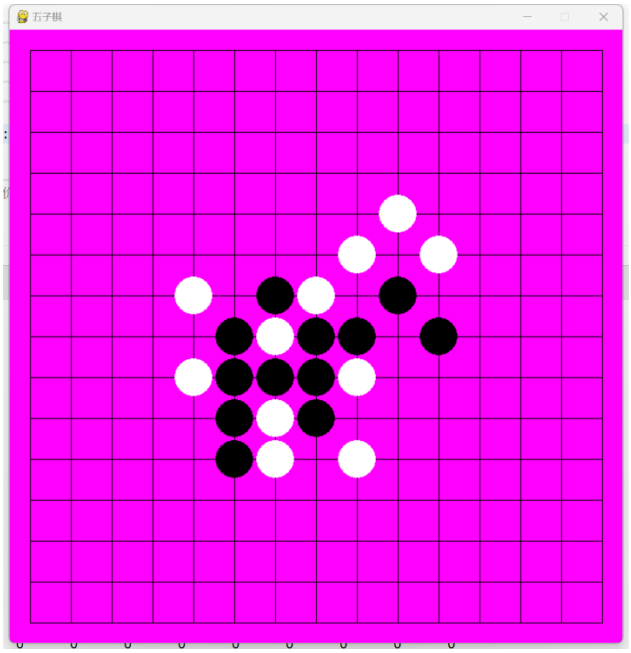
分析例子 1

以第二关中的一步防守为例：

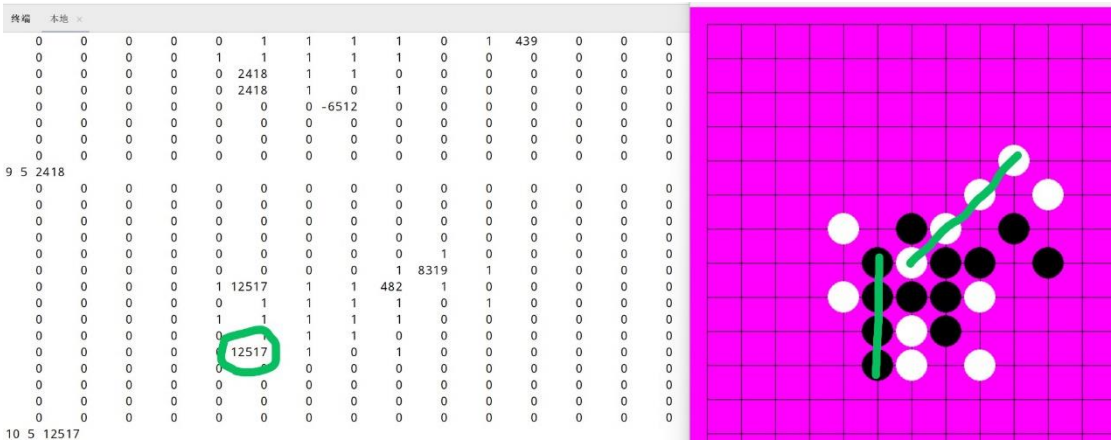


在此情况下，下一步黑棋只有一种选择，那就是去防守白棋的连冲四，否则将会直接输掉，在前导的版本中，黑棋会贪图自己的

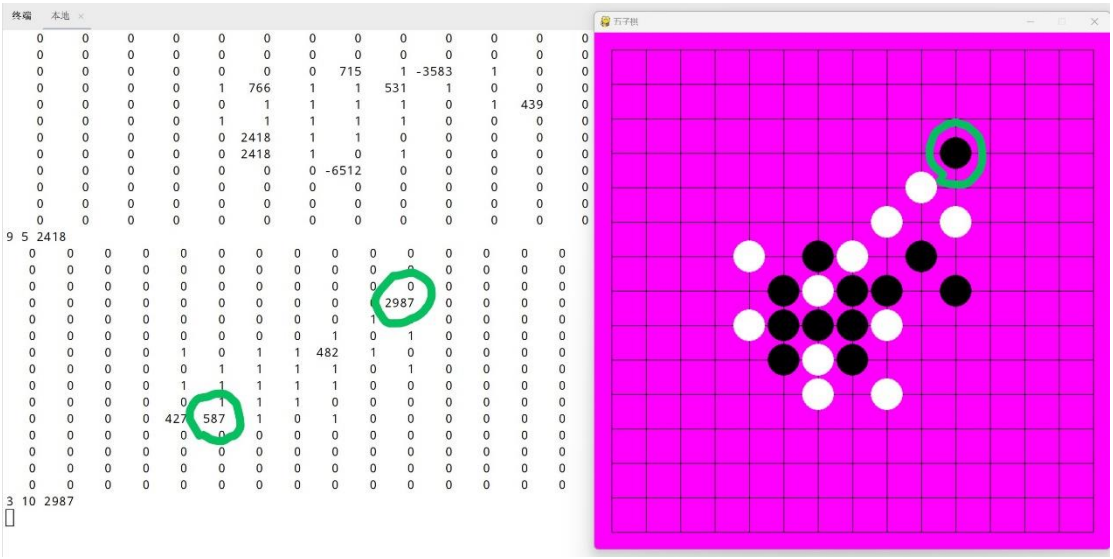
活四而放弃防守：（如下图所示）



通过打印得分矩阵进行调试：（其中 1 代表该位置已有棋子，0 代表还未对其评分，有分数则说明已经给出评分）



可以看到此时继续进攻，形成活四的分数达到了 12507，而且已经进行剪枝，可以看到对白棋进行防守的位置上是 0，即还没被遍历到。通过调整评估函数，把各棋型的分数进行调整：

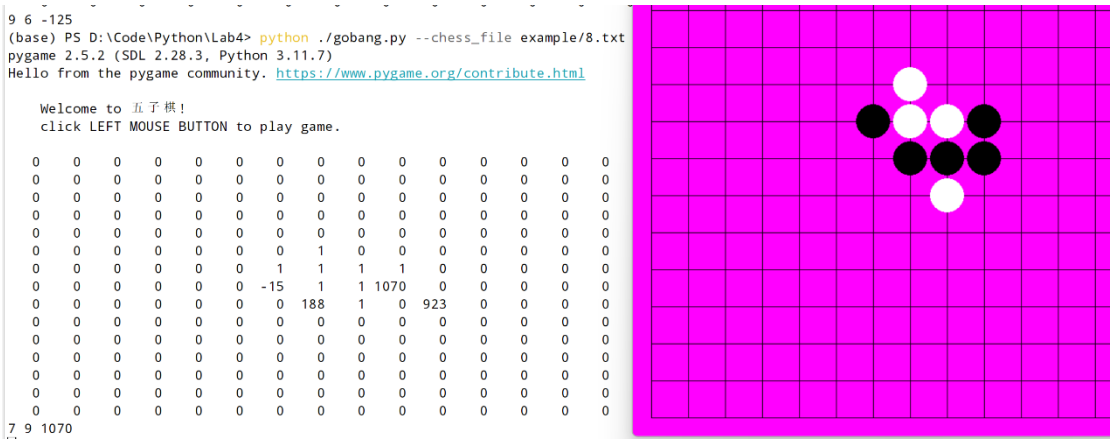


通过调整，使得在防守位置落子的得分超过了形成活四位置的落子的得分分别为 2987 和 587，从而会选择在正确的位置上落子。

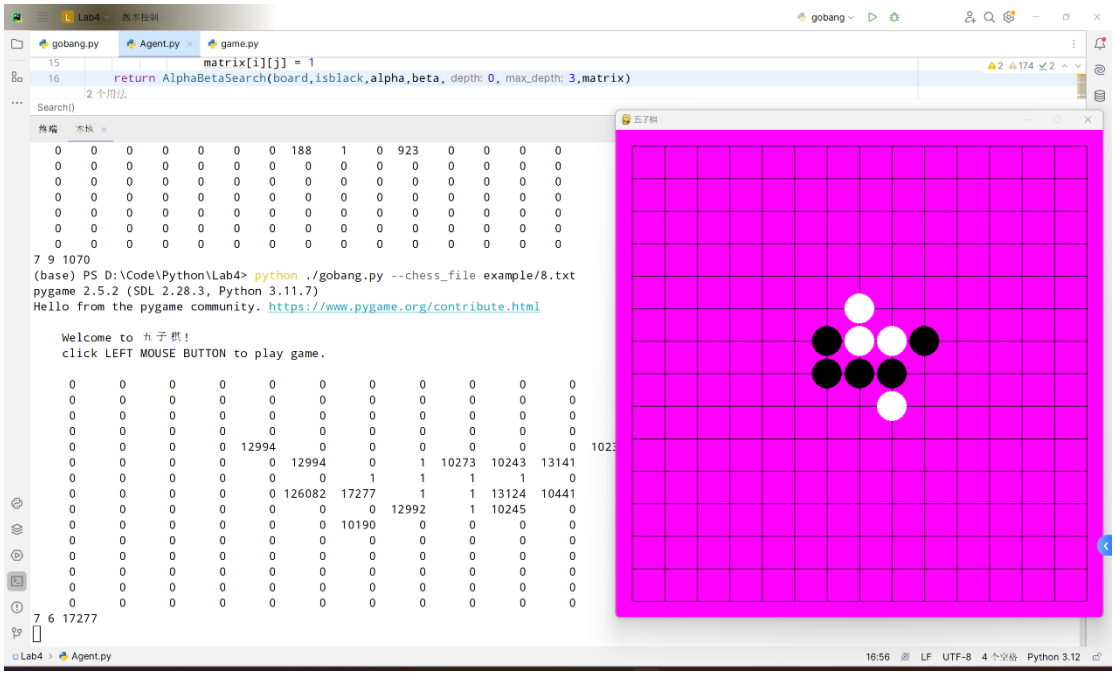
分析例子 2：（未解决的问题）

在修改分值和修改搜索优先级后顺利通过了前 7 关，而对于第 8 关则是在十几手后输掉，故先尝试自己通过第 8 关，再分析为什么 AI 没有过关，过关的正确落子顺序如下：

通过测试后发现，AI 很难找到这条最佳路径，最大的原因是在第一步开始就错误：

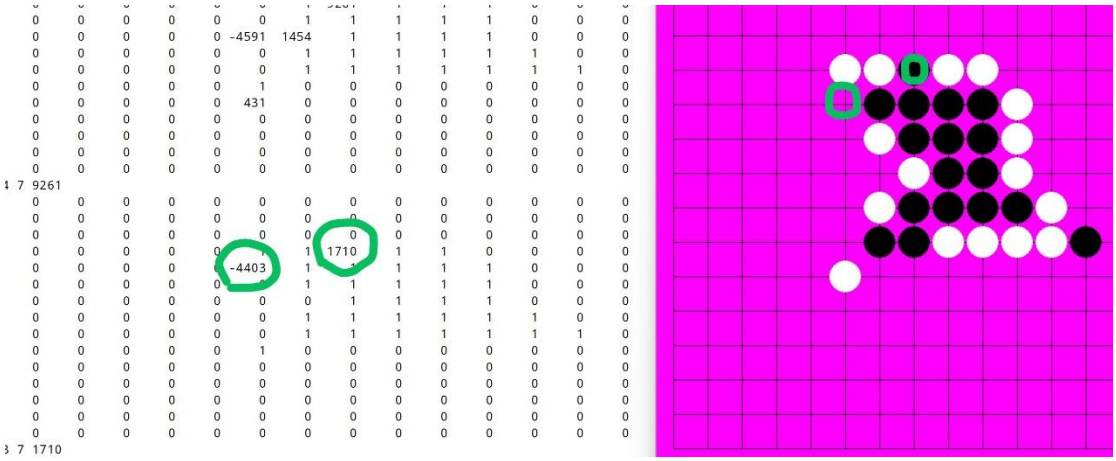


因为按照棋型的匹配，眠三确实不如活三，所以在这个位置得分较高且剪枝并落子，并没有逻辑上的错误。为了解决这个问题，我们尝试将递归深度加到 3 层以让 AI 察觉到能顺利过关的路径，遗憾的是由于深度加深，复杂度为指数级别，AI 下一步花费的时间会很久，超出了可以接受的范围，并且可以看到，仍然没有找到正确的落子位置。



分析例子 3：

在对第十八关进行测试的时候发现，在修改完防守的 bug 后，AI 对进攻和防守之间的倾向还是不够好，如下图所示：

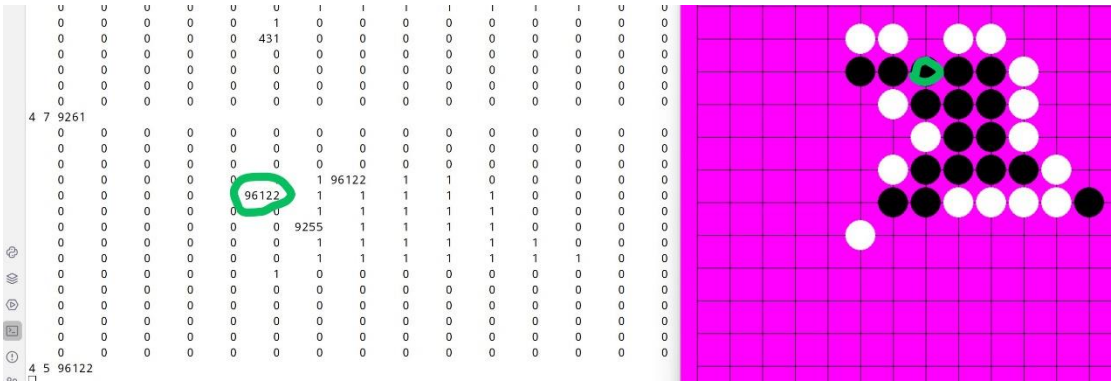


这一步本来可以直接形成五连，但是可以看到输出的矩阵中，进攻得到五连和防守对方的冲四的分数分别为-4403 和 1710。导致错失赢的机会。经过检查后发现，可以通过修改打分细则改善：

```
p15:(999999,899999)
```

将 paterrn 中第 15 个，也就是五子连珠的分数从原来双方都赋 999999

改为黑方赋 999999，白方赋 899999，修改后重新测试，查看矩阵和落子情况：



可以看见修改后正确落子并通关。

四、 参考资料

1. 使用 Min-Max 搜索和启发式评估函数实现五子棋 AI - 知乎 (zhihu.com)
2. python 中 yield 的用法（生成器的讲解） python yield-CSDN 博客
3. 基于 alpha-beta 剪枝技术的五子棋 - 知乎 (zhihu.com)
4. 【断奶班】五子棋零基础自学一本通（转载） - klchang - 博客园 (cnblogs.com)