

# 中山大學



## 中山大学计算机学院人工智能 实验报告

题 目	:	博弈树搜索 Alpha-beta剪枝
教学班级	:	20230349
姓 名	:	张超
学 号	:	22336290
专 业	:	计算机科学与技术（系统结构）
日 期	:	2024/04/16

# 实验题目

编写一个五子棋博弈程序，要求用Alpha-beta剪枝算法，实现人机对弈。

## 实验内容

### 算法原理

#### alpha-beta剪枝算法

##### 基本原理

- 博弈树搜索**: 在博弈树中，每个节点代表一种游戏局面，节点的子节点代表在该局面下可能的下一步行动。通过递归地搜索这棵树，可以找到最好的下一步行动。
- 最大化和最小化节点**: 在博弈中，轮到某一方行动时，该方的目标是最大化自己的收益；而对手的目标是最小化该方的收益。因此，每个节点可以分为最大化节点和最小化节点。
- 剪枝**: 在搜索过程中，可以通过剪枝来减少搜索的分支。Alpha-beta剪枝算法通过维护两个值，alpha和beta，来实现剪枝。

#### Alpha-beta剪枝算法步骤

- 递归搜索**: 从根节点开始递归搜索博弈树。在搜索过程中，每层交替进行最大化和最小化。
- 更新alpha和beta**: 在搜索的同时，维护两个值，alpha和beta。Alpha表示最大化节点的最佳值，Beta表示最小化节点的最佳值。在搜索过程中，当发现某一分支的值已经超出了当前节点的最优值范围时，可以剪枝。
- 剪枝条件**: 当最小化节点的beta值小于等于最大化节点的alpha值时，说明最小化节点不会选择该分支，因为对它来说已经有更好的选择了；当最大化节点的alpha值大于等于最小化节点的beta值时，说明最大化节点不会选择该分支。
- 递归终止条件**: 当搜索到达叶子节点或达到设定的搜索深度时，停止递归。

#### 优势和应用

- 减少搜索空间**: Alpha-beta剪枝算法能够有效地减少搜索的节点数量，从而在相同的时间内搜索更深的树，提高搜索效率。
- 广泛应用**: 这个算法在许多博弈游戏的人工智能中被广泛应用，例如国际象棋、围棋等。通过这种方式，Alpha-beta剪枝算法在搜索博弈树时能够显著提高效率，使得计算机能够在合理的时间内找到较好的行动策略。

#### ac自动机算法

AC自动机（Aho-Corasick自动机）是一种用于多模式串匹配的高效算法，可以在一个主串中同时查找多个模式串的出现位置。该算法以其高效的时间复杂度和内存利用率而广泛应用于字符串匹配领域，比如关键词过滤、字符串搜索等。

## 基本原理

1. **构建trie树**: 首先将所有模式串构建成一颗trie树（字典树），并为每个节点添加一个指向其父节点的fail指针。
2. **构建失败指针**: 接着，对trie树进行遍历，在每个节点上设置fail指针，该指针指向当前节点的父节点的fail指针所指向的节点的子节点中与当前节点字符相同的节点，若没有则继续沿着fail指针向上回溯直至找到根节点。这个过程可以使用广度优先搜索（BFS）来实现。
3. **搜索过程**: 在搜索过程中，从文本串的第一个字符开始，按照trie树的结构进行匹配。如果当前字符匹配失败，就根据fail指针转移到下一个状态继续匹配，直到达到trie树的叶子节点或文本串的末尾。
4. **匹配输出**: 当达到trie树的叶子节点时，说明找到了一个模式串的匹配，可以记录下该模式串在文本串中的位置或进行其他操作。

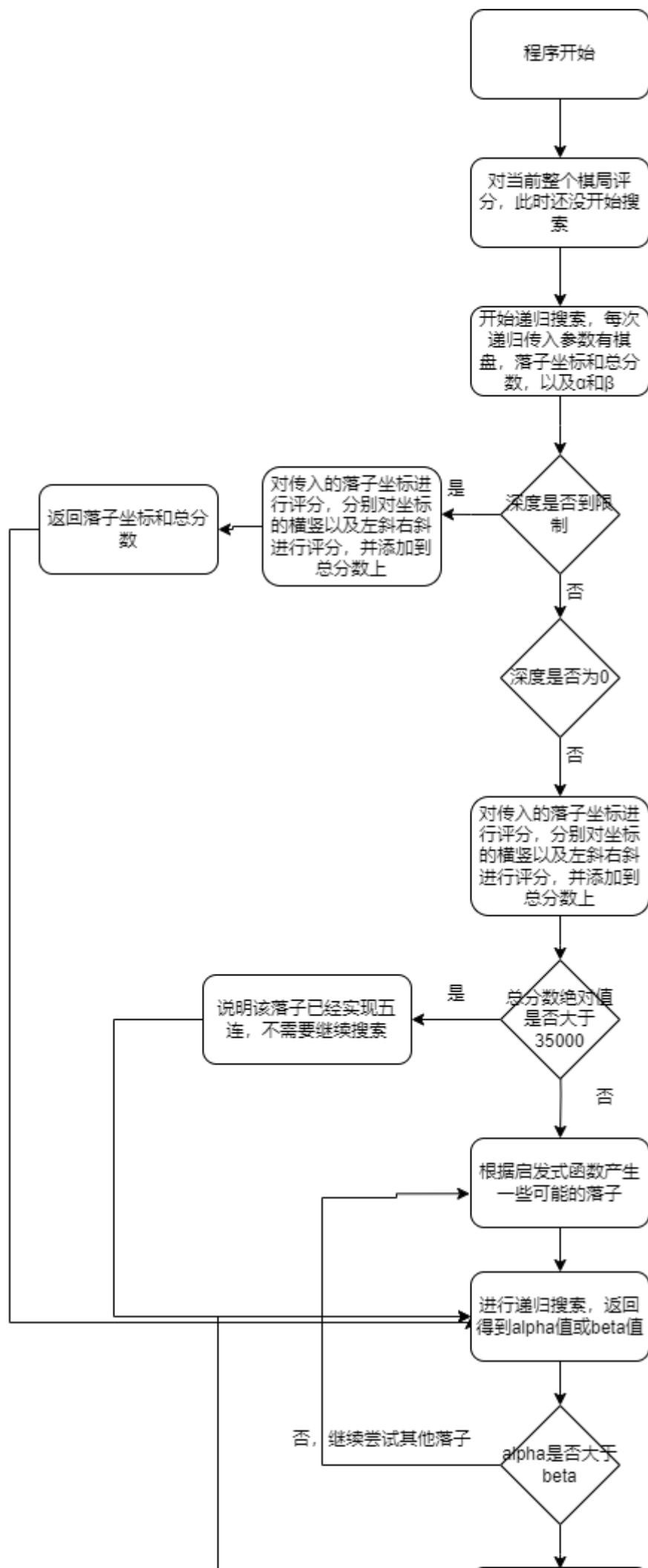
## AC自动机的优势和应用

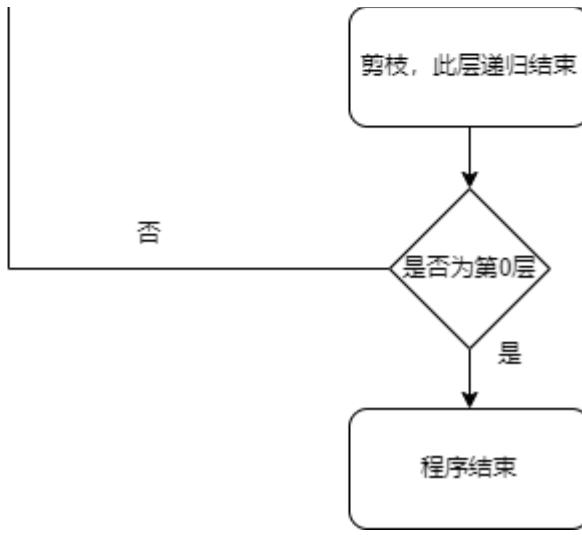
- **高效性**: AC自动机的搜索时间复杂度为 $O(n+m+k)$ ，其中n为文本串长度，m为模式串总长度，k为所有匹配结果的数量。相比传统的暴力匹配算法，AC自动机具有更高的效率。
- **多模式串匹配**: AC自动机适用于多模式串匹配，可以同时在一个文本串中查找多个模式串的出现位置，因此在字符串匹配、关键词过滤等场景中有广泛应用。  
通过构建trie树和失败指针，AC自动机算法实现了高效的多模式串匹配，是字符串匹配领域的重要算法之一。在五子棋的博弈算法中，需要对相应的棋形进行打分，这里就涉及到模式串匹配问题，并且每次打分的棋形都有可能不同，每次需要打分的棋形有多种，这就涉及到多模式串匹配。

## 流程图

---

下面为alpha-beta剪枝算法的流程图





简而言之，alpha-beta剪枝算法的实现是利用一个递归函数，该递归函数维护alpha和beta两个值，利用递归函数搜索博弈树，博弈树的每层根据落子方分为Max节点和Min节点，然后在该节点产生一系列可能的落子坐标，进行一系列的递归调用，再通过返回值和alpha或beta比较实现剪枝。

## 关键代码展示

### alpha-beta剪枝算法

```

# 最小最大搜索算法，同时使用了alpha-beta剪枝
# 参数意义：
# n:当前递归层数，起始为0
# player:本层下棋方，初始为1，即黑棋先手
# depth_limit:层数的最大限制，可以更改
# alpha,beta:用于剪枝
# old_board:上一层递归的棋盘（上层未落子） board:本层的棋盘（上层已落子）
# grade:总分数
def MinMax(n, player, depth_limit, alpha, beta, old_board, board, coordinate, grade):
    # 如果当前层数已经达到限制，计算分数并返回
    if n == depth_limit:
        # 减去落子坐标不落子的分数
        grade -= analysis_point((coordinate[0], coordinate[1], old_board))
        # 加上落子坐标落子的分数，其实这两行代码就是为了得到落子坐标处的净增加分数
        grade += analysis_point((coordinate[0], coordinate[1], board))
        return (coordinate, grade)

    if n != 0:
        # 如果不在第0层，则需要计算净分数
        grade -= analysis_point((coordinate[0], coordinate[1], old_board))
        grade += analysis_point((coordinate[0], coordinate[1], board))
        if abs(grade) > 35000:
            # 总分数绝对值大于35000，则该落子产生五连，继续搜索已经没有意义，所以返回分数
            return (coordinate, grade)

    if player == 1:
        # 黑棋落子
        next_state = get_successors(board, player, -1)
        coordinate = []
        for x, y, state in next_state:
            # 该落子的返回分数

```

```

        temp_alpha = MinMax(n + 1, 0, depth_limit, alpha, beta, board, state,
(x, y), grade)[1]
        if temp_alpha > alpha:
            # 若大于alpha，则更新alpha，并记录坐标
            alpha = temp_alpha
            coordinate = (x, y)
        if beta <= alpha:
            # beta小于等于alpha则发生剪枝
            break
    return (coordinate, alpha)
else:
    # 白棋落子，以下代码同黑棋落子
    next_state = get_successors(board, player, -1)
    coordinate = ()
    for x, y, state in next_state:
        temp_beta = MinMax(n + 1, 1, depth_limit, alpha, beta, board, state,
(x, y), grade)[1]
        if temp_beta < beta:
            beta = temp_beta
            coordinate = (x, y)
        if beta <= alpha:
            break
    return (coordinate, beta)

```

该算法的执行步骤与上图的流程图相同，可参照流程图理解该算法。

## 评估函数

### 评分规则

该评分规则参考了论文[《计算机五子棋博弈系统的研究与实现》](#)

我们给主要的规则评分如下：

- |                     |                     |
|---------------------|---------------------|
| (1) “00000” → 50000 | (2) “+0000+” → 4320 |
| (3) “+000++” → 720  | (4) “++000+” → 720  |
| (5) “+00+0+” → 720  | (6) “+0+00+” → 720  |
| (7) “0000+” → 720   | (8) “+0000” → 720   |
| (9) “00+00” → 720   | (10) “0+000” → 720  |
| (11) “000+0” → 720  | (12) “++00++” → 120 |
| (13) “++0+0+” → 120 | (14) “+0+0++” → 120 |
| (15) “+++0++” → 20  | (16) “++0+++” → 20  |

上面的评分是从计算机方棋子有这些特征，给正分。如果是对手有这些特征，则得分取反，得负分。例如对手如果有“00000”特征，将得-50000 分。

黑棋棋形的分数为正，白棋棋形的分数为负，因为ai为黑棋，则分数越高ai下出该棋形的可能性就越大

```

grade_dict = {"#####":50000, ' #### ':4320, '# #代表黑棋, *代表白棋, ' '空格代表无棋
' ### ':720, ' ## # ':720,
' ## # ':720, ' # ## ':720,
'#### ':720, ' #####':720,
'## ## ':720, '# ####':720,
'### # ':720, ' ## ':120,
' # # ':120, ' # # ':120,
}

```

```

        '# ':20, '# ':20,
"*****": -50000, ' **** ': -4320,
' *** ': -720, ' *** ': -720,
' ** * ': -720, ' * ** ': -720,
' **** ': -720, ' ***** ': -720,
'* * * ': -720, '* *** ': -720,
'*** * ': -720, ' ** ': -120,
' * * ': -120, ' * * ': -120,
'* * ': -20, ' * ': -20}

```

## 评分函数

将某一行（可以为横竖斜）转化为字符串，然后利用ac算法查找其中包含的棋形

```

def _heuristic_priority(state:tuple):
    # 坐标x
    x = state[0]
    # 坐标y
    y = state[1]
    # 棋局
    board = state[2]
    # 当前落子颜色
    color = state[3]
    # 如果当前坐标附近两格没有棋子，则该棋子不可能是目标落子点
    if not near_center(board,x,y):
        return -priority_board[x][y] #返回棋子的优先级的相反数，越在棋盘中间越高，主要是为了完成ai的第一步落子
    grade = 0
    # 减去未修改棋局时的分数
    grade -= analysis_point((x,y,board))
    flag = True
    # 如果当前坐标为空，则落子
    if board[x][y] != -1:
        flag = False
    if flag:
        board[x][y] = color
    # 加上落子之后的分数，则grade为落子增加的净分数
    grade += analysis_point((x,y,board))
    if flag:
        board[x][y] = -1
    # 如果为黑棋，则返回分数的相反数，因为黑棋为Max节点，优先返回分数高的节点，而排序是从小到大的，所以此处取反
    if color == 1:
        return -grade
    # 如果为白棋，则直接返回分数
    else:
        return grade

```

## 启发式函数

我们知道alpha-beta剪枝的效率与每次产生新点的顺序有关，比如在Max层，如果产生的第一个点就可以得到比beta更大的算法，则可以立马发生剪枝。

一个好的启发式函数可以帮助我们对产生的新节点进行排序，从而大大的提高剪枝效率，相反，在最坏情况下，错误的顺序会使alpha-beta剪枝退化为深度优先搜索。

上面我们已经提到了评分规则，此处的启发式函数便可以利用评分规则，通过估计该点的分数来给该点赋予一个代价，从而根据代价排序

```
def analysis_point(coordinate):
    # 坐标x
    x = coordinate[0]
    # 坐标y
    y = coordinate[1]
    # 棋盘
    board = coordinate[2]
    grade = 0
    # 得到横行的分数
    grade += get_grade(board[x], ac)
    line = []
    # 得到竖列
    for i in range(15):
        line.append(board[i][y])
    # 得到竖列的分数
    grade += get_grade(line, ac)
    line.clear()
    temp_x = x + y
    temp_y = 0
    # 得到右斜列
    while temp_x >= 0:
        if vaild(temp_x, temp_y):
            line.append(board[temp_x][temp_y])
        temp_x -= 1
        temp_y += 1
    # 得到右斜列分数
    grade += get_grade(line, ac)
    line.clear()
    temp_x = x - y
    temp_y = 0
    # 得到左斜列
    while temp_x <= 14:
        if vaild(temp_x, temp_y):
            line.append(board[temp_x][temp_y])
        temp_x += 1
        temp_y += 1
    # 得到左斜列分数
    grade += get_grade(line, ac)
    # 返回分数
    return grade
```

## 创新点&优化

### ac自动机

如何将棋局评分？这需要我们查找棋局上是否存在我们期望的棋形。

从棋局上查找对应棋形的过程显然是一个模式匹配的问题，并且由于棋形的多样化，这还是一个多模式匹配的问题。

如果假设模式串的长度为 $M_i$ ,主串的长度为N。

- 最朴素的模式匹配算法的时间复杂度:

$$O((\sum_i M_i) * N)$$

- ac自动机算法的时间复杂度:

$$O((\sum_i Mi) + N)$$

不难看出使用ac自动机能够极大的减少程序进行模式匹配的时间，从而提高博弈树搜索的深度和速度。

## 启发式函数

AlphaBeta剪枝有严重依赖于每一层节点扫描的顺序，因为前面节点生成的alpha直接决定了后面节点剪枝的多少。如果我们针对每一层生成的节点事先排好序，那么程序的速度将极大地提升。然而，对生成的节点进行绝对准确地排序是不可能的，因为需要巨大的运算量（等同于再进行一次alphabeta剪枝的搜索），所以我们只能对节点进行粗略地排好序。

我们已经讨论了如何对棋盘上的位置进行评分，我们可以利用这个评分，对生成的可能的点进行排序。排序开销小，然而对整体的搜索速度带来巨大的提升。

## 保存评分

大多数alpha-beta是在到达最后一层时对整个局面进行评分，来得出一个局面的评分。这样虽然可行，但是效率低下。

因为在alpha-beta剪枝函数的每一次迭代当中，棋盘中的棋子只会增加（减少）一个，棋盘中的大部分区域的评分是不变的，所以不用每次到达最后都对整个棋盘来进行一次完整的扫描。可以保存好前一次迭代里棋盘的评分，然后在当前递归中扫描当前下的棋子所在的行、列、斜列，结合前一次递归的棋盘评分，得出新的评分。这样就免去了每次都要重新扫描整个棋盘。效率可以有不小的提升。这一优化体现在搜索算法中grade参数的传递。

## 一些额外的剪枝

- 若该某个坐标的周围两格内都没有棋子（初始情况除外），则显然该坐标几乎不可能会得到一个较高的分数，也就是说我们在生成每一层的节点时可以不考虑这样的节点，从而减少分支因子和提升上述启发式排序的速度。

以下为具体函数的代码：

```
def near_center(board,x,y):
    # 检测该坐标两格以内
    start_X = x - 2
    end_X = x + 2
    start_y = y - 2
    end_y = y + 2
    # 排除一些边界情况
    if start_X < 0:
        start_X = 0
    if end_X > len(board)-1:
        end_X = len(board)-1
    if start_y < 0:
        start_y = 0
    if end_y > len(board)-1:
        end_y = len(board)-1
    for i in range(start_X,end_X+1):
        for j in range(start_y,end_y+1):
            if board[i][j] == 0:
                return True
    return False
```

```

for j in range(start_y, end_y+1):
    # 如果不为空, 返回True, 说明附近有棋子
    if board[i][j] != -1:
        return True
    # 否则返回False, 说明附近没有棋子
return False

```

- 在启发式搜索的优化中, 已经对下一步可能出现的位置粗略地排好了序, 极大地提高了AlphaBeta剪枝的效率。但是由于每层产生的点太多, 导致计算机仍然需要搜索数量巨大的节点。事实上, 只要对位置进行评分的函数足够好, 我们就可以保证最优解基本出现在排好序的节点的前十之中。所以我们只需要扫描产生的排好序的点的前十个点就可以了。这个不起眼的优化提升非常巨大, 有了这个小优化之后, 博弈树的搜索层数大大增加, 甚至多数时候只扫描前五个点亦可, 最高使搜索树到7层。

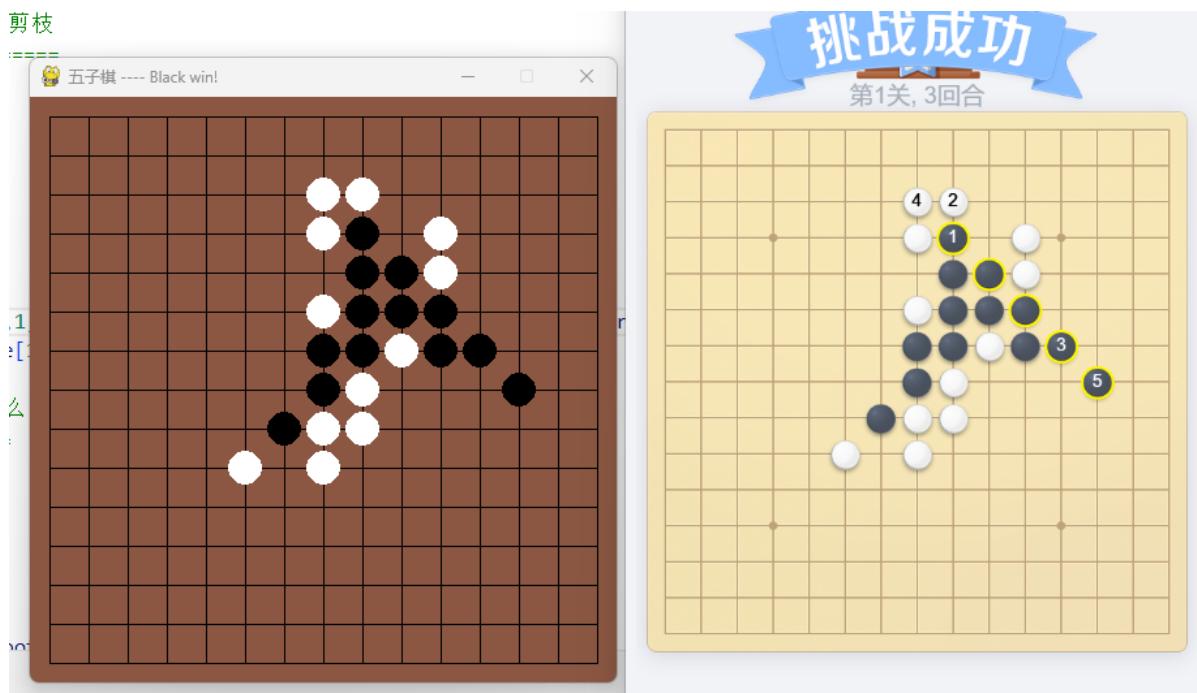
# 实验结果及分析

## 实验结果展示

经过优化后的代码可通过“欢乐五子棋”残局闯关前20关的所有关卡

具体棋局如下:

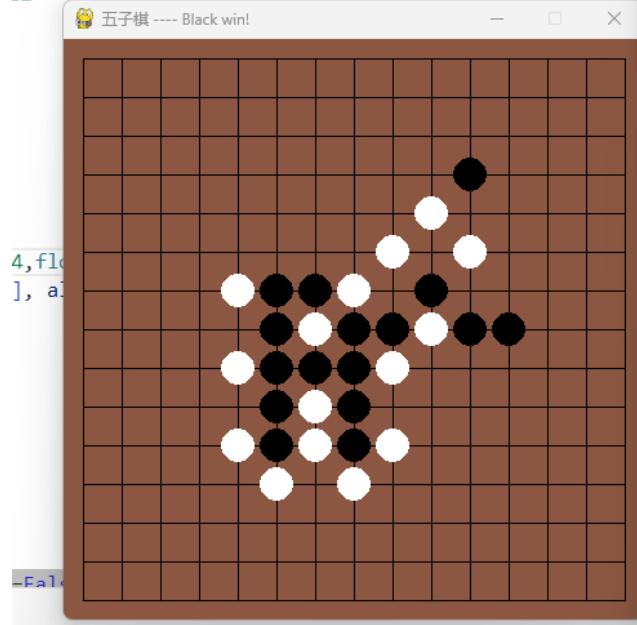
第一关 (搜索层数4) :



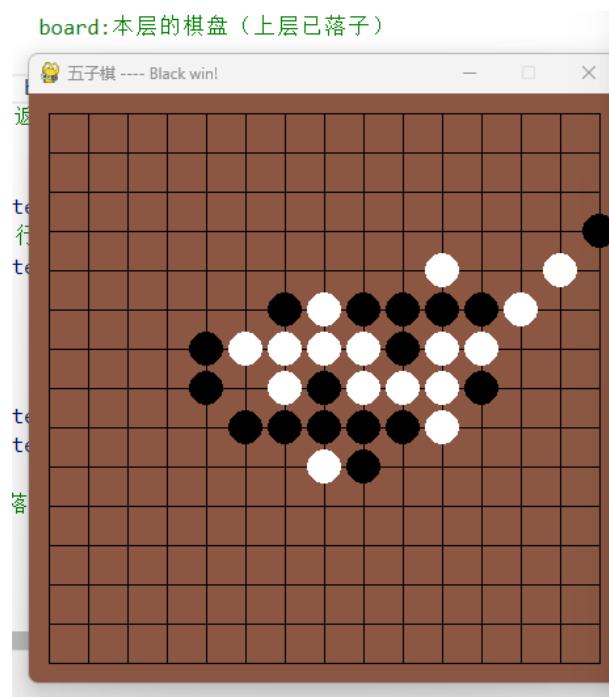
第二关 (搜索层数4) :

支

==

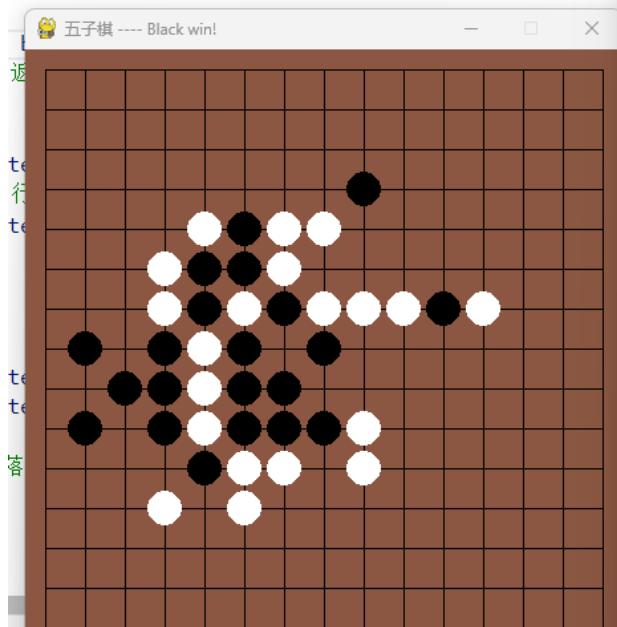


第三关 (搜索层数5) :



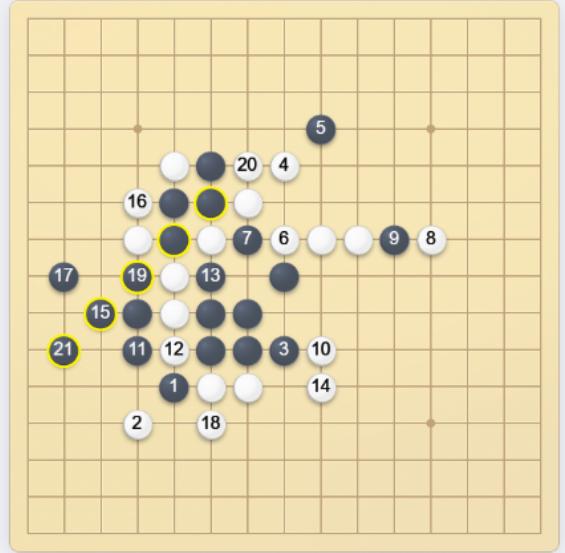
第四关 (搜索层数5) :

board:本层的棋盘 (上层已落子)

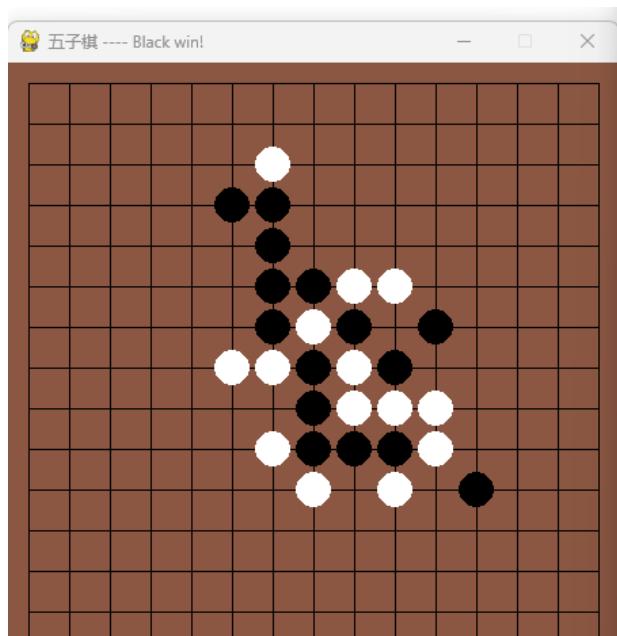


挑战成功

第4关, 11回合

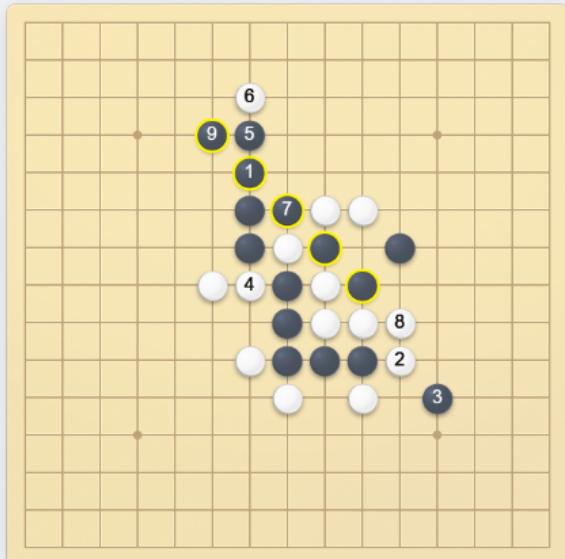


第五关 (搜索层数5) :

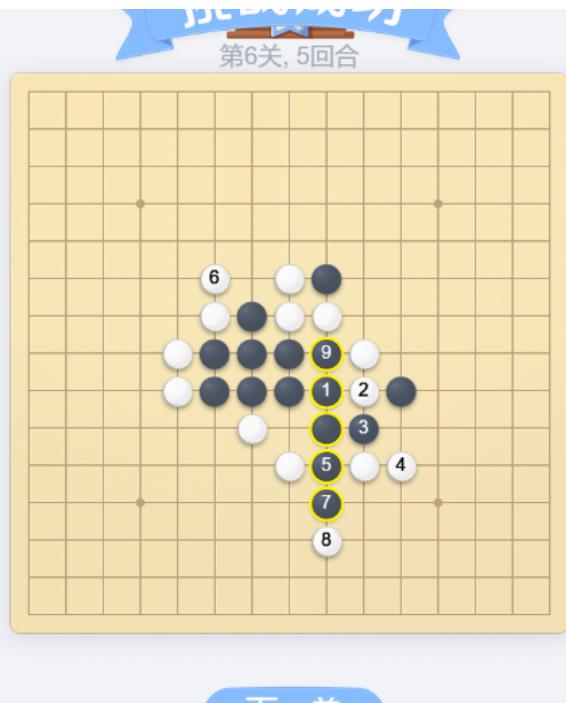
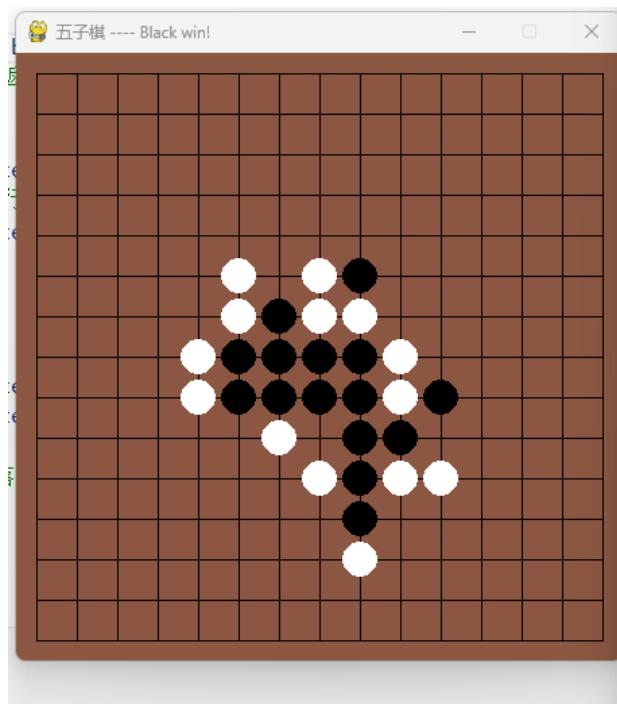


挑战成功

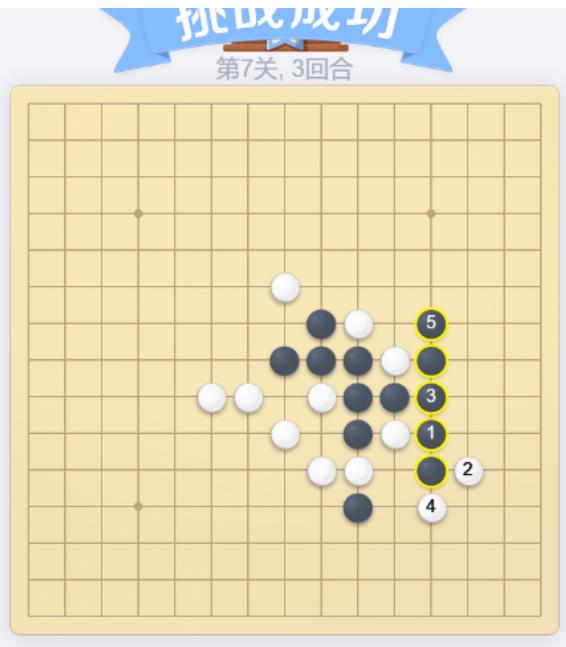
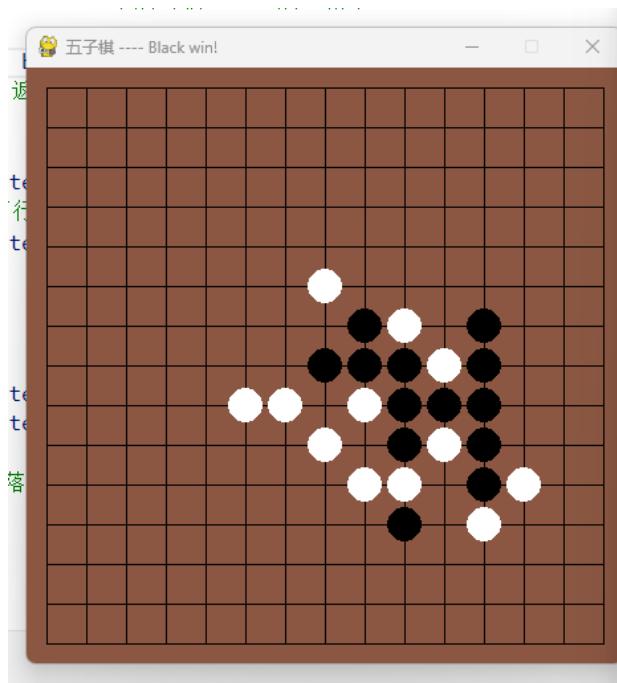
第5关, 5回合



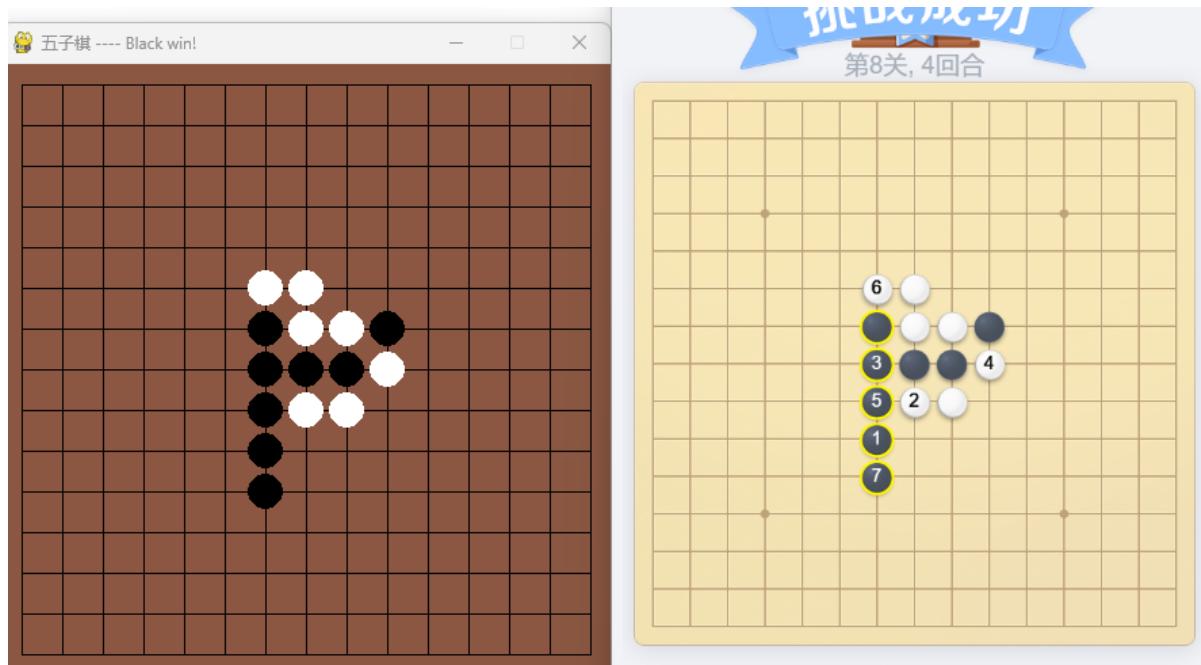
第六关 (搜索层数5) :



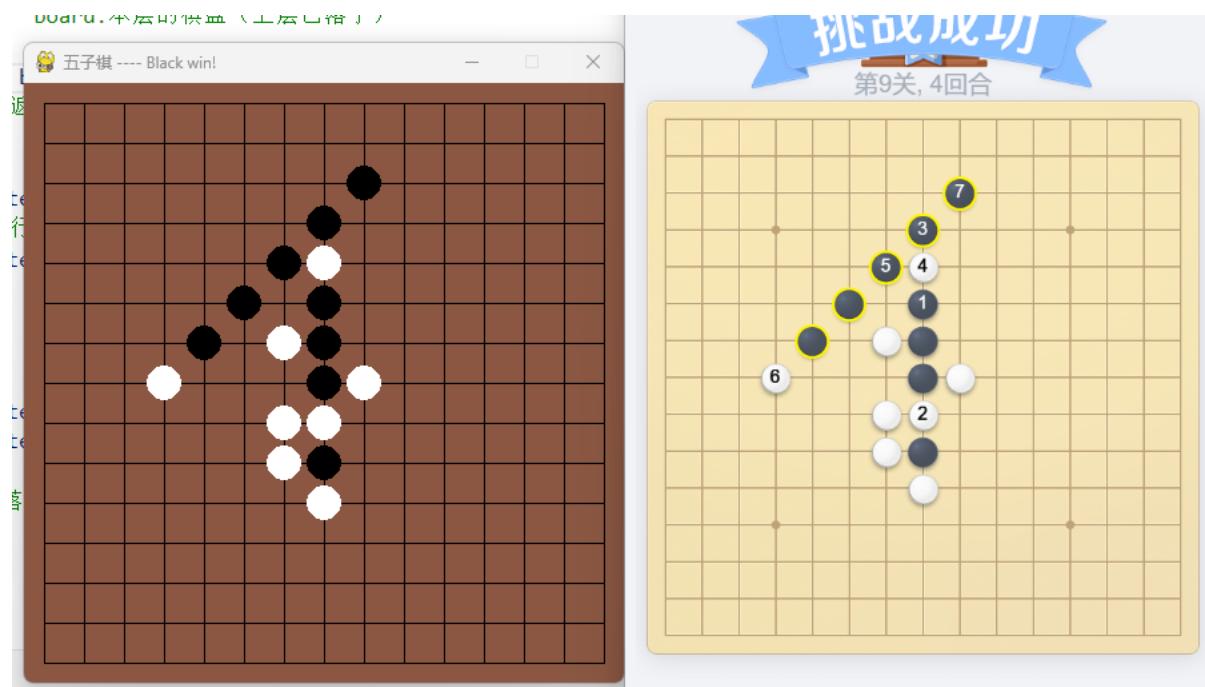
第七关 (搜索层数5) :



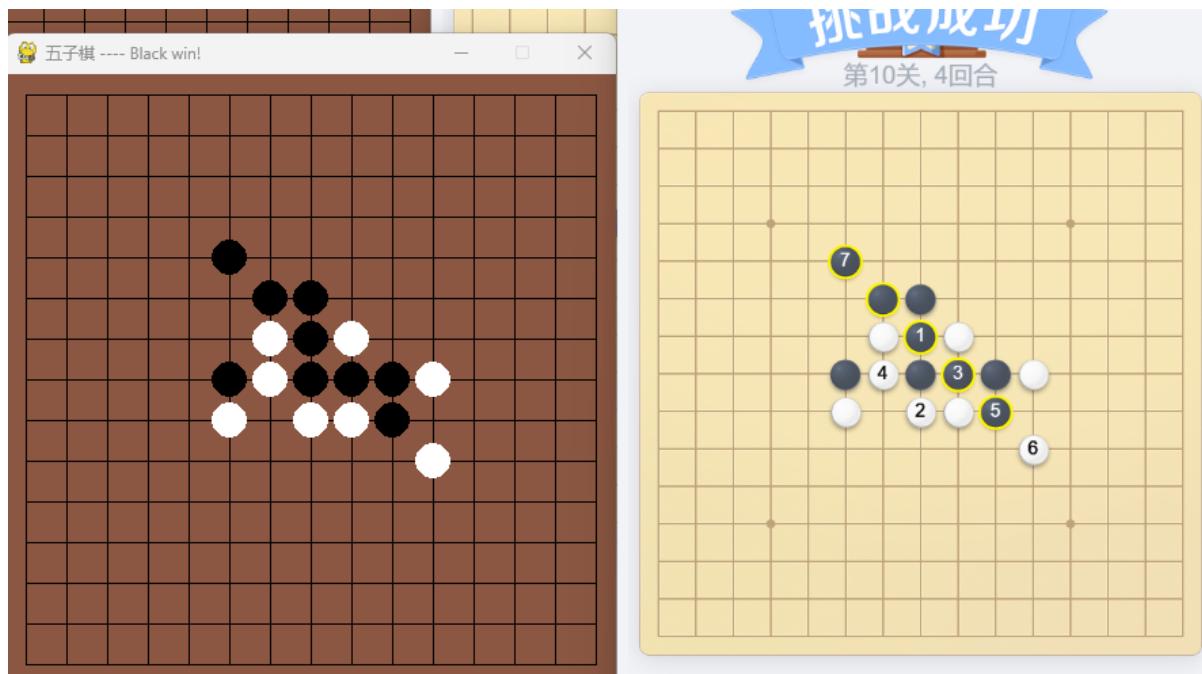
第八关 (搜索层数5) :



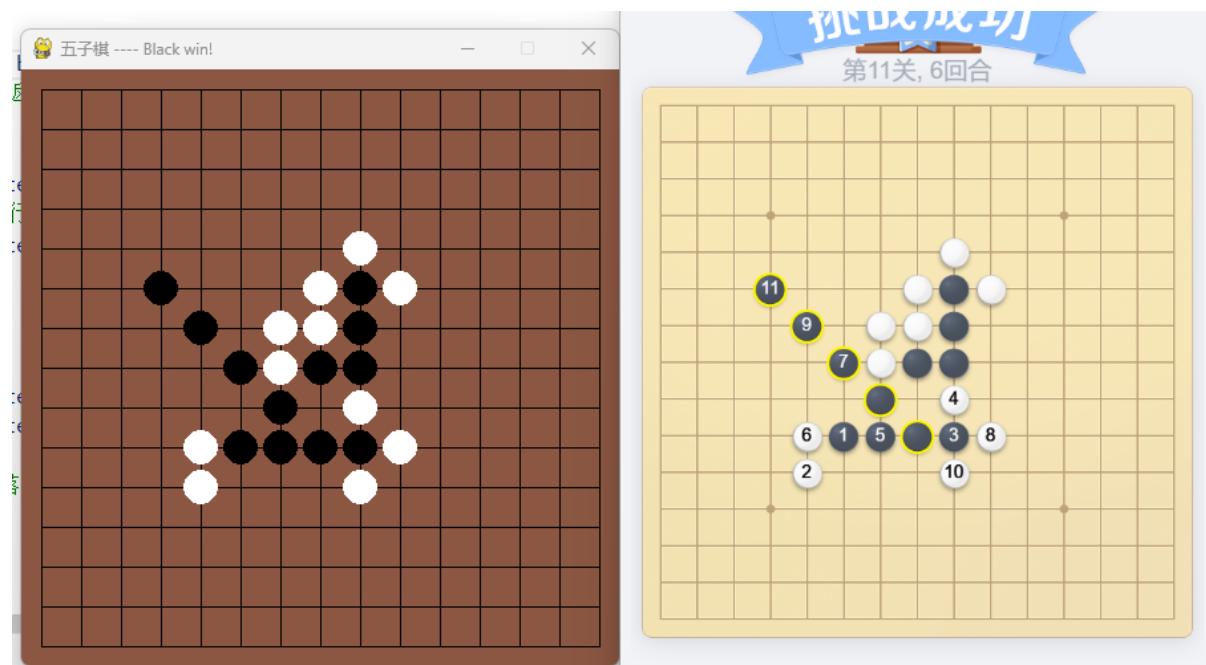
第九关 (搜索层数5) :



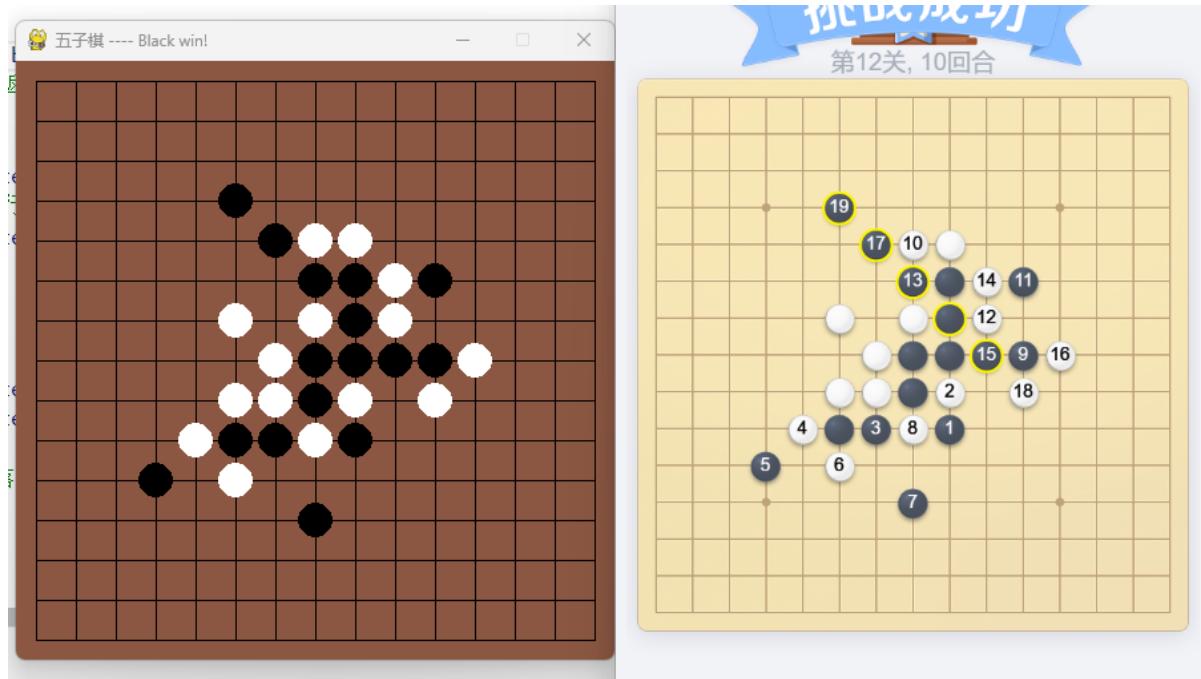
第十关 (搜索层数5) :



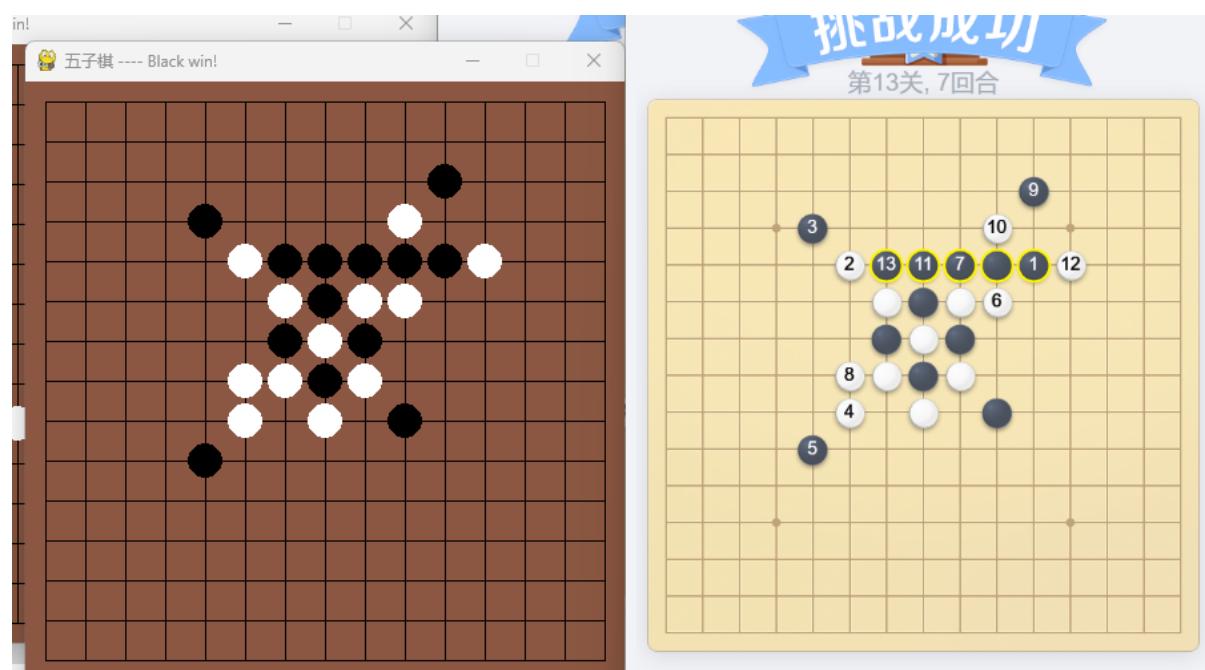
第十一关（搜索层数5）：



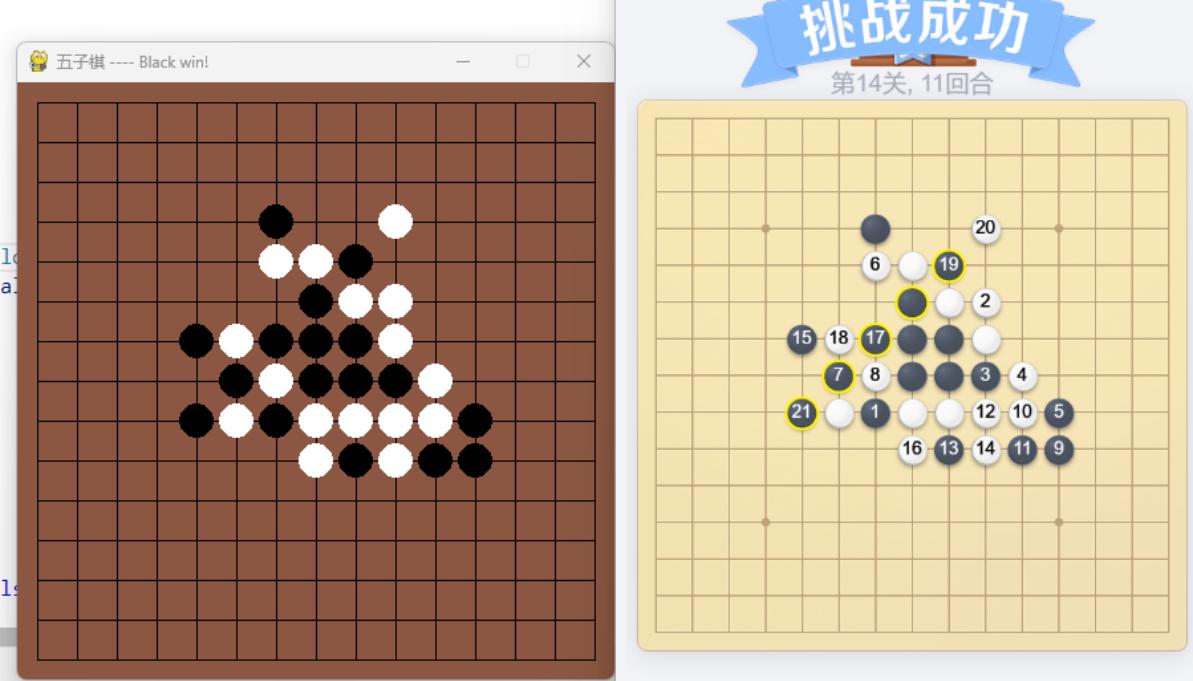
第十二关（搜索层数5）：



第十三关（搜索层数5）：



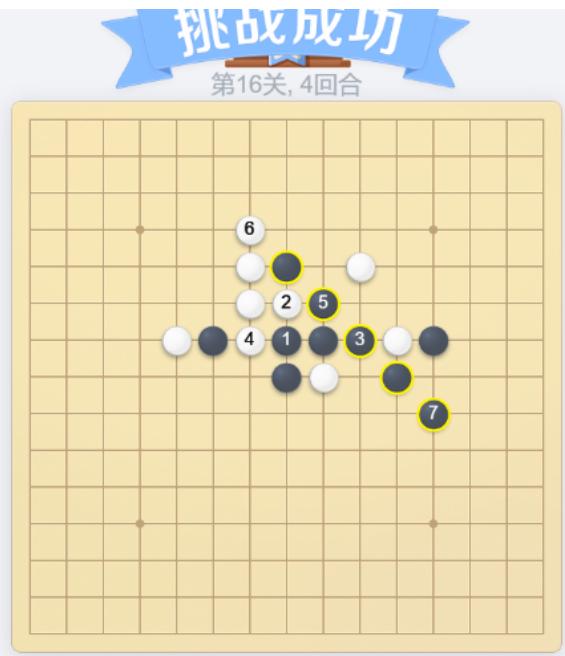
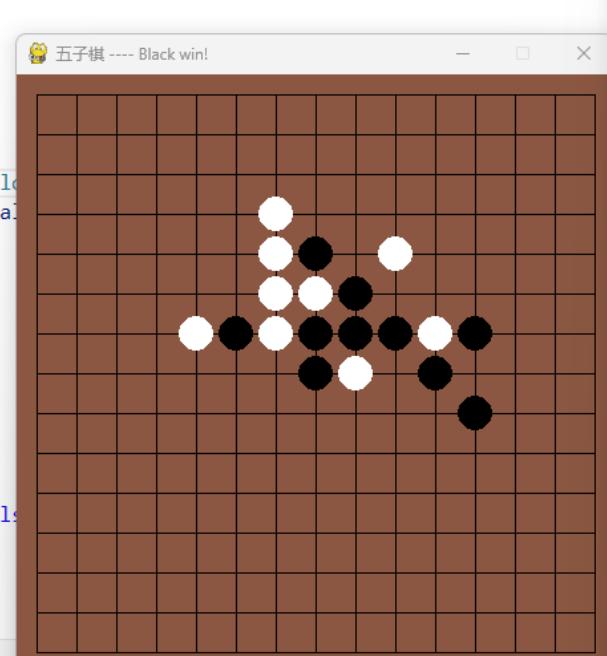
第十四关（搜索层数6）：



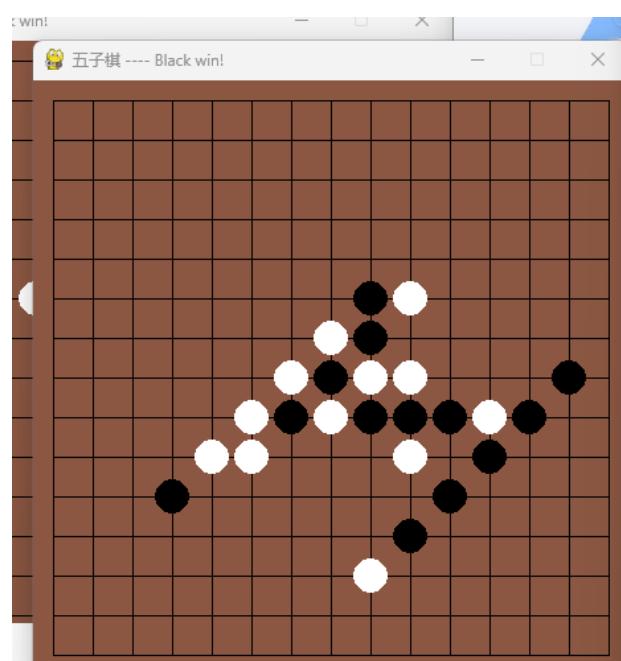
第十五关（搜索层数5）：



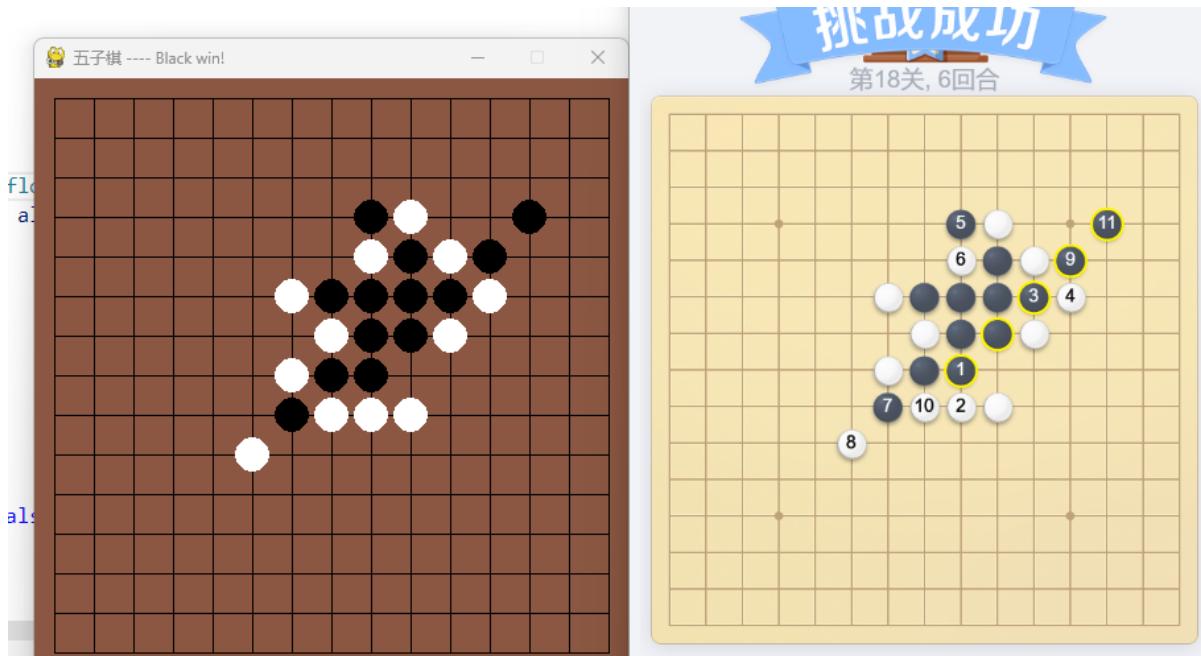
第十六关（搜索层数5）：



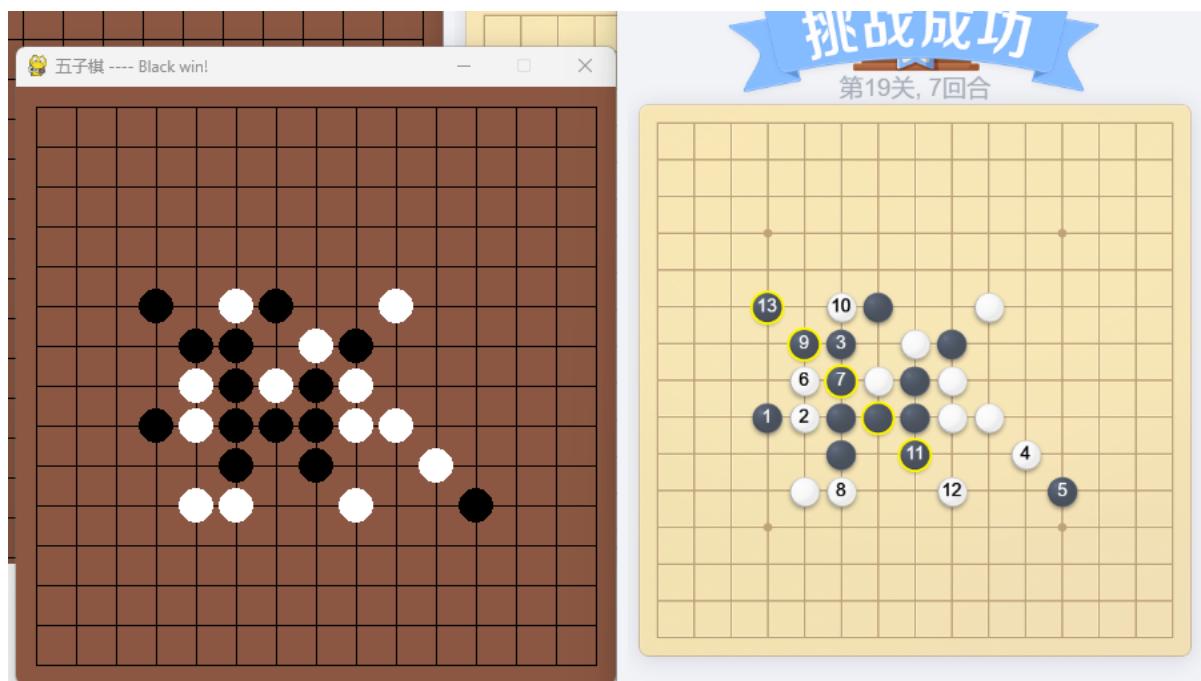
第十七关（搜索层数5）：



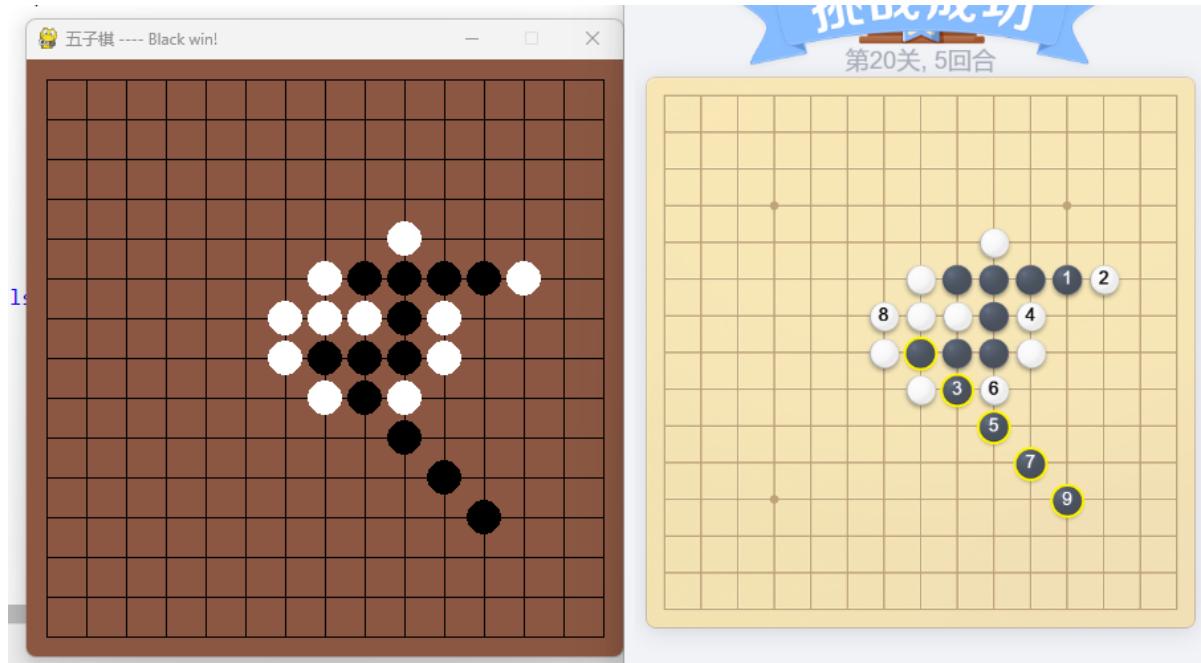
第十八关（搜索层数5）



第十九关 (搜索层数5) :



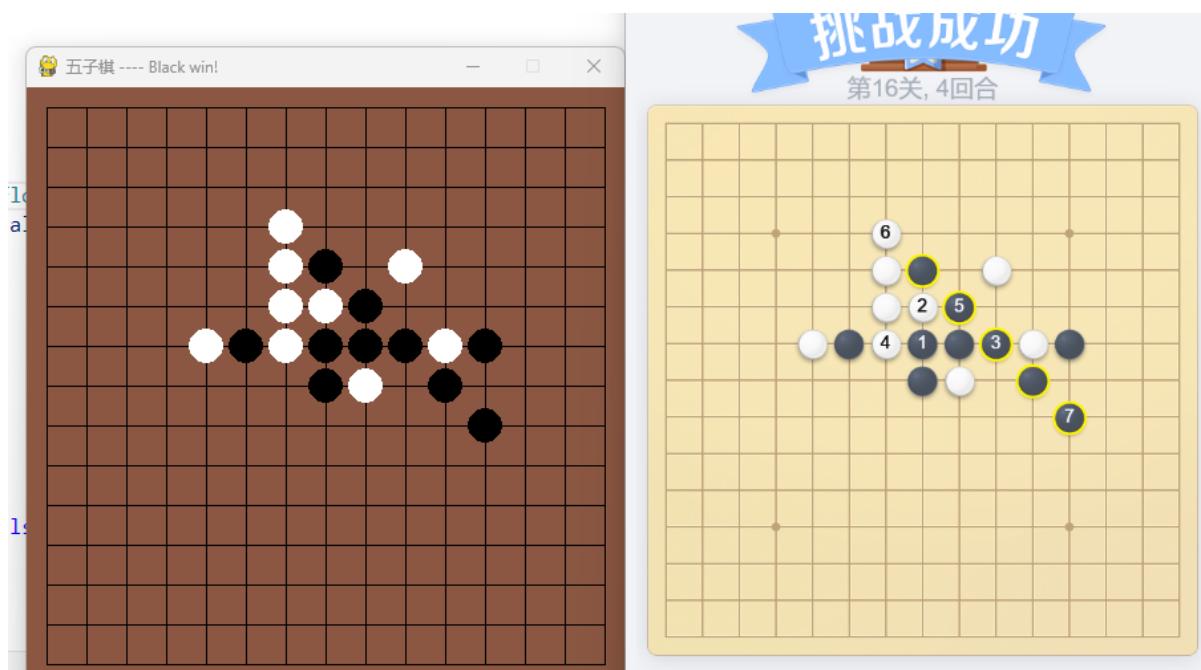
第二十关 (搜索层数5) :



## 实验结果分析

我们以第十六关为例，分析ai每步落子的原因。

由于我搜索了五层，每层大约有十个节点，总共就有100000个节点，对每个节点都进行分析显然不太可能，所以我们首先对第一层的节点进行分析。

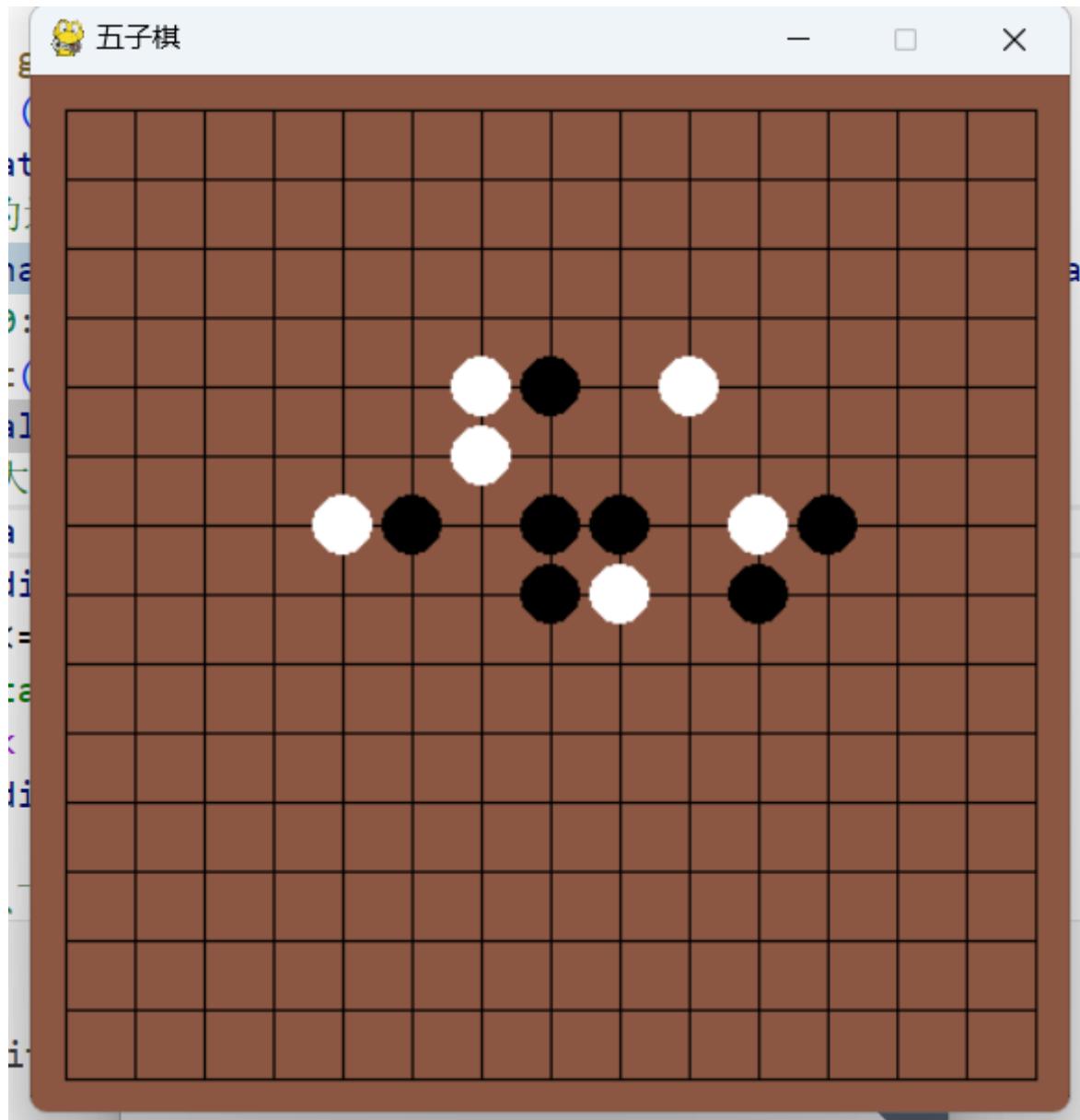


其实从这个图中，我们就可以看出，黑棋其实在下第一步的时候就已经找到了必胜路线，因为ai可以搜索五层，也即可以几乎预料到五步之后的结果，从上图看下完第五步时，ai已经形成活四，到达必胜的局面。接下来我们分析每一步来印证我们的想法

第一步（黑棋）：

黑棋的落子点为 (6, 7)，评分为3720，该评分接近4320也即活四的分数，也就是说ai在搜索时已经找到了必到活四的路线。

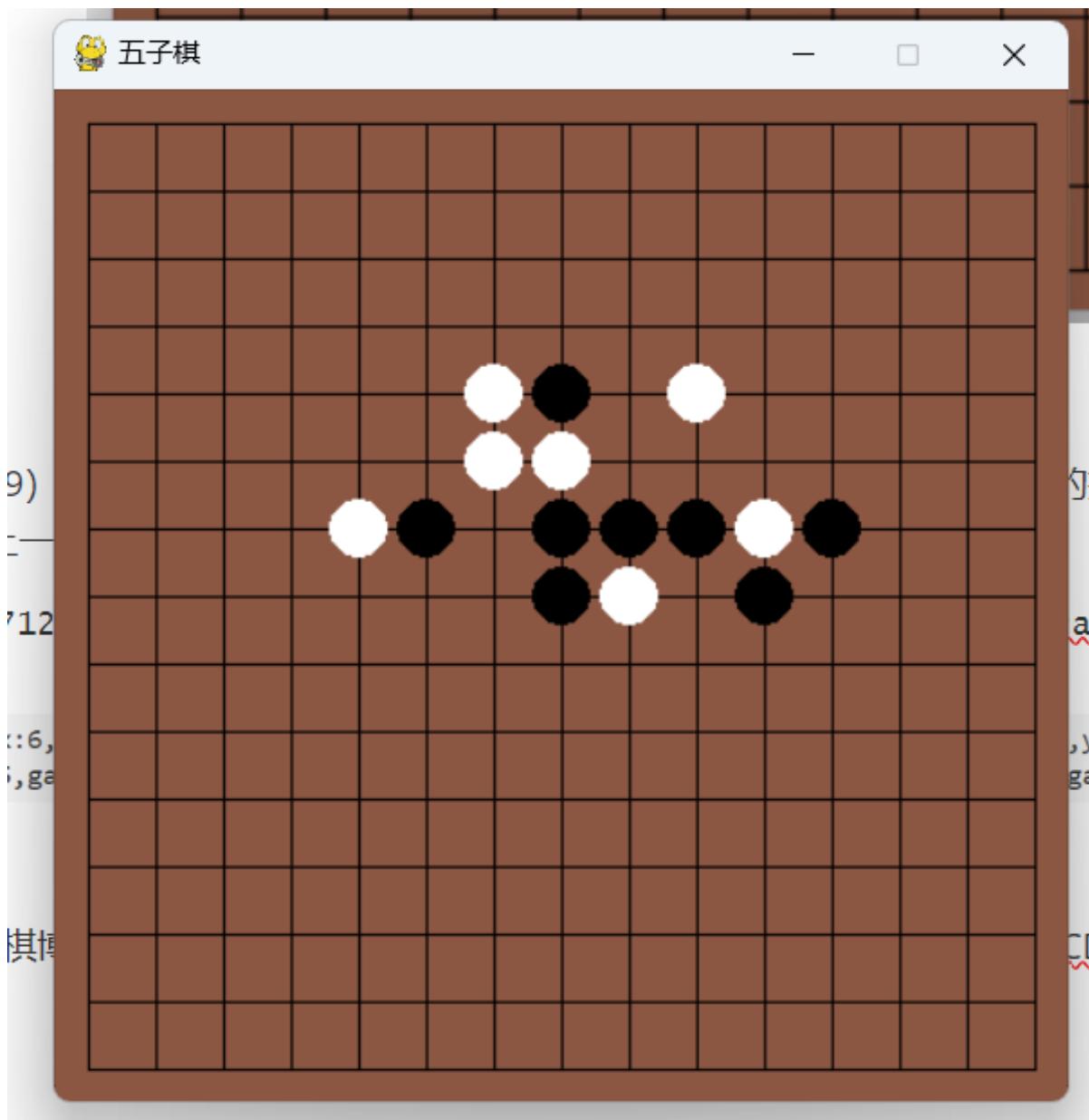
```
x:8,y:9,garde:1100  x:5,y:9,garde:1100  x:8,y:6,garde:1100  x:5,y:12,garde:1100  x:6,y:7,garde:3720  x:5,y:7,garde:3720  x:5,y:8,garde:3720  x:6,y:9,garde:3720  x:9,y:5,garde:3720  x:4,y:10,garde:3720  x:4,y:13,garde:3720  6 7 3720
```



第二步（黑棋）：

黑棋的落子点为 (6, 9)，分数为45000，接近50000也就是五连的分数，说明ai在这次的搜索中已经知道必到五连的路线，这也映衬了我们上一步的想法，上一步中ai可以搜索到活四，自然的，这一步必然可以搜索到五连。

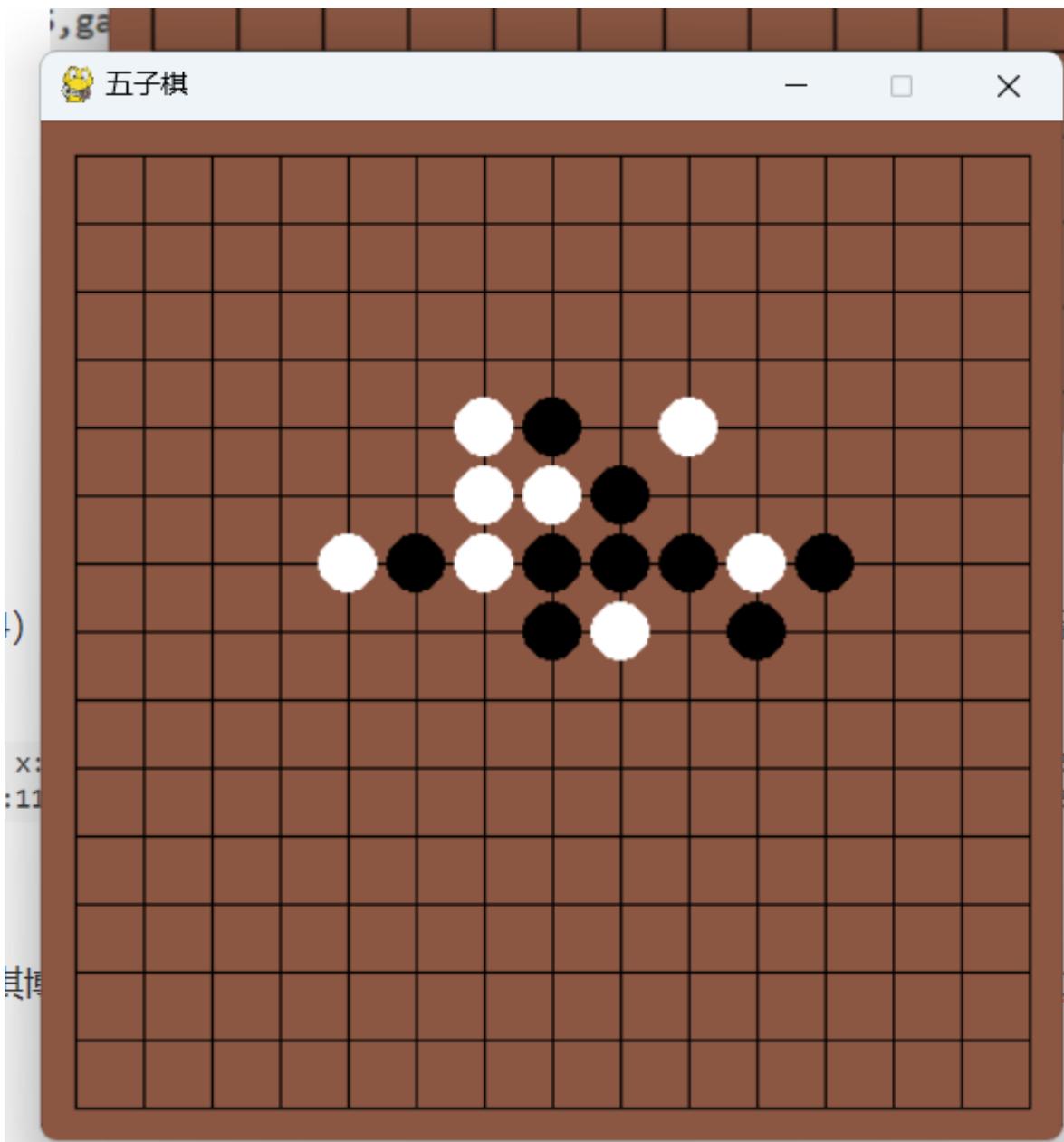
```
x:5,y:9,garde:3840  x:6,y:9,garde:45000  x:8,y:6,garde:45000  x:5,y:12,garde:45000  x:8,y:9,garde:45000  x:6,y:6,garde:45000  x:5,y:8  
,garde:45000  x:9,y:5,garde:45000  x:4,y:10,garde:45000  x:4,y:13,garde:45000  x:5,y:5,garde:45000  6 9 45000
```



第三步（黑棋）：

黑棋的落子点为 (5, 4) , 分数为45000, 说明ai仍然可以搜索到必胜的局面, 并且搜索到五连之后, ai不会继续搜索, 所以这一步仍然是45000分。可以看到在上一步完成后, 白棋形成活三, 但是黑棋并没有去防守, 因为如果去防守, 则会失去必胜的局面, 也就达不到45000分, 所以ai并不会选择防守白棋, 事实也是黑棋将会更快地完成绝杀。

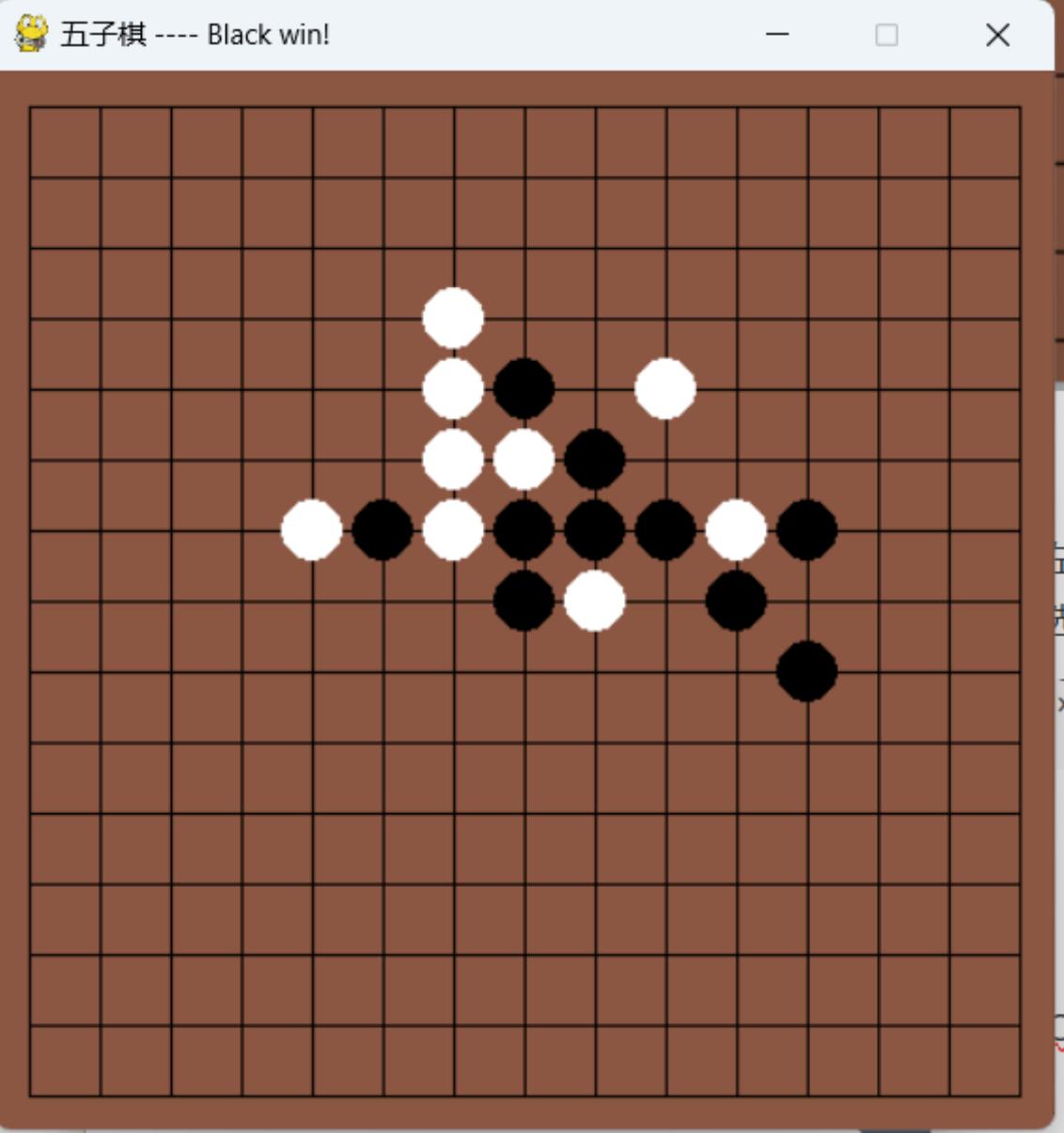
```
x:5,y:8,garde:45000  x:8,y:6,garde:45000  x:3,y:6,garde:45000  x:7,y:6,garde:45000  x:5,y:9,garde:45000  x:8,y:9,garde:45000  x:5,y:1  
2,garde:45000  x:8,y:11,garde:45000  x:2,y:6,garde:45000  x:9,y:12,garde:45000  x:9,y:5,garde:45000  5 8 45000
```



第四步（黑棋）：

黑棋选择落子（8， 11）， 分数为45000，形成五连，可以看到第一层中，有许多-50000左右的分数，这是因为白棋已经形成活四，如果黑棋不走（8， 11），那么将会迎来白棋的必胜局面，此时的分数就会是-50000左右。ai选择分数最高的一步，形成五连。

```
x:8,y:11,garde:45000  x:2,y:6,garde:-50200  x:7,y:6,garde:-50220  x:8,y:9,garde:-49540  x:5,y:12,garde:-49620  x:8,y:6,garde:-49620  
x:5,y:9,garde:-49660  x:9,y:12,garde:-50220  x:9,y:5,garde:-50340  x:4,y:10,garde:-50380  x:4,y:13,garde:-50380  8 11 45000
```



## 评测指标展示及分析

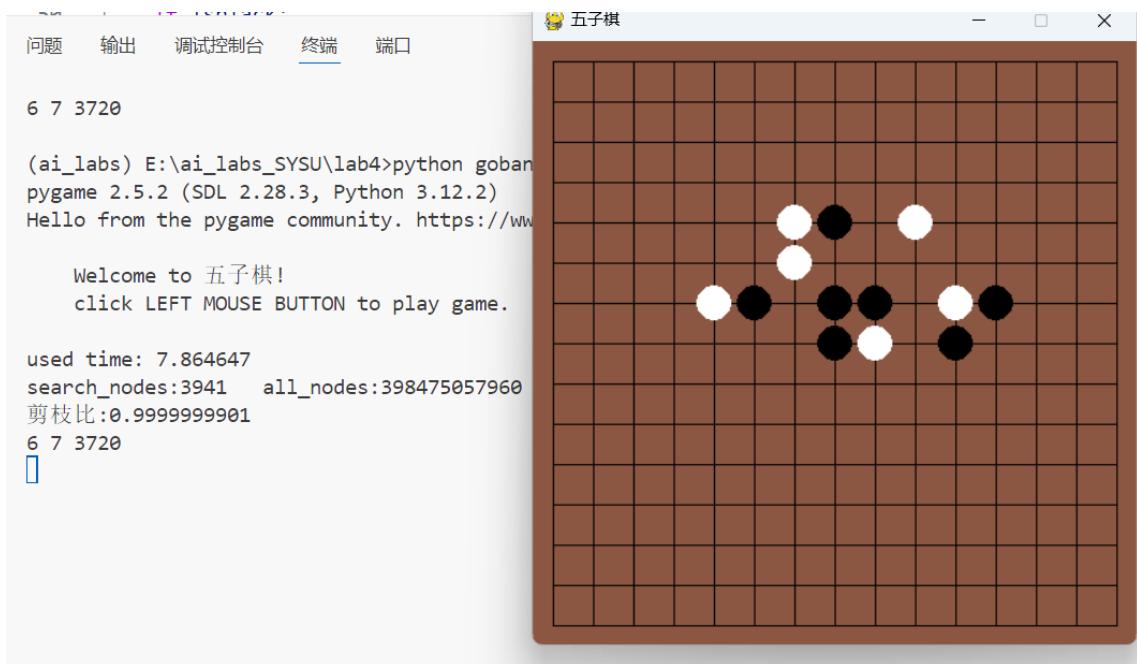
我们仍然以第十六关为例，分析搜索时间，剪枝比等。

我们来对比不同的启发式搜索对游戏结果和搜索效率的影响。

- 优化后

如图为十六关第一步，可以看到程序的搜索时间很快，仅在十秒之内，并且剪枝比已经几乎快达到**100%**！正是因为如此高的剪枝比，才能通过使用python在数秒内搜索五层。

并且可以看到ai落子的评分为3720，说明已经找到活四的局面，也就是说ai在该残局的第一步就几乎达到了**100%**的胜率。

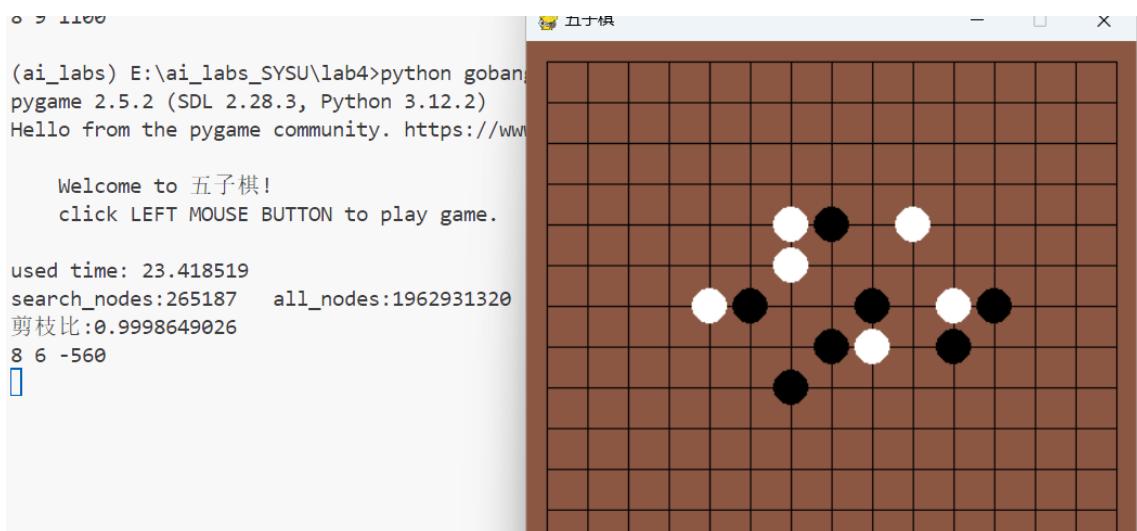


- 优化前

如果去掉启发式搜索和一些额外的剪枝，仅仅使用普通的alpha-beta剪枝。此时如果继续把搜索树设置在五层，则很难在有限时间内得出结果（超过15分钟），所以此处只能设置为4层进行测试，如下图所示。

虽然剪枝比仍然为较高的99.98%，但是这只搜索了四层的节点数都已经达到265187，而优化后的搜索五层才3941，搜索的节点数增加了**67倍**，但是在启发式搜索下一层节点处节约了时间，所以时间上只增加到约**3倍**。

并且最后的落子点评分为-560，说明因为只搜索了4层，ai并没有找到一个比较好的落子点，这大大增加了最后ai失败的概率，而优化后的ai在这一步时已经找到了必胜路线。



#### 参考资料：

- [1] [计算机五子棋博奕系统的研究与实现 - 百度学术 \(baidu.com\)](#)
  - [2] [AC 自动机 - OI Wiki \(oi-wiki.org\)](#)