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）

一、 实验题目

五子棋--博弈树搜索 Alpha-beta 剪枝

二、 实验内容

1. 算法原理

Alpha-beta 剪枝

Alpha-beta 剪枝算法是在极小化极大（Minimax）算法的基础上发展而来的。Minimax 算法是一种在零和游戏中寻找最优策略的算法，它通过递归地搜索游戏树来评估可能的走法和结果。然而，Minimax 算法可能会搜索整个游戏树，这在复杂的游戏中会导致巨大的计算量。为了解决这个问题，alpha-beta 剪枝算法被提出，其核心思想是通过剪掉那些明显不会影响最终决策的分支来减少搜索的节点数。

算法传入棋盘的列表，当前的 alpha 和 beta 值，以及计算深度（受计算机性能限制，深度只能设置为 2）。

4 月 20 日算法优化。算法维护当前棋盘真正落子的边界，这基于一个观察，对当前棋盘来说，有效的落子仅仅分布在有棋子的附近，因此远离棋子边界的落子不予考虑。例如，对当前棋盘的状态，黑白棋子仅在中心 5*5 的范围内，那么只需考虑中心 6*6 的范围落子，有效剪枝。

那么结合 Alpha-Beta 剪枝和落子边界剪枝的**双剪枝算法**能有效提升算法的效率。具体体现在，未优化前搜索深度最大只能确定为 2，如果设置为 3，计算过程极为缓慢；优化后可以在 30s-1min 内运行深度为 3 的搜索，提升落子质量。

2. 伪代码

```
def AlphaBetaSearch(board, boundary, EMPTY, BLACK, WHITE, isblack, alpha, beta, depth = 2) -> (int, int, int):
    procedure AlphaBetaSearch(board, boundary, EMPTY, BLACK, WHITE, isblack, alpha, beta, depth = 2) returns (int, int, int)
        i := 0
        j := 0
        if depth = 0
            return 0, 0, evaluate(board, boundary, EMPTY, BLACK, WHITE)
        end if
```

```
        if isblack
            next_states := get_successors(board, 1)
            for x, y, state , new_boundary in next_states
                _, _, alpha1 := AlphaBetaSearch(state, new_boundary, EMPTY, BLACK,
    WHITE, False, alpha, beta, depth - 1)
                if alpha1 > alpha
                    i := x
                    j := y
                    alpha := alpha1
                end if
                if alpha ≥ beta
                    break
                end if
            end for
            return i, j, alpha
    end procedure
```

3. 关键代码展示（带注释）

```
black_pattern_scores = {(-1, 1, 1, -1) : 50,          #活二
                        (-1, 1, -1, 1, -1) : 50,      #活二
                        (1, 1, -1, 1, -1) : 200,      #眠三
                        (-1, 1, 1, 1) : 500,          #眠三
                        (1, 1, 1, -1) : 500,          #眠三
                        (-1, 1, 1, 1, -1) : 5000,     #活三
                        (-1, 1, -1, 1, 1, -1) : 5000, #活三
                        (-1, 1, 1, -1, 1, -1) : 5000, #活三
                        (1, 1, 1, -1, 1) : 6000,      #冲四
                        (1, 1, -1, 1, 1) : 6000,      #冲四
                        (1, -1, 1, 1, 1) : 6000,      #冲四
                        (1, 1, 1, 1, -1) : 6000,      #冲四
                        (-1, 1, 1, 1, 1) : 6000,      #冲四
                        (-1, 1, 1, 1, 1, -1) : 100000, #活四
                        (1, 1, 1, 1, 1) : 99999999}   #连五
white_pattern_scores = {(-1, 0, 0, -1) : 30,          #活二
                        (-1, 0, -1, 0, -1) : 30,      #活二
                        (0, 0, -1, 0, -1) : 100,      #眠三
                        (-1, 0, 0, 0) : 300,          #眠三
                        (0, 0, 0, -1) : 300,          #眠三
                        (-1, 0, 0, 0, -1) : 3000,     #活三
                        (-1, 0, -1, 0, 0, -1) : 3000, #活三
                        (-1, 0, 0, -1, 0, -1) : 3000, #活三
                        (0, 0, 0, -1, 0) : 4000,      #冲四
                        (0, 0, -1, 0, 0) : 4000,      #冲四
                        (0, -1, 0, 0, 0) : 4000,      #冲四
```

```
                (0, 0, 0, 0, -1) : 4000,                    #冲四
                (-1, 0, 0, 0, 0) : 4000,                    #冲四
                (-1, 0, 0, 0, 0, -1) : 80000,               #活四
                (0, 0, 0, 0, 0) : 99999999}                 #连五
def evaluate(board : list[list], boundary : int, EMPTY : int = -1, BLACK : int = 1, WHITE : int = 0) -> int:
    '''评价函数，传入棋盘，返回棋盘的评价值   black_score - white_score'''
    black_score = 0
    white_score = 0
    cnt = 0                                                 #统计活三和冲四的总数，如果
大于等于2，则必胜，返回一个较大的值
    for pattern, score in black_pattern_scores.items():
        if score >= 3000:
            cnt += 1
        black_score += count_pattern(board, boundary, pattern) * score
    for pattern, score in white_pattern_scores.items():
        if score >= 3000:
            cnt -= 1
        white_score += count_pattern(board, boundary, pattern) * score
    if abs(cnt) >= 2:
        return black_score - white_score + cnt * 20000
    return black_score - white_score
```

4. 创新点&优化（如果有）

1、落子顺序优化

在给出的代码中，落子优先级为从左上到右下，这里修改为从中间到四周进行检查可以落子的空间。生成的后继棋盘就按照这种顺序进行排序，在后面的 alpha-beta 剪枝算法中，优先对落子更靠近中间的棋盘进行深度优先遍历。采取这种遍历顺序，有很大概率落子在边缘位置的棋盘被剪枝，从而提高算法的性能。具体实现为在函数 _coordinate_priority 中，返回 max(abs(x - 7), abs(y - 7))，实际为坐标(x, y)到(7, 7)的无穷范数（切比雪夫距离）。

2、评价函数的优化

评价函数中，对五子棋常见棋型（活二、眠三、活三、冲四、连五等）进行评价。首先，该算法的目标是运行五子棋经典残局，往往黑方主动进攻便可速胜，因此在评价函数中，我**适当提高黑方的进攻属性**，降低防守属性，可以更好的找到解答，具体做法是降低白色方相应阵型的评价得分。（如果是黑白对弈，那么攻防兼备可能更优）

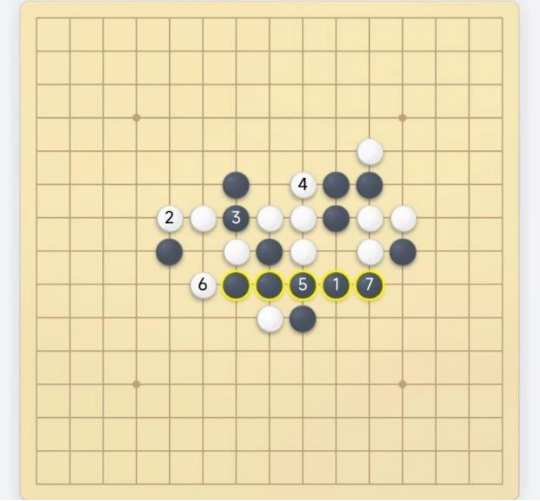
其次，为了更方便剪枝，我在评价函数中增加了 cnt 变量，统计活三、冲四等必须进行防守的阵型总数，如果 abs(cnt) >= 2，相当于棋盘上同时出现两个活三，那么此时必胜或者必败，为了**更加快速的进行剪枝**，返回一个比较大的值。

3、维护一个棋盘边界

对该部分的优化说明见**算法原理**部分。

三、 实验结果及分析

1. 博弈树分析（深度设置为 2，关卡为第三关）



1、 图中有多个“活二”，任意下一步可形成连三或者冲四。但只有图示位置可以形成两个冲四，评价值最高，因此 AI 落子于图中 1 位置

2、 白棋落子于 2，此时白棋马上连五，因此该步进行防守，落子于图中 3 位置

3、 白棋落子于 4，活三，但黑方已形成两个冲四，已经成为必胜局面，因此不理会白棋的活三，进行进攻。最终取得胜利。

```
0       0       0       0       0       0       0       0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0       0       0       0       0       0       0       0
0       0       -8540   -8540   -8540   -8540   -8540   -8540   -8540   -8540   -8540   -8540   -8540   0       0
0       0       -8540   -8510   -8510   -8510   -8510   -8510   -8510   -3310   -8510   -8510   -8340   0       0
0       0       -8540   -8510   -8510   -8510   -8510   -8510   -8510   -1560   0       -8010   -8540   0       0
0       0       -8540   -8510   -8410   -8460   0       -8400   970     0       0       -2060   -3310   0       0
0       0       -8540   -8510   140     0       -8250   0       0       0       0       0       -8540   0       0
0       0       -8540   -8510   0       -8510   0       0       0       -1560   0       0       -8540   0       0
0       0       -8540   -8510   1740    -1480   0       0       -3180   3440    -8510   -8510   -8540   0       0
0       0       -8540   -8510   -8460   -7510   -8460   0       0       -8510   -8510   -8510   -8540   0       0
0       0       -8540   -8510   -8310   -8510   -8510   -8460   -8460   -7510   -8510   -8510   -8540   0       0
0       0       -8540   -8510   -8510   -8510   -8460   -8510   -8460   -8510   -8310   -8510   -8540   0       0
0       0       -8540   -8540   -8540   -8540   -8540   -8540   -8540   -8540   -8540   -8540   -8540   0       0
0       0       0       0       0       0       0       0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0       0       0       0       0       0       0       0
8 9 3440
```

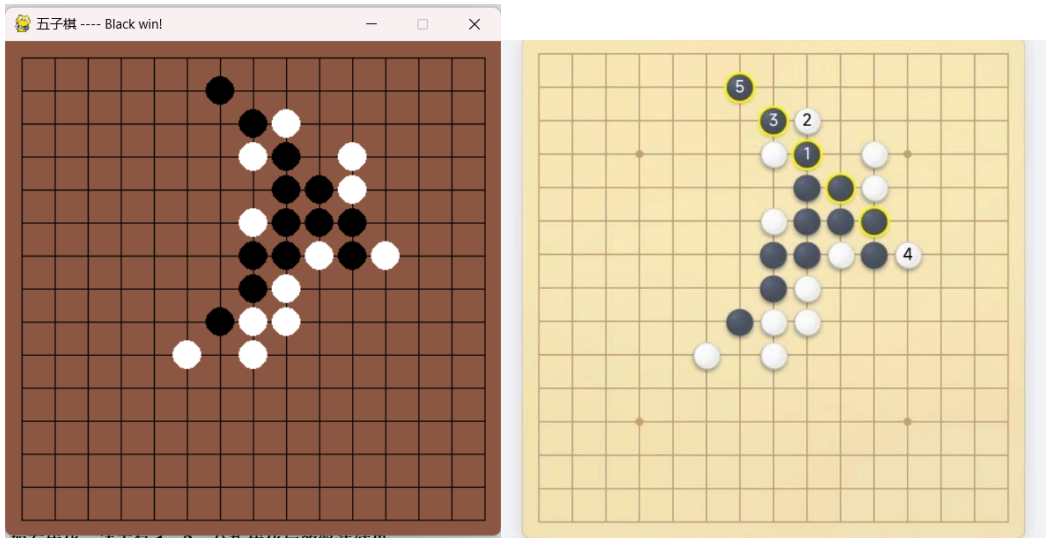
第一步的所有评价值，周围为 0 表示 boundary 剪枝，中间的 0 值表示有棋子。可以看到第八行第九列的评价值最高 3440，因此落子在第八行第九列。

2. 实验结果展示示例（可图可表可文字，尽量可视化）

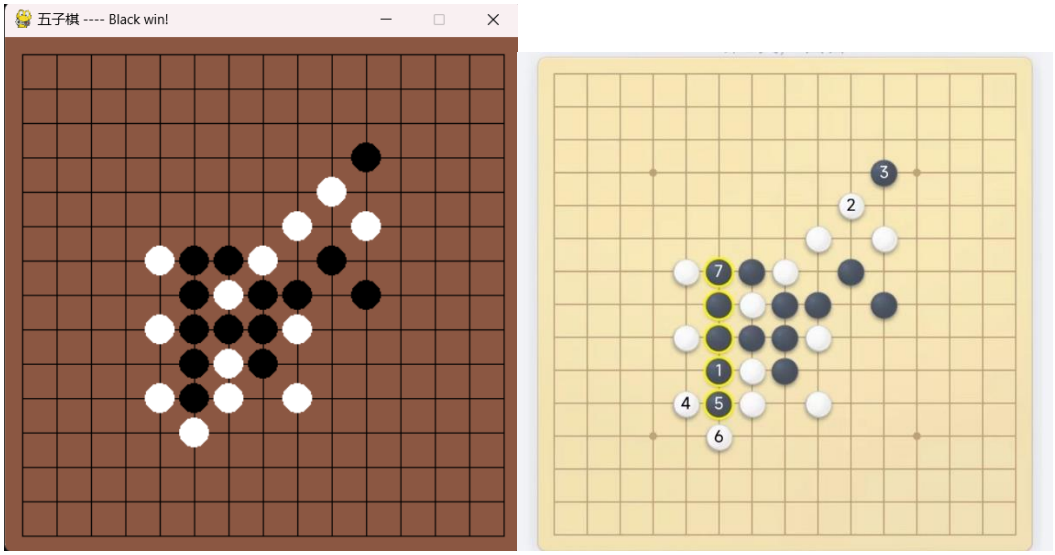
如果没有特别声明，alpha-beta 剪枝的搜索深度设置为 2 即可通关，某些较难的关卡请将深度设置为 3，此时每一步落子耗时大约 1-2min，请耐心等待。

算法在前二十关中未通过 9、14、16、17，其他均顺利通过，其中第八关搜索深度设置为 3

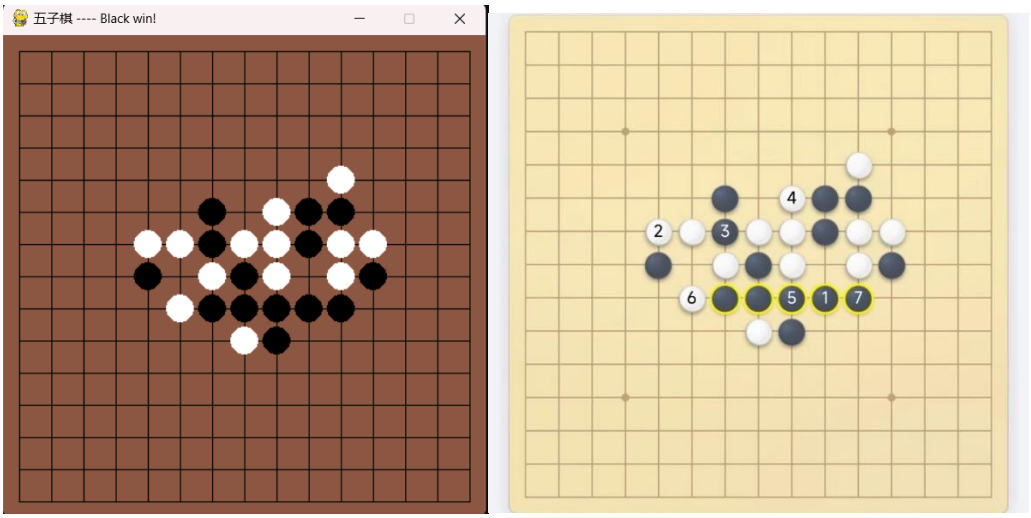
测试用例 1：


五子棋 ---- Black win!
5
3
2
1
4

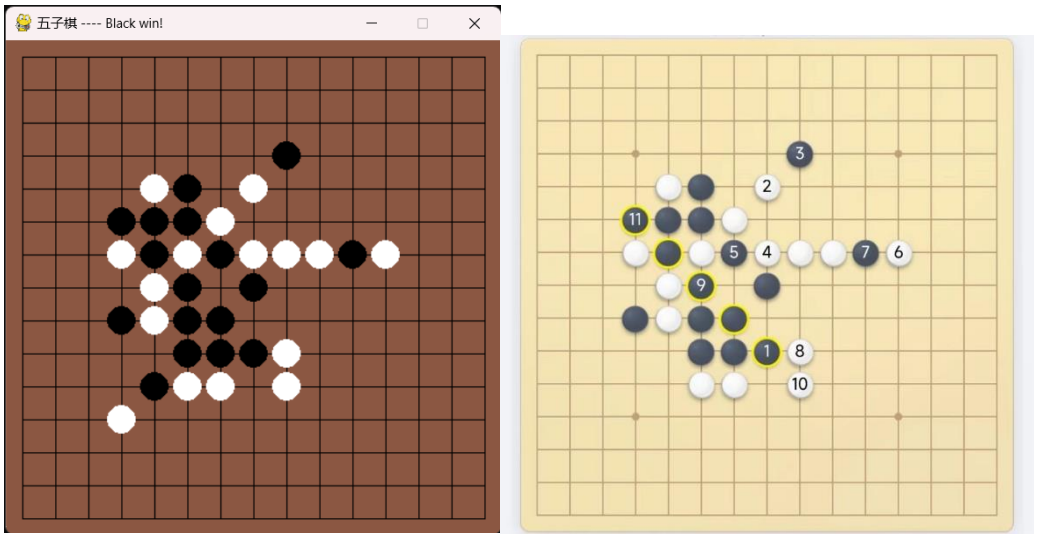
测试用例 2：


五子棋 ---- Black win!
3
2
7
1
4
5
6

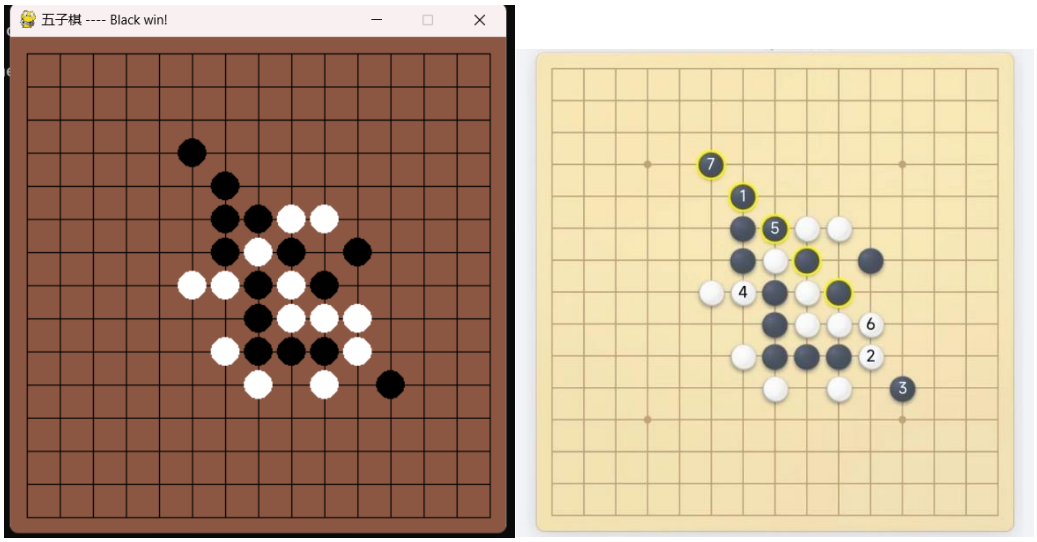
测试用例 3：


五子棋 ---- Black win!
4
2
3
6
5
1
7

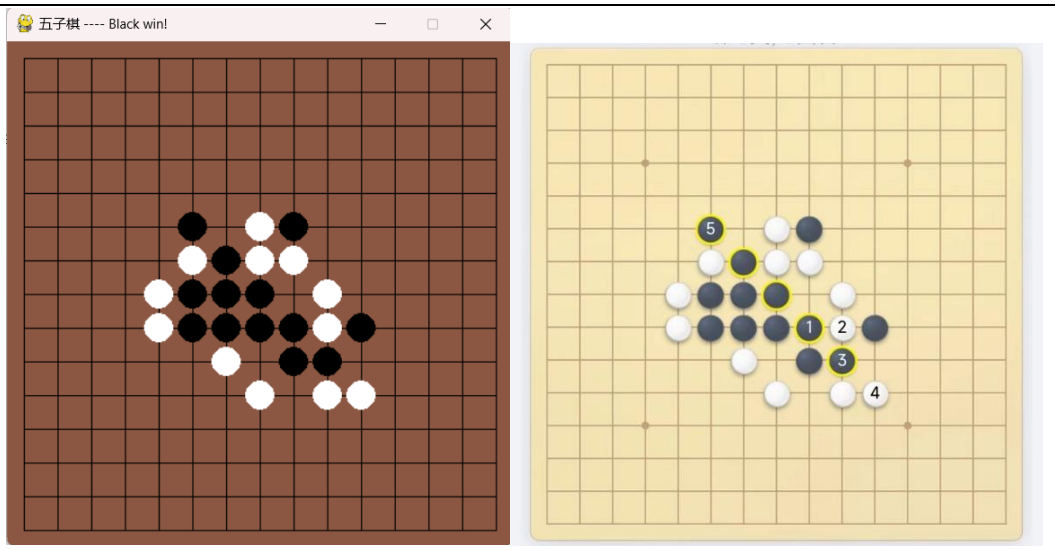
测试用例 4：



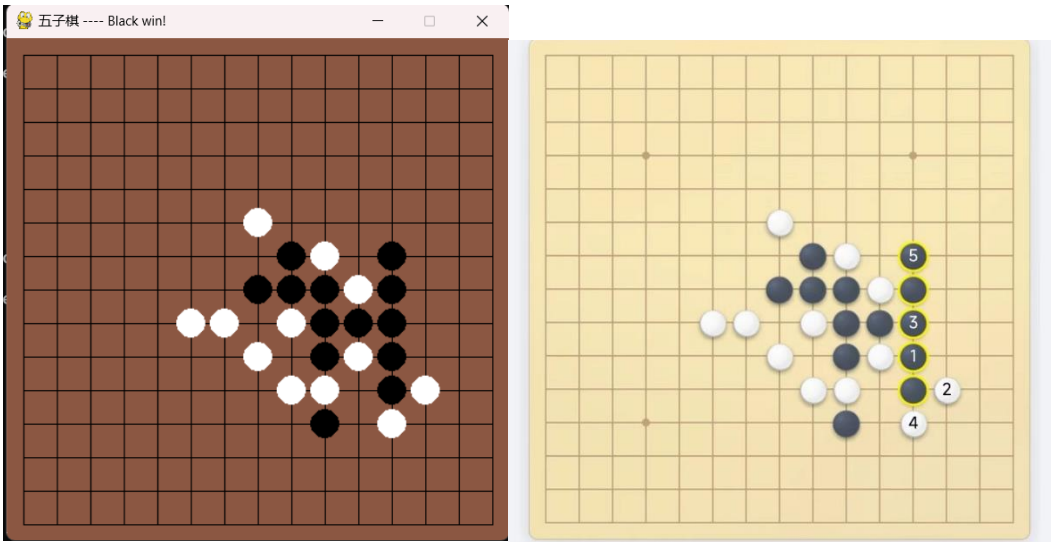
测试用例 5：



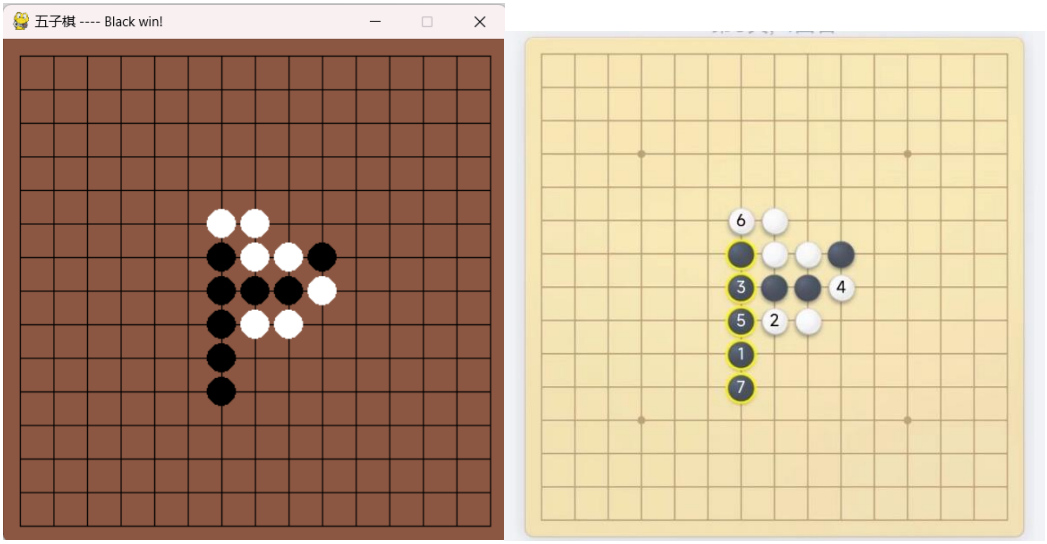
测试用例 6：



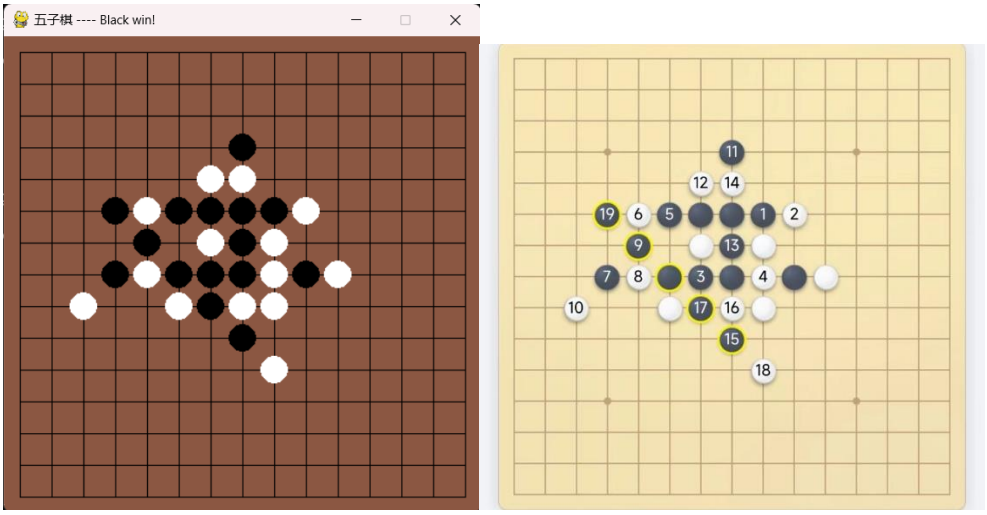
测试用例 7：



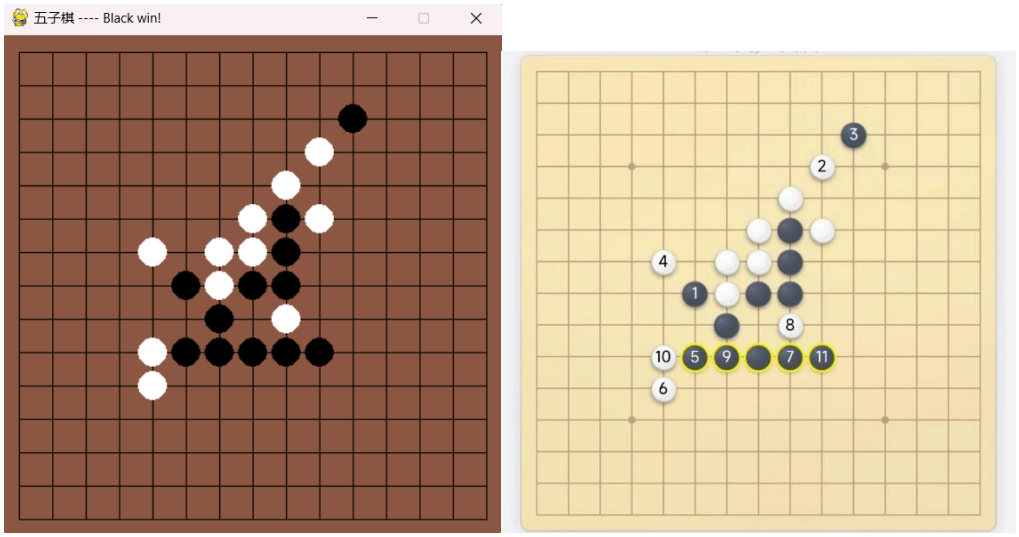
测试用例 8：将搜索深度设置为 3 可以通过



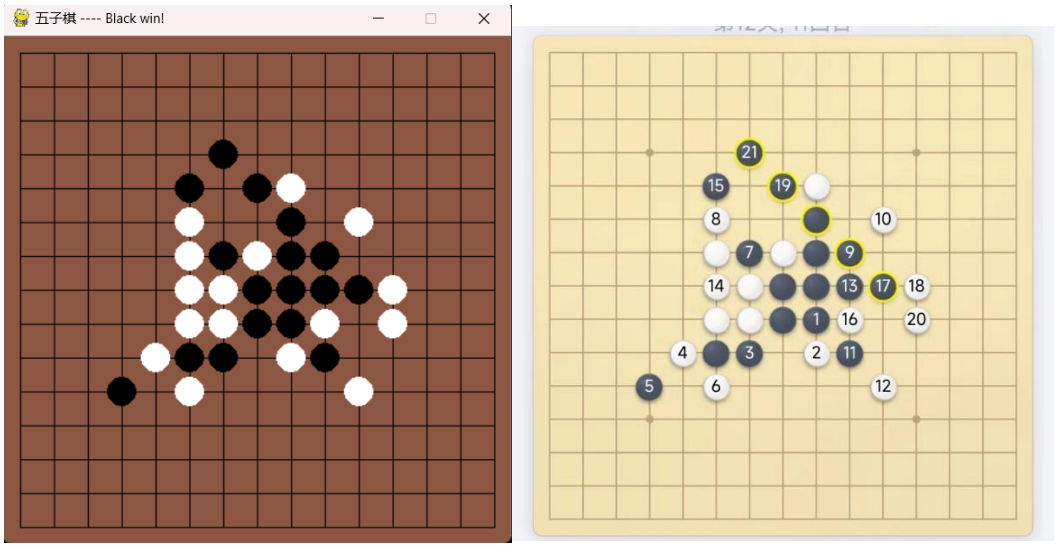
测试用例 10：



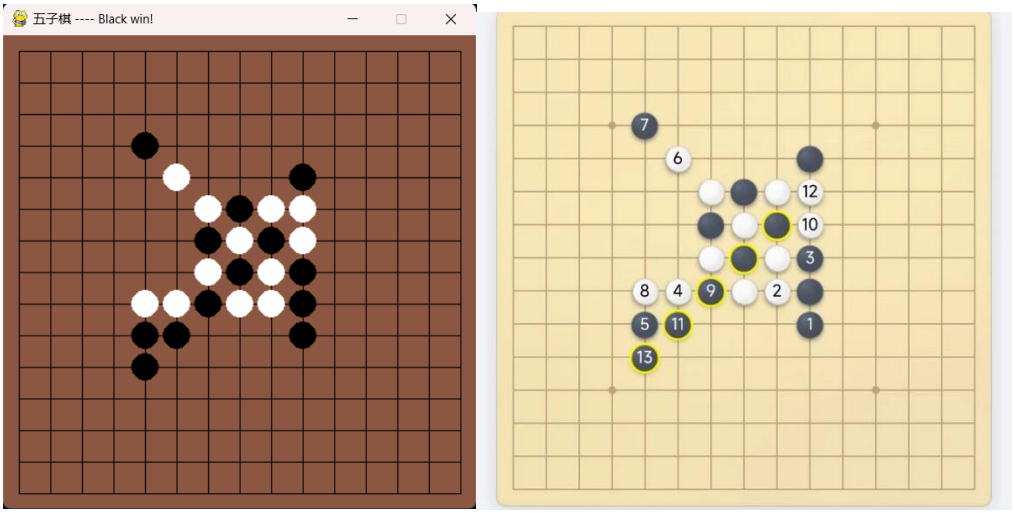
测试用例 11：



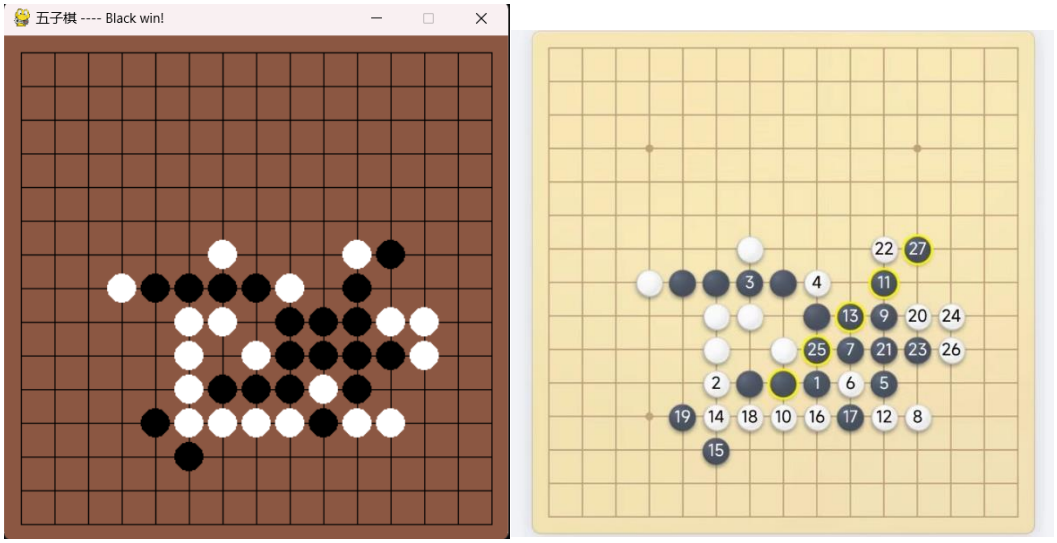
测试用例 12：



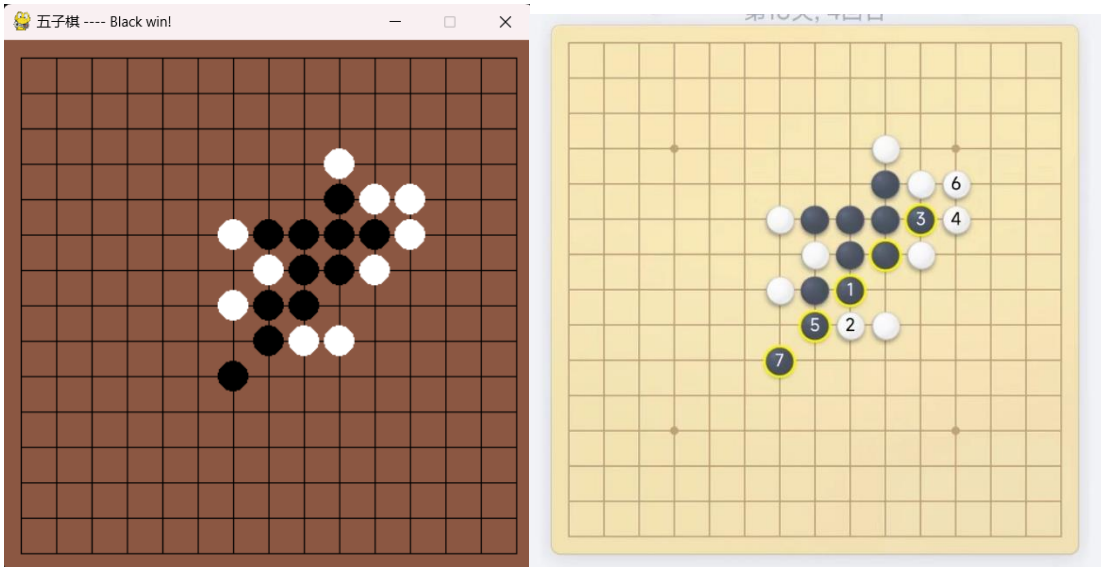
测试用例 13：



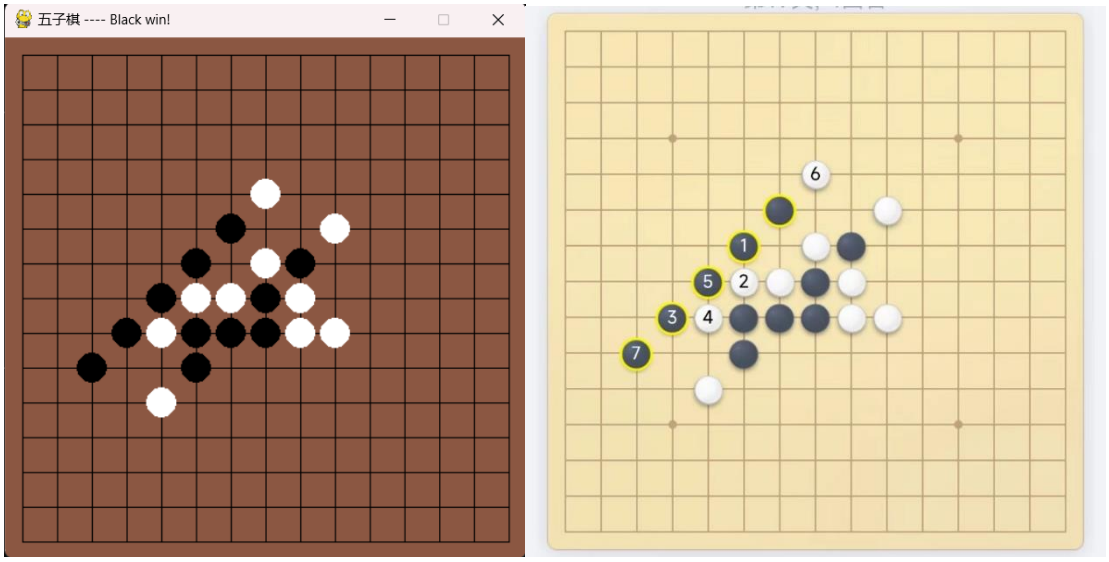
测试用例 15：



测试用例 18：



测试用例 19：



测试用例 20：