中山大学计算机学院

人工智能

本科生实验报告

（2022学年春季学期）
课程名称：Artificial Intelligence

教学班级	专业(方向)	学号	姓名
计科三班	计算机科学与技术	21307185	张礼贤

一、实验题目

使用Alpha - Beta剪枝算法编写五子棋博弈程序，实现人机对弈

二、实验内容

1、算法原理

1. **构建博弈树：**

- 首先需要构建一棵博弈树，树的每个节点表示棋盘上一种可能的棋局，树的深度表示搜索的步数。
- 其中，根节点表示MAX节点，它的下一层为MIN节点，每个MAX节点要从它的MIN节点子节点中**选取评估值最大**的作为α值；MIN节点则相反，选取子节点中**最小的值**作为β值。当递归到叶子节点的时候，则对当前的棋局进行评分，并返回相应的分数。

2. **评估函数：**

- 为了决定哪些节点需要继续搜索，需要使用一个评估函数来评估每个节点的价值。在五子棋中，评估函数可以计算当前局面的分数，如棋子数量、连续棋子数等等。在本次实验中，我选择了棋形匹配的方法，对每个棋形都进行了分数的统计和计算，最终将自己和对手的分数相减，得到一个最终的分数。

- 而对于各种棋形的统计，下面也列出了棋形的表示，其中以黑子为模板，0表示空位，1表示黑子，2表示白子：

棋形表示	棋形名称
"11111"	连五
"011110"	活四
"211110" , "210111" , "211011" , "211101"	冲四
"011010","01110"	活三
"001112","010112","011012","10011","10101","2011102"	眠三
"001100","010100","010010","001010"	活二
"211000","210100","210010","10001","2100012","2010012","2001012","2001102","2010102"	眠二

3. **Alpha-Beta剪枝：** 在搜索过程中，每个节点都有一个最优值和一个次优值，表示当前节点能够得到的最大价值和次大价值。Alpha表示当前最大值的下限，Beta表示当前最小值的上限。当搜索到某个节点时，如果发现它的最优值小于等于Alpha，那么就可以剪掉它的子树，因为对于当前玩家来说，它永远不会选择这个节点。同理，如果发现它的次优值大于等于Beta，那么就可以剪掉它的子树，因为对于对手来说，它永远不会选择这个节点。这样就可以大幅减少搜索的节点数量，提高搜索速度。

4. **最佳走法：** 当搜索到最后一层时，就可以使用评估函数计算出每个叶子节点的分数，并选择分数最高的节点作为最佳走法。

2、伪代码和流程图

伪代码:

```
def AlphaBetaSearch(board, alpha, beta, Player, depth):
    moves = create_move()
    best_score = inf if turn == 0 else -inf
    best_move = None
    if(depth == 0):
        value = calculate()
        return  best_move, value
    #轮数的判定和剪枝
    if(Player is Black):
        for each m in moves:
            set move in board
            value = AlphaBetaSearch(board, alpha, beta, 0, depth-1)[1]
            undo move in board
            if(value > best_score):
                flush best_move and best_score
            alpha = max(alpha,best_score)
            #if(alpha >= beta):break
    else:
        for m in moves:
            set move in board
            value = AlphaBetaSearch(board, alpha, beta, 0, depth-1)[1]
            undo move in board
            if(value < best_score):
                flush best_move and best_score
            beta = min(alpha,best_score)
            #if(alpha >= beta):break

    return best_move,best_score
```

3、关键代码展示

```
import re
from collections import Counter
from numpy import Infinity as inf

node_num = 0 #全局变量，统计搜索的节点数量
White = (255,255,255)
Black = (0,0,0)
pos = [[(7 - max(abs(x - 7), abs(y - 7))) for x in range(15)] for y in
range(15)]
#将棋盘的每个位置赋予权值，中心为7，往外扩展一圈则递减1
pattern_white = {
    "22222" : "five", #连五 white
    "022220": "live_four", #活四 white

"122220|120222|122022|122202|022221|222021|220221|202221":"skip_four", #
冲四 white
    "022020|02220|020220":"live_three" ,#活三 white
    "002221|020221|022021|20022|20202|1022201" : "sleep_three", #眠三
white
    "002200|020200|020020|002020" : "live_two", #活二 white
    "122000|120200|120020|20002|1200021|1020021|1002021|1002201|1020201"
: "sleep_two", #眠二 white
    } #模式匹配串，方便进行正则匹配
pattern_black = {
    "11111" : "five", #连五 black
    "011110": "live_four", #活四 black

"211110|210111|211011|211101|011112|111012|110112|101112":"skip_four", #
冲四 black
    "011010|01110|010110":"live_three"  ,#活三 black
    "001112|010112|011012|10011|10101|2011102" : "sleep_three", #眠三
black
    "001100|010100|010010|001010" : "live_two", #活二 black
    "211000|210100|210010|10001|2100012|2010012|2001012|2001102|2010102"
: "sleep_two", #眠二 black
    }

def evaluate(board,patterns):
    """返回pattern相应的匹配字典"""
    rows = len(board)
```

```
    cols = len(board[0])
    all_lines = []
    for i in range(rows):   #将每一行转换为字符串
        line = "".join(str(x) for x in board[i])
        all_lines.append(line)
    for j in range(cols):   #将每一列转换为字符串
        line = "".join(str(board[i][j]) for i in range(rows))
        all_lines.append(line)
    for k in range(-rows+1, cols):  #将每一左斜行转换为字符串
        line = "".join(str(board[i][i+k]) for i in range(rows) if 0 <=
i+k < cols)
        all_lines.append(line)
    for k in range(rows+cols-1):    #将每一右斜行转换为字符串
        line = "".join(str(board[i][k-i]) for i in range(rows) if 0 <= k-
i < cols)
        all_lines.append(line)
    scores = Counter()  #调用counter函数，统计每种棋形的个数，以字典的方式存
储
    for line in all_lines:
        for pattern, score in patterns.items():
            if re.search(pattern, line):    #进行正则匹配
                scores[score] += 1
    return scores   #返回存储了各种棋形以及其个数的字典

def calculate(pos,board,scores_mine,scores_opponent):
    """分数计算函数"""
    score1 = 0  #自己方的分数的初始化
    for key in scores_mine:
        #遍历存储了自己棋形的字典并赋分
        if(key == "five"):return 10000000   #遇到连五直接返回
        if(key == "skip_four" and scores_mine[key] > 1):score1 += 1000000
        if(key == "live_four"):score1 += 40000
        if(key == "skip_four"):score1 += 10000
        if(key == "live_three" and scores_mine[key] > 1):score1 += 3000
        if(key == "live_three" and scores_mine[key] == 1):score1 += 500
        if(key == "live_two" and scores_mine[key] > 1):score1 += 40
        if(key == "live_two" and scores_mine[key] == 1):score1 += 40
        if(key == "sleep_two"):score1 += scores_mine[key]
    if("live_three" in scores_mine and "sleep_three" in
scores_mine):score1 += 1000*scores_mine["sleep_three"]
    if("live_three" in scores_mine and "live_two" in scores_mine):score1
+= 500*scores_mine["live_two"]
    for i in range(len(board)):
```

```
        for j in range(len(board[i])):
            if(board[i][j] == 1): score1 += pos[i][j]
    score2 = 0
    for key in scores_opponent:
        if(key == "five"):return  -10000000
        if(key == "skip_four" and scores_opponent[key] > 1):score2 +=
100000
        if(key == "live_four"):score2 += 40000
        if(key == "skip_four"):score2 += 10000
        if(key == "live_three" and scores_opponent[key] > 1):score2 +=
3000
        if(key == "live_three" and scores_opponent[key] == 1):score2 +=
500
        if(key == "live_two" and scores_opponent[key] > 1):score2 += 40
        if(key == "live_two" and scores_opponent[key] == 1):score2 += 40
        if(key == "sleep_two"):score2 += scores_opponent[key]
    for i in range(len(board)):
        for j in range(len(board[i])):
            if(board[i][j] == 2): score2 += pos[i][j]
    return  4*score1 - score2   #调整参数，使其更富于进攻性

def evaluate_point(board,x,y,turn,patterns):
    """统计当前落点的四个方向的棋形及个数,方便连五的特判和返回"""
    lines = []
    line = "".join(str(board[x][i]) for i in range(15))
    lines.append(line)
    line = "".join(str(board[i][y]) for i in range(15))
    lines.append(line)
    line = "".join(str(board[x+k][y+k]) for k in range(-15,15) if
0<=x+k<15 and 0<=y+k<15)
    lines.append(line)
    line = "".join(str(board[x+k][y-k]) for k in range(-15,15) if
0<=x+k<15 and 0<=y-k<15)
    lines.append(line)
    scores = Counter()
    for line in lines:
        for pattern, score in patterns.items():
            if re.search(pattern, line):
                scores[score] += 1
    return scores

def create_move(pos,board,turn):
        '''生成移动序列'''
```

```
        moves = []
        left = 15
        right = 0
        up = 15
        down = 0    #初始化边界值
        flag = 0
        for i in range(len(board)):
            for j in range(len(board[i])):
                if(board[i][j] != 0):
                    left = min(left,j)
                    right = max(right,j)
                    up = min(i,up)
                    down = max(i,down)
                    flag = 1
        #统计当前落子集群的上下左右边界
        if(flag == 0):  #如果棋盘上没有落子，则直接下在中间的位置(先手)
            moves.append((7,7,7))
            return moves
        if(left - 1 >= 0):left -= 1
        if(right + 1 < len(board)):right += 1
        if(up - 1 >= 0): up -= 1
        if(down + 1 < len(board)):down += 1
        #对上面统计的边界进行扩展，提高其落子的准确性
        val = 1 if turn == 1 else 2
        pattern1 = pattern_black if turn == 1 else pattern_white
        pattern2 = pattern_white if turn == 1 else pattern_black
        for i in range(up,down+1):
            for j in range(left,right+1):
                if(board[i][j]==0): moves.append((pos[i][j],i,j))   #将落
点的位置权重作为排序依据加入到列表中
        moves.sort(reverse = True)  #对列表进行排序，优先选择靠近中间的落点
        return moves

def AlphaBetaSearch(board, alpha, beta, turn, depth):
    """剪枝函数"""
    global node_num
    node_num += 1   #利用全局变量统计搜索的节点
    moves = create_move(pos,board,turn) #生成可移动序列
    best_score = inf if turn == 0 else -inf #定义best_score
    best_move = None    #定义best_move
    if(depth == 0 or len(moves) == 0):  #终止条件判断
        oppo = evaluate(board,pattern_white)  #对手的棋形
        mine = evaluate(board,pattern_black)  #自己的棋形
```

```
            value = calculate(pos,board,mine,oppo)
            return  best_move, value,node_num  #计算分数返回
        #轮数的判定和剪枝
        if(turn == 1):
            for m in moves:
                board[m[1]][m[2]] = 1   #模拟在此位置进行落点
                cur = evaluate_point(board, m[1], m[2], turn, pattern_black)
#计算此位置是否形成连五
                if("five" in cur):  #如果已经有，则撤销上一步操作，直接返回
                    board[m[1]][m[2]] = 0
                    return m,10000000,node_num
                value = AlphaBetaSearch(board, alpha, beta, 0, depth-1)[1]  #
递归调用函数，计算value值
                board[m[1]][m[2]] = 0   #回溯，撤销上一步的操作
                if(value > best_score): #对于自己方，如果value大于当前的
best_score，则更新best_move 和 best_score
                    best_score , best_move = value ,m
                alpha = max(alpha,best_score)   #更新alpha值
                if(alpha >= beta):break    #进行判断剪枝
        else:
            #白棋进行落子，操作与上面的对称
            for m in moves:
                board[m[1]][m[2]] = 2
                cur = evaluate_point(board, m[1], m[2], turn, pattern_white)
                if("five" in cur):
                    board[m[1]][m[2]] = 0
                    return m,-10000000,node_num
                value = AlphaBetaSearch(board, alpha, beta, 1, depth-1)[1]
                board[m[1]][m[2]] = 0
                if(value < best_score):
                    best_score , best_move = value , m
                beta = min(beta,best_score)
                if(alpha >= beta):break

        return best_move,best_score,node_num    #返回
```

4、创新点和优化

匹配方式

对于每个棋盘的棋形统计，如果采用字符串匹配的方式，即采用遍历的方法进行匹配会十分慢，于是考虑使用正则表达式匹配字符串，并用字典存储相应的棋形和个数，方便进行分数的统计。

例如：

```
pattern_black = {
    "11111" : "five", #连五 black
    "011110": "live_four", #活四 black

"211110|210111|211011|211101|011112|111012|110112|101112":"skip_four", #
冲四 black
    "011010|01110|010110":"live_three"  ,#活三 black
    "001112|010112|011012|10011|10101|2011102" : "sleep_three", #眠三
black
    "001100|010100|010010|001010" : "live_two", #活二 black
    "211000|210100|210010|10001|2100012|2010012|2001012|2001102|2010102"
: "sleep_two", #眠二 black
    }
```

匹配的逻辑是用于判断五子棋中黑子的不同棋型，并给出对应的名称。它使用了正则表达式来匹配特定的棋型，如"11111"表示连五，"011110"表示活四等等。

具体来说，这段代码中定义了一个名为"pattern_black"的字典，它包含了不同的棋型和对应的名称。每个键都是一个正则表达式，用来匹配相应的棋型。如果匹配成功，则返回对应的名称。例如，如果棋盘上出现了五个黑子连成一条线的情况，则匹配到"11111"这个正则表达式，并返回对应的名称"five"。

并且该代码中使用了 "|" 来表示正则表达式的"或"操作，同时使用括号来分组表示相同棋型的不同情况。例如，"211110|210111|211011|211101|011112|111012|110112|101112"表示有八种不同的冲四情况。

棋盘落子剪枝

对于一张棋盘可能的落子情况，如果遍历整张棋盘的空位，会产生非常多的子状态，会极大减慢搜索的速度，而且很多空位并没有必要搜索。

于是，可以先遍历棋盘，确定现有的黑子和白子的集群的上下左右边界，在将边界进行适当扩展，这样做既可以确保落子的正确性，又可以确保搜索状态的缩小。

并且，当获得了落子序列后，可以根据每个将要落子的地方进行位置权重排序，可以优先选择靠近中间的落点，更好的提升搜索简剪枝效率

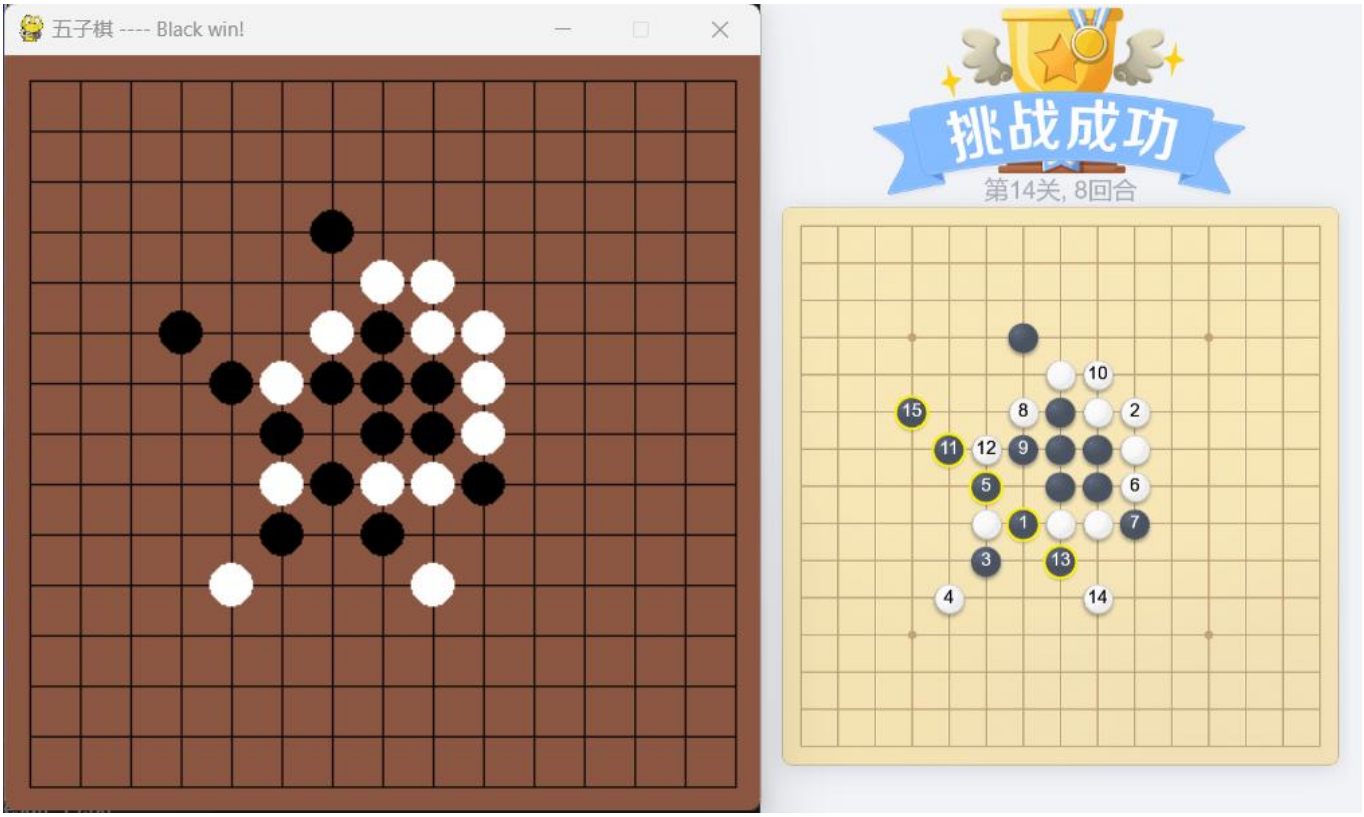
三、实验结果展示

test14

过关步骤及截图

对于14关，使用αβ剪枝的深度为两层即可以过，以下是过关的状态分析以及过关图像

1. 第一步黑棋下到坐标为（8，6）的位置，既防止了对方形成活四，又形成活三，其alpha值为-3720，此时白棋占优
2. 第二步白棋下至（5，9），防止黑方形成活四，第三步黑子下至（9，5）alpha值输出为254，此时黑棋已有冲四，局面占优
3. 第四步白棋下至（11，3），防止黑棋形成连五，第五步黑子下至（7，5）alpha值为-231，形成活三
4. 第六步白棋下至（7，9），封堵黑棋的活三并且形成自己的活三，为攻守兼备的打法；第七步黑子下至（8，9）封堵白棋的活三，并且形成黑方的活三，alpha值为285，局面占优
5. 第八步白棋下至（5，6），封堵黑棋的活三；第九步黑棋下至（6，6），形成眠三和活三的阵型，alpha值为309，局面占优
6. 第十步白棋下至（4，8），封堵黑棋的活三；第十一步黑棋下至（6，4），形成冲四逼迫白棋防守，alpha值为5660，局势大优
7. 第十二步白棋下至（5，6），被动防守；第十三步黑棋下至（9，7），形成活四，白棋无力回天
8. 第十四步白棋下至（10，8）封堵，但是无力回天，黑棋随后下至（5，3）形成连五赢得胜利



alpha剪枝效率分析

- **剪枝前 and 优化前：**

步数	黑方落子位置	落子时间
step1	(8,6)	4.876 seconds
step2	(9,5)	4.490 seconds
step3	(7,5)	8.988 seconds
step4	(8,9)	9.308 seconds
step5	(6,6)	9.018 seconds
step6	(6,4)	7.948 seconds
step7	(9,7)	7.980 seconds
step8	(5,3)	5.086 seconds

总的搜索节点数为26072

- **剪枝后 and 优化后：**

步数	黑方落子位置	落子时间
step1	(8,6)	0.206 seconds
step2	(9,5)	0.741 seconds
step3	(7,5)	1.969 seconds
step4	(8,9)	0.521 seconds
step5	(6,6)	0.552 seconds
step6	(6,4)	0.807 seconds
step7	(9,7)	0.658 seconds
step8	(5,3)	0.890 seconds

总共的搜索节点数为3149

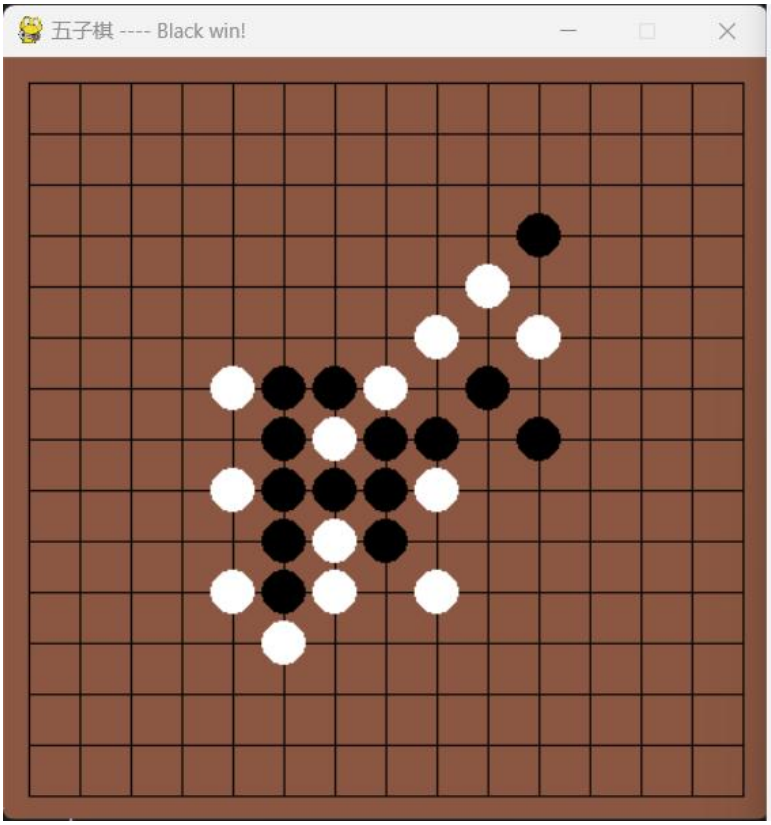
- **剪枝效率：87.92%**

---

test2

过关步骤以及截图

1. 第一步黑子落在（9，5），形成双活三，alpha值为2148，局面占优
2. 第二步白子落在（4，9），形成冲四；第三步黑子落在（3，10）顺势防守，alpha值为2156
3. 第二步白子落在（10，4），防守其中一个活三；第三步黑子落在（10，5）顺势形成活四，alpha值为40168，胜局已定
4. 第四步白子落在（11，5）；第五步黑子落在（6，5）形成连五胜利



αβ剪枝效率分析

- **剪枝前 and 优化前：**

步数	黑方落子位置	落子时间
step1	(9,5)	4.490 seconds
step2	(3,10)	6.886 seconds
step3	(10,5)	10.071 seconds
step4	(6,5)	0.847 seconds

总的搜索节点数为11122

- **剪枝后 and 优化后：**

步数	黑方落子位置	落子时间
step1	(9,5)	0.689 seconds
step2	(3,10)	6.125 seconds
step3	(10,5)	0.879 seconds
step4	(6,5)	0.372 seconds

总共的搜索节点数为3636

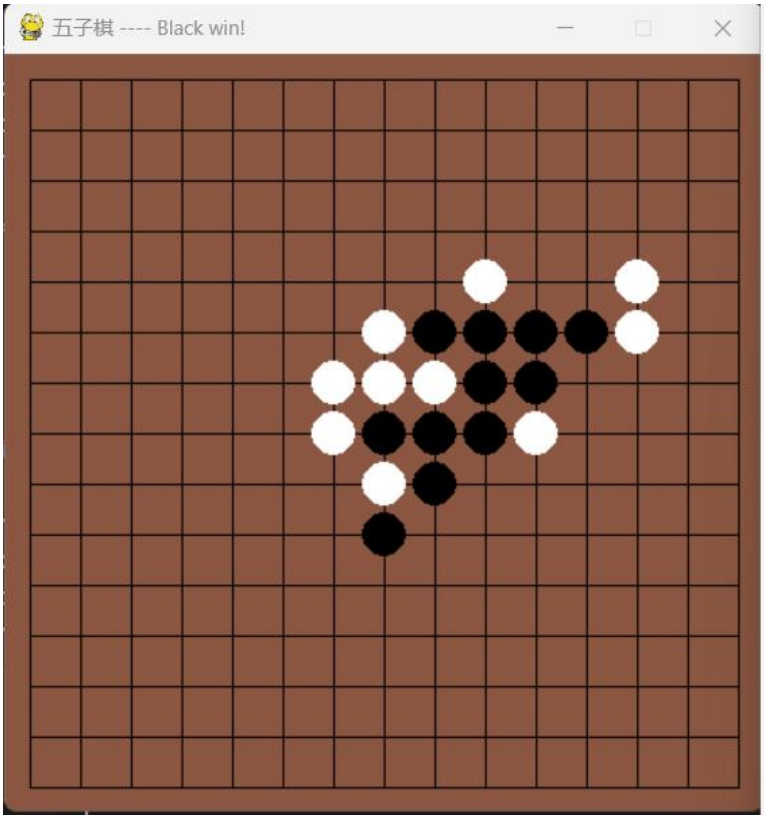
- **剪枝效率：67.31%**

---

test20

过关步骤及截图

1. 第一步黑子先落在（8，8）形成双活二局面，因为预见到了后面会形成活三以及冲四，因此下在这里，alpha值为44275，局面占优
2. 第二步白子落在（6，6）形成双活二眠三；黑棋顺势下在（5，11）形成活三冲四模型，alpha值为160286，局面大优
3. 第四步白子只能落在（5，12）进行封堵；黑棋落在（6，10）形成活四，锁定胜局
4. 第六步白子落在（4，12）封堵活四；但是黑棋顺势落在（9，7）形成连五，胜利！



αβ剪枝效率分析

剪枝前 and 优化前

步数	黑方落子位置	落子时间
step1	(8,8)	86.067 seconds
step2	(5,11)	79.285 seconds
step3	(6,10)	168.757 seconds
step4	(9,7)	0.920 seconds

总共的搜索节点数为227580

剪枝后 and 优化后

步数	黑方落子位置	落子时间
step1	(8,8)	9.853 seconds
step2	(5,11)	12.685 seconds
step3	(6,10)	7.751 seconds
step4	(9,7)	0.466 seconds

总共的搜索节点数为15568

- **剪枝效率：93.16%**

对比分析结果

- 当搜索层次越深，剪枝的效率就越高，test14 和 test2 都只搜索了两层，但是test20搜索了三层，可以得到90%以上的剪枝率，明显效果更为显著
- 剪枝前搜索两层的话，平均一步要5秒左右；如果进行剪枝的话，平均一步只需要0.7秒，在速度方面体现出了相当大的优势，因此可以具有足够的理由向搜索三层进行拓展，在第二十关证明这种假设是合理的
- 如果搜索三层，剪枝前每一步平均需要两分钟，剪枝后平均一步7秒左右，速度优化更为明显，于是可以将深度适当拓展为4层，这个时候对于不同的棋形搜索时间不同，平均下来为两分钟一步，在合理范围内

五、参考资料

[棋形与评估函数方面的参考] (https://blog.csdn.net/marble_xu/article/details/90450436)