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
2班	计算机科学与技术	21307174	刘俊杰

一、实验题目

Week 7 博弈树搜索 Alpha-beta剪枝

二、实验内容

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。棋局评估方法可以参考已有文献（基于 alpha-beta 剪枝技术的五子棋 - 知乎 (zhihu.com)），但正式实验报告要引用参考过的文献和程序。

微信小程序“欢乐五子棋”中的残局闯关，闯过前20关。将落子位置输出到文件中作为结果。第14关较难，作为附加关卡。

结果分析

选择其中3关进行分析。将每一步的所有可能位置对应的评估函数值输出，分析是否合理；分析评估函数的剪枝效果，如剪掉节点数的占比

加分项（供参考）

算法实现优化分析，分析优化后时间与搜索深度的变化

不同评估函数对比，分析不同评估函数的剪枝效果

1. 算法原理

(1)极小化极大（Minimax）算法原理

极小化极大算法在完全信息零和博弈中，基于己方努力使得在N步后优势最大化（即评估函数输出值最大化）和对方努力使得N步后己方优势最小化这两个出发点，构建决策树。在决策树上通过这两个出发点的内在逻辑进行搜索，最后给出行动策略。

但是显然，极小化极大算法需要展开整个决策树，对于局面复杂的问题，其搜索空间将会非常大。

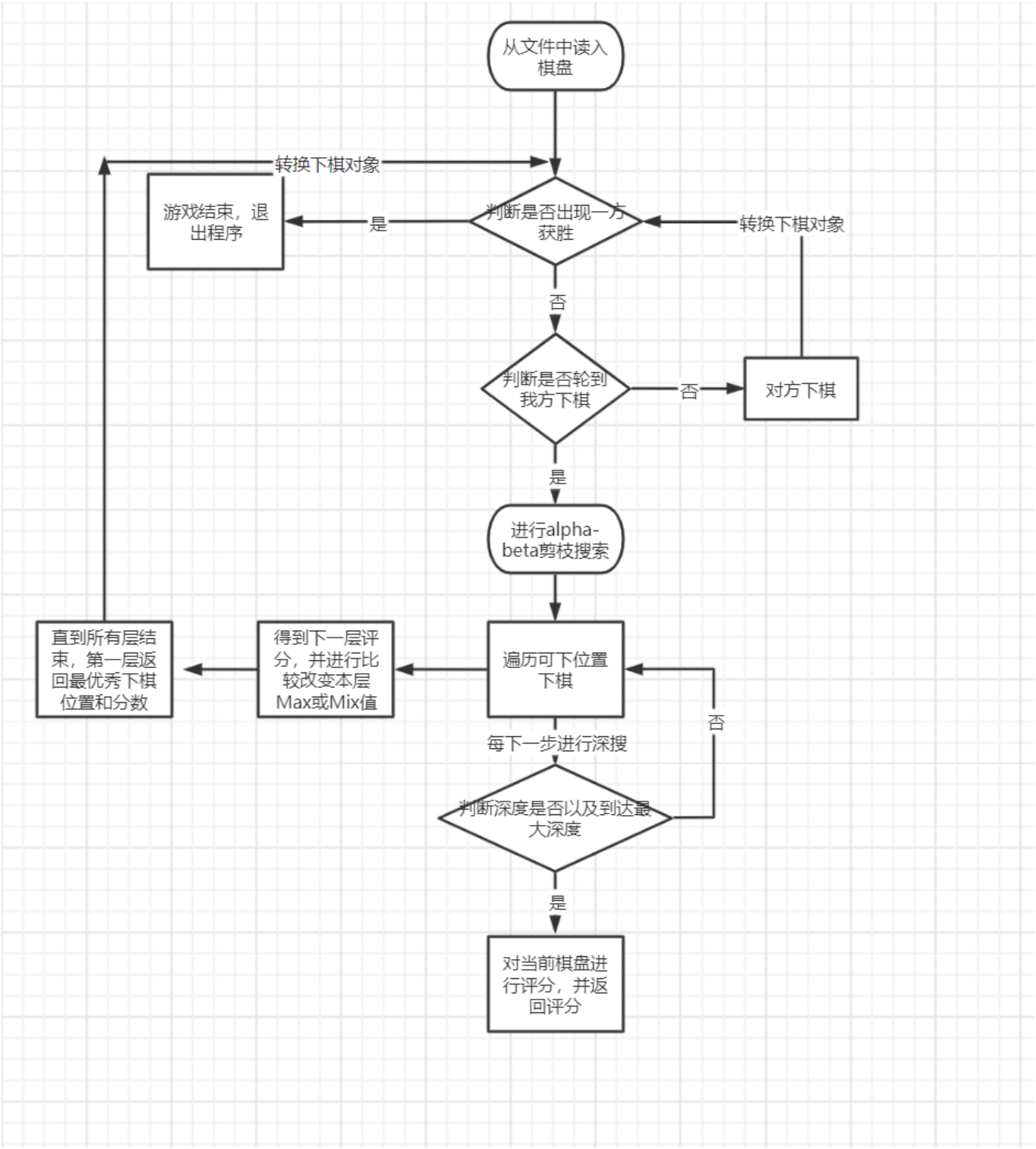
(2)Alpha-Beta剪枝算法:

#####有部分节点是否被搜索不会影响最后的结果，因此，无需展开此类节点以及计算此类节点的子节点的估值。通过上述方法，可节省算法的搜索时间。

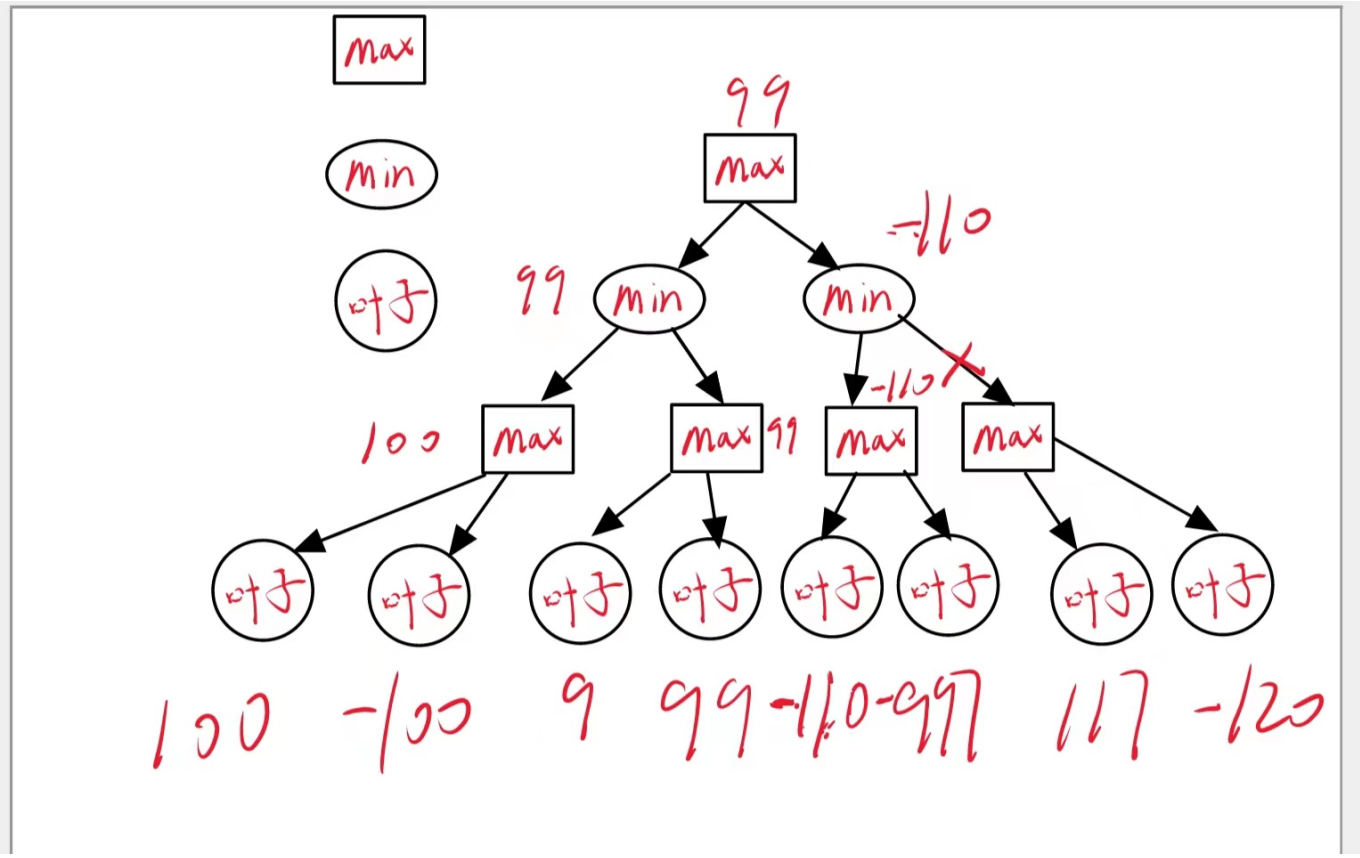
(1)开始构建决策树;
(2)将估值函数应用于叶子节点;
(3)使用深度优先搜索顺序构建和搜索决策树，传递并更新α 、 β 、 节 点 min 、max 值 、 alpha、beta、节点minimax值α、β、节点minimax值;
max结点更新α值(下限)，Min结点更新β值(上限)。
(4)从根结点选择评估值最大的分支，作为行动策略。

2.流程图

1. 该程序下棋的流程图(具体alpha-beta剪枝在下一张图)



2.搜索三层的alpha-beta剪枝(假设每个状态的后继有两个，且给定叶子节点分数)



3.关键代码展示

(1).构造协助我们评分的类

```
class ChessAI():
    def __init__(self, chess_len):
        self.len = chess_len#棋盘的宽度
        self.record = [[[0,0,0,0] for x in range(chess_len)] for y in
range(chess_len)]#记录每一个位置的竖直 横向 左斜 右斜方向是否被访问评分过
        self.count = [[0 for x in range(CHESS_TYPE_NUM)] for i in range(2)]#记录黑
白棋子各种棋形的数目
        self.pos_score = [[(7 - max(abs(x - 7), abs(y - 7))) for x in
range(chess_len)] for y in range(chess_len)]#给棋盘上每个位置的初始评分，由中间向四周
递减
```

(2).棋盘评分重置函数，每次评分前都需要重置

```
    def reset(self):#棋盘评分重置函数
        for y in range(self.len):
            for x in range(self.len):
                for i in range(4):
                    self.record[y][x][i] = 0 #将各个方向访问评分标志置为0，代表未评分

        for i in range(len(self.count)):
```

```
            for j in range(len(self.count[0])):
                self.count[i][j] = 0 #将黑白各种棋形数目置为0
```

(3).寻找可下棋子的位置，我们可下的位置是周围两步之内有棋子的位置，在空盘或者是第一次下时可下位置可以不考虑上述条件限制。

```
    def hasNeighbor(self, board, x, y, radius):#判断x、y周围是否有棋子
        start_x, end_x = (x - radius), (x + radius)
        start_y, end_y = (y - radius), (y + radius)
        global sum
        if sum==0:return True
        sum+=1
        for i in range(start_y, end_y+1):
            for j in range(start_x, end_x+1):
                if i >= 0 and i < self.len and j >= 0 and j < self.len:
                    if board[i][j] != -1:
                        return True
        return False

    # get all positions near chess
    def get_move(self, board):#获取可下棋子位置
        moves = []
        radius = 2
        for y in range(self.len):
            for x in range(self.len):
                if board[y][x] == -1 and self.hasNeighbor(board, x, y, radius):
                    score = self.pos_score[y][x]
                    moves.append((score, y, x))
        moves.sort(reverse=True)
        return moves
```

(4). 估价函数:每次博弈树到达叶子节点时对整个棋盘进行评分，对每个棋子的竖直、横向、左斜、右斜的四个方向的直线进行评分、统计黑白棋子的棋形数目，每个棋子的四个方向都只能评分一次，避免重复计算棋形，再通过棋形进行评分。

```
    def getLine(self, board, x, y, dir_offset, mine, opponent):#获取x、y这个位置的
dir_offset方向的直线棋子列表
        line = [0 for i in range(9)]
        tmp_x = x + (-5 * dir_offset[0])
        tmp_y = y + (-5 * dir_offset[1])
        for i in range(9):
            tmp_x += dir_offset[0]
            tmp_y += dir_offset[1]
            if (tmp_x < 0 or tmp_x >= self.len or
                tmp_y < 0 or tmp_y >= self.len):
                line[i] = opponent # 将超出范围的位置当作敌方棋子
```

```
            else:
                line[i] = board[tmp_y][tmp_x]

        return line

    def evaluatePoint(self, board, x, y, mine, opponent):#对x、y四个方向评分
        dir_offset = [(1, 0), (0, 1), (1, 1), (1, -1)] # direction from left to
right
        for i in range(4):
            if self.record[y][x][i] == 0:
                self.analysisLine(board, x, y, i, dir_offset[i], mine, opponent,
self.count[mine])

    def evaluate(self, board, turn, checkWin=False):#估价函数
        self.reset()
        mine=turn#turn 1代表黑子下，0代表白子下
        opponent=abs(1-turn)
        for y in range(self.len):
            for x in range(self.len):
                if board[y][x] == mine:
                    self.evaluatePoint(board, x, y, mine, opponent)
                elif board[y][x] == opponent:
                    self.evaluatePoint(board, x, y, opponent, mine)
        mine_count = self.count[turn]
        opponent_count = self.count[abs(turn-1)]
        if checkWin:
            return mine_count[FIVE] > 0
        else:
            mscore, oscore = self.getScore(mine_count, opponent_count)
            return (mscore - oscore)

    def analysisLine(self, board, x, y, dir_index, dir_offset, mine, opponent,
count):#对x、y某个方向的棋形进行统计
        # record line range[left, right] as analysized
        def setRecord(self, x, y, left, right, dir_index, dir_offset):#标志某个方向
的某些棋子已经被评分了，避免重复计算
            tmp_x = x + (-5 + left) * dir_offset[0]
            tmp_y = y + (-5 + left) * dir_offset[1]
            for i in range(left, right+1):
                tmp_x += dir_offset[0]
                tmp_y += dir_offset[1]
                self.record[tmp_y][tmp_x][dir_index] = 1

        empty = -1
        left_idx, right_idx = 4, 4

        line = self.getLine(board, x, y, dir_offset, mine, opponent)#获取对应方向的
棋子列表

        while right_idx < 8:
            if line[right_idx+1] != mine:
                break
            right_idx += 1
        while left_idx > 0:
```

```
            if line[left_idx-1] != mine:
                break
            left_idx -= 1

        left_range, right_range = left_idx, right_idx
        while right_range < 8:
            if line[right_range+1] == opponent:
                break
            right_range += 1
        while left_range > 0:
            if line[left_range-1] == opponent:
                break
            left_range -= 1

        chess_range = right_range - left_range + 1
        if chess_range < 5:
            setRecord(self, x, y, left_range, right_range, dir_index, dir_offset)
            return 0

        setRecord(self, x, y, left_idx, right_idx, dir_index, dir_offset)

        m_range = right_idx - left_idx + 1

        # M:mine chess, P:opponent chess or out of range, X: empty
        if m_range == 5:
            count[FIVE] += 1

        # Live Four : XMMMMX
        # Chong Four : XMMMMP, PMMMMX
        if m_range == 4:
            left_empty = right_empty = False
            if line[left_idx-1] == empty:
                left_empty = True
            if line[right_idx+1] == empty:
                right_empty = True
            if left_empty and right_empty:
                count[FOUR] += 1
            elif left_empty or right_empty:
                count[SFOUR] += 1

        # Chong Four : MXMMM, MMMXM, the two types can both exist
        # Live Three : XMMMXX, XXMMMX
        # Sleep Three : PMMMX, XMMMP, PXMMMXP
        if m_range == 3:
            left_empty = right_empty = False
            left_four = right_four = False
            if line[left_idx-1] == empty:
                if line[left_idx-2] == mine: # MXMMM
                    setRecord(self, x, y, left_idx-2, left_idx-1, dir_index,
dir_offset)
                    count[SFOUR] += 1
                    left_four = True
                left_empty = True
```

```
            if line[right_idx+1] == empty:
                if line[right_idx+2] == mine: # MMMXM
                    setRecord(self, x, y, right_idx+1, right_idx+2, dir_index,
dir_offset)
                    count[SFOUR] += 1
                    right_four = True
                right_empty = True

            if left_four or right_four:
                pass
            elif left_empty and right_empty:
                if chess_range > 5: # XMMMXX, XXMMMX
                    count[THREE] += 1
                else: # PXMMMXP
                    count[STHREE] += 1
            elif left_empty or right_empty: # PMMMX, XMMMP
                count[STHREE] += 1

        # Chong Four: MMXMM, only check right direction
        # Live Three: XMXMMX, XMMXMX the two types can both exist
        # Sleep Three: PMXMMX, XMXMMP, PMMXMX, XMMXMP
        # Live Two: XMMX
        # Sleep Two: PMMX, XMMP
        if m_range == 2:
            left_empty = right_empty = False
            left_three = right_three = False
            if line[left_idx-1] == empty:
                if line[left_idx-2] == mine:
                    setRecord(self, x, y, left_idx-2, left_idx-1, dir_index,
dir_offset)
                    if line[left_idx-3] == empty:
                        if line[right_idx+1] == empty: # XMXMMX
                            count[THREE] += 1
                        else: # XMXMMP
                            count[STHREE] += 1
                        left_three = True
                    elif line[left_idx-3] == opponent: # PMXMMX
                        if line[right_idx+1] == empty:
                            count[STHREE] += 1
                            left_three = True



                    elif line[left_idx-3] == mine:  # MMXMM
                        setRecord(self, x, y, left_idx-1, left_idx-2, dir_index,
dir_offset)
                        count[SFOUR] += 1
                        right_three = True

                left_empty = True

            if line[right_idx+1] == empty:
                if line[right_idx+2] == mine:
                    if line[right_idx+3] == mine:  # MMXMM
```

```
                        setRecord(self, x, y, right_idx+1, right_idx+2, dir_index,
dir_offset)
                        count[SFOUR] += 1
                        right_three = True
                    elif line[right_idx+3] == empty:
                        #?????setRecord(self, x, y, right_idx+1, right_idx+2,
dir_index, dir)
                        if left_empty:  # XMMXMX
                            count[THREE] += 1
                        else:  # PMMXMX
                            count[STHREE] += 1
                        right_three = True
                    elif left_empty: # XMMXMP
                        count[STHREE] += 1
                        right_three = True

                right_empty = True

            if left_three or right_three:
                pass
            elif left_empty and right_empty: # XMMX
                count[TWO] += 1
            elif left_empty or right_empty: # PMMX, XMMP
                count[STWO] += 1

        # Live Two: XMXMX, XMXXMX only check right direction
        # Sleep Two: PMXMX, XMXMP
        if m_range == 1:
            left_empty = right_empty = False
            if line[left_idx-1] == empty:
                if line[left_idx-2] == mine:
                    if line[left_idx-3] == empty:
                        if line[right_idx+1] == opponent: # XMXMP
                            count[STWO] += 1
                left_empty = True

            if line[right_idx+1] == empty:
                if line[right_idx+2] == mine:
                    if line[right_idx+3] == empty:
                        if left_empty: # XMXMX
                            #setRecord(self, x, y, left_idx, right_idx+2,
dir_index, dir)
                            count[TWO] += 1
                        else: # PMXMX
                            count[STWO] += 1
                elif line[right_idx+2] == empty:
                    if line[right_idx+3] == mine and line[right_idx+4] == empty: #
XMXXMX
                        count[TWO] += 1

        return 0


    def getScore(self, mine_count, opponent_count):#根据棋形评分
```

```
        mscore, oscore = 0, 0
        if mine_count[FIVE] > 0:#我方连五，返回获胜
            return (1000000, 0)
        if opponent_count[FIVE] > 0:#敌方连五，返回失败
            return (0, 1000000)

        if mine_count[SFOUR] >= 2:
            mine_count[FOUR] += 1

        if opponent_count[FOUR] > 0:
            return (0, 9050)
        if opponent_count[SFOUR] > 0:
            return (0, 9040)

        if mine_count[FOUR] > 0:
            return (9030, 0)
        if mine_count[SFOUR] > 0 and mine_count[THREE] > 0:
            return (9020, 0)

        if opponent_count[THREE] > 0 and mine_count[SFOUR] == 0:
            return (0, 9010)

        if (mine_count[THREE] > 1 and opponent_count[THREE] == 0 and
opponent_count[STHREE] == 0):
            return (9000, 0)

        if mine_count[SFOUR] > 0:
            mscore += 2000

        if mine_count[THREE] > 1:
            mscore += 500
        elif mine_count[THREE] > 0:
            mscore += 100

        if opponent_count[THREE] > 1:
            oscore += 2000
        elif opponent_count[THREE] > 0:
            oscore += 400

        if mine_count[STHREE] > 0:
            mscore += mine_count[STHREE] * 10
        if opponent_count[STHREE] > 0:
            oscore += opponent_count[STHREE] * 10

        if mine_count[TWO] > 0:
            mscore += mine_count[TWO] * 4
        if opponent_count[TWO] > 0:
            oscore += opponent_count[TWO] * 4

        if mine_count[STWO] > 0:
            mscore += mine_count[STWO] * 4
        if opponent_count[STWO] > 0:
            oscore += opponent_count[STWO] * 4
```

```
        return (mscore, oscore)

    def check_win(self, board,i, j):#检测下在i、j位置是否能赢
        if board[i][j] == -1:
            return False
        color = board[i][j]
        for dire in dir:
            x, y = i, j
            x1,y1=i,j
            chess = []
            while board[x1][y1] == color:
                chess.append((x1, y1))
                x1, y1 = x1+dire[0], y1+dire[1]
                if x1 < 0 or y1 < 0 or x1 >= self.len or y1 >= self.len:
                    break
            x1,y1=x-dire[0],y-dire[1]
            if x1 < 0 or y1 < 0 or x1 >= self.len or y1 >= self.len:
                continue
            while board[x1][y1] == color:
                chess.append((x1, y1))
                x1, y1 = x1-dire[0], y1-dire[1]
                if x1 < 0 or y1 < 0 or x1 >= self.len or y1 >= self.len:
                    break
            if len(chess) >= 5:
                return True
        return False
```

(5).Alpha-beta剪枝搜索

```
def search(board,Ai,alpha,beta,if_max,turn,depth,limit):#深度优先搜索
    moves=Ai.get_move(board)
    max_score=-1000000#要和连五的分数一样
    min_score=1000000#要和连五的分数一样
    alpha_tmp=alpha
    beta_tmp=beta
    x_ans=-1
    y_ans=-1
    for move in moves:
        x,y=move[1],move[2]
        board[x][y]=turn
        if turn == 0:#如果发现能获胜或是失败，直接赋值，防止出现搜索过多使得能连五不连，
对方即将连五不堵
            if Ai.check_win(board,x,y):
                x_ans=x
                y_ans=y
                min_score=-1000000#
        else:#如果发现能获胜或是失败，直接赋值，防止出现搜索过多使得能连五不连，对方即将
连五不堵
            if Ai.check_win(board,x,y):
                x_ans=x
                y_ans=y
```

```
                max_score=1000000
        if depth==limit:#到达搜索深度限制，即叶子节点
            score=Ai.evaluate(board,turn)
            if turn%2 == 0 :
                score=-score
        else:
            x_tmp,y_tmp,score=search(board,Ai,alpha_tmp,beta_tmp,1-if_max,1-
turn,depth+1,limit)
        board[x][y]=-1
        if if_max == 1:
            if score>max_score:
                x_ans=x
                y_ans=y
                max_score=score
            if max_score>beta_tmp or max_score==beta_tmp:#Alpha-beta剪枝
                break
            if max_score>alpha_tmp:#更新Alpha
                alpha_tmp=max_score
        else :
            if score<min_score:
                x_ans=x
                y_ans=y
                min_score=score
            if min_score<alpha_tmp or min_score==alpha_tmp:#Alpha-beta剪枝
                break
            if min_score<beta_tmp:#更新beta
                beta_tmp=min_score
    if if_max==1:
        return (x_ans,y_ans,max_score)
    else :
        return (x_ans,y_ans,min_score)

def AlphaBetaSearch(board1, EMPTY, BLACK, WHITE, black):#alpha beta剪枝搜索
    board=rebuild(board1)
    Ai=ChessAI(len(board))
    if_max=1#1表示是最大层，0表示是最小层
    turn=0
    alpha=-10000000000
    beta=100000000000
    if black:#如果是黑棋下，turn=1即下黑棋
        turn=1
    limit=2#搜索深度
    x,y,score=search(board,Ai,alpha,beta,if_max,turn,1,limit)#深度优先搜索
    return (x,y,score)
```

4.创新点&优化

1.寻找可下棋子的位置，我们可下的位置是周围两步之内有棋子的位置，在空盘或者是第一次下时可下位置可以不考虑上述条件限制。

```
    def hasNeighbor(self, board, x, y, radius):#判断x、y周围是否有棋子
        start_x, end_x = (x - radius), (x + radius)
        start_y, end_y = (y - radius), (y + radius)
        global sum
        if sum==0:return True
        sum+=1
        for i in range(start_y, end_y+1):
            for j in range(start_x, end_x+1):
                if i >= 0 and i < self.len and j >= 0 and j < self.len:
                    if board[i][j] != -1:
                        return True
        return False

    # get all positions near chess
    def get_move(self, board):#获取可下棋子位置
        moves = []
        radius = 2
        for y in range(self.len):
            for x in range(self.len):
                if board[y][x] == -1 and self.hasNeighbor(board, x, y, radius):
                    score = self.pos_score[y][x]
                    moves.append((score, y, x))
        moves.sort(reverse=True)
        return moves
```

2.下棋出现了有活四可以连五或者是多处可以出现连五，但是不连，虽然最后也能赢，但是消耗了时间。

问题：

(1).通过分析，发现问题是，搜索三层时，黑子在第一步能赢，在第二步也能赢，黑子可能先获得了第二步赢的层的返回评分，导致后续第一步能赢的评分不会高于第一步的，导致下的棋子位置不会改变，导致多次黑子能赢但不赢，实际上是黑子先搜索到了后续赢的结果，导致它下出了后几步赢的棋路。

(2).同样和上面一样，对方要连五但是不堵它，原因是，黑子搜索到后面自己连五了，于是先返回自己赢的分数忽视了白子先赢了。

解决方法：搜索到前面几步就能赢或对方赢的位置，直接改变相应的位置，这样就能防止能赢不赢或者能防止对方赢。

```
if turn == 0:#如果发现能获胜或是失败，直接赋值，防止出现搜索过多使得能连五不连，对方即将连五不堵
    if Ai.check_win(board,x,y):
        x_ans=x
        y_ans=y
        min_score=-1000000#
else:#如果发现能获胜或是失败，直接赋值，防止出现搜索过多使得能连五不连，对方即将连五不堵
    if Ai.check_win(board,x,y):
        x_ans=x
        y_ans=y
        max_score=1000000
```

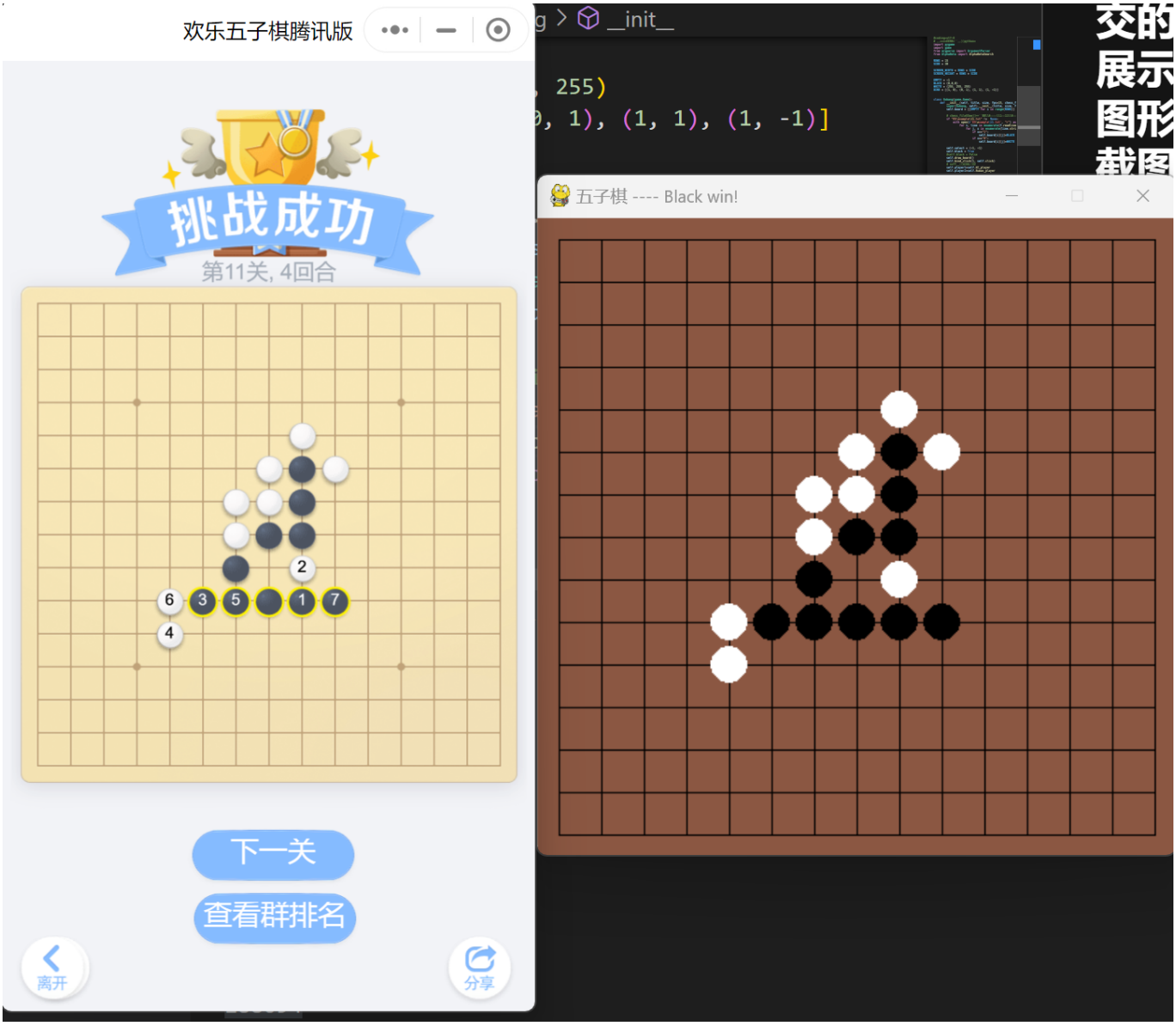
三、实验结果及分析(result文件中展示的都为用最小搜索层数解决的结果)

1. 实验结果展示示例(实验结果放在了提交的result文件夹中，这里为了方便直接展示展示黑子下棋的位置顺序和分数、图形化界面和小程序获胜后的下棋步骤截图)

```
20关可以全部通过
需要搜2层过的: 1 2 3 4 5 6 7 10 13 14 15 17 18
需要搜3层过的: 8 9 11 16 19 20
需要搜4层过的: 12
```

展示结果都为优化后的，以下展示关卡为11、13、14关，且14关展示分别为搜索2层和搜索3层的

(1)第11关(搜索层数：3) 搜索节点:188094



黑子下棋的位置顺序和分数

```
9 8 9020
9 5 9030
9 6 1000000
9 9 1000000
```

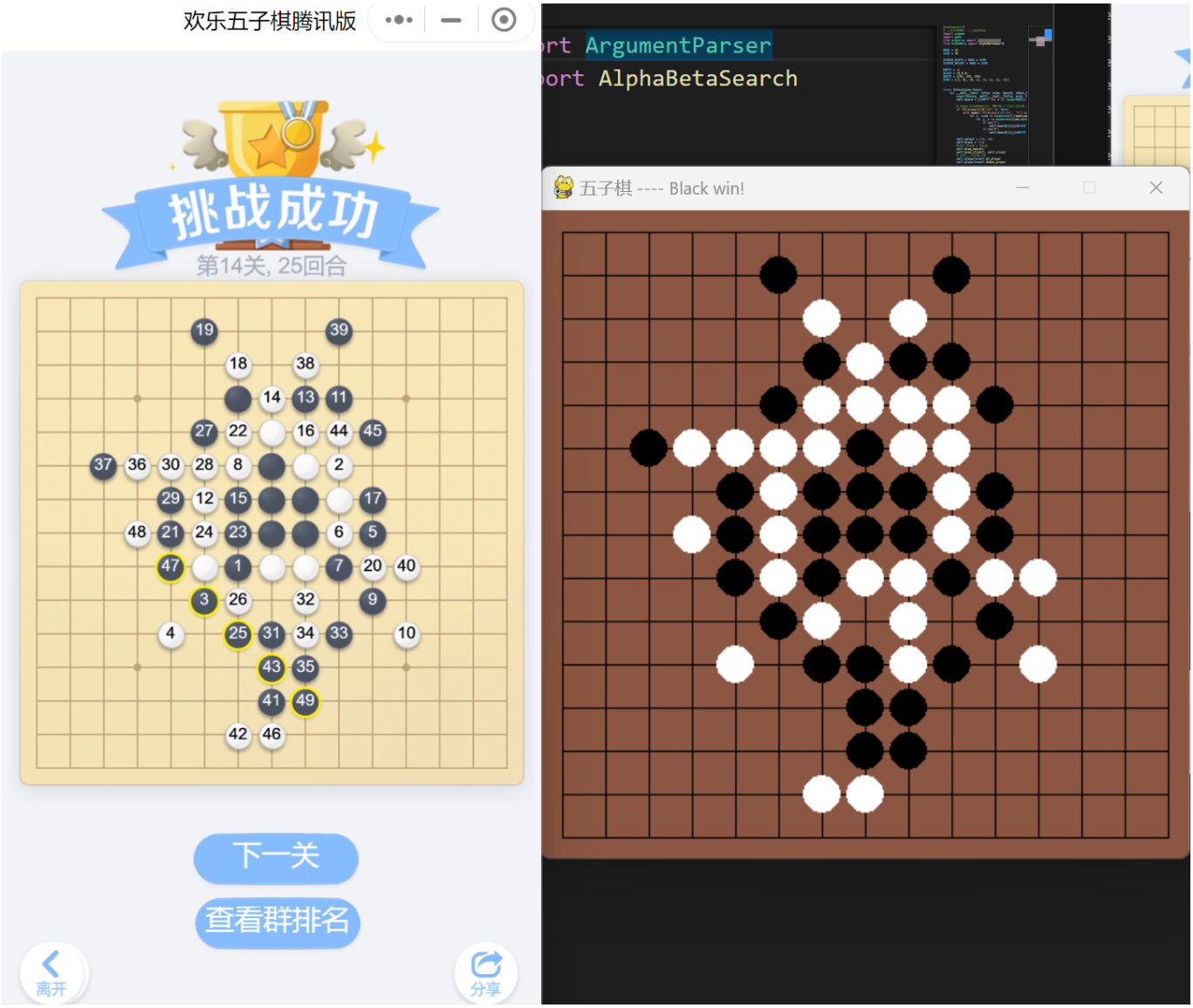
(2)第13关(搜索层数：2)搜索节点:17483

黑子下棋的位置顺序和分数

```
8 8 -1606
3 4 -1596
9 4 6
7 5 6
8 6 14
6 4 20
9 9 16
9 7 10
5 9 4
```

```
7 9 9010
10 6 9040
11 5 100000
```

(3-1)第14关(搜索层数：2) 搜索节点:73286

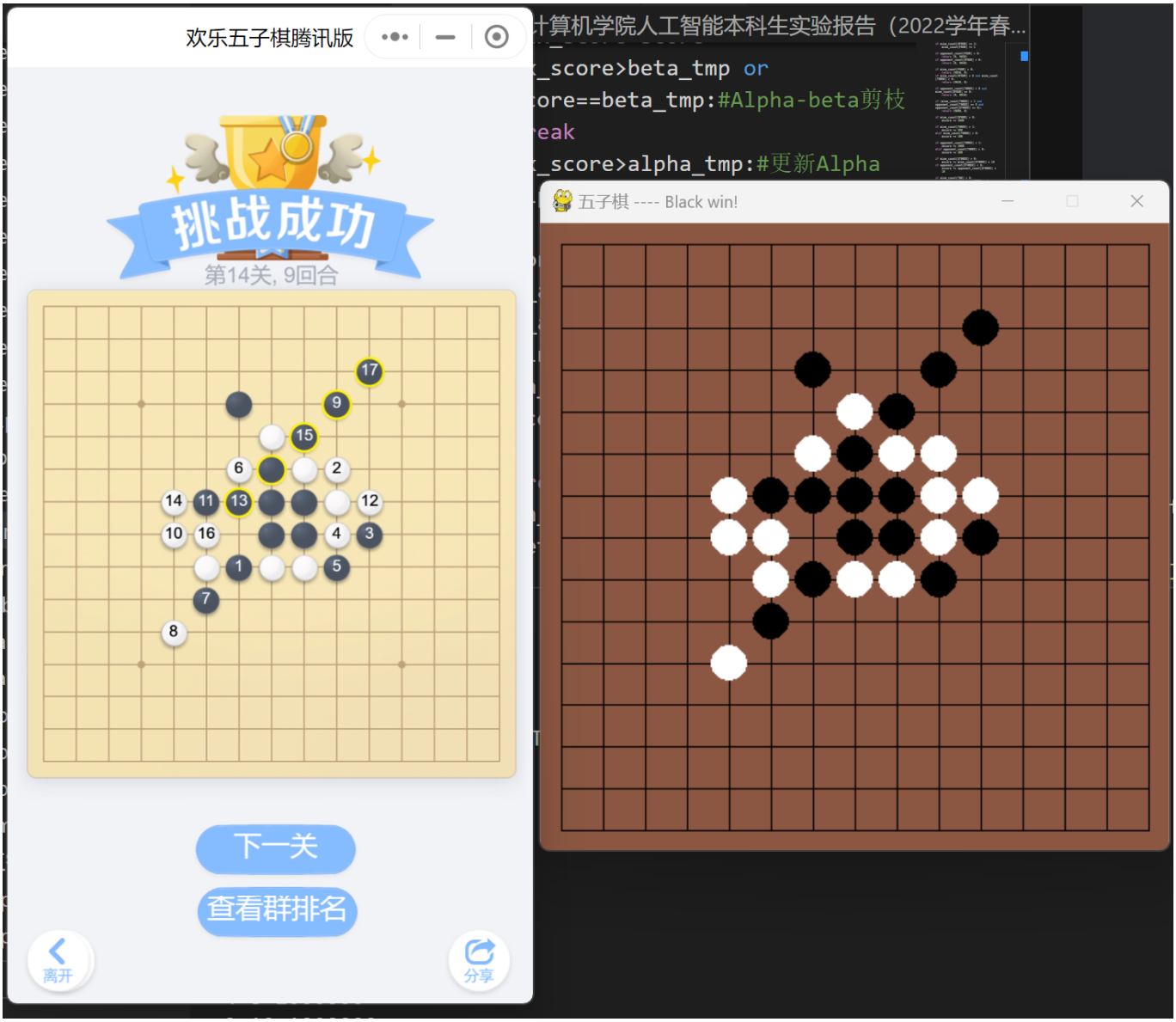


黑子下棋的位置顺序和分数

```
8 6 -1588
9 5 -6
7 10 -96
8 9 -1600
9 10 -2
3 9 -92
3 8 -1608
6 6 -1608
6 10 -1618
1 5 -1608
7 4 -104
```

```
7 6 -104
10 6 -114
4 5 -2004
6 4 -2018
10 7 -1628
10 9 -1624
11 8 -1624
5 2 -1614
1 9 -1604
12 7 -10
11 7 -12
4 10 9010
8 4 9040
12 8 1000000
```

(3-2)第14关(搜索层数：3)搜索节点:821374



黑子下棋的位置顺序和分数

```
8 6 116
7 10 1604
8 9 118
9 5 1998
3 9 122
6 5 2000
6 6 9030
4 8 1000000
2 10 1000000
```

分析：

1.对上述3关结果的分数分析，发现每次的棋盘评分也符号对棋盘棋形数目统计的结果。

2.14关用不同层数搜索，14关用2层搜索需要25步，用3层搜索需要9步，大大优化了下棋的步骤。

2. 评测指标展示及分析

剪枝之前且未优化：

(对搜索两层的进行了剪枝前的测试，搜索三层时间太长了)

关卡	搜索层数	步数	总共搜索节点	平均每步搜索节点	平均每步所花时间
13	2	12	544450	45371	几十秒到几分钟多
14	2	25	1113053	44522	几十秒到几分钟多

剪枝之前且优化后：

关卡	搜索层数	步数	剪枝后总共搜索节点	剪枝后平均每步搜索节点	剪枝后平均每步所花时间	剪掉节点数的占比
11	3	4	188094	47012	几秒到几十秒(能在1min内完成)	剪枝前未跑
13	2	12	17483	1457	零点几秒到一两秒	96.7%
14	2	25	73286	2931	零点几秒到一两秒	93.4%
14	3	9	821374	91264	几秒到几十秒(能在1min内完成)	剪枝前未跑

分析

1.通过实验的结果可以明显看出搜索的层数越大，那搜索的节点会更多，导致搜索的时间更长。

2.但更深层次的搜索，下棋的位置会更优，能解决的残局会更多，还有可以大大优化下的步骤，找出步骤最少的解决方法，如14关用2层搜索需要25步，用3层搜索需要9步，大大优化了下棋的步骤。用更层的搜索，考虑也会更优，从而能解决的关卡越多。

3.同时未剪枝前，搜索三层的每一步时间大概需要十几分钟，这效率太低，在剪枝后只需要几秒到几十秒(能在1min内完成)，大大减少了搜索的节点数目和搜索时间。

4.可以从上述表格看到未剪枝搜索的节点和时间都是巨大的，剪枝后大概能减少搜索90%多的节点，大大减少搜索所需要的时间。

5.同时与其他同学的讨论与比较中得出，使用不一样的估价方法，对下棋的解有一定影响，我的估价函数参考了参考资料第一份资料，像我2层就能解决的，别人可能需要3层，像我的12关有的人需要2或3层，而我却需要4层，这说明寻找到一个适合且效率高的估价函数能大大优化下棋的选择，但往往设计一个好的估计方法还是比较复杂的。

6.同时为了进一步测试我的代码，我在小程序上与其他玩家下棋，发现2层的搜索基本上瞬间得到下棋的位置，也几乎都能赢，但是也输了一两把；用三层搜索，没有在小程序上输过，但是有的步数需要45s以上，导致一步下棋超时，说明要应用在实际中还是需要进一步的优化。

四、参考资料

- Python 五子棋AI实现(2):棋型评估函数实现:https://blog.csdn.net/marble_xu/article/details/90450436?spm=1001.2014.3001.5506
- 基于 alpha-beta 剪枝技术的五子棋 - 知乎