实现6X6的黑白翻转棋的人机对战

（1）算法原理

1. 问题形式化

状态：棋盘当前棋子放置的情况。

搜索策略：minimax-alpha、beta剪枝

评估函数：作为每一步下子的评估值

2. minimax-alpha、beta剪枝算法

采用负极大值方法实现，对每一层的本方，都是求极大值，往上一层的结果就进行取负。对于AI，生成一个虚拟的对手与其进行指定层数的博弈，每次进入minimax函数，参数里有本方分数（alpha）和对手分数（beta)，首先判断当前是否到达指定深度，如果是，则返回当前状态的评估函数值作为预估值，否则进行查找所有可以下子的合法位置，能够没有合法位置，返回当前状态的预估值，令当前最佳的分数等于本方分数，遍历每一个合法下子位置，采用深度优先策略，先下子，然后调用对手的minimax函数，同时将自己的alpha、beta取负交换传递，因为本方采用alpha策略，求最高，但是对方在下一层是beta策略，求最小，因此需要取反求最高，接收返回的分数值，回溯撤销下子操作，将接收的分数取反（对手采用最小策略），然后跟当前最佳分数比较更新，如果最佳分数大于对方分数，说明接下来传回来的分数一直小于最佳分数，进行剪枝，加快搜索效率，最后返回最佳分数给上一层。

3. 评估函数：

A：根据棋子个数差表示的评估函数

求双方棋子之差作为预估。

B：根据位置矩阵权重的评估函数

根据每个位置不同给予不同权重，对于每个位置，本方棋子累加该位置权重，敌方棋子减去该位置权重。

C：综合位置矩阵权重、棋子差、行动力、稳定子的评估函数

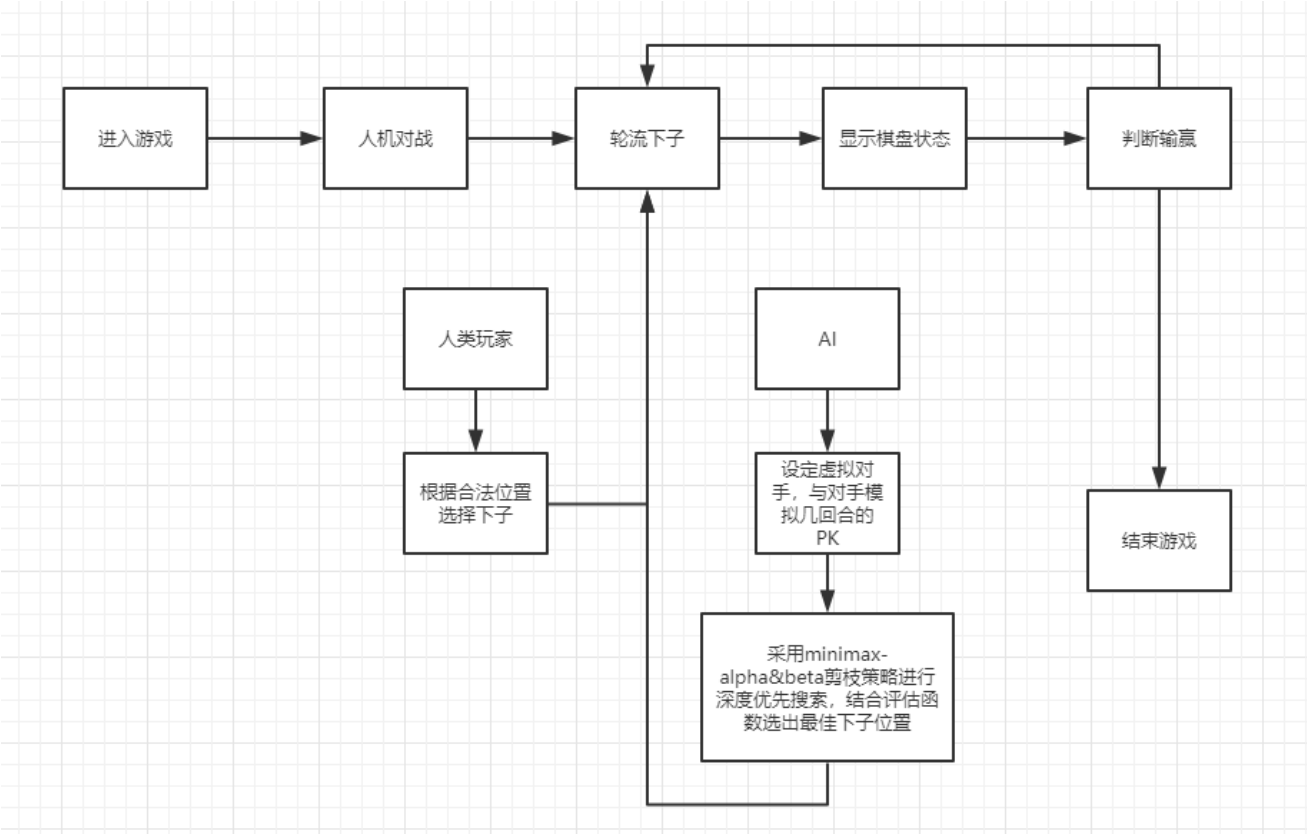
结合A和B，同时加入每次下子所有可选位置的个数作为行动力，任何情况不能被翻的棋子个数为稳定子，进行加权求和作为预估。

D：综合考虑棋子各个位置不同情况（非位置矩阵权重，而是根据不同棋盘状态改变位置权重）、棋子数目差、棋子行动力的评估函数

在位置权重进行改善，棋盘各个位置的权重会随着棋盘状态变化而相应做出变化，分为4种不同类型的位置，产生4个因素，同时加入棋子数目差、棋子行动力2个因素，对6个因素进行加权求和作为预估。

（2） 伪代码与流程图

1. 流程图



2. minimax-alpha、beta剪枝

```
Algorithm minimax-alpha&beta
  input:oppo depth alpha beta
  output:best_score best_nest_step
    #如果深度为0或者无子可下，返回预估值
    if depth == 0 or no valid position to put chess do
        return evaluation
    #令最佳分数等于alpha值
    best_score <- alpha
    #探索所有合法位置
    for every step in valid step do
        put chess #下子
        score,next_step <- oppo.minimax-alpha&beta(self,depth-1,-beta,-alpha)  #递归调用下一层探索
        cancel put chess #撤销下子

        score <- -score #策略相反，分数取反
        #更新最佳分数
        if score > best_score do
            best_score <- score
            best_nest_step <- step
        #如果大于beta，说明没有继续探索的意义，剪枝
        if best_score > beta do
            break
    return best_score,best_next_step
```

(3) 关键代码

1. 游戏棋盘设计：将游戏棋盘设计类，在类的基础上设计黑白棋游戏过程中用到的各种函数。

- 初始化棋盘，用二维列表存储棋盘，‘.'表示空格，'X'表示黑棋，表示先下子的一方，'O'表示白琪，表示后下子的一方，提前下好4步作为初始值。

```
    #初始化棋盘
    def __init__(self):
        self.size = 6
        self.space = '.'  #棋盘为空的标记
        self.board = []
        for i in range(self.size):
            tmp = [self.space] * self.size
            self.board.append(tmp)
        self.board[2][2] = 'O'
        self.board[3][3] = 'O'
        self.board[2][3] = 'X'
        self.board[3][2] = 'X'
```

- 判断是否已经结束，如果双方没有能够下子的合法位置，说明游戏结束。

```
    #判断是否已经结束
    def is_over(self):
        n1 = self.find_right_flipping_position('X')
        n2 = self.find_right_flipping_position('O')
        #如果双方都没有可以走下一步的位置，则说明游戏结束
        if len(n1) == 0 and len(n2) == 0:
            return True
        return False
```

- 游戏结束后，通过计算当前黑白棋的数目，得出最终的胜利者或者平局。

```
    #得出胜利者
    def winner(self):
        x_count = 0
        o_count = 0
        #计算白琪和黑棋的个数
        for i in range(self.size):
            for j in range(self.size):
                #黑棋个数+1
                if self.board[i][j] == 'X':
                    x_count += 1
                #白琪个数+1
                elif self.board[i][j] == 'O':
                    o_count += 1
        #黑棋胜利
        if x_count > o_count:
```

```
            return 0
        #平局
        elif x_count == o_count:
            return 1
        #白棋胜利
        else:
            return 2
```

- 打印棋盘状态，显示当前的博弈情况，同时显示下一方能够下子的位置。

```
    #打印棋盘状态
    def print_state_ts(self,mark):
        board = []
        #得出当前可以下子的合法位置
        valid_position = self.find_right_flipping_position(mark)
        #复制棋盘，防止后面操作改变原来的棋盘
        for i in range(self.size):
            tmp = []
            for j in range(self.size):
                tmp.append(self.board[i][j])
            board.append(tmp)
        #将合法位置改为'+'
        for i in range(len(valid_position)):
            x = valid_position[i][0]
            y = valid_position[i][1]
            board[x][y] = '+'
        #打印棋盘
        print(' ', ' '.join(list('123456')))
        for i in range(self.size):
            print(i + 1, ' '.join(board[i]))
```

- 游戏玩家下子，将选择的位置标记为玩家的标记，同时调用翻子函数，将能够翻子的位置全部进行翻子。

```
    #放棋子
    def move(self,mark,next_step):
        self.board[next_step[0]][next_step[1]] = mark #下子
        flipping_pieces_list = self.flipping_pieces(mark,next_step)  #将所有可以翻的位置都进行翻子
        return flipping_pieces_list #返回翻子的二维坐标的列表，方便后面回溯恢复
```

- 翻子函数，根据当前的位置，对八个方向进行遍历，如果遇到空格或者边界，说明不能翻子，结束该方向的探索，如果遇到自己同色的棋子，则将探索过的部分加入到翻子的列表中，遍历完8个方向后，统一对翻子列表中的位置全部进行翻子。

```
    #翻子
```

```
    def flipping_pieces(self,mark,next_step):

        flipping_list = []
        #检查当前位置的各方向
        for direction_line in self.direction_lines(next_step):
            for i,j in enumerate(direction_line):
                #找到相同颜色的，将前面探索的位置放进队列
                if self.board[j[0]][j[1]] == mark:
                    flipping_list.extend(direction_line[:i])
                    break
                #遇到空位，结束探索
                elif self.board[j[0]][j[1]] == self.space:
                    break
        #对所有可以翻子的位置进行翻子
        for i in range(len(flipping_list)):
            self.board[flipping_list[i][0]][flipping_list[i][1]] = mark
        return flipping_list
```

- 由于使用深度优先搜索，因此探索到指定层数后，就要返回，返回就要将之前的下子和翻子进行恢复，因此将之前下子的位置恢复为空格，利用之前记录的翻子位置列表一一恢复为对方的棋子。

```
    #回溯，撤销放棋子
    def un_move(self,mark,next_step,flipping_list):
        if mark == 'X':
            anti_mark = 'O'
        else:
            anti_mark = 'X'
        #将下子的位置恢复原状
        self.board[next_step[0]][next_step[1]] = self.space
        #将所有翻子的位置恢复原状
        for i in range(len(flipping_list)):
            self.board[flipping_list[i][0]][flipping_list[i][1]] = anti_mark
```

- 由于每次下子都要找到所有下子的位置，因此需要一个函数计算所有合法的放棋子位置。遍历棋盘，判断当前的棋子如果是对方颜色的，则进行8个方向的一步探索，检查是否越界、是否为空格、是否已经算入合法放棋子列表，然后再检查是否能够引起翻子，如果都可以，则放入合法放棋子列表。

```
    #找到所有合法的放棋子位置
    def find_right_flipping_position(self,mark):
        if mark == 'X':
            anti_mark = 'O'
        else:
            anti_mark = 'X'
        #8个方向
        move_direct = [(1,0),(1,1),(0,1),(-1,1),(-1,0),(-1,-1),(0,-1),(1,-1)]
        valid_flipping_position = []
        for i in range(self.size):
            for j in range(self.size):
```

```
                #找到相同颜色的子，进行下一步探索
                if self.board[i][j] == anti_mark:
                    for move in move_direct:
                        x = i + move[0]
                        y = j + move[1]
                        #检查合法性
                        if x < self.size and x >= 0 and y < self.size and y >= 0 and
self.board[x][y] == self.space and (x,y) not in valid_flipping_position:
                            next_step = (x,y)
                            #检查是否能够翻子，可以说明合法
                            if self.is_fipping(mark,next_step) == True:
                                valid_flipping_position.append((x,y))

        return valid_flipping_position
```

- 给当前位置计算8个方向的探索数组，便于在翻子函数中使用，由于代码有所重复，因此只列举部分方向创建。

```
        #左右方向
        left = board_array[x][0:y]
        right = board_array[x][y+1:]
        #上方向
        top = []
        for i in range(x):
            top.append(board_array[i][y])
       #下方向
        bottom = []
        for i in range(x+1,self.size,1):
            bottom.append(board_array[i][y])
        #斜方向
        x1 = x-1
        y1 = y+1
        right_top = []
        while x1 >= 0 and y1 < self.size:
            right_top.append(board_array[x1][y1])
            x1 = x1-1
            y1 = y1+1
```

2. 人类玩家：首先检查能够下子的合法位置，如果位置不存在，本轮下子跳过，输入下子的位置，检查位置是否越界，同时检测下子是否在合法位置内，在的话返回下子的位置，进行下一步行动。

```
    #决策思考下一步
    def decision_making(self):
        #得到合法的所有位置
        valid_flipping_position = self.gameboard.find_right_flipping_position(self.mark)
        #如果没有找到合法的位置，返回None
        if len(valid_flipping_position) == 0:
```

```
            return None
        while True:
            #提示输入格式
            input1 = input("please input the next step: format: row col.")
            input2 = input1.split()
            row = int(input2[0]) - 1
            col = int(input2[1]) - 1
            #检查是否在边界内
            if row >= 0 and row < self.size and col >= 0 and col < self.size:
                valid_flipping_position = self.gameboard.find_right_flipping_position(self.mark)
                #检查输入的下标是否在合法的位置内
                if (row,col) in valid_flipping_position:
                    return (row,col)
```

3. AI玩家：AI玩家进行决策，生成一个虚拟对手与其进行博弈，调用minimax搜索算法，预先进行若干轮比拼，返回评估函数值最高的下子位置，选择该下子位置作为下一步的下子。

```
    #决策
    def decision_making(self):
        print("It turns to AI player.")
        #生成一个虚拟的对手，进行训练
        opponent = AI(self.anti_mark,self.gameboard)
        # best_score,next_step = self.minimax(opponent,4)
        #调用极大极小算法
        best_score,next_step = self.minimax_alpha_beta(opponent,8,-1*float('inf'),float('inf'))
        #返回最佳的下子步骤
        return next_step
```

4. **minimax搜索策略**：采用博弈树搜索策略，对于AI玩家，设置一个虚拟玩家与其进行博弈，进行指定层数的回合比拼，每个回合选取评估函数值最优的步骤进行下子，直到最终找到最优结果，作为当前下子的依据。进入函数，首先判断是否达到指定深度，如果是，返回当前的评估函数值，寻找所有可以下子的合法位置，如果没有合法下子的位置，返回当前评估函数值最高的，设置最佳评估值为-9999，用于选出各个最优的子节点路径，初始化最佳下子位置，遍历所有合法下子位置，首先对选出的位置进行下子，并用变量flipping_pieces_list记录所有翻子的位置，递归调用对手的minimax函数，等到递归调用结束，接收返回值，并且将下子恢复，将之前翻子的位置全部复原，由于是minimax博弈，当前是找最高的best_score，那么到了下一层对对手而言，是应该找最低的，因此需要将返回的score取负，判断当前分数是否大于最佳分数，如果大于，则更新最佳分数和最佳下子步骤，最后返回最佳分数和最佳下子步骤。

```
    #极大极小值算法
    def minimax(self,opponent,depth):
        #如果深度为0，返回当前状态的评估函数值
        if (depth == 0):
            score = self.evaluation_pos_weight()
            return score,None
        #找出所有可以下子的合法位置
        valid_flipping_position = self.gameboard.find_right_flipping_position(self.mark)
```

```
        #如果不存在可以下子的合法位置，则返回当前状态的评估函数值
        if (len(valid_flipping_position) == 0) :
            score = self.evaluation_pos_weight()
            return score, None

        best_score = -99999  #最佳评估值
        best_next_step = None  #最佳下子位置
        #遍历所有合法下子位置
        for step in valid_flipping_position:
            #下子，并返回所有翻子的位置
            flipping_pieces_list = self.gameboard.move(self.mark,step)
            #递归调用该算法
            score,next_step = opponent.minimax(self,depth-1)
            #回溯，撤销下子，并且将翻子的所有位置恢复
            self.gameboard.un_move(self.mark,step,flipping_pieces_list)
            #由于极大极小值算法相邻层追求最优相反，因此从下层返回的score需要取反，让同一层取得最优
            score = (-1) * score
            #如果当前分数大于最佳分数，更新最佳分数和最佳下子步骤
            if score > best_score:
                best_score = score
                best_next_step = step
        #返回最佳分数、最佳下子步骤
        return best_score,best_next_step
```

5. **minimax+alpha、beta剪枝搜索策略**

在minimax基础上加入剪枝，参数中加入my_score（代表本方分数）,opponent_score（代表对方分数），令best_score = my_score，作为当前最佳分数，遍历所有合法下子位置，对选出合法下子位置下子，并且递归调用对手的minimax_alpha_beta函数，同时由于到下一层，此时该函数本方分数为现在的对手分数，对手分数为现在的本方分数，因此两个参数需要调转，同时因为本函数是求最高对best_score，因此到下一层是求最低的分数，因此需要对my_score,opponent_score进行取负，接收递归返回结果，撤销下子步骤，同时对分数取负，根据分数大小决定是否更新best_score、best_next_step，同时判断此时best_score > opponent_score是否成立，如果成立，则进行剪枝，不用继续探索剩余的分支，返回结果给上一层。

```
    #极大极小值，alpha、beta剪枝
    def minimax_alpha_beta(self,opponent,depth,my_score,opponent_score):
        #如果深度为0，返回当前状态评估函数的值
        if (depth == 0):
            # score = self.evaluation_pos_weight()
            score = self.evaluation_Comprehensive()
            return score,None
        #求出当前状态的所有的合法位置
        valid_flipping_position = self.gameboard.find_right_flipping_position(self.mark)
        #如果没有可以下子的合法位置，返回当前状态的评估函数的值
        if (len(valid_flipping_position) == 0) :
            # score = self.evaluation_pos_weight()
            score = self.evaluation_Comprehensive()
            return score, None
        #令当前最好的分数值等于本方的分数值
        best_score = my_score
        #初始化最好下一步下子步骤
```

```
        best_next_step = None
        #对于所有合法位置进行遍历
        for step in valid_flipping_position:
            #下子，并且返回所有翻子的下标
            flipping_pieces_list = self.gameboard.move(self.mark,step)
            #递归调用下一层，由于是极大极小值算法，因此是对方opponent对象调用该函数，同时
opponent_score、my_score取负号交换传参
            score,next_step = opponent.minimax_alpha_beta(self,depth-
1,-1*opponent_score,-1*my_score)
            #回溯恢复到下子之前的步骤
            self.gameboard.un_move(self.mark,step,flipping_pieces_list)

            #由于下一层的策略是相反的，因此需要去负号
            score = (-1) * score
            #如果分数大于当前最好的分数
            if score > best_score:
                best_score = score #更新最好的分数值
                best_next_step = step #更新最佳下一步的下子步骤

            #如果alpha>beta，进行剪枝
            if best_score > opponent_score:
                break
        #返回最佳分数、最佳下子步骤
        return best_score,best_next_step
```

6. **评估函数**

A：根据棋子个数差表示的评估函数

```
    #根据棋子个数差表示的评估函数
    def evaluation_score(self):
        score1 = 0
        score2 = 0
        for i in range(self.size):
            for j in range(self.size):
                #如果是对手棋子，对方分数加1
                if self.gameboard.board[i][j] == self.anti_mark:
                    score2 += 1
                #如果是对手棋子，自己分数加1
                elif self.gameboard.board[i][j] == self.mark:
                    score1 += 1
        #用自己分数减去对方分数的结果作为评估函数的分数
        return score1 - score2
```

B：根据位置矩阵权重的评估函数

```
    #根据位置权重的评估函数

    def evaluation_pos_weight(self):
```

```
        #统计分数
        score = 0
        for i in range(self.size):
            for j in range(self.size):
                #如果是对方棋子，减去当前位置的权重
                if self.gameboard.board[i][j] == self.anti_mark:
                    score -= self.pos_weight[i][j]
                #如果是本方棋子，加上当前位置的权重
                elif self.gameboard.board[i][j] == self.mark:
                    score += self.pos_weight[i][j]
        #以最终的权重作为评估函数的结果
        return score
```

C：综合位置矩阵权重、棋子差、行动力、稳定子的评估函数

- 计算位置权重的分数，上面已经附上，不再重复。
- 计算棋子差，上面已经附上，不再重复。
- 行动力计算，计算每次下子所有合法下子的个数作为行动力。

```
    valid_position = self.gameboard.find_right_flipping_position(self.mark)
        action_score = len(valid_position)
```

- 计算稳定子。

```
        my_stabilizer = 0
        opp_stabilizer = 0
        if self.gameboard.board[0][0] == self.mark:
            my_stabilizer += 1
        elif self.gameboard.board[0][0] == self.anti_mark:
            opp_stabilizer += 1
        if self.gameboard.board[0][self.size-1] == self.mark:
            my_stabilizer += 1
        elif self.gameboard.board[0][self.size-1] == self.anti_mark:
            opp_stabilizer += 1
        if self.gameboard.board[self.size-1][0] == self.mark:
            my_stabilizer += 1
        elif self.gameboard.board[self.size-1][0] == self.anti_mark:
            opp_stabilizer += 1
        if self.gameboard.board[self.size-1][self.size-1] == self.mark:
            my_stabilizer += 1
        elif self.gameboard.board[self.size-1][self.size-1] == self.anti_mark:
            opp_stabilizer += 1
```

- 消除内四角的潜在危害。

```
        if self.gameboard.board[1][1] == self.mark:
            total_score -= 32
```

```
        elif self.gameboard.board[1][1] == self.anti_mark:
            total_score += 32
        if self.gameboard.board[1][4] == self.mark:
            total_score -= 32
        elif self.gameboard.board[1][4] == self.anti_mark:
            total_score += 32
        if self.gameboard.board[4][1] == self.mark:
            total_score -= 32
        elif self.gameboard.board[4][1] == self.anti_mark:
            total_score += 32
        if self.gameboard.board[4][4] == self.mark:
            total_score -= 32
        elif self.gameboard.board[4][4] == self.anti_mark:
            total_score += 32
```

- 根据各个因素，按照给定权重进行加权求和得到最终分数。

```
weights = [10,10,10,100]
        total_score = pos_score * weights[0] + num_score * weights[1] + action_score *
weights[2] + (my_stabilizer - opp_stabilizer) * weights[3]
```

D：综合考虑棋子各个位置不同情况（非位置矩阵权重，而是根据不同棋盘状态改变位置权重）、棋子数目差、棋子行动力的评估函数

- 遍历整个棋盘，如果当前位置为空格，则跳过，否则判断棋子归属，如果是本方棋子，amount为1，否则为-1，每遇到一个棋子，更新chess_num，chess_num表示本方棋子与对方棋子之差。amount是为了下面更新角和边的数量的单位，如果是本方棋子，应该是加1，否则是减1。

```
#综合考虑棋子各个位置不同情况、棋子数目差、棋子行动力
    def evaluation_best(self):
        chess_num = 0
        out_corner,out_edge,inner_corner,inner_edge = 0,0,0,0
        for i in range(self.size):
            for j in range(self.size):
                if self.gameboard.board[i][j] == '.':
                    continue
                amount = 1 if self.gameboard.board[i][j] == self.mark else -1
                chess_num += amount
```

- 计算最外层的边和角，如果是四个角，更新out_corner，如果是边，更新out_edge。

```
    #计算最外层边的情况，四个角和四条边的情况
                if i == 0 or j == 0 or i == self.size-1 or j == self.size-1:
                    if (i == 0 and j == 0) or (i == 0 and j == self.size-1) or (i == self.size-1
and j == 0) or (i == self.size-1 and j == self.size-1):
                        out_corner += amount
                    else:
                        out_edge += amount
```

- 计算从外往里第二层的边的情况。对于第二层的边，判断其相邻的最外层的三个格子是否为空格，如果为空格，证明有危险，下次有可能被对方下子翻掉，有多少空格就给inner_edge加上多少个amount。

```
        #计算上下从外往内第二层的边
        elif i == 1 or i == self.size-2 and (j > 1 and j < self.size-2):
            x = self.size-1 if i == self.size-2 else 0
            for k in range(j-1,j+2):
                if self.gameboard.board[x][k] == '.':
                    inner_edge += amount


        #计算左右从外往内第二层的边
        elif j == 1 or j == self.size-2 and (i > 1 and i < self.size-2):
             y = self.size-1 if j == self.size-2 else 0
             for k in range(i-1,i+2):
                 if self.gameboard.board[k][y] == '.':
                     inner_edge += amount
```

- 计算从外往里第二层四个角的情况。以内层左上角为例，首先判断其对应的最外角是否为空格，如果为空格，则在这个位置是很有可能被别人占角，因此给inner_corner加上amount，然后分别对该位置的左边两个相邻格子和上面两个格子判断是否为空格，如果是，则将其inner_edge加上amount。

```
                #内层左上角
                elif j == 1 and i == 1:
                    if self.gameboard.board[0][0] == '.':
                        inner_corner += amount
                    for k in range(1,3):
                        if self.gameboard.board[k][0] == '.':
                            inner_edge += amount
                    for k in range(1,3):
                        if self.gameboard.board[0][k] == '.':
                            inner_edge += amount
```

- 计算行动力，计算当前棋子所有合法下子位置的个数作为下一步的行动力。

```
        #行动力
        valid_position = self.gameboard.find_right_flipping_position(self.mark)
        action_num = len(valid_position)
```

- 根据各个因素，赋予权值，相加得出最终的结果，由于inner_edge、inner_corner为负面因素，因此需要给予负数的权值，经过多次训练，得出较优化的权值参数为：15,6,-10,-4,8,8。对应的是：角、边、内层角、内层边、棋子差、行动力。

```
        #最终分数
        final_score = out_corner*self.ele_weights[0] + out_edge*self.ele_weights[1] +
inner_corner*self.ele_weights[2]
        + inner_edge*self.ele_weights[3] + chess_num*self.ele_weights[4] +
action_num*self.ele_weights[5]
```

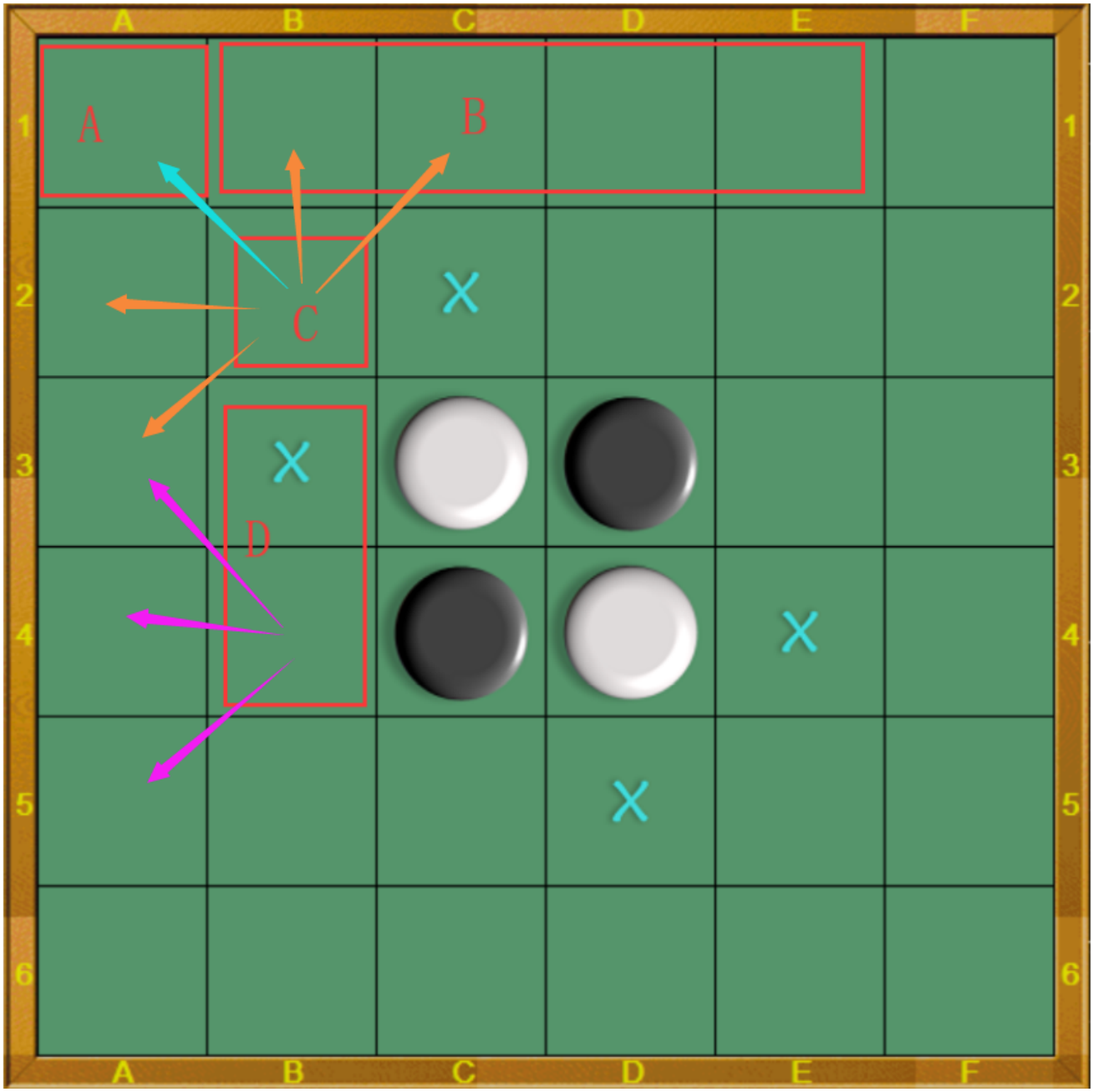
（4）创新点&优化

1. 棋盘位置权重矩阵

   根据棋盘各个位置分析，下子时不同的位置会起到不同的效果，因此需要给每个位置赋予不同的权值，其中四个角是最应该占领的，因此权值最大，而角相邻的两边的位置则是不好的位置，占领则容易让对方占领角，失去先机，因此权重为负数，角相对的内一层的角位置则是最差的位置，同时更容易让对方占领角，因此权重为负且最小，其余在边的位置是应该要占领的，因此权重仅次于角的权重，从外到内第二层的边容易让对手占边，因此权重相应调小，根据权重矩阵能够使得下子更加科学合理。

2. 棋盘位置权重随局势进行变化

   由于上面棋盘位置权重矩阵的各个值是固定的，在下子过程中，状态不同，因此相同的位置不一定就好，也不一定就差，要根据实际情况而定，因此基于不同位置对不同情况判断进行优化。



如图，棋盘分成4个部分，对于A部分，即角位置，计算本方棋子与敌方棋子的差作为out_corner的值；B则是外层四条边，对于每个格子，如果是本方棋子，加1，否则减1，作为out_edge的值；C则是从外到内第二层的角位置，此时要根据相邻向外的值确定该位置的情况好坏，如果青色箭头指向的角位置为空，则随时很可能被敌方占领，因此inner_corner加1，同时黄色箭头指向的格子中，有x个格子为空，代表有可能这x个格子会被敌方占领，这是不利的，因此需要让inner_edge加1；对于D区域，是从外到内第二层的边，以下面那个格子为例，粉色箭头表示与格子相邻的外层格子，同理，x个格子为空，有可能被占，因此让inner_edge加1；

最后对这四个因素根据重要性各自加上权重，其中out_corner、out_edge为正面因素，权重为正，inner_corner、inner_edge为负面因素，权重为负，经过训练，得出最合适的权重参数，其中corner权重的绝对值相对较大，说明影响较大。

3. 稳定子

稳定子是无论对方怎样下子都不会让其翻子的位置，四个角当然是稳定子，同时在下子过程中，有些位置会成为稳定子，对局势影响比较大，因此通过计算稳定子可以进一步优化评估函数，但是实际下棋过程中，除了四个角，其他位置出现稳定子几率很小，因此每一步计算会增加计算量，从而减慢运行效率，因此在25步后加入稳定子判断是比较合理。

4. 预定结局

博弈到了后期比较接近结局时，如果搜索层数较深，则会提前遇到结果，如果搜索结果有利于本方，则直接返回最优值，从而加快本方下子效率，提高本方胜利的概率。

实验结果及分析

（1）实验结果展示

- 展示人机对战连续5个回合。'X'为人类，'O'为AI。

  score为评估函数的值，根据各个因素进行加权求和，因此分数相对较高。

1. 人类玩家下子（2,2），此时AI下子（1,4），成功吃掉3个黑子，分数为1308，由于分数是由评估函数根据加权求和求出，因此数值会有不同，但相对是同一标准。

```
please input the next step: format: row col.2 2
move step:  (2, 2)
X: 11  O: 16  Empty: 9
  1 2 3 4 5 6
1 . + X + O O
2 + X X X O O
3 O X X O O O
4 O X O X X O
5 O O X X + O
6 O + + + + O
It turns to AI player.
move step:  (1, 4)
X: 8  O: 20  Empty: 8
score:  1308
  1 2 3 4 5 6
1 . + X O O O
2 + X O O O O
3 O O X O O O
4 O X O X X O
5 O O X X . O
6 O + . . . O
```

2. 人类玩家下子（6,2），此时吃掉（5,1）的白字，但是目前仍处于劣势，AI继续下子，将优势继续扩大到5:25。

```
please input the next step: format: row col.6 2
move step:  (6, 2)
X: 10  O: 19  Empty: 7
  1 2 3 4 5 6
```

```
1 . + X O O O
2 + X O O O O
3 O O X O O O
4 O X O X X O
5 O X X X + O
6 O X + + + O
It turns to AI player.
move step:  (6, 3)
X: 5  O: 25  Empty: 6
score:  1538
  1 2 3 4 5 6
1 . . X O O O
2 . X O O O O
3 O O X O O O
4 O X O X O O
5 O O O O . O
6 O O O + . O
```

3. 人类下子（6,4），将黑子数量扩大为8，由于当前步骤并没有明显的优势，因此仍然存在很大劣势，AI下子，改写比分为6:26。

```
please input the next step: format: row col.6 4
move step:  (6, 4)
X: 8  O: 23  Empty: 5
  1 2 3 4 5 6
1 . + X O O O
2 + X O O O O
3 O O X O O O
4 O X O X O O
5 O O X X + O
6 O O O X + O
It turns to AI player.
move step:  (6, 5)
X: 6  O: 26  Empty: 4
score:  1578
  1 2 3 4 5 6
1 . . X O O O
2 . X O O O O
3 O O X O O O
4 O X O X O O
5 O O X O + O
6 O O O O O O
```

4. 此时人类只有一个选择，只能下（5,5），AI抓住机会，往最有优势的角位置下子（1,1），比分来到4:30。

```
please input the next step: format: row col.5 5
move step:  (5, 5)
X: 8  O: 25  Empty: 3
  1 2 3 4 5 6
1 + + X O O O

2 + X O O O O
```

```
3 O O X O O O
4 O X O X O O
5 O O X X X O
6 O O O O O O
It turns to AI player.
move step:  (1, 1)
X: 4  O: 30  Empty: 2
score:  1582
  1 2 3 4 5 6
1 O + X O O O
2 + O O O O O
3 O O O O O O
4 O X O O O O
5 O O X X O O
6 O O O O O O
```

5. 此时四个角以被对方占领，并且大多数棋子属于对方，以无力回天，选择(1,2)，AI走完剩下的一步，游戏结束，最终以31:5战胜人类。

```
please input the next step: format: row col.1 2
move step:  (1, 2)
X: 7  O: 28  Empty: 1
  1 2 3 4 5 6
1 O X X O O O
2 + X O O O O
3 O X O O O O
4 O X O O O O
5 O O X X O O
6 O O O O O O
It turns to AI player.
move step:  (2, 1)
X: 5  O: 31  Empty: 0
score:  1572
  1 2 3 4 5 6
1 O X X O O O
2 O O O O O O
3 O O O O O O
4 O X O O O O
5 O O X X O O
6 O O O O O O
Game Over!!!
Player2 Win!!!
X: 5  O: 31  Empty: 0
```

（2）评测指标展示

为了更好地展示实验指标，让采用不同的标准的AI互相对战，得出实验比较。

先手结果是指表格中第一种策略先下，第二种策略后下的最终结束游戏时的棋子数目比；后手结果是指表格中第一种策略后下，第二种策略先下的最终结果。

- 各个评估函数的比较

  搜索策略统一采用minimax-alpha_beta剪枝

A：根据棋子个数差表示的评估函数

B：根据位置矩阵权重的评估函数

C：综合位置矩阵权重、棋子差、行动力、稳定子的评估函数

D：综合考虑棋子各个位置不同情况（非位置矩阵权重，而是根据不同棋盘状态改变位置权重）、棋子数目差、棋子行动力的评估函数

（先手和后手都是相对于评估函数1而言）

评估函数1	评估函数2	先手结果	后手结果
D	A	36 : 0	29 : 7
D	B	24 : 12	32 : 3
D	C	16 : 20	22 : 14
C	A	25 : 11	23 : 13
C	B	22 : 14	20 : 16
A	B	8 : 28	23 : 13

由表可见，评估函数D、C相对于A、B考虑比较全面，因此在博弈中无论是先手还是后手都占优势，C和D各有各自的优点，因此博弈中都是后手占优，先手失败，同理A和B也是如此，同时说明通过这样的对战并不能保证哪个策略一定是最优，不具有传递性，需要根据特定的对手而定，当然，通过这样的比较有利于进一步优化评估函数。

- 搜索策略的比较

策略A	策略B	先手结果	后手结果
minimax算法	随机选取	32 : 4	33 : 3

采用特定的搜索算法比随机选取更有效，minimax算法完胜随机选取策略，同时加上alpha-beta剪枝能够提高搜索效率，有利于加快搜索更深层的效率。

- 搜索层数的比较

  先手和后手是针对改变搜索层数AI来说，对手是指定为搜索层数为4的AI，两者评估函数一致。

层数	先手结果	后手结果
1	2 : 34	0 : 32
2	14 : 22	3 : 32
3	15 : 21	8 : 28
4	25 : 11	11 : 25
5	28 : 8	24 : 12
6	32 : 3	22 : 14
7	32 : 4	34 : 2
8	32 : 0	34 : 2

从此可见，搜索层数越低，对棋盘形势估计能力越差，难以得出有效的下一步，因此会完败，随着层数加深，当搜索层数超过对手时，此时对形势估计考虑得更加长久深入，因此往往更容易取得胜利，当层数为8时，可以完胜对手。

- 以评估函数D为例，展示部分各个参数进行调参的结果

  D：综合考虑棋子各个位置不同情况（非位置矩阵权重，而是根据不同棋盘状态改变位置权重）、棋子数目差、棋子行动力的评估函数

  参数顺序：out_corner、out_edge、inner_corner、inner_edge、双方棋子数目差、行动力。

先手参数比例	后手参数比例	先手结果	后手结果
10 6 6 -4 3 3	10 5 -5 -4 3 3	30 : 6	22 : 14
10 6 6 -4 3 3	10 5 -8 -4 4 4	27 : 9	14 : 22
10 6 6 -4 3 3	15 6 -8 -4 4 4	3: 30	14 : 22
15 6 -8 -4 4 4	15 6 -8 -4 5 5	22 : 14	14 : 22
15 6 -8 -4 4 4	18 6 -8 -4 4 4	22 : 14	11 : 25
15 6 -8 -4 4 4	15 10 -8 -4 4 4	22 : 14	33 : 3
15 6 -8 -4 4 4	15 6 -12 -4 4 4	22 : 14	11 : 25
15 6 -8 -4 4 4	15 6 -10 -4 4 4	22 : 14	11 : 25
18 6 -8 -4 4 4	10 5 -8 -4 4 4	10 : 26	14 : 22
15 6 -10 -4 4 4	18 6 -8 -4 4 4	25 : 11	11 : 25
15 6 -10 -4 4 4	15 6 -10 -4 6 6	25 : 11	11 : 25
15 6 -10 -4 4 4	15 6 -10 -4 8 8	25 : 11	11 : 25
15 6 -10 -4 8 8	10 5 -18 -4 8 8	23 : 13	14 : 22

根据调参结果，可以发现out_edge、out_corner、双方棋子数目差、行动力4个是正面因素，因此需要正权重，并且out_corner重要性最强，因此需要调大，调到15-18为较理想状态，out_edge虽然为正面因素，但是在某种情况下反而会有负面影响，因此权重不宜过大，5-6最佳，双方棋子数目差、行动力在6-8范围效果较好，inner_edge虽然为负面因素，但也有可能在某种情况下成为正面因素，因此负权重为-4较合适，inner_corner影响比较大，很有可能导致被占角，因此需要占较大的负权重，-10至-12较为合适。

总结

1. 结合minimax-alpha&beta搜索策略，设置搜索深度越深，能够对局势作出一个更长远的判断，更有利于下一步的决策。
2. 不同的评估函数对下子的影响非常大，同时评估函数好坏是相对的，不同的AI对战，其优胜结果不一定具有传递性，之前得出较好的策略在另一个对手不一定就完全适用，有可能起到反效果，与当前项目AI两两互战的局限性有关，不过在大方向上通过这样的方式，可以进一步优化自己的评估函数。
3. minimax搜索策略在每一层中，本方都是基于自己最优策略做出判断，而采用负极大值算法能够简化minimax算法的代码实现，将两者合并，有利于更好书写递归，同时达到理想的搜索效果。