黑白棋搜索策略与强化学习优化

实现

- **手动实现蒙特卡洛树搜索算法，选择两种不同策略，一是模拟训练到终止，二是模拟训练到指定层数，以评估函数作为reward反馈，尝试多次迭代模拟，优化AI性能。**
- **手动实现遗传算法，设置不同的种群大小、迭代次数进行多次训练，得出最优化的评估函数权重参数。**
- **优化评估函数，选择不同的评估策略进行比较，加入多种考虑因素：棋子差、行动力、棋盘权重、棋盘边界位置动态调整权重、稳定子。**
- **尝试用tensorflow训练蒙特卡洛树，将其与之前的优化版本进行比对，观察是否有进一步优化。**
- **尝试使用多线程对minimax搜索进行优化，缩短搜索时间。（实际效果不好，与python多线程版本有关）**
- **根据搜索深度复杂度，动态选择MCTS搜索的层数，中间搜索层数适当减少，后面增加模拟迭代次数。**

（1）算法原理

A. 蒙特卡洛树搜索策略（MCTS）

1. 算法引入：
    - minimax可以在指定层数条件下得到最优下子步骤，然而搜索层数有限制，当层数增加时，minimax需要扩展更大的游戏博弈树，双方博弈到游戏中段时，其树的分支非常巨大，为了优化，可以使用alpha-beta剪枝进行优化，减少搜索空间，但是最好的效果也不会超过minimax；为了尝试在减少搜索空间的条件下进一步优化搜索策略，决定采用蒙特卡洛树搜索策略。

2. 算法原理：
    - 蒙特卡洛树搜索基于对双方博弈进行多次模拟，根据最后的模拟结果选择最佳的下一步。minimax为了节省搜索时间，是在指定层数的条件下，遍历所有下子可能的结果，从中选择最优的策略；而MCTS则是进行多次迭代，每次迭代根据UCB公式选择需要扩展的结点，在这一步基础上进行若干次模拟，每次模拟都是随机下子，因此不需要遍历所有下子的可能。

3. 相关公式：UCB算法公式

$$\underset{v' \in v's-children}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2lnN(v)}{N(v')}}$$

v表示父结点，v'表示v的子结点，Q表示结点的行动值，N表示结点的访问次数，c表示一个常数，用于调整访问次数的影响。

当前结点根据UCB公式选择最优的子结点，首先计算将子结点的Q除以N表示平均行动值，同时要考虑访问次数的限制，在迭代时，MCTS希望能够优先考虑没有访问过的或者访问次数比较少的子结点，因此引入$c\sqrt{\frac{2lnN(v)}{N(v')}}$的限制，c根据经验公式取$\frac{1}{\sqrt{2}}$，所有迭代完成后，此时只需要考虑平均行动值，不需要考虑访问次数的限制，根据这个选择最优的下一步。

4. 算法步骤：

   进行多次迭代，每一次迭代都经过以下4个步骤：

   - selection：在树的初始结点找到下一个能够探索的结点，一般是从子结点入手，如果此时初始结点为叶子结点，则进入expansion步骤；如果此时子结点中还没有完全探索完，那么选择没有探索过的子结点进行探索；如果所有子结点都探索过，那么根据UCB公式选择最优的子结点，以该子结点为初始结点按照上面方法继续寻找值得探索的结点。
   - expansion：在当前结点随机地选择下一步创建一个新的子结点，该子结点不能与之前创建的子结点重复，以该子结点为探索的结点，进入simulation步骤。
   - simulation：以探索的结点为起点，双方开始模拟对弈，每一次对弈都是根据随机策略下子，直到游戏结束或者指定的深度，得到当前游戏状态的得分，进入backpropagation步骤。
   - backpropagation：把得分从探索的结点一步步向上更新，更新向上路径中每一个结点的访问次数和行动分数值，为了下一次迭代计算UCB值。

   迭代完成后，根据UCB公式选出初始结点中最佳的子结点，以此行动作为下一步的下子。

5. 算法策略：

   - simulation采用两种策略：
     - 一次模拟博弈直到游戏结束状态为止，以胜负作为行动得分。
     - 一次模拟博弈直到指定深度为止，结合评估函数，以当前状态的评估函数值作为行动得分。
   - 调参
     - 搜索迭代次数
     - 搜索深度
     - 评估函数的不同计算方式

B. 遗传算法

1. 算法引入

   对黑白棋的评估函数进行调参，需要手动调整参数并且让AI互相对战，这样训练效率比较低，而且往往陷入局部最优，难以找到最优解，为了能够系统地训练参数，使用遗传算法，指定多个权重个体组成第一代种群，每次进行循环对战，计算其胜负情况，得出适应度，从而更好地判断训练情况，为了避免个体取值在指定范围徘徊，会进行交叉和变异操作，使其产生新的个体，经过多次迭代训练，可以找到一个相对较好的解。

2. 算法策略

   - 适应度计算
     - 胜负计算差
     - 双方终局棋子差
   - 迭代收敛判断
     - 指定次数迭代
     - 到某一代最好的个体与最差的个体之间的差距低于某一阈值
   - 遗传算法与蒙特卡洛树算法结合，使用遗传算法训练好评估函数的权值参数，使用蒙特卡洛树进行搜索，指定层数，进行多次迭代模拟，结合评估函数，强化AI能力。

3. 算法步骤

- 初始化种群

  设定种群的大小g、个体的长度d以及迭代次数i，个体长度为评估函数的参数权值的个数，种群大小和迭代次数影响训练效果，需要进行调参，为了避免训练过程中陷入局部最优收敛，一开始个体的赋值应该尽量广泛化。

- 进行多次迭代

  - 计算适应度：

    让种群中的个体两两相互对战，分为先手和后手两局，进行循环赛，记录每个个体的对战结果，让赢的次数减去输的次数作为适应度值，统计所有个体的适应度值。

  - 选择：

    根据适应度选择前10%的个体，将其加入到下一代种群，剩余的个体则进入交叉和变异操作。

  - 交叉：

    使用**轮盘赌法**，将每个个体的适应度值占所有个体的总适应度值的比例作为遗传下一代的概率，每次根据概率选取两个个体进行交叉，根据概率选择一个点，从该点后两个个体的所有参数进行交换，产生两个新的个体。

  - 变异：

    在交叉产生的新个体基础上，遍历该个体的所有权值参数，对于每一个值，根据一定的概率选择进行变异，如果概率达到某种条件，进行变异，否则不进行变异，进行变异时，对该值乘以一个0.7-1.3范围的权重，该权重也由随机数产生，将产生变异的个体加入到下一代种群。

- 选择最优个体

  根据适应度，选择最优的个体作为评估函数的权重。

4. 轮盘赌法

- 比例选择法：各个个体被选中的概率与其对应的适应度成正比。
- 步骤：
  - 计算适应度，循环对战得到每一个个体的适应度。
  - 将所有适应度求和，求出各个个体的占比：

    $$P(x_i) = \frac{f(x_i)}{\sum_{j=1}^{N} f(x_j)}$$

- 根据该概率进行选择进行交叉变异，产生新个体加入到下一代种群中。

5. 算法评价

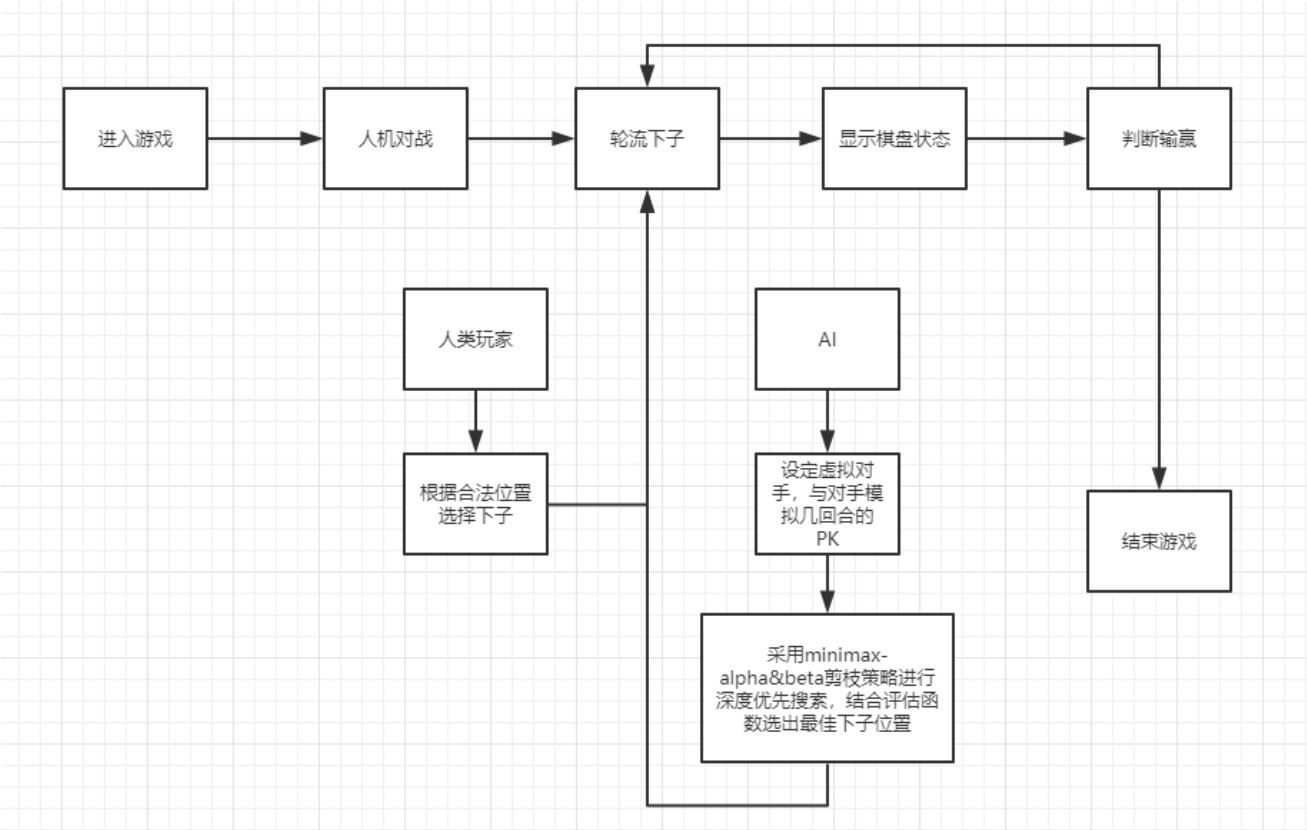
- 遗传算法经过多次训练未必能够达到全局最优解，到了一定次数迭代可能就处于局部最优解的状态就收敛。
- 由于最优一代其最好一个未必就比前几代就好，因此在训练比较充分时，可以近似认为此时总体已经比较优秀。
- 由于黑白棋对战没有传递性，因此不能绝对确定双方对战的优劣

C 评估函数

1. 根据棋子个数差表示的评估函数（A）
    - 求双方棋子之差作为预估。
2. 根据位置矩阵权重的评估函数。（B）
    - 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。
3. 综合位置矩阵权重、棋子差、行动力、稳定子的评估函数。（C）
    - 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。
4. 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。（D）
    - 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. 伪代码

- MCTS

```
Algorithm MCTS
    #进行多次迭代
```

```
    for i <- 0 to iterions:
        #每次迭代进行必要初始化
        init gameboard
        #选择结点
        while node has not been chosen:
            #如果当前结点的所有孩子被选择了，根据UCB选择最优的
            if root.children have been visited:
                choose best child according UCB and let root <- child
            else:
                #否则进行扩展新的孩子结点，选择该结点
                expand new_child from root and select new_child chosen
                break
        #根据选择的结点进行模拟，一直到指定条件或者游戏结束
        simualtion according to select_node until game finish
        #从结果得到反馈值
        get action_value from the result of the game
        #从选择结点向上更新行动值和访问次数，一直到根结点
        update action_value & visit_count from select_node to root
    #迭代完成后，选择最优的一个孩子结点就是下一步要走的行动
    choose best child from root
```

- GA

```
Algorithm GA
    #进行多次迭代
    for i <-0 to iterions:
        #让个体之间进行循环赛，算出适应度
        caculate fitness from population
        #按比例得到种群中最好的一批
        get best individuals from population
        #保留这些个体进入下一代
        put best individuals into next generation
        #计算剩余个体各自的遗传概率，与适应度成正比
        caculate genetic rate of rest individual
        #进行若干次选择
        for several choice
            #根据轮盘赌法，选择两个个体
            choose two individuals according to genetic rate
            #让这两个个体进行交叉
            let two individuals crossover into new individuals
            #新个体根据一定概率产生变异
            let new individuals mutate according to specific rate
            #让新的个体进入下一代
            choose new individuals into next generation
    #迭代完成，计算所有个体的适应度
    caculate fitness from population
    #选择最优的作为结果
    get best individual from population
```

（3） 关键代码

A. 蒙特卡洛树搜索

1. 使用树结点存储结构，记录子结点列表、父母结点、行动下子的坐标、访问次数、行动的分数，所有合法的走法。

```
#树结点
class TreeNode():

    #初始化
    def __init__(self,parent=None,action=None):
        self.children = []  #孩子结点
        self.parent = parent  #父母
        self.action = action  #行动
        self.visit_count = 0  #访问次数
        self.action_value = 0.0  #行动分数
        self.valid_choice = []  #合法走法
        self.child_visit_index = []  #访问过的孩子下标，方便计算
```

2. 设置部分函数方便判断，比如判断是否为叶子结点，插入新的结点时设置父子关系，判断是否完全扩展。

```
    #判断当前结点是否为叶子结点
    def is_leaf(self):
        #如果当前没有孩子，返回正确
        if len(self.children) == 0:
            return True
        #如果有孩子，证明不是叶子结点
        return False

    #插入新结点
    def insert_new_child(self,sub_node):
        sub_node.parent = self #设置父母
        self.children.append(sub_node)  #添加结点入孩子列表

    #判断是否完全扩展
    def is_total_expand(self):
        #如果该结点的所有孩子结点都已经被访问，则相当于完全扩展
        if len(self.valid_choice) == len(self.children) and len(self.children) != 0:
            return True
        return False
```

3. 构建蒙特卡罗树进行搜索，首先必要变量的初始化。

```
#蒙特卡罗搜索树
class MonteCarloTreeSearch():
    #初始化
    def __init__(self,mark,anti_mark,gameboard,size,weights,node,iter1):
        self.turn = 0  #轮流顺序
        self.chess = {0:mark,1:anti_mark}  #棋子颜色
        self.gameboard = gameboard  #棋盘对象
        self.size = size  #棋盘大小
        self.ele_weights = weights  #权重
        self.iter = 80  #迭代次数
        self.node = node  #结点
        self.constant = 1 / math.sqrt(2.0)  #常数
```

4. MCTS整个搜索实现过程，进行指定次数的迭代，对于每次迭代，进行必要的重新初始化，同时复制拷贝新的棋盘，首先选择并且扩展一个结点，根据这个结点进行模拟，返回一个行动值分数，然后再进行回溯一路更新结点的访问次数和行动值分数，所有迭代进行完毕后，会选择平均行动值最高的结点进行下子。

```
    #搜索实现
    def search(self):
        #进行多次迭代
        for i in range(self.iter):
            self.turn = 0  #轮流顺序
            self.depth = 10  #深度
            #复制棋盘，防止影响原来的状态
            gameboard = Gameboard()
            gameboard.board = []
            #复制每一个位置
            for i in range(self.size):
                tmp = [x for x in self.gameboard.board[i]]
                gameboard.board.append(tmp)
            #选择并扩展一个结点
            select_node = self.selection_expansion(self.node,gameboard)
            #进行模拟
            action_value = self.simulation(select_node,gameboard)
            #回溯更新行动值和访问次数
            self.back_propagation(select_node,action_value)
        #得到最好的行动步骤
        best_action = self.choose_best_action(self.node)
        return best_action
```

5. 在游戏状态还没结束前，首先判断当前子结点是否已经访问过，如果是，则根据UCB公式选择最好的子结点进入下一层，继续寻找合适的子结点；如果当前结点为叶结点，则计算得到所有合法下子的列表，更新其下子选择，从中选择一个子结点创建并且作为要扩展的目标返回；如果不是叶结点并且还没有全部访问，也选择一个还没有访问的子结点创建并且作为要扩展的目标返回。

```
    #选择并且扩展结点
    def selection_expansion(self,node,gameboard):
```

```
        #直到终点为止
        while gameboard.is_over() != True:
            #如果子结点已经全部被扩展
            if node.is_total_expand() == True:
                #选择最好的子结点
                node = self.get_best_child(node)
                #根据当前选择的子结点下子
                gameboard.move(self.chess[self.turn],node.action)
                #改变角色
                self.turn = (self.turn+1) % 2
            #如果是叶子结点
            elif node.is_leaf() == True:
                #找出合法下子的所有坐标
                valid_choice =
self.gameboard.find_right_flipping_position(self.chess[self.turn])
                #如果当前没法下子，直接返回
                if len(valid_choice) == 0:
                    return None
                #更新当前结点的子结点下子选择列表
                node.valid_choice = valid_choice
                #扩展新的子结点
                new_sub_node = self.expansion(node)
                #下子
                gameboard.move(self.chess[self.turn],new_sub_node.action)
                #改变角色
                self.turn = (self.turn+1) % 2
                #返回选中的子结点
                return new_sub_node
            else:
                #扩展新的子结点
                new_sub_node = self.expansion(node)
                #下子
                gameboard.move(self.chess[self.turn],new_sub_node.action)
                #改变角色
                self.turn = (self.turn+1) % 2
                return new_sub_node
        #返回结点
        return node
```

6. 根据扩展的结点状态开始进行模拟双方随机下子，搜索到指定的深度，返回当前棋盘下子状态的评估函数值。

```
    #模拟
    def simulation(self,node,gameboard):
        #搜索到最低或到指定深度
        while gameboard.is_over() != True and self.depth >= 0:
            #搜索合法走法
            valid_choice = gameboard.find_right_flipping_position(self.chess[self.turn])
            #如果当前没有合法走法，则换到另一方下子
            if len(valid_choice) == 0:
                self.turn = (self.turn+1) % 2
```

```
                continue
            #随机得到一个合法走子
            index = random.randint(0,len(valid_choice)-1)
            #得到行动步骤
            action = valid_choice[index]
            #下子
            gameboard.move(self.chess[self.turn],action)
            #转换角色
            self.turn = (self.turn+1) % 2
            #深度减一
            self.depth -= 1
        #使用评估函数评估当前得分
        final_reward = self.evaluation_best(self.chess[0],gameboard)

        return final_reward
```

7. 根据返回的行动值，从扩展的结点向上一步步更新其访问次数和行动值。

```
    #回溯
    def back_propagation(self,select_node,action_value):
        #回溯，直到父结点为空
        while select_node != None:
            #访问次数加一
            select_node.visit_count += 1
            #行动值增加
            select_node.action_value += action_value
            #向上追溯
            select_node = select_node.parent
```

8. 根据UCB公式，对于每一个结点，将其行动值除以访问次数得出平均行动值，并且考虑访问次数的限制，优先考虑访问次数较少的或还没有访问的因素，两者相结合作为选择子结点的标准。

```
    #选择最好的子结点，使用UCB评判方式
    def get_best_child(self,node):
        optimal_score = -10000
        optimal_son_node = None
        #遍历所有子结点，找出最优的
        for son_node in node.children:
            #公式前半部分，平均行动力
            value1 = son_node.action_value / son_node.visit_count
            #公式后半部分，加入访问次数限制，优先访问已访问次数少的
            value2 = (math.log(node.visit_count)*2.0) / son_node.visit_count
            tmp_score = value1 + self.constant*math.sqrt(value2)
            #如果找到更优的值，进行更新
            if tmp_score > optimal_score:
                optimal_score = tmp_score
                optimal_son_node = son_node
        #返回最优子结点
```

```
        return optimal_son_node
```

9. 从合法下子选择一个行动，如果该行动已经有创建结点，则继续重新取，根据取出的行动下子坐标，创建一个新的子结点，加入到父结点的孩子队列，返回子结点。

```
    #扩展新的子结点
    def expansion(self,node):
        #合法下子列表
        length = len(node.valid_choice)
        #循环
        while True:
            #随机得到一个下子下标
            index = random.randint(0,length-1)
            #如果没有扩展过，则跳出，否则一直随机找下标
            if index not in node.child_visit_index:
                break
        #添加下子访问下标列表
        node.child_visit_index.append(index)
        #创建新的结点
        new_node = TreeNode(node,node.valid_choice[index])
        #加入到父结点的子结点列表
        node.children.append(new_node)
        return new_node
```

B. 遗传算法

1. 进行初始化，设置种群的规模，每个个体的大小，包含多少个权值参数，定义迭代的次数，初始化种群的数值，既可以随机初始化，这样可以避免陷入局部进化，但是进化时间很长，又可以提前设定指定范围的值，可以较快收敛，但是容易有局部最优限制。

```
#遗传算法
class GeneticAlgorithm():
    #初始化
    def __init__(self,population_size,individuals_size,iter_size):
        self.population_size = population_size #种群数量
        self.individuals_size = individuals_size  #个体大小
        self.popul = []  #种群，存储各个个体，每个个体是一个权值列表
        self.fit_score = []  #适应度列表
        self.iter_size = iter_size  #迭代次数
        self.negative = [2,3]  #负数参数列表
        self.popul = [
            [5,5,-10,-4,4,4],
            [15,6,-8,-4,2,2],
            [13,5,-9,-3,3,3],
            [15,3,-6,-6,4,2],
            [10,8,-7,-4,3,5]
        ]
```

2. 在种群中进行循环赛，个体与个体之间两两进行比赛，分为先手后手比赛，记录比赛胜负结果，作为适应度值。

```
    #计算适应度
    def caculate_fitness(self):
        #初始化适应度列表
        self.fit_score = [0] * self.population_size
        #循环赛PK
        for x in range(self.population_size):
            for y in range(self.population_size):
                if x != y:
                    #选择两个个体进行比赛，x为先手，y为后手
                    x_count,o_count = chess_game(self.popul[x],self.popul[y])
                    if x_count > o_count :
                        result1 = 2
                        result2 = 0
                    elif x_count == o_count:
                        result1 = 1
                        result2 = 1
                    else:
                        result1 = 0
                        result2 = 2
                    #将输赢的结果作为分数更新到对应的个体的适应度值
                    self.fit_score[x] += result1
                    self.fit_score[y] += result2
```

3. 根据选择的两个个体，选中一个点，该点后面的部分进行交换。

```
    #交叉操作
    def crossover_operation(self,individual1,individual2):
        #得到交叉点
        index = random.randint(0,self.individuals_size-1)
        #根据交叉点的位置，将两者后面的列表进行交换
        new_individual1 = individual1[:index] + individual2[index:]
        new_individual2 = individual2[:index] + individual1[index:]

        return new_individual1,new_individual2
```

4. 遍历个体的所有值，对于每一个值，都有一个概率决定是否发生变异，如果发生变异，也有一个概率决定变化的趋势。

```
    #变异操作
    def mutation_operation(self,individual):
        #遍历整个个体列表
        for i in range(self.individuals_size):
            #对于每一个位置，都有一个概率，如果概率满足一定条件，进行变异
            pro = np.random.random()
            if pro <= MUTATION_RATE:
                #得到一个在0.7-1.3的权重
                weight = random.uniform(0.7,1.3)
                #将值乘以权重得到一个新的变异值
                individual[i] = individual[i] * weight
        #返回新的个体
        return individual
```

5. 计算每个个体被选择进行交叉的概率，根据轮盘赌法，概率与适应度大小成正比。

```
    #计算遗传到下一代的概率
    def genetic_rate(self):
        total_sum = 0
        #计算适应度总和
        length1 = len(self.fit_score)
        for i in range(length1):
            total_sum += self.fit_score[i]
        gene_rate = []
        #计算每一个个体的遗传概率
        for i in range(length1):
            tmp = (float)(self.fit_score[i]) / total_sum
            gene_rate.append(tmp)

        return gene_rate
```

6. 遗传算法训练过程，进行多次迭代，每一次迭代，通过循环赛得出各个个体的适应度值，选择前10%的个体保留到下一代，剩下的根据轮盘赌法选择两两个体进行交叉，根据一定概率产生变异，得到新的个体，加入到下一代，直到训练收敛，最后选择最优的个体作为评估函数的权重参数。

```
    #遗传算法训练过程
    def GA_Train(self):
        #进行多次迭代
        for i in range(self.iter_size):
            #计算适应度
            self.caculate_fitness()
            print("fitness: ",self.fit_score)
            # print(self.popul)
            #得到最高的适应度值
            max_fit_score = max(self.fit_score)
            #选择适应度分数最高的个体

            max_fit_index = self.fit_score.index(max_fit_score)
```

```
            #定义下一代的列表
            new_popul = []
            #把适应度最高的个体加入到下一代
            new_popul.append(self.popul.pop(max_fit_index))
            #将该个体从当前适应度列表删除
            self.fit_score.pop(max_fit_index)
            length1 = len(self.fit_score)
            #计算遗传概率
            gene_rate = self.genetic_rate()
            print(gene_rate)
            #交叉、变异
            for j in range((int)(length1/2)):
                #根据轮盘转法，根据遗传概率选择两个个体进行交叉
                select_choice = np.random.choice(length1,2,False,gene_rate)
                #进行交叉操作
                individual1,individual2 =
self.crossover_operation(self.popul[select_choice[0]],self.popul[select_choice[1]])
                #对个体进行按概率变异操作
                individual1 = self.mutation_operation(individual1)
                individual2 = self.mutation_operation(individual2)
                #将产生的新个体放到下一代
                new_popul.append(individual1)
                new_popul.append(individual2)
            #更新种群
            self.popul = new_popul
        #迭代完成后，选择最优的权重参数
        max_fit_score = max(self.fit_score)
        max_fit_index = self.fit_score.index(max_fit_score)
        print(self.popul[max_fit_index])
```

C. 评估函数

1. 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
```

2. 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
```

3. 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]
```

4. 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)
```

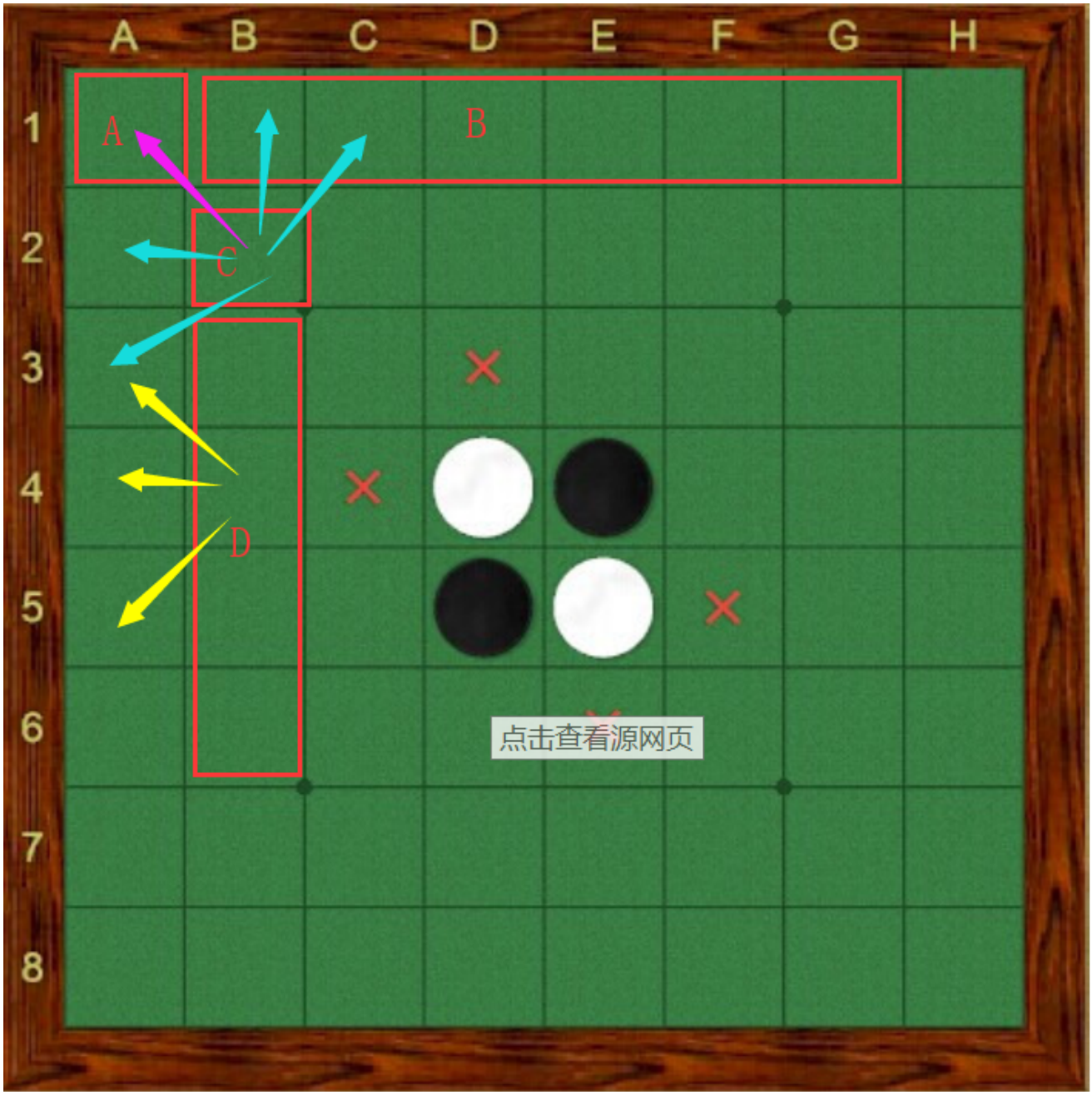
- 根据各个因素，赋予权值，相加得出最终的结果，由于inner_edge、inner_corner为负面因素，因此需要给予负数的权值，经过多次训练，得出较优化的权值参数为：15,6,-10,-4,8,8。对应的是：角、边、内层角、内层边、棋子差、行动力。

```
        #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

- 使用蒙特卡洛树搜索策略优化搜索空间，提高搜索效率。
- 将蒙特卡洛树与经过优化的评估函数结合。
- 蒙特卡洛树与遗传算法结合，利用遗传算法进行训练，得到最优的评估函数的权值参数，将其应用到蒙特卡洛树的模拟中，进一步强化行动分数值的合理性，提高AI的学习能力。
- 使用多种不同的评估函数进行比较，选择最优的考虑策略。
    - 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，如果紫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，同时青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，以第二个格子为例，黄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。

实验结果及分析

（1）实验结果与评测

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。

1. **利用遗传算法训练评估函数D，使用不同的种群大小、迭代次数，得到不同的权重参数，将其与原来的权重参数进行pk。**

   原来权重为15 6 -8 -6 4 4；先手指遗传算法版本的先下子，后手指遗传算法版本的后下子，结果显示都统一为前面的棋子数字是遗传算法的，后面的棋子数字是原来版本的，棋子比为最后终局的结果。

序号	种群大小	迭代次数	先手	后手
1	5	10	20:44	25:39
2	5	30	29:35	24:40
3	10	10	20:44	23:41
4	10	20	24:40	30:34
5	10	50	35:29	24:40
6	20	30	29:35	40:24
7	20	50	34:30	36:28
8	20	80	40:24	50:14

由表可知，遗传算法效果与种群大小和迭代次数有关，种群比较小时，个体与个体之间相互比较次数有限，交叉和变异能够产生更好的参数更少，同时如果迭代次数比较少，那么训练效果会比较弱，因此一开始还是不能打赢之前的版本，如果增大种群数量、迭代次数，那么此时遗传算法的优势开始逐渐体现出来，可以找到更好的参数。

然而，遗传算法每一次迭代的时间需要很长，如果种群设得太大，比如20个个体，那么两两循环对战，先手和后手各一局，那么就得20*19=380局，因此minimax层数如果设得太高，一局PK时间就会非常长，因此为了训练，将层数设小，尽可能训练更多的迭代，根据上面结果，虽然能够打赢同一搜索层数的原来版本，但是原来版本如果增加层数，还是打不赢。

因此，如果想要更好的训练效果，就必须得增大种群数量，最好30-50个，迭代次数至少要几百次，理论上应该能够得到比较好的效果，但是时间至少需要持续训练一周，由于电脑CPU有限，因此没有办法持续训练这么久，只能按照上面的参数进行训练，放到集群上跑，开多个进程同时训练，大概跑了8个小时。

2. **依据蒙特卡洛树，进行多次迭代模拟，与之前使用minimax搜索的进行比对，用评估函数D，权重为15 6 -8 -6 4 4。**
   - 先手指MCTS算法版本的先下子，后手指MCTS算法版本的后下子，结果显示都统一为前面的棋子数字是MCTS算法的，后面的棋子数字是原来版本的，棋子比为最后终局的结果。

- 以搜索到终局为模拟结束条件：

迭代次数	先手	后手
10	20:44	30:34
20	29:35	24:40
30	34:30	23:41
50	22:40	36:28
80	29:25	40:24
100	50:16	34:30
120	30:34	42:22

由上面的表可以看到，MCTS并没有收敛，迭代次数增高，并没有明显得出能力一定提高，随机性还是比较强，但总体而言，迭代次数增加，相对来说，AI思考能力有相对提高。

- 以搜索到指定层数，根据评估函数D的评估作为反馈，从而选择最优下一步。

搜索层数	迭代次数	先手	后手
10	200	30:34	20:44
10	300	35:29	10:54
15	150	43:21	28:36
15	200	35:29	45:19
20	180	22:42	34:30
20	200	25:38	22:42
25	300	34:30	46:18

总体而言，搜索层数越高，不一定效果就越好，迭代次数提高大致能够提高训练的效果，但是没有绝对性的优化，理论上，应该加大迭代次数到一定程度，才有有所效果，只是迭代次数如果上1000次，那么每做一步决策就得花很长时间。

3. **评估函数参数优化比较**

以评估函数D为例：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	50 : 14	42 : 22
10 6 6 -4 3 3	10 5 -8 -4 4 4	55 : 9	29 : 35
10 6 6 -4 3 3	15 6 -8 -4 4 4	13: 51	20 : 42
15 6 -8 -4 4 4	15 6 -8 -4 5 5	35 : 29	28 : 36
15 6 -8 -4 4 4	18 6 -8 -4 4 4	34 : 30	27 : 37
15 6 -8 -4 4 4	15 10 -8 -4 4 4	35 : 29	22 : 42
15 6 -8 -4 4 4	15 6 -12 -4 4 4	35 : 29	23 : 41
15 6 -8 -4 4 4	15 6 -10 -4 4 4	34 : 30	35 : 29
18 6 -8 -4 4 4	10 5 -8 -4 4 4	25 : 39	36 : 28
15 6 -10 -4 4 4	18 6 -8 -4 4 4	36 : 28	29 : 35
15 6 -10 -4 4 4	15 6 -10 -4 6 6	33 : 31	30 : 34
15 6 -10 -4 4 4	15 6 -10 -4 8 8	40 : 24	26 : 38
15 6 -10 -4 8 8	10 5 -18 -4 8 8	41 : 23	38 : 26

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. 遗传算法可以通过训练种群，经过不断迭代，从而找到最优的解，但是遗传算法容易陷入局部最优，多次训练后，得到当代第一个最好的并不一定就比前面的几代其他个体好，并且在局部收敛后，就无法进一步训练，因此在初始化种群时尽量让个体数目多点，具有代表性，同时迭代结束条件可以设定为一代中第一名和最后一名水平差异不大时结束，此时可以认为该代的总体水平较高。
2. MCTS搜索策略利用UCB选择顶点进行随机模拟，为了比较不同情况，对模拟结束条件设定为终局结束和指定层数以评估函数值作为反馈更新行动值，无论是哪种策略，模拟迭代次数应该尽可能高，这样可以保证其作出决策的合理性，总体来说，迭代数不高时，MCTS难以收敛，作出决策不稳定，并且容易陷入被动，因此需要进一步优化。
3. 优化评估函数能够提高搜索的合理性，通过更新棋盘边界的权重，可以根据不同情况作出更好的决策，根据形势的占边策略在大多数情况还是有明显优势，不会陷入当前棋子数差的限制，从而考虑得更加长远，因此在PK对战中比较有优势。