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	计科 1 班	专业（方向）	计算机科学与技术
学号	21307077	姓名	凌国明

一、实验题目

黑白棋博弈树搜索

通过 $MiniMax$ 搜索算法实现 8×8 的黑白翻转棋的人机对战
通过 $AlphaBeta$ 剪枝削减博弈树的规模，加速搜索。
通过 **遗传算法** 和 **模拟退火算法** 优化评估函数，并比较两种算法的表现
通过 $matplotlib$ 库设计 UI 使得对战更清晰流畅

二、实验内容

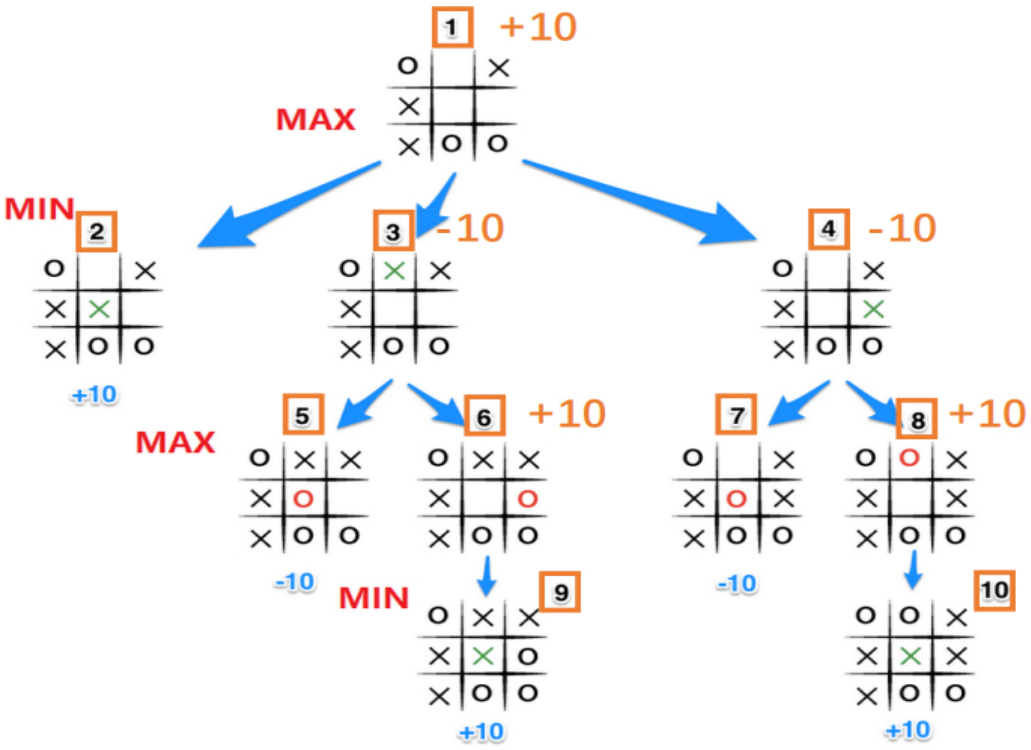
1、算法原理

$MiniMax$ 搜索

两方的**完全信息的零和博弈**：

1. 两名玩家**轮流行动**，进行博弈
2. **行动数量有限**，不存在随机性。
3. **零和博弈：一方的损失相当于另一方的收益，总收益为0**，结局有三种可能：玩家A获胜、玩家B获胜、平局
4. 所有**信息公开**透明，博弈双方知道所有信息。
5. **博弈双方足够聪明**，总能作出最佳的选择。

搜索过程：定义一个效益值，Max 玩家的目的是**最大化这个效益值**，Min 玩家的目的是**最小化这个效益值**（因为是零和博弈，所以 Max 玩家的效益减少意味着 Min 玩家效益的增大）
构建搜索树时，设定搜索最多 N 层，两个玩家都会作出 N 步后 使得 自身效益值最大化 或 对方效益值最小化 的决策。

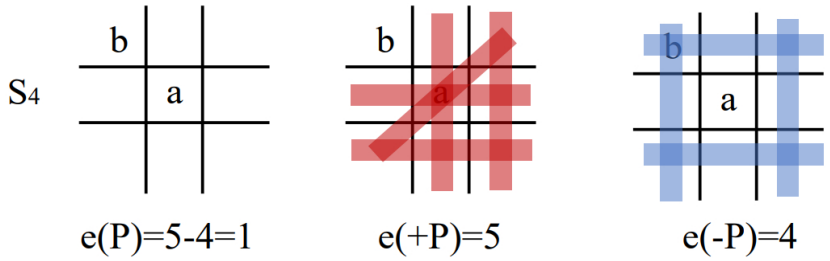


1. 构建决策树：Max 玩家会作出 N 步后 使得 自身效益值最大化 的决策，这一层称为 Max层；Min 玩家会作出 N 步后 使得 对方效益值最小化 的决策，这一层称为 Min层
2. 计算叶子节点的效益值：当不限制搜索层数时，叶子节点的效益值是游戏结束时 Max 玩家的效益值；当限制最多搜索 N 层，且游戏未结束时，叶子节点的效益值是对Max 玩家的效益值的估计。
3. 自底向上计算每个结点的 $MiniMax$ 值：Max 节点的值为它的子节点的效益值的最大值；Min 节点的值为它的子节点的效益值的最小值
4. 从根结点选择 $MiniMax$ 值最大的分支，作为行动策略。

评估函数

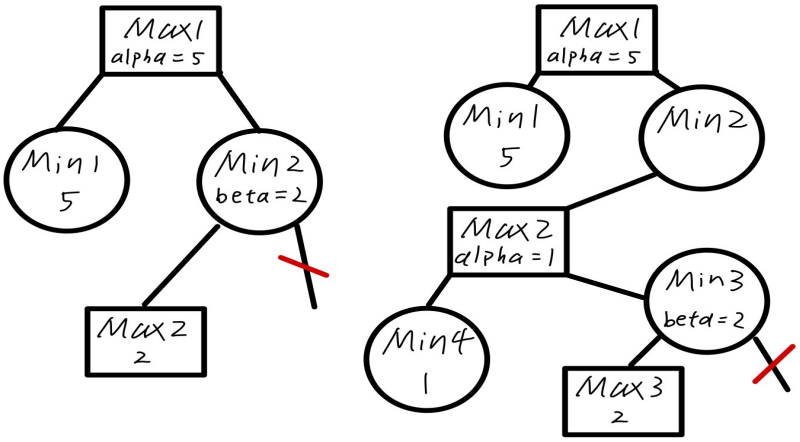
实际应用中，$MiniMax$ 博弈树的**规模庞大**，从开始状态扩展到“博弈结束”的叶子节点的代价是巨大的。因此需要**限制博弈树搜索的深度**。
但是当搜索深度受限时，原来的内部节点会变为现在的叶子节点，也就是说无法通过游戏的最终局面来确定叶子节点的效益值。这个时候我们就要**通过评估函数来评价博弈双方的局面**，**估计**Max 玩家的**效益值**。
井字棋中一个评估函数的例子如下：

- 设棋局为P, 估价函数为e(P)
- e(P)=e(+P)-e(-P)
- e(+P)表示棋局P上有可能使a成为三子成一线的数目
- e(-P)表示棋局P上有可能使b成为三子成一线的数目



$AlphaBeta$ 剪枝

剪掉不可能影响决策的分支，尽可能地削减部分搜索树。



考虑 Min 节点的 $beta$ 剪枝：

1. 上图左边的基本情况：$Max1$ 节点的 $alpha$ 值为 5，$Min2$ 节点的 $beta$ 值为 2，则 $Min2$ 节点的最终效益值一定小于等于 2， $Max1$ 节点的最终效益值一定大于等于 5。因此 $Min2$ 节点的其他分支对 $Max1$ 节点的效益值计算已经无用，可以剪枝。
2. 上图右边的泛化情况：$Max1$ 节点的 $alpha$ 值为 5，$Min2$ 节点的 $beta$ 值为 2。$Min3$ 节点如果要对 $Max1$ 节点有贡献，则要求 $Min2$ 节点的最终效益值大于等于5，这要求 $Max2$ 节点的最终效益值至少要大于等于5，这要求 $Min3$ 节点的最终效益值至少大于等于5。但是已知 $Min3$ 节点的 $beta$ 值为 2，这意味着$Min3$ 节点的最终效益值小于等于2，意味着 $Min3$ 节点对 $Max1$ 节点 毫无贡献。
3. 更深层的情况也类似，Max 节点的 $alpha$ 剪枝也类似。可以通过归纳法证明，当 Min 节点的 $beta$ 值 ≤ 祖先 Max 节点的最大 $alpha$ 值时，可以进行剪枝。当 Max 节点的 $alpha$ 值 ≥ 祖先 $Minx$ 节点的最小 $beta$ 值时，可以进行剪枝。

高级搜索算法

本实验中，需要用到启发式评估函数来对局势进行判断，从而得出博弈树中叶子节点的值。黑白棋的评估函数包括棋盘每个位置的权重，子数的权重，稳定子的权重，行动力的权重等等。我们要通过遗传算法和模拟退火算法来**确定一组最优的参数**，从而提**升评估函数对局势判断的准确程度**（最优的参数意味着评估函数对局势的判断最准确，这可以帮助搜索算法找出最优的选择），以此来改善 $MiniMax$ 搜索的表现，**提高胜率。**

认为模拟退火算法是爬山法的改进版本，增加了“以一定概率接受劣解”的机制，从而使得算法可以**跳出局部最优**，更可能解决全局最优。退火算法中，如果新解优于旧解，则接受新解；如果**新解劣于旧解**，则**温度越高，接受新解的概率越高。**这使得退火算法**前期可以跳出局部最优，后期可以稳定收敛。**

遗传算法是模拟自然界优胜劣汰的算法，包括种群交叉、基因变异、自然选择三个部分，遗传算法可以**很快地收敛到一组较优的解空间**，**全局搜索性较强，但局部搜索性能稍差。**

关于遗传算法和模拟退火算法的**优缺点**会在**创新点**部分提到。

2、伪代码

判断落子是否有效

Algorithm 1 落子有效性

```
procedure 落子有效性(落子位置，棋子颜色，棋盘)
    if 落子位置越界 或 落子位置已经有棋子 then
        return 无效
    end if
    for 八个方向 do
        if 这个方向上邻接着对方的棋子 then
            位置 = 落子位置向这个方向走一步
            标志 |= 有效性搜索(位置，颜色，方向，棋盘)
        end if
    end for
    return 标志
end procedure
procedure 有效性搜索(位置，颜色，方向，棋盘)
    if 从位置 向 方向 走一步 不会越界 then
        位置 = 位置向这个方向走一步
        if 位置上是对方棋子 then
            return 有效性搜索(位置，颜色，方向，棋盘)
        end if
    end if
    if 位置上是己方棋子 then
        return 有效
    end if
    return 无效
end procedure
```

$MiniMax$ 搜索（带$AlphaBeta$剪枝)

Algorithm 2 深搜

```
procedure 深搜(层数，最大层数，玩家，最大alpha，最小beta)
    if 层数 == 最大层数 then
        return 分数(棋盘)
    end if
    for 棋盘的每个位置 do
        if 落子有效(位置，玩家，棋盘) then
            在这个位置上落子，并翻转棋子
            分数 = 深搜(层数+1，最大层数，另一个玩家，最大alpha，最小\beta)
            子节点分数（数组） += 分数
            撤回这个位置上的落子，并撤回棋子的翻转
            if 玩家 == MAX玩家 then
                if 分数 >= 最小\beta then
                    break
                else if 分数 > 最大alpha then
                    最大alpha = 分数
                end if
            end if
            if 玩家 == Min玩家 then
                if 分数 <= 最大alpha> then
                    break
                else if 分数 < 最小\beta then
                    最小\beta = 分数
                end if
            end if
        end if
    end for
    if 子节点分数数组为空（意味着没有位置可走） then
        return 深搜(层数+1，最大层数，另一个玩家，最大alpha，最小\beta)
    end if
    if 玩家 == Max玩家 then
        return 最大值(子节点分数数组)
    else
        return 最小值(子节点分数数组)
    end if
end procedure
```

$MiniMax$ 搜索的主体部分是深度优先搜索

$AlphaBeta$ 剪枝只是增加了 Max_Alpha 和 Min_Beta 两个参数的维护

当 Min 节点的 $beta$ 值 ≤ 祖先 Max 节点的最大 $alpha$ 值时，可以进行剪枝。

当 Max 节点的 $alpha$ 值 ≥ 祖先 $Minx$ 节点的最小 $beta$ 值时，可以进行剪枝。

当剪枝具体逻辑见 \IF{玩家 == MAX玩家} 的两个判断条件

遗传算法

Algorithm 3 遗传算法

```
procedure 遗传算法()
    初始化 遗传代数，种群规模，初始竞争次数
    初始化 交叉概率，变异概率，变异步长
    随机生成 种群
    for 初始竞争次数 do
        种群 = 自然选择(种群)
    end for
    for 遗传代数 do
        种群 = 种群交叉(种群，交叉概率)
        种群 = 基因变异(种群，变异概率，变异步长)
        种群 = 自然选择(种群)
    end for
    while 种群规模 > 1 do
        种群 = 自然选择(种群)
    end while
    return 种群[0]
end procedure
procedure 自然选择(种群)
    初始化新种群为空，初始化一个从 0 到 种群大小 的数组 arr，并随机打乱 arr
    for i 从 0 到 种群大小 / 2 do
        开启一个新进程，种群[i] 和 种群[arr[i]] 对打，将胜者加入新种群中
    end for
    return 新种群
end procedure
```

遗传算法中各个参数的选择、种群初始化方法，自然选择的方法，在创新点中详细介绍

模拟退火算法

Algorithm 4 模拟退火

```
procedure 模拟退火()
    初始化 初始温度，内层循环，退温系数
    while 温度 > 临界温度 do
        for 内层循环 do
            新解 = 扰动(当前解)
            分数 = 对战(新解，当前解)
            if 分数 > 0（新解更好） 或 e^(-分数/温度) > 0-1的随机数 then
                当前解 = 新解
            end if
        end for
        温度 *= 退温系数
    end while
    return 当前解
end procedure
```

3、关键代码展示（带注释）

判断落子合法性

```
# 判断落子合法性
def is_valid(self, row, col, player):
    if self.board[row][col] != 0:
        return False, []
    # 各个方向是否可以翻转
    flag = False
    valid_direction = []
    for i in range(8):
        if 0 <= row + self.dr[i] < 8 and 0 <= col + self.dc[i] < 8:
            r = row + self.dr[i]
            c = col + self.dc[i]
            # 第一次调用时，确保位置是对方棋子
            if self.board[r][c] == -player:
                # 如果此方向可以翻转，则置 flag 为 True，valid_direction 加上 i
                if self.valid_search(r, c, i, player):
                    flag = True
                    valid_direction.append(i)
    return flag, valid_direction



# 判断落子合法性，注意第一次调用时，确保当前位置是对方棋子
def valid_search(self, row, col, direction, player):
    # 判断下个位置是否越界
    if 0 <= row + self.dr[direction] < 8 and 0 <= col + self.dc[direction] < 8:
        # 下个位置
        row = row + self.dr[direction]
        col = col + self.dc[direction]
        # 判断下个位置是否是对方棋子，如果是，判断下下个位置
        if self.board[row][col] == -player:
            return self.valid_search(row, col, direction, player)
    # 如果下个位置是己方棋子，则可行
    if self.board[row][col] == player:
        return True
    # 否则落子不合法
    return False
```

$MiniMax$ 搜索（带$AlphaBeta$剪枝）

```
def minimax_search_with_abcut(self, ceng, max_ceng, player, show=False, max_alpha=float('-inf'),
    # ceng 为偶数时 Max 玩家下子
    if ceng == 60:  # 棋局结束
        return self.final_score(), []
    if ceng == max_ceng:  # 搜到最大层
        return self.score(player), []
    pos = []  # 解的位置
    res = []  # 解的分数
    counter = 0
    for i in range(8):  # 行
        for j in range(8):  # 列
            if self.is_valid(i, j, player)[0]:  # 位置有效
                [directions, steps] = self.drops(i, j, player)[1:3]  # 下子
                res.append(self.minimax_search_with_abcut(ceng + 1, max_ceng, -player, show, max
                pos.append([i, j])
                self.withdraw(i, j, directions=directions, steps=steps)  # 撤回，回溯
                if player == 1:  # Max玩家
                    if res[counter] >= min_beta:  # Max玩家分数 大于 祖先min节点 的 最小beta
                        break  # 剪枝
                    elif res[counter] > max_alpha:  # 维护 最大alpha
                        max_alpha = res[counter]
                elif player == -1:  # Min玩家
                    if res[counter] <= max_alpha:  # Min玩家分数 小于 祖先max节点 的 最大alpha
                        break  # 剪枝
                    elif res[counter] < min_beta:  # 维护 最小beta
                        min_beta = res[counter]
                counter += 1
                # print(i, j, directions, steps)
                if show:
                    self.show_board(title='withdraw')
    # 没有位置可走
    if len(res) == 0:
        return self.minimax_search_with_abcut(ceng+1, max_ceng, -player, show)[0], []
    if player == 1:  # Max玩家
        return max(res), pos[res.index(max(res))]
    else:  # Min玩家
        return min(res), pos[res.index(min(res))]
```

遗传算法

```
# 交叉
def cross_over(pre_pop, p=0.75):
    arr1 = np.array(range(0, len(pre_pop)))
    arr2 = np.array(range(0, len(pre_pop)))
    np.random.shuffle(arr1)  # 随机打乱
    np.random.shuffle(arr2)
    cross_overed_pop = copy.deepcopy(pre_pop)  # 深拷贝
    for e in range(len(pre_pop)):
        fa1 = pre_pop[arr1[e]]
        fa2 = pre_pop[arr2[e]]
        child = copy.deepcopy(fa1)  # 深拷贝
        for i in range(4):
            for j in range(1 + i):
                if np.random.rand() < p:
                    child[i][j] = (fa1[i][j] + fa2[i][j]) / 2 + 1.5 * abs(fa1[i][j] - fa2[i][j])
        if np.random.rand() < p:
            child[4] = (fa1[4] + fa2[4]) / 2 + 1.5 * abs(fa1[4] - fa2[4]) * (np.random.rand() -
        if np.random.rand() < p:
            child[5] = (fa1[5] + fa2[5]) / 2 + 1.5 * abs(fa1[5] - fa2[5]) * (np.random.rand() -
        cross_overed_pop.append(child)
    return cross_overed_pop


# 变异
def mutation(pre_pop, p=0.1, r=50):
    for e in range(int(len(pre_pop)/2), len(pre_pop)): # 对每个策略
        for i in range(4):  # 对每个参数
            for j in range(1+i):
                if np.random.rand() < p:
                    pre_pop[e][i][j] += random.randint(-r, r)
        if np.random.rand() < p:
            pre_pop[e][4] += random.randint(-r, r)
        if np.random.rand() < p:
            pre_pop[e][5] += random.randint(-r, r)
    return pre_pop


# 锦标赛
def selection(pre_pop):
    size = len(pre_pop)
    now_pop = []
    arr = np.array(range(0, size))  # 生成对打的对手
    np.random.shuffle(arr)  # 随机打乱
    process = []
    que = multiprocessing.Queue()  # 所有进程往里面塞消息
    for i in range(int(size/2)):
        process.append(multiprocessing.Process(target=race, args=(pre_pop[i], pre_pop[int(size/2
        process[i].start()  # 进程开始工作
```

```
    for i in range(int(size / 2)):
        process[i].join()  # 等待所有进程工作完成（栅栏）
    now_pop = [que.get() for p in process]
    return now_pop


# 两个 weight 对打
def race(weight1, weight2, que):
    chess1 = Chess(weight1)
    chess2 = Chess(weight2)
    for step in range(30):
        [res, pos] = chess1.minimax_search_with_abcut(step, step + 3, 1, show=False)
        if len(pos) != 0:
            chess1.drops(pos[0], pos[1], 1)
            chess2.drops(pos[0], pos[1], 1)
        [res, pos] = chess2.minimax_search_with_abcut(step, step + 3, -1, show=False)
        if len(pos) != 0:
            chess1.drops(pos[0], pos[1], -1)
            chess2.drops(pos[0], pos[1], -1)
    final_score = chess1.final_score()
    # 赢家加入 now_pop
    if final_score >= 0:
        que.put(weight1)
    else:
        que.put(weight2)


if __name__ == '__main__':
    epochs = 200  # 迭代代数
    pop_size = 64  # 种群大小
    init_time = 3  # 初始竞争次数
    pop = []
    for s in range(pop_size * (2 ** init_time)):
        pop.append(随机)  # 随机初始化
    for s in range(init_time):  # 竞争初始化
        pop = selection(pop)
    print(pop)
    start = time.time()
    for epoch in range(epochs):
        new_pop = cross_over(pop, p=0.75)  # 交叉
        new_pop = mutation(new_pop, p=0.1, r=75)  # 变异
        new_pop = selection(new_pop)  # 选择
        pop = new_pop
        print('epoch', epoch, ': ', new_pop[0])

    while len(new_pop) > 1:
        new_pop = selection(new_pop)  # 最优个体
    print(new_pop)
```

模拟退火算法

```
def new_weight_generate(old_weight):
    n_weight = old_weight
    for i in range(4):  # 对每个参数
        for j in range(i+1):   # 对每个参数
            if np.random.rand() < 0.2:
                n_weight[i][j] += random.randint(-50, 50)  # 扰动
    if np.random.rand() < 0.2:
        n_weight[4] += random.randint(-50, 50)  # 扰动
    if np.random.rand() < 0.2:
        n_weight[5] += random.randint(-50, 50)  # 扰动
    return n_weight


if __name__ == '__main__':
    T = 100
    inner_loop = 10
    weight = [[704.1267941755075], [-42.96383722364534, -167.91415998636808], [183.1490661265754
    while T > 0.01:
        for loop in range(inner_loop):
            new_weight = new_weight_generate(weight)
            chess1 = Chess(new_weight)  # 新解
            chess2 = Chess(weight)  # 旧解
            for epoch in range(30):  # 对战
                [res, pos] = chess1.minimax_search_with_abcut(epoch, epoch + 3, 1, show=False)
                if len(pos) != 0:  # 无子可下
                    chess1.drops(pos[0], pos[1], 1)
                    chess2.drops(pos[0], pos[1], 1)

                [res, pos] = chess2.minimax_search_with_abcut(epoch, epoch + 3, -1, show=False)
                if len(pos) != 0:  # 无子可下
                    chess1.drops(pos[0], pos[1], -1)
                    chess2.drops(pos[0], pos[1], -1)

            final_score = chess1.final_score()  # 最终得分
            if final_score > 0:  # 新解比旧解好，接受新解
                weight = new_weight
            elif math.exp(-final_score / T > np.random.rand()):  # 新解比旧解差，以一定概率接受新解
                weight = new_weight
            print('T:', T, ' inner_loop:', loop)
            print('score:', final_score/10000)
            print('weight:', weight, '\n')
        T *= 0.98  # 退温
```

4、创新点&优化

遗传算法 & 退火算法 的比较

模拟退火算法

本实验的模拟退火算法，是新解和旧解的迭代过程。在退火的过程中，始终只有两个解在对弈，这存在许多问题：

1. 当初始解较差时，模拟退火算法**收敛很慢**，而且很有可能收敛不到很好的解。这是因为旧解很差，即使新解战胜了旧解，也**不能保证新解就是对棋局局势的更好的理解**。
2. 退火算法**可能不收敛**：比如在猜拳的博弈中，旧解是始终出石头，新解是始终出布，则新解战胜旧解，选择新解；第二轮退火中新解变为始终出剪刀，第三轮又变回始终出石头了。在棋类游戏中，有可能也存在像石头剪刀布这样**三种相互克制的策略**，这样的话退火算法有可能会在这三个策略之间循环，导致收敛不了。

遗传算法

本实验中的模拟退火算法就像一个棋手自己跟自己对战，用自己本来的套路，和自己新想出来的套路对战，如果新想出来的套路赢了，就选新套路，否则按一定概率选择新套路。经实验得出：这个**闭门造车**的过程得不出很好的结果。
遗传算法中，种群具有一定的规模，代表着很多**不同的策略**（对棋局/局势的理解），像很多棋手两两进行淘汰制对决一样，通过**锦标赛**的方法对种群进行选择，可以**收敛到较好的结果**。

遗传算法中的实现细节

要优化的参数

遗传算法**要优化的是评估函数中的参数**，包括各位置的权重，子数的权重，行动力的权重等等。
注意各位置的权重：因为棋盘是对称的，所以 64 个位置中，只需要优化 10 个位置的权重，其他位置的权重可以通过对称得到。

种群初始化

随机初始化种群，在初始种群中举行 n 次**锦标赛**，每次淘汰掉一半的个体，这样可以**提高初代种群的质量**，**加快算法收敛**的同时**使得收敛的结果更好**。

参数设置

遗传算法的参数包括遗传代数，种群规模，交叉概率，变异概率等等。设置遗传代数为 200，种群规模为 64，交叉概率为 0.75，变异概率为 0.1。

1. 遗传代数太小，算法不容易收敛，种群还没有成熟；遗传代数太大，算法已经熟练或者种群过于早熟不可能再收敛，继续进化没有意义，只会增加时间开支和资源浪费
2. 群体规模太小，很明显会出现近亲交配，产生病态基因；同时，遗传算子存在随机误差（模式采样误差），妨碍小群体中有效模式的正确传播，使得种群进化不能按照模式定理产生所预期的期望数量。群体规模太小，很明显会出现近亲交配，产生病态基因。
3. 变异概率太小，种群的多样性下降太快，容易导致有效基因的迅速丢失且不容易修补；变异概率太大，尽管种群的多样性可以得到保证，但是高阶模式被破坏的概率也随之增大。
4. 与变异概率类似，交叉概率太大容易破坏已有的有利模式，随机性增大，容易错失最优个体；交叉概率太小不能有效更新种群

多进程加速

通过锦标赛的方法模仿自然选择时，将个体两两配对进行博弈，胜者加入新种群。每对个体的**多场博弈之间是相互独立的，可以并行进行**。
通过 python 的 GIL 使用多线程的时候，同一时间只能有一个线程在 CPU 上运行，而且是单个 CPU 上运行。如果想要充分地使用多核 CPU 的资源，在 python 中大部分情况需要使用多进程。
因此使用 **multiprocessing** 库的**多进程函数**，对各个锦标赛进行实现。

UI 设计

通过 $matplotlib$ 库中的 plt 函数展现棋盘

三、实验结果与分析

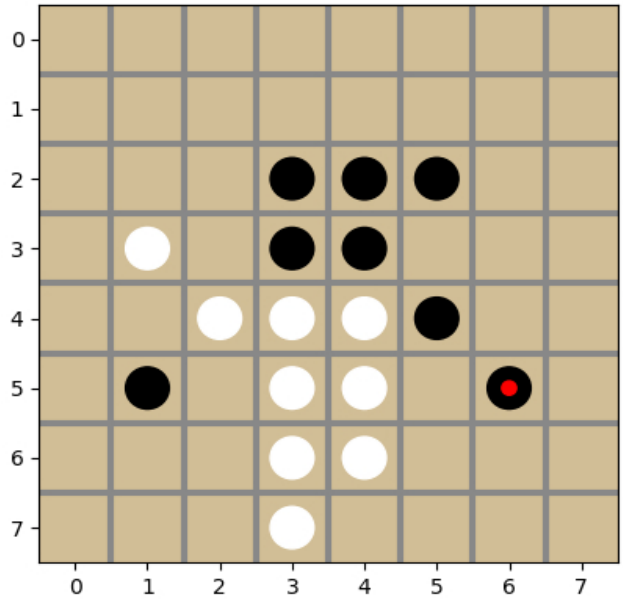
1、实验结果展示示例

权重

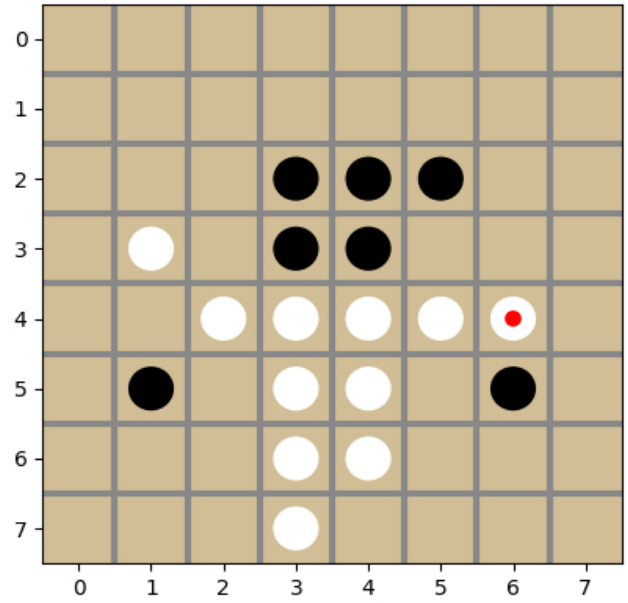
[[293], [-18, -67], [39, -62, -10], [40, -4, 13, 1], 72]
角的权重为 293，角临接的位置权重是负值。因为角的位置很重要，角可以形成很多稳定子，所以角的权重很大。而且角旁边的位置下了之后，容易被对方占到角，所以权值为负。

对局示例如下：（随便下的，只为了演示）
我执黑，电脑执白，我先手后手都打不过电脑

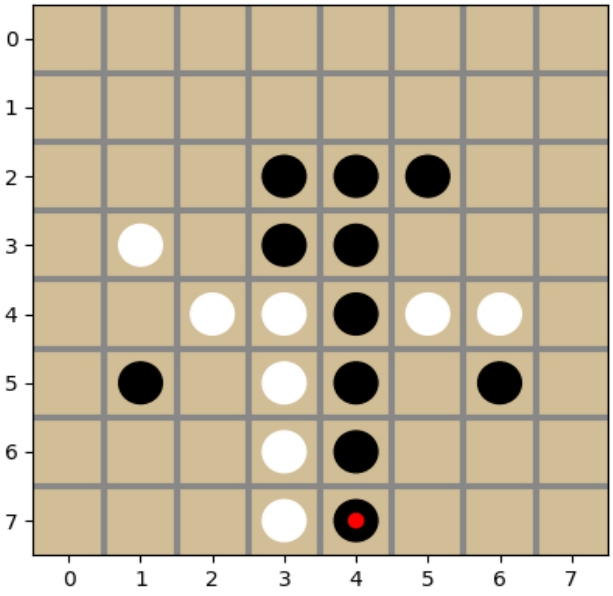
score: -1



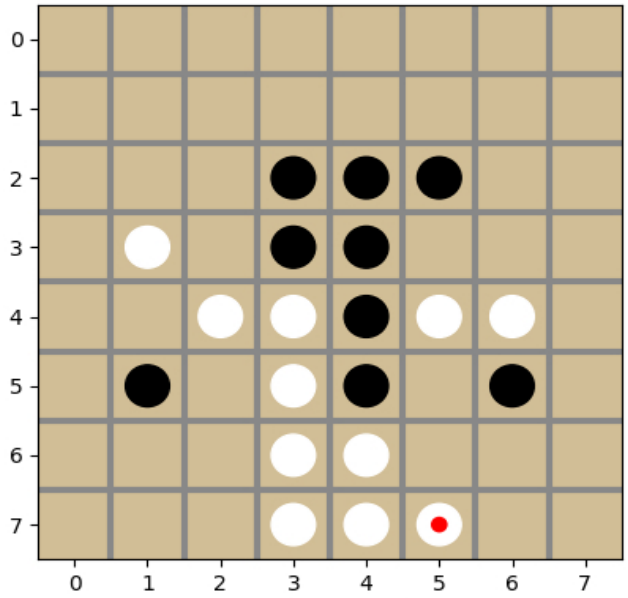
score: -4



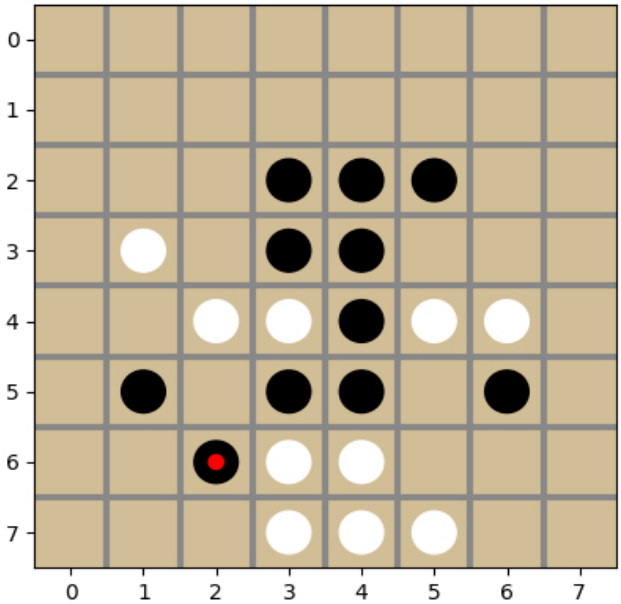
score: 3



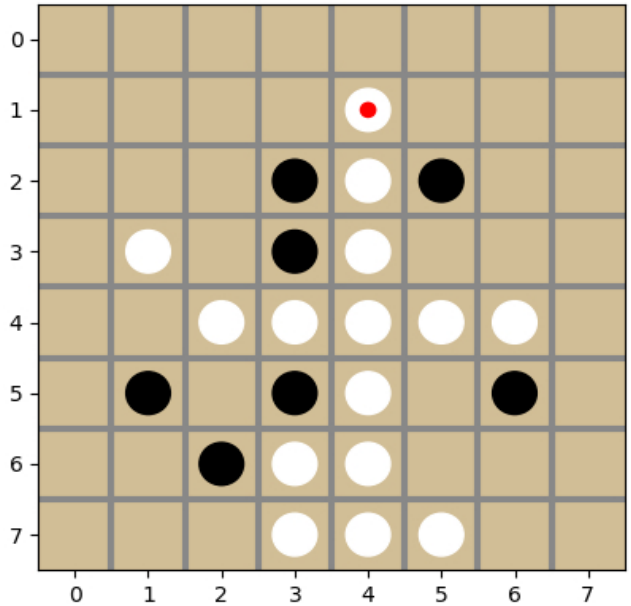
score: -2



score: 1



score: -8



2、运行时间分析

在 24 个 6 核 cpu 的服务器上跑遗传算法，通过多进程来并行执行锦标赛，可以使得遗传算法加速约 60 倍 （种群规模大小为 64，交叉后变为 128，锦标赛需打 64 场）
$alpha\_beta$ 剪枝可以使得程序平均运行时间缩短到原来的 1/3
$MiniMax$ 搜 3 层可以在 0.1 秒内出结果，$MiniMax$ 搜 5 层可以在3秒内出结果

四、参考资料

https://www.zhihu.com/question/25271618 黑白棋技巧
https://zhuanlan.zhihu.com/p/35121997 黑白棋局面估计
https://blog.csdn.net/cq947820606/article/details/78653070 判断落子合法性
https://blog.csdn.net/carlyll/article/details/105900317 遗传算法参数调整
https://blog.csdn.net/Big_Head_/article/details/78966363 plt获取鼠标位置
https://zhuanlan.zhihu.com/p/194349143 python多线程