中山大學
SUN YAT-SEN UNIVERSITY

《人工智能实验》

实验报告

（期末 proj-黑白棋）

学 院 名 称 ： 数据科学与计算机学院

专业（班级）： 17 计算机科学与技术

姓名学号 ： 17341067 江金昱　16327143 仲逊

一、算法原理

本次实验中我们实现了结合遗传算法的带有$alpha-beta$剪枝的$minimax$搜索策略。其中遗传算法用于确定各个评估函数返回的评估值之间的合理权重。

1.1 minimax 搜索策略

在博弈过程中，有两个玩家，双方轮流更新状态，每个玩家作出的决策都受另一个玩家的影响。类似于黑白棋这种双方博弈的问题，就非常适合用博弈树的相关算法解决。而$minimax$策略就是其中一种解决双方博弈的问题。

在博弈中，假设对方总时能作出最优的行动，己方总是作出能最小化对方获得的利益的行动。我们通过最小化对方的收益可以最大化自己的收益。在博弈树的叶结点（终止状态）存在一个效益值，表示到达此节点根节点可获得的收益。根节点为max节点，表示总是作出使自己效益最大化的决策，即：

$$U(n)=min\{U(c):c是n的子节点\}\ 当n是min节点时$$

根节点的子节点为min节点，表示总是作出使得对方效益最小化的决策，即

$$U(n)=max\{U(c):c是n的子节点\}\ 当n是max节点时$$

max与min节点随层数交替出现。

1.2 alpha-beta 剪枝

由于$Minimax$搜索需要搜索的游戏状态数目会随着博弈的进行呈指数增长，为了减小搜索的时间，$alpha-beta$剪枝应运而生。$alpha-beta$剪枝可以剪掉不可能影响决策的分支，减小部分搜索树。极大节点需要选择能使得自己效益最高的路径，最高效益记为α，极小节点需要选择能使得对方效益最低的路径，最低效益记为β。当算法运行到某个节点，观察到$\alpha \geq \beta$的时候，我们就可以停止搜索该节点的子节点。分两种情况解释，如果该节点为一个$mini$节点，那么其父节点一定不会选择走向该子节点，因为该子节点会使得最终max的收益变小。如果该节点为一个max节点，那么父结点一定不会选择走向该子节点，因为走向该节点会使得max最终收益变大。

1.3 遗传算法

遗传算法是模拟自然选择和生物进化的一种计算模型，能够过模拟自然进化过程以此来搜索最优解。主要过程包括染色体的选择，交叉，变异，复制等等。下面将分步骤介绍遗传算法的原理和我们如何在黑白棋问题中应用遗传算法。

- 建模

遗传算法中的一个染色体可以用一个向量（或者矩阵）来表示，我们首先需要对问题进行建模，问题的一个解即为一条染色体，这个向量（染色体）的每个元素就是这个染色体的基因。在黑白棋问题中，我们选定的染色体代表的是每个评估函数的权重的向量，对他们加权求和才能得到最后的评估函数值。

- 选择适应度函数

遗传算法在运行的过程中会进行迭代，每次迭代都会生成若干条染色体。我们需要评判这些染色体的适应度，然后将适应度较低的染色体淘汰掉，只保留适应度较高的染色体，这个评估标准就称为适应度函数。适应度函数的选择十分重要，它反映了我们使用何种指标来评估这个解的优秀程度。

在黑白棋策略中，我们使用的适应度函数是对评估函数加权求和得到的值，即使得当前棋局分数最大的评估值，这个值越大，则表明该染色体（权重向量）越优秀。

- 交叉

遗传算法每一次迭代被称为一次进化，交叉就是使得每次迭代都能够生成同样数量的新染色体的方法，即两个染色体的某一相同位置处被切断，前后两串分别交叉组合形成两个新的染色体，如下示例，父母染色体从“|”处被切断，拼接生成子染色体：

$$\left.\begin{matrix}\text{父：}\ [a,b,|\,c,d,e,f,g]\\ \text{母：}\ [h,i,|\,j,k,l,m,n]\end{matrix}\right\}\rightarrow \text{子：}\ [a,b,|\,j,k,l,m,n]\ \ [h,i,|\,c,d,e,f,g]$$

- 变异

交叉虽然使得每次进化留下优良的基因，但它只对原有的基因集进行了选择，仅仅交换了组合顺序，这只能保证经过一定次数的迭代后，结果接近于局部最优解，而达不到全局最优解，变异就是为了解决这一问题，可以理解为基因突变，当通过交叉生成了一条新的染色体后，我们在新染色体上随机选择若干个基因修改为随机值，这样就引入了新的基因，突破了当前搜索的限制，更有利于寻找到全局最优解。

- 复制

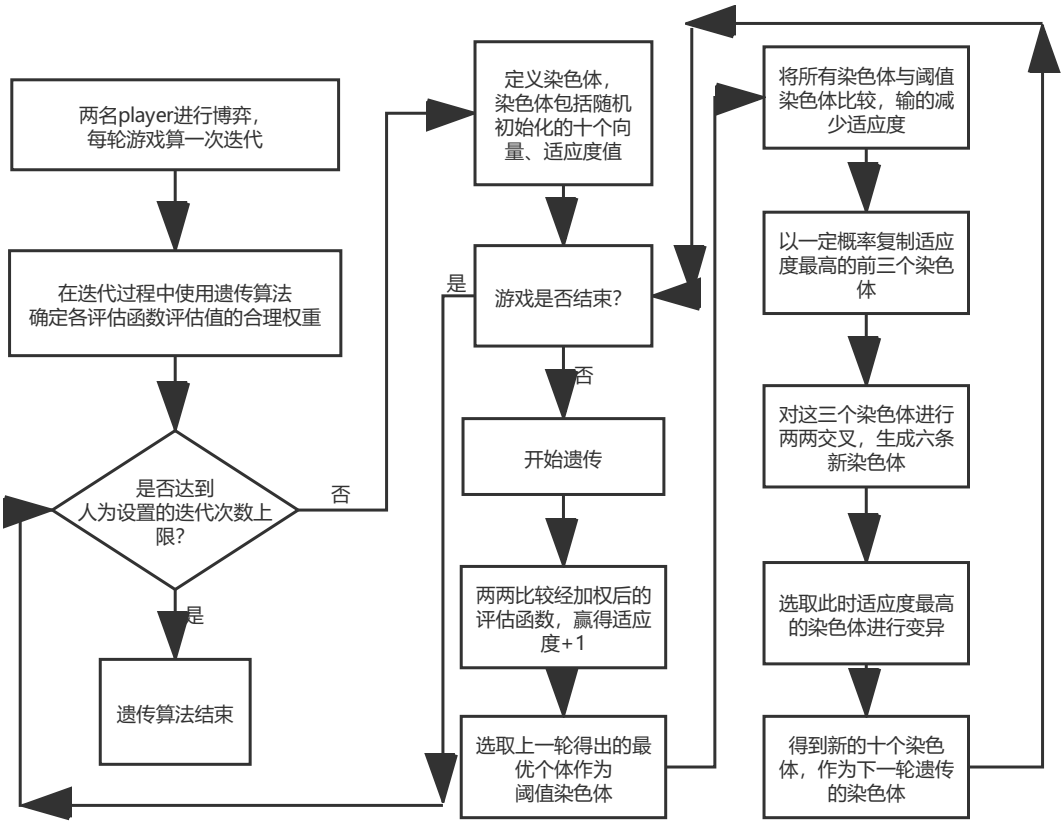
在遗传算法的进化过程中，为了保留上一代的优秀染色体，需将上一代适应度最高的几条染色体直接复制给下一代。设每次进化生成N条染色体，则每次进化中，通过交叉方式需要生成$N-M$条染色体，剩余M条染色体通过复制上一代适应度最高的M条染色体得到。

用数学语言描述即，因为我们不能保证每次交叉变异得到的新权重向量是更优的，因此保留上一代最优的几条权重向量能够防止局部最优解的产生。

➢ 阈值染色体

为了加快收敛的速度，尽快选出优秀的个体，遗传算法还可以选取每一轮遗传中适应度最大的染色体作为阈值染色体，在下一轮遗传中，要比较新生成的染色体与该阈值染色体得到的效用值哪个比较大，将效用值比阈值染色体小的染色体的适应度减小，以增加改染色体的淘汰概率（因为该染色体肯定不是最优的）

1.4 遗传算法流程图



二、实现之评价函数

2.1 给 8X8 的棋盘赋予分数，不同落子点的分数不同。

以下给出我们在对弈中设定的权重。

```
board_weights=[
    [400, -30, 11, 8, 8, 11, -30, 400],
    [-30, -70, -4, 1, 1, -4, -70, -30],
    [11, -4, 2, 2, 2, 2, -4, 11],
    [8, 1, 2, -3, -3, 2, 1, 8],
    [8, 1, 2, -3, -3, 2, 1, 8],
```

```
    [11, -4, 2, 2, 2, 2, -4, 11],
    [-30, -70, -4, 1, 1, -4, -70, -30],
    [400, -30, 11, 8, 8, 11, -30, 400]
]
```

当我们在黑白棋中，占据左上角、右上角、左下角、右下角的落子点时，我们赢得概率较大，因为此时该落子点得棋子不能被变更颜色，因此可以被赋予较大得权重。而在四个边角的周围 3 个点，假如我们占据了这 3 个点，那么对方就极有可能占据边角的点。因此这三个点可以被赋予负的权重。

2.2 根据黑白棋子的比例计算分数

由于最终黑白棋判定胜负的依据是各个棋子的数量，因此我们将黑白棋子比例作为评估的依据之一。当黑棋数目大于白棋数目时，对于黑棋来说，其效用值为正。当黑棋数目大于白棋数目时，对于白棋来说，其效用值为负。白棋亦然。最终，我们将比例乘以 100 作为最终的效用值。如下所示：

```
# 根据黑白棋子的比例进行计算分数
if mystonecount > opstonecount:
    rateeval = 100.0 * mystonecount / (mystonecount + opstonecount)
elif mystonecount < opstonecount:
    rateeval = -100.0 * opstonecount / (mystonecount + opstonecount)
else:
    rateeval = 0
```

2.3 根据四个角周围的棋子个数来评估

如果对方占据了棋盘四个角周围三个点的棋子，那么我方就有极大的概率占得四个角，因此赢得概率较大，因此我们计算四个角周围三个点我方棋子和对方棋子个数，再乘以一定比例值作为该评估函数得效用值，实现如下：

```
# 看四个角的周围是否有黑白棋
# mynear 表示四个角的周围的 color 颜色的棋子数
# oppnear 表示四个角的周围 opponent_color 颜色的棋子数
# corner_pos 存储棋盘四个角的位置
# direction 表示八个方向
mynear = 0
oppnear = 0
for i in range(4):
    x = corner_pos[i][0]
    y = corner_pos[i][1]
```

```
    if board2.board[x][y] == 0:
        for j in range(8):
            nx = x + direction[j][0]
            ny = y + direction[j][1]
            if 0 <= nx < 8 and 0 <= ny < 8:
                if board2.board[nx][ny] == color:
                    mynear += 1
                elif board2.board[nx][ny] == opponent_color:
                    oppnear += 1
neareval = -24.5 * (mynear - oppnear)
```

2.4 根据我方和对方可下棋子的位置来评估

可直观地理解到，当我们可以下的点多的时候，我们拥有更多制约对手的机会，因此可将我方和对方可下棋子的位置数目来评估。

```
# 通过我方和对方可下的地方的数量的比值来估计
# 我方可以下的地方的数量
mymove = len(board2.get_valid_moves(color))
# 对方可以下的地方的数量
opmove = len(board2.get_valid_moves(opponent_color))
if mymove == 0:
    moveeval = -450
elif opmove == 0:
    moveeval = 150
elif mymove > opmove:
    moveeval = (100.0 * mymove) / (mymove + opmove)
elif mymove < opmove:
    moveeval = -100.0 * opmove / (mymove + opmove)
else:
    moveeval = 0
```

2.5 根据该棋子是否可能被替代来评估

如果一枚棋子八个方向都占满了棋子，那么该枚棋子就不可能被改变，我们就认为该枚棋子时“稳定”的，稳定的棋子越多，我们赢得概率就越大，效用值也就越大。

```
# 某个棋子的八个方向是否被占满了棋子
for i in range(8):
    for j in range(8):
        # is_stable 判断该枚棋子是否稳定
        if board2.board[i][j] != 0 and is_stable(board2.board, i, j):
            if board2.board[i][j] == color:
                mystable += 1
```

```
            else:
                oppstable += 1
stableeval = 12.5 * (mystable - oppstable)
```

2.6 根据边界的棋子数目来评估

当占据边界的时候，该枚拥有较小的概率被改变颜色，且有较大概率借助该枚棋子去改变非边界棋子的颜色，所以我们也应该将边界棋子的数目纳入考虑范围。

```
# 我方边界点数量
myside = 0
# 对方边界点数量
opside = 0
# 四个角的我方棋子数
mycorner = 0
# 四个角对方棋子数
opcorner = 0
for i in range(4):
    if board2.board[corner_pos[i][0]][corner_pos[i][1]] == color:
        mycorner += 1
        for j in range(8):
            # 沿着列
            if board2.board[corner_pos[i][0]][j] == color:
                myside += sideVal[j]
            else:
                break
        for j in range(8):
            # 沿着行
            if board2.board[j][corner_pos[i][1]] == color:
                myside += sideVal[j]
            else:
                break
    elif board2.board[corner_pos[i][0]][corner_pos[i][1]] == opponent_color:
        opcorner += 1
        for j in range(8):
            # 沿着列
            if board2.board[corner_pos[i][0]][j] == opponent_color:
                opside += sideVal[j]
            else:
                break
        for j in range(8):
            # 沿着行
            if board2.board[j][corner_pos[i][1]] == opponent_color:
                opside += sideVal[j]
```

```
            else:
                break
sidestableeval = 2.5 * (myside - opside)
cornereval = 25 * (mycorner - opcorner)
```

三、实现之遗传算法

我们的目标是使用遗传算法来确定各个评估函数返回的效用值之间的权重。

我们采取人为选定迭代次数（即下多少盘棋）的方式作为迭代结束的标志。

在每次评估棋面的时候都调用遗传算法来调整权重。

适应度的评估：以染色体的权重乘以各个评估函数返回的效用值的加权和作为评估依据。

1. 定义染色体

我们使用一个类来封装染色体，染色体的属性有归一化后的权重、适应度

```
# 定义染色体
class chromosome:
    def __init__(self,weight_len):
        # 适应度
        self.adapt=0
        # 向量
        self.weight=[random.random() for i in range(weight_len)]
        # 归一化
        sum_weight=sum(self.weight)
        self.weight=[weight/sum_weight for weight in self.weight]
```

2. 随机生成 10 条染色体作为初始染色体

```
# 随机生成 10 条染色体,并初始化适应度为 0
randweight=[chromosome(weight_len) for i in range(chromo_num)]
```

在定义好染色体后，接下来就可以进行 N 轮遗传了，下面将展示每轮遗传的操作

3. 两两染色体比较得分

使用我们得出的评估函数的效用值乘以染色体对应的权重来作为评估适应度的依据。值大的染色体适应度+1

```
# 两两进行比较得分，赢的适应度+1
for i in range(chromo_num):
```

```
    for j in range(i+1,chromo_num):
        res1=sum([randweight[i].weight[k]*dim[k] for k in range(weight_len)])
        res2=sum([randweight[j].weight[k]*dim[k] for k in range(weight_len)])
        if res1 > res2:
            randweight[i].adapt += 1
        else:
            randweight[j].adapt += 1
```

4. 为了加快收敛速度，引入“阈值染色体”

选取每一轮遗传中适应度最大的染色体作为阈值染色体，在下一轮遗传中，要比较新生成的染色体与该阈值染色体得到的效用值哪个比较大，将效用值比阈值染色体小的染色体的适应度减小，以增加改染色体的淘汰概率（因为该染色体肯定不是最优的）。

```
# 把比阈值染色体差的染色体的适应度减少
for j in range(chromo_num):
    res1=sum([limit_chromo.weight[k]*dim[k] for k in range(weight_len)])
    res2=sum([randweight[j].weight[k] * dim[k] for k in range(weight_len)])
    if res1>res2:
        randweight[j].adapt-=2
```

5. 根据适应度大小对染色体进行排序

```
# 根据适应度大小从大到小排序
randweight.sort(key=lambda e:e.adapt,reverse=True)
```

6.复制

选取染色体最大的前 3 个进行复制。为了避免在遗传过程中造成局部最优解，因此我们将设定复制的概率，以避免局部最优解的情况。

```
# 选择前 copy_num 个进行复制
copy_num=3
# 以 90%的概率复制
if random.random()<=0.9:
    for i in range(copy_num):
        chromo=randweight[i]
        chromo.adapt=0
        new_randweight.append(chromo)
else:
    # 否则随机生成 copy_num 个
    for i in range(copy_num):
        new_randweight.append(chromosome(weight_len))
```

7. 交叉

选择适应度最大的前四个进行交叉操作，即进行 12、13、14、23、24、34 交叉，这样可以得到 6 条染色体。每次交叉过程中，我们首先成一个随机数，表示交叉点，之后生成一个模 2 的随机数，表示两条染色体先截取哪一条。

```
# 交叉部分，12,13,14,23,24,34 交叉，产生交叉后新的 6 条染色体
for i in range(4):
    for j in range(i+1,4):
        tmp_chromo=chromosome(weight_len)
        tmp_chromo.weight=[]
        rand2=random.randint(1,2)
        tmpsum=0
        # 交叉点
        cut=random.randint(0,weight_len-1)
        for k in range(weight_len):
            if rand2==0:
                if k<=cut:
                    tmp_chromo.weight.append(randweight[i].weight[k])
                    tmpsum+=randweight[i].weight[k]
                else:
                    tmp_chromo.weight.append(randweight[j].weight[k])
                    tmpsum += randweight[j].weight[k]
            else:
                if k <= cut:
                    tmp_chromo.weight.append(randweight[j].weight[k])
                    tmpsum += randweight[j].weight[k]
                else:
                    tmp_chromo.weight.append(randweight[i].weight[k])
                    tmpsum += randweight[i].weight[k]
        for k in range(weight_len):
            tmp_chromo.weight[k]/=tmpsum
        new_randweight.append(tmp_chromo)
```

8. 变异

对适应度最大的染色体进行变异。变异方法为随机选择染色体上的一维权重替换为一个（0，1）之间的数，最后对变异后的染色体归一化

```
# 变异
tmp_chromo=randweight[0]
tmp_chromo.adapt=0
```

```
# 随机生成一个替换的数
replace_num=random.random()
# 随机生成一个替换的位置
replace_index=random.randint(0,6)
weight_sum=0
for k in range(weight_len):
    if k==replace_index:
        weight_sum+=replace_num
    else:
        weight_sum+=tmp_chromo.weight[k]
for k in range(weight_len):
    if k==replace_index:
        tmp_chromo.weight[k]=replace_num/weight_sum
    else:
        tmp_chromo.weight[k]/=weight_sum
new_randweight.append(tmp_chromo)
```

四、实现之 minimax 搜索策略

采用原理部分提到的带有$alpha-beta$剪枝的$minimax$搜索策略。代码部分如下:

这里采用了改进版的剪枝策略，即“负值极大算法”。注意下方代码标红部分，每次将返回的评分取负值，那么对于极小节点，我们同样也可以通过选取最大分数的方式来处理。(相当于分数取反之后，原来的最小值变成了现在的最大值)。对于极大节点，由于经历了两次取反，因此还是选取最大分数。这样做的效果是代码量更加精简。

```
def minimax(self, board, depth, player, opponent,
            alfa=-INFINITY, beta=INFINITY):
    bestmove=(0,0)
    bestChild = board
    # 到达了叶子节点
    if depth == 0:
        return ( self.evaluate(player,board), board, bestmove )
    for (child,move) in board.next_states(player):
        # 返回分数，最优子节点，最优分数
        score, newChild,_= self.minimax(
            child, board, depth - 1, opponent, player, -beta, -alfa)
        score = -score
        if score > alfa:
            alfa= score
            bestChild = child
            bestmove= move
        # 剪枝
```

```
        if beta <= alfa:
            break
    return self.evaluate(player, board), bestChild, bestmove
```

五、实验结果分析

1. 遗传算法生成权重展示

由于不同的评价函数对棋局的落棋的重要性不同，因此他们最终收敛时的权重也不同。以下展示了 10 次迭代中每个评估函数的权重以及最终接近收敛时的权重。

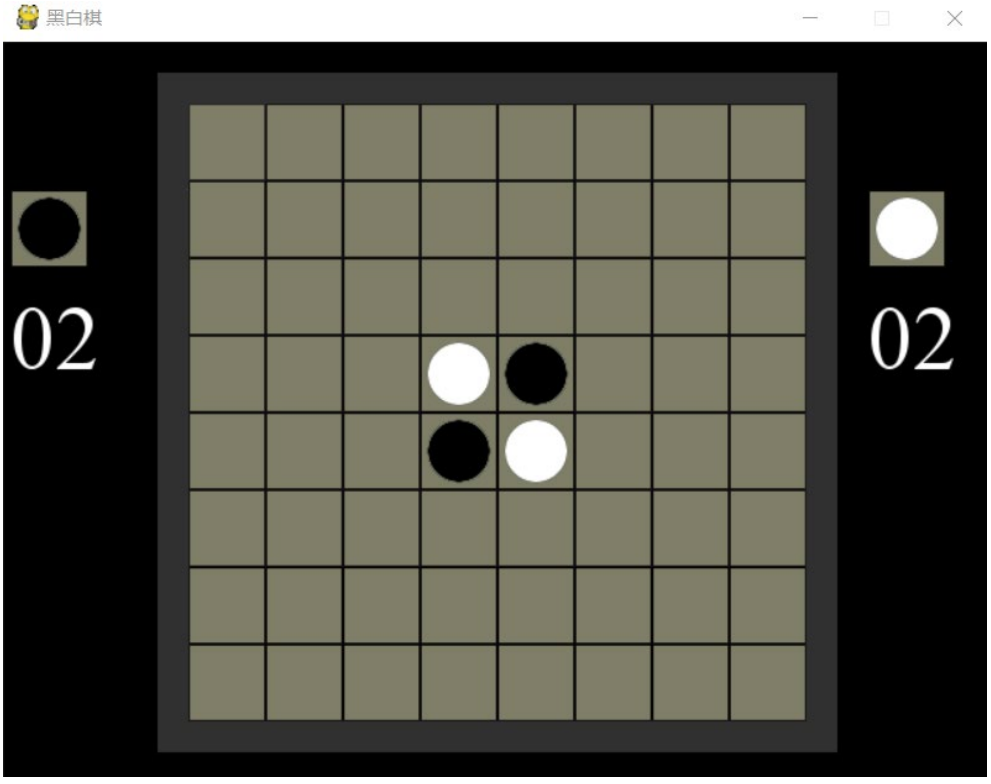
	e1	e2	e3	e4	e5	e6	e7
1	0.180024	0.158421	0.115637	0.148765	0.131673	0.132948	0.132531
2	0.191451	0.1672	0.131378	0.13384	0.131794	0.119331	0.125007
3	0.200374	0.165651	0.131179	0.130374	0.116644	0.127976	0.127801
4	0.194079	0.177424	0.126681	0.128813	0.120654	0.123919	0.12843
5	0.190731	0.172747	0.130905	0.1286	0.117932	0.126022	0.133063
6	0.190668	0.16921	0.132587	0.12919	0.119907	0.126999	0.13144
7	0.188288	0.171056	0.132032	0.128138	0.121742	0.126442	0.1323
8	0.187779	0.172125	0.132753	0.128525	0.121523	0.126784	0.13051
9	0.188166	0.172904	0.132355	0.127499	0.119512	0.129152	0.130412
10	0.186478	0.175266	0.130976	0.127504	0.117202	0.130896	0.131678
收敛	0.190777	0.173278	0.125869	0.118276	0.115697	0.135494	0.140609

2. 模拟对战（UI）展示

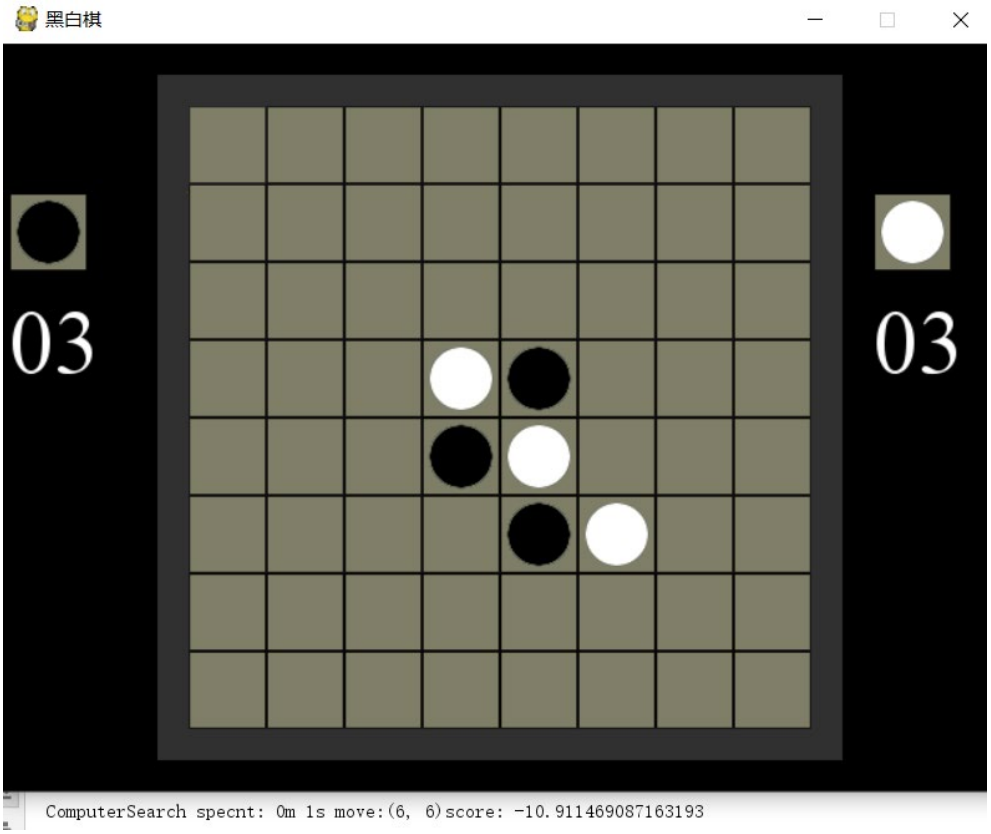
① 选择先手、后手，AI 搜索深度等（默认为玩家先手）

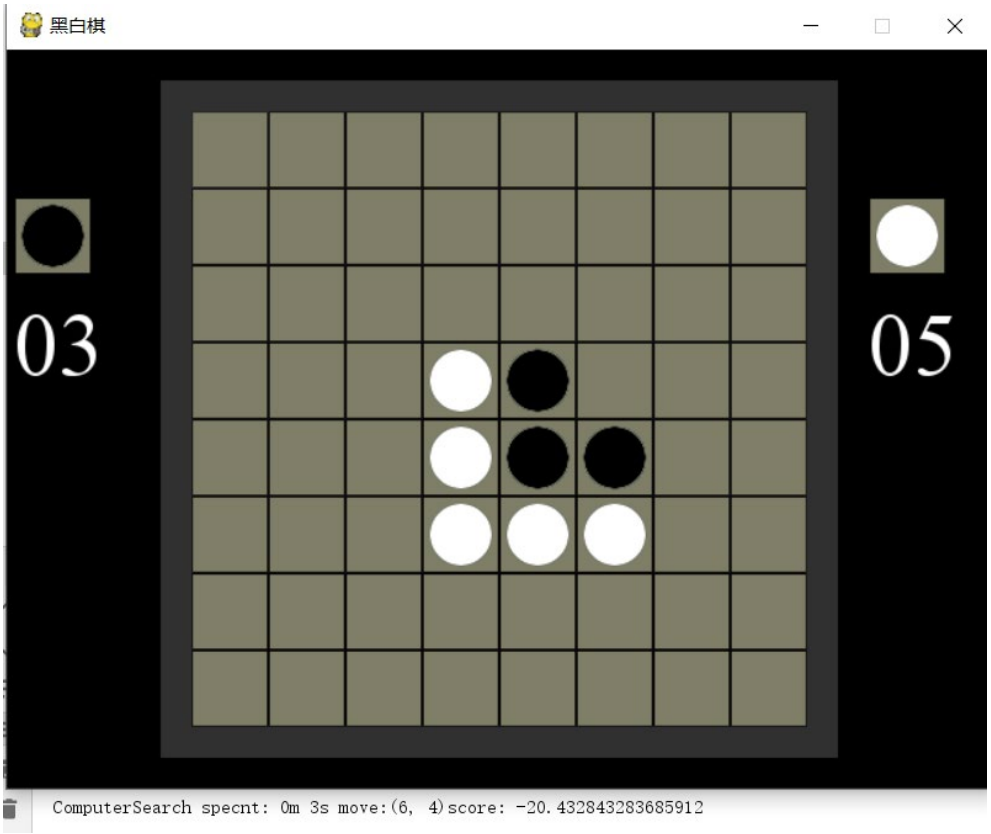

黑白棋
Othello
Start
First Player
Second Player
AI Level

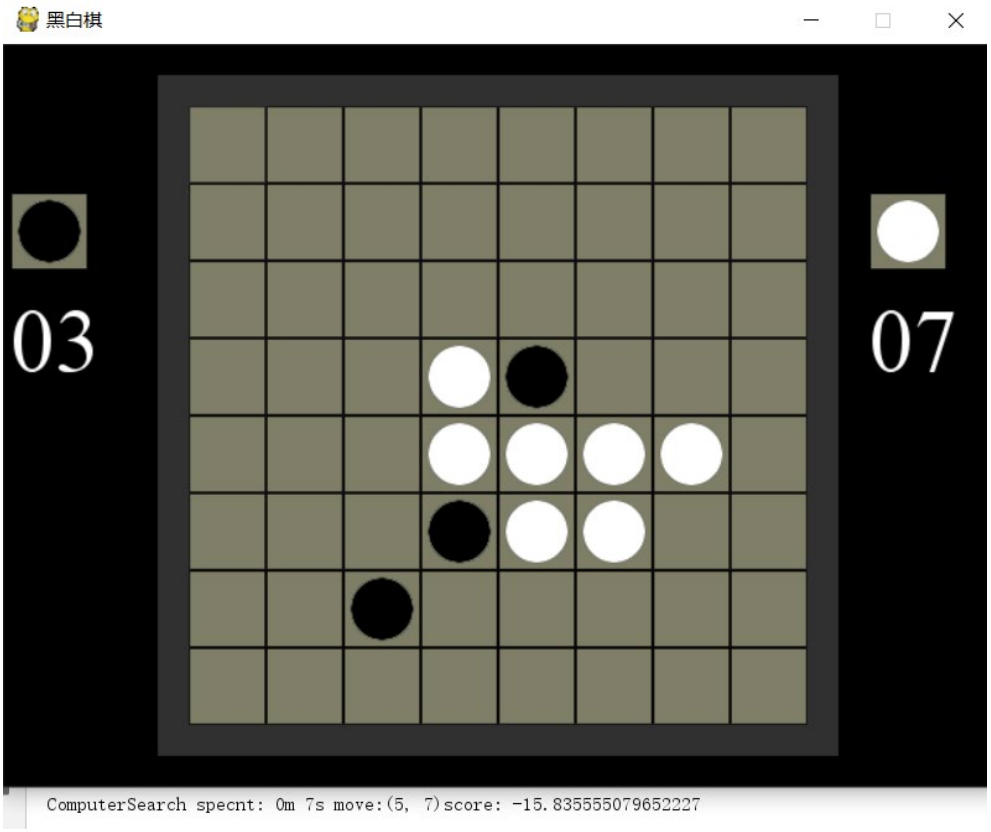
② 初始界面


黑白棋
02
02

③ 对战三轮截图（截图下方是分数）







3. 使用算法与随机落子进行博弈比较结果

对弈局数	10	20	30	50	100
（算法）胜率	1.0	1.0	1.0	0.98	0.97

可以看到，当我们与无脑的随机落子进行博弈的时候，我们的算法是行之有效的

六、小组分工及贡献

姓名	贡献
江金昱	遗传算法主要实现，编写主要代码，撰写部分报告
仲逊	确定遗传算法思路，模型实现与修改，撰写部分报告