《人工智能实验》

实验报告

期末 PROJECT

强化学习

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	17 级计算机科学与技术
组员 A 学号 ：	17341088
组员 A 姓名 ：	梁超
组员 B 学号 ：	17341178
组员 B 姓名 ：	薛伟豪

期末PROJECT：强化学习

1. 算法原理

1.1.遗传算法的主要思想

遗传算法（GA）是一种基于自然选择和基因遗传学原理的优化搜索算法。遗传算法仿照自然进化过程来实现对最优解的进行寻找的方法，它应用了达尔文进化论中的选择与变异的原理。先进行定向的选择，再进行不定向的变异，根据能体现目标的适应度函数，来体现每一代个体的适应度，从而进行选择操作，然后再进行遗传迭代，产生带有新的基因组合的个体。

遗传算法其本质是一种暴力求解的方法，利用计算机不知疲倦的高速运算，不断向最终解靠近，直到收敛出一个解。遗传算法不保证能得到最终解，也不保证所得到的解是最优解，但这是一种探索未知的好办法。当我们不知道对某一个问题如何求出最终的解，但只要能对最终可行的解进行编码，就可以用遗传算法进行探索。

1.2.遗传算法的具体步骤

遗传算法开始时先随机地产生一组问题的可行解，即第一代染色体。再根据适应度函数计算每个染色体的适应度，根据适应度对诸染色体进行选择、交换、变异等遗传操作，剔除适应度低，即性能不佳的染色体，留下适应度高，即性能优良的染色体，从而得到新的群体。

由于新群体的个体是上一代群体的“优秀者”，继承了上一代的优良性能，因而在总体上明显优于上一代。遗传算法按照上述过程反复迭代，向着更优的方向进化，直至满足某种预定的优化指标。一般而言，可以采用收敛或这人工限定迭代次数来使遗传算法结束。

下面对遗传算法的几个重要环节进行说明：

- **选择**

选择操作是从上一代种群中选择多对较优个体（这里所说的个体即为染色体），我们将一对较优个体称为一对父母，让父母们将它们的基因传递到下一代，直到下一代个体数量达到种群数量上限。

在选择操作前，将上一代种群中个体按照适应度从小到大进行排列，采用某种选择方法选出个体，每一个个体被选中的概率与其适应度函数值大小成正相关。需要指出的是，一些如轮盘赌这样的选择方法具有一定的随机性，在选择的过程中可能会丢掉较好的个体。所以可以使用精英机制，直接选择上一代的最优个体。

- **复制**

我们在选择部分选择出适应度较高的几个染色体，这些染色体的全部或部分可以直接进入下一次迭代的染色体组合,如下所示：

$$[a,b,c,d,e]_{old} \xrightarrow{\text{复制}} [a,b,c,d,e]_{new}$$

- **交叉**

自然界中，同源染色体的等位基因有一定概率会发生交叉互换。遗传算法中，两个待交叉的不同的染色体(即选择操作中选出的一对父母)，会按某种方式交换其部分“等位基因”，从而形成两个新的染色体。如下所示，染色体v_1的a_1基因和v_2的a_2基因发生了交叉互换：

$$\begin{matrix} v_{1-old} = [a_1,b_1,c_1,d_1,e_1] \\ v_{2-old} = [a_2,b_2,c_2,d_2,e_2] \end{matrix} \xrightarrow{\text{交叉}} \begin{matrix} v_{1-new} = [a_2,b_1,c_1,d_1,e_1] \\ v_{2-new} = [a_1,b_2,c_2,d_2,e_2] \end{matrix}$$

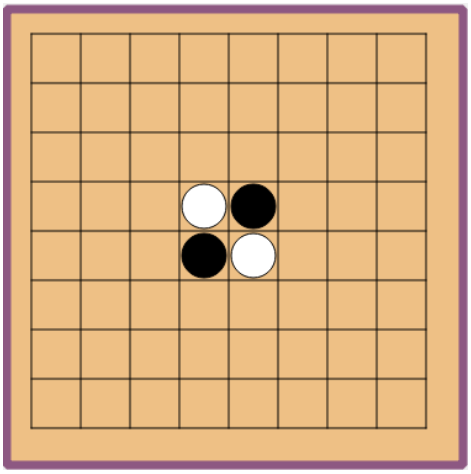
- **变异**

自然界中，染色体上的基因有可能发生变异。在遗传算法迭代的过程中，染色体上的基因会有一定的概率“变异”成该基因的等位基因，从而形成新的染色体。如下所示，染色体上的e_1基因变异成e_2基因：

$$[a,b,c,d,e_1]_{old} \xrightarrow{\text{变异}} [a,b,c,d,e_2]_{new}$$

1.3.遗传算法在黑白棋中的应用

1.3.1.黑白棋的规则



黑白棋又称为翻转棋。棋盘为8×8共64格。开局时，棋盘正中央的4格放置黑白相隔的4个棋子。通常黑子先行，双方轮流落子。当落子的点和棋盘上任意一枚己方棋子在一条线上(横、直、斜线皆可)，并且两个棋子之间夹的都是对方的棋子，就能将对方的棋子转变为己方棋子。如果在当前局势下，棋盘没有可以落子的地方，则继续由对方下子。如果双方皆不能下子，或棋盘已满，或一方棋子被另一方完全吃掉，则计算棋子个数，棋子个数多的一方胜。

1.3.2.评价函数

在本实验中，我将遗传算法与博弈树相结合。由于博弈树的叶子节点需要进行局面评估。为此，我们需要确定局面的评价函数。对于每一个局面，我们可以从多个方面进行评估，如：位置权重评估值、黑白子比例评估值等，将这些子评估值与遗传算法迭代得到的权重分别相乘，计算出每个局面的总评估值。具体细节如下：

- **位置权重评估值**

黑白棋和围棋相似，有“金边银角草肚皮”的说法。棋子在四个角的优势特别大，因为在四个角的棋子无法被翻转；而在四角的邻近位的优势最小，因为这些位置很容易让对方占角；四条边上的其他位置优势也比较大，因为迅速占边可以比较容易获得边界稳定子的优势；而在棋盘的中心位置优势较低，因为这些位置的棋子很容易被翻转。

棋盘的所有位置的权重值如下所示：

200	-100	20	10	10	20	-100	200
-100	-150	0	5	5	0	-150	-100
20	0	5	5	5	5	0	20
10	5	5	-5	-5	5	5	10
10	5	5	-5	-5	5	5	10
20	0	5	5	5	5	0	20
-100	-150	0	5	5	0	-150	-100
200	-100	20	10	10	20	-100	200

将本方所占位置的权重和减去对方所占位置的权重和，即可得到当前位置权重评估值。

- **黑白子比例评估值**

黑白子比例评估值基于此时棋面的黑白子棋子个数。

如果本方棋子多于对方棋子，则计算本方棋子占所有棋子的比例，乘以100得到结果；如果对方棋子多于本方棋子，则计算对方棋子占所有棋子的比例，乘以100后取负数得到结果；如果双方棋子数相同，则令黑白子比例评估值为0。

- **占角评估值**

近角评估值基于当前棋面双方所占四个角的个数。棋子在四个角的优势特别大，因为在四个角的棋子无法被翻转。因此在占角评估值的计算上，我们先用本方占角棋子数减去对方占角棋子，再乘以一个系数。

- **近角评估值**

近角评估值基于四个角周围最近的三个位置本方棋子和对方棋子的个数。如果本方棋子下在这些位置，那么很容易让对方棋子占领四个角。因此在近角评估值的计算上，我们先用本方近角棋子数减去对方近角棋子数，再乘以一个负系数。

- **行动力评估值**

行动力评估值基于当前棋面下双方可落子的位置数。如果本方可落子位置数越多、对方可落子位置数越少，则行动力估值越大。

具体地，如果本方没有地方落子，则设置一个很低的行动力估计值；如果对方没有地方落子，则设置一个很高的行动力估计值；如果本方可落子位置数大于对方，则将本方可落子位置数除以双方可落子位置数的总和，再乘以100得到结果；如果本方可落子位置数小于对方，则将对方可落子位置数除以双方可落子位置数的总和，再乘以-100得到结果；如果双方可落子位置数相同，则令行动力估计值为0。

- **稳定子评估值**

稳定子评估值基于当前棋面下双方稳定子的个数。所谓的稳定点，指的是该位置所在的各条线上，均被本方或者对方的棋子所占据，没有空子存在。

对于当前棋面，统计本方和对方稳定子的个数。将本方稳定子个数减去对方稳定子个数，再乘以一个系数，就可得到当前的稳定子评估值。

- **边界棋子数评估值**

边界棋子数评估值基于当前棋面双方边界棋子的个数。边界棋子指的是位于棋盘边界的棋子。总的来说，边界棋子越多，局面越好。

如果本方边界棋子多于对方，则计算本方边界棋子占所有边界棋子的比例，乘以100得到结果；如果对方边界棋子多于本方，则计算对方边界棋子占所有边界棋子的比例，乘以100后取负数得到结果；如果双方边界棋子数相同，则令边界棋子数评估值为0。

1.3.3.遗传算法在黑白棋中的实现方法

如上所说，在确定了评估函数之后，我们可以计算当前局面下评估函数的各个子评估值，而最终的评估值在本次实验中需要通过遗传算法计算得到。假设子评估值的个数为N，对于每一个棋面，算法的具体步骤如下：

1) 随机10个N维向量，对每个向量进行归一化处理。这些向量即为第一代染色体。同时初始化这10个向量的适应度为0。
2) 计算每一个向量与N个子评估值相乘的结果，将这10个向量两两进行比较，较大者适应

度+1。比较完成后，按照适应度大小对十个向量进行排序。

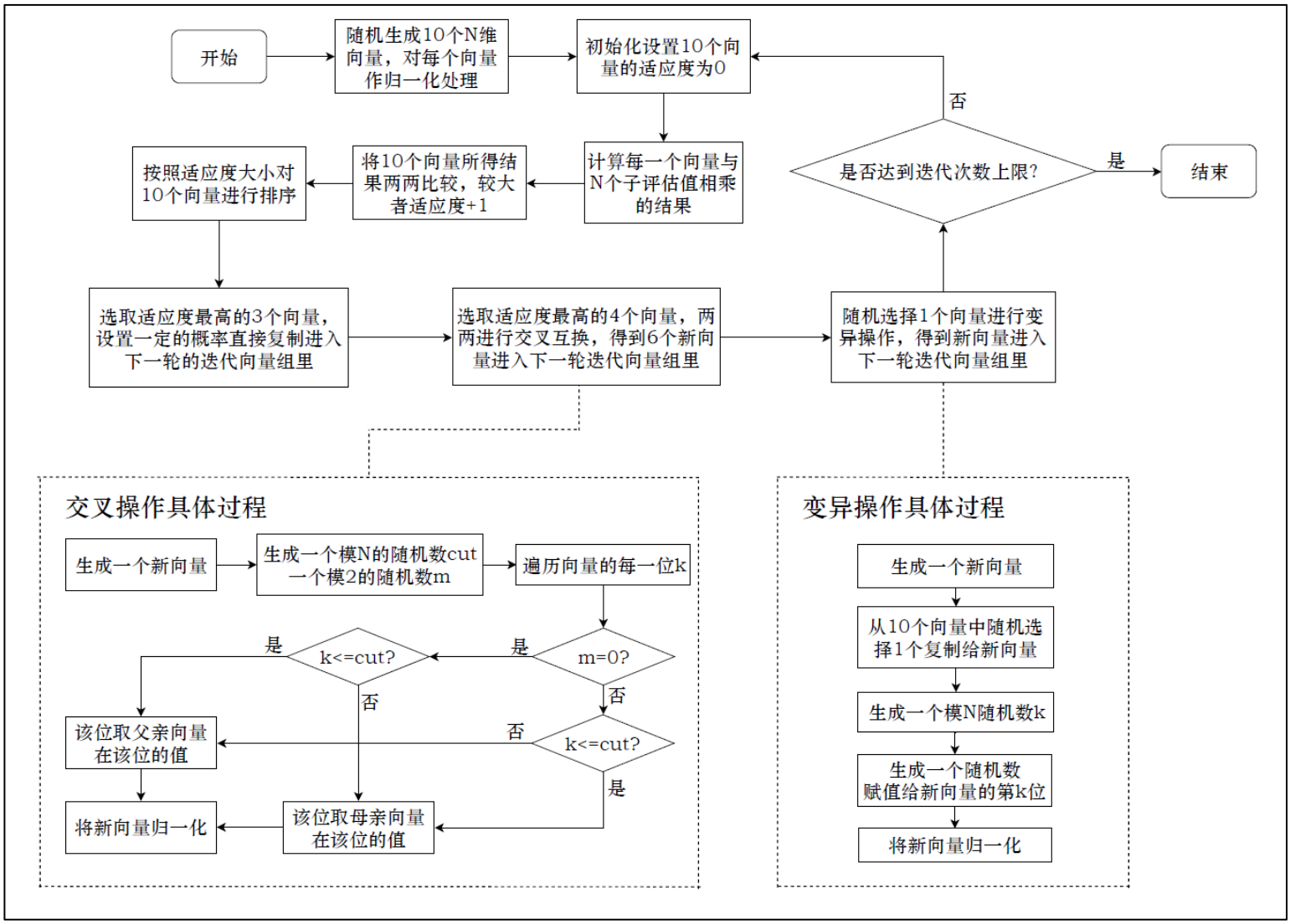
3) 根据此前计算得到的10个向量的适应度大小，我们选取适应度最高的3个向量，设置一定的概率直接复制进入下一轮的迭代向量组里。

4) 选取适应度最高的3个向量（标记为1、2、3），两两一组进行单点交叉，共三组。对于每一组交叉，标记这两个向量为a和b，进行如下操作：随机生成一个模N的数k，表示截取位置。截取位置前的部分不变，截取位置后的部分进行交换，模拟交叉，得到2个新的向量。最后将得到的共6个新向量做归一化处理后，加入下一轮的迭代向量组里。

5) 选择一个向量作为突变向量。从该向量中随机选择一个位置，用随机数替换，模拟突变的过程。最后将新向量做归一化处理后，加入下一轮的迭代向量组里。

6) 这样一来我们就得到新一组共10个向量，重复以上的过程。当达到迭代次数上限时，结束迭代。

2. 流程图



3. 关键代码展示

- **遗传算法—主体部分**

```
weightstruct BlackWhiteChess::ga(vector<double> subeval) {
    // 遗传算法计算子评估值的权重向量
    srand(time(0));
    vector<weightstruct> weightvec = init_weightvec(10);
    int iter = 20;  // 进行20次迭代
    while (iter--) {
        // 对当前10个向量两两进行比较
        for (int i = 0; i < 10; i++) {
            for (int j = i + 1; j < 10; j++) {
                if (calEval(weightvec[i].subweight, subeval) > calEval(weightvec[j].subweight, subeval))
                    weightvec[i].fitness++;
                else
                    weightvec[j].fitness++;
            }
        }
        //排序
        for (int i = 0; i < 10; i++)
            for (int j = i+1; j < 10; j++)
                if (weightvec[i].fitness < weightvec[j].fitness)
                    swap(weightvec[i], weightvec[j]);

        vector<weightstruct> new_weightvec;
        //复制
        copy(weightvec, new_weightvec);
        //交叉
        cross(weightvec, new_weightvec);
        //变异
        mutate(weightvec, new_weightvec);
        weightvec = new_weightvec;
    }
    return weightvec[0];
}
```

- **遗传算法—复制部分**

```
void BlackWhiteChess::copy(vector<weightstruct>& weightvec, vector<weightstruct>& new_weightvec){
    // 复制部分，有90%的概率选取适应度最高的3个进行复制
    if (rand() % 10) {
        new_weightvec.push_back(weightvec[0]);
        new_weightvec.back().fitness = 0;
        new_weightvec.push_back(weightvec[1]);
        new_weightvec.back().fitness = 0;
        new_weightvec.push_back(weightvec[2]);
        new_weightvec.back().fitness = 0;
    }
    //有10%的概率生成随机向量
    else {
        vector<weightstruct> rand_weight = init_weightvec(3);
        for (int i = 0; i < 3; i++)
            new_weightvec.push_back(rand_weight[i]);
    }
}
```

- **遗传算法—交叉部分**

```
void BlackWhiteChess::cross(vector<weightstruct>& weightvec, vector<weightstruct>& new_weightvec){
    // 交叉，选取适应度最高的4个两两进行单点交叉
    for (int i = 0; i < 4; i++) {
        for (int j = i + 1; j < 4; j++) {
            weightstruct new_weight1, new_weight2;
            new_weight1.fitness = 0;
            new_weight2.fitness = 0;
            double sum1 = 0, sum2 = 0;
            int cut = rand() % 7;   //cut为随机选择的单点交叉点
            for (int k = 0; k < 7; k++) {
                //如果位置<=cut，保持不变，直接从父母复制
                if (k <= cut) {
                    new_weight1.subweight.push_back(weightvec[i].subweight[k]);
                    sum1 += weightvec[i].subweight[k];
                    new_weight2.subweight.push_back(weightvec[j].subweight[k]);
                    sum2 += weightvec[j].subweight[k];
                }
                //如果位置>cut，父母向量对应位置进行交换
                else {
                    new_weight1.subweight.push_back(weightvec[j].subweight[k]);
                    sum1 += weightvec[j].subweight[k];
                    new_weight2.subweight.push_back(weightvec[i].subweight[k]);
                    sum2 += weightvec[i].subweight[k];
                }
            }
            //归一化
            for (int k = 0; k < 7; k++) {
                new_weight1.subweight[k] = new_weight1.subweight[k] / sum1;
                new_weight2.subweight[k] = new_weight2.subweight[k] / sum1;
            }
            new_weightvec.push_back(new_weight1);
            new_weightvec.push_back(new_weight2);
        }
    }
}
```

- **遗传算法—变异部分**

```
void BlackWhiteChess::mutate(vector<weightstruct>& weightvec, vector<weightstruct>& new_weightvec){
    // 变异部分，从适应度最高的3个向量中随机选择一个向量，随机选择一个向量元素赋予新的随机值
    int index = rand() % 3;
    weightstruct new_weight = weightvec[index];
    new_weight.fitness = 0;
    double sum = 0;
    // 随机选择一个位置赋予新的随机值
    int replace_pos = rand() % 7;
    for (int k = 0; k < 7; k++) {
        if (k == replace_pos){
            double replace_ele = (rand() / double(RAND_MAX));
            new_weight.subweight.push_back(replace_ele);
            sum += replace_ele;
        }
        else
            sum += new_weight.subweight[k];
    }
    // 归一化
    for (int k = 0; k < 7; k++)
        new_weight.subweight[k] = new_weight.subweight[k] / sum;
    new_weightvec.push_back(new_weight);
}
```

- **计算位置权重评估值**

```
double BlackWhiteChess::calPosEval(int board[8][8], int color){
    // 计算位置权重评估值
    int opp = color == 1 ? 2 : 1;
    double poseval = 0;
    for (int i = 0; i < 8; ++i){
        for (int j = 0; j < 8; ++j){
            if (board[i][j] == color)
                poseval += square_weights[i][j];
            else if (board[i][j] == opp)
                poseval -= square_weights[i][j];
        }
    }
    return poseval;
}
```

- **计算黑白子比例评估值**

```
double BlackWhiteChess::calRateEval(int board[8][8], int color){
    // 计算黑白子比例评估值
    int opp = color == 1 ? 2 : 1;
    int mycount = 0, opcount = 0;
    for (int i = 0; i < 8; ++i){
        for (int j = 0; j < 8; ++j){
            if (board[i][j] == color)
                mycount++;
            else if (board[i][j] == opp)
                opcount++;
        }
    }
    if (mycount > opcount)
        return 100.0 * mycount / (mycount + opcount);
    else if (mycount < opcount)
        return -100.0 * opcount / (mycount + opcount);
    else
        return 0;
}
```

- **计算占角评估值**

```
double BlackWhiteChess::calCornerEval(int board[8][8], int color){
    // 计算占角评估值
    int opp = color == 1 ? 2 : 1;
    int corner_pos[4][2] = { { 0, 0 },{ 0, 7 },{ 7, 0 },{ 7, 7 } };
    int mycorner = 0, opcorner = 0;
    for (int i = 0; i < 4; ++i){
        if (board[corner_pos[i][0]][corner_pos[i][1]] == color)
            mycorner++;
        else if (board[corner_pos[i][0]][corner_pos[i][1]] == opp)
            opcorner++;
    }
    return 25 * (mycorner - opcorner);
}
```

- **计算近角评估值**

```
double BlackWhiteChess::calNearCornerEval(int board[8][8], int color){
    // 计算近角评估值
    int opp = color == 1 ? 2 : 1;
    int corner_pos[4][2] = { { 0, 0 },{ 0, 7 },{ 7, 0 },{ 7, 7 } };
    int mynear = 0, oppnear = 0;
    for (int i = 0; i < 4; i++) {
        int x = corner_pos[i][0], y = corner_pos[i][1];
            if (board[x][y] == 0)
                for (int j = 0; j < 8; j++) {
                    int nx = x + direction[j][0], ny = y + direction[j][1];
                    if (is_on_board(nx, ny))
                        if (board[nx][ny] == color)
                            mynear++;
                        else if (board[nx][ny] == opp)
                            oppnear++;
                }
    }
    return -25 * (mynear - oppnear);
}
```

- **计算行动力评估值**

```
double BlackWhiteChess::calMoveEval(int board[8][8], int color){
    // 计算行动力评估值
    int opp = color == 1 ? 2 : 1;
    int mymove = find_places(board, color).size();
    int opmove = find_places(board, opp).size();
    //如果本方没有地方落子，则设定很低的行动力评估值
    if (mymove == 0)
        return -450;
    //如果对方没有地方落子，则设定很高的行动力评估值
    else if (opmove == 0)
        return 150;
    else if (mymove > opmove)
        return (100 * mymove) / (mymove + opmove);
    else if (mymove < opmove)
        return -(100 * opmove) / (mymove + opmove);
    else
        return 0;
}
```

- **计算稳定子评估值**

```
double BlackWhiteChess::calStableEval(int board[8][8], int color){
    // 计算稳定子评估值
    int mystable = 0, oppstable = 0;
    for (int i = 0; i < 8; ++i)
        for (int j = 0; j < 8; ++j)
            if (board[i][j] != 0 && is_stable(board, i, j))
                if (board[i][j] == color)
                    mystable++;
                else
                    oppstable++;
    return 12.5 * (mystable - oppstable);
}
```

- **计算边界棋子评估值**

```
double BlackWhiteChess::calSideEval(int board[8][8], int color){
    //计算边界棋子评估值
    int opp = color == 1 ? 2 : 1;
    int index[2] = {0, 7};
    int myside = 0, oppside = 0;
    for (int i = 0; i < 2; i++){
        for (int j = 0; j < 8; j++){
            if (board[index[i]][j] == color)
                myside++;
            else if (board[index[i]][j] == opp)
                oppside++;
        }
    }
    for (int i = 0; i < 2; i++){
        for (int j = 1; j < 7; j++){
            if (board[j][index[i]] == color)
                myside++;
            else if (board[index[i]][j] == opp)
                oppside++;
        }
    }
    return 2.5 * (myside - oppside);
}
```

4. 可选择的创新&优化

- 为了能使优良基因更多地遗传下去，本次试验我不使用轮盘赌，而是将种群中的个体按照适应度大小进行排列，选取排名前3的个体进行复制。当然，在操作过程中，也可以设置排在前面的个体复制两份，中间的个体复制一份，后面的个体不复制，更多地继承父母的特性。

- 在查阅资料后，我们了解到，简单遗传算法在任何情况下（交叉概率Pc,变异概率Pm，任意初始化，任意交叉算子，任意适应度函数）都是不收敛的，且不能搜索到全局最优解。如果在选择作用前（或后）保留当前最优解，则能保证收敛到全局最优解。

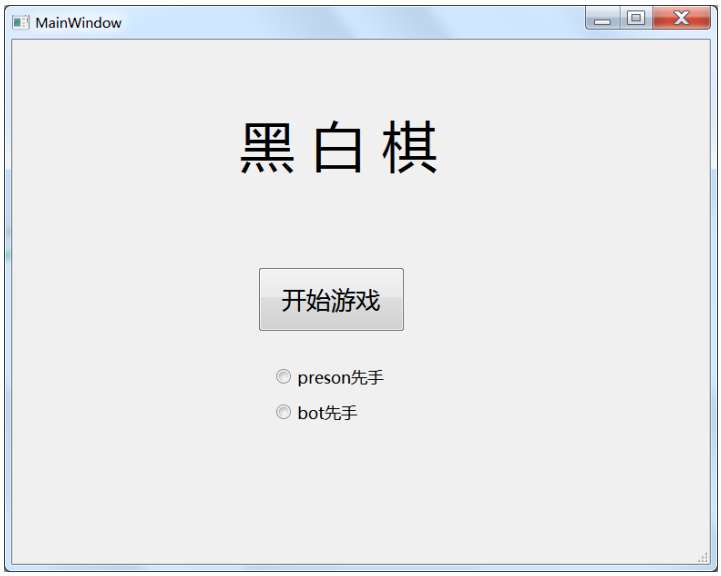
- 在迭代的过程红，群体中所有的个体有可能都陷于同一极值而停止变化。可使用“遗传-灾变”算法，即在遗传算法的基础上，模拟自然界的灾变现象，提高遗传算法的性能。当判断连续数代最佳染色体没有任何变化时，或者各个染色体已过于近似时，即可实施灾变。灾变方法很多，可以突然增大变异概率或对不同个体实施不同规模的突变。

- 对于交叉过程，也可以进一步改进。可以将某一对父母A、B进行n次交叉操作，生成2n个不同的个体，选出其中一个最高适应度的个体，送入子代对个体中。反复随机选择父母对，直到生成设定个数的子代个体为止。

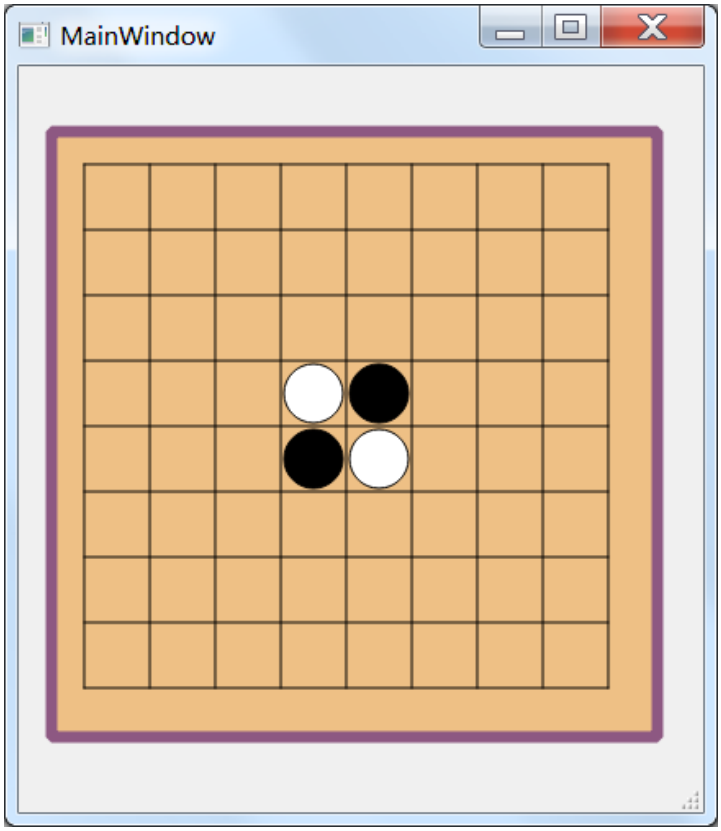
5. 实验结果展示&分析

我们运行程序，开始游戏。

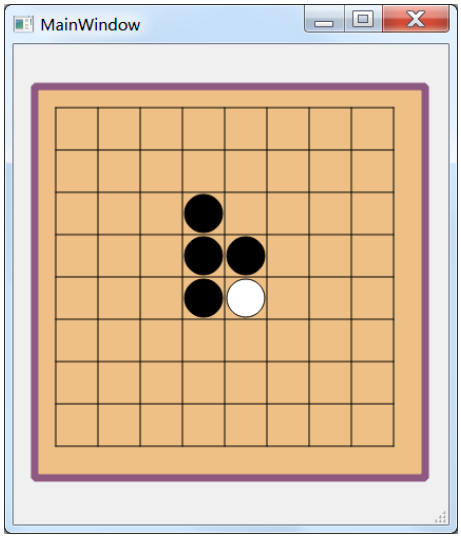
根据实验要求，用户可以选择先手或者后手。因此，在正式游戏前，程序会出现如下界面，用户可以点击“person先手”或者“bot先手”。



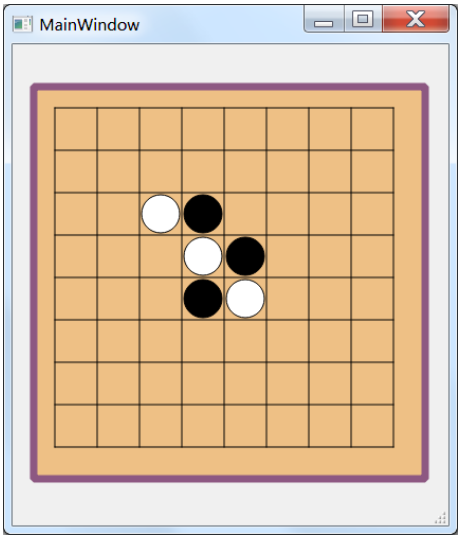
我们以person先手为例，开始黑白棋对弈。在等待游戏初始化完成后，我们可以看到如下的初始状态的棋盘。为说明方便，我们将纵坐标标记为A-H，横坐标标记为1-8。



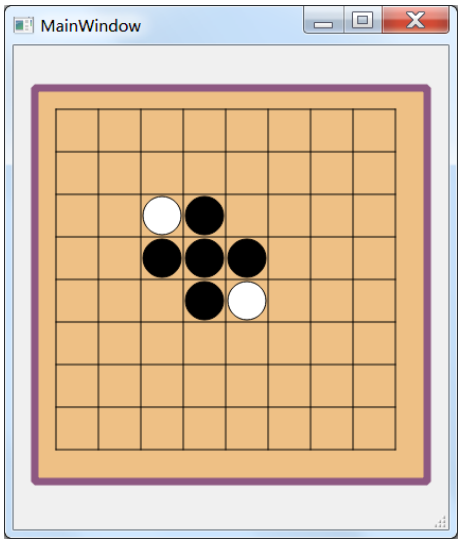
Step1：用户落子D3



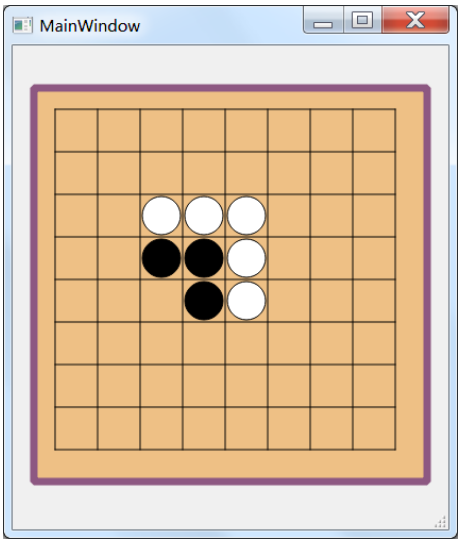
Step2：AI落子C3



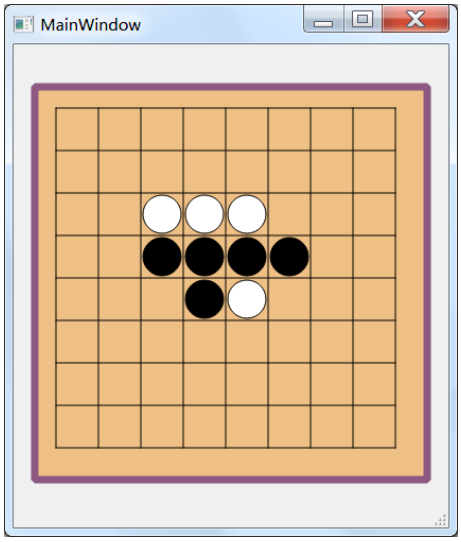
Step3：用户落子D3



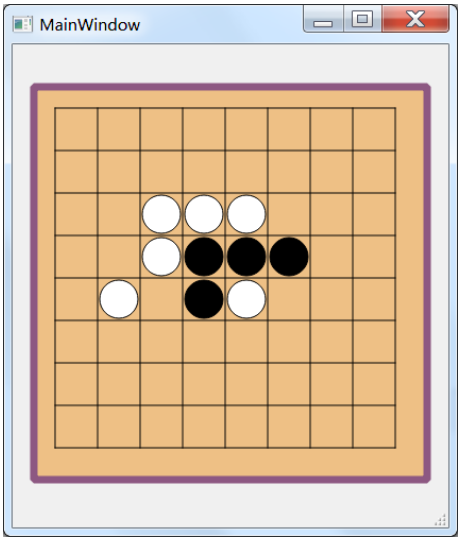
Step4：AI落子E3



Step5：用户落子F4



Step6：AI落子B5



我们将初始化过程中遗传算法迭代得到的权重向量进行打印，部分结果如下所示：

```
0.167067   0.174289   0.169405   0.228061   0.104787   0.0539907  0.102401
0.233932   0.0978068  0.154535   0.237974   0.00310065 0.0419198  0.230732
0.189527   0.0024374  0.202612   0.109696   0.115787   0.145948   0.233993
0.205642   0.00264464 0.219839   0.119023   0.125631   0.158357   0.168863
0.0346405  0.30855    0.0174988  0.40574    0.00744592 0.0780642  0.14806
0.205642   0.00264464 0.219839   0.119023   0.125631   0.158357   0.168863
0.0346405  0.30855    0.0174988  0.40574    0.00744592 0.0780642  0.14806
0.0346405  0.30855    0.0174988  0.40574    0.00744592 0.0780642  0.14806
0.262674   0.00337809 0.271576   0.0393144  0.299972   0.0941269  0.0289589
0.190228   0.287919   0.20987    0.0451624  0.052943   0.0738364  0.140041
0.0766635  0.142092   0.127009   0.19322    0.274007   0.138118   0.0488904
0.345265   0.0911244  0.196471   0.0309052  0.0326123  0.0465396  0.257744
0.082224   0.00202974 0.260155   0.217602   0.258676   0.161273   0.0180406
0.36179    0.0931529  0.171274   0.0280046  0.0328293  0.134333   0.178616
0.099762   0.0308286  0.257338   0.210607   0.222963   0.159562   0.0189386
0.366644   0.0965104  0.238803   0.0389363  0.0456443  0.263103   0.167939
0.094073   0.162636   0.139884   0.18823    0.224621   0.140041   0.0494422
0.330666   0.0871559  0.162076   0.0347392  0.0300936  0.21498    0.141165
0.0298452  0.264627   0.0151911  0.352231   0.0112584  0.288236   0.0316941
0.368714   0.0973131  0.17563    0.0282635  0.030676   0.126293   0.179623
0.0309279  0.0366485  0.072319   0.304892   0.230101   0.175827   0.0550383
0.355026   0.0937006  0.187306   0.0305398  0.0358012  0.0486738  0.248952
0.0678773  0.00110837 0.0401884  0.0815598  0.411471   0.347076   0.0507192
0.355026   0.0937006  0.187306   0.0305398  0.0358012  0.0486738  0.248952
0.0268808  0.237728   0.0135119  0.40793    0.00584349 0.0706403  0.202679
0.409645   0.108116   0.266811   0.0435029  0.0509977  0.0924757  0.0284509
0.0298142  0.0365197  0.260616   0.142261   0.195935   0.310871   0.023982
0.409645   0.108116   0.266811   0.0435029  0.0509977  0.0924757  0.0284509
0.0298694  0.264924   0.0150247  0.348373   0.0110078  0.0839189  0.240777
0.409645   0.108116   0.266811   0.0435029  0.0509977  0.0924757  0.0284509
0.0298978  0.265251   0.0150432  0.348802   0.0110335  0.0838096  0.240464
0.345265   0.0911244  0.196471   0.0309052  0.0326123  0.0465396  0.257744
0.172771   0.00357257 0.102262   0.124611   0.314143   0.266768   0.0158892
0.333746   0.0880841  0.160267   0.0344881  0.0387442  0.223329   0.142551
0.182055   0.0345173  0.0847172  0.102684   0.275345   0.210399   0.110281
0.193875   0.30399    0.213987   0.0460483  0.0539816  0.0758783  0.112241
0.171127   0.0052721  0.150723   0.179592   0.30119    0.146901   0.0451954
0.193875   0.30399    0.213987   0.0460483  0.0539816  0.0758783  0.112241
```

由于使用遗传算法计算权重的过程，对弈双方采取的随机下棋的方法，存在偶然性。不同的棋局存在差异，所以最终迭代得到的权重向量会有差异。但是我们可以从中看出一些规律出来，比如第一项（位置权重评估值）和第四项（占角评估值）所对应的权重在大部分的时候比其他子评估值的权重占比要大。这也印证了黑白棋在对弈过程中本方抢占四个角、避免对方抢占四个角的重要性。

为了观察程序的效果，在用遗传算法得到我们所求的相对较优的权重值后，我们设置用户采取随机下的方式，与AI进行对战100局（搜索深度为4）。在这100局中，AI赢72局，输23局，平5局，胜率为72%。而直接使用我们自己设定的权重值与AI进行对战，同样设置用户随机下，搜索深度为4，AI的胜率为69%。

综合模拟对战得到的胜率，我们可以看到，使用遗传算法计算子评估值对应的权重向量，对黑白棋胜率的提升有略微的帮助，但是帮助有限。

6. 小组分工

组员信息		负责工作
17341088	梁超	黑白棋UI设计
		棋面各项评估值的设计
		遗传算法改进
		实验报告撰写
17341178	薛伟豪	博弈树构建
		遗传算法编写
		棋面各项评估值的设计
		实验报告撰写