第三次作业：对抗搜索五子棋 AI

1953194 庞鹏瞩

设计思想

极大极小算法和 ab 剪枝算法，都是用来进行博弈与对抗游戏的算法，且要求这个博弈是一个无偏博弈。一些我们熟悉的棋类游戏，形如象棋、五子棋、围棋都是无偏博弈，它们都满足下面的三个要求：

1.完全信息，所有游戏者都能看到整个局势。这排除了类似桥牌一类的游戏。

2.无随机行动。所有行动都确定性地将目前局势转变到下一个局势。

3.在有限步行动之后按照规则游戏必将终止，此时有唯一的一方成为赢家。

在无偏博弈中，每一个特定的局势都对应着一个 nim 数，这是由 SG 定理得到。知道局面的 SG 函数值，可以推理得到当前局面为必胜或必败。但是在棋类游戏中，一般是没有机会直接从终局开始倒推得到各个局面的 nim 数，所以常用的方法就是搜索，然后对于局面，给出一个估价函数，来判断当前局面的优劣。然后通过极大极小值搜索来求得当前局面下的最优步骤，alpha-beta 的方法来进行剪枝优化加快搜索速度。

本次作业中，由于要求采用极大极小值搜索和 alpha-beta 剪枝，所以最好的实现方式还是一个棋类游戏。在大作业当中是通过中国象棋的方式来进行 AI 的编写，所以这次换成了一个五子棋游戏来进行编写。游戏包含两种玩法，本地双人和人机对战。其中的人机对战中，玩家可以与五子棋的 AI 进行较量。

下面是这个程序中所使用的五子棋的局面估价函数：

```
int evaluate(int color)
{
    //返回行，列，左斜和右斜的评价函数之和
    Point a, b;
    a.status = color;
    b.status = 3 - color;
    int val = 0;
```

```
        val += LigatureOnLine(a) - LigatureOnLine(b);
        val += LigatureOnRow(a) - LigatureOnRow(b);
        val += LigatureOnDia(a) - LigatureOnDia(b);
        val += LigatureOnBdia(a) - LigatureOnDia(b);
        return val;
    }
```

每一个方向上的评价函数（以横向为例）：

```
    int LigatureOnLine(const Point now)
    {
        int num = 0, val = 0;            //同子个数,对方个数，价值,是否被计分
        for (int i = 0; i < SIZE; i++)
            for(int j = 0; j < SIZE; j ++)
                if (chess[i][j] == now.status)
                {
                    num = 0;
                    for (int k = j; k < j + 5; k++)
                    {
                        if (chess[i][k] == now.status)
                            num++;
                        if (k == SIZE)
                            break;
                    }
                    //五子连珠
                    if (num == 5)
                    {
                        val += 1000000;
                        j += 4;
                        continue;
                    }
                    if (num == 4)
                    {
                        //中间缺
                        if (chess[i][j + 2] == BLANK)
                        {
                            val += 2600;
                            j += 4;
                            continue;
                        }
                        else if (chess[i][j + 1] == BLANK || chess[i][j +
3] == BLANK)
                        {
                            val += 3000;
```

```
            j += 4;
            continue;
        }
    }

    //四个
    num = 0;
    for (int k = j; k < j + 4; k++)
    {
        if (chess[i][k] == now.status)
            num++;
        if (k == SIZE)
            break;
    }
    //四子
    if (num == 4)
    {
        //边界情况
        if (j == 0)
        {
            if (chess[i][j + 4] == BLANK)
            {
                val += 2500;
                j += 3;
                continue;
            }
        }
        else if (j == 11)
        {
            if (chess[i][j - 1] == BLANK)
            {
                val += 2500;
                j += 3;
                break;
            }
        }
        else if (chess[i][j - 1] != BLANK)
        {
            if (chess[i][j + 4] == BLANK)
            {
                val += 2500;
                j += 3;
                continue;
            }
```

```
                    }
                    else if (chess[i][j + 4] == BLANK)
                    {
                        val += 300000;
                        j += 3;
                        continue;
                    }
                    else
                    {
                        val += 2500;
                        j += 3;
                        continue;
                    }
                }

                //三子
                num = 0;
                for (int k = j; k < j + 3; k++)
                {
                    if (chess[i][k] == now.status)
                        num++;
                    if (k == SIZE)
                        break;
                }
                if (num == 3)
                {
                    //边界
                    if (j == 0)
                    {
                        if (chess[i][j + 3] == BLANK)
                        {
                            val += 500;
                            j += 2;
                            continue;
                        }
                    }
                    else if (j == 12)
                    {
                        if (chess[i][j - 1] == BLANK)
                        {
                            val += 500;
                            j += 2;
                            continue;
                        }
```

```
                    }
                    else if (chess[i][j - 1] != BLANK)
                    {
                        if (chess[i][j + 3] == BLANK)
                        {
                            val += 500;
                            j += 2;
                            continue;
                        }
                    }
                    else
                    {
                        //活三
                        if (chess[i][j + 3] == BLANK)
                        {
                            val += 3000;
                            j += 2;
                            continue;
                        }
                        else
                        {
                            val += 500;
                            j += 2;
                            continue;
                        }
                    }
                }
                if (j != 0 && j != 13 && chess[i][j] == chess[i][j +
1]
                    && chess[i][j] != BLANK
                    && chess[i][j - 1] == BLANK && chess[i][j + 2] ==
BLANK)
                {
                    val += 50;
                    j += 1;
                }
            }
        return val;
    }
```

以上是估价函数的代码。

在完成估价函数之后，我们要进行极大极小值搜索。由于 alpha-beta 搜索只

是在普通的极大极小值搜索的基础上加上了一个 beta 值来进行剪枝操作，所以这里直接用了加上 alpha-beta 剪枝的极大极小值搜索。

在 alpha-beta 剪枝中，若想要使剪枝的效果最好，那么应当优先选择那些更优的着法，这样的话可以将很多不那么优秀的着法剪枝，从而提高搜索的效率。同时，我们应该选择那些有意义的着法来减小搜索的范围，如果对方将棋子放在棋盘的正中心，那么将棋子放在现有的棋子的旁边的效果肯定比放在角落的效果更好。如果一个棋子放在一个很远的地方，那么它的效果肯定很差，我们在选择着法的时候就可以将其忽略掉。

下面是判断这个棋子的位置是否有意义的代码。

```
inline int tooFar(int x, int y)
{
    if (x > 0 && chess[x - 1][y] != BLANK)
        return 0;
    if (x > 1 && chess[x - 2][y] != BLANK)
        return 0;
    if (x != SIZE && chess[x + 1][y] != BLANK)
        return 0;
    if (x != SIZE - 1 && chess[x + 2][y] != BLANK)
        return 0;
    if (y > 0 && chess[x][y - 1] != BLANK)
        return 0;
    if (y > 1 && chess[x][y - 2] != BLANK)
        return 0;
    if (y != SIZE && chess[x][y + 1] != BLANK)
        return 0;
    if (y != SIZE - 1 && chess[x][y + 2] != BLANK)
        return 0;
    //左上角
    if (x > 0 && y > 0 && chess[x - 1][y - 1] != BLANK)
        return 0;
    if (x > 1 && y > 1 && chess[x - 2][y - 2] != BLANK)
        return 0;
    //右下角
    if (x != SIZE && y != SIZE && chess[x + 1][y + 1] != BLANK)
        return 0;
    if (x != SIZE - 1 && y != SIZE - 1 && chess[x + 2][y + 2] != BLANK)
        return 0;
    //右上角 x+ y-
    if (x != SIZE && y > 0 && chess[x + 1][y - 1] != BLANK)
        return 0;
```

```
        if (x != SIZE - 1 && y > 1 && chess[x + 2][y - 2] != BLANK)
            return 0;
        //左下角 x- y+
        if (x > 0 && y != SIZE && chess[x - 1][y + 1] != BLANK)
            return 0;
        if (x >1 && y != SIZE - 1 && chess[x - 2][y + 2] != BLANK)
            return 0;
        return 1;
```

下面是 ab 搜索的代码。

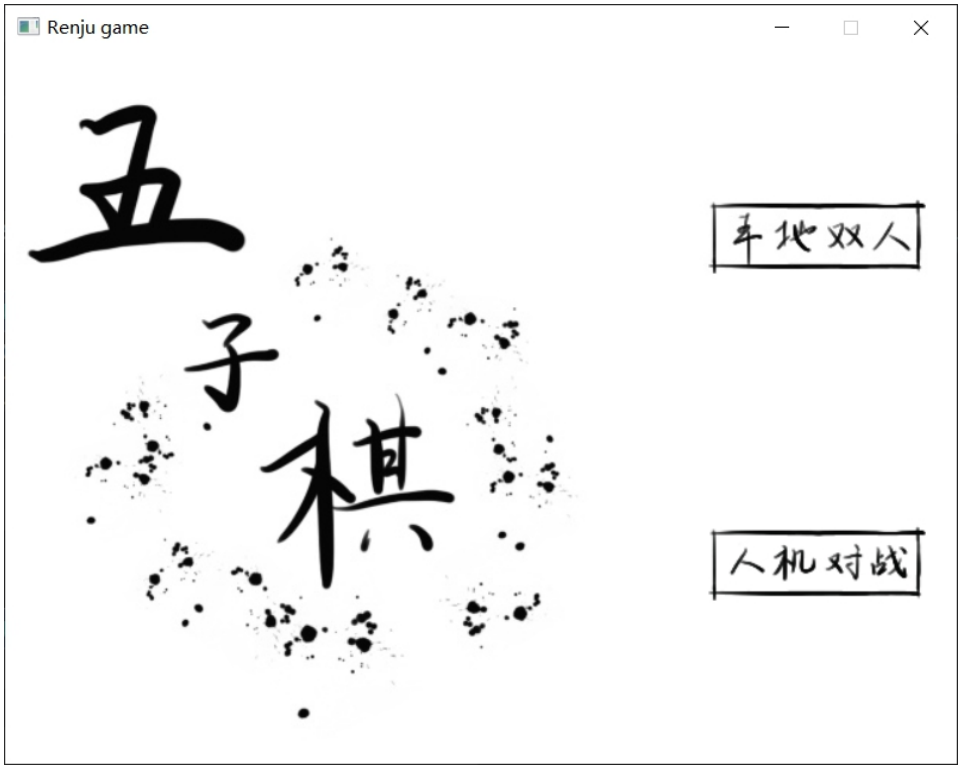
在这个搜索中设置的搜索深度为 4，如果要让搜索的效果更好可以让时间延长，或者用迭代加深搜索控制搜索的时间。

```
    value getMaxValue(int leftstep, int color, int alpha, int beta, int
dep)
    {
        value temp;
        temp.col = 0; temp.row = 0; temp.val = 0;
        if (leftstep == 0)
        {
            temp.val = evaluate(color);
            return temp;
        }

        value step[200];
        int d = 0;

        for (int i = 0; i < SIZE; i ++)
            for (int j = 0; j < SIZE; j ++)
                if (chess[i][j] == BLANK && !tooFar(i, j))
                {
                    int val;
                    chess[i][j] = color;
                    val = evaluate(color);//走子之后的局面判断
                    if (val > 800000)
                    {
                        chess[i][j] = 0;
                        if (leftstep == dep)
                        {
                            temp.row = i;
                            temp.col = j;
                        }
                        temp.val = val;
                        return temp;
                    }
```

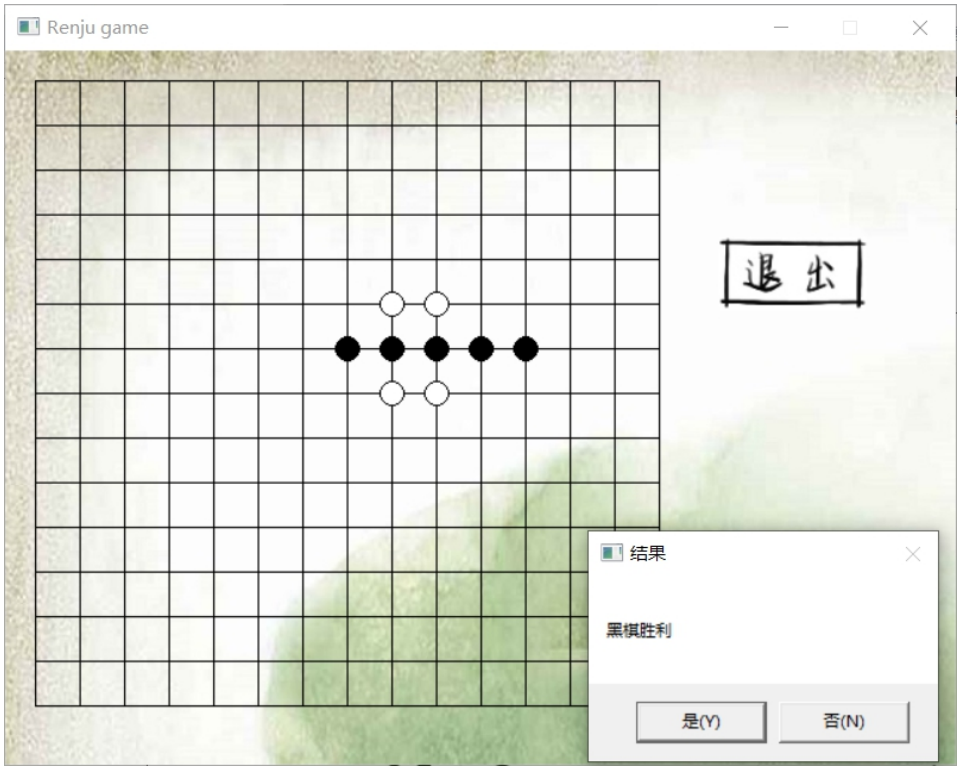
```
                    step[++d].row = i; step[d].col = j;
                    step[d].val = val;
                    chess[i][j] = BLANK;
                }
        std::sort(step + 1, step + 1 + d, cmp);//进行着法排序，先走更优秀的着法
        for (int i = 1; i <= d; i++)
        {
            int val, p = step[i].row, q = step[i].col;
            chess[p][q] = color;
            val = -getMaxValue(leftstep - 1, 3 - color, -beta, -alpha, dep).val;
            chess[p][q] = BLANK;
            if (val >= beta)
            {
                temp.val = beta;
                return temp;
            }
            if (val > alpha)
            {
                alpha = val;
                if (leftstep == dep)
                {
                    temp.row = p;
                    temp.col = q;
                }
            }
        }
        temp.val = alpha;
        return temp;
    }
```

实现效果

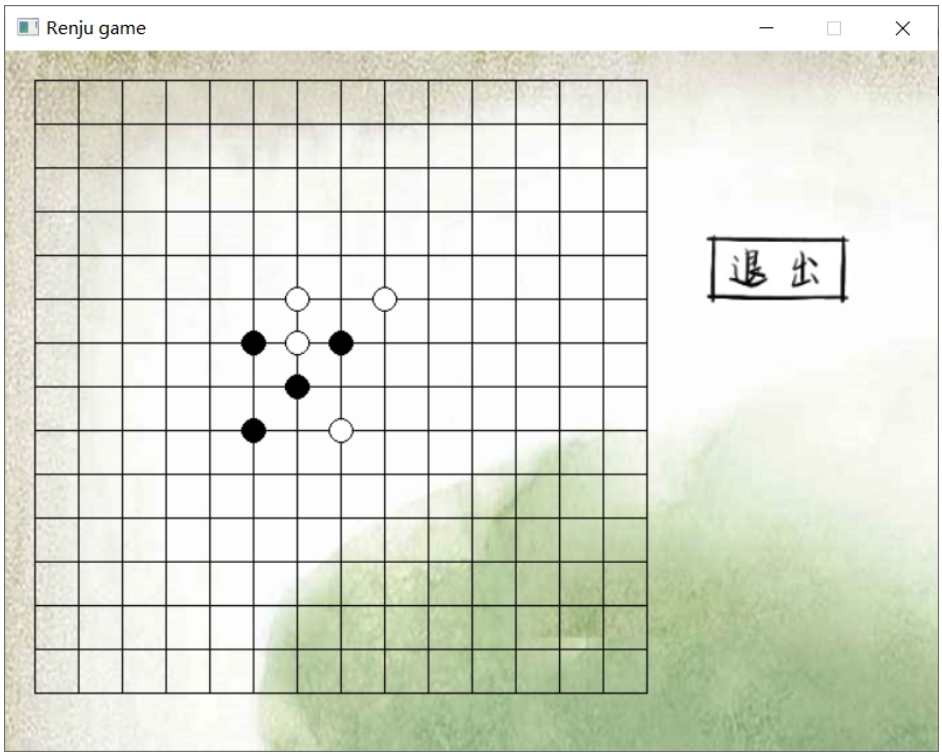
开始界面

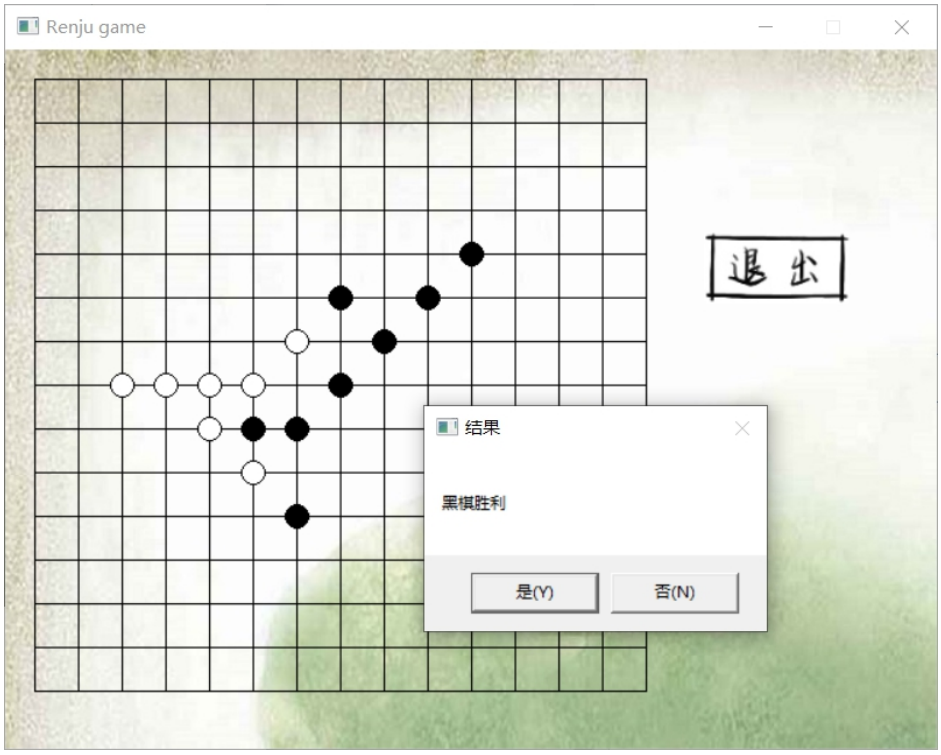


本地双人（都是玩家操作）



人机对战（玩家执黑）





总结：AI 有一定的走子能力，不过还不算很厉害。一个是因为五子棋的估值函数不是十分好写，五子棋的各种各样的局面很多，对于各种不同的局面的估值大小不是一件轻易就可以精确掌握的事情。二是这个搜索的层数还不是很深，后

续还可以加入很多高级的搜索方式来提高效率增加层数，这在之后的大作业中可以得到实现。