一、 实验描述

1. 实验目的

练习使用博弈树上的 alpha-beta 剪枝算法实现人工智能小游戏——五子棋。

2. 实验问题描述

五子棋是世界智力运动会竞技项目之一，是一种两人对弈的纯策略型棋类游戏，通常双方分别使用黑白两色的棋子，下在棋盘直线与横线的交叉点上，先形成 5 子连线者获胜。

3. 实验原理

当 AI 玩家落子的时候，通过对当前棋盘局面的探索和判断，找出最有利于自己落子的位置，为了使 AI 玩家更聪明一点，可以让 AI 向后思考几步，而这时就要使用 alpha-beta 剪枝算法，通过对整个棋盘局面的棋型及组合进行利弊的综合评估，得出最佳落子位置。

4. 实验环境

Windows10 系统，Qt，visual studio 2017

一、 算法详细设计

本次实验用到的主要算法是 alpha-beta 剪枝算法。Alpha-beta 剪枝算法其实是 MinMax 算法的优化，Alpha-Beta 剪枝用于裁剪搜索树中没有意义的不需要搜索的树枝，以提高运算速度，它们产生的结果完全相同，不过算法运行的时间效率却不同，前者正是利用剪枝将一些没有必要的搜索的树枝剪去，这样就大大加快了搜索时间。

1. 问题转化

五子棋实际是两个人的博弈问题，这就涉及到博弈树，通过对博弈树的搜索，来找到最佳着棋法。下面对博弈树模型的各部分进行解释。

（1） 结点：树的节点就是每落一个子之后，形成的新的棋盘状态（ChessPadStatus），根据棋盘状态判断棋型（ChessType）及组合，之后再按照棋型的等级进行棋盘局面（Situation）评分，得到一个分数（Score），此即为博弈树结点的含义。

（2） 树枝：两个节点之间的连线。它连接的是两个棋盘的相邻局面（棋盘状态），也代表着棋盘局面的转换，即父节点是上一个玩家落一个子，子节点就是对手玩家在此基础上再落一个子形成的新局面。

（3） 层：对应于博弈树搜索的深度，它也和回合相对应，根节点为第 0 层，以下层数递增。

2. 算法描述（核心算法）

Alpha-Beta 剪枝算法的基本依据是：棋手不会做出对自己不利的选择。依据这个前提，如果一个节点明显是不利于自己的节点，那么就可以直接剪掉这个节点。

现在假设 AI 会在 MAX 层选择最大节点，玩家会在 MIN 层选择最小节点。

在 MAX 层：如果在当前层已将找到一个最大值 X，假设又在下一层找到一个小于 X 的

值 Y，玩家选择的始终是不利于 AI 行棋的值，即最小值也就是着在 MIN 层中这个节点的值不会超过 Y；那么就没有必要再去扩展 MIN 层这个节点，也就是说这个结点的值不可能会对 MAX 层最大值的增大有贡献。（此亦即 alpha 剪枝）

在 MIN 层：假设当前已经搜索到一个最小值 Y，若发现下一层（MAX）还有一个比 Y 还大的结点值 X,因为 AI 总是会选择有利于自己行棋的结点值，即最大值，也就是在 MAX 层中这个节点的值不会小于 X；那么此时也没有必要去扩展 MAX 层该结点，也就是着这个结点值不会对减小 MIN 层最小值有贡献。（此亦即 beta 剪枝）

通过对 alpha-beta 剪枝的描述，其实算法就很容易了。Alpha 为最大下界，beta 为最小上界，alpha 初始化为负无穷，beta 初始化为正无穷。首先可以对整个棋盘的每个位置都进行评分，按照思考（搜索）层数，从 MAX 层开始，假设 AI 落子进行评分，接下来 MIN 层，玩家落子，进行评分，隔一层一个 MAX 层，直到搜索的深度 MaxDepth,此时就对叶子结点的棋盘局面进行评估，得到一个分数，再进行回溯，按照上述剪枝的方法对 alpha，beta 的值进行修改，若 alpha>=beta 则进行剪枝，直到根节点，也就是最初进入思考的这个位置，就会的到一个棋局评估最大值，AI 也就会在此落子。

3. **算法涉及到的类**

(1) AI.h//人工智能类

```
struct  Point {
    int x;//0到14，行
    int y;//0到14，列
};
struct Status {//当前位置的形式，打分根据这个来打
    int win5;//5连珠
    int alive4;//活4
    int rush4_1;//冲四1
    int rush4_2;//冲四2->低级版本
    int alive3;//活3
    int sleep3_2;//眠三1->跳3
    int sleep3_1;//眠三2
    int alive2;//活2
    int sleep2_2;//低级活2，眠二1
    int sleep2_1;//眠二2
    int nothreat;//没有威胁(死3，死4，死2，死1)
};
```

```
class AI{
private:
    //棋局类型
     static const int WIN5 = 0;
     static const int ALIVE4 = 1;
     static const int RUSH4_1 = 2;
```

```
public:
    //判断一个方向的棋型
    int JudgeChessType(const int chess[9]);
    //综合四个方向进行棋局类型计数并评估分数
    int Evaluate(const int state[15][15],  Point point, int color);
```

(2) AI_Chesser(AI 玩家)

```
class AI_Chesser
{
private:
    int color;
    AI AI_player;
public:
    AI_Chesser(int color);//构造函数
    Chess GiveNextChess(const int ChessPad[15][15], int &x, int &y);
     //给出下一个落子的位置
};
```

(3) Chess.h(棋子的属性，包括颜色，坐标等)

```cpp
class Chess {
private:
    int color;
    int x;
    int y;
public:
    Chess(int color, int x, int y);
    int GetColor();//得到棋子的颜色
    void GetPoint(int &x, int &y);//得到位置的坐标
    void SetColor(int color);//设置棋子颜色
    void SetPoint(int x, int y);//设置坐标
};
```

(4) ChessBoard.h(棋盘的状态，落一颗棋子，及棋子的横纵坐标)

```cpp
class ChessBoard
{
private:
    int LastX;//上一次落子的横坐标
    int LastY;//上一次落子的纵坐标
    int ChessPadStatus[15][15];//棋盘状态

public:
    ChessBoard();
    void AddChess(Chess chess);//下一个棋子
    void GetChessStatus(int ChessPadStatus[15][15]);//得到当前棋盘的状态
};
```

(5) Judge.h(判断胜负结果)

```cpp
class Judge
{
private:
    int CurChesser; //0-black  1-white
public:
    Judge();
    int JudgeResult(const int ChessPad[15][15]);//判断两个玩家的胜负结果
    int JudgeNextChesser();//判断下一个棋手
};
```

(6) PE_Chesser.h(人玩家的动作，下棋及棋的颜色)

```
class PE_Chesser
{
private:
     int color;//人玩家的棋子颜色

public:
     PE_Chesser(int color);//人玩家的构造
     Chess GiveNextChess(int x, int y);//落下一颗棋子
};
```

(7) Mainwindow1.h(模式选择窗口)

```
#ifndef MAINWINDOW1_H
#define MAINWINDOW1_H
#include "mainwindow.h"
#include <QButtonGroup>
#include <QMainWindow>
namespace Ui {
class MainWindow1;
}
class MainWindow1 : public QMainWindow
{
    Q_OBJECT
public:
    explicit MainWindow1(QWidget *parent = nullptr);
    int mode, fp;
    int flag1 = 0, flag2 = 0, flag3 = 0, flag4 = 0, flag5 = 0, flag6 =
0;
    ~MainWindow1();
private slots:
    void on_radioButton_clicked(bool checked);
    void on_radioButton_2_clicked(bool checked);
    void on_pushButton_2_clicked();
    void on_checkBox_clicked(bool checked);
    void on_checkBox_2_clicked(bool checked);
    void on_radioButton_3_clicked(bool checked);
    void on_radioButton_4_clicked(bool checked);
private:
    Ui::MainWindow1 *ui;
    MainWindow *m = new MainWindow;
};
#endif // MAINWINDOW1_H
```

(8) 五子棋主操作窗口

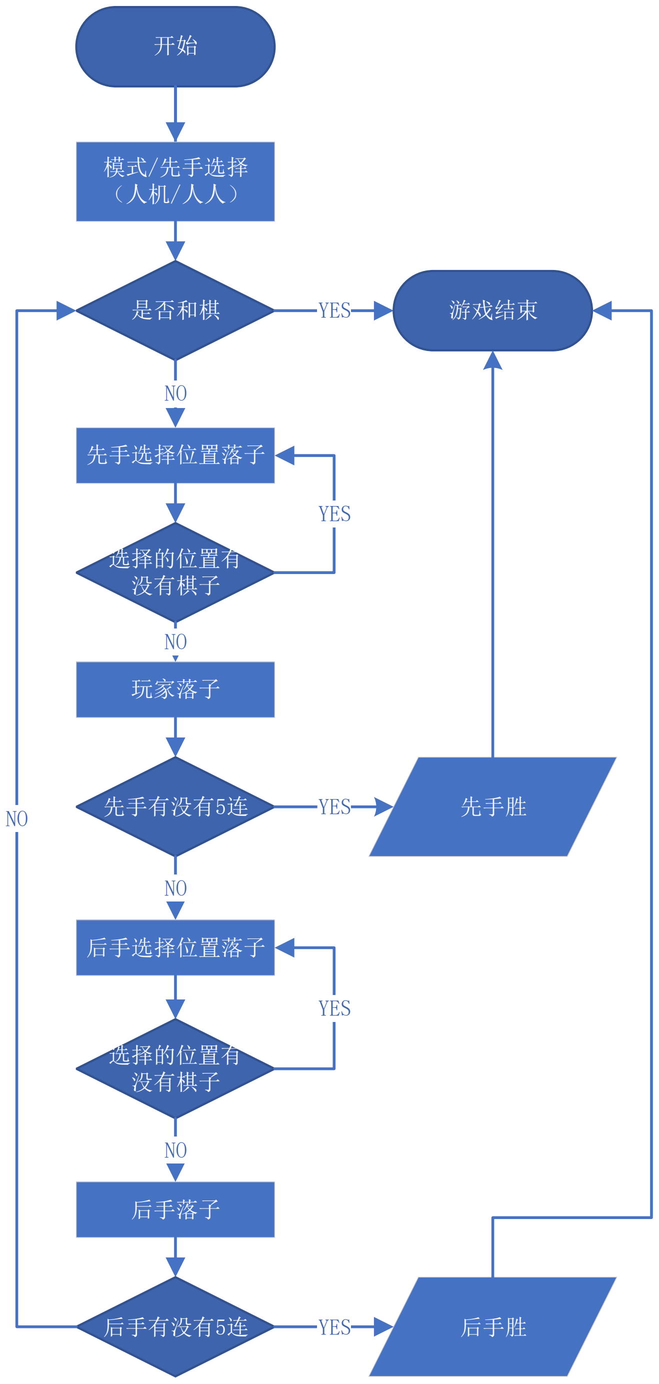
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include "AI_Chesser.h"
#include "ChessBoard.h"
#include "Judge.h"
#include "PE_Chesser.h"
#include <QMainWindow>
namespace Ui {
class MainWindow;
}
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    int fp;   //first player
    int mode; //game mode
    int PvAflag = 0;//人机对战标志
    explicit MainWindow(QWidget *parent = nullptr);
    void paintEvent(QPaintEvent *);//绘制棋子及棋盘
    void mouseMoveEvent(QMouseEvent *event);//鼠标移动事件
    void mouseReleaseEvent(QMouseEvent *);//鼠标左键释放事件（AI 行棋）
    void mousePressEvent(QMouseEvent *);//鼠标点击事件（PE 行棋）
    int player;
    ~MainWindow();

private:
    Ui::MainWindow *ui;
    Judge judge;
    ChessBoard chessboard;
    int Status[15][15] = {{0}};
    int moveX, moveY, currentX, currentY; //mouse move position
private slots:
    void PE_Chess();
    void AI_Chess();
};
#endif
```

4. 算法总结

以上算法的类就是本次五子棋实验的所有相关类，一个完整的项目就是将这些类加一组合并调用。其中最主要的当属 AI 类，它主要包括了判断棋型，棋型的等级划分、为当前棋盘状态估分，alpha-beta 剪枝等主要操作。

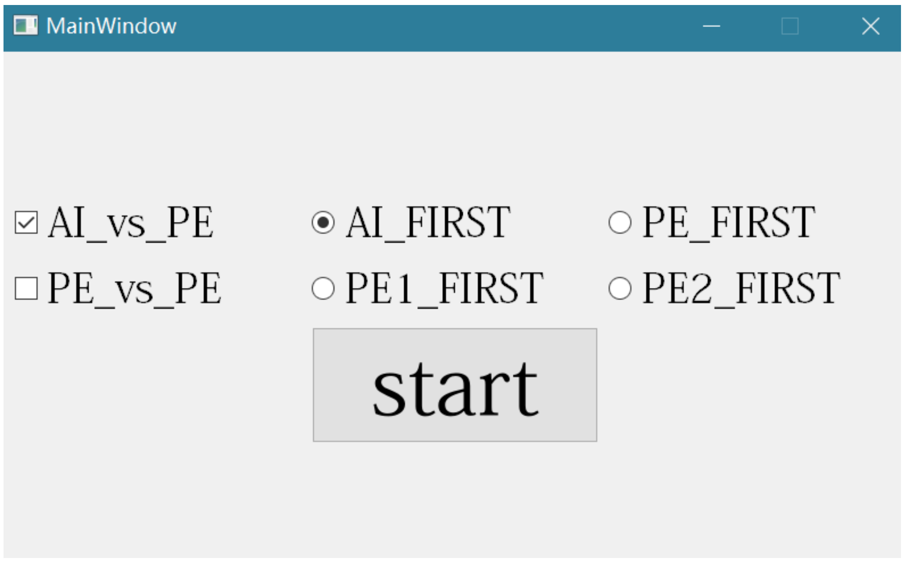
将这些操作以类的形式进行封装，使得每个步骤都有了清晰的层次和次序关系，整个程序算法的结构更加突出，可以很好的理清每个动作的执行以及类与类之间的依赖关系。

二、 程序框图



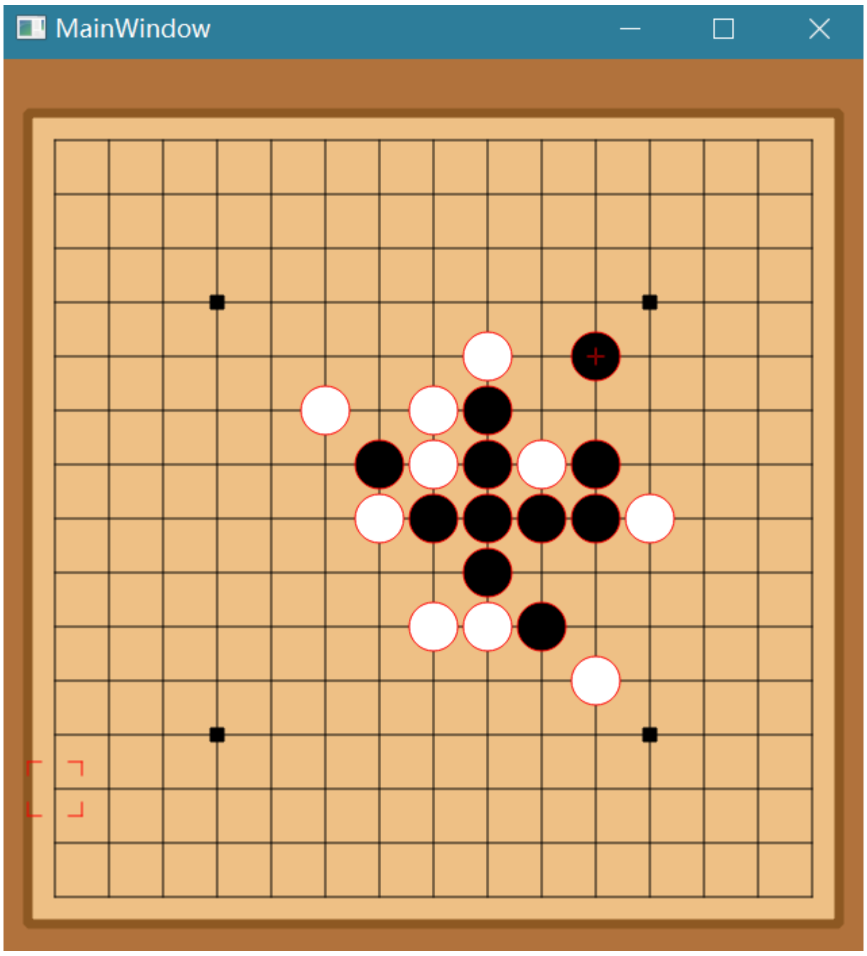
三、 游戏截图

1. 模式选择界面



[✓] AI_vs_PE　　● AI_FIRST　　○ PE_FIRST
☐ PE_vs_PE　　○ PE1_FIRST　　○ PE2_FIRST

2. 游戏界面



四、　小结

通过本次“五子棋”实验的过程分析及程序编写，我更加清楚的认识到 alpha-beta 剪枝算法的重要性以及其搜索的高效率，对博弈树树的 alpha-beta 剪枝算法有了更深一步的认识。再者就是，通过写五子棋程序，对 C++类的封装性，以及其清晰明了的程序架构和执行顺序也有了更进一步的认识，尤其是各个类之间的依赖性以及相关调用，使得程序的连通性和完整性有了更大的提升。

同时还存在的问题就是传统的剪枝算法对于思考深度较高的程序会有一定的局限性和效率限制，所以在以后的学习中还要对 alpha-beta 剪枝算法进行更深层次的优化和改进，让五子棋变得更加聪明和迅速。

五、 源程序

1. AI.cpp

```
/*AlphaBeta 剪枝*/
int AI::AlphaBeta(int curState[][15], Point point, int alpha, int beta, int
depth,int curPlayer)
{
    int value;
    if (depth ==4)
        return Evaluate(curState, point, curPlayer);
    else
    {
        if (curPlayer == 1)
        {
            int i, j;
            for (i = 0; i < 15; i++)
                for (j = 0; j < 15; j++)
                {
                    if (curState[i][j] == NO)
                    {
                        point.x = i;
                        point.y = j;
                        value = AlphaBeta(curState, point, alpha, beta, depth + 1, 2);
                        if (alpha < value)
                            alpha = value;
                        if (alpha >= beta)//alpha 剪枝
                            return alpha;
                    }

                }
            return alpha;
        }
        else
        {
            int i, j;
            for (i = 0; i < 15; i++)
                for (j = 0; j < 15; j++)
                {
                    if (curState[i][j] == NO)
                    {

                        point.x = i;
                        point.y = j;
                        value = AlphaBeta(curState, point, alpha, beta, depth + 1, 1);
                        if (beta > value)
                            beta = value;
                        if (alpha >= beta)//beta 剪枝
                            return beta;
                    }

                }
            return beta;
        }
    }
}

/*计算下一步棋子的位置*/
Point AI::GetNextPoint(const int chesspad[15][15], int color)
{
    int me = color + 1; //棋盘标志
    int rival;

    int MyScore[15][15] = { 0 };    //我的分数矩阵
    int RivalScore[15][15] = { 0 }; //对手的分数矩阵
    int temp[15][15] = { 0 };       //临时棋局状态

    //判断是否第一次下棋
    int flag = 0;
    int k = 0, h = 0;
    for (k = 0; k < 15; k++) {
        for (h = 0; h < 15; h++) {
            if (chesspad[k][h] > 0) {
                flag = 1;
                break;
            }
        }
        if (flag)
            break;
    }
    if (k == 15 && h == 15) { //第一次下棋
        Point point = { 7, 7 }; //默认最中间
        return point;
    }

    //把最后一步标志还原
    for (int i = 0; i < 15; i++) {
        for (int j = 0; j < 15; j++) {
            if (chesspad[i][j] > 2)
                temp[i][j] = chesspad[i][j] - 2;
            else
                temp[i][j] = chesspad[i][j];
        }
    }

    int max = -INF;
    int score;

    Point point;
    for (int i = 0; i < 15; i++)
    {
        for (int j = 0; j < 15; j++)
        {
            point.x = i;
            point.y = j;
            score = AlphaBeta(temp, point, 0, -inf, inf, me);
            if (score > max)
            {
                max = score;
                point.x = i;
                point.y = j;
            }
        }
        cout << endl;
    }
    return point;
 }

/*棋盘状态估分*/
int AI::Evaluate(const int state[15][15], Point point, int color)
{
    Situation status = { 0 }; //对当前棋盘状态棋型记录
    if (state[point.x][point.y])
```

```
        return class15; //此处不空，不能下棋子

    for (int direction = 0; direction < 4; direction++) { //四个方向,0 横，1 竖，2 左上
右下，3 右上左下
        int type;
        type = GetLinearType(state, point, color, direction); //取得某个方向上的线性分
析类型

        switch (type) { //根据类型对 Situation 设置
        case WIN5:
            status.win5++;
            break;
        case ALIVE4:
            status.alive4++;
            break;
        case RUSH4_1:
            status.rush4_1++;
            break;
        case RUSH4_2:
            status.rush4_2++;
            break;
        case ALIVE3:
            status.alive3++;
            break;
        case SLEEP3_2:
            status.sleep3_2++;
            break;
        case SLEEP3_1:
            status.sleep3_1++;
            break;
        case ALIVE2:
            status.alive2++;
            break;
        case SLEEP2_2:
            status.sleep2_2++;
            break;
        case SLEEP2_1:
            status.sleep2_1++;
            break;
        case NOTHREAT:
            status.nothreat++;
            break;
        default:
            break;
        }
    }
    return Score(status); //返回棋局状态评估分数
}

/*等级评分*/
int AI::Score(Situation status)
{
    int rush4_1 = status.rush4_1 + status.rush4_2;
    int alive3 = status.alive3 + status.sleep3_2;
    int alive2 = status.alive2 + status.sleep2_2;

    if (status.win5 >= 1)
        return class1; //赢5

    if (status.alive4 >= 1 || rush4_1 >= 2 || (rush4_1 >= 1 && alive3 >= 1))
        return class2; //活4 双冲4 冲4 活3

    if (alive3 >= 2)
        return class3; //双活3

    if (status.sleep3_1 >= 1 && status.alive3 >= 1)
        return class4; //眠3 高级活3

    if (status.rush4_1 >= 1)
        return class5; //冲41

    if (status.rush4_2 >= 1)
        return class6; //冲42

    if (status.alive3 >= 1)
        return class7; //单活3

    if (status.sleep3_2 >= 1)
        return class8; //眠3（跳活3）

    if (status.alive2 >= 2)
        return class9; //双活2

    if (status.alive2 >= 1)
        return class10; //活2

    if (status.sleep2_2 >= 1)
        return class11; //眠22
    if (status.sleep3_1 >= 1)
        return class12; //眠 3

    if (status.sleep2_1 >= 1)
        return class13; //眠 21

    return class14; //死 4,3,2,1
}

/*得到某个方向的线性模型*/
int AI::GetLinearType(const int state[15][15], Point point, int color, int
direction)
{
    int type;
    int chess[9] = { 0 };
    LinearAnalysis(chess, state, point, color, direction);
    type = JudgeChessType(chess);
    return type;
}

/*将某个方向的棋子进行线性分析*/
void AI::LinearAnalysis(int chess[9], const int state[15][15], Point point, int
color, int direction)
{
    int rival; //若超出边界则用对手的棋子颜色填充
    if (color == BLACK)
        rival = WHITE;
    else
        rival = BLACK;
    chess[4] = color;
    switch (direction) {
    case 0: //横向

        for (int i = point.x, j = 1; j <= 4; j++) { //往左拷贝四个
            int column = point.y - j;
            if (column < 0) {
                for (; j <= 4; j++)
                    chess[4 - j] = rival; //出界设置对手颜色
                break;
            }
            chess[4 - j] = state[i][column]; //没出界，复制 state 数组
        }
        for (int i = point.x, j = 1; j <= 4; j++) { //往右拷贝四个
```

```
            int column = point.y + j;
            if (column > 14) {
                for (; j <= 4; j++)
                    chess[4 + j] = rival; //出界设置对手颜色
                break;
            }
            chess[4 + j] = state[i][column]; //没出界，复制 state 数组
        }
        break;
    case 1:                                         //纵向
        for (int j = point.y, i = 1; i <= 4; i++) { //往上拷贝四个
            int row = point.x - i;
            if (row < 0) {
                for (; i <= 4; i++)
                    chess[4 - i] = rival; //出界设置对手颜色
                break;
            }
            chess[4 - i] = state[row][j]; //没出界，复制 state 数组
        }
        for (int i = 1, j = point.y; i <= 4; i++) { //往下拷贝四个
            int row = point.x + i;
            if (row > 14) {
                for (; i <= 4; i++)
                    chess[4 + i] = rival; //出界设置对手颜色
                break;
            }
            chess[4 + i] = state[row][j]; //没出界，复制 state 数组
        }
        break;
    case 2:                                        //左上
        for (int i = 1, j = 1; i <= 4; i++, j++) { //往左上拷贝四个
            int row = point.x - i;
            int column = point.y - j;
            if (row < 0 || column < 0) { //其中一个出边界
                for (; i <= 4; i++)
                    chess[4 - i] = rival; //出界设置对手颜色
                break;
            }
            chess[4 - i] = state[row][column]; //没出界，复制 state 数组
        }
        for (int i = 1, j = 1; i <= 4; i++, j++) { //往右下拷贝四个
            int row = point.x + i;
            int column = point.y + j;
            if (row > 14 || column > 14) { //其中一个出边界
```

```
                for (; i <= 4; i++)
                    chess[4 + i] = rival; //出界设置对手颜色
                break;
            }
            chess[4 + i] = state[row][column]; //没出界，复制state 数组
        }
        break;
    case 3:                                       //右上
        for (int i = 1, j = 1; i <= 4; i++, j++) { //往左下拷贝四个
            int row = point.x + i;
            int column = point.y - j;
            if (row > 14 || column < 0) { //其中一个出边界
                for (; i <= 4; i++)
                    chess[4 - i] = rival; //出界设置对手颜色
                break;
            }
            chess[4 - i] = state[row][column]; //没出界，复制state 数组
        }
        for (int i = 1, j = 1; i <= 4; i++, j++) { //往右上拷贝四个
            int row = point.x - i;
            int column = point.y + j;
            if (row < 0 || column > 14) { //其中一个出边界
                for (; i <= 4; i++)
                    chess[4 + i] = rival; //出界设置对手颜色
                break;
            }
            chess[4 + i] = state[row][column]; //没出界，复制state 数组
        }
        break;
    default:
        break;
    }
}

/*由棋型种类可知，相连的九个棋子即可判断某个棋型*/
int AI::JudgeChessType(const int chess[9])
{
    int left, right;   //第一个个中心线相连的同色棋子间断的左右位置
    int Lc, Rc;        //和中心线相连同色棋子间断的左右两边棋子的颜色
    int count = 1;     //中心线两边相连的同色棋子个数，初始化为1
    int me = chess[4]; //我的棋子颜色
    int rival;         //对手棋子的颜色
    rival = (me == BLACK) ? WHITE : BLACK;
```

```
for (int i = 1; i <= 4; i++) {
    if (chess[4 - i] == me)
        count++;
    else {
        left = 4 - i;      //保存间断点
        Lc = chess[4 - i]; //保存间断点颜色
        break;
    }
}
for (int i = 1; i <= 4; i++) {
    if (chess[4 + i] == me)
        count++;
    else {
        right = 4 + i;     //保存间断点
        Rc = chess[4 + i]; //保存间断点颜色
        break;
    }
}

if (count >= 5)  //中心线 5 连
    return WIN5; //5 连珠
if (count == 4)  //中心线 4 连
{
    if (Lc == NO && Rc == NO)           //两边断开位置均空
        return ALIVE4;                 //活四
    else if (Lc == rival && Rc == rival) //两边断开位置均非空
        return NOTHREAT;               //没有威胁
    else if (Lc == NO || Rc == NO)      //两边断开位置只有一个空
        return RUSH4_1;                //死四
}
if (count == 3) { //中心线 3 连
    int Lc1 = chess[left - 1];
    int Rc1 = chess[right + 1];

    if (Lc == NO && Rc == NO) //两边断开位置均空
    {
        if (Lc1 == rival && Rc1 == rival) //均为对手棋子
            return SLEEP3_1;
        else if (Lc1 == me || Rc1 == me) //只要一个为自己的棋子
            return RUSH4_2;
        else if (Lc1 == NO || Rc1 == NO) //只要有一个空
            return ALIVE3;

    }
```

```
        else if (Lc == rival && Rc == rival) //两边断开位置均非空
        {
            return NOTHREAT;            //没有威胁
        }
        else if (Lc == NO || Rc == NO) //两边断开位置只有一个空
        {
            if (Lc == rival) {    //左边被对方堵住
                if (Rc1 == rival) //右边也被对方堵住
                    return NOTHREAT;
                if (Rc1 == NO) //右边均空
                    return SLEEP3_1;
                if (Rc1 == me)
                    return RUSH4_2;
            }
            if (Rc == rival) {    //右边被对方堵住
                if (Lc1 == rival) //左边也被对方堵住
                    return NOTHREAT;
                if (Lc1 == NO) //左边均空
                    return SLEEP3_1;
                if (Lc1 == me) //左边还有自己的棋子
                    return RUSH4_2;
            }
        }
    }
    if (count == 2) { //中心线 2 连
        int Lc1 = chess[left - 1];
        int Rc1 = chess[right + 1];
        int Lc2 = chess[left - 2];
        int Rc2 = chess[right + 2];

        if (Lc == NO && Rc == NO) //两边断开位置均空
        {
            if ((Rc1 == NO && Rc2 == me) || (Lc1 == NO && Lc2 == me))
                return SLEEP3_1; //死 3
            else if (Lc1 == NO && Rc1 == NO)
                return ALIVE2; //活 2
            if ((Rc1 == me && Rc2 == rival) || (Lc1 == me && Lc2 == rival))
                return SLEEP3_1; //死 3
            if ((Rc1 == me && Rc2 == me) || (Lc1 == me && Lc2 == me))
                return RUSH4_2; //冲 4
            if ((Rc1 == me && Rc2 == NO) || (Lc1 == me && Lc2 == NO))
                return SLEEP3_2; //眠三->跳活 3
            //其他情况在下边返回 NOTHREAT
        }
        else if (Lc == rival && Rc == rival) //两边断开位置均非空
            return NOTHREAT;
        else if (Lc == NO || Rc == NO) //两边断开位置只有一个空
        {
            if (Lc == rival) {                      //左边被对方堵住
                if (Rc1 == rival || Rc2 == rival) {  //只要有对方的一个棋子
                    return NOTHREAT;                //没有威胁
                }
                else if (Rc1 == NO && Rc2 == NO) { //均空
                    return SLEEP2_1;                //眠2
                }
                else if (Rc1 == me && Rc2 == me) { //均为自己的棋子
                    return RUSH4_2;                 //冲4
                }
                else if (Rc1 == me || Rc2 == me) { //只有一个自己的棋子
                    return SLEEP3_1;                //眠3
                }
            }
            if (Rc == rival) {                      //右边被对方堵住
                if (Lc1 == rival || Lc2 == rival) {  //只要有对方的一个棋子
                    return NOTHREAT;                //没有威胁
                }
                else if (Lc1 == NO && Lc2 == NO) { //均空
                    return SLEEP2_1;                //死2
                }
                else if (Lc1 == me && Lc2 == me) { //均为自己的棋子
                    return RUSH4_2;                 //冲4
                }
                else if (Lc1 == me || Lc2 == me) { //只有一个自己的棋子
                    return SLEEP3_1;                //眠3
                }
            }
        }
    }
    if (count == 1) { //中心线1连
        int Lc1 = chess[left - 1];
        int Rc1 = chess[right + 1];
        int Lc2 = chess[left - 2];
        int Rc2 = chess[right + 2];
        int Lc3 = chess[left - 3];
        int Rc3 = chess[right + 3];

        if (Lc == NO && Lc1 == me && Lc2 == me && Lc3 == me)
            return RUSH4_2;
        if (Rc == NO && Rc1 == me && Rc2 == me && Rc3 == me)
            return RUSH4_2;

        if (Lc == NO && Lc1 == me && Lc2 == me && Lc3 == NO && Rc == NO)
            return SLEEP3_2;
        if (Rc == NO && Rc1 == me && Rc2 == me && Rc3 == NO && Lc == NO)
            return SLEEP3_2;

        if (Lc == NO && Lc1 == me && Lc2 == me && Lc3 == rival && Rc == NO)
            return SLEEP3_1;
        if (Rc == NO && Rc1 == me && Rc2 == me && Rc3 == rival && Lc == NO)
            return SLEEP3_1;

        if (Lc == NO && Lc1 == NO && Lc2 == me && Lc3 == me)
            return SLEEP3_1;
        if (Rc == NO && Rc1 == NO && Rc2 == me && Rc3 == me)
            return SLEEP3_1;
        if (Lc == NO && Lc1 == me && Lc2 == NO && Lc3 == me)
            return SLEEP3_1;
        if (Rc == NO && Rc1 == me && Rc2 == NO && Rc3 == me)
            return SLEEP3_1;

        if (Lc == NO && Lc1 == me && Lc2 == NO && Lc3 == NO && Rc == NO)
            return SLEEP2_2;
        if (Rc == NO && Rc1 == me && Rc2 == NO && Rc3 == NO && Lc == NO)
            return SLEEP2_2;

        if (Lc == NO && Lc1 == NO && Lc2 == me && Lc3 == NO && Rc == NO)
            return SLEEP2_2;
        if (Rc == NO && Rc1 == NO && Rc2 == me && Rc3 == NO && Lc == NO)
            return SLEEP2_2;
    }
    return NOTHREAT; //返回没有威胁
}
```

2. AI_Chesser.cpp

```
AI_Chesser::AI_Chesser(int color)
{
    this->color = color;
}

Chess AI_Chesser::GiveNextChess(const int ChessPad[15][15], int &x, int &y)
{
    Point p;
    p = AI_player.GetNextPoint(ChessPad, color);
    x = p.x;
    y = p.y;
    return Chess(color, p.x, p.y);
}
```

3. Chess. cpp

```
#include "Chess.h"
Chess::Chess(int color, int x, int y)
{
    this->color = color;
    this->x = x;
    this->y = y;
}

int Chess::GetColor()
{
    return color;
}

void Chess::GetPoint(int &x, int &y)
{
    x = this->x;
    y = this->y;
}

void Chess::SetColor(int color)
{
    this->color = color;
}

void Chess::SetPoint(int x, int y)
{
    this->x = x;
    this->y = y;
}
```

4. ChessBoard. cpp

```
#include "ChessBoard.h"
ChessBoard::ChessBoard()
{
    LastX = 0;
    LastY = 0;
    for (int i = 0; i < 15; i++)
        for (int j = 0; j < 15; j++)
            ChessPadStatus[i][j] = 0;
}
void ChessBoard::AddChess(Chess chess)
{
    if (ChessPadStatus[LastX][LastY])//有棋子
        ChessPadStatus[LastX][LastY] -= 2;//不是最后一步了

    chess.GetPoint(LastX, LastY); //LastX=chess.x;LastY=chess.y;

    if (chess.GetColor()) //  color=chess.color=this.color;  color=1-white  color=0-
black
        ChessPadStatus[LastX][LastY] = 4;
    else
        ChessPadStatus[LastX][LastY] = 3;
}

void ChessBoard::GetChessStatus(int ChessPadStatus[15][15])
{
    for (int i = 0; i < 15; i++)
        for (int j = 0; j < 15; j++)
            ChessPadStatus[i][j] = this->ChessPadStatus[i][j];
}
```

5. Judge. cpp

```
#include "Judge.h"
Judge::Judge()
{
    CurChesser = 1; //white fang
}

int Judge::JudgeResult(const int chesspadstate[15][15])
{
    int lastrow, lastcolumn;
    int i, j;
    int count, result;
    int rowmin, rowmax, columnmin, columnmax;

    int flag = 0;
    for (i = 0; i < 15; i++) {
```

```
    for (j = 0; j < 15; j++) {
        if (chesspadstate[i][j] == 0) {
            flag = 1;
            break;
        }
    }
    if (flag)
        break;
}
if (i == 15 && j == 15)
    return 3; //和局

flag = 0;
for (i = 0; i < 15; i++) { //最后一步的坐标
    for (j = 0; j < 15; j++)
        if (chesspadstate[i][j] > 2) {
            lastrow = i;
            lastcolumn = j;
            result = chesspadstate[i][j] - 2; //返回当前旗手赢的标志
            flag = 1;
            break;
        }
    if (flag)
        break;
}
if (i == 15 && j == 15) //还没开始下棋
    return 0;

//横向
count = 0;
columnmin = lastcolumn - 4 < 0 ? 0 : lastcolumn - 4;
columnmax = lastcolumn + 4 > 14 ? 14 : lastcolumn + 4;
for (i = lastrow, j = columnmin; j <= columnmax; j++) {
    if (chesspadstate[i][j] == result
        || chesspadstate[i][j] == result + 2) { //返回结果标志和棋标志相同
        count++;
        if (count == 5) //赢了
            return result;
    } else
        count = 0; //重头数起
}

//纵向
count = 0;
```

```
rowmin = lastrow - 4 < 0 ? 0 : lastrow - 4;
rowmax = lastrow + 4 > 14 ? 14 : lastrow + 4;
for (i = rowmin, j = lastcolumn; i <= rowmax; i++) {
    if (chesspadstate[i][j] == result
        || chesspadstate[i][j] == result + 2) { //返回结果标志和棋标志相同
        count++;
        if (count == 5) //赢了
            return result;
    } else
        count = 0; //重头数起
}

//西北斜
count = 0;
rowmin = lastrow - 4;
columnmin = lastcolumn - 4;
if (rowmin < 0 || columnmin < 0) {     //出界
    if (lastrow > lastcolumn) {       //出界步数小先出界
        columnmin = 0;                //先出界的为边界值
        rowmin = lastrow - lastcolumn; //后出界的根据斜率1
    } else {
        rowmin = 0;
        columnmin = lastcolumn - lastrow;
    }
}
rowmax = lastrow + 4;
columnmax = lastcolumn + 4;
if (rowmax > 14 || columnmax > 14) {      //出界
    if (14 - lastrow < 14 - lastcolumn) { //出界步数小先出界
        rowmax = 14;                     //先出界的为边界值
        columnmax = lastcolumn + 14 - lastrow;
    } else {
        columnmax = 14;
        rowmax = lastrow + 14 - lastcolumn;
    }
}
for (i = rowmin, j = columnmin; i <= rowmax; i++, j++) {
    if (chesspadstate[i][j] == result
        || chesspadstate[i][j] == result + 2) { //返回结果标志和棋标志相同
        count++;
        if (count == 5) //赢了
            return result;
    } else
        count = 0; //重头数起
    }

    //东北斜
    count = 0;
    rowmin = lastrow - 4;
    columnmax = lastcolumn + 4;
    if (rowmin < 0 || columnmax > 14) {      //出界
        if (lastrow - 0 < 14 - lastcolumn) { //出界步数小先出界
            rowmin = 0;                      //先出界为边界值
            columnmax = lastcolumn + lastrow;
        } else {
            columnmax = 14;
            rowmin = lastrow - (14 - lastcolumn);
        }
    }
    rowmax = lastrow + 4;
    columnmin = lastcolumn - 4;
    if (rowmax > 14 || columnmin < 0) {      //出界
        if (14 - lastrow < lastcolumn - 0) { //出界步数小先出界
            rowmax = 14;
            columnmin = lastcolumn - (14 - lastrow);
        } else {
            columnmin = 0;
            rowmax = lastrow + lastcolumn - 0;
        }
    }
    for (i = rowmin, j = columnmax; i <= rowmax; i++, j--) {
        if (chesspadstate[i][j] == result
            || chesspadstate[i][j] == result + 2) { //返回结果标志和棋标志相同
            count++;
            if (count == 5) //赢了
                return result;
        } else
            count = 0; //重头数起
    }
    return 0; //未分胜负
}

int Judge::JudgeNextChesser()
{
    return CurChesser = (CurChesser == 1) ? 0 : 1;
}
```

6. PE_Chesser

```cpp
#include "PE_Chesser.h"
PE_Chesser::PE_Chesser(int color)
{
    this->color = color;
}

Chess PE_Chesser::GiveNextChess(int x, int y)
{
    return Chess(color, x, y);
}
```

7. MainWindow1.cpp

```cpp
#include "mainwindow1.h"
#include "mainwindow.h"
#include "ui_mainwindow1.h"
#include <iostream>
#include <QTimer>
using namespace std;
MainWindow1::MainWindow1(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow1)
{
    ui->setupUi(this);
    setFixedSize(800, 500);
}

MainWindow1::~MainWindow1()
{
    delete ui;
}
void on_Qbg1_toggled(int, bool) {}
void MainWindow1::on_checkBox_clicked(bool checked)
{
    if (checked) {
        m->mode = 0; //人机
        flag1 = 1;
    }
}
void MainWindow1::on_checkBox_2_clicked(bool checked)
{
    if (checked) {
        m->mode = 1; //人人
        m->player = 1;
        flag2 = 1;
    }
}

void MainWindow1::on_radioButton_clicked(bool checked)
{
    if (checked) {
        m->fp = 0; //AI
        m->PvAflag = 1;
        flag3 = 1;
    }
}

void MainWindow1::on_radioButton_2_clicked(bool checked)
{
    if (checked) {
        m->fp = 1; //PE
        flag5 = 1;
    }
}

void MainWindow1::on_radioButton_3_clicked(bool checked)
{
    if (checked) {
        m->fp = 0;
        flag4 = 1;
    }
}

void MainWindow1::on_radioButton_4_clicked(bool checked)
{
    if (checked) {
        m->fp = 1;
        flag6 = 1;
    }
}
void MainWindow1::on_pushButton_2_clicked()
{
    if ((flag1 == 1 && (flag3 || flag5)) || (flag2 && (flag4 || flag6)))
        m->show();
    else
        ;
}
```

8. MainWindow.cpp(主要函数有 AI 和 PE 下棋函数，其余的为鼠标点击事件和绘制棋盘事件)

```cpp
void MainWindow::AI_Chess() //ai's turn to add a chess
{
    if (mode == 0 && fp == 0) {
        AI_Chesser ai(0);              //ai is black
        if (!(judge.JudgeNextChesser())) //curchesser=0 ,ai'turn==black's turn
            chessboard.AddChess(ai.GiveNextChess(Status, currentX, currentY));
    }
    if (mode == 0 && fp == 1) {
        AI_Chesser ai(1);            //ai is white
        if (judge.JudgeNextChesser()) //curchesser=1, ai's turn==white's turn
            chessboard.AddChess(ai.GiveNextChess(Status, currentX, currentY));
    }
    update();
}
void MainWindow::PE_Chess() //pe's turn to add a chess
{
    if (Status[currentX][currentY] == 0) {
        if ((mode == 0 && fp == 0) || (mode == 1 && fp == 1 && player == 1)) {
            PE_Chesser pe(1);            //pe is white
            if (judge.JudgeNextChesser()) //1,curchesser=1,pe's turn==white's turn
                chessboard.AddChess(pe.GiveNextChess(currentX, currentY));
        }
        if ((mode == 0 && fp == 1) || (mode == 1 && fp == 0 && player == 0)) {
            PE_Chesser pe(0);               //pe is black
            if (!(judge.JudgeNextChesser())) //0,curchesser=0,pe's turn==black's turn
                chessboard.AddChess(pe.GiveNextChess(currentX, currentY));
        }
        if (mode == 1 && fp == 0 && player == 1) {
            PE_Chesser pe1(1); //pe1 is white
            if (judge.JudgeNextChesser())
                chessboard.AddChess(pe1.GiveNextChess(currentX, currentY));
        }
        if (mode == 1 && fp == 1 && player == 0) {
            PE_Chesser pe1(0); //pe1 is black
            if (!(judge.JudgeNextChesser()))
                chessboard.AddChess(pe1.GiveNextChess(currentX, currentY));
        }
        update();
    }
}
```

七、 参考文章

1. 部分代码实现方法：
https://github.com/cstackess/Gobang
2. 五子棋型：
https://blog.csdn.net/zjh776/article/details/84506350
https://blog.csdn.net/mahabharata_/article/details/79511727
3. alpha-beta 剪枝算法参考：
https://www.cnblogs.com/tk55/articles/6012314.html