五子棋人机博弈问题

摘 要

五子棋博弈是一种两人对弈的纯策略型棋类游戏，由横线和竖线的棋盘构成，下在线与线的交叉点上，先形成五子连珠者获胜，是一个经典的人工智能博弈问题的游戏。本题种要求设计AI，能与人进行对弈，博弈树的创建和搜索中利用到了 alpha-beta 剪枝算法。本实验即为基于博弈树的启发式搜索、alpha-beta 剪枝算法和评价函数，开发了一个五子棋人机博弈游戏，其中设计的数据结构可以评估棋势、选择落子、判断胜负，又利用 Qt 设计了美观的 ui 界面。

关键词：五子棋，alpha-beta 剪枝算法，博弈树，人工智能

装

订

线

目 录

装 订 线

1 实验概述

1.1 实验目的

熟悉和掌握博弈树的启发式搜索过程、α-β 剪枝算法和评价函数，并利用 α-β 剪枝算法开发一个五子棋人机博弈游戏。

1.2 实验内容

1.2.1 五子棋游戏程序的实现

以五子棋人机博弈问题为例，实现 α-β 剪枝算法的求解程序（编程语言不限），要求设计适合五子棋博弈的评估函数。五子棋游戏的规则为，对局双方各执一色棋子，空棋盘开局，黑先、白后，交替下子，每次只能下一子；棋子下在棋盘直线与横线的交叉点上，先形成五子连线者获胜。

1.2.2 界面显示

要求初始界面显示 15*15 的空白棋盘，电脑执白棋，人执黑棋，界面置有重新开始、悔棋等操作。

1.2.3 数据结构设计

设计五子棋程序的数据结构，具有评估棋势、选择落子、判断胜负等功能。

1.2.4 提交要求

撰写实验报告，提交源代码（进行注释）、实验报告、汇报 PPT。

1.3 本实验所做的工作

本实验基于 Qt 平台，利用 alpha-beta 算法，设计了合适的评估函数求解了五子棋人机博弈问题，具有评估棋势，选择落子，判断胜负的功能。其中，还加入了悔棋操作和认输（即结束游戏，重新开始）的功能，以及加入了双人模式，即两人博弈的功能。界面显示方面，利用了 Qt 平台设计了一个美观的 ui 界面，并配有音效。

2 实验方案设计

2.1 总体设计思路与总体架构

在这次实验的设计中，总体可以分为四个模块：

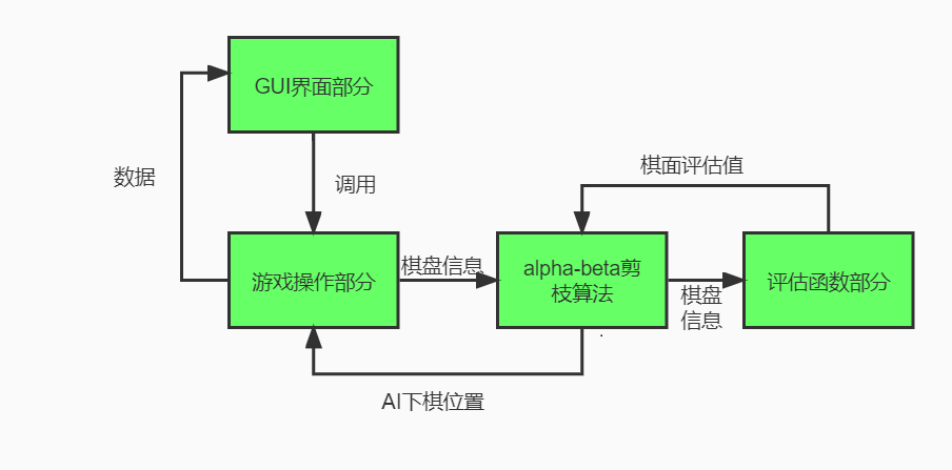
（1） alpha-beta 剪枝部分，即利用 alpha-beta 剪枝算法计算出最佳下棋位置的部分。

（2） 评估函数部分，即根据棋面信息算出当前棋局的某方评估值的部分。

（3） GUI 界面部分，即基于 Qt 平台实现的 ui 界面的代码部分。

（4） 游戏操作部分，即衔接每个动作完成总体游戏功能的部分

关系图如下：



这四个模块相辅相成实现了最终的实验成果，在 2.3 中将分别介绍这四个模块的具体设计

2.2 核心算法及基本原理

2.2.1 Alpha-beta 剪枝算法

Alpha-beta 剪枝算法是在极大极小值算法的基础上，尽可能地消除部分搜索树，使得搜索的效率更高，具体遵循的原理为：

（1）alpha 剪枝：任何极小层（由 MIN 落子）的节点的 beta 值都不大于其前驱节点（MAX 节点）的 alpha 值，即搜索过程中，只要找到一个 MIN 节点的评估值不大于其前驱 MAX 节点的评估值，则可舍弃后续的搜索，这表示当前 MIN 节点落子对 MAX 是有利的。

（2）beta 剪枝：任何极大层（由 MAX 落子）的节点的 alpha 值都不小于其前驱节点（MIN 节点）的 beta 值。即搜索过程中，只要找到一个 MAX 节点的评估值不小于其前驱 MIN 节点的评估值，则可舍弃后续的搜索，这表示当前 MAX 节点落子对 MAX 是有利的。

算法伪代码如下：

```
function  alphaBeta(node, alpha, beta , depth)
    if node is a terminal node or depth = 0
        return the evaluate value of node  //使用评估函数返回局面得分
    else
        if AI's turn
```

```
            foreach child of node
            val := alphaBeta(child, alpha, beta, depth-1)
            if(val > alpha)    alpha:= val
            if(alpha >= beta)    break
        return alpha
    else player's turn
        foreach child of node
            val := alphaBeta(child, alpha, beta, depth-1)
            if(val < beta)  beta:= val
            if(alpha >= beta)    break
        return b
```

2.2.2 五子棋棋形原理

在五子棋AI的设计中,尽管采用什么样的搜索算法很重要,但最重要的还是评估函数的设计，只有设计出合理的评估函数，才能使得 AI 的选择变得更加正确。五子棋中有很多种棋形，从研究中可以看到有几种重要棋形如下：

棋形名称	棋形估值
活四	300000
死四	2500
活三	3000
死三	1000
活二	800
死二	300
五	1000000

表格 1

通过参考表格 1 中的棋形知识，我们可以更好地设计评估函数，在评估函数部分会更详细的说明。

2.2.3 局部搜索提高效率

根据五子棋的游戏规则可知，一般而言，应将棋下在周围有棋子的地方，否则是没有意义的。因此在搜索的时候可以忽略周围一定范围内没有棋子的地方，进而提高效率。本程序中设计了一个函数 canSearch,即对于某处（x,y），周围 1 格有棋子则进行搜索。具体的算法是遍历他周围的八个位置，若有棋子则返回 true，否则返回 false。

2.3 模块设计

2.3.1 alpha-beta 剪枝部分

这一模块利用了如 2.2.1 中所述 alpha-beta 算法，主要功能是从游戏操作部分传来的棋盘信息递归算得最佳落子位置，传回游戏操作部分。

2.3.2 评估函数部分

这一部分是本程序中至关重要的部分。首先我们将整个棋面分解为线，对于整个棋盘，我们可以按四个方向将棋盘转化为 15*6 个长度不超过 15 的一维向量（斜向的分为上下两个半区）。

再根据线状态进行评估，线状态的评分汇总加和即可得棋面的评分。

对线状态的评分即为对不同棋形的评估，可见 2.2.2 中的表格 1。基于表格 1 中所示棋形估值，即可设计好全部的评估函数。

2.3.3 GUI 界面部分

这一模块是基于 Qt 平台完成的，利用了 QPainter、QMouseEvent 等工具，实现了人机交互，以及添加了一些音效和弹窗，使得游戏界面更简洁美观。

这一部分的功能即为将人的操作和 AI 的操作显示出来，然后将信号传输给游戏操作模块，游戏操作模块再将具体的棋盘信息传给 GUI 界面部分。

2.3.4 游戏操作部分

这一部分主要设计了这一部分创建了一个名为 GameModel 的类，类中含有存储棋盘信息的变量以及对棋盘进行操作的函数，还包括搜索算法用到的函数。功能含有开始游戏、人下棋、机器下棋、更新棋盘、判断游戏是否胜利、判断是否和棋、悔棋等等。这一部分和其他的三个部分紧密相连，承接 GUI，调取 Alpha-beta 部分，进而调取评估函数部分，后两者都是基于 GameModel 类进行的操作。这一部分可以称之为游戏的主要框架。这里简要介绍一下悔棋操作的思路：存储上一步棋人和 AI 分别的落子位置，在 GUI 界面部分调取悔棋函数时，将该位置重置为 0，再在 GUI 界面打印出来，即可完成悔棋操作。

3 实验过程

3.1 环境说明

3.1.1 操作系统

64 位 Microsoft Windows 系统

3.1.2 开发语言

C++

3.1.3 开发环境

开发环境：Qt Creator(4.3.0)
核心使用库：QPainter、QMenu、QMouseEvent等

3.2 文件及主要函数清单

3.2.1 GameModel.cpp

```
void GameModel::startGame(GameType type)
void GameModel::updateGameMap(int row, int col)
void GameModel::actionByPerson(int row, int col)
bool GameModel::canSearch(int x, int y)
int GameModel::nextType(int type)
int GameModel::getPieceType(int A, int type)
int GameModel::getPieceType(int x, int y, int type)
int GameModel::evaluateLine(int line[], bool all)
int GameModel::evaluateLine(int line[])
int GameModel::getValue(int cnt, int blk)
int GameModel::evaluateState(int type)
int GameModel::evaluatePiece(int x, int y, int type)
int GameModel::minMax(int x, int y, int type, int depth, int alpha, int beta)
void GameModel::actionByAI(int& clickRow, int& clickCol)
bool GameModel::isWin(int row, int col)
bool GameModel::isDeadGame()
```

3.2.2 mainwindow.cpp

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
MainWindow::~MainWindow()
void MainWindow::initGame()
```

```
void MainWindow::initPVPGame()
void MainWindow::initPVEGame()
void MainWindow::retract()
void MainWindow::gotolost()
void MainWindow::paintEvent(QPaintEvent *event)
void MainWindow::mouseMoveEvent(QMouseEvent *event)
void MainWindow::mouseReleaseEvent(QMouseEvent *event)
void MainWindow::chessOneByPerson()
void MainWindow::chessOneByAI()
```

3.2.3 mainwindow.h

3.2.4 GameModel.h

3.2.5 main.cpp

```
int main(int argc, char *argv[])
```

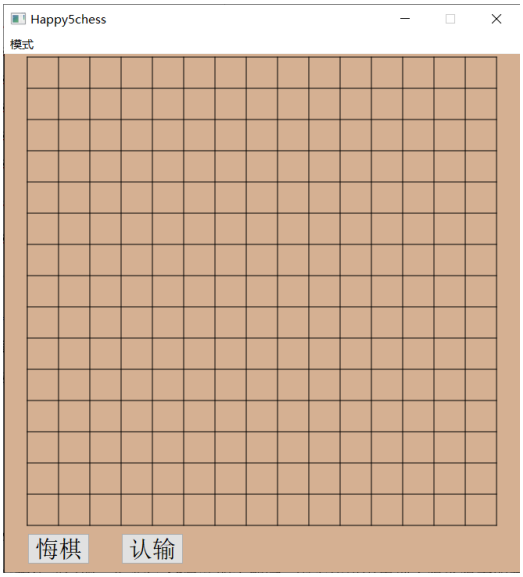
3.2.6 helloworld2.pro

3.2.7 其他资源文件

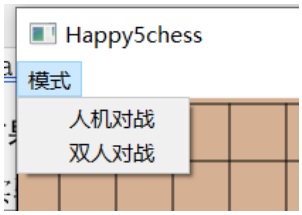
3.3 实验结果展示

3.3.1 界面展示

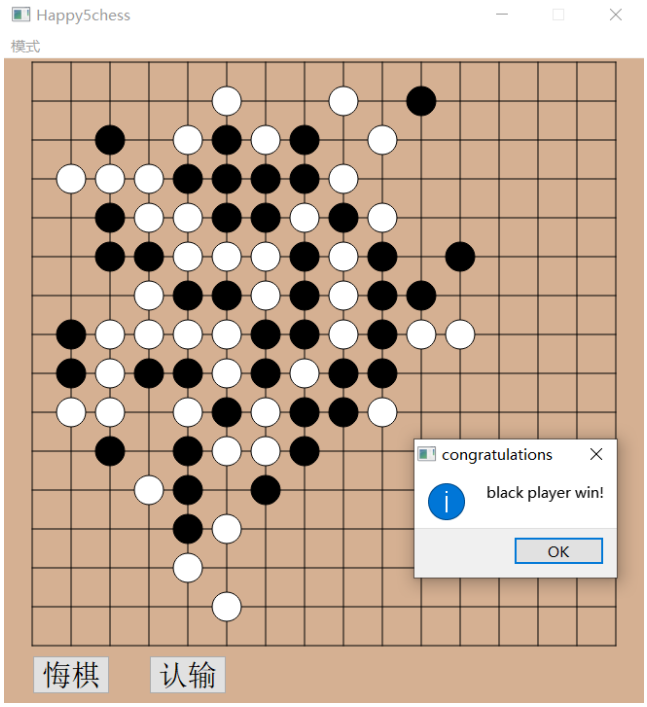
本次实验中设计的五子棋 AI 颇为智能，首先初始界面如下：



界面为 15*15 的棋盘格，并含有悔棋、认输功能，左上角的模式中点开还可切换人机对战和双人对战模式。如图所示：



然后就可以进行下棋了，在某方获胜的时候会弹出获胜消息，如图所示：

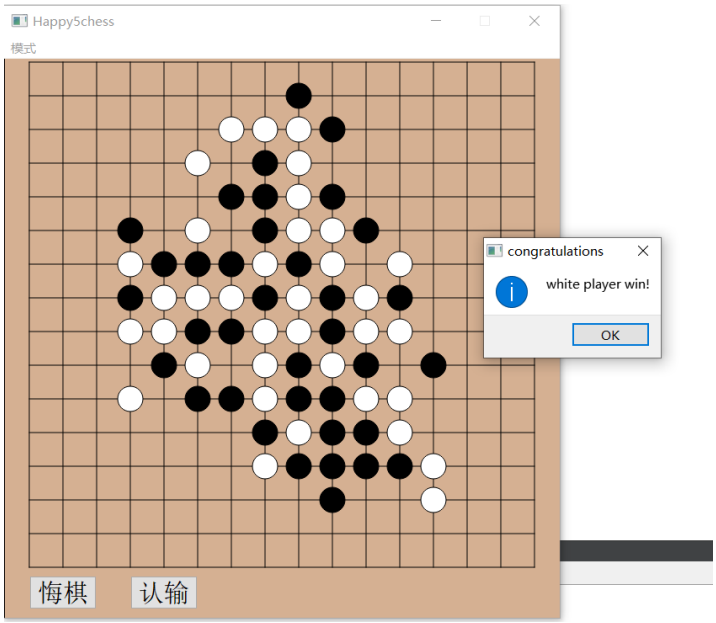


3.3.2 AI 搜索效率

进行了优化后的算法效率很高，速度很快，为了使得玩家感受到对局的真实性，还设置了一定的延迟，但延迟后也是 AI 迅速下棋，完全无需等待。

3.3.3 测试棋局对弈

实验者与本实验所设计的 AI 进行对决，在经过了激烈的对局后，AI 赢过了实验者



虽然棋艺还没有到达特别精湛的地步，但是也颇为智，可以说是顺利完成任务。

装 订 线

4 总结

4.1 实验中存在的问题及解决方案

1.搜索过慢问题。最开始进行搜索的时候，尽管采用了 alpha-beta 剪枝，但是速度还是有些慢。于是采用了局部搜索，即利用上文提到的 canSearch 函数进行筛选，只搜索部分位置，大大提高了效率，解决了此问题。

2.设计出的程序棋艺不精。最开始尽管搜索深度加深，但是 AI 的棋艺依旧很有限，有时还会出现乱下的情况。于是我考虑到这应该是评估函数设计的问题。在重新设计了评估函数并且不断改变对几种棋形的评估值之后，选择了最终的这种评估函数评估值，很大程度的提升了棋艺。

4.2 心得体会

通过这次的实验，我熟悉和掌握了博弈树的启发式搜索过程、alpha-beta 剪枝算法和五子棋评价函数的设计。除此之外，我还提高了自己的编程能力，尤其是类的设计。并且通过此次作业，我学会了使用 Qt 进行 UI 界面的设计，掌握了一些初级的 UI 界面的设计方式，感受到受益匪浅。并且此次评估函数的设计也非常复杂，增强了我思考问题的能力。

4.3 后续改进方向

五子棋作为一个很复杂的问题，其实还可以进行更多的完善。首先，评估函数还有许多可以提升的地方，比如加入算杀等更多的五子棋知识。其次，通过在网络上的资料寻找，发现网上有很多优秀的五子棋模型，他们也给我提供了后续优化的方向，比如加入更复杂的算法使得一些基本问题上面效率更高，比如数组的传送等等，借此达到优化的效果。后续可以先从评估函数下手进行优化，再优化一下其他部分的效率。界面上也可以进行一定程度的优化。

附件：源代码

GameModel.cpp

```
#include <utility>
#include <stdlib.h>
#include <time.h>
#include "GameModel.h"
/*空的构造函数*/
GameModel::GameModel()
{

}
/*开始游戏，对游戏进行初始化*/
void GameModel::startGame(GameType type)
{
    gameType = type;
    // 初始棋盘
    gameMapVec.clear();
    for (int i = 0; i < kBoardSizeNum; i++)
    {
        std::vector<int> lineBoard;
        for (int j = 0; j < kBoardSizeNum; j++)
            lineBoard.push_back(0);
        gameMapVec.push_back(lineBoard);
    }

    // 如果是 AI 模式，需要初始化评分数组
    if (gameType == BOT)
    {
        scoreMapVec.clear();
        for (int i = 0; i < kBoardSizeNum; i++)
        {
            std::vector<int> lineScores;
            for (int j = 0; j < kBoardSizeNum; j++)
                lineScores.push_back(0);
            scoreMapVec.push_back(lineScores);
        }
    }

    // 己方(人黑)下为 true,对方(AI 白)下为 false
    playerFlag = true;
    //悔棋设置为零
    lastcolAI=0;
    lastcolPer=0;
    lastrowAI=0;
    lastrowPer=0;
    belost=0;
```

```
    gameMapTmp=gameMapVec;
}
/*落子后更新游戏棋盘*/
void GameModel::updateGameMap(int row, int col)
{
    if (playerFlag)
        gameMapVec[row][col] = -1;//黑子
    else
        gameMapVec[row][col] = 1;//白子

    // 换手
    playerFlag = !playerFlag;
}
/*人下*/
void GameModel::actionByPerson(int row, int col)
{
    if(playerFlag){//黑子
        lastrowPer=row;
        lastcolPer=col;
    }
    else
    {
        lastrowPer2=row;
        lastcolPer2=col;
    }
    updateGameMap(row, col);
}
/*若 xy 位置周围 1 格有棋子则搜索*/
bool GameModel::canSearch(int x, int y) {

    int tmpx = x - 1;
    int tmpy = y - 1;
    for (int i = 0; tmpx < kBoardSizeNum && i < 3; ++tmpx, ++i) {
        int ty = tmpy;
        for (int j = 0; ty < kBoardSizeNum && j < 3; ++ty, ++j) {
           if (tmpx >= 0 && ty >= 0 && gameMapTmp[tmpx][ty]!=0)
               return true;
           else
               continue;
        }
    }
        return false;
}

/*给出后继节点的类型，type 为棋子类型*/
int GameModel::nextType(int type) {
    return type == MAX_NODE ? MIN_NODE : MAX_NODE;
}
```

```
/*A 是带判断棋子，type 是此时我方棋子的类型，返回我方、空、敌方*/
int GameModel::getPieceType(int A, int type) {
    return A == type ? AI_MY : (A == 0 ? AI_EMPTY : AI_OP);
}
int GameModel::getPieceType(int x, int y, int type) {
    if (x < 0 || y < 0 || x >= kBoardSizeNum || y >= kBoardSizeNum)// 超出边界按敌方
棋子算
        return AI_OP;
    else
        return getPieceType(gameMapTmp[x][y], type);
}

/*以 center 作为评估位置进行评价一个方向的棋子*/
int GameModel::evaluateLine(int line[], bool all) {
    int value = 0; //估值
    int cnt = 0; // 连子数
    int blk = 0; // 封闭数
    if(all)
        cnt=0;
    for (int i = 0; i < kBoardSizeNum; ++i) {
        if (line[i] == AI_MY) { // 找到第一个己方的棋子
                                // 还原计数
            cnt = 1;
            blk = 0;
            // 看左侧是否封闭
            if (line[i - 1] == AI_OP)
                ++blk;
            // 计算连子数
            for (i = i + 1; i < kBoardSizeNum && line[i] == AI_MY; ++i, ++cnt);
            // 看右侧是否封闭
            if (line[i] == AI_OP)
                ++blk;
            // 计算评估值
            value += getValue(cnt, blk);
        }
    }
    return value;
}

/*以 center 为评估评价一个方向的棋子（前后四格）*/
int GameModel::evaluateLine(int line[]) {
    int cnt = 1; // 连子数
    int blk = 0; // 封闭数
                 // 向左右扫
    for (int i = 3; i >= 0; --i) {
        if (line[i] == AI_MY) ++cnt;
        else if (line[i] == AI_OP) {
            ++blk;
```

```
            break;
        }
        else
            break;
    }
    for (int i = 5; i < 9; ++i) {
        if (line[i] == AI_MY) ++cnt;
        else if (line[i] == AI_OP) {
            ++blk;
            break;
        }
        else
            break;
    }
    return getValue(cnt, blk);
}

/*根据连子数和封堵数给出一个评价值*/
int GameModel::getValue(int cnt, int blk) {
    if (blk == 0) {//活
        switch (cnt) {
        case 1:
            return AI_ONE;
        case 2:
            return AI_TWO;
        case 3:
            return AI_THREE;
        case 4:
            return AI_FOUR;
        default:
            return AI_FIVE;
        }
    }
    else if (blk == 1) {//单项封死
        switch (cnt) {
        case 1:
            return AI_ONE_S;
        case 2:
            return AI_TWO_S;
        case 3:
            return AI_THREE_S;
        case 4:
            return AI_FOUR_S;
        default:
            return AI_FIVE;
        }
    }
    else {//双向封死
```

```
        if (cnt >= 5)
            return AI_FIVE;
        else
            return AI_ZERO;
    }
}

/*评价一个其面上的一方*/
int GameModel::evaluateState(int type) {
    int value = 0;
    // 分解成线状态
    int line[6][17];
    int lineP;
    for (int p = 0; p < 6; ++p)
        line[p][0] = line[p][16] = AI_OP;

      //从四个方向产生
    for (int i = 0; i < kBoardSizeNum; ++i) {
          //产生先状态
        lineP = 1;

        for (int j = 0; j < kBoardSizeNum; ++j) {
            line[0][lineP] = getPieceType(i,j, type); /* | */
            line[1][lineP] = getPieceType(j,i, type); /* - */
            line[2][lineP] = getPieceType(i + j,j, type); /* \ */
            line[3][lineP] = getPieceType(i - j,j, type); /* / */
            line[4][lineP] = getPieceType(j,i + j, type); /* \ */
            line[5][lineP] = getPieceType(kBoardSizeNum - j - 1,i + j, type); /* / */
            ++lineP;
        }
        // 估计
        int special = i == 0 ? 4 : 6;
        for (int p = 0; p < special; ++p) {
            value += evaluateLine(line[p], true);
        }
    }
    return value;
}
/*对一个状态的一个位置放置一种类型的棋子的优劣进行估价*/
int GameModel::evaluatePiece(int x, int y, int type) {
    int value = 0; // 估价值
    int line[17];  //线状态
    bool flagX[8];// 横向边界标志
    flagX[0] = x - 4 < 0;
    flagX[1] = x - 3 < 0;
    flagX[2] = x - 2 < 0;
    flagX[3] = x - 1 < 0;
    flagX[4] = x + 1 > 14;
```

```
flagX[5] = x + 2 > 14;
flagX[6] = x + 3 > 14;
flagX[7] = x + 4 > 14;
bool flagY[8];// 纵向边界标志
flagY[0] = y - 4 < 0;
flagY[1] = y - 3 < 0;
flagY[2] = y - 2 < 0;
flagY[3] = y - 1 < 0;
flagY[4] = y + 1 > 14;
flagY[5] = y + 2 > 14;
flagY[6] = y + 3 > 14;
flagY[7] = y + 4 > 14;

line[4] = AI_MY; // 中心棋子
                 // 横
line[0] = flagX[0] ? AI_OP : (getPieceType(gameMapTmp[x - 4][y], type));
line[1] = flagX[1] ? AI_OP : (getPieceType(gameMapTmp[x - 3][y], type));
line[2] = flagX[2] ? AI_OP : (getPieceType(gameMapTmp[x - 2][y], type));
line[3] = flagX[3] ? AI_OP : (getPieceType(gameMapTmp[x - 1][y], type));

line[5] = flagX[4] ? AI_OP : (getPieceType(gameMapTmp[x + 1][y], type));
line[6] = flagX[5] ? AI_OP : (getPieceType(gameMapTmp[x + 2][y], type));
line[7] = flagX[6] ? AI_OP : (getPieceType(gameMapTmp[x + 3][y], type));
line[8] = flagX[7] ? AI_OP : (getPieceType(gameMapTmp[x + 4][y], type));

value += evaluateLine(line);

//纵
line[0] = flagY[0] ? AI_OP : getPieceType(gameMapTmp[x][y - 4], type);
line[1] = flagY[1] ? AI_OP : getPieceType(gameMapTmp[x][y - 3], type);
line[2] = flagY[2] ? AI_OP : getPieceType(gameMapTmp[x][y - 2], type);
line[3] = flagY[3] ? AI_OP : getPieceType(gameMapTmp[x][y - 1], type);

line[5] = flagY[4] ? AI_OP : getPieceType(gameMapTmp[x][y + 1], type);
line[6] = flagY[5] ? AI_OP : getPieceType(gameMapTmp[x][y + 2], type);
line[7] = flagY[6] ? AI_OP : getPieceType(gameMapTmp[x][y + 3], type);
line[8] = flagY[7] ? AI_OP : getPieceType(gameMapTmp[x][y + 4], type);

value += evaluateLine(line);

//左上-右下
line[0]=flagX[0]||flagY[0]?AI_OP:getPieceType(gameMapTmp[x-4][y-4],type);
line[1]=flagX[1]||flagY[1]?AI_OP:getPieceType(gameMapTmp[x-3][y-3],type);
line[2]=flagX[2]||flagY[2]?AI_OP:getPieceType(gameMapTmp[x-2][y-2],type);
line[3]=flagX[3]||flagY[3]?AI_OP:getPieceType(gameMapTmp[x-1][y-1],type);

line[5]=flagX[4]||flagY[4]?AI_OP:getPieceType(gameMapTmp[x+1][y+1],type);
line[6]=flagX[5]||flagY[5]?AI_OP:getPieceType(gameMapTmp[x+2][y+2],type);
```

```
    line[7] = flagX[6] || flagY[6] ? AI_OP : getPieceType(gameMapTmp[x + 3][y + 3], type);
    line[8] = flagX[7] || flagY[7] ? AI_OP : getPieceType(gameMapTmp[x + 4][y + 4], type);

    value += evaluateLine(line);

    // 右上-左下
    line[0] = flagX[7] || flagY[0] ? AI_OP : getPieceType(gameMapTmp[x + 4][y - 4], type);
    line[1] = flagX[6] || flagY[1] ? AI_OP : getPieceType(gameMapTmp[x + 3][y - 3], type);
    line[2] = flagX[5] || flagY[2] ? AI_OP : getPieceType(gameMapTmp[x + 2][y - 2], type);
    line[3] = flagX[4] || flagY[3] ? AI_OP : getPieceType(gameMapTmp[x + 1][y - 1], type);

    line[5] = flagX[3] || flagY[4] ? AI_OP : getPieceType(gameMapTmp[x - 1][y + 1], type);
    line[6] = flagX[2] || flagY[5] ? AI_OP : getPieceType(gameMapTmp[x - 2][y + 2], type);
    line[7] = flagX[1] || flagY[6] ? AI_OP : getPieceType(gameMapTmp[x - 3][y + 3], type);
    line[8] = flagX[0] || flagY[7] ? AI_OP : getPieceType(gameMapTmp[x - 4][y + 4], type);

    value += evaluateLine(line);

    return value;
}
/*
type:当前层的标记:MAX/MIN，分别为AI（1）和人（-1）
depth：当前层深
alpha：父层alpha的值
beta：父层beta的值
*/
int GameModel::minMax(int x, int y, int type, int depth, int alpha, int beta)
{
    gameMapTmp[x][y]=nextType(type);
    int weight = 0;
    int max = -INF;//下层权值上界
    int min = INF;//下层权值下界
    if (depth < MAX_DEPTH) {
       // iswin
       if(evaluatePiece(x,y,nextType(type))>=AI_FIVE){
           gameMapTmp[x][y]=0;
           if(type==MIN_NODE)
               return AI_FIVE;
           else
               return -AI_FIVE;
       }
       int i, j;
       for (i =1 ; i < kBoardSizeNum; ++i) {
            for (j = 1; j < kBoardSizeNum; ++j) {
               if (gameMapTmp[i][j]==0 && canSearch(i, j)) {
                   weight = minMax(i, j, nextType(type), depth + 1, min, max);
                   if (weight > max)
                       max = weight;
```

```
                    if (weight < min)
                        min = weight;
                // alpha-beta
                    if (type == MAX_NODE) {
                       if (max >= alpha){
                           gameMapTmp[x][y]=0;
                             return max;
                       }
                    }
                    else {
                        if (min <= beta){
                            gameMapTmp[x][y]=0;
                             return min;
                        }
                    }
                }
                else
                    continue;
            }
        }

        if (type == MAX_NODE){
            gameMapTmp[x][y]=0;
            return max;
        }
        else{
            gameMapTmp[x][y]=0;
            return min;
        }
    }
    else {
        weight = evaluateState(MAX_NODE);//评估我方局面
        weight -= type == MIN_NODE ? evaluateState(MIN_NODE) * 10 :
evaluateState(MIN_NODE);//评估对方局面
        gameMapTmp[x][y]=0;
        return weight;
    }
}
/*AI 下*/
void GameModel::actionByAI(int& clickRow, int& clickCol)
{
    // 计算评分
    int weight;
    int maxScore = -INF;
    std::vector<std::pair<int, int>> maxPoints;

    scoreMapVec.clear();
    for (int row = 1; row < kBoardSizeNum; row++)
```

```
        for (int col = 1; col < kBoardSizeNum; col++)
        {
            // 前提是这个坐标是空的
            gameMapTmp=gameMapVec;
            if (gameMapVec[row][col] == 0&&canSearch(row,col))
            {
                weight=minMax(row,col,nextType(MAX_NODE),1,-INF,maxScore);
                scoreMapVec[row][col]=weight;
                if (weight > maxScore)          // 找最大的数和坐标
                {
                    maxPoints.clear();
                    maxScore = scoreMapVec[row][col];
                    maxPoints.push_back(std::make_pair(row, col));
                }
                else if (weight== maxScore)     // 如果有多个最大的数，都存起来
                    maxPoints.push_back(std::make_pair(row, col));
            }
        }

    // 随机落子，如果有多个点的话
    srand((unsigned)time(0));
    int index = rand() % maxPoints.size();

    std::pair<int, int> pointPair = maxPoints.at(index);
    clickRow = pointPair.first; // 记录落子点
    clickCol = pointPair.second;
    updateGameMap(clickRow, clickCol);
    lastrowAI=clickRow;
    lastcolAI=clickCol;
    gameMapTmp=gameMapVec;
}

/*判断是否获胜*/
bool GameModel::isWin(int row, int col)
{
    // 横竖斜四种大情况，每种情况都根据当前落子往后遍历 5 个棋子，有一种符合就算赢
    // 水平方向
    for (int i = 0; i < 5; i++)
    {
        // 往左 5 个，往右匹配 4 个子，20 种情况
        if (col - i > 0 &&
            col - i + 4 < kBoardSizeNum &&
            gameMapVec[row][col - i] == gameMapVec[row][col - i + 1] &&
            gameMapVec[row][col - i] == gameMapVec[row][col - i + 2] &&
            gameMapVec[row][col - i] == gameMapVec[row][col - i + 3] &&
            gameMapVec[row][col - i] == gameMapVec[row][col - i + 4])
            return true;
    }
```

```
    // 竖直方向(上下延伸 4 个)
    for (int i = 0; i < 5; i++)
    {
        if (row - i > 0 &&
            row - i + 4 < kBoardSizeNum &&
            gameMapVec[row - i][col] == gameMapVec[row - i + 1][col] &&
            gameMapVec[row - i][col] == gameMapVec[row - i + 2][col] &&
            gameMapVec[row - i][col] == gameMapVec[row - i + 3][col] &&
            gameMapVec[row - i][col] == gameMapVec[row - i + 4][col])
            return true;
    }

    // 左斜方向
    for (int i = 0; i < 5; i++)
    {
        if (row + i < kBoardSizeNum &&
            row + i - 4 > 0 &&
            col - i > 0 &&
            col - i + 4 < kBoardSizeNum &&
            gameMapVec[row + i][col - i] == gameMapVec[row + i - 1][col - i + 1] &&
            gameMapVec[row + i][col - i] == gameMapVec[row + i - 2][col - i + 2] &&
            gameMapVec[row + i][col - i] == gameMapVec[row + i - 3][col - i + 3] &&
            gameMapVec[row + i][col - i] == gameMapVec[row + i - 4][col - i + 4])
            return true;
    }

    // 右斜方向
    for (int i = 0; i < 5; i++)
    {
        if (row - i > 0 &&
            row - i + 4 < kBoardSizeNum &&
            col - i > 0 &&
            col - i + 4 < kBoardSizeNum &&
            gameMapVec[row - i][col - i] == gameMapVec[row - i + 1][col - i + 1] &&
            gameMapVec[row - i][col - i] == gameMapVec[row - i + 2][col - i + 2] &&
            gameMapVec[row - i][col - i] == gameMapVec[row - i + 3][col - i + 3] &&
            gameMapVec[row - i][col - i] == gameMapVec[row - i + 4][col - i + 4])
            return true;
    }

    return false;
}
/*判断棋盘是否被填满*/
bool GameModel::isDeadGame()
{
    // 所有空格全部填满
    for (int i = 1; i < kBoardSizeNum; i++)
```

```
        for (int j = 1; j < kBoardSizeNum; j++)
        {
            if (!(gameMapVec[i][j] == 1 || gameMapVec[i][j] == -1))
                return false;
        }
    return true;
}
/*悔棋功能的实现*/
void GameModel::retractGame()
{
    if (gameType == BOT){
        gameMapVec[lastrowAI][lastcolAI]=0;
        gameMapVec[lastrowPer][lastcolPer]=0;
    }
    else{
        gameMapVec[lastrowPer][lastcolPer]=0;
        gameMapVec[lastrowPer2][lastcolPer2]=0;
    }
}
```

mainwindow.cpp

```
#include <QPainter>
#include <QTimer>
#include <QSound>
#include <QMouseEvent>
#include <QMessageBox>
#include <QMenuBar>
#include <QMenu>
#include <QColor>
#include <QPushButton>
#include <QAction>
#include <QDebug>
#include <math.h>
#include "mainwindow.h"

// -------全局遍历-------//
#define CHESS_ONE_SOUND ":/res/sound/chessone.wav"
#define WIN_SOUND ":/res/sound/win.wav"
#define LOSE_SOUND ":/res/sound/lose.wav"

const int kBoardMargin = 30; // 棋盘边缘空隙
const int kRadius = 15; // 棋子半径
const int kMarkSize = 6; // 落子标记边长
const int kBlockSize = 40; // 格子的大小
const int kPosDelta = 20; // 鼠标点击的模糊距离上限

const int kAIDelay = 700; // AI 下棋的思考时间
```

```
// -------------------- //

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    // 设置棋盘大小
    setFixedSize(kBoardMargin * 2 + kBlockSize * kBoardSizeNum, kBoardMargin * 3 +
kBlockSize * kBoardSizeNum);
//    setStyleSheet("background-color:yellow;");
    gameheight=kBoardMargin * 2 + kBlockSize * kBoardSizeNum;
    gamewidth=gameheight;

    // 开启鼠标 hover 功能，这两句一般要设置 window 的
    setMouseTracking(true);
//    centralWidget()->setMouseTracking(true);

    //添加悔棋按钮
    QPushButton *btn = new QPushButton("悔棋",this);
    btn->move(kBoardMargin,10+kBoardMargin + kBlockSize * kBoardSizeNum);
    btn->resize(kBlockSize*2,kBlockSize);
    btn->setFont(QFont("宋体", 18));
    connect(btn,&QPushButton::clicked,this,&MainWindow::retract);

    //添加认输按钮
    QPushButton *btn2 = new QPushButton("认输",this);
    btn2->move(kBoardMargin+kBlockSize*3,10+kBoardMargin + kBlockSize *
kBoardSizeNum);
    btn2->resize(kBlockSize*2,kBlockSize);
    btn2->setFont(QFont("宋体", 18));
    QObject::connect(btn2,&QPushButton::clicked,this,&MainWindow::gotolost);

    // 添加菜单
    QMenu *gameMenu = menuBar()->addMenu(tr("模式")); // menuBar 默认是存在的，直接加
菜单就可以了
    QAction *actionPVE = new QAction("人机对战", this);
    connect(actionPVE, SIGNAL(triggered()), this, SLOT(initPVEGame()));
    gameMenu->addAction(actionPVE);

    QAction *actionPVP = new QAction("双人对战", this);
    connect(actionPVP, SIGNAL(triggered()), this, SLOT(initPVPGame()));
    gameMenu->addAction(actionPVP);

    // 开始游戏
    initGame();
}

MainWindow::~MainWindow()
{
```

```
    if (game)
    {
        delete game;
        game = nullptr;
    }
}

void MainWindow::initGame()
{
    // 初始化游戏模型
    game = new GameModel;
    initPVEGame();//默认为人机
}

void MainWindow::initPVPGame()
{
    game_type = PERSON;
    game->gameStatus = PLAYING;
    game->startGame(game_type);
    update();
}

void MainWindow::initPVEGame()
{
    game_type = BOT;
    game->gameStatus = PLAYING;
    game->startGame(game_type);
    update();
}
void MainWindow::retract()
{
    game->retractGame();
    update();
}
void MainWindow::gotolost()
{
    game->belost=1;
    update();
}

void MainWindow::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    // 绘制棋盘
    painter.setBrush(QColor(213,176,146));

    painter.drawRect(this->rect());
    painter.setRenderHint(QPainter::Antialiasing, true); // 抗锯齿
```

装 订 线

```
//    QPen pen; // 调整线条宽度
//    pen.setWidth(2);
//    painter.setPen(pen);
    for (int i = 0; i < kBoardSizeNum + 1; i++)
    {
        painter.drawLine(kBoardMargin + kBlockSize * i, kBoardMargin, kBoardMargin +
kBlockSize * i, gameheight - kBoardMargin);
        painter.drawLine(kBoardMargin, kBoardMargin + kBlockSize * i, gamewidth -
kBoardMargin, kBoardMargin + kBlockSize * i);
    }

    QBrush brush;
    brush.setStyle(Qt::SolidPattern);
    // 绘制落子标记(防止鼠标出框越界)
    if (clickPosRow > 0 && clickPosRow < kBoardSizeNum &&
        clickPosCol > 0 && clickPosCol < kBoardSizeNum &&
        game->gameMapVec[clickPosRow][clickPosCol] == 0)
    {
        if (game->playerFlag)
            brush.setColor(Qt::black);
        else
            brush.setColor(Qt::white);
        painter.setBrush(brush);
        painter.drawRect(kBoardMargin + kBlockSize * clickPosCol - kMarkSize / 2,
kBoardMargin + kBlockSize * clickPosRow - kMarkSize / 2, kMarkSize, kMarkSize);
    }

    // 绘制棋子
    for (int i = 0; i < kBoardSizeNum; i++)
        for (int j = 0; j < kBoardSizeNum; j++)
        {
            if (game->gameMapVec[i][j] == 1)
            {
                brush.setColor(Qt::white);
                painter.setBrush(brush);
                painter.drawEllipse(kBoardMargin + kBlockSize * j - kRadius,
kBoardMargin + kBlockSize * i - kRadius, kRadius * 2, kRadius * 2);
            }
            else if (game->gameMapVec[i][j] == -1)
            {
                brush.setColor(Qt::black);
                painter.setBrush(brush);
                painter.drawEllipse(kBoardMargin + kBlockSize * j - kRadius,
kBoardMargin + kBlockSize * i - kRadius, kRadius * 2, kRadius * 2);
            }
        }
    if (game->belost && game->gameStatus == PLAYING)
    {
```

```
        qDebug() << "lose";
        game->gameStatus = WIN;
        QSound::play(LOSE_SOUND);
        QString str;
        if(game->playerFlag)
            str = "您已认输，白棋获胜";
        else
            str="您已认输，黑棋获胜";
        QMessageBox::StandardButton btnValue = QMessageBox::information(this, "so sad",
str);

        // 重置游戏状态，否则容易死循环
        if (btnValue == QMessageBox::Ok)
        {
            game->startGame(game_type);
            game->gameStatus = PLAYING;
        }
    }


    // 判断输赢
    if (clickPosRow > 0 && clickPosRow < kBoardSizeNum &&
        clickPosCol > 0 && clickPosCol < kBoardSizeNum &&
        (game->gameMapVec[clickPosRow][clickPosCol] == 1 ||
            game->gameMapVec[clickPosRow][clickPosCol] == -1))
    {
        if (game->isWin(clickPosRow, clickPosCol) && game->gameStatus == PLAYING)
        {
            qDebug() << "win";
            game->gameStatus = WIN;
            QSound::play(WIN_SOUND);
            QString str;
            if (game->gameMapVec[clickPosRow][clickPosCol] == 1)
                str = "white player";
            else if (game->gameMapVec[clickPosRow][clickPosCol] == -1)
                str = "black player";
            QMessageBox::StandardButton btnValue = QMessageBox::information(this,
"congratulations", str + " win!");

            // 重置游戏状态，否则容易死循环
            if (btnValue == QMessageBox::Ok)
            {
                game->startGame(game_type);
                game->gameStatus = PLAYING;
            }
        }
    }
```

```
    // 判断死局
    if (game->isDeadGame())
    {
        QSound::play(LOSE_SOUND);
        QMessageBox::StandardButton btnValue = QMessageBox::information(this, "oops",
"dead game!");
        if (btnValue == QMessageBox::Ok)
        {
            game->startGame(game_type);
            game->gameStatus = PLAYING;
        }
    }
}

void MainWindow::mouseMoveEvent(QMouseEvent *event)
{
    // 通过鼠标的 hover 确定落子的标记
    int x = event->x();
    int y = event->y();

    // 棋盘边缘不能落子
    if (x >= kBoardMargin + kBlockSize / 2 &&
            x < gamewidth - kBoardMargin &&
            y >= kBoardMargin + kBlockSize / 2 &&
            y < gameheight- kBoardMargin)
    {
        // 获取最近的左上角的点
        int col = x / kBlockSize;
        int row = y / kBlockSize;

        int leftTopPosX = kBoardMargin + kBlockSize * col;
        int leftTopPosY = kBoardMargin + kBlockSize * row;

        // 根据距离算出合适的点击位置,一共四个点，根据半径距离选最近的
        clickPosRow = -1; // 初始化最终的值
        clickPosCol = -1;
        int len = 0; // 计算完后取整就可以了

        // 确定一个误差在范围内的点，且只可能确定一个出来
        len = sqrt((x - leftTopPosX) * (x - leftTopPosX) + (y - leftTopPosY) * (y -
leftTopPosY));
        if (len < kPosDelta)
        {
            clickPosRow = row;
            clickPosCol = col;
        }
```

```
        len = sqrt((x - leftTopPosX - kBlockSize) * (x - leftTopPosX - kBlockSize) +
(y - leftTopPosY) * (y - leftTopPosY));
        if (len < kPosDelta)
        {
            clickPosRow = row;
            clickPosCol = col + 1;
        }
        len = sqrt((x - leftTopPosX) * (x - leftTopPosX) + (y - leftTopPosY - kBlockSize)
* (y - leftTopPosY - kBlockSize));
        if (len < kPosDelta)
        {
            clickPosRow = row + 1;
            clickPosCol = col;
        }
        len = sqrt((x - leftTopPosX - kBlockSize) * (x - leftTopPosX - kBlockSize) +
(y - leftTopPosY - kBlockSize) * (y - leftTopPosY - kBlockSize));
        if (len < kPosDelta)
        {
            clickPosRow = row + 1;
            clickPosCol = col + 1;
        }
    }

    // 存了坐标后也要重绘
    update();
}

void MainWindow::mouseReleaseEvent(QMouseEvent *event)
{
    // 人下棋，并且不能抢机器的棋
    if (!(game_type == BOT && !game->playerFlag))
    {
        chessOneByPerson();
        // 如果是人机模式，需要调用 AI 下棋
        if (game->gameType == BOT && !game->playerFlag)
        {
            // 用定时器做一个延迟
            QTimer::singleShot(kAIDelay, this, SLOT(chessOneByAI()));
        }
    }
}

void MainWindow::chessOneByPerson()
{
    // 根据当前存储的坐标下子
    // 只有有效点击才下子，并且该处没有子
    if (clickPosRow != -1 && clickPosCol != -1 &&
game->gameMapVec[clickPosRow][clickPosCol] == 0)
```

```
    {
        game->actionByPerson(clickPosRow, clickPosCol);
        QSound::play(CHESS_ONE_SOUND);

        // 重绘
        update();
    }
}

void MainWindow::chessOneByAI()
{
    game->actionByAI(clickPosRow, clickPosCol);
    QSound::play(CHESS_ONE_SOUND);
    update();
}
```

GameModel.h

```
#ifndef GAMEMODEL_H
#define GAMEMODEL_H

// ---- 五子棋游戏模型类 ---- //
#include <vector>
// 游戏类型，双人还是 AI（目前固定让 AI 下黑子）
enum GameType
{
    PERSON,
    BOT
};

// 游戏状态
enum GameStatus
{
    PLAYING,
    WIN,
    DEAD
};

// 棋盘尺寸
const int kBoardSizeNum = 15;

class GameModel
{

public:
    GameModel();
    static const int INF = 106666666;
    static const int ERROR_INDEX = -1;
    static const int AI_ZERO = 0;
```

```
    static const int AI_ONE = 10;
    static const int AI_ONE_S = 1;
    static const int AI_TWO = 800;
    static const int AI_TWO_S = 300;
    static const int AI_THREE = 3000;
    static const int AI_THREE_S = 1000;
    static const int AI_FOUR = 300000;
    static const int AI_FOUR_S = 2500;
    static const int AI_FIVE = 1000000;

    static const int AI_EMPTY = 0; // 无子
    static const int AI_MY = 1; // 待评价子
    static const int AI_OP = 2; // 对方子或不能下子
    static const int MAX_NODE = 1;//AI 白棋
    static const int MIN_NODE = -1;//人黑棋
    static const int MAX_DEPTH=3;

public:
    std::vector<std::vector<int>> gameMapVec; // 存储当前游戏棋盘和棋子的情况,空白为0，白子 1，黑子-1
    std::vector<std::vector<int>> gameMapTmp;
    std::vector<std::vector<int>> scoreMapVec; // 存储各个点位的评分情况，作为 AI 下棋依据
    bool playerFlag; // 标示下棋方
    GameType gameType; // 游戏模式
    GameStatus gameStatus; // 游戏状态
    int lastrowPer,lastcolPer;//黑子
    int lastrowPer2,lastcolPer2;
    int lastrowAI,lastcolAI;
    bool belost;
    int value;

    void startGame(GameType type); // 开始游戏
    void actionByPerson(int row, int col); // 人执行下棋
    void actionByAI(int &clickRow, int &clickCol); // 机器执行下棋
    void updateGameMap(int row, int col); // 每次落子后更新游戏棋盘
    bool isWin(int row, int col); // 判断游戏是否胜利
    bool isDeadGame(); // 判断是否和棋
    void retractGame();

    /*搜索算法用到的函数*/
    int minMax(int x, int y, int type, int depth, int alpha, int beta);
    int evaluateState(int type);
    int getValue(int cnt, int blk);
    int evaluateLine(int line[]);
    int evaluateLine(int line[], bool ALL);
    int evaluatePiece(int x, int y, int type);
    int getPieceType(int x, int y, int type);
```

```
    int getPieceType(int A, int type);
    int nextType(int type);
    bool canSearch(int x, int y);
};

#endif // GAMEMODEL_H
```

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "GameModel.h"

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

protected:
    // 绘制
    void paintEvent(QPaintEvent *event);
    // 监听鼠标移动情况，方便落子
    void mouseMoveEvent(QMouseEvent *event);
    // 实际落子
    void mouseReleaseEvent(QMouseEvent *event);

private:
    GameModel *game; // 游戏指针
    GameType game_type; // 存储游戏类型
    int clickPosRow, clickPosCol; // 存储将点击的位置
    int gameheight,gamewidth;

    void initGame();
    void checkGame(int y, int x);

private slots:
    void chessOneByPerson(); // 人执行
    void chessOneByAI(); // AI 下棋

    void initPVPGame();
    void initPVEGame();

    void retract();
    void gotolost();
```

```
};

#endif // MAINWINDOW_H
```

main.cpp

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

装

订

线