“BetaGo” 五子棋游戏开发文档

一. 主要功能

本程序模拟经典的五子棋益智游戏，采用 15*15 的棋盘，具有人机对弈/双人对弈两种玩法，并且可以选择简单、中级、高级三种难度模式。同时玩家在开始游戏前可以选择新建游戏或者恢复之前存盘的棋局。在游戏结束时（即五颗棋子连成一条直线），程序会弹出对话框，提示输赢结果，并退出游戏。

二. 主要模块

1．前段交互

游戏界面：棋盘区域、功能选择等

2．后端策略

核心算法：限制深度优先搜索

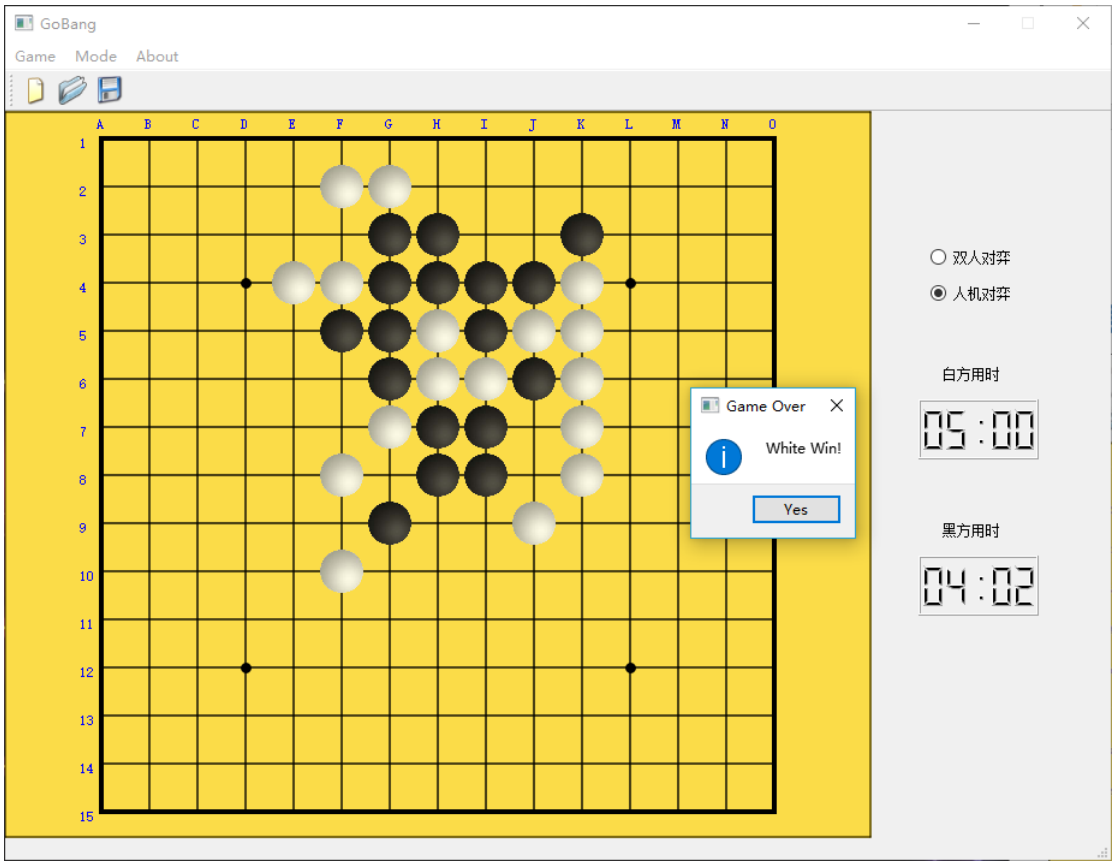
3．性能测试

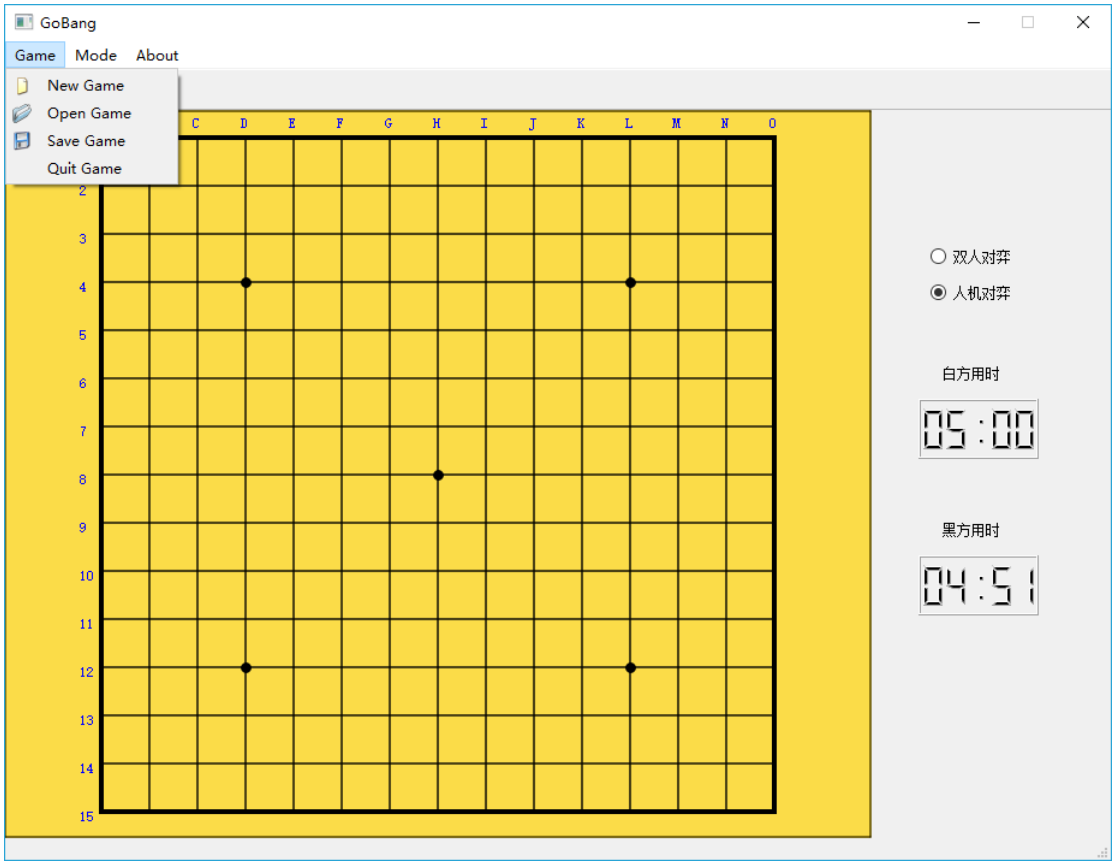
性能指标：运行时间、占用内存统计等

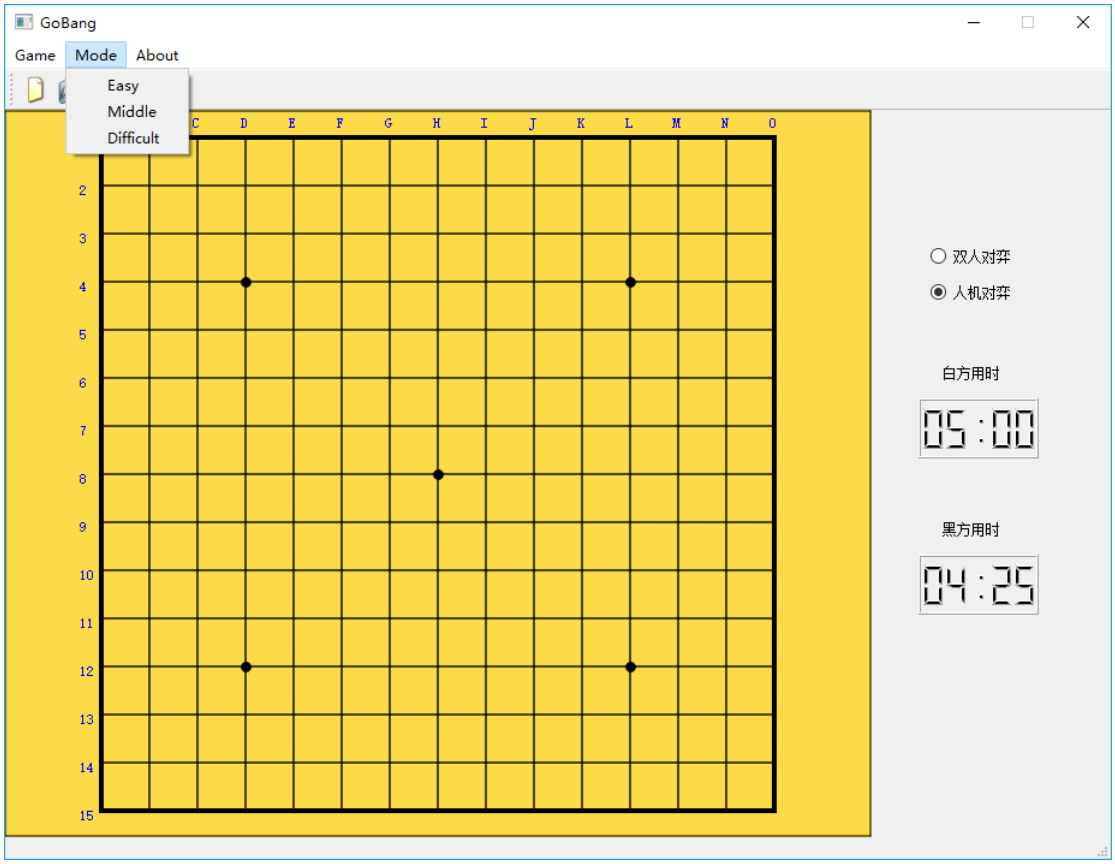
三. 开发平台与工具

1. 使用 git 作为版本控制工具，并且项目开源在 github 网站，便于组内各成员合作开发。
2. 操作系统： Windows 10
3. 集成开发环境（IDE）：Qt creator 4.2.1 + Qt 5.8.0（MSVC 2015）
4. 使用 Qt Widgets Application 项目模板，所有程序代码为 C++实现。

四. 界面设计


GoBang
Game Mode About
A B C D E F G H I J K L M N O
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Game Over
White Win!
Yes
双人对弈
人机对弈
白方用时
05 : 00
黑方用时
04 : 02


GoBang
Game Mode About
New Game
Open Game
Save Game
Quit Game
C D E F G H I J K L M N O
2 3 4 5 6 7 8 9 10 11 12 13 14 15
双人对弈
人机对弈
白方用时
05 : 00
黑方用时
04 : 51



1. 棋盘绘制：

棋盘使用 QPen 与 QBrush 进行绘制；棋子使用贴图，通过 QImage 从资源文件中载入。

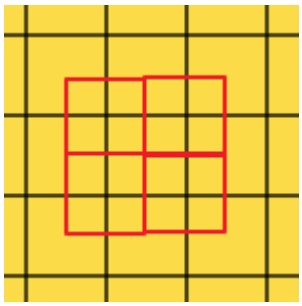
```
void MainWindow::paintEvent(QPaintEvent *){
    // get the current status
    //game.getStatus(chessboard);

    QPainter paint(this);
    paint.setRenderHint(QPainter::Antialiasing,true);
    //background
    double factor = 0.73;
    paint.setPen(QPen(QColor::fromRgbF(1, 211.0/255.0, 0,0.7),2,Qt::SolidLine));
    paint.setBrush(QBrush(QColor::fromRgbF(1, 211.0/255.0,
0,0.7),Qt::SolidPattern));
    paint.drawRect(0,Y*factor,SIZE,SIZE-2*Y*factor);
    paint.setPen(QPen(QColor::fromRgbF(0,0,0,1),1,Qt::SolidLine));
    paint.drawLine(0,Y*factor,SIZE,Y*factor);
    paint.drawLine(0,-Y*factor+SIZE,SIZE,-Y*factor+SIZE);
    paint.drawLine(0,Y*factor,0,-Y*factor+SIZE);
    paint.drawLine(SIZE,Y*factor,SIZE,-Y*factor+SIZE);
    //horizonal line * (CHECK_NUM-1)
    paint.setPen(QPen(QColor::fromRgbF(0,0,0,0.7),2,Qt::SolidLine));
    for(int i=1;i<CHECK_NUM;i++){
```

```
paint.drawLine(X,Y+CHECK_WIDTH*i,X+CHECK_WIDTH*(CHECK_NUM),Y+CHECK_WIDTH*i);//dra
w line: from (x1,y1) to (x2,y2)
    }
    //vertical line * (CHECK_NUM-1)
    for(int i=1;i<CHECK_NUM;i++){
      paint.drawLine(X+CHECK_WIDTH*i,Y,X+CHECK_WIDTH*i,Y+CHECK_WIDTH*(CHECK_NUM));
    }
    //frame * 4
    paint.setPen(QPen(QColor::fromRgbF(0,0,0,1),4,Qt::SolidLine));
    paint.drawLine(X,Y,X+CHECK_WIDTH*CHECK_NUM,Y);

paint.drawLine(X,Y+CHECK_WIDTH*CHECK_NUM,X+CHECK_WIDTH*CHECK_NUM,Y+CHECK_WIDTH*CH
ECK_NUM);
    paint.drawLine(X,Y,X,Y+CHECK_WIDTH*CHECK_NUM);

paint.drawLine(X+CHECK_WIDTH*CHECK_NUM,Y,X+CHECK_WIDTH*CHECK_NUM,Y+CHECK_WIDTH*CH
ECK_NUM);
    //text
    paint.setPen(QPen(QColor::fromRgbF(0,0,1,1),4,Qt::SolidLine));
    if(1){
        int num = 1;
        char str[3],char_a[2]="A";
        for(short i = 0;i < 15;i++){
            //itoa(num++,str,10);
            sprintf(str, "%d", num++);
            paint.drawText(X*0.78,Y*1.1+i*CHECK_WIDTH,str);
            paint.drawText(X*0.95+i*CHECK_WIDTH,Y*0.9,char_a);
            char_a[0]++;
        }
    }
    //dot * 5
    int dot = CHECK_WIDTH/8;
    paint.setPen(QPen(QColor::fromRgbF(0,0,0,1),4,Qt::SolidLine));
    paint.setBrush(QBrush(Qt::black,Qt::SolidPattern));
    paint.drawEllipse(X-dot/2+3*CHECK_WIDTH,Y-dot/2+3*CHECK_WIDTH,dot,dot);//draw
ellipse: start(X,Y),CHECK_WIDTH,height
    paint.drawEllipse(X-dot/2+3*CHECK_WIDTH,Y-dot/2+11*CHECK_WIDTH,dot,dot);
    paint.drawEllipse(X-dot/2+11*CHECK_WIDTH,Y-dot/2+3*CHECK_WIDTH,dot,dot);
    paint.drawEllipse(X-dot/2+11*CHECK_WIDTH,Y-dot/2+11*CHECK_WIDTH,dot,dot);
    paint.drawEllipse(X-dot/2+7*CHECK_WIDTH,Y-dot/2+7*CHECK_WIDTH,dot,dot);
    //chess pieces
    int radix = CHECK_WIDTH*0.9;
    QImage black_chess,white_chess;
```

```
    black_chess.load(":/images/black");
    white_chess.load(":/images/white");
    black_chess = black_chess.scaled(radix,radix,Qt::KeepAspectRatio);
    white_chess = white_chess.scaled(radix,radix,Qt::KeepAspectRatio);
    if(1){
        int pos_x,pos_y;
        for(short i = 0; i < 15; i++){
            for(short j = 0; j < 15; j++){
                pos_x = X-radix/2+CHECK_WIDTH*i;
                pos_y = Y-radix/2+CHECK_WIDTH*j;
                if(chessboard[i][j]==1){
                    paint.drawImage(pos_x,pos_y,black_chess);
                }
                else if(chessboard[i][j]==-1){
                    paint.drawImage(pos_x,pos_y,white_chess);
                }
            }
        }
    }
}
```

2. 鼠标事件处理：

当鼠标落在整张棋盘中的有效位置时，处理鼠标事件。并且按照下图中红色区域划分所属落子点，并将相应颜色的棋子落下。



```
void MainWindow::mousePressEvent(QMouseEvent *mouseEvent)
{
    int x = mouseEvent->x();
    int y = mouseEvent->y();

    if(x>0&&x<SIZE&&y>0&&y<SIZE&&this->chessboard[int((x-
X)*1.0/CHECK_WIDTH+0.5)][int((y-Y)*1.0/CHECK_WIDTH+0.5)]==0)
    {

        this->chessCounts += 1;
        if (ui->radioType->isChecked()&&(!is_win()))
        {
```

```
            this->chessboard[int((x-X)*1.0/CHECK_WIDTH+0.5)][int((y-
Y)*1.0/CHECK_WIDTH+0.5)] = 1;
            this->chessCounts += 1;
            this->update();
            computer();
        }
        else if(ui->radioType_2->isChecked()&&(!is_win()))
        {
            this->chessboard[int((x-X)*1.0/CHECK_WIDTH+0.5)][int((y-
Y)*1.0/CHECK_WIDTH+0.5)] = (this->chessCounts%2 == 0)?1:-1;
            this->chessboard[int((x-X)*1.0/CHECK_WIDTH+0.5)][int((y-
Y)*1.0/CHECK_WIDTH+0.5)] = (this->chessCounts%2 == 1)?1:-1;
        }
        this->update();      //update the window
    }
    if(is_win() ==1){
        cout << "is win" << endl;
        QMessageBox::information(NULL, "Game Over", "Black Win!",
QMessageBox::Yes, QMessageBox::Yes);
        this->close();
    }
    else if(is_win()==-1){
        cout << "is not win" << endl;
        QMessageBox::information(NULL, "Game Over", "White Win!",
QMessageBox::Yes, QMessageBox::Yes);
        this->close();
    }
}
```

3. **胜负判断：**

每次落子后，棋盘二维数组得到更新。is_win()函数对棋盘中的横、竖和两条对角线方向分别字符串转换，并进行“11111”（黑子*5）或者“00000”（白子*5）的子串查找。函数返回值为 0 代表胜负未分，返回 1 代表黑方胜，返回-1 代表白方胜。

```
int MainWindow::is_win(){
    int stat = 0;//0: not ended; 1: black win; -1: white win;
    if (timeValuePlayer->minute() == 0 && timeValuePlayer->second() == 0)
        return -1;
    else if (timeValueComputer->minute() == 0 && timeValueComputer->second() == 0)
        return 1;


    const QString black_win("11111");
    const QString white_win("00000");
    QString line;//-- or | or \ or /
```

```
    const int len = 15;
    bool left_right = false;
    bool up_down = false;
    bool leftUp_rightDown = false;
    bool leftDown_rightUp = false;
    //  left-right
    for(int i = 0;i<len;i++){
        for(int j=0;j<len;j++){
            switch(chessboard[j][i]){
            case 1:line[j]='1';break;
            case 0:line[j]='2';break;
            case -1:line[j]='0';break;
            }
        }
        if(line.contains(black_win)){
            stat = 1;
            left_right = true;
            continue;
        }
        else if(line.contains(white_win)){
            stat = -1;
            left_right = true;
            continue;
        }
    }
    //  up-down
    if(!left_right){
        for(int i = 0;i<len;i++){
            for(int j=0;j<len;j++){
                switch(chessboard[i][j]){
                case 1:line[j]='1';break;
                case 0:line[j]='2';break;
                case -1:line[j]='0';break;
                }
            }
            if(line.contains(black_win)){
                stat = 1;
                up_down = true;
                continue;
            }
            else if(line.contains(white_win)){
                stat = -1;
                up_down = true;
                continue;
```

```
            }
        }
    }
    //  leftUp-rightDown
    if((!left_right)||(!up_down)){
        for(int i = 4;i<len;i++){
            for(int j = 0;j<i+1;j++){
                switch(chessboard[len-(i-j)][j]){
                case 1:line[j]='1';break;
                case 0:line[j]='2';break;
                case -1:line[j]='0';break;
                }
            }
            if(line.contains(black_win)){
                stat = 1;
                leftUp_rightDown = true;
                continue;
            }
            else if(line.contains(white_win)){
                stat = -1;
                leftUp_rightDown = true;
                continue;
            }

            for(int j = 0;j<i+1;j++){
                switch(chessboard[i-j][len-j]){
                case 1:line[j]='1';break;
                case 0:line[j]='2';break;
                case -1:line[j]='0';break;
                }
            }
            if(line.contains(black_win)){
                stat = 1;
                leftUp_rightDown = true;
                continue;
            }
            else if(line.contains(white_win)){
                stat = -1;
                leftUp_rightDown = true;
                continue;
            }
        }
    }
```

```
    //  leftDown-rightUp
    if((!left_right)||(!up_down)||(!leftUp_rightDown)){
        for(int i = 4;i<len;i++){
            for(int j = 0;j<i+1;j++){
                switch(chessboard[i-j][j]){
                case 1:line[j]='1';break;
                case 0:line[j]='2';break;
                case -1:line[j]='0';break;
                }
            }
            if(line.contains(black_win)){
                stat = 1;
                leftDown_rightUp = true;
                continue;
            }
            else if(line.contains(white_win)){
                stat = -1;
                leftDown_rightUp = true;
                continue;
            }

            for(int j = 0;j<i+1;j++){
                switch(chessboard[len-(i-j)][len-j]){
                case 1:line[j]='1';break;
                case 0:line[j]='2';break;
                case -1:line[j]='0';break;
                }
            }
            if(line.contains(black_win)){
                stat = 1;
                leftDown_rightUp = true;
                continue;
            }
            else if(line.contains(white_win)){
                stat = -1;
                leftDown_rightUp = true;
                continue;
            }
        }
    }
    return stat;
}
```

4. 菜单设计

包含三个子菜单：

- 游戏菜单，包括新游戏，打开游戏，保存游戏，以及退出游戏
- 模式菜单，设置游戏难度（人机对战时），有低中高三个难度
- 关于菜单，弹出项目信息

```
QMenuBar *MenuBar = new QMenuBar(this);
QMenu *GameMenu = new QMenu(tr("Game"), MenuBar);
QMenu *ModeMenu = new QMenu(tr("Mode"), MenuBar);
QMenu *AboutMenu = new QMenu(tr("About"), MenuBar);

const QIcon newIcon = QIcon::fromTheme("document-new", QIcon(":/icon/new.png"));
QAction *newGame = new QAction(newIcon, tr("New Game"), GameMenu);
const       QIcon       openIcon       =       QIcon::fromTheme("document-open",
QIcon(":/icon/open.png"));
QAction *openGame = new QAction(openIcon, tr("Open Game"), GameMenu);
const       QIcon       saveIcon       =       QIcon::fromTheme("document-save",
QIcon(":/icon/save.png"));
QAction *saveGame = new QAction(saveIcon, tr("Save Game"), GameMenu);
QAction *quitGame = new QAction(tr("Quit Game"), GameMenu);

GameMenu->addAction(newGame);
GameMenu->addAction(openGame);
GameMenu->addAction(saveGame);
GameMenu->addAction(quitGame);
MenuBar->addMenu(GameMenu);

    QAction *easyMode = new QAction("Easy", ModeMenu);
    QAction *middleMode = new QAction("Middle", ModeMenu);
    QAction *difficultMode = new QAction("Difficult", ModeMenu);
ModeMenu->addAction(easyMode);
ModeMenu->addAction(middleMode);
ModeMenu->addAction(difficultMode);
MenuBar->addMenu(ModeMenu);

QAction *aboutInfo = new QAction("About", AboutMenu);
AboutMenu->addAction(aboutInfo);
MenuBar->addMenu(AboutMenu);
```

另外设置了工具栏，方便快捷地实现一些常用功能，包括开始新游戏，打开游戏，保存游戏。

```
QToolBar *GameToolBar = addToolBar(tr("Game"));
GameToolBar->addAction(newGame);
GameToolBar->addAction(openGame);
GameToolBar->addAction(saveGame);
```

通过信号槽将对应的按钮与函数联系起来:

```
connect(newGame, SIGNAL(triggered(bool)), this, SLOT(newGame()));
```

```
connect(openGame, SIGNAL(triggered(bool)), this, SLOT(openGame()));
connect(saveGame, SIGNAL(triggered(bool)), this, SLOT(saveGame()));
connect(quitGame, SIGNAL(triggered(bool)), this, SLOT(quitGame()));
connect(easyMode, SIGNAL(triggered(bool)), this, SLOT(setEasyMode()));
connect(middleMode, SIGNAL(triggered(bool)), this, SLOT(setMiddleMode()));
connect(difficultMode, SIGNAL(triggered(bool)), this, SLOT(setDifficultMode()));
connect(aboutInfo, SIGNAL(triggered(bool)), this, SLOT(showAbout()));
```

（1）、游戏菜单

游戏菜单包括新游戏、打开游戏、保存游戏以及退出游戏。

- 新游戏 newGame()：初始化棋盘，初始化计时模块
- 打开游戏 openGame()：打开保存过的游戏文件，在原游戏基础上开始游戏
- 保存游戏 saveGame()：保存当前棋盘信息，方便下次使用。
- 退出游戏 quitGame()：关闭当前游戏窗口

（2）、模式菜单

通过设置搜索算法中的深度来控制游戏难度。

- 简单模式 easyMode()：对应的搜索深度为 1
- 中等模式 middleMode()：对应的搜索深度为 2
- 复杂模式 difficultMode()：对应的搜索深度为 3

（3）、关于菜单

弹出项目信息的对话框。

5. 功能按钮

功能按钮包括对战模式选择，有人机对战和人人对战两种模式，通过 radioButton 实现切换，每次切换对战模式，游戏重新初始化。

6. 计时模块

在 mainwindow.ui 中新建 lcdComputer 和 lcdPlayer 用于输出时间，并且通过 timer 控制每 1000 毫秒（即 1 秒）更新一次时间。

在 mainwindow.h 对应的语句为：

```
QTimer* timer;
QTime* timeValueComputer;
QTime* timeValuePlayer;
```

在 mainwindow.cpp 对应的语句为：

```
timerComputer = new QTimer();
timerPlayer = new QTimer();
timer = new QTimer();
ui->lcdComputer->display(this->timeValueComputer->toString());
ui->lcdPlayer->display(this->timeValuePlayer->toString());
this->timer->start(1000);
connect(timer,SIGNAL(timeout()),this,SLOT(setDisplay()));
```

槽函数中用于更新倒计时的函数为：

```
void MainWindow::setDisplay()
{
    if (this->chessCounts%2 == 0 && !is_win())
        this->timeValuePlayer->setHMS(0,          this->timeValuePlayer->addSecs(-
1).minute(), this->timeValuePlayer->addSecs(-1).second());
```

```
    else if (this->chessCounts%2 == 1 && !is_win())
        this->timeValueComputer->setHMS(0,  this->timeValueComputer->addSecs(-
1).minute(), this->timeValueComputer->addSecs(-1).second());

    ui->lcdComputer->display(this->timeValueComputer->toString());
    ui->lcdPlayer->display(this->timeValuePlayer->toString());
}
```

函数中通过棋盘上的棋子数目判断当前的下棋方，通过槽函数更新倒计时。

7. **设计 Game 类**

用于存储当前棋盘信息，用于保存游戏新建游戏等操作。

五. 算法部分

1. **五子棋 AI 实现的基本思路：**

五子棋的本质上是一棵博弈树，在这颗树中，从根节点 0 开始，奇数层表示电脑可能的走法，偶数层表示玩家可能的走法。假设电脑先手，那么第一层就是电脑的所有可能的走法，第二层就是玩家的所有可能走法，以此类推。

在这里我们假定棋盘的大小是15×15， 整颗博弈树的规则是这样的：

1） 电脑走棋的层我们称为 MAX 层，这一层电脑要保证自己利益最大化，那么就需要选分最高的节点。

2） 玩家走棋的层我们称为 MIN 层，这一层玩家要保证自己的利益最大化，那么就会选分最低的节点。

整个算法的过程可以用一个图示来进行表示：

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

而每一个节点的分数，都是由子节点决定的，因此我们只能对博弈树进行深度优先搜索而无法进行广度优先搜索。但是我们需要注意到一点，那就是整个博弈树的搜索空间是很大的。我们看到第一层有15×15 = 225 种可能的走法，第二层的每一个节点又有 224 种，假设我们进行 4 层思考，也就是电脑和玩家各走两步，那么这颗博弈树的最后一层的节点数为2.5×10^9个。可以预见，其算法的时间和空间的复杂度很高，所以我们需要更多的策略来进行修正。

所以我们没有办法使得在有限的，较短的时间内每一次都搜索到搜索树的底部，只能在有限深度的层数下进行搜索。故而在每一层我们都应该有一个效用函数（*Utility*

Function)，在这里的效用函数思考是这样的（下面的加分分析都是针对电脑来说的，玩家的分数则要相应的减分，因为假设电脑在这里是 MAX 层，玩家是 MIN 层）：

1） 如果在水平方向的检测上发现电脑有已经获胜的情况加上 100 分，如果发现玩家有已经获胜的情况，那么就减去 100 分；

2） 如果在水平方向的检测上发现有“活四”的情况，也就是有四个子相连，但是两端都没有另一方的子，那么这种情况就被称为“活四”。如果电脑发现有“活四”的情况，那么就加上 10 分，如果是玩家出现“活四”的情况，就减去 10 分；

3） 如果在水平方向的检测上发现有“活三”的情况，也就是有三个子相连，但是两端都没有另一方的子，那么这种情况就被称为“活三”。如果电脑发现有“活三”的情况，那么就加上 10 分，如果是玩家出现“活三”的情况，就减去 10 分；

4） 如果在水平方向的检测上发现有“活二”的情况，也就是有两个子相连，但是两端都没有另一方的子，那么这种情况就被称为“活二”。如果电脑发现有“活二”的情况，那么就加上 2 分，如果是玩家出现“活二”的情况，就减去 2 分；

5） 如果在水平方向的检测上发现有“死四”的情况，也就是有四个子相连，但是一端有另一方的子，那么这种情况就被称为“死四”。如果电脑发现有“死四”的情况，那么就加上 5 分，如果是玩家出现“死四”的情况，就减去 5 分；

6） 如果在水平方向的检测上发现有“死三”的情况，也就是有三个子相连，但是一端有另一方的子，那么这种情况就被称为“死三”。如果电脑发现有“死三”的情况，那么就加上 5 分，如果是玩家出现“死三”的情况，就减去 5 分；

在其他的方向上的检测（竖直方向，45 度方向，-45 度方向）也是一样的原则，需要对局面中所有可能的进行判断，那么就可以得出一个合理的效用函数了。

2. 五子棋的减少搜索空间思路

1） 首先在每一层进行选择的时候，我们认为一般的棋手不会随便丢弃自己正在搏杀的阵地，也就是说，棋手的落子位置必须要和当前局面上的棋子有空间位置上的联系。这样做可以较大限度地减少前几层的搜索空间，也就是可以很大程度上减少了时间和空间的复杂度。

2） *Alpha Beta* 剪枝：

从上面的复杂度分析中可以看出，我们需要对这颗搜索树进行剪枝，否则搜索的空间太大，根本在规定的时间内给出下一步的坐标点，而且会造成程序的崩溃。我们至少要让电脑可以保持至少需要进行 4 层思考，4 层是中等棋手的水平。

Alpha Beta 剪枝的策略是如下的原理：

前面讲到过，电脑会在 *MAX* 层选择最大节点，而玩家会在 *MIN* 层选择最小节点。那么如下两种情况就是分别对双方不利的选择：

1〉 在 *MAX* 层，假设当前层已经搜索到一个最大值 *X*，如果发现下一个节点的下一层（也就是 *MIN* 层）会产生一个比 *X* 还小的值，那么就直接剪掉此节点。也就是在 *MAX* 层的时候会把当前层已经搜索到的最大值 *X* 存起来，如果下一个节点的下一层会产生一个比 *X* 还小的值 *Y*，那么之前说过玩家总是会选择最小值的。也就是说这个节点玩家的分数不会超过 *Y*，那么这个节点显然没有必要进行计算。换言之，就是电脑发现这一步是对玩家更有利的，那么当然不会走这一步。

2〉 在 *MIN* 层，假设当前层已经搜索到一个最小值 *Y*，如果发现下一个节点的下一

层（也就是 *MIN* 层）会产生一个比 *Y* 还大的值，那么就直接剪掉此节点。换言之，就是玩家发现这一步棋是对电脑更有利，玩家必定不会走这一步。

剪枝的策略可以如下的程序伪代码进行解释：

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a), α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

3. **五子棋的防御策略：**

一般来说，五子棋在进行策略评价的时候， 效用函数是没有办法能够很好得按照我们的人类思想来进行思考和权衡的。我们都知道，在面对必输或者必胜的局面下，电脑必须赶紧落到正确的位置，防止自己输掉或者对方再反扑。

所以我们的五子棋设计得希望能够偏于“防御优先”，满足之前所说的要求。具体来说，这个防御策略将会搜索如下的局面并且做出相应的动作。具体采取那个动作，可以对动作进行优先级的设定来完成（以水平方向上的防御策略来进行说明）：

1） 如果发现自己的一方已经可以获胜（自己一方已经有 4 个子，这 4 个子可以是连在一起的，也可以是三个子连在一起而剩下的子相差有一个子的距离），那么这个可以获胜的动作的优先级是 10；
2） 如果发现是对方已经可以获胜，那么这个“必须拦截”的动作的优先级是 5；
3） 如果发现自己的一方出现“活三”的情况，那么应该强先攻占，优先级是 4；
4） 如果发现对方出现“活三”的情况，也有需要拦截的价值，优先级是 3；

其他的方向上（竖直方向，45 度方向，-45 度方向）的情况也是以此类推。

4. **Cpp 函数的说明：**

`void miniMaxSearchForFive(const MatchState& currentState, int& r, int& c)`
是进行极大极小搜索的函数，其中包含我们的效用估计策略和防御策略，结合来判断当前应该下在哪一个坐标。currentState 是当前的局面，r 是行，c 是列；

`int Min_Value(MatchState& current_state, int depth, int alpha, int beta)`
进行极小搜索的函数，currentState 是当前的局面，depth 是限制深度搜索的参数（博弈树的搜索不能超过这个值，否则会出现搜索空间过大的问题），alpha 和 beta 是 *Alpha Beta* 剪枝策略的参数；

`int Max_Value(MatchState& current_state, int depth, int alpha, int beta)`
进行极大搜索的函数，currentState 是当前的局面，depth 是限制深度搜索的参数（博弈树的搜索不能超过这个值，否则会出现搜索空间过大的问题），alpha 和 beta 是 *Alpha Beta* 剪枝策略的参数；

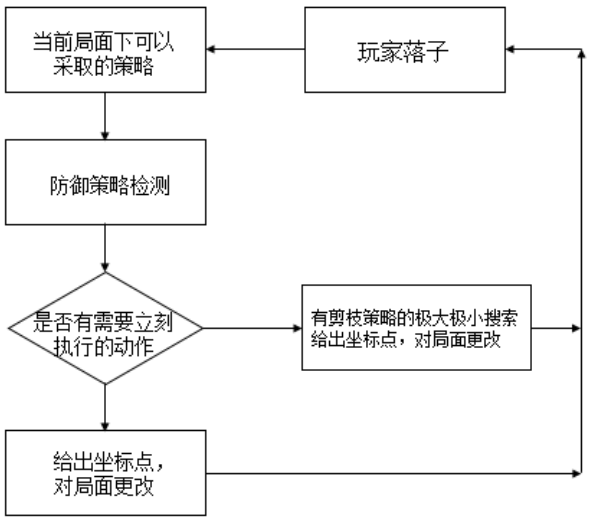
int Utility(MatchState& current_state)
这个函数是对当前局面的效用值的计算；

int Terminal_test(MatchState& current_state)
这个函数是对当前局面判断是不是达到了游戏结束的状态；

int guarantee(const MatchState& current_state, int& r, int& c)
这个函数是对当前局面进行防御的策略检测，如果发现有符合防御策略的，必须要落子的位置，那么就返回 1 并且更改 r 和 c，否则返回 0；

MatchState* actionResult(const MatchState& currentState, Action action, const int player)
返回在当前的 player 操作下，采取特定的 action，对当前局面的更改；

void getAvailableAction(const MatchState& currentState, vector<Action>& actions)
返回在当前的局面下，双方可以采取的落子点。
整个算法程序的表示如下：



六. 测试部分

1.性能测试内容：

一个算法的性能要考虑它的空间和时间复杂度。我们所利用的算法是基于博弈树的深度搜索，当深度不断加深时，电脑能考虑更复杂全面的形式，但搜索所占用的时间和空间会急剧上升。我们在进行实战测试时，偶尔会出现电脑面对复杂棋局迟迟无法落子的情况，可能是由于时间复杂度过大或者内存不足造成的。因此我们从决策所需时间和内存占用来对算法的性能进行测试。

```
int miniMaxSearchForFive(const MatchState& currentState, int& r, int& c)
{
    QVector<Action> actions;
    QVector<double> values;
    int t = 0;
    //首先应该先检验一下是否出现了危险的境地，也就是失误之后必输的情景
    t = guarantee(currentState, r, c);
    if (t != 0){
        return 0;}
    getAvailableAction(currentState, actions);
    int depth = 0;
    //在这里需要我们限制极大极小的深度搜索
    //为了能够减少搜索的时间复杂度，在这里采用alpha beta剪枝策略
    for (int i = 0; i < (int)actions.size(); ++i)
    {
        values.push_back(Min_Value(*actionResult(currentState, actions[i], 1), depth, -10000, 10000));
    }
    int max_idx = 0;
    double max_val = values[max_idx];
    for (int i = 1; i < (int)values.size(); ++i)
    {
        if (values[i] > max_val)
        {
            max_idx = i;
            max_val = values[i];
        }
    }
    r = actions[max_idx].row;
    c = actions[max_idx].col;
    return 1;
}
```

在上图的搜索函数 miniMaxSearchForFive 中，函数体首先对棋局是否出现危险情况进行判断，这个判断由 guarantee 函数实现，倘若对于电脑来说，棋局出现的危险的境地，也就是失误之后必输的情景（guarantee!=0），那么电脑首先的策略一定是防守，而不会调用 Max_Value 和 Min_value 进行深度搜索。所以这种情况下电脑执行防守策略，不进行搜索，占用的时间和内存可以忽略不计，我们设定这时函数 miniMaxSearchForFive 的返回值为 0；当电脑没有面临危险的情景（guarantee == 0）时，它会进行深度搜索，这时候函数需要占用大量的时间和内存，我们需要对这些时间、内存进行统计。我们设定这时函数 miniMaxSearchForFive 的返回值为 1。

在调用 miniMaxSearchForFive 函数的代码中，我们用 clockBegin,clockEnd 分别记录搜索前后的时间，dwValue1,dwValue2 分别记录搜索前后的可用内存。strategy 判断 miniMaxSearchForFive 函数采取的是搜索策略（1）还是防御策略（0）。

```
clock_t clockBegin, clockEnd; //分别记录搜索前后的时间
DWORD dwValue1,dwValue2;//分别记录搜索前后可用内存
int strategy;//strategy 用来判断miniMaxSearchForFive采取的是搜索策略（1）还是防御策略（0）
```

如下图，GlobalMemoryStatue(&men)用来记录当前系统的内存情况，men.dwAvailPhys 是判断当前系统可用物理内存的函数，men.dwAvailPhys/mb 是当前可用物理内存有多少 Mb。在搜索函数执行前后分别进行可用内存和时间的记录。当搜索策略执行时，我们记录下相应的内存消耗（dwValue1- dwValue2）和时间消耗（clockEnd- clockBegin），并进行相应的处理，如记录最大内存消耗和统计平均内存消耗等。

```
GlobalMemoryStatus(&mem);
dwValue1 = mem.dwAvailPhys / mb;
clockBegin = clock();
strategy = miniMaxSearchForFive(currentState,r,c);
GlobalMemoryStatus(&mem);
dwValue2 = mem.dwAvailPhys / mb;
clockEnd = clock();
```

```
if(strategy){    //只有在搜索策略下才记录相应的内存和时间消耗
    this->count = this->count +1;
    if(this->max_memory<(dwValue1-dwValue2)){
        this->max_memory = dwValue1-dwValue2;
    }
    if(this->max_time<(clockEnd-clockBegin)){
        this->max_time = clockEnd - clockBegin;
    }
    this->sum_memory = this->sum_memory + dwValue1-dwValue2;
    this-> average_memory = this-> sum_memory / this->count;
    this->sum_time = this->sum_time + clockEnd-clockBegin;
    this->average_time = this->sum_time / this->count;
}
```

2. 性能测试结果

我们把性能测试的结果放在每局的结束之后。其中包括时间性能(time performance)和内存性能(memory performance)。具体的测试项目包括：

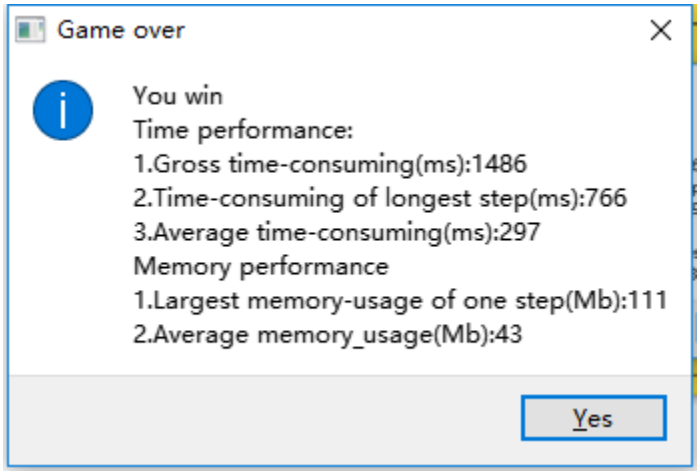
Time performance: #时间性能

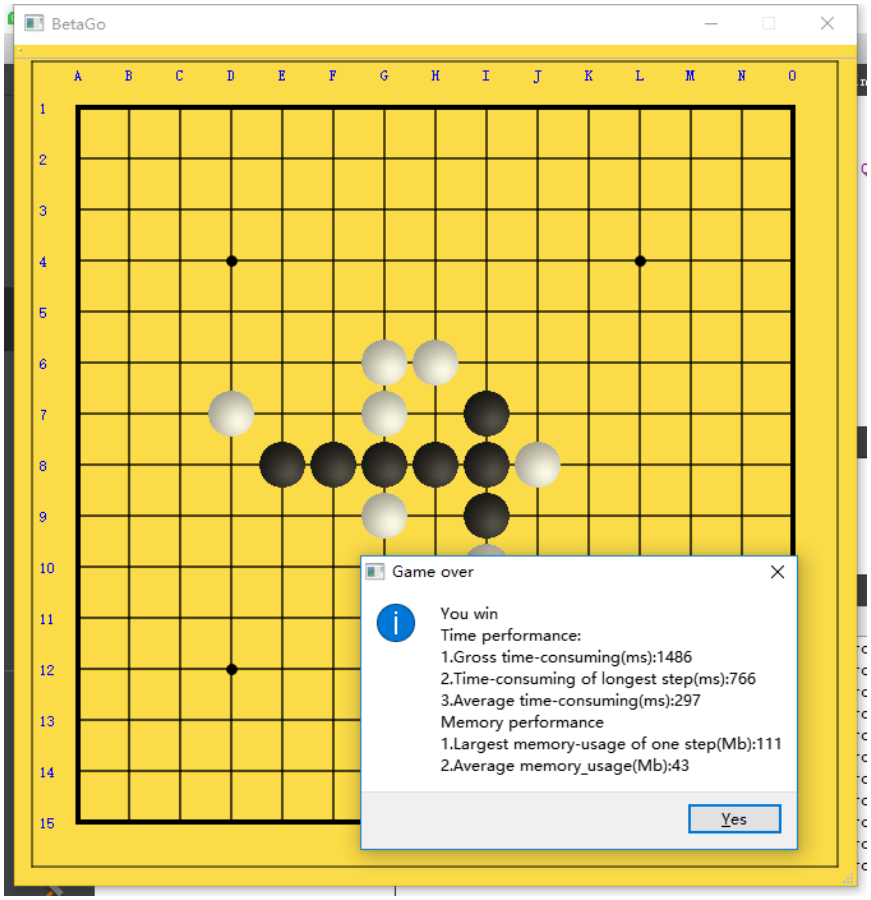
1. *Gross time_consuming(ms) #总搜索时间*
2. *Time-consuming of longest step(ms) #最长一步的搜索时间*
3. *Average time-consuming(ms) #平均每步搜索时间*

Memory performance: #内存测试

1. *Largest memory-usage of one step(Mb) #内存耗费最大的一部*
2. *Average memory_usage(Mb) #平均每步的内存耗费*

每局测试结束后，除了给出输赢信息，还会给出性能测试的结果，如下图





3. 性能测试结果统计

在搜索深度为 3（默认）的条件下，我们进行了 5 次测试，并得出了相关时间、内存参数的均值和标准偏差。电脑平均单步耗费时间为 324.2ms，平均每局最大的耗时为 853.8ms，单步的时长基本小于 500ms，除了少数的情况外，电脑在实战中一般不会拖太久，体验较好。但内存方面，平均每步耗费内存为 39.2Mb，平均单步最大内存耗费为 101.6Mb，有时候会造

成卡顿现象。说明在内存耗费方面，我们的算法还需要进行优化。

项目	最大时间耗费（ms）	平均时间耗费（ms）	最大内存耗费（Mb）	平均内存耗费（Mb）
均值	853.8	324.2	101.6	39.2
标准偏差	1033.542	317.1935	84.18907	28.81319

七. 组内分工

1. 基础功能框架与前后端合调试：**潘如晟**
2. 功能扩展与软件调试：**汪利军**
3. 后端算法及改进优化：**徐晓刚**
4. 性能测试与评价：**金康**
5. 开发文档撰写：**潘如晟、汪利军、徐晓刚、金康**