第九小组·第一阶段·阶段性报告

Part Ⅰ：小组分工

一、图形界面设计[马博靖]

1.登录界面、模式选择界面、游戏界面；

2.棋盘绘制、棋子绘制；

3.模糊化落子位置选择；

二、胜负判定逻辑、自定义倒计时[祝凯]

1.并查集算法；

2.Timer的使用；

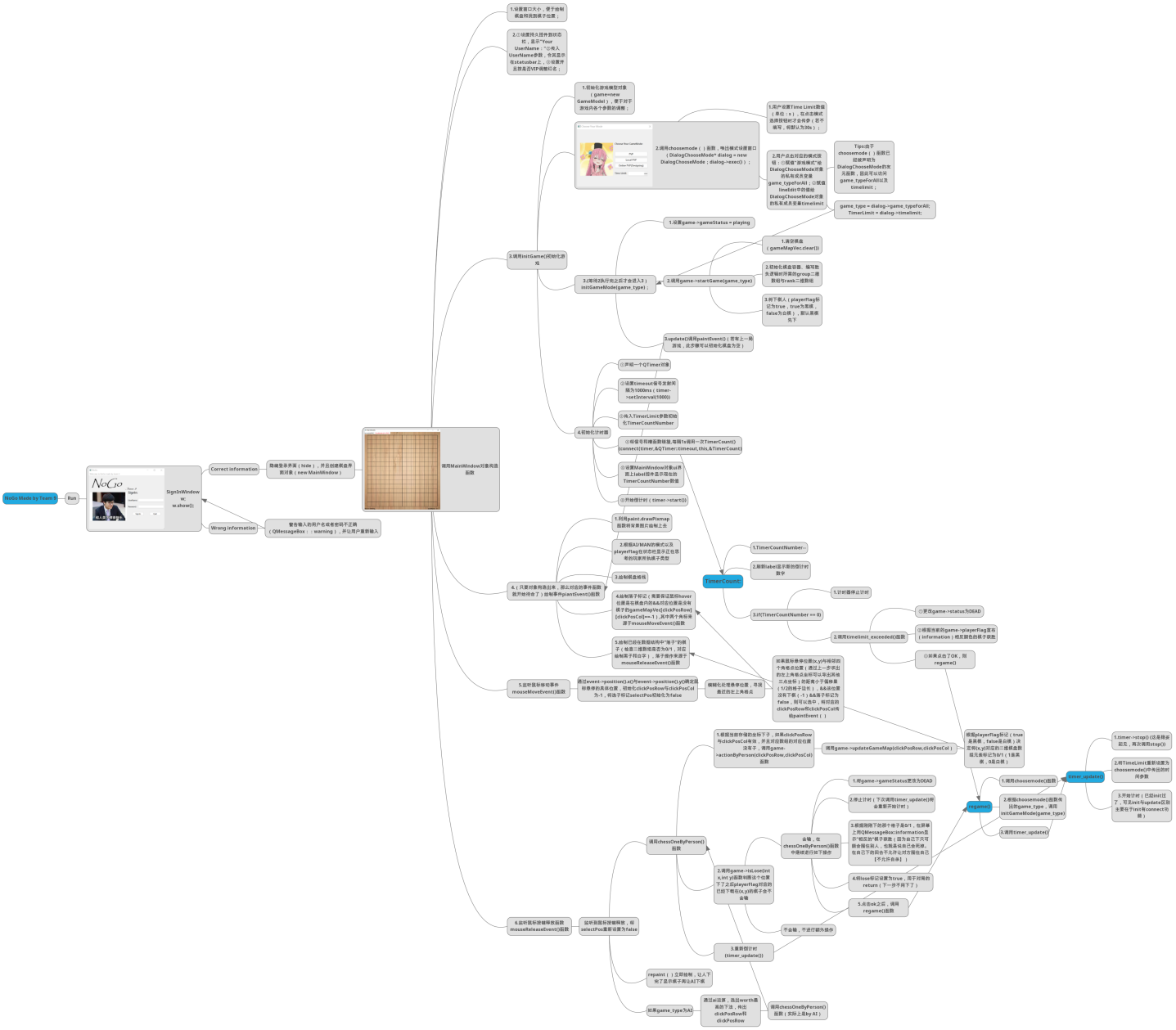
三、初代AI（ahead of schedule）[李雨城]

1.评分机制；

2.全局试探性检验；

Part Ⅱ：代码框架设计

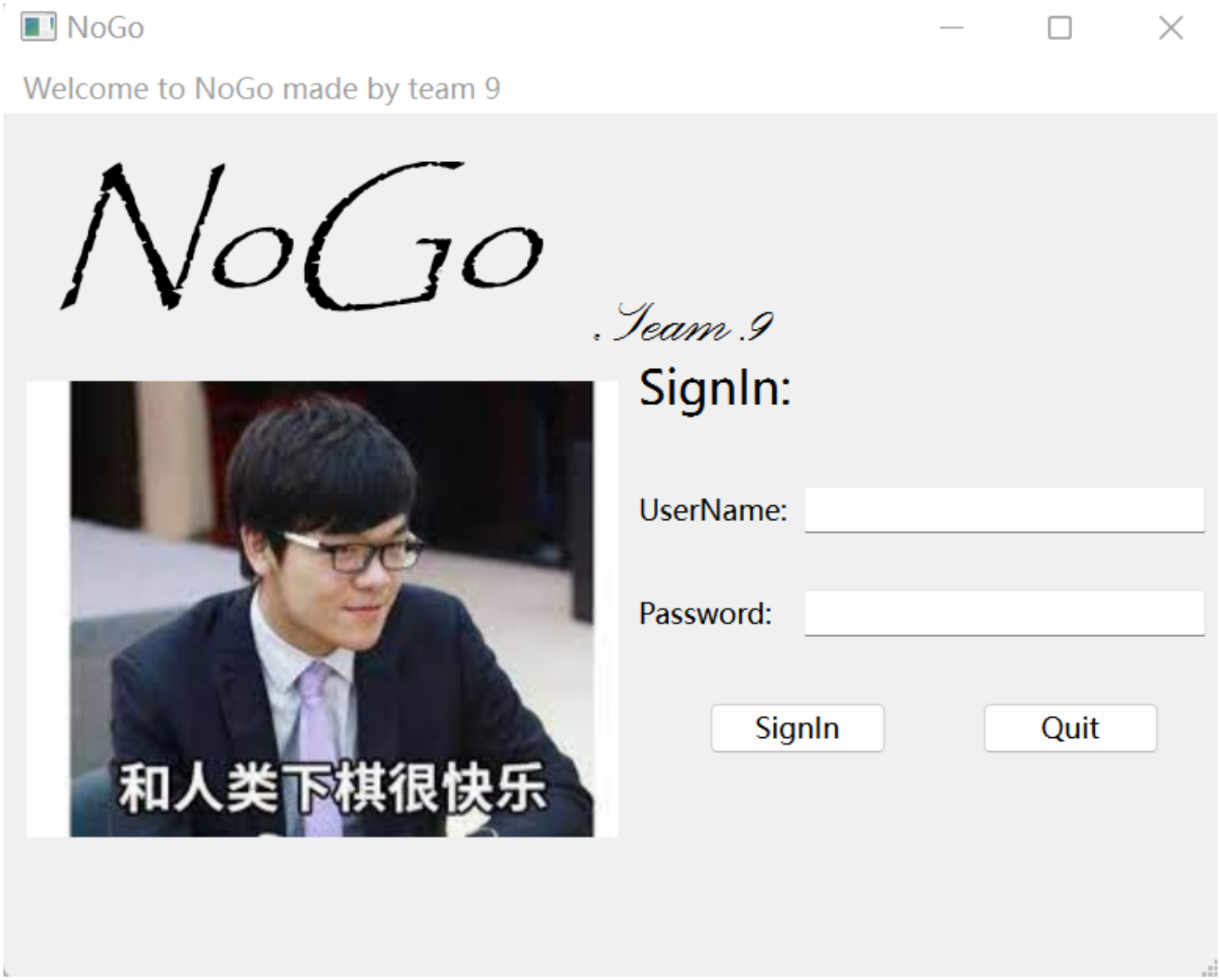
一、整体框架



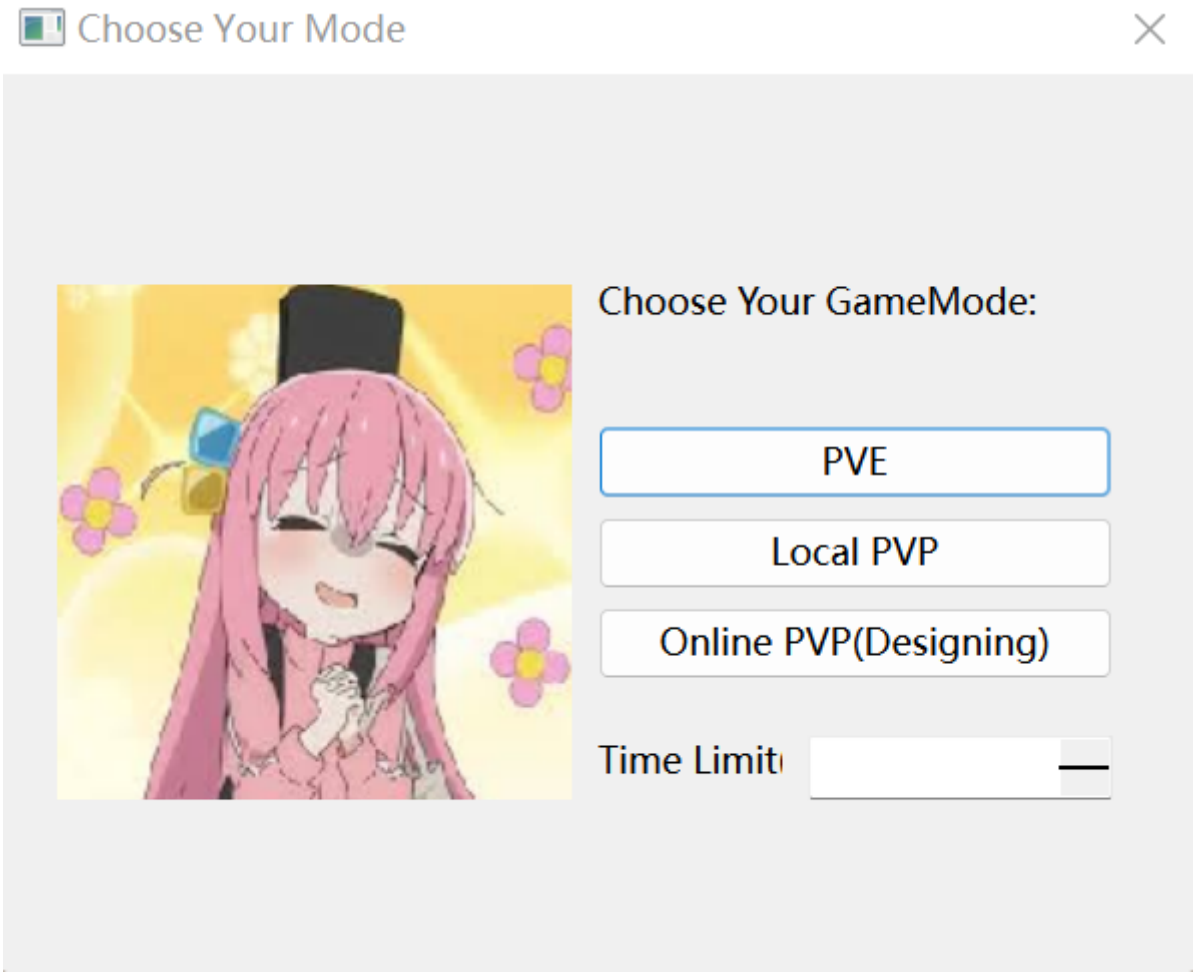
（如果需要详细阅读请戳此链接：MindMap）

以下是应用运行截图：

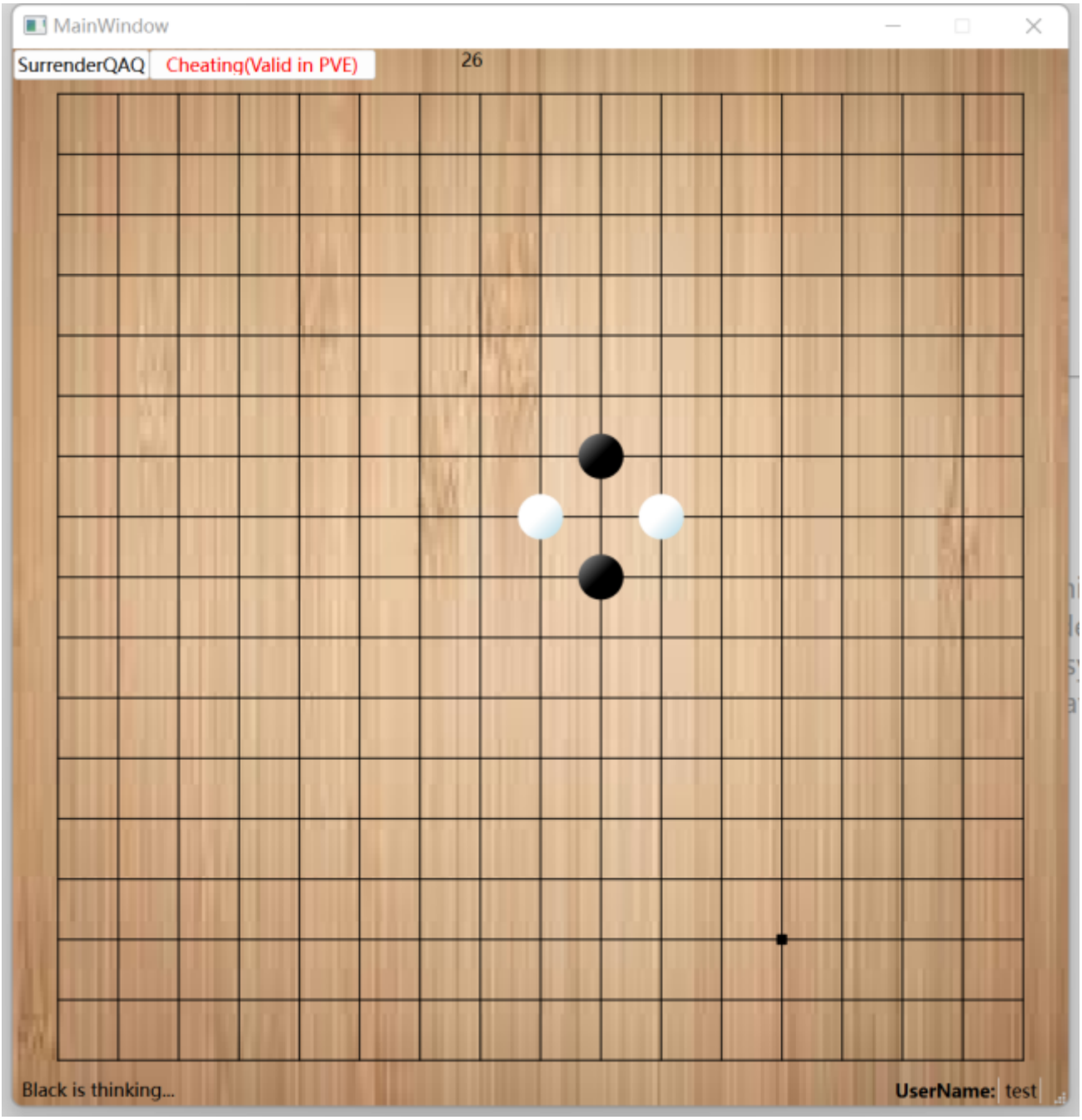
登录界面：


NoGo
Welcome to NoGo made by team 9
NoGo
.Team.9
和人类下棋很快乐
SignIn:
UserName:
Password:
SignIn
Quit

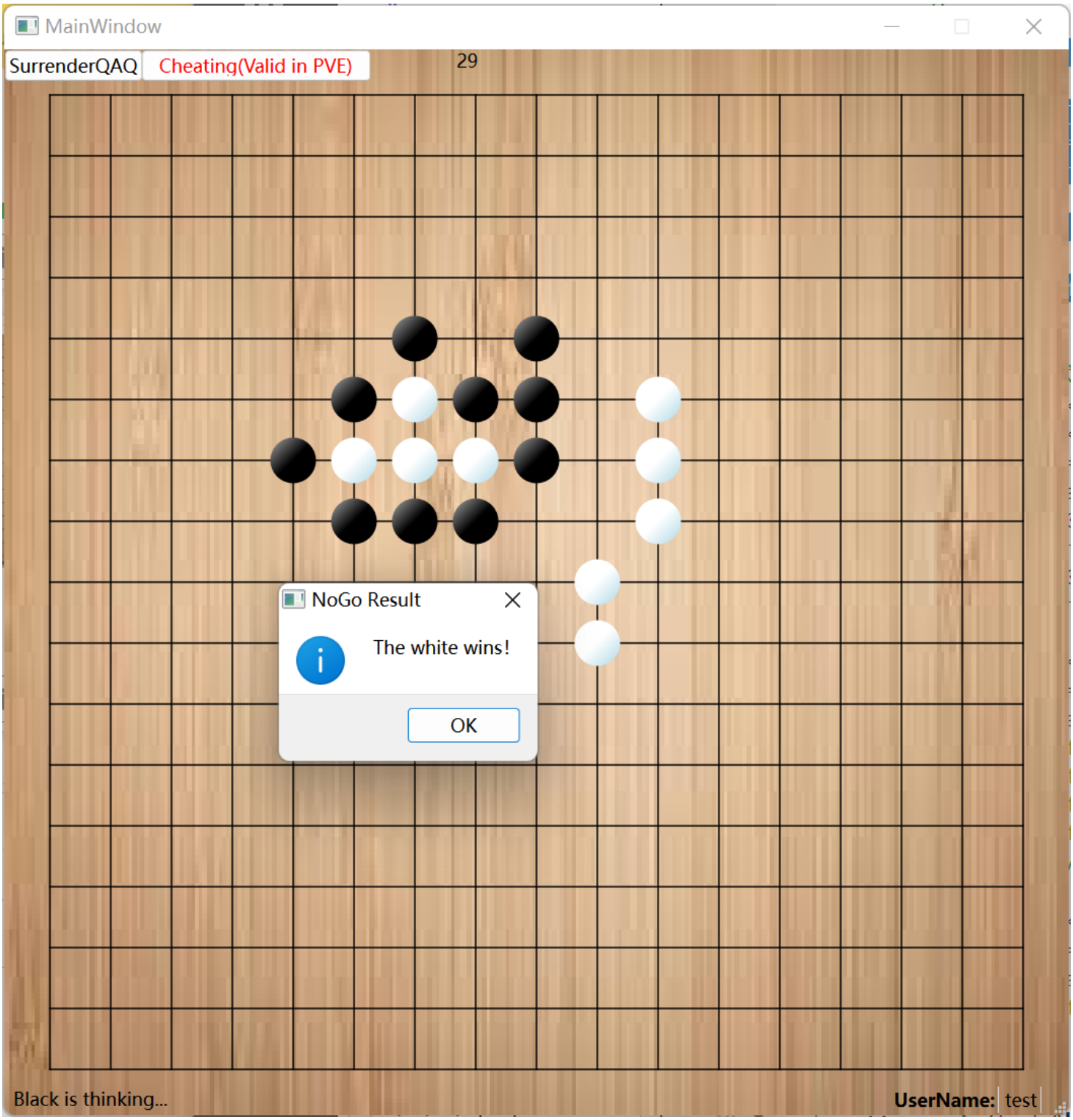
模式选择界面：


Choose Your Mode
Choose Your GameMode:
PVE
Local PVP
Online PVP(Designing)
Time Limit

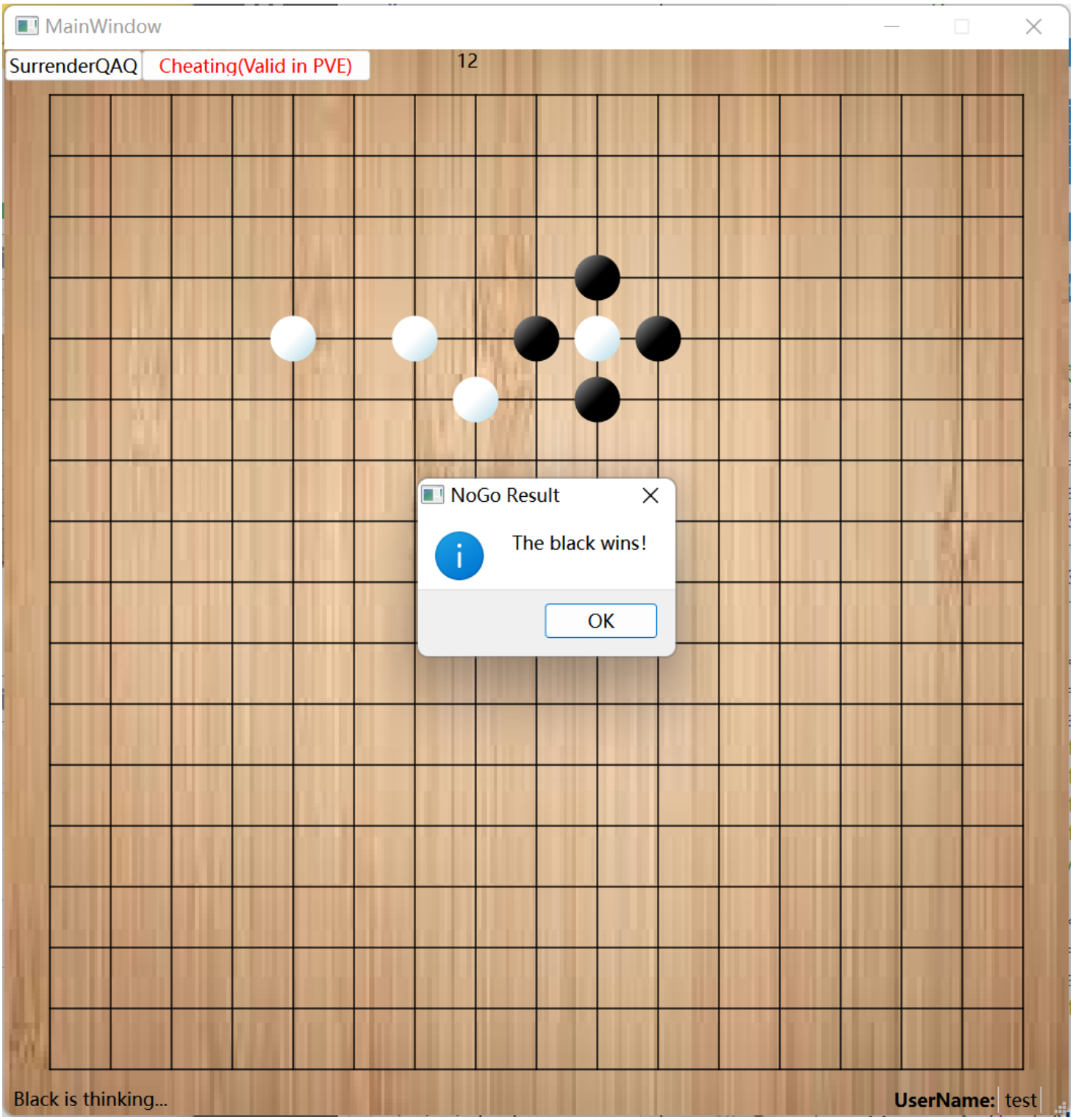
正式游戏界面（棋盘界面）：


MainWindow
SurrenderQAQ
Cheating(Valid in PVE)
26
Black is thinking...
UserName: test

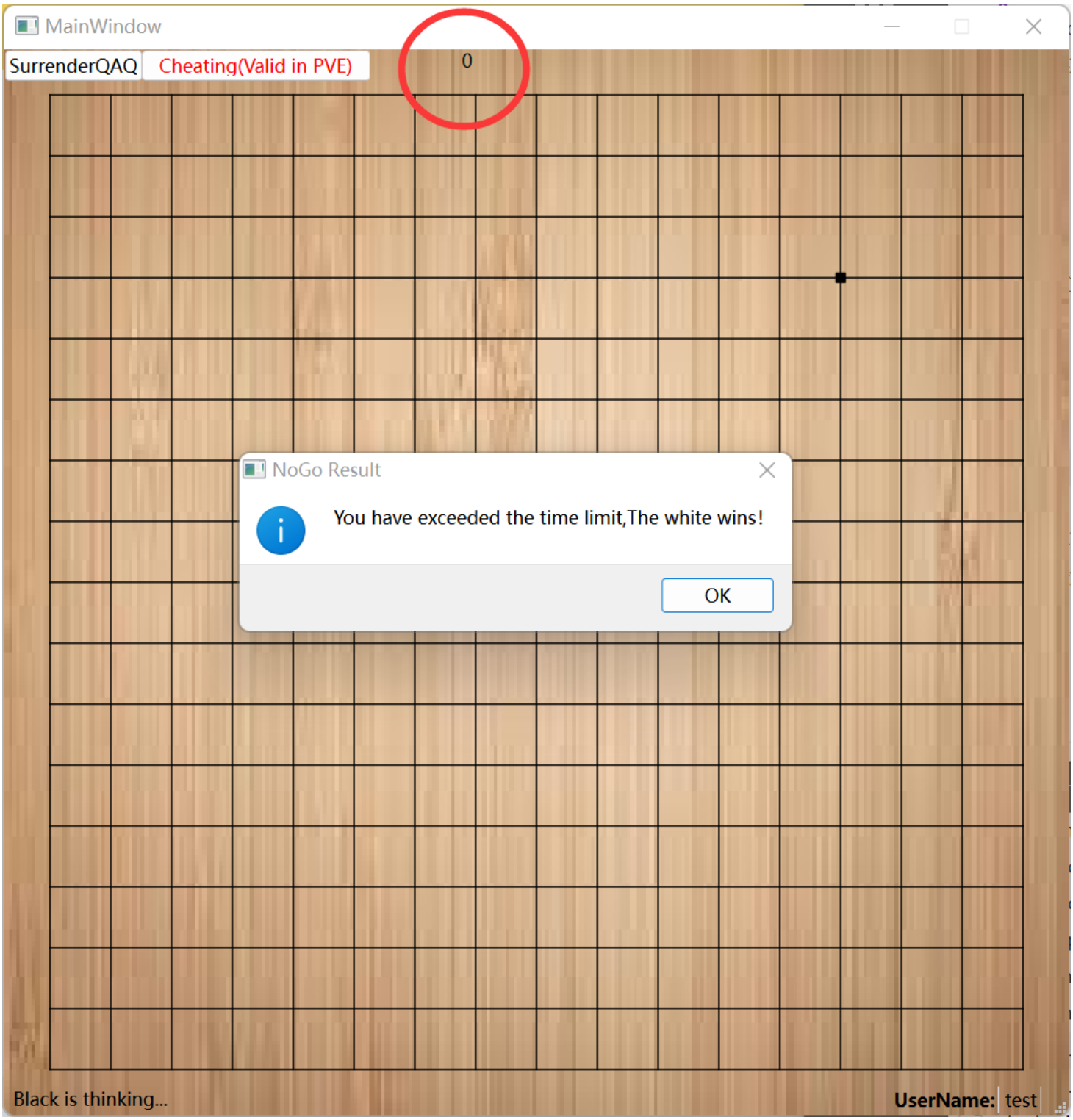
正常“不围”取胜：


MainWindow
SurrenderQAQ
Cheating(Valid in PVE)
29
NoGo Result
The white wins!
OK
Black is thinking...
UserName: test

自杀判负：


MainWindow
SurrenderQAQ
Cheating(Valid in PVE)
12
NoGo Result
The black wins!
OK
Black is thinking...
UserName: test

超时判负：



二、胜负判定逻辑

不围棋落子后只会出现两种情况，落子一方输或无事发生。而落子判负又有两种情况，情况①被围自杀，情况②围住对方棋子群判负。

由于上下左右相邻棋子的气共同判定，考虑使用并查集算法处理同色棋子之间的集合所属关系。

使用二维数组维护并查集，数组的第一维下标为棋子颜色（黑色1，白色0），第二维下标为棋子二维坐标的一维映射（15路棋盘）：

```
#define point_index(row,col) (100 * row + col)
int group[2][group_size];
int rank[2][group_size];
```

简单的并查集实现（带有路径压缩与按秩合并优化）时间复杂度为 O(α(n)):

```
int GameModel::find(int x,int player)
{
    return (x == group[player][x]) ? x : (group[player][x] = find(group[player][x],player));
}

void GameModel::merge(int i,int j,int player)
{
    int x = find(i,player),y = find(j,player);
    if (rank[player][x] <= rank[player][y])
        group[player][x] = y;
    else
        group[player][y] = x;
    if (rank[player][x] == rank[player][y])
        rank[player][y]++;
}
```

每一次落子首先判断是否围住对方棋子:

```
bool GameModel::eat(int row,int col,int player)
{
    int i,new_row,new_col;
    int opp_player = (player + 1) % 2;
    for (i = 0;i < 4;i++) {
        new_row = row + direction[i][0];
        new_col = col + direction[i][1];
        if (if_out(new_row,new_col))
            continue;
        if (gameMapVec[new_row][new_col] == opp_player) {
            if (!if_air(find(point_index(new_row,new_col),opp_player),opp_player)) //对手的某个gr
                return true;
        }
    }
    return false;
}
```

扫描落子周围所有位置，如果有同色棋子对其群组判断是否有气，此处使用了李雨城同学提出的思路，即依次扫描棋盘每一个空格，对空格四周的棋子群组气加一（重复无所谓），对此思路进行改进，在确

定即将判断的棋子群组下标后，找到一个气即可跳出循环返回true，否则返回false。（复杂度常数为棋盘中空白个数*4次扫描，15路棋盘为225*4 = 900）

当然，扫描时可以用数组记录已经扫描的点，但每次判定都要初始化该数组，优化收益并不大:

```
bool GameModel::if_air(int group,int player)
{
    int i,j,k,new_i,new_j;
    for (i = 1;i < BOARD_GRAD_SIZE;i++) {
        for (j = 1;j < BOARD_GRAD_SIZE;j++) {//从1开始 边界没有air
            if (gameMapVec[i][j] == -1) { //扫描每一个空格
                for (k = 0;k < 4;k++) {
                    new_i = i + direction[k][0];
                    new_j = j + direction[k][1];
                    if (if_out(new_i,new_j))
                        continue;
                    if (gameMapVec[new_i][new_j] == player) {
                        if (find(point_index(new_i,new_j),player) == group)
                            return true;
                    }
                }
            }
        }
    }
    return false;
}
```

然后判断是否被围，判定之前首先进行合并操作:

```
int GameModel::merge_group(int row,int col)
{
    int point = point_index(row,col),new_row,new_col,new_point;
    for (int i = 0;i < 4;i++) {
        new_row = row + direction[i][0];
        new_col = col + direction[i][1];
        if (if_out(new_row,new_col))
            continue;
        new_point = point_index(new_row,new_col);
        if (gameMapVec[new_row][new_col] == gameMapVec[row][col])//四周有同色棋子
            merge(point,new_point,gameMapVec[row][col]);
    }

    return find(point,gameMapVec[row][col]);
}
```

随后对返回的棋子群组判断是否有气即可。

完整胜负判断代码如下:

```
bool GameModel::isLose(int row, int col)//merge_group并查集算法已统一边界标准，人人对战无需多次decla
{
    //ai newai;
    //return !newai.ai_check(gameMapVec,BOARD_GRAD_SIZE);
    if (eat(row,col,gameMapVec[row][col]))
        return true; //吃对方判负
    int group = merge_group(row,col);
    if (!if_air(group,gameMapVec[row][col]))
        return true; //紫砂判负
    return false;
}
```

三、初代AI

不围棋的判断输赢的方式，是当回合有子被吃就输。

也就是说，棋盘上有一些地方，只有我方可以落子，设数量为a； 或者只有对方可以落子，设数量为b（程序中并无这两个变量，只是方便理解）；或者双方都能落子。我们要让我方能落子的地方尽可能多，让对方能落子的地方尽可能少。即a-b尽可能大。

所以，用ai_calc()计算在某一点落子的价值。按理来说，价值等于a-b的变化量，不过对于某一步来说，a-b大，即a-b的变化量大，所以代码中只考虑a-b的值:

```
int ai::ai_calc(brd &board,int use,int size){
    //玩家use已落子，判断这个子对他的作用
        int val=0;
        int enemy=1-use;
        for(int i=1;i<size;i++)
            for(int j=1;j<size;j++){
                if(board[i][j]==ai_empty){
                    board[i][j]=use;
                    if(ai_check(board,size)){
                        val+=100;
                    }
                    board[i][j]=ai_empty;


                    board[i][j]=enemy;
                    if(ai_check(board,size)){
                        val-=100;
                    }
                    board[i][j]=ai_empty;
                }
            }
        return val;
    }
```

对于假设在某点落子后的局面，每碰见一个我方能下的地方，价值val加100；发现对方能下的地方，价值val减100。双方都能下的点，价值就会变成100-100=0，是合理的。

另一方面来说，我们下一个点以后，对方就不能下这个点了。也就是说，假如这个点对对方很重要，我们也要考虑下在这个点。因此，主函数run()调用两次ai_calc()，将这个点对双方价值之和作为这个点的价值。

在开局时，各个点价值都一致，我们要尽量下在边角，因为边角更容易被对方围住，对方在这里下的棋就有限了。并且，我们尽量不要把点连起来，连起来的点不容易被对方围住，所以加强边角的价值和我方棋子对角线的价值。

之后主函数run()选择价值最高的点作为返回值:

```
pii ai::run(brd &board,int use,int size){
    //程序的接口
        const int enemy=1-use;
        int worth[size][size];
        for(int i=1;i<size;i++)
            for(int j=1;j<size;j++){
                if(board[i][j]!=ai_empty)worth[i][j]=-1e7;
                else{
                    if(i<=2||i>=size-2)worth[i][j]+=rand()%10;
                    if(j<=2||j>=size-2)worth[i][j]+=rand()%10;
                    bool zs=(i-1>=0&&j-1>=0&&board[i-1][j-1]==use);
                    bool ys=(i-1>=0&&j+1<size&&board[i-1][j+1]==use);
                    bool zx=(i+1<size&&j-1>=0&&board[i+1][j-1]==use);
                    bool yx=(i+1<size&&j+1<size&&board[i+1][j+1]==use);
                    int summ=(int)zs+(int)ys+(int)zx+(int)yx;
                    worth[i][j]+=summ*20;
                    board[i][j]=use;
                    if(ai_check(board,size)){
                        worth[i][j]=ai_calc(board,use,size);
                    }
                    else worth[i][j]=-1e6;
                    board[i][j]=ai_empty;
                    board[i][j]=enemy;
                    if(ai_check(board,size)){
                        worth[i][j]+=ai_calc(board,enemy,size);
                    }
                    board[i][j]=ai_empty;
                }
            }
        int mn=-1e8;
        int x=1,y=1;
        for(int i=1;i<size;i++)
            for(int j=1;j<size;j++)
                if(worth[i][j]>mn){
                    mn=worth[i][j];
                    x=i;
                    y=j;
                }
        return make_pair(x,y);
    }
```

如何判断一个点是否可以下？即：如何计算a和b?我们要假设一方下在了这里，调用check()判断是否有子被围住。

```
bool ai::ai_check(brd &board,int size){
        //检测当前棋盘状态
        for(int i=1;i<10000;i++){ai_fa[i]=i;air[i]=0;}
        for(int i=1;i<size;i++)
            for(int j=1;j<size;j++)
                if(board[i][j]!=ai_empty){
                if(i>0&&board[i][j]==board[i-1][j])
                    ai_merge(i*size+j,i*size-size+j);
                if(j>0&&board[i][j]==board[i][j-1])
                    ai_merge(i*size+j,i*size+j-1);
            }
        for(int i=1;i<size;i++)
            for(int j=1;j<size;j++)
                if(board[i][j]==ai_empty){
                    if(i>=1)air[findai_fa((i-1)*size+j)]++;
                    if(j>=1)air[findai_fa((i)*size+j-1)]++;
                    if(i<size-1)air[findai_fa((i+1)*size+j)]++;
                    if(j<size-1)air[findai_fa((i)*size+j+1)]++;
                    //是否重复无所谓
                }
        for(int i=1;i<size;i++)
            for(int j=1;j<size;j++)
                if(board[i][j]!=ai_empty){
                    if(air[findai_fa(i*size+j)]==0)return false;
                }
        return true;
    }
```

判断方式：用并查集findai_fa()ai_merge()，把所有相连的子合并成组，每个空白处给四面的棋组加气:

```
int ai::findai_fa(int i){
        //并查集
        if(i==ai_fa[i])return i;
        else ai_fa[i]=findai_fa(ai_fa[i]);
        return ai_fa[i];
}
void ai::ai_merge(int sour,int dest){
        //并查集
        ai_fa[findai_fa(sour)]=findai_fa(dest);
    }
```

不用考虑气是否重复，因为气只有“有”“无”两个状态。

其他细节：不能落子的空地，价值减一大截；已经有子的地方，价值减更大一截。

时间复杂度：设棋盘14*14，共200点。run()枚举每一方每一点，400次；calc()查询每点有哪方能下，400次；check

一共400*400*400=64,000,000。按理说不到1秒就能算完，可惜要卡一下，就当作ai在思考了。由此可见，我们不可

而且假如我们枚举了两步，可能我们走完第一步之后，对方就把我们要走的第二步填上了，得不偿失。

Part III、遇到的问题和解决方式

(由于第一阶段比较简单，所以遇到的问题也很初级，但确实是真实遇到，并且也花了一定时间去处理，在此说明)

一、窗口hide()之后未真正关闭，导致游戏判断超时出错

开始的时候，我们通过dialogchoosemode这个窗口打开mainwindow（棋盘界面），并且当一局游戏结束后将mainwindow进行hide()操作、弹出新的dialogchoosemode窗口，通过该窗口打开新的棋盘界面。

这样导致旧窗口虽然被隐藏，但没有真正关闭，其计时器继续计时，导致在进行第二局游戏的时候出现超时提醒。

并且还有潜在的问题是内存溢出，一直打开窗口但没能真正关闭，占用大量不必要的资源。

解决方案：用mainwindow的构造函数打开dialogchoosemode窗口，并且保证mainwindow这个窗口一直不被关闭。一局游戏结束时，只是再次初始化倒计时参数以及棋盘格局，这样就能不会在新的一局游戏时出现上一局游戏的超时提醒了；也防止了内存的资源浪费。

二、绘制棋子渐变时只能显示0位置颜色，而无法实现渐变

解决方案:
gradient构造函数中的四个参数应该分别为x1,y1,x2,y2，对应渐变的0起始点和1终止点（线性渐变的两个端点）；

drawEllipse函数的四个参数分别为椭圆所在**矩形的最左上角（之前误以为是椭圆中心）**、长轴长度、短轴长度；

传参的时候注意这一点（之前是误以为两个参数应该保持一致了）。

Part IV、可能的优化

一、关于胜负判断

棋子坐标的映射可以hash一下节省一些数组空间，为了代码清晰（lazy），暂时没有增加hash操作。

二、关于AI

1.落子应该贴着对方落子。这样可以更容易被吃。还要减少我方棋子上下左右四个点的价值评估。

2.我们知道场面上有多少点只有黑子能下，有多少点只有白子能下，就可以轻松地实时给出胜率。

3.开局的子倾向于贴着左上角走，看着很呆，可以优化一下。

三、完成Task1过程中想到的除了任务要求以外的功能（当然会先做完正常任务要求的功能）

1.保存对局，下次登录可以继续下上一盘棋；

2.练习模式（棋谱模式）；

3.实现gpt的接口（dreaming，暂时还不知道怎么弄）；