实验报告

2022080910014　陈梦桢

一、游戏规则及基本功能

在对局中，无禁手和时间限制，其他规则与传统的五子棋规则相同。有如下基本功能：棋局开始前玩家可选先后手；落子后有高亮提示；可多步悔棋。

二、具体实现

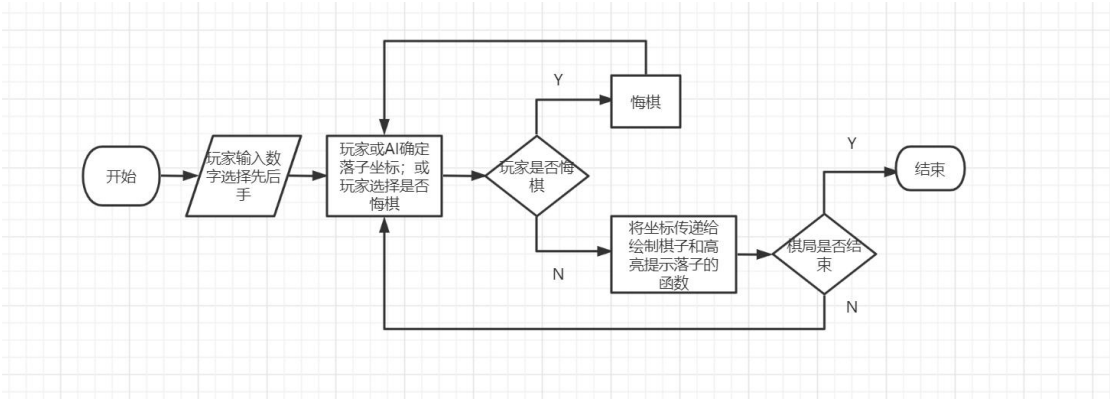
（一）、备战

使用了 Visual Studio 2019 作为开发工具，并配置了 eaxyx 以实现人通过鼠标与电脑之间的交互，以纯色图片作为背景图片并通过图形绘制作为棋盘。

（二）、实战

整个程序分为五部分文件：FIR、Basic、Stack、Human 和 AI。FIR 是包含 main 函数的文件，主要负责整个棋局的实现。Basic 负责实现棋局的各种功能，如加载菜单、绘制棋盘、判断棋局是否结束等功能。Stack 内主要是栈的结构声明和操作，用于辅助完成悔棋的功能。Human 负责实现玩家落子以及悔棋的功能。AI 负责实现 AI 的所有的功能，如 Minmax 搜索，α - β 剪枝，棋局评估等功能。

1、在 FIR 文件中，定义了一些整型全局变量来表示当前落子颜色、棋盘、当前落子方、先手方、后手方，以及用于记录各个回合的棋局的全局栈。整个对局大致流程如下：



2、在 Basic 文件中

（1）menu 函数为玩家提供选择先后手的服务；

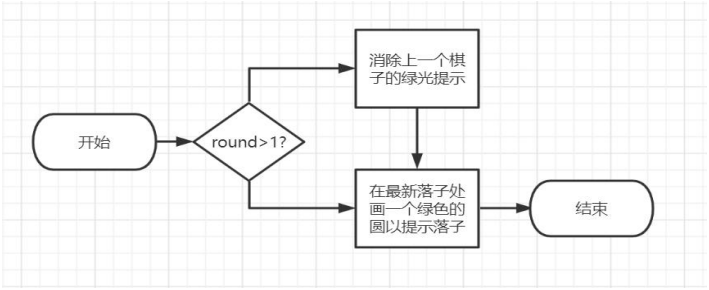
（2）drawLine 函数负责画棋盘的线，drawPoint 函数负责画天元等 5 个小点儿，drawBoard 函数则结合前两个函数共同绘制最终的棋盘；

（3）correct 函数将获取到的鼠标单击的分辨率坐标转化为棋盘（二维数组）坐标，允许的鼠标单击误差为以棋盘交线为中心，半个格子的长度为半径的圆。

```
int correct(int t) {
    // 修正坐标 以棋盘网格线左上角的点为原点,一个LSTEP为一个单位长度
    int t1, t2, T;
    t -= LSTEP / 2;
    t1 = t / LSTEP * LSTEP;
    t2 = (t / LSTEP + 1) * LSTEP;

    if (abs(t1 - t) >= abs(t2 - t)) T = t2 / LSTEP;
    else T = t1 / LSTEP;

    return T;
}
```

（4）drawPiece 函数接收落子的棋盘坐标，将其转化为分辨率坐标，后通过绘制填充颜色的圆形以绘制棋子；

（5）hightlight 函数接收五个整型参数：上一棋子坐标、当前落子坐标和当前回合数，完成落子提示的相关功能，其流程如下：



具体实现代码如下：

```
void highlight(int x, int y, int last_x, int last_y, int round) {
    if (round > 1) {
        last_x = last_x * LSTEP + LSTEP / 2;
        last_y = last_y * LSTEP + LSTEP / 2;
        if (gPiece == BPIECE) {
            setfillcolor(WHITE);
            fillcircle(last_x, last_y, LSTEP / 2 - 1);
        }
        else {
            setfillcolor(BLACK);
            fillcircle(last_x, last_y, LSTEP / 2 - 1);
        }
        setlinecolor(BLACK);
        circle(last_x, last_y, LSTEP / 2 - 1);
    }

    //最新落子
    x = x * LSTEP + LSTEP / 2;
    y = y * LSTEP + LSTEP / 2;
    setlinecolor(GREEN);
    circle(x, y, LSTEP / 2 - 2);
    circle(x, y, LSTEP / 2 - 1);
}
```

（6）endJudge 函数接收落子坐标，在该坐标的棋子上分别判断纵向、横向、正斜向、负斜向是否有连续五个相同颜色的棋子，从而判断棋局是否结束；

（7）turnRound 函数转换落子颜色；

（8）end 函数接收两个整型参数：flag（获胜方）和 round（round 大于或等于 255 即为平局），最后结束棋局。

3、在 Stack 文件中

（1）定义了两个数据结构以存储棋局信息，STACKDATE 记录某一回合棋盘上的落子情况以及最新落子的坐标；STACK_RECORD 记录整个棋局的对局落子情况。代码如下：

```
typedef struct {
    int x;
    int y;
    char board[225];
}STACKDATE;
```

```
typedef struct {
    int top;
    STACKDATE date[226];
}STACK_RECORD;
```

（2）push_record 函数完成入栈操作，即记录棋局情况；

（2）pop_withdraw 函数完成出栈操作，即将前 2 回合的棋局数据导入表示棋盘的二维数组中（不满 2 回合则导入最初的数据，即棋盘上没有棋子），代码如下：

```
int pop_withdraw(int* x, int* y) {
    int count = 0, flag = 0;
    if (gStack.top <= 0) {
        return 0;
    }
    else if (gStack.top < 2) {
        gStack.top = 0;
    }
    else if(gStack.top >= 2){
        gStack.top -= 2;
    }
    *x = gStack.date[gStack.top].x;
    *y = gStack.date[gStack.top].y;
    for (int i = 0; i < 15; i++) {
        for (int j = 0; j < 15; j++) {
            gBoard[i][j] = gStack.date[gStack.top].board[count] - '0';
            count++;
        }
    }
    return 1;
}
```

4、在 human 文件中

（1）human 函数负责获取鼠标单击的分辨率坐标，判断单击的区域从而判断玩家是要落子或是悔棋。若是落子则将分辨率坐标转化为棋盘坐标，若是悔棋则调用 withdraw 函数。

（2）withdraw 函数通过调用记录棋局的栈及其出栈操作（定义在 Stack 文件中），重新绘制棋盘及棋子，并高亮提示最后落下的棋子。需要注意的是，若是若回合数大于 2 则悔棋后落子方不变，若回合数为 2 则悔棋后落子方需要交换。代码如下：

```
void withdraw() {
    int x = 0, y = 0;
    loadimage(&gBgp_withdraw, _T("..\\resource\\background.png"));
    if (gStack.top == 1) {
        if (gPlayer == gPlayer2) {
            gPlayer = gPlayer1;
        }
        else {
            gPlayer = gPlayer2;
        }
        gPiece = BPIECE;
    }
    pop_withdraw(&x, &y);
    //重画棋盘及全部棋子
    clearrectangle(0, 0, 600, 600);
    putimage(0, 0, &gBgp_withdraw);
    drawLine();
    drawPoint();
    for (int i = 0; i < 15; i++) {
        for (int j = 0; j < 15; j++) {
            if (gBoard[i][j] != EMPTY) {
                drawPiece(i, j);
            }
        }
    }
    //高亮提示最后的棋子
    if (gStack.top > 0) {
        //形参round=0，则无需用到参数last_x和last_y，故二者取值任意
        highlight(x, y, 0, 0, 0);
    }
}
```

5、在 AI 文件中

（1）findBoundary 函数的作用是缩小博弈树的有效搜索范围，起到一定程度的预剪枝作用。该函数通过遍历表示棋盘的二维数组来确定能囊括所有棋子的最小矩形区域，并在此基础上在四个方向上分别向外至多扩展指定大小的单位长度，在本程序中设定为扩展至多一个

单位长度，为博弈树搜索做准备。

（2）getScore 函数和 evaluate 函数完成 AI 对棋局的评估。评估的标准及评分的方法很大程度上影响了 AI 的智能程度。

① 评分标准：首先先定义棋型。棋型分为三大类：活棋（某一方向上，若干个连续同色棋子的两侧没有异色棋子或边界）、眠棋（某一方向上，若干个连续同色棋子的两侧只有一侧有异色棋子或边界）和死棋（某一方向上，若干个连续同色棋子的两侧均有异色棋子或边界）。一般情况下，连续的棋子数量相同时，就威胁性而言，活棋>眠棋>死期。例如，“活X”指在某一方向上，存在 X 个连续同色棋子且棋型为“活”。其次确定各个棋型的权重。活一到活五分别为 20,300,1800,5000,100000；眠一到眠五分别为 10,100,600,2100,100000；死一到死五分别为 0,0,0,0,100000。

② getScore 函数通过遍历整个棋盘（二维数组），依次遍历四个方向，

```
for (int dir = 0; dir < 4; dir++) {
    for (int i = 0; i <= 14; i++) {
        for (int j = 0; j <= 14; j++) { ... }
    }
}
return score;
```

统计己方在某一方向上连续的棋子数，以及确认两侧是否有对方棋子或边界，

```
while (x >= 0 && x <= 14 && y >= 0 && y <= 14 && vBoard[x][y] == player ){
    n_player++;
    x = x + DIR[dir].x, y = y + DIR[dir].y;
}
if (x >= 0 && x <= 14 && y >= 0 && y <= 14) {
    if (vBoard[x][y] == rival || x < 0 || x > 14 || y < 0 || y > 14) {
        n_rival++;
    }
}
```

从而确定棋型和分数。

```
    if (n_player > 5) n_player = 5;
    score += SCORE[n_rival][n_player];
}
```

③ evaluate 函数通过调用 getScore 函数分别算出玩家的得分和 AI 的得分，相减后便得到了最后的得分。（这在一定程度上保证了 AI 的智能程度，避免了片面性）

```
aiScore = getScore(vBoard, ai);
huScore = getScore(vBoard, human);

return aiScore - huScore;
```

（3）maxminSearch 函数，主要是运用了 α - β 剪枝算法的极大极小值搜索。

在零和博弈中，玩家均会在可选的选项中选择将其 N 步后优势最大化或者令对手优势最小化的选择。将双方决策过程视作一颗决策树，若决策树某一层均为己方决策依据状态（即接下来是己方进行动作），则己方必定会选择使得己方收益最大化的路径，将该层称为 MAX 层。若决策树某一层均为对手决策依据状态（即接下来是对手进行动作），则对手必定会选择使得己方收益最小化的路径，将该层成为 MIN 层。由此，一个极小化极大决策树将包含 max 节点（MAX 层中的节点）、min 节点（MIN 层中的节点）和终止节点（博弈终止状态节点或 N 步时的状态节点）。每个节点对应的预期收益成为该节点的 minmax 值。

博弈树通过递归实现，并在搜索前调用 foundBoundary 函数进行预剪枝，提高算力（这在上文中已提到），依次试探落子。

```
findBoundary(boundary, vBoard);
y1 = boundary[0];
y2 = boundary[1];
x1 = boundary[2];
x2 = boundary[3];
```

在 max 层改变 α 值，其取值来源于叶结点或者下一层的 α 值和 β 值中的较大值；在 min 层改变 β 值，其取值来源于叶结点或者下一层的 α 值和 β 值中的较小值；

```
//max层
if (depth % 2 == 0) {
    for (int i = x1; i <= x2; i++) {
        for (int j = y1; j <= y2; j++) {
            if (vBoard[i][j] == EMPTY) {
                vBoard[i][j] = player;
                score = maxminSearch(depth - 1, player, vBoard, alpha, beta);
                vBoard[i][j] = EMPTY;
                if (depth == SDEPTH) {
                    RECORD_SCORE[i][j] = score;
                }
                if (score > alpha){          // 对于AI，更新极大值
                    alpha = score;
                }
                if (alpha >= beta){          // Alpha-剪枝
                    return alpha;
                }
            }
        }
    }
    return alpha;
}
```

```
//min层
else {
    for (int i = x1; i <= x2; i++) {
        for (int j = y1; j <= y2; j++) {
            if (vBoard[i][j] == EMPTY) {
                vBoard[i][j] = player;
                score = maxminSearch(depth - 1, player, vBoard, alpha, beta);
                vBoard[i][j] = EMPTY;
                if (depth == SDEPTH) {
                    RECORD_SCORE[i][j] = score;
                }
                if (score < beta) {          // 对于human，更新极小值
                    beta = score;
                }
                if (alpha >= beta) {         // Beta-剪枝
                    return beta;
                }
            }
        }
    }
    return beta;
}
```

在终止节点（叶结点）调用 evaluate 函数进行评估返回分数。

```
//叶结点
if (depth == 0) {
    return evaluate(vBoard);
}
```

递归结束后，将最终得分记录到二维数组中，与落子点坐标相对应。

（4）AI 函数包含 AI 确定落子坐标的全部过程。

我们知道，对于五子棋而言，棋盘的中心（天元）价值是非常高的，所以只要天元上未落子，AI 必定在该处落子，为了显得更自然，调用了 Sleep 函数。

```
if (gBoard[CENTRE][CENTRE] == EMPTY) {
    Sleep(500);
    *x = CENTRE;
    *y = CENTRE;
    flag = 1;
}
```

若 AI 后手，一般情况下，玩家第一回合都是落子于天元处，为了节省时间，第二回合不采用极大极小搜索，而是随机在天元四周的八个点落子。

```
else if (round == 2) {
    Sleep(500);
    secondRound(x, y);
    flag = 1;
}
```

```
void secondRound(int* x, int* y) {
    srand((unsigned int)time(0));
    int t;
    t = rand() % 8;
    *x = CENTRE + DIR[t].x;
    *y = CENTRE + DIR[t].y;
}
```

其他情况下则采用博弈树搜索。出于安全考虑，先将表示棋盘的二维数组的数据复制到另外一个数组，并用该数组进行博弈树搜索。

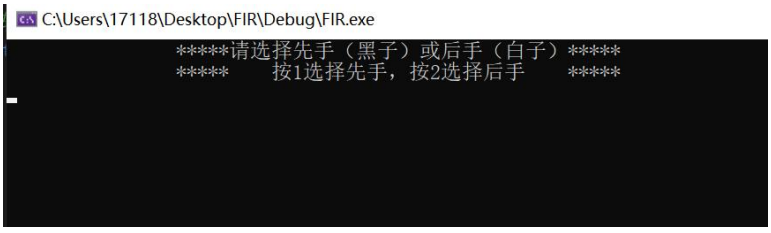
```
else {
    int** vBoard = (int**)malloc(sizeof(int*) * 15);
    for (int i = 0; i < 15; i++) {
        if (vBoard != NULL) {
            vBoard[i] = (int*)malloc(sizeof(int) * 15);
        }
    }
    //把当前棋盘复制到假想棋盘中
    for (int i = 0; i < 15; i++) {
        for (int j = 0; j < 15; j++) {
            if (vBoard != NULL) {
                if (vBoard[i] != NULL) {
                    vBoard[i][j] = gBoard[i][j];
                }
            }
        }
    }

    if (player == BPIECE) {
        player = WPIECE;
    }
    else {
        player = BPIECE;
    }
    maxminSearch(SDEPTH, player, vBoard, alpha, beta);
```

最后，遍历记录分数的数组，最大值对应的坐标即为 AI 落子的坐标。

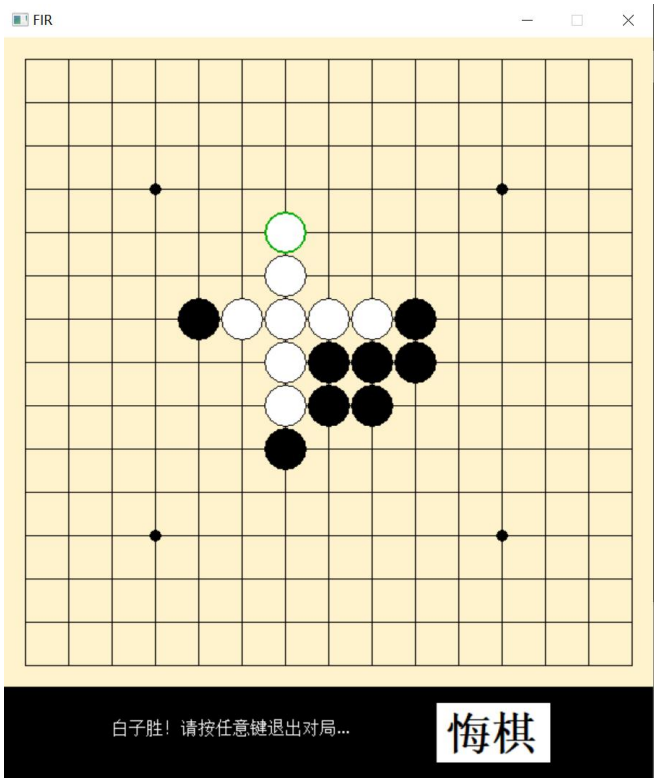
```
for (int i = x1; i <= x2; i++) {
    for (int j = y1; j <= y2; j++) {
        if (RECORD_SCORE[i][j] > maxScore && gBoard[i][j] == EMPTY) {
            maxScore = RECORD_SCORE[i][j];
            *x = i;
            *y = j;
        }
    }
}
```

三、运行情况及 AI 棋力

开始



对局



值得说明的是，随着棋局的进行，棋子会越来越靠近边界，foundBoundary 函数划定的搜索范围会越来越大，搜索速度也会越来越慢。由于算法的局限性，在死局的时候，AI 偏向于走活四或者眠四，无法通过 VCF、VCT 扭转局势。

至于 AI 的算力，在不限时的情况下能够战胜普通人，胜率较高（测试对象为自己，室友，以及一些朋友）。一般人不留神的话，很容易被 AI 打败。

四、参考文献

1、五子棋人工智能算法设计与实现_刘瑞
2、GitHub、CSDN 等博客帖子（实在找不到是哪些了）