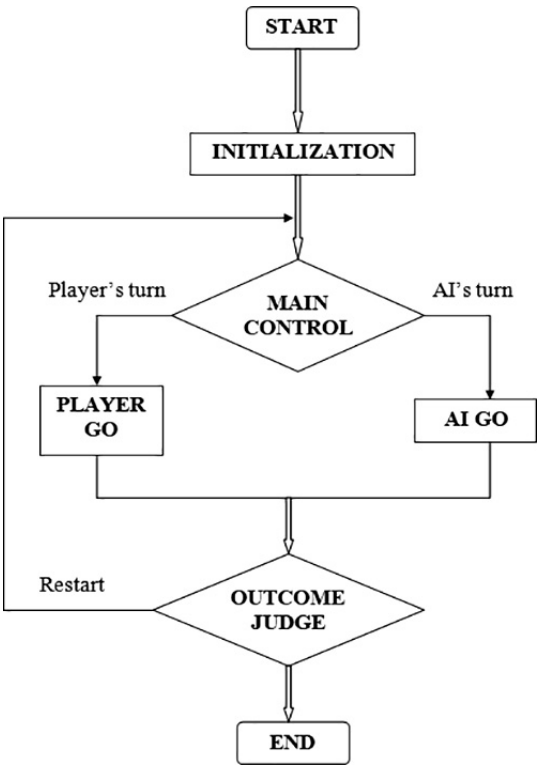
五子棋：人机博弈

一、五子棋人机博弈背景

五子棋是一款经典的两人对弈的纯策略型棋类游戏. 相对于国际象棋、中国象棋、围棋、日本将棋，五子棋简单易学，但是精通五子棋并非易事. 五子棋棋盘通常 15×15，即 15 行，15 列，共 225 个交叉点，即棋子落点；棋子由黑白两色组成，黑棋 123 颗，白棋 122 颗。游戏规则为黑先白后，谁先五子连成一条直线谁赢，其中直线可以是横的、纵的、45 度、135 度. 而人机博弈，顾名思义就是玩家和电脑进行博弈，玩家首先五子一线，则玩家获胜，否则电脑获胜.

二、人机博弈需求分析

实现一个简单的五子棋游戏，能够让玩家与电脑按次序在棋盘上进行落子，并能够判断输赢，简要流程图如下：



通过参考 L. Venkateswara Reddy 的论文 [1]，将五子棋人机博弈分成以下 5 个模块，分别为：

Module:

- 游戏环境模块 (Game environment build Module)
- 游戏控制模块 (Game control Module)
- 游戏状态模块 (Game state Module)
- AI 模块 (AI Module)
- 输出判断模块 (Outcome judge Module)

三、人机博弈详细设计与实现

该项目使用 *Java* 语言来一一实现上述的 5 个模块，*Java* 是面向对象的编程语言，可以将上述每个模块都封装成一个类，通过编写类方法来实现各个模块的功能，最终将这 5 个模块的功能集中到一个主类中，实现五子棋人机博弈. 实现各模块之前，需要定义几个常量，如下：

```
1           /**
2            * 棋盘的大小
3            */
4           public static final int N = 15;
5           /**
6            * 棋盘间距
7            */
8           public static final int size = 40;
9           /**
10           * 交点是否落子
11           */
12          public static int[][] isPlaced = new int[N][N];
```

3.1 游戏环境模块的实现

游戏环境模块 **Environment.java** 类主要实现了棋盘绘制、棋子绘制以及落子三个类方法.

3.1.1 棋盘绘制

用 java.awt.Graphics 类中的 drawLine 方法就可以绘制水平直线和垂直直线，通过 for 循环以及偏移量，水平和垂直的直线各画 15 条即可形成棋盘，实现代码如下：

```java
/**
 * 画棋盘
 *
 * @param pen
 */
public void drawChessBoard(Graphics pen) {
    // 画棋盘
    pen.setColor(Color.BLACK);

    for (int i = 0; i < N; i++) {
        pen.drawLine(x, y + size * i, x + size * (N - 1), y + size * i);
    }
    for (int j = 0; j < N; j++) {
        pen.drawLine(x + size * j, y, x + size * j, y + size * (N - 1));
    }
}
```

3.1.2 棋子绘制

用 java.awt.Graphics 类中的 fillOval 方法就可以实现棋子的绘制，实现代码如下：

```java
/**
 * 画棋子
 *
 * @param pen
 */
public void drawChessStone(Graphics pen) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            // 计算交点
            int countX = j * size + x;
            int countY = i * size + y;
            if (isPlaced[i][j] == 1) {
                pen.setColor(Color.BLACK);
                pen.fillOval(countX - size / 4, countY - size / 4, size / 2, size / 2);
            }
            if (isPlaced[i][j] == 2) {
                pen.setColor(Color.WHITE);
                pen.fillOval(countX - size / 4, countY - size / 4, size / 2, size / 2);
            }
        }
    }
}
```

3.1.3 落子

通过传入鼠标点击的位置，如果鼠标点击的位置在棋盘交点周围一定范围内，则计算出该交点的位置，并在该交点处进行落子，部分代码如下：

```
1           /**
2            * 下棋
3            *
4            * @param state
5            * @param gobang
6            * @param pen
7            * @param px
8            * @param py
9            */
10          public void playChess(State state, Gobang gobang, Graphics pen, int px, int py) {
11              // 处理坐标
12
13              // 判断坐标是否在交点一定范围内
14              if (rangeX < size / 4 && rangeY < size / 4) {
15                  // 计算交点
16                  int countX = isPlacedX * size + x;
17                  int countY = isPlacedY * size + y;
18                  // 获取画笔
19                  pen = gobang.getChessBoard().getGraphics();
20                  // 落子
21
22              }
23          }
```

3.2 游戏控制模块实现

游戏控制模块 **Control.java** 类主要实现了游戏的开始和结束两个类方法.

3.2.1 游戏开始

将游戏状态改变成开始状态，然后清空初始界面，并进行棋盘绘制，实现代码如下：

```
1           /**
2            * 开始游戏
3            *
4            * @param environment
5            * @param state
6            * @param chessBoard
7            * @param pen
8            */
9           public void startGame(Environment environment, State state, JPanel chessBoard, Graphics
                pen) {
```

```
            // 设置状态为开始
            state.setGameState(1);
            // 清空棋盘
            chessBoard.repaint();
            // 清空棋子
            State.stoneCounts = 0;
            // 轮次为黑
            state.setTurn(1);
            // 画棋盘
            environment.drawChessBoard(pen);
            // 画棋子
            for (int i = 0; i < Gobang.N; i++) {
                for (int j = 0; j < Gobang.N; j++) {
                    Gobang.isPlaced[i][j] = 0;
                }
            }
            environment.drawChessStone(pen);
        }
```

3.2.1 游戏结束

将游戏的状态改变成结束状态，然后清空棋盘，回到初始界面，实现代码如下：

```
        /**
         * 结束游戏
         *
         * @param state
         * @param chessBoard
         */
        public void endGame(State state, JPanel chessBoard) {
            // 设置状态为未开始
            state.setGameState(0);
            // 弹出消息
            JOptionPane.showMessageDialog(null, "你输啦");
            // 清空棋盘
            chessBoard.repaint();
            // 清空棋子
            State.stoneCounts = 0;
        }
```

3.3 游戏状态模块

游戏状态模块 State.java 类主要设置了棋盘状态 (棋盘是否满)、轮次状态 (伦次是黑棋还是白棋) 以及游戏状态 (游戏是否开始) 三个状态变量，以及对应的获取和修改状态变量的方法，代码如下：

```
1           /**
2            * 棋盘状态
3            */
4           public static int stoneCounts = 0;
5
6           /**
7            * 当前伦次是黑还是白
8            */
9           private int turn;
10
11          /**
12           * 棋盘状态是开始还是结束
13           */
14          private int gameState;
```

3.4 AI 模块 (核心)

五子棋看起来有各种各样的走法，而实际上把每一步的走法展开，就是一棵巨大的博弈树. 在这个树中，从根节点为 0 开始，偶数层表示电脑可能的走法，奇数层表示玩家可能的走法. 假设玩家先手，那么第一层就是玩家的所有可能的走法，第二层就是电脑的所有可能走法，以此类推. 我们假设平均每一步有 50 种可能的走法，那么从根节点开始，往下面每一层的节点数量是上一层的 50 被，假设我们进行 4 层思考，也就是玩家和电脑各走两步，那么这棵博弈树的最后一层的节点数为 $50^4 = 625W$ 个. 遍历这颗博弈树，就可以找到电脑最优的落子点，因为博弈树的深度是 4，能够比普通玩家多想 4 步，自然能够占据一定的优势. 但博弈树的节点随着深度的增加是指数级增长的，要在短时间内搜索出最优落子点，需要对传统的博弈树搜索算法进行优化. AI 模块 **AI.java** 类共分为 3 部分别是 ai 落子、评估函数、搜索算法.

3.4.1 ai 落子

ai 落子很简单，只需要告诉 ai 在哪一个交点落子即可，至于如何告诉 ai 在哪一个交点落子，这是搜索算法所要做的，实现代码如下：

```
1           /**
2            * ai画棋子
3            *
4            * @param state
5            * @param pen
6            */
7           public void aiPlayChess(State state, Graphics pen) {
8               int isPlacedX = position.getY();
9               int isPlacedY = position.getX();
10
```

```
11                  // 计算交点
12                  int countX = isPlacedX * Gobang.size + Gobang.x;
13                  int countY = isPlacedY * Gobang.size + Gobang.y;
14
15                  if (state.getTurn() == 2) {
16                      pen.setColor(Color.WHITE);
17                      pen.fillOval(countX - Gobang.size / 4, countY - Gobang.size / 4, Gobang.size /
                            2,
18                              Gobang.size / 2);
19                      Gobang.isPlaced[isPlacedY][isPlacedX] = state.getTurn();
20                      state.setStone(new Stone(isPlacedY, isPlacedX, state.getTurn()));
21                      state.setTurn(state.getTurn() - 1);
22                      State.stoneCounts++;
23                  }
24              }
```

3.4.2 评估函数

评估函数是 ai 模块的核心，我这里所用的评估函数是对整个棋盘进行评估，在进行评估函数编写之前，需要对五子棋的一些棋型进行了解. 常见的棋型为**连五**、**活四**、**冲四**、**活三**、**眠三**、**活二**、**眠二**等，各棋型如下图所示 [1]：

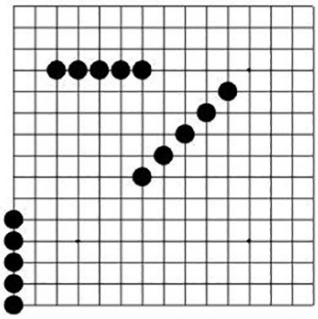


图 1: 连五

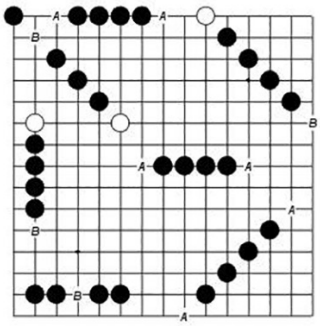


图 2: 活四、冲四

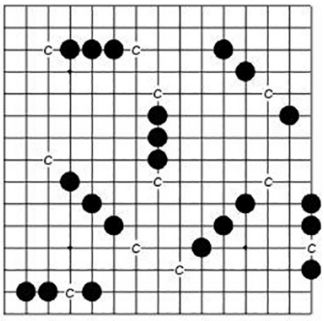


图 3: 活三

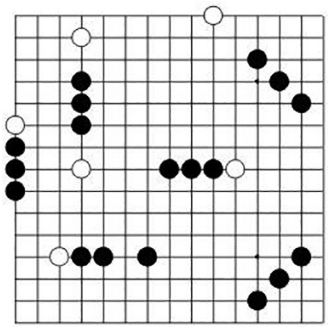


图 4: 眠三

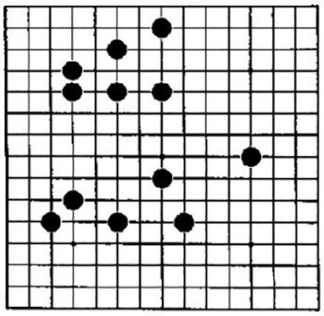


图 5: 活二

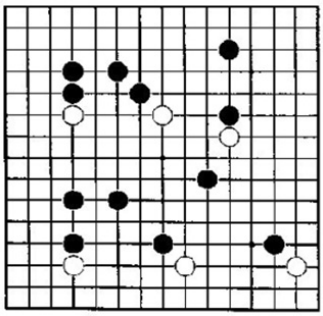


图 6: 眠二

很显然，这些棋型可以划分等级，不妨规定规则为：**连五** > **活四** > **冲四** = **活三** > **眠三** = **活二** > **眠二**，按照这样的规则可以为每种棋型设立一个权重，将棋盘上出现的上述棋型作为**评估函数**的评估值，设立的权重如下：

```
        /**
         * 定义一些棋型权重
         */
        public static final int OTHER = 0;
        public static final int WHITE_FIVE = 1000000;
        public static final int BLACK_FIVE = -10000000;
        public static final int WHITE_LIVE_FOUR = 50000;
        public static final int BLACK_LIVE_FOUR = -100000;
        public static final int WHITE_RUSH_FOUR = 400;
        public static final int BLACK_RUSH_FOUR = -100000;
        public static final int WHITE_LIVE_THREE = 400;
        public static final int BLACK_LIVE_THREE = -8000;
        public static final int WHITE_SLEEP_THREE = 20;
        public static final int BLACK_SLEEP_THREE = -50;
        public static final int WHITE_LIVE_TWO = 20;
        public static final int BLACK_LIVE_TWO = -50;
        public static final int WHITE_SLEEP_TWO = 1;
        public static final int BLACK_SLEEP_TWO = -3;
        public static final int WHITE_ONE = 1;
        public static final int BLACK_ONE = -3;
```

可以发现黑棋各棋型的权重都大于白棋，这是因为玩家先落子，电脑后落子，当电脑落子后，玩家又有了优先权，故相同的棋型，玩家的权重要比电脑的权重大，这是合理的. 各棋型的权重有了，接下来就是对棋型的描述，这里采用的是 6 个字符组成的一个字符串来描述棋型，如 011111、111110 等都可以用来表示**黑连五**，011110、022220 分别表示**黑活四**和**白活四**，于是可以用类似的字符串将所有的棋型表示出来，将这些字符串作为字典的键，将上述棋型的权重作为键对应的值，部分代码如下：

```
        // 白方连5
        mapTable.put("222222", WHITE_FIVE);
        mapTable.put("222220", WHITE_FIVE);
        mapTable.put("022222", WHITE_FIVE);
        mapTable.put("222221", WHITE_FIVE);
        mapTable.put("122222", WHITE_FIVE);
        mapTable.put("322222", WHITE_FIVE);
        mapTable.put("222223", WHITE_FIVE);
        // 黑方连5
        mapTable.put("111111", BLACK_FIVE);
        mapTable.put("111110", BLACK_FIVE);
        mapTable.put("011111", BLACK_FIVE);
        mapTable.put("111112", BLACK_FIVE);
        mapTable.put("211111", BLACK_FIVE);
```

```
15        mapTable.put("111113", BLACK_FIVE);
16        mapTable.put("311111", BLACK_FIVE);
```

通过这张 Hash 表，就可以将棋型与对应的权重联系起来. 接下来的评估函数就是通过搜索整张棋盘中的各种棋型，在找到棋型对应的权重，将这些权重加起来，最终就得到了整张棋盘的评估分数，即电脑的局面分数. 因为各棋型都是直线，故搜索棋型也很简单，只需要水平搜索 15 行、垂直搜索 15 行、左斜搜索、右斜搜索即可将所有的棋型记录下来，部分代码如下：

```
1             /**
2              * 局面评估函数
3              *
4              * @param isPlaced
5              * @return
6              */
7             public int evaluate(int[][] isPlaced) {
8                 // 六元组
9                 SixTuple sixTuple;
10
11                // 棋局分数
12                int score = 0;
13
14                // 加入棋盘边界
15                // 因为被边界夹住的斜线也有可能构成连五，所以要构建一个包含边界的二维数组
16
17                // 判断水平棋型
18
19                // 判断垂直棋型
20
21                // 判断左斜棋型
22
23                // 判断右斜棋型
24
25                return score;
26            }
```

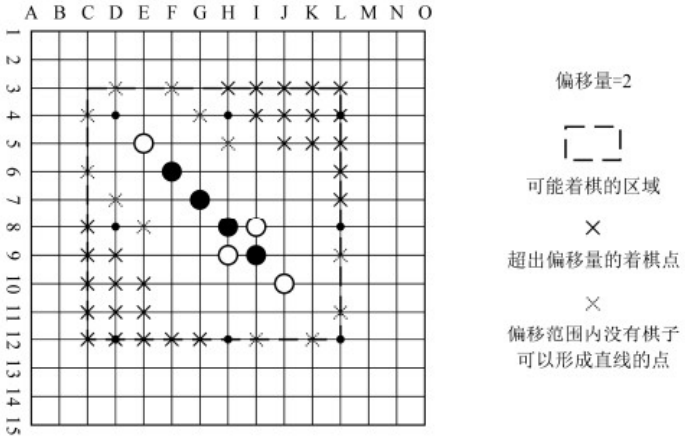
3.4.3 搜索算法

上文已经讲到五子棋博弈其实是一棵巨大的博弈树，当第一个黑棋放下之后，白棋就有 224 种放法，但是不同的落子点影响着局面的分数，要寻找在这剩下的落子点寻找一个最优落子点，对于普通人来说是很难的，因为不可能在很短的时间内模拟 224 种落子方法，但是电脑可以. 但是仅仅是走一步，对于电脑来说显然没有什么意义，电脑需要看得更远，它应该继续想到，当它走完这一步之后，玩家会走哪一步用来进攻或者防守，而它又将走哪一步用来进攻或者防守，想得越深，电脑就可能越聪明，因为它能

够看的更远. 而在电脑思考的过程中，就形成了一棵博弈树，如果电脑思考了 4 步，那么这棵博弈树的深度就是 4. 接下来就是如何选择博弈树的分支的问题，因为电脑总是会选择对自己有利的落子方案，而玩家总是会选择对电脑不利的落子方案，因为上述评估函数是对电脑局面的评估，故电脑总会选择评估函数值最大的点进行落子，而玩家恰恰相反. 因此，电脑所在的层数可以称为 Max 层，玩家所在的层数称为 Min 层，于是这棵博弈树也可以称为**极大极小树**.

而博弈树每增加一层，增加的节点都是指数级的，所以必须要进行优化，于是就有了 $\boldsymbol{\alpha-\beta}$ **剪枝算法** [2]，剪去极大极小树中不必要的分支，从而减少计算量. $\alpha-\beta$ 剪枝算法中每一个节点对应有一个 α 和一个 β，α 表示目前该节点的最好下界，β 表示目前该节点的最好上界. 在最开始时，α 为负无穷，β 为正无穷。然后进行搜索，Max 层节点每搜索它的一个子节点，就要更新自己的 α（下界），而 Min 层节点每搜索它的一个子节点，就要更新自己的 β（上界）。如果更新之后发现 $\alpha \geq \beta$ 了，说明后面的子节点已经不需要进行搜索了，直接进行剪枝操作.(**$\boldsymbol{\alpha-\beta}$ 剪枝算法**可以参考文献 [2])

除了对博弈树进行剪枝操作来优化搜索速度，还可以进行**局部搜索**，减少搜索的节点，在五子棋博弈过程中，没有哪个人类棋手会对整个棋盘进行计算，其思考过程往往是从中心向四周发散，并且五子棋着棋往往不会超出原有棋子两格的位置，而未经优化的搜索算法会将任何一个空白的节点作为可能的落子点，因此局部搜索可以将搜索节点大大减少，特别是棋型呈现斜向狭长走势，如下图所示 [3]：



如果 $\alpha-\beta$ 剪枝算法越早搜索到较优走法，剪枝就会越早发生. 于是可以对局部搜索得到的节点按照局面评估分数递减**排序**，然后再扩展下一层节点，这样可以使得剪枝操作更早的发生，从而加快搜索速度，上述**局部搜索**和 $\boldsymbol{\alpha-\beta}$ **剪枝算法**的实现代码如下：

```
1           /**
2            * 局部搜索
3            *
4            * @param isPlaced
```

```
5             * @param turn
6             * @return
7             */
8            public ArrayList<Position> seekBestPoints(int[][] isPlaced, int turn) {
9                // 记录要搜索的点(初始值为false)
10               boolean[][] boolBoard = new boolean[Gobang.N][Gobang.N];
11               // 搜索偏移量
12               int offset = 3;
13               // 局部搜索棋子可以放置的点
14               for (int i = 0; i < Gobang.N; i++) {
15                   for (int j = 0; j < Gobang.N; j++) {
16                       if (isPlaced[i][j] != 0) {
17                           for (int k = 1; k <= offset; k++) {
18                               // 水平搜索
19                               if (j + k < Gobang.N && isPlaced[i][j + k] == 0) {
20                                   boolBoard[i][j + k] = true;
21                               }
22                               if (j - k >= 0 && isPlaced[i][j - k] == 0) {
23                                   boolBoard[i][j - k] = true;
24                               }
25                               // 垂直搜索
26                               if (i + k < Gobang.N && isPlaced[i + k][j] == 0) {
27                                   boolBoard[i + k][j] = true;
28                               }
29                               if (i - k >= 0 && isPlaced[i - k][j] == 0) {
30                                   boolBoard[i - k][j] = true;
31                               }
32                               // 左斜搜索
33                               if (i + k < Gobang.N && j + k < Gobang.N && isPlaced[i + k][j + k]
                                     == 0) {
34                                   boolBoard[i + k][j + k] = true;
35                               }
36                               if (i - k >= 0 && j - k >= 0 && isPlaced[i - k][j - k] == 0) {
37                                   boolBoard[i - k][j - k] = true;
38                               }
39                               // 右斜搜索
40                               if (i - k >= 0 && j + k < Gobang.N && isPlaced[i - k][j + k] == 0) {
41                                   boolBoard[i - k][j + k] = true;
42                               }
43                               if (i + k < Gobang.N && j - k >= 0 && isPlaced[i + k][j - k] == 0) {
44                                   boolBoard[i + k][j - k] = true;
45                               }
46                           }
47                       }
48                   }
49               }
50
```

```
51              // 搜寻最佳位置
52              ArrayList<Position> positions = new ArrayList<Position>();
53              for (int i = 0; i < Gobang.N; i++) {
54                  for (int j = 0; j < Gobang.N; j++) {
55                      if (boolBoard[i][j] == true) {
56                          isPlaced[i][j] = turn;
57                          positions.add(new Position(i, j, this.evaluate(isPlaced)));
58                          isPlaced[i][j] = 0;
59                      }
60                  }
61              }
62              // 进行排序
63              Collections.sort(positions);
64
65              return positions;
66          }
67
68          /**
69           * alpha - beta 剪枝
70           *
71           * @param isPlaced
72           * @param turn
73           * @param depth
74           * @param alpha
75           * @param beta
76           * @return
77           */
78          public int maxMinSearch(int[][] isPlaced, int turn, int depth, int alpha, int beta) {
79
80              if (depth == 0) {
81                  ArrayList<Position> positions = this.seekBestPoints(isPlaced, turn);
82                  return positions.get(0).getScore();
83              } else if (depth % 2 == 0) {
84                  // max层
85                  ArrayList<Position> positions = this.seekBestPoints(isPlaced, turn);
86
87                  for (int i = 0; i < positions.size(); i++) {
88                      isPlaced[positions.get(i).getX()][positions.get(i).getY()] = turn;
89                      turn--;
90                      int a = this.maxMinSearch(isPlaced, turn, depth - 1, alpha, beta);
91                      turn++;
92                      isPlaced[positions.get(i).getX()][positions.get(i).getY()] = 0;
93                      // 记录棋子
94                      if (a >= alpha && depth == 4) {
95                          position = new Position(positions.get(i).getX(), positions.get(i).getY()
                                ,
96                                  positions.get(i).getScore());
```

```
97                      }
98                      alpha = Math.max(beta, Math.max(alpha, a));
99                      // 剪枝
100                     if (alpha >= beta) {
101                         break;
102                     }
103                 }
104                 return alpha;
105             } else {
106                 // min层
107                 ArrayList<Position> positions = this.seekBestPoints(isPlaced, turn);
108
109                 for (int i = 0; i < positions.size(); i++) {
110                     isPlaced[positions.get(i).getX()][positions.get(i).getY()] = turn;
111                     turn++;
112                     int b = - this.maxMinSearch(isPlaced, turn, depth - 1, alpha, beta);
113                     turn--;
114                     isPlaced[positions.get(i).getX()][positions.get(i).getY()] = 0;
115                     if (depth == 1) {
116                         beta = Math.min(b, beta);
117                     } else {
118                         beta = Math.min(alpha, Math.min(beta, b));
119                     }
120                     // 剪枝
121                     if (beta <= alpha) {
122                         break;
123                     }
124                 }
125                 return beta;
126             }
127         }
```

3.5 输出判断模块

输出判断模块 **Judge.java** 类主要实现了判断输赢和判断棋盘满两个类方法.

3.5.1 判断输赢

只需要在每次落子后，对该子进行水平、垂直、左斜、右斜进行搜索，如果有任何一个方向出现连五，则该子对应的颜色方获得胜利，否则棋局继续，部分代码如下：

```
1           /**
2            * 检查是否胜利
3            *
4            * @param state
5            * @param chessBoard
```

```
     */
    public void checkWin(State state, JPanel chessBoard) {
        // 获取当前棋子
        Stone stone = state.getStone();

        // 棋子参数
        int x = stone.getX();
        int y = stone.getY();
        int color = stone.getColor();

        // 水平、竖直、左斜、右斜计数器
        int rowCount = 0;
        int columnCount = 0;
        int leftCount = 0;
        int rightCount = 0;

        // 检查水平棋子

        // 检查垂直的棋子

        // 检查左斜的

        // 检查右斜的

        // 当任意一个计数器达到5时结束
        if (rowCount == 5 || columnCount == 5 || leftCount == 5 || rightCount == 5) {
            String message = null;
            if (state.getTurn() == 2 && state.getGameState() == 1) {
                message = "黑方获胜";
            } else if (state.getTurn() == 1 && state.getGameState() == 1) {
                message = "白方获胜";
            }
            JOptionPane.showMessageDialog(null, "游戏结束: " + message);
            // 设置状态为未开始
            state.setGameState(0);
            // 清空棋盘
            chessBoard.repaint();
            // 清空棋子
            State.stoneCounts = 0;
        }
    }
```

3.5.2 判断棋盘满

如果状态参数 $stoneCounts == 225$ 则说明，棋盘满了，实现代码如下：

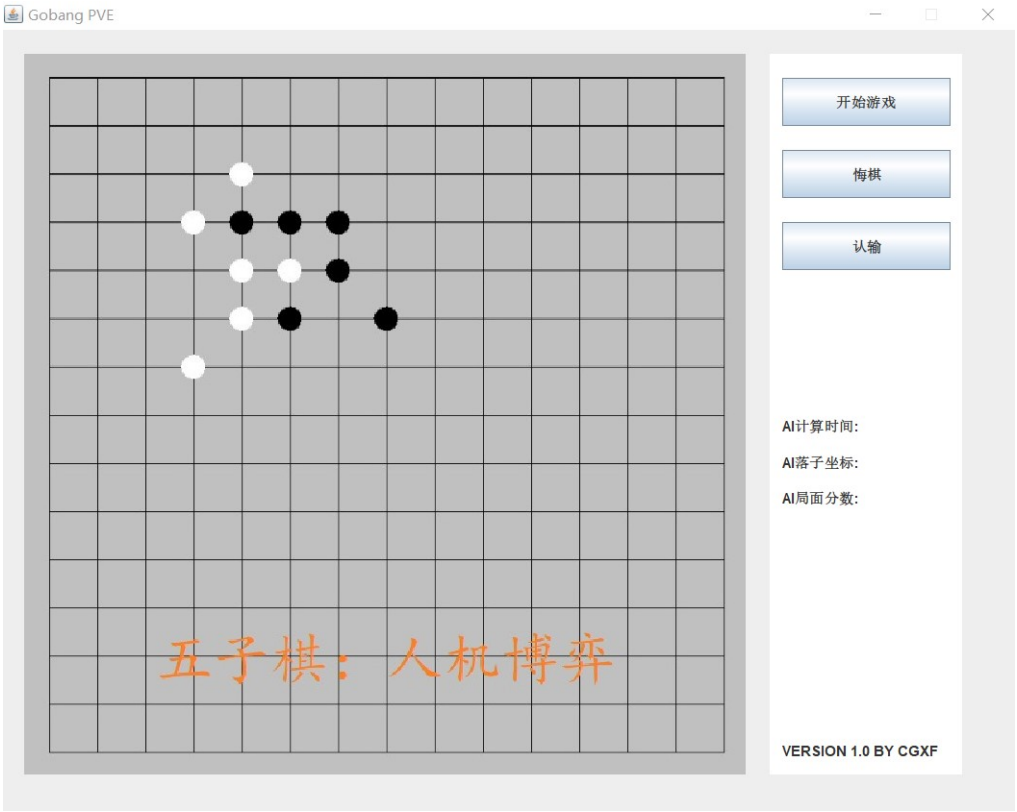
```
    /**
```

```
2             * 判断棋盘满
3             *
4             * @param state
5             * @param chessBoard
6             */
7            public void isFull(State state, JPanel chessBoard) {
8                if (State.stoneCounts == Gobang.N * Gobang.N) {
9                    JOptionPane.showMessageDialog(null, "游戏结束: 平局 -- 棋盘满了");
10                   // 设置状态为未开始
11                   state.setGameState(0);
12                   // 清空棋盘
13                   chessBoard.repaint();
14                   // 清空棋子
15                   State.stoneCounts = 0;
16               }
17           }
```

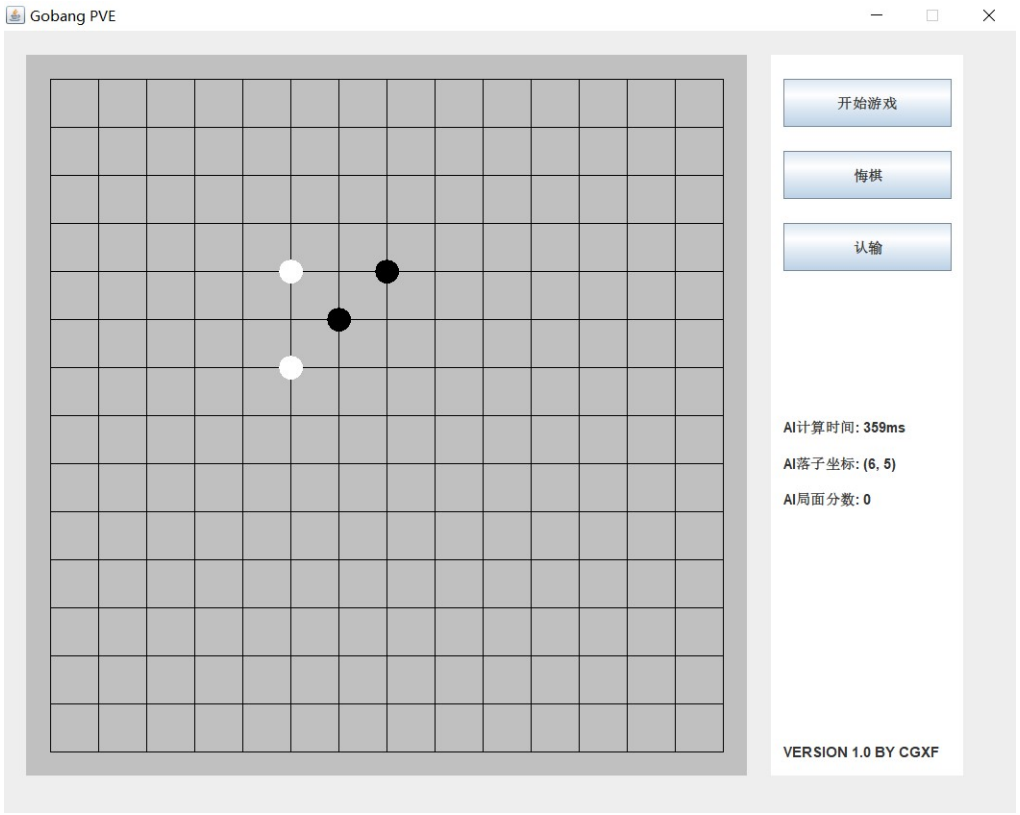
四、人机博弈演示

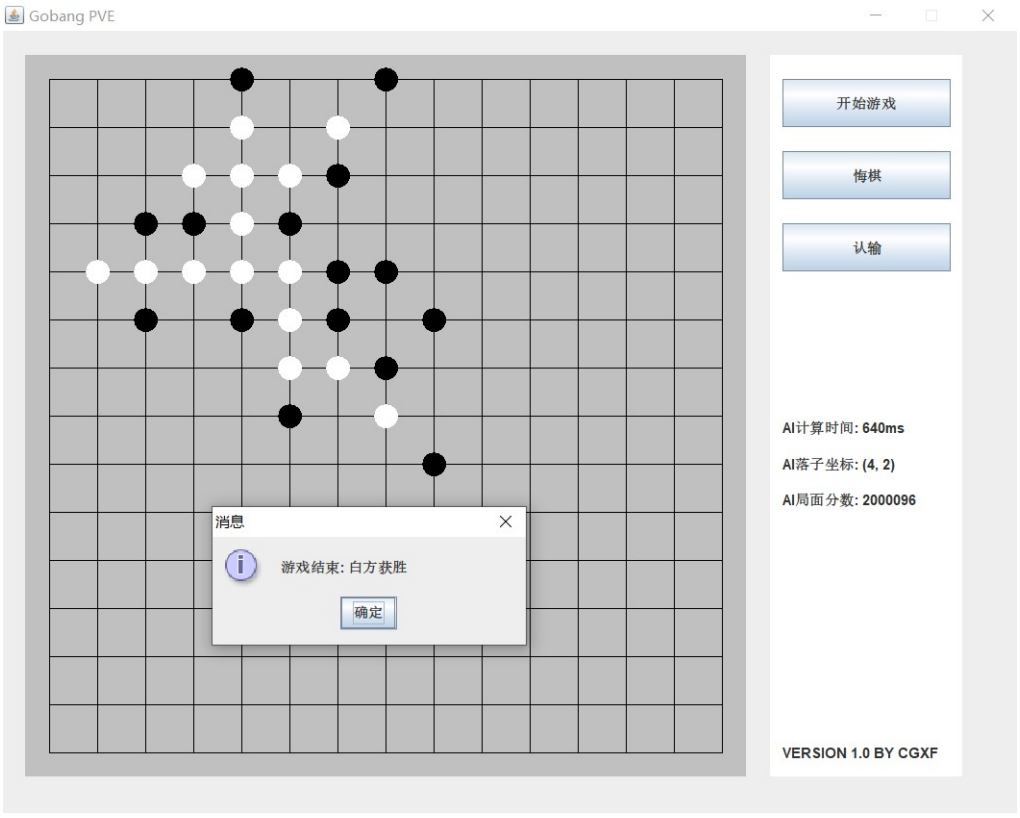
4.1 开始界面

包含一个棋盘界面和操作界面，如下图：



4.2 人机博弈演示


Gobang PVE
开始游戏
悔棋
认输
AI计算时间: 359ms
AI落子坐标: (6, 5)
AI局面分数: 0
VERSION 1.0 BY CGXF


Gobang PVE
开始游戏
悔棋
认输
AI计算时间: 640ms
AI落子坐标: (4, 2)
AI局面分数: 2000096
VERSION 1.0 BY CGXF
消息
游戏结束: 白方获胜
确定

五、设计小结

本次实现的五子棋：人机博弈游戏，仅仅只搜索了 4 层极大极小树，但是最终测试的时候，电脑也能成功打败一些普通玩家. 如果能将搜索深度加到 8 层，那么电脑的智能程度会进一步提升，但所需要支付的代价是更慢的搜索速度，需要对搜索算法进一步优化，如**深度迭代算杀**、**zobrist 缓存优化**和**多线程**等更深的技术，或许以后有时间来进行深入的研究.

这个项目大部分时间都用在了 AI 模块的实现，AI 模块首要的就是评估函数，好的评估函数是 AI 智能的一部分，查阅了很多论文，其中大多数论文都提到了构建一个对棋盘局面分数的评估函数，理论虽然讲得通俗易懂，真正实现起来还是很有难度的. 尝试了很多方法之后，最终选择了六字符的字符串来表示棋型，并通过遍历棋盘四个方向，来记录当前棋盘上的棋型，根据映射表加和得到最终局势分.

总的来说，五子棋：人机博弈还是十分有趣的一个项目，还存在很多可以深入挖掘的东西.

注：源代码目录 Gobang；可执行程序目录 gobangFinal；

参考文献

[1] L. Venkateswara Reddy, S. Saravana Kumar, S. Sugumaran, K. Lavanya, Design and development of artificial intelligence (AI) based board game (Gobang) using android, Materials Today: Proceedings, 2021.

[2] Donald E. Knuth, Ronald W. Moore, An analysis of alpha-beta pruning, Artificial Intelligence, Volume 6, Issue 4, 1975.

[3] 郑健磊, 匡芳君. 基于极小极大值搜索和 Alpha Beta 剪枝算法的五子棋智能博弈算法研究与实现 [J]. 温州大学学报 (自然科学版),2019,40(03):53-62.

[4] 张明亮, 吴俊, 李凡长. 五子棋机器博弈系统评估函数的设计 [J]. 计算机应用,2012,32(07):1969-1972+1990.