《人工智能课程设计》
实验二：
五子棋人机博弈（c#）

同济大学电子与信息工程学院

班级：计算机科学与技术2班

姓名：夏文勇

学号：1851506

指导老师：王俊丽

2021 年 5 月 14日

五子棋人机博弈

摘 要

五子棋的人机博弈是 two-agent 对抗搜索算法的一个很具有标志性的应用，类似于其他的各种棋类问题，五子棋问题的求解最终都趋向于 Minimax 算法（极小化极大算法）。本次实验即利用这一算法给出了五子棋棋局形势的评价函数，通过该评价函数对当前棋局进行层次搜索，最终确定落子位置。

为了节约搜索时长，本次实验的搜索树采用了α-β剪枝算法，剪去了不会改变最终决策的分支，避免了大量无谓的计算。同时，本项目小组还根据五子棋的特点对普适的α-β剪枝算法进行了优化，例如不对周围八个位置没有棋的格点进行搜索等。

而在用户 UI 方面，本次实验采用了 c#来实现，通过使用 c#提供的一系列图形界面封装函数，可以实现鼠标点击位置的检测，从而实现简单的 UI 交互。

关键词：五子棋博弈，Minimax算法，α-β剪枝算法，c#

目录

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.实验内容及要求

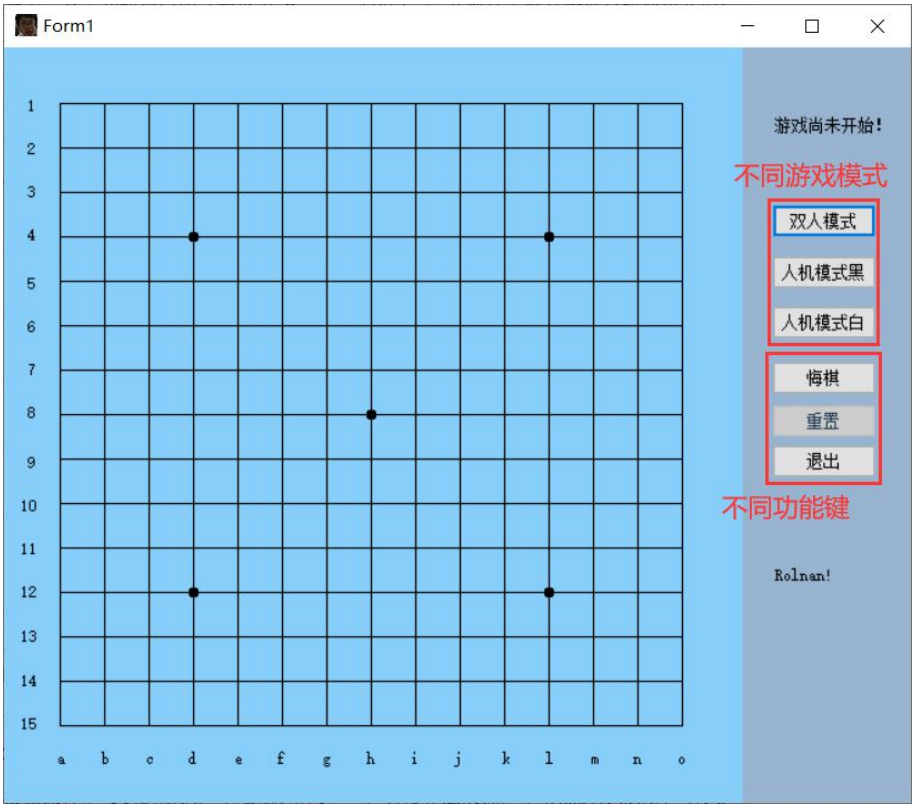
1.以五子棋人机博弈问题为例，实现 α - β 剪枝算法的求解程序（编程语言不限），要求设计适合五子棋博弈的评估函数。

2.要求初始界面显示 15*15 的空白棋盘，有双人模式、人机对战模式，界面置有重新开始、悔棋等操作。

3.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。

4.撰写实验报告，提交源代码（进行注释）、实验报告、汇报 PPT。

1.3.实验效果呈现



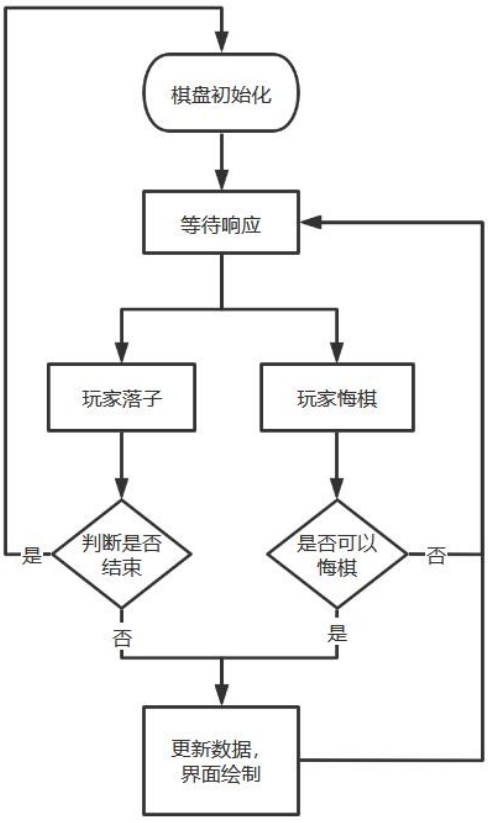
2. 实验方案设计

2.1.总体设计思路与总体架构

整个程序总体上分为基本功能与人工智能落子两部分，基本功能部分主要包括棋盘的绘制、落子、悔棋等功能；人工智能落子部分主要包括棋局评估函数、α - β 剪枝算法、Zobrist 哈希、置换表等。

2.1.1 基本功能架构

1. 使用一个二维数组存储当前棋局，0/1/2 分别表示空、黑、白；另外使用一个一维数组存储每一步的落子情况，用于悔棋时的回退。
2. 每当玩家落子时，读入玩家落子位置，与当前棋局数组比较，判断是否能落子，能则更新棋盘与玩家状态，否则不响应。
3. 玩家落子后，调用监测函数，从落点向八个方向查询，判断当前玩家一方是否获胜，若没有获胜，再进行后续的检索步骤。
4. 使用悔棋按键，是棋盘回到上一步的状态，同时消去最新一步的落子。
5. 使用重置按键，棋盘重置，棋盘数组清空，等待落子。



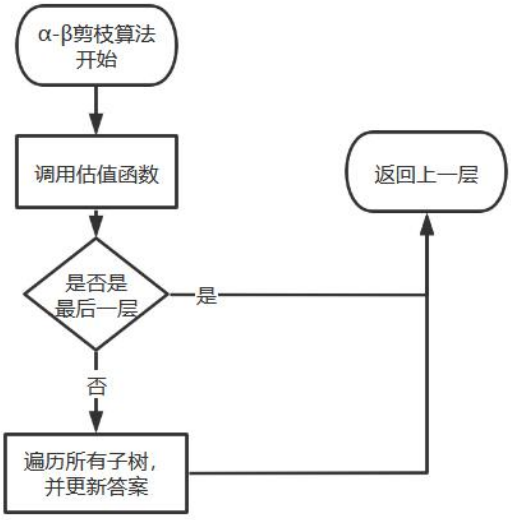
2.1.2 人工智能落子架构

1. 采用 α - β 剪枝算法开始遍历构建当前棋局的搜索博弈树，根据落子点周围的情况与上一步落子的位置安排博弈树的检索遍历顺序与范围，尽可能小地压缩检索时间。

2. 每一层搜索树都需要调用评估函数，首先根据每个点与周围 10*10 方格内所有点的关系，找出活四、活三等的棋子状态，并根据这些状态对每个点进行状态评估。

3. 搜索树每个点的权值都会作为参考，深度更小的点相对价值更大（近大远小）。

4. 选取最优落点进行落子。



2.2.核心算法及基本原理

1. Zobrist 哈希：

Zobrist 哈希通过一种特殊的置换表,也就是对棋盘上每一位置的各个可能状态赋予一个编码索引值,来实现在极低冲突率的前提下在一个整型数据上对棋盘进行编码。其编码步骤描述如下:

1) 将棋盘分为最小单位，即单个点，求出每个单位上不同状态数，在五子棋中，有空、黑、白三种状态。

2) 为每个单位上的每种状态生成一个一定范围内（如 64 位整数）随机数。

3) 对于特定的棋局，将每个单位上的状态对应的随机数作异或运算，所得即为哈希值。

用 Zobrist 哈希为棋局状态编码至少具备两个优点:

1) 当随机数的范围足够大时,不同的棋局产生哈希冲突的概率非常小，在实际应用中通常可以忽略。

2) 在棋局进行过程中，不必每次重新开始计算棋局的哈希值，只需计算棋局状态发生改变的部分。

2. 置换表：

搜索树可以用图来表示，而置换结点可以引向以前搜索过的子树上。置换表可以用来检测这种情况，从而避免重复劳动。

这个散列数组是以“Zobrist 键值”为指标的。你求得局面的键值，除以散列表的项数得到余数，这个散列项就代表该局面。由于很多局面都有可能跟散列表中同一项作用，因此散列项需要包含一个校验值，它可以用来确认该项就是你要找的。通常校验值是一个 64 位的数，也就是上面那个例子的第一个域。

你从搜索中得到结果后，要保存到散列表中。如果你打算用散列表来避免重复工作，那么重要的是记住搜索有多深。如果你在一个结点上搜索了 3 层，后来又打算做 10 层搜索，你就不能认为散列项里的信息是准确的。因此子树的搜索深度也要记录。

在 α - β 搜索中，你很少能得到搜索结点的准确值。α 和 β 的存在有助你裁剪掉没有用的子树，但是用 α - β 有个小的缺点，你通常不会知道一个结点到底有多坏或者有多好，你只是知道它足够坏或足够好，从而不需要浪费更多的时间。

这就引发了一个问题，散列项里到底要保存什么值，并且当你要获取它时怎样来做。答案是储存一个值，另加一个标志来说明这个值是什么含义，在我的程序中，我存储了各个状态在每一层的值，这就可以避免这个问题。

3. Minimax 算法

本实验的核心即为 Minimax 算法（极大极小化算法），该算法通过构建博弈树的方式，旨在找出失败的最大可能性中损失最小的算法，该算法常用于多种零和博弈问题的求解，例如本实验中的五子棋人机博弈问题。

该算法的思路是从当前状态开始，根据接下来可能出现的情况构建子节点，再依照后续的情况对子节点绘制子节点，直至博弈结束或是到达提前预设的深度。然后调用评价函数，对所有出现的状态进行评分估值，开始模拟双方的博弈过程。

由于该博弈问题是零和博弈，所以双方的目的都是要让自己的最终得分最高，对方的得分最低，又由于双方适用于同一套规则，所以这里可以采用正负值的方式来统一双方的评价函数。一方希望最终状态评价函数值尽可能的高，一方希望其尽可能的低，双方轮流决定当前状态该指向哪一个子状态。

所以，Minimax 算法不找理论最优解，因为理论最优解往往依赖于对手是否足够愚蠢，Minimax 中我方完全掌握主动，如果对方每一步决策都是完美的，则我方可以达到预计的最小损失格局，如果对方没有走出完美决策，则我方可能达到比预计的最悲观情况更好的结局。总之我方就是要在最坏情况中选择最好的。

4. α-β剪枝算法

对于单纯的 Minimax 算法来说，其时间复杂度和空间复杂度都是惊人的，在具有 b 个可选策略，深度为 m 的博弈树中，该算法的时间复杂度为 O(bm)量级，呈现出来指数式增长的态势——而为了应对这一问题，可以对不影响策略选择结果的结点进行“剪枝”操作，即

采用α-β剪枝算法对 Minimax 算法进行优化。

α-β剪枝算法的基本思想是，边生成博弈树边计算评估各节点的倒推值，并且根据评估出的倒推值范围，及时停止扩展那些已无必要再扩展的子节点，即相当于减去了博弈树上的一些分支，提高了搜索效率。其具体的操作流程为：

（1）对于一个与节点 MIN，若能估计出其倒推值的上确界β，并且这个β值不大于 MIN 的父节点的估计倒推值下确界α，即α≥β，则就不必再扩展该 MIN 节点的其余子节点了，因为这些子节点的估值对 MIN 父节点的倒推值已无任何影响了，这一过程称为α剪枝。

（2）对于一个或节点 MAX，若能估计出其倒推值的下确界α，并且这个α值不小于 MAX 的父节点的估计倒推值上确界β，即α≥β，则就不必再扩展该 MAX 节点的其余子节点了，因为这些子节点的估值对 MAX 父节点的倒推值已无任何影响了，这一过程称为β剪枝。

2.3.模块设计

1. 棋盘与棋子绘制

在建立的 panel 部件上，使用 c#封装的 DrawLine 函数与 FillEllipse 函数，分别用于棋盘线的绘制以及棋子的绘制，同时还需要考虑最新一步棋的终点标记功能。

2. 落子功能

首先需要使用系统函数捕获当前鼠标位置，同时捕获鼠标单击事件，在鼠标单击事件发生时，获取鼠标在窗口中的点击位置，如果是在控件 panel 中（棋盘），则继续判断当前位置是否已经有棋子，有则不响应，否则落子成功。

3. 胜利判断

由最新的棋子开始搜索，向上、下、左、右、左上、左下、右上、右下八个方向各搜索四格，判断有多少连续同色棋子，如果任意一对相反方向连续同色棋子数大于等于 5，说明当前落子玩家胜利，游戏结束。

4. 评估函数

该模块根据当前棋局新增的棋子，向八个方向查找连续同色棋子或空点，然后根据四个方向判断有多少活四、活三等的状态，最后根据构建的评估函数来评估当前局势，给出评估值。

5. 博弈树构建

基于评估函数给出的评估值，采用α-β剪枝算法，根据本实验中自主添加的优化方式确定搜索顺序与范围，逐层构建起一棵深度为 3 的博弈树，并由该博弈树确定 AI 的落子位置。

2.4.其他创新内容或优化算法

1. 评估函数的构建

对于一个零和博弈问题过程来说，Minimax 算法与α-β剪枝策略是大部分博弈树在构建过程中都会采用的策略，而决定一个对抗 AI 策略选择好坏的根本核心，就在于其评估函数的选择上。在本次试验中，我们根据五子棋本身的特点，结合网络博客的分析，给出其评估函数为：

分别从横竖和两条对角线共计四个方向对当前棋局进行遍历分析，分析方式是向八个方向依次扩展读取至多 5 个相邻的格点位置，直到遇到不同色的棋子，然后记录有多少连续同色棋子，或者包含多少空白，每当出现如下的范式（0 表示空格点，1 表示同色棋子，2 表示异色棋子），就给其加上对应的评估值权重：

1）连五：顾名思义，五颗同色棋子连在一起，型为（1,1,1,1,1）；

2）活四：有两个连五点（即有两个点可以形成五），活四出现的时候，如果对方单纯过来防守的话，是已经无法阻止自己连五了，型为（0,1,1,1,1,0）；

3）冲四：有一个连五点（即有一个点可以形成五），相对比活四来说，冲四的威胁性就小了很多，因为这个时候，对方只要跟着防守在那个唯一的连五点上，冲四就没法形成连五，型为（2,1,1,1,1,0）（1,1,0,1,1）（1,0,1,1,1）；

4）活三：可以形成活四的三，活三棋型是我们进攻中最常见的一种，因为活三之后，如果对方不以理会，将可以下一手将活三变成活四，而我们知道活四是已经无法单纯防守住了。所以，当我们面对活三的时候，需要非常谨慎对待。在自己没有更好的进攻手段的情况下，需要对其进行防守，以防止其形成可怕的活四棋型，型为（0,1,1,1,0）（0,1,0,1,1,0）；

5）眠三：只能够形成冲四的三。图中白点代表冲四点。眠三的棋型与活三的棋型相比，危险系数下降不少，因为眠三棋型即使不去防守，下一手它也只能形成冲四，而对于单纯的冲四棋型，我们知道，是可以防守住的，型为（0,0,1,1,1,2）（0,1,0,1,1,2）（0,1,1,0,1,2）（1,0,0,1,1）（1,0,1,0,1）（2,0,1,1,1,0,2）；

6）活二：能够形成活三的二，活二棋型看起来似乎很无害，因为他下一手棋才能形成活三，等形成活三，我们再防守也不迟。但其实活二棋型是非常重要的，尤其是在开局阶段，我们形成较多的活二棋型的话，当我们将活二变成活三时，才能够令自己的活三绵绵不绝微风里，让对手防不胜防，型为（0,0,1,1,0,0）（0,1,0,1,0）（1,0,0,1）；

7）眠二：能够形成眠三的二，型为（0,0,0,1,1,2）（0,0,1,0,1,2）。

2. 搜索的优化

1）搜索规模限制：对于搜索算法来说，节点数量与搜索深度是限制的重要因素，而对于五子棋来说，对于一个局面来说，可能的落子位置其实并不多。我们可以维护一个当前棋子最小矩形，即一个能覆盖所有棋子的最小矩形，我们的落子只需要在这个最小矩阵想外再扩展 3-4 格即可。

2）搜索顺序优化：对于 α - β 剪枝算法来说，好的搜索顺序是很重要的，因为越早搜索

到最优解，就能在之后的搜索中进行更多的剪枝操作，因此，我们会优先对可能的点做排序，先对当前评估值更高的局面做搜索。

3. 实验过程

3.1.环境说明：操作系统、开发语言、开发环境及具体版本、核心使用库等

操作系统：windows10 版本 20H2

开发语言：c#

开发环境： Visual Studio2019

核心使用库：

C#：

System.Windows.Forms

System.Drawing

System.IO

System.Collections.Generic

3.2.源代码文件清单，主要函数清单

3.2.1 class Form1

主要用于数据的初始化，以及一些核心数据的存储。

1. 数据说明

```
        private bool start = false;     //现在的棋局状态，是否开始
        private bool Ai = false;        //是否人机模式
        private bool AiUse = false;     //电脑在下棋，不能下
        private bool who = true;        //谁下棋0 黑，1 白

        private int[,] chessNum = new int[Global.size, Global.size];    //每一步是第几步下的
        private Dictionary<int, bool> mp = new Dictionary<int, bool>();  //Zobrist 随机值判重
        private int prex = -1, prey = -1;         //上一步下的棋的位置
        private int[,] point = new int[230, 2];//下棋的记录
        private int totSteps = 0;
```

2. Zobrist 随机值，每个落点 3 个状态，随机赋值。

```
        private void Zobrist()
```

3. 退出按钮

```
private void button3_Click(object sender, EventArgs e)
```

4. 双人模式开始

```
private void button1_Click(object sender, EventArgs e)
```

5. 人机模式，玩家执白棋

```
private void buttonWhite_Click(object sender, EventArgs e)
```

6. 人机模式，玩家执黑棋

```
private void buttonBlack_Click(object sender, EventArgs e)
```

7. 初始化函数

```
private void init()
```

8. 棋盘初始化函数

```
private void panel2_Paint(object sender, PaintEventArgs e)
```

9. 鼠标落点捕获函数

```
private void moveNow(object sender, ref bool who, int x1, int y1)
```

10. 悔棋函数

```
private void button4_Click(object sender, EventArgs e)
```

3.2.2 class Global

用于储存一些全局变量

```
public const int CHESS_NONE = 0;
public const int CHESS_BLACK = 1;
public const int CHESS_WHITE = 2;
public const int size = 15;    //棋盘大小
public static int[,,] ZobristMap = new int[Global.size, Global.size, 3];
//Zobrist 随机值
public static int ZovristHash { get; set; }        //Zobrist 哈希值
public static Dictionary<Point, int> ReplacementTable = new Dictionary<
Point, int>();  //Zobrist 随机值判重
public static int[,] chessmap = new int[size, size];    //棋盘状态，0 空，
1 黑，2 白
```

3.2.3 class chessboard

用于对棋盘的一些操作

1. 数据说明

```
        private static int[,] dir = new int[8, 2] { { -1, 0 }, { -1, 1 }, { 0,
1 }, { 1, 1 }, { 1, 0 }, { 1, -1 }, { 0, -1 }, { -1, -1 } };
        //上，右上，右，右下，下，左下，左，左上
        private static int[] cntCon = new int[8];
        public static int nowLeft, nowRight, nowUp, nowDown;//现在已经下的子的边
界
        public const int pointGap = 32, Margin = 40;   //画布大小,行距，边界距离
        public const int chessRadius = 24;
```

2. 判断棋盘是否已满

```
        public static bool mapFull()
```

3. 判断是否有玩家获胜

```
        public static bool isVictory(int x, int y)
```

4. 计算第 i 行/列的坐标值

```
        public static int count_position(int i)
```

5. 棋盘绘制

```
        public static void draw_chessboard(Graphics graph)
```

6. 棋子绘制

```
        public static void draw_chess(Panel p, bool who, int x1, int y1, int nu
m, bool red = false)
```

3.2.4 class AI_solve

主要包括局面评估函数以及 α - β 剪枝算法，用于确定最优解

1. 数据说明

```
        private static int[,] dir = new int[8, 2] { { -1, 0 }, { -1, 1 }, { 0,
1 }, { 1, 1 }, { 1, 0 }, { 1, -1 }, { 0, -1 }, { -1, -1 } };
        //左，左下，下，右下，右，右上，上，左上
        private static int[] cntCon = new int[8];      //连续的相同棋子或空白
        private static int[,] cntNum = new int[8, 5];  //向一个方向延伸，遇到不同
颜色棋子或指定长度前，相同棋子数量
        private static int[] cntChong = new int[6];
        private static int[] cntLive = new int[6];
        private static int cntDie;
```

```
        public const int changeRange = 4;       //每个节点能够影响的范围
        private const int maxDeep = 4;           //搜索最大深度
        private const int maxSearchPoint = 10;  //每层搜索节点数量

        public static int[,,] Vmap = new int[2, Global.size, Global.size];
    //记录每个节点对于两种颜色的权值，1 黑 0 白
        private static bool[,] maxVmapVis = new bool[Global.size, Global.size];
      //Max 剪枝时取前十个最值
        private static int xnext, ynext;
```

2. 初始化函数

```
        public static void init()
```

3. 判断指定位置周围 10*10 内连续或空白点

```
        private static void countConNum(bool who, int i, int j)
```

4. 判断“冲”的状态，即两端点为空，中间为若干空白与同色棋子

```
        private static int statusChong(int fx)
```

5. 判断“活”的状态，即两端点为空，中间为若干空白与同色棋子

```
        private static int statusLive(int fx)
```

6. 根据“冲”与“活”的状态为具体点计算评估值

```
        public static int judge(bool who, int i, int j)
```

7. 为可能落子范围内所有点计算评估值

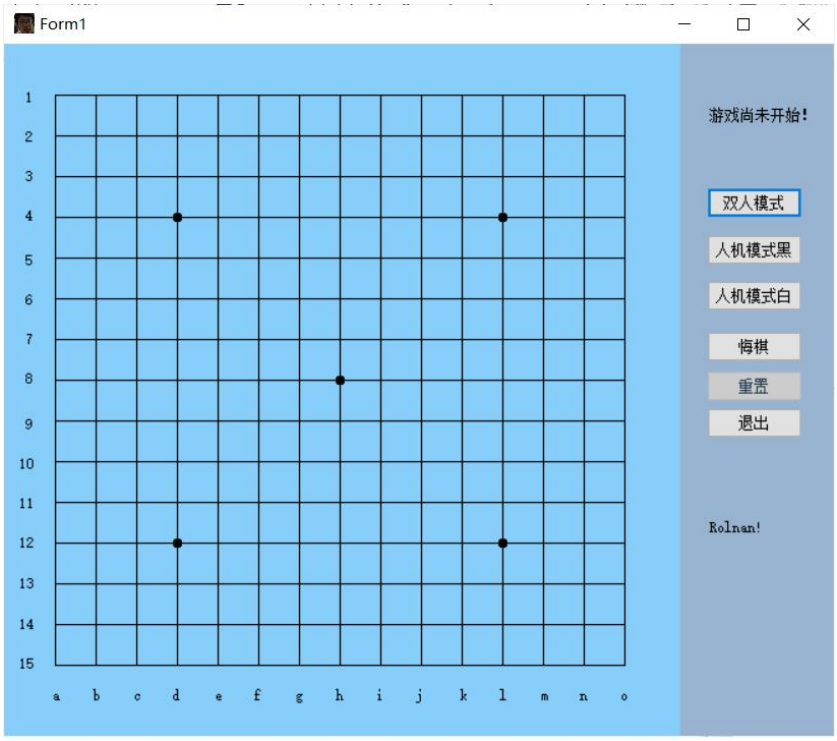
```
        public static void calV(bool who)
```

8. α - β 剪枝

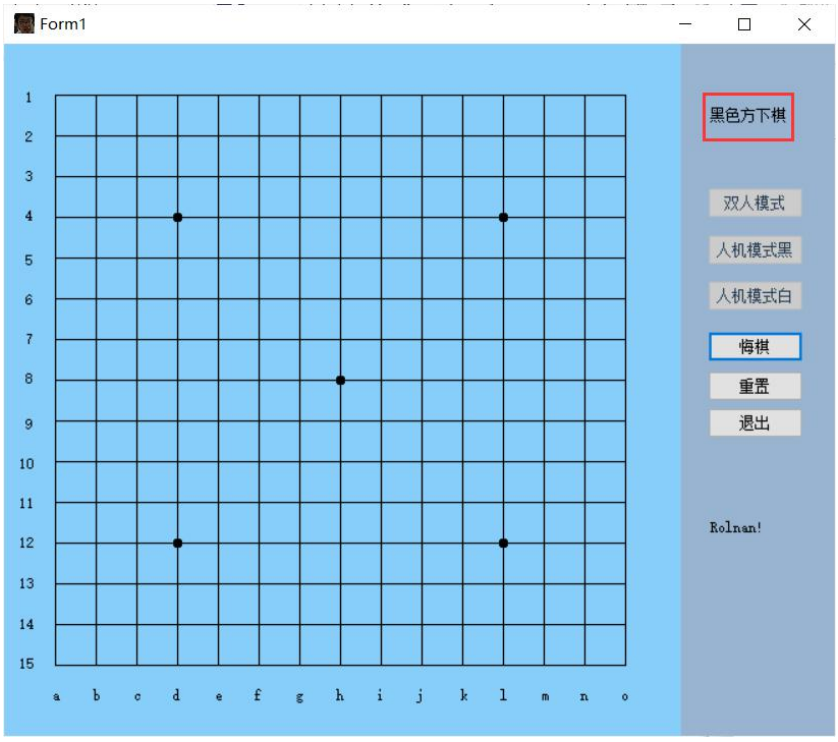
```
        private static int alpha_beta(int deep, bool who, int x, int y, int alp
ha, int beta)
```

3.3.实验结果展示

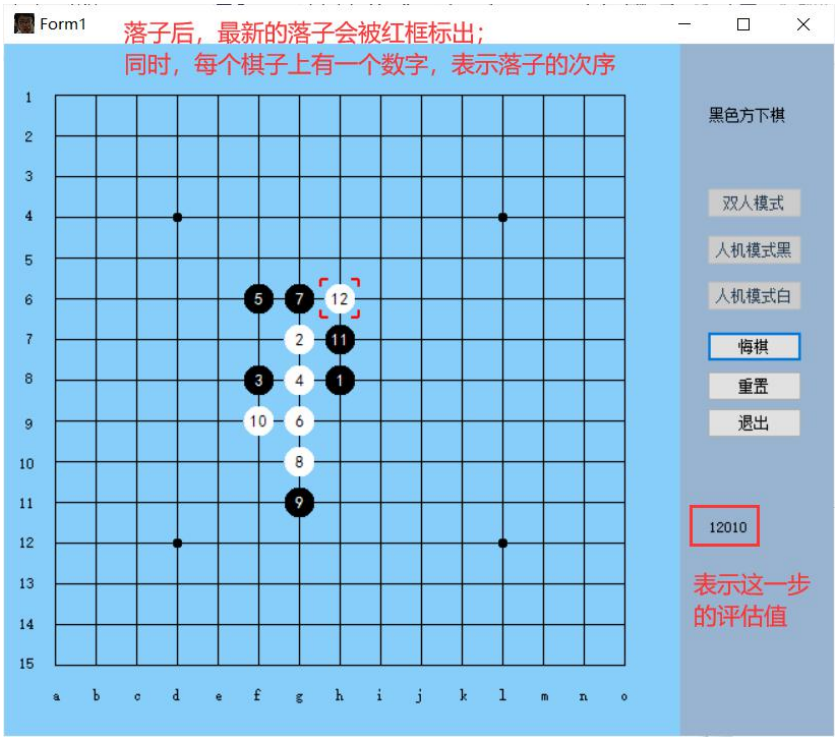
1、 初始状态，可以选择游戏模式



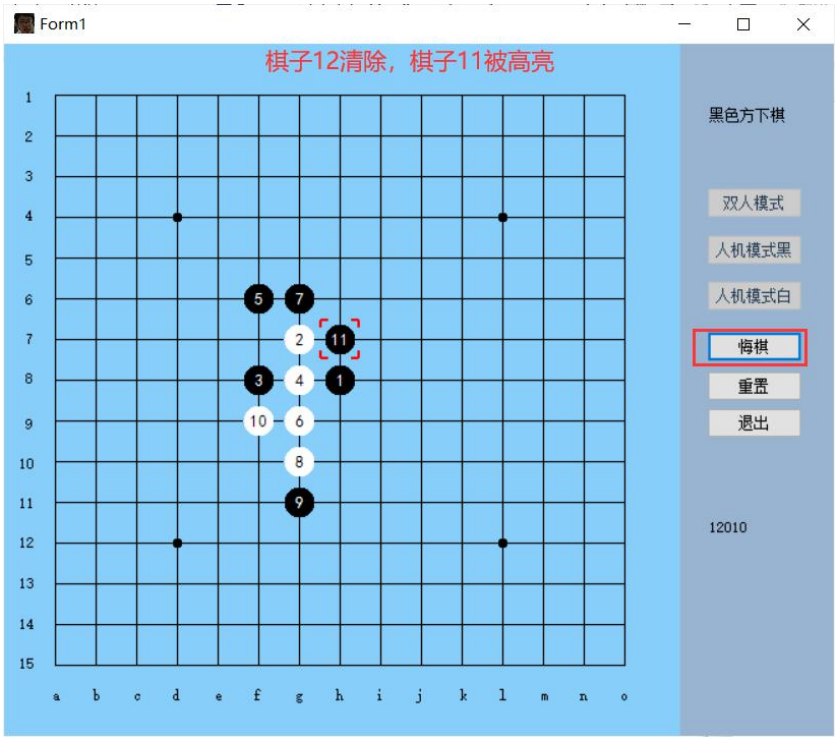
2、 选择模式后提示当前落子方



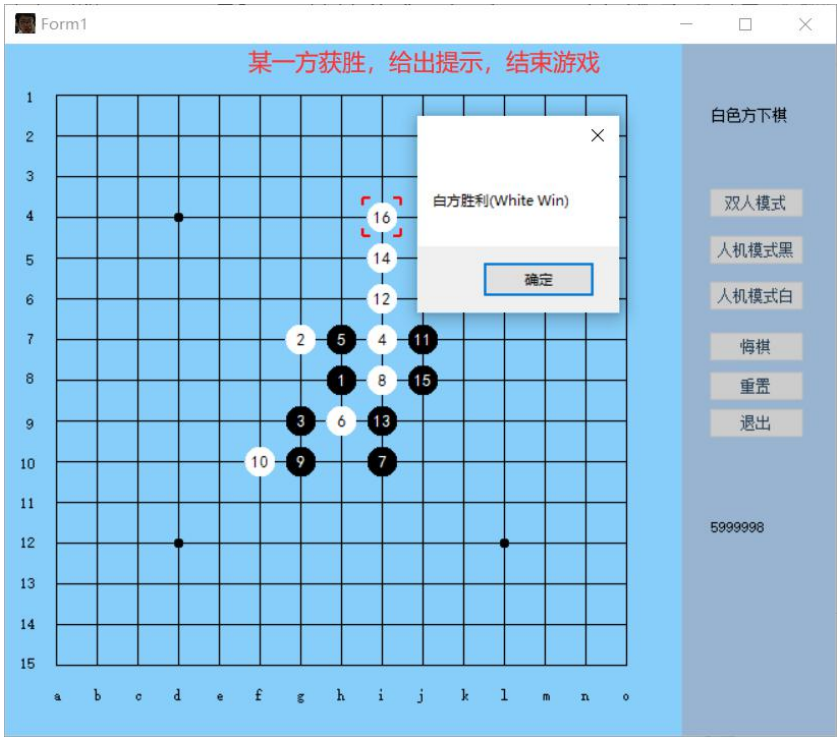
3、 正常游戏进行



4、 悔棋功能演示



5、 结束功能演示



4. 总结

4.1. 实验中存在的问题及解决方案

1、 进攻与防守兼顾问题

我们的评估函数是针对每一步，每一种颜色都会有一个评估值，那么该点的真实评估值应该需要兼顾两种颜色的评估值。

而如何兼顾的，最简单的想法就是两者之和，但是这样又会存在一定的问题，比如双方都有一个冲四，此时轮到自己下棋，那么理论上自己一定会选择获胜的方法，但是此时的结果却不是这样，原因就是防守对方的评估值可能更大。

因此我们需要加大一些进攻的评估值，以此来完善评估函数，在这里我尝试了许多参数，最终选取进攻参数乘以二加上防守参数，来计算最终的评估函数，取得了较好的效果。

4.2.心得体会

本次大作业，让我自学了 Zobrist 哈希、置换表、静态搜索、主要变例搜索等等一系列博弈游戏中的高级策略，对博弈游戏有了进一步的理解。

4.3.后续改进方向

1、估值函数优化

由于估值函数的构造具有固定性，难以适应所有情况（开局、中期、后期），因此可以考虑在未来增加多个估值函数来适应多种棋局的情况。

此外，查阅了一些比较先进的五子棋对战 AI 的实现逻辑，发现其往往是搭载了一定的机器学习内容，通过对大量棋谱的分析，根据前人的棋谱，能够给出当前棋局下适宜的几个落子位置，然后再基于这几个给出的位置进行优先搜索，从而能够减去更多的子节点分支。而随着剪枝效率的上升，也随之可以进一步加大搜索深度，提高搜索准确度。

5. 参考文献

[1] 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.

[2] 董慧颖,王杨.多种搜索算法的五子棋博弈算法研究[J].沈阳理工大学学报,2017,36(02):39-43+83.

[3] 周洋,邓莉,谢煜.一种五子棋博弈算法的分析[J].现代计算机(专业版),2017(10):8-10.

6. 成员分工与自评

整个项目由我个人独立完成，均为独立完成。其中，在评估函数方面参考了网络文献，其他部分均为完全自主独立完成。

总体的任务有：

1、界面设计

2、图形绘制部分

3、评估函数的构建

4、α-β剪枝部分

5、实验报告的撰写