# 第一次实验：MiniAlphaGo

| 序号 | 学号 | 专业班级 | 姓名 | 性别 |
|---|---|---|---|---|
| **22** | **3160103963** | 信息安全 | 罗天翔 | 男 |
| | | | | |
| | | | | |

## 1. Project Introduction

（1） 开发环境：

语言：C，C++

库：easyX

（2） 工作情况：

3.12-3.19

1.学习蒙特卡洛树搜索算法

2.学习黑白棋的基本逻辑

3.真人对战电脑，以学习黑白棋对战技巧

3.20-3.27

1.思考蒙特卡洛树搜索算法的实现，在这一过程中，仔细比较了 Min-Max，α - β 剪枝算法和蒙特卡洛搜索树的异同，确定了使用 UCT 方法实现蒙特卡洛树搜索作为最基本的搜索方法。

2.使用 C 语言实现了黑白棋的对战逻辑，GUI 则暂时使用了命令行界面代替

3.根据自己将要实现的 AI 算法对黑白棋的逻辑进行进一步修改和完善。

3.28-4.4

1.实现了蒙特卡洛树搜索的最基本形式，评价体系则是由最基本的棋子数量的多少来实现。

2.修复蒙特卡洛算法的部分 bug，使其能完整运行并和真人对战，虽然结果是惨败

3.和同一门课的其它同学进行讨论，完善和修改算法的过程和对局势的评价模型。

4.5-4.12

1.优化蒙特卡洛树搜索的效率，使其 1s 内能够搜索 8 层左右，但是随

着层数提高，搜索时间呈指数级增长，又没有想到其它更好的优化方法了。
2.增加了一个评价模型，对单独下到一半的棋盘进行评价，增加了位置因素进行选择，使 AI 更倾向于选择边角的位置。

4.13-4.19
1. 完善了命令行显示界面，可以使其打印出较好看的棋盘
2．将 AI 与高级电脑对战，获胜率 100%，感觉很良好。
3．进一步学习人工智能相关知识

4.20-4.26
1. 思考是否能将参数训练加入 AI 评价模型当中
2. 思考评价体系的学习方法
3. 与同学交流和沟通学习方法
4. AI 第一轮比赛，一轮游，对面的 AI 好像更智能点，与胜利的同学交流，发现其 AI 是无脑占边角，居然还要更强点

4.27-5.4
1. 使用 EasyX 库实现了简单的 GUI，能实现鼠标交互了。但是 EasyX 库的语言是 C++，因此调整过程十分困难。

5.4-5.12
1. 为 AI 实现了一个参数训练的方法，使其对每一步的两个评价占有不同的比例。
2. 与黑白棋冠军交流，发现其评价模型含有下一步能下的步数，觉得这个思考点很特别，同时其 AI 真的十分厉害，受其启发，增加了一个判断步数的参数。
3. 训练 AI

5.13-5.18
1.继续训练 AI
2.编写实验报告

## 2. Technical Details

（1） 算法实现：

1. 数据结构

蒙特卡洛搜索树节点：

```
struct PlayNode
{
    int Playableboard[64];
    int Chessboard[64];
    int PlayableNum;
    float value;
    int frequency;
    int turn;
    int runtime;
    struct PlayNode* Father;
    struct PlayNode* Random_Child;
    struct PlayNode** Child;
};
```

节点信息包含当前节点的棋盘落子情况，棋盘可下子位置和个数，同时还有评价值，遍历次数，棋盘轮到哪一方下以及轮数。对于树结构，由连接父节点的指针用于反向传播，连接孩子节点指针用于延伸结构，随机落点则是 UCT 方法下随机搜索结果的双向链表。

2. 搜索过程

AI 的搜索过程分成以下几步：

a. 从根节点开始搜索评价模型下评价值最大的节点
b. 拓展该节点，如果无法拓展则选择该节点
c. 从该节点出发，随机选择下一步的走法，直到对局结束
d. 对结束对局的棋盘形势进行判断，反向传播到根节点
e. 对选择的节点进行棋盘形式判断，反向传播到根节点，实现双重判断
f. 重复上述过程直到搜索结束

3. 反向传播过程

反向传播的过程如下：

a. 对于随机节点，对棋盘的形势进行判断，通过评价模型生成棋盘的值，反向传播，使遍历的数目加 1，评价值累加，结束后释放节点。

b. 对于选择节点，对棋盘的形势通过评价模型生成评价值，反向传播，遍历数目不加，因为前者已经加过了，评价值累加，结束后不释放节点。

（2） 选择模型设计

这里采用了 UCT 策略中常见的选择模型：

```
left = p -> value / p -> frequency;
right = 2 * p -> Father -> frequency / p -> frequency;
score = left + 2 * c * sqrt(right);
return score;
```

即

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

（3） 评价模型的设计与优化

评价的模型在设计的过程中不断地更改过，不过基本大体上可以被分成三大部分：

第一阶段：仅有对随机节点地棋子数量比较的评价模型

方法为：

计算终局棋子的差决定评价值

```
for (int i = 0; i < 64; ++i)
{
    value += Chessboard[i];
}
return value * turn_flag;
```

这种方法一直被很多同学诟病，原因很简单，因为黑白棋不是一个使用累加型的下棋手段能通常获胜的棋，下棋者有时在棋子数量上虽然有很大的优势，但完全有可能在最后几步被一举反超。因此评价棋子的数量差模型是十分不妥当的。

我在这个模型处思考了很久，最终还是将它留下来当中评价模型

之一，因为棋类最终的胜利仍然是被数量差所决定，越是往后面下，尤其是最后十几步的时候，棋子数量差的模型评价的越好，其胜率应该是越高的。

也就是说，在棋子的位置和棋子数量的选择上有一个临界点，评价模型不能靠单一的元素进行评价，否则容错性就会很低，因此我最终把这个留下来当作评价模型之一，在后面的测试中也取得了不错的效果。

第二阶段：对选择节点的局势进行判断

方法为：

对棋盘上每一个位置赋予权重，使 AI 倾向于选择边角的点

```
const int weight[64] = {
    400,-25,100,100,100,100,-25,400,
    -25,-50,5,5,5,5,-50,-25,
    100,5,1,1,1,1,5,100,
    100,5,1,1,1,1,5,100,
    100,5,1,1,1,1,5,100,
    100,5,1,1,1,1,5,100,
    -25,-50,5,5,5,5,-50,-25,
    400,-25,100,100,100,100,-25,400
};
```

在这里，角点占较高的权重，而角旁边的点则赋予负值，从而防止对手轻易的占到角点，边点的权重也得到提升。

在这个评价体系中，AI 比起数量会倾向于选择边角的点，从而使自己的棋子在比较优秀的位置。从而达到后面的反超甚至反盘的结果

第三阶段：加入对对手可下子的数目进行评价

这一段是我与冠军选手交流后得到的启示，其实我一开始的时候思考过这一方向的内容，但是后面我想了想又放弃了，因为除非对手是无子可下，否则即使有一个子可下也存在翻盘的空间，但是交流之后我发现自己想错了，越少可下的子数相当于越低的复杂度，AI 在这种情况下对局势的把握能力就越好，我后面添加了这种方法进入了训练当中，AI 的压制力有了显著的提高。

（4） 优化与加速

1.在这一部分，首先是对黑白棋的存储，使用了 bitmap 进行管理，缩小了黑白棋的内存占用空间

2.对搜索树的结构做了加速处理，本来是每次需要搜索 64 个点以确定是否能下，后面使用可下棋的阵列来表示可下点，同时在 AI 搜索的时候，对不同的步数有不同的搜索次数和层数。

3.计划中：是将棋盘的形势结果存储在哈希表中，再结构化的存在文件中，从而节省 AI 的搜索时间，同时提升效率。这一点是与冠军选手交流后觉得十分不错，但是仍未来得及实现。

（5） GUI 设计

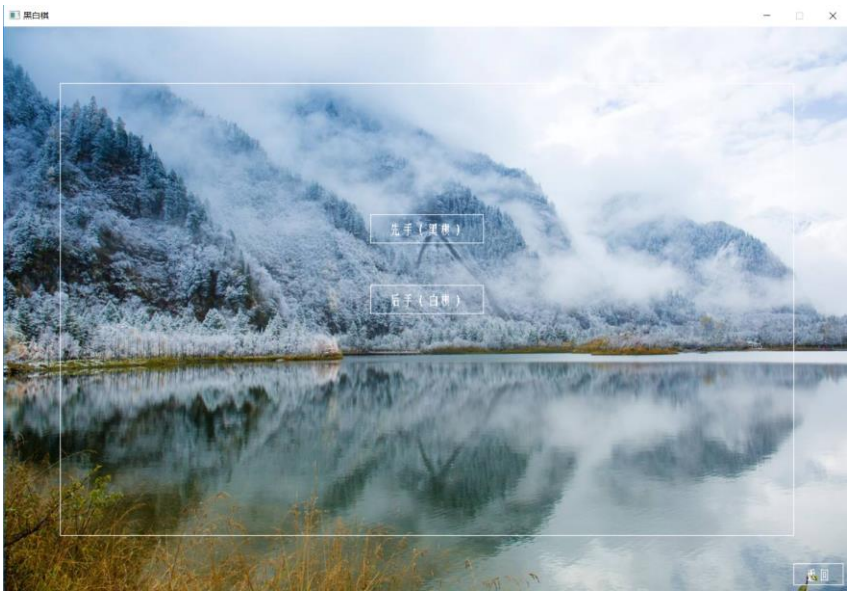
首先必须要说明的点是，我的 AI 程序的开发过程是在 Linux 平台 GCC 环境下开发的，但是在制作 GUI 的过程中，我发现 Linux 平台下没有比较简易明了的 GUI 开发库或者环境，opengl 又太过复杂了。同一时期，我发现 Windows 平台下有 EasyX 轻量级图形库，因此后期又把工程转移到了 Windows 平台下，使用 Visual stdio 2019(因为 EasyX 只支持这个)进行了转接。

因此，在 GUI 环境中，只有人机对战的部分，而缺少了自博弈学习的部分。同时，自博弈可在 Linux 平台下进行使用。

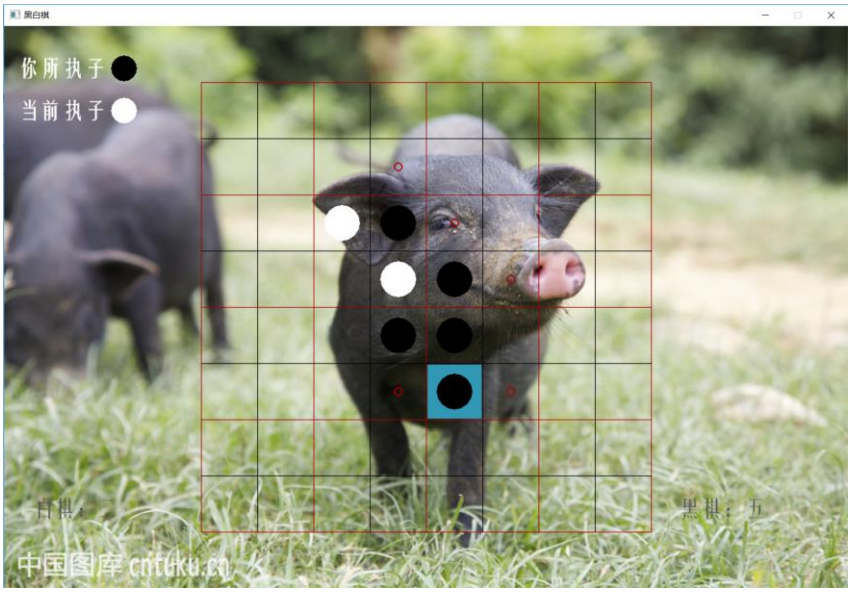
图形界面：

1.开始界面


2.选择先后手



3.开始博弈




# 3. Experiment Results

(1) 自博弈参数设计

在这一部分，我思考良久，决定寻找每一步的下子位置与对手下子数量做一个权重比：一共有 64 个参数，代表 64 步棋每一步的决定参数，参数对评价模型的影响如下：

```
float value = 0;
for (int i = 0; i < 64; ++i)
    {
```

```
            value += Chessboard[i] * weight[i];
        }
value = value * Constant + (float)PlayableNum * 10 * (1 - Constant);
```

其中，即每一步选择位置和可下棋子的数量的比重。

在每一局游戏开始时，AI 将会从文件读入这些参数，用以评价棋盘形势。

(2) 自博弈参数学习方法

我将学习的方法可以分为两个阶段：

1. 首先由真人（我）设计一个我认为比较科学的参数序列，作为 AI 的初始参数序列，然后与随机生成的序列进行博弈。在博弈过程中，连续获胜两场将会把序列替换成较强序列，否则重新生成序列。这一部分训练了一天左右。
2. 将训练很久后得到的较强序列作为模板，轻微震荡部分参数，测试是否有更优的序列。反复这个过程，得到更优序列。这一部分序列了两天左右。

将得到的参数与网络上 AI 进行对战。

(3) 自博弈参数学习结果

参数列表如下：

```
0.076600  0.246200  0.646800  0.350000  0.256300  0.100400  0.520300
0.143900  0.959800  0.164600  0.801900  0.956400  0.242700  0.230800
0.443500  0.572000  0.688800  0.865200  0.092300  0.997800  0.226000
0.490500  0.690500  0.655600  0.478800  0.352400  0.828900  0.598500
0.552700  0.846500  0.110300  0.264600  0.727900  0.392300  0.614600
0.984300  0.492700  0.134900  0.763400  0.087700  0.299600  0.565300
0.044100  0.177500  0.796200  0.122800  0.749500  0.120200  0.623300
0.477000  0.753200  0.849300  0.602800  0.443700  0.504900  0.081600
0.431300  0.969000  0.680100  0.984100  0.450800  0.425600  0.883900
0.178700
```

对于这个结果，我的预期是一个抛物线的类型，但是结果跟想象的不一样，我觉得原因还是多种的，比如训练时间不足，训练的参数不够稳定等等。

(4) 网络对战结果

在设计完的最初版的 AI 可以战胜各大小游戏网站上（4399）的黑白棋

AI，但是黑白棋 AI 网站上的 AI 普遍下不赢。

在自博弈学习后的 AI，在 AI 网站上能下过较弱的 AI，排行第一的 AI 下不过。

(5) 比赛结果

第一轮被一个无脑选边角的 AI 零封了，我觉得可能还是自己设计的不足，才导致 AI 判断不如其它的 AI。后面我仍不断改进自己的 AI，希望变得更强。

# Conclusion:

这次黑白棋 AI 的设计中，虽然比赛结果不尽人意，但是整体的过程仍然是充满趣味和成就感的，在这里，我将自己的几个心得总结一下：

1. 对于黑白棋 AI，评价模型是非常重要的。

   一个好的评价模型能把一个差的评价模型吊起来锤，但是评价模型很多时候靠的是设计人对于游戏或者事务的思考角度和理解，如果能让 AI 快速进行无监督学习训练，就非常厉害了。

2. 多种评价模型的结合可能不如单一的好的模型

   当初在进行比赛的时候，我的 AI 就是结合了位置权重和数量对局势进行评价的，我还想着再加上 min-max 评价和 α - β 剪枝的，但是却被无脑选边角的 AI 吊起来打，当时比赛的时候第二局我还修改了权重，希望 AI 能够更倾向于无脑选边角，结果还是输了。

3. 自博弈学习是一个漫长的过程

   由于时间和设备的原因，我的 AI 也没能得到最好的训练，训练结果应当属于较好,如果能够再多一点时间,AI 或许能够得到更好的结果。

4. 从统计学和 AI 的视角评价和从人类视角评价是截然不同的

   在调试 AI 的过程种，发现其选择的步数本质上是统计的结果，不具有发散性也一定程度上限制了 AI 的上限。

5. 机器学习就像是炼丹

   就像是我的自博弈参数一样，是一群比较参数中挑出来的比较好的而已，但却不一定是最佳的，最坏的可能就是矮子里面挑将军，机器学习在我印象中上限可以很高，但下限也可以很低。只有通过不断的统计学习才得到的结果需要不断被优化不断被加速，才能够提高下限，确保上限。