五子棋设计实验

摘 要

本实验的主要目标是利用α-β剪枝算法和合理的评价函数，实现五子棋的人机对弈的过程。关键技术是利用α-β剪枝算法有效减少搜索的节点数量，有效的提高运行的速度，实现了人机对弈的游戏。同时，合理的评价函数也非常重要，既要足够合理的评判局面又要足够简单，不能太复杂影响速度。本文介绍了具体实现方法。

关键词： α-β剪枝算法、评价函数，五子棋

装

订

线

目 录

装 订 线

1 实验概论

1.1 实验目的

五子棋的游戏大家都玩过，在 15*15 的棋盘上，黑白双方互相下棋，率先在横竖斜三个方向上连续的连成 5 个棋子的一方获胜。本实验旨在设计一款能够和人类对弈的机器人，要能够符合五子棋的规则，运用极小极大算法进行对抗式搜索，同时要用 α - β 剪枝算法来实现搜索的简化加速，也要选择合适的评估函数来判断局面的优劣。

1.2 实验内容

本实验的主要内容包含以下几个方面：首先，需要根据五子棋的规则来设计一个合理的评估函数，评估函数需要满足两个要求，第一，评估函数要能够有效的区分出不同落子对于局面的状态影响，也就是要能够显著区分局面的优劣；第二，评估函数要足够简单快速，因为在每个节点都要计算局面的评估值，如果太慢会显著拖慢整体的运行速度；其次，我们要根据五子棋的规则以及设计的评估函数以及之后拟采用的极小极大算法和 α - β 剪枝算法来设计合理的数据结构，数据结构既要包含必要的信息，又要尽可能的小，以减小内存开销。接着，根据前两步写出极小极大算法和 α - β 剪枝算法；最后，将游戏可视化界面完成，实现游戏人机对弈。

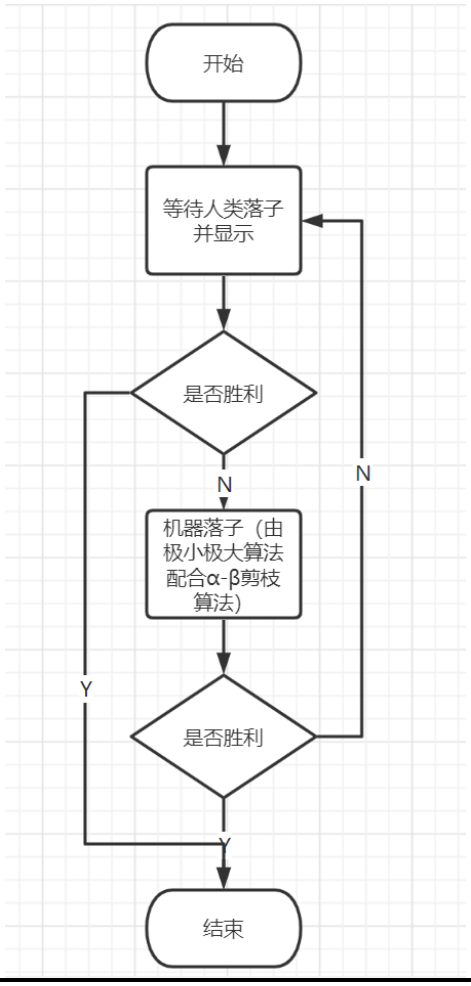
2 实验方案设计

2.1 总体设计思路和设计框架

本实验的主要设计思路如下：首先设计数据结构

```
class StornPoint():
    def __init__(self, x, y, value):
        '''
        :param x: 代表 x 轴坐标
        :param y: 代表 y 轴坐标
        :param value: 当前坐标点的棋子：0:没有棋子 1:黑子 2:白子
        '''
        self.x = x  # 初始化成员变量
        self.y = y
        self.value = value
```

每一个棋子都设置为一个类的对象，其中包含了棋子的坐标以及对应的棋子颜色。在已有数据结构的基础之上设计游戏的整体模块，先监听鼠标点击事件，等到第一次的鼠标点击之后，确认人类落子的位置并显示，然后判断胜负，胜利就结束，否则按照极小极大算法和 α - β 剪枝算法来计算出机器落子的最优点（这个最优是根据评估函数得到的，落子极大程度上依赖以这个值），然后在棋盘上显示，然后判断胜负，胜利就结束（人类失败），否则重复之前的过程。具体流程如图所示：



2.2 核心算法及基本原理

核心算法采用的是极小极大算法并用 α－β 剪枝算法进行加速优化。

首先是极小极大算法：这是在对抗搜索中常用到的算法。首先，五子棋问题中有两个 agent，一个是电脑，一个是人类。我们假设他们都是绝对理性的。我在这个算法中把人类看做正方，正方需要让局面的评估值尽可能的大，反方需要让局面评估值尽可能的小。

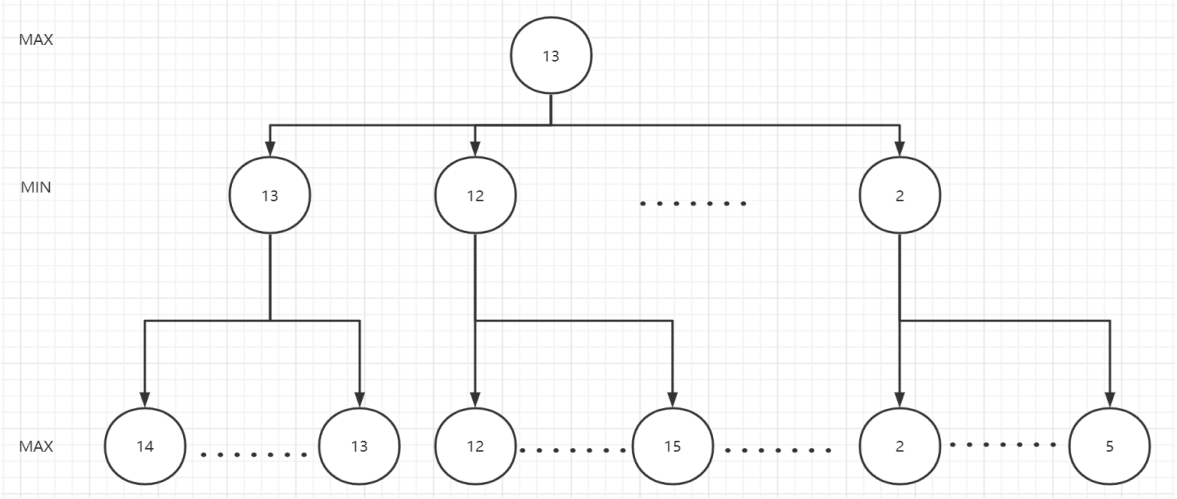


图 2-1

如图 2-1 所示，两个个 agent 都是分别取各自这一层最大或者最小的一个节点，也就是说他们在做一种对抗式的比较。反方在搜索过程中也要考虑正方（使局面评估值变大的一方）做的决策对于搜索的影响。由此得出当前状态的最优解（不能只考虑自己这一方的评估值的大小）。

但是由于我们的棋盘非常大（15*15），没一个节点要拓展的子节点数就要有 200 个左右（后期越来越少，因为可下的地方减少了）。这样的话即使搜索的深度只有两层，那么要拓展的节点数量也是要有 40000 多个，实际运算起来会非常的慢。于是就需要 α－β 剪枝算法来进行优化。

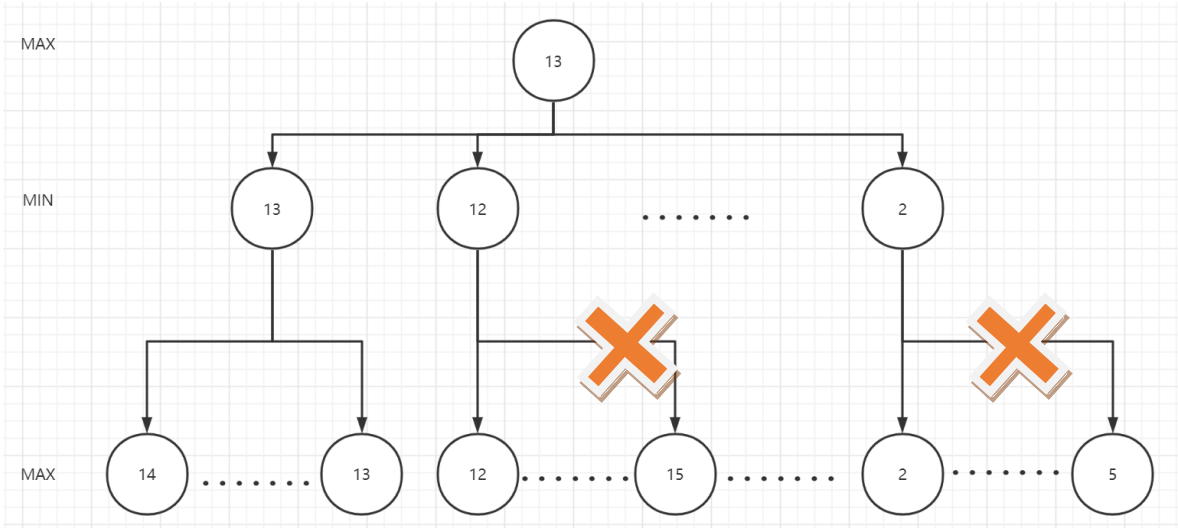


图 2-2

由图 2-2 可见，如果当我搜索完左边这一路结点，当我再去搜索中间这一路时，搜索到结点的评估值为 12<13，后面的结点及其子结点都不需要去搜索了，因为是取这些结点的最小值，已经存在比其父结点的兄弟结点小的结点，那么在 max 层选择时必然不会选择这些结点了，所以就不必再搜索，同样的原理还可以应用到 min 层的结点选择。这就是α－β剪枝算法，有效的在保证搜索完整性的额前提下减少了搜索次数，提高了运算速度。

2.3 模块设计

本实验主要分为五个大模块：主模块、图形模块、监听模块、算法模块、游戏判断模块

主模块

主要函数：main 函数

主要功能：连接其他所有的模块，实现整个游戏的运行，实现正常的人机交互功能，输出当前状态的评估值。设置游戏界面的参数，设置重新开始和悔棋功能。

图形模块

主要函数：initChessSquare 函数，以及 main 函数中相关部分

主要功能：实现图形的可视化，在屏幕上显示游戏界面，显示黑白棋子，显示游戏终止界面的两种不同状态（胜利/失败）。

监听模块

主要函数：eventHander 函数

主要功能：监听各种事件（包括鼠标点击等），然后利用判断模块以及计算点击位置返回不同的事件值，后续模块调用这个模块实现人机交互。

EVENT_WIN=1

EVENT_LOSE=2

EVENT_RESTART=3

EVENT_REGRET=4

EVENT_NORMAL=5

算法模块

主要函数：evaluatePoint 函数、getValue 函数、MaxValue 函数、MinValue 函数、alpha_beta 函数

主要功能：这个模块包含了这个游戏人工智能的计算模块，包括极小极大算法以及用α－β剪枝算法的应用。同时还有求评估值的函数。

游戏判断模块

主要函数：judgeResult

主要功能：判断落完子之后的结果的模块。

2.4 其他创新内容或优化算法

本实验对于评估函数的设计有所创新：
首先是网上主流的一种评估方式：依靠对于常见的几种棋型进行打分，比如说冲四，划三等棋型打出相应的分值。如表 2-1 所示：

术语	得分
成五	5000000 分
活四	100000 分
冲四	10000 分
单活三	8000 分
跳活三	7000 分
眠三	500 分
活二	50 分
眠二	10 分

表 2-1

这种评估方式非常符合我们一般人下棋时候的思路，但是由于要在各个方向上枚举出这些棋型然后进行比对，所以所需要的时间比较长，复杂度略微有点高，而且非常依赖于人类的主观评分，比如说对于跳获三和单活三具体哪个评分高主要依靠人的经验，但是每个人的经验不同，可能设计的 ai 就不同，比如这张表中单活三的分比跳活三的分高，但是我就觉得跳活三更加隐蔽，应该是跳活三分高，这就比较依赖人的主观经验了。

我采用另一种评估方式。首先来看一种极端情况（图 2-3）：当这种情况下时会发现，虽然白棋下的非常的松散，但是，在红框之中黑棋是不可能连成五子的，这是因为白棋之间在任意的横竖斜三个方向上的相邻距离都不会超过 4 个子的间距。由这一点受到启发，我们可以发现，五子棋的棋盘都可以划分为多个五元组（图 2-4），每个五元组就是棋盘上连续的五个可以下棋的落子点。五元组可以是横的也可以是竖的，还可以是斜的。

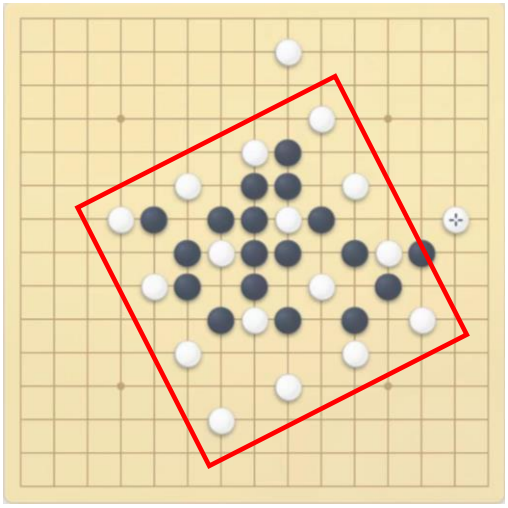


图 2-3

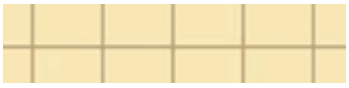


图 2-4

由此，我们总结出五子棋游戏是一个抢占五元组的游戏，谁先能够占领一个五元组谁就获胜，不难计算出 15*15 的棋盘一共有 15*10*2+10*10*2=500 个五元组。我们的评估函数为对于每个五元组评分求和来求得局面的总分。

首先我们明确，当一个五元组既有黑子又有白子的时候，这个五元组就没有意义了，因为不论是黑白双方都不可能在这个五元组中连成五个，这也是为什么图 2-3 中黑棋没法在红框中连成五子的原因，因为白棋最大限度的污染了五元组。因此，我们可以将五元组的情况分成以下几类：

1. 既有黑又有白
2. 空
3. 单色棋子 1 个
4. 单色棋子 2 个
5. 单色棋子 3 个
6. 单色棋子 4 个
7. 单色棋子 5 个

然后分别对这些情况赋分，例如我对情况 1 赋 0，情况 2 赋 7 情况 3 赋 35，情况 4 赋 800，情况 5 赋 15000，情况 6 赋 800000，情况 7 赋无穷大。这样通过遍历五元组然后求和就能得到局面的评估值。

这种算法的优势主要有三点：

1. 运算简单，速度快捷，不需要对比棋型，只需要遍历五元组然后计数即可，大大提高运算的速度。
2. 从计算机的角度理解五子棋，比起以人类的思维习惯来理解减少了许多有争议的棋型的

评估值。对于不同五元组的评分也没有特别细致的要求，不需要以人类的经验来进行评分，更加客观。

3. 代码书写遍历，只需要遍历五元组然后计数即可，利用宏定义也可以方便的修改五元组的评分。

综上，最终本项目采用以上方法作为评估函数。

3 实验过程

3.1 环境说明

操作系统：Windows11
开发语言：Python3.9
开发环境：Pycharm
核心使用库： pygame、time、sys

3.2 源代码文件清单，主要函数清单

文件清单：main.py：包含主函数以及图形化显示，监听事件的函数
chessGame.py:包含 ai 的算法， α - β 剪枝算法

函数清单：

函数名：main
函数参数：无
功能： 联结所有模块，实现人机对弈，同时还要处理一些细节上的问题，如图片位置的设置等
返回值： 无

函数名：judgeResult
函数参数：i：落子的横坐标（数组中的，不是棋盘上的坐标）
j：落子的纵坐标（数组中的，不是棋盘上的坐标）
value：落子的类型 （CHESS_WHITE=2 , CHESS_BLACK=1）
功能： 判断落子的胜负
返回值： 胜利返回 True,否则返回 False

函数名：eventHander
函数参数：无
功能： 监听事件，包括鼠标点击等，然后根据事件修改 initChessList 的值，然后还要判断落子后的胜负
返回值： 各种事件，宏定义如下
EVENT_WIN=1
EVENT_LOSE=2
EVENT_RESTART=3
EVENT_REGRET=4
EVENT_NORMAL=5

函数名：initChessSquare
函数参数：x 坐标横方向的偏移量
y 坐标纵方向的偏移量
功能： 初始化 initChessList 这个数组（初始化棋盘）（初始棋子的值都是空）
返回值： 无

函数名：evaluatePoint
函数参数：initChessList：棋盘数组
i：要评估的点的横坐标
j：要评估的点的纵坐标
功能： 计算一个点的评估值，评估值就是以那个点为头的五元组
返回值： 返回该点的评估值

函数名：getValue
函数参数：initChessList：棋盘数组
功能： 计算整个棋盘的评估值
返回值： 返回棋盘的评估值

函数名：MaxValue
函数参数：initChessList ：棋盘数组
depth ：搜索的当前深度
alpha ：极大极小算法的α值（max 父结点）
beta ：极大极小算法的β值（min 父结点）
功能：求子结点最大值
返回值： 子结点的最大值

函数名：MinValue
函数参数：initChessList ：棋盘数组
depth ：搜索的当前深度
alpha ：极大极小算法的α值（max 父结点）
beta ：极大极小算法的β值（min 父结点）
功能：求子结点最小值
返回值： 子结点的最小值

函数名：alpha_beta
函数参数：initChessList ：棋盘数组
功能：求当前局面的最优解，并且在棋盘上标注
返回值： 返回落子的棋子的坐标

3.3 实验结果展示

中间对弈的过程（图 3-1），黑棋先走：

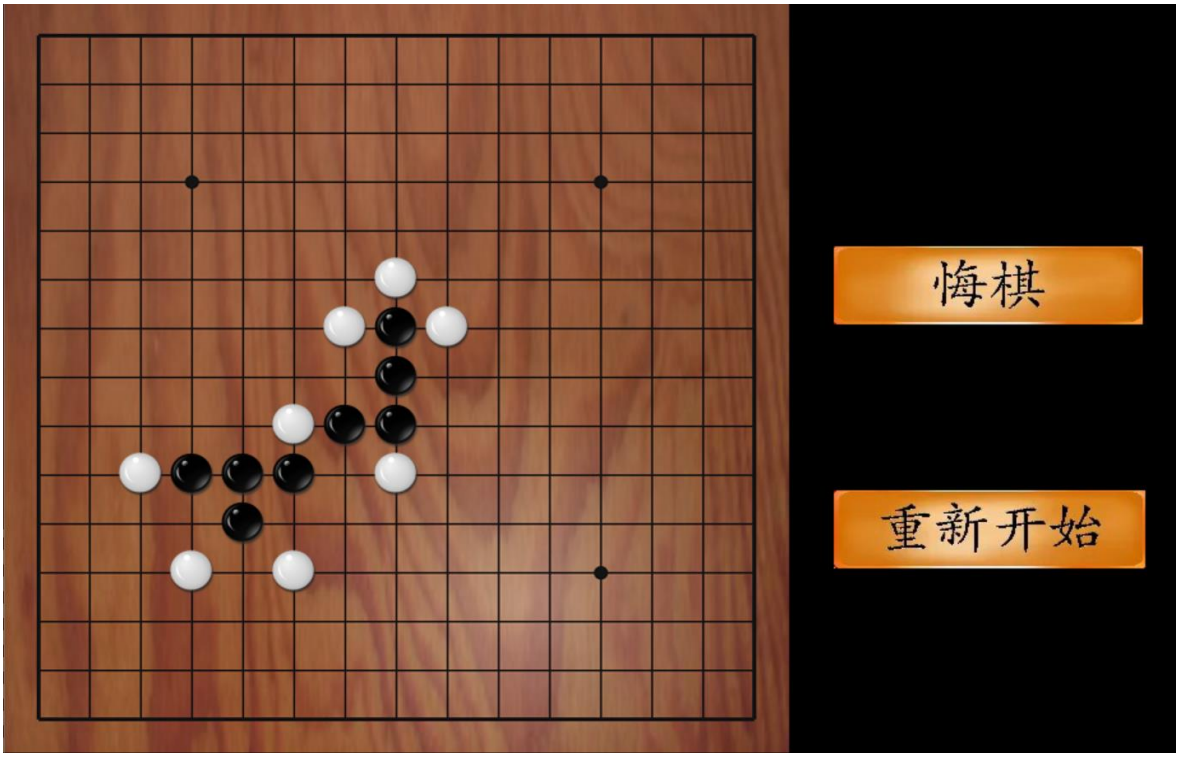


图 3-1

胜利界面（图 3-2）：

图 3-2

失败界面（图 3-3）：

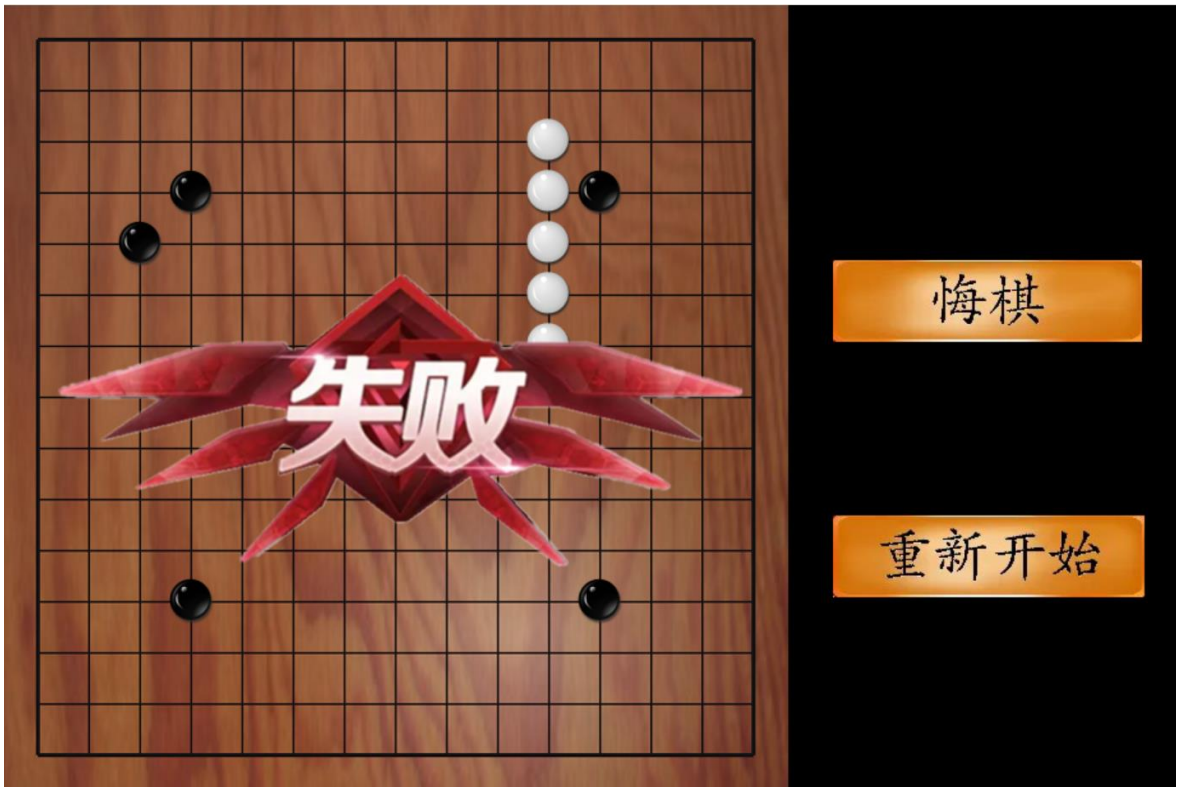


图 3-3

在 run 窗口输出局面评估值（3-4）：

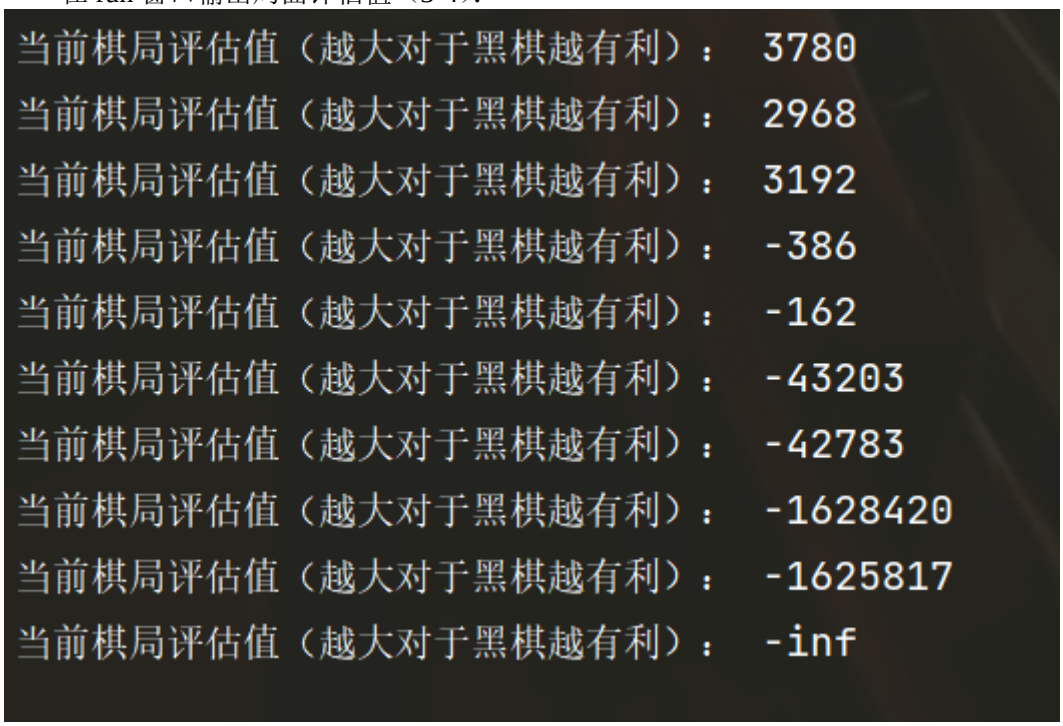


图 3-4

结论：使用 α - β 剪枝算法，最终能够找到有效的解，但是速度依旧不是非常的快速，我最终限制的深度为两层，也就是只看两步的深度，这样虽然不是最优解，但也是深度为 2 的局部最优解了。最后的效果也是比较不错，不仅能够有效的防守，而且也能够合理的进攻。

4 总结

4.1 实验中存在的问题及解决方案

在本次实验中，结构体的设计是一个问题。一开始我是打算在结构体里设置一个棋盘数组来记录棋盘状态，但是后来就发现这样做的话将会有非常大的内存开销，例如我只进行深度为 2 的搜索算法，加入不考虑剪枝的情况，那么我将要拓展 40000 多个结点，如果每个结点都有一个 15*15 的棋盘，那将是非常大的内存开销。这还只是深度为 2，如果深度为 3 就要再多 225 倍，这是非常可怕的。因此我设计将这个棋盘数组移出结构体，而具体的做法是在拓展一个结点之前直接在棋盘数组上落子，拓展完之后再把这个落子点置空，由于搜索的时候采用的是一种类似于深度优先的搜索方式，所以每次都可以在父结点的基础上落一个子就可以了。最终实现了全局只用一个棋盘数组就完成了搜索，大大减少了搜索时的内存开销。

其次是在 α - β 剪枝算法寻求答案时，一开始我选择把计算到的评估值存在每个棋子的类的成员变量里，然后在搜索结束后比对评估值找到那个评估值和函数返回值相同的点，在那里落子，但是由于搜索时，一个点不一定只会出现一次（不同分支可能出现多出），这就导致我的评估值被覆盖的可能，有时候走得不是最优解，甚至不落子（评估值被覆盖后找不到了）。对于这个问题，我的解决办法是用一个去全局的答案棋子结构体，一旦找到更优解就更新他，最后算法结束这个答案棋子就是我下一步要走的棋子。

4.2 心得体会

这次的五子棋实验中最大的难题在于由于搜索需要拓展结点的个数过多而导致的搜索速度过慢的问题。对于这个问题的解决的方法集中在对于搜索过程的简化和每次拓展一个结点之后的评估值的计算的简化。对于前者来说，本实验采用 α－β 剪枝算法，在保证搜索完整性的情况下有小减少了需要搜索拓展的结点，既能找到局部的（深度受限）最优解，又能提高运算速度。可以说这个算法是本实验的核心算法。而对于后者来说，本实验选用计算五元组评分的方式，方法简单，速度更快。两者的结合使得本实验能够快速的算出答案，最终实现了人机对弈的过程。本实验给我的最大感受首先就是合理算法的应用对于计算速度的影响之大，其次是数据结构的设计也非常重要，比如我最终将棋盘数组移出类，大大减少内存开销，而对于这一改变，也需要算法上的改变来配适，可见算法对于数据结构也有反向作用。做项目时应该先规划好，不然做到后面会混乱。

4.3 后续改进方向

本实验虽然顺利完成了，但是仍然有不少值得改进的地方，比如界面的美化，评估函数评估五元组的评分的优化等等。但是最重要的是要优化搜索算法，目前的搜索深度只能局限在深度为 2，深度再深就会比较慢了，我的下一步的目标就是要进一步提升搜索效率，这其中主要包括改进搜索算法，如限定搜索范围，在更小范围寻求局部最优解，或者进一步简化评估函数算法，提高拓展单个结点所需要的时间。