电子科技大学

计算机专业类课程

实验报告

课程名称：程序设计基础

学院专业：计算机科学与工程学院

学生姓名：杨惟楚

学　　号：2020080910007

指导教师：俸志刚

日　　期：2021 年 6 月 19 日

电 子 科 技 大 学

实 验 报 告

实验一

一、实验室名称：

电子科技大学清水河校区主楼 A2-412

二、实验项目名称：五子棋算法设计实现

Github 仓库链接：https://github.com/xenoppy/Tree

三、实验目的：

1．熟悉并完全掌握 C 与 C++程序设计
2．熟悉并使用各种数据结构
3．学会分析问题并寻找、使用相关算法
4．熟悉将程序写入多文件的程序设计方式
5．实现带有胜负判断的五子棋游戏程序
6．设计能进行对弈的五子棋 AI

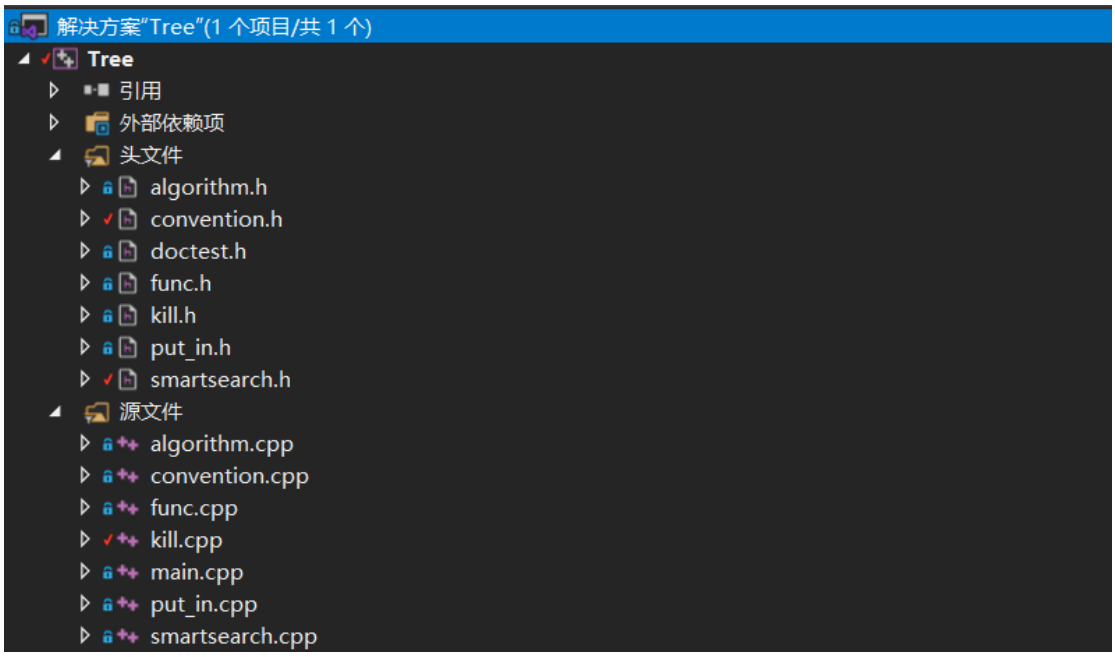
四、实验主要内容：

设计并实现五子棋游戏
设计并实现估值函数
设计并实现最大最小值搜索树
在最大最小值搜索树中增加 α - β 剪枝
增添可落子区域检测评估函数
增加算杀部分增强棋力
增加启发式搜索加快运算效率
测试

五、实验器材（设备、元器件）：

计算机型号：机械革命 code01
CPU：AMD R7-4800H
内存：16G
操作系统：Windows10
开发环境：Visual studio2019 + Visual studio code
测试环境：Visual studio2019

六、各文件的作用与其内容：



Doctest.h：doctest 的测试框架

Convention.h 和 Convention.cpp： 游戏预设，包括搜索深度，棋盘大小以及其他基本宏定义。

func.h 和 func.cpp：包含一些基本功能的函数，包括打印胜利结果，棋盘打印和可落子判定函数

put_in.h 和 put_in.cpp：包含两种向五子棋游戏输出的方式，一种是手工输入，一种是 AI 输入。

algorithm.h 和 algorithm：包含估值函数和极大极小值搜索。

kill.h 和 kill.cpp：包含算杀模块相关函数 killcheck 和 killSearch。

smartsearch.h 和 smartsearch.cpp：包含启发式搜索相关内容。其中有类 node 与类 nodevector。

main.cpp：主函数。包括棋盘初始化、先后手的处理以及对弈循环体。

七、实验步骤：

1. 问题描述

进行五子棋的对弈。

五子棋是一种两人对弈的纯策略型棋类游戏，通常双方分别使用黑白两色的棋子，下在棋盘直线与横线的交叉点上，横竖或斜方向先形成 5 子连线者获胜。

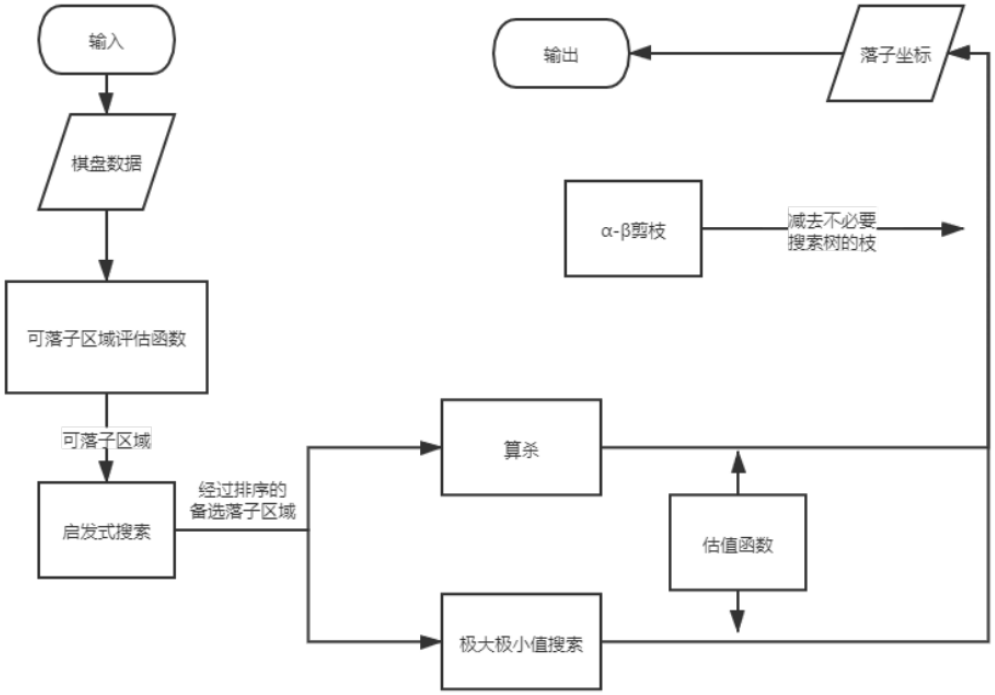
而该实验所设计的即五子棋对弈算法。

2. 算法分析与概要设计

输入:当前的棋盘数据。棋盘由一个 15*15 的 int 型的二维数组进行存储，空白记为 2，黑棋记为 1，白棋记为 0。

输出：一个有序数对(x, y)，为这一步所下棋的坐标。

输入与输出之间转换的算法：



3. 核心算法的详细设计与实现

a) 全局估值函数

对目前的棋盘进行估值。输入为一个二维数组的指针以及一个记录分数的整形指针，估值完成后将分数更新到记录分数的整形中，无返回值。

此估值函数对棋面进行分析，对不同类型的棋子分布情况进行打分。其中 User1（白棋）的分数默认为负，Users2（黑棋）的分数默认为正，总分数为两个分数相加。在此处认为：

活一 （两边都没有被堵的最大连续 1 个的棋形）：10

活二：100

活三：1000

活四：1000000

活五：1000000000000000

死一 （有一边被堵住的最大连续 1 个的棋形）：1

死二：10

死三：100

死四：1000

死五：1000000

双死一 （两边都被堵住的最大连续 1 个的棋形）：0

双死二：1

双死三：10

双死四：100

该算法由 4 次双重循环组成，分别检测横、竖、左上右下、左下右上四个方向上的连着的棋子的数量以及状况，并对分数进行修改。

```
73    //估值函数
74    void evaluate(int** board,int* temp) {
75        int sum1,sum2,minus1,minus2,score=0,a[6] = {1,10,100,1000,1000000,10000000000000
76        //横向
77        for (int i = 0; i < BOARD_LENTH; i++) { ... }
101       //纵向
102       for (int j = 0; j < BOARD_LENTH; j++) { ... }
128       //左下右上
129       for (int i = 0; i < BOARD_LENTH * 2 - 1; i++) { ... }
156       //左上右下
157       //该情况需要用两个while去遍历所有路径
158       for (int i = 1 - BOARD_LENTH; i <= BOARD_LENTH - 1; i++) { ... }
217       //结束
218       * temp = score;
219       return;
220   }
```

b) 极大极小值搜索

对后续的棋局进行预测，找到数轮之后能使自己分数尽可能高的下法。

在极大极小值搜索树中，每一个用户下一步棋都是树的新一层。每一层都默认选择对己方最有利的情形。

采用深度优先搜索来实现极大极小值搜索。后续增加 α - β 剪枝增加效率。(位于文件 Algorithm.cpp⬇)

```
228     //极大极小搜索
229     for (int i = 0; i < BOARD_LENTH; i++) {
230         for (int j = 0; j < BOARD_LENTH; j++) {
231             if (check(Board, i, j)) {
232                 //在i, j处落子
233                 t = *temp;
234                 Board[i][j] = user;
235                 //搜索树抵达底层
236                 if (times == 1) { ... }
244                 //搜索树未抵达底层
245                 else { ... }
257             }
258         }
259     }
260     return max_score * (user * 2 - 1);
```

c) α - β 剪枝

在 MAX 层时把当前层已经搜索到的最大值 X 存起来，如果下一个节点的下一层会产生一个比 X 还小的值 Y，也就是说这个节点对手的分数不会超过 Y，而对手不可能做出这样的决策，那么这个节点显然没有必要进行计算了。

同理，在 MIN 层时把当前层已经搜索到的最小值 X 存起来，如果下一个节点的下一层会产生一个比 X 还大的值 Y，也就是说这个节点对手的分数不会小于 Y，而对手不可能做出这样的决策，那么这个节点也有必要进行计算了。

因为 user 为 1 或 0，则乘上（2 * user – 1）可以自动完成 MAX / MIN 层的取大/取小比较。变量 extre 是来自上一层的极值，与该层的分数进行比较。（位于 algorithm.h 与 algorithm.cpp↓）

```
if (max_score > extre * (2 * user - 1)) {
    return extre;
}
```

d) 可落子区域检测评估

在一盘棋局中，大部分空子处其实都是无意义的。因此用该函数排除一部分区域。

检查每一个位置以自身为中心形成的 3*3 的正方形，其中是否有子。若无子则不对该位置进行搜索，掠过。

位于 func.h 与 func.cpp↓

```
bool check(int** board,int x, int y) {
    if (board[x][y] == EMPTY) {
        for (int i = max(0, x - 1); i < min(BOARD_LENTH, x + 2); i++) {
            for (int j = max(0, y - 1); j < min(BOARD_LENTH, y + 2); j++)
                if (board[i][j] != EMPTY) {
                    return true;

                }
        }

    }
    return false;
}
```

e) 算杀

考虑到 6 层的搜索有时还是偏弱，故考虑在杀棋（冲死四、冲活三）中搜 10 层。

Killcheck 用于判断是否有棋可杀，有棋杀就继续向深处搜索，无棋可杀就返回。

KillSearch 即算杀搜索树的实现。（位于 Killsearch.h 与 killSearch.cpp↓）

```
int KillSearch(int** Board, int user, int times, int* temp, int extre) {
    int max_score = -2147483647, save;
    for (int i = 0; i < BOARD_LENTH; i++) { ... }
bool checkKill(int** board, int user, int x, int y) { ... }
```

因为它实际上占用时间并不多，所以 KillSearch 被插入在每一层搜索中。下图红框中即搜索中插入的算杀搜索。（位于 algorithm.cpp↓）

```
219  int search(int ** Board, int user, int times,int* temp,int extre) {
220      int max_score = -2147483647,save;
221      int t = *temp;
222      //找到该层的极值
223      //同时展开的算杀搜索
224      KillSearch(Board, 1-user, KILLDEPTH, &t, max_score * (user * 2 - 1));
225      //极大极小搜索
226      for (int i = 0; i < BOARD_LENTH; i++) {
227          for (int j = 0; j < BOARD_LENTH; j++) {
228              if (check(Board, i, j)) {
229                  //在i, j处落子
230                  t = *temp;
231                  Board[i][j] = user;
232                  //搜索树抵达底层
233                  if (times == 1) { ... }
242                  //搜索树未抵达底层
243                  else { ... }
255              }
```

f）启发式搜索

将可落子范围中的位置进行简单的排序（此处采用一层搜索之后的估值评分来排序），在极大极小值搜索中优先搜索分数高的位置，来优化 α - β 剪枝的效率。

Smartevaluate 函数进行启发式搜索时的落子重要性评估。

定义类 Node，包含横坐标 x，纵坐标 y，以及分数 score 三个参数。

定义类 Nodevector，包含两个 vector<Node*>,其中一个存储未排序好的可落子的位置，另一个存储排序好的位置。（位于 smartsearch.h⬇）

```
int smartevaluate(int** Board, int x, int y,int user);
class node{
public:
    node(int a, int b) { x = a; y = b; score = 0; }
    int score, x, y;
};
class nodevector{
public:
    std::vector<node*>Nodevector;
    std::vector<node*>SortedNodevector;
    void sort() { ... }
    void initialize(int** Board,int user) { ... }
    void end() { ... }
};
```

之后按照启发式搜索给出的顺序进行算杀搜索与极大极小值搜索。

（位于 put_in.cpp⬇）

```
51      //开启第一层搜索
52      //因为第一层搜索用到了启发式搜索，所以单独拎出来。
53      for (std::vector<node*>::iterator i = Nodevector.Nodevector.begin();
54          i != Nodevector.Nodevector.end();
55          i++) {
56          if (check(Board, (*i)->x, (*i)->y)) {
57              Board[(*i)->x][(*i)->y] = user;
58              int tempp = t;
59              evaluate(Board, user, 0, 0, &tempp);
60              if (abs(tempp) > 10000000) { ... }
65              int save = search(Board, 1 - user, DEPTH - 1, &t, (int)max_score);
66              if ((user * 2 - 1) * save > (user * 2 - 1) * max_score) { ... }
71
72              //复原
73              Board[(*i)->x][(*i)->y] = EMPTY;
74          }
```

八、实验数据及结果分析：

(测试需要给出测试用例（从正常，边界，错误等各方面给出测试用例，建议用 3 个表格的形式给出 3 种不同类型的测试用例），贴图（运行结果截屏），分析。做看图说话。每一图都要给出图名。有表则给出表名。请根据实验实际，用大量语言进行描述讨论。)

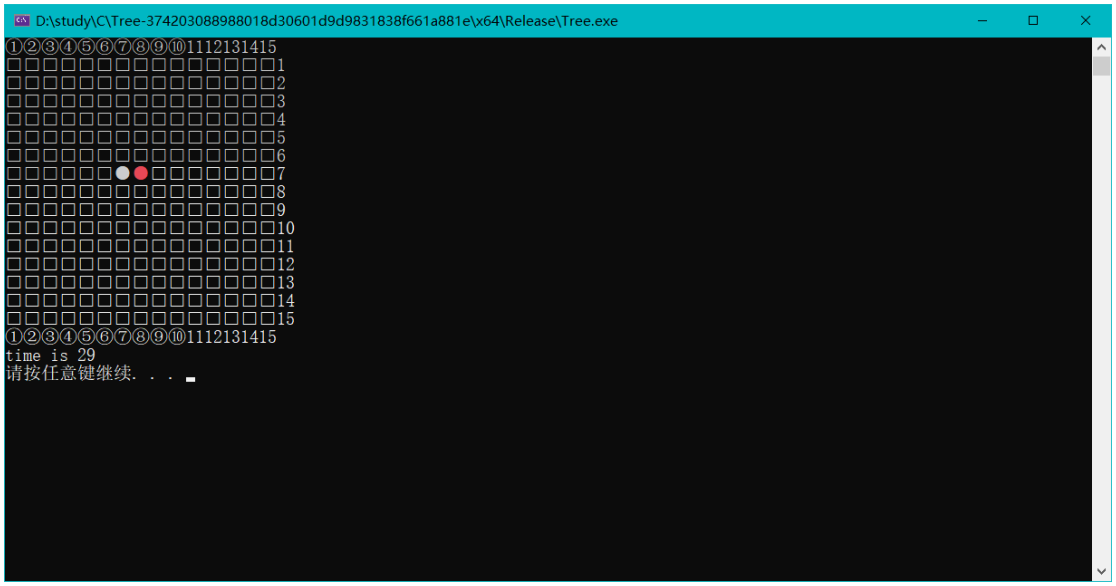
1）搜索 4 层，算杀搜索 10 层

user0 是用户，执红棋。

user1 是电脑，执白棋。

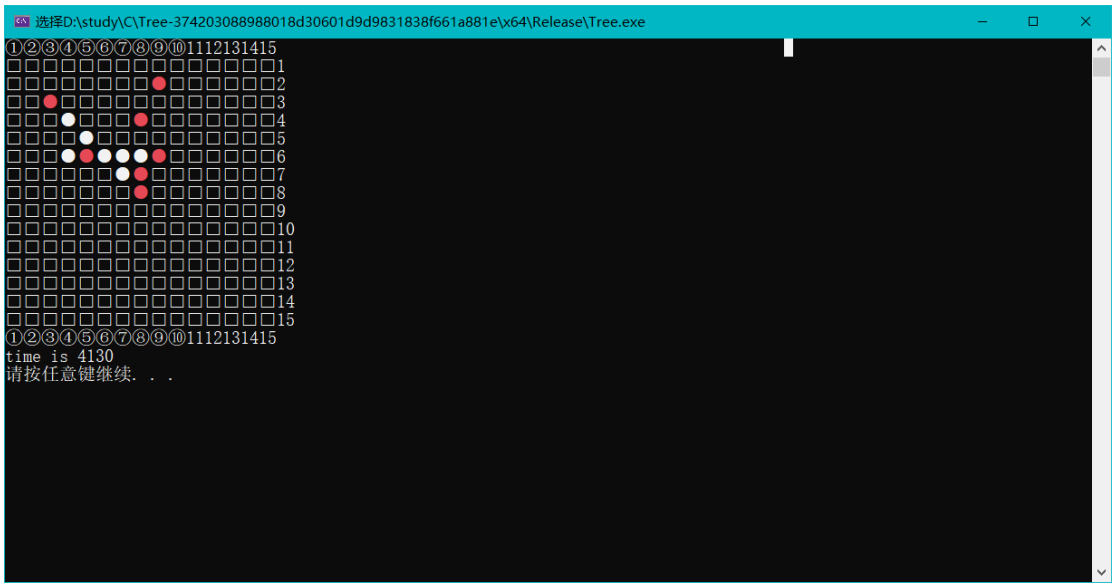
运行时间：

棋盘较小时计算速度较快，时间为 29 毫秒



棋盘逐渐变大之后搜索范围会变大，时间会长到 1000 毫秒数量级

棋力：



棋局 1：

红棋 User0（人）胜。⬇

棋局 2：

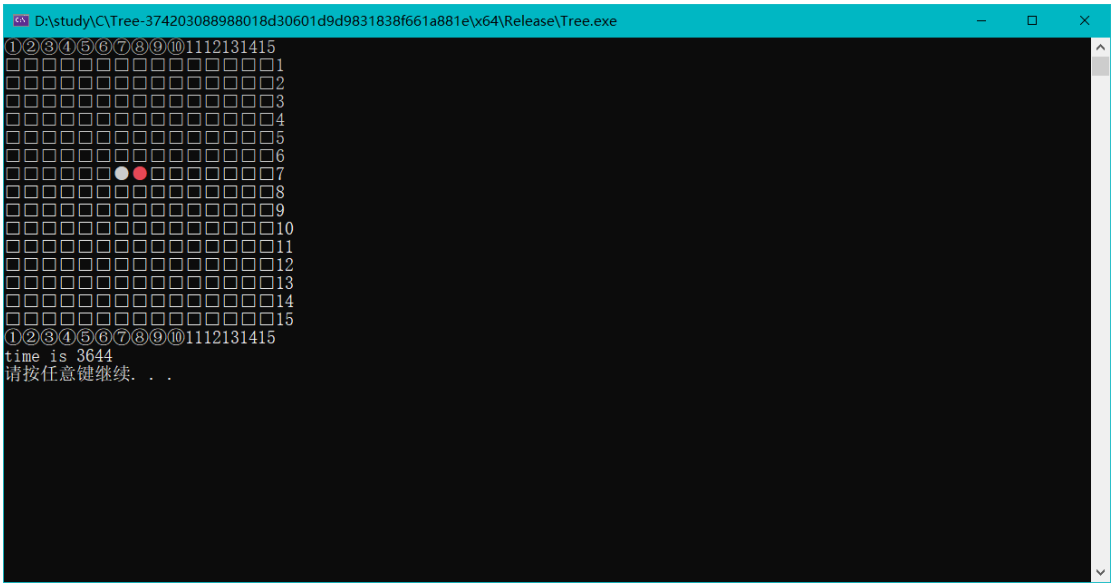
白棋 User1（AI）胜。⬇

偶尔能赢我，可见棋力已然不凡⬆

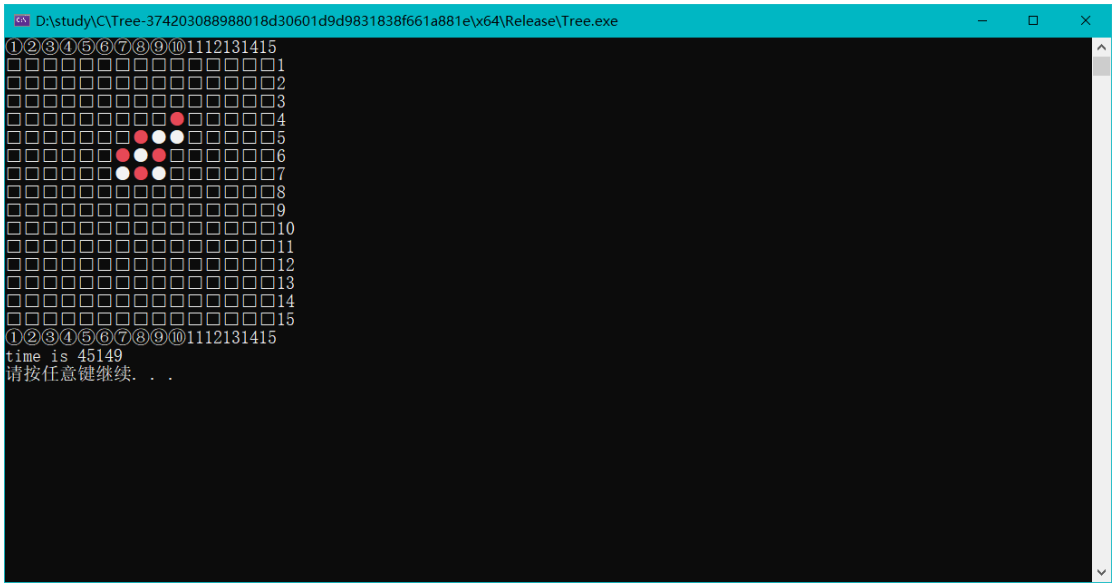
2） 搜索 6 层，算杀搜索 10 层。user0 是用户，执红棋。

运行速度：

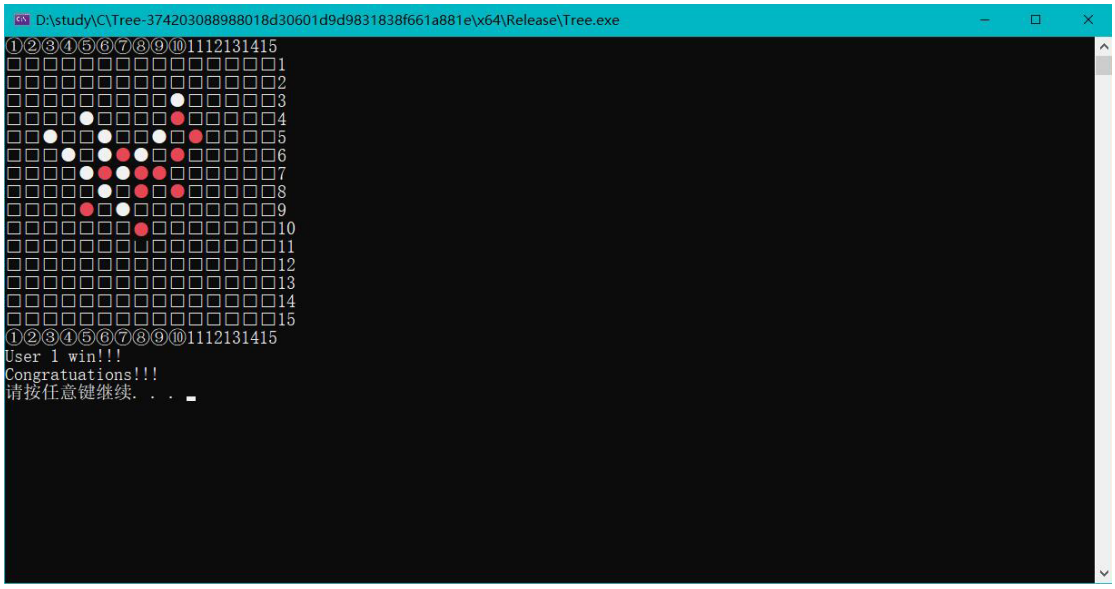
游戏初棋子覆盖范围较小时的速度尚可，在 1000ms 数量级



当棋子覆盖范围较大之后，速度逐渐变慢。但得益于启发式搜索和α-β剪枝，随着棋盘扩大带来的时间增加并没有特别特别严重，仍控制在10000ms数量级。



棋力评估：
六层的极大极小值搜索和10层的算杀，棋力已经较强



白棋User1（AI）胜利↑

九、总结及心得体会：

优点：

配合启发式搜索、α-β剪枝、落子判断模块和10层的算杀、4层的极大极小值搜索，该五子棋AI已经可以在具备一定棋力的同时达到较高的速度了。

将极大极小值搜索的层数加到6层，也就是10层的算杀、4层的极大极小值搜索，会拥有更高的棋力，不过花费的时间也会达到几秒乃至几十秒。

缺点：

1、当对方已经胜利之后（如出现双活三、活四等情况）会随便下，显得很没有体育精神。

2、当棋盘较大之后会速度会变得比较慢，在极大极小搜索深度达到6层时显得尤为严重。

3、未添加迭代加深模块，有时自己已经胜利之后（如出现双活三、活四等情况）不会选择直接下赢而是会去把能冲的四都冲完再终结比赛。。

心得体会：在这次实验过程中，意识到了代码的简洁性的重要，在代码不断重构与优化的过程中逐步升级了我的AI，也逐步消除了一些bug。

十、对本实验过程及方法、手段的改进建议及展望：

本实验过程老师采取了以自我学习为主，参考资料、老师指导为辅的教学方式，老师很好地培养了我自学的能力。此外，感谢老师为我们提的建议，受益匪浅。

展望：修改现有bug，进一步提高算力，进一步加快计算效率，达到快速、强力两不误。

报告评分：

指导教师签字：