装

订

线

同济大学

人工智能实验报告

（αβ剪枝算法 —五子棋）

学　号　：1453381

姓　名　：曾 鸣

班　级　：计算机二班

日　期　：2017 年 6 月 8 日

目录

1. 题目：基于α β剪枝算法的五子棋 1
1.1 五子棋介绍 1
1.2 引入 1
2. 实验目的和环境 2
2.1 实验目的 2
2.2 实验环境 2
3. 算法和实现 2
3.1 棋形介绍 2
3.1.1 术语 2
3.1.2 棋形 3
3.2 估值函数 3
3.3 算法 4
3.3.1 博弈树 4
3.3.2 极大极小值算法 4
3.3.3 α β 剪枝算法 5
3.3.4 算法优化 6
3.3.4.1 局部搜索 7
3.3.4.2 优先值启发 7
3.3.5 核心算法伪代码 7
3.3.6 核心算法流程图 8
4. 结果演示 8
5. 附件：源程序 10

装 订 线

1. 题目：基于α β剪枝算法的五子棋

1.1 五子棋介绍

简介：

五子棋是世界智力运动会竞技项目之一，是一种两人对弈的纯策略型棋类游戏，是世界智力运动会竞技项目之一，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，先形成 5 子连线者获胜。

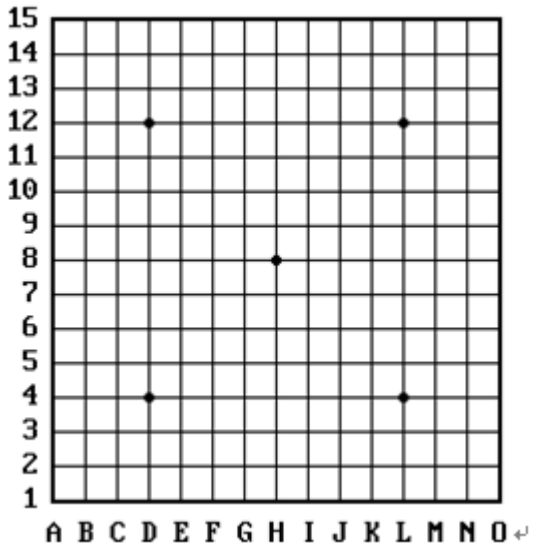
五子棋规则：

五子棋有多种规则，分为：原始规则、无禁类规则、有禁类规则；其中无禁类规则又有 Standard Gomoku 规则、Gomoku-Pro 规则、Swap 规则、Swap2 规则等。

本次五子棋采用原始规则：

行棋：黑子先行，一人轮流一著下于棋盘空点处。

胜负：先把五枚或以上己棋相连成任何横纵斜方向为胜。（长连仍算胜利）



棋盘示例图

1.2 引入

人工智能是一门综合性很强的边缘科学，它研究如何使计算机去做那些过去只能靠人的智力才能完成的工作。而 agent 博弈是人工智能的重要分支，在博弈问题中提高机器的智能水平，敌对搜索对这一问题的经典解决方法，而极大极小算法是敌对搜索中最为基础的算法，为了提高极大极小搜索的效率，在极大极小搜索算法的基础上使用 Alpha-Beta 剪枝所产生的 Alpha-Beta 搜索算法则是其中最重要的算法之一。

本次试验利用 Alpha-Beta 搜索算法实现人机博弈中的五子棋游戏，并在此基础上，利用局部搜索、优先值启发、限制深度等方法来提高 Alpha-Beta 搜索算法的效率。

2. 实验目的和环境

2.1 实验目的

熟悉人工智能系统中的问题求解过程；

学会利用对抗搜索解决博弈问题；

熟悉对抗搜索中的极大极小值算法，以及在此基础上的 Alpha-Beta 搜索算法的应用；

熟悉对五子棋问题的建模、求解及编程语言的应用。

2.2 实验环境

硬件环境：

计算机型号：惠普 Pavilion M4
内存：4.00GB
CPU：Intel Core i5　2.6GHz

软件环境：

操作系统：Windows10 版本
IDE：Visual Studio 2015 社区版
图形库：EasyX
实现语言：C++（C++11 标准）

3. 算法和实现

3.1 棋形介绍

3.1.1 术语

〖阳线〗　直线，棋盘上可见的横纵直线。
〖交叉点〗　阳线垂直相交的点，简称“点”。
〖阴线〗　斜线，由交叉点构成的与阳线成 45°夹角的隐形斜线。
〖落子〗　棋子直接落于棋盘的空白交叉点上。
〖轮走方〗　“行棋方”，有权利落子的黑方或白方。
〖着〗　在对局过程中，行棋方把棋子落在棋盘无子的点上，不论落子的手是否脱离棋子，均被视为一着。
〖回合〗　双方各走一着，称为一个回合。

〖开局〗 在对局开始阶段形成的布局。
〖连〗 同色棋子在一条阳线或阴线上相邻成一排。

3.1.2 棋形

〖长连〗 五枚以上同色棋子在一条阳线或阴线上相邻成一排。
〖五连〗 只有五枚同色棋子在一条阳线或阴线上相邻成一排。
〖成五〗 含有五枚同色棋子所形成的连，包括五连和长连。
〖四〗 在一条阳线或阴线上连续相邻的 5 个点上只有四枚同色棋子的棋型。
〖活四〗 有两个点可以成五的四。
〖冲四〗 只有一个点可以成五的四。
〖死四〗 不能成五的四。
〖三〗 在一条阳线或阴线上连续相邻的 5 个点上只有三枚同色棋子的棋型。
〖活三〗 再走一着可以形成活四的三。
〖连活三〗 连的活三（同色棋子在一条阳线或阴线上相邻成一排的活三）。简称“连三”。
〖跳活三〗 中间隔有一个空点的活三。简称“跳三”。
〖眠三〗 再走一着可以形成冲四的三。
〖死三〗 不能成五的三。
〖二〗 在一条阳线或阴线上连续相邻的 5 个点上只有两枚同色棋子的棋型。
〖活二〗 再走一着可以形成活三的二。
〖连活二〗 连的活二（同色棋子在一条阳线或阴线上相邻成一排的活二）。简称“连二”。
〖跳活二〗 中间隔有一个空点的活二。简称“跳二”。
〖大跳活二〗中间隔有两个空点的活二。简称“大跳二”。
〖眠二〗 再走一着可以形成眠三的二。
〖死二〗 不能成五的二。
〖先手〗 对方必须应答的着法，相对于先手而言，冲四称为“绝对先手”。
〖三三〗 一子落下同时形成两个活三。也称“双三”。
〖四四〗 一子落下同时形成两个冲四。也称“双四”。
〖四三〗 一子落下同时形成一个冲四和一个活三。

3.2 估值函数

五子棋是一种零和游戏，但是五子棋的搜索树有较多分支因子以及深度较大，限于有限的计算资源，实际中不可能从搜索树的根节点搜索到最终棋局的叶子节点状态，我们只能限定其搜索深度，然后对棋局状态进行评估。

进行棋局估分的具体方式是对棋盘上已经落子的每个棋子根据其在棋盘上的位置以及与周围棋子形成的棋形进行打分，整个棋局状态的估分由己方所有棋子估分和减去敌方棋子估分和，即 F(state)=$\sum G1_i$-$\sum G2_i$

棋形估值函数具体规则：

根据五子棋的特点和根据实际情况多次调整，本次采用的棋形估值如下：

```
#define VALUE_MAX   10000000      //长连/成五  获胜

#define VALUE_L4    150000        //活四
#define VALUE_W4    7500          //冲四
#define VALUE_D4     20           //死四

#define VALUE_L3    7500          //活三
#define VALUE_W3    2000          //冲三
#define VALUE_D3     10           //死三

#define VALUE_L2    1000          //活二
#define VALUE_W2    200           //冲二
#define VALUE_D2     5            //死二

#define VALUE_L1    100           //活一
#define VALUE_W1    50            //冲一
#define VALUE_D1     0            //死一
```

3.3 算法

3.3.1 博弈树

下棋双方在棋盘上下棋，把棋局状态作为节点，当一方任意下棋时，局面就会从一个状态转移到另一个状态。对于一个确定的局面状态，有很多种可行的走法，每一种走法都会使局面状态转移到一个不同的子状态，把子状态和生成它的父亲状态用一条边连接，然后向下递归，直到下棋结束，整个过程生成了一棵树，即为博弈树。

3.3.2 极大极小值算法

在通常的局面中，我们往往想通过少量的搜索，为当前局面选择一步较好的走法，然而在五子棋的棋局中，一个局面的评估往往不像输、平、赢 3 种状态这么简单，在分不出输赢的局面中棋局也有优劣之分。也就是说，要用更细致的方法来刻画局面的优劣，而不是仅仅使用 1，-1,0 三个数字刻画 3 种终了局面。

用估值函数可以为每一个局面的优劣评分。例如甲胜为正无穷，乙胜为负无穷，和局为 0,；其他的情形依据双方剩余棋子的数量及位置评定从负无穷到正无穷之间的具体分数。这样我们可以建立一棵固定深度的搜索树，其叶子节点不必是终了状态，只是固定深度的最深一层的节点，其值由上述函数评出；对于中间节点，如同前面提到的那样，甲方取子节点的最大值，乙方取子节点的最小值。

静态估值函数，用以取代固定深度的搜索。显然，我们无法拥有绝对精确的静态估值函数，否则，只要这个静态估值函数就可以解决所有的棋局了。估值函数给出的只是一个较粗略的评分，在此基础上进行的少量搜索的可靠性，理论上是不如前述的三种状态的博弈树的，但这种方法却是可以实现的。

极大极小策略是考虑双方对弈若干步之后，从可能的步中选一步相对好的步法来走，即在有限的搜索深度范围内进行求解。

定义一个静态估价函数 f，以便对棋局的态势做出优劣评估。

规定：

max 和 min 代表对弈双方；

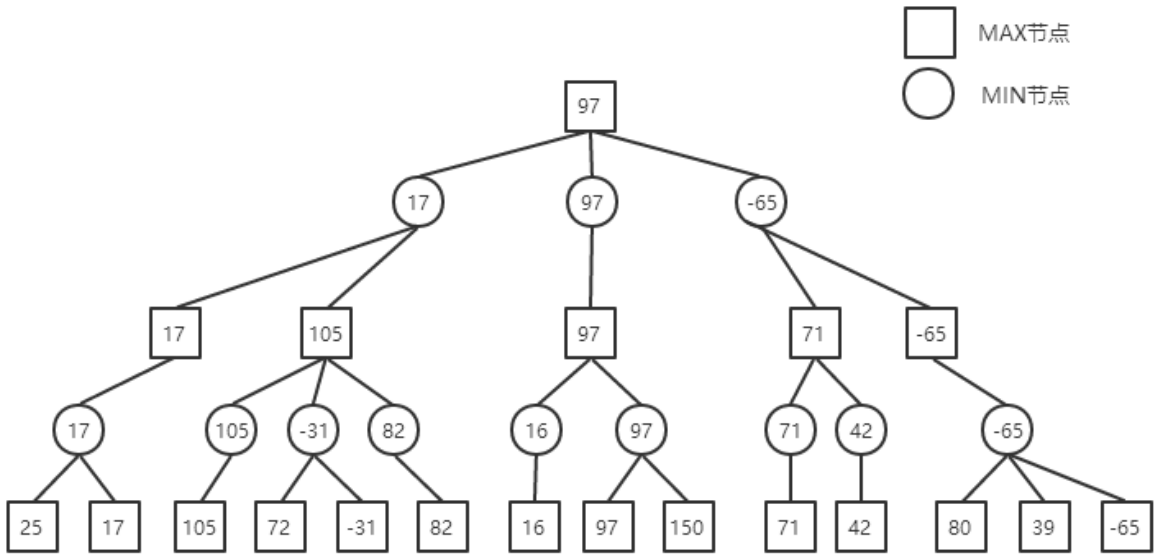
p 代表一个棋局（即一个状态）；

有利于 MAX 的态势，f(p)取正值；

有利于 MIN 的态势，f(p)取负值；

态势均衡，f(p)取零值；

MINMAX 的基本思想：（1）当轮到 MIN 走步时，MAX 应该考虑最坏的情况（即 f(p)取极小值）（2）当轮到 MAX 走步时，MAX 应该考虑最好的情况（即 f(p)取极大值）（3）相应于两位棋手的对抗策略，交替使用（1）和（2）两种方法传递倒推值。



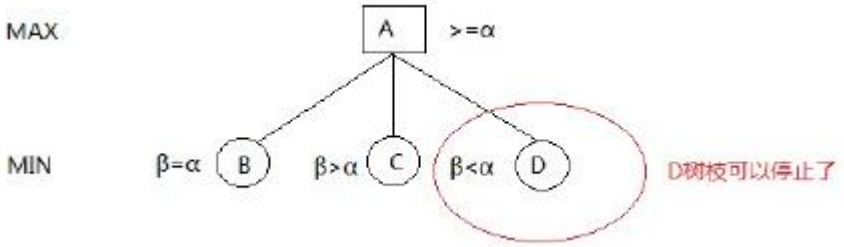
极大极小值算法搜索过程

3.3.3 α β剪枝算法

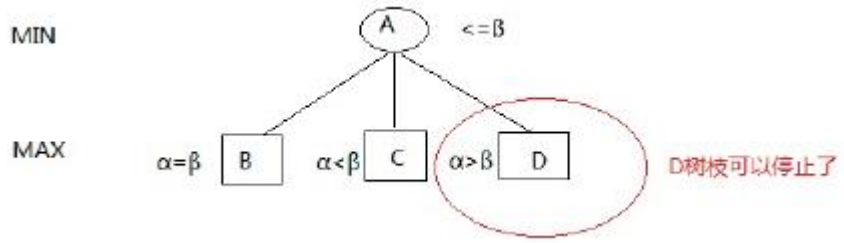
α β剪枝算法是对 Minimax 方法的优化，它们产生的结果是完全相同的，只不过运行效率不一样。其方法运用的前提假设与 Minimax 方法也是一样的。α β剪枝算法的基本思想是：边生成

博弈树边计算评估各节点的倒推值，根据倒推值范围及时减掉那些没有必要再扩展的子节点，从而减少搜索节点的数量，节约了机器开销，提高搜索效率。其内容如下：

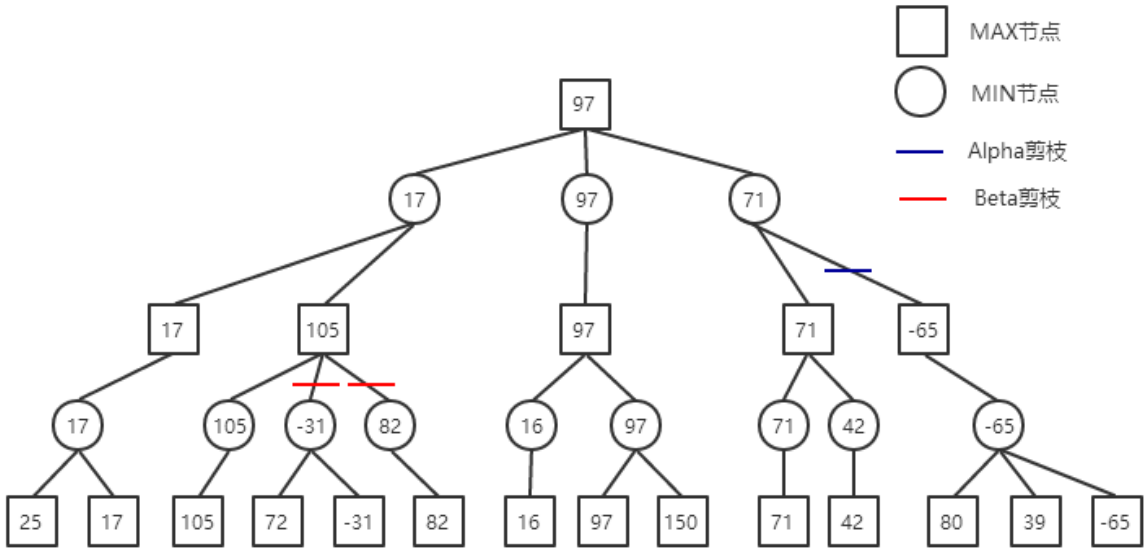
α 剪枝：如果当前节点的某子节点的值不比当前节点的前兄弟节点中的最大值大,则舍弃该子节点和该子节点的所有后兄弟节点。



β 剪枝：如果当前节点的某子节点的值不比当前节点的前兄弟节点中的最小值小,则舍弃该子节点和该子节点的所有后兄弟节点。



α β 剪枝算法搜索过程：



AlphaBeta剪枝算法搜索过程

3.3.4 算法优化

3.3.4.1 局部搜索

设定一个落子范围 MAX_EXTEND 进行限定，在当前棋盘落子位置的最左、最右、最上、最下点的 MAX_EXTEN 格之内且不超过棋盘边界的可落位置进行扩展搜索。这样在棋子较少的时候，搜索结点的数量大大减少。

3.3.4.2 优先值启发

当前节点的子节点的排列顺序对于搜索的速度起着至关重要的影响。如果一开始搜索的子节点更接近于最终返回值，那么再对后面节点进行搜索时剪枝函数会发挥更大的作用，算法效率得到极大提高。

我们可以对下一次的将要搜索的分支子节点计算一个启发值，按照启发值大小顺序进行排序搜索，如果子节点仍然较多，可以设定一个 MAX_SEARCH_STEP 进行限制。

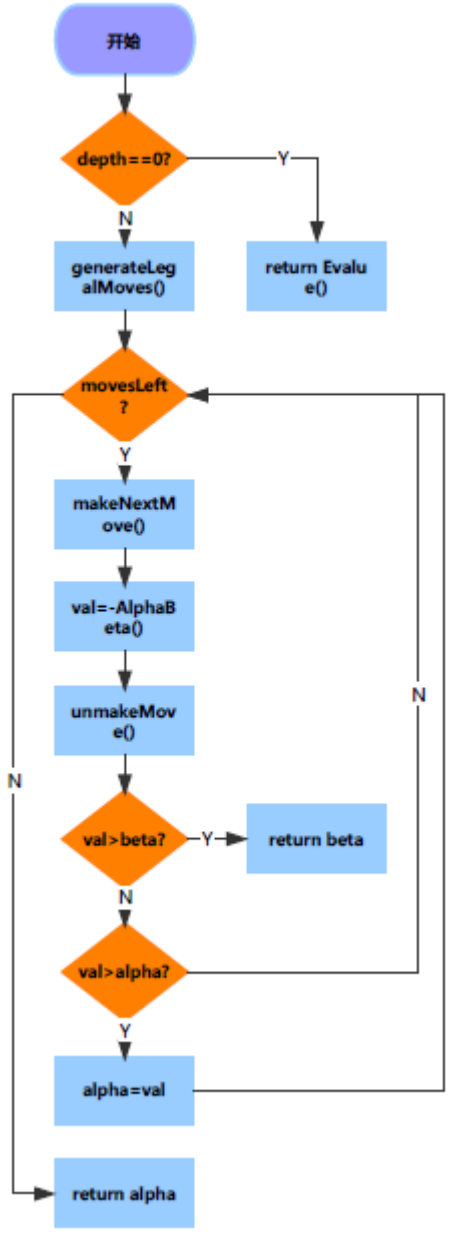
启发值设计为：G= max(G1+ADDSCORE,G2)；也就是该位置下己方棋子的估分加一个微调值和下对方棋子的估分值中这两个值中的较大值。加一个微调是当双方都形成活四棋局时，主动权在自己手中，应该让自己赢。

3.3.5 核心算法伪代码

```
int AlphaBeta(int depth, int alpha, int beta) {
    if (depth == 0) {          //到达一定深度，评估棋局，返回分值
        val = Evaluate();
        return val;
    }
    GenerateLegalMoves();      //生成下一步合法位置

    while (MovesLeft()) {
        MakeNextMove();        //下棋

        //注意 Negamax 风格的调用方式，前面有个负号，后面的参数是-beta 和-alpha
        // Negamax 的含义中 Nega 就是指这里的负号
        val = -AlphaBeta(depth - 1, -beta, -alpha);

        UnmakeMove();          //撤销

        if (val >= beta)       //剪枝情况判断
            return beta;

        if (val > alpha)
            alpha = val;
    }
```

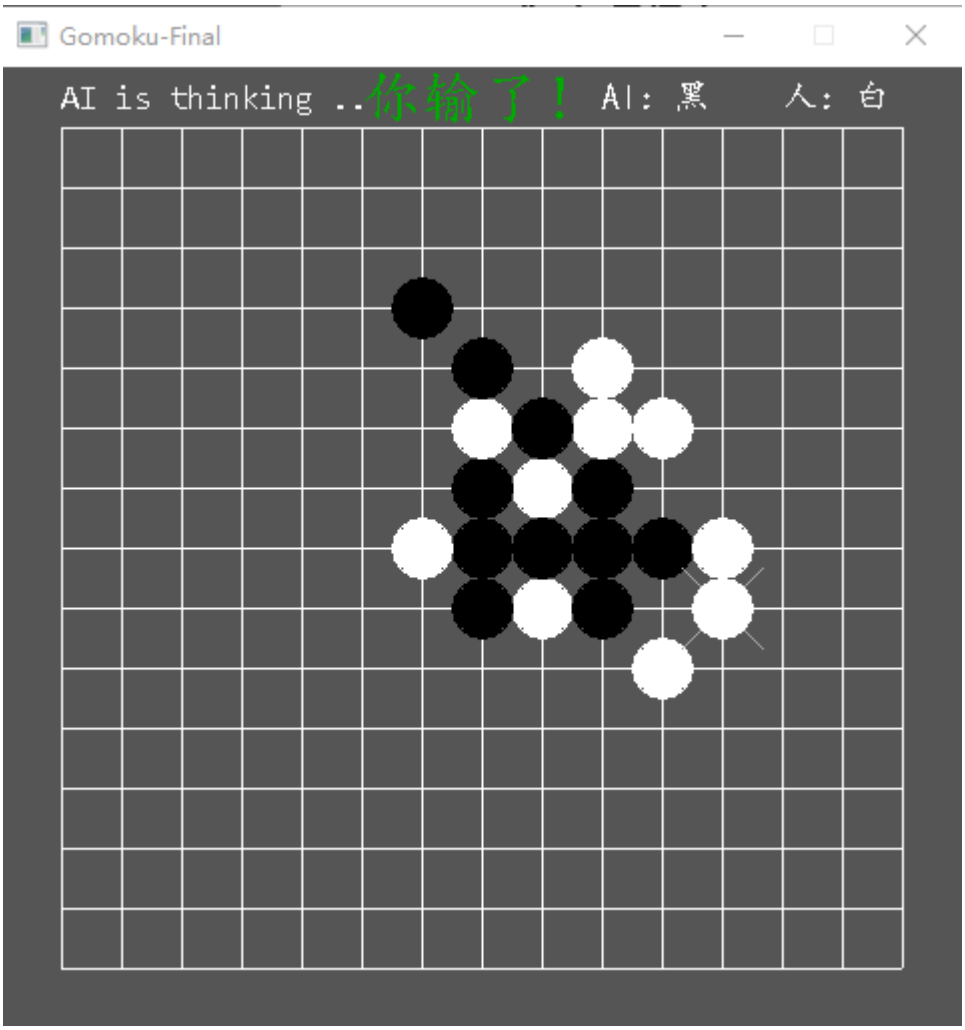
```
        return alpha;     // 此时的 alpha 就是记录了当前结点的所有子结点的最大的负评估值
    }
```

3.3.6 核心算法流程图



4. 结果演示

AI 为黑棋方：

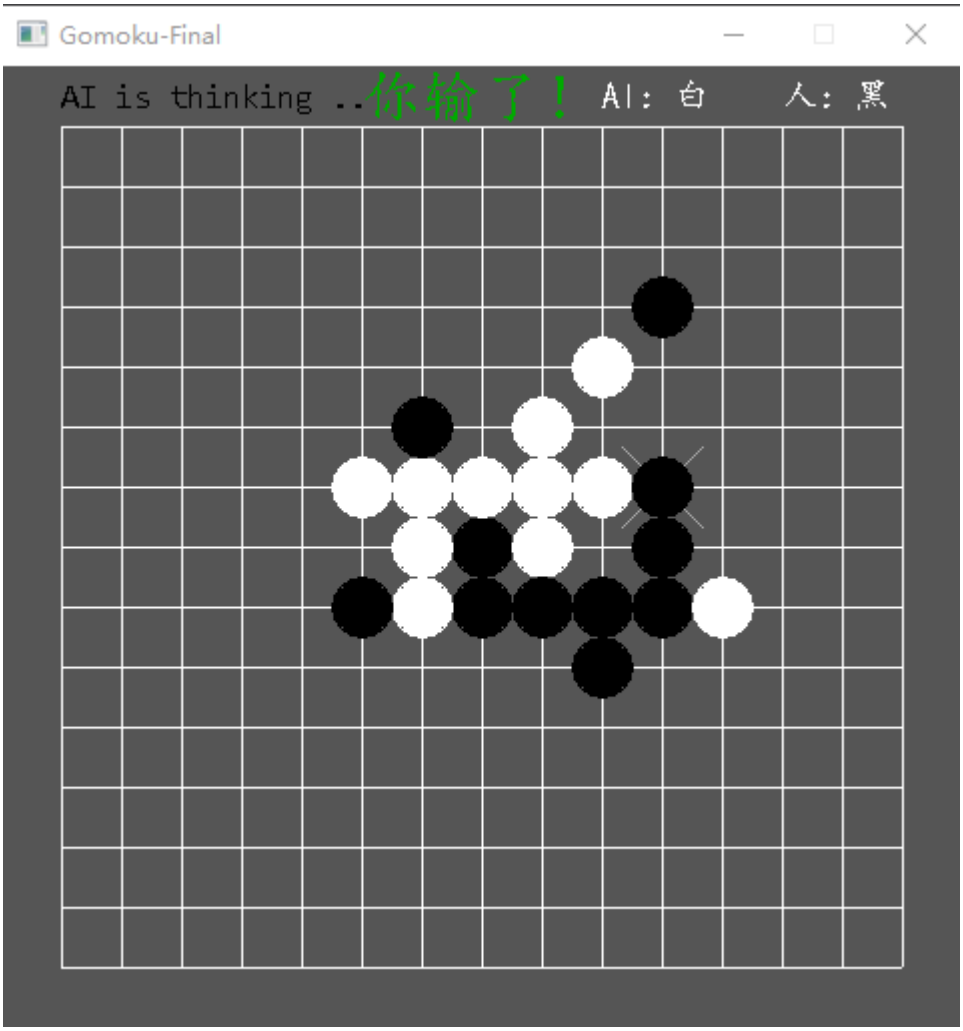


装

订

线

AI 为白棋方：



5. 附件：源程序

工程文件包括：
Global.h
Evaluate.h
Evaluate.cpp
AI.h
AI.cpp
Display.h
Display.cpp
main.cpp

```
//Global.h

#ifndef GLOBAL_H
#define GLOBAL_H

#define LEFT 5              //棋盘左边界
#define TOP 5               //棋盘上边界
#define RIGHT 19            //棋盘右边界
#define DOWN 19             //棋盘下边界
#define MAX_BORDER 25       //棋盘存储大小
#define CHESS_SIZE 15       //棋盘最大网格数

//下棋方定义
typedef enum {
    EMPTY = 0,
    BLACKCHESS = 1,
    WHITECHESS = 2
} SIDE;

//下棋步数定义
class STEP {
public:
     int row;
     int col;
     STEP(int r = 0, int c = 0) :row(r), col(c) {};
};

extern int chess[MAX_BORDER][MAX_BORDER];
void setChess(const STEP&step);
#endif
```

装 订 线

```
//Evaluate.h

#ifndef EVALUATE_H
#define EVALUATE_H

#include"Global.h"
// 棋形分数

#define INF 0x3f3f3f3f
#define VALUE_MAX 10000000      //长连/成五 获胜

#define VALUE_L4  150000        //活四
#define VALUE_W4  7500          //冲四
#define VALUE_D4     20         //死四

#define VALUE_L3  7500          //活三
#define VALUE_W3  2000          //冲三
#define VALUE_D3     10         //死三

#define VALUE_L2  1000          //活二
#define VALUE_W2  200           //冲二
#define VALUE_D2     5          //死二

#define VALUE_L1  100           //活一
#define VALUE_W1  50            //冲一
#define VALUE_D1     0          //死一

#define ADDSCORE  500

//计算返回某个方向的估分
int returnScore();

//对下棋位置的四个方向估分
int getSideScore(const STEP&step, SIDE side);

//对当前状态整个棋盘估分
int evaluateWholeChess(SIDE side);

//计算位置的启发值，优化搜索
int evaluateExtend(const STEP&step, SIDE side);

//判断某一方是否取胜
bool isWin(const STEP step, SIDE side);

#endif
```

```cpp
//Evaluate.cpp

#include<algorithm>
using namespace std;

#include"Evaluate.h"

const int dir[4][2] = { {-1,0},{0,-1},{-1,-1},{-1,1} };       //四个方向   |  -  \  /
const int live[6] = { 0,VALUE_L1,VALUE_L2,VALUE_L3,VALUE_L4,VALUE_MAX };   //活棋分数
const int with[6] = { 0,VALUE_W1,VALUE_W2,VALUE_W3,VALUE_W4,VALUE_W4 };    //冲棋分数
const int death[6] = { 0, VALUE_D1,VALUE_D2,VALUE_D3,VALUE_D4,VALUE_D4 };  //死棋分数

int score;
int cmid, cleft, cright, cnt, leftmid, midright;
int left1, left2, right1, right2;
int flag;

//计算返回某个方向的估分
int returnScore() {
    cmid = min(cmid + 1, 5);
    if (cmid == 5)
        return VALUE_MAX;
    cnt = min(cmid + cleft + cright, 5);
    leftmid = min(cleft + cmid, 5);
    midright = min(cmid + cright, 5);
    flag = left2 << 3 | left1 << 2 | right1 << 1 | right2;

    switch (flag) {
    case 0b0000:
        return live[cnt];
    case 0b0001:
    case 0b1000:
        return max(live[leftmid], with[cnt]);
    case 0b0010:
    case 0b0011:
        return with[leftmid];
    case 0b0100:
    case 0b1100:
        return with[midright];
    case 0b0101:
    case 0b1101:
        return midright >= 4 ? with[midright] : death[midright];
    case 0b1010:
    case 0b1011:
        return leftmid >= 4 ? with[leftmid] : death[leftmid];
    case 0b0110:
    case 0b0111:
    case 0b1110:
    case 0b1111:
        return death[cmid];
    case 0b1001:
        return cnt >= 4 ? max(live[cmid], with[cnt]) : death[cnt];
    }
    return 0;
```

```
}

//对下棋位置的四个方向估分
int getSideScore(const STEP&step, SIDE side) {
    score = 0;
    int row, col, i;
    for (i = 0; i < 4; ++i) {
        cmid = cleft = cright = 0;
        left1 = left2 = right1 = right2 = 0;

        for (row = step.row + dir[i][0], col = step.col + dir[i][1]; chess[row][col] == side;
++cmid) {
            row += dir[i][0];
            col += dir[i][1];
        }
        left1 = chess[row][col] != EMPTY;
        if (left1) {
            for (row += dir[i][0], col += dir[i][1]; chess[row][col] == side; ++cleft) {
                row += dir[i][0];
                col += dir[i][1];
            }
            left2 = chess[row][col] != EMPTY;
        }

        for (row = step.row - dir[i][0], col = step.col - dir[i][1]; chess[row][col] == side;
++cmid) {
            row -= dir[i][0];
            col -= dir[i][1];
        }
        right1 = chess[row][col] != EMPTY;
        if (right1) {
            for (row -= dir[i][0], col -= dir[i][1]; chess[row][col] == side; ++cright) {
                row -= dir[i][0];
                col -= dir[i][1];
            }
            right2 = chess[row][col] != EMPTY;
        }
        score += returnScore();
    }
    return score;
}

//对当前状态整个棋盘估分
int evaluateWholeChess(SIDE side) {
    int returnValue = 0;
    int v1 = 0, v2 = 0;
    for (int i = TOP; i <= DOWN; ++i)
        for (int j = LEFT; j <= RIGHT; ++j)
            if (chess[i][j] == side)
                v1 += getSideScore(STEP(i, j), side);
            else if (chess[i][j] != EMPTY) {
                v2 += getSideScore(STEP(i, j), (SIDE)chess[i][j]);
            }
            return (int)(v1 - v2);
```

装 订 线

```
}

//计算位置的启发值，优化搜索
int evaluateExtend(const STEP&step, SIDE side) {
    return max(getSideScore(step, BLACKCHESS) + (side == BLACKCHESS ? ADDSCORE : 0),
getSideScore(step, WHITECHESS) + (side == WHITECHESS ? ADDSCORE : 0));
}

//判断某一方是否取胜
bool isWin(const STEP step, SIDE side) {
    return getSideScore(step, side) >= VALUE_MAX;
}
```

装

订

线

```
//AI.h

#ifndef AI_H
#define AI_H

#include"Global.h"
#include<vector>

#define MAX_DEPTH 4             //最大搜索深度
#define MAX_EXTEND 5            //每一步棋最大扩展范围
#define MAX_SEARCH_STEP 10      //每一步棋最大搜索步数

//AI下棋接口
void AISearch(SIDE aiSide, STEP&step, STEP &nextStep);

//α-β剪枝算法
int alphaBetaSearch(SIDE side, STEP step, int depth, int alpha, int beta);

//扩展下一步棋的位置
void extendNextStep(SIDE side, const STEP &curStep, vector<pair<STEP, int>>&nextStep);

#endif
```

装 订 线

```
//AI.cpp

#include<vector>
#include<algorithm>
using namespace std;

#include "Global.h"
#include "Evaluate.h"
#include "AI.h"

STEP bestStep;
SIDE AIside;

//AI下棋接口
void AISearch(SIDE aiSide, STEP&step, STEP &nextStep) {
    AIside = aiSide;
    alphaBetaSearch(aiSide, step, 0, -INF, INF);
    nextStep = bestStep;
}

//α-β剪枝算法
int alphaBetaSearch(SIDE side, STEP step, int depth, int alpha, int beta) {
    if (depth >= MAX_DEPTH)
        return evaluateWholeChess(side);

    SIDE otherSide = side == WHITECHESS ? BLACKCHESS : WHITECHESS;
    if (isWin(step, otherSide))
        return -VALUE_MAX;

    vector<pair<STEP, int>>nextStep;
    extendNextStep(side, step, nextStep);
    int trySize = min((int)nextStep.size(), MAX_SEARCH_STEP);

    int value, flag = 0;
    for (int i = 0; i < trySize; ++i) {
        pair<STEP, int>&nspair = nextStep[i];
        STEP &nstep = nextStep[i].first;
        chess[nstep.row][nstep.col] = side;

        if (nspair.second >= VALUE_L4) {
            if (!depth) {
                bestStep = nstep;
                chess[nstep.row][nstep.col] = EMPTY;
                return alpha;
            }
            flag = 1;
        }
        if (!flag || nspair.second >= VALUE_L4)
            value = -alphaBetaSearch(otherSide, nstep, depth + 1, -beta, -alpha);

        chess[nstep.row][nstep.col] = EMPTY;
        if (value >= beta)
            return value;
        if (value > alpha) {
```

```
                alpha = value;
                if (!depth)
                    bestStep = nstep;
                if (alpha >= VALUE_MAX)
                    return alpha;
            }
        }

        return alpha;
    }

    //定义排序顺序
    bool myPairCmp(const pair<STEP, int>&p1, const pair<STEP, int>&p2) {
        return p1.second > p2.second;
    }

    //扩展下一步棋的位置
    void extendNextStep(SIDE side, const STEP &curStep, vector<pair<STEP, int>>&nextStep) {
        int left = max(curStep.col - MAX_EXTEND, LEFT);
        int right = min(curStep.col + MAX_EXTEND, RIGHT);
        int top = max(curStep.row - MAX_EXTEND, TOP);
        int down = min(curStep.row + MAX_EXTEND, DOWN);
        for (int i = top; i <= down; ++i)
            for (int j = left; j <= right; ++j)
                if (chess[i][j] == EMPTY) {
                    STEP temStep(i, j);
                    nextStep.push_back({ temStep,evaluateExtend(temStep,side) });
                }
        sort(nextStep.begin(), nextStep.end(), myPairCmp);
    }
```

装 订 线

```
//Display.h

#ifndef DISPLAY_H
#define DISPLAY_H

#define GRID_SIZE 30
#define WIDTH GRID_SIZE*(CHESS_SIZE+1)
#define HEIGHT GRID_SIZE*(CHESS_SIZE+1)
#define MOUSE_P1 10
#define MOUSE_P2 20

#define MOUSE_COLOR LIGHTGRAY
#define BACKGROUND_COLOR DARKGRAY

#define BLACKCHESS_COLOR 0
#define WHITECHESS_COLOR 0xFFFFFF

//初始化
void init(SIDE humanSide);

//画出棋盘网格
void drawChessBoard();

//显示各方棋色提示
void showSide(SIDE humanSide);

//指定位置画出一颗棋子
void showOneChess(int x, int y, int color);

//显示会话
void showDialog(char *dialog);

//显示博弈结果
void showResult(bool humanWin);

//显示鼠标位置
void showMouse(int x, int y, int color);

//移动鼠标时更新界面
void moveMouse(int x, int y, int nx, int ny);

//人类下棋前端交互
void humanAction(STEP &step, SIDE side);

#endif
```

装 订 线

```
//Display.cpp

#include "Global.h"
#include "display.h"
#include <graphics.h>

//画出棋盘网格
void drawChessBoard() {
    for (int i = 1; i <= CHESS_SIZE; ++i)
        line(GRID_SIZE, i*GRID_SIZE, CHESS_SIZE*GRID_SIZE, i*GRID_SIZE);
    for (int i = 1; i <= CHESS_SIZE; ++i)
        line(i*GRID_SIZE, GRID_SIZE, i*GRID_SIZE, CHESS_SIZE*GRID_SIZE);
}

//显示各方棋色提示
void showSide(SIDE humanSide) {
    settextstyle(18, 0, _T("楷体"));
    outtextxy(WIDTH - GRID_SIZE * 6, 6, _T(humanSide == BLACKCHESS ? "AI：白    人：黑" : "AI：
黑    人：白"));
}

//初始化
void init(SIDE humanSide) {
    initgraph(WIDTH, HEIGHT);
    setbkcolor(BACKGROUND_COLOR);
    cleardevice();
    FlushMouseMsgBuffer();
    drawChessBoard();
    showSide(humanSide);
    showSide(humanSide);
}

//指定位置画出一颗棋子
void showOneChess(int x, int y, int color) {
    x -= LEFT - 1, y -= TOP - 1;
    x *= GRID_SIZE, y *= GRID_SIZE;
    setcolor(color);
    setfillcolor(color);
    setfillstyle(BS_SOLID);
    fillcircle(x, y, GRID_SIZE / 2);
}

//显示会话
void showDialog(char *dialog) {
    settextstyle(20, 0, _T("Consolas"));
    outtextxy(30, 5, _T(dialog));
}

//显示博弈结果
void showResult(bool humanWin) {
    settextstyle(30, 0, _T("楷体"));
    if (humanWin) {
        setcolor(RED);
        outtextxy(WIDTH / 2 - GRID_SIZE * 2, 0, _T("你赢了！"));
```

```
    }
    else {
        setcolor(GREEN);
        outtextxy(WIDTH / 2 - GRID_SIZE * 2, 0, _T("你输了！"));
    }
}

//显示鼠标位置
void showMouse(int x, int y, int color) {
    static const int p1[4][2] =
{ { MOUSE_P1 , MOUSE_P1 }, { MOUSE_P1, -MOUSE_P1 }, { -MOUSE_P1, MOUSE_P1 }, { -MOUSE_P1, -MOUSE_P1 } };
    static const int p2[4][2] =
{ { MOUSE_P2 , MOUSE_P2 }, { MOUSE_P2, -MOUSE_P2 }, { -MOUSE_P2, MOUSE_P2 }, { -MOUSE_P2, -MOUSE_P2 } };
    setlinecolor(color);
    x *= GRID_SIZE, y *= GRID_SIZE;
    for (int i = 0; i < 4; ++i)
        line(x + p1[i][0], y + p1[i][1], x + p2[i][0], y + p2[i][1]);
}

//移动鼠标时更新界面
void moveMouse(int x, int y, int nx, int ny) {
    showMouse(x, y, BACKGROUND_COLOR);
    showMouse(nx, ny, MOUSE_COLOR);
}

//人类下棋前端交互
void humanAction(STEP &step, SIDE side) {
    MOUSEMSG m;
    int nowx = -1, nowy = -1;
    while (1) {
        m = GetMouseMsg();
        int windowx = (m.x + (GRID_SIZE >> 1)) / GRID_SIZE;
        int windowy = (m.y + (GRID_SIZE >> 1)) / GRID_SIZE;
        int chessx = windowx + LEFT - 1;
        int chessy = windowy + TOP - 1;
        if (chessx >= LEFT && chessx <= RIGHT&&chessy >= TOP && chessy <= DOWN) {
            switch (m.uMsg)
            {
            case WM_MOUSEMOVE:      //移动
                if (nowx != windowx || nowy != windowy)
                    moveMouse(nowx, nowy, windowx, windowy);
                break;
            case WM_LBUTTONDOWN:  //按下
                if (chess[chessy][chessx] != EMPTY)
                    break;
                showOneChess(chessx, chessy, side == BLACKCHESS ? BLACKCHESS_COLOR :
WHITECHESS_COLOR);
                step = { chessy , chessx };
                return;
            }
            nowx = windowx, nowy = windowy;
        }
    }
}
```

```cpp
//main.cpp

#include<cstring>
#include<graphics.h>
using namespace std;

#include "Global.h"
#include "Evaluate.h"
#include "AI.h"
#include "Display.h"

int chess[MAX_BORDER][MAX_BORDER];

void setChess(const STEP&step, SIDE side) {
    chess[step.row][step.col] = side;
}

#define compSide 1    //0：AI为黑  1：AI为白

void playChess() {
    STEP step, nextStep;
    SIDE humanSide = compSide ? BLACKCHESS : WHITECHESS;
    SIDE computerSide = compSide ? WHITECHESS : BLACKCHESS;
    init(humanSide);

    if (computerSide == BLACKCHESS) {
        nextStep = { 7 + TOP,7 + LEFT };
        setChess(nextStep, computerSide);
       showOneChess(nextStep.col, nextStep.row, computerSide == BLACKCHESS ? BLACKCHESS_COLOR :
WHITECHESS_COLOR);
    }

    while (1) {
        showDialog("Turn to you ...    ");
        humanAction(step, humanSide);
        setChess(step, humanSide);
        if (isWin(step, humanSide)) {
            showResult(true);
            break;
        }

        showDialog("AI is thinking ...");
        AISearch(computerSide, step, nextStep);
        setChess(nextStep, computerSide);
       showOneChess(nextStep.col, nextStep.row, computerSide == BLACKCHESS ? BLACKCHESS_COLOR :
WHITECHESS_COLOR);
        if (isWin(nextStep, computerSide)) {
            showResult(false);
            break;
        }
    }
    system("pause");
}
```

```
int main() {
    playChess();
    return 0;
}
```

装

订

线