Alpha-Beta 剪枝
求解五子棋 AI 问题

装 订 线

学　　号 [redacted]

姓　　名 [redacted]

专　　业 计算机科学与技术专业

授课老师 [redacted]

目 录

装 订 线

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

1.2.1 内容

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。

1.2.2 要求

1.要求初始界面显示15*15的空白棋盘，电脑执白棋，人执黑棋，界面置有重新开始，悔棋等操作。

2.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。

3.撰写实验报告，提交源代码（进行注释）、实验报告、汇报PPT。

2 实验方案设计

2.1 总体设计思路与总体框架

2.1.1 总体设计思路

本程序的总体设计思路是：

首先设计 seearchAlphaBeta 类实现对五子棋人机博弈问题的核心算法的求解。主要执行 AI 下棋的内部执行逻辑，即 alpha-beta 剪枝算法的实现。外部可调用的为一个 alphabetaGetBack 函数，程序输入对应的 15*15 的局面信息，便可返回下一步 AI 方的行棋位置。整个剪枝运算的过程被封装在该类中，因此，在 main 函数中使用时，只需要调用该类即可。

再设计一个 showUI 类实现对五子棋人机交互界面的实现。由于本函数需要根据用户指令切换不同的页面形式，因此对外函数接口针对过程中所需实现的功能均进行了对接。主要包含接口为：refresh 重置界面情况函数；uiLoad 加载背景界面信息函数；uiGetBack 玩家单步下棋执行函数；addWhite 棋盘添加 AI 方棋子函数；winShow 玩家胜利显示函数；loseShow 玩家失败显示函数；showThingking 切换背景图函数以及 showNormal 切换背景图函数。在 main 函数中，可以直接根据 earchAlphaBeta 类计算结果情况调用该类中的执行函数，并根据不同的情况在界面实现玩家下棋、AI 下棋、悔棋、重新开始等操作。

程序设计安排再 4 个 cpp 文件中实现，分别为：

minimaxAlphaBeta_head.h：存储整个项目类、结构体、函数、宏定义的声明，作为项目的头文件使用。

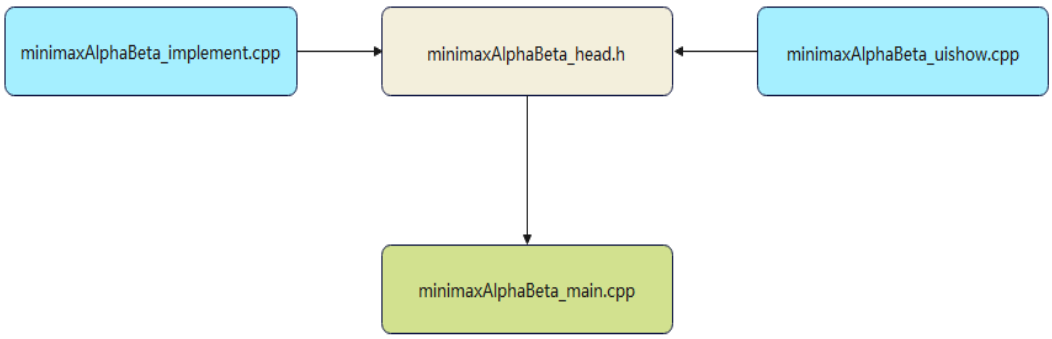
minimaxAlphaBeta_implement.cpp：内含 seearchAlphaBeta 类的具体实现函数，实现五子棋游戏的内部核心算法的求解。

minimaxAlphaBeta_uishow.cpp：内含 showUI 类的具体实现函数，实现五子棋游戏的可视化界面输入输出。还包含部分整个游戏所需工具函数的具体实现。

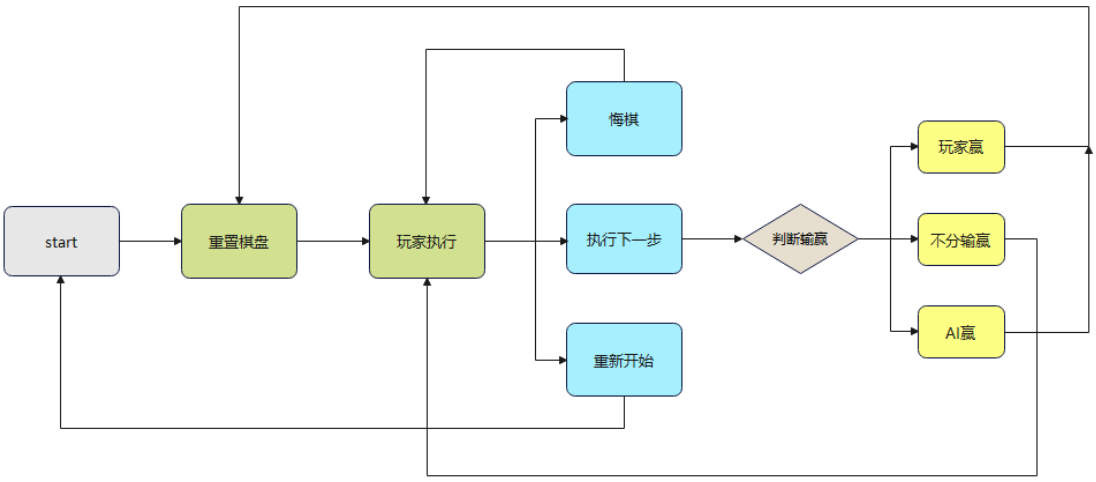
minimaxAlphaBeta_main.cpp：程序主函数所在地，调用各个类进行对五子棋问题的求解。

2.1.2 总体架构

程序实现的总体架构：



根据此总体程序实现过程，以期望实现的程序运行流程如下：



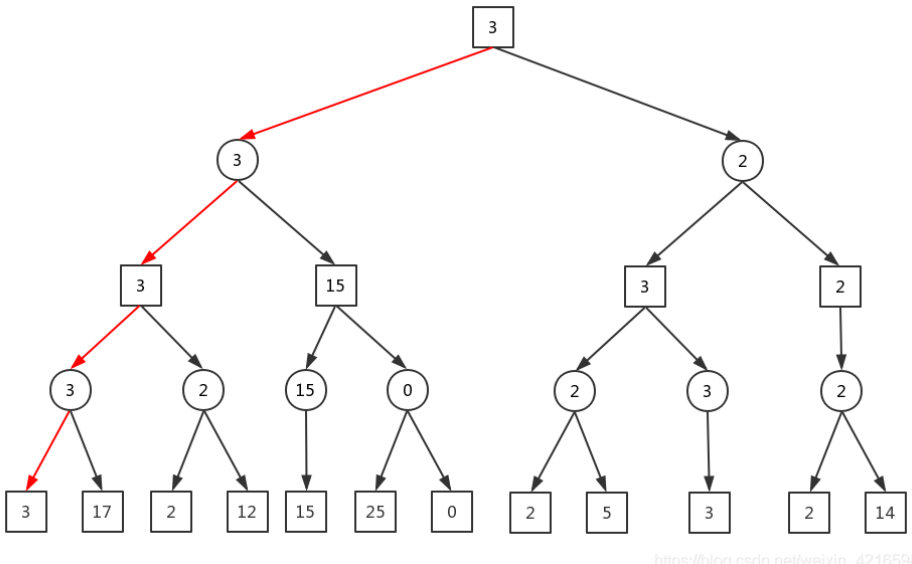
2.2 核心算法及基本原理

2.2.1 核心算法—MINMAX 算法

Minimax 算法（亦称 MinMax or MM）又名极小化极大算法，是一种找出失败的最大可能性中的最小值的算法。

Minimax 算法常用于棋类等由两方较量的游戏和程序。该算法是一个零总和算法，即一方要在可选的选项中选择将其优势最大化的选择，另一方则选择令对手优势最小化的方法。而开始的时候总和为 0。很多棋类游戏可以采取此算法，例如井字棋（tic-tac-toe）、五子棋（gobang）

根据 MINMAX 算法的解释，我们可以构建搜索树为：



但是，根据上图可以看出，如果依照 MINMAX 算法进行决策的话，需要的计算量是随着向后的步数的增加而呈指数级增长。但是，这些状态中其实包含了很多不必要存在的状态，所以我们可以进行剪枝。

2.2.2 核心算法 — Alpha-Beta 剪枝算法

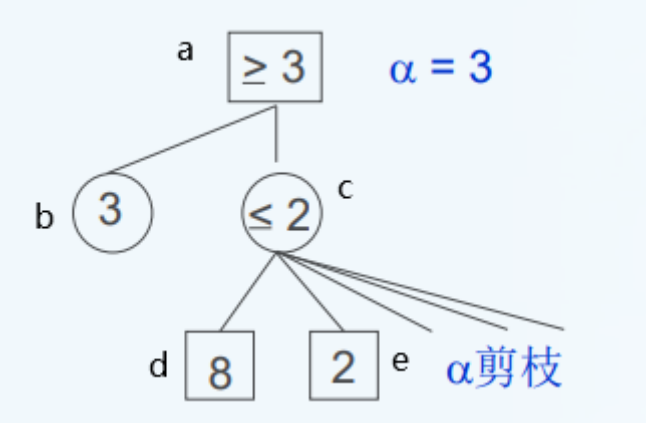
Alpha-beta(α－β)剪枝的名称来自计算过程中传递的两个边界，这些边界基于已经看到的搜索树部分来限制可能的解决方案集。其中，Alpha(α)表示目前所有可能解中的最大下界，Beta(β)表示目前所有可能解中的最小上界。

因此，如果搜索树上的一个节点被考虑作为最优解的路上的节点（或者说是这个节点被认为是有必要进行搜索的节点），那么它一定满足以下条件（N 是当前结点的评估值）：α ≤ N ≤ β

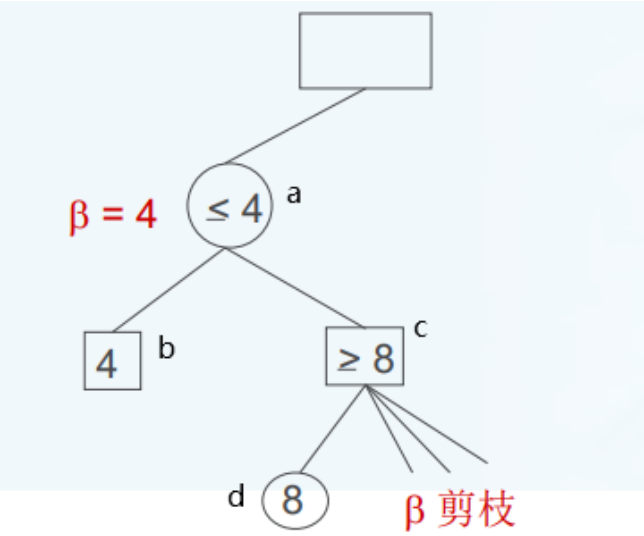
在我们进行求解的过程中，α 和 β 会逐渐逼近。如果对于某一个节点，出现了α ＞ β>的情况，那么，说明这个点一定不会产生最优解了，所以，我们就不再对其进行扩展（也就是不再生成子节点），这样就完成了对博弈树的剪枝。

因此，在搜索最优解的过程中，每当需要搜索同级结点过程中，可以根据 Alpha-beta 进行剪枝，消除其余无需遍历的结点。

Alpha 剪枝：



Beta 剪枝：



2.2.3 算法基本原理

本文关键 Alpha-Beta 剪枝算法基本过程可以简述为：

采用递归方法进行实现：

Alpha-Beta 剪枝算法方法为 alphabetaAlgorithm

首先从第 0 层(根结点)向下遍历

Ⅰ.若遍历置叶子结点，则直接返回局面的全局评估值

Ⅱ.若层数为奇数，则进入 MIN 结点判别

①调用增益评估函数，选择 n 个后继结点进行遍历

②依次遍历该结点的 n 个后继结点，并根据结点信息修改局面信息

③对每个后继调用方法 alphabetaAlgorithm，返回 value 值

④取 value 和结点 β 值的较小值作为该结点新 β 值

⑤若该结点的 α 值大于等于该结点 β，则无需继续遍历子结点（跳出循环）--> α 剪枝

Ⅲ.若层数为偶数，则进入 MAX 结点处理判别

①调用增益评估函数，选择 n 个后继结点进行遍历

②依次遍历该结点的 n 个后继结点，并根据结点信息修改局面信息

③对每个后继调用方法 alphabetaAlgorithm，返回 value 值

④取 value 和结点 α 值的较大值作为该结点新 α 值

⑤若该结点的 α 值大于等于该结点 β，则无需继续遍历子结点（跳出循环）--> β 剪枝

注意：a. n 可以根据自己需求计算指定

b. 第 0 层为偶数层

根据以上算法便可以较为轻易地根据 α－β 剪枝算法进行求解

2.2.4 评估函数算法设计

Ⅰ. 全面评估

针对不同的局面给出不同的评估值，有利于使得五子棋 AI 更加智能，有效选择更有利的步骤。因此，根据查阅资料，对五子棋棋局的评估常采用六元组的形式进行判别，即对六个点位的信息进行思考，再次基础上发现五子棋局面总共分成如下情况(以黑棋为例):

①连五/长连：即存在五个或者六个连续的黑棋，即黑棋赢

②活四：有两个位置可以形成连五

③冲四：有一个位置可以形成连五

④活三：走一步可以形成活四

⑤眠三：走一步可以形成冲四

⑥活二：走一步可以形成活三

⑦眠二：走一步可以形成眠三

⑧活一：走一步可以形成活二

因此根据黑棋和白棋在以上不同情况，给予不同的分数。

特别需要注意的是：

a. 由于当 AI 需要评判时，是模拟该空位白棋走了一步，因此，紧接着轮到黑棋行走。故，黑白棋出现同级别情况的时候，黑棋得分应当大于白棋。

b.一个局面过程中黑棋白棋均存在，因此应当对二者同时进行评估，给予白棋正分，黑棋负

分，分数越高即对于 AI 来说越有利。给出评分表如下为：

棋型说明	权重		棋型说明	权重
白连五	1000000		黑连五	-10000000
白活四	50000		黑活四	-100000
白冲四	400		黑冲四	-100000
白活三	400		黑活三	-8000
白眠三	20		黑眠三	-50
白活二	20		黑活二	-50
白眠二	1		黑眠二	-3
白活一	1		黑活一	-3

Ⅱ．增益评估

根据 alphabetaAlgorithm 可以看出，若每层对于所有可能的空结点进行评估，这显然无法再一定时间内得出结果，因为递归子结点个数是呈指数级增长。因此，我们需要对向下搜索的结点进行评估，以确定某些结点继续向下遍历。此时，便需要我们的增益评估函数。

增益评估函数是指对每个可以选择落点的位置，进行评估，给出落子前后的得分增益。因此，根据此思想，可以避免对原棋局的重复计算得分。只需要对该位置上-下;左-右;左上-右下;左下-右上四个方向的棋子进行滑窗六元组评估落子前后结果即可。

2.3 模块设计

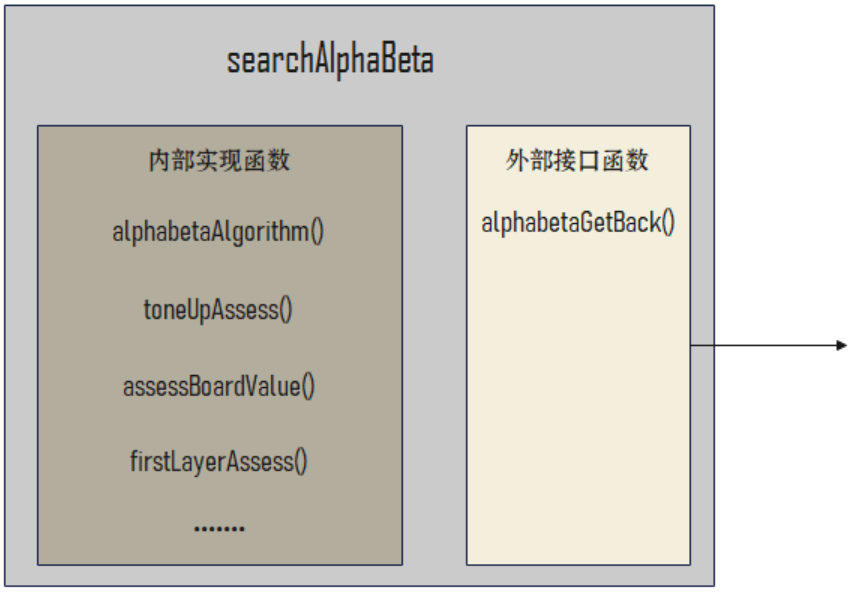
①searchAlphaBeta 模块

```
//@function:alpha-beta剪枝搜索类
class searchAlphaBeta {
protected:
    NODE head;
    char board[BOARDWIDTH + BOARDADD][BOARDLENGHT + BOARDADD] = { '\0' };
protected:

    int getScoreFromString(string& get);                                                                                    //得到一个string串的的嗯
    int toneUpAssessIn(char(*chessBoard)[BOARDLENGHT + BOARDADD], NODESCORE& site);                                         //增益函数内部评估
    void toneUpAssess(char(*chessBoard)[BOARDLENGHT + BOARDADD], NODESCORE& site, char sort);                               //增益评估
    bool searchThreeMust(char(*chessBoard)[BOARDLENGHT + BOARDADD], NODESCORE& get, char chessSort);                        //白棋活三特判
    bool searchFiveMust(char(*chessBoard)[BOARDLENGHT + BOARDADD], NODESCORE& get, char chessSort);                         //五子连珠特判
    bool fiveMust(string& get, char chessSort);                                                                             //五子连珠特判in
    bool firstLayerAssess(char(*chessBoard)[BOARDLENGHT + BOARDADD], int& feeBack);                                         //第一层首先评估
    bool chessAround(int row, int col, char(*chess)[BOARDLENGHT + BOARDADD]);                                               //判断NONECHESS周围是否存在CEHSS
    bool findBorder(char(*chessBoard)[BOARDLENGHT + BOARDADD], BORDER& bor);                                                //找到搜索边界
    int eachGet(string& extract);                                                                                           //评价函数中对不同的情况返回不同得分
    int assessBoardValue(char(*chessBoard)[BOARDLENGHT + BOARDADD]);                                                        //全局评估函数
    bool seekBestPoint(char(*chessBoard)[BOARDLENGHT + BOARDADD], char chessClass, vector<NODESCORE>& result);              //找到较优的几个点进行针对性拓展
    int alphabetaAlgorithm(char(*chessBoard)[BOARDLENGHT + BOARDADD], int depth, int alpha, int beta);                      //进行alpha-beta剪枝搜索

public:
    searchAlphaBeta(char(*chessBoard)[BOARDLENGHT + BOARDADD]);                                                             //构造棋盘函数
    int alphabetaGetBack(void);                                                                                             //执行函数
};
```

该模块的整体模块设计为：



②showUI 模块

```
//@function : 图像UI界面显示类
class showUI {
protected:

    //界面布局结构体定义
    SURFACE boardImage;
    SURFACE regretButton;
    SURFACE restartButton;
    SURFACE pictureSite;
    SURFACE winloseShow;

    //界面图片调用定义
    IMAGE backGround;
    IMAGE chessBoard;
    IMAGE chessBlack;
    IMAGE chessWhite;
    IMAGE chessWhitePre;

    IMAGE regretImage;
    IMAGE regretMouseImage;
    IMAGE restartImage;
    IMAGE restartMouseImage;
    IMAGE thinkingImage;
    IMAGE userImage;

    IMAGE winImage;
    IMAGE loseImage;

    bool findXY(int x, int y, int& boardX, int& boardY);          //找到xy所对应在棋盘中的位置
    void endShow(void);                                           //结束后只能按下重新开始按钮
public:
    showUI(void);                                                 //构造函数
    bool refresh(char(*chessBoard)[BOARDLENGHT + BOARDADD]);      //重置界面情况
    bool uiLoad(void);                                            //加载背景界面函数
    int uiGetBack(char(*chessBoard)[BOARDLENGHT + BOARDADD]);     //单步下棋执行函数
    bool addWhite(int site);                                      //在棋盘上添加白棋
    void winShow(void);                                           //成功显示
    void loseShow(void);                                          //失败显示
    void showThinking(void);                                      //切换图片
    void showNormal(void);                                        //切换图片
};
```

该模块的整理模块设计为：

装

2.4 其他创新内容或优化算法

1. 限定单结点向下遍历个数

根据递归向下剪枝可知，若每层不存在剪枝运算，将搜索结点个数将呈现指数级增长态势，因此随着个数的增加，我们将无法在一定时间内得到结果，也就丧失了人机对战的意味。故显然需要进行不完美的实时决策，对非终止结点打分。此时需要增益评估函数，对非终止结点进行完善的评估，选择出更为得分更高，更为优秀的结点继续向下遍历。

订

```
#define EACHSEARCH      20      // 定义每层的探索个数

//进行赋值返回
for (int i = 0; (i < (int)Total.size() && i + 1 <= EACHSEARCH); i++)
    result.push_back(Total[i]);
```

线

2. 空位落子策略

每次落子时，棋盘上的空白理论上均可落子，但根据查阅资料可知，仅在已有结点周围八个方向延申至多三层的结点对于本局有利，因此，设定 chessAround 函数，判定该空位周围是否在一定范围内存在棋子，即判断位置的拓展可行性。

```
#define DIRECTIONINNER 8    //8个方向上的节点进行拓展
//#define DIRECTIONOUTER 16

//@function:判断周围是否存在棋子
bool searchAlphaBeta::chessAround(int row, int col , char (* chess)[BOARDLENGHT + BOARDADD])
{
#ifdef DIRECTIONINNER
    for (int i = row - 1; i <= row + 1; i++) {
        for (int j = col - 1; j <= col + 1; j++) {
            if ((i >= 1 && i <= BOARDWIDTH) && (j >= 1 && j <= BOARDLENGHT) && (i != row || j != col)) {
                if(chess[i][j]!='*')
                    return true;  //存在棋子
            }
        }
    }
    return false;                 //不存在棋子
#endif
}
```

3. 优先队列增大剪枝概率

我们了解到 alpha-beta 剪枝将极大的优化效率，剪掉越多的结点可以获得更高效的搜索效率。对于 alpha-beta 剪枝有一个最优的剪枝状态，即优先搜索到极大或者较小得分值。

对于 MIN 结点，下一步走黑棋，首先搜索到的为得分最少的结点。当出现 alpha 剪枝时，便可在最大程度上剪去不必要结点。

对于 MAX 结点，下一步走白棋，首先搜索到的为得分最优的结点，当出现 beta 剪枝时，便可在最大程度上剪去不必要结点。

我们在 seekBestPoint 方法中实现这一过程：

```
//进行降序排序
if (chessClass == BLACKCHESS)        //黑棋升序-->易于剪枝
    sort(Total.begin(), Total.end(), compareAscend);
else if (chessClass == WHITECHESS)  //白棋降序-->易于剪枝
    sort(Total.begin(), Total.end(), compareDecend);

//进行赋值返回
for (int i = 0; (i < (int)Total.size() && i + 1 <= EACHSEARCH); i++)
    result.push_back(Total[i]);

return true;
```

4. 第一层初判

对于搜索，有些选择在第一层便可以得出，该选择可以为必走制胜步骤或必走抵抗步骤，而无需继续向下搜索。因此，在原有搜索的基础上，增加一个首层判断过程，将白棋能制胜的必走棋或抵抗黑棋的必走棋或白棋形成活四的必胜棋首先判断，若存在这样的必走棋，则无需再向下遍历，直接走该位置即可。

```
//@function:找到较优的探索结点（局部搜索+静态启发+剪枝）
bool searchAlphaBeta::seekBestPoint(char(*chessBoard)[BOARDLENGHT + BOARDADD], char chessClass, vector<NODESCORE>& result){ ... }

//@function:第一层白棋活三特判
bool searchAlphaBeta::searchThreeMust(char(*chessBoard)[BOARDLENGHT + BOARDADD], NODESCORE& get, char chessSort){ ... }

//@function:第一层五子连珠特判滑窗特判(五元组)
bool searchAlphaBeta::fiveMust(string& get, char chessSort){ ... }

//@function:第一层五子连珠特判
bool searchAlphaBeta::searchFiveMust(char(*chessBoard)[BOARDLENGHT + BOARDADD], NODESCORE& site, char chessSort){ ... }

//@function:第一层特判,如果存在必下棋，则返回必下棋局位置-->针对白棋视角
bool searchAlphaBeta::firstLayerAssess(char(*chessBoard)[BOARDLENGHT + BOARDADD], int& feeBack){ ... }
```

3 实验过程

3.1 环境说明

操作系统：Window
开发语言：C++
编译平台：Visual Studio 2022
核心库 ：

```
<vector>          // vector数组
<iostream>        // 标准库
<algorithm>       // 算法库
<stack>           // 堆栈
<graphics.h>      // EasyX库
<conio.h>         // EasyX 库
```

3.2 源代码文件清单

3.2.1 源代码清单

minimaxAlphaBeta_head.h：存储程序核心库声明、宏定义、结构体声明、函数声明等等程序，实现各个 cpp 文件之间的连接。

minimaxAlphaBeta_implement.cpp：alpha-beta 剪枝内部执行函数源文件，用于实现 α－β 剪枝算法运算。

minimaxAlphaBeta_uishow.cpp：五子棋界面显示源文件，用于实现人机间的五子棋对局交互。

minimaxAlphaBeta_main.cpp：五子棋程序主函数源文件，作为程序的主函数存放地，实现整体程序的执行。

3.2.2 主要函数清单

Ⅰ. searchAlphaBeta 类函数

```
int getScoreFromString(string& get);
//得到一个string串的评估得分
int toneUpAssessIn(char(*chessBoard)[BOARDLENGHT + BOARDADD], NODESCORE& site);
//增益函数内部评估
void toneUpAssess(char(*chessBoard)[BOARDLENGHT + BOARDADD], NODESCORE& site, char sort);
//增益评估
bool searchThreeMust(char(*chessBoard)[BOARDLENGHT + BOARDADD], NODESCORE& get, char chessSort);
//白棋活三特判
bool searchFiveMust(char(*chessBoard)[BOARDLENGHT + BOARDADD], NODESCORE& get, char chessSort);
//五子连珠特判
bool fiveMust(string& get, char chessSort);
//五子连珠特判内部执行函数
bool firstLayerAssess(char(*chessBoard)[BOARDLENGHT + BOARDADD], int& feeBack);
//第一层首先评估
bool chessAround(int row, int col, char(*chess)[BOARDLENGHT + BOARDADD]);
//判断NONECHESS周围是否存在CEHSS
bool findBorder(char(*chessBoard)[BOARDLENGHT + BOARDADD], BORDER& bor);
//找到搜索边界
int eachGet(string& extract);
//评价函数中对不同的情况返回不同得分
```

```
int assessBoardValue(char(*chessBoard)[BOARDLENGHT + BOARDADD]);
//全局评估函数
bool seekBestPoint(char(*chessBoard)[BOARDLENGHT + BOARDADD], char chessClass,
vector<NODESCORE>& result);
//找到较优的几个点进行针对性拓展
int alphabetaAlgorithm(char(*chessBoard)[BOARDLENGHT + BOARDADD], int depth, int alpha, int
beta);
//进行alpha-beta剪枝搜索
searchAlphaBeta(char(*chessBoard)[BOARDLENGHT + BOARDADD]);
//构造棋盘函数
int alphabetaGetBack(void);
 //执行函数
```

Ⅱ. showUI 类函数

```
bool findXY(int x, int y, int& boardX, int& boardY);          //找到xy所对应在棋盘中的位置
void endShow(void);                                            //结束后只能按下重新开始按钮
showUI(void);                                                  //构造函数
bool refresh(char(*chessBoard)[BOARDLENGHT + BOARDADD]);       //重置界面情况
bool uiLoad(void);                                             //加载背景界面函数
int uiGetBack(char(*chessBoard)[BOARDLENGHT + BOARDADD]);      //单步下棋执行函数
bool addWhite(int site);                                       //在棋盘上添加白棋
void winShow(void);                                            //成功显示
void loseShow(void);                                           //失败显示
void showThinking(void);                                       //切换图片
void showNormal(void);                                         //切换图片
```

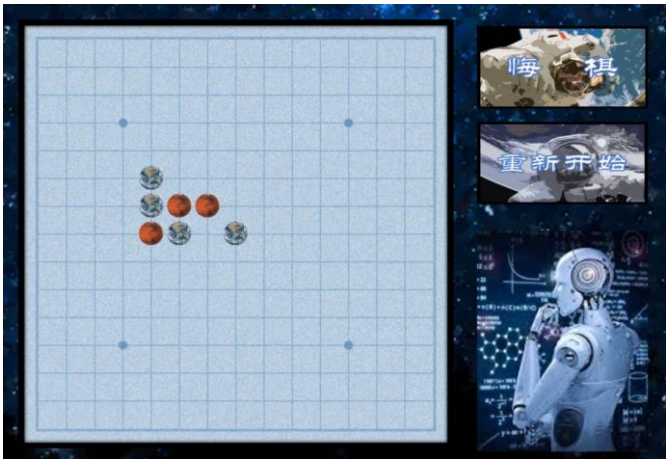
Ⅲ. 其余工具函数

```
int judgeWinner(char(*chessBoard)[BOARDLENGHT + BOARDADD]);        //判断是否成功
void refreshBoard(char(*chessBoard)[BOARDLENGHT + BOARDADD]);      //重置棋盘
```

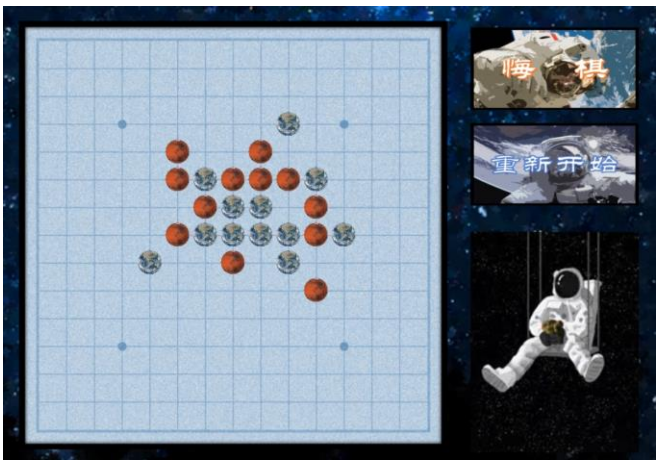
3.3 实验结果展示

3.3.1 界面结果展示

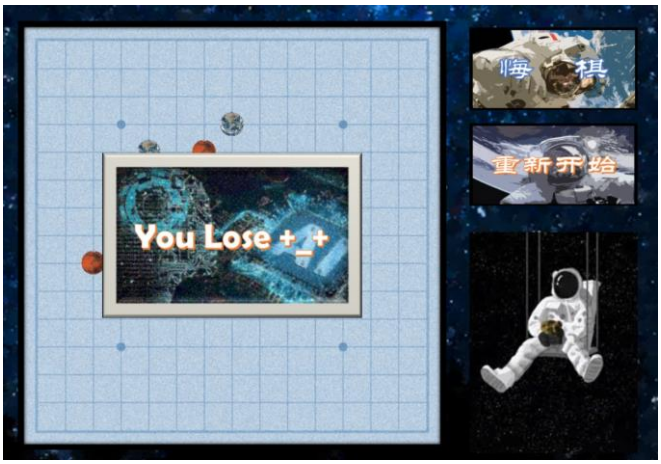
①对战界面:



②悔棋功能:



③重新开始功能:



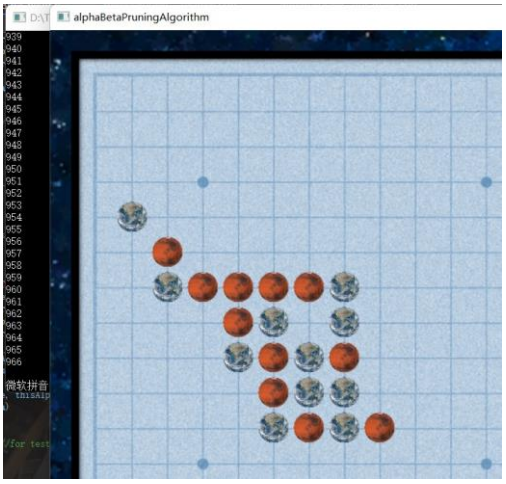
3.3.2 运行结果展示

本程序能自行控制搜索深度和每层搜索至多个数，因此在运行结果的测试上，能较为方便和快捷的进行测试，因此实验结果如下：

Ⅰ．优化前后测试

使用优先队列扩大剪枝效果后，原四层搜索深度、限定搜索至多搜索个数为 10 个：程序原本生成叶子结点在 10*10*10*10 个，可以剪去近九成的搜索结点，秩序遍历大概 1000 个结点，便可以得到最终结果，程序效果得到极大程度优化。

由此可见，优化后，能使得α－β剪枝发挥更大作用，效果不错。相信在更大数据量的情况下，将会产生更大的增益。



Ⅱ. 多次调试搜索深度和搜索个数后，得到效率表如下

深度	每层至多个数	平均执行时长
3	20	2.5s
3	10	1s
4	10	6s
4	5	2.6s
6	4	7.8s
6	2	1.3s

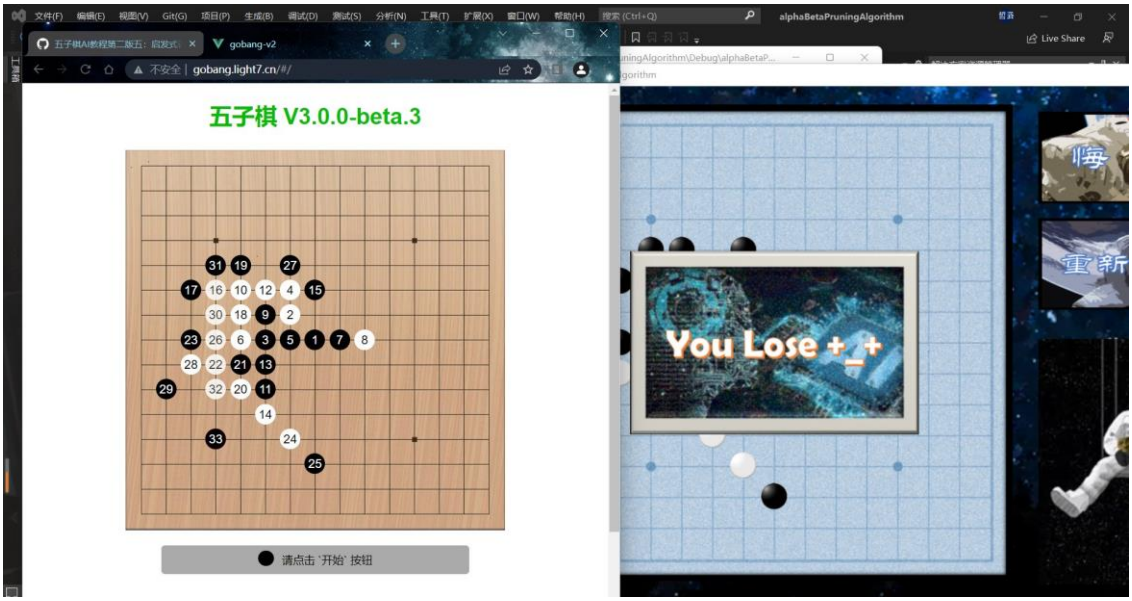
根据表格可以看出，每层个数将极大影响计算个数，并且在个数不变，深度增加的情况下，搜索时间将呈现指数级增长。

但是经过测试发现，随着减少每层个数，由于采用逐层贪心，选择最优结点的方案，对 AI 智能的影响不大。并且若深度越深，AI 的智慧增长情况显然能够弥补个数减少的影响。因此，其实深度为 6 层，每层个数为 2 便可以营造高速高智慧的 AI，对局体验极好，暂时也无法赢过 AI。

III. 对局结果

首先人机对战，当超过大于等于第四层时，作为人以及很难下过 AI，它的考虑以及评估已经十分完善，响应速度也不错。

当第三层时，便可战胜 github 上收藏最多的五子棋 AI，并且是 github 五子棋作为先手的情况下：



在对局体验上，十分流畅，适当减少搜索个数，加深搜索深度，能构建高速，高智的五子棋AI。

3.4 实验结论

根据实验结果可以看出，本次五子棋 AI 的实现效果极好，实现了高速反馈、高智选择的较高要求。根据分析，影响 AI 智慧的成败在于评估函数的选择和构建；影响 AI 速度的决定性因素在于 α－β 剪枝算法的实现程度。因此，可以看出，本次构建的评估函数是较为完备的；并且设计的剪枝优化算法也是能相当高效的剪去不必要的计算过程，提升整体效率。

总体而言，本次实验基本达到预期结果，成果较好。

4 总结

4.1 实验中存在的问题及解决方案

整个实验过程首先就遇到了较多问题。刚开始对于α-β剪枝算法不了解，试图构建 minmax 搜索树然后再尝试剪枝搜索，显然是南辕北辙。在程序实现过程中需要先对实验原理有较好的掌握，才能进一步进行程序设计。

除此之外，思考最久的就是五子棋 AI 的优化上了。最初实现全体可落子位置的搜索时，程序执行效率极低，常常需要等待 1-2min 才能呈现结果，这也促使我进一步优化程序。在深入理解α-β剪枝算法后，阅读较多文献后，设计出的优化算法能较好的实现人机对局，且反应快，智商高。

4.2 心得体会

经过本次实验，我对于α-β剪枝算法的实现过程有了更加清晰的认识。评估函数的选择极大程度决定构造 AI 的智慧水平 ，因此选择较为完备合理的局面评估函数，将在一定程度上，提升 AI 的智慧水平。

并且，在实际对局过程中，不可能实现所有情况的搜索，因此选择较好的不完美的实时策略，能极大程度提升程序效率。同时也要避免对 AI 思考深度和智慧水平的不利影响。

经过本次实验，对 C++地 EasyX 库有了更加清晰的了解，告别了 cmd 的小黑框，能实现较好的可视化效果。

4.3 后续改进方向

后续可以在评估函数的选择上提升，查阅更多文献，选择更完备的评估函数，进一步提升 AI 的智慧。

还可以设计更好的优化算法，在进一步提升 AI 思考深度的基础上，保证较高的执行效率。

4.4 总结

经过本次实验，对于 AI 有了更加深入的认识。原来人工智能离我们那么近，运用一定的算法便可以实现自己的 AI。这也激励我在人工智能领域进一步学习，多阅读相关文献，提升个人思考深度。

参考文献

[1] 赵美勇,宋思睿.博弈论算法在AI中的应用[J].计算机产品与流通,2019(09):278.

[2] 郑培铭，何丽．基于计算机博弈的五子棋AI设计[J].电脑知识与技术,2016,12(33):80-81+90.DOI:10.14004/j.cnki.ckt.2016.4580.

[3] 陈树彬，和昱旻，原菊梅．五子棋落子算法的研究[J].电脑与信息技术,2021,29(05):49-51+94.DOI:10.19414/j.cnki.1005-1228.2021.05.014.

[4] 李昊．五子棋人机博弈算法优化研究与实现[D].大连海事大学,2020.DOI:10.26989/d.cnki.gdlhu.2020.000523.

[5] 沈雪雁. 基于蒙特卡洛树与神经网络的五子棋算法的设计与实现[D].沈阳化工大学,2021.DOI:10.27905/d.cnki.gsghy.2021.000057.

[6] 刘阳. 基于人工智能的五子棋专家系统研究和设计[D].电子科技大学,2015.

[7] 林华．基于Self-Play的五子棋智能博弈机器人[D].浙江大学,2019.DOI:10.27461/d.cnki.gzjdx.2019.000894.

装 订 线