**同济大学计算机系**

**人工智能原理课程设计报告**

****

**成 员 1 1751918 温 鑫**

**成 员 2 1650917 杨 晨**

**成 员 3 1752132 王 森**

**成 员 4 1753309 毕晓栋**

**专 业 计算机科学与技术**

**授课老师 赵才荣**

目录

[1 实验概述 5](#_Toc274813223)

[1.1 实验要求 5](#_Toc1722850460)

[1.2 实验内容 5](#_Toc1881266657)

[2 实验方案设计 5](#_Toc263238483)

[2.1 总体设计思路与总体架构 5](#_Toc1728716082)

[2.2 核心算法及基本原理 6](#_Toc1070869288)

[2.2.1基于置换表的主要变例搜索算法 6](#_Toc11256893)

[2.2.2 静态搜索算法 10](#_Toc1307809591)

[2.2.3 带检验的空着裁剪算法 10](#_Toc2143863683)

[2.3 模块设计 11](#_Toc1697462768)

[2.3.1 UCCI交互模块 11](#_Toc1933229636)

[2.3.2 着法生成模块 13](#_Toc1232002802)

[2.3.3 局面估值模块 15](#_Toc268662113)

[2.3.4 着法搜索模块 15](#_Toc186952519)

[2.3.5 开局库模块 16](#_Toc854342618)

[2.4 其他创新内容或优化算法 17](#_Toc255871871)

[2.4.1 对将军的判断 17](#_Toc1645271609)

[2.4.2 走法排序策略 17](#_Toc345598992)

[2.4.3 动态计算局面估值 18](#_Toc2070440395)

[3 实验过程 18](#_Toc564070714)

[3.1 环境说明 18](#_Toc1410868809)

[3.1.1 操作系统 18](#_Toc1152799908)

[3.1.2 开发语言 19](#_Toc942037327)

[3.1.3 开发环境具体要求 19](#_Toc1239385775)

[3.1.4 核心使用库 19](#_Toc1633051063)

[3.2 源代码文件清单 19](#_Toc907787873)

[3.2.1 search.cpp：迭代加深搜索过程 19](#_Toc2103927946)

[3.2.2 search.h ：迭代加深搜索过程的一些定义和工具函数 21](#_Toc2003153525)

[3.2.3 book.h ：开局库着法的结构体的定义和一些工具函数 21](#_Toc473013041)

[3.2.4 book.cpp ：开局库着法的函数和方法的实现 21](#_Toc613732825)

[3.2.5 ucci.h ：ucci通讯的结构体和相关的ucci的操作 22](#_Toc1944756127)

[3.2.6 ucci.cpp ：ucci操作的实现 22](#_Toc747826264)

[3.2.7 movesort.h ：着法的效力的排序的定义和实现 23](#_Toc189099637)

[3.2.8 position.h ：棋盘的结构体和一些棋盘的有关操作 24](#_Toc1678539137)

[3.2.9 position.cpp ：棋盘结构体的一些操作的实现 27](#_Toc1011064748)

[3.2.10 hash.h ：置换表结构和信息的结构体和一些宏定义和工具函数 32](#_Toc1917815719)

[3.2.11 starfish.cpp ：象棋博弈中的主程序 32](#_Toc601924777)

[3.2.12 tools.h ：其他工具函数 33](#_Toc1022321641)

[3.3 实验结果展示 34](#_Toc1078141663)

[3.4 实验结论 34](#_Toc598304813)

[4 总结 35](#_Toc572300761)

[4.1 实验中存在的问题及解决方案 35](#_Toc863887651)

[4.2 心得体会 35](#_Toc1830307615)

[4.3 后续改进方向 36](#_Toc840962874)

[4.4 总结 37](#_Toc1050840170)

[5 成员分工与自评 38](#_Toc537166585)

[参考文献 39](#_Toc1096834745)

# 1 实验概述

## 1.1 实验要求

中国象棋是一项十分受欢迎的棋类运动。中国象棋是起源于中国的一种棋戏，属于二人[对抗性游戏](https://baike.baidu.com/item/%E5%AF%B9%E6%8A%97%E6%80%A7%E6%B8%B8%E6%88%8F/1959177" \t "/home/frank/Documents\x/_blank)的一种，在中国有着[悠久](https://baike.baidu.com/item/%E6%82%A0%E4%B9%85/9310358" \t "/home/frank/Documents\x/_blank)的历史。由于用具简单，趣味性强，成为流行极为广泛的棋艺活动。 [1]

中国象棋使用方形格状棋盘及红黑二色圆形棋子进行对弈，棋盘上有十条横线、九条竖线共分成90个交叉点；中国象棋的棋子共有32个，每种颜色16个棋子，分为7个兵种，摆放和活动在交叉点上。双方交替行棋，先把对方的将（帅）“将死”的一方获胜（因为[擒贼先擒王](https://baike.baidu.com/item/%E6%93%92%E8%B4%BC%E5%85%88%E6%93%92%E7%8E%8B/227413" \t "/home/frank/Documents\\x/_blank)）。已有几千年历史、充满东方智慧的中国象棋在中国的群众中基础远远超过围棋，一直是普及最广的棋类项目。

本次实验要求设计一个中国象棋棋类AI，可以使用UCCI[2]接口进行通讯作战。

## 1.2 实验内容

本次实验，我们采取模块化的方式：UCCI协议交互模块、着法生成模块、着法排序模块、局面估值模块、着法搜索模块、开局库模块。每个模块进行内部调试，模块之间定义标准化接口。结构清晰、便于查找错误。本次实验最核心的算法就是搜索算法，使用暴力搜索加剪枝的方式进行着法的生成和价值计算。然后进行着法的价值的排序，然后给出一个相对最优的着法。

我们采用象棋巫师[3]这个支持UCCI协议的程序作为测试效果的工具。

实验最后的结果表明，使用以上叙述的流程可以实现效果较好的中国象棋的棋类  
AI程序。

# 2 实验方案设计

## 2.1 总体设计思路与总体架构

主要包含5个模块，内容为：

(1)UCCI交互模块(ucci.cpp\ucci.h)：UCCI命令解释模块。

(2)着法生成模块(position.cpp\position.h)：主要描述着法和局面的数据结构及功能， 以及根据当前局面生成所有可行着法。

(3)着法排序模块(movesort.h):对已生成的着法进行排序。

(3)局面估值模块(search.cpp)：对着法进行估值。

(4)着法搜索模块(search.cpp\search.h)：按照一定的规则对着法进行搜索。

(5)开局库模块(book.cpp\book.h)：根­­据开局库返回最佳着法。

除该五个模块之外，还有一些工具函数(buffer.h\tools.h\hash.h)

总体架构流程图

ucci模块处理指令

查找开局库book

生成所有着法

搜索及局面评估

go

false

最优着法bestmove

true

结果输出

输入局面信息,生成棋局

position

开始

loop

首先通过ucci模块接受外部指令，position指令对局面进行描述，构建棋盘，go指令根据现有棋盘开始进行最佳着法的寻找，在着法寻找过程中，先判断当前棋局是否在开局库中，如果在，则直接返回最佳着法，否则生成所有着法，在一定时限内进行搜索，最终返回搜索到的最佳的着法bestmove，将结果进行输出。

## 2.2 核心算法及基本原理

### [2.2.1基于置换表的主要变例搜索](#直接计算生成函数法测定稳定常数的原理)算法

1. 极大极小搜索算法

Alpha-Beta剪枝算法是基于极大极小搜索算法的。极大极小搜索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走法来走，在有限的搜索范围内进行求解，可以理解为规定一个有限的搜索深度。

为此要定义一个静态估计函数f，以便对棋局的势态做出优劣的估计，这个函数可根据棋局的优劣势态的特征来定义。这里规定，MAX代表程序方，MIN代表对手方，P代表一个棋局（即一个状态）。有利于MAX的势态，f(p)取正值，有利于MIN的势态，f(p)去负值，势态均衡，f(p)取零。极大极小搜索的基本思想是：

1. 当轮到MIN走步的节点时，MAX应考虑最坏的情况（因此，f(p)取极小值）。
2. 当轮到MAX走步的节点时，MAX应考虑最好的情况（因此，f(p)取极大值）。
3. 当评价往回倒退的时候，相应于两位棋手的对抗策略，不同层上交替地使用（1）、（2）两种方法向上传递倒推值。所以这种搜索方法称为极大极小过程。实际上，这种算法是假定在模拟过程中双方都走出最好的一步，对MAX方来说，MIN方的最好一步是最坏的情况，MAX在不断地最大化自己的利益。
4. Alpha-Beta剪枝算法

极大极小搜索策略在一些棋盘AI中非常常见，但是它有个致命的弱点，就是非常暴力地搜索导致效率不高，特别是当讲搜索的深度加大时会有明显的延迟，Alpha-Beta在此基础上进行了优化。事实上，MIN、MAX不断的倒推过程中是存在着联系的，当它们满足某种关系时后续的搜索是多余的。Alpha-Beta剪枝算法把生成后继和倒推值估计结合起来，及时减掉一些无用分支，以此来提高算法的效率。

定义极大层的下界为Alpha，极小层的上界为Beta，Alpha-Beta剪枝规则描述如下：

1. Alpha剪枝。若任一极小值层结点的Beta值不大于它任一前驱极大值层结点的Alpha值，即Alpha(前驱层) >= Beta(后继层)，则可终止该极小值层中这个MIN结点以下的搜索过程。这个MIN结点最终的倒推值就确定为这个Beta值。
2. Beta剪枝。若任一极大值层结点的Alpha值不小于它任一前驱极小值层结点的Beta值，即Alpha(后继层) >= Beta(前驱层)，则可以终止该极大值层中这个MAX结点以下的搜索过程，这个MAX结点最终倒推值就确定为这个Alpha值。
3. 主要变例搜索算法

主要变例搜索(PVS, Principal Variation Search)是提高“Alpha-Beta”算法效率的一种方法。

当alpha<value<beta时，此估值一定是准确值；此搜索节点就称为PV（Principal Variation，主要变例）节点。当value≤alpha时，此估值将不是准确值，而只是估值上限；此搜索节点就称为alpha节点。当value≥beta时，此估值也不是准确值，而只是估值下限；这一搜索节点就称为beta节点。

由此可看出，Alpha-Beta剪枝算法重点关注的是(alpha, beta)区间，它也称为搜索窗口。相对于alpha节点和beta节点而言，程序搜索PV节点的过程是比较费时的。如果把搜索窗口的宽度减小，发生剪枝的可能性就会加大，程序搜索的时间也会短些。特别是当窗口宽度减为零，即搜索窗口为(t - 1, t)时，这就成了零宽窗口（也称为极小窗口）。这种零宽窗口的搜索结果只有二种，要么是估值在窗口之下的alpha节点，要么是估值在窗口之上的beta节点。利用零宽窗口搜索一分为二的特点，可以很方便地判别估值的取值范围。因为零宽窗口搜索不会出现PV节点，这种判别过程非常高效，所以它广泛应用于各种优化的Alpha-Beta剪枝算法中。

主要变例搜索（Principal Variation Search，简称PVS）算法就是一种采用零宽窗口技术、针对PV节点进行优化的剪枝算法，也称为极小窗口搜索（Minimal Window Search）算法。Tony Marsland和Murray Campbell（1982）首先提出了PVS算法[4]，其后Alexander Reinefeld（1983）也提出了相类似的NegaScout算法[5]，并证明其正确性[6]。

下面我们来看看PVS算法是怎样进行优化的。在前面所介绍的Alpha-Beta剪枝算法中，每下完一步棋，我们总是采用以下形式进行递归搜索：

value = -alpha\_beta(-beta, -alpha);

而对于PVS算法，它首先假设之前已经找到最佳估值（其值为alpha），然后应用零宽窗口搜索，以便快速判别此假设的正确性：

value = -alpha\_beta(-alpha-1, -alpha);

如果零宽窗口搜索的结果是value≤alpha，那么表明之前的假设是正确的，即这步棋不会有更好的估值，程序将直接舍弃这步棋。在这种情况下，由于采用了零宽窗口搜索，搜索时间将大大缩短。

但是如果零宽窗口搜索的结果表明之前的假设是错误的，那么还需要重新进行一次常规窗口的搜索。在这种情况下，PVS算法实际上多进行了一次零宽窗口搜索。但由于零宽窗口搜索耗时相对较少，而出现前一种情况的概率一般较大，额外进行一次零宽窗口搜索所花费的代价还是值得的。

算法伪代码如下：

int AlphaBeta(int depth, int alpha, int beta) {

　BOOL fFoundPv = FALSE;

　if (depth == 0) {

　　return Evaluate();

　}

　GenerateLegalMoves();

　while (MovesLeft()) {

　　MakeNextMove();

　　if (fFoundPv) {

　　　val = -AlphaBeta(depth - 1, -alpha - 1, -alpha);

　　　if ((val > alpha) && (val < beta)) { // 检查失败

　　　　val = -AlphaBeta(depth - 1, -beta, -alpha);

　　　}

　　} else

　　　val = -AlphaBeta(depth - 1, -beta, -alpha);

　　}

　　UnmakeMove();

　　if (val >= beta) {

　　　return beta;

　　}

　　if (val > alpha) {

　　　alpha = val;

　　　fFoundPv = TRUE;

　　}

　}

　return alpha;

}

由于PVS算法的高效性是建立在已找到最佳估值的假设之上，因此在尚未找到有效节点时，只需进行常规窗口搜索。

一般来说，PVS算法的总体速度会比常规Alpha-Beta剪枝算法快一些。如果将PVS算法与棋步排序、Hash表等优化技术相结合，算法的高效性将得到更充分的体现。

1. 基于置换表的启发式优化

中国象棋的搜索树可以用图来表示，而置换结点可以引向以前搜索过的子树上。置换表可以用来检测这种情况，从而避免重复劳动。

以局面的 Zobrist Key % HASH\_SIZE 作为索引值。每个置换表项存储的内容为：

（1）深度

（2）标志

（3）分值

（4）最佳走法

（5）Zobrist Lock 校验码

通过ProbeHash和RecordHash两个函数维护置换表。

RecordHash采用深度优先的替换策略，在判断深度后，将Hash表项中的每个值填上即可。

ProbeHash 对置换表信息的利用：

第一种情况：检查局面所对应的置换表项，如果Zobrist Lock校验码匹配，那么我们就认为命中了；

第二种情况：是否能直接利用置换表中的结果，取决于两个因素：1. 深度是否达到要求，2. 非PV节点还需要考虑边界。

第二种情况是最好的(完全利用)，ProbeHash返回一个非 -MATE\_VALUE 的值，这样就能不对该节点进行展开了。

如果仅仅符合第一种情况，那么该置换表项的信息仍旧是有意义的——它的最佳走法给了我们一定的启发(部分利用)。

1. 迭代加深和时间控制

迭代加深具有一石多鸟的功效，目前最明显的供效是充分发挥历史表的作用——浅一层搜索结束后，历史表中积累了大量非常宝贵的数据，这将大幅度减少深一层搜索的时间。

在迭代加深的基础上实现时间控制，这将是非常简单的：

for (i = 1; i < MAX\_DEPTH; i ++) {

　AlphaBeta(-INFINITY, INFINITY, i);

　if (超过最短搜索时间) {

　　break;

　}

}

　　当然，我们还可以加入其他结束迭代加深的条件，例如当程序算出了杀棋(分值接近INFINITY或-INFINITY)时，就没有必要进行更深的搜索了。

### 2.2.2 静态搜索算法

中国象棋中有些走法有固定的应对方式，而Alpha-Beta搜索不会特别针对这种情况的。你把深度参数传递给函数，当深度到达零就做完了，从而产生“水平线效应”。一个应对的方法称为“静态搜索”(Quiescent Search)。当Alpha-Beta用尽深度后，通过调用静态搜索来代替调用“Evaluate()”。这个函数也对局面作评价，只是避免了在明显有对策的情况下看错局势。

简而言之，静态搜索就是应对可能的动态局面的搜索。进入静态搜索时，要考虑两种情况，一是不被将军的情况，首先尝试不走是否能够截断，然后搜索所有吃子的走法(可以按照MVV/LVA排序)；二是被将军的情况，这时就必须生成所有的走法了(可以按照历史表排序)。注意静态搜索中没有传递“深度”这个参数。正因为如此，如果找到好的吃子，或者有一系列连续的强制性吃子的着法，那么搜索可能会非常深。

### 2.2.3 带检验的空着裁剪算法

空着向前裁剪是你搜索任何着法之前要做的事。你要问一个问题：“如果我在这里什么都不做，对手能做什么？”这就好比像打架时，根据自己的能力给对手一个出击的机会，来增加自己的信心。如果任凭对手攻击也无法击倒你，那么你攻击他的时候他会输掉。这个思想就给了对手出击的机会，如果你的局面仍然好到超过Beta的程度，你就假设如果你搜索了所有的着法也会超过Beta。

这个方法能节省时间的原因是，开始时用了减少深度的搜索。深度减少因子称为R，因此跟你用深度D搜索所有的着法相比，现在你是先以D - R搜索对手的着法。

不幸的是，空着向前裁剪在某些地方不能正常运行。我们作了一个重要的假定——假定走了一步棋会比不走棋有更高的分值。不幸的是，这个假定在很多典型的局面上并不成立，这些局面非常普遍，并且有一个名称——无等着局面(Zugzwang)。无等着局面指的是，如果你不走棋，局势会好些，但是你被强迫走子，这使得你的局势会崩溃。

“带检验的空着裁剪”(Verified Null-Move Pruning)指的是检验空着裁剪是否安全的算法，它首先由Tabibi发表在2002年的ICGA(原ICCA)杂志上[7]。其内容可以概括为：

1. 用R = 3的空着裁剪进行搜索；
2. 如果高出边界，那么做浅一层的搜索(这就意味着一层的搜索是无法做带验证的空着裁剪的)；
3. 做浅一层的搜索时，子结点用R = 3的不带验证的空着裁剪；
4. 如果浅一层的搜索高出边界，那么带验证的空着裁剪是成功的，否则必须重新做完全的搜索。

象棋百科全书网的作者认为这里存在很多问题：

1. 用R = 3非常冒进，还是用R = 2比较合适；
2. 检验时做浅一层的搜索太浪费时间，裁剪的层数可以跟空着裁剪一样的R值一样，而且窗口也用以Beta为界的零窗口；
3. 做检验时，子结点仍旧应该做带检验的空着裁剪，否则“连停着杀”就检测不出来了。

ElephantEye是否启用空着裁剪，分三种情况讨论：

1. 我方进攻子力达到3个，就使用不带检验的空着裁剪；
2. 我方进攻子力小于3个，则使用带检验的空着裁剪；
3. 我方只有仕(士)相(象)等守子，则禁用空着裁剪。

这里我们也采取了相同的策略。

## 2.3 模块设计

### 2.3.1 UCCI交互模块

UCCI指令分析模块由三各UCCI指令解释器组成。

其中第一个解释器"BootLine()"最简单，只用来接收引擎启动后的第一行指令

输入"ucci"时就返回"UCCI\_COMM\_UCCI"，否则一律返回"UCCI\_COMM\_UNKNOWN"

前两个解释器都等待是否有输入，如果没有输入则执行待机指令"Idle()"

而第三个解释器("BusyLine()"，只用在引擎思考时)则在没有输入时直接返回"UCCI\_COMM\_UNKNOWN"

1. 判断FEN串 bool ParsePos(UcciCommStruct& UcciComm, char\* lp)

功能描述：输入一个字符串lp，判断其是否指定了FEN串。

首先判断lp前缀是否时fen，如果是，则将后续字符串存入UcciComm.szFenStr

否则判断是否是"startpos"，如果都不是，返回false

然后继续寻找是否指定了后续着法，即是否有"moves"关键字，nMoveNum为后续着法数目，"moves"后面的每个着法都是1个空格和4个字符，将这些着法存储到UcciComm.lpdwMovesCoord中

1. 解释器一：UcciCommEnum BootLine(void)

功能描述：用来接受第一条指令，如果是ucci，则返回UCCI\_COMM\_UCCI，否则返回"UCCI\_COMM\_UNKNOWN"

1. 解释器二：UcciCommEnum IdleLine(UcciCommStruct& UcciComm, bool bDebug)

功能描述：用来接受除第一条指令外的所有指令，bDebug参数是进行测试时使用，将指令结果保存在一个UcciCommStruct对象中。

读到"isready"指令,返回"UCCI\_COMM\_ISREADY"

读到"position"指令，转移到ParsePos()，返回"UCCI\_COMM\_POSITION"

读到"go"指令，设置UcciComm.nTime为给出的时间，返回"UCCI\_COMM\_GO"

读到"quit"指令,返回"UCCI\_COMM\_QUIT"

读到其他指令,返回"UCCI\_COMM\_UNKNOWN"

1. 解释器三：UcciCommEnum BusyLine(UcciCommStruct& UcciComm, bool bDebug)

功能描述：引擎思考时的指令接受。包括"isready""stop"两条指令，分别返回"UCCI\_COMM\_ISREADY""UCCI\_COMM\_QUIT"

### 2.3.2 着法生成模块

首先，棋盘表示为一个16\*16的矩阵，用中间的9\*9的格子表示棋盘，把棋盘做成16x16的大小，得到行号和列号就可以用16除，这要比用9或10除快得多。16x16的棋盘 还有更大的好处，它可以防止棋子走出棋盘边界。

在中国象棋里，除车和炮以外的其他棋子，它们的着法都有固定的增量(行的增量，列的增量)，处理起来非常简单，也是着法生成技术的基础。例如马有8个着法，增量分 别是±0x0e、±0x12、±0x1f和±0x21，红方的过河兵有3个着法，增量分别是-0x10和 ±0x01。将所有棋子的着法增量列成表，供生成着法时使用。

棋盘上空的位置用0表示，棋子序号从0到47，其中0到15不用，16到31表示子，32到47表示黑子。每方的棋子顺序依次是：帅仕仕相相马马车车炮炮兵兵兵兵兵(将 士士象象马马车车炮炮卒卒卒卒卒)，判断棋子是红子用"pc < 32"，黑子用"pc >= 32"

1. 生成所有着法：int PositionStruct::GenMoves(int\* mvs, bool bCapture)

功能描述：生成当前局面下所有可能的着法。

首先遍历棋盘(0~256)，找到每一个己方棋子，再做后续判断

根据不同的棋子类型确定其不同的走法，具体如下：

(1) 将（帅）：只能在“九宫”内移动。每一步只可以前、后、左、右移动一格。走 之后，都不准造成帅、将在同一直线上直接对面。

(2) 士（仕）：只能在“九宫”内移动。每一步只可以沿对角线方向移动一格。

(3) 象（相）：河界的一侧。每一步只可以沿对角线方向斜走两格，即俗称“相（象）走田“字”。另外，在移动的过程中，若“田”字中心有棋子占据，则不允许移动，俗 称“塞相（象） 眼”。

(4) 马：任何位置。每一步走一直（或一横）一斜，可进可退，即俗称“马走日字”。另外， 如果在先直（横）的交叉点有棋子占据，俗称“蹩马腿”，则不许进或退。

(5) 车：任何位置，可以水平或垂直方向移动任意个无阻碍的点。

(6) 炮：任何位置，移动起来和车很相似，但它必须跳过一个棋子吃掉对方的一个棋子。

(7) 兵（卒）：任何位置，过“河界”之前，每步只能向前移动一格。过“河界”之后，每步可向前直走或横走一格。兵不允许后退。利用事先设置好的步长矩阵ccKingDelta[4]， ccAdvisorDelta[4]，ccKnightDelta[4][2]，ccKnightCheckDelta[4][2]，cucvlPiecePos[7][256]，ccKnightPin[512]，生成所有着法，函数将所有着法保存在mvs矩阵里，函数返回值为所有的着法数量。

1. 走一步棋： bool PositionStruct::MakeMove(int mv)

从着法中取一种着法，找到该棋子及目标位置是否有棋子，如果目标位置有棋子，则将其移除，替换为该棋子。

1. 判断走法是否合理：bool PositionStruct::LegalMove(int mv)

功能描述：判断走法是否合理

主要判断过程如下：

(1) 判断起始格是否有自己的棋子

(2) 判断目标格是否有自己的棋子

(3) 根据棋子的类型检查走法是否合理

1. 判断是否将军 bool PositionStruct::Checked()

功能描述：判断自己是否被将军

主要判断过程如下：

(1) 判断是否被对方的兵(卒)将军

(2) 判断是否被对方的马将军(以仕(士)的步长当作马腿)

(3) 判断是否被对方的车或炮将军(包括将帅对脸)

1. 判断是否被杀 bool PositionStruct::IsMate(void)

功能描述：判断是否被杀

1. 检测重复局面 int PositionStruct::RepStatus(int nRecur)

功能描述：检测重复局面

1. FEN串识别 void PositionStruct::FromFen(const char\* szFen)

功能描述：识别FEN串，生成棋盘。

FEN串的识别包括以下几个步骤：

1. 初始化，清空棋盘。

2. 读取棋盘上的棋子，转换为int型添加到棋盘变量中。

3. 确定轮到哪方走。

1. 生成FEN串 void PositionStruct::ToFen(char\* szFen)

功能描述：根据当前棋盘变量，生成FEN串。

1. 绘图 void PositionStruct::DrawBoard()

功能描述：将棋盘打印出来，便于调试与测试。

### 2.3.3 局面估值模块

在局面估值模块中，设定了局面评价估值函数，来评估局面。局面评估函数基于此假设：局面评估函数我们能把局面的性质量化成一个数字，这个数字可以对期盼上双方的棋力做出估计，也能一定反映出取胜的概率。

本程序的局面评估函数(Eval)是由双方子力之差所确定，下文中，另valBlack和valRed分别代表双方的子力，则形式化的说：

为了确定局面评估函数，就需要首先分别计算出双方的子力，子力的因素主要由于以下因素所确定：

棋子的种类和数目，不同种类的棋子是由不同的子力的，这个和棋子的机动性和威胁性有关。车、炮、马等攻击性较强的棋子，子力较大；而兵象士等棋子，攻击性较弱，则拥有较小的子力。而当拥有更多的棋子时，显然是有更强的子力的，多枚棋子的子力可以看作是单个棋子子力之和。

棋子的位置，棋子处于棋盘不同的位置，所拥有的子力也不尽相同。在不同的位置，棋子可能具有不同的走法，攻击性也不相同，所以位置是决定子力的关键因素。

在本程序中，定义了子力价值位置表table，table(k, x, y)表示单个棋子种类为k时，棋子位于棋盘坐标(x,y)处时，子力大小。

下面给出valBlack或者valWhile的形式化定义：

其中，当第15到31种棋子代表的是红方棋子，第32到47种棋子代表的是黑方棋子。总的估值函数为：

经过参数调整，本局面评估函数能够很好的反映出双方是优劣势和棋力大小。

### 2.3.4 着法搜索模块

着法搜索模块架构图如下：

图2.2 着法搜索模块架构图

其中，局面构建模块从ucci局面获得搜索所需的局面数据结构；走法生成模块根据当前局面生成可行走法；主搜索模块根据迭代加深搜索算法，逐渐增加最大深度，调用根节点搜索模块；根节点搜索模块依次选取可行走法，执行主要便利搜索模块；主要变例搜索模块首先尝试进行空着裁剪，对可能产生循环局面的走法进行裁剪，以防止静态搜索不能跳出，然后递归调用自身。当达到最大深度时，调用静态搜索模块。

### 2.3.5 开局库模块

　开局库在很多棋类中都是非常重要的。中国象棋已经被人类积累了大量知识和经验，要生成开局库，有个简单的方法就是找一个对局数据库，根据对局结果来选择可靠的着法。

本程序中，通过搜寻高手对局记录，生成了开局库。在开局库中，采用一个32位的变量dwZobristLock和两个16位的变量wmv和wvl，共同表示一个局面。将开局库的所有信息初始化储存在了buffer.h中bookBuffer数组中，使得程序不需要读取额外的文件，就能实现开局库的查询。

在book.cpp中，定义了BookDataStruc结构体，能够实现对buffer.h的随机访问。在BookDataStruct中，局面是以dwZobristLock为关键字升序排列的，则可以采用二分方法，实现在开局库中局面快速查找。

## 2.4 其他创新内容或优化算法

### 2.4.1 对将军的判断

判断将军是本程序的关键步骤之一，我们的做法很简单——生成所有走法，如果走任意一步都会被将军，则该局面被将军。

具体而言，我们有两种做法：

1. 让对方生成全部走法，看看其中有没有走法可以吃掉自己的帅(将)；
2. 按照判断走法是否符合规则的思路，采用更简单的做法。

第一种做法显然是正确的，然而计算成本过高，所以我们要尝试第二种做法。其实判断帅(将)是否被将军的过程并不复杂，只需要把棋子的角色进行互换：

1. 假设帅(将)是车，判断它是否能吃到对方的车和将(帅)(将帅不能对脸)；
2. 假设帅(将)是炮，判断它是否能吃到对方的炮；
3. 假设帅(将)是马，判断它是否能吃到对方的马；
4. 假设帅(将)是过河的兵(卒)，判断它是否能吃到对方的卒(兵)。

这样，一个复杂的走法生成过程就被简化成几个简单的走法判断过程。

### 2.4.2 走法排序策略

A. 杀手走法

杀手走法就是兄弟节点中产生Beta截断的走法。根据国际象棋的经验，杀手走法产生截断的可能性极大，所以我们在中国象棋里吸取了这个经验。显然，兄弟节点中的走法未必在当前节点下能走，所以在尝试杀手走法前先要对它进行走法合理性的判断。我如果杀手走法确实产生截断了，那么后面耗时更多的走法生成模块就可以不用执行了。

如何保存和获取“兄弟节点中产生截断的走法”呢？我们可以把这个问题简单化——距离根节点步数同样多的节点，彼此都称为“兄弟”节点，换句话说，亲兄弟、堂表兄弟以及关系更疏远的兄弟都称为“兄弟”。这样，我们可以把距离根节点的步数作为索引值，构造一个杀手走法表。

B. 历史表

“历史表启发”(History Heuristic)是杀手着法启发的扩展，历史表记录的是整个搜索树中着法的好坏。历史表的思想是：搜索树中某个结点上的一个好的着法，对于其他结点可能也是好的。没有什么非常可靠的理由来支持这个思想，但根据历史表来排序着法，总比不排序要好得多，而且实践证明这是一种效果非常好的启发算法[8]。与迭代加深结合，可以充分发挥历史表的作用——浅一层搜索结束后，历史表中积累了大量非常宝贵的数据，这将大幅度减少深一层搜索的时间。

C. 启发式走法排序策略

利用各种信息渠道(如置换表、杀手走法、历史表等)来优化走法顺序的手段称为“启发”。中国象棋引擎象棋小巫师[9]采用了多种启发方式：

1. 如果置换表中有过该局面的数据，但无法完全利用，那么多数情况下它是浅一层搜索中产生截断的走法，我们可以首先尝试它；
2. 然后是两个杀手走法(如果其中某个杀手走法与置换表走法一样，那么可以跳过)；
3. 然后生成全部走法，按历史表排序，再依次搜索(可以排除置换表走法和两个杀手走法)。

由于这种排序策略具有非常好的优化效果，我们在程序中也实现了该方法。

### 2.4.3 动态计算局面估值

在程序中，采用了局面动态估值的方法实现估值函数的快速计算，在走子过程中，程序便运算除了局面的股指评价函数。

在我们对一个局面搜索最佳着法之前，我们在搜索过程中评价任何局面，只要简单地把棋子在数组中的值加起来就行了。我们不必每一步都重新计算它们的和，在把棋子从一个格子移到另一个格子时，可以用下面的公式更新局面估值函数：

这样，就不必每次花费时间，遍历整个棋盘或者局面的评估，在搜索过程中，局面的评估函数便自然算出来了。

# 3 实验过程

## 3.1 环境说明

### 3.1.1 操作系统

Windows 10

### 3.1.2 开发语言

C++

### 3.1.3 开发环境具体要求

IDE：Visual Studio 2017

编译器：MSVC

### 3.1.4 核心使用库

Visual Studio 2017内的库

## 3.2 源代码文件清单

### 3.2.1 search.cpp：迭代加深搜索过程

const int IID\_DEPTH = 2; // 内部迭代加深的深度

const int SMP\_DEPTH = 6; // 并行搜索的深度

const int UNCHANGED\_DEPTH = 4; // 未改变最佳着法的深度

const int DROPDOWN\_VALUE = 20; // 落后的分值

const int RESIGN\_VALUE = 300; // 认输的分值

const int DRAW\_OFFER\_VALUE = 40; // 提和的分值

SearchStruct Search;

PositionStruct& Pos = Search.pos;

// 搜索信息，是封装在模块内部的

static struct {

int64\_t llTime; // 计时器

uint32\_t mvResult;                  // 走棋结果

bool bStop;              // 中止信号

int nUnchanged; // 未改变最佳着法的深度

uint16\_t wmvKiller[LIMIT\_DEPTH][2]; // 杀手着法表

HashStruct HashTable[HASH\_SIZE];    // 置换表

int nHistoryTable[65536];           // 历史表

MoveSortStruct MoveSort; // 根结点的着法序列

} Search2;

HashStruct\* hshItems = Search2.HashTable;

uint16\_t(\*wmvKiller)[2] = Search2.wmvKiller;

int\* nHistory = Search2.nHistoryTable;

void BuildPos(PositionStruct& pos, const UcciCommStruct& UcciComm) {

int i, mv;

pos.FromFen(UcciComm.szFenStr);

for (i = 0; i < UcciComm.nMoveNum; i++) {

mv = COORD\_MOVE(UcciComm.lpdwMovesCoord[i]);

if (mv == 0) {

break;

}

if (pos.LegalMove(mv) && pos.MakeMove(mv) && pos.Captured()) {

// 始终让pos.nMoveNum反映没吃子的步数

pos.SetIrrev();

}

}

}

// 重复裁剪

static int RepPruning(const PositionStruct& pos, int vlBeta) {

int vlRep = pos.RepStatus();

if (vlRep > 0) {

return pos.RepValue(vlRep);

}

return -MATE\_VALUE;

}

// 调整型局面评价函数

inline int Evaluate(const PositionStruct& pos) {

int vl;

vl = pos.Evaluate();

return vl == pos.DrawValue() ? vl - 1 : vl;

}

// 静态搜索过程

static int SearchQuiesc(PositionStruct& pos, int vlAlpha, int vlBeta) {

int vlBest, vl, nGenMoves;

int mvs[MAX\_GEN\_MOVES];

// 静态搜索例程包括以下几个步骤：

// 1. 重复裁剪；

vl = RepPruning(pos, vlBeta);

if (vl > -MATE\_VALUE) {

return vl;

}

// 2. 达到极限深度，直接返回评价值；

if (pos.nDistance == LIMIT\_DEPTH) {

return Evaluate(pos);

}

// 3. 初始化；

vlBest = -MATE\_VALUE;

// 4. 对于被将军的局面，生成全部着法；

if (pos.InCheck()) {

nGenMoves = pos.GenMoves(mvs);

std::sort(mvs, mvs + nGenMoves, CompareHistory);

}

else {

// 5. 对于未被将军的局面，在生成着法前首先尝试空着(空着启发)，即对局面作评价；

vl = Evaluate(pos);

if (vl >= vlBeta) {

return vl;

}

vlBest = vl;

vlAlpha = max(vl, vlAlpha);

// 6. 对于未被将军的局面，生成并排序所有吃子着法(MVV(LVA)启发)；

nGenMoves = pos.GenMoves(mvs, GEN\_CAPTURE);

std::sort(mvs, mvs + nGenMoves, CompareMvvLva);

}

// 7. 用Alpha-Beta算法搜索这些着法；

for (int i = 0; i < nGenMoves; i++) {

if (pos.MakeMove(mvs[i])) {

vl = -SearchQuiesc(pos, -vlBeta, -vlAlpha);

pos.UndoMakeMove();

if (vl > vlBest) {

if (vl >= vlBeta) {

return vl;

}

vlBest = vl;

vlAlpha = max(vl, vlAlpha);

}

}

}

// 8. 返回分值。

if (vlBest == -MATE\_VALUE) {

return pos.nDistance - MATE\_VALUE;

}

else {

return vlBest;

}

}

const bool NO\_NULL = false; // "SearchPV()"的参数，是否禁止空着裁剪

// 主变例搜索过程

static int SearchPV(int vlAlpha, int vlBeta, int nDepth, bool bNoNull = false) {

int nNewDepth, nHashFlag, vlBest, vl, nCurrTimer;

int mvBest, mvHash, mv;

MoveSortStruct MoveSort;

// 完全搜索例程包括以下几个步骤：

// 1. 在叶子结点处调用静态搜索；

if (nDepth <= 0) {

//      return Evaluate(Search.pos);

return SearchQuiesc(Search.pos, vlAlpha, vlBeta);

}

// 2. 重复裁剪；

vl = RepPruning(Search.pos, vlBeta);

if (vl > -MATE\_VALUE) {

return vl;

}

// 3. 置换裁剪；

vl = ProbeHash(Search.pos, vlAlpha, vlBeta, nDepth, mvHash);

if (Search.bUseHash && vl > -MATE\_VALUE) {

// 由于PV结点不适用置换裁剪，所以不会发生PV路线中断的情况

return vl;

}

// 4. 达到极限深度，直接返回评价值；

if (Search.pos.nDistance == LIMIT\_DEPTH) {

return Evaluate(Search.pos);

}

// 5. 尝试空着裁剪；

if (bNoNull && !Search.pos.InCheck() && Search.pos.NullOkay()) {

Search.pos.NullMove();

vl = -SearchPV(-vlBeta, 1 - vlBeta, nDepth - NULL\_DEPTH - 1, NO\_NULL);

Search.pos.UndoNullMove();

if (vl >= vlBeta) {

return vl;

}

}

// 6. 初始化；

mvBest = 0;

nHashFlag = HASH\_ALPHA;

vlBest = -MATE\_VALUE;

MoveSort.Init(mvHash);

// 7. 按照"MoveSortStruct::Next()"的着法顺序逐一搜索；

while ((mv = MoveSort.Next()) != 0) {

if (Search.pos.MakeMove(mv)) {

// 8. 尝试选择性延伸；

nNewDepth = (Search.pos.InCheck() ? nDepth : nDepth - 1);

// 9. 主要变例搜索；

if (vlBest == -MATE\_VALUE) {

vl = -SearchPV(-vlBeta, -vlAlpha, nNewDepth);

}

else {

vl = -SearchPV(-vlAlpha - 1, -vlAlpha, nNewDepth);

if (vl > vlAlpha && vl < vlBeta) {

vl = -SearchPV(-vlBeta, -vlAlpha, nNewDepth);

}

}

Search.pos.UndoMakeMove();

if (Search2.bStop) {

return vlBest;

}

// 10. Alpha-Beta边界判定；

if (vl > vlBest) {

vlBest = vl;

if (vl >= vlBeta) {

mvBest = mv;

nHashFlag = HASH\_BETA;

break;

}

if (vl > vlAlpha) {

vlAlpha = vl;

mvBest = mv;

nHashFlag = HASH\_PV;

}

}

nCurrTimer = (int)(GetTime() - Search2.llTime);

if (nCurrTimer > Search.nMaxTimer) {

Search2.bStop = true;

}

}

}

// 11. 更新置换表、历史表和杀手着法表。

if (vlBest == -MATE\_VALUE) {

return Search.pos.nDistance - MATE\_VALUE;

}

else {

RecordHash(Search.pos, nHashFlag, vlBest, nDepth, mvBest);

if (mvBest != 0) {

SetBestMove(mvBest, nDepth, Search2.wmvKiller[Search.pos.nDistance]);

}

return vlBest;

}

}

// 根结点搜索过程

static int SearchRoot(int nDepth) {

int nNewDepth, vlBest, vl, mv, nCurrTimer;

// 根结点搜索例程包括以下几个步骤：

// 1. 初始化

vlBest = -MATE\_VALUE;

Search2.MoveSort.Init(Search2.mvResult);

// 2. 逐一搜索每个着法

while ((mv = Search2.MoveSort.Next()) != 0) {

if (Search.pos.MakeMove(mv)) {

// 3. 尝试选择性延伸(只考虑将军延伸)

nNewDepth = (Search.pos.InCheck() ? nDepth : nDepth - 1);

// 4. 主要变例搜索

if (vlBest == -MATE\_VALUE) {

vl = -SearchPV(-MATE\_VALUE, MATE\_VALUE, nNewDepth, NO\_NULL);

}

else {

vl = -SearchPV(-vlBest - 1, -vlBest, nNewDepth);

if (vl > vlBest) { // 这里不需要" && vl < MATE\_VALUE"了

vl = -SearchPV(-MATE\_VALUE, -vlBest, nNewDepth, NO\_NULL);

}

}

Search.pos.UndoMakeMove();

if (Search2.bStop) {

return vlBest;

}

// 5. Alpha-Beta边界判定("vlBest"代替了"SearchPV()"中的"vlAlpha")

if (vl > vlBest) {

// 6. 如果搜索到第一着法，那么"未改变最佳着法"的计数器加1，否则清零

Search2.nUnchanged = (vlBest == -MATE\_VALUE ? Search2.nUnchanged + 1 : 0);

vlBest = vl;

// 7. 搜索到最佳着法时记录主要变例

Search2.mvResult = mv;

}

nCurrTimer = (int)(GetTime() - Search2.llTime);

if (nCurrTimer > Search.nMaxTimer) {

Search2.bStop = true;

}

}

}

RecordHash(Search.pos, HASH\_PV, vlBest, nDepth, Search2.mvResult);

SetBestMove(Search2.mvResult, nDepth, Search2.wmvKiller[Search.pos.nDistance]);

return vlBest;

}

// 主搜索例程

void SearchMain(int nDepth) {

int i, vl, vlLast;

int nCurrTimer, nLimitTimer;

int nBookMoves;

BookStruct bks[MAX\_GEN\_MOVES];

// 主搜索例程包括以下几个步骤：

// 2. 从开局库中搜索着法

if (Search.bUseBook) {

// a. 获取开局库中的所有走法

nBookMoves = GetBookMoves(Search.pos, bks);

if (nBookMoves > 0) {

vl = 0;

for (i = 0; i < nBookMoves; i++) {

vl += bks[i].wvl;

}

// b. 根据权重随机选择一个走法

vl = Search.rc4Random.NextLong() % (uint32\_t)vl;

for (i = 0; i < nBookMoves; i++) {

vl -= bks[i].wvl;

if (vl < 0) {

break;

}

}

// c. 如果开局库中的着法构成循环局面，那么不走这个着法

Search.pos.MakeMove(bks[i].wmv);

if (Search.pos.RepStatus(3) == 0) {

Search2.mvResult = bks[i].wmv;

Search.pos.UndoMakeMove();

uint32\_t result = MOVE\_COORD(Search2.mvResult);

printf("bestmove %.4s\n", (const char\*)& result);

fflush(stdout);

if (Search.bDebug)

Search.pos.DrawBoard(Search2.mvResult);

return;

}

Search.pos.UndoMakeMove();

}

}

// 3. 如果深度为零则返回静态搜索值

if (nDepth == 0) {

vl = SearchQuiesc(Search.pos, -MATE\_VALUE, MATE\_VALUE);

//      vl = Evaluate(Search.pos);

if (Search.bDebug) {

printf("info depth 0 score %d\n", vl);

fflush(stdout);

}

return;

}

// 4. 初始化时间和计数器

Search2.bStop = false;

Search2.nUnchanged = 0;

Search2.mvResult = 0;

ClearKiller(Search2.wmvKiller);

ClearHistory();

memset(Search2.HashTable, 0, sizeof(Search2.HashTable));

// 由于 ClearHash() 需要消耗一定时间，所以计时从这以后开始比较合理

Search2.llTime = GetTime();

vlLast = 0;

nCurrTimer = 0;

// 5. 做迭代加深搜索

for (i = 1; i <= nDepth; i++) {

// 6. 搜索根结点

vl = SearchRoot(i);

if (Search2.bStop) {

if (vl > -MATE\_VALUE) {

vlLast = vl; // 跳出后，vlLast会用来判断认输或投降，所以需要给定最近一个值

}

break; // 没有跳出，则"vl"是可靠值

}

if (Search.bDebug) {

printf("info depth %d score %d\n", i, vl);

fflush(stdout);

}

nCurrTimer = (int)(GetTime() - Search2.llTime);

// 7. 如果搜索时间超过适当时限，则终止搜索

nLimitTimer = Search.nMaxTimer;

// a. 如果当前搜索值没有落后前一层很多，那么适当时限减半

nLimitTimer = (vl + DROPDOWN\_VALUE >= vlLast ? nLimitTimer / 2 : nLimitTimer);

// b. 如果最佳着法连续多层没有变化，那么适当时限减半

nLimitTimer = (Search2.nUnchanged >= UNCHANGED\_DEPTH ? nLimitTimer / 2 : nLimitTimer);

if (nCurrTimer > nLimitTimer) {

vlLast = vl;

break; // 不管是否跳出，"vlLast"都已更新

}

vlLast = vl;

// 8. 搜索到杀棋则终止搜索

if (vlLast > WIN\_VALUE || vlLast < -WIN\_VALUE) {

break;

}

}

// 9. 输出最佳着法

uint32\_t result = MOVE\_COORD(Search2.mvResult);

printf("bestmove %.4s\n", (const char\*)& result);

fflush(stdout);

if (Search.bDebug)

Search.pos.DrawBoard(Search2.mvResult);

}

### 3.2.2 search.h ：迭代加深搜索过程的一些定义和工具函数

#include "tools.h"

#include "ucci.h"

#include "position.h"

#ifndef SEARCH\_H

#define SEARCH\_H

// 搜索前可设置的全局变量，指定搜索参数

struct SearchStruct {

    PositionStruct pos; // 待搜索的局面

    bool bQuit;                      // 是否收到退出指令

    bool bDebug;                     // 是否调试模式

    bool bUseHash, bUseBook; // 是否使用置换表裁剪和开局库

    RC4Struct rc4Random; // 随机数

    int nMaxTimer;                   // 最大使用时间

    char szBookFile[1024]; // 开局库

};

extern SearchStruct Search;

// UCCI局面构造过程

void BuildPos(PositionStruct& pos, const UcciCommStruct& UcciComm);

// 搜索的启动过程

void SearchMain(int nDepth);

#endif

### 3.2.3 book.h ：开局库着法的结构体的定义和一些工具函数

#include <cstdio>

#include "buffer.h"

#include "position.h"

#ifndef BOOK\_H

#define BOOK\_H

inline int BOOK\_POS\_CMP(const BookStruct& bk, const PositionStruct& pos) {

    return bk.dwZobristLock < pos.zobr.dwLock1 ? -1 :

        bk.dwZobristLock > pos.zobr.dwLock1 ? 1 : 0;

}

struct BookDataStruct {

    int nLen;

    void init() {

        nLen = 12081;

        return;

    }

    void Read(BookStruct& bk, int nPtr) const {

        bk = bookBuffer[nPtr];

    }

};

// 获取开局库着法

int GetBookMoves(const PositionStruct& pos, BookStruct\* lpbks);

#endif

### 3.2.4 book.cpp ：开局库着法的函数和方法的实现

#include "position.h"

#include "book.h"

int GetBookMoves(const PositionStruct& pos, BookStruct\* lpbks) {

    BookDataStruct BookData;

    PositionStruct posScan;

    BookStruct bk;

    int nScan, nLow, nHigh, nPtr;

    int i, j, nMoves;

    // 从开局库中搜索着法的例程，有以下几个步骤：

    BookData.init();

    // 1. 用折半查找法搜索局面；

    posScan = pos;

    for (nScan = 0; nScan < 2; nScan++) {

        nPtr = nLow = 0;

        nHigh = BookData.nLen - 1;

        while (nLow <= nHigh) {

            nPtr = (nLow + nHigh) / 2;

            BookData.Read(bk, nPtr);

            if (BOOK\_POS\_CMP(bk, posScan) < 0) {

                nLow = nPtr + 1;

            }

            else if (BOOK\_POS\_CMP(bk, posScan) > 0) {

                nHigh = nPtr - 1;

            }

            else {

                break;

            }

        }

        if (nLow <= nHigh) {

            break;

        }

        // 原局面和镜像局面各搜索一趟

        pos.Mirror(posScan);

    }

    // 2. 如果不到局面，则返回空着；

    if (nScan == 2) {

        return 0;

    }

    // 3. 如果找到局面，则向前查找第一个着法；

    for (nPtr--; nPtr >= 0; nPtr--) {

        BookData.Read(bk, nPtr);

        if (BOOK\_POS\_CMP(bk, posScan) < 0) {

            break;

        }

    }

    // 4. 向后依次读入属于该局面的每个着法；

    nMoves = 0;

    for (nPtr++; nPtr < BookData.nLen; nPtr++) {

        BookData.Read(bk, nPtr);

        if (BOOK\_POS\_CMP(bk, posScan) > 0) {

            break;

        }

        if (posScan.LegalMove(bk.wmv)) {

            // 如果局面是第二趟搜索到的，则着法必须做镜像

            lpbks[nMoves].nPtr = nPtr;

            lpbks[nMoves].wmv = (nScan == 0 ? bk.wmv : MOVE\_MIRROR(bk.wmv));

            lpbks[nMoves].wvl = bk.wvl;

            nMoves++;

            if (nMoves == MAX\_GEN\_MOVES) {

                break;

            }

        }

    }

    // 5. 对着法按分值排序

    for (i = 0; i < nMoves - 1; i++) {

        for (j = nMoves - 1; j > i; j--) {

            if (lpbks[j - 1].wvl < lpbks[j].wvl) {

                std::swap(lpbks[j - 1], lpbks[j]);

            }

        }

    }

    return nMoves;

}

### 3.2.5 ucci.h ：ucci通讯的结构体和相关的ucci的操作

#include "tools.h"

#ifndef UCCI\_H

#define UCCI\_H

const int UCCI\_MAX\_DEPTH = 32; // UCCI引擎思考的极限深度

const int LINE\_INPUT\_MAX\_CHAR = 8192;

const int MAX\_MOVE\_NUM = 1024;

// UCCI指令类型

enum UcciCommEnum {

    UCCI\_COMM\_UNKNOWN, UCCI\_COMM\_UCCI, UCCI\_COMM\_ISREADY,

    UCCI\_COMM\_POSITION, UCCI\_COMM\_GO, UCCI\_COMM\_QUIT

};

// UCCI指令可以解释成以下这个抽象的结构

union UcciCommStruct {

    /\* 1. "position"指令传递的信息，适合于"e\_CommPosition"指令类型

     \* "position"指令用来设置局面，包括初始局面连同后续着法构成的局面

     \* 例如，position startpos moves h2e2 h9g8，FEN串就是"startpos"代表的FEN串，着法数(MoveNum)就是2

     \*/

    struct {

        const char\* szFenStr; // FEN串，特殊局面(如"startpos"等)也由解释器最终转换成FEN串

        int nMoveNum; // 后续着法数

        uint32\_t\* lpdwMovesCoord; // 后续着法，指向程序"IdleLine()"中的一个静态数组，但可以把"CoordList"本身看成数组

    };

    /\* 2. "go"指令传递的信息，适合于"UCCI\_COMM\_GO指令类型 \*/

    struct {

        int nTime;

    };

};

// 下面三个函数用来解释UCCI指令，但适用于不同场合

UcciCommEnum BootLine(void); // UCCI引擎启动的第一条指令，只接收"ucci"

UcciCommEnum IdleLine(UcciCommStruct& UcciComm, bool bDebug); // 引擎空闲时接收指令

UcciCommEnum BusyLine(UcciCommStruct& UcciComm, bool bDebug); // 引擎思考时接收指令，只允许接收"stop"、"ponderhit"和"probe"

#endif

### 3.2.6 ucci.cpp ：ucci操作的实现

#include <cstdio>

#include <iostream>

#include "ucci.h"

#pragma comment(lib, "Shlwapi.lib")

/\* UCCI指令分析模块由三各UCCI指令解释器组成。

\*

\* 其中第一个解释器"BootLine()"最简单，只用来接收引擎启动后的第一行指令

\* 输入"ucci"时就返回"UCCI\_COMM\_UCCI"，否则一律返回"UCCI\_COMM\_UNKNOWN"

\* 前两个解释器都等待是否有输入，如果没有输入则执行待机指令"Idle()"

\* 而第三个解释器("BusyLine()"，只用在引擎思考时)则在没有输入时直接返回"UCCI\_COMM\_UNKNOWN"

\*/

static char szFen[LINE\_INPUT\_MAX\_CHAR];

static uint32\_t dwCoordList[MAX\_MOVE\_NUM];

static bool ParsePos(UcciCommStruct& UcciComm, char\* lp) {

    int i;

    //输入一个字符串lp，判断是否指定了FEN串

    //StrEqvSkip比较lp和fen，比较时自动忽略大小写，

    //若s1和s2相同则返回1，且lp指针后移strlen("fen")的长度，，否则返回0

    if (StrEqvSkip(lp, "fen ")) {

        strcpy(szFen, lp);

        UcciComm.szFenStr = szFen;//szFenStr为FEN串

     // 然后判断是否是startpos

     // 比较lp和startpos，比较时自动忽略大小写，

     // 若s1和s2相同则返回1，且lp指针后移strlen("fen")的长度，，否则返回0

     // 简单来说StrEqvSkip识别一个字符串然后指针移到这个字符串后面

    }

    else if (StrEqv(lp, "startpos")) {

        UcciComm.szFenStr = "rnbakabnr/9/1c5c1/p1p1p1p1p/9/9/P1P1P1P1P/1C5C1/9/RNBAKABNR w";

    }

    else {

        // 如果两者都不是，就立即返回

        return false;

    }

    // 然后寻找是否指定了后续着法，即是否有"moves"关键字

    UcciComm.nMoveNum = 0; //nMoveNum为后续着法数目

    if (StrScanSkip(lp, " moves ")) {

        //略过"moves"

        \*(lp - strlen(" moves ")) = '\0';

        UcciComm.nMoveNum = min((int)(strlen(lp) + 1) / 5, MAX\_MOVE\_NUM);   // "moves"后面的每个着法都是1个空格和4个字符

        for (i = 0; i < UcciComm.nMoveNum; i++) {

            dwCoordList[i] = \*(uint32\_t\*)lp; // 4个字符可转换为一个"uint32\_t"，存储和处理起来方便

            lp += sizeof(uint32\_t) + 1;// lp后移四个字符和一个空格

        }

        UcciComm.lpdwMovesCoord = dwCoordList;

    }

    return true;

}

UcciCommEnum BootLine(void) {

    //该函数用来接受第一条指令，如果是ucci，则返回UCCI\_COMM\_UCCI

    char szLineStr[LINE\_INPUT\_MAX\_CHAR];

    while (!std::cin.getline(szLineStr, LINE\_INPUT\_MAX\_CHAR)) {

        Idle();

    }

    if (StrEqv(szLineStr, "ucci")) {

        return UCCI\_COMM\_UCCI;

    }

    else {

        return UCCI\_COMM\_UNKNOWN;

    }

    //UCCI\_COMM\_UCCI和UCCI\_COMM\_UNKNOWN都是ucci指令类型

}

UcciCommEnum IdleLine(UcciCommStruct& UcciComm, bool bDebug) {

    //如果bDebug指令为1，输出当前读到的指令

    char szLineStr[LINE\_INPUT\_MAX\_CHAR];

    char\* lp;

    bool bGoTime;

    while (!std::cin.getline(szLineStr, LINE\_INPUT\_MAX\_CHAR)) {

        //输入一个指令

        Idle();

    }

    lp = szLineStr;

    if (bDebug) {

        printf("info idleline [%s]\n", lp);

        fflush(stdout);//清空输入缓冲区

    }

    if (false) {

        // "IdleLine()"是最复杂的UCCI指令解释器，大多数的UCCI指令都由它来解释，包括：

    }

    // 1. "isready"指令

    else if (StrEqv(lp, "isready")) {

        return UCCI\_COMM\_ISREADY;

    }

    // 2. "position {<special\_position> | fen <fen\_string>} [moves <move\_list>]"指令

    //如果读到position指令，调用ParsePos指令，着法列表保存在UcciComm里

    else if (StrEqvSkip(lp, "position ")) {

        return ParsePos(UcciComm, lp) ? UCCI\_COMM\_POSITION : UCCI\_COMM\_UNKNOWN;

    }

    // 3. "go [ponder | draw] <mode>"指令

    else if (StrEqvSkip(lp, "go time ")) {

        bGoTime = true;

        UcciComm.nTime = Str2Digit(lp, 0, 2000000000);

        return UCCI\_COMM\_GO;

    }

    // 4. "quit"指令

    else if (StrEqv(lp, "quit")) {

        return UCCI\_COMM\_QUIT;

    }

    // 5. 无法识别的指令

    else {

        return UCCI\_COMM\_UNKNOWN;

    }

}

UcciCommEnum BusyLine(UcciCommStruct& UcciComm, bool bDebug) {

    char szLineStr[LINE\_INPUT\_MAX\_CHAR];

    char\* lp;

    if (std::cin.getline(szLineStr, LINE\_INPUT\_MAX\_CHAR)) {

        if (bDebug) {

            printf("info busyline [%s]\n", szLineStr);

            fflush(stdout);

        }

        // "BusyLine"只能接收"isready"、"quit"这两条指令

        if (false) {

        }

        else if (StrEqv(szLineStr, "isready")) {

            return UCCI\_COMM\_ISREADY;

        }

        else if (StrEqv(szLineStr, "quit")) {

            return UCCI\_COMM\_QUIT;

        }

        else {

            lp = szLineStr;

            return UCCI\_COMM\_UNKNOWN;

        }

    }

    else {

        return UCCI\_COMM\_UNKNOWN;

    }

}

### 3.2.7 movesort.h ：着法的效力的排序的定义和实现

#include <cstring>

#include <algorithm>

#include "position.h"

#ifndef MOVESORT\_H

#define MOVESORT\_H

const int LIMIT\_DEPTH = 64; // 搜索的极限深度

const int SORT\_VALUE\_MAX = 65535; // 着法序列最大值

extern int\* nHistory; // 历史表

// 走法排序阶段

const int PHASE\_HASH = 0;

const int PHASE\_KILLER\_1 = 1;

const int PHASE\_KILLER\_2 = 2;

const int PHASE\_GEN\_MOVES = 3;

const int PHASE\_REST = 4;

const bool NEXT\_ALL = true; // 着法顺序函数"MoveSortStruct::NextQuiesc()"选项

const bool ROOT\_UNIQUE = true; // 着法顺序函数"MoveSortStruct::ResetRoot()"选项

// MVV/LVA每种子力的价值

static int cucMvvLva[24] = {

0, 0, 0, 0, 0, 0, 0, 0,

5, 1, 1, 3, 4, 3, 2, 0,

5, 1, 1, 3, 4, 3, 2, 0

};

extern PositionStruct& Pos;

extern uint16\_t(\*wmvKiller)[2];

// 求MVV/LVA值

inline int MvvLva(int mv) {

    return (cucMvvLva[Pos.ucpcSquares[DST(mv)]] << 3) - cucMvvLva[Pos.ucpcSquares[SRC(mv)]];

}

// "sort"按MVV/LVA值排序的比较函数

bool CompareMvvLva(const int lpmv1, const int lpmv2) {

    return MvvLva(lpmv1) > MvvLva(lpmv2);

}

// "sort"按历史表排序的比较函数

bool CompareHistory(const int lpmv1, const int lpmv2) {

    return nHistory[lpmv1] > nHistory[lpmv2];

}

// 着法序列结构

struct MoveSortStruct {

    int mvHash, mvKiller1, mvKiller2; // 置换表走法和两个杀手走法

    int nPhase, nIndex, nGenMoves; // 当前阶段，当前采用第几个走法，总共有几个走法

    int mvs[MAX\_GEN\_MOVES]; // 所有的走法

    void Init(int mvHash\_) { // 初始化，设定置换表走法和两个杀手走法

        mvHash = mvHash\_;

        mvKiller1 = wmvKiller[Pos.nDistance][0];

        mvKiller2 = wmvKiller[Pos.nDistance][1];

        nPhase = PHASE\_HASH;

    }

    int Next(void); // 得到下一个走法

};

// 清空历史表

inline void ClearHistory(void) {

    memset(nHistory, 0, sizeof(int[65536]));

}

// 清空杀手着法表

inline void ClearKiller(uint16\_t(\*lpwmvKiller)[2]) {

    memset(lpwmvKiller, 0, LIMIT\_DEPTH \* sizeof(uint16\_t[2]));

}

// 得到下一个走法

int MoveSortStruct::Next(void) {

    int mv;

    switch (nPhase) {

        // "nPhase"表示着法启发的若干阶段，依次为：

        // 0. 置换表着法启发，完成后立即进入下一阶段；

        case PHASE\_HASH:

            nPhase = PHASE\_KILLER\_1;

            if (mvHash != 0) {

                return mvHash;

            }

            // 技巧：这里没有"break"，表示"switch"的上一个"case"执行完后紧接着做下一个"case"，下同

         // 1. 杀手着法启发(第一个杀手着法)，完成后立即进入下一阶段；

        case PHASE\_KILLER\_1:

            nPhase = PHASE\_KILLER\_2;

            if (mvKiller1 != mvHash && mvKiller1 != 0 && Pos.LegalMove(mvKiller1)) {

                return mvKiller1;

            }

            // 2. 杀手着法启发(第二个杀手着法)，完成后立即进入下一阶段；

        case PHASE\_KILLER\_2:

            nPhase = PHASE\_GEN\_MOVES;

            if (mvKiller2 != mvHash && mvKiller2 != 0 && Pos.LegalMove(mvKiller2)) {

                return mvKiller2;

            }

            // 3. 生成所有着法，完成后立即进入下一阶段；

        case PHASE\_GEN\_MOVES:

            nPhase = PHASE\_REST;

            nGenMoves = Pos.GenMoves(mvs);

            std::sort(mvs, mvs + nGenMoves, CompareHistory);

            nIndex = 0;

            // 4. 对剩余着法做历史表启发；

        case PHASE\_REST:

            while (nIndex < nGenMoves) {

                mv = mvs[nIndex];

                nIndex++;

                if (mv != mvHash && mv != mvKiller1 && mv != mvKiller2) {

                    return mv;

                }

            }

            // 5. 没有着法了，返回零。

        default:

            return 0;

    }

}

// 对最佳走法的处理

inline void SetBestMove(int mv, int nDepth, uint16\_t\* lpwmvKiller) {

    nHistory[mv] += nDepth \* nDepth;

    if (lpwmvKiller[0] != mv) {

        lpwmvKiller[1] = lpwmvKiller[0];

        lpwmvKiller[0] = mv;

    }

}

#endif

### 3.2.8 position.h ：棋盘的结构体和一些棋盘的有关操作

#include <cstring>

#include "tools.h"

#ifndef POSITION\_H

#define POSITION\_H

const int MAX\_GEN\_MOVES = 128;          // 最大的生成走法数

const int MAX\_MOVES = 256;              // 最大的历史走法数

const int MATE\_VALUE = 10000; // 最高分值，即将死的分值

const int WIN\_VALUE = MATE\_VALUE - 100; // 搜索出胜负的分值界限，超出此值就说明已经搜索出杀棋了

const int NULLOKAY\_MARGIN = 200; // 空着裁剪可以不检验的子力价值边界

const int NULLSAFE\_MARGIN = 400; // 允许使用空着裁剪的条件的子力价值边界

const int DRAW\_VALUE = 20; // 和棋时返回的分数(取负值)

// 每种子力的类型编号

const int KING\_TYPE = 0;

const int ADVISOR\_TYPE = 1;

const int BISHOP\_TYPE = 2;

const int KNIGHT\_TYPE = 3;

const int ROOK\_TYPE = 4;

const int CANNON\_TYPE = 5;

const int PAWN\_TYPE = 6;

// 每种子力的开始序号和结束序号

const int KING\_FROM = 0;

const int ADVISOR\_FROM = 1;

const int ADVISOR\_TO = 2;

const int BISHOP\_FROM = 3;

const int BISHOP\_TO = 4;

const int KNIGHT\_FROM = 5;

const int KNIGHT\_TO = 6;

const int ROOK\_FROM = 7;

const int ROOK\_TO = 8;

const int CANNON\_FROM = 9;

const int CANNON\_TO = 10;

const int PAWN\_FROM = 11;

const int PAWN\_TO = 15;

// 棋盘范围

const int RANK\_TOP = 3;

const int RANK\_BOTTOM = 12;

const int FILE\_LEFT = 3;

const int FILE\_RIGHT = 11;

const int ADVANCED\_VALUE = 3; // 先行权分值

const int NULL\_MARGIN = 400; // 空步裁剪的子力边界

extern const bool ccInBoard[256]; // 棋盘区域表

extern const bool ccInFort[256]; // 城池区域表

extern const int8\_t ccLegalSpan[512]; // 合理着法跨度表

extern const int8\_t ccKnightPin[512]; // 马腿表

extern const uint8\_t cucvlPiecePos[7][256]; // 子力位置价值表

extern const int cnPieceTypes[48];  // 棋子序号对应的棋子类型

// "GenMoves"参数

const bool GEN\_CAPTURE = true;

// 起始局面的FEN串

const char\* const cszStartFen = "rnbakabnr/9/1c5c1/p1p1p1p1p/9/9/P1P1P1P1P/1C5C1/9/RNBAKABNR w";

// 棋子类型对应的棋子符号

const char\* const cszPieceBytes = "KABNRCP";

// 棋子类型对应的中文棋子符号标识

const char\* const cszPieceBytesInChineseBlack[7] = { "将","士","象","碼","車","砲","卒" };

const char\* const cszPieceBytesInChineseRed[7] = { "帅","仕","相","马","车","炮","兵" };

// 棋子类型

inline int PIECE\_TYPE(int pc) {

    return cnPieceTypes[pc];

}

// 棋子下标

inline int PIECE\_INDEX(int pc) {

    return pc & 15;

}

// 判断棋子是否在棋盘中

inline bool IN\_BOARD(int sq) {

    return ccInBoard[sq];

}

// 判断棋子是否在九宫中

inline bool IN\_FORT(int sq) {

    return ccInFort[sq];

}

// 获得格子的横坐标

inline int RANK\_Y(int sq) {

    return sq >> 4;

}

// 获得格子的纵坐标

inline int FILE\_X(int sq) {

    return sq & 15;

}

// 根据纵坐标和横坐标获得格子

inline int COORD\_XY(int x, int y) {

    return x + (y << 4);

}

// 翻转格子

inline int SQUARE\_FLIP(int sq) {

    return 254 - sq;

}

// 纵坐标水平镜像

inline int FILE\_FLIP(int x) {

    return 14 - x;

}

// 横坐标垂直镜像

inline int RANK\_FLIP(int y) {

    return 15 - y;

}

// 格子水平镜像

inline int MIRROR\_SQUARE(int sq) {

    return COORD\_XY(FILE\_FLIP(FILE\_X(sq)), RANK\_Y(sq));

}

inline int SQUARE\_FORWARD(int sq, int sd) {

    return sq - 16 + (sd << 5);

}

inline int SQUARE\_BACKWARD(int sq, int sd) {

    return sq + 16 - (sd << 5);

}

// 走法是否符合帅(将)的步长

inline bool KING\_SPAN(int sqSrc, int sqDst) {

    return ccLegalSpan[sqDst - sqSrc + 256] == 1;

}

// 走法是否符合仕(士)的步长

inline bool ADVISOR\_SPAN(int sqSrc, int sqDst) {

    return ccLegalSpan[sqDst - sqSrc + 256] == 2;

}

// 走法是否符合相(象)的步长

inline bool BISHOP\_SPAN(int sqSrc, int sqDst) {

    return ccLegalSpan[sqDst - sqSrc + 256] == 3;

}

// 相(象)眼的位置

inline int BISHOP\_PIN(int sqSrc, int sqDst) {

    return (sqSrc + sqDst) >> 1;

}

// 马腿的位置

inline int KNIGHT\_PIN(int sqSrc, int sqDst) {

    return sqSrc + ccKnightPin[sqDst - sqSrc + 256];

}

// 是否未过河

inline bool HOME\_HALF(int sq, int sd) {

    return (sq & 0x80) != (sd << 7);

}

// 是否已过河

inline bool AWAY\_HALF(int sq, int sd) {

    return (sq & 0x80) == (sd << 7);

}

// 是否在河的同一边

inline bool SAME\_HALF(int sqSrc, int sqDst) {

    return ((sqSrc ^ sqDst) & 0x80) == 0;

}

// 是否在同一行

inline bool SAME\_RANK(int sqSrc, int sqDst) {

    return ((sqSrc ^ sqDst) & 0xf0) == 0;

}

// 是否在同一列

inline bool SAME\_FILE(int sqSrc, int sqDst) {

    return ((sqSrc ^ sqDst) & 0x0f) == 0;

}

inline int SIDE\_TAG(int sd) {

    return 16 + (sd << 4);

}

inline int OPP\_SIDE\_TAG(int sd) {

    return 32 - (sd << 4);

}

struct ZobristStruct {

    uint32\_t dwKey, dwLock0, dwLock1;

    void InitZero(void) {

        dwKey = dwLock0 = dwLock1 = 0;

    }

    void InitRC4(RC4Struct& rc4) {

        dwKey = rc4.NextLong();

        dwLock0 = rc4.NextLong();

        dwLock1 = rc4.NextLong();

    }

    void Xor(const ZobristStruct& zobr) {

        dwKey ^= zobr.dwKey;

        dwLock0 ^= zobr.dwLock0;

        dwLock1 ^= zobr.dwLock1;

    }

    void Xor(const ZobristStruct& zobr1, const ZobristStruct& zobr2) {

        dwKey ^= zobr1.dwKey ^ zobr2.dwKey;

        dwLock0 ^= zobr1.dwLock0 ^ zobr2.dwLock0;

        dwLock1 ^= zobr1.dwLock1 ^ zobr2.dwLock1;

    }

}; // zobr

// Zobrist表

extern struct ZobristTable{

    ZobristStruct Player;

    ZobristStruct Table[14][256];

} Zobrist;  // 其实没错

// 初始化Zobrist表

void InitZobrist(void);

// 历史走法信息

struct MoveStruct {

    uint16\_t wmv;

    uint8\_t ucpcCaptured, ucbCheck;

    uint32\_t dwKey;

    void Set(int mv, int pcCaptured, bool bCheck, uint32\_t dwKey\_) {

        wmv = mv;

        ucpcCaptured = pcCaptured;

        ucbCheck = bCheck;

        dwKey = dwKey\_;

    }

}; // mvs

// 获得走法的起点

inline int SRC(int mv) {

    return mv & 255;

}

// 获得走法的终点

inline int DST(int mv) {

    return mv >> 8;

}

// 根据起点和终点获得走法

inline int MOVE(int sqSrc, int sqDst) {

    return sqSrc + (sqDst << 8);

}

// 走法水平镜像

inline int MOVE\_MIRROR(int mv) {

    return MOVE(MIRROR\_SQUARE(SRC(mv)), MIRROR\_SQUARE(DST(mv)));

}

inline uint32\_t MOVE\_COORD(int mv) { // 把着法转换成字符串

    union {

        char c[4];

        uint32\_t dw;

    } Ret;

    Ret.c[0] = FILE\_X(SRC(mv)) - FILE\_LEFT + 'a';

    Ret.c[1] = '9' - RANK\_Y(SRC(mv)) + RANK\_TOP;

    Ret.c[2] = FILE\_X(DST(mv)) - FILE\_LEFT + 'a';

    Ret.c[3] = '9' - RANK\_Y(DST(mv)) + RANK\_TOP;

    return Ret.dw;

}

inline int COORD\_MOVE(uint32\_t dwMoveStr) { // 把字符串转换成着法

    int sqSrc, sqDst;

    char\* lpArgPtr;

    lpArgPtr = (char\*)& dwMoveStr;

    sqSrc = COORD\_XY(lpArgPtr[0] - 'a' + FILE\_LEFT, '9' - lpArgPtr[1] + RANK\_TOP);

    sqDst = COORD\_XY(lpArgPtr[2] - 'a' + FILE\_LEFT, '9' - lpArgPtr[3] + RANK\_TOP);

    return (IN\_BOARD(sqSrc) && IN\_BOARD(sqDst) ? MOVE(sqSrc, sqDst) : 0);

}

// 局面结构

struct PositionStruct {

    int sdPlayer; // 轮到谁走，0=红方，1=黑方

    uint8\_t ucpcSquares[256];       // 棋盘上的棋子

    uint8\_t ucsqPieces[48];         // 每个棋子放的位置，0表示被吃

    int vlWhite, vlBlack; // 红、黑双方的子力价值

    int nDistance, nMoveNum; // 距离根节点的步数，历史走法数

    MoveStruct mvsList[MAX\_MOVES]; // 历史走法信息列表

    ZobristStruct zobr; // Zobrist

    void ClearBoard(void) { // 清空棋盘

        sdPlayer = vlWhite = vlBlack = nDistance = 0;

        memset(ucpcSquares, 0, 256);

        memset(ucsqPieces, 0, 48);

        zobr.InitZero();

    }

    void SetIrrev(void) { // 清空(初始化)历史走法信息

        mvsList[0].Set(0, 0, Checked(), zobr.dwKey);

        nMoveNum = 1;

    }

    void ChangeSide(void) { // 交换走子方

        sdPlayer = 1 - sdPlayer;

        zobr.Xor(Zobrist.Player);

    }

    void AddPiece(int sq, int pc) { // 在棋盘上放一枚棋子

        int pt;

        ucpcSquares[sq] = pc;

        ucsqPieces[pc] = sq;

        pt = PIECE\_TYPE(pc);

        // 红方加分，黑方(注意"cucvlPiecePos"取值要颠倒)减分

        if (pc < 32) {

            vlWhite += cucvlPiecePos[pt][sq];

            zobr.Xor(Zobrist.Table[pt][sq]);

        }

        else {

            vlBlack += cucvlPiecePos[pt][SQUARE\_FLIP(sq)];

            zobr.Xor(Zobrist.Table[pt + 7][sq]);

        }

    }

    void DelPiece(int sq, int pc) { // 从棋盘上拿走一枚棋子

        int pt;

        ucpcSquares[sq] = 0;

        ucsqPieces[pc] = 0;

        pt = PIECE\_TYPE(pc);

        // 红方减分，黑方(注意"cucvlPiecePos"取值要颠倒)加分

        if (pc < 32) {

            vlWhite -= cucvlPiecePos[pt][sq];

            zobr.Xor(Zobrist.Table[pt][sq]);

        }

        else {

            vlBlack -= cucvlPiecePos[pt][SQUARE\_FLIP(sq)];

            zobr.Xor(Zobrist.Table[pt + 7][sq]);

        }

    }

    int Evaluate(void) const { // 局面评价函数

        return (sdPlayer == 0 ? vlWhite - vlBlack : vlBlack - vlWhite) + ADVANCED\_VALUE;

    }

    bool InCheck(void) const { // 是否被将军

        return mvsList[nMoveNum - 1].ucbCheck;

    }

    bool Captured(void) const { // 上一步是否吃子

        return mvsList[nMoveNum - 1].ucpcCaptured != 0;

    }

    int MovePiece(int mv); // 搬一步棋的棋子

    void UndoMovePiece(int mv, int pcCaptured); // 撤消搬一步棋的棋子

    bool MakeMove(int mv); // 走一步棋

    void UndoMakeMove(void) { // 撤消走一步棋

        nDistance--;

        nMoveNum--;

        ChangeSide();

        UndoMovePiece(mvsList[nMoveNum].wmv, mvsList[nMoveNum].ucpcCaptured);

    }

    void NullMove(void) { // 走一步空步

        uint32\_t dwKey;

        dwKey = zobr.dwKey;

        ChangeSide();

        mvsList[nMoveNum].Set(0, 0, false, dwKey);

        nMoveNum++;

        nDistance++;

    }

    void UndoNullMove(void) { // 撤消走一步空步

        nDistance--;

        nMoveNum--;

        ChangeSide();

    }

    // 生成所有走法，如果"bCapture"为"TRUE"则只生成吃子走法

    int GenMoves(int\* mvs, bool bCapture = false) const;

    bool LegalMove(int mv) const; // 判断走法是否合理

    bool Checked(void) const; // 判断是否被将军

    int DrawValue(void) const { // 和棋分值

        return (nDistance & 1) == 0 ? -DRAW\_VALUE : DRAW\_VALUE;

    }

    int RepStatus(int nRecur = 1) const; // 检测重复局面

    int RepValue(int nRepStatus) const { // 重复局面分值

        int vlReturn;

        vlReturn = ((nRepStatus & 2) == 0 ? 0 : nDistance - MATE\_VALUE) +

            ((nRepStatus & 4) == 0 ? 0 : MATE\_VALUE - nDistance);

        return vlReturn == 0 ? DrawValue() : vlReturn;

    }

    bool NullOkay(void) const { // 判断是否允许空步裁剪

        return (sdPlayer == 0 ? vlWhite : vlBlack) > NULL\_MARGIN;

    }

    // 局面处理过程

    void FromFen(const char\* szFen);            // FEN串识别

    void ToFen(char\* szFen) const;           // 生成FEN串

    void Mirror(PositionStruct & posMirror) const; // 对局面镜像

    void DrawBoard(uint32\_t mv = NULL);         // 打印局面

};

#endif

### 3.2.9 position.cpp ：棋盘结构体的一些操作的实现

#include <cstdio>

#include "position.h"

// 判断棋子是否在棋盘中的数组

static const bool ccInBoard[256] = {

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,

0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,

0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,

0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,

0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,

0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,

0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,

0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,

0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,

0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

};

// 判断棋子是否在九宫的数组

static const bool ccInFort[256] = {

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

};

// 判断步长是否符合特定走法的数组，1=帅(将)，2=仕(士)，3=相(象)

static const int8\_t ccLegalSpan[512] = {

                     0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 3, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 2, 1, 2, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 2, 1, 2, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 3, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0

};

// 根据步长判断马是否蹩腿的数组

static const int8\_t ccKnightPin[512] = {

                             0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0,-16, 0,-16, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, -1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, -1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 16, 0, 16, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0

};

// 帅(将)的步长

static const int8\_t ccKingDelta[4] = { -16, -1, 1, 16 };

// 仕(士)的步长

static const int8\_t ccAdvisorDelta[4] = { -17, -15, 15, 17 };

// 马的步长，以帅(将)的步长作为马腿

static const int8\_t ccKnightDelta[4][2] = { {-33, -31}, {-18, 14}, {-14, 18}, {31, 33} };

// 马被将军的步长，以仕(士)的步长作为马腿

static const int8\_t ccKnightCheckDelta[4][2] = { {-33, -18}, {-31, -14}, {14, 31}, {18, 33} };

// 子力位置价值表

static const uint8\_t cucvlPiecePos[7][256] = {

{ // 帅(将)

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 11, 15, 11, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

}, { // 仕(士)

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 20, 0, 20, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 23, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 20, 0, 20, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

}, { // 相(象)

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 20, 0, 0, 0, 20, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 18, 0, 0, 0, 23, 0, 0, 0, 18, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 20, 0, 0, 0, 20, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

}, { // 马

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 90, 90, 90, 96, 90, 96, 90, 90, 90, 0, 0, 0, 0,

    0, 0, 0, 90, 96,103, 97, 94, 97,103, 96, 90, 0, 0, 0, 0,

    0, 0, 0, 92, 98, 99,103, 99,103, 99, 98, 92, 0, 0, 0, 0,

    0, 0, 0, 93,108,100,107,100,107,100,108, 93, 0, 0, 0, 0,

    0, 0, 0, 90,100, 99,103,104,103, 99,100, 90, 0, 0, 0, 0,

    0, 0, 0, 90, 98,101,102,103,102,101, 98, 90, 0, 0, 0, 0,

    0, 0, 0, 92, 94, 98, 95, 98, 95, 98, 94, 92, 0, 0, 0, 0,

    0, 0, 0, 93, 92, 94, 95, 92, 95, 94, 92, 93, 0, 0, 0, 0,

    0, 0, 0, 85, 90, 92, 93, 78, 93, 92, 90, 85, 0, 0, 0, 0,

    0, 0, 0, 88, 85, 90, 88, 90, 88, 90, 85, 88, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

}, { // 车

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0,206,208,207,213,214,213,207,208,206, 0, 0, 0, 0,

    0, 0, 0,206,212,209,216,233,216,209,212,206, 0, 0, 0, 0,

    0, 0, 0,206,208,207,214,216,214,207,208,206, 0, 0, 0, 0,

    0, 0, 0,206,213,213,216,216,216,213,213,206, 0, 0, 0, 0,

    0, 0, 0,208,211,211,214,215,214,211,211,208, 0, 0, 0, 0,

    0, 0, 0,208,212,212,214,215,214,212,212,208, 0, 0, 0, 0,

    0, 0, 0,204,209,204,212,214,212,204,209,204, 0, 0, 0, 0,

    0, 0, 0,198,208,204,212,212,212,204,208,198, 0, 0, 0, 0,

    0, 0, 0,200,208,206,212,200,212,206,208,200, 0, 0, 0, 0,

    0, 0, 0,194,206,204,212,200,212,204,206,194, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

}, { // 炮

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0,100,100, 96, 91, 90, 91, 96,100,100, 0, 0, 0, 0,

    0, 0, 0, 98, 98, 96, 92, 89, 92, 96, 98, 98, 0, 0, 0, 0,

    0, 0, 0, 97, 97, 96, 91, 92, 91, 96, 97, 97, 0, 0, 0, 0,

    0, 0, 0, 96, 99, 99, 98,100, 98, 99, 99, 96, 0, 0, 0, 0,

    0, 0, 0, 96, 96, 96, 96,100, 96, 96, 96, 96, 0, 0, 0, 0,

    0, 0, 0, 95, 96, 99, 96,100, 96, 99, 96, 95, 0, 0, 0, 0,

    0, 0, 0, 96, 96, 96, 96, 96, 96, 96, 96, 96, 0, 0, 0, 0,

    0, 0, 0, 97, 96,100, 99,101, 99,100, 96, 97, 0, 0, 0, 0,

    0, 0, 0, 96, 97, 98, 98, 98, 98, 98, 97, 96, 0, 0, 0, 0,

    0, 0, 0, 96, 96, 97, 99, 99, 99, 97, 96, 96, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

}, { // 兵(卒)

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 9, 9, 9, 11, 13, 11, 9, 9, 9, 0, 0, 0, 0,

    0, 0, 0, 19, 24, 34, 42, 44, 42, 34, 24, 19, 0, 0, 0, 0,

    0, 0, 0, 19, 24, 32, 37, 37, 37, 32, 24, 19, 0, 0, 0, 0,

    0, 0, 0, 19, 23, 27, 29, 30, 29, 27, 23, 19, 0, 0, 0, 0,

    0, 0, 0, 14, 18, 20, 27, 29, 27, 20, 18, 14, 0, 0, 0, 0,

    0, 0, 0, 7, 0, 13, 0, 16, 0, 13, 0, 7, 0, 0, 0, 0,

    0, 0, 0, 7, 0, 7, 0, 15, 0, 7, 0, 7, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

}

};

/\* 棋子序号对应的棋子类型

\*

\* 棋子序号从0到47，其中0到15不用，16到31表示红子，32到47表示黑子。

\* 每方的棋子顺序依次是：帅仕仕相相马马车车炮炮兵兵兵兵兵(将士士象象马马车车炮炮卒卒卒卒卒)

\* 提示：判断棋子是红子用"pc < 32"，黑子用"pc >= 32"

\*/

static const int cnPieceTypes[48] = {

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 6, 6, 6,

0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 6, 6, 6

};

ZobristTable Zobrist;

// 初始化Zobrist表

void InitZobrist(void) {

    int i, j;

    RC4Struct rc4;

    rc4.InitZero();

    Zobrist.Player.InitRC4(rc4);

    for (i = 0; i < 14; i++) {

        for (j = 0; j < 256; j++) {

            Zobrist.Table[i][j].InitRC4(rc4);

        }

    }

}

// 搬一步棋的棋子

int PositionStruct::MovePiece(int mv) {

    int sqSrc, sqDst, pc, pcCaptured;

    sqSrc = SRC(mv);

    sqDst = DST(mv);

    pcCaptured = ucpcSquares[sqDst];

    if (pcCaptured != 0) {

        DelPiece(sqDst, pcCaptured);

    }

    pc = ucpcSquares[sqSrc];

    DelPiece(sqSrc, pc);

    AddPiece(sqDst, pc);

    return pcCaptured;

}

// 撤消搬一步棋的棋子

void PositionStruct::UndoMovePiece(int mv, int pcCaptured) {

    int sqSrc, sqDst, pc;

    sqSrc = SRC(mv);

    sqDst = DST(mv);

    pc = ucpcSquares[sqDst];

    DelPiece(sqDst, pc);

    AddPiece(sqSrc, pc);

    if (pcCaptured != 0) {

        AddPiece(sqDst, pcCaptured);

    }

}

// 走一步棋

bool PositionStruct::MakeMove(int mv) {

    int pcCaptured;

    uint32\_t dwKey;

    dwKey = zobr.dwKey;

    pcCaptured = MovePiece(mv);

    if (Checked()) {

        UndoMovePiece(mv, pcCaptured);

        return false;

    }

    ChangeSide();

    mvsList[nMoveNum].Set(mv, pcCaptured, Checked(), dwKey);

    nMoveNum++;

    nDistance++;

    return true;

}

// 生成所有走法，如果"bCapture"为"true"则只生成吃子走法

int PositionStruct::GenMoves(int\* mvs, bool bCapture) const {

    int i, j, nGenMoves, nDelta, sqSrc, sqDst;

    int pcSelfSide, pcOppSide, pcSrc, pcDst;

    // 生成所有走法，需要经过以下几个步骤：

    nGenMoves = 0;

    pcSelfSide = SIDE\_TAG(sdPlayer);

    pcOppSide = OPP\_SIDE\_TAG(sdPlayer);

    for (sqSrc = 0; sqSrc < 256; sqSrc++) {

        // 1. 找到一个本方棋子，再做以下判断：

        pcSrc = ucpcSquares[sqSrc];

        if ((pcSrc & pcSelfSide) == 0) {

            continue;

        }

        // 2. 根据棋子确定走法

        switch (PIECE\_INDEX(pcSrc)) {

            case KING\_FROM:

                for (i = 0; i < 4; i++) {

                    sqDst = sqSrc + ccKingDelta[i];

                    if (!IN\_FORT(sqDst)) {

                        continue;

                    }

                    pcDst = ucpcSquares[sqDst];

                    if (bCapture ? (pcDst & pcOppSide) != 0 : (pcDst & pcSelfSide) == 0) {

                        mvs[nGenMoves] = MOVE(sqSrc, sqDst);

                        nGenMoves++;

                    }

                }

                break;

            case ADVISOR\_FROM:

            case ADVISOR\_TO:

                for (i = 0; i < 4; i++) {

                    sqDst = sqSrc + ccAdvisorDelta[i];

                    if (!IN\_FORT(sqDst)) {

                        continue;

                    }

                    pcDst = ucpcSquares[sqDst];

                    if (bCapture ? (pcDst & pcOppSide) != 0 : (pcDst & pcSelfSide) == 0) {

                        mvs[nGenMoves] = MOVE(sqSrc, sqDst);

                        nGenMoves++;

                    }

                }

                break;

            case BISHOP\_FROM:

            case BISHOP\_TO:

                for (i = 0; i < 4; i++) {

                    sqDst = sqSrc + ccAdvisorDelta[i];

                    if (!(IN\_BOARD(sqDst) && HOME\_HALF(sqDst, sdPlayer) && ucpcSquares[sqDst] == 0)) {

                        continue;

                    }

                    sqDst += ccAdvisorDelta[i];

                    pcDst = ucpcSquares[sqDst];

                    if (bCapture ? (pcDst & pcOppSide) != 0 : (pcDst & pcSelfSide) == 0) {

                        mvs[nGenMoves] = MOVE(sqSrc, sqDst);

                        nGenMoves++;

                    }

                }

                break;

            case KNIGHT\_FROM:

            case KNIGHT\_TO:

                for (i = 0; i < 4; i++) {

                    sqDst = sqSrc + ccKingDelta[i];

                    if (ucpcSquares[sqDst] != 0) {

                        continue;

                    }

                    for (j = 0; j < 2; j++) {

                        sqDst = sqSrc + ccKnightDelta[i][j];

                        if (!IN\_BOARD(sqDst)) {

                            continue;

                        }

                        pcDst = ucpcSquares[sqDst];

                        if (bCapture ? (pcDst & pcOppSide) != 0 : (pcDst & pcSelfSide) == 0) {

                            mvs[nGenMoves] = MOVE(sqSrc, sqDst);

                            nGenMoves++;

                        }

                    }

                }

                break;

            case ROOK\_FROM:

            case ROOK\_TO:

                for (i = 0; i < 4; i++) {

                    nDelta = ccKingDelta[i];

                    sqDst = sqSrc + nDelta;

                    while (IN\_BOARD(sqDst)) {

                        pcDst = ucpcSquares[sqDst];

                        if (pcDst == 0) {

                            if (!bCapture) {

                                mvs[nGenMoves] = MOVE(sqSrc, sqDst);

                                nGenMoves++;

                            }

                        }

                        else {

                            if ((pcDst & pcOppSide) != 0) {

                                mvs[nGenMoves] = MOVE(sqSrc, sqDst);

                                nGenMoves++;

                            }

                            break;

                        }

                        sqDst += nDelta;

                    }

                }

                break;

            case CANNON\_FROM:

            case CANNON\_TO:

                for (i = 0; i < 4; i++) {

                    nDelta = ccKingDelta[i];

                    sqDst = sqSrc + nDelta;

                    while (IN\_BOARD(sqDst)) {

                        pcDst = ucpcSquares[sqDst];

                        if (pcDst == 0) {

                            if (!bCapture) {

                                mvs[nGenMoves] = MOVE(sqSrc, sqDst);

                                nGenMoves++;

                            }

                        }

                        else {

                            break;

                        }

                        sqDst += nDelta;

                    }

                    sqDst += nDelta;

                    while (IN\_BOARD(sqDst)) {

                        pcDst = ucpcSquares[sqDst];

                        if (pcDst != 0) {

                            if ((pcDst & pcOppSide) != 0) {

                                mvs[nGenMoves] = MOVE(sqSrc, sqDst);

                                nGenMoves++;

                            }

                            break;

                        }

                        sqDst += nDelta;

                    }

                }

                break;

            default:

                sqDst = SQUARE\_FORWARD(sqSrc, sdPlayer);

                if (IN\_BOARD(sqDst)) {

                    pcDst = ucpcSquares[sqDst];

                    if (bCapture ? (pcDst & pcOppSide) != 0 : (pcDst & pcSelfSide) == 0) {

                        mvs[nGenMoves] = MOVE(sqSrc, sqDst);

                        nGenMoves++;

                    }

                }

                if (AWAY\_HALF(sqSrc, sdPlayer)) {

                    for (nDelta = -1; nDelta <= 1; nDelta += 2) {

                        sqDst = sqSrc + nDelta;

                        if (IN\_BOARD(sqDst)) {

                            pcDst = ucpcSquares[sqDst];

                            if (bCapture ? (pcDst & pcOppSide) != 0 : (pcDst & pcSelfSide) == 0) {

                                mvs[nGenMoves] = MOVE(sqSrc, sqDst);

                                nGenMoves++;

                            }

                        }

                    }

                }

                break;

        }

    }

    return nGenMoves;

}

// 判断走法是否合理

bool PositionStruct::LegalMove(int mv) const {

    int sqSrc, sqDst, sqPin;

    int pcSelfSide, pcSrc, pcDst, nDelta;

    // 判断走法是否合法，需要经过以下的判断过程：

    // 1. 判断起始格是否有自己的棋子

    sqSrc = SRC(mv);

    pcSrc = ucpcSquares[sqSrc];

    pcSelfSide = SIDE\_TAG(sdPlayer);

    if ((pcSrc & pcSelfSide) == 0) {

        return false;

    }

    // 2. 判断目标格是否有自己的棋子

    sqDst = DST(mv);

    pcDst = ucpcSquares[sqDst];

    if ((pcDst & pcSelfSide) != 0) {

        return false;

    }

    // 3. 根据棋子的类型检查走法是否合理

    switch (PIECE\_INDEX(pcSrc)) {

        case KING\_FROM:

            return IN\_FORT(sqDst) && KING\_SPAN(sqSrc, sqDst);

        case ADVISOR\_FROM:

        case ADVISOR\_TO:

            return IN\_FORT(sqDst) && ADVISOR\_SPAN(sqSrc, sqDst);

        case BISHOP\_FROM:

        case BISHOP\_TO:

            return SAME\_HALF(sqSrc, sqDst) && BISHOP\_SPAN(sqSrc, sqDst) &&

                ucpcSquares[BISHOP\_PIN(sqSrc, sqDst)] == 0;

        case KNIGHT\_FROM:

        case KNIGHT\_TO:

            sqPin = KNIGHT\_PIN(sqSrc, sqDst);

            return sqPin != sqSrc && ucpcSquares[sqPin] == 0;

        case ROOK\_FROM:

        case ROOK\_TO:

        case CANNON\_FROM:

        case CANNON\_TO:

            if (SAME\_RANK(sqSrc, sqDst)) {

                nDelta = (sqDst < sqSrc ? -1 : 1);

            }

            else if (SAME\_FILE(sqSrc, sqDst)) {

                nDelta = (sqDst < sqSrc ? -16 : 16);

            }

            else {

                return false;

            }

            sqPin = sqSrc + nDelta;

            while (sqPin != sqDst && ucpcSquares[sqPin] == 0) {

                sqPin += nDelta;

            }

            if (sqPin == sqDst) {

                return pcDst == 0 || pcSrc - pcSelfSide == ROOK\_FROM || pcSrc - pcSelfSide == ROOK\_TO;

            }

            else if (pcDst != 0 && (pcSrc - pcSelfSide == CANNON\_FROM || pcSrc - pcSelfSide == CANNON\_TO)) {

                sqPin += nDelta;

                while (sqPin != sqDst && ucpcSquares[sqPin] == 0) {

                    sqPin += nDelta;

                }

                return sqPin == sqDst;

            }

            else {

                return false;

            }

        default:

            if (AWAY\_HALF(sqDst, sdPlayer) && (sqDst == sqSrc - 1 || sqDst == sqSrc + 1)) {

                return true;

            }

            return sqDst == SQUARE\_FORWARD(sqSrc, sdPlayer);

    }

}

// 判断是否被将军

// 注意：pc代表棋子编号，sq代表棋子位置

bool PositionStruct::Checked() const {

    int i, j, sqSrc, sqDst;

    int pcSelfSide, pcOppSide, pcDst, nDelta;

    pcSelfSide = SIDE\_TAG(sdPlayer);    //16 = 6'b010000, 16~31 = 6'b01xxxx

    pcOppSide = OPP\_SIDE\_TAG(sdPlayer); //32 = 6'b100000, 32~47 = 6'b10xxxx, &运算非零判断同侧

    // 找到棋盘上的帅(将)，再做以下判断：

    sqSrc = ucsqPieces[pcSelfSide];

    if (sqSrc == 0) {

        return false;

    }

    // 1. 判断是否被对方的兵(卒)将军

    pcDst = ucpcSquares[SQUARE\_FORWARD(sqSrc, sdPlayer)];

    if ((pcDst & pcOppSide) != 0 && PIECE\_INDEX(pcDst) >= PAWN\_FROM) {

        return true;

    }

    for (nDelta = -1; nDelta <= 1; nDelta += 2) {

        pcDst = ucpcSquares[sqSrc + nDelta];

        if ((pcDst & pcOppSide) != 0 && PIECE\_INDEX(pcDst) >= PAWN\_FROM) {

            return true;

        }

    }

    // 2. 判断是否被对方的马将军(以仕(士)的步长当作马腿)

    for (i = 0; i < 4; i++) {

        if (ucpcSquares[sqSrc + ccAdvisorDelta[i]] != 0) {

            continue;

        }

        for (j = 0; j < 2; j++) {

            pcDst = ucpcSquares[sqSrc + ccKnightCheckDelta[i][j]];

            if ((pcDst & pcOppSide) != 0 && (PIECE\_INDEX(pcDst) == KNIGHT\_FROM

                || PIECE\_INDEX(pcDst) == KNIGHT\_TO)) {

                return true;

            }

        }

    }

    // 3. 判断是否被对方的车或炮将军(包括将帅对脸)

    for (i = 0; i < 4; i++) {

        nDelta = ccKingDelta[i];

        sqDst = sqSrc + nDelta;

        while (IN\_BOARD(sqDst)) {

            pcDst = ucpcSquares[sqDst];

            if (pcDst != 0) {   //扫描线上第一个棋子

                if ((pcDst & pcOppSide) != 0 && (PIECE\_INDEX(pcDst) == ROOK\_FROM

                    || PIECE\_INDEX(pcDst) == ROOK\_TO || PIECE\_INDEX(pcDst) == KING\_FROM)) {

                    return true;

                }

                break;

            }

            sqDst += nDelta;

        }

        sqDst += nDelta;

        while (IN\_BOARD(sqDst)) {

            int pcDst = ucpcSquares[sqDst];

            if (pcDst != 0) {   //扫描线上第二个棋子

                if ((pcDst & pcOppSide) != 0 && (PIECE\_INDEX(pcDst) == CANNON\_FROM

                    || PIECE\_INDEX(pcDst) == CANNON\_TO)) {

                    return true;

                }

                break;

            }

            sqDst += nDelta;

        }

    }

    return false;

}

// 检测重复局面

int PositionStruct::RepStatus(int nRecur) const {

    bool bSelfSide, bPerpCheck, bOppPerpCheck;

    const MoveStruct\* lpmvs;

    bSelfSide = false;

    bPerpCheck = bOppPerpCheck = true;  // 长将标记

    lpmvs = mvsList + nMoveNum - 1; // 指向历史表中最后节点

    while (lpmvs->wmv != 0 && lpmvs->ucpcCaptured == 0) {

        if (bSelfSide) {

            bPerpCheck = bPerpCheck && lpmvs->ucbCheck;

            if (lpmvs->dwKey == zobr.dwKey) {

                nRecur--;

                if (nRecur == 0) {

                    return 1 + (bPerpCheck ? 2 : 0) + (bOppPerpCheck ? 4 : 0);

                }

            }

        }

        else {

            bOppPerpCheck = bOppPerpCheck && lpmvs->ucbCheck;

        }

        bSelfSide = !bSelfSide;

        lpmvs--;

    }

    return 0;

}

inline char PIECE\_BYTE(int pt) {

    return cszPieceBytes[pt];

}

// FEN串中棋子标识，注意这个函数只能识别大写字母，因此用小写字母时，首先必须转换为大写

int FenPiece(int nArg) {

    switch (nArg) {

        case 'K':

            return 0;

        case 'A':

            return 1;

        case 'B':

        case 'E':

            return 2;

        case 'N':

        case 'H':

            return 3;

        case 'R':

            return 4;

        case 'C':

            return 5;

        case 'P':

            return 6;

        default:

            return 7;

    }

}

// FEN串识别

void PositionStruct::FromFen(const char\* szFen) {

    int i, j, k;

    int pcWhite[7];

    int pcBlack[7];

    const char\* lpFen;

    // FEN串的识别包括以下几个步骤：

    // 1. 初始化，清空棋盘

    pcWhite[0] = SIDE\_TAG(0) + KING\_FROM;

    pcWhite[1] = SIDE\_TAG(0) + ADVISOR\_FROM;

    pcWhite[2] = SIDE\_TAG(0) + BISHOP\_FROM;

    pcWhite[3] = SIDE\_TAG(0) + KNIGHT\_FROM;

    pcWhite[4] = SIDE\_TAG(0) + ROOK\_FROM;

    pcWhite[5] = SIDE\_TAG(0) + CANNON\_FROM;

    pcWhite[6] = SIDE\_TAG(0) + PAWN\_FROM;

    for (i = 0; i < 7; i++) {

        pcBlack[i] = pcWhite[i] + 16;

    }

    /\* 数组"pcWhite[7]"和"pcBlack[7]"分别代表红方和黑方每个兵种即将占有的序号，

     \* 以"pcWhite[7]"为例，由于棋子16到31依次代表“帅仕仕相相马马车车炮炮兵兵兵兵兵”，

     \* 所以最初应该是"pcWhite[7] = {16, 17, 19, 21, 23, 25, 27}"，每添加一个棋子，该项就增加1，

     \* 这种做法允许添加多余的棋子(例如添加第二个帅，就变成仕了)，但添加前要做边界检测

     \*/

    ClearBoard();

    lpFen = szFen;

    if (\*lpFen == '\0') {

        SetIrrev();

        return;

    }

    // 2. 读取棋盘上的棋子

    i = RANK\_TOP;

    j = FILE\_LEFT;

    while (\*lpFen != ' ') {

        if (\*lpFen == '/') {

            j = FILE\_LEFT;

            i++;

            if (i > RANK\_BOTTOM) {

                break;

            }

        }

        else if (\*lpFen >= '1' && \*lpFen <= '9') {

            j += (\*lpFen - '0');

        }

        else if (\*lpFen >= 'A' && \*lpFen <= 'Z') {

            if (j <= FILE\_RIGHT) {

                k = FenPiece(\*lpFen);

                if (k < 7) {

                    if (pcWhite[k] < 32) {

                        if (this->ucsqPieces[pcWhite[k]] == 0) {

                            AddPiece(COORD\_XY(j, i), pcWhite[k]);

                            pcWhite[k] ++;

                        }

                    }

                }

                j++;

            }

        }

        else if (\*lpFen >= 'a' && \*lpFen <= 'z') {

            if (j <= FILE\_RIGHT) {

                k = FenPiece(\*lpFen + 'A' - 'a');

                if (k < 7) {

                    if (pcBlack[k] < 48) {

                        if (this->ucsqPieces[pcBlack[k]] == 0) {

                            AddPiece(COORD\_XY(j, i), pcBlack[k]);

                            pcBlack[k] ++;

                        }

                    }

                }

                j++;

            }

        }

        lpFen++;

        if (\*lpFen == '\0') {

            SetIrrev();

            return;

        }

    }

    lpFen++;

    // 3. 确定轮到哪方走

    if (\*lpFen == 'b') {

        ChangeSide();

    }

    // 4. 把局面设成“不可逆”

    SetIrrev();

}

// 生成FEN串

void PositionStruct::ToFen(char\* szFen) const {

    int i, j, k, pc;

    char\* lpFen;

    lpFen = szFen;

    for (i = RANK\_TOP; i <= RANK\_BOTTOM; i++) {

        k = 0;

        for (j = FILE\_LEFT; j <= FILE\_RIGHT; j++) {

            pc = this->ucpcSquares[COORD\_XY(j, i)];

            if (pc != 0) {

                if (k > 0) {

                    \*lpFen = k + '0';

                    lpFen++;

                    k = 0;

                }

                \*lpFen = PIECE\_BYTE(PIECE\_TYPE(pc)) + (pc < 32 ? 0 : 'a' - 'A');

                lpFen++;

            }

            else {

                k++;

            }

        }

        if (k > 0) {

            \*lpFen = k + '0';

            lpFen++;

        }

        \*lpFen = '/';

        lpFen++;

    }

    \*(lpFen - 1) = ' '; // 把最后一个'/'替换成' '

    \*lpFen = (this->sdPlayer == 0 ? 'w' : 'b');

    lpFen++;

    \*lpFen = '\0';

}

// 对局面镜像

void PositionStruct::Mirror(PositionStruct& posMirror) const {

    int sq, pc;

    posMirror.ClearBoard();

    for (sq = 0; sq < 256; sq++) {

        pc = ucpcSquares[sq];

        if (pc != 0) {

            posMirror.AddPiece(MIRROR\_SQUARE(sq), pc);

        }

    }

    if (sdPlayer == 1) {

        posMirror.ChangeSide();

    }

    posMirror.SetIrrev();

}

// 绘图

inline const char\* PIECE\_BYTE\_IN\_CHINESE(int pt, bool type) {

    if (type == true)

        return cszPieceBytesInChineseRed[pt];

    else

        return cszPieceBytesInChineseBlack[pt];

}

void PositionStruct::DrawBoard(uint32\_t mv)

{

    int i, j, pc;

    const char\* c;

    if (mv != NULL) {

        pc = ucpcSquares[SRC(mv)];

        ucpcSquares[SRC(mv)] = 0;

        ucpcSquares[DST(mv)] = pc;

    }

    for (i = RANK\_TOP; i <= RANK\_BOTTOM; i++) {

        for (j = FILE\_LEFT; j <= FILE\_RIGHT; j++) {

            pc = ucpcSquares[COORD\_XY(j, i)];

            if (pc == 0) {

                printf(" .");

            }

            else {

                c = PIECE\_BYTE\_IN\_CHINESE(PIECE\_TYPE(pc), pc < 32);

                printf("%s", c);

            }

        }

        printf(" %d\n", 12 - i);

    }

    printf(" a b c d e f g h i\n");

}

### 3.2.10 hash.h ：置换表结构和信息的结构体和一些宏定义和工具函数

#include <cstring>

#include "position.h"

#ifndef HASH\_H

#define HASH\_H

// 置换表标志，只用在"RecordHash()"函数中

const int HASH\_SIZE = 1 << 20; // 置换表大小

const int HASH\_ALPHA = 1; // ALPHA节点的置换表项

const int HASH\_BETA = 2; // BETA节点的置换表项

const int HASH\_PV = 3; // PV节点的置换表项

const int NULL\_DEPTH = 2; // 空着裁剪的深度

// 置换表结构，置换表信息夹在两个Zobrist校验锁中间，可以防止存取冲突

struct HashStruct {

    uint8\_t ucDepth, ucFlag;    // 深度，标志位

    int16\_t svl;                // 分值

    uint16\_t wmv, wReserved;    // 最佳着法，内存填充

    uint32\_t dwLock0, dwLock1;  // Zobrist校验锁

}; // hsh

extern HashStruct\* hshItems;

// 提取置换表项

static int ProbeHash(const PositionStruct& pos, int vlAlpha, int vlBeta, int nDepth, int& mv) {

    bool bMate; // 杀棋标志：如果是杀棋，那么不需要满足深度条件

    HashStruct hsh;

    hsh = hshItems[pos.zobr.dwKey & (HASH\_SIZE - 1)];

    if (hsh.dwLock0 != pos.zobr.dwLock0 || hsh.dwLock1 != pos.zobr.dwLock1) {

        mv = 0;

        return -MATE\_VALUE;

    }

    mv = hsh.wmv;

    bMate = FALSE;

    if (hsh.svl > WIN\_VALUE) {

        hsh.svl -= pos.nDistance;

        bMate = TRUE;

    }

    else if (hsh.svl < -WIN\_VALUE) {

        hsh.svl += pos.nDistance;

        bMate = TRUE;

    }

    if (hsh.ucDepth >= nDepth || bMate) {

        if (hsh.ucFlag == HASH\_BETA) {

            return (hsh.svl >= vlBeta ? hsh.svl : -MATE\_VALUE);

        }

        else if (hsh.ucFlag == HASH\_ALPHA) {

            return (hsh.svl <= vlAlpha ? hsh.svl : -MATE\_VALUE);

        }

        return hsh.svl;

    }

    return -MATE\_VALUE;

};

// 保存置换表项

static void RecordHash(const PositionStruct& pos, int nFlag, int vl, int nDepth, int mv) {

    HashStruct hsh;

    hsh = hshItems[pos.zobr.dwKey & (HASH\_SIZE - 1)];

    if (hsh.ucDepth > nDepth) {

        return;

    }

    hsh.ucFlag = nFlag;

    hsh.ucDepth = nDepth;

    if (vl > WIN\_VALUE) {

        hsh.svl = vl + pos.nDistance;

    }

    else if (vl < -WIN\_VALUE) {

        hsh.svl = vl - pos.nDistance;

    }

    else {

        hsh.svl = vl;

    }

    hsh.wmv = mv;

    hsh.dwLock0 = pos.zobr.dwLock0;

    hsh.dwLock1 = pos.zobr.dwLock1;

    hshItems[pos.zobr.dwKey & (HASH\_SIZE - 1)] = hsh;

};

#endif

### 3.2.11 starfish.cpp ：象棋博弈中的主程序

#include <cstdio>

#include "tools.h"

#include "ucci.h"

#include "position.h"

#include "hash.h"

#include "search.h"

inline void PrintLn(const char\* sz) {

    printf("%s\n", sz);

    fflush(stdout);

}

int main(void) {

    UcciCommStruct UcciComm;

    if (BootLine() != UCCI\_COMM\_UCCI) {

        return 0;

    }

    InitZobrist();

    Search.pos.FromFen(cszStartFen);

    Search.pos.nDistance = 0;

    Search.bQuit = false;

    Search.bDebug = true;

    Search.bUseHash = true;

    Search.bUseBook = true;

    PrintLn("ucciok");

    // 以下是接收指令和提供对策的循环体

    while (!Search.bQuit) {

        switch (IdleLine(UcciComm, Search.bDebug)) {

            case UCCI\_COMM\_ISREADY:

                PrintLn("readyok");

                break;

            case UCCI\_COMM\_POSITION:

                BuildPos(Search.pos, UcciComm);

                Search.pos.nDistance = 0;

                break;

            case UCCI\_COMM\_GO:

                Search.nMaxTimer = (int)(0.95 \* UcciComm.nTime);

                SearchMain(UCCI\_MAX\_DEPTH);

                break;

            case UCCI\_COMM\_QUIT:

                Search.bQuit = true;

                break;

            default:

                break;

        }

    }

    PrintLn("bye");

    return 0;

}

### 3.2.12 tools.h ：其他工具函数

#include <sys/timeb.h>

#include <cstdint>

#include <cstdio>

#include <cstring>

#include <algorithm>

#include <windows.h>

#include <shlwapi.h>

#ifndef TOOLS\_H

#define TOOLS\_H

inline int64\_t GetTime() {

    timeb tb;

    ftime(&tb);

    return (int64\_t)tb.time \* 1000 + tb.millitm;

}

inline void Idle(void) {

    Sleep(1);

}

inline char\* strcasestr(const char\* sz1, const char\* sz2) {

    return StrStrI(sz1, sz2);

}

inline int strncasecmp(const char\* sz1, const char\* sz2, size\_t n) {

    return \_strnicmp(sz1, sz2, n);

}

inline void StrCutCrLf(char\* sz) {

    char\* lpsz;

    lpsz = strchr(sz, '\r');

    if (lpsz != NULL) {

        \*lpsz = '\0';

    }

    lpsz = strchr(sz, '\n');

    if (lpsz != NULL) {

        \*lpsz = '\0';

    }

}

inline bool StrEqv(const char\* sz1, const char\* sz2) {

    return strncasecmp(sz1, sz2, strlen(sz2)) == 0;

}

inline bool StrEqvSkip(const char\*& sz1, const char\* sz2) {

    if (strncasecmp(sz1, sz2, strlen(sz2)) == 0) {

        sz1 += strlen(sz2);

        return true;

    }

    else {

        return false;

    }

}

inline bool StrEqvSkip(char\*& sz1, const char\* sz2) {

    if (strncasecmp(sz1, sz2, strlen(sz2)) == 0) {

        sz1 += strlen(sz2);

        return true;

    }

    else {

        return false;

    }

}

inline bool StrScan(const char\* sz1, const char\* sz2) {

    return strcasestr(sz1, sz2) != NULL;

}

inline bool StrScanSkip(const char\*& sz1, const char\* sz2) {

    const char\* lpsz;

    lpsz = strcasestr(sz1, sz2);

    if (lpsz == NULL) {

        return false;

    }

    else {

        sz1 = lpsz + strlen(sz2);

        return true;

    }

}

inline bool StrScanSkip(char\*& sz1, const char\* sz2) {

    char\* lpsz;

    lpsz = strcasestr(sz1, sz2);

    if (lpsz == NULL) {

        return false;

    }

    else {

        sz1 = lpsz + strlen(sz2);

        return true;

    }

}

inline bool StrSplitSkip(const char\*& szSrc, int nSeparator, char\* szDst = NULL) {

    const char\* lpsz;

    lpsz = strchr(szSrc, nSeparator);

    if (lpsz == NULL) {

        if (szDst != NULL) {

            strcpy(szDst, szSrc);

        }

        szSrc += strlen(szSrc);

        return false;

    }

    else {

        if (szDst != NULL) {

            strncpy(szDst, szSrc, lpsz - szSrc);

            szDst[lpsz - szSrc] = '\0';

        }

        szSrc = lpsz + 1;

        return true;

    }

}

inline bool StrSplitSkip(char\*& szSrc, int nSeparator, char\* szDst = NULL) {

    char\* lpsz;

    lpsz = strchr(szSrc, nSeparator);

    if (lpsz == NULL) {

        if (szDst != NULL) {

            strcpy(szDst, szSrc);

        }

        szSrc += strlen(szSrc);

        return false;

    }

    else {

        if (szDst != NULL) {

            strncpy(szDst, szSrc, lpsz - szSrc);

            szDst[lpsz - szSrc] = '\0';

        }

        szSrc = lpsz + 1;

        return true;

    }

}

inline int Str2Digit(const char\* sz, int nMin, int nMax) {

    int nRet;

    if (sscanf(sz, "%d", &nRet) > 0) {

        return min(max(nRet, nMin), nMax);

    }

    else {

        return nMin;

    }

}

struct RC4Struct {

    uint8\_t s[256];

    int x, y;

    void InitZero(void) {

        int i, j;

        uint8\_t uc;

        x = y = j = 0;

        for (i = 0; i < 256; i++) {

            s[i] = i;

        }

        for (i = 0; i < 256; i++) {

            j = (j + s[i]) & 255;

            uc = s[i];

            s[i] = s[j];

            s[j] = uc;

        }

    }

    uint8\_t NextByte(void) {

        x = (x + 1) & 255;

        y = (y + s[x]) & 255;

        std::swap(s[x], s[y]);

        return s[(s[x] + s[y]) & 255];

    }

    uint32\_t NextLong(void) {

        union {

            uint8\_t uc[4];

            uint32\_t dw;

        } Ret;

        Ret.uc[0] = NextByte();

        Ret.uc[1] = NextByte();

        Ret.uc[2] = NextByte();

        Ret.uc[3] = NextByte();

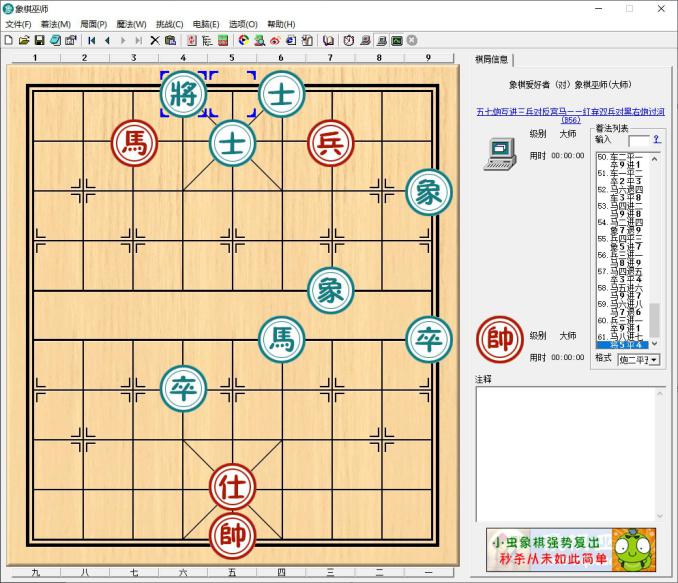
        return Ret.dw;

    }

};

#endif

## 3.3 实验结果展示

使用象棋巫师的UCCI接口的图形化表示的实验结果截图如下：  
 

## 3.4 实验结论

1. 实现了正确的UCCI通讯交互
2. 实现了较为高效的搜索剪枝算法
3. 实现了估值的算法
4. 实现了可以和其他支持UCCI的软件进行对战

# 4 总结

## 4.1 实验中存在的问题及解决方案

1. Hash表的初始化一直不能成功，导致局面判重出现错误，进而导致过早截断，不能正常执行搜索过程。后来发现在外部不能修改static变量，修改变量属性后解决；
2. 静态搜索没有最大深度参数，在搜索到循环局面后会一直搜索下去，导致程序出现死循环，在加入局面判重模块后解决；
3. 判断是否超时后，总不能在时限内退出搜索，在对最大时限乘上参数0.95后解决。
4. 测试界面过于简单，无法根据实时情况输出棋盘状态。后来利用每次搜索的结果，即bestmove操作，对棋盘进行修改然后输出，从而直观的得到AI的走棋情况。
5. 在处理棋盘上棋子位置时，棋子的编号及表示出现错误，对后续的步骤产生影响，后经调试，找到错误原因。
6. 估值函数不能反映空间，棋子机动性等更多信息，可以重构估值函数，使得其对局面的反映更加准确。
7. 开局库内容较少，可以提取更多局面，增加开局库的内容。
8. 单纯的暴力搜索的方式最造成一些算力的浪费

## 4.2 心得体会

1. 程序的编写要循序渐进，预先做好架构设计，从简单到复杂，从基础到拓展一步步完成，逐渐进行迭代，否则程序编制难度较大；
2. 要广泛学习各种方法，最大化利用现有资源，不要好高骛远，强化学习、MCTS看看就好。
3. 读别人的代码是学习的一个重要途径，从别人的代码中可以学到很多数据表示的方法，工具函数的构造方法等。
4. 一个大的项目需要合理拆分模块，有利于团队分工及工程构建，同时便于修改调试，也方便迁移。
5. 在编写程序的过程中，养成良好的Debug习惯，可以采用assert函数的方法，帮助debug。
6. 明确小组分工，使得各个模块功能明确，可以提高小组工作效率。
7. 一个大的任务只要认真的分析，广泛的了解最新的资料，最终也是可以做出来的。
8. 处理好系统的接口问题，经过广泛的了解和阅读之后，确定接口然后不要轻易改变接口。

## 4.3 后续改进方向

1. 增加后台思考功能，充分利用所有有效时间；
2. 使用位棋盘技术，提高运算效率；
3. 运用启发式策略改进估值函数；
4. 加入残局库，优化对残局的处理。
5. 将结果以图形化界面展示；
6. 优化搜索策略，提高运算效率。
7. 深入研究估值函数，使得估值函数更加多样，可以采用深度学习的方法，通过Ai和大师们的对弈，实现估值函数系数的学习。
8. 采用迭代更新的方式改善估值函数。
9. 可以适当的采用一些深度学习的技术进行剪枝方面的处理和判断。

## 4.4 总结

1. 搜索层数的深度对于决策十分的重要，增加搜索的深度对于改善象棋的效果有着重要的影响
2. 评估函数对于象棋的走向是有很大的影响，对于估值函数的创新和改善是十分重要和有意义的方向。
3. 想要增加搜索的深度，我们一定要在处理搜索任务的时候，充分的剪枝，减少任务的复杂度

# 5 成员分工与自评

表5.1 成员分工表

|  |  |
| --- | --- |
| 姓名 | *任务* |
| 温鑫 | 搜索功能的实现 |
| 杨晨 | UCCI通讯的实现 |
| 王森 | 棋盘结构体的实现 |
| 毕晓栋 | 估值函数和开局库的实现 |

表5.2 自评打分表

|  |  |
| --- | --- |
| 姓名 | 评分 |
| 温鑫 | 10 |
| 杨晨 | 9 |
| 王森 | 9 |
| 毕晓栋 | 10 |

# 参考文献

1. 中国象棋大师网. 中国象棋的起源. 2015-12-23. <https://baike.baidu.com/reference/278314/98c0ggYMmlZkjf_dA685U3twvWa6_ewkts-foyj01tcoy9IXEEXAZCeLheEWOYJo34OQlA8XvA9uq_HJ8BpZtQQ1-uLwnyrZ>
2. 象棋百科全书网.中国象棋程序设计探索[EB/OL]. <https://www.xqbase.com/protocol/cchess_ucci.htm>
3. 象棋百科全书网.象棋巫师[EB/OL].

<https://www.xqbase.com/xqwizard/xqwizard.htm>

1. Marsland T A , Campbell M . Parallel Search of Strongly Ordered Game Trees.[J]. Acm Computing Surveys, 1982, 14(4):533-551.
2. Reinefeld A. An improvement to the Scout tree search algorithm[J]. ICGA Journal, 1983, 6(4): 4-14.
3. Reinefeld A. Spielbaum-Suchverfahren[M]. Springer-Verlag, 2013.
4. Tabibi O D, Netanyahu N S. Verified null-move pruning[J]. ICGA Journal, 2002, 25(3): 153-161.
5. 象棋百科全书网.中国象棋程序设计探索[EB/OL]. http://www.xqbase.com/computer/eleeye\_heuristic.htm, 2007–05/2019–5–25.
6. 象棋百科全书网.电脑象棋循序渐进[EB/OL]. http://www.xqbase.com/computer/stepbystep5.htm, 2008–04/2019–5–25.