《对弈程序基本技术》专题

概述

象棋百科全书网 (webmaster@xqbase.com) 2005年9月

我花了大量时间翻译了F. D. Laramée的《国际象棋程序设计》(Chess Programming)连载，现在已经具备了设计象棋引擎的能力，但是还需要一些资料。为此，我查找了各个象棋程序设计的网站，收集了很多专题性的文章，并且整理成Laramée写他的连载时形成的体系。

我整理的文章除了T. A. Marsland的《国际象棋程序剖析》一文外，把它们分成四类，它们分别对应了Laramée的连载的四个部分。由于是从不同的网站上收集的，作者也各不相同，他们分别是：

- D. Eppstein，加州爱尔文大学(UC Irvine)的计算机教授，我收录了他1997到1999年做专题讲座的9篇文章，基本上是概述性的，也有细节上的东西，包括源代码(这可能是读者最感兴趣的)。
- J. Swafford，公开源代码的国际象棋引擎Galahad的作者(Galahad并不算很成功，现在已经销声匿迹的)，Laramée的连载中提到过他的文章，我收录的3篇文章(都是关于位棋盘的)则是一个叫Allen Liu的网友翻译的，其实肯定不止这3篇。不过现在他的网站现在已经关闭了，因此没有他的原文，这点让我非常遗憾。
- B. Moreland，微软(Microsoft)的程序设计师，业余从事国际象棋引擎Ferret的开发，在电脑奥林匹克大赛国际象棋业余组里，它和著名的Crafty一样处于领先水平。Moreland的文章涵盖的范围最广(除了局面评价以外都涉及到了)，所以我收集得最多。
- M. Fierz，瑞士Aargau学院的物理学家，国际象棋业余棋手，业余从事棋类程序的设计，作品有Muse(国际象棋)、Cake(西洋跳棋)等。我收录了Fierz写的关于残局库、开局库等其他方面的译文，使得整个专题更为完整。

一、数据结构

这部分内容极其丰富，不仅包括棋盘的表示方法，还包括着法产生当中需要用到的数据结构。我把关于“着法产生”的文章也并入其中，因为这部分内容实在很难归为一类，因此它代表了Laramée的连载的第二和第三部分。

1.1 数据结构——简介(D.Eppstein)；
1.2 数据结构——位棋盘(J.Swafford)；
1.3 数据结构——旋转的位棋盘(J.Swarford)；
1.4 数据结构——着法生成器(J.Swarford)；
1.5 数据结构——0x88着法产生方法(B.Moreland)；
1.6 数据结构——Zobrist键值(B.Moreland)。

二、基本搜索方法

目前象棋程序的搜索算法，都是在“带置换表的启发式Alpha-Beta搜索”基础上发展起来的，这就涵盖了最小-最大搜索、Alpha-Beta搜索、迭代加深和置换表四个内容，把他们归入基本搜索方法的范畴是非常恰当的。

2.1 基本搜索方法——简介(一)(D.Eppstein)
2.2 基本搜索方法——简介(二)(D.Eppstein)
2.3 基本搜索方法——简介(三)(D.Eppstein)
2.3 基本搜索方法——“最小-最大”搜索(B.Moreland)

三、高级搜索方法

这些方法都是“带置换表的启发式Alpha-Beta搜索”的扩展，其中静态搜索和空着裁剪是消除“水平线效应”的基本手段，而期望窗口、主要变例搜索(PVS)和MTD(*f*)都是Alpha-Beta搜索的改进。并非所有的棋类游戏都会用到这些方法，而且使用起来会有一些负作用，因此归入高级搜索方法一点也不过分。

四、局面评估函数

局面评估函数是对弈程序的核心，很少有特别详细的报道，通常引擎的作者是不愿意公开的。

五、其他策略

这里包含的内容比较杂，除了后台思考(属于时间控制的范畴)以外，其他方法都不能提高引擎的效率，但能时程序更完善，这些方法主要是针对国际象棋的。

- 上一篇 国际象棋程序设计(六)：局面评估函数
- 下一篇 数据结构——简介
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

棋盘的表示

David Eppstein */文
* 加州爱尔文大学(UC Irvine)信息与计算机科学系

要让机器下棋，程序首先必须对两个对象进行存储和操作，它们是局面和着法。这两个对象的表示使得程序可以进行下列操作：

(1) 执行一个着法(不仅仅是用户所指出的着法，而是在搜索过程中要用到的)；
(2) 撤消一个着法(作用同上)；
(3) 向用户显示棋盘；
(4) 产生所有可能的着法；
(5) 对棋盘的局面作出评估。

除了显示棋盘以外，所有的操作都应该尽可能地迅速，因为它们会在搜索的循环内被调用。(而显示棋盘的操作毕竟不是经常要做的。【译注：在UCI协议(国际象棋通用引擎协议)中，引擎从不显示棋盘，因此这个操作事实上是多余的。】)

着法的内部表示应该是非常简洁的(因为我们不需要花太多的时间来生成一系列着法)而且能快速解读，但是非常重要的是，它应该能代表所有的着法！在国际象棋中，机器内典型的着法表示，就是记录移动棋子的起点格子和终点格子，例如王前兵进两格就表示为“e2e4”(e2代表这个兵起点位置而e4代表终点位置)。不管是否吃子，被吃的棋子都不必记录，因为它可以由终点格子来判断。在机器中位置能表示为6位的数值，所以一个着法可以用2个字节表示。尽管很多程序都是这样表示的，但是这种表示方法不足以表示所有的着法！王车易位时，有两个子要移动，此时我们把它当作特殊情况来考虑，只记录王的移动。问题在于，当兵从第七行走到第八行时，可以升变为后、车、马或象，上述表示方法不能让我们指定升变为哪个棋子。因此在设计着法表示时需要非常仔细，要确认这种表示能够涵盖棋局中所有可能发生的特殊情况。

【一般而言，棋类的着法有两种形式，即添子式和移动式。五子棋、围棋、黑白棋等属于添子式，记录着法时只要记录棋子所下到的那个格子。其中五子棋最简单，下完的棋子不再会改变；黑白棋稍复杂些，下完的棋子可能会被后续着法所变换黑白，但每下一子棋盘上就多一子；围棋是最复杂的，由于存在提子的着法，所以局势是可逆的，打劫这样的着法需要很复杂的处理过程。

国际象棋和中国象棋都属于移动式，相比较而言中国象棋更简单，当一个棋子从起点移到终点时，只要简单地做一下终点的覆盖即可；而国际象棋由于三条特殊规则的存在而必须做特别处理，有时别的特殊位置的棋子要跟着移动(王车易位)，有时别的特殊位置的子要被吃掉(吃过路兵)，有时移动的棋子要被其他棋子所替代(升变)。】

棋盘的表示可能稍许复杂了些，但也不应该太庞大。它必须包括所有相关的信息，而不仅仅是表观上的，但无关的信息不包括在其中。例如在国际象棋里，我们必须知道棋子在棋盘上的位置(表观上的信息)，而且需要知道一些不可见的信息——现在是谁在走，每方是否还有易位权，哪个过路兵可以吃，从吃子或进兵到现在一共走了多少步。我们还需要知道以前发生过哪些重复的局面(因为三次重复局面即导致和棋)，而不需要知道以前所有的着法。

【在网络通讯中常常用一种称为FEN串的6段式代码来记录局面，在国际象棋中它的6段代码依次为：棋盘、走子方、王车易位权、吃过路兵的目标格、可逆着法数以及总回合数，基本上涵盖了国际象棋某个局面的所有信息。但是FEN字符串无法记录重复局面，因此UCI协议中规定，局面由棋局的最后一个不可逆局面(发生吃子、进兵或升变的局面)和它的后续着法共同表示，这样就涵盖了重复局面的信息。】

举例说明国际象棋中的诸多棋盘表示方法

国际象棋中有很多表示棋盘的方法，以下这些是程序中可能用到的：

一、棋盘格子的8x8数组

每个格子的值代表格子中的棋子(例如：enum { empty, wK, wN, wB, wR, wQ, wP, bK, bN, bR, bQ, bP })。它的优点在于：(1) 简单；(2) 容易计算子力价值：

```
for (i = 0; i < 8; i ++)
   for( j = 0; j < 8; j ++)
      score += value[square[i, j]];
```

计算可能的着法有些麻烦但并不非常困难，可以找遍棋盘的每个格子，根据棋子的颜色和类型来做分枝：

```
for (i = 0; i < 8; i++) {
   for(j = 0; j < 8; j++) {
      switch (board[i, j]) {
      case wP:
         if (board[i + 1, j] = empty) {
            生成到(i + 1, j)的着法;
         }
         if (i == 2 && board[i + 1, j] == empty && board[i + 2, j] empty) {
            生成到(i + 2, j)的着法;
         }
         if (j > 0 && board[i + 1, j - 1]有黑子) {
            生成吃(i + 1, j - 1)子的着法;
         }
         if (j < 7 && board[i + 1, j + 1]有黑子) {
            生成吃(i + 1, j + 1)子的着法;
         }
         break;
         ...
      }
   }
}
```

但是很多检测边界的判断(例如，兵在车一路就不能吃更外侧的子)，使得代码非常复杂，速度非常缓慢。

二、扩展数组

包括扩展边界的10x10数组，在棋子类型中应包括“boundary”这个值。这样就可以花很少的代价，来简化一些判断。【在下面提到的0x88方法问世以前，最普遍的做法却是用12x12的数组(即有双重的扩展边界)，这样连马的着法也不用担心跳出棋盘了。】

三、0x88

这个名称来自一个技巧——通过判断格子编码是否包含136这个数(在16进制中是0x88)来检测着法是否合理。下表就是格子的编码(用一个字节)，高4位表示行，低4位表示列。

120	121	**122**	123	**124**	125	**126**	127
96	**97**	98	**99**	100	**101**	102	**103**
80	81	**82**	83	**84**	85	**86**	87

64	**65**	66	**67**	68	**69**	70	**71**
48	49	**50**	51	**52**	53	**54**	55
32	**33**	34	**35**	36	**37**	38	**39**
16	17	**18**	19	**20**	21	**22**	23
0	**1**	2	**3**	4	**5**	6	**7**

这样，格子*i*的左边一格是*i* – 1，右边是*i* + 1，上边是*i* + 16，下边是*i* – 16，等等。棋盘被表示为128个格子的数组(其中64格代表棋盘上存在的格子)，这种表示的优势在于：(1) 数组仅用1个指标，这样写的程序要比用2个指标快；(2) 可以快速简单地判断某个格子*i*是否在棋盘上——当且仅当(i & 0x88) == 0时*i*在棋盘上。(因为列超出范围时*i* & 0x08不为零，而行超出范围时*i* & 0x80不为零。)这是一个非常有效而且常用的技巧。

四、位棋盘

我必须在这里多花些笔墨，因为这个技术还不被人所熟悉，但是我认为它可能会很好用的。以前用一个格子数组，每个元素含有一个棋子，现在取而代之的是一个棋子数组，每个元素代表棋盘上哪几个格子有这个棋子，按位压缩表示。由于棋盘上有64个格子，所以棋盘可以压缩成一个64位的字(即两个32位的字)。这种表示最大的优势在于，可以通过布尔操作(位操作)来加快局面评估和着法生成操作的速度。试想一下，如此烦琐的事情可以用压缩的字运算来解决，例如下面的局面：

白方的兵(这个64位数字记作wP)应该由这些位构成：

```
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 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 1
0 0 0 0 0 0 0 0
```

而被黑方占据的格子可以用下面的公式来计算：

```
bOcc = bP | bN | bB | bR | bQ | bK
```

(这里bP等等代表黑方不同兵种的棋子)，类似地可以计算白方占据的格子，还可以把这两个格子作“或”运算得到所有被占的格子。这些白兵前进一格可能到达的格子，可以用下面的公式计算：

single_pawn_moves = (wP << 8) & ~occupied

现在仔细看一下过程，先将wP左移8位，来产生每个兵前面的格子：

0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

然后求被占格子的“非”运算得到空的格子：

0 0 1 1 0 0 1 0
1 0 1 0 1 0 0 0
1 1 1 0 0 0 1 1
1 0 1 1 1 0 1 1
1 0 1 1 0 1 1 1
1 0 1 1 1 0 1 1
0 0 0 1 1 1 1 0
0 1 0 1 0 0 1 0

对以上两个位棋盘按位作“与”运算，就得到这些兵前面的没有被占的格子了：

0 0 0 0 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
0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

参照兵走一格的方法，可以找到兵走两格的着法，即再左移8位，“与”上没有子的格子，再“与”上一个只有第四行都占满的常数棋盘(这是兵走两格能到达的唯一一行)：

0 0 0 0 0 0 0 0
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
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

值得注意的是，这个常数棋盘是在编译的时候生成的，而不需要在每次着法生成的时候再计算出来。兵吃子的情况也类似，先左移7位或9位，再“与”上一个常数棋盘以过滤左边线或右边线的格子【对于左移7位产生吃右边子时，需要考虑子在右边线的情况，此时左移7位是同一行的左边线，因此需要排除这个格子，即“与”上一个常数棋盘，代表除左边线以外的所有格子】，最后“与”上bOcc【前面提到过这个棋盘，代表黑子所占的格子】。

位棋盘这个技术不能简化代码，但是它能一次就产生兵的所有着法，而不是一次只产生一个着法。此外，这个过程中有些表达式需要多次反复使用(例如bOcc)，但只要计算一次就可以了。因此，位棋盘最终变得非常高效，在棋子数比国际象棋少的棋类中，我想位棋盘的效果会更好。

有个复杂的问题产生了：计算位棋盘中有多少非零位，或者找到【最低或最高的】一个非零位(例如把兵可以到达的位棋盘转化为一系列着法)，这些操作往往非常重要。计算位数的操作可以一次计算一个字节，做一个长度为256的数组，每个元素代表它的指标所含有多少个非零位，然后通过查表来完成。在找到一个非零位的计算中有个技巧：$x \wedge (x-1)$(“^”代表异或)，会得到一个二进制为...000111...的数，$x \wedge (x-1)$的第一位就是x的最后一位非零位。如果要求得这个数字【$x \wedge (x-1)$，即型如...000111...的数】的第一位，就把它对某个特定的数M取余数(不同的...000111...这样的数对M取余数必须不同)，然后去查表。例如，以下的代码可以找到一个字节的最后一个非零位：

```
int T = { -1, 0, 7, 1, 3, -1, 6, 2, 5, 4, -1, -1 };
int last_bit(unsigned char b) {
   return T[(b^(b-1)) % 11];
}
```

【把原作者提到的这个技巧扩展到16位或32位的情况，不妨可以试探一下(只要找得到合适的除数即可)。但是原作者没有充分考虑计算机的运算特点，因此译者认为这不是一个合理的设计。由于“电脑一做除法就成了傻瓜”，所以代码中取余数的过程严重影响了效率，为此译者收集了一些使用炫技的程序，可以迅速完成求位数和查找位的操作。

(1) 求一个32位数中有几位非零位的运算——Count32操作：

```
int Count32(unsigned long Arg) {
   Arg = ((Arg >> 1) & 0x55555555) + (Arg & 0x55555555);
   Arg = ((Arg >> 2) & 0x33333333) + (Arg & 0x33333333);
   Arg = ((Arg >> 4) & 0x0f0f0f0f) + (Arg & 0x0f0f0f0f);
   Arg = ((Arg >> 8) & 0x00ff00ff) + (Arg & 0x00ff00ff);
   return (Arg >> 16) + (Arg & 0x0000ffff);
}
```

(2) 求最低位非零位是第几位的运算——Lsb32操作(LSB = Least Significant Bit)：

```
int Lsb32(unsigned long Arg) {
   int RetVal = 31;
   if (Arg & 0x0000ffff) { RetVal -= 16; Arg &= 0x0000ffff; }
   if (Arg & 0x00ff00ff) { RetVal -= 8; Arg &= 0x00ff00ff; }
   if (Arg & 0x0f0f0f0f) { RetVal -= 4; Arg &= 0x0f0f0f0f; }
   if (Arg & 0x33333333) { RetVal -= 2; Arg &= 0x33333333; }
   if (Arg & 0x55555555) RetVal -=1;
   return RetVal;
}
```

(3) 求最高位非零位是第几位的运算——Msb32操作(MSB = Most Significant Bit)：

```
int Msb32(unsigned long Arg) {
  int RetVal = 0;
  if (Arg & 0xffff0000) { RetVal += 16; Arg &= 0xffff0000; }
  if (Arg & 0xff00ff00) { RetVal += 8; Arg &= 0xff00ff00; }
  if (Arg & 0xf0f0f0f0) { RetVal += 4; Arg &= 0xf0f0f0f0; }
  if (Arg & 0xcccccccc) { RetVal += 2; Arg &= 0xcccccccc; }
  if (Arg & 0xaaaaaaaa) RetVal += 1;
  return RetVal;
}
```

对64位数字进行操作，把它分解成两个32位字，分别对两个字调用上面的函数就可以了。如果程序能运行在64位的处理器上，只要对上面三个函数略加改动就可以了。】

如何撤消着法?

还记得吗？我们说过在棋盘表示方法中需要涉及撤消着法的操作。现在有两种解决方案：(1) 用一个堆栈来保存棋盘，执行一个着法前先将棋盘压入堆栈，撤消着法就从堆栈弹出棋盘，恐怕这太慢了..... (2) 用一个堆栈来保存着法，执行一个着法前先将该着法及其相关信息压入堆栈，撤消着法就根据该着法还原棋盘及其相关信息。例如，在国际象棋中只要保存被吃的棋子(如果吃子的话)，以及王车易位和吃过路兵的权利等信息。

重复检测

某些棋类在发生重复局面时要用到特殊的规则，如围棋和国际象棋(在国际象棋中，第三次出现重复局面时，制造重复局面的一方就有权宣布和棋)。那么如何知道是否出现重复局面呢？答案很简单：用散列函数把局面转换成一个相当大的数(我们以后要谈到这个问题，因为这个技术还可以加快搜索速度)，然后保存一系列前面出现过的局面的散列值，从这些值里面找就是了。一个典型的散列函数，先随机产生一张64x13的表，如果棋子y在位置x上，就把表中$[x, y]$这个数加到散列值上(忽略溢出)[即Zobrist值]。值得注意的是，当棋子y从位置x走到位置z时，可以快速地更新散列值：减去$[x, y]$并加上$[z, y]$即可。

原文：http://www.ics.uci.edu/~eppstein/180a/970408.html
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 概述
- 下一篇 数据结构——位棋盘
- 返 回 象棋百科全书——电脑象棋

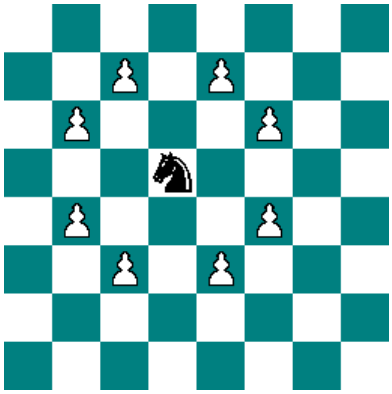
《对弈程序基本技术》专题

旋转的位棋盘

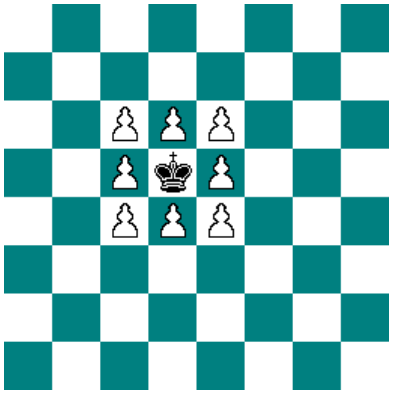
作者：James Swafford

“位棋盘”可以记录的最有用的棋盘状况之一就是“棋盘上哪些格子受到某一特定棋子的攻击”。幸运的是，这样的计算耗时不多。对于那些活动能力不强的棋子，这可以说非常容易(请看上节“位棋盘”)。原因是：当马，国王或兵在特定格子上时，无论周围情况怎样变化，所攻击的格子数目不变。你不用管那些格子上是否有棋子或者周围是否有特殊情况。

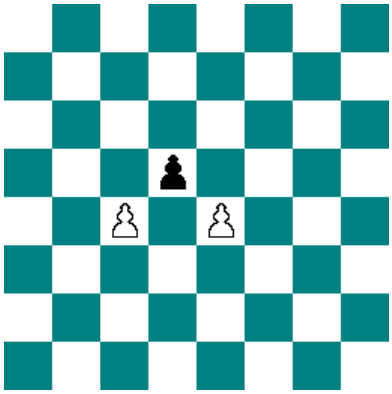
马在D5格可以攻击到的格子



国王在D5格可以攻击到的格子



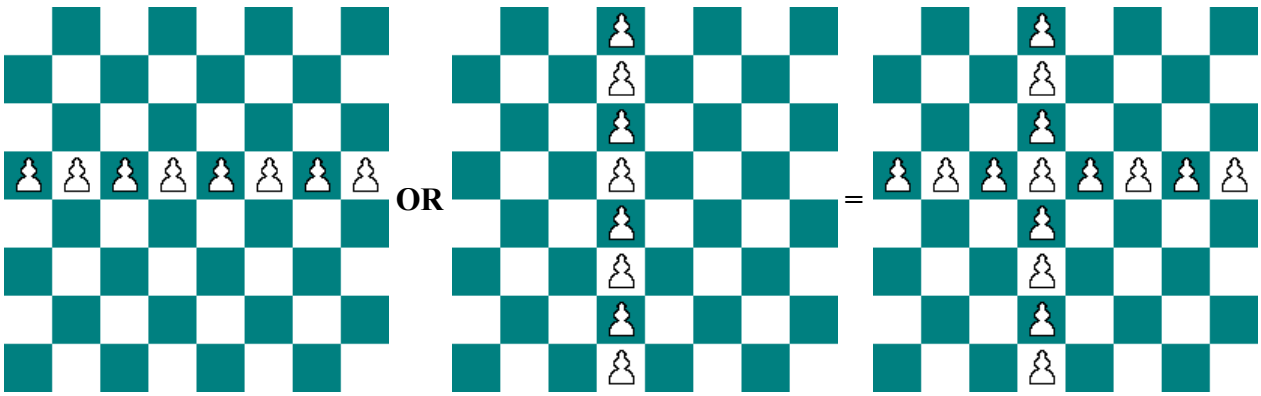
黑棋的兵在D5格可以攻击到的格子



但是对于车，象和皇后，事情就不那么简单了。例如，车攻击横线方向和竖线方向上的格子，直到在这两个方向遇到第一个有棋子的格子为止(包括这个有棋子的格子)；如果没有遇到有棋子的格子，则到棋盘边界为止。也就是说，车所在横线和竖线上的棋子分布情况不同，受到这个车攻击的格子的数量也不一样。因为每条横线或竖线都有8个格子，每个格子又有2种状态（有棋子或者没有棋子），所以每条横线或竖线都会有2^8(256)种棋子分布状态。

用旋转的位棋盘计算受车攻击的格子

如何获得整个棋盘上受到某个车攻击的格子呢？最简单的方法是：先计算出一个位棋盘记录车所在横线上受其攻击的格子，在计算出另一个位棋盘记录车所在竖线上受其攻击的格子，最后用“逻辑或”把这两个位棋盘“结合”到一起。



整个操作的关键在于“预先计算”。对于64格中的每一格，都预先计算出当车在这一格时，不同情况下横线上受到攻击的格子(共256种情况)和不同情况下竖线上受到攻击的格子(也是256种)。当要寻找某一情况下受车攻击的格子时，用横线(或竖线)上的“棋子分布状态”作索引从预先计算出来的位棋盘数组中取得。

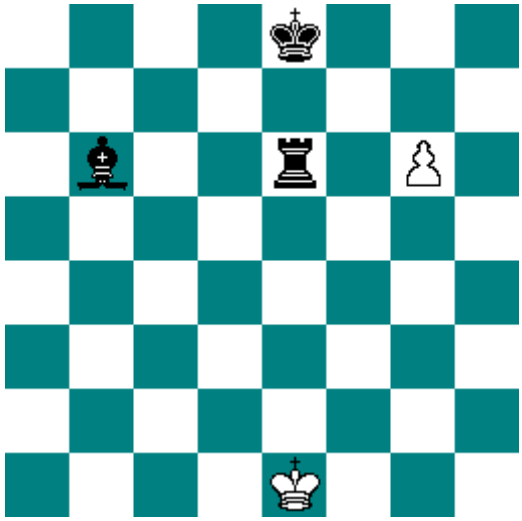
所以，第一步：预先计算出数组rank_attacks[64][256]和file_attacks[64][256]。

```
/* 车或后横向移动的预先计算 */
for (棋盘上的每一格)(0-63) {
   for (每行的棋子分布状态)(0-255) {
      计算该状态下被占的格子
      计算从这个格子朝左走的着法
      计算从这个格子朝右走的着法
      rank_attacks[格子][横线状态] = attack_board;
   }
}
/* 车或后纵向移动的预先计算 */
for (棋盘上的每一格)(0-63) {
   for (每条纵线的棋子分布状态)(0-255) {
      计算该状态下被占的格子
      计算从这个格子朝上走的着法
      计算从这个格子朝下走的着法
      file_attacks[格子][纵线状态] = attack_board;
   }
}
```

第二步：索引rank_attacks[64][256]，数组的第一个索引号为车所在格的编号。第二个索引号为车所在行上的“棋子分布状态”。

请看下面这个例子：

车在E6格可以攻击到的格子



第一个索引号是E6格的编号。在我的程序里，它的编号是20。第二个索引号是棋盘上第六行的棋子分布状态。

0 1 0 1 0 0 1 0

【编者注：在James Swafford的程序里，由于A8格编号为0，所以这串数据的顺序正好和棋盘相反，跟黑方看上去的顺序相同。】

要分离出上面这个数字，我们需要对位棋盘执行一个“位移(shift)”和一个“逻辑与”指令。在我的程序中，A8格的编号为0。如果车在第8行的任意格子上则不需要执行位移；若在第7行则要右移8位；在第6行要右移16位；以此类推。

需要右移的位数

0	0	0	0	0	0	0	0
8	8	8	8	8	8	8	8
16	16	16	16	16	16	16	16
24	24	24	24	24	24	24	24
32	32	32	32	32	32	32	32
40	40	40	40	40	40	40	40
48	48	48	48	48	48	48	48
56	56	56	56	56	56	56	56

我们需要第6行的棋子分布情况(从第16位到第23位)，就右移16位，现在它被右移到了最低的8位，我们把它与255(0xff)做逻辑与运算。所得到的东西即是我们需要作为rank_attacks的第二个索引号的数字。

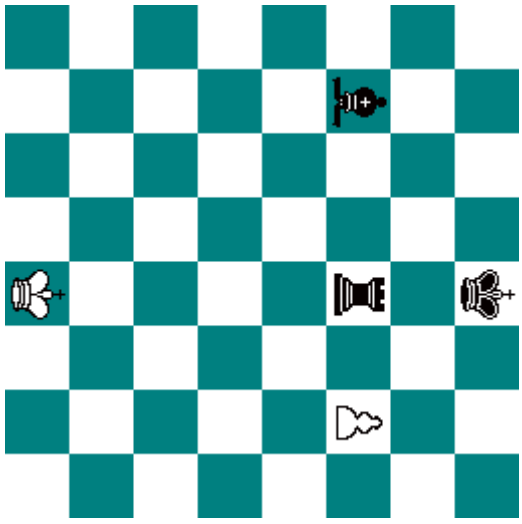
第三步：索引数组file_attacks[64][256]。

如何计算出file_attacks位棋盘呢？还是通过索引数组。和上面一样，第一个索引号是车所在格的编号。第二个索引号是车所在列的棋子分布状态。

1
0
0
0
0
1
0
1【编者注：同样道理，这是黑方看上去的位置。】

要分离出这个索引号比分离rank_attacks的要复杂一些。仅执行“位移”和“与”指令是无法做到的。首先，我们要把棋盘右转90度【编者注：即顺时针转90度。】。因此现在它看上去应该是这个样子：

右转90度后的棋盘



这个旋转后的棋盘是怎么得到的？你可以在开始做下一层搜索的时候构建这个棋盘，或者简单地在“走”下一步棋或“恢复”上一步棋的时候逐步更新它。下面的方法演示了如何在开始做下一层搜索时构建它。(代码并不那么优雅，但较易理解.....)

```
int r90R_map[64] = {
   H8, H7, H6, H5, H4, H3, H2, H1,
   G8, G7, G6, G5, G4, G3, G2, G1,
```

```
    F8, F7, F6, F5, F4, F3, F2, F1,
    E8, E7, E6, E5, E4, E3, E2, E1,
    D8, D7, D6, D5, D4, D3, D2, D1,
    C8, C7, C6, C5, C4, C3, C2, C1,
    B8, B7, B6, B5, B4, B3, B2, B1,
    A8, A7, A6, A5, A4, A3, A2, A1
};
BitBoard Rotated90R = 0;
for (棋盘上的每一格)(0-63) {
    if (格子被占)
        Rotated90R |= mask[r90R_map[square]];
}
```

这样，原来在A8格上的棋子被“映射”到H8格上，原来在H8格上的棋子则被“映射”到H1格上……

现在，我们可以对这个旋转后的位棋盘执行“位移”和“与”操作了。

第四步：获得rook_attacks位棋盘。

得到rank_attacks和file_attacks位棋盘后，你只要简单的对这两个位棋盘执行“与”运算就获得了记录了所有受到那个车攻击的格子的位棋盘了。

```
rook_attack = rank_attack | file_attack;
```

用旋转的位棋盘计算受象攻击的格子

计算受象攻击的格子的方法与车类似。对应于车的把横线与竖线做“或”运算，这里我们对象所在的两条斜线做“或”运算。我们把第一条斜线(就是执白时，从左上到右下这条斜线)叫做diag_A8H1；另一条斜线叫做diag_H8A1。相对车这里还多了一步，原因是每条斜线的长度不同。先预先计算出数组diag_A8H1_attacks[64][256]和diag_H8A1_attacks[64][256]。A8格编号为0，H8格为7，A1格为56，H1格为63。

```
int length_A8H1_diag[64] = {
    8, 7, 6, 5, 4, 3, 2, 1,
    7, 8, 7, 6, 5, 4, 3, 2,
    6, 7, 8, 7, 6, 5, 4, 3,
    5, 6, 7, 8, 7, 6, 5, 4,
    4, 5, 6, 7, 8, 7, 6, 5,
    3, 4, 5, 6, 7, 8, 7, 6,
    2, 3, 4, 5, 6, 7, 8, 7,
    1, 2, 3, 4, 5, 6, 7, 8
};
int length_H8A1_diag[64] = {
    1, 2, 3, 4, 5, 6, 7, 8,
    2, 3, 4, 5, 6, 7, 8, 7,
    3, 4, 5, 6, 7, 8, 7, 6,
    4, 5, 6, 7, 8, 7, 6, 5,
    5, 6, 7, 8, 7, 6, 5, 4,
    6, 7, 8, 7, 6, 5, 4, 3,
    7, 8, 7, 6, 5, 4, 3, 2,
```

```
    8, 7, 6, 5, 4, 3, 2, 1
};
/* 象或后在A8-H1方向移动的预先计算 */
for (棋盘上的每一格)(0-63) {
    /* 状态数会随对角线长度而变化 */
    for (每条对角线的棋子分布状态)(0-(2^对角线长-1)) {
        计算该状态下被占的格子(注意某些状态不要把8个格子都考虑进去)
        计算从这个格子朝左上方走的着法
        计算从这个格子朝右下方走的着法
        diag_A8H1_attacks[格子][状态] = attack_board;
    }
}
/* 象或后在H8-A1方向移动的预先计算 */
for (棋盘上的每一格)(0-63) {
    /* 状态数会随对角线长度而变化 */
    for (每条对角线的棋子分布状态)(0-(2^diag length)-1) {
        计算该状态下被占的格子(注意某些状态不要把8个格子都考虑进去)
        计算从这个格子朝右上方走的着法
        计算从这个格子朝左下方走的着法
        diag_H8A1_attacks[square][diag state] = attack_board;
    }
}
```

索引 diag_A8H1_attacks[64][256] 数组。

不说你也应该知道，第一个索引号是象所在格的编号。第二个索引号是它所在A8->H1方向斜线上的棋子分布情况。没错，接下去又要旋转位棋盘，位移，最后用“逻辑与”分离出我们需要的东东。为了对齐 A8->H1 斜线上的格子，我们要把棋盘左转45度。(现在我建议你去拿罐可乐，外加一些提神的药物……)

```
int r45L_map[64] = {
    28, 21, 15, 10, 6, 3, 1, 0,
    36, 29, 22, 16, 11, 7, 4, 2,
    43, 37, 30, 23, 17, 12, 8, 5,
    49, 44, 38, 31, 24, 18, 13, 9,
    54, 50, 45, 39, 32, 25, 19, 14,
    58, 55, 51, 46, 40, 33, 26, 20,
    61, 59, 56, 52, 47, 41, 34, 27,
    63, 62, 60, 57, 53, 48, 42, 35
};
```

注意每个格子的映射次序，是从右上往左下的。这样映射后的位棋盘就会沿着A8-H1斜线对齐了(所有的斜线都是沿着这个方向走的)。

```
BitBoard Rotated45L = 0;
for (棋盘上的每一格)(0-63) {
    if square is not empty
    otated45L |= mask[r45L_map[square]];
}
```

现在“右移”这个棋盘，但是移动几位呢?

需要右移的位数

28	21	**15**	10	**6**	3	**1**	0
36	**28**	21	**15**	10	**6**	3	**1**
43	36	**28**	21	**15**	10	**6**	3
49	**43**	36	**28**	21	**15**	10	**6**
54	49	**43**	36	**28**	21	**15**	10
58	**54**	49	**43**	36	**28**	21	**15**
61	58	**54**	49	**43**	36	**28**	21
63	**61**	58	**54**	49	**43**	36	**28**

位移完成后，最后一步就是用“屏蔽模版”将需要的数据分离出来。“屏蔽模版”根据斜线的长度不同而变化。

```
Mask_Length = (2 ^ diag_length) - 1;
```

按照这个公式，当斜线的长度为7时，屏蔽模版等于127(二进制：1111111b)。如果斜线的长度为1时屏蔽模版也为1。在程序中你可以用这个公式，但是使用查询表会更快一些。

索引 diag_H8A1_attacks[64][256] 数组。
为了对齐H8->A1方向的斜线，我们要把棋盘向右旋转45度。(再来一杯可乐？)

```
int r45R_map[64] = {
   0, 1, 3, 6, 10, 15, 21, 28,
   2, 4, 7, 11, 16, 22, 29, 36,
   5, 8, 12, 17, 23, 30, 37, 43,
   9, 13, 18, 24, 31, 38, 44, 49,
   14, 19, 25, 32, 39, 45, 50, 54,
   20, 26, 33, 40, 46, 51, 55, 58,
   27, 34, 41, 47, 52, 56, 59, 61,
   35, 42, 48, 53, 57, 60, 62, 63
};
BitBoard Rotated45R = 0;
for (棋盘上的每一格)(0-63) {
   if square is not empty
      Rotated45R |= mask[r45R_map[square]];
}
```

需要右移的位数

0	1	**3**	6	**10**	15	**21**	28
1	**3**	6	**10**	15	**21**	28	**36**
3	6	**10**	15	**21**	28	**36**	43
6	**10**	15	**21**	28	**36**	43	**49**
10	15	**21**	28	**36**	43	**49**	54
15	**21**	28	**36**	43	**49**	54	**58**
21	28	**36**	43	**49**	54	**58**	61

28	36	43	49	54	58	61	63

最后，像对第一根斜线所做的那样，分离出所需数据。
对这两个位棋盘做“逻辑或”，便得到记录受象攻击的格子的位棋盘。

```
bishop_attack = diag_A8H1_attacks | diag_H8A1_attacks;
```

最后，要得到受皇后攻击的格子的位棋盘，只要把受车攻击的格子“或”上受象攻击的格子：

```
queen_attack = rook_attack | bishop_attack;
```

出处：不详
译者：Allen Liu (http://lostboy.myrice.com/)
类型：不详
编辑：象棋百科全书网 (webmaster@xqbase.com)

- 上一篇 数据结构——位棋盘
- 下一篇 数据结构——着法生成器
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

着法生成器

作者 James Swafford

“着法生成器”在不同的引擎中差异较大。本文将向你演示使用位棋盘(aka bitmaps)【编者注：即前两章所说的BitBoard】时生成着法的基本原理。高级的国际象棋引擎通常具备一次只生成一小部分着法的能力。例如，仅生成象走的着法，马走的着法，导致升变的着法，所有的吃子着法等等，这正是位棋盘的强项。那为什么用这种方式生成着法呢？原因是生成着法耗费一定的时间。如果你的引擎在检查了一部分着法后发现了必须走的棋，那它就无需生成余下的棋步了。因此，你可能想先生成所有吃子的着法，如果没有满意的棋再生成余下的着法。(用来减少耗时的着法生成策略很多——发挥你的想象力吧)。

大名鼎鼎的免费国际象棋引擎Crafty(其作者是Robert Hyatt博士)使用三个着法生成函数。一个用来生成所有伪合法吃子着法，一个生成所有伪合法不吃子着法，最后一个生成所有摆脱被将军状态的着法。注意前两个函数生成的是伪合法的着法。就是说，这些函数生成的着法并非都是合法的。例如，你要生成所有将军的着法并且发现了一步你想走的棋，但随后发现这步不合法再把它抛弃。这看起来很奇怪，但它确实比那种在所有局面下都严格生成合法着法的策略更快！Hyatt博士曾经这样解释：当国王被将时，你需要生成摆脱被将的着法，这时大部分生成的着法是不合法的，在这种局面中你使用生成所有合法着法的策略会帮你节省时间；但在大多数局面中，生成的着法都是合法的，推迟验证合法性会更有效率。

记住上面讲的内容，让我们看一些例程！下面这段代码选自我目前正在开发的引擎(其设计理念同时受到了Crafty和Tom's Simple Chess Program的影响)。它生成伪合法的马吃子的着法。

```
gen_rec *GenWhiteKnightCaps(chess_pos *posptr, gen_rec *m) {
  piece_map = posptr -> whiteknights;
  while (piece_map) {
    from_sq = MSB(piece_map);
    piece_map ^= mask[from_sq];
    to_map = nmoves[from_sq] & posptr -> blackpieces;
    while (to_map) {
      to_sq = MSB(to_map);
      to_map ^= mask[to_sq];
      m -> m.b.from = from_sq;
      m -> m.b.to = to_sq;
      m -> m.b.promote = 0;
      m -> m.b.bits = 1;
      m -> score = piece_val[posptr -> piece[to_sq] + 7] - KNIGHT_value;
      m ++;
    }
  }
  return m;
}
```

gen_rec是记录着法和它的得分的数据结构。上面的函数获得的第一个参数是一个指针指向需要从中生成着法的位棋盘的位置。第二个参数同样是一个指针，指向着法栈(储存生成着法的栈)中下一个存放着法的位置。这个函数返回一个指针，指向着法栈中下一个可以存放着法的位置。

Piece_map是一个位棋盘，最初记录所有白棋马的位置。然后，这些“位”(对应于棋盘上有白棋马的格)依次被解析出来。解析出来的“位”被用来生成另一个位棋盘，它记录了白马在该“位”(格子)上时所有受其攻击且上面有黑棋的“位”(格)。这些“位”再被依次取出，成为着法，被压入着法栈，附带存入这步棋的得分(虽然我是这样做的，但并非必要)。

我们再来看下面的函数。它被设计成生成黑棋象和皇后的伪合法、不吃子着法。注意它并未生成所有皇后的不吃子着法，仅仅是它的沿着斜线走的着法。

```
gen_rec *GenBlackBishopQueenNonCaps(chess_pos *posptr, gen_rec *m) {
  piece_map = posptr -> blackbishops | posptr -> blackqueens;
  while (piece_map) {
    from_sq = LSB(piece_map);
    piece_map ^= mask[from_sq];
    to_map = BishopMoves(from_sq, posptr) & ~(posptr -> whitepieces | posptr ->
blackpieces);
    while (to_map) {
      to_sq = LSB(to_map);
      to_map ^= mask[to_sq];
      m -> m.b.from = from_sq;
      m -> m.b.to = to_sq;
      m -> m.b.promote = 0;
      m -> m.b.bits = 0;
      m -> score=history[from_sq][to_sq];
      m++;
    }
  }
  return m;
}
```

这个函数同上一个相比并没有太多的不同。它接受一样的参数，返回指向最后一个被压入的着法的下面一个位置的指针。

Piece_map位棋盘最初记录所有黑棋象和皇后的位置，这些“位”被依次析出；生成棋子在该“位”上所有走斜线的着法的位棋盘，加上着法的得分并压入着法栈。

但这两个函数是存在一些不同的，例如这个函数中用“LSB()”来析取“位”，LSB(Least Significant Bit)意思是最低位。我在这个函数中用LSB是因为黑棋棋子靠近A8格，在我的程序中A8格编号为0。相反，白棋则靠近H1，它在程序中的编号为63，所以在生成白棋着法的例程中我使用MSB(Most Significant Bit，最高位)。

另一个区别是评分算法。注意，在生成着法的时候就给每个着法打分并非必要，只是我的程序中是这样做的。在生成吃子着法的例程中所使用的评分策略是MVV/LVA算法(最有价值的受害者/最没有价值的攻击者)。其实就是每步棋的得分等于被吃子的价值减掉吃它的子的价值。对于不吃子着法的例程，使用的是历史启发算法。历史启发算法将在以后给你介绍。

下面的代码将生成给定一方的所有伪合法的吃子着法。

```
gen_rec *GenCaps(chess_pos *posptr, gen_rec *m) {
  if (posptr -> player) {
    // white to move
    m = GenWhiteKnightCaps(posptr, m);
    m = GenWhiteBishopCaps(posptr, m);
    m = GenWhiteRookCaps(posptr, m);
    m = GenWhiteQueenCaps(posptr, m);
```

```
        m = GenWhiteKingCaps(posptr, m);
        m = GenWhitePawnCaps(posptr, m);
      } else {
        // black to move
        m = GenBlackKnightCaps(posptr, m);
        m = GenBlackBishopCaps(posptr, m);
        m = GenBlackRookCaps(posptr, m);
        m = GenBlackQueenCaps(posptr, m);
        m = GenBlackKingCaps(posptr, m);
        m = GenBlackPawnCaps(posptr, m);
      }
      return m;
    }
```

这个函数非常“直观”。它获得两个指针，分别指向棋盘位置和着法栈中相应的位置。然后生成走棋一方的所有棋子的吃子着法。最后返回一个指针，指向着法栈的下一个空位(就是你压入的最后一个着法的下面一个位置)。

现在你完全可以开始动手编写你自己的国际象棋引擎了！如果你想看更多的代码，请看下面。祝你好运！

出处：不详
译者：Allen Liu (http://lostboy.myrice.com/)
类型：不详
编辑：象棋百科全书网 (webmaster@xqbase.com)

- 上一篇 数据结构——旋转的位棋盘
- 下一篇 数据结构——0x88着法产生方法
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

0x88着法产生方法

Bruce Moreland / 文

历史

我在1995年香港举行的世界电脑国际象棋锦标赛(WCCC)上和 David Kittinger 交流时，他跟我说了这个着法产生的技术，当时我就把它忘了。如今回过头来看，我已经很多次提到这个技术了。由于当时我还不知道它的名称，我就给它起了别的名字。它名称应该是0x88，即十六进制的88。

之所以称为0x88，是因为0x88概括了这个技术的要点。

0x88技术的优势在于：

(1) 它很容易理解；

(2) 代码非常短小，一点也不复杂；

(3) 很容易检查棋子是否超出边界。

棋盘表示和基本原理

通常的国际象棋棋盘是8x8的格子，格子的“标准”编号应该从0到63，a1格是0，b1格是1，a2格是8，h8格是63，其余的格子由你自己填上。

0x88采用稍微不同的棋盘，它有128个格子，a1到h1仍旧是0到7，但是在h列右边还有从未用到的i到p列，简单地说就是把一个虚的棋盘放在实的棋盘的右边。

因此a2被编为16，a8是112，h8是119。

任意一个格子都符合下面的公式：

```
指标 = 行号 x 16 + 列号
```

你可能会问为什么要这样做，其实有很多理由，这里我们只讨论最关键的。这个理由就是，这样做可以检查射线是否走到了棋盘的左边界或右边界。

你可能仍旧不明白。假设你仍然用8x8的棋盘，并且考虑a3格的车，在8x8的棋盘上它的编号是16。现在来产生这个车的目标格子，先从纵线上的着法开始。在纵线上前进一格，就增加8，16加上8就得到24。这个格子是否在棋盘上呢？在8x8的棋盘上可以检查它是否小于64。现在24小于64，所以它在棋盘上。下一个格子是32，然后依次是40、48和64，64不小于64，所以它在棋盘外边，我们就停了下来。

非常好，现在我们从a3往下走。a3是16，减去8得到8，它在棋盘上吗？此时要检查它是否大于或等于零，而8确实是的。再下一个格子是–8，因为小于零，所以不在棋盘上。

我们不厌其烦做了两次测试，如果只需要做一次测试那就更好了。现在测试棋子是否在棋盘上的代码是：

```
if ((index < 0) || (index >= 64))
```

其实这就包含了两次测试，我们远远没有满足，因为它完全可以只做一次测试：

```
if (!(index & 0x40))
```

这就涵盖了判断棋子超出棋盘下边界和超出棋盘上边界的两种情况。超出棋盘上边界时，由于大于64而0x40置为1，超出棋盘下边界时，用补码表示负数的机制使得0x40也置为1。

如果你还没有明白，那么你最好停下来别看下去了，否则下面要说的东西对你来说就是天书。

如果你留意一下，你会注意到我们只让车朝上朝下走，而没有让它朝左朝右走。我们之所以不让它朝左朝右走，是因为这套机制不允许这么做，它无法判断朝左或朝右的射线是否到达棋盘的边界。如果你从a3开始朝右走，每走一格加1直到h3，此时你可以继续加1到达a4。a4仍就在棋盘上，然而没有什么技巧可以让你判断是否超越了h列。朝左走也一样，从a3格开始减1，就会到h2，它仍然在棋盘上，但是国际象棋里没有哪个棋子能这么走。

而0x88机制恰恰可以解决这个问题。用一个16x8的棋盘，就得到一个标志位，你就能知道棋子是否到了右边那个没用的棋盘上，因为在这个棋盘上0x08位置为1。例如h1标号是7，加1后就是8，而0x08位变成了1。左边的实棋盘上没有一格的0x08位是1，而右边的虚棋盘每个格子的0x08位都是1。如果你在a3并且要朝左走，你会到达p2，这是在虚棋盘上，0x08位是1。

可以把0x08的检测和0x80的检测结合起来(原来的0x40变成了0x80，因为现在棋盘变成128个格子了)，并且两次测试要同时完成。把0x80和0x08结合起来就是0x88，这就是这套方案的名称。

如果你知道我在说什么，你就明白0x88是怎样运作的。你的着法产生的循环就应该写成下面的样子：

```
while (!(index & 0x88)) {
  GenerateMove(index);
  index += delta;
}
```

这就非常简洁了，你可以这么写：

```
void GenerateMoves(int square, int * ptab) {
  for (; *ptab; ptab++) {
    int index;
    for (index = square; !(index & 0x88); index += *ptab) {
      GenerateMove(index);
    }
  }
}
int argdBishop[] = { 17, 15, -17, -15, 0 };
void GenerateBishopMoves(int square) {
  GenerateMove(square, argdBishop);
}
```

这样写就非常快了，并且代码很短。车或者后跟象的区别只是表中的数据不同罢了，因此你可以花很短的时间把其他棋子的代码加上，而且不需要任何改动。

当然，你仍然需要为兵和王车易位另写代码，但每个体系都得如此。0x88体系能给你一点帮助，可是代码写出来仍然很丑。

奖励

0x88体系还可以快速判断攻击，这就是要使用16x8棋盘的另一个原因。如果你将两格子的号码相减，你会得到两个格子之间的关系。

例如，如果两个格子减下来是1，那么第二个格子就在第一个格子的左边。如果减下来是16，那么第二个格子就在第一个格子的上面。

这在8x8棋盘上是做不到的。d1 – c1 = 1确实如此，但是a2 - h1也是1。这个“回圈”问题可以在128个格子的棋盘上解决。

你在写判断将军或者一个子是否在捉另一个子的函数时，可以利用以上这个技巧。

你可以用一个大约257个元素(−128 ~ +128)的数组，里面存放一些代表棋子的位域，来说明哪些棋子能在某两格间移动，而移动的增量就是数组的指标。你可以用少于257个的元素，但是我没有尝试过。

例如在增量为1的元素里，应该有王、后、车的位置是置1的。如果增量是17，那么置1的是王、后、象和白兵。

这样就可以写出比较快的检查将军的程序了。你把攻击子的格子和目标格子相减得到增量，加上128以避免负的指标，然后去找预先计算好的攻击数组，看看是否符合这个攻击子的类型，以确认它有可能一步到达目标格。

如果你确认这个子可能到达目标格，那么你还要检查它是否是长兵器(后、车或象)。如果不是，那就完成判断了，否则你要沿着从攻击子到目标格的射线去找有没有阻挡的棋子。

我不想提供这个程序的代码，因为它很容易写。这个程序的速度是比较快的。

原文：http://www.seanet.com/~brucemo/topics/0x88.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译

- 上一篇 数据结构——着法生成器
- 下一篇 数据结构——Zobrist键值
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

Zobrist键值

Bruce Moreland / 文

比较局面的方法

国际象棋局面包含了棋盘上的棋子、哪一方走棋、是否能易位、是否能吃过路兵等信息。

在写国际象棋的程序时，需要比较两个局面看它们是否相同。如果你比较每个棋子的位置，或许不需要花很多时间，但是实战中你每秒种需要做成千上万次比较，因此这样会使比较操作变成瓶颈的。另外，需要比较的局面数量多得惊人，要存储每个棋子的位置，需要占用非常大的空间。

一个解决方案是建立一个标签，通常是64位。由于64位不足以区别每个局面，所以仍然存在冲突的标签。但是实战中这种情况非常罕见，你可以有充分的把握不会发生冲突。

用32位是否足够，目前争论很多，而用64位通常是明智的。

Zobrist键值是一个常用的建立标签的方法。

实现

你必须从多维的64位数组开始，每个数组含有一个随机数。在C语言中，“rand()”函数返回一个15位的值(0~32767)，所以要得到64位的随机数还需要再加工一下。实际上我已经为你做好了，这个64位随机数的函数是：

```
U64 rand64(void) {
   return rand() ^ ((U64)rand() << 15) ^ ((U64)rand() << 30) ^ ((U64)rand() << 45) ^
((U64)rand() << 60);
}
```

这个函数用来填满一个“U64”的元素(你需要自己来定义这个类型，这取决于你的编译器，试一下“long long”或者“__int64”)，它是通过调用一系列“rand()”来实现的。

无论如何你的数组要有三维：棋子的类型、棋子的颜色和棋子的位置：

```
U64 zobrist[pcMAX][coMAX][sqMAX];
```

程序启动时就把这个数组用随机数填满，随机数是必须非常随机的，我看到Usenet(新闻组网络系统)上很多人说“rand()”用在这里随机性不是很好，但是我一直都用“rand()”并且没有出现问题。如果你想使数组更随机，就去找更强大的随机函数，但是你要确保它的随机性不比“rand()”差。

要为一个局面产生Zobrist键值，首先把键值设成零，然后找棋盘上的每个子，并且让键值跟“zobrist[pc][co][sq]”做异或(通过“^”运算符)运算。

如果局面由白方走，那么别去动它，如果是黑方走，你还要在键值上异或一个64位的随机常数。

为什么Zobrist键值特别有用

用Zobrist技术产生的键值，表面上跟局面没什么关系。如果一个棋子动过了，你会得到完全不同的键值，所以这两个键值不会挤在一块儿或者冲突。当你把它们用作散列表键值的时候会非常有

效。

另一个优点在于，键值的产生是可以逐步进行的。例如，你的白兵在e5格，那么键值里一定异或过一个“zobrist[PAWN][WHITE][E5]”。如果你再次异或这个值，那么根据异或的工作原理，这个兵就从键值里删除了。

这就是说，如果你有当前局面的键值，并且需要把白兵从e5移到e6，你只要异或一个“白兵在e5”的键值，把兵从e5格移走，并且异或一个“白兵在e6”的键值，把兵放在e6上。比起从头开始一个个棋子去异或，这样做可以得到同样的键值。

如果你要改变着子的一方，只要异或一个“改变着子方”的键值就可以了。你可以用类似的方法处理王车易位和吃过路兵的标志。

用这种方法，你可以在搜索根结点的时候构造一个Zobrist键值，在搜索时通过走子函数“MakeMove()”来更新键值，一直让它保持和当前局面同步。

Zobrist键值的用途

Zobrist键值通常用在散列键值当中，而散列键值在国际象棋程序里有以下几个作用：

(1) 用Zobrist键值来实现置换表。置换表是一个巨大的散列表，来保存以前搜索过的局面，让你节省很多搜索的时间。如果你需要对某个局面搜索9层，你可以从置换表中查找该局面，如果它已经搜索过9层，那么你不必去重复搜索。置换表的一个并不起眼的作用是，它可以帮助你改善着法的顺序。

(2) 用Zobrist键值来实现兵型的散列表。你可以用一个键值只记录棋盘上的兵，对兵型做了很复杂的分析后，在散列表中记录分析的结果。在实战中兵型是很少发生变化的，所以这个散列表的命中率会非常高，它可以为你评估兵型节省很多时间。

(3) 可以用Zobrist键值制造一个很小的散列表，来检测当前着法路线中有没有重复局面，以便发现长将或其他导致和局的着法。

(4) 可以用Zobrist键值创建支持置换的开局库。

原文：http://www.seanet.com/~brucemo/topics/zobrist.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译

- 上一篇 数据结构——0x88着法产生方法
- 下一篇 基本搜索方法——简介(一)
- 返　回 象棋百科全书——电脑象棋

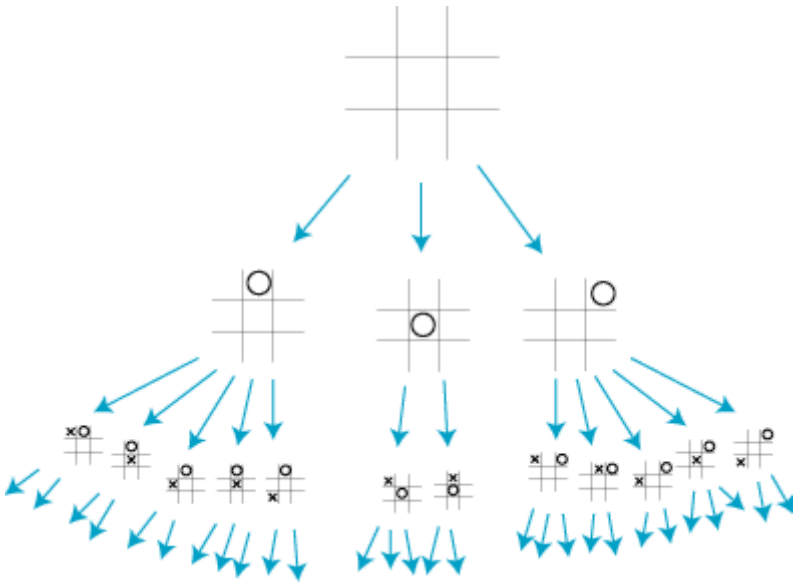
《对弈程序基本技术》专题

最小-最大和负值最大搜索

David Eppstein */文
* 加州爱尔文大学(UC Irvine)信息与计算机科学系

搜索树

任何棋类游戏都要定义一棵有根的树(即“博弈树”)，一个结点就代表棋类的一个局面，子结点就是这个局面走一步可以到达的一个局面。例如下图是井子棋(Tic-tac-toe)的搜索树：



(实际上，这个搜索树的根结点应该有9个子结点，但是我去掉了一些对称的情况。如果同样的棋盘是由两个不同的着法顺序形成的，那么我们就建立两个结点，所以这的确是树的结构。稍后我们会在讨论散列技术的时候谈到如何利用重复的结点来提高搜索速度——我们只要搜索第一个，另一个就用第一个搜索结果来代替。另外我们假设棋手是轮流下棋的，没有人一次走多步或跳过不走的，那些复杂的情况可以把它走的一系列着法看作一个着法来处理。【译注：复杂的情况是指一些一次能走很多步的棋类游戏，例如跳棋、西洋跳棋、黑白棋等，按照原作者的方案，可以把一方连续走的几步棋看成一步棋。而译者更愿意把一方连续的几步棋拆成几个回合，只是另一方都别无选择地走了空着。】最后，我们假设搜索树是有限的，这样我们就不会遇到永无止境的棋局或者一步有无限多种着法的棋局。)

搜索树中有三种类型的结点：

(1) 偶数层的中间结点，代表棋手甲要走的局面；

(2) 奇数层的中间结点，代表棋手乙要走的局面；

(3) 叶子结点，代表棋局结束的局面，即棋手甲或棋手乙获胜，或者是和局。

博弈树的评价

假设某个中间结点的所有子结点都是叶子结点，那么棋局会在一回合内结束。现在我们假设棋手会挑选最好的着法，如果有一个着法能使他赢下棋局，那么他一定会走这步。如果没有可以赢的着法，但是有取得和局的着法，那么他会走这步取得和局的着法。但是，如果所有的着法都使得对手获胜，那么无论如何他都会输。

因此在叶子结点的上一层结点，我们就能知道棋局的结果。现在我们知道了这个结点的结果，那么我们可以用同样的方法作推演，知道叶子结点的上两层结点的结果，然后是上三层结点，等等，直到我们达到搜索树的根结点。在每个结点上，棋手只要找到一个子结点能让他获胜，那么他就可以赢下棋局；他只要找到一个形成和局的子结点，棋局就和了；如果获胜与和局的子结点都没有，那么肯定是输的。如果我们有足够多的时间来计算，那么这就给了我们一个可以下棋的完美算法。但是对于任何常规的棋类游戏，我们都不可能有足够的计算时间，因为搜索树实在太大了。

另外，“正确”的评价函数只有三个值，赢、输或者和局。在实际的棋类程序中，我们通常使用一个更宽泛的实数来作评价值，就是因为赢、输或者和局是不确定的。如果棋手甲获胜的值用+1表示，和局的值用0表示，棋手乙获胜的值用–1表示，那么博弈树的每个中间结点的值就是子结点的最大值或最小值，这取决于棋手甲还是棋手乙着棋。

部分的博弈树

在实战中，我们的搜索算法只能对博弈树展开一部分。我们用一些“中止规则”来决定搜索树展开到哪个结点就停下来，例如我们在8步变化以后听下来。由于棋局没有在叶子结点结束，我们只能用评价函数来猜哪一方获胜。现在我们来假设在我们展开的结点中，棋手甲总是希望棋局到达评价函数大的局面，而棋手乙总是希望棋局到达评价函数小的局面。

如果双方都用这种方法来下棋，那么我们可以使用同样的最小-最大过程，来确定到达的叶子结点的评价值，这个过程如下：对每个中间结点，计算子结点的最大值或最小值，这取决于是棋手甲还是棋手乙走棋。到达叶子结点的线路称为“主要变例”(Principal Variation)。最小-最大博弈树的基本原理，就是对博弈树作部分展开，去找主要变例，并走出变例中的第一步。

广度优先和深度优先搜索，负值最大代码

正如上面所讲的，计算博弈树的值是一个广度优先的过程(我们要自下而上地，一次一层地计算)。然而实战中，我们使用深度优先(即后序遍历)的递归过程来遍历搜索树，即要得到一个结点的值，就对子结点做递归，然后根据它们的返回值来决定自身的返回值。这样要节省很多空间，因为不需要来存储整个博弈树，而只是存储一条线路(相对来说非常短，例如上面提到的8步中止的规则)。下次我们讨论Alpha-Beta搜索时，会发现深度优先的遍历会有很大的优势，你可以用目前得到的信息来决定某些结点是不需要访问的，这样就节省下很多的时间。

只要对搜索树的值作一个很小的改动，就可以用一个求最大值的操作来代替最小值和最大值交替的过程。在搜索树的奇数层(即轮到棋手乙下棋的结点)，就对上面提到的评价值取负数。因此在每个结点上，这些值都可以由子结点的负值求得。我把博弈树搜索的源代码写出来，恐怕就会清楚很多了。

```
// 将博弈树搜索到一定的深度，返回根结点的评价值
double negamax(int depth) {
  if (depth <= 0 || 棋局结束)
    return eval(pos);
  else {
    double e = -infty;
    for (当前局面所有可能的着法 m) {
      执行着法 m;
      e = max(e, -negamax(depth - 1));
      撤消着法 m;
    }
    return e;
  }
```

```
}
```

注意，这个过程只能找到评价值，但是无法得知着法。我们只要在搜索树的根结点找到着法(尽管很多程序都返回整个主要变例)就可以了，要做到这一点，可以对根结点的搜索稍作改动：

```
// 将博弈树搜索到一定的深度，返回根结点的着法
move rootsearch(int depth) {
  double e = -infty;
  move mm;
  for (当前局面所有可能的着法 m) {
    执行着法 m;
    double em = -negamax(depth - 1);
    if (e < em) {
      e = em;
      mm = m;
    }
    撤消着法 m;
  }
  return mm;
}
```

负值最大的分析：分枝因子和深度

人们通常简单地根据博弈树的形状来对博弈树算法进行分析。我们假设每个中间结点有同样多的子结点，其数量称为分枝因子(Branching Factor)。我们还假设搜索到固定的深度(就如前面所提到的算法一样)，并且棋局不会很早地结束(在达到搜索深度以前结束)。

在这些假设下，很容易写下负值最大程序所花的时间，即正比于展开结点的数量。(看上去需要乘上一个系数，以反映调用负值最大时的那个循环，但是这个循环所花的时间已经被包括在递归函数里了。)【译者也不理解这句话的意思，但译者认为程序中eval()函数所花费的时间最多，而它只是在搜索到叶子结点时才被调用，因此只计算叶子结点的数量就可以了，即b^d。】如果分枝因子是b，深度是d，那么这个数就是：

$$1 + b + b^2 + b^3 + ... + b^d = b^d\ (1 - 1\ /\ b^d)\ /\ (1 - 1\ /\ b).$$

公式右端括号里的数值接近于1，所以整个运算所花费的时间接近于b^d。

如果棋类游戏不符合以上假定，我们可以反过来定义一个“有效分枝因子”(Effective Branching Factor)，使得这个b能够符合程序运行所花费的时间。更简单些，可以把“分枝因子”描述为某个棋类游戏中“典型”局面的可能着法数的平均值。

这个公式可以告诉我们什么呢？首先它是指数形式的，这就意味着我们不可能搜索太多层，如果电脑的速度翻了番，那么我们只能把d增加很小一点。其次搜索取决于分枝因子b，在分枝因子很小的棋类中(像西洋跳棋，通常每个局面只有3个着法)，我们就可以搜索的比国际象棋(一个局面有30种左右的着法)或围棋(一个局面有几百种着法)深得多，因此我们喜欢让b越小越好。很不幸的是搜索函数更多地决于棋类游戏本身，而不是我们写程序的水平。但是下一次我们要讨论一个算法，称为Alpha-Beta裁剪，它可以很大程度地减少分枝因子，如果运气好的话，它可以减少到没有裁剪的博弈树的平方根那么多，这就意味着我们可以搜索原来深度(即不用Alpha-Beta搜索的深度)的两倍那么深。【b的平方根即$b^{1/2}$，用一下中学数学学过的公式，$(b^{1/2})^d = b^{d/2}$，还记得吗？】

迭代加深

负值极大的代码还留给我们一个问题：我们如何来给定搜索深度？简单的棋类程序只把它设成一个固定值，这就可能使得程序走的每步棋时花的时间长短变化非常大。因此你最好根据搜索所需的时间，来决定搜索的深度。幸运的是指数特征的搜索有这样一个好处：通过“迭代加深”(Iterated Deepening)这个手段，可以很容易地对搜索进行控制，刚开始搜索时浅一些，然后增加深度重复搜索直到时间用完为止：

```
depth = 0
while (有足够的时间来进行下一层的搜索) {
  depth ++;
  m = rootsearch(depth);
}
执行着法 m;
```

这看上去似乎在浪费时间，因为除了最后一次搜索外，前面的搜索都白费了。但是根据前面分析过的结果，白费的时间是很少的：不同层数所花的时间加起来是 $1 + b + b^2 + \ldots$，我们已经知道它接近于最后一项b^d了。所以，迭代加深所花的代价并不多，而它给我们提供了很好的时间控制的手段。它还有一个很大的作用：在做较深的搜索时，可以用浅一层搜索得到的着法顺序，在Alpha-Beta搜索中，着法顺序是影响搜索的速度的决定性因素。

【Iterative Deepening，字面意思是“重复加深”，就如上文所讲的。但它最主要的作用是改善着法的顺序，它是Alpha-Beta搜索的一种主要的启发方式，浅一层最好的着法在深一层的搜索中首先被尝试，本质上是一种迭代的过程，所以译为“迭代加深”。】

原文：http://www.ics.uci.edu/~eppstein/180a/970417.html
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 数据结构——Zobrist键值
- 下一篇 基本搜索方法——简介(二)
- 返　回 象棋百科全书——电脑象棋

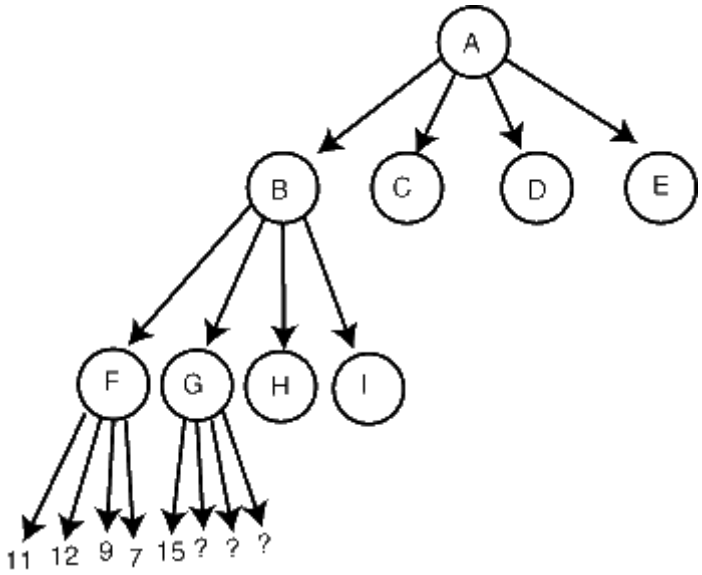
《对弈程序基本技术》专题

Alpha-Beta搜索

David Eppstein */文
* 加州爱尔文大学(UC Irvine)信息与计算机科学系

浅的裁剪

假设你用最小-最大搜索(前面讲到的)来搜索下面的树：



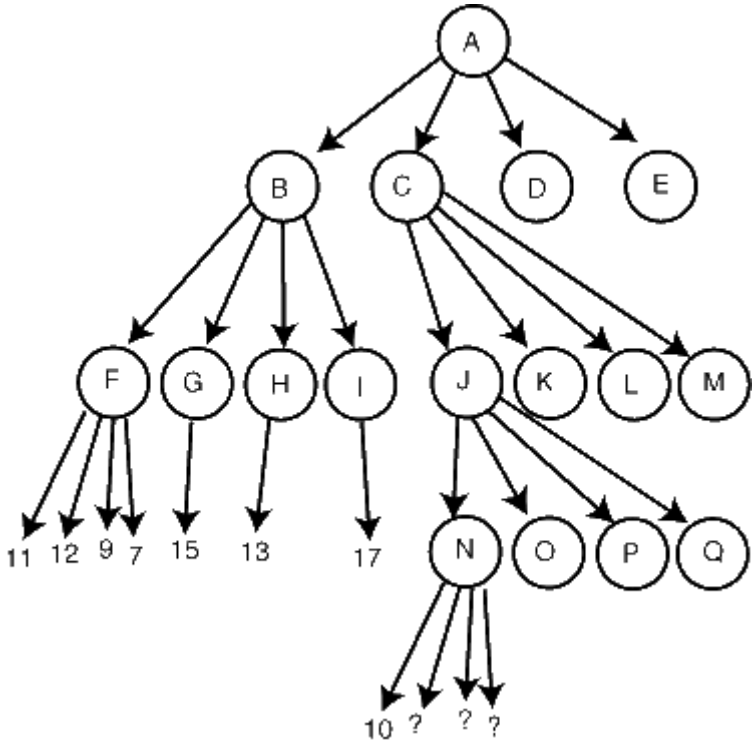
你搜索到F，发现子结点的评价分别是11、12、7和9，在这层是棋手甲走，我们希望他选择最好的值，即12。所以，F的最小-最大值是12。

现在你开始搜索G，并且第一个子结点就返回15。一旦如此，你就知道G的值至少是15，可能更高(如果另一个子结点比G更好)。这就意味着我们不指望棋手乙走G这步了，因为就棋手乙看来，F的评价12要比G的15(或更高)好，因此我们知道G不在主要变例上。我们可以裁剪(Prune)结点G下面的其他子结点，而不要对它们作出评价，并且立即从G返回，因为对G作更好的评价只是浪费时间。

一般来说，像G一样只要有一个子结点返回比G的兄弟结点更好的值(对于结点G要走棋的一方而言)，就可以进行裁剪。

深的裁剪

我们来讨论更复杂的可能裁剪的情况。例如在同一棵搜索树中，我们评价的G、H和I都比12好，因此12就是结点B的评价。现在我们来搜索结点C，在下面两层我们找到了评价为10的结点N：



我们能用更为复杂的路线来作裁剪。我们知道N会返回10或更小(轮到棋手乙走棋，需要挑最小的)。我们不知道J能否返回10或更小，也不知道J的哪个子结点会更好。如果从J返回到C的是10或者更小的值，那么我们可以在结点C上作裁剪，因为它有更好的兄弟结点B。因此在这种情况下，继续找N的子结点就毫无意义。考虑其他情况，J的其他子结点返回比10更好的值，此时搜索N也是毫无意义的。所以我们只要看到10，就可以放心地从N返回。

Alpha-Beta的伪代码

一般来说，如果返回值比偶数层的兄弟结点好，我们就可以立即返回。如果我们在搜索过程中，把这些兄弟结点的最小值Beta作为参数来传递，我们就可以进行非常有效的裁剪。我们还用另一个参数Alpha来保存奇数层的结点。用这两个参数来进行裁剪是非常有效的，代码就写在下边。像上次一样，我们用负值最大(Negamax)的形式，即搜索树的层数改变时取负值。

```
double alphabeta(int depth, double alpha, double beta) {
  if (depth <= 0 || 棋局结束) {
    return evaluation();
  }
  就当前局面，生成并排序一系列着法;
  for (每个着法 m) {
    执行着法 m;
    double val = -alphabeta(depth - 1, -beta, -alpha);
    撤消着法 m;
    if (val >= beta) {
      return val;
    }
    if (val > alpha) {
      alpha = val;
    }
  }
  return alpha;
```

}

下次我们会解释为什么排序这一步是很重要的。

期望搜索

在根结点上我们如何为Alpha和Beta设定初值?

Alpha和Beta定义了一个评价的实数区间(Alpha, Beta)，这个区间是我们“感兴趣的”。如果某值比Beta大我们就会做裁剪并立即返回，因为我们知道它不是主要变例的一部分，我们对它的准确值不感兴趣，只需要知道它比Beta大。如果某值比Alpha小，我们不作裁剪，但是仍然对它不感兴趣，因为我们知道搜索树里肯定有一个着法会更好。

但是在搜索树的根结点，我们不知道感兴趣的评价是在哪个范围内，如果我们要保证不会因为意外而裁剪掉重要的部分，我们就设Alpha = –Infinity，Beta = Infinity(无穷大)。

但是，如果我们使用迭代加深，就可能有办法知道主要变例是怎么样的。假设我们猜其值为x(例如x就是前一次搜索到$D-1$深度时的值)，并设Epsilon为一个很小的值，它代表从$D-1$深度到D深度搜索评价的期望变化范围。我们可以尝试调用alphabeta(D, x – Epsilon, x + Epsilon)，那么可能发生三种情况:

(1) 搜索的返回值会落在区间(x – Epsilon, x + Epsilon)内。这种情况下，我们知道它返回的是正确值，我们就能放心地选择这个着法，在搜索树中这个着法指向具有返回值的那个结点。

(2) 搜索会返回一个值$v \geq x$ + Epsilon。这种情况下，我们知道搜索结果也至少是 x + Epsilon，但是我们不知道它到底是几(正确的主要变例可能被裁剪掉了，因为我们看到有别的着法的值大于Beta)。我们必须把我们所猜的值x调整得更高，然后再试一次(可能还要用更大的Epsilon)。这种情况称为“高出边界”(Fail High)。

(3) 搜索会返回一个值$v \leq x$ – Epsilon。这种情况下，我们知道搜索结果也最多是 x + Epsilon，但是我们不知道它到底是几。我们必须把我们所猜的值x调整得更低，然后再试一次(可能还要用更大的Epsilon)。这种情况称为“低出边界”(Fail Low)。

即便有两种可能失败的情况，使用期望搜索(用一个比(–Infinity, Infinity)更小的区间(Alpha, Beta))总体来说效率会有所提高，因为它作了更多的裁剪。

分析

让我们对Alpha-Beta搜索作一下分析，来知道它为什么是个很有用的算法。跟普通的算法不同，我们采用“Beta情况的分析”，即假设任何可能的情况下都会发生Alpha-Beta裁剪。下一次我们会知道如何让Alpha-Beta搜索接近我们的所分析的情况。在这里我只考虑浅的裁剪，因为它会让分析变得更加简单。

在最好的情况下，除了主要变例上的结点不会裁剪外(如果这个结点也被裁剪了，那么整个算法会高出边界或低出边界，这当然不是最好的情况)，在裁剪前，深$D-1$层的每个结点只会搜索一个深D层的子结点。

但是在深$D-2$层时，谁也没有被裁剪，因为所有的子结点都返回大于或等于Beta的值，而$D-2$层是要取负数，因此它们都小于或等于Alpha。

继续朝树根走，$D-3$层的每个结点(除了主要变例外)都被裁剪，而$D-4$层谁也没被裁剪，等等。

因此，如果搜索树的分枝因子是B，那么在搜索树一半的深度上，结点以因子B作增长，而在另一半的深度上则保持不变(我们忽略了主要变例)。所以这个搜索树所有要搜索的结点数，粗略地写成$B^{D/2} = \text{sqrt}(B)^D$。因此Alpha-Beta搜索最终可以将分枝因子减少为原来的平方根那么多，因此它可以让我们搜索原来两倍的深度。正因为这个原因，它是所有基于最小-最大策略的棋类对弈程序的最重要的算法。

【译注：原作者一开始提到的“浅的裁剪”和“深的裁剪”这两个概念，实际上包含了Alpha-Beta搜索的两个层次，前者只是用过传递参数Beta对搜索树作了部分裁剪，可以称为Beta搜索，而后

者增加一个传递参数Alpha，使得裁剪更加充分，这就形成了Alpha-Beta搜索。
Beta搜索的伪代码是：

```
double alphabeta(int depth, double beta) {
  if (depth <= 0 || 棋局结束) {
    return evaluation();
  }
  就当前局面，生成并排序一系列着法;
  double alpha = -infty;
  for (每个着法 m) {
    执行着法 m;
    double val = -alphabeta(depth - 1, -alpha);
    撤消着法 m;
    if (val >= beta) {
      return val;
    }
    if (val > alpha) {
      alpha = val;
    }
  }
  return alpha;
}
```

对红色部分加一些改进，就变成Alpha-Beta搜索的伪代码了。】

原文：http://www.ics.uci.edu/~eppstein/180a/970422.html
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 基本搜索方法——简介(一)
- 下一篇 基本搜索方法——简介(三)
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

散列技术和着法排序

David Eppstein */文
* 加州爱尔文大学(UC Irvine)信息与计算机科学系

我还没有讲完Alpha-Beta呢，因为我的伪代码里包含神秘的“排序着法”还没有解释，暂时先扔在一边，在讲完散列技术后，我将继续这部分内容。

散列技术的思想非常简单，很多棋类会出现“置换”的着法，即着法顺序的不同会导致相同的局面。例如在国际象棋中，开局走“1. d4 Nf6 2. c4”和“1. c4 Nf6 2. d4”都会导致一样的局面(称为印度防御)，白方两次进兵可以按不同的顺序走。再看一个更加复杂置换：“1. e4 c6 2. d4 d5 3. ed Qxd5 4. Nc3 Qd6”(卡罗-卡恩防御)，“1. e4 d5 2. ed Qxd5 3. Nc3 Qd6 4. d4 c6”(斯堪的那维亚防御)，以及“1. e4 Nf6 2. e5 Ng8 3. d4 d6 4. ed Qxd6 5.Nc3 c6”(阿廖欣防御)都会在走不同的步数后到达相同的局面。

因为换位，相同的局面能在Alpha-Beta搜索树的很多位置上找到。如果有一种数据结构能够保存以前找过的每一个位置，那么我们不必重新搜索，取而代之的是查找局面。但是有两个困难摆在面前，一是我们没有足够的存储器来存放所有搜索过的局面，二是查找速度要足够快以至于超过搜索的时间。幸运的是，找不到已经搜索过的局面也没关系，再搜索一遍就是了，因为毕竟这种情况不会经常发生。

这种数据结构就是“散列表”(Hash Tables)，一个很大的数组，大到不超出物理存储器为止(不要扩展到虚拟存储器，否则会非常慢的)。

```
struct {
  long checksum; // long long 可能会更好
  int depth;
  enum { exact, lower_bound, upper_bound } entry_type;
  double eval;
} hashtable[HASH_TABLE_SIZE]; //【译注：完整的散列表还应记录最佳着法。】
```

对于每一个需要搜索的局面，先计算散列值x作为散列表的指标，另计算散列值y来检查这个局面是否是所要找的局面。

在搜索一个局面前，先去找 hashtable[x]，如果 hashtable[x].checksum == y，hashtable[x].entry_type == exact，并且 hashtable[x].depth 不比现在需要搜索的深度浅，那么就可以返回 hashtable[x].eval。

每搜索完一个局面，就把散列值y、当前搜索的深度和搜索的评分保存到 hashtable[x] 里。

如何计算散列值?

Zobrist 散列技术(已经在“重复检测”这个话题中探讨过了)是指：在下棋以前(但在程序里)产生随机数组Z[square, piecetype]。棋盘的散列值就是棋盘上各个棋子的Z[s, p]相加，再加上一些额外的信息如是否能王车易位等。通常求和被“异或”运算所代替，因为它操作方便且快速，但算术加法会更好些。走完一步棋后，不要把散列值整个都算一遍，只要减去原来位置的棋子和格子值，并加上新位置的棋子和格子值，就实现了散列值的更新。这个技术同时适用于散列值x和校验值y(但要使用不同的随机数表)。

在散列技术中，另有一些十分有用的诀窍：

(1) 在走完一步后，不要把散列表清空，这样不但浪费时间，而且原来的内容或许对后面的走法有用。【这里指的是电脑完成整个局面的搜索，走完一步后不需要清空散列表，电脑进行下一回合

时，上一回合留下的信息或许有用。当然，是否清空散列表不能一概而论，还要取决于散列表的覆盖策略，以后的文章将谈到这个问题。】

(2) 如果相同的局面出现在搜索树的不同深度中(例如上面讲到置换时的第二个例子)，那么你可能得到比预期更深的搜索结果，这样会非常好。【译者把这个现象称为“搜索树的迁移式延伸”，当这种情况出现在残局时反而不是好事，因为你有可能找到一步杀棋，而它却不是最短路线，你就停止了搜索。后面的文章会说明一个问题，当你搜索到杀棋时，如果你的着法不是最短路线，那么在实战中可能明知杀棋但怎么也杀不了。】

(3) 不要在搜索树靠近叶子的局面上用散列技术，因为这些局面太多了(它们可能取代别的更重要的局面)，并且不去搜索这些局面也不会省掉很多时间。【如果把搜索分为完全搜索和静态搜索两个阶段，那么静态搜索的结果是肯定不写入散列表的，因为静态搜索的分枝因子非常小(只考虑吃子着法)，所以查找散列表的时间或许比搜索的时间还长。在完全搜索中，这条法则是就“始终覆盖”这一策略而言的，若使用“深度优先”策略则无所谓。】

散列技术如何跟Alpha-Beta联系起来?

国际象棋程序中很多的错误都跟散列技术有关，原因是它们要跟Alpha-Beta搜索联系起来，但你无法绕开这个话题，因为散列技术和Alpha-Beta都是非常有效的搜索方法。现在重新来看Alpha-Beta，当你在一个局面中调用 alphabeta(depth, alpha, beta) 时，可能有三种情况发生：(1) 高出边界(Fail High)，即返回评分至少是Beta，但到底多少却不知道；(2) 低出边界(Fail Low)，即返回评分最多是Alpha，但到底多少也不知道；(3) 准确值，即返回评分在Alpha和Beta之间。如果我们知道准确结果，那么散列表中只记录准确评分，但是高出边界和低出边界的情况，仍然在以后的裁剪中有用。可以跟记录准确评分一样，散列表中也可以记录这两种情况的评分，“下界标志”(lower_bound)代表评分至少是Beta，“上界标志”(up_bound)代表评分最多是Alpha，我们用 entry_type 这个域来表示评分属于哪个类型。如果搜索散列表时返回这两个类型，那么我们需要看它是否在搜索结点之前发生裁剪，如果能发生裁剪，那么直接返回该评分，否则继续搜索。下面是用散列技术的Alpha-Beta搜索的伪代码，散列表的索引值x和校验值y是全局变量，并且在执行着法和撤消着法的时候更新。

```
double alphabeta(int depth, double alpha, double beta) {
  if (depth <= 0 || 棋局结束) {
    return evaluation();
  }
  if (hashtable[x].checksum == y && hashtable[x].depth >= depth) {
    switch (hashtable[x].entry_type) {
    case exact: //【就C语言的语法而言是错的，应该写成“case hashtable[x].exact:”，下同】
      return hashtable[x].eval;
    case lower_bound:
      if (hashtable[x].eval >= beta) {
        return (hashtable[x].eval);
      }
      break;
    case upper_bound:
      if (hashtable[x].eval <= alpha) {
        return (hashtable[x].eval);
      }
      break;
    }
  }
```

```
    int eval_is_exact = 0;
    就当前局面，生成并排序一系列着法;
    for (每个着法 m) {
      执行着法 m;
      double val = -alphabeta(depth - 1, -beta, -alpha);
      撤消着法 m;
      if (val >= beta) {
        hashtable[x].checksum = y;
        hashtable[x].depth = depth;
        hashtable[x].entry_type = lower_bound;
        hashtable[x].eval = val; //【译者认为 eval = beta 更加合理。】
        return val;
      }
      if (val > alpha) {
        alpha = val;
        eval_is_exact = 1;
      }
    }
    hashtable[x].checksum = y;
    hashtable[x].depth = depth;
    if (eval_is_exact) {
      hashtable[x].entry_type = exact;
    } else {
      hashtable[x].entry_type = upper_bound;
    }
    hashtable[x].eval = alpha;
    return alpha;
  }
```

Alpha-beta和着法顺序

我们现在就回到Alpha-Beta。上次我们的分析指出，在最乐观的情况下，如果能裁剪的地方都裁剪了，那么搜索深度将增加一倍。“能裁剪的地方都裁剪了”这个条件说得再简单一些，就是好的着法在坏的着法之前搜索。着法没有必要完全排序好，但是最好的着法必须首先搜索，或者最好的几个着法必须在比较前面的几个位置搜索。如果不这样做，就不会有裁剪并且不会搜索得很深。

我们把结点分为A型(所有的子结点都搜索过)和B型(在搜索到好的子结点后即裁剪掉了)，那么着法的顺序对两种结点都很重要：在B型结点中，需要从最好的子结点开始，以裁剪掉剩余的节点；而在A型结点中，需要找到足够好的结点，来告诉其他结点，它们都是B型的。

找到最好的着法当然是很难的，这就是我们需要进行搜索的原因。【言下之意，要在搜索之前找到最好的着法，理论上是不可能的。】但是我们可以根据以下线索来找：(1) 迭代加深时，散列表会记录前面一次搜索过的结点，散列表中的值可以用来作近似(因为这是相同局面浅一些的搜索)；【原作者在散列表中没有记录最佳着法，其实这个着法最应该被首先搜索。】(2) 针对特定的棋类游戏，我们需要一些知识，例如在国际象棋中，吃子的着法往往是好的着法，应该首先尝试；(3) 杀手启发：如果相邻结点有好的着法，并且在现在的结点上该着法也合理，那么应该首先尝试。

因此在搜索子结点之前要加上一步：按照着法的质量来排序，然后按照这个顺序来搜索。(有时你们可以直接修改着法生成器，让它进行粗略的排序，比如先生成吃子的着法，从而节省下重新排序的时间。)

还有一个诀窍：如果你希望发生裁剪，那么你不必对所有着法排序，只要排序前面很少几个着法就可以了。那么你应该用这样的搜索算法，最佳着法可以一个一个地挑出，然后早早地停止排

序，例如选择排序和堆排序。【译者认为冒泡排序是最符合原作者的意图的，因为排序时每扫描一趟即可找到一个最佳着法，在扫描第二趟之前先搜索这个着法。而几乎没有程序是用堆排序的，尽管它的复杂度是所有排序算法中最低的，并且不消耗额外的存储器，但是对100个以下的数进行排序，它的速度是非常缓慢的。】

原文：http://www.ics.uci.edu/~eppstein/180a/970424.html
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 基本搜索方法——简介(二)
- 下一篇 基本搜索方法——最小-最大搜索
- 返　回 象棋百科全书——电脑象棋

www.xqbase.com

《对弈程序基本技术》专题

最小-最大搜索

Bruce Moreland / 文

从浅显的地方开始

在国际象棋里，双方棋手都知道每个棋子在哪里，他们轮流走并且可以走任何合理的着法。下棋的目的就是将死对方，或者避免被将死，或者有时争取和棋是最好的选择。

国际象棋程序通过使用“搜索”函数来寻找着法。搜索函数获得棋局信息，然后寻找对于程序一方来说最好的着法。

一个浅显的搜索函数用“树状搜索”(Tree-Searching)来实现。一个国际象棋棋局通常可以看作一个很大的*n*叉树(“*n*叉树”意思是树的每个结点有任意多个分枝通向其他结点)，棋盘上目前的局面就是“根局面”(Root Position)或“根结点”(Root Node)。从根局面走一步棋，局面就到达根局面的“分枝”(Branch)，这些局面称为“后续局面”(Successor Position)或“后续结点”(Successor Nodes)。每个后续局面后面还有一系列分枝，每个分枝就是这个局面的一个合理的着法。

国际象棋的树非常庞大(通常每个局面有35个分枝)，又非常深。

每盘棋局都是一棵巨大的*n*叉树，如果能通过树状搜索找到棋局中对双方来说都最好的着法就好了。这个浅显的算法在这里称为“最小-最大搜索”(Min-max Search)。

用最小-最大搜索来解诸如井字棋的简单棋局是可行的(即完全了解每一种变化)。井字棋的博弈树既不烦琐也不深，所以整个树可以遍历，棋局的所有变化都可以知道，任何局面都可以保证找到一步最佳着法。

数学上用这种方法处理国际象棋也是可以的，但是目前和不久的将来用计算机去实现，却是不可行的。即便如此，我们仍然可以用基于最小-最大搜索的程序来下国际象棋。相比最小-最大地搜索整个树，在一个给定的局面下搜索前几步则是可能的。由于叶子结点的局面没能搜索出杀棋或和棋，所以要用一个称为“评价”(Evaluate)的启发函数给这些局面赋值。尽管程序设计师希望这些值能够通过知识来得到，但它们确实都是猜的。

基于最小-最大的评价函数

我不打算在这里谈很多关于评价函数的细节。这里我只说明它是怎样确定的，在以后的章节中会详细展开。评价函数首先应该返回局面的准确值，在没办法得到准确值的情况下，如果可能的话启发值也可以。它可以由两种方法来决定：

(1) 如果黑方被将死了，那么评价函数返回一个充分大的正数；如果白方被将死了，那么返回一个充分大的负数；如果棋局是和棋(例如某一方逼和，或者双方都只有王)，那么返回一个常数，通常是零或接近零。如果不是棋局结束局面，那么它返回一个启发值。我将不详细介绍这个启发值是如何确定的，但是我有把握说子力平衡是首先要考虑的(如果白方盘面上多子的话，这个值就大)，而其他位置上的考虑(兵型、王的安全性、重要的子力等等)也需要加上。如果白方是赢棋或者很有希望赢，那么启发函数通常会返回正数；如果黑方是赢棋或者很有希望赢，那么返回负数；如果棋局是均势或者是和棋，那么返回在零左右的数值。

(2) 这个函数的工作原理跟第一个一样，只是如果当前局面要走子的一方优势，那么它返回正数，反之是负数。

最小-最大搜索是如何运作的

最小-最大搜索是一对几乎一样的函数，或者说两个逻辑上重复的函数。我写了很少的代码，用一个更好的函数来完成同一件事，但是写出来时却收到一些意见，因此我首先写出纯粹的(不完美的)最小-最大函数，代码如下：

```
int MinMax(int depth) {
  if (SideToMove() == WHITE) {   // 白方是“最大”者
    return Max(depth);
  } else {                       // 黑方是“最小”者
    return Min(depth);
  }
}

int Max(int depth) {
  int best = -INFINITY;
  if (depth <= 0) {
    return Evaluate();
  }
  GenerateLegalMoves();
  while (MovesLeft()) {
    MakeNextMove();
    val = Min(depth - 1);
    UnmakeMove();
    if (val > best) {
      best = val;
    }
  }
  return best;
}

int Min(int depth) {
  int best = INFINITY;   // 注意这里不同于“最大”算法
  if (depth <= 0) {
    return Evaluate();
  }
  GenerateLegalMoves();
  while (MovesLeft()) {
    MakeNextMove();
    val = Max(depth - 1);
    UnmakeMove();
    if (val < best) {    // 注意这里不同于“最大”算法
      best = val;
    }
  }
  return best;
}
```

上面的代码可以这样调用：

```
val = MinMax(5);
```

这样可以返回当前局面的评价，它是向前看5步的结果。

这里的“评价”函数用的是我上面所说第一种定义，它总是返回对于白方来说的局面。

我简要描述一下这个函数是如何运作的。假设根局面(棋盘上当前局面)是白方走，那么调用的是“Max”函数，它产生白方所有合理着法。在每个后续局面中，调用的是“Min”函数，它对局面作出评价并返回。由于现在是白走，因此白方需要让评价尽可能地大，能得到最大值的那个着法被认为是最好的，因此返回这个着法的评价。

“Min”函数正好相反，当黑方走时调用“Min”函数，而黑方需要尽可能地小，因此选择能得到最小值的那个着法。

这两个函数是互相递归的，即它们互相调用，直到达到所需要的深度为止。当函数到达最底层时，它们就返回“Evaluate”函数的值。

如果在深度为1时调用“MinMax”函数，那么“Evaluate”函数在走完每个合理着法之后就调用，选择一个能达到最佳值的那个着法导致的局面。如果层数大于1，那么另一方有权选择局面，并找一个最好的。

以上内容应该不难理解，但是代码很长，下面有个更好的办法。

负值最大函数

负值最大只是对最小-最大的优化，“评价”函数返回我所说的第二种定义，对于当前结点上要走的一方，占优的情况返回正值，其他结点也是对于要走的一方而言的。这个值返回后要加上负号，因为返回以后就是对另一方而言了。代码如下：

```
int NegaMax(int depth) {
  int best = -INFINITY;
  if (depth <= 0) {
    return Evaluate();
  }
  GenerateLegalMoves();
  while (MovesLeft()) {
    MakeNextMove();
    val = -NegaMax(depth - 1); // 注意这里有个负号。
    UnmakeMove();
    if (val > best) {
      best = val;
    }
  }
  return best;
}
```

在这个函数里，当走子一方改变时就要对返回值取负值，以反映当前局面评价的更改。就根结点是白先走的情况，如果没有剩下的层数，那么“评价”返回的值是就白方而言的，如果有剩下的层数，就产生后续局面，函数对这些局面逐一做递归，每个次递归都得到就黑方而言的评价，黑方走得越好值就越大。当评价值返回时，它们被取负数，变成就白方而言的评价。

该函数在遍历时结点的顺序同“最小-最大”搜索的函数是一样的，产生的返回值也一样。它的代码更短，同时减少了移植代码时出错的可能，代码维护起来也比较方便。

原文：http://www.seanet.com/~brucemo/topics/minmax.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译

- 上一篇 基本搜索方法——简介(三)
- 下一篇 基本搜索方法——Alpha-Beta搜索
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

Alpha-Beta搜索

Bruce Moreland / 文

最小-最大的问题

Alpha-Beta 同“最小-最大”非常相似，事实上只多了一条额外的语句。最小最大运行时要检查整个博弈树，然后尽可能选择最好的线路。这是非常好理解的，但效率非常低。每次搜索更深一层时，树的大小就呈指数式增长。

通常一个国际象棋局面都有35个左右的合理着法，所以用最小-最大搜索来搜索一层深度，就有35个局面要检查，如果用这个函数来搜索两层，就有35^2个局面要搜索。这就已经上千了，看上去还不怎样，但是数字增长得非常迅速，例如六层的搜索就接近是二十亿，而十层的搜索就超过两千万亿了。

要想通过检查搜索树的前面几层，并且在叶子结点上用启发式的评价，那么做尽可能深的搜索是很重要的。最小-最大搜索无法做到很深的搜索，因为有效的分枝因子实在太大了。

口袋的例子

幸运的是我们有办法来减小分枝因子，这个办法非常可靠，实际上这样做绝对没有坏处，纯粹是个有益的办法。这个方法是建立在一个思想上的，如果你已经有一个不太坏的选择了，那么当你要作别的选择并知道它不会更好时，你没有必要确切地知道它有多坏。有了最好的选择，任何不比它更好的选择就是足够坏的，因此你可以撇开它而不需要完全了解它。只要你能证明它不比最好的选择更好，你就可以完全抛弃它。

你可能仍旧不明白，那么我就举个例子。比如你的死敌面前有很多口袋，他和你打赌赌输了，因此他必须从中给你一样东西，而挑选规则却非常奇怪：

每个口袋里有几件物品，你能取其中的一件，你来挑这件物品所在的口袋，而他来挑这个口袋里的物品。你要赶紧挑出口袋并离开，因为你不愿意一直做在那里翻口袋而让你的死敌盯着你。

假设你一次只能找一只口袋，在找口袋时一次只能从里面摸出一样东西。

很显然，当你挑出口袋时，你的死敌会把口袋里最糟糕的物品给你，因此你的目标是挑出“诸多最糟的物品当中是最好的”那个口袋。

你很容易把最小-最大原理运用到这个问题上。你是最大一方棋手，你将挑出最好的口袋。而你的死敌是最小一方棋手，他将挑出最好的口袋里尽可能差的物品。运用最小-最大原理，你需要做的就是挑一个有“最好的最差的”物品的口袋。

假设你可以估计口袋里每个物品的准确价值的话，最小-最大原理可以让你作出正确的选择。我们讨论的话题中，准确评价并不重要，因为它同最小-最大或Alpha-Beta的工作原理没有关系。现在我们假设你可以正确地评价物品。

最小-最大原理刚才讨论过，它的问题是效率太低。你必须看每个口袋里的每件物品，这就需要花很多时间。

那么怎样才能做得比最小-最大更高效呢?

我们从第一个口袋开始，看每一件物品，并对口袋作出评价。比方说口袋里有一只花生黄油三明治和一辆新汽车的钥匙。你知道三明治更糟，因此如果你挑了这只口袋就会得到三明治。事实上只要我们假设对手也会跟我们一样正确评价物品，那么口袋里的汽车钥匙就是无关紧要的了。

现在你开始翻第二个口袋，这次你采取的方案就和最小-最大方案不同了。你每次看一件物品，并跟你能得到的最好的那件物品(三明治)去比较。只要物品比三明治更好，那么你就按照最小-最大

方案来办——去找最糟的，或许最糟的要比三明治更好，那么你就可以挑这个口袋，它比装有三明治的那个口袋好。

比方这个口袋里的第一件物品是一张20美元的钞票，它比三明治好。如果包里其他东西都没比这个更糟了，那么如果你选了这个口袋，它就是对手必须给你的物品，这个口袋就成了你的选择。

这个口袋里的下一件物品是六合装的流行唱片。你认为它比三明治好，但比20美元差，那么这个口袋仍旧可以选择。再下一件物品是一条烂鱼，这回比三明治差了。于是你就说“不谢了”，把口袋放回去，不再考虑它了。

无论口袋里还有什么东西，或许还有另一辆汽车的钥匙，也没有用了，因为你会得到那条烂鱼。或许还有比烂鱼更糟的东西(那么你看着办吧)。无论如何烂鱼已经够糟的了，而你知道挑那个有三明治的口袋肯定会更好。

算法

Alpha-Beta就是这么工作的，并且只能用递归来实现。稍后我们再来谈最小一方的策略，我希望这样可以更明白些。

这个思想是在搜索中传递两个值，第一个值是Alpha，即搜索到的最好值，任何比它更小的值就没用了，因为策略就是知道Alpha的值，任何小于或等于Alpha的值都不会有所提高。

第二个值是Beta，即对于对手来说最坏的值。这是对手所能承受的最坏的结果，因为我们知道在对手看来，他总是会找到一个对策不比Beta更坏的。如果搜索过程中返回Beta或比Beta更好的值，那就够好的了，走棋的一方就没有机会使用这种策略了。

在搜索着法时，每个搜索过的着法都返回跟Alpha和Beta有关的值，它们之间的关系非常重要，或许意味着搜索可以停止并返回。

如果某个着法的结果小于或等于Alpha，那么它就是很差的着法，因此可以抛弃。因为我前面说过，在这个策略中，局面对走棋的一方来说是以Alpha为评价的。

如果某个着法的结果大于或等于Beta，那么整个结点就作废了，因为对手不希望走到这个局面，而它有别的着法可以避免到达这个局面。因此如果我们找到的评价大于或等于Beta，就证明了这个结点是不会发生的，因此剩下的合理着法没有必要再搜索。

如果某个着法的结果大于Alpha但小于Beta，那么这个着法就是走棋一方可以考虑走的，除非以后有所变化。因此Alpha会不断增加以反映新的情况。有时候可能一个合理着法也不超过Alpha，这在实战中是经常发生的，此时这种局面是不予考虑的，因此为了避免这样的局面，我们必须在博弈树的上一个层局面选择另外一个着法。

在第二个口袋里找到烂鱼就相当于超过了Beta，如果口袋里没有烂鱼，那么考虑六盒装流行唱片的口袋会比三明治的口袋好，这就相当于超过了Alpha(在上一层)。算法如下，醒目的部分是在最小-最大算法上改过的：

```
int AlphaBeta(int depth, int alpha, int beta) {
   if (depth == 0) {
      return Evaluate();
   }
   GenerateLegalMoves();
   while (MovesLeft()) {
      MakeNextMove();
      val = -AlphaBeta(depth - 1, -beta, -alpha);
      UnmakeMove();
      if (val >= beta) {
         return beta;
      }
      if (val > alpha) {
         alpha = val;
```

```
        }
      }
      return alpha;
    }
```

把醒目的部分去掉，剩下的就是最小-最大函数。可以看出现在的算法没有太多的改变。
这个函数需要传递的参数有：需要搜索的深度，负无穷大即Alpha，以及正无穷大即Beta：

```
val = AlphaBeta(5, -INFINITY, INFINITY);
```

这样就完成了5层的搜索。我在写最小-最大函数时，用了一个诀窍来避免用了“Min”还用“Max”函数。在那个算法中，我从递归中返回时简单地对返回值取了负数。这样就使函数值在每一次递归中改变评价的角度，以反映双方棋手的交替着子，并且它们的目标是对立的。

在Alpha-Beta函数中我们做了同样的处理。唯一使算法感到复杂的是，Alpha和Beta是不断互换的。当函数递归时，Alpha和Beta不但取负数而且位置交换了，这就使得情况比口袋的例子复杂，但是可以证明它只是比最小-最大算法更好而已。

最终出现的情况是，在搜索树的很多地方，Beta是很容易超过的，因此很多工作都免去了。

可能的弱点

这个算法严重依赖于着法的寻找顺序。如果你总是先去搜索最坏的着法，那么Beta截断就不会发生，因此该算法就如同最小-最大一样，效率非常低。该算法最终会找遍整个博弈树，就像最小-最大算法一样。

如果程序总是能挑最好的着法来首先搜索，那么数学上有效分枝因子就接近于实际分枝因子的平方根。这是Alpha-Beta算法可能达到的最好的情况。

由于国际象棋的分枝因子在35左右，这就意味着Alpha-Beta算法能使国际象棋搜索树的分枝因子变成6。

这是很大的改进，在搜索结点数一样的情况下，可以使你的搜索深度达到原来的两倍。这就是为什么使用Alpha-Beta搜索时，着法顺序至关重要的原因。

原文：http://www.seanet.com/~brucemo/topics/alphabeta.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译

- 上一篇 基本搜索方法——最小-最大搜索
- 下一篇 基本搜索方法——迭代加深
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

迭代加深

Bruce Moreland / 文

一个听起来不怎样的思想

如果你准备开始搜索一个国际象棋的局面了，你要搜索多深呢？事先预测搜索将进行多少时间，这有些困难，因为完成*D*层搜索所需要的时间取决于很多不确定的因素。在复杂的中局局面里，你可能不会搜索得很深，而在残局中你可能会搜索得非常深，在某些王兵残局里你可能会搜索100多层【译注：这也太夸张了点吧】。

有一个思想，就是一开始只搜索一层，如果搜索的时间比分配的时间少，那么搜索两层，然后再搜索三层，等等，直到你用完时间为止。

这足以保证很好地运用时间了。如果你可以很快搜索到一个深度，那么你在接下来的时间可以搜索得更深，或许你可以完成。如果局面比你想象的复杂，那么你不必搜索得太深，但是至少有合理的着法可以走了，因为你不太可能连1层搜索也完不成。

这个思想称为“迭代加深”(Iterative Deepening)，因为你在迭代搜索，每次都比一次前一次加深1层(多1层没有什么奥妙的，当然你可以试试多两层，但是1层比较好)。

代码如下：

```
for (depth = 1; ; depth ++) {
   val = AlphaBeta(depth, -INFINITY, INFINITY);
   if (TimedOut()) {
      break;
   }
}
```

这是一个非常有效的搜索方法，你可能会感到吃惊。如果你能增强Alpha-Beta使得它返回一条“主要变例”，你可以用主要变例中的着法来做下一次迭代搜索。

例如，一层的搜索显示“1. e4”是最好的着法，那么在做两层的搜索时你先搜索“1. e4”。如果返回“1. e4 e5”，那么你在做三层的搜索时仍旧先搜索这条路线。

这样做之所以有好的效果，是因为第一次搜索的线路通常是好的，而Alpha-Beta对着法的顺序特别敏感。如果着法顺序很坏，那么在国际象棋中你的“分枝因子”将接近35。如果你的着法很好，那么分枝因子将接近于6。前一次迭代的搜索函数得到的主要变例通常是非常好的着法。

迭代加深的思想给了你一个简单的方法，它可以在时间用完时中断搜索，并且会提高你的搜索效率。

【有可能的话，你可以把检测超时的程序做到“AlphaBeta”函数里去，而“TimeOut”只是由“AlphaBeta”函数返回的超时检测结果(如果超时的话，就直接跳出函数体了)。很多情况下，程序没有必要搜索整个一层才给出最佳着法。由于迭代加深的原因，新的一层搜索的第一个着法总是上一层搜索得到的最佳着法，如果新的一层可以搜索出另一个更好的着法，就已经很满意了，有时没有必要找到最好的着法。换句话说，即使你没有足够的时间把这一层搜索完，得到的着法至少不会比上一层最好着法要坏。】

原文：http://www.seanet.com/~brucemo/topics/iterative.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 基本搜索方法——Alpha-Beta搜索
- 下一篇 基本搜索方法——置换表
- 返　回 象棋百科全书——电脑象棋

www.xqbase.com

《对弈程序基本技术》专题

置换表

Bruce Moreland / 文

一个多功能的数据结构

国际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。如果“1. e4 d6 2. d4”以后的局面已经搜索过了，那就没有必要再搜索“1. d4 d6 2. e4”以后的局面了。

这个原因可能鼓舞着早期的电脑国际象棋程序的设计师们，而现在事实上这还是置换表的次要用途。在某些局面，例如在没有通路兵的王兵残局中，检查到的置换的数量是惊人的，以至于搜索可以在短达时间内达到很深的深度。

省去重复的工作，这是置换表的一大特色，但是在一般的中局局面里，置换表的另一个作用更为重要。每个散列项里都有局面中最好的着法，我在“迭代加深”这一章里解释过，首先搜索好的着法可以大幅度提高搜索效率。因此如果你在散列项里找到最好的着法，那么你首先搜索这个着法，这样会改进你的着法顺序，减少分枝因子，从而在短的时间内搜索得更深。

实现

主置换表是一个散列数组，每个散列项看上去像这样：

```
#define hashfEXACT 0
#define hashfALPHA 1
#define hashfBETA 2
typedef struct tagHASHE {
   U64 key;
   int depth;
   int flags;
   int value;
   MOVE best;
} HASHE;
```

这个散列数组是以“Zobrist键值”为指标的。你求得局面的键值，除以散列表的项数得到余数，这个散列项就代表该局面。由于很多局面都有可能跟散列表中同一项作用，因此散列项需要包含一个校验值，它可以用来确认该项就是你要找的。通常校验值是一个64位的数，也就是上面那个例子的第一个域。

你从搜索中得到结果后，要保存到散列表中。如果你打算用散列表来避免重复工作，那么重要的是记住搜索有多深。如果你在一个结点上搜索了3层，后来又打算做10层搜索，你就不能认为散列项里的信息是准确的。因此子树的搜索深度也要记录。

在Alpha-Beta搜索中，你很少能得到搜索结点的准确值。Alpha和Beta的存在有助你裁剪掉没有用的子树，但是用Alpha-Beta有个小的缺点，你通常不会知道一个结点到底有多坏或者有多好，你只是知道它足够坏或足够好，从而不需要浪费更多的时间。

当然，这就引发了一个问题，散列项里到底要保存什么值，并且当你要获取它时怎样来做。答案是储存一个值，另加一个标志来说明这个值是什么含义。在我上面的例子中，比方说你在评价域中保存了16，并且在标志域保存了“hashfEXACT”，这就意味着该结点的评价是准确值16；如果你

在标志域中保存了“hashfALPHA”，那么结点的值最多是16；如果保存了“hashfBETA”，这个值就至少是16。

当你在搜索中遇到特定情况时，很容易决定评价和标志应该保存哪些内容。然而避免错误是非常重要的，散列表是非常容易犯错误的，而且一旦犯下错误就很难捕捉出来。

我的散列项的最后一个域，保存着上次搜索到这个局面时的最佳着法。有时我没有得到最佳着法，比如任何低出边界的情况(返回一个小于或等于Alpha的值)，而其他情况必定有最佳着法，比如高出边界的情况(返回一个大于或等于Beta的值)。【译注：只有叶子结点才没有最佳着法，即便是Alpha结点，所有的着法都是差的，也应该从中找一个最好的着法，它对更深一层的搜索会带来很大的好处。】

如果找到最佳着法，那么它应该首先被搜索。

下面是示范程序，是根据Alpha-Beta函数修改的，改动的地方用醒目的字标出：

```
int AlphaBeta(int depth, int alpha, int beta) {
  int hashf = hashfALPHA;
  if ((val = ProbeHash(depth, alpha, beta)) != valUNKNOWN) {
    // 【valUNKNOWN必须小于-INFINITY或大于INFINITY，否则会跟评价值混淆。】
    return val;
  }
  if (depth == 0) {
    val = Evaluate();
    RecordHash(depth, val, hashfEXACT);
    return val;
  }
  GenerateLegalMoves();
  while (MovesLeft()) {
    MakeNextMove();
    val = -AlphaBeta(depth - 1, -beta, -alpha);
    UnmakeMove();
    if (val >= beta) {
      RecordHash(depth, beta, hashfBETA);
      return beta;
    }
    if (val > alpha) {
      hashf = hashfEXACT;
      alpha = val;
    }
  }
  RecordHash(depth, alpha, hashf);
  return alpha;
}
```

以下就是两个新的函数的代码：

```
int ProbeHash(int depth, int alpha, int beta) {
  HASHE *phashe = &hash_table[ZobristKey() % TableSize()];
  if (phashe->key == ZobristKey()) {
    if (phashe->depth >= depth) {
      if (phashe->flags == hashfEXACT) {
        return phashe->val;
```

```
            }
            if ((phashe->flags == hashfALPHA) && (phashe->val <= alpha)) {
               return alpha;
            }
            if ((phashe->flags == hashfBETA) && (phashe->val >= beta)) {
               return beta;
            }
         }
         RememberBestMove();
      }
      return valUNKNOWN;
   }

   void RecordHash(int depth, int val, int hashf) {
      HASHE *phashe = &hash_table[ZobristKey() % TableSize()];
      phashe->key = ZobristKey();
      phashe->best = BestMove();
      phashe->val = val;
      phashe->hashf = hashf;
      phashe->depth = depth;
   }
```

你所看到的代码，并不像航天科学一样准确，而是很可能有错误的，而且细节上的问题我还没有讨论。如果你的程序中有错误，或许就是很严重的错误。

【以上代码有个速度上的瓶颈，即“ZobristKey() % TableSize()”这个表达式。由于“电脑一做除法就成了傻瓜”，所以“TableSize”最好是一个2^n的常量，只有当除数是2^n时除法才可以由右移指令取代。最好的方法是设一个“TableSizeMask”的变量：

```
int TableSizeMask = TableSize() - 1;
HASHE *phashe = &hash_table[ZobristKey() & TableSizeMask];
```

而这里“TableSize()”也必须是2^n。正是这个道理，在很多可以设定置换表大小的国际象棋程序中，允许的设定值总是呈倍数增长的，要么是3M、6M、12M、24M等等(如果每个散列项有12字节)，要么是4M、8M、16M、32M等等(如果每个散列项有16字节)。】

替换策略

最主要的细节就包括，什么时候该覆盖散列项。在上面的例子中，我用了“始终替换”的策略，即简单地覆盖已经存在的值。这或许不是最好的策略，事实上已经有大量的工作试图找出哪个策略是最好的。

另一个策略是“同样深度或更深时替换”。除非新局面的深度大于或等于散列表中已经有的值，否则已经存在的结点将被保留。

还有很多试验的余地。1994年我在Usenet(新闻组网络系统)的新闻组rec.games.chess(如今是rec.games.chess.computer)上问了这个问题，得到了Ken Thompson的答复。

他的回答是使用两个散列表。一个使用“始终替换”策略，另一个使用“同样深度或更深时替换”。当你做试探时，两个散列表都去试探，如果其中一个可以产生截断，那就可以了。如果两者都不能产生截断，那么你可能至少得到一个最佳着法，实际上更多的可能是得到两个不同的着法，两者都应该首先(或第二个)尝试。

记录的时候，你只要简单地根据替换策略来执行。

如果你使用“同样深度或更深时替换”的策略，那么你的散列表可能最终会被过期的但很深的结点所占满。解决方案就是每次你走棋时都清除散列表，或者在散列项中加入“顺序”这个域，从而使这个策略变成变成“同样深度，或更深，或原来是旧的搜索，才替换”。

我在我的程序Ferret中使用了Thompson的策略，并且运行得很好。另一个程序Gerbil也使用这个策略，你可以去看它的源代码。

【根据译者研究的结果，只用“深度优先覆盖”策略(即“同样深度或更深时替换”)，效果会比“始终替换”好得多，而代码则并不复杂，只有醒目的部分是新增的：

```
void RecordHash(int depth, int val, int hashf) {
  HASHE *phashe = &hash_table[ZobristKey() & (TableSize() - 1)];
  if (phashe->hashf != hashfEMPTY && phashe->depth > depth) {
    return;
  }
  phashe->key = ZobristKey();
  phashe->best = BestMove();
  phashe->val = val;
  phashe->hashf = hashf;
  phashe->depth = depth;
}
```

如果使用这个代码，那么每走一步以前都必须把散列表中所有的标志项置为“hashfEMPTY”。】

不稳定性的问题

当你用置换表时，如果你允许搜索过程根据散列项来截断，那就会产生另一个问题，你的搜索会受“不稳定性”的捆扰。

不稳定性至少是由以下因素引起的：

1. 你可能在做6层的搜索，但是如果你在散列项中得到10层搜索的结果，就可能根据这个值来截断。在后来的搜索中，这个散列项被覆盖了，因此你在这个结点上得到了两个不同的值。

2. Zobrist键值无法记录到达结点的线路，这个结点上不是每条线路都有相同结果的。如果某条线路遇到重复局面，那么散列项的值就会跟路线有关。因为重复局面会导致和局的分值，或者至少不一样的分值。

就我所知，还没有什么办法能处理这些问题。

【另外，如果搜索过程中找到杀棋，那么评价值会接近“INFINITY”或“–INFINITY”，此时记录散列表时不能简单地记录这些评价值，在后面介绍的“胜利局面”的处理中，会谈到这个问题。】

原文：http://www.seanet.com/~brucemo/topics/hashing.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 基本搜索方法——迭代加深
- 下一篇 高级搜索方法——简介(一)
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

什么样的结点需要搜索？全部还是选择性的？

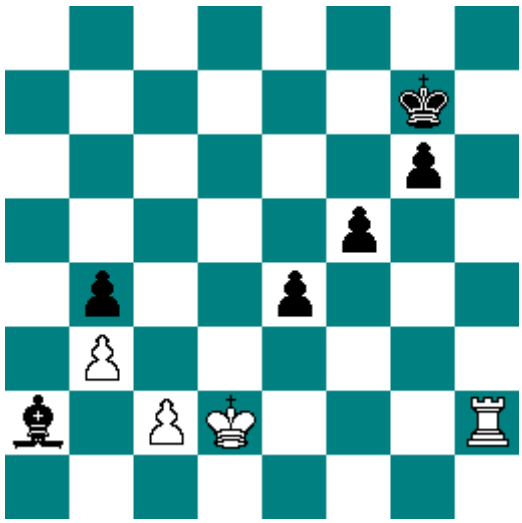
David Eppstein */文
* 加州爱尔文大学(UC Irvine)信息与计算机科学系

Alpha-Beta告诉我们怎样搜索，但是我们仍然需要知道，何时需要展开结点(搜索它的子结点)，以及何时停下来从而调用评价函数。

水平线效应

在前面我给你们看的伪代码中，每一步棋都搜索到一个固定的深度，这个深度被称为“水平线”(Horizon)。对于看到水平线以内会发生的威胁，这个方法非常有效，但是它显然不能检查到水平线以后的威胁。例如在8层的搜索中(即搜索4个回合)，就可能得不到在5步内有杀棋的任何信息。它不知道的事情，就无法作出防御，而且只是简单地忽略那些遥远的威胁。然而当局势面临中等深度的威胁而丢子不可避免时，固定深度的搜索有时会走出更糟的棋，因为某些丢子会在搜索水平线以内，而有些却不在。在这种情况下，程序会走出糟糕的并且无意义的棋，试图来延缓丢子的发生，使得它出现在程序看不到的未来。这种现象称为“水平线效应”(Horizon Effect)。

这里有个例子。在下面的局势中，黑象被白兵包围，不管黑的怎么走，象总是会在几步内被吃掉；例如白车可以沿着h2-h1-a1-a2吃掉象。这是一个8层深的变化，那么假设黑方的程序也搜索8层。可能当前局面对黑方来说，最好的走法就是用象换兵，即象吃掉兵，兵吃掉象。在后面的残局里，黑方的三个连兵足以战胜或守和白车。【译注：其实守和的希望也不大，译者认为黑方还是会输掉的，因为白王的位置非常好，可以独挡三黑兵，使得白车有时间拔掉b线的黑兵。】但是程序搜索8层，很有可能会挺黑兵将军白王。白必须应对(例如自己来吃掉兵)，这个应对使得丢象被暂时避免，拖延到程序看不到的步数内，并且程序认为象是安全的。实际上在这个局面里，固定深度的程序可能会连续地送吃兵，把象被吃的结果延缓几步，但是最终可能输掉整盘棋。



对付水平线效应的一个方法，就是在你的程序里增加一些知识：如果从评价中知道象被包围，那么搜索程序就不会通过弃兵来延缓丢象。另一个方法是让搜索更快更深：你的程序搜索的层数越多，因超过水平线而延缓丢象的做法，发生的可能性就越小。但是对于普通局面来说，最有效的做法就是让搜索深度更灵活，使得程序在丢兵的路线上搜索得更深，而在其它路线上不必要搜索得很深。

蛮力和选择性

在Shannon【申朗，参阅译文《电脑国际象棋简史》】最早关于电脑国际象棋的文章中，提到程序调整搜索深度的两种策略。

最明显的就是我给你们看的伪代码：全部范围，固定深度的蛮力搜索。只要把“深度”这个参数输入到你的程序中，每搜索一层就减一，到达零时停下来。这样做的好处在于，只要在搜索水平线以内，一些甚至很怪异的线路也看得到。但是高的分枝因子就意味着任何线路都不可能很深(即学士程度，对任何事物都只知道个皮毛【原文是“Bachelor's degree: knows nothing about everything”】)。更遭的是，程序可能会栽倒在水平线效应下。

Shannon提到的另外一个方法就是选择性裁剪：不是搜索固定的深度，而是通过搜索每个结点的部分着法来减少分枝因子(避免那些“明显是坏棋”的着法)。因此这样可以搜索得很深，但是有些线路完全看不见(即博士程度，只对某个狭窄方面的懂得很多【原文是“Ph.D.: knows everything about nothing”，源于一句用来讽刺博士的笑话：“A PhD knows more and more about less and less until he knows everything about nothing”】)。Shannon认为这个思想非常好，因为它更接近人类的思考方式。Turing【图灵，参阅译文《电脑国际象棋简史》】的思想有所不同，只搜索吃子的着法。更典型的做法是，对所有的子结点作评价，而只对最好的k个作展开，这里k是小于实际分枝因子的一个参数。

不幸的是，“明显是坏棋”的着法往往根本不坏，而是取得胜利的精彩弃子。如果你没有找到你应该走的那步着法，你就必须更努力地去找其他可以获胜的方法。更糟的是，对手可能会在后面几步作出有力的反击，如果你没有看出来，那么你会掉入陷阱从而输掉棋局。

如今没有哪个思想会被单纯地使用的。我们把两者结合起来：选择性地延伸。每条路线都搜索固定的深度，但是某些路线要搜索得比水平线深。有时我们也会做裁剪(不是像Alpha-Beta那样的安全裁剪)，这通常也是非常保守的，因为只挑出好的着法实在太困难了，但是有时对于确实很坏的着法，我们可以挑出并且忽略它们。除了国际象棋以外，有些棋类有更高的分枝因子，这就有必要使用更冒进的裁剪技术了。【例如五子棋，每个局面都有100种以上的合理着法，但是我们只会挑有意义的着法，要么有进攻作用(己方棋子周围的一些着点)，要么有防守作用(敌方棋子邻近的着点)，除此以外一概不予考虑。】

什么时候需要延伸?

延伸的目标是什么呢？是获得更好(更准确)的评价。所以下面的结点必须延伸：

(1) 目前的评价可能不准确时；

(2) 目前的着棋路线在整个博弈搜索树中是非常重要的；

或者以上两个情况的组合。

静态搜索

在国际象棋或其他棋类中，有吃子和不吃子的着法(西洋跳棋、围棋、Fanorano等)，如果有吃子的情况，那么每次吃子时评价都会改变。

“静态搜索”(Quiescence Search)的思想是，到达主搜索的水平线后，用一个图灵型的搜索只展开吃子(有时是吃子加将军)的着法。其他棋类不同于国际象棋，可能只包括一些会很大程度上改变评价的着法。静态搜索还必须包括放弃的着法，来决定停止吃子。因此，主Alpha-Beta搜索中每个调用评价函数的地方，都会被一个类似Alpha-Beta的但只搜索吃子着法的函数代替，如果当前结点的评价函数足以高出边界，那么就让搜索停下来。代码如下：

```
// 静态搜索
// 主Alpha-Beta搜索中，原来出现“eval()”的地方现在调用这个函数
quiesce(int alpha, int beta) {
   int score = eval();
```

```
    if (score >= beta) {
      return score;
    }
    for (每个吃子着法 m) {
      执行着法 m;
      score = -quiesce(-beta,-alpha);
      撤消着法 m;
      if (score >= alpha) {
        alpha = score;
        if (score >= beta) {
          break;
        }
      }
    }
    return score;
  }
```

有人还把将军加入到静态搜索中，但是你要当心，由于没有深度参数，静态搜索会有巨大的结点数。吃子通常是有限的(在棋子全部吃完以前你只能有16次子)，而将军可以一直进行下去并导致无限制递归。【对于是否展开将军着法的问题，可以尝试一种做法，如果局面被将军，就展开全部着法，即做应将处理，而不对当前局面作评价，参阅“静态搜索”一文。】

选择性延伸

如果局面在前面的路线中非常活跃，那么这就证明后面会有进一步的手段，或者前面的着法使得这些手段推迟了，从而在很深的地方会有好的局面。因此如果搜索到一个“感兴趣”的着法例如吃子或将军，就要增加搜索深度。在Alpha-Beta的伪代码中，调用搜索过程时参数“depth – 1”可以用“depth – 1 + extension”来代替。你必须小心不要经常做这些事，否则最终会把搜索树延伸得特别庞大(甚至可能无限大)。

有一个技巧可以使这样的延伸终止，即延伸一个分数的层数。比较好的做法是，“深度”参数记录的是你想要搜索的层数乘上一个因子，比如说“深度 = 层数 x 24”。在递归调用Alpha-Beta搜索的时候，就传递参数“Depth – 24 + Extension”。如果延伸值总是小于24，那么这个方法能保证终止，这个方法还可以有选择地多延伸些或少延伸些。

在评价函数里加上“局面如何难以评价”这个知识，也是有用的，这样就可以在局面太难评价的时候延伸搜索。我的程序就做到了这点，程序对当前结点调用评价函数。如果局面十分复杂，而且深度接近零，那么评价会返回一个特殊的值【表示评价失败，这个值不能在“–Infinity”和“Infinity”之间，否则会和正常值混淆】，通知搜索继续进行下去。如果深度达到一个负得很大的数【原作者的意思是评价连续失败导致延很长】，那么评价函数总是成功的，这样搜索将会终止。

如何把准确性和重要性结合起来?

到目前为止，我们都在讨论并试图找到评价局面可能不准确的原因。但是对于搜索树的一些不重要的部分，或许我们不在乎它不准确，而我们真正关心的是主要变例上的结点。我们如何来关注选择性延伸的重要性呢?

(1) 在Alpha-Beta搜索时，不要只根据准确性作延伸，而忽视了重要性;

(2) 对主要变例部分(或附近)的线路作延伸，例如单步延伸(Singular Extension)，深蓝(Deep Blue)及其前身就用这个方法，如果一个局面的某个着法远比其他着法好，就延伸这个着法。

(3) 把视线从Alpha-Beta上移开。有一个称为“对策数搜索”(Conspiracy Number Search)的策略，即某些局面会使得能应对的好的着法很少【根据原文，译者也无法了解这个策略的确切含义】，这些局面要搜索得深一些。

【选择性延伸通常运用在强制着法上，强制着法的界定在各个程序中有所不同，但主要有以下几种判断方法：

(1) 将军：此时必须应将，显然属于强制着法；

(2) 单一应着：走子的一方只有一种合理着法时，显然属于强制着法；

(3) 杀棋威胁：当一方不走子时就会被对方在几步内杀掉，那么解除杀棋威胁也属于强制着法，这种判断比较困难，通常利用下面所介绍的“空着搜索”来做判断。

(4) 吃回被吃棋子：这很有可能是兑子过程，因此大多数情况下为强制着法；

(5) 通路兵挺进：在王兵残局中，最简单的处理就是搜索到兵升变时的局面，从而绕开正方型法则、关键格理论等知识，这就需要对通路兵的挺进作延伸。(如果兵挺5格才能到达升变格，那么原来搜索8层还看不到升变，作了延伸以后搜索5层就能看到了，因为在连续挺兵的这条线路上已经延伸到了10层。)

大多数程序都结合以上若干种判断，以决定是否进行延伸。】

空着搜索

这个思想符合本文的整个主题，即在适当时机调整搜索层数，但是它是通过相反的方式来表现的。这个思想不是在复杂的局面上延伸，而是在简单的局面上减少搜索。

这个思想是建立在国际象棋知识的基础上的：有害的着子是非常罕见的(除了残局以外)。通常如果轮到你走，你一定能让局面更好些。所有可能的着法都使局面变得更糟，这样的局面称为“无等着”(Zugzwang)(德语，意思为强迫着子)，通常只发生在残局中。像其他棋类，例如五子棋，无等着根本不会发生。因此，如果你改变国际象棋的规则，允许有“弃权”的着法，那么弃权通常是错误的，它不会使棋局有进展。

因此，假设你搜索一个希望高出边界的结点(即Alpha-Beta搜索的返回值至少是Beta)，空着搜索就是先搜索“弃权”着法【即“空着”(Null-Move)】，即使它通常不是最好的。如果弃权着法高出边界，那么真正最好的着法也可能高出边界，你就可以直接返回Beta而不要再去搜索了。要把搜索做得更快，弃权着法搜索的深度通常比常规着法浅。

你必须小心，这种启发会改变搜索结果，也可能使你忽略棋局中的一些重要的线路。不要连续两次用空着(因为这样你的搜索会退化，结果只返回评价函数)，而且要小心，只能在不会出现无等着的情况下使用。在国际象棋中，这就意味着当局面还有很多子的时候才能使用。

```
// 带空着启发的Alpha-Beta搜索
alphabeta(int depth, int alpha, int beta) {
  if (赢棋 || depth <= 0) {
    return score;
  }
  放弃着子;
  if (上一步不是空着 && 局面不是无等着局面 &&
      alphabeta(depth - 3, beta, beta + 1) >= beta) {
    // 【深度参数如果是 depth - 1，那就是纯粹的“启发”算法，而现在搜索浅了(depth - 3)，因此称为“空着裁剪”更为恰当。】
    return beta;
  }
  /* 【“放弃着子”蕴涵着交换着子方的操作，空着启发做完后还必须交换回来。这样，上面用醒目颜色标出的代码(是由译者标出的)就存在一些问题，应该改为：

  if (上一步不是空着 && 局面不是无等着局面) {
```

```
      放弃着子;
      int val = alphabeta(depth - 3, beta, beta + 1);
      撤消放弃着子; // 如果只作简单处理，放弃着子和撤消放弃着子都只交换着子方。
      if (val >= beta) {
        return beta;
      }
    }
】 */
    for (每个可能的着法 m) {
      执行着法 m;
      alpha = max(alpha, -alphabeta(depth - 1, -beta, -alpha);
      撤消着法 m;
      if (alpha >= beta) {
        break;
      }
    }
    return alpha;
}
```

原文：http://www.ics.uci.edu/~eppstein/180a/990204.html
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇　基本搜索方法——置换表
- 下一篇　高级搜索方法——简介(二)
- 返　回　象棋百科全书——电脑象棋

《对弈程序基本技术》专题

Alpha-Beta搜索的诸多变种

David Eppstein */文
* 加州爱尔文大学(UC Irvine)信息与计算机科学系

尽管我们已经讨论过Alpha-Beta搜索简单有效，还是有很多方法试图更有效地对博弈树进行搜索。它们中的大部分思想就是，如果认为介于Alpha和Beta间的评价是感兴趣的，而其他评价都是不感兴趣的，那么对不感兴趣的评价作截断会让Alpha-Beta更有效。如果我们把Alpha和Beta的间距缩小，那么感兴趣的评价会更少，截断会更多。

首先让我们回顾一下原始的Alpha-Beta搜索，忽略散列表和“用当前层数调整胜利值”的细节。

```
// 基本的Alpha-Beta搜索
int alphabeta(int depth, int alpha, int beta) {
   move bestmove;
   if (棋局结束 || depth <= 0) {
      return eval();
   }
   for (每个合理着法 m) {
      执行着法 m;
      score = -alphabeta(depth - 1, -beta, -alpha);
      if (score >= alpha) {
         alpha = score;
         bestmove = m;
      }
      撤消着法 m;
      if (alpha >= beta) {
         break;
      }
   }
   return alpha;
}
```

超出边界(Fail-Soft)的Alpha-Beta

以上代码总是返回Alpha、Beta或在Alpha和Beta之间的数。换句话说，当某个值不感兴趣时，从返回值无法得到任何信息。原因就是当前值被变量Alpha所保存，它从感兴趣的值的窗口下沿开始一直增长的，没有可能返回比Alpha更小的值。一个对Alpha-Beta的简单改进就是把当前评价和Alpha分开保存。下面的伪代码用常数“WIN”表示最大评价，它可以在Alpha-Beta搜索中返回任何评价。

```
// 超出边界的Alpha-Beta搜索
int alphabeta(int depth, int alpha, int beta) {
   move bestmove;
   int current = -WIN;
   if (棋局结束 || depth <= 0) {
      return eval();
   }
```

```
    for (每个可能的着法 m) {
      执行着法 m;
      score = -alphabeta(depth - 1, -beta, -alpha);
      撤消着法 m;
      if (score >= current) {
        current = score;
        bestmove = m;
        if (score >= alpha) {
          alpha = score;
        }
        if (score >= beta) {
          break; // 【译注：这里可以直接返回score、current或alpha，如果返回beta，则是不会高出边界的Alpha-Beta搜索。】
        }
      }
    }
    return current;
}
```

这样改动以后，就可以知道比以前稍多一点的信息。如果返回值x等于或小于Alpha，我们仍然不知道局面的确切值(因为我们可能在搜索中裁剪了一些线路)，但是我们知道确切值最多是x。类似地，如果x大于或等于Beta，我们就知道搜索值至少是x。这些微小的上界和下界变化不会影响搜索本身，但是它们能导致散列表命中率的提高。超出边界的Alpha-Beta搜索对下面要介绍的MTD(f)算法有重要作用。

期望搜索

这个方法不是代替Alpha-Beta的，只是从外部改边一个途径来调用搜索过程。通常用Alpha-Beta找最好路线时，可以调用：

```
alphabeta(depth, -WIN, WIN)
```

这里 –WIN 和 WIN 之间的范围很大，表明我们不知道搜索值是多少，因此任何可能的值都必须考虑。随后，要走的那步保存在搜索函数外部的变量中。

期望搜索的不同之处在于，我们可以人为地指定一个狭窄窗口(用前一个搜索值为中心)，对搜索通常是有帮助的。如果你搜索失败，必须加宽窗口重新搜索：

```
// 期望搜索
int alpha = previous - WINDOW;
int beta = previous + WINDOW;
for ( ; ; ) {
  score = alphabeta(depth, alpha, beta);
  if (score <= alpha) {
    alpha  = -WIN;
  } else if (score >= beta) {
    beta = WIN;
  } else {
    break;
  }
```

```
}
```

权衡狭窄搜索所节约的时间，和因为失败而重新搜索的时间，可以决定常数 WINDOW 的大小。在国际象棋中，典型的值也许是半个兵。期望搜索的变体有，搜索失败时适当增加窗口宽度，或者用初始窗口而没必要以前一次搜索结果为中心，等等。

MTD(*f*)

这个技术跟期望搜索一样，只是在调用Alpha-Beta时对初始值进行调整。搜索窗口越窄搜索就越快，这个技术的思想就是让搜索窗口尽可能的窄：始终用“beta = alpha + 1”来调用Alpha-Beta。用这样一个“零宽度”搜索的作用是用Alpha和确切值作比较：如果搜索的返回值最多是Alpha，那么确切值本身最多是Alpha，相反确切值大于Alpha。

我们可以用这个思想对确切值作二分搜索：

```
int alpha = -WIN;
int beta = +WIN;
while (beta > alpha + 1) {
   int test = (alpha + beta) / 2;
   if (alphabeta(depth, test, test + 1) <= test) {
      beta = test;
   } else {
      alpha = test + 1;
   }
}
```

但是，这样会导致大规模的搜索(即 WIN 和 -WIN 的差的对数)。而MTD(*f*)的思想则是用超出边界的Alpha-Beta对搜索进行控制：每次调用超出边界的Alpha-Beta就会返回一个值更接近于最终值，如果用这个搜索值作为下次测试的开始，我们最终会达到收敛。

```
// MTD(f)
int test = 0;
for ( ; ; ) {
   score = alphabeta(depth, test, test + 1);
   if (test == score) {
      break;
   }
   test = score;
}
```

不幸的是，它和散列表之间的复杂作用会导致这个过程陷入无限循环，所以必须加上额外的代码，如果迭代次数太多而没有收敛，就必须中止搜索。

MTD(*f*)的一个大的优势在于我们可以简化Alpha-Beta搜索，因为它只需要两个参数(深度和Alpha)而不是三个。【据说这样做可以提高并行计算的效率，遗憾的是译者对并行计算一窍不通。】

【为了对MTD(*f*)有更详细的了解，译者查阅了该算法的发明者Plaat的网站，他提供的MTD(*f*)代码中用了两个边界，可以防止迭代过程中出现振荡而不收敛的情况，代码如下(原来是Pascal语言，现被译者转写为C语言)：

```
int MTDF(int test, int depth) {
   int score, beta, upperbound, lowerbound;
```

```
    score = test;
    upperbound = +INFINITY;
    lowerbound = -INFINITY;
    do {
      beta = (score == lowerbound ? score + 1 : score);
      score = alphabeta(depth, beta - 1, beta);
      (score < beta ? upperbound : lowerbound) = score;
    } while (lowerbound < upperbound);
    return score;
  }
```

而关于MTD(*f*)的使用另有以下几点技巧：

(1) 通常试探值并不一定设成零，而是用迭代加深的形式由浅一层的MTD(*f*)搜索给出；

(2) 局面评价得越粗糙，MTD(*f*)的效率越高，例如国际象棋中可使一个兵的价值由100降低为10，其他子力也相应比例降低，以提高MTD(*f*)的效率(但粗糙的局面评价函数却不利于迭代加深启发，因此需要寻求一个均衡点)；

(3) 零宽度窗口的搜索需要置换表的有力支持，因此称为“用存储器增强的试探驱动器”(Memory-enhanced Test Driver，即MTD)，它只需要传递两个参数(深度*n*和试探值*f*)，故得名为MTD(*n*,*f*)，缩写为MTD(*f*)。】

PVS

或许最好的Alpha-Beta变体，要算是这些名称了：负值侦察(NegaScout)和主要变例搜索(Principal Variation Search，简称PVS)。这个思想就是当第一次迭代搜索时找到最好的值，那么Alpha-Beta搜索的效率最高。对着法列表进行排序，或者把最好的着法保存到散列表中，这些技术可能让第一个着法成为最佳着法。如果真是如此，我们就可以假设其他着法不可能是好的着法，从而对它们快速地搜索过去。

因此PVS对第一个搜索使用正常的窗口，而后续搜索使用零宽度的窗口，来对每个后续着法和第一个着法作比较。只有当零窗口搜索失败后才去做正常的搜索。

```
// 主要变例搜索(超出边界的版本)
int alphabeta(int depth, int alpha, int beta) {
  move bestmove, current;
  if (棋局结束 || depth <= 0) {
    return eval();
  }
  move m = 第一个着法;
  执行着法 m;
  current = -alphabeta(depth - 1, -beta, -alpha);
  撤消着法 m;
  for (其余的每个着法 m) {
    执行着法 m;
    score = -alphabeta(depth - 1, -alpha - 1, -alpha);
    if (score > alpha && score < beta) {
      score = -alphabeta(depth - 1, -beta, -alpha);
    }
    撤消着法 m;
    if (score >= current) {
      current = score;
```

```
            bestmove = m;
            if (score >= alpha) {
               alpha = score;
            }
            if (score >= beta) {
               break;
            }
         }
      }
      return current;
   }
```

这个算法跟MTD(*f*)有个同样的优势，即搜索树的大多数结点都以零宽度的窗口搜索，可以用双参数的Alpha-Beta。由于“Beta > Alpha + 1”的调用非常少，因此不必担心额外的工作(例如保存最佳着法以供将来使用)会占用很多时间。【原作者的意思是，调用零宽度窗口的搜索时，可以免去保存最佳着法等操作，因此可以省下不少时间。】

推荐

我自己的程序结合了期望搜索(用在整个搜索过程以外)和PVS(用在搜索过程内部)。但是不同的棋类游戏不一样，由于这些搜索不难实现，所以要在这些方法中进行选择或调节参数，就必须对它们逐一实现并做一些试验。它们都必须返回同样的搜索值(如果受散列表影响，那么至少是相近的值【例如常规的Alpha-Beta搜索和超出边界的Alpha-Beta搜索，在使用散列表时可能会返回不同的值】)，但搜索的结点数会不同。在你的棋类的典型局面中，能使搜索树最小的方法则被采纳。

原文：http://www.ics.uci.edu/~eppstein/180a/990202b.html
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 高级搜索方法——简介(一)
- 下一篇 高级搜索方法——静态搜索
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

静态搜索

Bruce Moreland / 文

国际象棋中会有很多强制的应对。如果有人用马吃掉你的象，那么你最好吃还他的马。

Alpha-Beta搜索不是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，即使一方的后被捉。

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。

典型的静态搜索只搜索吃子着法。这会引发一个问题，因为在国际象棋中吃子通常不是强制的。如果局势很平静，而且你面对的吃子只有QxP(后吃兵，导致丢后)，你不会强迫后去吃兵的，所以静态搜索不应该强迫吃子。

因此，走子一方可以选择是吃子还是不吃子。这就好比打扑克牌时可以只用手中的牌而不去抓牌【译注：术语“Standing Pat”】。

代码如下：

```
int Quies(int alpha, int beta) {
   val = Evaluate();
   if (val >= beta) {
      return beta;
   }
   if (val > alpha) {
      alpha = val;
   }
   GenerateGoodCaptures();
   while (CapturesLeft()) {
      MakeNextCapture();
      val = -Quies(-beta, -alpha);
      UnmakeMove();
      if (val >= beta) {
         return beta;
      }
      if (val > alpha) {
         alpha = val;
      }
   }
   return alpha;
}
```

这看上去和“Alpha-Beta”非常相似，但是区别是很明显的。这个函数调用静态评价，如果评价好得足以截断而不需要试图吃子时，就马上截断(返回Beta)。如果评价不足以产生截断，但是比Alpha好，那么就更新Alpha来反映静态评价。

然后尝试吃子着法，如果其中任何一个产生截断，搜索就中止。可能它们没有一个是好的，这也没问题。

这个函数有几个可能的结果：可能评价函数会返回足够高的数值，使得函数通过Beta截断马上返回；也可能某个吃子产生Beta截断；可能静态评价比较坏，而任何吃子着法也不会更好；或者可能任何吃子都不好，但是静态评价只比Alpha高一点点。

注意这里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。

我的示范程序不检测将军，因此它不能找到杀棋。先询问要走的一方是否被将军，这是可以尝试的做法。如果被将军了，就不能用“Evaluate()”来尝试截断，而应该以一层的深度调用Alpha-Beta。例如：

```
int Quies(int alpha, int beta) {
  if (InCheck()) {
    return AlphaBeta(1, alpha, beta);
  }
  val = Evaluate();
  if (val >= beta) {
    return beta;
  }
  if (val > alpha) {
    alpha = val;
  }
  GenerateGoodCcaptures();
  while (CapturesLeft()) {
    MakeNextCapture();
    val = -Quies(-beta, -alpha);
    UnmakeMove();
    if (val >= beta) {
      return beta;
    }
    if (val > alpha) {
      alpha = val;
    }
  }
  return alpha;
}
```

这个版本会找到带有连续吃子的杀棋。至于用哪个版本，取决于个人喜好和测试结果。

“好的”吃子的界定也是仁者见仁智者见智的。如果你允许任何吃子，并且以任何原始的顺序来搜索，那么你会降低搜索效率，并且导致静态搜索的膨胀，从而大幅度降低搜索深度，并使程序崩溃。

有一些简单的做法可以避免静态搜索的膨胀。

MVV/LVA

MVV/LVA 意思是“最有价值的受害者/最没价值的攻击者”(Most Valuable Victim/Least Valuable Attacker)。这是一个应用上非常简单的着法排序技巧，从而最先搜索最好的吃子着法。这个技术假设最好的吃子是吃到最大的子。如果不止一个棋子能吃到最大的子，那么假设用最小的子去吃是最好的。

这就意味者PxQ(兵吃后)首先考虑(假设王的将军另外处理)。接下来是NxQ或BxQ，然后是RxQ、QxQ、KxQ。接下来是PxR、B/NxR、RxR、QxR、KxR，等等。

这个工作总比不做要好，但是很明显有很严重的问题。即使车被保护着，RxR仍旧排在PxB的前面。

MVV/LVA 可以解决静态搜索膨胀的问题，但是它仍然留给你比较庞大的静态搜索树。

MVV/LVA 的优势在于它实现起来非常方便，而且可以达到很高的NPS值(每秒搜索的结点数)。它的缺点是搜索效率低——你花大量的时间来评估吃亏的吃子，所以你的搜索不会很深。

SEE

SEE 意思是“静态交换评价”(Static Exchange Evaluation)。它比 MVV/LVA 更复杂，但是着法排序更准确，从而很多吃子都不必考虑。

在 MVV/LVA 中你无法去掉QxP的着法，因为很可能兵是没保护的，这就意味着QxP是好的着法。当然，如果兵是保护的，我不相信你还能在搜索这步时有什么好的发现。用了 SEE 后，如果QxP是吃亏的着法，你就可以把它挑出来，然后在吃子的顺序里把它排在最后，或简单地把它裁剪掉。

现在我还没有 SEE 的示范代码，你可以设计一个处理吃子的过程，然后根据它看上去能得到的价值进行排序。【当考虑吃子着法时，需要考虑能吃到的该子的所有攻击子，也要保护该子的所有防守子，因此处理吃子的过程是相当复杂的。】

SEE 能让你非常有效地裁剪掉坏的着法，很多重要的吃子不会被错误地裁剪，并且能改进着法的顺序。你能做的另一件事是，如果着法看起来还不算充分好，那么你可以裁剪掉这些好的着法。如果你领先一个车，那么得一个兵的吃子或许不值得在静态搜索中尝试。

我很怀疑用 SEE 的程序能比相同的但用了 MVV/LVA 的程序强。问题在于代码的编写上，需要很好地设计数据结构，才能让 SEE 变得有效。

静态搜索不是完美的

静态搜索极有可能会犯错误。这是一个不幸的现实，但是它更多地在搜索看上去非常愚蠢(例如一层的搜索，它根本不会是非常好的)的情况下会犯错误，因此这不是一个严重的问题。

如果可能用更准确的静态搜索而不降低速度，那么我肯定这个程序会比以前更强。但是我们必须明白的是，你在耗费时间的前提下试图让静态搜索更准确，需要找到平衡点。如果为了让静态搜索更聪明，你花费了几层完全搜索的时间，那么这就不值得让它更聪明了。

原文：http://www.seanet.com/~brucemo/topics/quiescent.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 高级搜索方法——简介(二)
- 下一篇 高级搜索方法——空着裁剪
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

空着向前裁剪

Bruce Moreland / 文

没有副作用即可获得额外的速度

空着向前裁剪(Null-Move Forward Pruning)，运用可能忽视重要路线的冒险策略，使得国际象棋的分枝因子锐减，它导致搜索深度的显著提高，因为大多数情况下它明显降低了搜索的数量。它的工作原理是裁剪大量无用着法而只保留好的。

这个技术在很多刊物上报道过，但是使得大家都来关注空着的，则是由Chrilly Donniger发表在1993年9月的ICCA杂志上的一篇文章。

试想国际象棋搜索树中的某个局面，你的程序将以*D*层搜索这个局面的每个着法。如果其中任何一个着法的分数超过Beta，你就会马上返回Beta。如果任何一个超过Alpha，但是没有超过Beta，你就要记住着法和分值，因为这有可能是主要变例的一部分。如果它们全部小于或等于Alpha，你就要返回Alpha。

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？”

记得在刚才，你没有问这个问题。你只是去找最佳的着法去打击对手。问对手是否会打击你，这个问题却有所不同。

但是事实证明很多情况下对手无法打击你。比如说你送了一个车，而其他棋子都没有作用，在这种情况下，对手随便走哪步你都吃亏，因为你丢了一个车。空着向前裁剪的要点，就是简单地去掉那些没用的着法，而不要在这上面多花时间。

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。

我们不讨论这个策略了，现在来谈它是如何运用在电脑国际象棋中的。在你搜索着法以前(事实上在你生成着法以前)，你做一个减少深度的搜索，让对手先走，如果这个搜索的结果大于或等于Beta，那么你简单地返回Beta而不需要搜索任何着法。

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。一个比较好*R*是2，如果你要对所有的着法搜索6层，你最终只对对手所有的着法搜索了4层。【译注：因为放弃着法后层数应该减1，所以实际在对手上面搜索的层数是$D-1-R$。】

这就使得很多时间节约下来了，实践证明可以使搜索增加一到两层。效果真的非常可观！

代码如下，醒目的文字是在Alpha-Beta搜索的基础上增加的部分：

```
#define R 2
int AlphaBeta(int depth, int alpha, int beta) {
  if (depth == 0) {
    return Evaluate();
  }
  MakeNullMove();
  val = -AlphaBeta(depth - 1 - R, -beta, -beta + 1);
  UnmakeNullMove();
  if (val >= beta) {
    return beta;
  }
```

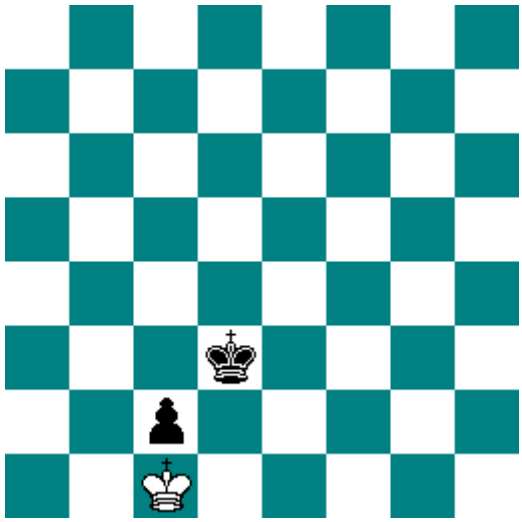
```
    GenerateLegalMoves();
    while (MovesLeft()) {
      MakeNextMove();
      val = -AlphaBeta(depth - 1, -beta, -alpha);
      UnmakeMove();
      if (val >= beta) { // 把这部分去掉，就用纯粹的最小-最大代替了Alpha-Beta。
        return beta;
      }
      if (val > alpha) {
        alpha = val;
      }
    }
    return alpha;
  }
```

在这个代码中我用了一个诀窍。我需要知道空着搜索的值是否是Beta或更好，如果还不如Beta，我不关心它到底比Beta有多糟，因此我用了极小窗口，试图让裁剪做得更快。实际上我用(Beta – 1, Beta)调用了搜索，但是由于递归时必须把Alpha和Beta颠倒并取负数，这就变成源代码中的样子了。

不用说，这个代码在一方被将军时不能发挥作用(因为对手立刻把王吃掉了)。什么地方允许调用空着向前裁剪，必须掌握好分寸，因为如果你允许一次连续地这么做，那么搜索将退化成什么都不做了。一个很简单的尝试，就是当中没有间隔一个实在着法的时候，不要让两个空着搜索连在一起。另一个思想是在一个实在着法之前，允许连续两个空着裁剪。实践证明这两个方法都做得很好。

嗯，还是有副作用的

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。下面是个典型的例子：

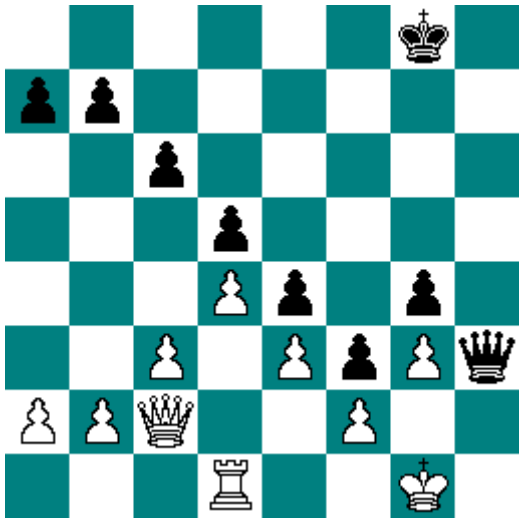


在这个局面里，如果白方被迫走子，他走 Kb2，而黑走 Kd2 助兵变后。如果白方不走棋，那么黑的走 Kc3，局面就是和棋。(事实上这是一个互相的无等着局面，因为双方都被先走棋所困扰，这个问题不在我们的讨论范围内。)

如果到达这个局面，而且试图用空着向前裁剪，那么可能会认为黑方没有威胁白方，因为现在黑方确实没威胁白方。而现在黑方在等待白方毁掉局势，这就完全不同了。

由于这个原因，空着向前裁剪不能在残局中使用，或者至少不能直接地使用。如果你试图在残局中用，则会出现很糟糕的事情。

有一个更困扰人的例子，是这样的：



这个局面选自Goetsch和Campbell的《空着启发的试验》(*Experiments with the Null-Move Heuristic*)一文，发表在《电脑、象棋与认知》(*Computers, Chess and Cognition*)(ISBN 0-387-97415-6)一书中。

这个局面轮到白方走，白的下一步就被将死了，而且无能为力。对这个局面做两层的搜索，毫无疑问：1. <任何着法> Qg2#。

如果你在这里试图用空着向前裁剪，那么不幸发生了。我们原来打算做两层的完全搜索，现在要做的是对对方做零层搜索，并试图找到威胁。在零层搜索中，黑方不走子，所以调用“评价”函数，由于白方领先一个车而返回 +5 左右。这或许会大于Beta，因此搜索返回Beta。

这是我们不希望发生的！搜索应该返回一个很小的值，表示白方被杀。

我们这里看到的是一种水平线效应，经常发生在启用空着向前裁剪的时候。没有空着向前裁剪时，白方走了一步没用的着法，然后有足够的搜索深度(在这个例子中只需要一层)让黑杀掉白。用了空着向前裁剪并且用一个足够高的R值(例如$R = 2$)，白方不做任何事情，但是黑方也没有足够深度来看到胜利了。

这个例子或许让你感到奇怪，或许它只会在很浅的搜索中才发生，但是有很多例子在足够的搜索深度下仍然会发生这样的事，即用正常的搜索可以看到的棋，用了空着向前裁剪后就被忽视了。实际上，这个黑后在h3格并且黑兵在f3格的棋型，对于国际象棋程序来说都是很危险的。【原作者的意思是，如果程序遇到这种棋型，应该考虑适当延伸搜索深度，或者用更小的R值做空着裁剪。】

空着向前裁剪的另一个问题在于它会导致“搜索的不稳定性”。空着搜索的成功与否取决于Beta的值。这个结点下次访问时，Beta可能不同，因此搜索会有不同的值，这是很不合理的。你可能在传递窗口(7, 30)时搜索高出边界，但是传递(7, 50)时，却低出边界。

在实战中，比起遇到偶然的对策错误，以及搜索不稳定性的增加来说，搜索速度的加快重要得多。

一些思想

尝试调整一个不同于2的R值，这是非常有趣的事。你可以试试1、3、4或更大的数，或者根据搜索深度、棋盘上的子力等等因素改变。我从来没有得到比直接用$R = 2$更好的结果，但是一些研究表明其他值或许会更好，而且有些人至少在搜索树的部分结点上用$R = 3$的。

通过一些验证式的搜索，我们也可以试图找到残局中使用空着向前裁剪的方法。如果你的空着搜索高出边界，你就减少深度来做常规搜索，看它是否也高出边界。我觉得这看上去有些破费，但是还是值得尝试的。

还有其他的改进方法值得尝试，但是我不想把它们一一列举出来。你可以去看Donninger的文章，看《电脑、象棋与认知》(*Computers, Chess and Cognition*)的文章，或者看Ernst Heinz的跟空着向

前裁剪有关的文章。

【译者有必要在这里介绍一下这两个思想：

(1) 根据不同情况来调整*R*值的做法，称为“适应性空着裁剪”(Adaptive Null-Move Pruning)，它首先由Ernst Heinz发表在1999年的ICCA杂志上。其内容可以概括为：

a. 深度小于或等于6时，用$R = 2$的空着裁剪进行搜索；

b. 深度大于8时，用$R = 3$；

c. 深度是6或7时，如果每方棋子都大于或等于3个(译者没注意到是否包括王)，则用 $R = 3$，否则用 $R = 2$。

参阅：Heinz EA: ***Adaptive Null-Move Pruning***, *ICCA J.* 22 (3): 123-132, **1999**

(2) 验证空着裁剪是否安全的做法，称为“带验证的空着裁剪”(Verified Null-Move Pruning)，它首先由Tabibi发表在2002年的ICGA(原ICCA)杂志上。其内容可以概括为：

a. 用$R = 3$的空着裁剪进行搜索；

b. 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)；

c. 做浅一层的搜索时，子结点用$R = 3$的不带验证的空着裁剪；

d. 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。

参阅：Tabibi OD, Netanyahu NS: ***Verified Null-Move Pruning***, *ICGA J.* 25 (3): 153-161, **2002**】

原文：http://www.seanet.com/~brucemo/topics/nullmove.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 高级搜索方法——静态搜索
- 下一篇 高级搜索方法——期望窗口
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

期望窗口

Bruce Moreland / 文

期望窗口是对迭代加深的改进。迭代加深的最简单的实现方法是这样的：

```
for (depth = 1; ; depth ++) {
   val = AlphaBeta(depth, -INFINITY, INFINITY);
   if (TimedOut()) {
      break;
   }
}
```

这里调用了一个“窗口”为正负无穷大的Alpha-Beta搜索，以假定返回值可能是很大的正数或负数。

假设下一次迭代时，搜索的值不会比上一次有太多的变动，那么就可以对以上的程序作改进，代码如下：

```
#define valWINDOW 50 // 【译注：valWINDOW是窗口的半宽，在国际象棋中通常是半个兵的价值。】
alpha = -INFINITY;
beta = INFINITY;
for (depth = 1; ; ) {
   val = AlphaBeta(depth, alpha, beta);
   if (TimedOut()) {
      break;
   }
   if ((val <= alpha) || (val >= beta)) {
      alpha = -INFINITY;
      beta = INFINITY;
      continue;
   }
   alpha = val - valWINDOW;
   beta = val + valWINDOW;
   depth ++;
}
```

这个代码有点烂，如果可能的话你可以把“continue”前面的部分改一下试试。这个思想是用前一次Alpha-Beta迭代的值来作用到这次的迭代中。大多数情况下你得到的返回值会在Alpha和Beta之间。如果没有，那么你可以尝试一个宽一些的窗口。

我的重新搜索并不那么有效，但是我相信你可以做得更好。一个很明显的思想是，如果返回值大于或等于Beta，你就扩展Beta，如果返回值小于等于Alpha，你就扩展Alpha，而不是我所做的Alpha和Beta都扩展。

搜索不稳定性的问题

这是你要小心搜索不稳定性的另一个地方。当我首次在我的国际象棋程序中加入期望窗口时，我设想如果返回值大于或等于Beta，那么重新搜索的结果就应该得到更大的值。完全错了，我的程序陷入了非常严重的故障！下面是它的代码：

```
alpha = -INFINITY;
beta = INFINITY;
for (depth = 1; ; ) {
   val = AlphaBeta(depth, alpha, beta);
   if (TimedOut()) {
      break;
   }
   if (val <= alpha) { // 错!
      alpha = -INFINITY;
      beta = val + 1;    // 【这行没有就对了，但效率会低很多。】
      continue;
   }
   if (val >= beta) {   // 错!
      beta = INFINITY;
      alpha = val - 1;
      continue;
   }
   alpha = val - valWINDOW;
   beta = val + valWINDOW;
   depth++;
}
```

如果你能确认重新搜索会有结果，那么上面的代码会非常有效的。但是在我这里情况却是：
1. 搜索(Alpha, Beta)时高出边界；
2. 重新搜索(Beta - 1, INFINITY)时低出边界；
3. 重新搜索(-INFINITY, Alpha + 1)时高出边界；
4. 等等，等等……

无论如何你必须避免发生这种情况。

《对弈程序基本技术》专题

搜索的不稳定性

Bruce Moreland / 文

没有这个，生活会更有趣

当你试图写很强或很完美的程序时，搜索的不稳定性就可能出现。有很多原因可以导致不稳定性，当我讨论搜索的诸多改进方法时，顺便讨论了它们是如何导致搜索不稳定的。其他我没有讨论的搜索技巧也必须考虑不稳定的可能。

不稳定的搜索会返回无效的值，你用(5, 25)的Alpha-Beta窗口会高出边界，因此你用(24, INFINITY)重新搜索，却低出边界。这不应该发生，因为高出边界很明显说明返回值应该是25或者更高，那怎么又会低出边界呢？

事实就是如此，很多工作可以让国际象棋程序运行得更快或更好，但是它们或许会做一些蠢事，在用不同的窗口做搜索时返回略微不同的值。如果你没有得到你所期望的值，那么你的程序可能会陷入故障，或者产生一个使你的程序走出昏着的错误。

一些国际象棋的程序设计师没有把握好搜索不稳定性的思想，他们宁可不用非常好的搜索算法，以避免这种情况的发生，或者他们认为这样就能够避免。

我希望有可能完全排除搜索的不稳定性，但是就目前使用的非常基本的技术而言，很存在问题。我想解决办法就是对故障作一些防御，而别去深究不稳定性的原因。

原文：http://www.seanet.com/~brucemo/topics/instability.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译

- 上一篇 高级搜索方法——主要变例搜索
- 下一篇 局面评估函数——简介(一)
- 返　回 象棋百科全书——电脑象棋

【译者研究认为，在用了后面要谈到的“主要变例搜索”以后，期望窗口就几乎没有作用了，因此很多程序都没有使用这个技术，尽管它的思想非常明显，看上去似乎可以起到好的效果。】

原文：http://www.seanet.com/~brucemo/topics/aspiration.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 高级搜索方法——空着裁剪
- 下一篇 高级搜索方法——主要变例搜索
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

主要变例搜索

Bruce Moreland / 文

对Alpha-Beta的改进

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

在Alpha-Beta搜索中，任何结点都属于以下三种类型：

1. Alpha结点。每个搜索都会得到一个小于或等于Alpha的值，这就意味着这里没有一个着法是好的，可能是因为这个局面对于要走的一方太坏了。

2. Beta结点。至少一个着法会返回大于或等于Beta的值。

3. 主要变例(PV)结点。有一个或多个着法会返回大于或等于Alpha的值(即PV着法)，但是没有着法会返回大于或等于Beta的值。

有些时候你可以很早地判断出你要处理的是哪类结点。如果你搜索的第一个着法高出边界(返回一个大于或等于Beta的值)，那么很明显你得到Beta结点。如果低出边界(返回一个小于或等于Alpha的值)，假设你的着法顺序非常好，那么你有可能得到Alpha结点。如果返回值在Alpha和Beta之间，你可能得到PV结点。

当然，有两种情况你可能会判断错误。当你高出边界时，你返回Beta，因此你不会犯错误，但是如果第一个着法低出边界或者是PV着法时，仍然有可能在下一个着法得到更高的值。

主要变例搜索作了假设，如果你在搜索一个结点时找到一个PV着法，那么你就得到PV结点。也就是说假设你的着法排序已经足够好了，使得你不必在其余的着法中找更好的PV着法或者高出边界的着法(这就会使结点变成Beta结点)。

你找到一个着法其值在Alpha和Beta之间，那么对其余的着法，搜索的目标就是证明他们都是坏的。跟要搜索出更好的着法相比，这种搜索也许要快一些。

如果这个算法发现判断是错的，即其中一个后续着法比第一个PV着法好，那么它会被再一次搜索，这次使用正常的Alpha-Beta搜索方法。这种情况有时会发生，这样就浪费时间了，但是这些时间通常不会超过面所说的“证明是坏着法”所节约下来的时间。

算法如下，是从Alpha-Beta算法改过来的，改过的地方用醒目的字标出：

```
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;
    }
```

算法的核心部分就是函数中间醒目的“if”块中的内容。如果没有找到PV结点，“AlphaBeta()”函数就正常调用，如果找到了一个，那么情况就变了。不是用常规的窗口(Alpha, Beta)，而是用(Alpha, Alpha + 1)来搜索。这样做的前提是，搜索必须返回小于或等于Alpha的值，如果确实这样，那么把窗口的上面部分去掉就会导致更多的截断。当然，如果前提是错的，返回值是Alpha + 1或更高，那么搜索必须用宽的窗口重做。

据报道PVS可以提高10%的效率。我没有试图检测PVS用在我的程序里到底提高了多少，但是确实提高了，所以我用了这个算法。

搜索不稳定性的问题

如果你用(Alpha, Alpha + 1)这个窗口去做搜索，返回值超过了窗口(但是没有超过Beta)，你就必须重新搜索。你认为重新搜索的值必定在Alpha和Beta之间，但是恐怕不一定是。这很有可能是由“搜索的不稳定性”引起的，我会在别的章节中讨论这个问题。

上面写的那个程序对这个情况作了防御，并对这种情况的发生作了正确的处理。如果你要使用这个程序并且作一些改动，就要特别当心你的搜索是否总是稳定的。如果你得到不期望得到的返回值，就必须采取措施避免让程序陷入故障。

原文：http://www.seanet.com/~brucemo/topics/pvs.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译

- 上一篇 高级搜索方法——期望窗口
- 下一篇 高级搜索方法——搜索的不稳定性
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

搜索的不稳定性

Bruce Moreland / 文

没有这个，生活会更有趣

当你试图写很强或很完美的程序时，搜索的不稳定性就可能出现。有很多原因可以导致不稳定性，当我讨论搜索的诸多改进方法时，顺便讨论了它们是如何导致搜索不稳定的。其他我没有讨论的搜索技巧也必须考虑不稳定的可能。

不稳定的搜索会返回无效的值，你用(5, 25)的Alpha-Beta窗口会高出边界，因此你用(24, INFINITY)重新搜索，却低出边界。这不应该发生，因为高出边界很明显说明返回值应该是25或者更高，那怎么又会低出边界呢?

事实就是如此，很多工作可以让国际象棋程序运行得更快或更好，但是它们或许会做一些蠢事，在用不同的窗口做搜索时返回略微不同的值。如果你没有得到你所期望的值，那么你的程序可能会陷入故障，或者产生一个使你的程序走出昏着的错误。

一些国际象棋的程序设计师没有把握好搜索不稳定性的思想，他们宁可不用非常好的搜索算法，以避免这种情况的发生，或者他们认为这样就能够避免。

我希望有可能完全排除搜索的不稳定性，但是就目前使用的非常基本的技术而言，很存在问题。我想解决办法就是对故障作一些防御，而别去深究不稳定性的原因。

原文：http://www.seanet.com/~brucemo/topics/instability.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译

- 上一篇 高级搜索方法——主要变例搜索
- 下一篇 局面评估函数——简介(一)
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

评价函数

David Eppstein */文
* 加州爱尔文大学(UC Irvine)信息与计算机科学系

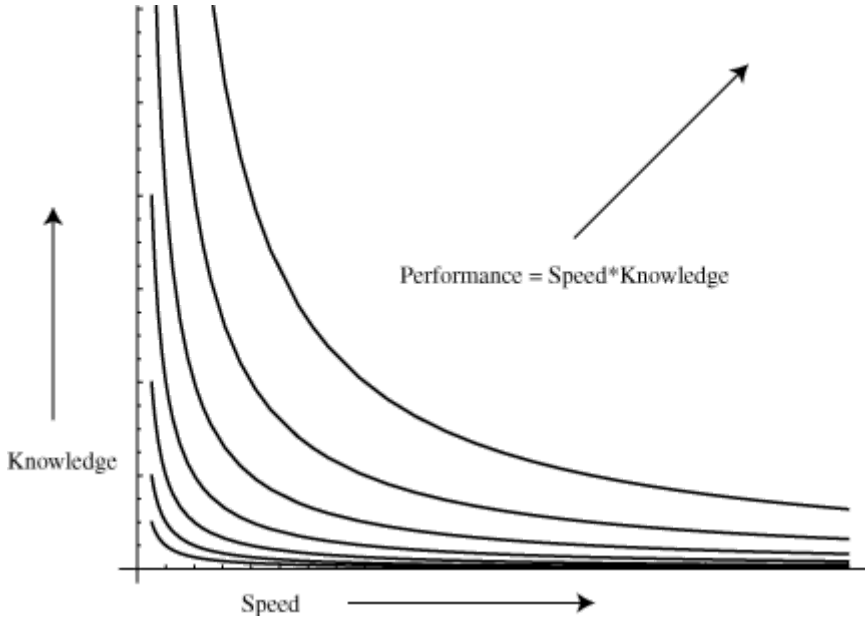
整体考虑

在你的程序中，评价函数综合了大量跟具体棋类有关的知识。我们从以下两个基本假设开始：

(1) 我们能把局面的性质量化成一个数字。例如，这个数字可以是对取胜的概率作出的估计；但是大多数程序不给这个数字以如此确定的含义，因此这仅仅是一个数子而已。

(2) 我们衡量的这个性质应该跟对手衡量的性质是一样的(如果我们认为我们处于优势，那么反过来对手认为他处于劣势)。真实情况并非如此，但是这个假设可以让我们的搜索算法正常工作，而且在实战中它跟真实情况非常接近。

评价可以是简单的或复杂的，这取决于你在程序中加了多少知识。评价越复杂，包含知识的代码就越多，程序就越慢。通常，程序的质量(它棋下得怎样)可以通过知识和速度的乘积来估计：



因此，如果你有一个快速而笨拙的程序，你通常可以加一些知识让它慢下来，使它工作得更好。但是同样是增加知识让程序慢下来，对一个比较聪明但很慢的程序来说，可能会更糟；知识对棋力的增长作用会减少的。类似地，你增加程序的速度，到一定程度后，速度对棋力的提高作用也会减少，你最好在速度和知识上寻求平衡，达到图表中间的位置。平衡点也会随着你面对的对手而改变；对于击败其他电脑，速度的表现更好，而人类对手则善于寻找你的程序中对于知识的漏洞，从而轻松击败基于知识的程序。【译注：如果你的程序要和人类棋手比，那么最好给程序加上足够多的知识。】

实现方法

就评价方法而言主要有两个类型。第一个是“终点评价”(End-Point Evaluation)，即用你擅长的评价算法，简单地评价每个局面，而不受其他局面的影响。这通常会给出好的结果，但是非常慢。

因此一些程序设计师用了下面的诀窍，称为预先计算(Pre-Computation)，一阶评价(First-Order Evaluation)，或棋子-格子数组(Piece-Square Tables)。

在我们对一个局面搜索最佳着法之前，我们认真检查棋局本身，在数组T[格子，棋子类型]中保存计算值。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评价值：

score += T[新的格子，棋子] - T[旧的格子，棋子]

下面就举一个例子说明国际象棋中的棋子-格子数组：当王被易位到棋盘的角落时，王前面的几个兵对防御来说是非常有用的。它们前进后防御能力就变差。因此，如果搜索的开始局面王在角落里，我们就应该为这些兵建立一个棋子-格子数组，其值如下：

...	...	...	...	...
...	1	1	1	1
...	1	1.1	1.1	1.1
...	1	1.2	1.2	1.2
...	-	-	-	-

在王前的三列中，为了使兵尽量离王近些，就在距离近的时候给它们更高的值。

不幸的是棋子-格子数组会很快失效，如果你要通过棋子-格子数组来增加一些知识，那么这种方法会显得非常愚蠢。在棋子-格子数组建立之初，这些联系就根据棋子的原始位置粗略计算好了，因此它们不能建立起几个移动过的棋子之间的联系。因此，如果我们搜索很长的一系列着法，例如王走到了棋盘的另半边，那么原来的棋子-格子数组的值就会非常不准确，因为它只是让兵防御王原来待的地方，而不是防御王本身。

用棋子-格子数组的程序通常需要结合终点评价。另一个建立棋子-格子数组的策略，就是把数组的建立延迟到后面的搜索中。例如，你要搜索9步后续着法，那么可以在5步的后续着法下建立数组，为剩下的4步搜索作准备。如果你想这么做，就应该使一个5步着法产生的数组和其他着法产生的数组保持一致，使得所有的评价值都有可比性。在我的课上O. Dave提出另一个改进的建议：用增量的办法修改棋子-格子数组，例如当王走掉时，对王前几个兵的奖励值也去掉了。这看上去是个不错的思想，但是我不知道如何来实现，也不知道如果实现了，效果会是怎样。

如何组合评价要素

把评价要素组合起来，通常就和上面所说的一阶评价一样，评价函数是很多项的和，每一项是一个函数，它负责找到局面中的某个特定因素。为什么要用加法呢？因为这种简单的组合信息的方法在实践中非常好用。

我自己感觉，棋类程序应该充分尝试各种可能的评价函数：把各种胜利的可能性结合起来，包括很快获胜(考虑进攻手段)，很多回合以后能获胜，以及在残局中获胜(国际象棋中就必须考虑通路兵的优势)的可能性，然后把这些可能性以适当的方式结合起来。如果黑方很快获胜的可能性用bs表示而白方用ws，在很多回合以后获胜(即不是很快获胜)的可能性是bm或wm，而在残局中获胜的可能性是be或we，那么整个获胜的可能性就是：

bs + (1 - bs - ws) * bm + (1 - bs - ws - bm - wm) * be，或者
ws + (1 - bs - ws) * wm + (1 - bs - ws - bm - wm) * we

我想，通过和类似上面的公式把若干单独概率结合起来，在评价函数中或许是个很好的估计概率的思路。每种概率是否估计得好，这就需要用程序的估计来和数据库中棋局的真实结果来作比

较，这就需要让程序具有基本判断的能力(判断某种攻击是否能起到效果)。但是这纯粹是我的假想，并没有在程序中测试过，如果你只用加法将不会犯多大的错。

评价函数中要加入哪些信息?

典型的评价函数，要把下列不同类型的知识整理成代码，并组合起来：

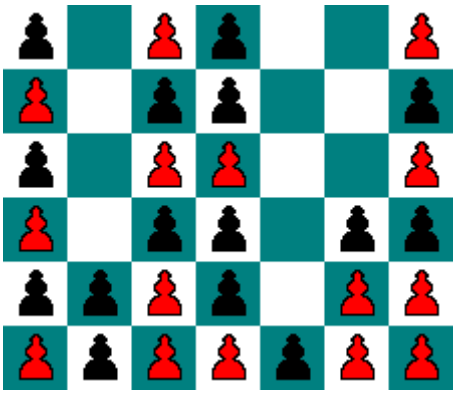
(1) 子力(Material)：在国际象棋中，它是子力价值的和，在围棋或黑白棋中，它是双方棋盘上棋子的数量。这种评价通常是有效的，但是黑白棋有个有趣的反例：棋局只由最后的子数决定，而在中局里，根据子力来评价却是很差的思路，因为好的局势下子数通常很少。其他像五子棋一样的游戏，子力是没有作用的，因为好坏仅仅取决于棋子在棋盘上的位置，看它是否能发挥作用。

(2) 空间(Space)：在某些棋类中，棋盘可以分为一方控制的区域，另一方控制的区域，以及有争议的区域。例如在围棋中，这个思想被充分体现。而包括国际象棋在内的一些棋类也具有这种概念，某一方的区域包括一些格子，这些格子被那一方的棋子所攻击或保护，并且不被对方棋子所攻击或保护。在黑白棋中，如果一块相连的棋子占居一个角，那么这些棋子就不吃不掉了，成为该棋手的领地。空间的评价就是简单地把这些区域加起来，如果有说法表明某个格子比其他格子重要的话，那么就用稍复杂点的办法，增加区域重要性的因素。

(3) 机动(Mobility)：每个棋手有多少不同的着法？有一个思想，即你有越多可以选择的着法，越有可能至少有一个着法能取得好的局势。这个思想在黑白棋中非常有效，国际象棋中并不那么有用。(它也曾被使用，但现在国际象棋程序设计师们把它从程序中去掉了，因为它看起来对整个局面的评价质量没什么提高。)

(4) 着法(Tempo)：这和机动性有着密切的联系，它指的是在黑白棋或连四子棋中(以及某些国际象棋残局中)，某方被迫作出使局面变得不利的着法。和机动性不同的是，起决定作用的是着法数的奇偶而不是数量。

例如，考虑下面的连四子棋局面：



【连四子棋的规则是：每个着子都必须位于某列的最底下一个空格，获胜条件是直线、横线或斜线有四个子相连。可以把连四子棋想象成带重力的五子棋。】

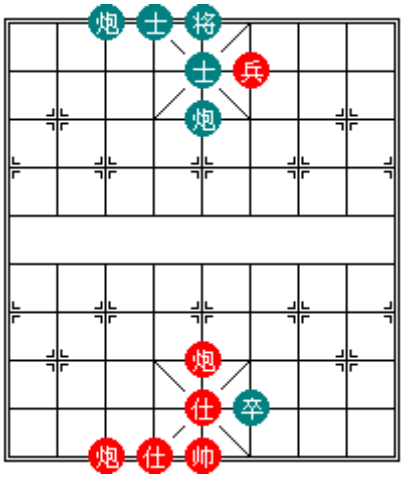
1、3、4、7列已经填满，因此只能在2、5、6列落子。第6列的着子是中立的，哪方都不能利用该列得胜或失利。黑方控制第2列，即红方不能在这里落子，因为这样可以让黑连成四子【注意第3列到第5列，已经有3个黑子形成斜线】。任何一方都不能在第5列着子，因为对方可以马上胜利【注意该列倒数第3行的空格，任何一方走到该格，都会在第4列到第7列(红方斜线，黑方横线)连成四子】。如果接下来是红先走，那么在第6列走了3步之后，黑方被迫走第2列放弃对该列的控制，又是3步后只能走第5列让红方取得胜利。但是如果下一步是黑走，那么3步之后会迫使红方输棋。

像这种连四子棋的残局中，偶数空格的列是无关紧要的，重要的是计算只有一方可以走的奇数列。如果一方控制更多的奇数列，那么他就可能赢。如果双方控制的奇数列一样多，如上面的棋盘

所示(没有一列受红方控制，而黑方只控制一个偶数列)，那么中立列的数量就非常重要了——如果它是奇数那么先走的一方会赢，如果它是偶数那么先走的一方会输。当然，这个简单的分析是建立在掌握复杂局势的基础上的，需要知道一方是如何控制某列上方的格子的。

像围棋这样的游戏，并不存在这样严格的奇偶规则，但是哪方有“主动权”【即“先手”】仍然很重要，一方能选择走哪里，而另一方只能在同一个地方被迫应对。走一系列着子，每步都赢得一块小的地盘并让对手被迫应着，然后再来走棋以取得大地盘并让对手获得主动权，这通常是个好的思路。【这里指的是在收官阶段，先走先手的小官子，然后再走后手的大官子。】

【这里有一个中国象棋的排局，也包括类似奇偶性的主动权问题：



在这个局面中，双方的兵(卒)都不能离开原位，否则对方平帅(将)即可造成铁门栓杀。双方的中炮不能离开中线，而三七路炮也不能离开该线，否则对方就会有闷宫杀。这样的棋型只能有一种取胜方法——用自己的两个炮顶住对方的两个炮，迫使对方让开兵(卒)或三七路炮。

这就衍生出一个数学游戏：有两堆石子，双方轮流从石子中拿去几颗，每次只能从一堆石子中拿走至少一颗石子，先拿完最后一堆者获胜。这个游戏的诀窍是：始终让对方面临两堆石子一样多的窘境。上面这个象棋局面中，两路炮之间的空格就好比两堆石子的数量，现在先走一方占有主动，因为两堆石子数量不一样多，他只要走一步让两堆石子数目一样就可以了。以红方先走为例，红方杀法及其黑方最顽强的抵抗如下：

1. 炮七进四 炮 3 进 1
2. 炮五进一 炮 3 进 1
3. 炮五进一 炮 3 进 1
4. 炮五进一 炮 3 进 1
5. 炮五进一 炮 3 退 1

若黑走炮 3 平 5，则仕五进六、前炮平 2、炮七平五做杀无解(若黑走炮 2 平 5 解杀则构成长将)。

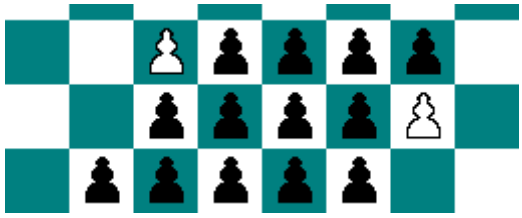
6. 炮七进一 炮 3 退 1
7. 炮七进一 炮 3 退 1
8. 炮七进一 炮 3 退 1
9. 炮七进一 卒 6 平 7
10. 帅五平四 卒 7 进 1
11. 帅四进一 卒 7 平 8
12. 兵四进一

红方第一步若不走炮七进四，不管进哪个炮，主动权都让给了黑方，走炮七进八可以守和，其他着法都会让黑取胜。可见，主动权这一问题在很多棋类中都是存在的，然而这个知识写入棋类程序中很有难度。】

(5) 威胁(Threat)。对手是否会有很恶劣的手段？你有什么很好的着法？例如在国际象棋或围棋中，有什么子可能要被吃掉？在五子棋或连四子棋中，某一方是否有可以连起来的子？在国际象棋或西洋棋中，有没有子将会变后或变王？在黑白棋中，一方是否要占角？这个因素必须根据威胁的远近和强度来考虑。

(6) 形状(Shape)。在围棋中，如果连起来的一串子围成两个独立的区域(称为“眼”)，那么它们就是安全的。在国际象棋中，并排的兵通常要比同一列的叠兵强大。形状因素是非常重要，因为局面的长远价值在几步内不会改变，也不会因为搜索而变化，这正是形状因素需要衡量的。(搜索可以找到短期的手段来改进局面，所以评价本身需要包括更多的长远眼光，使得搜索可以察觉到。)【在中国象棋中，空头炮(指被对手摆空头炮)和窝心马是不好的形状，不太深的搜索不会察觉到它们的坏处，但是长远来看是这些形状会存在严重弊端，大多数程序的评价函数会直接对空头炮和窝心马进行罚分。】

(7) 图案(Motif)。一些常见的具有鲜明特点的图案，蕴涵着特殊的意义。例如在国际象棋中，象往往可以吃掉边兵，却会被边上的兵困住。当象被困住时，对手可能还需要很多步才会吃掉它，因此被困的情形不容易被计算机的搜索程序所发现。有些程序通过特殊的评价因素来警告电脑，吃掉那个边兵可能会犯错误。在黑白棋中，在角落的邻近格子上放一个子来牺牲一个角，往往是非常有用的，这样当对手占领这个角时，就可以在这个子的边上放一个提不掉的子，从而在另一个角上取得优势。



上图中，白方牺牲了右下角。

当黑方走右下角时，白方在边上走子，然后赢得了左下角。【黑方只得到一个角，而白方得到了一个角连同边线上的6子，白大优。】

对这种牺牲做特别的评价也许是很有必要的，它可以决定是否需要做牺牲，或者来衡量边线上的子是否能抵挡这种牺牲手段。

原文：http://www.ics.uci.edu/~eppstein/180a/990114.html
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 高级搜索方法——搜索的不稳定性
- 下一篇 局面评估函数——简介(二)
- 返 回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

调整评价函数

David Eppstein */文
* 加州爱尔文大学(UC Irvine)信息与计算机科学系

上次我谈到了局面评价中的很多函数，把这些函数加起来就可以组合成评价函数。但是数值从哪里来?

例如在黑白棋中，你可以说出四种函数:

(1) *f*(局面) = 子力(我的子数 – 对手的子数);

(2) *g*(局面) = 角(我控制的 – 对手控制的);

(3) *h*(局面) = 机动性(我可以走的)。

你必须组合这些函数(可能还有其他项)来构成一个评价函数: eval = $a \cdot f + b \cdot g + c \cdot h$。例如，你可以尝试: eval = $-1 \cdot f + 10 \cdot g + 1 \cdot h$。但是这些数值从哪里来？哪种数值的组合可以得到最好的效果?

下面是手工找到这些数值的一些方法:

(1) 规格化(Normalize)。如果你只关心评价的顺序，而通常不怎么关心评价值，那么你就可以把每一项都乘以同样的常数。这就意味着你对某个特定的项目(比如说兵的价值)可以硬性设一个值，其他值就表示成它们相当于多少个兵。这个做法可以让你减少一个需要设定的参数。

(2) 约束法(Deduce Constraints)。你希望让电脑作出什么样的判断，考虑这些问题就可以确定一些参数了。例如在国际象棋中，即使你赚到一个兵，用车换象或马通常还是坏的，但是如果你赚到两个兵那还是好的，因此子力价值要满足R>B+P(防止换单兵)和R<B+2P(鼓励换双兵)。这样的不等式你给得越多，合适的权重组合就越少。在一开始设定权重值的时候，这个方法通常可以得到合适的值，但是后面你仍然需要做一些调整。

(3) 交手法(Hand Tweaking)。这是很常用的方法，仅仅是让你的程序对弈足够多的次数，来找到它的优势和弱点，猜测哪些参数会让程序更好，然后挑选新的参数。这个方法可以很快得到合理的结果，但是你需要对这种棋类有足够的了解，这样就可以根据程序的对局来做分析，知道程序的问题在哪里。(也就是说，当程序很笨但是你很聪明时，这个方法最有用。)

不需要人工干预的方法有:

(4) 爬山法(Hill-Climbing)。类似于交手法，每次对权重作很小的改变，测试改变后的表现，仅当成绩提高时才采纳这个改变，需要重复很多次。这个方法看上去很慢，并且只能找到“局部最优”的组合(即评价可能很差，但是任何很小的改变都会使评价更差)。

(5) 模拟退火法(Simulated Annealing)。类似于爬山法，也是对权重做改变来提高成绩的。但是如果改变没有提高成绩，有时候(随机地，给定一个几率)也采纳改变，试图跳出全局最优。这个方法需要给定一些几率，从几率高、梯度大的条件开始，然后逐渐减小。模拟退火法比爬山法更慢，但是最终可能得到比较好的值。

(6) 遗传算法(Genetic Algorithms)。爬山法和模拟退火法可以得到一组好的权重，它们是逐渐变化的。相反，遗传算法可以得到几组不同的好的权重，不断增加新的组合跟原来的做比较(取用某组中的某个权重，另一组中的另一个权重，互相交换得到新的)，通过淘汰坏的组合来控制种群的数量。

(7) 神经网络(Neural Networks)。实际上这更多地是一种评价函数的类型，而不是用来选择权重的：神经元是阈值(输入权重的和)的函数，第一层神经元输入的关于局面的性质(例如位棋盘表示中的某几个位)就可以构造网络，然后前一层的结果输入到后一层。因此单输入神经元的单层网络就等同于我们上次讨论过的一阶评价函数，但是接下来就可以构造更复杂的神经网络了，而且用这种方法作为评价函数是不难的(只要根据输入的改变来重新计算神经元的输出就可以了)。问题仍然像前面所说的，如何设置权重？除了前面的方法外，针对神经网络还发展出一些方法，例如“暂时差别学习”(Temporal Difference Learning)。其基本思想是确定网络何时会作出坏的评价，并且让每个权重增加或减小看是否会评价得更好，这很类似于爬山法。跟其他自动学习的方法相比，神经网络的好处就在于它不需要很多人类的智慧：你不需要懂得太多的棋类知识，就可以让程序有个比较好的评价函数。但是根据目前我们掌握的情况，根据自己的智慧来做评价函数，要比机器学习做得好，并且做得快。

这些方法都需要依靠一些自动化的技术，以便对程序的性能进行评估：

(1) 我们可以用程序处理大量的测试局面(比如人类棋手的高质量对局中提取的局面)并且看它是否得到正确的结果。

(2) 我们可以让程序对阵一些著名的对手(比如其他程序)来看它能赢几盘。或者我们可以让程序和它自己对阵，以及和它自己的其他版本对阵，例如在爬山法中，用修改过的版本对阵没修改过的版本。这个方法有自身的缺点，除非系统中增加一些随机因素，否则两个程序每次会下出一样的棋，因此你只是看到一局棋的结果而无法代表全部比赛。一个合适的方法就是拿一组测试局面，从每个局面开始就可以下出不同的棋。

(3) 我们可以比较两个结果，一个用评价函数得到，另一个用评价和搜索相结合得到。如果评价是好的，那么两者应该接近，但是反过来说行吗？

那么如何来自动掌握评价中的权重呢？可以参考Jay Scott的“博弈中的机器学习”这个网站。他列举了两个实验方法，我认为比较有趣：

(1) John Stanback(著名的商业国际象棋程序设计师)尝试用遗传算法来设置他的程序Zarkov中评价函数的权重组合。他只测试了2000-3000局，我认为太少，得到的值还不错，但是仍然比手工调整的差。这个例子可以看出遗传算法确实有效，但是需要遗传很多代，或者有一个好的权重组合作为祖先。

(2) Risto Miikkulainen，是得克萨斯大学里遗传算法的研究员，他曾经针对黑白棋的实验做了个报告。他用遗传算法来调整神经网络型评价函数中的权重。评价网络通过对阵固定程序来建立，如果这个固定程序下棋是随机的，那么神经网络会以棋子-格子数组的形式掌握评价(棋子在角上是好的，在角的邻近格子上是坏的，等等)，直到它一直能赢了才停止学习。然后对阵一个浅的搜索结合棋子-格子数组的程序，它最终(通过几个星期的计算)学会了基于机动性的策略。但是如果对手已经是很聪明的基于激动性的程序，那么它始终会输并且不会学习。这个例子可以看出，要进行学习就必须对阵水平相当的程序，例如在遗传算法的同一个种群里，让两个不同的评价的程序进行对阵，也可以让某个程序对阵不同棋力的对手。

原文：http://www.ics.uci.edu/~eppstein/180a/970415.html
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译

- 上一篇 局面评估函数——简介(一)
- 下一篇 其他策略——胜利局面
- 返　回 象棋百科全书——电脑象棋

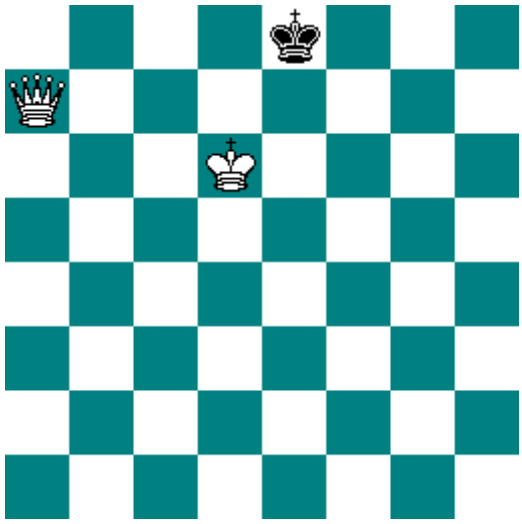
《对弈程序基本技术》专题

胜利局面中的强制过程

David Eppstein */文
* 加州爱尔文大学(UC Irvine)信息与计算机科学系

如果棋局到达一个能用强制着法获胜的局面，那么Alpha-Beta搜索会找到它。但是奇怪的是，每次都走一步能赢棋，不是总能赢下来的。这个问题出现在西洋棋或国际象棋中，可以走一步棋到达强制获胜的局面，但是无法使胜利更近一步。

例如，考虑下面的国际象棋局面：



白先走可以立即获胜：把后走到e7格将死黑王。但是白也可以走其他着法只是赢起来慢些，实际上白方只有一种着法不能取得胜利。例如，假设白把王走到e6，黑只能走d8和f8，两者都会被白将死。如果黑走d8，那么白王走回d6仍然可以赢。但是在走了“1. Ke6 Kd8 2. Kd6 Ke8”之后，我们回到了一开始！白走了获胜的着法，但是他没有在获胜上取得进展。

如果Alpha-Beta搜索给任何获胜的局面以相同的评价，就很容易掉进这个陷阱。要防止这种现象，我们需要改变对胜利局面的评价，让着数少的胜法比推延获胜稍微好些。代码很直观：如果我们保留一个层数变量，代表搜索的当前局面距离根局面有多远，我们可以通过减去层数来调整胜利局面的值。以下伪代码用常数 WIN 代表棋类分值的最大值(在国际象棋中，典型的 WIN 可以是兵的价值的100或1000倍)。

```
// 根据层数来调整胜利值的Alpha-Beta搜索
int ply; // 全局变量，在开始搜索时设为零
int alphabeta(int depth, int alpha, int beta) {
  if (棋局结束 && 当前棋手获胜) {
    return WIN - ply;
  } else if (棋局结束 && 当前棋手失利) {
    return -WIN + ply;
  } else if (depth <= 0) {
    return eval();
  }
  ply ++;
  for (每个可能的着法 m) {
    执行着法 m;
```

```
        alpha = max(alpha, alphabeta(depth 1, beta, alpha));
        撤消着法 m;
        if (alpha >= beta) {
          break;
        }
      }
      ply --;
      return alpha;
    }
```

现在再来看上面的例子，ply = 1 时立即将死，得到的分值是999(WIN – 1)，而把王走到e6可以在 ply = 3 时获胜，分值是997。程序会使局面到达最大的分值，因此选择立即将死。

对于像黑白棋一样的棋类，棋局的长度有个适当的限制，每着棋都会在棋盘上增加一个子，因此棋局结束前最多只有64着棋。对于这些棋类，没有可能使局面产生无限循环，因此我们可以只用 WIN 或 –WIN 而不必考虑层数的调整。

这个层数调整的技巧有一个更为复杂的情况：如何跟散列表发生作用？问题是在散列表中存储的局面，其层数可能跟我们搜索到的局面有所不同。为了得到正确的层数调整值，我们应该在散列表中存储相对于当前局的分值，而不是相对于根局面的。

这样，把局面存储到散列表中，就用下面的伪代码，这里 MAX_PLY 是比搜索中可能的最大深度还大的常数(WIN = 1000 的话，可以让 MAX_PLY = 100)。变量*x*只是当前局面在散列表中的指标。

```
if (score > WIN - MAX_PLY) {
  hash[x].score = score + ply;
} else if (score < -WIN + MAX_PLY) {
  hash[x].score = score - ply;
} else {
  hash[x].score = score;
}
```

从散列表中获取局面时，需要做相反的调整：

```
if (hash[x].score > WIN - MAX_PLY) {
  score = hash[x].score - ply;
} else if (hash[x].score <-WIN + MAX_PLY) {
  score = hash[x].score + ply;
} else {
  score = hash[x].score;
}
```

原文：http://www.ics.uci.edu/~eppstein/180a/990202a.html
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译

- 上一篇 局面评估函数——简介(二)
- 下一篇 其他策略——主要变例的获取
- 返　回 象棋百科全书——电脑象棋

www.xqbase.com

《对弈程序基本技术》专题

获取主要变例

Bruce Moreland / 文

要点

经常有人问，如何在搜索完成后提取主要变例。主要变例是程序认为的对双方来说都是最好的着法线路。它不会由未修改的“Alpha-Beta函数”来获得，所有的Alpha-Beta都只返回数值。

我们需要做的是对普通的Alpha-Beta搜索作修改，使得它能获取主要变例。修改的部分用醒目的颜色标出：

```
typedef struct tagLINE {
   int cmove;                  // 路线中着法的数量；
   MOVE argmove[moveMAX]; // 路线。
} LINE;

int AlphaBeta(int depth, int alpha, int beta, LINE *pline) {
   LINE line;
   if (depth == 0) {
      pline->cmove = 0;
      return Evaluate();
   }
   GenerateLegalMoves();
   while (MovesLeft()) {
      MakeNextMove();
      val = -AlphaBeta(depth - 1, -beta, -alpha, &line);
      UnmakeMove();
      if (val >= beta) {
         return beta;
      }
      if (val > alpha) {
         alpha = val;
         pline->argmove[0] = ThisMove();
         memcpy(pline->argmove + 1, line.argmove, line.cmove * sizeof(MOVE));
         pline->cmove = line.cmove + 1;
      }
   }
   return alpha;
}
```

这个函数或许效率很低，因为它用到很多的堆栈空间，但是代码工作起来并不慢。有了这些改动后，如果函数返回Alpha和Beta之间的值，那么它不仅仅返回一个值，还包括能产生这个值的路线(一系列预测的着法)。这对于搜索树的任何结点都是有效的，包括根结点(它是最值得这么做的)。你可以这么来调用Alpha-Beta：

```
val = AlphaBeta(depth, -INFINITY, INFINITY, &line);
```

你得到的是局面的值，以及在“line”这个存储区里保存的预测路线。“line”的数据结构是一个着法数组，以构成一个路线，另有一个整数来告诉你路线中有多少着法。

如果你用深度等于零去调用Alpha-Beta，那么这个函数只调用“Evaluate()”并返回它的值。一个着法也没搜索，因此一个着法也没选择，从而和分值一起返回的路线其长度为零。

如果你用某个深度调用这个搜索函数，那么你可以找到一个着法其值在Alpha和Beta之间，而你得到的不仅仅是分值，而且包括能产生这个值的一系列着法。为了在Alpha-Beta过程中建立路线，你拿出这个搜索到的着法，把它存入传递的路线存储区中【译注：即函数中的“*pline”】，并把局部的路线存储区【即函数中的“line”】也加到传递的存储区中。

你可能会问：“既然有传过来的路线存储区，为什么又要在每次递归的堆栈中新分配一个？”因为你在搜索树中找到一个局部的线路，那么原来的线路被推翻了，但你不能毁坏已经建立好的线路。如果你找到某个返回值在Alpha和Beta之间的结点，你就认为这个结点在搜索树的根结点的路线上，这是不对的。有很多零碎的主要变例是被丢弃的。

让你们感到惊讶的是，我在循环内用了“memcpy”这个函数【事实上这也是个循环，因此可能会认为它的效率很低】，它几乎不花时间，因为它很少被执行。

另一个思想

另一个一目了然的方法，就是在搜索值返回后，从主置换表中提取主要变例。置换表项中有一个区域存放这局面的最佳着法。由于每个PV结点都有一个值在Alpha和Beta之间，散列项中必定保存着“最佳着法”。

从置换表中提取主要变例，可以沿着散列项组成的链来提取，这就像吃馅饼一样简单。

我知道很多程序(包括一些专业程序)是这样做的，但是我没有试过。

【情况可能会比想象中的复杂，因为置换表中的数据会被覆盖，这样做就必须选择合适的覆盖策略。显然根结点是不会被覆盖的(它总是最后一个写入置换表)，因此至少能提取出一个着法(即程序要走的那步棋)，但是后续着法就很难保证了。深度优先的覆盖策略会比较有利，除此之外，也可以考虑PV结点优先的策略，因为PV结点的数量比Alpha结点和Beta结点少得多，所以这个覆盖策略对置换表不会产生很大的影响。

另外，直接从Alpha-Beta函数提取的主要变例，会因为置换表的运用而中断，除非置换表里有一项用来存储主要变例(这不是不可能的，因为PV结点的数量非常少)。如果要得到完整的主要变例，可以考虑不在置换表中写入PV结点(某些程序甚至只在置换表中写入Beta结点)。】

原文：http://www.seanet.com/~brucemo/topics/pv.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 其他策略——胜利局面
- 下一篇 其他策略——重复检测
- 返 回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

重复检测

Bruce Moreland / 文

重复检测为什么那么重要

我们有必要提一下重复和局的问题。如果棋局面(同一方走的情况下)重复三次，就可以宣布和棋。如果程序领先一个车但是它陷入长将，那将是非常糟糕的，对手会在你即将取得胜利的时候宣布和棋。

要解决这个问题，就必须检测以前出现过的局面，并采取对策。如果程序懂得重复，它就可以根据盘面上局势的需要，来谋求重复或避免重复。如果程序即将输棋，那么它应该试图寻找长将，反之应该避免。

【译注：中国象棋出现重复局面时，要根据双方的着法来判断胜负(例如单方面长将一方要被判负)，规则非常复杂，因此策略也会不同。】

一个相当麻烦的选择

有两种局面可能会重复：棋局的历史局面，即在棋局中盘面上走过的局面；搜索树局面，即程序搜索的路线上出现的局面，它们没有在盘面上出现过，但是程序的思考中会不断地撤消和执行着法。

很明显，搜索树中的重复局面应该能被立即检测出，并且会得到处理。一般来说会返回一个和局分值，但是我听说有些程序会在中局遇到长将时，如果程序一方在将军，就故意返回一个正值。这是为了说明，如果你能找到长将，那么局面通常会好些。如果搜索树中的重复没有被处理，那么程序就不会看到长将或其他重复和局的情况。

对于搜索树与棋局中局面出现的重复局面，该怎么做就不清楚了。如果某个局面在棋局中出现两次，在搜索中出现一次，那么应该当作和局处理，因为棋局中再出现一次就会和了。困难在于某个局面只在棋局中出现一次，在搜索树中也出现一次。

很多程序把这些局面当作和局处理。这可以使得程序在陷入困境或对手试图制造重复局面时，能有效地通过重复检测来确定是否达到和局，缺点是程序有时会走出一些不正常的着法。【如果程序只检测到两次重复局面(即棋局中的一次和搜索树中的一次，或者两次都在搜索树中)就返回和局分值，那么对搜索效率是有所提高的，因为程序节省了第二次到第三次重复之间的线路，这样就至少在搜索树的局部分枝上减少了4层。】

我看过一些比赛的例子，其中一盘是GNUChess对阵我的程序。有个能赢的王兵残局被GNUChess【WinBoard自带的程序，源代码是公开的】错过了，这两个程序就进入一系列和局局面。最后，他们又走回那个被GNUChess错过的能赢的局面。我的程序很高兴看到这个重复局面，因为它是作为和局来评分的(它已经出现在棋局的历史局面中)。当然，第二次GNUChess找到了获胜的途径。【第二次重复不会被判和棋(尽管原作者的程序认为已经和了)，要到第三次重复才判和棋。】

还有一盘棋是我的程序对阵一位叫Greg Kennedy的人类大师。Kennedy领先两个兵，但是他走了一步臭棋可以导致对手一马踏双，并能让对手得回一个兵。Kennedy看到他的王被将军了，必须走他的王，他就把王走到原来待过的地方。然而我的程序走回了上一步，让局面回到两步前的样子，而没有通过吃兵来达到仅落后一个兵的局面。重复问题使得程序期待Kennedy会把王走回来，并且让程序再对他一马踏双【这样他的王就第三次回到原来的位置上了】。当然他不会这么做，这样就让他继续领先两个兵了。

其他假设也是有可能的。一个很强的人类棋手发动了压倒性的进攻，但是后来没走好而让程序逃掉了王，因此人类棋手就只有长将了，因为他子力落后并没有杀棋。程序会乐于把王走回逃跑前

原来的位置，因为这些位置是重复的并且会得到和局。【被长将会导致和局，把王走回原来危险的位置也被程序认为是和局，如果程序选择了后者，就给了对手第二次尝试杀棋的机会。】

我已经在我的程序里做了修改，忽略棋局中出现一次并且搜索树中出现一次的重复局面。这样就解决了上面提到的问题，但是带来了新的问题。

程序会把一个局面重复几次，这使得棋局有时非常烦琐。这可能会干扰人类对手，或者在电脑国际象棋比赛中干扰对手，因为棋局到达复杂残局时，可能不会有进展，并且可能周旋很长时间，从而离50回合限制越来越近。【人类棋手在棋局没有进展时，第二次出现重复局面就会握手言和，而采用这种策略的程序则不肯提和，这会让他的对手感到不舒服。】

选择哪种做法，是需要斟酌的。【从效率上说，第二次重复就返回和局分值的做法比较好，然而这种做法给了对手一个机会，当对手第一次犯错误时，第二次就有机会纠正了。】

可能的实现

有很多方法可以检测重复。其中一个在Tom Kerrigan的TSCP中使用，他把这个方法归功于John Stanback。在他的代码中他声明了这一点，但是没有任何详细的信息来告诉我们这是如何做的，因此我也不知道。如果你想知道它，就不得不到TSCP的源程序中挖掘。

我能知道的实现方法已经在“Zobrist键值”一文中提到。如果你要实现“置换散列表”，那么你应该先实现Zobrist键值，这才能让置换表得以实现。你需要对Zobrist键值作处理，从搜索树的当前局面往回找，看有没有和当前局面相等的键值。

一个实现方法就是根据当前路线建立一个先进后出的堆栈，把键值加到历史局面中。为了检测重复，就要一层层地读取堆栈，比较其中的键值，以确认当前键值是否已经存在于堆栈中。

没有必要找遍整个列表，只要往回找，直到遇到进兵或吃子的着法，因为这些着法在棋局中是不可逆的。你不可能在最后一次吃子或进兵的前面找到重复局面。

这样做看上去有点花时间，恐怕是吧，但是我相信有些程序就是这么做的。

在我写国际象棋程序的早期，我在Usenet上问了个关于散列技术的问题，得到Belle(尤物)作者Ken Thompson的回答。【贝尔实验室的Ken Thompson，可能是影响力仅次于John Von Nouma(冯-诺依曼)的计算机科学家了，他是C语言的前身B语言的发明者，Unix系统的创立者之一，有关他在电脑国际象棋上作出的贡献，可参阅译文《电脑国际象棋简史》。】他告诉我他用置换表来检测重复局面，他是这样做的：

当探测置换表时，在表项中设置“打开”标志。这个标志一直被设置着，直到不再搜索这个局面为止，即从局面返回时才关闭。任何时刻打开的结点不是历史局面就是在搜索树的当前路线上，因此如果探索散列表时遇到一个打开的结点，就一定是前面发生过的局面。

这种方法具有数据结构上的优势，因为这样的数据结构在典型的国际象棋中都用到，但是这个思想有一些问题。当进入一个结点时，必须写入散列项，因此需要使用“始终替换”的策略。对于Thompson来说这不是问题，因为他的策略包含了“始终替换”的散列表，但是其他替换策略就无法使用这种方法。另一个问题出现在散列表项冲突的情况下，这个问题必须得到处理。当两个局面具有相同的64位键值时，我不讨论散列键值的冲突问题。现在我讨论过两个局面共用一个散列项时，应该保留哪一个。如果两个打开的结点要占用同一个散列项，除了要检测第二个局面是否重复以外，要做什么并不清楚。可能这个问题要通过重新散列的策略来解决，但是这个方法跟散列表的主要用途没有关系，所以要加这个功能比较麻烦【重新散列(Re-Hashing)是一个解决散列冲突的常用方法，但是在国际象棋程序中并不常用】。最后一个问题就是如何适应多处理器搜索，因为很多线程可能会读取一个散列表。遇到打开的结点时，可能根本就不是重复局面，因为它可能属于另一个处理器的搜索路线。这个问题解决起来看上去很复杂。【译者认为，可以在散列项中记录一个被打开的处理器号，每个处理器只对自己打开的结点作重复检测。】

另一个简单的策略就是用一个很小的散列表【如果考虑50回合限着(即100个着法)，那么100到200个散列项就足够了】。进入结点时探测散列表，如果当前局面的Zobrist键值已经在散列表里，就返回平局分值。否则就加入这个键值。当结点退出时，键值就删除。这看上去很直观，并且散列项的冲突处理起来很容易，因为散列表不是满的，散列项以先进后出的顺序存取，也不存在替换策略的问题。

我不愿说这个方法比其他方法好，因为毕竟有Ken Thompson的方法。我的这个方法有一些问题，因为它需要靠额外的数据结构和额外的函数的支持。

当程序改成多处理器搜索时，这种方法有点令人担忧，但是比起Thompson的策略在这方面遇到的问题，我的这个问题看上去不那么严重。

如果你们想看这个第二散列表的策略，就去检查Gerbil的源程序。这里我不准备列出源代码的示例，这只是实现上的问题罢了。

【中国象棋程序也可以通过探测散列表进行重复检测，但是不能立即返回平局分值。当检测出重复局面时，必须逐个分析两次重复局面之间的着法，根据着法的性质来判定胜负，这一点比国际象棋麻烦得多。】

原文：http://www.seanet.com/~brucemo/topics/repetition.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 其他策略——主要变例的获取
- 下一篇 其他策略——藐视因子
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

藐视因子

Bruce Moreland / 文

为杀棋定义一个值很容易——只需要一个很大的正数，如果你被将死了就用一个很大的负数。但是决定平局用什么值就稍难一些了。

如果程序对阵比它强的棋手，那么一个平局可能对程序来说非常不错，但是当程序对阵一个很弱的棋手时，平局却很失败。

“藐视因子”(Contempt Factor)只是平局分值的另一个名称而已，这就意味着平局分值必须根据对平局的渴望程度来调整。

典型的藐视因子是一个稍负一点的数，因此程序即使在局面稍差的情况下也会试图取胜，但是一些程序根本不用藐视因子，并且在平局时只返回零。

我认为有必要根据比赛进行的状况选择不同的藐视分值。如果是很激烈的中局并且程序稍微落后，通常发生在程序执黑时，可以考虑给个稍大些的值，这种条件下程序就可以为胜利而战。

在残局中即使有子力，用零或许就是好的。

我认为还有个诀窍，藐视因子在某些情况下应该取负数，这取决于哪一方走棋。否则的话从Alpha-Beta树向上返回时符号会出错。

我认为开局阶段合理的藐视值是–0.50个兵；整体上合理的藐视因子是–0.25；在残局中0.00比较合理，或者用一个不是很负的值。用一个负值来处理兵残局则不是一个好的策略。

【译注：某些棋类程序可由用户定义程序的下棋风格，如保守、均衡或冒进，译者认为这可能和藐视因子有关，风格保守时藐视因子正些，风格冒进时负些(注：藐视因子越负说明藐视程度越大，程序越想赢棋)。另外，参加比赛的程序还应该根据赛程来调整藐视因子，例如最后一轮比赛中程序只要守和就可以夺冠，那么这一轮藐视因子应该设得正些。】

原文：http://www.seanet.com/~brucemo/topics/contempt.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 其他策略——重复检测
- 下一篇 其他策略——后台思考
- 返 回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

后台思考

Bruce Moreland / 文

对手在走棋的时候你会做什么

国际象棋程序能够在规定的时间内思考局面，走出它认为的最佳着法，然后坐在一边等对手下，对手出了着法后，再做同样的事。

对于程序来说这太简单了，但是缺点是浪费了很多可以处理的时间，因为对手在思考时，程序做在一边空闲着。

在一些国际象棋书上，有一些思想说的是你不着棋时该做什么。一个想法就是在对手思考时，你也在考虑对策，关注到你走棋时将出现的路线。这对人类棋手来说是个不错的主意，但是如果要在电脑国际象棋程序上实现，这并不简单。

我用我的程序参加1995年在香港举行的世界电脑国际象棋锦标赛(WCCC)时，自以为合理地解决了这个问题。轮到对手走棋时，我的程序就扮演对手的角色，也坐在那里想。当我的程序又开始走棋时，“散列表”里已经记录了很多很有用的信息，前面几层能够很快地搜索完。

我认为这是一个更好的方法

这个方法用得很好，使得我没有得最后几名，但是那次比赛我学到了一个更好的办法，我想现在所有人都用这个办法了(可能那时候就是如此)。这个技术称为“沉思”(Pondering)或其他诸如德语中翻译过来的“永久智能”(Permanent Brain)。我不懂德语，但是更好的翻译可能应该是“连续思考”。

当你的程序在走棋时，它会产生一个“主要变例”。这个主要变例通常不止一个着法，主要变例中的第一个着法就是下到棋盘上的，而剩下的那些只是显示在屏幕上。

第二个着法就是程序认为的对手会走的着法，它通常是个好的着法，并且在很多情况下对手最终会走这个着法。

这有点像是在赌对手会走这个着法，赌赢了就告诉程序对手确实走了这个着法。

因此对手在思考一个着法的时候，你的程序就猜了一个可能会选择的着法，并且在此基础上思考你的应对。

如果你的程序猜得正确，你会在走下一步着法的时候有一个很好的开端，你的程序从而很可能想得更远，并且很快就能出子。

有一种情况经常出现在两台计算机的比赛中：一个程序在思考着法，然后出子，另一个程序就马上回应。第一个程序又开始思考并出子，而第二个又马上回应。这种情况会持续很多步，直到第二个程序在自己的时钟上积累了足够的时间，使得它认为没有必要马上出子。

【译注：根据剩下的时间和需要走的步数，国际象棋程序可以决定一步棋允许花多少时间的(国际比赛通常是时段制的，当然加时制也可以作相应的处理)。当立即出子时，剩下的时间没有减少，而需要走的步数减少了，因此一步棋允许花的时间就多了，这就相当于积累了时间。

某些程序可以估计它多搜索一层所需要的时间，如果它没有积累到足够的时间来多搜索一层，那么在后台思考中已经把前面的搜索做完了，它就会马上出子。如果它积累到了足够的时间，那么它通常会选择多搜索一层，让棋下得更好。】

如果对手没有走出预测的着法，你的程序只要简单地撤消这个预测着法，执行对手走的那个，然后开始为自己思考着法。

另一个思想

在1995年的WCCC上，我的程序有机会挑战Hitech。我的程序下出了要输的局面并且开始长时间思考。在我的程序思考走哪步时，Hitech对我将要走的那步想了足够长的时间，并且开始考虑另一种着法了。如果我的程序可能走出的着法Hitech都能想到，那么它当然可以迅速出子。

我不知道这样做是好是坏，但这是很值得考虑的。

【为对手预测着法来进行后台思考，是目前普遍的做法。在UCI协议(国际象棋通用引擎协议)中，引擎甚至没有必要自己设计这样的策略。UCI协议只要求引擎在给出着法的同时给出后台思考的预测着法(即主要变例中的第二个着法)，例如：

bestmove e2e4 ponder e7e5

如果界面的后台思考功能打开，界面就会根据预测着法把局面告诉引擎，让引擎进行后台思考，例如：

position startpos moves e2e4 e7e5
go ponder <time_settings>

此时引擎就在对手着棋的时间里思考，而自己的时钟是关闭的。当对手出子后，如果预测命中，界面就会向引擎发出后台思考命中的指令：

ponderhit

让引擎开启自己的时钟，继续思考，直到它想出着法为止，例如：

bestmove g1f3 ponder b8f6

如果预测没有命中，那么界面就必须中止引擎的思考：

stop
bestmove g1f3 ponder b8f6

中止思考后，引擎同样会返回一个着法，但界面不会理会。随后界面把对手的着法告诉引擎，并让引擎重新思考，例如：

position startpos moves e2e4 c7c5
go <time_settings>

在UCI协议中，原作者提到Hitech的这种后台思考策略(即预测着法想了足够长的时间，开始想第二个预测着法了)是无法实现的。】

后台思考对棋力的增强

我没有考虑过这个问题，但是它会使搜索进行得更深，或者时间花得更短，所以肯定会增强棋力的。

原文：http://www.seanet.com/~brucemo/topics/pondering.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 其他策略——藐视因子
- 下一篇 其他策略——残局库
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

残局库

Martin Fierz */ 文
* 瑞士Windisch应用科学学院(Aargau学院)

简介

残局库在很多棋类游戏中扮演着非常重要的角色，例如九人Morris，Awari和西洋跳棋(Checkers)(其中九人Morris已经完全可解了，这要归功于残局库的使用)。在其他棋类中，残局库就不那么重要了(例如国际象棋)，而某些棋类制作残局库是不现实的(如连四子棋、黑白棋和围棋)。你是否已经发现这些棋类的差异了？通常只有在盘面上棋子数量很少的情况下，残局库才能实现。适用残局库有个确切的棋子数量的界限，它取决于棋类的复杂性。有些棋类随着对局的进行，棋子数量是减少的，那么残局库就是可行的；而有些棋类的棋子数量是增加的或者不变的，那么残局库就是无法计算的(除非棋类异常简单)，无论如何残局库取决于具体的棋类。例如在国际象棋中，有多达6子的残局库(王车兵对王车兵等)，而这种残局(包括其他不超过6子的残局)在实战中不是经常出现的，因此残局库对棋力的影响并不是很大。而在(盎格鲁-萨克森式的)西洋跳棋里，有多达8子的残局库，而10子的残局库也已经在计算了，这就意味着在有吃必吃的规则下，很多棋局会很快走到有残局库的局面中，因此残局库使得西洋跳棋的棋力有了很大的提高。要让某种棋类完全可解，通常总是要借助于残局库的——从起始局面开始向前搜索，结合残局库，就能解出这盘棋。Gasser用这种办法解决了九人Morris，而Chinook(最著名的西洋跳棋程序)的小组正在用这个方法解西洋棋。

残局库的不同类型

残局库有不同的类型，而它们都可以知道残局库中某个特定的局面是赢棋、是输棋还是和棋。如果这就是残局库包含的所有信息，就称为“胜负和”(WLD)残局库；如果知道多少步以后棋局会结束，就称为“杀棋步数”(DTM，Distance-to-Mate)残局库；如果只知道多少步以后会转换为另一种类型的局面，就称为“转换步数”(DTC，Distance-to-Conversion)残局库。WLD残局库有个问题，即便程序处于胜利局面，也未必能赢下棋局。尽管残局库知道某个局面已经是胜利局面，并且知道走哪步能继续保持胜利局面，但是保持胜利局面的着法可能会距离杀棋更远，而程序又不知道该选择哪步保持胜利局面的着法。【译注：更具体的说明请参阅《胜利局面中的强制过程》一文。】DTM残局库显然在这个方面做得比较好，因为你只要选择DTM(转换步数)最少的那个保持胜利局面的着法。DTC残局库也能够在胜利局面下走出取得胜利的着法，但程序走的步数可能会比必要的步数多。至今还有人使用WLD残局库，你可能很怀疑。原因很简单，储存胜负和的信息只需要很小的空间，如果残局库比计算机的存储器大得多(通常如此)，那么残局库有很多部分可以放入存储器。从磁盘上读取残局库不是一个好的选择，因为磁盘跟存储器比起来慢得多。

残局库的生成

残局库的生成是一个相对比较简单的过程，尽管有很多细节，但这不影响生成过程的理解。残局库生成的基本算法称为“后退式分析”(Retrograde Analysis)，它是由Ken Thompson发明的(至少是首先使用的)。以下就是整个过程：

以我们要生成“王车对王”的残局为例，你要从“索引函数”(Index Function)开始，每个可能的王车对王局面都有一个数字。索引函数必须能倒过来映射到以数字x为代表的局面。理想情况下，索引函数会把所有N个合理的王车对王局面映射为0, 1, ..., $N-1$。如果是这种情况，我们称之为“无间隙”(Gapless)的索引。一般情况下，索引函数把所有合理局面编号为0, 1, ..., $M-1$，而$M > N$，我们

称其为“有间隙”(Gapped)的索引。有间隙的索引往往更简单，因为构造一个有间隙的索引函数要比无间隙的索引函数简单得多。后退式分析需要的存储器跟索引号的最大值成正比，因此如果构造了一个*M*比*N*大得多的索引函数，那么你会浪费很多存储器。

一旦有了索引函数，后退式分析算法就只要做以下几件事：

(1) 初始化：生成两个长度都为N的数组，分别存放结果(WIN/LOSS/DRAW，代表胜负和)和DTC。把所有的结果都设成UNKNOWN(代表未知)，所有的DTC计数器都设成0。你会发现，你需要4个数来表示结果，因此数组的数据类型是4个值的数(即2位)。当然，还有些根本不存在的局面，你需要自行处理。

(2) 杀棋遍历：逐一检查每个局面是否是杀棋局面，如果是的，就把这个局面的结果设成LOSS，表示即将走的一方输了。如果棋类允许“逼和”的存在，也必须作逼和的检查，并赋值为DRAW。把每个UNKNOWN局面的DTC计数器加1。

(3) 对每个UNKNOWN的局面，生成所有后续局面并看它们都有哪些结果。只要其中有一个后续局面是LOSS，就把这个局面设成WIN。如果所有后续局面都是WIN，就把这个局面设成LOSS。如果你遍历了所有局面但没有一个局面能设WIN或LOSS的，就跳到第5步。

(4) 把每个UNKNOWN局面的DTC计数器加1，并回到第3步。

(5) 把每个UNKNOWN局面设成DRAW，算法就结束了。

这个算法显然是确保完成的。在第3步里当吃子发生时，你就要读取少一个子的残局库。很明显，没有王车对王的残局库，你无法独立地生成王车对王马的残局库。如果你只要生成WLD残局库，就可以不要DTC计数器。如果你需要生成DTM残局库，你就需要在局面转换时传递DTM值。以上算法有很多优化方案，但是我不想展开讨论，最基本的算法已经非常简单明了了，为什么再舍弃它呢?

明白了整个算法后，你就知道为什么叫做“后退式”了——该算法总是从已知局面(杀棋或能转换为低级别的残局库局面)向后找，按照上述算法的第3和第4步，每次遍历就后退半步。我们拿一个局面举例说明，白王在g3，黑王在h1，白车在a2，黑先走【8/8/8/8/8/6K1/R7/7k b - - 0 1】。在遍历杀棋局面时，按照上述算法找到白王在g3，白车在a1，黑王在g1，黑先走【8/8/8/8/8/6K1/8/R5k1 b - - 0 1】，这个局面是杀棋，所以把它设为LOSS。在我们要讨论的这个局面中，根本不能检查到什么。在主循环的第一次遍历中，会为“白王在g3，黑王在g1，白车在a2”【8/8/8/8/8/6K1/R7/6k1 w - - 0 1】产生所有着法，发现走了Rb2-b1后就是LOSS局面，根据规则这个局面就设为WIN。下一步，为黑王在h1的局面【8/8/8/8/8/6K1/R7/7k b - - 0 1】找后续局面，发现所有的后续局面都是WIN局面(这个局面的后续局面只有一个)。最后把这个局面设成LOSS，就得到了正确的结果。

索引函数

索引函数对这个算法非常重要，你无法存储整个局面并对他们设定结果，因为结果只需要2位，而整个局面需要用大量存储器来描述。如果你存储整个局面，你就会浪费大量的存储器。用了索引函数以后，你就能够简单地用一个数字来代表局面了，你不需要把结果和索引数字都记下来，而只需要以索引数字为数组的指标，只在数组中存储结果。那么如何才能找到符合上述性质的索引函数呢？看上去这是个很吓人的工作，实际上用简单的方法来构造索引函数是可行的。对于棋子各不相同的残局(例如白王、白车和黑王)，就非常简单，把格子标号为0到63，就可以写下这样的公式：

```
index = wK_index + 64 * wR_index + 64 * 64 * bK_index;
```

这个函数完成了局面到数字的转换，并且它是可逆的(wK_index = index % 64, wR_index = (index / 64) % 64，等等)，但是它会产生一些不合理局面(例如几个子在同一个格子上，或两个王紧挨着)。这个函数也没有利用棋盘的对称性。这些细节问题是完全可以解决的，但是在这里我不想多说了。那么如果棋盘上有多于一个的相同棋子，例如王双车对王，怎么办呢？我们照样写：

```
index = wK_index + 64 * wR1_index + 64 * 64 * wR2_index + 64 * 64 * 64 * bK_index;
```

这样当然很管用，但是非常愚笨，因为1号车在x格而2号车在y格，情况跟2号车在x格而1号车在y格是一样的。这样我们的索引就比必要的数多了一倍！为了解决这个问题，我们用组合公式来表示2个相同的子在64个位置上的情况：在数学课上你肯定学过用$N = 64 \times 63 / 2$来做。因此我们可以写成：

```
index = wK_index + 64 * combinedindex + 64 * N * bK_index;
```

剩下来的问题就是由车的具体位置来计算“组合的车的索引”了，它是0到$64 \times 63 / 2 - 1$之间的数。用r_1和r_2表示两个车的位置，并且$r_1 < r_2$(这样就等同于除以2了!)。我们有：

```
combinedindex = bicoef(r1, 1) + bicoef(r2, 2);
```

这里bicoef(x, y)代表x和y($x > y$)的二项式系数，即$x! \times y! / (x - y)!$，任意数量的棋子都可以由这个组合索引公式产生。它的逆公式有点复杂，如果是k个子在n个格子上的组合索引，我们就必须用顺序搜索的办法来分析它的组成：因为组合索引的最后一项总是最大的，所以我们要依次计算$i = n - 1, n - 2, ...$的bicoef(i, k)，直到比组合索引数小为止。一旦找到了i，我们就知道它在第i格上，然后在组合索引上减去bicoef(i, k)。然后我们依次计算$j = i - 1, i - 2, ...$的bicoef(j, $k - 1$)，因为我们已经在建立索引函数的时候知道$j < i$了。

压缩

当你开始生成残局库时，你一定会马上意识到你要建立的残局库是非常庞大的。例如，8子的西洋跳棋残局库如果没有压缩，就需要大约40GB的磁盘空间。如果你需要在1GB的RAM的计算机上使用这个残局库，你就必须对它进行压缩。压缩这类残局库的标准方法是“行程编码”(RLE，Run-Length Encoding)：当你在查找后退式分析所产生的数组时，它看上去会是这样的：

```
....WWWBWWLLDBDBDDDWLBLLLWWBDDD...
```

这里WLDB代表胜负和坏，坏的意思是局面不合理，使用有间隙的索引，或者不走棋的一方被将军了，这种情况就会发生。要对此进行压缩，我们首先注意到对坏的标志可以任意处理，因为它们没有用，因此我们将它们设定为最方便压缩的值：

```
....WWWWWWLLDDDDDDDWLLLLLWWDDDD...
```

行程编码存储的就是一对对数值和长度，上面的例子就转换为：

```
(W,6) (L,2) (D,7) (W,1) (L,5) (W,2) (D,4)
```

如果行程非常长(我没有耐心来造一个行程非常长的例子)，那么这种压缩的效果就非常好。8子的西洋跳棋残局库可以压缩到大约4GB，压缩因子是10。

在搜索中读取数据库

压缩完残局库以后，你必须写一个飞跃式(on-the-fly)的解压缩程序，根据局面找到结果。或许这还不够，如果残局库大到超过你的RAM，你就必须为自己的残局库写一个存储器管理程序。你不会指望Windows(或其他操作系统)来帮你管理残局库，因为你自己写的程序是高效的。管理残局库的通

常做法是把压缩的残局库分成一个个几千字节的块(Chunks)，如果你需要知道某个局面的结果，就一次读取整个块。从磁盘读取1字节或1千字节是没有差别的，速度只取决于磁盘的寻找时间，而跟传输速度无关。一次读取整个块，就把很多相似的局面装入存储器，这些局面可能是你以后要用到的。一般来说，你会用“最近最少用到”(Least-Recently-Used)的策略，来决定在装入一个块的时候哪个块应该被覆盖掉。即便你用了块，由于磁盘比起存储器来说实在太慢，你无法对所有局面都去查找残局库。通常你只会在第*x*层以外去读取磁盘上的残局库局面，而在*x*层以内你只会在存储器中查找这些局面。

原文：http://www.fierz.ch/strategy3.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 其他策略——后台思考
- 下一篇 其他策略——开局库
- 返 回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

开局库

Martin Fierz */ 文
* 瑞士Windisch应用科学学院(Aargau学院)

不管你是否喜欢，开局库在很多棋类中都是非常重要的。很多棋类已经被人类积累了大量知识(例如国际象棋、西洋跳棋等)，而有些棋类是新诞生的(或是因为不普及)因而没什么发展。对于人类非常精通的棋类(例如国际象棋)，建立开局库有个非常简单的办法，即手工产生，选择高手走的棋。这种方法不是我们感兴趣的，而对于一些没有理论可寻的，或者电脑足以战胜人类(例如西洋跳棋)的棋类，用这个方法是不可取的。我们将会关注一种没有人工干预的生成开局库的方法。

从数据库自动生成

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。我的国际象棋程序的开局库就是这样生成的，很多国际象棋程序也都使用这种方法。对于国际象棋来说，有高质量的数据库，其中包括很多高等级分棋手下的，因此你可以决定仅仅使用高等级分棋手的对局。对于每个局面，可以生成所有着法的统计数字，例如这步着法的分数，下这步着法的棋手平均等级分，这步着法得到的平均结果，以及用结果衡量和用棋手等级分衡量的不同之处。有的对局会以符号“!”(好着)或“!!”(妙着)作评论，把这种评论考虑进去也是可行的，但我不知道是否有人这么做。我的国际象棋程序的开局库只参照统计数字，采用棋手等级分最高的着法。

由“脱离棋谱的扩展”自动生成

如果没有人类积累的棋类理论，那么你就不得不自己计算开局库了。任何计算开局库的方法，都需要用称为“博弈树”的图来表达，图上的每个结点代表一个局面。对于每个局面，你需要计算所有可能着法的值，因此每个候选着法都要扩展。跟这个图表相应，你还必须找到一个方法来决定哪些结点是需要展开的，也就是说，图表上的哪个叶子结点需要再添加结点。这里我将介绍一个称为“脱离棋谱的扩展”(DOE，Drop-out Expansion)的方法，它是由Thomas Lincke发明的。DOE的基本思想就是产生一个开局库，使得比赛时可以尽可能迟地脱离棋谱。建立开局库时有个策略，每个局面的一些好的着法需要扩展，然后从扩展出的局面继续，这样总是会在当前结点处扩展出一些着法(例如固定数量的着法，或者和跟最好着法的评价差距在一定程度内的这些着法)。但是这个策略在遇到不恰当的局面时往往很糟糕，采用这种开局库的程序通常只会走开局库中最好的着法，而对手会犯错误(要注意对手所谓的错误可能也是好的着法，仅仅是对于你的开局库而言是错误的)。因此，开局库生成器只能对一方生成最好的一些着法，而对方应该能走好的或不好的着法。首先，DOE生成器会需要产生一个图表，它对每个结点的后续着法都有一个评价。然后需要一个优先函数，对每个可能的路径(在DOE限制下，一方只走最好的着法)都可以给出一个优先值，对优先级高的路径的叶子结点作扩展。这个策略使得它要做很烦琐的并行计算，如果你有计算机集群，那么你可以在主机上运行DOE生成器，而众多子机上对每个局面做精确的计算。路径优先级的计算相对来说比较随意，你可以用类似下面的方法来做：

```
delta = sum_of_errors(path);
depth = length(path);
priority = -delta - CONST * depth;
```

这个优先函数会使对手的好的着法搜索得深一些，对手犯错误的着法搜索得浅一些。你可以在优先级的计算中加入其他因素，例如衡量结点重要性的因素。Thomas Lincke在他的优先函数中使用过“策略数”(Conspiracy Number)，我的程序则衡量结点的单一性(Singularity)，也就是说，最好和次好的着法差距很大，这条路线就搜索得深一些。

DOE的现状

DOE是一个比较新的技术，在西洋跳棋(我和Ed Gilbert的程序)和Awari(Thomas Lincke的程序)中用得比较成功。我的西洋跳棋的DOE生成器已经在一台单机上运行了近2年了，生成的开局库基本上不再有差错了(在一台很快的个人电脑上计算了1年半以后)，或许还能够纠正人们对这种棋类在开局上的错误认识。我之所以说“基本上”，是因为目前我知道的着法中没有错误，但是也有可能开局库的成千上万个着法里会隐藏着几个坏的着法。半年前我和Ed Gilbert让我们的引擎比了624盘棋，我的程序Cake战绩是3胜1负，其余620盘是和棋，和棋率是那么的高！输的那局揭示了我的开局库里的一个问题，现在已经修补好了。

原文：http://www.fierz.ch/strategy4.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译

- 上一篇 其他策略——残局库
- 下一篇 其他策略——策略和技巧
- 返　回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

策略和技巧

Martin Fierz */ 文
* 瑞士Windisch应用科学学院(Aargau学院)

我通过以下问题的讨论来结束这个棋类游戏程序设计的讲座。这些问题大都没有非常明确的答案，因此我作了比较深入的研究。我觉得这些问题非常有趣，希望你们也这么认为。

速度的需求

棋类程序设计师们都对程序的速度非常关注，有些人会不顾孩子的出生而让程序再提高10%的速度，这是何苦呢？原因是速度提高了几个百分点，棋就会下得不一样，事实上这个不一样的程度会超出你的想象。Ken Thompson对他的国际象棋程序“尤物”(Belle)做了很多试验后发现，搜索每多一层棋力就增加200个ELO等级分，换句话说当你的对手比你低200分的时候，平均有75%的棋局是你赢下的。很多人都在做这个试验，并且得到同样的结果。因此人们就开始预测搜索多少层才能成为世界冠军，并且预测“消亡转折”，即你在搜索到一定深度时，再多搜索一层时棋力不再有原先预计的增长。人们试图找过这个转折，但是找到它要比预期的困难得多。我用自己早期设计的西洋跳棋程序Cake++来寻找这个转折点，指定固定的搜索深度，如5层、7层、9层等等，和少搜索两层的程序比赛。以下就是试验结果，很明显可以看到消亡转折。我用Chinook做了类似的试验来作比较，由于对局数太少，数字不太确切。一些在国际象棋中很难找到的规律，在西洋跳棋里就很容易找到，因为西洋跳棋的搜索深度要多得多。

【译注：消亡转折有着重大的理论和实际意义，它可以用来估计电脑程序能够到达的极限。因为随着计算机速度的不断提高(目前仍旧以每一年半翻一番的速度在提高)，电脑可以计算的深度也越来越深(估计每两年就可多搜索一层)。而随着消亡转折的到来，即使电脑搜索得再深，棋力也不会有太大的长进，这就可能是电脑棋力的极限了。】

搜索深度	胜:和:负(Cake++)	胜率	偏差	胜率(Chinook)
5:3	196:53:33	78.9%	2.1%	-
7:5	153:100:29	72.0%	2.0%	77.5%
9:7	181:75:26	77.5%	2.0%	65.0%
11:9	130:111:41	65.8%	2.1%	72.5%
13:11	134:116:32	68.1%	2.0%	58.8%
15:13	119:136:27	66.3%	1.9%	58.8%
17:15	89:165:28	60.8%	1.8%	61.3%
19:17	78:176:28	58.9%	1.8%	57.5%
21:19	60:189:33	54.8%	1.7%	57.5%

又笨又快还是又好又慢?

跟消亡转折类似的问题是：程序能否通过速度来补偿知识的不足？对于这个问题，Hans Berliner做过一个很有名的试验，用他的国际象棋程序Hitech很好地回答了这个问题。他去掉了程序中大量的局面评价函数，由此产生了另一个程序，命名为Lotech。然后他用Lotech对阵Hitech，并且给Lotech多搜索一层的优势。一个令人惊奇的结果是：Lotech总是能赢Hitech。因此，有人认为一定存在一个

转折点，使得Hitech能打败Lotech。在国际象棋中，这个转折点很难找到。我曾经用自己的西洋跳棋程序来找这个转折点，但是当时我发现Lotech-Cake总是能赢Hitech-Cake。

【如果计算同样深度，Hitech棋力比Lotech高100分的话，但由于多搜索一层使得Lotech增加了200分，所以棋力反而比Hitech强。根据消亡转折来推断的，超过某个深度以后，多搜索一层使得程序的棋力没有太大长进，只要长进低于100分，那么这个时候Hitech就能击败Lotech了。如今电脑思考得越来越深，导致的结果就是：程序设计师们对知识的重视程度增加了。】

机器学习

机器学习是一个很吸引人的主题。计算机程序利用前人经验的方法有很多，其中很多方法被程序作者称为“学习”(Learning)，我将在这里总体介绍一下。最普遍的学习是机械学习，当你的程序输掉一局棋时，它就把这局棋记住了。这个方法可以在开局库的学习中实现，你的程序会把一个开局库着法标记为劣着，如果这个着法会输棋，那么下一盘棋它就选择别的着法。这个方法也可以用一个小的置换表来实现，输掉的棋局里的局面都以输的局面存储起来，这样，程序就不会走到这些局面里去。但是，这种学习的形式实在是太具体了，你的西洋跳棋程序会认为某个具体的局面是坏的，类似典型的局面也是坏的，你的程序却视而不见。在西洋跳棋中“牵制”(Holds)非常重要，例如你的3个兵被对手的2个兵牵制住，如果不考虑棋盘上的其他棋子，那么对手将要获胜(至少获胜的机会很大)，因为它很有可能要升变了。你用最基本的机械学习会知道这种牵制的其中一种局面会输，但是还有成千上万种类似的局面呢，遇到跟这个局面有微小差别的其他局面，再下一盘还是会输。这种类型的学习不是真正意义上的学习，把这些程序称为“智能”(Intelligent)或“吃一堑长一智”(Learning from Mistakes)纯粹是糊弄人的。

另一种类型的机器学习用来自动调整评价函数的权重，这属于遗传算法(Genetic Algorithms)的范畴，把遗传算法作用在评价理论上，就自然地得到了最适合的程序。以下就是这种算法的工作原理，你产生一个评价函数(典型的评价函数是线性的)，每一项都有一个权重，因此有：

```
eval = w_1 * f_1 + w_2 * f_2 + ... + w_n * f_n
```

f_i就是局面的各个因素，例如在西洋跳棋中，你可以把f_1定义为黑方有强大的底线，因此w_1越大，你的程序就越会试图让底线留下棋子。因此当你选择这些因素时，就必须顺便优化相应的权重。遗传算法在一开始时使用一些随机产生的程序，然后通过它们之间的对阵来找出哪些是好的哪些是坏的。一些坏的程序被筛选掉了，合适的程序幸存下来并得到繁衍——或者是在两个好的程序之间作交换(随机地在两个好的程序中选择w_i)，或者随机改变一个好的程序。现在你只管继续这样的工作，指望最终能得到一个好的权重的组合。这就是上世纪50年代Arthur Samuel在他著名的西洋跳棋程序里用的方法。另外还有其他调节权重的方法，深蓝(Deep Blue)的小组用了大量特级大师的对局，然后让程序吻合这些局面。他们对程序和特级大师走出一样着法的频率作出统计，然后修改了权重再作尝试。如果程序能解决更多的局面，那么他们就保留这个修改，否则就重新修改。最近我知道的调整权重的办法，是Michael Buro在他的黑白棋(Othello)世界冠军程序Logistello中用到的。他建立了一个巨大的局面数据库，用来储存对局结果，然后他把评价函数作用在每个局面上，通过优化评价函数使得产生的分数尽可能地准确。

所有这些方法都比机械学习更有效。如果某个程序有对西洋跳棋中类似牵制战术的评价，那么你在评价函数中提高这项的权重，会让程序认为所有类似情形的局面都是坏的，并且会避免这些局面。那么这样说来，我们是否已经学到了知识了？我认为还没有。如果你的程序本身就不包括这个评价呢？即便再怎样调节权重，也是不会有效果的。因此我们将涉及到下面的学习：

神经网络(Neural Networks)能够发明一些评价模式，而不需要有人告诉它，它能对局面作出评价，并且结合到棋类程序中。神经网络由很多输入口，一些隐含的层次结构，以及一个出口组成。这些连接着入口和出口的层次，实际上由很多结点组成，结点有连接上一层的入口和连接下一层的出口。每个结点都可以和其他层有连接，每个连接是有强有弱的。如果同时改变连接及其强度，你就能得到一些评价模式，并确定他们的权重。

把有评分的局面告诉神经网络，它就会得到训练，而在结合了遗传算法后，神经网络也会自我学习。有个西洋跳棋程序Blondie24，就是用这种方法实现了自我学习的，并且下得还不错。我认为这才是真正的学习，它和人类的思考很接近。Blondie的作者把他们的程序吹嘘得很过分，但对于目前大多数研究者来说，事实上就是如此，你必须大造声势来获得资金。

改进你的程序

你花了无数时间来写你的程序，它现在下得是否还像初学者？这是很正常的现象，但是请不要失望，我会告诉你一些技巧。首先并且最重要的是，很多问题来源于代码中的错误。Alpha-Beta算法是最容易藏匿错误的，你的程序评价了成千上万个局面，而你只得到一个着法，即使10%的局面是随机评价的，你的程序仍然在大多数时间内可以走得正确。因此，你需要你需要做一些额外检查，以确认程序的每个部分都正常工作。一个最典型的技巧就是检验评价的对称性，如果同样的局面黑白交换后，你的评价函数没有返回同样的值，那么肯定有什么地方错了。如果棋盘有其他的对称性(在国际象棋的中局里你可以按d列和e列作左右镜像操作)，你也可以利用这样的对称性。代码要尽可能地写得清晰，优化不要操之过急，否则就得不偿失了。在程序用Alpha-Beta算法走第一盘棋之前，不要加任何锦上添花的代码。只有当你根除了错误，才可以着手对程序进行调整。它是否太慢了？你可以用AMD CodeAnalyst这样的分析测试软件来考察程序的关键部分，并加以改进。让程序和别的程序下棋，这是个改进程序的非常有效的办法。程序下了几百盘棋以后，所有的统计数字就都有了，对你的代码修改一些地方(搜索算法，评价函数等等)，然后再打很多比赛来确认你改得是否有效。如果你改变的只是着法顺序，就可以用一组对比测试来代替长时间的比赛，看看花的时间是否减少了，就知道着法顺序是否改进了。

原文：http://www.fierz.ch/strategy5.htm
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 其他策略——开局库
- 下一篇 结语——国际象棋程序剖析
- 返 回 象棋百科全书——电脑象棋

《对弈程序基本技术》专题

国际象棋程序剖析

T.A. Marsland */ 文
* 阿尔伯达大学计算机科学系

一、简介

从理论上说，让机器下国际象棋是一个简单的游戏，只要每一步都考虑各种可能的着法，以及每种着法的后续变化，如此下去直到将死或和棋。但是实际操作中，这种策略是不可行的，因为需要考虑的每种着法加起来会是天文数字。

不管人类还是机器去下棋，都有一整套下棋的理论，人类好几百年前就开始下棋，在近二百年里有很多理论资料，而机器下棋只有五十多年的历史，在机器下棋的诸多理论中，最著名的属Claude Shannon在1949-50年提出的两个完全对立的策略——用蛮力考虑所有着法的策略(A策略)，以及靠知识来选择其中一部分着法的策略(B策略)。尽管早在Shannon以前，就有一些电子机械可以下比国际象棋简单的棋类，然而Shannon的理论却是现代电脑象棋发展的基础。

如今的象棋程序把棋局看成树状结构，每个局面表示成棋局树中的一个结点，每个着法代表它的一个分枝(一个结点和深一个结点的连接)。这样，树就由双方不同层面上的所有着法组成(树的一个层面用“层”(Ply)这个术语表示，代表一方走的一步棋)。

现在通常的策略是，把树由浅至深分为三个阶段，第一阶段用蛮力搜索(Shannon的A策略)，第二阶段用选择性搜索(B策略)，第三阶段用称为“静态搜索”(Quiescence Search)的策略，来处理最后变化非常剧烈的局面，在最后一个阶段里，程序只评价吃子着法，兵升变的潜力，将军产生的后果，以及一些强制性着法。所有的程序都使用相同的算法——深度优先的Alpha-Beta算法，而不同之处在于每个阶段选择什么样的深度。

通常搜索的深度不是固定的，而是根据当前搜索到的着法在小范围内变动，例如某个结点是将军的局面，那么它的分枝数很少，这个区域内搜索应当加深一些。策略上有非常多的选择，使得程序使用相同模型的时，他们的走棋风格和搜索速度会有显著的差别。

二、树状搜索

人类下棋时，往往分析一小部分着法并作展开，相比选择着法而言，机器更愿意考虑得完备，所以需要发展一套逐步求精的技术。“迭代加深”(Iterative Deepening)这一技术取代了固定的深度，让机器一次次做更深层的搜索(先是搜索N层，然后搜索$N+1$层，$N+2$层，等等)，直到按计划分配的时间用完为止，这样就可以让机器在有限的时间内找出最好的着法。诸如反驳表和置换表等技术，在与迭代加深结合起来后，可以对着法重新排序，使得下一次迭代时“主要变化”(Principal Variation)(上一次找到的最好着法)优先考虑。

另一个排序着法的技术就是记录一些“杀手”着法，这些着法要优先试探，杀手着法就是在搜索树的其他位置能产生截断或裁剪的着法。杀手着法通常是吃子的着法，因此可以简单地认为，吃子的着法应该在其他着法之前考虑。在很多程序中，这个技术被拓展为“历史启发表”(History Heuristic Table)，通常是64x64的数组，每个元素存放了该着法引起裁剪的次数。

排序着法可以大幅度提高深度优先Alpha-Beta搜索的效率，此外还有三个算法也起到同样作用——Pearl的侦察(Scout)算法以及相关的负值侦察(NegaScout)和主要变例搜索(Principal Variation Search，简称PVS)方法，它们具有同一个思想：当主要变例找到时，可以先去验证其他着法是否都

不如这个着法好，如果验证下来不是这样，那么它们必须重新搜索(因为它们是更好的着法，必须做完全的搜索)。

另一个能减少搜索量的技术称为“期望搜索”(Aspiration Search)，根据当前局面估计一个范围(通常正负半个兵)，用这样一个窗口进行搜索。尽管期望搜索并不那么有效，但是用它的人还是很多的，因为它比主要变例搜索更容易理解。

深层次的搜索到底能提高多少精确度，这是很难衡量的。任意局面其搜索树的大小有很大的差异，很多残局每方有约8种着法，而复杂的中局里每方可能有多达80种着法，就现在的技术而言，一般程序在中局阶段能完全考虑7到10层的变化(有个程序设计师声称选择性地搜索能达到40层！)。

“选择性延伸”(Selective Extension)是对一些关键着法作更深入的探索，每个程序设计师对这些着法都有自己的界定——可以是防御杀棋的着法，或是为避免失子而作的反击。选择性延伸不能和“单步延伸”(Singular Extension)混淆起来，后者是对最好的一步着法进行的更深层次的检查，来验证这个着法是否仍旧是最好的。从某种意义上说它对主要变例进行了短的延伸，是一种很花时间但很有意义的方法。

而用得最广泛的做法是“空着启发”(Null-Move Heuristic)，即假设一方连走两步，如果产生的结果更糟的话，后面那步变化就应该抛弃掉。由于水平线效应，一些看不到的不可避免的失子，通过这种方法能看到。很多向前裁剪方法起不到很好的效果，但空着向前裁剪往往是很有效的。

三、置换表

置换表的作用如同缓冲存储器，用来记录前面搜索过的局面，这些局面通常在迭代加深的早期搜索过。之所以称为“置换表”(Transposition Table)，是因为当着法次序发生置换时会有相同的局面。置换表中存储的信息除了局面以外，还有局面的评分，它的最佳着法，以及前一次搜索的深度。

“评分”指的是在搜索树的顶端结点(搜索能够到达的水平)对局面进行的评价，这个评价通常包括了静态搜索，来消除由吃子等着法引起的局面的不确定性，包括考虑兵的升变等。

在残局的搜索中，置换表的作用发挥得更为明显，每个局面只有很少的着法需要考虑，其他着法都因为置换的发生而不必再去搜索。

置换表并不增加代码的长度和复杂性，为置换表分配空间是小事一桩。置换表的每个局面一般需要约10个字节，通常程序使用可以记录32K到1M个局面的置换表。1993年超级电脑(Supercomputer)程序自称他们使用了1G个局面的置换表，而对于程序员来说这完全取决于存储器的容量。

四、程序的性能及其测评方法

尽管各种程序基本方法是一样的，但是在相同硬件条件下，它们的表现有很大程度的差异，一定程度上这体现了程序设计师对程序所作的投入。

比如说，尽管每个程序都有开局库，但开局库是没有哪本书规定的，每个程序小组都在开发自己的开局库。现在开局库的大小从10,000个局面到500,000个局面不等(也有程序试验过1.7M个局面的开局库)。

相反，只有少数人使用Ken Tompson制作在CD-ROM上的5子到6子的残局库，一方面原因是目前数据的读取非常缓慢，另一方面原因是大部分棋局都会在用到这些残局库之前结束(程序设计师们在利用时间的问题上，可能考虑得比较现实吧)。

在运行速度的层面上，当今的程序在单处理器上每秒钟可以分析3,000到500,000个局面。在相同机器上程序速度的差别是巨大的，这当然有很多解释，用汇编语言写的程序能有比较快的速度，用同一语言来写的程序，用不同的编译器也会导致速度的不同。

其中更多的因素在于蛮力搜索、选择性搜索和静态搜索这三个阶段所占的比例。选择性搜索阶段需要额外的时间，因为它还判断哪些着法需要搜索，而在这个慢速、以知识为基础的阶段中，诸多程序在速度上的差别是非常明显的。

另一个影响搜索速度和强度的因素，是程序使用的置换表的大小。

尽管很多象棋程序非常相似，它们的棋力差别仍旧可能非常大。测试棋力是很复杂的过程，由于在正式比赛中程序仍然是可调节的，因此“瑞典排名表”(Swedish Rating List)的制作小组提出了一个传统的方案——所有的商业程序都连续自动地进行对决，得到几百场比赛的胜负结果，并从这些结果中计算出ELO等级分，即美国和其他地方棋手们通用的等级分。

美国棋手的平均等级分从1500开始，专家的分数在2000-2200，大师在2200-2400。世界排名300以内的特级大师，等级分则更高。在第八届世界电脑象棋锦标赛中，大多数程序的等级分在2100-2500。现在瑞典排名表刊登在每期的《国际电脑象棋协会杂志》(*International Computer Chess Association Journal*)上。

五、展望

如今顶极的电脑可以挑战特级大师了，尤其是在快棋赛中，单机要比多处理器的并行系统更有优势。单机之所以快，是因为它们不必通过网络来传输着法，而由10到100个处理器组成的多处理器系统，则在两小时内完成40步的标准赛制中表现得更好。不久我们将会有由1000个高性能处理器组成的系统，到那个时候电脑毫无疑问能比人类棋手算得更远，这将多少能弥补电脑的不足——不具备人类棋手所擅长的战略构思。

人类棋手对于局势的简化和基本形势的判断方面都具有高超技巧，他们也善于在对手身上找到弱点，除非局面是无法挽回的。尽管人类棋手有这些优势，我们一直以来都预测说，机器在5年内能击败人类，现在这个目标正越来越近。要实现这个目标可能需要花上十多年，但是最终肯定会实现的。

《国际象棋程序剖析》参考资料

1.《电脑象棋概论》(*Computer Chess Compendium*)，D.N.L. Levy著，Springer-Verlag出版社，1988年；

2.《电脑象棋与搜索》(*Computer Chess and Search*)，T.A. Marsland文，《人工智能百科全书》(*Encyclopedia of Artificial Intelligence*)一书的224-241页，S. Shapiro主编，J. Wiley & Sons出版社，1992年第二版；

3.《ICCA(国际电脑象棋协会)杂志》(*International Computer Chess Association Journal*)，自1983年以来每年四期，ICCA编辑部([荷兰马斯特里赫特]林堡大学(University of Limburg)计算机系，P.O. Box 616, 6200 MD Maastricht, The Netherlands)出版；

4. 电脑、象棋与认知(*Computers, Chess and Cognition*)，T.A. Marsland和J. Schaeffer主编，Springer-Verlag出版社，1990年；

5. 电脑象棋进展(*Advances in Computer Chess*)，一至七卷，从1977到1994年由多人主编，由多个出版社出版。

出处：http://www.cs.ualberta.ca/~tony/ICCA/anatomy.html
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译

- 上一篇 其他策略——策略和技巧
- 下一篇
- 返 回 象棋百科全书——计算机博弈

www.xqbase.com