论文题目： 电脑象棋的设计与实现

专业： 计算机软件与理论

硕（博）士生： 涂志坚

指导老师： 姜云飞教授

摘要

计算机博弈是人工智能领域中的一个重要主题，而当前对中国象棋博弈的研究也在不断地发展着，本文通过对象棋程序“纵马奔流”（取得了第 8 届 Computer Olympiad 象棋软件金牌）的数据结构、搜索算法与评价函数的描述，以及对所采用的数据结构、搜索方法进行基于测试数据集的分析和检验，阐述了一个可达人类大师水平的象棋程序的设计及实现的原理、方法。且在当中提出了新的 BitFile、BitRank 数据结构实现以及 Transposition Table 存储的 Fail Low 修正算法，有效地提高了实际搜索运算的效率。

关键字： 计算机博弈，电脑象棋，启发式搜索

Title：Design and Implementation of Computer Xiangqi

Major: Computer Science

Name: Zhijian Tu

Supervisor: Prof. Yunfei Jiang

Abstract

Computer game playing is an interesting subject in artificial intelligence research. Also, the research on computer xiangqi(Chinese Chess) have been developed rapidly in recent years. In this article we describe the data structure, search algorithms, and evaluation function of xiangqi program ZMBL, which won the gold medal in 8th Computer Olympiad held at Graz, Austria. In addition, there are analyses and statistics basing on test data suites to express the benefits that the searching methods could bring. More over, this article provides several innovations such as the data structure BitFile, BitRank and an improve algorithm in Transposition Table storing when fails low.

Key words: Gaming Theory, Computer Xiangqi, Heuristics Search

目 录

第一章 引言

1.1 电脑象棋简介

在人类文明发展的初期，人们便开始进行棋类博弈的游戏了。到了近 50 年前，随着电子计算机的诞生，科学家们开始通过电脑模拟人的智能逐步向人类智能发起挑战，香农(1950)与图灵(1953)提出了对棋类博弈程序的描述，随着电脑硬件和软件的高速发展，从 1980 开始，电脑博弈便开始逐渐大规模地向人的智能发起了挑战，到了 1997 年，IBM 超级电脑 Deeper Blue 击败了当时国际象棋世界冠军卡斯帕罗夫，成为了人工智能挑战人类智能发展的一个重要旅程碑。

而在空间复杂度以及搜索复杂度比国际象棋更高的中国象棋（以下简称象棋）方面，当前的研究也正蓬勃地发展着，水平也一步步地提高，并出现了不少优秀的象棋程序，包括台湾大学许舜钦教授及其团队的 ELP、美国吴韧博士的梦入神机、台湾郑明政的象棋世家、法国 Pascal Tang 的 XieXie 等，出现了研究和竞技百家争鸣的好景象。随着研究的深入以及电脑硬件速度的增长，在不远的将来，象棋程序将会向顶尖人类高手发起强而有力的挑战。

表 1-1 各种棋类复杂度比较

	空间复杂度	搜索树复杂度
黑白棋	10^{30}	10^{58}
国际象棋	10^{50}	10^{123}
中国象棋	10^{60}	10^{160}
围棋	10^{160}	10^{400}

电脑象棋的研究常常通过比赛对弈的方式来验证各自研究的成果，当前的主要比赛包括：由 ICGA 每年举办的 Computer Olympiad，以及从 2004 年起在台湾成功大学举办的世界电脑象棋争霸赛。

1.2 象棋程序“纵马奔流”

本文将会介绍象棋程序“纵马奔流”（以下简称 ZMBL）的设计原理与实现方法。ZMBL 从 2001 年 10 月份开始编写，其主要由开局知识库、搜索引擎、评价函数以及一系列的知识学习机制组成，主要的特点包括快速的数据结构设计实

现、以及高效率的搜索，这些机制保证了 ZMBL 在应用大量领域知识的同时能保持高速的搜索速度与深度，做到知识与速度的均衡。从 2002 年 2 月开始，ZMBL 通过 Telnet 方式在专业象棋网站 www.movesky.net（对弈者包括大量的专业象棋大师）进行自动运行的测试，在 2002 年底取得快棋排名第一，并且 2003 年开始取得慢棋保持前 3 名的成绩。与此同时，ZMBL 开始参加电脑象棋的比赛，并于 2003 年 11 月在奥地利格拉兹举行的 8th Computer Olympiad 的取得了电脑象棋的世界冠军。

1.3 本文架构

第一章将会阐述计算机博弈发展以及电脑象棋介绍、电脑象棋的状态空间复杂度以及文章架构。

第二章首先描述了电脑象棋表示的一些基本数据结构，然后提出了 ZMBL 所采用的双向映射数组的盘面表达方法，以及优化的快速运算数据结构----BitFile、BitRank 的设计实现原理。

第三章将重点讲述博弈树的搜索方法，包括博弈树构建、Alpha-Beta 算法介绍、NegaScout 算法与 Minimal Window、Quiescence Search、Transposition Table、Iterative Deepening、Move Ordering、选择性剪枝和延伸，以及提出新的 Transposition Table 存储的 Fail Low 修正算法，最后对整体搜索框架进行总结。

第四章主要描述 ZMBL 的评价函数架构，包括子力分数、子力灵活度评价、棋盘控制以及一些重要特征的计算。

最后，第五章是全文的总结及展望。

第二章 数据结构

在一个搜索体系，有效地对研究目标构架一个好的数据结构表示，既能快速便于搜索地进行，更能大大提高搜索的效率。

而在中国象棋博弈的研究中，显然怎样很好的表示棋盘，也是决定搜索效率的关键之一，一般来说，博弈运算的时间耗费主要在搜索和评价中。搜索时，不断根据当前棋盘，生成可行移动，然后进行搜索树的扩展；同时评价函数也是根据当前棋盘情况进行相关特征的计算和估价。因此，好的棋盘数据结构的表示，能大大提高以上这两个部分的运行效率。

2.1 棋盘数组和棋子数组的双向映射模式

在本人编写的中国象棋程序中，采用了是一个棋盘数组和棋子数组双向映射表的形式。

以下首先说说一些辅助知识：

2.1.1 棋盘坐标

棋盘坐标是指对棋盘建立坐标系，通过坐标标示棋盘上每一点的方法。

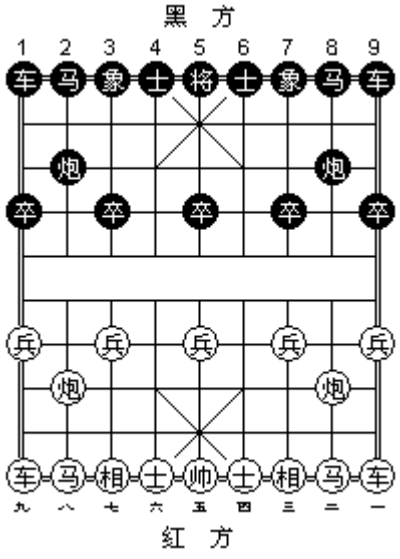


图 2-1

2.1.1.1 二维坐标和一维坐标的比较

二维坐标，表示比较直观，譬如棋盘横坐标从左开始 0 到 8，纵坐标从上到下为 0 到 9 的话，棋盘左上角的点就是 Board[0][0]，整个棋盘坐标用 Board[9][10] 来定义，这样表示比较直观。

一维坐标，举个例子，棋盘第一行 0，1，2，3，4，5，6，7，8，第二行 9，10，11，12，13，14，15，16，17，如此类推，用一维数组表示坐标。计算棋盘上某个点的横坐标纵坐标，只需作 N%9 和 N/9 计算便可。

比较两种方式，由于在搜索、评价时，访问用一维数组的方式访问棋盘比用二维数组的方式快（少了一次地址偏移操作），采用的一维数组有速度上的优势。

2.1.1.2 冗余辅助点

冗余辅助点：当进行树扩展时（Generate Move），譬如棋子车，马，炮，兵等生成可移动步时，举个例子，车的向左移动，每次都要和左边界进行比较，看当前坐标是否已经超出棋盘，浪费了效率。所以通过在棋盘四周建立冗余点，记录特殊标志，避免边界检测运算。

2.1.1.3 使用的棋盘坐标表示方法

使用一维数组，结合冗余辅助点，建立如下的棋盘坐标：

0	1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38
39	40	41	42	43	44	45	46	47	48	49	50	51
52	53	54	55	56	57	58	59	60	61	62	63	64
65	66	67	68	69	70	71	72	73	74	75	76	77
78	79	80	81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100	101	102	103
104	105	106	107	108	109	110	111	112	113	114	115	116
117	118	119	120	121	122	123	124	125	126	127	128	129
130	131	132	133	134	135	136	137	138	139	140	141	142
143	144	145	146	147	148	149	150	151	152	153	154	155

156	157	158	159	160	161	162	163	164	165	166	167	168
169	170	171	172	173	174	175	176	177	178	179	180	181

图 2-2

其中周围的橙色格为冗余辅助点，中间部分为实际上的棋盘。
上下左右的偏移量为：{**-13**，**+13**，**-1**，**+1**}

2.1.2 棋盘数组表示

棋盘数组是指上面所描述的棋盘坐标上每个点所对应的数据，即根据棋盘坐标建立研究目标表示的索引。然后我们进行数据编码：

(1) **空格**：0

(2) **冗余点**：Dummy(30)

(3) **棋子表示**：

红车	红马	红炮	红兵	红士	红相	红帅
11	12	13	14	15	16	17
黑车	黑马	黑炮	黑卒	黑士	黑象	红将
21	22	23	24	25	26	27

其中，十位表示棋子颜色，个位表示棋子类别（依次是车，马，炮，兵，士，象，帅）。

2.1.3 棋子数组表示

棋子数组：即根据棋子列表进行索引。在评价函数和生成步中，按照棋盘数组从 90 个点一个一个寻找棋子，然后进行评价或者生成步的话，我们就需要每次都白白循环 90 次。为了提高运算的效率，我们另外增加了棋子数组 ChessPos[1….32] 形成有效的棋子索引，如果 ChessPos[K] = 0，1≤K≤32 的话，表示该棋子不存在，否则的话，表示棋子的实际在棋盘上的坐标。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	红车		红马		红炮		红兵					红士		红相		红帅

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
黑车		黑马		黑炮		黑卒					黑士		黑象		黑将

2.1.4 双向映射数组

棋盘数组(2.1.2)、棋子数组(2.1.3)均可对一个棋盘对象进行完整的描述。单用棋盘数组(2.1.2)能方便地检阅棋盘上任一点上的数据，但要依次检索某一方棋子的时候，要对整个棋盘进行一次检索（需要 90 次比较），效率较低。单用棋子数组(2.1.3)，能方便知道某棋子在棋盘上的坐标，但要检阅棋盘状况，却要对棋子数组扫描一次（需要 32 次比较）。为了充分利用两者各自的优点，又同时避免两者，本人主要把两种方法结合起来，构建两者的映射数组，进行棋子移动的时候根据该数组进行棋盘数组和棋子数组很少的调整，以达到同时容纳棋盘数组方式和按棋子数组的方式对数据进行快速的访问。

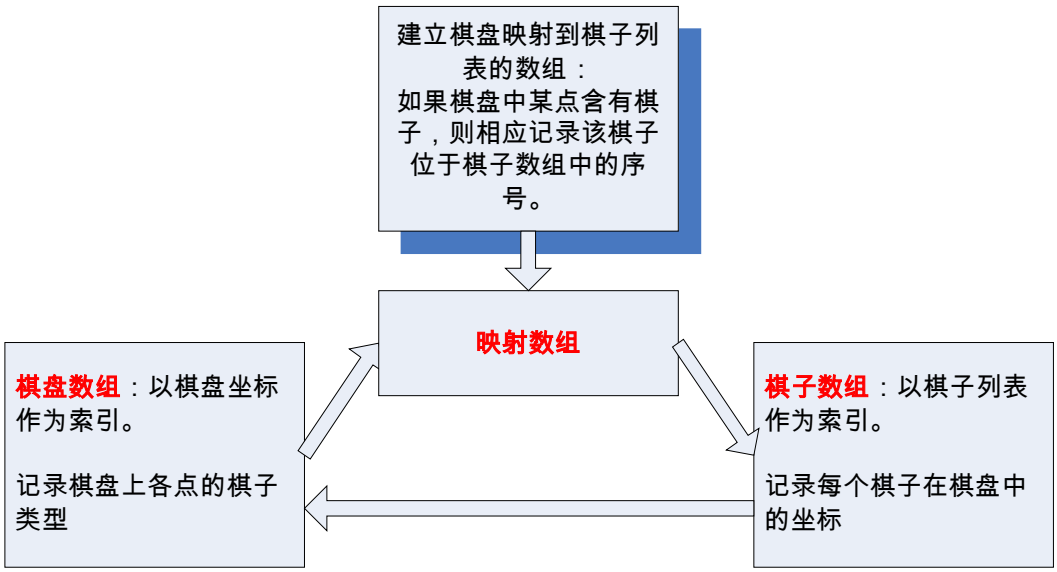


图 2-3

2.2 优化的快速数据结构（BitFile，BitRank）实现

要提高一个象棋软件的对弈水平，在不修改评价函数的前提下可以这样认为，就是以在搜索相同的深度里耗费的时间最少作为目标，搜索的深度越深，能发现一些微妙情况的概率就越大。而要提高这个速度，有两种方法，一是优秀的

效率更高的搜索算法（这将在本文其他章节讨论），二是提高单位时间内搜索的结点数目。采用好的数据表示方法，能达到计算的快速实现，基于这个道理，本人建立了一个新的数据结构表示方法----BitFile，BitRank 来实现运算的加速，就是对棋盘的每一行，每一列，以 Bit 0 或者 1 来记录该点上是否有棋子，这样的话每一行、每一列都有一个编码值。其中是 2^9=512 的编码空间，列是 2^10=1024 的编码空间。这样在大量的直线车炮评价、将军判断、步法生成的时候，就不用对其所在行进行一次遍历（这样的遍历需要对该行每一个坐标分别遍历，最多达 8+9=17 次），只需获得当前的 BitFile，BitRank 的编码值，并在搜索前预算计算对应编码值的特征估价值，通过查表立刻可得。

2.2.1 预计算以及延展递增计算的数学模型

在数学上，评价函数是对一个盘面定性的静态评价，其计算方法为：

$$EvaluationScore = \sum K_i F_i(P)$$

其中 K_i 为特征系数，$F_i(P)$为特征函数，P 为当前棋盘。

对于特征函数 $F_i(P)$的计算，实际上只需要棋盘 P 的局部数据。所以，$F_i(P)$可简化为

$F_i(P) = f_i(x_1, x_2, ..., x_k)$，其中 $x_1, x_2 \ldots . x_k$ 为局部数据，一般指可直接得到的原子数据。

如果存在函数 f’，使

$f'_i(y_1, y_2, ..., y_n) = f_i(x_1, x_2, ..., x_k)$，其中 $y_i = g_i(P)$，满足条件：

（1） y_i 的分布范围比较狭小，同时 n 的值也较少

（2） 搜索扩展的时候，y_i 的值在棋盘 P 变更的时候，仅需作计算量较少的维护，便可以得出新棋盘的 y_i 值。也就是说，对于树的中间结点 P_1，通过着法移动生成结点 P_2，由于一步着法产生的棋盘变化比较少。所以，可以快速地根据 $y_i(P_1)$，以及移动产生的少量变化，直接计算出 $y_i(P_2)$，而不需要完全重新计算出 $y_i(P_2)$。

那么的话，我们就可以建立延展递增的方法，来计算 $F_i(P)$。

计算方法

（1） 首先在搜索前，预先计算出对于所有可能出现的($y_1,y_2,.....y_n$)集合，对应 $f'_i(y_1,y_2,.....,y_n)$ 的值，存储在 Pattern[y_1][y_2].....[y_n]中，设 y_i 的定义域长度函数为 DefLen()，则所需的内存空间为： DefLen(y_1) * DefLen(y_2) * …DefLen(y_n) * sizeof(int)

（2） 然后在树扩展的时候，对中间结点进行递增改变 y_i。根据着法(棋盘 P1➔P2)产生的少量数据变化，从 $y_i(P_1)$计算 $y_i(P_2)$。

（3） 叶子结点的 $F_i(P)$计算，仅需查表获取 Pattern[y_1][y_2].....[y_n]的值。

计算耗费及评价：

原来的 $F_i(P)$计算是在叶子结点中进行，根据当前棋盘计算 $F_i(P)$，对于整颗搜索树来说，其时间耗费约为：

T_1 = 叶子结点数目 * $F_i(P)$的计算的耗费

延展递增的计算方法是在搜索树延展的时候，通过递增改变的计算方式，减轻在叶子端点的计算耗费。其时间耗费约为：

T_2 = Pattern[y_1][y_2].....[y_n]的预计算时间 + 结点扩展次数 * 递增改变的数据维护计算 + 叶子结点数目 * 查表获取 Pattern 值的时间耗费

其额外的空间耗费，主要为模式预计算结果的保存空间：

DefLen(y_1) * DefLen(y_2) * …DefLen(y_n) * sizeof(int)

比较 T_1 与 T_2 的计算公式，由于 Pattern[y_1][y_2].....[y_n]的预计算时间是一个定值，不随搜索树的增大而变化，并且一般计算量比较少，其计算时间几乎可以忽略；另外查表获取 Pattern 值的时间耗费，只是一次性访问内存所需时间，其计算时间也可以忽略。而对于电脑象棋的搜索树来说，结点扩展次数仅比叶子结点数略多，所以当递增改变的数据维护耗费比较远远少于直接计算特征函数 $F_i(P)$ 的时候，$T_2 < T_1$。

2.2.2 BitFile、BitRank 实现的主要思想

在理论分析的角度看，数据结构 BitFile、BitRank 体现了上面所叙述的预计算和延展递增计算的思想，将部分复杂的评价函数计算部分，通过延展递增、预计算的方法来计算，有效地提高了计算效率，并在将军判断、步法生成等部分，通过预计算的方法，降低计算耗费。

在实现的角度看，数据结构 BitFile、BitRank 的主要思想是

（1） 数据映射：将多个数据映射到一个数据，减少多个数据计算的复杂性。

（2） 利用空间换取时间：在数据映射的基础上，通过预计算，在实际搜索运算过程中可以快速查表获取结果。

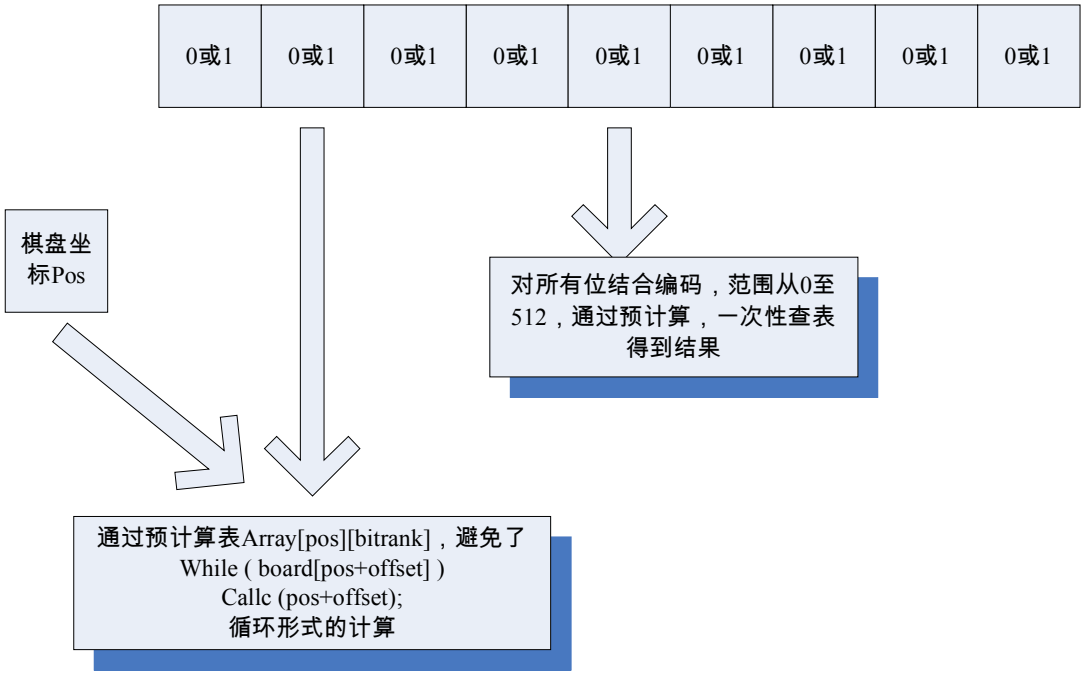


图 2-4

将复杂的循环条件代码简化：

```
for ( loop1 = 0; loop1 < 3; loop1++ )
{
   while ( board[pos+offset[loop1] == 0 )
   {
            Add Mobility Score;
            Add Control Score;
   }
}
```

```
Result += EvalBitRank[pos and color][bitrank];
Result += EvalBitFile[pos and color][bitfile];
```

图 2-5

2.2.3 BitFile、BitRank 的性能测试

使用 BitFile、BitRank 数据结构和不使用 BitFile、BitRank 的搜索结点数比较(测试数据集为 50 个中局盘面)：

表 2-1

	不使用 BitFile、BitRank		使用 BitFile、BitRank		
搜索深度	结点数	耗时(S)	结点数	耗时(S)	Δ 耗时 (%)
1	29,115	1.515	29,115	1.391	-8.18
2	68,274	1.516	68,274	1.406	-7.26
3	202,535	1.625	202,535	1.469	-9.60
4	1,031,448	2.312	1,031,448	1.953	-15.53
5	3,696,393	4.985	3,696,393	3.375	-32.30
6	14,089,815	12.781	14,089,815	8.875	-30.56
7	45,134,350	36.734	45,134,350	24.734	-32.67
8	164,786,147	135.375	164,786,147	87.047	-35.70

从上表可得，通过 BitFile、BitRank 的快速数据结构实现，可以在不改变搜索结点数的基础上减少用时的 30%以上。

第三章 搜索方法

3.1 传统 Alpha-Beta 算法介绍

在最大最小树搜索的过程中，实际相当一部分结点的搜索并不会影响最终搜索树的值，Bruno 在 1963 年首先提出了 Alpha Beta 算法，1975 年 Knuth 和 Moore 给出了 Alpha Beta 的数学正确性证明。Alpha Beta 算法通过下界（Alpha）和上届（Beta）对搜索树值的最终范围进行了划定，当某些子树其值被证明会在上述界限之外，无法影响整颗树的值时，便可进行剪枝。Alpha Beta 剪枝算法的效率与子结点扩展的先后顺序相关，在最理想情况下，其生成的结点数目为：

$$N_D = 2B^{D/2} - 1 \quad (D\text{为偶数}) \qquad N_D = B^{(D+1)/2} + B^{(D-1)/2} - 1 \quad (D\text{为奇数})$$

(其中，B 为 Branch Factor，D 为深度)

由于不使用 Alpha Beta 剪枝算法时，$N_D = B^D$，所以最理想情况下 Alpha-Beta 算法搜索深度为 D 的结点数仅相当于不使用 Alpha-Beta 时搜索深度为 D/2 的结点数。

```
int AlphaBeta(int alpha, int beta, int depth)
{
    if (game over or depth <= 0)
        return winning score or eval();

     GenMove();

    for ( i = 0; i < MoveListLength; i++ )
    {
        make move m;
        val = - AlphaBeta(-beta, -alpha, depth-1);

        undo make move m;
        if ( val > alpha )
        {
            if ( val >= beta ) //beta cut
                return val;
            alpha = val;
        }
    }
    return alpha;
}
```

表 3-1 NegaMax 形式的 Alpha-Beta 算法伪代码

以下为以 NegaMax 形式（每层之间更改 Alpha,Beta 值符号，使 Alpha 剪枝形式在代码中也简化成通过 Beta 剪枝的形式表现出来）表现的 Alpha-Beta 算法伪代码：

Alpha-Beta 剪枝图：

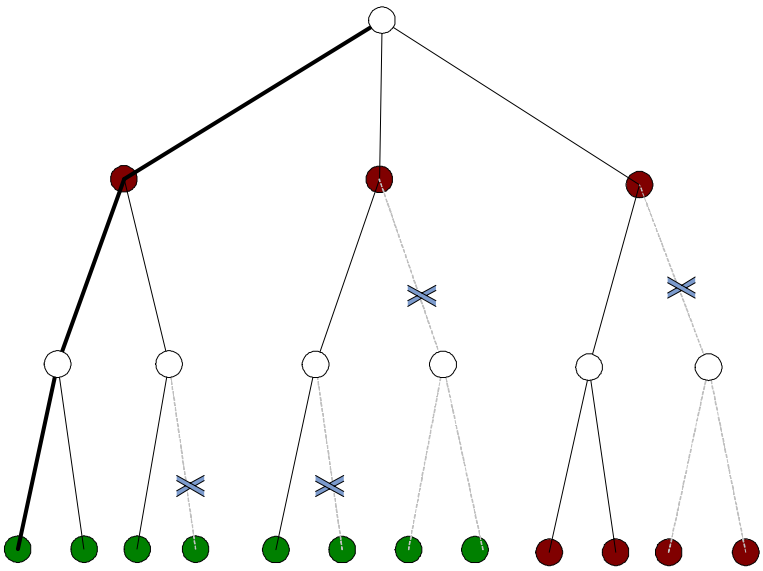


图 3-1 Alpha-Beta 剪纸图

3.2 NegaScout 算法及 Minimal Window

Minimal Window：是指窗口为 0 的 Alpha、Beta 限制范围，譬如[N,N+1]，因为不可能存在一个整数既大于 N 又小于 N+1，所以这个范围内不可能存在真实结果。同时因 Minimal Window 的 Alpha、Beta 限制范围最小，所以产生剪枝的概率也比正常的搜索窗口的大。

采用 Minimal Window 的意义：如果搜索一个子结点的时候，当其值大于 Beta，即产生 Beta 剪枝的概率比较大的时候，我们可以先用[Beta-1,Beta]进行搜索，如果搜索的结果 Fail High，即说明了结果>= Beta，这时我们可以立刻进行 Beta 剪枝，相反如果结果 Fail Low，则说明结果不可能大于 Beta，也就说无法剪枝，那么我们刚才用[Beta-1,Beta]的 Minimal Window 就无法得到真实结果，所以我们必须重新用[Alpha,Beta]进行搜索。虽然当无法 Beta Cut 需要重新搜索的时候，实际比原来多出了 Minimal Window [Beta-1,Beta]的搜索过程，但是由于 Minimal Window 内的子树产生剪枝的概率比较大，所以实际增加的搜索结点数相对比较

少，同时当我们后面章节所提到的 Move Ordering 比较好的时候，大部分结点能够产生 Beta 剪枝的概率相当大，这样我们就可以避免更繁重的全 Alpha、Beta 窗口搜索。

在采用 Minimal Window 的 NegaScout 算法中，每次递归扩展的时候，第一个节点用正常的 Alpha-Beta 窗口扩展，而后面的都用 Minimal Window 进行扩展，这样的话如果最优的孩子结点最先扩展，则此后的其他孩子结点通过 Alpha 值对应的 Minimal Window 进行搜索都可以轻松剪枝，并且由于采用了 Minimal Window，Alpha-Beta 窗口缩到最小，剪枝也更容易，搜索节点也更少，反之，如果发现该扩展结点使用 Minimal Window 不产生剪枝（即值小于 Beta），且值仍大于 Alpha，我们才用正常的 Alpha-Beta 窗口进行再搜索（Re-Search）。

```
int NegaScout(int alpha, int beta, int depth)
{
    if (game over or depth <= 0)
        return winning score or eval();

     GenerateMove();

    for ( i = 0; i < MoveListLength; i++ )
    {
        make move m;
        if ( i == 0 )
            val = - NegaScout(-beta, -alpha, depth-1);
        else {
            val = - NegaScout(-alpha-1, -alpha, depth-1);

        if ( (val > alpha) && (val < beta) )
              val = - NegaScout(-beta, -alpha, depth-1);

        undo make move m;
        if ( val > alpha )
        {
            if ( val >= beta ) //beta cut
                return val;
            alpha = val;
        }
    }
    return alpha;
}
```

表 3-2 NegaScout 算法伪代码

3.2.1 Negascout 算法性能测试

使用 Alpha Beta 算法和使用 Negascout 算法的搜索结点数比较(测试数据集为 50 个中局盘面)：

表 3-3

	Alpha Beta	NegaScout	
搜索深度	结点数	结点数	Δ 结点数(%)
1	29,292	29,115	-0.60

2	69,117	68,274	-1.22
3	199,864	202,535	1.34
4	1,112,866	1,031,448	-7.32
5	3,768,023	3,696,393	-1.90
6	15,604,356	14,089,815	-9.71
7	49,494,233	45,134,350	-8.81
8	184,374,485	164,786,147	-10.62

Negascout 算法能减少 10%左右的搜索节点数。

3.3 宁静搜索(Quiescence Search)

一般 Alpha-Beta 搜索用固定的深度进行搜索，这样就产生了水平效应，譬如搜索深度为 9，它就无法看到第 10 层后发生的事情，即使第 10 层会失去一车。所以为了尽量地减少这种水平效应，我们可以采用对 Alpha-Beta 搜索树的叶子结点(Depth=0)以下的一些有兴趣的路径继续搜索，直到局面达到平静为止，这样对一些比较混乱的局面能有个较精确的估计。

在象棋中，在搜索树叶子之上的一层结点（即深度为 1 的结点），如果不会有任何延伸的话，那么进行吃子将会立刻得到分数上的提升，不管对方是否有保护和能否回吃。因为它走完这一步之后，深度已经变为 0，也就是不会继续搜索对方的棋步了，所以对方在“搜索逻辑上”无法回吃。这造成分数起伏很大，并无法准确体现局面的正确评价值。

所以我们应该对深度为 0 的叶子结点以下的所有吃子进行延伸，但是当认为不需要吃子的时候，可以选择不走棋（吃子），这时候将结束宁静搜索的继续搜索。

Quiescence Search 的伪代码：

```
quiesce(int alpha, int beta)
{
    int score = eval();
    if (score >= beta) return score;
    for (each capturing move m) {
        make move m;
        score = -quiesce(-beta,-alpha);
        unmake move m;
        if (score >= alpha) {
            alpha = score;
            if (score >= beta) break;
        }
    }

    return score;
}
```

表 3-4 Quiescence Search 伪代码

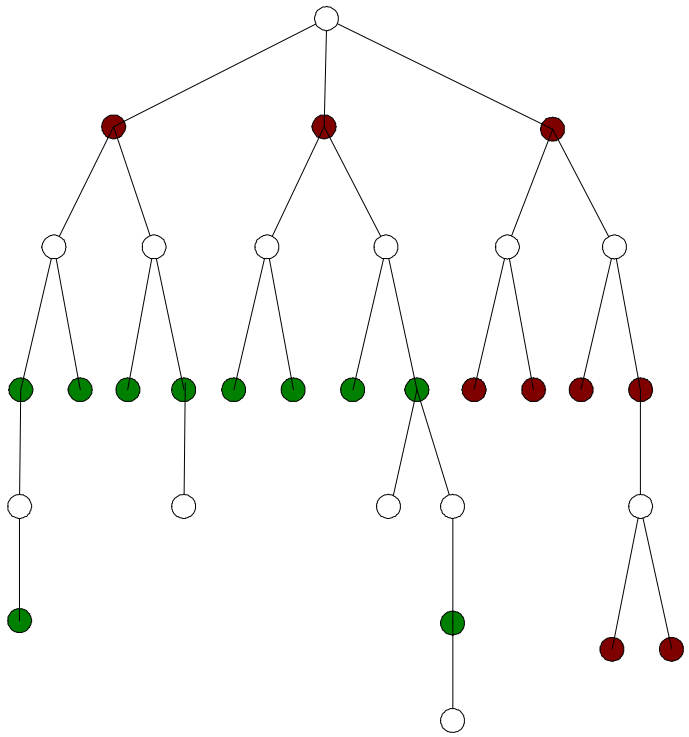


图 3-2 Quiescence Search 示意图

3.4 Transposition Table

在一颗搜索树中，不少结点之间虽然是经过不同的路径到达的，但其状态是完全一致的。通过建立 Transposition Table 保存已搜索结点的信息，那么再次遇到相同状态的结点时便可以套用之前的搜索结果。在 Transposition Table 中，记录的信息为最优着法、分值、深度，为了快速检测当前结点是否已经搜索过，我们一般对当前状态进行编码，然后利用 Hash 表的方式进行查找，这里采用的编码方式是 Zobrist Hash 方法。

Zobrist Hash 方法：在搜索之前，生成大随机数(64 位)数组 ZB[棋盘坐标][棋子类型]与代表走棋方的随机数 MoveSideHash。然后当前棋盘的 Hash 值便是棋盘上所有存在棋子的坐标上对应 ZB[s][p]的异或之和。这样产生移动后，并不需要重新计算棋盘的 Hash 值，只需将当前 Hash 值
(1) 异或移动棋子原坐标对应 ZB 值
(2) 异或移动棋子新坐标对应 ZB 值
(3) 如果产生吃子的话，需要异或被吃子在其对应坐标的 ZB 值
(4) 异或 MoveSideHash，表示改变待走棋一方的颜色。
这是根据两次异或同一个数结果保持不变的原理，避免重新计算整个棋盘的 Hash 值，并且位的异或在计算机内部运算中速度较快。

Transposition Table 是一种空间换取时间的思想，由于实际物理内存空间有限，所以对存储入 Transposition Table 的点会有所限制，在 ZMBL 中，Quiescence Search 的节点不存入 Transposition Table。并且为了尽量提高 Transposition Table 的效率，根据结点深度、访问的局部性理思想，确定存储替换算法。

Transposition Table 结合 AlphaBeta 搜索：在 Alpha Beta 搜索的过程中，一个结点会出现以下三种情况之一：(1) Fail High，结点值至少>=Beta，但不知道其具体值。(2)Fail Low，结点值至多<=Alpha，但不知道具体值。(3)Exact，Alpha <= 结点分值 <= Beta，此值为准确值。一般只有 Exact 类型，才可作为当前结点的准确值存入 Transposition Table，但 Fail High(<=Beta)、Fail Low(>=Alpha)所对应的边界值仍可帮助我们作进一步的剪枝，所以我们在 Transposition Table 用字段 entry_type 表示类型，其中 Exact 类型为准确值，其余的则表示值的范围。
在搜索过程中，首先检查 Transposition Table 中保存的结果能否直接代表当前结结的值或使当前结点产生 Alpha、Beta 剪枝，不能的话则继续进行该结点的搜索。

Transposition Table 结合搜索的伪代码：

```
int NegaScout(int alpha, int beta, int depth)
{
    if (game over or depth <= 0)
        return winning score or eval();

    //查找 Transposition Table
    if ( ProbeTT() > 0 )
    {
        if ( depth <= hashtable[position].depth )
        {
            switch (hashtable[position].entry_type)
            {
                case EXACT:
                    return hashtable[position].score;
                case LOWERBOUND:
                    if (hashtable[position].score >= beta)
                        return (hashtable[position].score);
                    else break;
                case UPPERBOUND:
                    if (hashtable[position].score <= alpha)
                        return (hashtable[position].score);
                    else break;
            }
        }
    }

    GenMove();

    for ( i = 0; i < MoveListLength; i++ )
    {
        make move m;
        if ( i == 0 )
            val = -NegaScout(-beta, -alpha, depth-1);
        else {
            val = - NegaScout(-alpha-1, -alpha, depth-1);

        if ( (val > alpha) && (val < beta) )
            val = - NegaScout(-beta, -alpha, depth-1);

        undo make move m;
        if ( val > alpha )
        {
            bestmove = m;
```

```
            if ( val >= beta ) //beta cut
              {
                //插入 Transposition Table
                 InsertTT(hashvalue,val,LOWERBOUND,depth,bestmove);
                   return val;
              }
            alpha = val;
        }
    }

    //插入 Transposition Table
    if ( old_alpha == alpha )
        InsertHash(hashvalue,alpha,UPPERBOUND,depth,bestmove);
    else   InsertHash(hashvalue,alpha,EXACT,depth,bestmove);

    return alpha;
}
```

表 3-5 使用 Transposition Table 的伪代码（加粗倾斜部分）

3.4.1 Transposition Table 的作用

(1) 如果在 Transposition Table 访问中能直接得到结果的话，则可以避免该结点以下子树的搜索，减少了搜索时间。

(2) 如果在 Transposition Table 中能查找到当前结点信息，并且存储深度比当前结点大，那么实际上是增加了当前子树的搜索深度，也即增加了结果的准确性。

(3) 结合 Iterative Deepening，也即在逐层深入的搜索中，获取本结点之前的最优步，作为第一个儿子结点进行扩展，有效地提高 Move Ordering，减少搜索结点数，这部分将会在以下章节中进行详细论述。

3.4.2 Transposition Table 的存储替换算法

ZMBL 的 Transposition Table 采用的是双层存储方式，其替换算法如下：

(1) 如果待置入结点的深度大于等于 Transposition Table 第 1 层结点存储的深度，将当前第 1 层存储内容移至第 2 层，第 1 层存储写入待置入结点信息

(2) 如果待置入结点的深度小于 Transposition Table 第 1 层结点存储的深度，将待置入结点信息写入第 2 层存储中。

基于原理：首先是深度优先，因为深度越大，一般来说其子树的结点数越多，更容易显示 Transposition Table 的优势，减少搜索结点数；其次是局部性原理，如果当前结点深度小于第 1 层存储单元的深度，则无条件把当前结点信息写入第 2 层存储单元，因为最近访问的内容往往在不远的将来会被再次被访问。

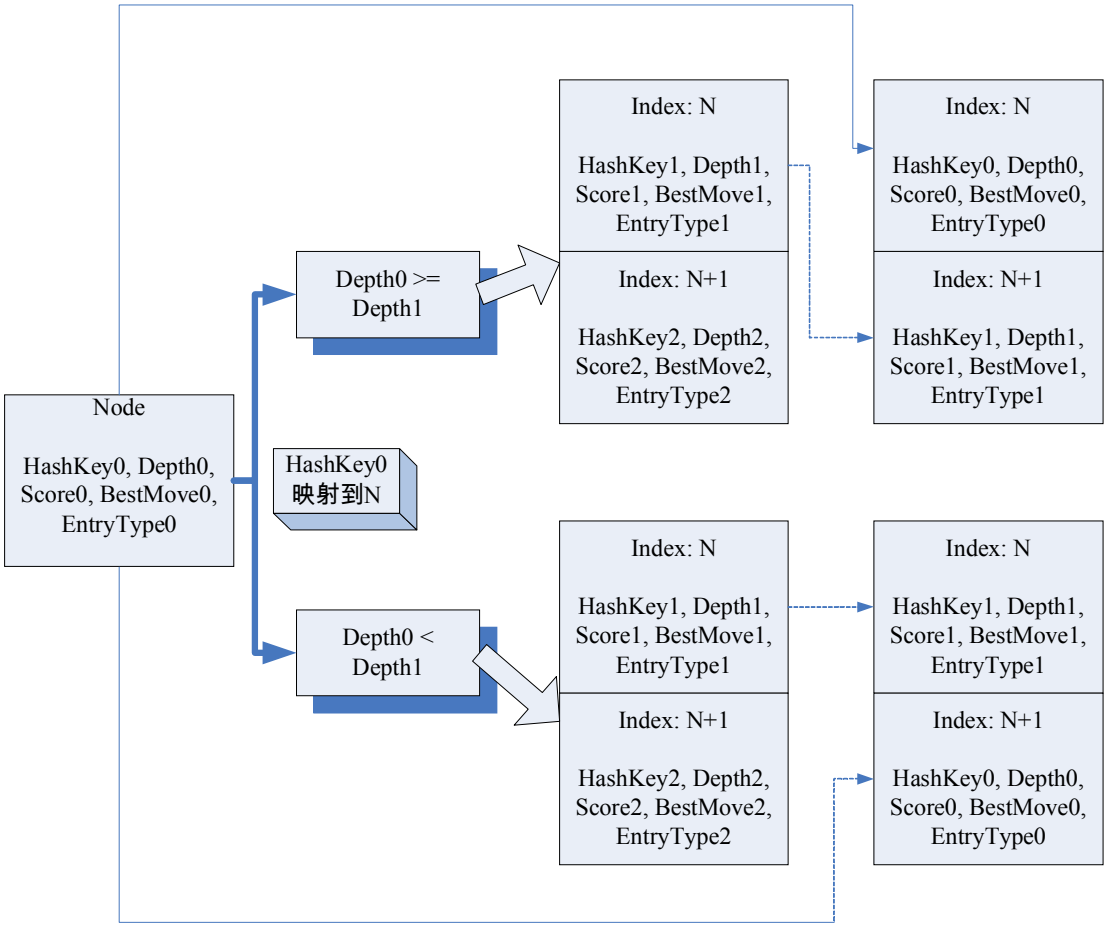


图 3-3 双层 TT 替换算法示意图

3.4.3 Transposition Table 性能测试

使用 24M 的 Transposition Table（2097152 个单元）和不使用 Transposition Table 的搜索结点数比较(测试数据集为 50 个中局盘面)：

表 3-6

	不使用 Transposition Table	使用 Transposition Table	
搜索深度	结点数	结点数	Δ结点数(%)
1	29,115	29,115	0
2	68,601	68,274	-0.48
3	222,557	202,535	-9.00

4	1,175,702	1,031,448	-12.2796
5	4,644,780	3,696,393	-20.42
6	20,325,116	14,089,815	-30.68
7	77,369,221	45,134,350	-41.66
8	328,477,041	164,786,147	-49.83

从表中我们可以看出，Transposition Table 能大大减少搜索的结点数，特别随着搜索深度的增加，Transposition Table 所起的效用增大。

3.5 叠代加深(Iterative Deepening)

Iterative Deepening 是先进行深度为 1 的搜索，在进行深度为 2 的搜索，如此类推，先 N 层搜索，然后 N+1 层搜索，直到搜索分配的时间耗尽。这样的优点是，通过上面的 Transposition Table 的辅助，我们可以记录结点的当前最优孩子，到深度 Depth+1 的时候，就可以先搜索深度 Depth 的最优孩子，因为深度 Depth 的最优孩子往往也是深度 Depth+1 的最优孩子或者是个较优的估计，这样的话，产生剪枝的机会就更大，搜索的节点大大地减少。并且由于每增加一层搜索深度，其以指数时间增加，相对来说，深度为 Depth 的搜索树节点数相对 Depth+1 的搜索树节点数几乎可以忽略。另外，一般比赛都有限时要求，如果直接搜索 N 层的话，有可能没搜索完成而无法找到自己的最佳应步，但通过 Iterative Deepening，至少保证在用时耗尽的时候能得到某一搜索深度的最佳应步值，满足搜索要求。

3.5.1 Iterative Deepening 的性能测试

使用 Iterative Deepening 和不使用 Iterative Deepening 的搜索结点数比较(测试数据集为 50 个中局盘面)：

表 3-7

	不使用 Iterative Deepening	使用 Iterative Deepening	
搜索深度	结点数	结点数	Δ结点数(%)
1	29,115	29,115	0
2	64,361	68,274	6.08
3	210,549	202,535	-3.81
4	1,092,843	1,031,448	-5.62

5	3,964,300	3,696,393	-6.76
6	18,318,398	14,089,815	-23.08
7	61,751,644	45,134,350	-26.91
8	219,059,690	164,786,147	-24.78

3.6 启发函数（Move Ordering）

在搜索 MiniMax 数的时候，要提高搜索的效率，就意味着要尽量使构建的搜索树结点数更小。在宏观上数量的角度来看，这个“更小”是指构建搜索树的时候所访问的结点数目尽量地少，而在微观角度上来看，除了主路径(Principal Variation，也即搜索树的解路径)外，就是要求每个结点访问的时候，在访问少量子结点为根结点的子树后，便发生 Beta 剪枝，为了最大地提高搜索效率，我们在搜索中往往考虑的是访问第一个子结点所在子树后，立刻产生 Beta 剪枝的概率，这个第一孩子Beta剪枝概率同时也可以作为衡量Move Ordering的一个标准。

表 3-8

启发函数效率衡量	
宏观上（从数量上）	微观（从结点搜索的角度看）
搜索树的结点数多少，相同的搜索深度，所需搜索结点数越少越好	从整颗树来看，首孩子结点 Beta 剪枝的概率

在启发式搜索中，通常我们在每个非叶子结点的搜索中，均通过一个启发式函数，计算所有子结点中应该谁先搜索、谁后搜索，在 MiniMax 树搜索我们通常称这种启发式方法为 Move Ordering，意味着子结点扩展的先后顺序。对于 Move Ordering 所含的启发式信息，我们通常包括静态的启发信息和动态的启发信息。

3.6.1 静态启发信息

静态的启发信息：顾名思义，在搜索过程中是静态的，不会因搜索过程的变化而改变，这些信息一般是与研究对象领域相关的信息。譬如在象棋中，主要就是吃子，这是因为在搜索树内部，一个结点所代表盘面如果一般有 42 种走棋步，往往一半步以上会打乱原来子力之间联系保护的平衡，对于这些会丢失保护造成失子的情况，子结点首先通过检测吃子来寻找最好步往往具有优异性，当然一般只有有利的吃子才是有效的，要尽量搜索“有利”的吃子，可采用几种方法，包括：

(1)（被吃棋子的值-吃子的棋子的值）作为排序根据，因为分值小的棋子捕吃分值大的棋子，即使会被回吃，交换中也能获得利益，并且从概率来说，被吃子棋子越大，产生的分值增加的幅度以及概率也越大。其优点是计算耗费少，但缺点是计算欠缺准确性，这是因为分值大的棋子捕吃分值小的棋子的时候，往往会因对方回吃而造成在子力交换中损失利益。
(2) 交换静态评估(Static Exchange Evluation)，通过建立所有棋子对被吃子所在点的攻击或者保护关系，按照最大最小过程，评估交换所带来的利益。其优点是计算准确度较高，但缺点是要计算过程耗费比较大。

3.6.2 动态启发信息

动态启发信息，顾名思义，是动态的，它跟搜索的过程以及当前状态密切相关。在树搜索的过程中，其实也是一个信息积累的过程。利用这些信息进行挖掘，便可以得到我们所需要的启发信息。

定义：好孩子，即最优孩子或者能够产生 Beta Cut 的孩子。

3.6.2.1 Hash Move

若本结点曾搜索过并在 Transposition 记录下以前搜索时的好孩子，往往该好孩子仍然是当前的最优或较优。特别在 Iterative Deepening 中，通过获取该结点第 N 层搜索时的好孩子，作为扩展的第一个结点，往往会得到较满意的 Move Ordering 及剪枝机会。 由于 Hash Move 比较准确，所以一般作为最先扩展的子结点考虑。

3.6.2.2 Killer Heuristics

是指由于相关性，相同深度的邻近节点的最优孩子（或产生 Beta Cut 的孩子）往往也是本结点的最优孩子（或产生 Beta Cut 的孩子）。在 ZMBL 中，为每一层保存 2 个 Killer Move，其中替换方式为 FIFO，但不保存吃子的步（因 Killer Move 在 Capture Move 后搜索）。

3.6.2.3 History Heuristics

是一个从树全局的信息中获取的优先准则，它类似一个全局的 Killer Heuristics，为每个 Move 设置计数器，每当一个 Move 作为最优孩子或者产生 Beta Cut 后，根据当前高度增加相应计数器的值，这是因为高层的最优步，在树中往往具有一定的全局性。

3.6.3 剪枝分析

ZMBL 采用的 Move Ordering，以 Hash Move->Capture Move->Killer Move-> History Move 作为先后顺序，这是因为 Hash Move 由于是上一次搜索的最佳步，所以先搜索，同时对于整棵树来说，移动造成子力之间威胁或失去保护的几率较大，所以放在第二位搜索，同时通过简单的静态交换评价把造成失子的 Caputer Move 放在 Killer Move 候搜索。

各启发特征的产生剪枝百分率比较，测试数据集为 50 个中局盘面
表 3-9

	Hash Cut	Capture Cut	Killer Cut	History Cut
搜索深度	%	%	%	%
1	--	--	--	--
2	3.32898	63.9317	23.3093	9.42997
3	23.0829	55.02	15.0899	6.80715
4	16.849	61.1913	16.159	5.8007
5	25.7065	56.3733	12.9247	4.99554
6	20.264	61.5469	13.3929	4.79623
7	24.0754	59.7553	11.8193	4.34993
8	21.2268	62.664	11.7996	4.30963

3.7 Transposition Table 存储的 Fail Low 修正算法

在结点搜索中出现 Fail Low 情况的时候，因为所有子结点的搜索值都<= Alpha，也即不存在有效的最优孩子，所以存储入 Transposition Table 的时候，不再有最优孩子的存储。但实际 Iterative Deepening 的搜索过程常是一个震荡过程，常常第 N 层最优步为 A，第 N+1 层最优步为 B，第 N+2 层又恢复为 A，这样，如果在 Fail Low 的情况下，仍把之前 Fail High 或者 Exact 情况下的最优孩子保

存，那么如果下一次由于震荡本结点出现非 Fail Low 的话，则会大大提高剪枝的概率（也即提高 Move Ordering）的。由于这种存储不会增加 Transposition Table 负担，且从概率上有 Hash Move 比因 Fail Low 而无 Hash Move 只能靠其他启发特征的情况好。

算法描述如下：

如果当前结点搜索属于 Fail Low，则本次没有 BestMove，若本结点存在于 Transposition Table 中，并且原 BestMove 不为空，则将本次结点搜索结果存入 Transposition Table 的时候，置 BestMove 为上一次搜索的 BestMove。

伪代码如下：

```
ProbeTT();
BestMove = 0;
……..
Recursive Search
……
if ( FailLow 情况出现  && Trasnposition Table 中有存储 BestMove)
    BestMove = OldBestMove; // OldBestMove 为原来 TT 中存储的 BestMove
InsertHash(hashvalue,alpha,TYPE,depth,bestmove);
```

表 3-10

3.7.1 Transposition Table 存储的 Fail Low 修正算法的性能测试

使用 Fail Low 修正算法和不使用 Fail Low 修正的搜索结点数比较(测试数据集为 50 个中局盘面)：

表 3-11

	不使用 Transposition Table 的 Fail Low 修正算法的	使用 Transposition Table 的 Fail Low 修正算法的	
搜索深度	结点数	结点数	Δ结点数(%)
1	29,115	29,115	0
2	68,274	68,274	0
3	208,686	202,535	-2.95
4	1,060,127	1,031,448	-2.71
5	3,910,833	3,696,393	-5.48
6	15,494,351	14,089,815	-9.06
7	49,731,944	45,134,350	-9.24
8	186,986,790	164,786,147	-11.87

可看出，当搜索层次增加到一定程度后，采用 Fail High 修正算法，能有效地减少 10%左右的搜索结点数。

3.8 选择性搜索

在人类下棋的时候，往往根据其自身对棋的认识，只选择几个认为较好的着法进行思考和深层次计算，而跳过其余的譬如会造成失子或造成危险局面的较差着法。这就涉及到对如何延伸较好着法的计算，及忽略较差着法的思考方式。在程序中，由于缺乏甚至不存在绝对准确的评价函数，所以我们无法准确判断那些是好着法，那些是坏着法，只能对着法进行估算，当该着法达到某种标准的时候，对其进行选择性剪枝。同时，我们对一些不稳定或容易产生分数起伏的着法进行延伸，避免水平效应，我们称之为选择性延伸。

3.8.1 选择性剪枝

上面的搜索方法和剪枝技巧，在理论上具有无损性，即不会影响结果的正确性。但在实际应用中，采用对搜索结果正确性影响很少或者较少的选择性剪枝策略，会大大地提高搜索的效率。对于相同的搜索时间来说，通过选择性剪枝增加实际搜索深度，能弥补选择搜索过程带来的不准确性，在实质上提高了搜索结果的优异性。

3.8.2 选择性延伸

一般来说，搜索 N+1 层的结果为什么比搜索 N 层更好，这是因为它考虑的深度更高，减少了犯错误的概率。所以我们可以考虑一个很好的想法，就是选择性延伸，增加一些我们感兴趣的结点的搜索深度，这样就可以提高该结点评价的准确性。当然，我们不能过度地对结点进行延伸，因为这样往往会造成搜索树过于庞大，以至实际上降低了搜索的效率。实际上我们上面所阐述的 Quiescence Search 也是一种选择性延伸，它是在所有搜索树的叶子结点进行吃子延伸。

当前 ZMBL 使用的主要延伸包括：将军延伸，单步着法的延伸。

将军延伸是指当本方受对方将军的时候延伸，由于我们必须逃避对方对本方帅的攻击，所以实际解将的可走着法不多，所以我们可以对当前结点的搜索深度增加一层，以更准确的评估攻击的危险性，减少水平效应。

当本方受对方将军的时候，并且解将步只有一步，这时候由于搜索量不大，我们在将军延伸之外，还对其进行另外的延伸，我们称之为单步着法延伸，通常单步着法延伸的增加深度是 2/3 层（在 ZMBL 中，一层深度实际上用 64 来表示，如果根结点搜索 N 层的话，也就是 Search(N*64)，这样的话我们可以接收小于 1 层的增量，当这些小于 1 层的增量累积起来，超过层数的整数表示值，便可进行额外的延伸，这样的做法，可以使到延伸的同时减少了搜索的负担）。

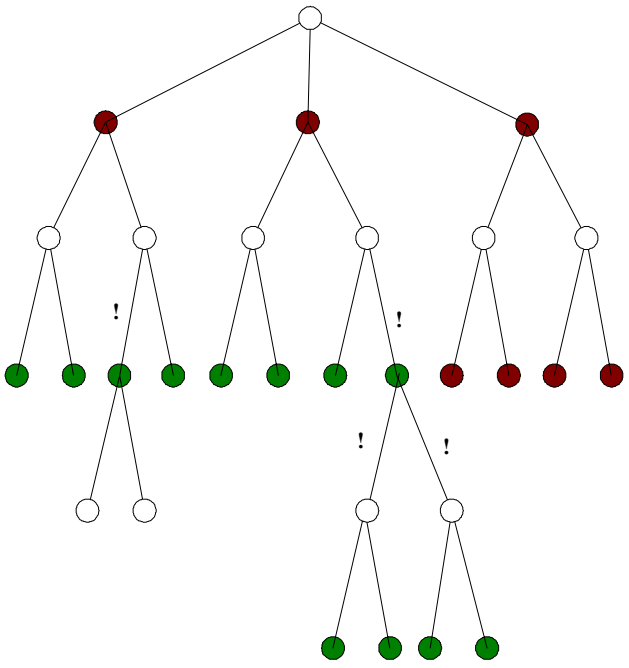


图 3-4 选择性延伸图

3.8.3 Null Move 搜索

Null Move 选择剪枝算法是由 1993 年 Chrilly Donninger 在 ICCA 杂志上首先提出的，它是基于如果本方不走棋，让给对方连续走棋，在减少深度的浅层搜索下如果也能使分值超过 Beta 的话，则进行剪枝。它避免了原先基于静态函数来进行前向剪枝的危险性，并且实现较为简单，现已被几乎所有博弈棋类（除一些 Null Move 没有意义的棋类）所采用。

在实现的时候应该注意的是，搜索不能连续进行 Null Move，应为双方连续不走棋，实际还原到原来的状态，这不在 Null Move 剪枝的意义内。并且在一些情况下，Null Move 需要避免，譬如当子力少，连续多走一步也不一定能获得利益的时候，并且在残局中，甚至出现轮到谁走谁不利的情况出现。

Null Move 搜索的伪代码，在上面 NegaScout 基础上加以修改：

```
int NegaScout(int alpha, int beta, int depth, int bNullMove)
{
    if (game over or depth <= 0)
        return winning score or eval();

    //Null Move
    if ( (bNullMove != 1) && ( condition that should not use Null Move) )
    {
        val = -NegaScout(depth-NULLDEPTH,-beta,-beta+1,1);
        if ( val >= beta )
            return val;
    }

    GenMove();

    for ( i = 0; i < MoveListLength; i++ )
    {
        make move m;
        ……..
        NegaScout 递归搜索 部分
        ……..
        undo make move m;
        if ( val > alpha )
        {
            if ( val >= beta ) //beta cut
                return val;
            alpha = val;
        }
    }
    return alpha;
}
```

表 3-12 NullMove 伪代码

在应用中，Null Move 搜索减少的层数 **NULLDEPTH** 可作为一个变量出现，其根据(1)当前强子数目多少 (2)当前结点搜索深度。强子数目较少时，亦即开始进入残局阶段的时候，可减少 NULLDEPTH 的大小，因为这时候 Null Move 本身实现原理中的一方连走两步棋所带来的优势开始减少，所以可适当减少 NULLDEPTH，降低 Null Move 所带来的不准确性。另外，当结点搜索深度较低的时候，可以适当减少 NULLDEPTH 的大小，这样可以保证搜索树内搜索深度较低的部分结点进行 Null Move 浅层搜索时的深度不会太少，以增强准确性。

3.8.3.1 Null Move 剪枝性能分析

表 3-13

	不使用 Null Move 剪枝算法	使用 Null Move 剪枝算法	
搜索深度		结点数	Δ结点数(%)
1	29,115	29,115	0
2	63,551	68,274	+7.43
3	235,686	202,535	-14.07
4	1,039,410	1,031,448	-0.76
5	5,893,066	3,696,393	-37.28
6	26,384,488	14,089,815	-46.60
7	140,163,627	45,134,350	-67.80
8	662,967,845	164,786,147	-75.14

可看出，随着搜索层数的增加，NullMove 选择性剪枝是一种非常有效的减少搜索树结点数目的方法，并且层数越大，搜索结点数减少的比例越大，当搜索层数达到 8 时，使用 NullMove 搜索的结点数已经仅需要原来的 1/4 了。同时，从上面的数据可以看到，奇数层采用 Null Move 减少的幅度往往大于偶数层，这说明在奇数层搜索（在搜索路径上本方比对方多走一步）时，Null Move 更加有效。

3.9 搜索主体架构

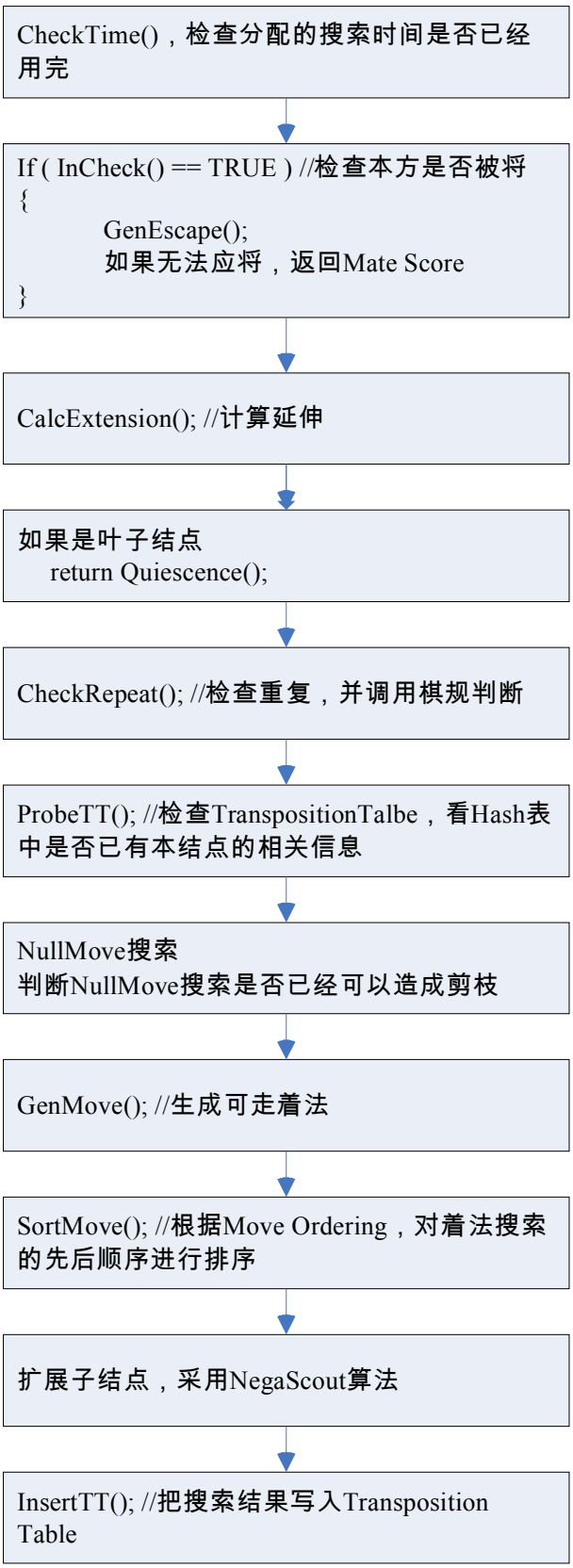


图 3-5

第四章 评价函数

在搜索之外，评价函数是最重要的部分，因为对实际局面的评价的好坏，影响着今后局势发展的趋势。更高的目标结点评价函数值是搜索树所追求的目标，与研究对象变化规律越接近的评价函数，则越能反映未来局势的变化。由于大部分研究对象不存在绝对准确体现最终结果的评价函数，所以使评价函数值尽量能刻划研究对象是评价函数的主要目标，这就需要大量的领域相关的知识。一般来说领域知识常以感性的形式存在，在计算机实现的过程中，则必须将其理性化，也就是说建立研究对象的数学模型。同时在实现这个数学模型的过程中，必须考虑运算的耗费，因为运算量太大的评价计算，往往影响了整个搜索的速度，降低了搜索的深度，而运算量少的一些近似评价，通过搜索深度的增加，仍能大大弥补近似评价所丢失的不准确性。

在数学上，评价函数是对一个盘面定性的静态评价，其计算方法为：

$$EvaluationScore = \sum K_i F_i(P)$$

其中 K_i 为特征系数，$F_i(P)$为特征函数，P 为当前棋盘。

也就是说评价函数分别对当前棋盘的若干个特征进行计算，获得其特征值，所有这些特征值及其特征系数乘积的累加，得出最后的评价函数值。

4.1 象棋的评价函数简介

象棋程序 ZMBL 的评价函数主要有几部分组成：子力价值，子力灵活性，子力对棋盘的控制，和一些对棋局影响比较大的重要特征计算。

4.2 子力分值评价

在象棋程序的评价值中，首先计算的就是双方的子力价值，子力价值在象棋中相当重要，如果一方多子（即子力价值比对方多），这是体现优势的一个主要方面。

$$\text{子力评价值} = \sum \text{本方棋子分值} - \sum \text{对方棋子分值}$$

根据当前象棋理论的研究，一般比较认同的棋子分值是

表 4-1

棋子	车	马	炮	士	相	兵
分值	9	4	4.5	2	2	1

上面的数据只是专业象棋研究者对棋子分值的大致估计，而人在实际对弈中是不会真的精确地计算具体棋子分数值的，他们往往只会通过大致棋子分值判断当前局面优劣和判断子力交换是否恰当。而在程序中实现，则需比较准确地估计了，当前 ZMBL 的子力分值是随着局势变化而变化的，这个局势也就是当前盘面的强子数目，当强子减少的时候，兵卒的作用就原来越大，所以分值也相应的增加，另外，残局的时候马和炮的分数差异也慢慢减少，这是因为残局子力减少，马活动空间会越来越大的缘故。

表 4-2

棋子	车	马	炮	士	相	兵
分值	1050	450	500	180	160	100

4.3 子力灵活性

子力的灵活性主要指棋子可走步数的多少，因为棋子可移动的步越多，那么它能到达满意盘面的几率就越大，并且可逃避攻击的几率也大。

一个子的灵活性分数= 可移动空格数 X 灵活度参数

一般只考虑攻击性强子的灵活性，其中灵活度系数如下：

表 4-3

	灵活度系数
车	7
马	12
炮	1

4.4 棋盘控制

一方对棋盘控制的棋盘区域越多，那么他就越具有主动性。由于棋盘在不同区域的控制具有不同的重要性，所以先要制定一个棋盘控制分数表，也就是越接近对方九宫的控制分数越大，然后累加计算各个主要子力对棋盘的控制分数。以下为棋盘控制分值表：

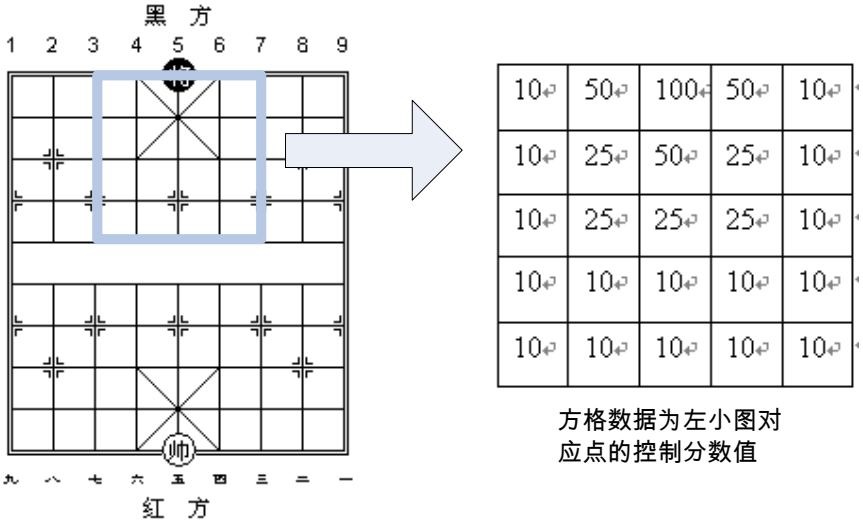


图 4-1

每个强子可采用不同的控制分数表，甚至每个强子的某一方向也可以使用自己的控制分数表

4.5 重要特征计算

除了上面几个基本评价外，评价函数还应该考虑一些影响棋局发展的特征，譬如空头炮、无士相边的沉底炮，还有一些棋子组合的加分以及特定棋子的攻击性价分，这包括马炮组合价分、对方缺士同时本方有双车和有车对无车等的动态加分等。

第五章 结论及展望

5.1 总结

本文论述了象棋程序“纵马奔流”的设计与实现的方法与原理。通过测试数据集验证程序的理论效率值（我们将其称之为技术指标值）提高，并通过网络与专业棋手进行大量的对弈来验证搜索、评价函数的实际效果（我们将其称之为实践指标值）。

在这不断研究理论、应用于实践，最终又由实践反馈于理论的过程中，本象棋程序得到了很大的提高，通过优秀的剪枝算法以及快速的数据结构实现，有效地提高了程序的相对和绝对搜索速度；通过的大量的延伸，有效地提高了搜索的质量；在不影响准确性的基础上，快速的评价函数运算，也大大减少了评价耗费时间，而通过更深层次的搜索来增加博弈树评价的准确性。因此，程序在实践中取得了相当不错的成绩，包括在网络上和专业棋手的对弈以及参加的电脑象棋比赛（8^{th} Computer Olympiad 金牌）上。

5.2 未来的展望

在未来的发展中，将会着重对并行搜索以及评价参数学习修正的研究。通过并行搜索，可充分利用当前的多 CPU 硬件平台，在不改变程序大部分结构的基础上实现速度的倍增，由于对相同的搜索与评价函数来说，程序棋力随速度的增加而增加，而这种增加其实是搜索深度增加与程序棋力的一种相对关系。另外，对于复杂的棋类（譬如中国象棋、国际象棋等），计算机博弈研究界还没有一种很好的参数的学习修正方法，大部分通过学习得出来的结果还比人类手工调整差，这主要因为像象棋这种复杂对象，其规律有很多的特征所组成，并且各种特征交错作用，同时有些重要特征随机出现几率也并不高，在这种情况下，常用的知识方法只能学习出一些较为简单的大参数值，譬如子力价值，而对大部分其它参数学习的效果则不太理想。

当前的电脑象棋发展水平正在迅速地提高着，部分优秀的象棋程序甚至已达

到了大师的水平。但由于象棋搜索树复杂度比国际象棋大，而且投入的研究较少，实际对弈并未能达到特级大师的水平，但随着硬件的不断发展，搜索研究的不断投入，可以相信，在不远的将来象棋程序会具备挑战特级大师的水平。

附录

附录 A 与其他象棋程序的对弈比较

在2003年11月，ZMBL参加了在奥地利格拉兹举行的8th Computer Olympiad比赛，参加电脑象棋的参赛者有5个，除本人之外，包括：

程序	作者
ELP	許舜欽，鄭武堯，陳志昌（台湾大学）
Contemplation	吳光哲，陳志昌，徐讚昇，許舜欽（台湾大学）
XieXie	Pascal Tang (France)
Lock	郑明政，颜士净（台湾东华大学）

比赛每方用时为30分钟，比赛结果如下：

ZMBL vs ELP	ZMBL vs Contemplation
2-0	2-0
ZMBL vs XieXie	ZMBL vs Lock
1.5-0.5	1.5-0.5

ZMBL在和4个对手的8盘分先对弈中，取得了6胜2和的成绩。

附录 B 参加第 8 届 Computer Olympiad 的比赛结果

Standing

Gold ZMBL (Z. Tu)
Silver Xie Xie (P. Tang)
Bronze ELP (J-C. Chen)

Place	Name	Score	Berg.	Wins	
1	ZMBL	7	22.75	6	
2-3	XieXie	5	13.75	4	play-off 1.5-0.5
	ELP	5	11.00	5	
4-5	Contemplation	1.5	6.75	0	
	Lock	1.5	2.25	1	

XieXie and ELP shared second place. In the play-off XieXie defeated ELP with 1.5-0.5.

附录 C 参加第 8 届 Computer Olympaid 的比赛对局选

标题: ZMBL vs XieXie
结果: 和棋

1. 兵七进一 炮 2 平 3	2. 炮二平五 象 3 进 5	3. 马二进三 车 9 进 1
4. 马八进九 车 9 平 4	5. 车九平八 士 4 进 5	6. 车一平二 车 4 进 3
7. 士六进五 马 8 进 9	8. 兵三进一 卒 9 进 1	9. 马三进四 车 4 平 6
10. 炮八进二 马 2 进 4	11. 炮八平九 车 1 平 2	12. 车八进九 马 4 退 2
13. 车二进六 马 2 进 4	14. 马九退七 炮 8 平 7	15. 相七进九 卒 1 进 1
16. 炮九平八 炮 3 平 1	17. 马七进六 炮 1 进 4	18. 炮五平四 车 6 平 2
19. 炮四平八 车 2 平 6	20. 兵五进一 炮 1 平 9	21. 兵五进一 卒 5 进 1
22. 车二退三 卒 9 进 1	23. 前炮进一 卒 5 进 1	24. 兵七进一 车 6 退 3
25. 后炮平四 车 6 平 7	26. 马四退三 卒 7 进 1	27. 马三进一 卒 7 进 1
28. 相三进五 卒 7 平 8	29. 车二平三 卒 9 进 1	30. 马六进五 象 5 进 3
31. 车三平六 马 4 进 2	32. 炮四平三 炮 7 平 8	33. 马五退三 卒 8 平 7
34. 炮三进六 炮 8 进 7	35. 相五退三 马 9 退 7	36. 车六平一 卒 1 进 1
37. 车一平二 炮 8 平 9	38. 炮八退三 象 3 退 5	39. 炮八平一 马 2 进 3
40. 炮一退一 卒 5 进 1	41. 车二退三 炮 9 平 7	42. 车二平三 马 7 进 8
43. 车三进二 卒 5 平 6	44. 炮一进五 卒 7 进 1	45. 车三平六 马 8 退 6
46. 车六进四 卒 1 进 1	47. 相九进七 卒 1 平 2	48. 炮一平四 马 3 进 5
49. 车六平七 马 5 退 6	50. 车七平四 卒 6 平 5	51. 相七退五 卒 5 进 1
52. 帅五平六 卒 2 平 3	53. 车四平五 卒 3 平 4	54. 车五退四 马 6 进 7
55. 车五平八 卒 7 平 6	56. 车八进七 士 5 退 4	57. 车八退七 马 7 进 8
58. 车八进四 卒 6 平 5	59. 车八平二 马 8 进 7	60. 车二平三 马 7 退 5
61. 车三平四 象 5 退 3	62. 帅六进一 象 3 进 5	63. 帅六退一 象 5 退 3
64. 帅六进一 象 7 进 9		

标题: XieXie vs ZMBL
结果: 黑方胜

1. 炮八平五 马 2 进 3	2. 马八进七 车 1 平 2	3. 车九平八 马 8 进 7
4. 马二进一 卒 3 进 1	5. 炮二平三 车 9 平 8	6. 车一平二 炮 2 进 4
7. 车二进六 炮 8 平 9	8. 车二平三 车 8 进 2	9. 车三退二 马 7 进 8
10. 车三平二 马 8 退 6	11. 车二平四 马 6 进 8	12. 炮三进七 士 6 进 5
13. 车四平三 炮 9 进 4	14. 兵七进一 卒 3 进 1	15. 车三平七 马 3 进 4
16. 炮五进四 象 3 进 5	17. 炮三平六 将 5 平 4	18. 车七平六 炮 9 退 2
19. 炮五进二 马 8 进 6	20. 兵五进一 车 8 进 2	21. 兵三进一 将 4 平 5
22. 炮五平九 炮 2 退 3	23. 相七进五 车 2 平 3	24. 车八进二 车 3 进 6
25. 士六进五 马 6 退 5	26. 兵五进一 车 8 平 5	27. 马七退六 炮 9 进 1

28. 兵三进一 马 5 进 7 29. 车六平二 车 3 平 8 30. 车二平四 车 5 进 2
31. 车四进四 炮 9 平 2 32. 车八平九 车 5 平 3 33. 车四平一 前炮进 4
34. 马六进七 马 4 进 2 35. 马七退八 炮 2 进 6 36. 车一进一 将 5 进 1
37. 车九平六 车 3 进 3 38. 士五退六 马 7 进 5 39. 车一退一

标题: ZMBL vs ELP

结果: 红方胜

1. 兵七进一 马 8 进 7 2. 马八进七 卒 7 进 1 3. 马二进三 车 9 进 1
4. 车一进一 象 3 进 5 5. 车一平四 马 7 进 8 6. 炮二进五 炮 2 平 8
7. 车九进一 马 8 进 7 8. 车四进二 卒 7 进 1 9. 车九平四 士 6 进 5
10. 炮八进六 车 9 进 1 11. 前车进五 炮 8 进 2 12. 前车平三 炮 8 平 7
13. 炮八退一 车 9 退 2 14. 车四进五 炮 7 平 5 15. 士四进五 车 1 进 2
16. 炮八退一 象 7 进 9 17. 车三退四 车 9 平 7 18. 车三进五 象 9 退 7
19. 炮八退三 马 7 退 8 20. 车四平五 炮 5 平 7 21. 车五平二 马 8 进 7
22. 马三退一 炮 7 平 9 23. 马一进二 马 7 退 6 24. 车二平四 马 6 进 8
25. 炮八进一 炮 9 平 8 26. 炮八平二 炮 8 进 2 27. 帅五平四 士 5 进 6
28. 车四进一 象 5 退 3 29. 车四进二 将 5 进 1 30. 车四退一 将 5 退 1
31. 马七进六 车 1 平 7 32. 炮二平五 马 2 进 3 33. 车四退二 将 5 进 1
34. 马六进五 车 7 平 5 35. 炮五进三 马 3 进 5 36. 车四平五 象 3 进 5
37. 兵九进一 炮 8 退 1 38. 车五平七 卒 9 进 1

标题: Lock vs ZMBL

结果: 黑方胜

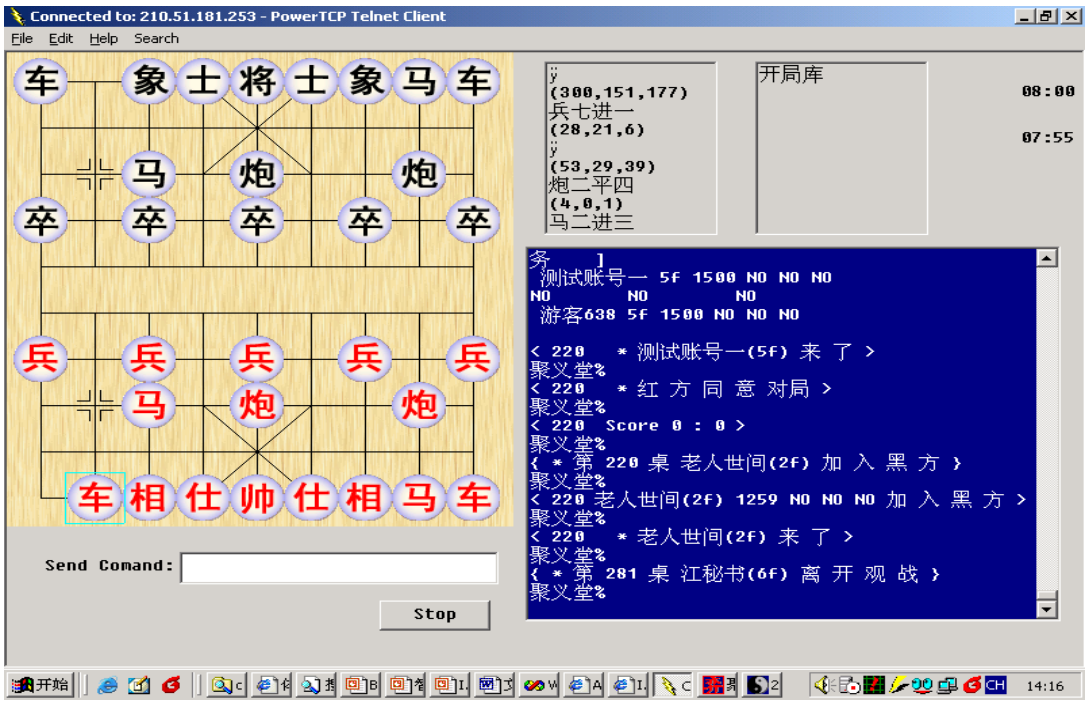
1. 炮二平五 马 8 进 7 2. 马二进三 车 9 平 8 3. 车一平二 马 2 进 3
4. 马八进九 卒 7 进 1 5. 炮八平七 车 1 平 2 6. 车九平八 炮 8 进 4
7. 车八进六 炮 2 平 1 8. 车八平七 车 2 进 2 9. 车七退二 马 3 进 2
10. 车七平八 马 2 退 4 11. 兵九进一 象 7 进 5 12. 车二进一 车 2 进 3
13. 马九进八 马 4 进 2 14. 炮七退一 炮 1 进 3 15. 炮五平八 车 8 进 5
16. 马八进六 车 8 平 4 17. 马六进八 炮 8 退 2 18. 马八进七 将 5 进 1
19. 炮七平八 将 5 平 4 20. 后炮平六 将 4 平 5 21. 炮六平八 将 5 平 4
22. 士四进五 炮 1 退 1 23. 前炮平六 将 4 平 5 24. 炮八平六 车 4 平 2
25. 相三进五 马 2 退 1 26. 马七退六 将 5 退 1 27. 前炮进七 士 6 进 5
28. 前炮退一 马 1 进 3 29. 后炮进一 马 3 退 4 30. 炮六进六 炮 1 平 3
31. 马六进七 将 5 平 4 32. 炮六退六 炮 3 退 2 33. 兵一进一 车 2 退 4
34. 马七退五 象 3 进 5 35. 马三进一 炮 8 退 4 36. 车二进六 炮 8 平 7
37. 兵七进一 车 2 进 3 38. 马一退二 将 4 平 5 39. 车二平一 车 2 平 5
40. 兵五进一 车 5 进 1 41. 兵七进一 马 7 进 6 42. 车一退一 象 5 进 3
43. 马二进四 车 5 进 1 44. 车一进三 炮 7 平 6 45. 兵一进一 马 6 进 4
46. 炮六平八 马 4 进 2 47. 车一退二 炮 3 平 6 48. 车一进二 前炮进 2
49. 兵一平二 前炮平 5 50. 兵二平三 车 5 平 6 51. 车一平四 将 5 平 6

52. 帅五平四 车 6 平 7 53. 兵三平四 炮 5 进 2 54. 兵四进一 卒 5 进 1
55. 相五退三 炮 5 平 6 56. 帅四平五 卒 5 进 1 57. 相三进一 卒 5 平 6
58. 兵四平五 炮 6 平 5 59. 帅五平四 卒 6 进 1 60. 马四进六 卒 6 进 1
61. 士五进四 炮 5 退 1 62. 马六进八 炮 5 平 6 63. 帅四平五 马 2 进 4
64. 帅五进一 车 7 进 2 65. 帅五进一 马 4 退 5 66. 马八进六 炮 6 平 8
67. 士四退五 车 7 退 2 68. 士五退四 炮 8 退 1 69. 炮八进一 车 7 平 2
70. 帅五退一 炮 8 平 5

标题: ZMBL vs Contemplation
结果: 红方胜

1. 兵七进一 炮 2 平 3 2. 炮二平五 卒 3 进 1 3. 炮五进四 卒 3 进 1
4. 相七进九 马 2 进 1 5. 炮八平五 卒 3 进 1 6. 马二进三 车 1 平 2
7. 车一平二 马 8 进 7 8. 前炮退二 炮 8 平 9 9. 车二进六 车 2 进 3
10. 车二退一 车 2 平 3 11. 马八进六 卒 3 进 1 12. 马六进四 卒 3 平 4
13. 马四进三 卒 4 平 5 14. 前马进五 车 3 平 5 15. 相三进五 卒 7 进 1
16. 车二进一 炮 3 进 1 17. 车二进一 象 7 进 5 18. 炮五进二 马 7 进 5
19. 车九平八 车 9 进 1 20. 车八进六 炮 3 退 1 21. 车二退一 马 5 进 3
22. 车二平一 车 9 平 6 23. 相九进七 车 6 进 6 24. 马三退二 车 6 退 3
25. 马五退六 卒 1 进 1 26. 车一平四 车 6 退 1 27. 车八平四 马 1 进 2
28. 车四平九 马 3 退 4 29. 车九退一 马 2 进 3 30. 马二进四 炮 3 退 1
31. 马六进五 马 4 进 5 32. 车九平五 炮 3 平 9 33. 车五进一 前炮平 6
34. 士六进五 炮 9 平 7 35. 车五平六 马 3 进 2 36. 兵九进一 炮 6 进 4
37. 兵一进一 炮 6 退 4 38. 兵九进一 炮 7 平 6 39. 马四进二 前炮平 8
40. 兵三进一 炮 6 平 8 41. 兵三进一

附录 D 以 Telnet 形式在 www.movesky.net 进行自动对弈



参考文献

[1]Heinz,E.A.(1999).Adaptive null-move pruning, ICGA Journal, Vol.22, No.3, pp.123-132

[2] Ernst A. Heinz. (1997). How DarkThought Plays Chess, ICCA Journal 20(3), 166-176

[3] Schaeffer, J. (1983). The history heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16–19.

[4] Schaeffer, J. (1989). The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, No. 11, pp. 1203–1212. ISSN 0162-8828

[5] Plaat, A. , J. Schaeffer, W. Pijls, A. Bruin. . (1996)Best-first Fixed- depth Minimax Algorithms, Artificial Intelligence, Vol. 87, issue 1,2 , 255- 293.

[6] Hyatt, RM and Newborn, M. (1997). Crafty Goes Deep. ICCA Journal, Vol. 20, No. 2, pp. 79-86

[7] Donninger, C. (1993). Null move and deep search: Selective search heuristics for obtuse chess programs. *ICCA Journal*, Vol. 16, No. 3, pp. 137–143.

[8] Zobrist, A.L. (1970). A New Hashing Method with Application for Game Playing. Technical Report 88, Computer Science Department, The University of Wisconsin, Madison, WI, USA. Reprinted (1990) in *ICCA Journal*, Vol.13, No. 2, pp. 69-73.

[9] Knuth, D.E. and Moore, R.W. (1975). An Analysis of Alpha-beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293-326.

[10] D.M. Breuker, J.W.H.M. Uiterwijk and H.J. van den Herik.(1996). Replacement Schemes and Two-Level Tables. ICCA Journal, Vol 19, No.3, pp.175-180

[11] Breuker, D.M., Uiterwijk, J.W.H.M. and Herik, H.J. van den (1994).

Replacement Schemes for Transposition Tables. ICCA Journal, Vol. 17, No. 4, pp. 183-193.

[12] Reinefeld, A. (1983). An improvementto the Scout tree-search algorithm. ICCA journal, Vol.6, No.4, pp.4-14

[13] Zobrist, A.L. (1970). A New Hashing Method with Application for Game Playing. Technical Report 88, Computer Science Department, The University of Wisconsin, Madison, WI, USA. Reprinted (1990) in *ICCA Journal*, Vol.13, No. 2, pp. 69-73.

[14] Akl, S.G. and Newborn, M.M. (1977). The Principal Continuation and the Killer Heuristic. 1977 ACM Annual Conference Proceedings, pp. 466-473. ACM, Seattle.

[15] Beal D.F.(1990).”A Generalized Quiescence Search Algorithm”, Artificial Intelligent Journal. Vol.43,No.1, pp.85-98.

[16] 颜士净，Computer Games, http://www.csie.ndhu.edu.tw/~sjyen/Course/92/AI92/Games.pdf

[17] 黄少龙，《象棋对策论》

致谢

在这里，首先要感谢姜云飞老师在本人研究生学习期间的关怀和学习指引，在他身上，我除了接受知识的灌输之外，还学习到做学问的正气，其严谨、认真的治学态度也是我学习的榜样。他的教诲，使我受益匪浅，终生难忘。

感谢软件所的所长李磊老师、邓克像书记以及秘书小罗，他（她）们在我在软件所学习期间，给予了学习上、生活上等多方面的帮助。

最后，要特别感谢的是我父母，这么多年来，他（她）们在各方面一直默默地支持着我，在我受挫折的时候安慰我、鼓励我，也在我取得成绩的时候分享我的喜悦，这份情是我一直难以偿还的。

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究作出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期： 年 月 日