摘要

计算机博弈是人工智能领域最具有挑战性的科研课题之一。相比于国际象棋，中国象棋的历史悠久，博弈难度水平较高，且棋盘盘面规模更大、着法更为特殊、变化也更加复杂，同时中国象棋也是一种完全知识博弈。一个完备的中国象棋引擎一般包括以下几个组成部分：局面数据结构、局面评价函数、搜索方法、开局库残局库。

本文设计并实现了一个较完整的中国象棋对弈引擎，还介绍了中国象棋引擎应有的基本模块以及搜索方法部分，详细剖析其中的算法原理，通过实验数据分析每个模块的提升点、缺陷等。本文的搜索方法是一种基于博弈树搜索，以 Alpha-Beta 算法为核心的多种剪枝算法的加强组合型算法，主要利用 Alpha-Beta 算法、主要变例搜索、迭代加深法、空着裁剪法来增强引擎的搜索能力。

根据实验结果，本文所设计的引擎能完成所有的第一类残局以及部分的第二类残局，能和人及机器对弈。本文主要针对中国象棋的搜索方法进行研究，评价函数等部分较为欠缺，有待以后提升。

关键词：计算机博弈，中国象棋，博弈树搜索，Alpha-Beta 算法，空着裁剪

Abstract

Computer game is one of the most challenging research topics in the field of artificial intelligence. Compared to chess, Chinese chess has a long history, the difficulty level of the game is higher, and the board is larger, the method is more special, the change is more complicated. Meanwhile, Chinese chess is also a complete knowledge game. A complete Chinese chess engine generally includes the following components: Data structure, Evaluation function, Search function, Start database and End database.

This paper designs and implements a complete Chinese chess game, also introduces the necessary part of the Chinese chess engine: basic module and search method part, details analysis of the principle of the algorithm, through the experimental data analysis of each module of the lifting point, defects, etc. The search method is a kind of enhanced combination algorithm based on game tree search and multiple pruning algorithms with Alpha-Beta algorithm as the core, mainly uses the Alpha-Beta algorithm, Principal Variation Search, iterative deepening method, Null-Move Forward Pruning to enhance the search engine capacity.

According to the experimental results, the engine designed in this paper can complete all the first type of residual and some of the second type of residual, and play with human and machine. This article mainly aims at the Chinese chess search method to carry on the research, the evaluation function and so on part is relatively lack, need to be promoted later.

Keyword: Computer game, Chinese chess, game tree, Alpha-Beta algorithm, Null-Move Forward Pruning

目 录

第一章 绪论

1.1 课题的背景

早在人类文明发展初期，人们就已经开始进行棋类博弈的游戏了。可以说，进行棋类博弈是人类智慧的一种体现。

在人工智能领域，机器博弈一直被认为是最具有挑战性的课题。让计算机拥有博弈的能力，与就意味着计算机拥有了一定的智能。因此，人工智能领域的学者可以在机器博弈这个平台上检验自己的研究成果；对机器博弈进行研究而产生出来的技术也已广泛应用于政治、经济、军事等领域中，并取得了大量引人瞩目的成果。也可以说机器博弈是人工智能的一块实验田。

早在计算机发展的初级阶段，香农（1950）以及图灵（1953）就已经提出了对国际象棋博弈策略及程序的描述。香农指出，计算机博弈的实质就是生成博弈树，并对博弈树进行搜索。他还提出了两种博弈树搜索策略：穷尽搜索所有博弈树节点的 A 策略和有选择地搜索一部分节点的 B 策略。香农也因此成为计算机博弈的创始人。图灵写出了第一个国际象棋计算机博弈的程序。但由于当时的计算机成本高昂，尚未普及，而且运算能力也非常有限，图灵不能将自己的程序放在机器上运行。但是他想出了一个办法，就是把自己当成一个 CPU，并严格按照程序指令的顺序进行操作，平均每走一步就要花费半个小时左右。尽管当时的程序棋力和效率都非常低，但是科学家们这种契而不舍的精神鼓励着人们继续对计算机博弈进行研究。

随着计算机软硬件水平的高速发展，对国际象棋计算机博弈的研究也逐步取得进展，计算机博弈系统水平不断提高。

1957年，伯恩斯坦采用香农的B策略，设计出了一个完整的象棋程序，这个程序在IBM704上运行,从此第一台能进行人机对弈的计算机诞生了。这台机器的运算速度为每秒200步。

1973年，美国西北大学开发出来了CHESS4.0，成为未来程序的基础。1978年，CHESS4.7达到A级(相当于国际象棋一级)水平，1979年的更高版本CHESS 4.9夺得了全美国际象棋计算机大赛冠军，并达到了专家级水平(相当于国际象棋1-3段)。

1983年,肯.汤普森开发了一台专门下国际象棋的机器BELLE，可以搜索到八至九层，达到了精通级水平（相当于国际象棋4-6段）。

1993年,“深思”二代击败了丹麦国家队，并在与世界优秀女棋手小波尔加的对抗中获胜。

1997年，卡内基梅隆大学组成了“深蓝”小组研究开发出“更深的蓝”。这个计算机棋手拥有强大的并行处理能力，可以在每秒钟计算2亿步，并且还存储了百年来世界顶尖棋手的10亿套棋谱，最后“超级深蓝”以3.5比2.5击败了棋王卡斯帕罗夫。成为人机博弈的历史上最轰动的事件。

1.2 中国象棋计算机博弈研究现状

由于文化背景的差异，很少有西方学者对中国象棋计算机博弈问题展开研究，而国内计算机软硬件发展水平的落后，也成为制约中国学者进行研究的不利因素。因此,在国际象棋计算机博弈研究迅猛发展的时侯，对中国象棋计算机博弈的研究却十分滞后。无论是在博弈水平还是在研究规模上，都与国际象棋存在着差距。

对中国象棋计算机博弈的研究是从台湾开始的。当时可供参考的资料非常少,只能 借鉴国际象棋的成功经验。1981年台湾大学的张耀腾发表的硕士论文《人造智慧在电脑 象棋中的应用》，成为第一篇关于中国象棋计算机博弈的文章，虽然只用了残局来做实验,但是他介绍了评估函数的组成部分，并提出了当时的评估函数存在的问题,对以后的研究很有参考价值。

1982年台湾交通大学的廖嘉成在他的硕士论文《利用计算机下象棋之实验》中实现了一个完整的象棋程序，分为开局，中局，残局三部分，其中开局打谱，不超过二十步，中局展开搜索，残局记录杀着，已经具备相当的智能。而这种针对不同的对局阶段分别进行处理的方法也被现在的象棋软件普遍采用。

从1985年开始，台湾大学的许舜钦教授开始了全面的研究工作。在许舜钦1991年的论文《电脑对局的搜寻技巧》中总结了自1944年Von Neumann和Morgenstern提出的极大极小算法到当时为止最新发展的几乎所有算法，在他同年的另外一篇论文《电脑象棋的盲点解析》中从“审局函数偏差”和“搜寻深度不足”两大方面细致地论述了电脑象棋算法的7种误区。这些研究对计算机象棋的发展起了巨大的指导作用。许舜钦教授也因为自己的突出贡献而被称为“中国计算机象棋之父”。

随后对中国象棋计算机博弈的研究逐步展开。到目前为止出现了许多优秀的中国象棋软件，如台湾的许舜钦及其团队的“ELP”、赵明阳的“象棋奇兵″、中山大学涂志坚的“纵马奔流”、东北大学的“棋天大圣”、上海计算机博弈研究所黄晨的“象眼”等。其中东北大学人工智能与机器人研究所还专门聘请了“深蓝之父″许峰雄博士作为“棋天大圣”的顾问，获得过第十一、十二届电脑奥赛金牌和首届全国计算机博弈 锦标赛冠军。“象眼”是应用于“象棋巫师”上的搜索引擎，其创作者黄晨为了方便大家学习交流，公开了源码，并且发布了专门的网站作为计算机象棋知识和技术的交流平台。本文的程序就参考了一些相关技术。

作为中国的传统棋类游戏，中国象棋的空间复杂度比国际象棋高，规则也更为特殊，因而对它的研究也更具有挑战性。因此我们希望在目前中国象棋博弈系统研究成果的基础上，借鉴国际象棋的成功经验，总结出优秀的中国象棋搜索引擎有哪些共同点，尝试 新的研究方向，并且在一般配置的计算机上实现一个具有一定能力的中国象棋博弈系统，为今后中国象棋计算机博弈工作的深入开展提供帮助。

1.3 本文主要工作及难点

本文设计并实现了一个较完整的中国象棋对弈引擎（代码链接参见目录 1），经过测试，此引擎具有一定的棋力，能完成所有的第一类残局挑战及部分的第二类残局挑战，能和人进行对弈，但棋力不强，还有很多提升空间。

本文介绍了中国象棋引擎应有的基本模块，包括局面数据结构、局面评价函数、着法排序三个部分，深入阐述了其中的重要原理、算法流程等，同时对基本模块的不足作出了分析，提出了待改进的地方。

本文的重点是搜索模块部分，介绍了 4 种有效的剪枝搜索方法及它们的有机结合应用，还介绍了克服水平效应的方法。本文的难点除了无逻辑漏洞的基本模块的实现之外，还有对 Alpha-Beta 算法的搜索能力的提升。如何能够在保证搜索结果准确性的情况下，加深引擎搜索深度，减少引擎搜索时间，是本文研究的重点、难点、关键。

1.4 本文的结构安排

本文的主要研究内容是提升博弈树搜索算法的能力，并将其应用到中国象棋引擎上。本文共分为六章，各章节结构安排如下：

第一章，介绍课题的背景及意义、主要研究内容及难点。

第二章，阐述本文设计的引擎中三大基本子模块的基本原理。即局面数据结构、局面评价函数、着法排序的基本原理。

第三章，阐述以 Alpha-Beta 算法为中心的搜索方法的原理，分析增强博弈树搜索能力的方法，并展示测试结果。

第四章，展示本文引擎分别在残局测试以及对弈测试上的结果。

第五章，总结全文的成果，并在此基础上指出下一步工作。

第二章 基本模块

2.1 引言

本文所设计的引擎的基本模块包括局面数据结构、局面评价函数、着法排序三个部分。基本模块为第四章所涉及的搜索方法提供了底层基础支持，搜索方法可以任意调用其中的外函数来局面消环、哈希判重、生成着法及递归搜索等等。基本模块是引擎的第一层，很多函数需要多次重复地被调用，因此，基本模块的效率至关重要，如果函数复杂度过高，会影响引擎的搜索能力。基本模块应当做到前后逻辑通顺、无严重的漏洞、高效率。

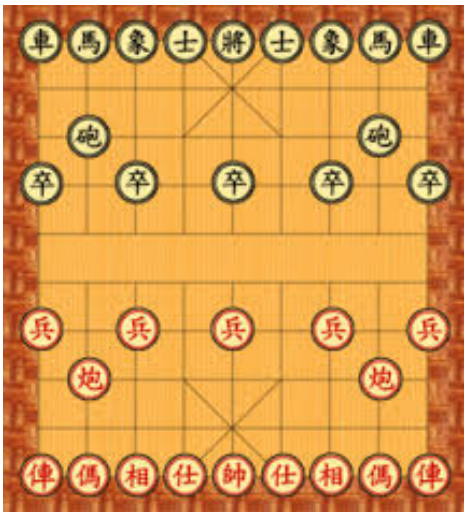
2.2 局面数据结构

局面数据结构部分主要包括棋盘表示、着法生成、哈希函数、置换表、回滚列表五个部分。其中棋盘表示是局面数据结构的核心，它负责描述棋盘局面；着法生成负责走棋，它能够高效的将一个局面转化为另一个局面。此外，哈希函数负责将局面单一的转化为计算机可识别的变量类型；置换表负责有效的存储及提供搜索过程中搜索过的局面。

本文的引擎采用面向对象的方法来表示棋盘，相关代码见附录 2。

2.2.1 局面表示

局面表示的作用是将棋盘数据转化为电脑语言，让计算机能够识别棋盘。因



此，局面表示的关键在于转化方法，转化方法不仅要有逻辑性，还要有效率。

图 2-1 中国象棋开盘局面

设棋盘数组 square 来表示棋盘。中国象棋棋盘有 10 行 9 列，用 square[256]来表示棋盘。将 10x9 的棋盘，转化为 16x16 的大棋盘，这样的设计有诸多好处。图 2-1 为标准的中国象棋开盘局面，其转化后的 square 数组如下表：

表 2-1 square 数组

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	39	37	35	33	32	34	36	38	40	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	41	0	0	0	0	0	42	0	0	0	0	0
0	0	0	43	0	44	0	45	0	46	0	47	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	27	0	28	0	28	0	30	0	31	0	0	0	0
0	0	0	0	25	0	0	0	0	0	26	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	23	21	19	17	16	18	20	22	24	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

如表 2-1 所示，这样的设计有如下好处：

(1) 不需要对棋子走出棋盘进行判断，只需判断 square[p]是否是 0 即可。将原本的“if i≥0 AND i＜10 AND j≥0 AND j＜9”四次判断转化为“if square[p]等于 0”一次判断。

(2) 方便用一个 16 进制整数存储每行每列的棋子位置状态。表 2-1 所示第 4 行的位置状态即为“0001111111111000”，第 4 列的位置状态即为“0001001001001000”。

(3) 设走子方为 p（p 为 0 表示红方，p 为 1 表示黑方），则 16+16*p+x（x 从 0 到 15）表示 p 方的第 x 个棋子

(4) 设棋子为 x，用 x&16 的结果即可判断棋子的颜色。若 x&16 为 16，则 x 为红色；若 x&16 为 0，则 x 为黑色。

(5) 设棋子为 x，用 x&15 的结果即可判断棋子的种类。

2.2.2 着法生成

着法生成的作用是通过合法着子将当前局面生成为另一个局面，搜索方法可以调用其中的外函数来生成吃子着法、非吃子着法、将军着法等。设棋子位置数组 piece，piece 数组大小为 48，piece[x]表示第 x 个棋子所在的位置，piece 与 square 互为逆函数,即 square[piece[x]]=x。与表 2-1 对应的 piece 数组如下表：

表 2-3 piece 数组

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
piece	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
piece	199	198	200	197	201	196	202	195	203	164	170	147	149	151	153	155
x	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
piece	55	54	56	53	57	52	58	51	59	84	90	99	101	103	105	107

除车、炮外，其它棋子归为一类，生成着法的算法如下：

(1) 预生成每个棋子在每个位置能走到的位置集合 s

(2) 枚举起始位置 p1 的集合 s 的每个位置 p2

(3) 若 square[p2]不为 0 且 COLOR square[p1]等于 COLOR square[p2]，则返回(2)

(4) 若 square[p1]为象或者马,设相应的象脚或马脚位置为 p3,若 square[p3]不为 0，则返回(2)

(5) 通过从 p1 到 p2 的着法生成新局面，返回(2)

对于车、炮，一种简单的生成着法算法如下：

(1) 枚举 4 个方向，若枚举结束，则退出

(2) 设起始位置为 p1，往该方向的下一步位置为 p2

(3) 若 p2 超出棋盘，则回到(1)

(4) 若 square[p2]不为 0，则令 n++

(5) 若 square[p1]为车且 n 为 1，则通过从 p1 到 p2 的着法生成新局面

(6) 若 square[p1]为炮且 n 为 2，则通过从 p1 到 p2 的着法生成新局面

(7) 返回(1)

鉴于以上车炮的生成着法算法复杂度较高，本引擎采用行位数组 bitRow 以及列位数组 bitCol 来优化。bitRow[i]及 bitCol[j]分别表示 square 数组的第 i 行及第 j 列的位置状态（见 2.2.1(2)）。

2.2.3 哈希函数及置换表

哈希函数作用是将每个局面单一的无冲突的映射为计算机可识别的类型，方便快速判断重复局面。因此，哈希函数的关键在于无冲突，保证任意两个不同的局面可映射为两个不同的值。

本文采用经典的 zobrist 哈希法，原理是给不同类型棋子在不同位置赋值为随机的 64 位整形数，一个局面的 zobrist 值通过多个整形数相互异或而得到。zobrist 哈希法的好处诸多，尤其是在生成着子时，修改 zobrist 值很方便，只需进行三次异或运算即可。zobrsit 哈希法能在一定程度上保证哈希不冲突，然而，本文采用双重哈希法，再大大降低哈希冲突的可能性。生成两个 zobrist 值 zobrist1 及 zobrist2 来表示一个局面，代价是多花一倍的哈希所需时间。

置换表是博弈树搜索过程中的重要部分，作用是避免重复搜索，即搜索过的局面不搜索，直接返回上次搜索结果。置换表应当有一定的容量（本引擎的置换表大小为 256MB）。对可置换的局面的判断方法应当保证合理性，否则会将不该置换的局面置换，导致搜索出错。

本引擎的置换表含有以下元素：局面深度、局面哈希值、局面着法、局面分值、节点类型。节点类型分为 alpha 节点、beta 节点、pv 节点三种（见 3.2.1 节）。置换表的维护包括初始化、删除、清空、更新、查询五个部分。其中更新置换表在每个节点递归搜索结束时进行（见 3.2.1 节），即获得每个节点最优着法及最优分值时进行。只有当新局面深度比置换表存储局面深度大，或者相同但新局面分值比存储局面分值高时，才会更新置换表。

此节只以置换表的查询函数来阐述置换表的原理，查询函数算法如下：

函数参数：查询局面 s1，搜索窗口[alpha,beta]

函数作用：递归搜索前调用，避免重复搜索

(1) 设通过的 s1 的 zobrist 值得到应在置换表中的位置 p

(2) 设置换表 p 位置所存储的局面为 s2，s2 的局面分值为 vl

(3) 若 s1 和 s2 一致，则到(5)；否则到(9)

(4) 若 s1 的深度不小于 s2，则到(6)；否则到(9)

(5) 若 s2 的类型为 pv 类型，则返回 vl；否则到(7)

(6) 若 s2 的类型为 beta 类型且 vl≥beta，则返回 vl，否则到(8)

(7) 若 s2 的类型为 alpha 类型且 vl≤alpha，则返回 vl；否则到(9)

(8) 返回最低分值

2.3 局面评价函数

2.3.1 引言

评价函数是直接体现引擎下棋智慧的精华部分。评价函数不单单直接影响着引擎的棋力，还影响着搜索方法，准确的评价函数可使得博弈树搜索裁剪地更准确。在第五章中，本文提出了基于评价函数的动态博弈树的新方法，评价函数的好坏直接影响着动态博弈树的优劣。

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

(1) 评价函数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。但是一般情况下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) 衡量的这个性质应该跟对手衡量的性质是一样的（如果引擎认为本方处于优势，那么反过来对方认为自己处于劣势）。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，程序的质量可以通过知识和速度的乘积来估计：

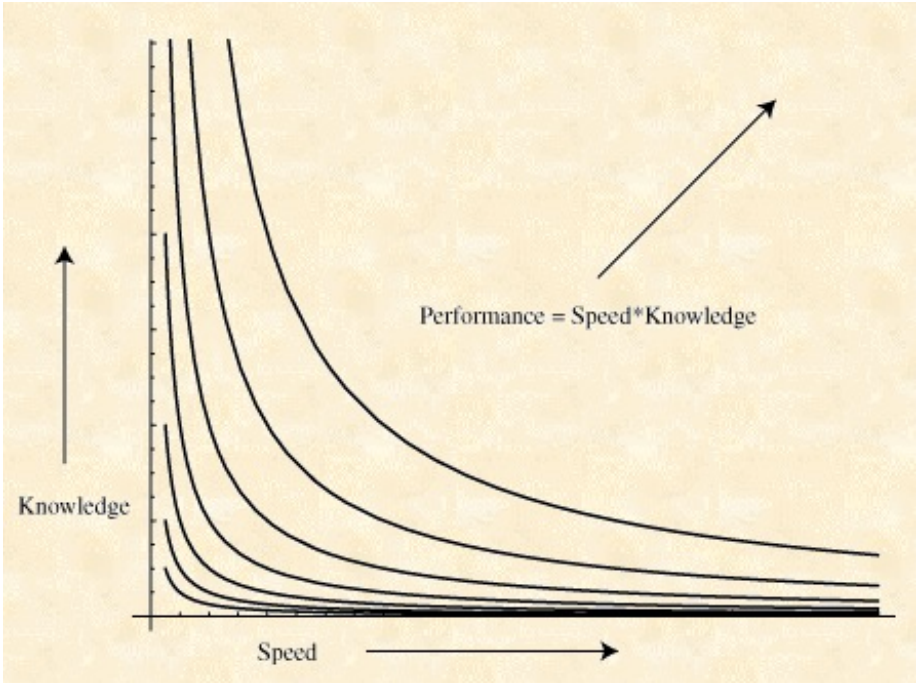


图 2-2 知识与速度的关系图

因此，如果有一个快速而笨拙的程序，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，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，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程序中对于知识的漏洞，从而轻松击败基于知识的程序。

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。

本引擎的评价函数采用中国象棋对弈程序象眼（参考文献[2]）中的方法。主要评价是子力平衡评价，其次是车的灵活性以及马的阻碍。

2.3.2 子力平衡评价

子力平衡评价是通过评估双方每个棋子的权值，从而得到当前局面的分数。评估方法如图 2-3 所示，是通过求出本方权值以及对方权值，将二者相减从而得出当前局面分数。

$$V = V_{本方} - V_{对方}$$

$$V_{本方} = \sum V_{棋子i} + V_{调整}$$

$$V_{棋子i} = \frac{K_{中局} \times V_{中局} + K_{残局} \times V_{残局}}{K_{中局} + K_{残局}} \text{ (帅、马、车、炮、兵)}$$

$$V_{棋子i} = \frac{K_{进攻} \times V_{进攻} + K_{防守} \times V_{防守}}{K_{攻防最大}} \text{ (相、仕)}$$

$$V_{调整} = \frac{K_{象士防守} \times (K_{攻防最大} - K_{防守})}{K_{攻防最大}}$$

$$K_{中局} + K_{残局} = 66$$

$$K_{攻防最大} = 8$$

$$K_{象士防守} = 80$$

图 2-3 子力平衡公式组图

在本节，只阐述本方权值的求法，对方权值的求法类似。本方权值由两个部分组成，其一是所有本方棋子的权值之和，其二是权值调整值。本方棋子的权值要分两类来求，一类是帅、马、车、炮、兵，通过中局和残局的加权权值得到，二类是相、仕，通过进攻和防守的加权权值得到。由图 2-3 可知，要求得当前局面分数，只需求得中局比例值（$K_{中局}$）、进攻比例值（$K_{进攻}$）、防守比例值（$K_{防守}$）即可。而中局权值（$V_{中局}$）、残局权值（$V_{残局}$）、进攻权值（$v_{进攻}$）、防守权值（$v_{防守}$），均是根据棋子位置设定好的。

中局比例值通过计算所有本方棋子带权和而得。帅不计，相、仕、兵计 1 分，车计 6 分，马、炮计 3 分。计算出中局比例值后，残局比例值也随之而得。进攻比例值通过计算本方过河棋子的带权和以及本方与对方马车炮数量差而得。本方过河棋子的权值分配是，马、车计 2 分，炮、兵计 1 分。计算双方马车炮数量差时，车的差值多算一倍。而防守比例值是以同样的方法但计算对方棋子的带权和以及对方与本方车马炮数量差而得。进攻比例值和防守比例值都分别需要与攻防最大值取最小值。不同棋子中局权值、残局权值、进攻权值、防守权值，根据棋子的位置而设定。本节只以马的中局权值和残局权值来表述，如下表 2-4 及表 2-5：

表 2-4 马的中局权值分布

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	96	90	96	90	90	90	0	0	0	0
0	0	0	90	96	103	97	94	97	103	96	90	0	0	0	0
0	0	0	92	98	99	103	99	103	99	98	92	0	0	0	0
0	0	0	93	108	100	107	100	107	100	108	93	0	0	0	0
0	0	0	90	100	99	103	104	103	99	100	90	0	0	0	0
0	0	0	90	98	101	102	103	102	101	98	90	0	0	0	0
0	0	0	92	94	98	95	98	95	98	94	92	0	0	0	0
0	0	0	93	92	94	95	92	95	94	92	93	0	0	0	0
0	0	0	85	90	92	93	78	93	92	90	85	0	0	0	0
0	0	0	88	85	90	88	90	88	90	85	88	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

表 2-5 马的残局权值分布

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	92	94	96	96	96	96	96	94	92	0	0	0	0
0	0	0	94	96	98	98	98	98	98	96	94	0	0	0	0
0	0	0	96	98	100	100	100	100	100	98	96	0	0	0	0
0	0	0	96	98	100	100	100	100	100	98	96	0	0	0	0
0	0	0	96	98	100	100	100	100	100	98	96	0	0	0	0
0	0	0	94	96	98	98	98	98	98	96	94	0	0	0	0
0	0	0	94	96	98	98	98	98	98	96	94	0	0	0	0
0	0	0	92	94	96	96	96	96	96	94	92	0	0	0	0
0	0	0	90	92	94	92	92	92	94	92	90	0	0	0	0
0	0	0	88	90	92	90	90	90	92	90	88	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

2.3.3 特殊棋形及牵制

在中国象棋中，主要的进攻棋子为车、马、炮，而在残局时兵也会变得越来越重要。如果对手缺少一个士或两个士，而我方拥有一个或两个车，那么我方将可能很快的将死对手，即缺士怕双车。而炮在中国象棋中有着至关重要的作用，从大量盘局面分析，很多时候炮能起到对对手很大的牵制效果，比如中炮、沉底炮，尤其当对方的车被炮牵制时，这些都应该考虑到特殊棋形里。除此外，炮与马的配合、马与马的配合、车与炮的配合等等都可以考虑到评价函数中去。

本引擎的特殊棋形是根据士的形状来打分的，士的形状分为 3 种，即开局形、左防形、右防形。若却少一个士且对手拥有双车，要有一定的罚分。若不满足上述三种棋形，则不罚分。而对上述三种情况，分别考虑对手的炮在中炮位置、沉底位置以及对我方车的牵制，来进行打分。

至于牵制，主要指的是车马车、车炮车、车马帅、车炮帅、炮马帅、炮马车这 6 种情况，“车马车”中，第一个车与马及第二个车不同颜色，这样就形成了一个棋子牵制两个棋子的效果，这样的情形应当考虑到评价函数中。本引擎在处理这样清醒时，要保证被牵制的中间棋子不受其它棋子保护。

根据实验而知，特殊棋形和牵制能起到增强搜索能力的作用，原因是其使得不同子局面的分值差异合理的增大，导致搜索剪枝时比以前更快。根据实验发现，本引擎拥有特殊棋形及牵制，可使得博弈树多搜索一层到两层。

2.3.4 车的灵活性及马的阻碍

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。

除子力平衡之外，本引擎评价函数还包括了车的灵活性及马的阻碍，这二者正是考虑区域因素的体现。车的灵活性指车往四个方向能够走到的空格总数量，通过计算本方车的灵活性分值减去对方车的灵活性分值而得。马的阻碍通过枚举马的八个可走到的位置，若位置上无棋子，且位置不受对方棋子保护，且位置不处于棋盘边界，则令 n++。若 n 为 0，则计负 10 分；若 n 为 1，则计负 5 分。马的阻碍也是通过计算本方马的阻碍分值减去对方马的阻碍分值而得。

2.4 着法排序

2.4.1 引言

Alpha-Beta 算法对着法排序非常敏感，所以采用启发式方法对其进行优化是提高算法效率的关键所在。本文引用参考文献[5]中的方法，并结合本引擎的实际情况，提出一种着法排序方案。在该方案中，将使用置换表启发、静态评价启发以及动态启发等技术。

2.4.2 置换表启发

在搜索中，经常会在不同的路径上遇到相同的棋局，这样的子树没有必要重复搜索，把最优着法、分值、深度和节点类型等信息保存在置换表中，再次遇到时直接运用即可。假设对某局面进行 d1 层搜索，窗口是[a, b]，而发现该局面在置换表中已存在，其评价值为 v，类型为 t，深度为 d2，当 d2≥d1 时，如果 t 为精确型，便可直接返回 v 而代替搜索；如果 t 为高出边界型且 v≥b，便可进行剪枝。否则进行重新搜索以保证所取得数据的准确，此时置换表中保存的最佳着法给了我们一定的启发，它为搜索提供一个较优着法，该着法应排在前面优先搜索，这使得总体着法顺序得到了一定程度的优化，置换表的一个主要作用是它对着法顺序的启发。值得一提的是，本引擎在使用置换表启发时，不考虑深度的影响，即只要查询局面在置换表中，则直接调用置换表中的着法。

2.4.3 静态评价启发

静态评价启发主要用来优化吃子着法的顺序。对吃子着法进行静态评价启发，就是要找出表面上占有较大优势的吃子着法。中国象棋的主要进攻手段就是吃子，首先通过检测吃子来寻找最优着法往往会产生较好的效果。如果我们用 V 表示攻击子的价值，用 W 表示被吃子的价值（各个棋子的价值如表 2-4 所示），那么某个吃子着法的价值 U 可表示为：

(1) $V-W$ （被吃子有保护）

(2) W（被吃子无保护）

表 2-4 棋子的交换价值

帅（将）	车	马	炮	仕（士）	相（象）	兵（卒）
5	4	3	3	1	1	2

当吃子着法的值 U＞0 时为表面上能得到获得很大利益的着法，这样的着法往

往是好的着法；U＝0 可能是等价值的换子，这样的着法也值得一试；而 U＜0 的着法往往是吃亏的。因此我们可将吃子着法依据 U 进行排序，并对 U≥0 的着法优先进行搜索。

2.4.4 动态启发

相对来说，对于中国象棋中的某一局面，U≥0 的吃子着法是很少的，大多数着法都是 U＜0 的吃子着法和非吃子着法。对于这些着法仅用之前所述的静态评价启发是不够的，还要进行动态启发。树搜索的过程，实际也是信息积累的过程。对这些信息进行挖掘，可以得到我们所需要的启发信息。

历史启发的思想是：搜索树中某个节点上的好着法，对于其它节点可能也是好的。所谓好着法是指可以引发剪枝的着法，或者其兄弟节点中最好的着法。一经遇到这样的着法，算法就给其历史得分一个相应的增量，使其具有更高的优先搜索权，这个增量通常为$d\times d$（d 为当前节点需要搜索的深度）。具体地说，就是设立一个90×90的数组，红方和黑方的着法都记录在这个数组中，前一个指标代表起始格，后一个指标代表目标格；或者设立一个14×90的数组，红方着法和黑方着法分别记录，前一个指标代表兵种，后一个指标代表目标格。历史启发是从全局信息中获取优先准则的一种方法。对非吃子着法和表面上不能立刻得到实惠的吃子着法根据其历史得分进行排序，就能获得较佳的着法顺序。由于着法的历史得分随搜索而改变，对节点的排列也会随之动态改变。杀手着法实际上是历史启发的特例。它是把同一层中，引发剪枝最多的着法成为杀手，当下次搜索时，搜索到同一层次，如果杀手是合法走步的话，就优先搜索杀手。

2.4.5 结论及测试结果

针对中国象棋中的着法排列问题，本文将置换表着法、静态评价较优的着法和杀手着法排在前面，其余着法按历史得分依次降序排在后面，从而得到了一个较好的着法生成顺序。实验数据表明上述转化在同样的时间内使搜索时访问的平均节点数降低了一个数量级。

本引擎的着法排序流程如下所示：

(1) 查找置换表，若局面在置换表中，则将置换表中的着法排在第一个位置

(2) 生成吃子着法，将吃子着法按静态评价值降序排序

(3) 将杀手排在静态评价≥0 的吃子着法之后，<0 的吃子着法之前

(4) 生成非吃子着法，排在静态评价<0 的吃子着法之后

(5) 将静态评价<0 的吃子着法以及所有非吃子着法按历史得分降序排序

使用启发式方法对着法顺序进行优化可以大大减少搜索的节点数，实验结果如表 2-5 所示（测试数据选自参考文献[4]中有代表性的 10 个中局局面），其中优化前的着法顺序指的是以棋子为主键的排列顺序。从表 2-5 中可以看出随着搜索深度的增加，优化的着法顺序起的效用越来越大，这也验证了剪枝的效率与着法顺序是高度相关的。

表 2-5 不同着法顺序搜索访问的平均节点数对比

搜索深度	优化前的着法排序	优化后的着法排序	降低程度(%)
1	40	40	0
2	670	450	-33
3	5198	2766	-47
4	31408	10882	-65
5	263397	65516	-75
6	1422290	240260	-83
7	11400735	1257174	-88
8	20110101	1976541	-90
9	26099184	2243123	-91

值得指出的是，对杀手的位置进行了实验后，发现将杀手放在静态评价≥0 的吃子着法之后可获得更高的剪枝效率。表 2-6 对比了将杀手排在吃子着法之前与之后搜索时访问的平均节点数（测试数据选自参考文献[4]中有代表性的 10 个中局局面）。从表 2-6 中可以看出，杀手位置的调整对于搜索效率的提升是较为明显的。

表 2-6 杀手排在吃子着法前与之后搜索访问的平均节点数对比

搜索深度	杀手在吃子着法前	杀手在吃子着法后	降低程度(%)
1	40	40	0
2	502	450	-10
3	3400	2766	-18
4	14115	10882	-23
5	87275	65516	-25
6	332417	24260	-28
7	1798713	1257174	-30
8	2534636	1976541	-33
9	2901231	2243123	-35

2.5 本章小结

本章介绍了本文引擎中的基本模块，主要包含局面数据结构、局面评价函数、着法排序三个部分。这三个模块是搜索方法的底层、基础，根基，要保证逻辑及实现不出错，且具有一定的效率。

局面数据结构部分具有一般中国象棋引擎所必需具有的功能，即局面表示、按不同需求生成着法、哈希函数、置换表等等，且保证各函数的复杂度最优或相对最优。然而，与参考文献[2]中的象眼引擎做对比，本引擎的局面数据结构仍有许多地方需要改进、优化。在局面判环的问题上，由于官方规则描述的不精确，不能保证完全符合中国象棋规则，但能够解决绝大部分的判环问题。

评价函数部分包括子力平衡评价、特殊棋形、牵制、车的灵活性、马的阻碍五个部分，具有一定的评价能力，但知识面较为欠缺，体现出了较爱吃子、忽视残局兵及兵过河的重要性等弱点，且评价均为静态型评价，没有运用自学习型算法来优化评价函数。

着法排序部分采用分吃子着法、非吃子着法的方式排序，将置换表着法排第一，吃子着法按静态评价值排序，并在中间插入杀手着法，再将部分吃子着法及所有非吃子着法按历史得分排序。经过实验证明，着法排序部分具有使搜索时访问的平均节点数降低一个数量级的能力。

总的来说，本引擎的基本模块保证逻辑及实现不出错，且具有一定的效率，为搜索方法提供了根基，但仍有一些地方不足，需要改进。

第三章 搜索方法

3.1 引言

中国象棋是有完整信息的、确定的、轮流行动的、两个游戏者的零和游戏。参与博弈的双方是对抗性的双方，搜索的路径不仅取决于一方的意愿，同时还要参考对方采用的应对措施，由此而产生的搜索树通常称为博弈树。中国象棋的博弈树可以看成依此把计算机和人类棋手所有可行的着法和局面列举出来而就构成的一棵树。图 3-1 所示是一棵分支为 3 的博弈树。

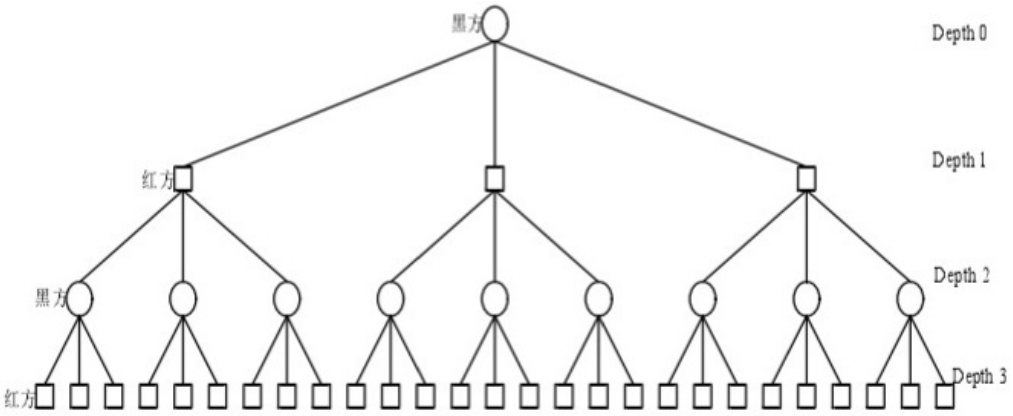


图 3-1 博弈树

图中每个节点都代表一个局面，方节点代表红方要走的局面，圆节点代表黑方要走的局面；树枝代表着法。其中根节点表示初始局面，叶节点代表最终局面（胜、负、和），中间层的节点表示对弈过程中可能出现的合理局面。叶节点到根节点之间的最大的层数，称为搜索深度。图中的博弈树深度为 3。

博弈树搜索模块的任务，就是从初始局面开始有层次地进行着法生成，找到从根节点到分出胜负的叶节点之间的最佳路径，并返回这条最佳路径的第一步着法。如果可以建立完全博弈树，也就是电脑总能找到一条取胜的路径，那么中国象棋的问题也就得到了解决。但是事实上，这是不可能的。

解决的方法就是只把博弈树展开到一定深度，在未到达指定的搜索深度之前，持续不断地针对特定的局面生成所有合理的着法，直到指定的搜索深度为止，然后对这一深度下所有的叶节点进行局面评估，从而确定哪个局面是最佳的，并存储下最佳的着法。博弈树展开的深度越深，就越接近最终局面，得到的结果也就越精确，为了能找到对自己尽量有利的着法，就要求在一定的搜索时间之内搜索的深度尽可能深。由此可见，博弈树搜索是连结着法生成和局面评估的纽带，是着法生成器中的核心。下面对博弈树的搜索原理进行研究。

3.2 剪枝算法

如何对博弈树进行正确的裁剪是本章的难题，这里的“正确”包含了两个方面，一是保证搜索结果的正确性，即保证搜索不出错，不在裁剪的过程中降低了引擎的棋力，二是提升搜索算法的搜索能力，主要表现在提升引擎搜索深度方面。

本节介绍引擎包含的 4 个主要剪枝算法，包括 Alpha-Beta 算法、主要变例搜索、空着裁剪法、迭代加深法。

3.2.1 Alpha-Beta 算法

Alpha-Beta 同最小最大算法非常相似，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 35x35 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。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。Alpha-Beta 算法正是对最小最大算法的改进。

现在搜索算法的瓶颈是要搜索更多层数使得计算机看到的局面尽可能的接近对弈的终局和所花费时间之间的矛盾。根据实际情况有些节点和这个节点下的分枝是不需要做处理的。如图 3-2 所示。

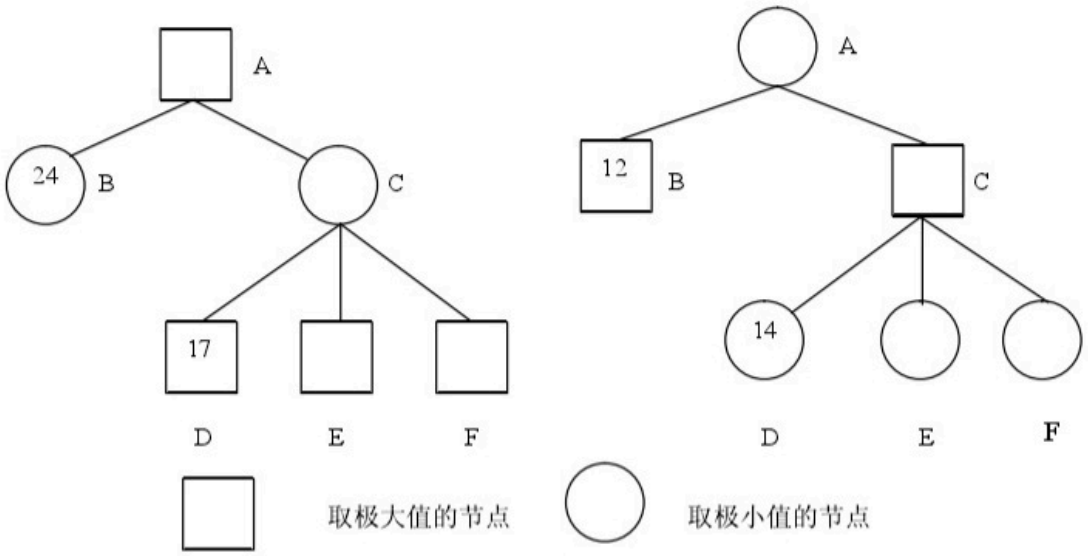


图 3-2 Alpha-Beta 算法

如图 3-2 所示的情况，B 节点的评估值为 24，C 的子节点 D 的评估值为 17，

由于C是取极小值的节点，故该节点的评估值计算方式：valueof(C)=min(valueof(D), valueof(E), valueof(F))，因为D节点的评估值是17，所以C节点的评估值不会大于17，再看A节点，A节点属于取极大值的节点，其评估值的计算方式为：valueof(A)=max(valueof(A), valueof(B))，因为D节点的评估值为17，valueof(B)≤17，无论D节点兄弟节点的评估值是多少C节点都不会被A选中作为扩展的分枝，因此对于节点D兄弟节点的分析是没有必要的，应该将它们剪掉以节省计算机的开销，诸如这样的情况被叫做alpha剪枝。如图3-2所示的情况，B节点的评估值为12，C的子节点D的评估值为14，由于C是取极大值的节点，故该节点的评估值计算方式为：valueof(C)=max(valueof(D), valueof(E), valueof(F))，因为D节点的评估值是14所以C节点的评估值不会小于14，再看A节点，A节点属于取极小值的节点，其评估值的计算方式为：valueof(A)=min(valueof(A), valueof(B))，因为D节点的评估值为14，valueof(B)≥14，无论D节点的兄弟节点的是多少C节点都不会被A选中作为扩展的分枝，因此对于节点D的兄弟节点的分析也是没有必要的，应该将它们剪掉以节省计算机的开销，诸如这样的情况被叫做beta剪枝。

Alpha-Beta算法的伪代码如下：

```
int AlphaBeta ( int depth, int alpha, int beta )
      bestv = -10000 //最优分值，初始化为负无穷
      if ( depth <= 0 ) //叶子节点
            return evaluation () //调用估值函数，返回估值
      for ( each possibly move m ) //对每一种可能的行动进行探测
            MakeMove ( m ) //虚拟执行走法，修改棋局状态
            value = - AlphaBeta ( depth-1, -beta, -alpha); //递归搜索
            UnMakeMove ( m ) //撤销走法回滚到上一个棋局状态
            if ( value >= alpha )
                  alpha = value //取最大值
            if ( value >= beta )
                  bestv = value
                  break // 剪枝
       return bestv
```

经过几十年的发展出现了很多的改进的博弈树搜索算法，但是到目前为止Alpha-Beta剪枝算法仍是使用最广泛的博弈树搜索算法。Alpha-Beta算法是博弈树搜索的基础技术。

3.2.2 主要变例搜索

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。

有些时候可以很早地判断出要处理的是哪类结点。如果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)，假设的着法顺序非常好，那么有可能得到 Alpha 结点。如果返回值在 Alpha 和 Beta 之间，可能得到 PV 结点。

当然，有两种情况可能会判断错误。当高出边界时，返回 Beta，因此不会犯错误，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，如果在搜索一个结点时找到一个 PV 着法，那么就得到 PV 结点。也就是说假设的着法排序已经足够好了，使得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)。

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。

增加了主要变例搜索后，伪代码如下：

```
int AlphaBeta ( int depth, int alpha, int beta )
      bestv = -10000
      if ( depth <= 0 )
            return evaluation ()
      for ( each possibly move m )
            MakeMove ( m )
            value = - AlphaBeta ( depth-1, -alpha-1, -alpha )
            if ( value > alpha || value < beta )
                  value = - AlphaBeta ( depth-1, -beta, -alpha )
            UnMakeMove ( m )
            if ( value >= alpha )
                  alpha = value
            if ( value >= beta )
                  bestv = value
                  break
      return bestv
```

算法的核心部分就是函数中间“if (value > alpha || value < beta)”的部分，不是用常规的窗口(alpha, beta)，而是用(beta-1, beta)来搜索。若返回的 value 值不大于 alpha 或者不小于 beta，则不需要再搜索，否则要重新进行一次正常的 Alpha-Beta 搜索。虽然看上去似乎要多花时间，不值得，但实际不然。首先，窗口由(alpha, beta)变为(beta-1, beta)，相当于令 alpha 为 beta-1，提高了 alpha，增加了剪枝率，其次，在主要变例搜索中，会进行空着裁剪，这将是提升 Alpha-Beta 搜索能力的关键。

3.2.3 空着裁剪

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，使得博弈树的分枝因子锐减，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。

试想博弈树树中的某个局面，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,就会马上返回Beta。如果任何一个超过Alpha，但是没有超过 Beta，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，就要返回 Alpha。

空着向前裁剪是搜索任何着法之前要做的事。好比问一个问题：“如果我方在这里什么都不做，对手能做什么？”记得在刚才，我方没有问这个问题。只是去找最佳的着法去打击对手。问对手是否会打击我方，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我方。比如说我方送了一个车，而其他棋子都没有作用，在这种情况下，对手随便走哪步我方都吃亏，因为我方丢了一个车。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。在搜索着法以前，先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，那么简单地返回 Beta 而不需要搜索任何着法。

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，因此跟用深度 D 搜索所有的着法相比，现在是先以 D-R 搜索对手的着法。一个比较好 R 是 2，如果要对所有的着法搜索 6 层，最终只对对手所有的着法搜索了 4 层。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。

增加了空着裁剪后，伪代码如下：

```
int AlphaBeta ( int depth, int alpha, int beta, bool bNoNull = 0 )
      bestv = -10000
      if ( depth <= 0 )
            return evaluation ()
      if ( !bNoNull && !checked && evaluation() > 200 )
            NullMove ()
            value = - AlphaBeta ( depth - 3, -beta, -alpha, true )
            UndoNullMove ()
            if ( value >= beta )
                  if ( AlphaBeta ( depth - 2, alpha, beta, true ) >= beta )
                        return value
```

```
    for ( each possibly move m )
          MakeMove ( m )
          value = - AlphaBeta ( depth-1, -alpha-1, -alpha )
          if ( value > alpha || value < beta )
                value = - AlphaBeta ( depth-1, -beta, -alpha )
          UnMakeMove ( m )
          if ( value >= alpha )
                alpha = value
          if ( value >= beta )
                bestv = value
                break
    return bestv
```

上述代码中的“NullMove()”为走一步空着裁剪着法，即不移动棋子，只改变走子方，正是上文中所提的“如果我方在这里什么都不做，对手能做什么？”。但注意不是任何情况下都应该采用空着裁剪，只有当认为我方局面比较优时，才会采用，本引擎判定如果局面分数超过 200，则认为局面较优。

3.2.4 迭代加深

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，直到用完时间为止。这足以保证很好地运用时间了。如果可以很快搜索到一个深度，那么在接下来的时间可以搜索得更深，或许可以完成。如果局面比想象的复杂，那么不必搜索得太深，但是至少有合理的着法可以走了，因为不太可能连 1 层搜索也完不成。这个思想称为“迭代加深”(Iterative Deepening)。

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。例如，一层的搜索显示“e4a5”是最好的着法，那么在做两层的搜索时先搜索“e4a5”。如果返回“e4a5 e5b6”，那么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，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。

虽然看起来进行了很多层重复的搜索，但实际上，每一层的搜索都为下一层做了很大的贡献，2.4 节中提到，将置换表着法放到第一位，而正是因为迭代加深搜索，在搜索第 d 层后，再搜索 d+1 层时，才能有效的将置换表着法放到第一位。根据实验说明，迭代加深能很有效的提高搜索效率，且不需要去考虑搜索深度的问题，只需要根据需求控制搜索时间即可。

在上述基础上，可以得到一个“迭代加深启发”，将其运用于 Alpha-Beta 算法中，主要思想如下，在进行生成着法前，如果发现不能在置换表中找到当前局面，则不妨进行一次减少深度的搜索，以获得置换表局面。

增加“迭代加深启发”后，伪代码如下：

```
int AlphaBeta ( int depth, int alpha, int beta, bool bNoNull = 0 )
    bestv = -10000
    if ( depth <= 0 )
        return evaluation ()
    mv = GetMoveFromHashTable ()
    if ( mv == 0 && depth > 2 )
        AlphaBeta ( depth - 2, alpha, beta )
    if ( !bNoNull && !checked && evaluation() > 200 )
        NullMove ()
        value = - AlphaBeta ( depth - 3, -beta, -alpha, true )
        UndoNullMove ()
        if ( value >= beta )
            if ( AlphaBeta ( depth - 2, alpha, beta, true ) >= beta )
                return value
    for ( each possibly move m )
        MakeMove ( m )
        value = - AlphaBeta ( depth-1, -alpha-1, -alpha )
        if ( value > alpha || value < beta )
            value = - AlphaBeta ( depth-1, -beta, -alpha )
        UnMakeMove ( m )
        if ( value >= alpha )
            alpha = value
        if ( value >= beta )
            bestv = value
            break
    return bestv
```

进行减少 2 层深度的搜索，平均只会占用当前局面搜索所需的 1/36 的时间，并且能为搜索提供置换表着法。根据实验可知，有效的置换表着法加上迭代加深搜索能很大程度上提高搜索效率。

3.3 克服水平效应

当搜索节点深度为0时，它是一个叶子节点。搜索进行到叶子节点时进行评估，并返回评估值。但有时到叶子节点是一个吃子着法或将军着法，这可能得到一个很好的评分，但如果是一个换子或解将，可能局面又是一个平手。在叶子节点，局面可能产生剧烈动荡，除非评估函数能非常精确的反映这一点，否则返回值不能很好的反映局面真是情况。这种现象被称为“水平效应”。

3.3.1 静态搜索

中国象棋中会有很多强制的应对，如果对方用马吃掉我方的象，那么我方最好吃还对方的马。Alpha-Beta搜索不是特别针对这种情况的。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。

伪代码如下：

```
int Quies ( int alpha, int beta )
      val = evaluation ()
      if ( val >= beta )
            return beta
      if ( val > alpha )
            alpha = val
      GenerateGoodCaptures ()
      while ( CapturesLeft() )
            MakeMove()
            val = - Quies ( -beta, -alpha )
            UndomakeMove ()
            if (val >= beta)
                  return beta
            if (val > alpha)
                  alpha = val
      return alpha
```

这段代码看上去和Alpha-Beta的非常相似，但是有明显区别的。此函数调用静态评价，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。此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。值得一提的是，上段代码中涉及到生成好的吃子着法一步，这里“好的吃子着法”指的是 2.4 节中静态评价大于 0 的着法，譬如“吃有被保护的象”、“车吃被保护的马”等明显无意义的着法不会在静态搜索中出现。

3.3.2 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。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，通知搜索继续进行下去。如果深度达到一个负得很大的数，那么评价函数总是成功的，这样搜索将会终止。根据静态搜索的启发，可以意识到，不仅可以在叶子节点进行适当的延伸，还可以在非叶子节点进行延伸。需要思考的时，什么情况下需要进行延伸，本文设计的选择性延伸在当前局面被将军或者着法是唯一的时候进行。

将 3.2.1 节中的伪代码改进，如下：

```
int AlphaBeta ( int depth, int alpha, int beta )
      bestv = -10000
      if ( depth <= 0 || distance >= 60 )
            return evaluation ()
      for ( each possibly move m )
            MakeMove ( m )
            newdepth = ( checked || onlymove ) ? depth : depth - 1
            value = - AlphaBeta ( newdepth, -beta, -alpha);
            UnMakeMove ( m )
            if ( value >= alpha )
                  alpha = value //取最大值
            if ( value >= beta )
                  bestv = value
                  break // 剪枝
      return bestv
```

根据实验可知，这样的选择性延伸在处理连续将军或唯一着法的情况下非常有效，可以达到将第四章中提到的第一类及第二类残局统一起来处理的效果。值得一提的是，为防止延伸过度，搜索膨胀，最好控制下延伸深度。

3.4 结果测试

本节选用参考文献[4]中 10 个有代表性的中局局面，限制引擎总搜索时长为 20 秒，测试结果如下：

表 3-1 引擎搜索能力测试

局面编号	搜索深度	返回着法	期待着法	期待-发回分差
78	9	c0e2	f5b5	0
79	8	f5f8	f5b5	0
80	7	i3i4	c1b1	-1
81	8	c3g3	c5d5	-8
82	9	e4e5	e4e5	0
83	9	e2e6	e2e6	0
84	8	f0g0	b9b5	-4
86	9	f4e6	f4e6	0
87	13	d5e7	a6d6	-21
88	7	a2b2	h6h9	-19

从表 3-1 中可知，引擎最少搜索了 7 层，最多搜索了 13 层，平均搜索了 8.7 层。其中 3 个局面返回着法完全等于期待着法，5 个局面的期待着法与返回着法的分值差为 0，还有 5 个局面的期待着法与返回着法存在一定分之差，其中编号为 87、88 局面的分之差较大。导致分之差的原因是评价函数评估不准确，亦或者返回着法与期待着法等效。

3.5 本章小结

本章重点介绍了引擎所含有的 4 个主要的剪枝搜索的基本原理、算法过程，通过逐层介绍的方式显示如何将它们有机结合起来。同时还介绍了克服水平效应的方法，使得搜索评价更为精准。最后展示引擎搜索能力的测试结果。

本引擎的搜索算法是以 Alpha-Beta 为核心的基于博弈树搜索的算法，因此，评价函数越能准确的区分不同局面的差异，搜索能力越强。然而本引擎的评价函数较为缺乏，这在一定程度上阻碍了引擎搜索能力的提升。

在 4 个主要的剪枝搜索中，主要变例搜索是通过修改[alpha,beta]区间窗口来剪枝的，迭代加深与置换表启发结合，空着裁剪采用新思路大幅度剪枝。而基于 Alpha-Beta 算法还可以有很多剪枝方法待发现或待实现，从而可使得引擎搜索能力更上一层。

第四章 引擎棋力的测试结果

4.1 引言

本文采用参考文献[1]中提供的对弈平台——象棋巫师软件。此软件上不仅可以提供机器与机器的对弈，还包含了数百种残局闯关挑战，其中软件上的趣味象棋模块有 240 个残局，局面均具有一定的代表性，很合适作为测试数据使用。此软件还提供了参考文献[2]中所提到的象眼引擎，可以根据不同难度进行棋力测试，还可限定引擎搜索时间、深度来测试。

在残局挑战中，可以根据局面来划分为两类残局。第一类残局的特点是，红方只需经过不断将军即可完成挑战，而第二类残局挑战需要红方多考虑非将军的走法，总的来说，第一类残局比第二类残局容易，需要搜索的节点少。本文分为两类残局来测试，引擎需在每一步 20 秒内落子。

除了残局挑战外，本文还测试了与象眼引擎对弈的结果，分别根据软件上划分好的难易程度测试。每种难度测试三局，并且限定我方每一步 20 秒内落子。

4.2 第一类残局

第一类残局比较容易，红方只需不断将军即可完成挑战。举如下例子：

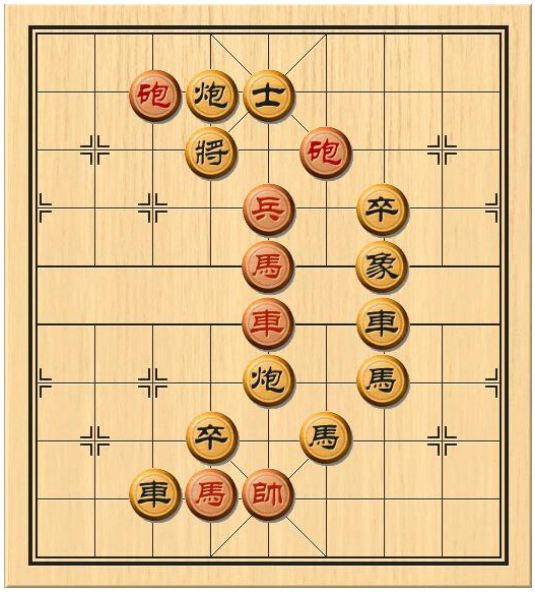


图 4-1 第 001 局 月上柳梢

本引擎执红方，象眼执黑方，测试结果如下：

兵五平六，将 4 平 5，兵六进一，士 5 进 4，马五进三，将 5 平 6，车五进三，

象 7 退 5，马三退五。将死！

再举如下例子：

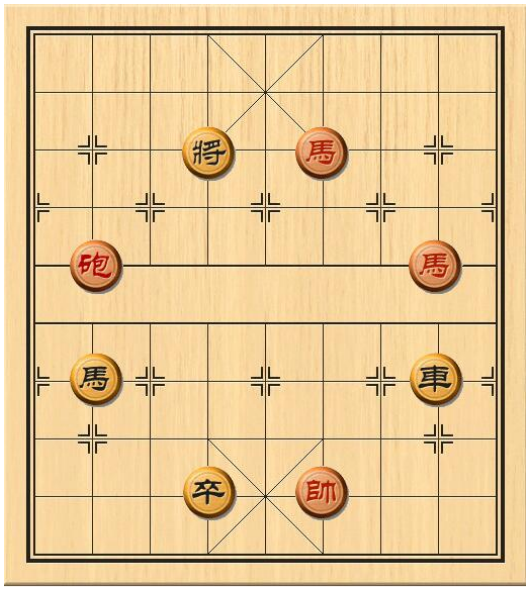


图 4-2 第 006 局 八公德水

本引擎执红方，象眼执黑方，测试结果如下：

马二进四，将 4 进 5，后马退六，将 5 平 4，马六进八，将 4 平 5，马八进七，将 5 平 4，马四退五，将 4 退 1，马五进七，将 4 退 1，前马退 5，将 4 平 5，炮八平五。将死！

在软件上选取 10 局第一类残局，测试结果如下：

表 4-1 第一类残局测试结果

编号	名称	总步数	结果
001	月上柳梢	9	获胜
006	八公德水	15	获胜
024	雀起柳萌	25	获胜
031	沾花惹草	17	获胜
045	釜底抽薪	11	获胜
055	浑水摸鱼	15	获胜
080	亦步亦趋	29	获胜
139	花心轻折	13	获胜
210	尽忠报国	19	获胜
231	移花接木	21	获胜

4.3 第二类残局

第二类残局相对较难，原因是需要引擎多考虑许多非将军着法，这样搜索树变大，搜索层数变低，也考验引擎搜索能力。举如下例子：

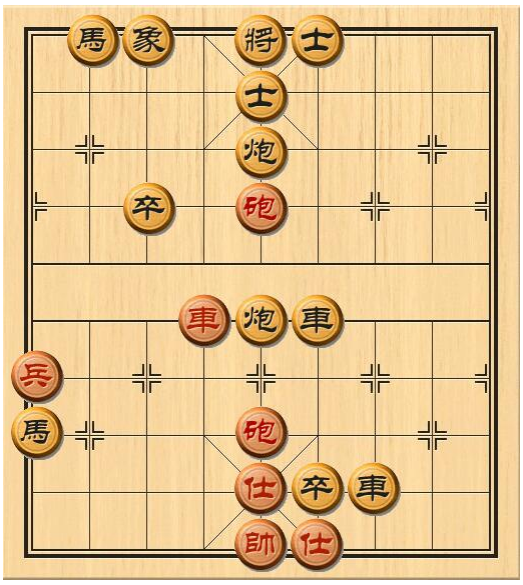


图 4-3 第 010 局 御驾亲征

本引擎执红方，象眼执黑方，测试结果如下：

帅五平六，马 2 进 3，后炮平八，象 3 进 1，炮八平七，马 3 进 5，车六进五。将死！

再举如下例子：

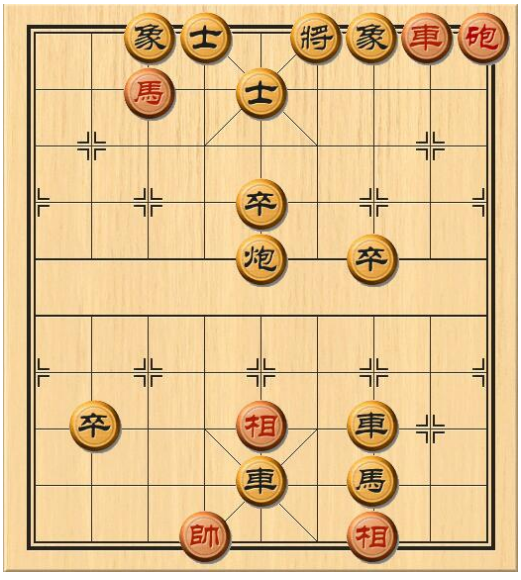


图 4-4 第 029 局 渴骥饮泉

本引擎执红方，象眼执黑方，测试结果如下：

车二退七，将 6 进 1，车二平三，车 5 进 1，帅六进一，卒 2 进 1，红方认输！

象眼执红方又执黑方，测试结果如下：

马七退五，车 5 进 1，帅六进一，车 5 平 4，帅六退一，马 7 退 5，相三进五，车 7 进 2，相五退三，象 3 进 1，车二平三，将 6 进 1，马五退三，将 6 进 1，车三退二，将 6 退 1，车三平二，将 6 退 1，车二进二，将死！

对比来看，本局关键在第一步马七退五，而本引擎没有着此步的主要原因是搜索深度不够，根本原因是引擎的搜索能力不够。

在软件上选取 10 局第二类残局，测试结果如下：

表 4-2 第二类残局测试结果

编号	名称	总步数	结果
10	御驾亲征	7	获胜
12	弈海双星	35	获胜
29	渴骥饮泉	6	认输
47	捉襟见肘	12	认输
67	兵相斗士	19	获胜
83	蜀道伏兵	14	认输
100	捕风捉影	59	和局
167	风云幻变	38	和局
181	近在咫尺	25	获胜
240	石破天惊	15	获胜

4.4 对弈

将本引擎与象眼引擎进行对弈，限制本引擎 20 秒内下一步，再根据限制象眼引擎搜索深度，分别下三盘，得到如下结果：

表 4-3 第二类残局测试结果

象眼搜索深度	第一盘	第二盘	第三盘	总比分
4	获胜	获胜	获胜	3:0
5	获胜	认输	获胜	2:1
6	认输	获胜	认输	1:2
7	认输	认输	认输	0:3

4.5 本章小结

由本章表格可以看出，本引擎具有一定的处理残局挑战的能力，可以处理所有的第一类残局，部分第二类残局。第三章中提到的克服水平效应，尤其是选择性延伸的方法，对处理第一类残局非常有帮助，因为第一类残局即为需要不断将军的残局，而选择性延伸正是在局面被将军时进行延伸，这使得第一类残局能被有效的解决。而第二类残局所需搜索的平均子节点数比第一类多，搜索树增大，从而搜索深度降低，搜索能力减弱。本文设计的方法，从实验结果来看，只能处理 50%的第二类残局，还有待提高。

在对弈方面，本引擎可以打败搜索深度控制为 5 的象眼引擎，当搜索深度大于 5 时，显露出棋力不足而认输。主要原因还是搜索深度不够以及评价函数能力不足以及缺少开局库，根本原因是底层模块及函数的复杂度不够优越以及评价函数的模型实力欠缺。

本引擎的平均搜索层次是 6-7 层，但由于尚未对基本模块进行优化，引擎存在漏洞，评价函数知识欠缺等原因，棋力稍显偏弱。由此可知，需要对基本模块进行优化，降低每个模块和函数的时间复杂度，对引擎每个模块进行数据测试，找到存在的漏洞，增强评价函数来进一步提高引擎棋力。

第五章 全文总结及展望

5.1 全文总结

本文主要介绍了所设计的引擎中的基本模块、搜索方法两个部分。基本模块主要包含局面数据结构、局面评价函数、着法排序三个部分。搜索方法部分重点介绍了引擎所含有的 4 个主要的剪枝搜索的基本原理、算法过程，通过逐层介绍的方式显示如何将它们有机结合起来。同时还介绍了克服水平效应的方法，使得搜索评价更为精准。

由第四章的测试结果可看出，本文设计的引擎只能处理 50%的第二类残局，还有待提高。在对弈方面，本引擎可以打败搜索深度控制为 5 的象眼引擎，当搜索深度大于 5 时，显露出棋力不足而认输。引擎的棋力存在许多待提升改进的地方。同时，引擎缺少开局库，导致开局布局能力不强。

由第三章可知，本引擎可平均搜索 6-7 层，具有一定的搜索能力，但从第四章可知，由于评价函数知识的欠缺，引擎的棋力及搜索能力受到了限制。因此，总的来说，本文所设计的引擎具有一定的棋力，能完成所有的第一类残局挑战及部分的第二类残局挑战，能和人进行对弈，但棋力不强，还有很多提升空间。

5.2 后续工作展望

本引擎的平均搜索层次是 6-7 层，但由于尚未对基本模块进行优化，引擎存在漏洞，评价函数知识欠缺等原因，棋力稍显偏弱。由此可知，需要对基本模块进行优化，降低每个模块和函数的时间复杂度，对引擎每个模块进行数据测试，找到存在的漏洞，增强评价函数等来进一步提高引擎棋力。

下一步工作的最终任务是：让引擎具有 1 秒钟内平均搜索 9 层的能力，能打败象眼引擎的大师级别。要做到这点，必须从多方面对引擎的棋力进行提升。在局面数据结构部分，主要方式是将本引擎与象眼引擎的局面数据结构执行效率做对比，不断优化其中的函数、算法，增强引擎底层处理能力。对每个模块进行数据测试，找出其中的漏洞和不足，并祢补，尤其是对置换表部分的检测、查错，非常重要。除此外，引擎还需增加开局库来增强布局能力。至于局面评价函数及着法排序部分，先要求做到与象眼引擎一致，再考虑用数学建模、自学习型算法等方法来增强。搜索方法部分，尝试优化每个模块，使得搜索速度增大，再考虑新的剪枝技巧，增强引擎搜索能力。

致谢

大学四年的生活即将结束，在学习过程中虽然遇到了很多问题，但是都在老师、家人、同学的帮助下顺利克服，在此对他们表示感谢。

首先，我要感谢我的指导老师——杨鹏老师。在完成毕业设计的过程中，杨老师对我进行了无私的指导与帮助，帮助我进行方法的改进与论文的完善，感谢您对我的帮助与鼓励。杨老师对学术对科研的严谨态度，使我一生收益。

其次，我要感谢上海贤趣信息技术有限公司，感谢你们的开源杰作象眼引擎以及象棋巫师网站，给我了很大的技术支持，在我屡次遇到困难时能有所依，有所靠。感谢你们对我的问题的回复，让我在完成毕设过程中扫清障碍，顺利前行。

感谢杨鹏教练以及校 ACM 集训队的同学。与你们一起在队内训练的两个暑假，使我终生难忘的经历，这使得我在编程与算法方面都有非常大的提高。

最后，感谢我的父母一直以来对我的支持与帮助，你们是我最坚强的后盾。

参考文献

[1] 上海贤趣信息技术有限公司.象棋百科全书网[EB/OL].http://www.xqbase.com/,Dec 15, 2015

[2] 上海贤趣信息技术有限公司.中国象棋对弈程序 ElephantEye(象眼)[CP/OL].https://github.com/xqbase/eleeye,March 16,2016

[3] 蒋鹏,雷贻祥,陈园圆.C/C++中国象棋程序入门与提高[M].北京:电子工业出版社,2009,1-333

[4] 黄少龙.象棋中局精妙战法[M].北京:金盾出版社,2004,94-105

[5] 岳金鹏,冯速.博弈树搜索算法在中国象棋中的应用[J].计算机系统应用,2009,(9):1-4

[6] 裴祥豪.基于剪枝策略的中国象棋搜索引擎研究[D].河北:河北大学,2009,6-8

[7] 本书编写组.象棋竞赛规则 201 试行[M].北京:人民体育出版社,2011,1-132

[8] 霍文会,王巍.象棋竞赛规则入门导读[M].北京:北京体育大学出版社,2006,1-181

附录

1. 引擎代码链接

https://github.com/peteryuanpan/lazyboy/

2. 局面数据结构（部分）

```
struct PositionStruct {
    // 基本成员
    int player;   // 轮到哪方走，0 表示红方，1 表示黑方
    int square[256];  // 每个格子放的棋子，0 表示没有棋子
    int piece[48];    // 每个棋子放的位置，0 表示被吃
    int bitRow[16];   // 每行的位压
    int bitCol[16]; // 每列的位压
    std::pair<ULL, ULL> zobrist; // zobrist 值，双哈希
    int nDistance; // 搜索深度，初始值为 0
    bool check; // 将军态
    bool checked; // 被将军态
    bool chased; // 被捉态
    int vlRed; // 红方子力值
    int vlBlk; // 黑方子力值
};
```

外文资料原文

The History Heuristic and Alpha-Beta Search Enhancements in Practice

Jonathan Schaeffer

Abstract

Many enhancements to the alpha-beta algorithm have been proposed to help reduce the size of minimax trees. A recent enhancement, the history heuristic, is described that improves the order in which branches are considered at interior nodes. A comprehensive set of experiments is reported which tries all combinations of enhancements to determine which one yields the best performance. Previous work on assessing their performance has concentrated on the benefits of individual enhancements or a few combinations. However, each enhancement should not be taken in isolation; one would like to find the combination that provides the greatest reduction in tree size. Results indicate that the history heuristic and transposition tables significantly out-perform other alpha beta enhancements in application generated game trees. For trees up to depth 8, when taken together, they account for over 99% of the possible reductions in tree size, with the other enhancements yielding insignificant gains.

Introduction

Many modifications and enhancements to the alpha-beta (αβ) algorithm have been proposed to increase the efficiency of minimax game tree searching. Some of the more prominent ones in the literature include iterative deepening, transposition tables, refutation tables, minimal window search, aspiration search and the killer heuristic. Some of these search aids seem to be beneficial while others appear to have questionable merit. However, each enhancement should not be taken in isolation; one would like to find the combination of features that provides the greatest reduction in tree size. Several experiments assessing the relative merits of some of these features have been reported in the literature, using both artificially constructed and application generated trees.

The size of the search tree built by a depth-first αβ search largely depends on the

order in which branches are considered at interior nodes. The minimal αβ tree arises if the branch leading to the best minimax score is considered first at all interior nodes. Examining them in worst to best order results in the maximal tree. Since the difference between the two extremes is large, it is imperative to obtain a good ordering of branches at interior nodes. Typically, application dependent knowledge is applied to make a "best guess" decision on an order to consider them in.

In this paper, a recent enhancement to the αβ algorithm, the history heuristic, is described. The heuristic achieves its performance by improving the order in which branches are considered at interior nodes. In game trees, the same branch, or move, will occur many times at different nodes, or positions. A history is maintained of how successful each move is in leading to the highest minimax score at an interior node. This information is maintained for every different move, regardless of the originating position. At interior nodes of the tree, moves are examined in order of their prior history of success. In this manner, previous search information is accumulated and distributed throughout the tree.

A series of experiments is reported that assess the performance of the history heuristic and the prominent αβ search enhancements. The experiments consisted of trying all possible combinations of enhancements in a chess program to find out which provides the best results. The reductions in tree size achievable by each of these enhancements is quantified, providing evidence of their (in)significance. This is the first comprehensive test performed in this area; previous work has been limited to a few select combinations. Further, this work takes into account the effect on tree size of ordering branches at interior nodes, something not addressed by previous work.

The results show that the history heuristic combined with transposition tables provides more than 99% of the possible reductions in tree size; the others combining for an insignificant improvement. The history heuristic is a simple, mechanical way to order branches at interior nodes. It has none of the implementation, debugging, execution time and space overheads of the knowledge-based alternatives. Instead of using explicit knowledge, the heuristic uses the implicit knowledge of the "experience" it has gained from visiting other parts of the tree. This gives rise to the apparent paradox that less knowledge is better in that an application dependent knowledge based ordering method can be approximated by the history heuristic.

外文资料译文

历史启发式和 Alpha-Beta 搜索增强实践

Jonathan Schaeffer

摘要

已经提出了对 alpha-beta 算法的许多改进以帮助减小极小树的大小。描述了最近的增强，历史启发式，改善了内部节点考虑分支的顺序。报告一组全面的实验，尝试所有增强功能的组合，以确定哪一个产生最佳性能。以前关于评估其绩效的工作集中在个人增强或少数组合的好处。但是，每个增强都不应该孤立; 人们希望找到提供最大减少树大小的组合。结果表明，历史启发式和转置表在应用程序生成的游戏树中显着地超出了其他 alpha beta 增强。对于深度为 8 的树，当它们合在一起时，它们占树的大小可能减少的 99％以上，其他增强效果不大。

简介

已经提出了对 alpha-beta（a-b）算法的许多修改和增强，以提高极小游戏树搜索的效率。一些文学中更突出的包括迭代深化，转置表，反驳表，最小窗口搜索，抽象搜索和杀手启发式。其中一些搜索工具似乎是有益的，而其他搜索工具似乎有疑问的优点。但是，每个增强都不应该孤立;人们希望找到能够最大限度减少树形大小的功能的组合。在文献中已经报道了一些评估这些特征的相对优点的实验，使用人工构建的和应用生成的树。

通过深度优先的 a-b 搜索构建的搜索树的大小很大程度上取决于在内部节点处考虑分支的顺序。如果在所有内部节点首先考虑导致最佳极小分数的分支，则出现最小 a-b 树。在最坏的情况下检查他们最好的顺序会导致最大的树。由于两个极端之间的差异很大，所以必须在内部节点获得良好的分支排序。通常，应用依赖的知识被应用于对订单进行“最佳猜测”决定以考虑它们。

在本文中，最近对 a-b 算法的增强，历史启发式进行了描述。启发式通过改进在内部节点处考虑分支的顺序来实现其性能。在游戏树中，相同的分支或移动将在不同的节点或位置发生多次。维持每个举措在导致内部节点的最高极小分数方面取得成功的历史。不管起始位置如何，这个信息都是针对每个不同的动作进行维护的。在树的内部节点处，按照先前的成功历史进行检查。以这种方式，先

前的搜索信息被累积并分布在整个树中。

报道了一系列实验，以评估历史启发式的表现和突出的 alpha-beta 检索增强。实验包括在国际象棋程序中尝试所有可能的增强组合，以找出哪些提供最好的结果。 通过这些增强功能可以实现的树形尺寸的减少量被量化，提供了它们（重要性）的证据。这是在这一领域进行的第一次综合测试；以前的工作已经被限制在几种选择组合中。此外，这项工作考虑到在内部节点上订购分支的树形大小的影响，以前的工作没有解决。

结果表明，历史启发式结合转置表提供了超过 99％的可能的树形大小的减少；其他组合改善不大。历史启发式是一种简单，机械的方式在内部节点上分配分支。它没有基于知识的替代方案的实现，调试，执行时间和空间开销。启发式使用隐含的知识，而不是使用显式知识，而是从访问树的其他部分获得的“经验”的隐含知识。这导致明显的悖论，较少的知识更好，因为基于应用的依赖知识的排序方法可以通过历史启发式近似。