G05

学校代号 10532　　学　　号 J06010006

分 类 号 TP393　　密　　级

湖南大学
HUNAN UNIVERSITY

高校教师硕士学位论文

基于 OpenMP 的并行混合 PVS 算法及其应用

学位申请人姓名 邹竞

培 养 单 位 信息科学与工程学院

导师姓名及职称 谢鲲 副教授

学 科 专 业 计算机应用技术

研 究 方 向 计算机博弈

论文提交日期 2012 年 6 月 20 日

湖 南 大 学

学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写的成果作品。对本文的研究做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后果由本人承担。

作者签名：邹竞　　日期：2012 年 6 月 20 日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权湖南大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于

1、保密□，在_____年解密后适用本授权书。

2、不保密☑。

（请在以上相应方框内打“√”）

作者签名：邹竞　　日期：2012 年 6 月 20 日

导师签名：　　日期：2012 年 6 月 20 日

学校代号：10532

学　　号：J06010006

密　　级：

湖南大学高校教师硕士学位论文

基于 OpenMP 的并行混合 PVS 算法及其应用

学位申请人姓名：　邹竞

导师姓名及职称：　谢鲲 副教授

培　养　单　位：　信息科学与工程学院

专　业　名　称：　计算机应用技术

论文提交日期：　2012 年 6 月 20 日

论文答辩日期：　2012 年 7 月 7 日

答辩委员会主席：　廖波 教授

Parallel Hybrid PVS Algorithm Based on OpenMP And Its Application

by

Zou Jing

B.E.(Xiangtan University)2001

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of science

in

Computer Application Technology

in the

Graduate school

of

Hunan University

Supervisor

Associate Professor Xie Kun

June, 2012

摘要

计算机博弈是人工智能中一个非常具有挑战性的研究方向，对各种博弈树搜索算法和优化措施的研究和组合，又是计算机博弈中研究的重点。国际象棋计算机博弈已经获得了巨大的成就，早已具备击败人类冠军的智能。中国象棋计算机博弈的研究起步较晚，难度更大，挑战性更强，吸引了越来越多的研究者对其进行研究，也取得了不俗的成就。

OpenMP 是一种基于共享内存的并行化程序设计的标准，具有开发简单、抽象度高、可移植性强等诸多优点。多核 CPU 的问世和普及，使廉价的普通 PC 也能进行基于共享内存的并行计算。使用 OpenMP 标准，将已有算法在多核 PC 环境下进行并行计算，能充分应用硬件资源，具有很强的实用性。

本文对各种博弈树搜索算法及优化措施进行了分析和比较，并阐述了 OpenMP 标准下的共享内存的并行程序设计方法。针对中国象棋计算机博弈，本文将空着裁剪、置换表、吃子启发、置换表启发、历史启发、杀手启发融入博弈树搜索的 PVS（主要变例搜索）算法，设计了一种混合 PVS 算法，提高了剪枝效率，使算法能在相同的时间内搜索更深的层次。进一步，以广泛普及的多核 PC 为环境，在 OpenMP 2.5 标准下，以 PVSplitting（主要变例分裂）策略对混合 PVS 算法进行了并行化设计，相比于串行 PVS 算法，并行优化后，可充分利用了多核 CPU 资源，提高了搜索效率。

本文还用面向对象方法设计了一个真实的多核 PC 环境下的中国象棋计算机博弈系统，将 OpenMP 下的并行混合 PVS 算法运用于搜索引擎中，对其进行了实际试验，同时针对优化估值函数的自适应遗传算法进行了改进，并使用 OpenMP 2.5 进行了并行化设计，为多核 PC 环境下中国象棋计算机博弈系统的设计与优化提供了一种便捷而有效的思路。

关键字：计算机博弈；PVS 算法；空着裁剪；置换表；启发策略；并行计算；OpenMP；中国象棋

Abstract

Computer game is a challenging research branch in artificial intelligence, a study and combination of various search algorithm and optional approach of game tree, and a key point among the computer game researches. Chess computer game has got great achievement in the world and has owned the intelligence to beat the human champion. However, the research on Chinese chess computer game started much later, and therefore it has more difficulties and greater challenge. More and more researchers have interest in it and have had good success.

OpenMP is a standard of parallel programming design based on shared-memory. It has many advantages, such as simple development, high abstract degree and probability. With the appearance and popularity of multi-core processor, the common PC with low-price can be used to parallel computing. If we put the former algorithm into the PC with multi-core processor and have the parallel computing under the OpenMP standard, we can make full use of hardware sources, so it has more practicability.

This paper analyzes and compares kinds of search algorithm and optional approach of game tree and elaborates the designing methods of shared- memory parallel programming under the OpenMP standard. In the light of Chinese chess computer game, the paper combines the null move pruning, translation table, capturing heuristic, killer heuristic, history heuristic and translation table heuristic into the PVS algorithm aiming at game tree search and mix them into PVS algorithm so that we can improve the efficiency of pruning and make the algorithm search more deeply with the same time consuming. Moreover, by means of PVSplitting in the popular multi-core processor under the OpenMP standard, the paper parallel optimizes a hybrid PVS algorithm to fully utilize multi-core processor sources and improve the efficiency of search.

What’s more, the paper designs a system of Chinese chess computer game in the condition of a real multi-core processor by the object-oriented approach and experiments on the parallel hybrid PVS algorithm. At the same time the paper improves and designs parallel the adaptive genetic algorithm to optimize evaluation function in the OpenMP 2.5 standard and provides a convenient and effective thought used to design and optimize the system of Chinese chess computer game in

the condition of multi-core processor.

Key Words: Computer game; PVS algorithm; Null move pruning; Translation table; Heuristic approach; OpenMP; Chinese chess

目录

插图索引

附表索引

第1章 绪论

1.1 研究背景

人工智能（Artificial Intelligence，AI）是一门新兴的综合性学科，它研究的是人类智能行为在的自动化[1]。简单来说，人工智能研究的是如何让计算机拥有人类的智能。博弈（Game Playing）指的是在多个决策主体的目标存在冲突，行为具有相互作用时，各主体根据所掌握信息及对自身能力的认知，做出有利于自己的决策的一种行为[2]。博弈是伴随着古代棋类游戏而产生的。人类社会发展至今，博弈问题已经无处不在，小到人与人的辩论、下棋、打牌等对抗性游戏，大到商家与商家的竞争，国与国战争、外交等，只要局面中多个决策主体存在利益冲突，博弈就成为体现矛盾和寻求对策的过程和方式[3]。计算机博弈（Computer Game）也称为机器博弈，是人工智能的一个重要研究分支，也是人工智能领域最具挑战性的研究方向之一。早在 20 世纪 50 年代中后期，就有学者开始研究如何让计算机模拟人的思维，让机器学会“思考”，和人类来下棋，实现计算机和人的对弈[4]。计算机博弈目标明确，是研究计算机思维的最好载体，在计算机博弈研究中获取的成果，亦可推广到人工智能的其他领域，所以计算机博弈理论和技术对于人工智能的意义，就相当于果蝇对生物学的意义一样重大[5]。

国内外大多数传统的棋类游戏，如井字棋、五子棋、围棋、国际象棋、中国象棋等，都具备二人零和、全信息、非偶然的特点[2,6]。二人零和，指的是参与博弈的只有两方，两方的利益完全对立，一方获益了多少，另一方就损失了多少；全信息，指的是博弈双方对影响博弈的全部要素，包括己方、对方和环境的全部信息，以及历史信息和当前格局都是知晓的；非偶然，指的是博弈双方每一步所做的决策都需要以得失大小为依据进行分析，不是任意的。这种二人零和、全信息、非偶然的博弈，是一种相对简单和基础的博弈，也是计算机博弈的主要研究对象。

国际上最初对计算机博弈的研究，都是基于国际象棋的。经过半个多世纪的发展，研究者们在计算机博弈这一领域取得了重大成就。1956 年，Los Alamo 实验室研制出了一个真正能够在 MANIAC-I 机器上运行的棋类游戏的模拟程序，但是这个程序还不能算一个真实环境下的计算机博弈程序，因为这个程序对棋盘、棋子、规则等都进行了很多简化[7]，但这却是一个良好的开端。1957 年，Bemstein 在 IBM 740 机器上编写出第一个完整的计算机国际象棋程序，采用的是深度优先搜索策略，每次搜索 4 层，IBM 740 机器成为了世界上第一台能和人类下棋的

机器，被取名为“思考”[7]。1967 年，麻省理工大学的 Greenblatt 等人在 PDP-6 机器上开发出 Mac Hack Ⅵ程序，参加麻省国际象棋锦标赛，第一次击败人类选手[7]。1978 年，Ken Thompson 开发国际象棋机 BELLE，每秒钟搜索十万个局面，达到精通水准。1979 年，国际象棋软件 Chess 4.9 已然达到专家水准[3,7]。1987 年，美国的 Carnegie Mellon 大学研发的国际象棋计算机程序“深蓝”面世，每秒钟能思考 75 万个局面，并于 1988 年击败丹麦著名棋手拉尔森[3,7]。1989 年，已经拥有 6 台专用处理器的“深蓝”每秒钟能思考的局面数达到 200 万个[7]！1991 年，ChessBase 公司开发的国际象棋计算机程序 Fritz 问世[7]。1993 年，“深思二代”战胜世界顶级女棋手小波尔加[3,7]。1997 年，“深蓝”的升级版——使用大型计算机拥有 256 个处理芯片的 IBM 超级计算机“更深的蓝”以 3.5:2.5 的成绩战胜当时世界最顶尖的国际象棋冠军卡斯帕罗夫[3,7]，举世震惊，这一事件在计算机博弈史上具有重大意义，标志着计算机战胜人类顶级棋手已非不可能之事，卡斯帕罗夫本人也惊叹“这一天（计算机智能战胜人类顶尖棋手的时刻）终于来临了！”2001 年到 2002 年期间，Fritz 的升级版 More Fritz 战胜了除了克拉姆尼尔之外的世界排名前 10 位的国际象棋顶尖男棋手，在与克拉姆尼尔的对弈中，也不落下风，以 4:4 战平[7]。2005 年，超级计算机 Hydra 在和英国国际象棋大师迈克尔·亚当斯的对弈中，以 5.5:0.5 大胜[7]。Hydra 在这次比赛期间仅仅利用了其 64 台 PC 中的 32 台，每秒钟能够计算 2 亿个局面，还能够预测 40 步棋[7]，远非人脑所能及。在国际象棋上，计算机的博弈水平已经远远超越人脑了。

并行计算（Parallel Computing）是从 20 世纪 70 年代出现的一种通过多个计算处理单元（节点）分担负载的方法，对海量数据或超大规模运算同时进行处理的新技术，这些海量数据或者繁重的计算负载被分布在了不同的计算处理单元中，每个处理单元分担其中一部分任务进行处理[8]。20 世纪 80 年代，并行计算开始应用于国际象棋计算机博弈。计算机能够战胜人类顶尖棋手，除了各种相关算法越来越先进成熟之外，并行计算当然也是功不可没。

中国象棋不仅是在中国普及最广的棋类游戏，也是世界上历史最悠久的棋类游戏，蕴含了博大精深的中国古代文化[3]。但是，中国象棋计算机博弈研究的起步却比较晚，不过发展较快。上世纪 70 年代末 80 年代初，中国台湾的学者率先开始了中国象棋计算机博弈的研究工作，并于上世纪 80 年代初开发出了基于苹果机（6502）和 IBM PC（8088）的中国象棋计算机博弈程序，但是智能不高。1985 年，台湾的许舜钦教授开始着手中国象棋计算机博弈算法的研究，在近 20 年的时间内，许舜钦教授和他的学生开发出的很多中国象棋计算机博弈程序，如“ELP”[9]、“千虑（Contemplation）”[9]等，均在台湾乃至世界的计算机象棋大赛中取得了非凡的成绩，许舜钦也被誉为“计算机中国象棋之父”[9]。台湾的郑明政等人开发出的“象棋世家”[7]等也在计算机象棋大赛中获奖。中国大陆也

有不少研究者相继投入到了中国象棋计算机博弈程序的研究和开发，涌现出了一些优秀的作品，如中山大学的涂志坚开发的“纵马奔流（ZMBL）”[7,9]、北京的陈朝阳开发的“象棋旋风（Cyclone）”[7,9]、东北大学的徐心和、王骄等人开发的“棋天大圣（NEUChess）”[3,7,9]、北京的赵明阳开发的“象棋奇兵（XQMaster）”[7,9]、上海机器博弈研究所的黄晨开发的开源软件“象眼（ElephantEye）”[7,9]等，都是其中的佼佼者，这些作品都是在中国象棋计算机博弈的国际奥林匹克大赛中的获奖作品，代表了国内中国象棋计算机博弈研究的最尖端水平，其中尤以“棋天大圣”因“深蓝”的开发团队都参与研究而最引人关注[9]。随着中国象棋在国际上的影响逐步扩大，在国外，也有研究者开发出了非常优秀的中国象棋计算机博弈程序，如法国的“谢谢大师（XieXieMaster）”[7]、美国的“神乎其技（SHCC）”[7]等。但总体而言，由于中国象棋的计算机博弈研究起步较晚，中国象棋的棋盘的可落子点更多，局面更为复杂，搜索难度和估值难度超过国际象棋很多，专门针对中国象棋计算机博弈的资料不多，兼之中国象棋蕴含了丰富的东方文化，在国际上的普及率不如国际象棋，研究者相对较少，因此，总体而言，中国象棋的计算机博弈的研究高度还远不如国际象棋计算机博弈的研究高度，尚无法战胜胡荣华、许银川这些人类棋手的顶尖大师[10]。国际象棋与中国象棋的复杂度的比较，如表 1.1 所示。

表 1.1 国际象棋与中国象棋的复杂度比较

游戏名称	棋盘大小	状态空间复杂度	博弈树复杂度
国际象棋	64（8×8）	10^{46}	10^{123}
中国象棋	90（10×9）	10^{48}	10^{150}

随着研究者的不断努力，借鉴计算机战胜国际象棋绝顶高手的经验，相信计算机战胜中国象棋冠军的这一天终会来临。

1.2 国内外研究现状

很早以前，研究者就确定了二人零和、全信息、非偶然的博弈问题的数学模型——博弈树。这样，对这种计算机博弈问题的研究，就转换成了对博弈树的研究，针对棋类游戏的计算机博弈系统，学者研究的关键技术包括[3]：

（1）棋盘和棋子的数据结构（Data Structure for Board and Piece）。要开发棋类游戏的计算机博弈系统，首先要解决的问题，就是对棋盘、棋子等数据进行合理有效的组织和存储，使这些信息能够方便的被计算机识别并处理。合理的数据结构不仅能较好的表示棋盘和棋子的信息，使计算机能够明确知道博弈进行中双方的状态，还能较快的生成着法。

（2）着法生成（Move Generation）。着法可以看作博弈树中的结点从一种

状态到达另一种状态的算子。着法生成研究的是如何快速产生机器在某一局面下的所有可能的着法，并判断人类棋手的着法是否符合规则。

（3）估值函数（Evaluation Function）。棋类游戏的博弈，不是几步棋之内就能决出高下的，尤其是当双方都是高手时，要经过很多的着法，才能分出胜负。在博弈树中，每个局面对应一个结点。在难以判定输赢的时候，就应该对结点所表示的局面的好坏做一个量化。估值函数研究的是为某个局面估计一个分值，是评价局面优劣程度的标准。

（4）搜索算法（Search Algorithm）。中国象棋、国际象棋等棋类游戏在某个局面下后继状态的数量十分庞大，计算机无法分析出最终的胜负状态，只能向后搜索到一定的深度。搜索算法的作用就是从当前局面开始，对博弈树进行敌对搜索，根据局面估值系统对博弈树结点计算出的分值，找出当前最优的一种着法。

（5）开局库（Opening Book）。国际象棋、中国象棋等棋类游戏的开局方式虽然非常多，搜索算法要经过繁重的搜索才能获得一个开局的着法，但有一定会水平的人类棋手都是为数不多的十几种高质量的着法，开局库就是将棋类游戏的开局方法建成一个数据库，开局阶段每次搜索之前先查找数据库，以减少开局时的搜索时间，提高开局的质量。

其中，搜索算法是这些关键技术中最核心的部分，大多数棋类游戏的搜索算法都具有一定的通用性，与具体的棋类游戏的规则关系不大，比如 α-β 搜索/剪枝算法[22]，既能应用于国际象棋，也能应用于中国象棋、五子棋等，搜索算法是国内外关于计算机博弈研究的最热门的研究方向。估值函数依赖于具体的棋类游戏的规则，不同的棋类游戏，估值函数的设计是不同的，针对某个具体的棋类游戏，如何完善和优化估值函数，也是计算机博弈研究的一个热点。

目前，国内外对搜索算法的研究主要体现在两个方面：一是发现更好的博弈树搜索算法，或者针对某个具体的棋类游戏将某些已有的搜索算法和优化措施结合起来，发挥更好的效果；二是将已有的搜索算法进行并行化优化，利用并行计算机的硬件优势，减少计算机的思考时间，或者让计算机能思考更多的局面。

博弈树搜索算法很多，总体可以分为两大类[11,12]：一类是基于深度优先的 α-β 搜索/剪枝策略的算法，目前使用最为广泛的 PVS 算法[35]和 MTD(f)算法[38]都是基于 α-β 搜索/剪枝策略的衍生算法，这类算法都根据一个估值上界 α 和一个估值下界 β 构成的窗口[α,β]来限制搜索的范围，对估值不在这个范围内的结点实施剪枝；另一类是基于状态图最佳优先搜索策略的算法，如 SSS*算法[12]等。虽然 SSS*算法在搜索博弈树时，遍历的结点数量少于 α-β 搜索/剪枝算法，但是算法复杂难以理解，需要维护一个非常庞大的有序 OPEN 队列，巨大的内存空间开销和维护有序队列的额外时间开销使其在实际应用中并不比 α-β 搜索/剪枝算法效果更好[12]。而 α-β 搜索/剪枝算法灵活性强，经过置换表、启发式搜索等

增强措施不断优化，衍生为 PVS、MTD(f)等算法时性能早已远高于最初的 α-β 搜索/剪枝算法。1995 年，Aske Plaat 发现，结合置换表，基于空窗搜索的 α-β 搜索/剪枝算法完全可以替代 SSS*算法[11,12]。因此 PVS、MTD(f)等算法成为了主流的博弈树搜索算法，著名的国际象棋计算机博弈程序 Cilkchess 就使用了 MTD(f)算法[12]，国内著名的中国象棋计算机博弈程序“纵马奔流”和“象眼”都采用了 PVS 算法[13,14]。而对 SSS*算法的研究逐渐趋向冷门。

博弈树搜索算法的并行化也很自然地分为两个大类[11]：一类是基于 α-β 搜索/剪枝策略的并行博弈树搜索算法，如著名的国际象棋引擎 Crafty 使用的 DTS（Dynamic Tree Splitting，动态树分裂）算法[11]；另一类是基于其它搜索策略的并行博弈树搜索算法，如 1990 年 Steinberg 和 Solomo 提出的 ER（Evaluate-Refute，评估-拒绝）算法[11]，以及 2002 年 Shoham 和 Toledo 提出的算法 PRBFM（Parallel Randomized Best-First Minimax Search，并行随机最佳优先极小极大值搜索）算法[11]。由于 α-β 搜索/剪枝算法及其衍生算法依旧是主流的博弈树搜索算法，因此基于 α-β 搜索/剪枝策略的并行博弈树搜索算法依旧是博弈树搜索算法并行化研究的热门。

对基于 α-β 搜索/剪枝算法及其衍生算法的并行化，在具体实现上有两种不同的方式[15]：

（1）基于窗口的分割。这种并行化方式将搜索的窗口[α,β]划分成多个子窗口，每个计算处理单元以其中一个子窗口去搜索整个博弈树。如基于窗口分割的 α-β 搜索/剪枝算法[16]。事实上 MTD(f)算法本身也体现了这一思想，MTD(f)算法在一个大的窗口范围内不断地尝试空窗探测，这个大的窗口在搜索中不断缩小，可以划分为多个子窗口进行并行化处理。

（2）基于子树的分割。这种并行化方式将博弈树以某个结点为根，划分成多个子树，对每个子树的搜索进行并行计算。如果某个计算处理单元在一个子树上找到了更好的结点，可以动态更新窗口的边界值，并通知其它计算处理单元窗口的边界值已更新，窗口范围比原先要小，这样使得其它计算处理单元可能引发更多的剪枝。基于子树分割的并行方法有 Ferguson 和 Korf 提出的 DTS 算法[15]、Hyratt 提出的 PVSplitting 算法[17,18]、王京辉和乔卫民提出的一种基于 PVM 的博弈树的网络并行搜索算法[19]。

近年来，随着双核、四核等多核处理器在微机中的日益普及，多核处理器占据了 PC 机市场的主流，普通的 PC 机也具备了并行计算的硬件条件。多核 PC 从体系结构来看，属于共享内存多处理器（SMP）系统。如何有效利用多核技术，将算法从传统的串行模式改进为多核环境下的并行模式，充分利用 CPU 资源，提高算法效率，成为近期热门的研究方向之一。OpenMP[20]是一种面向共享内存的多核多线程并行程序设计技术，已经成为工业标准之一，其 API（应用程序接

口）由 SGI 公司发起，用于编写可移植的多线程应用程序，通过与 Fortran、C 和 C++语言结合实现并行计算，支持大多数的类 Unix 系统和所有的 Windows 系统，对于同步共享变量、合理分配计算负载等，都提供了有效的支持，具有开发简单、通用性好，可扩充性强等优点。如果能将某个主流的博弈树串行搜索算法，针对共享多核 CPU 架构特性使用 OpenMP 编程接口对搜索算法进行并行分解，将计算负载均衡分配到多个 CPU 处理单元上进行并行计算，利用多核优势提高博弈树的搜索效率，在应用上无疑具有较大意义。

1.3 研究内容与论文结构

1.3.1 本文的主要工作

限于时间和自身水平，本文仅对计算机博弈研究的核心内容——博弈树搜索算法及其优化措施进行研究，将多种算法和增强措施相结合，探索一种适合中国象棋计算机博弈的混合博弈树搜索算法，并将该算法使用 OpenMP 模型进行并行化，以推进中国象棋计算机博弈的研究。本文的研究工作主要从以下方面展开：

1．以中国象棋和国际象棋作为载体，对计算机博弈的发展现状和当今学术界研究的热点进行总结，明确了研究的意义所在。

2．分析 α-β 搜索/剪枝算法及其各种衍生算法，以及相关的搜索策略和优化措施，深刻理解各种各种算法和增强措施的优势和不足，该如何结合使用；分析了 OpenMP 模型的机制，总结了在 OpenMP 环境下进行共享内存的多核并行程序设计需要注意的问题。

3．根据中国象棋的特点，以 PVS 算法作为主要框架，将尽可能多的优化措施结合起来，形成一种针对中国象棋计算机博弈的混合 PVS 算法，并将这种混合 PVS 算法在 OpenMP 2.5 下进行并行化设计。

4．使用面向对象方法和 Visual C++ 2010 平台开发出一个实际的中国象棋计算机博弈系统，以此为真实的实验平台，对本文提出的 OpenMP 2.5 下的混合 PVS 算法进行实验和分析，并使用 OpenMP 2.5 环境下的混合遗传算法对估值函数进行了优化。

1.3.2 本文的组织结构

本文包含绪论（第 1 章）、主体（第 2 章到第 5 章）、结论三大主要部分。

第 1 章为绪论，本章以国际象棋和中国象棋为例，介绍了本课题的研究背景、博弈树搜索算法的研究现状，阐述了本课题的研究内容和本文的行文结构。

第 2 章介绍 α-β 搜索/剪枝算法及其各种衍生算法，以及相关的搜索策略和

优化措施。主要包括 α-β 搜索/剪枝算法、置换表、启发式搜索、PVS 算法、空着裁剪、MTD(f)算法等，以及一些并行策略。

第 3 章介绍 OpenMP 标准和机制，以及在 OpenMP 环境下进行共享内存的并行程序设计的方法。

第 4 章根据中国象棋的特点，研究出一种融合了多种增强措施、适合中国象棋计算机博弈的混合 PVS 算法，并使用 PVSplitting 并行思想在 OpenMP 2.5 下将其并行化。

第 5 章介绍了一种基于面向对象方法和 Visual C++ 2010 平台的中国象棋计算机博弈程序的设计，对 OpenMP 下的混合 PVS 算法进行实验和分析，并介绍了使用 OpenMP 下的并行遗传算法对估值函数进行优化的过程。

结论部分，对本文进行了总结，指出了 OpenMP 下并行 PVS 算法的优点和不足，并指出了下一步的研究方向。

第2章 博弈树搜索算法及其优化措施

计算机博弈中，计算机的智能和思考速度，主要取决于博弈树搜索算法。α-β搜索/剪枝算法及其衍生算法是博弈树搜索的主流算法，本章将介绍α-β搜索/剪枝算法及其各种衍生算法，以及相关的优化措施和并行策略。

2.1 博弈树

假设甲乙双方在进行“二人零和、全信息、非偶然”的棋类游戏，从某个初始局面开始，如果轮到甲方走棋，甲方有很多种着法，但只能选择一个着法进行走棋。甲方走棋后，局面发生了变化，轮到乙方走棋，乙方也有很多种着法，但也只能选择一个着法。甲乙两方交替走棋，局面的变化可以表示成一个树形结构，这就是博弈树（game-tree）[21]。一种井字棋的博弈树，如图 2.1 所示。

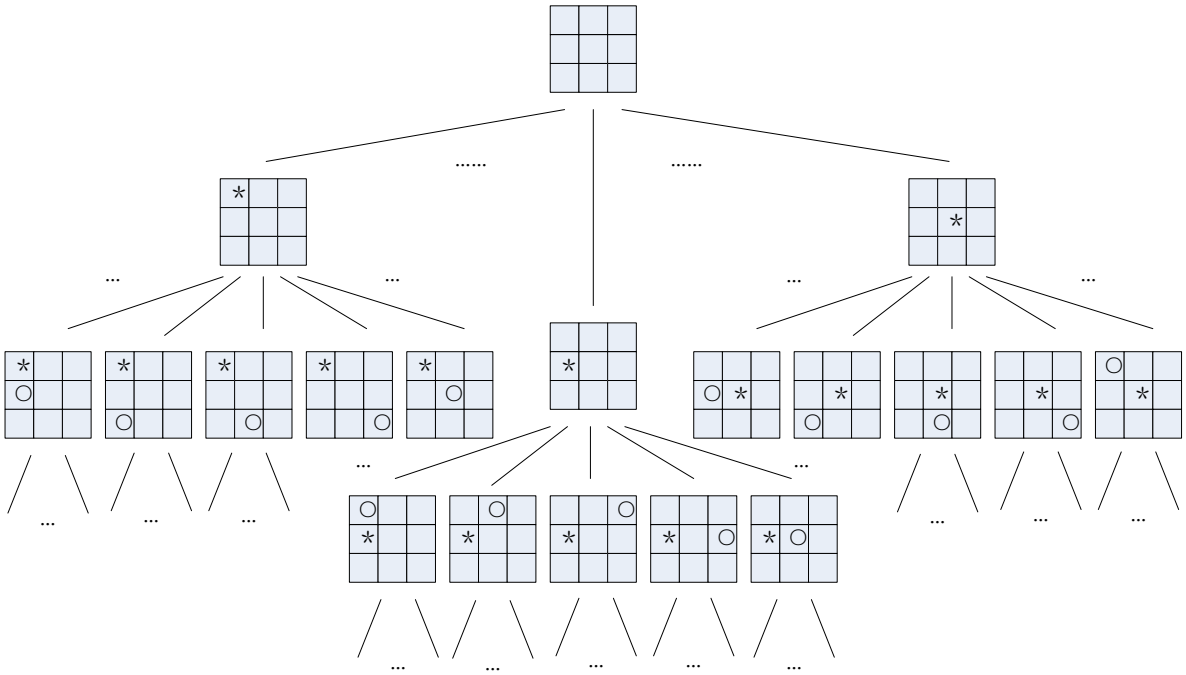


图 2.1 井字棋博弈树示例

每个局面可以用博弈树的一个结点来表示，某方获胜、失败或双方平局的结点构成了叶子结点。博弈树是一种特殊的与或树，“或”结点和“与”结点是逐层交替出现的。己方扩展的结点之间是“或”关系，对方扩展的结点之间是“与”关系[21]。

2.2 博弈树搜索算法与优化措施

2.2.1 极小极大值算法与负极大值算法

极小极大值算法（Minimax Algorithm）与负极大值算法（Negamax Algorithm）的性质是一样的，这里放在一起介绍。

2.2.1.1 极小极大值算法

博弈双方的目的正好相反，一方总是倾向选择对自己最有利、最能钳制对手的决策，同时还要充分考虑对方的应对策略。一方做出决策后，都会引发局面的变化。一般会根据具体的问题，设计一个估值函数，通过估值函数计算出一个分值，对局面的优劣程度进行定量分析。可将某方获胜局面的分值定义为+∞（用一个非常大的正数来表示），另一方获胜局面的分值定义为-∞（用一个非常小的负数来表示），其他局面的分值为-∞到+∞之间的一个值，平局的分值为 0 或者接近 0 的值[22]。把获胜局面的分值为+∞的一方称为 MAX 方，另一方称为 MIN 方，轮到 MAX 方选择着法的局面，称为极大值结点，轮到 MIN 方选择着法的局面，称为极小值结点。实际的博弈树因规模太大，通常只能搜索到某一固定深度，一次搜索对应的叶子结点，不一定就是胜、负、平的局面，绝大多数情况下是达到最大搜索深度的结点。叶子结点的估值，可以直接通过估值函数计算出来，而非叶子结点的的估值，需要通过以这个结点为根的子树从叶子结点开始，自下而上的倒推计算。推算的方法是：对“或”结点，选择其孩子结点中一个最大的得分作为父结点的得分，即在自己的备选决策中选一个对自己最有利的决策；对“与”结点，选择其孩子结点中一个最小的得分作为父结点的得分，即立足于最坏的情况。这就是极小极大算法[22]，这个算法可以用递归过程实现，用类 C/C++伪代码可表示如下：

```
int MinMax(board, depth)
{
    if(GameOver(board) || depth <= 0) return Evaluation(board);
    if(board 的当前走棋方是 MAX 方) best = -∞;
    else best = +∞;
    w = CreateSuccessors(board, p); //生成局面 board 下的 w 种着法 p0…pw-1
    for(i = 0; i < w; i++)
    {
        MakeMove(board, pi); //执行着法 pi 生成新的局面 board
        value = MinMax(board, depth - 1);
```

```
            RestoreMove(board, pi); //撤销着法 pi 恢复原先局面 board
            if(board 的当前走棋方是 MAX 方) best = max(best, value);
            else best = min(best, value);
        }
        return best;
    }
```

2.2.1.2 负极大值算法

极小极大值算法中，一方试图获取极大值，另一方试图寻找极小值，其实二者可以统一起来。1975 年 Knuth 和 Moore 提出了负极大值算法[22]，它的思想是：父结点的估值是其孩子结点估值相反数的最大值，这样避免了奇数层取极小值而偶数层取极大值的麻烦。负极大值算法可用类 C/C++伪代码表示如下：

```
int NegaMax(board, depth)
{
    if(GameOver(board) || depth <= 0) return Evaluation(board);
    best = -∞;
    w = CreateSuccessors(board, p); //生成局面 board 下的 w 种着法 p0…pw-1
    for(i = 0; i < w; i++)
    {
        MakeMove(board, pi); //执行着法 pi 生成新的局面 board
        value =- NegaMax (board, depth - 1);
        RestoreMove(board, pi); //撤销着法 pi 恢复原先局面 board
        if(value > best) best = value;
    }
    return best;
}
```

负极大值算法与极小极大算法的实质和效率是一样的，都属于完全搜索，只是在表达形式上，负极大值算法稍显简洁。

2.2.2 α-β 搜索/剪枝算法

用极小极大值算法或者负极大值算法对实际的博弈树进行搜索，即使只做少数几层的搜索，也会因为分支因子太大而超过计算机的处理能力。有很多结点的并不会影响搜索的结果，完全没必要展开。搜索算法应该能够识别并舍弃这种明显无意义的分支，提高搜索效率。

1963 年，Brudno 提出了 α-β 搜索/剪枝算法（α-β Search Algorithm）[22]，通过设置下界值和上界值的方法对博弈树待搜索结点的估值范围进行了划定，并利用这个边界值进行剪枝，本文通过图 2.2 来说明这一过程。

图 2.2 中，极小极大算法求出极小值结点 B 的估值为 8，再搜索到极大值结点 H 的估值为 7，此时可知取极小值的父结点 C 的上界为 7，此时结点 C 对于父结点 A 来说已无意义（C 结点肯定不如 B 结点好），可停止搜索 C-I 分支和 C-J 分支，并略去对 C 的估值；当搜索到极大值结点 K 的估值为 10 时，可知极小值结点 D 的估值的上界为 10，当搜索到极小值结点 N 时得到极大值结点 L 的估值的下界为 14，此时结点 L 对父结点 D 已无意义，可以停止搜索 L-O 分支和 L-P 分支，并忽略对结点 L 的估值。图 2.2 中的虚线即表示剪枝。

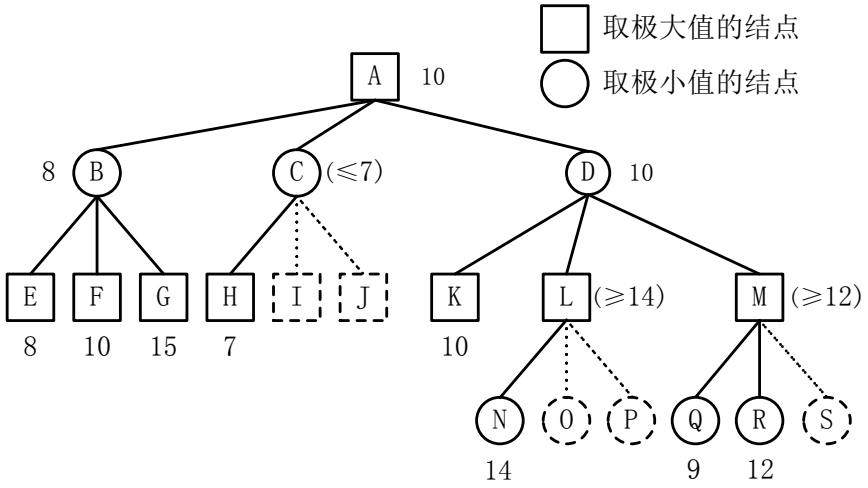


图 2.2 α-β搜索/剪枝算法示例

取极大值一方结点的当前最优值被称为 α 值，取极小值一方结点的当前最优值被称为 β 值，区间[α, β]构成了搜索的窗口。搜索过程中，α 值不减，β 值不增。若取极小值层的结点的 β 值不大于其双亲结点的 α 值，可以停止该最小层结点的搜索过程，这个最小层结点的倒推值就确定为博弈树的 β 值，这个过程被称为 α 剪枝，如图 2.2 中停止对结点 C 的搜索就是 α 剪枝；若取极大值层的结点的 α 值不小于其双亲结点的 β 值，可以停止该最大层结点的搜索过程，这个最大层结点的倒推值就确定为博弈树的 α 值，这个过程被称为 β 剪枝，如图 2.2 中停止对结点 L 的搜索就是 β 剪枝。α 剪枝与 β 剪枝交替进行，就构成了 α-β 搜索/剪枝算法[21]。α-β 搜索/剪枝算法成为了绝大多数主流博弈树搜索算法的基础。

本文对搜索窗口的边界——α 值和 β 值的理解如下：α 值表示当前决策方决策的优势程度，α 值越大表示这种决策对自己越好，对对方越不利。一旦搜索到更高的值，则更新 α 值。β 值表示对方的劣势程度，β 值越小表示对方对应的决

策对对方越好，对自己越不利。因为对方总是倾向找到一个在对方看来不比 β 值更坏的决策，所以搜索过程中如果返回一个大于或者等于 β 值，对方肯定不会选择这种决策，不必搜索。

α-β 搜索/剪枝算法的实现一般采用负极大值的形式，这样在任意一层只进行 β 剪枝，避免了在奇数层和偶数层分别进行 α 剪枝和 β 剪枝。负极大值算法的形式的 α-β 搜索/剪枝算法可用类 C/C++伪代码表示如下：

```
int AlphaBeta (board, depth, alpha, beta)
{
    if(GameOver(board) || depth <= 0) return Evaluation(board);
    best = -∞;
    w = CreateSuccessors(board, p); /*生成 board 下的 w 种着法 p0…pw-1*/
    for(i = 0; i < w; i++)
    {
        MakeMove(board, pi); //执行着法 pi 生成新的局面 board
        value = -AlphaBeta(board, depth - 1, -beta, -alpha); //递归
        RestoreMove(board, pi); //撤销着法 pi 恢复原先局面 board
        if(value >= beta) return value; //β 剪枝
        if(value > best)
        {
            best = value;
            if(value > alpha) alpha = value;
        }
    }
    return best;
}
```

设 v1 < v2，如果以 AlphaBeta(board, depth, v1, v2)的形式调用上述渴望搜索算法，返回的结果 v 可分为 3 种情况[23]：（1）v≤v1，（2）v≥v2，（3）v1 < v < v2。情况(3)显然能得到结点 board 的精确估值，此时结点 board 称为 PV 结点[23,24]；情况（1）被称为“fail-low”，意味着结点 board 的估值同样≤v，此时结点 board 称为 α 结点，也称为 All 结点[23,24]；情况（2）被称为“fail-high”，意味着结点 board 的估值同样≥v，此时结点 board 称为 β 结点，也称为 Cut 结点[23,24]。（1）和（2）两种情况下只能分别得到结点 board 估值的上界和下界。

2.2.3 极小树

α-β 搜索/剪枝算法中。如果博弈树展开结点进行搜索时，每一层下第一个访问的结点就是最优的，则剪枝效率最高，这种情况下的博弈树称为极小树[22]。假设博弈树的平均分支因子为 w，对其进行深度为 d 的搜索，需要估值的极小树的叶子结点数为[22]

$$N_{w,d} = w^{\lfloor d/2 \rfloor} + w^{\lceil d/2 \rceil} - 1 \tag{2.1}$$

图 2.3 显示了一个极小树，其中实线表示需要估值的分支，虚线表示剪枝。极小树是 α-β 搜索/剪枝算法在理论上的最佳情况，实际中很难达到。

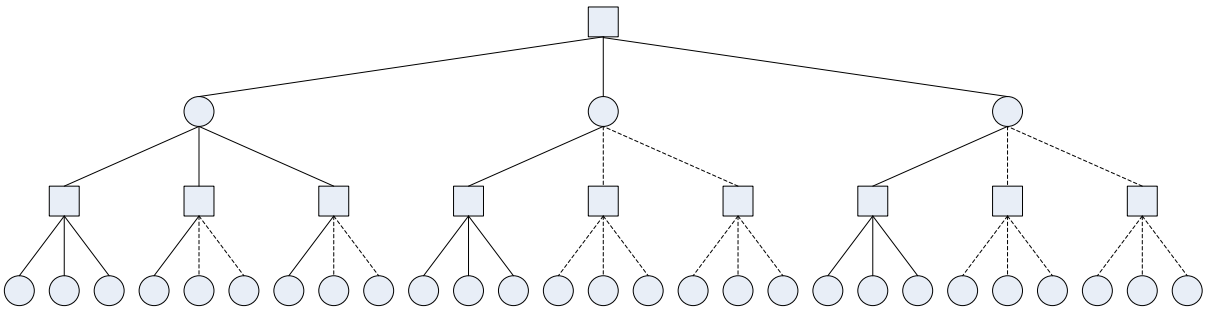


图 2.3 极小树示例

Knuth 和 Moore 提出，在极小树中，根结点是 PV 结点，PV 结点下的第一个孩子结点也是 PV 结点，其他孩子结点为 Cut 结点，Cut 结点的孩子结点为 All 结点，All 结点的孩子结点为 Cut 结点，Cut 结点下只要对第一个孩子结点进行估值，就可以引发对其他孩子结点的剪枝，而 All 结点下要对所有的孩子结点进行估值[24]。极小树中的这三类结点如图 2.4 所示，其中矩形表示 PV 结点，圆形表示 Cut 结点，五角星表示 All 结点，三角形和虚线表示被剪枝的结点及其分支。

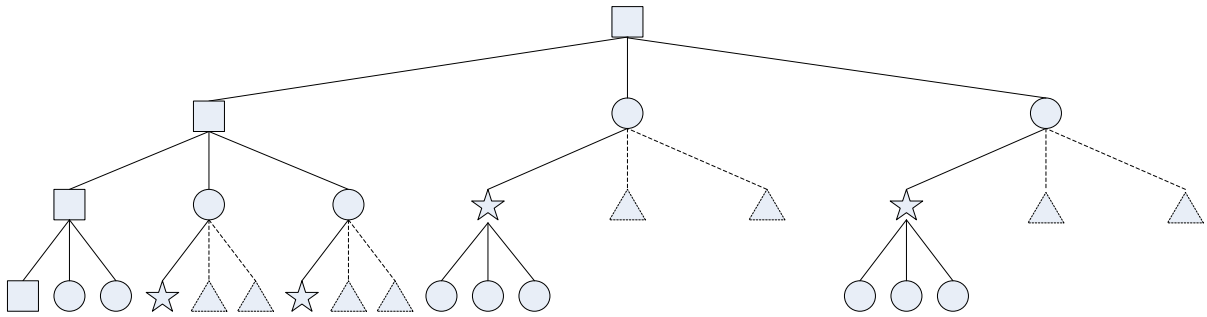


图 2.4 极小树中的PV结点、All结点和Cut结点

2.2.4 置换表

在博弈树的搜索过程中，经常会在不同的路径上搜索到表示相同局面的结点。图 2.5 以中国象棋博弈树为例，结点 F 和结点 G，位于不同的路径，但所表示的局面完全相同。显然，随着搜索深度的递增，这种情况出现的次数也越多。

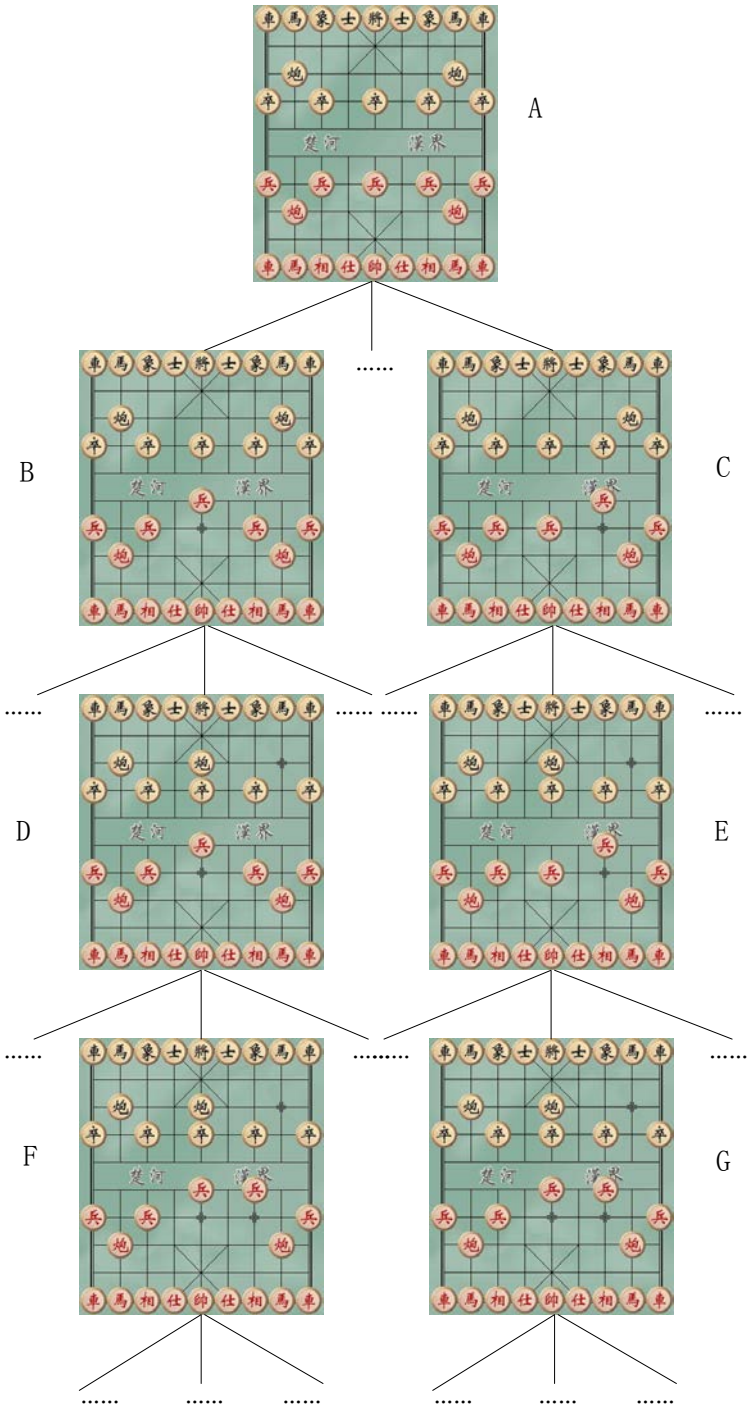


图 2.5 中国象棋博弈树中的重复局面示例

在图 2.3 中，α-β 搜索/剪枝算法完成了对结点 F 的搜索，如果搜索到结点 G 以前，未发生剪枝，需要对结点 G 估值，而 G 对应的局面已经搜索过，如果依旧对 G 展开孩子结点并估值，显然是没必要的重复计算。应该采取相应措施，保存已经搜索过的结点的特征、估值和其他相关信息，当再次搜索到重复局面时，直接查找事先记录的结果即可。

置换表（Translation Table，TT）[25,26]的原理是采用哈希表技术将已搜索的结点的局面特征、估值和其他相关信息记录下来，如果待搜索的结点的局面特征在哈希表中已经有记录，在满足相关条件时，就可以直接利用置换表中的结果。

对一个结点进行估值时，应先查找置换表，置换表命中失败，再对该结点进行搜索。置换表在使用时要及时更新，当计算出一个结点的估值时，应立即将这个结点的相关信息保存到置换表。为了加快处理速度，一般不采用再散列技术，一旦在写入置换表的时候发生冲突，直接覆盖相关的数据项，只要保证在读取操作时避免读取到错误数据即可，因此置换表的设计应使得发生冲突的概率很小。

待搜索的结点，满足如下条件可以直接使用置换表中的记录：

首先，确定哈希函数，将结点对应局面映射为一个哈希值，这个哈希值通常为一个 32 位的整数，根据这个值计算出哈希地址。一种快速而简单的方法就是将这个哈希值对置换表的长度取余数，作为待访问的哈希表元素的地址。

其次，哈希函数可能产生地址冲突，即不同的哈希值映射到了同一地址，上述 32 位的哈希值是不安全的。置换表中的数据项，还应必须包含一个唯一标识局面特征的校验值，这个校验值通常是一个 64 位的整数，从理论上来说，64 位整数也有可能发生冲突，但这个概率极小，在实际中可以忽略不计。使用哈希函数通过哈希值找到置换表数据项的地址之后，再验证该数据项的校验值和待搜索结点对应的局面的特征值是否一致，只有二者一致，才认为查找命中。

再次，对于 PV 结点、All 结点和 Cut 结点，后两种情况并非对应结点的精确估值。因此置换表置换表中的数据项，不仅要记录对应结点的估值结果，还应同时记录这个估值的类型，究竟是一个精确值，还是一个上界值或者下界值。

最后，结点的估值结果与搜索深度有关，搜索深度越深，估值越准确。故置换表中的数据项，还应记录结点对应的搜索深度。如果下次搜索到的局面 A，在置换表中找到了同样的局面 A'，如果 A 对应的搜索深度为 Depth，置换表中 A' 对应的搜索深度为 Depth'，显然只有当 Depth' ≥ Depth 时，才能直接使用置换表中 A'的估值信息，但如果 Depth > Depth'，则置换表中对应结点的估值信息就没有意义了，因为需要再向前搜索几步才能得到一个更准确的值。

因此，置换表中的一个数据项至少应包含如下数据：结点局面的 64 位校验值、搜索深度、估值以及估值的类型[25]。置换表的数据结构可用类 C/C++伪代码定义如下：

```
struct HashItem
{
    unsigned long long checksum; //表示棋盘局面的 64 位校验值
    ENTRYTYPE type; /*估值的类型，EXACT 表示精确值，LOWERBOUND 表示下界值，UPPERBOUND 表示上界值*/
    int depth; //结点对应的搜索深度
    int value; //结点的估值
}HashTable[HashTableSize];
```

一种高效的生成一个特定局面下的 32 位哈希值和 64 位校验值方法就是 Zobrist 哈希方法[25]：创建一个 64 位数组 Z[type][pos]，其值为 type 类型的棋子在棋盘坐标 pos 的一个 64 位随机整数。对棋盘上存在的所有的棋子的随机数求异或，结果保存在 64 位变量 BoardKey 内，就得到了该局面的一个校验值。这样，类型为 type1 的棋子从 pos1 移动到 pos2 时，只要将当前的 BoardKey 值作如下操作：

（1）将要移动的棋子从棋盘上去除，BoardKey = BoardKey ^ Z[type1][pos1]，(“^”表示按位异或运算，下同)；

（2）如果目的位置有对方类型为 type2 的棋子被吃掉，也将其去除，BoardKey = BoardKey ^ Z[type2][pos2]；

（3）将移动后的棋子置入目的坐标，BoardKey = BoardKey ^ Z[type1][pos2]。

使用 Zobrist 哈希方法构造一个局面下的 32 位哈希值的方法与构造 64 位校验值的方法是一样的，只需将 64 位的整型变量改为 32 位即可。Zobrist 哈希方法使结点哈希值的产生可以随着着法的执行或撤销逐步进行。

结合了置换表的 α-β 搜索/剪枝算法可用类 C/C++伪代码表示如下：

```
int AlphaBeta_TT(board, depth, alpha, beta)
{
    value = SearchTT(HashKey, alpha, beta, depth);
    if(置换表命中) return value;
    if(GameOver(board) || depth == 0)
    {
        value = Evaluation(board);
        if(depth == 0) InsertHashTable(HashKey, EXACT, value, depth);
        return value;
    }
    best = -∞; ValueIsExact = 0;
    w = CreateSuccessors(board, p); /*生成 board 下的 w 种着法 p0…pw-1*/
```

```
    for(i = 0; i < w; i++)
    {
        HashKey = MakeMoveWithTT(board, pi);
        value = -AlphaBeta_TT(board, depth - 1, -beta, -alpha);
        HashKey = RestoreMoveWithTT(board, pi);
        if(value >= beta)
        {
            InsertHashTable(HashKey, LOWERBOUND, value, depth);
            return value; //β 剪枝
        }
        if(value > best)
        {
            best = value;
            if(value > alpha)
            {
                alpha = value; ValueIsExact = 1;
                InsertHashTable(HashKey, EXACT, value, depth);
            }
        }
    }
    if(ValueIsExact) InsertHashTable(HashKey, EXACT, value, depth);
    else InsertHashTable(HashKey, UPPERBOUND, value, depth);
    return best;
}
```

置换表的长度（所包含的数据项数量）也是需要考虑的因素。置换表越长，发生地址冲突的概率越小，从而能保存更多的局面信息，置换表命中率越高，算法性能越好。在国际象棋计算机博弈中，置换表的长度每增加 1 倍，命中率约提高 7%[26]。但置换表的长度也并非越大越好，一旦置换表的长度超过物理内存的承受能力，导致使用了硬盘中的虚拟内存，性能反而下降的很快。

置换表数据的更新有两种策略，即深度优先和随时替换[27,28]。深度优先策略不比较校验值，写入置换表的数据对应的搜索深度大于置换表相应数据项的深度，才能替换原有数据；始终替换策略注重实时性，对新估值的局面信息不作任何判断，立即更新置换表中对应的数据项。

有文献提出了“双置换表”的方法[13,27,28]，以提高置换表的命中率。即设置两个置换表，第一层置换表以深度优先策略进行更新，第二层置换表以随时替换

策略进行更新，读取时，先后读取这两层置换表，任意一层置换表命中即算命中。这种双置换表的方法在未引入启发策略的 α-β 搜索/剪枝中获得了较好的效果，但在博弈树倾向良序时性能提高有限，而且要付出多一倍的空间代价。

也有研究着提出置换表中可以不保存叶子结点的估值信息，因为叶子结点数量巨大，并且哈希查找和插入也有一定的时间耗费，对叶子结点进行哈希处理显然增加了算法时间和空间的开销[29]。本文认为应该具体情况具体分析，在第 4 章中将继续讨论这个问题。

2.2.5 启发式搜索策略

启发式搜索策略即为结点排序技术。α-β 搜索/剪枝算法的剪枝效率对同一结点下的孩子结点的排列顺序非常敏感，这些结点的排列越理想，则剪枝越早发生，需要展开和估值的结点数越少，算法效率越高，反之搜索效率越低。虽然无法事先预知哪个结点是最佳结点，但却可以采取一些措施，让能产生剪枝的几率越高的结点越排在前面，以优先搜索，使博弈树尽可能接近最小树。以中国象棋和国际象棋为例，常见的结点排序技术，有吃子启发、置换表启发、历史启发、杀手启发等。

2.2.5.1 吃子启发

吃子启发（Capturing Heuristic）也称为静态启发（Static Heuristic）[34]，仅针对吃子着法。吃子启发通过 MVV/LVA（Most Valuable Victim/Least Valuable Attacker，最大价值的受害者/最小价值的攻击者）思想为吃子着法定义了一种“吃子得分”， 在被吃的棋子有对方其他棋子保护的情况下，一个吃子着法的吃子得分为：对方被吃子价值 MVV 减去己方攻击子价值 LVA；如果被吃的棋子没有保护，则只考虑对方被吃子价值 MVV。将吃子着法按照吃子得分从大到小排列，值越大的越优先搜索。

以中国象棋为例，在开局状态下，最初搜索时，置换表启发、历史启发、杀手启发这些动态启发起的作用很小甚至来不及起作用，此时吃子启发起的作用较大。因此，在着法生成时，考虑首先生成车、马、炮的着法，最后生成帅的着法，往往是很有效的[30]。

有文献认为，吃子着法不多，分析了对应的局面后不用经过搜索，就大致应该知道哪些着法应该首先尝试，很多情况下能立刻得到较好的效果，在各种结点排序技术中应该优先考虑[7,30]。

有些棋类游戏，如五子棋，不存在吃子启发。

2.2.5.2 置换表启发

置换表除了记录结点局面估值信息以供查询外，还能提供启发的功能。置换表启发（Translation Table Heuristic）即在置换表的数据项中，增加一个域，用来保存该局面的估值对应的着法。再次搜索到同样局面的结点时，先查找置换表，如果命中，但因置换表内记录的深度为达到搜索深度要求，或者边界值条件不成立，则估值结果不能被利用，但置换表对应数据项中记录的着法依旧有一定意义，可优先搜索这个着法，多数情况下能更快产生剪枝[30]。

PV 结点对应的着法显然是最佳着法，无疑应该保存到置换表中。β 结点能产生剪枝，其对应的着法多数情况下也是较优的着法，显然也应保存到置换表[30]。至于 α 结点，有文献认为 α 结点的每个着法均返回 α 值，无法确定哪个着法是最好的，因此无需保存 α 结点对应的着法，仅保存估值信息即可[14]；也有文献认为 α 结点结点对应的着法也可以保存到置换表[30]，只要下次搜索时该着法是个合理的着法就可以优先搜索；而著名的中国象棋程序“纵马奔流”则采取了一个更好的方法，称为“低出/高出边界的修正”策略，使得某些 α 结点也存在置换表着法，减少了搜索结点数[13]。

“低出边界的修正”指的是，当某个 α 结点覆盖某个相同局面的置换表数据项时，保留该表项的最佳着法[13]。“高出边界的修正”指的是，当某个 β 结点没能覆盖某个相同局面的置换表项（该表项记录了一个没有着法的 α 结点）时，只把这个 β 结点的截断着法写入该置换表项[13]。

2.2.5.3 历史启发

历史启发（History Heuristic）[31,32]是 J.Schaeffer 于 20 世纪 80 年代提出的博弈树结点排序技术。在搜索过程中，以前搜索到的某一局面下的最佳着法，在其他相差不大的局面下，仍有很大可能也是最佳着法。某个着法被证明是最佳的次数越多，成为相应局面最佳着法的可能性也越大。当一个着法作为兄弟着法中的最佳者，或者能引发了剪枝，则被认为是一个“好”的着法。历史启发的思想是为每个着法设置一个历史得分，每次搜索到一个“好”的着法，该着法的历史得分就累加上一个增量，一般认为，搜索深度越深，搜索结果越可靠，所以这个增量值的大小与相应结点靠近根结点的距离有关，越靠近顶层的结点，这个增量值就越大，越靠近叶子结点则越小。一个“好”的着法被搜索到的次数越多，其历史得分就会较高。生成某一局面下的全部着法时，将所有着法根据历史得分重新排列，历史得分越高的着法越优先搜索，越早引发剪枝的概率越大。

一些文献认为，历史启发仅适用于非吃子着法[7,30]，但也有文献将历史启发

使用在任何类型的着法中[12]。

在众多的启发策略中，历史启发的效果是最好的[14,30]。历史启发不依赖于具体棋类游戏的规则和知识，是一种较为通用的结点排序技术，它付出的代价很少，却往往就能收到极好的效果。在不同的棋类博弈游戏中历史启发的效果差异很大，如在国际象棋、中国象棋中能获得很好的效果，但在五子棋中，历史启发的效果就不如国际象棋和中国象棋。

2.2.5.4 杀手启发

杀手启发（Killer Heuristic）[30,33]可以看作是历史启发的一种特例。所谓“杀手”，指的是同一层中引发剪枝最多的结点。杀手启发的方法是为每一层记录引发剪枝最多的杀手着法（一般是记录 2 个杀手着法），当下次搜索到同一层时，如果杀手着法是合法的着法，则优先搜索杀手着法。一般认为，杀手着法不仅可以是引发剪枝的 β 结点对应的着法，也可以是 PV 结点对应的着法，或者是 α 结点估值最好的非空着法；但也有人认为，杀手着法应该只是引发剪枝的 β 结点对应的着法，才配的上“杀手”这个名称[30]。杀手启发的开销要小于历史启发，但效果也弱于历史启发。

2.2.5.5 各种启发策略的组合与顺序

吃子启发不依赖于以往的搜索结果，也被称为静态启发（Static Heuristic），而置换表启发、历史启发、杀手启发依赖于以往的搜索结果，属于动态启发（Dynamic Heuristic）[14,30]。多种启发策略混合使用效果更好。如果将多种启发策略混合使用，要考虑策略顺序。

不同的棋类博弈游戏，最好的策略顺序会有所不同。以中国象棋计算机博弈为例，多数文献为一种较好的策略顺序是[7,30,34]：

（1）置换表启发：若置换表中的着法不为空，则首先搜索置换表中的着法；

（2）吃子启发：生成所有吃子着法，按照 MVV/LVA 值从大到小排序；

（3）杀手启发：如果杀手着法是合理的着法，再搜索杀手着法；

（4）历史启发：生成所有非吃子着法，着法按照历史得分从大到小排序。

2.2.6 PVS 算法

博弈树结点的估值一般都是取整数，如果在 α-β 搜索/剪枝算法中，采用窗口[v, v + 1]对结点 p 搜索估值，这种极端情况称之为极小窗口探测，也称为空窗探测。极小窗口探测的速度很快，但目的不是找到结点 p 的精确估值，而是探测出结点 p 的估值是≤v，还是≥v+1，因为这种返回值只有这两种情况。

PVS（Principal Variation Search，主要变例搜索）算法[35]也称为 NegaScout 算法，是 A.Reinefeld 在 20 世纪 80 年代中期提出的。PVS 算法利用极小窗口探测引发大量剪枝，它基于如下假设：假定第一个孩子结点是最佳着法，其后续则次之，直到另一结点被证明是最佳的。第一个分支以一个完整窗口[α, β]搜索最佳移动，并得到一个位于该窗口区间内的值 v，后续的分支则以一个极小窗口[v, v + 1]进行搜索，这种极小窗口探测将引发大量的剪枝。如果猜测被证实不准确，发生了 fail-high，就需要以[v + 1, beta]为窗口重新搜索；如果猜测不准确发生了 fail-low，说明这个结点不如已有的最佳结点，不必继续搜索。

图 2.6 描述了负极大值形式的 PVS 算法的剪枝过程，其中粗实线为主要变例，虚线为剪枝。深度优先搜索过程中以全窗口[-∞, +∞]搜索完以结点 D 为根的子树后，D 获得了一个值 6，回溯到结点 B 后再以极小窗口[5, 6]搜索结点 E，E 将窗口边界值变号，以极小窗口[-6, -5]搜索到结点 K 时，K 的值-7 小于窗口下界-6，发生 fail-low，从而引发了对结点 L 的剪枝，于是 E 获得了下界值 7，B 最终获得了值-6，回溯到 A 后，以极小窗口[-7, -6]搜索以结点 C 为根的子树，搜索到结点 M 时，M 的值-9 小于 F 传递给 M 的窗口下界-7，发生 fail-low，又引发了对结点 N 和结点 O 的剪枝，结点 F 获得了下界值 9，再以极小窗口[6, 7]搜索以结点 G 为根的子树，G 以极小窗口[-7, -6]搜索结点 P，因 P 的值为-4，大于窗口上界-6，发生了 fail-high，再以窗口[-6, +∞]对 P 重新搜索。最终结点 A 获得估值 6。

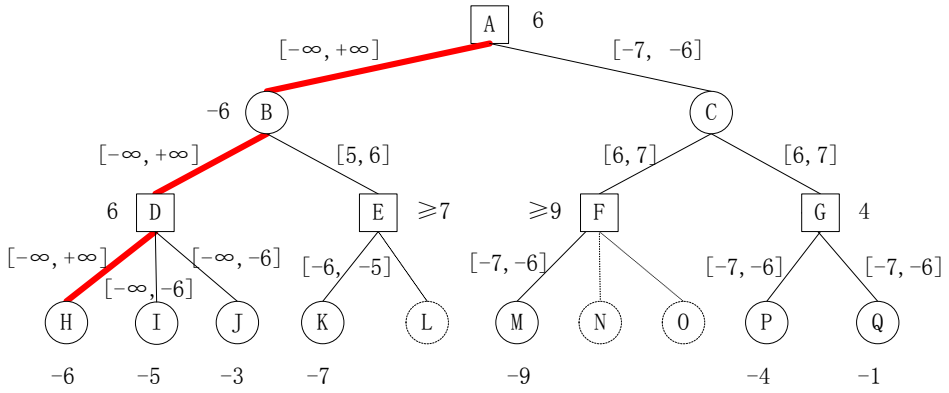


图 2.6 PVS算法的负极大值形式示例

PVS 算法的具体过程，可以用类 C/C++伪代码描述如下：

```
int PVS (board, depth, alpha, beta)
{
    best = -∞, PVFlag = 0;
    if(GameOver(board) || depth == 0) return Evaluation(board);
```

```
        w = CreateSuccessors(board, p); /*生成 board 下的 w 种着法 p0…pw-1*/
        for(i = 0; i < w; i++)
        {
            MakeMove(board, pi); //在棋盘局面 board 下执行着法 pi
            if(PVFlag) //如果已经获得主要变例的估值
            {
                value = -PVS(board, depth - 1, -alpha - 1, -alpha);/*空窗探测*/
                if(value > alpha && value < beta ) value = -PVS(board, depth - 1,
-beta, -value); //较大窗口探测
            }
            else value = -PVS(board, depth - 1, -beta, -alpha); /*全窗口探测*/
            RestoreMove (board, pi); //恢复棋盘局面
            if(value >= beta) return value; //剪枝
            if(value > best)
            {
                best = value;
                if(value > alpha) { PVFlag = 1; alpha = value; }
            }
        } //end of for
        return best;
    }
```

PVS 算法已经从理论上被证明在平均情况下比传统的 α-β 搜索/剪枝过程的效率高出很多，很多实验也证实了这一点[12,35]，特别是当博弈树同一层的结点有序性较强时，PVS 算法拥有很大的优势。

2.2.7 空着裁剪

空着裁剪（Null Move Pruning）[36]是 Chrilly Donninger 于 1993 年提出一种强大的前置剪枝策略。轮到己方走棋时，己方放弃走棋，让对方继续走棋（即对方连走两步棋），如果此时己方在较浅层的搜索中依旧具有优势，得到的估值大于或等于 β 值，这个分支就没有必要进行完全深度的搜索了。

假设完全搜索时的深度为 depth，空着搜索的目的只是为了验证自己的优势，所以一般来说，没必要在执行空着之后做深度为 depth - 1 的搜索，而是做深度为 depth - 1 - R 层的较浅搜索即可。多数情况下，R 值取 2 较为安全，如果想让搜索更快，在 depth 值较大的情况下也可让 R 为 3[36]。也有人提出了 “自适应

空着裁剪（Adaptive Null Move Pruning）”[37]，令 R 值随着最大搜索深度而变化。

空着裁剪只要探测是否能引发剪枝，所以只要使用极小窗口[β - 1, β]进行搜索即可[14]。空着裁剪的效果非常可观，目前不少棋类计算机博弈程序使用到了空着裁剪（除了空着没有意义的棋类）。但空着裁剪需要谨慎使用，这是因为：

（1）在残局中，可能会出现不走棋反而比走任何一步棋要好的局面；

（2）发生将军时要慎用空着裁剪，如果此时使用空着裁剪，可能造成被将军一方既避免了将军，又吃掉了对方用来将军的棋子，造成搜索不准确；

（3）显然，空着不能连续两次使用，即一方使用空着后，另一方再使用空着是没有意义的；

（4）在少数情况下（如搜索深度 depth 较小时），空着裁剪可能会忽视重要路线的冒险策略，造成后继搜索对某种较好的冒险着法的漏判。

2.2.8 MTD(f)算法

MTD(f)算法（Memory-enhanced Test Driver with f and n）[38,39]是 Aske Plaat 于 1996 年提出博弈树搜索算法，有较高的效率。算法开始时，给定一个初始猜想估值 f，一个上界 upperbound 和一个下界 lowerbound。初始估值要尽可能的靠近最优结点的估值，上界和下界要保证能把这个最优结点的估值包含在内。该算法多次调用 α-β 搜索/剪枝算法以极小窗口[f - 1, f]进行探测，返回估值 value，如果发生 fail-high，则更改上界 upperbound 为 value，如果发生 fail-low，则更改下界 lowerbound 为 value，接着令 value 的值为新的猜想估值，即 f = value，继续在更新后的上下边界搜索……反复如此，上下边界构成的范围不断缩小，当下边界的值大于或等于上边界时，搜索完成。

MTD(f)算法可用类 C/C++伪代码描述如下：

```
int MTD_f(board, f, depth)
{
    value = f; //初始猜想估值存入 value
    upperbound = +∞; //上界初始值为正无穷大
    lowerbound = -∞; //下界初始值为负无穷大
    while(lowerbound < upperbound) //当下界小于上界时，不断搜索
    {
        if(value == lowerbound) beta = value + 1;
        else beta = value;
        value = AlphaBeta (board, depth, beta - 1, beta); //空窗探测
```

```
            if(value < beta) upperbound = value; /*若 fail-high 则更改上界为
value*/
            else lowerbound = value; //若 fail-low 则更改下界为 value
        }
        return value;
    }
```

空窗探测在 PVS 算法和 MTD(f)算法中发挥了巨大的作用，多数人认为窗口变小能引发更多的剪枝。但张明亮等人指出，影响博弈树窗口搜索效率的首要原因是窗口位置，次要原因才是窗口大小[40]。

2.3 并行搜索策略

充分利用多处理器资源，对博弈树的搜索进行并行处理，也是提高博弈树搜索效率的重要手段。下面介绍几种常见的并行搜索策略。

2.3.1 Tree Splitting 并行策略

Tree Splitting（树分裂）[41]并行策略最为简单，它的思想是，多个处理器组成一个树形结构，每个处理器对应树中的一个结点，位于叶子结点处的处理器执行串行的 α-β 算法，非叶子结点处的处理器执行主算法。算法开始时，博弈树根结点由处理器树的根结点处理，根结点处理器产生博弈树中相应的孩子结点，并把它交给下一层的子处理器处理。孩子结点对应的处理器继续产生下一层的孩子结点……反复如此，直到某个结点交给叶子结点处的处理器进行处理，再将返回的估值层层向上级处理器传递，最终根结点处的处理器获得待搜索结点的估值。如果某个叶子结点位置的处理器更新了 α 值，将更新结果向上传递，上层处理器通知它的子孙处理器更新搜索的窗口边界。

2.3.2 PVSplitting 并行策略

PVSplitting（Principal Variation Splitting，主要变例分裂）[24,41]并行策略的思想是，将一个处理器作为主处理器，其余处理器为从处理器。主处理器从待搜索的博弈树根结点开始，从最左侧展开每一层的首个 PV 结点，直到最大搜索深度 MaxDepth 层的最左边的结点被展开，将估值传回第 MaxDepth-1 层的父结点，再和从处理器一起对剩余的结点进行并行搜索，将每个子树的搜索交给一个处理器处理。将搜索到的估值再传回给第 MaxDepth-2 层的父结点，所有的处理器再对剩余的结点进行并行搜索，将搜索到的估值再传回给第 MaxDepth-3 层的父结点……反复如此，直到根结点下剩余结点的并行搜索结束，传回最终的估值，搜索结束。并行搜索过程中，如果某个处理器修改了窗口边界，其他处理器随之更

新窗口边界。PVSplitting 策略比 Tree Splitting 策略性能优越很多，博弈树越接近最小树，PVSplitting 策略的效果越好。

2.3.3 DTS 并行策略

DTS（Dynamic Tree Splitting，动态树分裂）[11,15,24]并行策略的思想与 PVSplitting 有类似之处，但每个处理器的地位相同。搜索开始时，先让一个处理器从根结点开始向下沿最左侧的分支做浅层的 α-β 搜索，初始化博弈树，将已知的博弈数子树划分到不同的“块”中，每个“块”复制到共享内存中（也可以利用置换表），这个处理器将这些“块”中的子树的搜索工作分配其他闲置的处理器。当某个处理器完成了自己的搜索线程处于闲置状态时，对其他所有的处理器广播一个消息，询问其他处理器是否需要自己协助搜索。如果收到另外的处理器请求协助工作的回应，就协同这个处理器一起工作，以加快搜索速度。当某个处理器允许其他处理器协同工作，必须将子树的信息复制到共享内存中以供对方读取。DTS 策略使所有的 CPU 始终处于饱和工作状态，性能好于 PVSplitting 策略，著名的国际象棋搜索引擎 Crafty 和台湾的中国象棋搜索引擎“深象”就使用了 DTS 并行策略，都是利用 Unix 下的 PThread 模型实现的[24]。但是 DTS 对内存要求很高，实现难度也较大。

2.4 小结

本章介绍了博弈树和极小树的概念，以及极小极大值搜索和负极大值搜索的原理，归纳性的介绍了博弈树搜索算法的基础——α-β 搜索/剪枝算法，以及由其衍生的两个主流的博弈树搜索算法——PVS 算法和 MTD(f)算法，并以中国象棋和国际象棋的计算机博弈为例，穿插介绍了置换表、吃子启发、置换表启发、历史启发、杀手启发、空着裁剪等启发策略和优化措施，最后介绍了 Tree Splitting、PVSplitting、DTS 三种并行策略。

第3章 OpenMP 并行模型介绍

OpenMP 是基于共享存储环境的一种多线程并行程序设计标准，有着良好的易用性和可移植性，适用于在多核 PC 中使用进行并行程序设计。

3.1 OpenMP 并行编程标准

3.1.1 OpenMP 简介

OpenMP[20,42,43]是一种基于共享内存多处理器体系结构的多线程并行编程模型，也是在共享存储下编写并行程序的一组 API。OpenMP 的规范由 SGI 提出，获得了微软、Sun、Intel、IBM、HP 等软硬件业界巨头的认可和支持，目前支持的语言是 C/C++和 Fortran。

OpenMP 1.0 标准诞生于 1997 年。2005 年，OpenMP 的结构审议会（Architecture Review Board，ARB），推出了 OpenMP 2.5 标准。2008 年，OpenMP 3.0 标准被推出[20]，但目前使用最广泛的还是 OpenMP 2.5 标准。

OpenMP 中，CPU 中的多个处理单元（核）共享同一个内存设备，所有的 CPU 处理单元访问相同的内存空间编址，这些处理单元通过共享的内存变量进行通信。OpenMP 编程方式十分方便，现已成为一个工业标准，支持多种平台，包括大多数的类 UNIX 系统和几乎所有的 Windows NT 系统等。OpenMP 具有开发简单、抽象度高、通用性好、可扩充性和可移植性强、支持并行的增量开发和数据并行等优点[42,43]，广泛应用于多核 PC 环境下的并行程序设计。

3.1.2 OpenMP 并行执行模式

OpenMP 是基于多线程的，并行执行模型采用 Fork-Join 的形式，以线程为基础，通过编译指导指令显示地指定程序的并行部分。程序开始时只有一个主线程，主线程一直串行执行，直到遇到第一个并行域才开始并行执行，执行过程如下[20,42,43]：

Fork：Fork 表示创建新线程或者唤醒处于等待状态的线程。主线程创建一系列并行的线程，之后，并行域中的代码在不同的线程中并行执行，主线程和其他并行线程同时工作；

Join：Join 表示表示多个线程的汇合。当并行域中的代码执行完之后，并行线程退出或者阻塞，不再工作，控制流程回到单独的主线程来执行。

这种从串行执行到并行执行的改变，只需要少量 OpenMP 编译指导指令就可以完成。OpenMP 的 Fork-Join 并行执行模式，如图 3.1 所示。

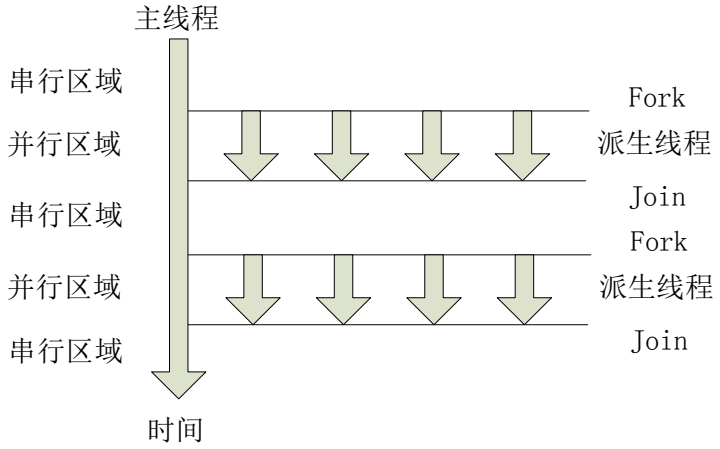


图 3.1 OpenMP的Fork-Join并行执行模式

从图 3.1 可以看出，主线程在运行的过程中，遇到并行指导指令，会根据环境变量生成多个派生线程，进入 Fork 模式，多个派生线程与主线程共存，同时运行，当主线程和多个派生线程遇到一个隐含的同步操作（barrier）后，汇合成原有的线程继续运行，进入 Join 模式。

OpenMP 使用的是一种并非十分严格的共享内存模型，所有的并行线程都可以访问指定的内存变量。按照变量对线程的可见性进行分类，OpenMP 中的变量可以分为共享变量（shared variable）和私有变量（private variable）[20,42,43]。某个线程的私有变量只能被该线程访问，对其他线程来说是不可见的。一个线程的私有变量可以在某个并行代码区域内定义，也可以通过相应的编译指导指令来定义。如果线程 A 派生出了线程 B，还可以通过相应的编译指导指令使线程 B 的私有变量的值初始化为线程 A 的同名的变量的值。并行代码区域内，每个并行线程操作的私有变量相互独立，虽然同名，但每个线程都有自己的拷贝。而共享变量是被并行代码区域内所有的线程所共享的，当多个线程对同一共享变量进行写操作时，会产生数据竞争，其结果是危险的、不可预知的。并行代码区域内对共享变量进行写操作的代码称为关键代码，对关键代码应采用某种保护措施，避免数据竞争。

OpenMP 中，并行线程的私有变量和共享变量可以复制到 CPU 的高速缓存（cache）中，这样线程可以在高速缓存内的“临时视图”中进行读写，加快了读写速度，但也带来了副作用，即内存中的数据与高速缓存的临时视图中的数据不一致以及多个 CPU 处理单元的临时视图不同的问题。这就需要在 OpenMP 中引入相应机制，及时更新内存中的变量值，确保二者的一致性。

3.2 OpenMP 编程模型

OpenMP 是一组编译指导指令、库函数和环境变量的集合。使用传统的 Win32/64 API 和 MFC 直接进行多线程编程非常复杂和繁琐，需要程序员自己处理线程间的同步、互斥、死锁等具体问题。但在 OpenMP 模式下，程序员通过使用编译指导指令、库函数，设置环境变量来设定 C/C++或 Fortran 程序的可并行部分，并在必要之处加入同步和互斥，编译平台可以自动将程序进行并行化。若编译平台不能识别 OpenMP 的编译指导语句，则将这些语句当作注释，程序依旧可以以串行方式执行。这样就降低了编写并行程序的复杂性，将程序员从繁冗复杂的细节中解脱出来，使得编写并行编程变得简单、正确和高效。

3.2.1 OpenMP 的编译指导指令

OpenMP 中的编译指导指令包含了 OpenMP 并行化编程的一些语义，显式指明如何对应用程序进行并行化处理。OpenMP 的编译指导指令的格式为[20,42,43]：

#pragma omp 指令 [子句[[,]子句]...] 换行符

OpenMP 的所有编译指导指令均以# pragma omp 开始，后面跟具体的功能指令，指令之后还可以包含相关的子句。这些的子句是可选的，子句给出了相应的编译指导指令的参数，影响这些指令的具体行为。每一个编译指导指令都有一系列适合它的子句，其中有 5 个编译指导语句（master、critical、flush、ordered、atomic）不能跟相应的子句。

OpenMP 2.5 中，常用的编译指导指令如下[20,42,43]：

parallel：用在一个复合语句结构块之前，表示这段代码将被多个线程并行执行；

for：用于 for 循环语句之前，表示将循环计算任务分配到多个线程中并行执行，以实现任务分担，程序员必须保证每次迭代之间无数据相关性；

parallel for：parallel 和 for 指令的结合，也是用在 for 循环语句之前，表示 for 循环体的代码将被多个线程并行执行，它同时具有并行域的产生和任务分担两个功能；

sections：用在可被并行执行的代码段之前，用于实现多个结构块语句的任务分担，可并行执行的代码段各自用 section 指令标出；

parallel sections：parallel 和 sections 两个语句的结合，类似于 parallel for；

single：用在并行域内，表示一段只被单个线程执行的代码；

critical：用在一段代码临界区（关键代码段）之前，保证每次只有一个线程进入；

flush：保证各个线程的数据影像的一致性；

barrier：用于并行域内代码的线程同步，线程执行到 barrier 时要停下等待，直到所有线程都执行到 barrier 时才继续往下执行；

atomic：用于指定一个简单的数据操作需要原子性地完成；

master：用于指定一段代码必须由主线程执行；

threadprivate：用于指定一个或多个变量是线程专用。

相应的 OpenMP 编译指导指令的子句如下：

private：指定一个或多个变量在每个线程中都有它自己的私有副本；

firstprivate：指定一个或多个变量在每个线程都有它自己的私有副本，这些副本进入并行域时，以主线程中的同名变量的值作为初值；

lastprivate：是用来指定将线程中的一个或多个私有变量的值在并行处理结束后复制到主线程中的同名变量中，负责拷贝的线程是 for 或 sections 任务分担中的最后一个线程；

reduction：用来指定一个或多个变量是私有的，并且在并行处理结束后这些变量要执行指定的归约运算，并将结果返回给主线程同名变量；

nowait：指出并发线程可以忽略其他制导指令暗含的路障同步；

num_threads：指定并行域内的线程的数目；

schedule：指定 for 任务分担中的任务分配调度类型；

shared：指定一个或多个变量为多个线程间的共享变量；

ordered:用来指定 for 任务分担域内指定代码段需要按照串行循环次序执行；

copyprivate：配合 single 指令，将指定线程的专有变量广播到并行域内其他线程的同名变量中；

copyin：用来指定一个 threadprivate 类型的变量需要用主线程同名变量进行初始化；

default：用来指定并行域内的变量的使用方式，缺省是 shared。

3.2.2 OpenMP 的库函数

OpenMP 的库函数是一组控制并行线程的行为的 API。OpenMP 2.5 中，常用的库函数如下[20,42,43]：

omp_in_parallel：判断当前是否在并行域中；

omp_get_thread_num：返回线程的唯一编号；

omp_set_num_threads：设置并行域中的线程数量；

omp_get_num_threads：返回当前并行区域中的线程数量；

omp_get_max_threads：获取并行域可用的最大线程数量；

omp_get_num_procs：返回系统中 CPU 处理单元的数量（核数）；

omp_get_dynamic：判断是否支持动态改变线程数量；

omp_set_dynamic：启用或关闭线程数目的动态改变；

omp_get_nested：判断系统是否支持并行嵌套；

omp_set_nested：启用或关闭并行嵌套；

omp_init(_nest)_lock：初始化一个（嵌套）锁；

omp_destroy(_nest)_lock：销毁一个（嵌套）锁；

omp_set(_nest)_lock：（嵌套）加锁操作；

omp_unset(_nest)_lock：（嵌套）解锁操作；

omp_test(_nest)_lock：非阻塞的（嵌套）加锁。

使用 OpenMP 的库函数在其头文件 omp.h 中定义。和编译指导语句不同的是，编译指导语句在不支持 OpenMP 的编译平台上可以被自动忽略，而库函数要求编译平台必须支持 OpenMP 才能使用。

3.2.3 OpenMP 的环境变量

OpenMP 的环境变量用以设置和获取执行环境相关的信息，可以在一定程度上控制 OpenMP 程序的行为。OpenMP 2.5 中的环境变量如下[20,42,43]：

OMP_SCHEDULE：用于 for 循环并行化后的调度，它的值就是循环调度的类型；

OMP_NUM_THREADS：设置并行域中的线程数；

OMP_DYNAMIC：用于确定是否允许动态设定并行域内的线程数；

OMP_NESTED：用于确定是否支持并行嵌套。

3.3 OpenMP 编程的关键问题

使用 OpenMP 模型进行多核并行程序设计，首先要保证并行算法的正确性，如果算法并行化后提供的结果不正确或者错误无法容忍，性能提升再多也是没有意义的，必须设计能够提供正确结果或者错误在可容忍范围内的算法。除了正确性之外，还要考虑一些新问题。

3.3.1 任务分解与循环并行化问题

显然，在并行化的收益大于代价的前提下，应该让算法中并行部分尽可能多。找到算法的可并行部分，将这部分的计算任务进行分解，分解后的每个子任务可以并行。一般而言，算法的很大一部分耗时集中在循环中，对循环进行并行化处理非常关键。

OpenMP 2.5 中进行循环并行化处理是有一定限制的，主要体现在[42,43]：

1．仅限于 for 循环，其他循环需要改写成 for 循环的形式；

2．for 循环中的循环控制变量必须是带符号整形，比较操作必须包含循环控制变量，并且是<、<=、>、>=的形式，循环的步长必须是一个不变的整数，其目的是在进入循环前，编译器能预知循环的次数；

3．循环必须是单入口、单出口的；

4．必须具备循环无关性（loop-independent dependence），即某一次循环的结果不能依赖于其前面的循环的结果。

3.3.2 线程过少或过多问题

OpenMP 中，程序的并行线程数，应等于 CPU 处理单元数[20,43]。如果并行线程太少，无法利用多核资源将加速比最大化。但如果并发线程太多（如超过了 CPU 核数的两倍），一方面线程的启动、终止、切换、等待等方面的开销大增，另一方面，过多的并发线程数也会导致共享的存储资源的开销增大。

3.3.3 数据竞争与锁操作问题

OpenMP 中，多个并行线程对共享数据的写操作，会发生数据竞争，数据最终写入的值依赖于这些并发线程的时间特性，导致了不正确、不可预知的结果。

解决数据竞争的最简单方法是进行锁操作或者指定关键代码段[20,42,43]。可以通过锁操作或者指定关键代码段，将对某个共享变量的更新操作的代码保护起来，使得每次最多只有一个线程执行这段代码，从而保证数据的完整性和正确性。

在满足条件的情况下，还可以进行数据分解，即数据本地化处理，将数据分解成若干独立的数据子块，每个线程处理其中的一个或者多个子块[42]。

3.3.4 线程调度问题

OpenMP 提供了静态调度（static scheduling）、动态调度（dynamic scheduling）、指数调度（guided scheduling）、运行时调度（runtime scheduling）四种调度策略[20,42,43]。静态调度将所有循环迭代划分成相等大小的块静态分配给线程，块的大小由语法指定。动态调度也是将所有循环迭代划分成相等大小的块，但使用一个队列，当线程可用时，为其分配由块大小指定数量的循环迭代，线程执行完成后，将从队首取出下一组迭代。指数调度和动态调度类似，但块大小刚开始较大，然后逐渐减少，从而减少了线程用于访问任务队列的时间。运行时调度是在运行时通过环境变量来确定使用上述三种调度策略中的某一种。静态调度的额外开销最小，但最不利于负载平衡；动态调度最有利于负载平衡，但额外开销最大。

3.3.5 伪共享问题

CPU 读取 Cache 时是以行为单位读取的，如果两个硬件线程的两个不同的内存变量位于同一 Cache 行里，那么当两个硬件线程同时在对各自的内存变量进行写操作时，将会造成两个硬件线程写同一 Cache 行的问题，此时虽然这两个内存变量没有出现数据竞争，但对应的 Cache 行出现了数据竞争，造成性能下降。

伪共享问题在 OpenMP 编程中经常可以碰到的，比如两个并行线程同时写一个数组的相邻部分，或者写两块相邻的内存。解决伪共享问题，一种方法是尽可能使用专用数据，另一种方法是利用编译器的优化功能或者优化内存分配算法来避免不同的线程操作的私有变量映射到同一 Cache 行[42,43]。

3.4 多核并行算法的性能评价

并行算法的性能评估包含的内容较多，出了时间/空间复杂度之外，主要还包括加速比、并行效率、可扩展性等[8]。

3.4.1 加速比

加速比（speed up）[8,42]是评价多核并行算法的最重要的性能指标。对于一个指定的算法，加速比被定义为同一个任务在串行处理器系统和并行处理器系统中运行需要的时间的比值，用来衡量并行系统或程序并行化的性能和效果。如果用 S(p)来表示算法在使用 p 个 CPU 处理单元的条件下并行的加速比，该算法在串行模式的最优执行时间为 T(1)，在使用 p 个 CPU 处理单元进行并行运行的执行时间为 T(p)，则加速比可以表示为

$$S(p)=\frac{T(1)}{T(p)} \tag{3.1}$$

可见，加速比反映了在相同的问题规模下，并行算法的执行速度相对相应的串行算法的速度加快多少倍，直接体现了在并行计算机上执行并行算法求解实际问题所能获得的好处。

3.4.2 并行效率

并行效率（parallel efficiency）[8]反映了并行算法中CPU处理单元的利用程度。如果算法在串行模式的最优执行时间为T(1)，在使用p个CPU处理单元进行并行运行的执行时间为T(p)，则并行效率可以表示为

$$E(p)=\frac{S(p)}{p}=\frac{T(1)}{p\cdot T(p)} \tag{3.2}$$

对于一些并行算法而言，可能会出现较少 CPU 数或较少核数的环境下运行拥有较为理想的并行效率，但是在较多 CUP 数或较多核数的环境下运行时并行效率下降的情况，设计并行算法时应尽可能使算法在 CUP 数或核数增加时，减少并行效率下降的幅度。

3.4.3 可扩展性

可扩展性（scalability）[8]是指并行算法有效利用 CPU 处理单元增加的能力的一个度量。可扩展性和并行效率有关，随着多处理器或者 CPU 处理单元的增加，如果并行效率基本不变，或者并行效率下降幅度相对不大，则该并行算法在并行计算机上的可扩展性较好，反之则该并行算法的可扩展性较差。

随着多处理器或者 CPU 处理单元的增加，并行算法并行效率下降达到什么程度，其可扩展性就算“较差”呢？显然，对于不同的实际应用，这个阀值是不同的，不能一概而论。影响一个并行算法的并行效率和可扩展性的因素很多，评价标准和具体度量也是不同的。一个可扩展性良好的算法必然有良好的加速比和并行效率。

3.5 小结

本章对 OpenMP 标准做了综述，并重点介绍了 OpenMP 多线程并行编程方法。OpenMP 是一种基于共享内存的多线程编程模型，并行执行模型采用 Fork-Join 的形式，具有良好的易用性、高度抽象性、可扩充性和可移植性。OpenMP 的组成要素包含编译指导指令、库函数与环境变量，使用 OpenMP 编程，应充分考虑任务分解与循环并行化问题、线程数量设置问题、数据竞争与锁操作问题、线程调度策略、伪共享问题等常见问题。评价 OpenMP 下的并行算法的性能指标，包含加速比、并行效率、可扩展性等。

第4章 混合 PVS 算法及在 OpenMP 下的并行化

第 2 章中介绍的博弈树的各种搜索算法及优化措施，各有特点，各有所长：有的侧重估值的精确性，如 α-β 搜索/剪枝算法；有的侧重空窗探测以减少展开的估值结点数，如 PVS 算法、MTD(f)算法；有的侧重结点搜索的优先级，如历史启发、吃子启发、杀手启发；有的侧重“以空间换时间”，如置换表。要开发一个拥有较高棋力和较快走棋速度的棋类游戏的博弈树搜索引擎，需要结合实际情况，选择一个高效的搜索算法，并尽可能引入更多的对实际问题有利的优化措施和搜索策略，将多种方法结合使用，提高搜索引擎的智能和思考速度。随着硬件的发展，在多核微机环境下的博弈树搜索引擎，还应当尽可能利用硬件资源，如 CPU 和内存资源，将博弈树搜索算法进行并行化设计，以降低搜索需要的耗时，提高计算机的思考速度。

4.1 一种混合 PVS 算法

在中国象棋计算机博弈中，α-β 搜索/剪枝算法、历史启发、杀手启发、置换表、PVS 算法、MTD(f)算法这些搜索算法、策略和优化措施完全可以适用于中国象棋计算机博弈程序。在条件满足的情况下（如没有出现将军局面，并且不是残局），也可以在算法中融合空着裁剪。

这里针对多核 PC 环境下的中国象棋计算机博弈系统，设计一个适合中国象棋的博弈树搜索算法。鉴于 PVS 算法与 MTD(f)算法是目前两大主流的博弈树搜索算法，而从本章后述的表 4.1 可以看出，在本文开发的中国象棋计算机博弈程序中，PVS 算法的效率略优于 MTD(f)算法（实验过程和实验结果详见本章第 3 节），故本文采用 PVS 算法作为搜索引擎的主要算法框架，并结合历史启发、杀手启发、置换表这些优化措施，以利于搜索引擎看的更深、更快、更准。

4.1.1 PVS 算法框架

负极大值形式的 PVS 算法是本文设计的中国象棋计算机博弈系统搜索引擎的算法框架，融入了置换表、各种启发策略、空着裁剪等优化措施。算法框架对应的活动图，如图 4.1 所示。

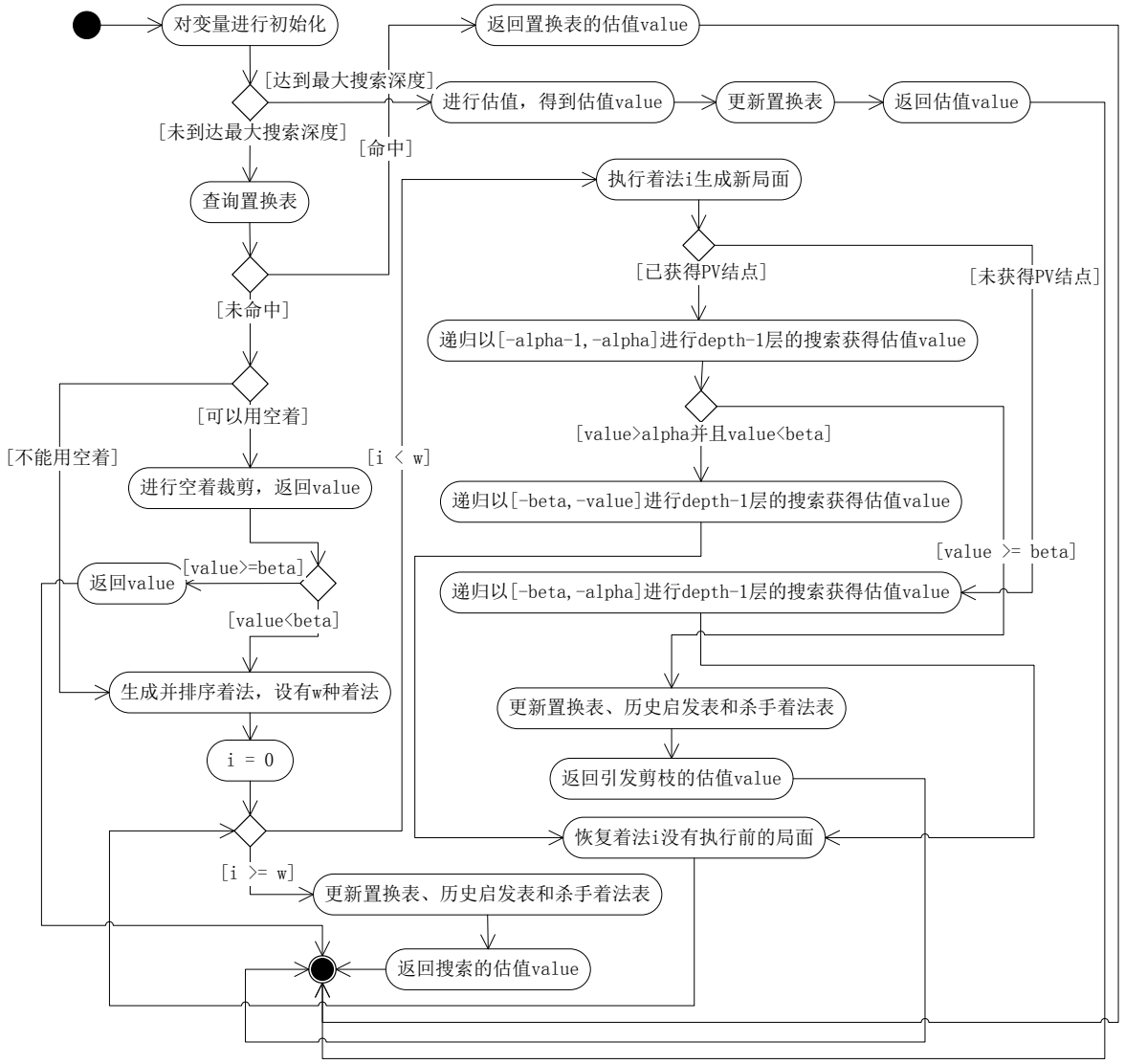


图 4.1 混合PVS算法的基本框架活动图

该混合 PVS 算法的参数中，有 3 个参数 depth、alpha、beta 分标表示搜索深度、下界值、上界值。并引入变量（对象）GoodMove、BestMove 和 best 分别表示搜索到的“好”的着法、最大搜索深度下的最佳着法及其估值，将 best 初始化为-∞。

算法首先判断搜索是否到达了叶子结点或者最大搜索深度（depth 为 0），如果到达最大搜索深度，则对当前结点 board 调用估值函数，将当前结点 board 的精确值 value 以及对应的搜索深度 depth 保存到置换表 HashTable 中，返回估值 value。

如果搜索未到达叶子结点，也没到达最大搜索深度，则根据当前结点 board 及其搜索深度 depth 查找置换表 HashTable，如果置换表命中，则返回置换表中查询到的估值 value。

如果空着裁剪的条件满足，算法尝试空着裁剪。执行空着后，递归以极小窗口[-beta, -beta + 1]进行深度为 depth - 1 - R 的搜索，撤销空着后，如果发现返回的估值 value 大于或等于 beta 值，则返回 value。

如果空着裁剪失败，算法生成当前结点 board 下当前走棋方的所有着法，p_0、p_1、...、p_{w-1} 共 w 个，放在着法数组 p 中，并使用着法排序技术，对 p_0、p_1、...、p_{w-1} 排序。顺序扫描排序后的着法数组 p，对每一个着法 p_i，执行下列操作：

执行着法 p_i，生成孩子结点替换 board，即 board 为结点执行着法 p_i 后对应的局面。

如果 i 为 0，将这个结点 board 作为初始的 PV 结点，递归以完整的窗口[-beta, -alpha]进行深度为 depth - 1 的搜索，搜索的估值结果保存在变量 value 中。

如果 i > 0，说明已经获得了 PV 结点，此时需要寻找一个更好的 PV 结点，递归以极小窗口[-alpha - 1, -alpha]进行深度为 depth - 1 的搜索，检查返回的估值 value，如果 alpha < value < beta 成立，说明这个结点是新的 PV 结点，再针对 board，递归以较大窗口[-beta, -value]进行深度为 depth - 1 的搜索，搜索的估值结果保存在变量 value 中。如果 alpha < value < beta 不成立，说明这个结点没有已找到的 PV 结点好，或者引发了剪枝。

p_i 对应的分支搜索完毕后，将结点 board 还原成没有执行着法 p_i 前的状态。

检查返回的值 value，如果 value ≥ beta，意味着发生了剪枝，此时的估值 value 并非结点的精确估值，而是个下界值，在历史得分表 History 中为着法 p_i 增加一定的历史得分，更新杀手着法表 KillerMoves 的信息，并在置换表 HashTable 中添加结点 board 在搜索深度为 depth 时的下界值 value，最后直接返回 value 的值；否则说明没有发生剪枝，执行下述操作：

如果估值 value > best，则更新 best 为 value 的值，如果此时 value > alpha 成立，则说明更新 alpha 为 value 的值，当前层次下搜索到的“好”着法 GoodMove 更新为着法 p_i，如果 board 此时是带搜索的博弈树子树的根结点，则将最佳着法 BestMove 更新为着法 p_i。

所有的孩子结点的搜索都结束后，将搜索到的“好”的着法 GoodMove 在历史得分表 History 中累加一定的分值，并更新杀手着法表 KillerMoves 的信息。如果此时 value 是子树根结点估值的精确值，则在置换表 HashTable 中添加结点 board 在搜索深度为 depth 时的精确值 value；如果 value 对应的是上界值，则在置换表 HashTable 中添加结点 board 在搜索深度为 depth 时的上界值 value。算法最终将估值 value 作为最初待搜索结点的估值，返回这个估值，并将下一步的最佳着法保存在全局变量 BestMove 中。

4.1.2 置换表的设计

和大多数棋类游戏的计算机博弈程序一样，本文在置换表中保存 PV 结点、α 结点和 β 结点的估值信息、局面特征及搜索深度。对于 PV 结点和 β 结点，还保存这个估值对应的着法，对于 α 结点，本文还采用第 2 章介绍的“低出/高出边界的修正”策略，将一部分 α 结点对应的着法也保存到置换表。置换表数据项的 32 位哈希值和 64 位校验码使用第 2 章中介绍的 Zobrist 哈希方法生成。如果将混合 PVS 算法在 OpenMP 下进行并行化处理，为了减少关键代码段的锁操作，每个线程需要一个独立的的置换表（将在本章 4.2 节介绍），对内存要求太高，因此这里不使用第 2 章提到的“双置换表”方法。

基于上述考虑，这里将置换表中的元素的数据结构定义如下：

```
struct HashItem
{
    unsigned long long checksum; //表示棋盘局面的 64 位校验码
    ENTRYTYPE  type; /* 估值的类型，EXACT 表示精确值，LOWERBOUND 表示下界值，UPPERBOUND 表示上界值*/
    int depth; //结点对应的搜索深度
    int value; //结点的估值
    MOVE goodmove; //对应的着法
};
```

置换表 HashTable 包含动态分配的一个数组成员 Item，数组中的每个元素的数据类型都是 HashItem 类型。

置换表的查找操作算法可用类 C/C++伪代码表示如下：

```
bool SearchTT(board, alpha, beta, depth, &value, &mov)
{
    用哈希函数计算当前局面 board 的 Zobrist 键值 checksum 的映射索引 p;
    A = HashTable.Item[p];
    if(A.checksum != checksum) {mov = NULLMOVE; return false;}
    mov = A.goodmove;
    if(A.depth >= depth)
    {
        switch(A.type)
        {
        case EXACT: value = A.value; return true;
        case LOWERBOUND:
```

```
                if(A.value >= beta) { value = A.value; return true;}
            case UPPERBOUND:
                if(A.value <= alpha) { value = A.value; return true;}
            }
        }
        return false;
    }
```

本文对置换表的更新操作，采取如下策略：如果置换表中已经保存了博弈树中刚搜索到的结点的信息，则采用“深度优先覆盖”的策略，即博弈树中搜索到的结点对应的搜索深度大于或等于置换表中该数据项的搜索深度时，才使用新搜索到结点的估值信息替换置换表中对应数据项的信息；如果置换表中没有保存了博弈树中刚刚搜索到的结点的信息，说明搜索过的结点是一个新局面，则采用“始终替换”的办法，直接用博弈树中搜索到的结点的估值信息覆盖该置换表中对应数据项已经存在的值。置换表在更新前都不必清空，因为一方面采取这种策略，前面搜索记录的结果对后面的搜索并无任何不良影响，另一方面清空置换表的操作本身也会耗费一定的时间。

置换表的更新操作算法可用类 C/C++伪代码表示如下：

```
void InsertTT(board, value, depth, type, mov)
{
    用哈希函数计算当前局面 board 的 Zobrist 键值 checksum 的映射索引 p;
    A = HashTable.Item[p];
    if(A.checksum == checksum && A.depth >= depth)
    {   /*高出边界的修正*/
        if(A.goodmove == NULLMOVE) A.goodmove = mov;
        return;
    }
    A.checksum = checksum; A.depth = depth;
    A.value = value; A.goodmove = mov; A.type = type;
}
```

第 2 章中提到一些研究者提出不对叶子结点进行哈希处理，以节省时间。但本文认为不能一概而论，要针对具体的应用作具体分析。如果估值函数简单，如五子棋等，计算量很小，对博弈树结点的估值瞬间完成，可以不对叶子结点进行哈希处理；但对于中国象棋、国际象棋、围棋等棋类游戏，估值函数涉及的因素很多，非常复杂，对博弈树结点的估值耗费时间相对较多，显然将叶子结点的估值信息保存到置换表还是很有必要的，以减少调用估值函数，提高搜索速度。本

文研究的混合 PVS 算法主要针对中国象棋，因此将达到最大搜索深度的叶子结点的信息也保存到置换表。

4.1.3 空着裁剪的设计

对于中国象棋或者国际象棋对应的博弈树，空着裁剪能较大程度提高剪枝效率，使待搜索的分支锐减。但要完美使用空着裁剪的难度较大，稍有不慎就有可能造成漏判或者导致搜索的不稳定性。为了保证搜索的正确性，这里采取一种比较保守的做法，具体措施如下：

（1）当搜索深度大于或等于 9，并且己方的棋子估值比对方的棋子估值高出 600 分以上时，说明己方占有非常明显的优势，此时取 R 的值为 3，否则取 R 的值为 2（局面估值设计详见第 5 章）；

（2）被将军时不使用空着裁剪；

（3）待搜索子树的根结点初始的 beta 值一般为+∞，所以根结点没必要进行空着裁剪；

（4）残局时不使用空着裁剪。这里对“残局”的界定是：己方的进攻棋子少于 3 个；

（5）空着裁剪是极小窗口搜索，且搜索深度相对较浅，空着之后的局面在后续的搜索中是不存在的，所以本文认为空着裁剪时不必使用置换表，也不更新历史得分表和杀手着法表，这样可以减少搜索的不稳定性。

4.1.4 启发策略的设计

博弈树中结点的有序性越强，PVS 算法的剪枝效率越高。本文综合使用第 2 章介绍的置换表启发、吃子启发、杀手启发、历史启发四种启发策略对着法进行排序。

上述查询置换表的算法 SearchTT 中已体现了置换表启发，查询置换表时，置换表内命中的数据项，不管对应的深度是否达到要求，只要能获得一个合理的着法，就优先搜索。

对于每一个吃子着法，算法在应用吃子启发时，在被吃的棋子有对方其他棋子保护的情况下，考虑 MVV - LVA 的值，即对方被吃子价值减去己方攻击子价值，作为静态启发值，而在被吃的棋子没保护的情况下，只考虑 MVV 的值作为静态启发值，然后对这种策略产生的吃子得分作排序。

算法设置了一个二维着法数组 KillerMoves[MaxPly][2]，用以保存每一个搜索深度下的 2 个杀手着法，KillerMoves[ply][0]和 KillerMoves[ply][1]始终保持为走棋步数为 ply 的最新搜索到的 2 个杀手着法，同一走棋步数下的 2 个杀手着法

采取普通队列的先进先出的更新方式。先后判断着法 KillerMoves[ply][0]和KillerMoves[ply][1]是否是合理的着法，如果合法，则搜索这个着法相关的结点。本文将“杀手”着法理解为不仅仅是引发剪枝的 β 结点对应的着法，还包括 PV 结点对应的着法，以及 α 结点估值最好的非空着法。

算法还有一个用于历史启发的历史得分表 History，这是一个二维数组，History[from][to]表示着法(from, to)的历史得分，当着法(from, to)被认为是“好”的着法时，则将 History[from][to]累加上一个值，本文定义这个累加的增量值为 2^{Depth}，Depth 为距离叶子结点的深度。当展开待搜索结点下的全部着法时，历史得分越高的着法被认为越“好”。

混合 PVS 算法中，需要将搜索到的一个“好”的着法的信息记录到历史得分表 History 并更新杀手着法表 KillerMoves，这个过程用算法 SaveGoodMove 表示，其类 C/C++伪代码表示如下：

```
void SaveGoodMove(const CMove & mv, int depth, int ply)
{
    History[mv.from][mv.to] += (2 << depth);
    KillerMoves[ply][1] = KillerMoves[ply][0];
    KillerMoves[ply][0] = mv;
}
```

本文将上述 4 种启发策略混合使用，并提出一种特殊的组合顺序，将历史启发同时用于吃子着法和非吃子着法，在函数 CreateAndSortSuccessors 中实现，具体步骤如下：

（1）生成所有静态评价值 ≥0 的吃子着法，按照静态评价值降序排列，如果静态评价值相同，则历史得分高的排在前面；

（2）在上述排列好的静态评价值 ≥0 的吃子着法中，检查是否包含杀手着法，如果包含杀手着法，则将杀手着法排在静态评价值 ≥ 0 的吃子着法的最前面；

（3）生成所有静态评价值 <0 的吃子着法和所有的非吃子着法，按照历史得分值降序排列；

（4）在上述排列好的静态评价值 < 0 的吃子着法和所有的非吃子着法中，检查是否包含杀手着法，如果包含杀手着法，则将杀手着法排在静态评价值 <0 的吃子着法和所有的非吃子着法的最前面；

（5）如果置换表中保存的着法是非空着法，则将置换表中的着法排在全部着法的最前面，第一个搜索。

在上述组合策略中，加强了历史启发的使用，这里历史启发不仅用于非吃子着法的排序，也应用于吃子着法的排序。

4.1.5 串行混合 PVS 算法描述

中国象棋计算机博弈中，结合了着法排序技术（置换表着法、杀手启发、吃子启发、历史启发）置换表和空着裁剪的串行模式下的混合 PVS 算法可用类 C/C++伪代码描述如下：

```
int PVS (board, depth, alpha, beta, nullmove)
{
    best = -∞, PVFlag = 0, ValueIsExact = 0, side = (MaxDepth - depth) % 2;
    if(GameOver(board) || depth == 0) //胜负已分或到达最大搜索深度
    {
        value = Evaluation(board); //对结点 board 估值
        if(depth == 0) InsertHashTable(board, EXACT, value, depth, NULLMOVE); /*到达最大搜索深度，在置换表中添加该局面的估值信息*/
        return value;
    }
    value = SearchTT(board, value, alpha, beta, depth, mov);
    if(HashTable 命中) return value;
    if(nullmove == 0 && !(check(board)) && 不是残局状态)
    {
        MakeNullMove(); ply++; //执行空着
        value = -PVSWithoutTT(board, depth - 1 - R, -beta, -beta + 1, 1);
        RestoreNullMove(); ply--; //撤销空着
        if(value >= beta) return value;
    }
    w = CreateAndSortSuccessors(board, p, depth, ply, mov);
    for(i = 0; i < w; i++)
    {
        board = MakeMoveWithTT(board, p_i); /*局面 board 下执行着法 p_i*/
        ply++; //走棋步数加 1
        if(PVFlag) //如果已经获得主要变例的估值
        {
            value = -PVS(board, depth - 1, -alpha - 1, -alpha, 0);//空窗探测
            if(value > alpha && value < beta ) value = -PVS(board, depth - 1, -beta, -value, 0); //较大窗口探测
        }
```

```
            else value = -PVS(board, depth - 1, -beta, -alpha, 0); /*全窗口探测*/
            board = RestoreMoveWithTT(board, pi); //恢复棋盘局面
            ply--; //走棋步数减 1
            if(value >= beta) //发生剪枝
            {    //更新着法历史得分表、杀手着法表和置换表的信息
                SaveGoodMove(pi, depth, ply);
                InsertTT(board, LOWERBOUND, value, depth, pi);
                return value; //剪枝
            }
            if(value > best) //找到了更好的着法
            {
                best = value;
                if(value > alpha) //更新窗口下界
                {
                    GoodMove = pi; ValueIsExact = 1;
                    PVFlag = 1; alpha = value; //找到更好的结点
                    if(depth == MaxDepth) BestMove = pi;
                }
            }
        } //end of for
        SaveGoodMove(GoodMove, depth, ply); //更新历史启发表和杀手着法表
        //更新置换表
        if(ValueIsExact) InsertTT(board, exact, value, depth, GoodMove);
        else
        {
            if(mov != NULLMOVE) GoodMove = mov; /*低出边界的修正*/
            InsertTT(board, UPPERBOUND, value, depth, GoodMove);
        }
        return best;
    }
```

4.2 混合 PVS 算法的 OpenMP 并行化设计

混合 PVS 算法用到了置换表、历史启发表和杀手着法表，非常适合采用基于共享内存的模式进行并行计算。因此，在 OpenMP 下对混合 PVS 算法进行并行化处理，不仅能大大减少硬件和环境的开销，也符合混合 PVS 算法的特点。

4.2.1 基于 PVSplitting 的任务分解

第1章介绍了博弈树的并行搜索有两种方式，即基于窗口的分割和基于子树的分割。第2章中提到影响博弈树窗口搜索效率的首要原因是窗口位置，次要原因才是窗口大小。因此，本文认为，实际的博弈树每个结点的估值在窗口[α,β]的分布情况很难预判，基于子树的分割的总体效果比基于子树的分割总体效果要好。

串行混合PVS算法的计算的最大工作量，是针对当前局面board下生成的w种着法进行搜索，这个计算集中在一个for循环内，分支搜索的顺序，会影响剪枝效率，但不会影响对当前局面board的估值，因此并行化设计时首先要让这个for循环并行执行。

如果将博弈树第一层的结点展开后，就直接把每个分支的搜索以线程形式分配给CPU处理单元，这种方式最为简单，但是却无法获得良好的加速比和效率。因为在这种方式下工作，每个CPU处理单元最初分配到的线程都必须以全窗口的方式进行搜索，大大降低了效率，这种方法不可取。

目前的启发策略已经能使得博弈树的结点排列倾向良序，相对接近极小树。根据这个特点，本文采用PVSplitting并行策略。

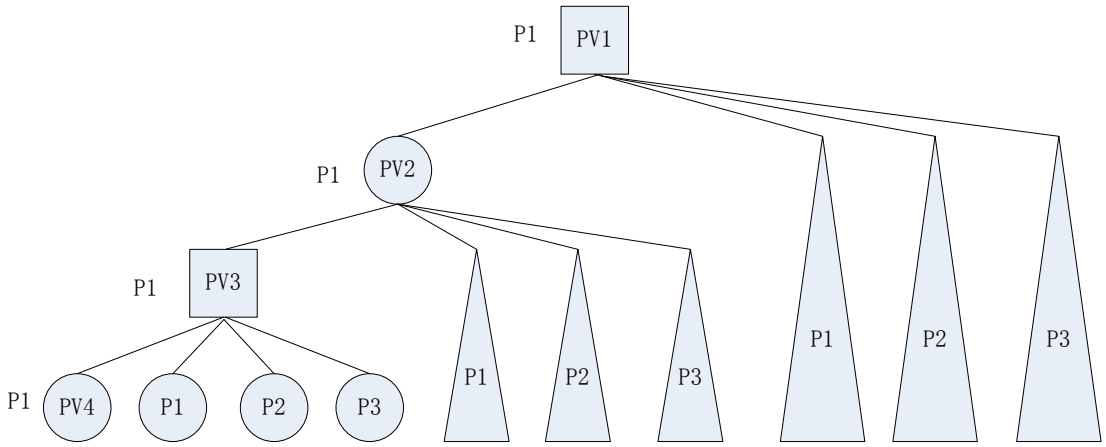


图 4.2 基于PVSplitting的并行PVS算法示意

图4.2演示了OpenMP下这种并行PVS算法的执行过程。假设最大搜索深度为4，并行线程数为3，主线程P1从根结点PV1沿着最左边的分支一直搜索到第4层的第一个叶子结点PV4，将PV4的估值传回给第3层最左边的结点PV3，再创建出派生线程P2和P3，与主线程P1一起，对PV3的其他孩子结点并行搜索，派生线程P2和P3完成各自的搜索后被销毁，从而PV3获得估值，这个估值传回给第2层最左边的结点PV2，主线程P1再重新创建线程P2和P3，3个线程对PV2剩下的孩子

结点并行搜索，派生线程P2和P3完成各自的搜索后被销毁，PV2的估值传给PV1，主线程P1重新创建线程P2和P3，一起对PV1剩下的孩子结点并行搜索，派生线程P2和P3完成各自的搜索后被销毁，根结点PV1获得了最终的估值。对于不在最左边的分支中的结点，每个线程首先以进行空窗探测，空窗探测失败后再以较大窗口进行重新搜索。

根据PVSplitting思想，展开根结点下的w个孩子结点时，可对第一个孩子结点p_0单独进行递归处理，对剩余结点p_1、p_2、……、p_{w-1}的并行处理可以在前述混合PVS算法中的for循环内处理。在这个for循环内，当发生剪枝时将通过return语句返回剪枝分支的估值，并非严格意义上的单入口单出口的循环。可以通过简单的处理将这个循环改为单入口单出口的结构，这里引入一个二值变量EndLoop表示for循环内是否发生了剪枝，并用变量val记录这个引发剪枝的估值。for循环内所有的搜索分支在执行PVS搜索前首先判断EndLoop的值，发生了剪枝，则执行continue语句，否则才执行搜索操作。for循环结束后，再次通过EndLoop的值判断是否已发生剪枝，如果是则直接返回val。这样，这个for循环变成了单入口单出口的循环，可以采用编译指导指令#pragma omp parallel for实现并行处理。可以看出，这种任务分解方式的粒度比较适中，但每个线程的工作量不尽相同甚至差别较大，多数线程在空窗探测发生fail-low即可结束工作，少数线程在空窗探测发生fail-high时需要以较大窗口重新搜索，需要耗费相对较多的时间。如果待搜索的博弈树结点排序越好，越接近最小树，并行算法效率越高。

对于最左边的分支，生成着法的操作和对着法排序的操作可以根据实际情况采取相应的并行处理。例如，在中国象棋计算机博弈算法中，可以针对当前走棋方的不同棋子类型（如，车、马、炮、兵、将、士、相7种类型棋子）的棋子并行生成不同着法，最后个再合并入着法列表p。而对着法列表p的排序操作，使用并行快速归并排序算法进行排序，过程由下列类C/C++伪代码表示：

```
#pragma omp parallel for //假设将数据分为4个区间
    for(i = 0; i < 4; i++) {对区间i进行串行排序}
……
#pragma omp parallel for /*对排序好的各个相邻区间进行并行归并操作*/
    for(i = 0; i < 4; i += 2) {使用串行归并算法对区间i和区间i+1进行归并}
……
```

搜索结束后，需要将“好”的着法 GoodMove 的相关信息更新历史得分表 History 和杀手着法表 KillerMoves，并将其估值信息写入置换表 HashTable。这些操作瞬间即可完成，并非算法的性能瓶颈，但为了充分利用 OpenMP 中的多核编程的优势，可以使用编译指导指令#pragma omp parallel sections 中嵌套#pragma omp section 指令的方式，将这两步操作同时进行。

4.2.2 线程数量的设置

第3章中提到，一般情况下，OpenMP并行线程数应等于CPU处理单元（核）数。串行混合PVS算法是递归算法，如果不加任何修改直接进行并行化处理，因嵌套并行产生的线程数量急剧增加，将远远超过一般多核CPU的处理单元数，线程间的创建、销毁、等待等方面的额外开销大大增加，引发性能下降。虽然最新的OpenMP 3.0版本中提供了基于任务的#pragma omp task编译指导指令，但串行PVS算法发生剪枝而返回的条件依赖for循环内的计算结果，Visual Studio 2010等开发工具暂时也不支持OpenMP 3.0。

为了避免线程数量因嵌套并行剧增，本文将混合PVS算法在形式上做一些变化，将算法分解成两个形式上和混合PVS算法都类似的算法PVS_Parallel和PVS_Serial，PVS_Parallel算法是优化后的并行算法的入口，PVS_Serial算法是递归形式的串行执行的混合PVS算法。PVS_Parallel算法对待搜索的博弈树，除了整个树最左侧的分支调用PVS_Parallel算法自身单独递归处理之外，其他的结点展开后，在for循环内不直接调用自身进行递归，而是对每个结点以串行形式调用递归算法PVS_Serial进行搜索，每个线程发生递归时，仅由该线程自己处理，不再启动子线程。这样，每个CPU处理单元在每一层中只有执行完了某个结点为根的子树的搜索线程，才能执行其它搜索线程。

4.2.3 数据分解与临界区的处理

并行混合PVS算法，每个线程都要生成在多个搜索层次下的着法列表，对当前局面进行走棋、搜索、恢复棋盘的过程，每个分支返回各自的不同估值，因此棋盘局面board、每个线程生成的着法列表p、当前线程搜索深度depth、某个分支返回的估值value等应为每个线程复制一份私有的副本，并继承进入循环前相应对象的值，OpenMP中可以通过firstprivate子句设置。而搜索结束后找到的“好”的着法和最佳着法只保留一个，其估值也是唯一的，所以“好”的着法GoodMove和最佳着法BestMove及其估值best应设置为共享对象，窗口下界值alpha也可设置为共享对象，如果某个线程在一个子树上找到了更好的结点，可以动态更新窗口的边界值，并通知其它计算处理单元窗口的边界值已更新，形成更小的窗口范围，使得其它线程可能引发更多的剪枝。OpenMP中可以通过shared子句进行设置共享对象。这些对象的更新操作，需要使用编译指导指令#pragma omp critical对其进行关键代码段的保护，或者实施锁机制进行保护。前述的二值变量EndLoop和第一个剪枝的分支的估值val也应设置为共享对象，但任意一个线程因发生剪枝更新了EndLoop的值，EndLoop的值就不会再发生变化，故不需要对其进行关键

代码段或者锁机制的保护。

混合PVS算法对历史启发表、杀手着法表和置换表进行读写操作很频繁。串行模式下只使用一个历史得分表、一个杀手着法表和一个置换表，并行化时，可以将历史得分表、杀手着法表和置换表设置为线程的私有对象，但更新历史得分表、杀手着法表和置换表的代码部分，都要采取保护措施，严重影响算法的时间效率。这里采用“以空间换时间”的解决办法进行数据分解，设置一个历史得分表数组History、一个杀手着法表数组KillerMoves和一个置换表数组HashTable，使每个并行的线程拥有各自的一个历史得分表、杀手着法表和置换表。某个线程搜索到更好的着法时，只更新对应CPU处理单元拥有的历史得分表、杀手着法表和置换表，但是可以对全部的置换表、历史得分表和杀手着法表进行读操作。例如，要在置换表中查找某个局面对应的估值信息时，则查找置换表数组中所有的置换表，可以让每个线程并行查找自己的置换表，如果有一个置换表命中，则在置换表中查找成功，如果所有的置换表都没有命中，则查找失败，如果出现多个置换表都命中的情况，则选择搜索层次最深的项值。某个着法的历史得分，是历史得分表数组中全部历史得分表中该着法的历史得分的累计。而每个并行线程在某一搜索深度下，仅设置一个杀手着法即可。这样避免了对这些对象的写操作实施加锁、解锁的操作，只要将历史得分表数组、杀手着法表数组和置换表数组的数组名通过shared子句进行设置，并行算法的时间效率得以提升。

4.2.4 调度策略

在 OpenMP 下，将每一个子树的搜索交给一个线程来处理，在最理想的情况下，应该让所有的线程按照执行所需总时间均匀分配给 CPU 处理单元，使每个 CPU 处理单元的工作负载大致相等。但前面已经分析，空窗探测发生 fail-high 的线程与空窗探测发生 fail-low 的线程的负载差别较大，具体会发生哪种情况无法预知，因此在实际中无法预判每个 CPU 处理单元应该分配哪些线程，但应当使每个 CPU 处理单元的工作时间尽量接近。

由于算法引入了结点排序技术，排序越靠前的结点应越优先搜索。对某一层而言，应为每个刚创建的线程分配第一个尚未搜索的结点对应的分支。具体实现时，可设置一个数组 Searched，二值变量 $Searched_i$ 表示第 i 个结点是否已有线程负责搜索。每个线程总是选择第一个尚未有线程负责搜索的结点 p_i 进行搜索，搜索前更新 $Searched_i$ 的值。线程查询 Searched 数组的代码段显然属于关键代码段。在 for 循环内，每个 CPU 处理单元静态分配 1 次迭代，这样一旦某个线程发生空窗探测失败以较大窗口进行搜索，分配给相应的 CPU 处理单元的后续线程就少，在 OpenMP 中可以通过 schedule(static, 1)子句实现。这种每次只分配一

个进程给 CPU 处理单元的做法虽然增加了线程分配的开销，但最有利于负载平衡。因为每个分支搜索的时间消耗差距相对较大，负载平衡带来的好处要远多于线程分配的开销。在中国象棋中，每个结点下最多也就 50 多个分支，这种线程分配的开销也是非常小的。

大多数情况下，需要进行较大窗口搜索的分支往往排列靠前，排列靠后的分支空窗探测时发生 fail-high 的概率较大，因此这种将每个结点对应的线程，依次顺序分配给每个 CPU 处理单元的调度策略，绝大多数情况下不会出现当某个处理单元以较大窗口搜索排列靠后的结点迟迟未能结束计算，而其他处理单元却因空窗探测发生 fail-low 结束计算而闲置太长时间的情况。

4.2.5 解决伪共享问题

从上述并行化设计可见，伪共享问题有可能会发生在多个线程对置换表数组、着法历史得分表数组和杀手着法表数组的更新操作中，但是这种概率很低，因为每个线程拥有自己独立的置换表、着法历史得分表和杀手着法表，尤其是每个线程的置换表长度很大，对两个相邻的置换表来说，在某个置换表的尾部与其相邻置换表的首部发生伪共享问题的概率可以认为接近于零，而着法历史得分表数组和杀手着法表数组发生伪共享问题的概率稍高。

在 OpenMP 并行混合 PVS 算法的具体实现时，可以为置换表数组、着法历史得分表数组和杀手着法表数组动态申请内存单元，并针对着法历史得分表数组和杀手着法表数组采用 Cache 行对齐的内存管理方式，即对每个线程的着法历史得分表和杀手着法表分配存储单元时，多分配一个 Cache 行的大小（在 Intel 处理器中 Cache 行大小为 64 字节），从首地址开始向后找到第一个 Cache 行的大小的倍数的地址，以这个地址作为每个着法历史得分表和杀手着法表逻辑上的起始地址，这样就使得更新着法历史得分表和杀手着法表时，任意两个并行线程都不会对 CPU 的同一个 Cache 行进行写操作。

严格来说，置换表数组也应该这样处理，但前面已经提到，在实际中置换表发生伪共享的几率非常低，因此即使对置换表不做 Cache 行对齐的处理，也没有什么影响。

4.2.6 并行混合 PVS 算法的描述

根据上述优化思想，以初局和中局的情况为例，得到基于OpenMP的并行混合PVS算法PVS_Parallel的UML活动图，如图4.3所示，其中，“执行搜索过程”的子活动图，如图4.4所示。

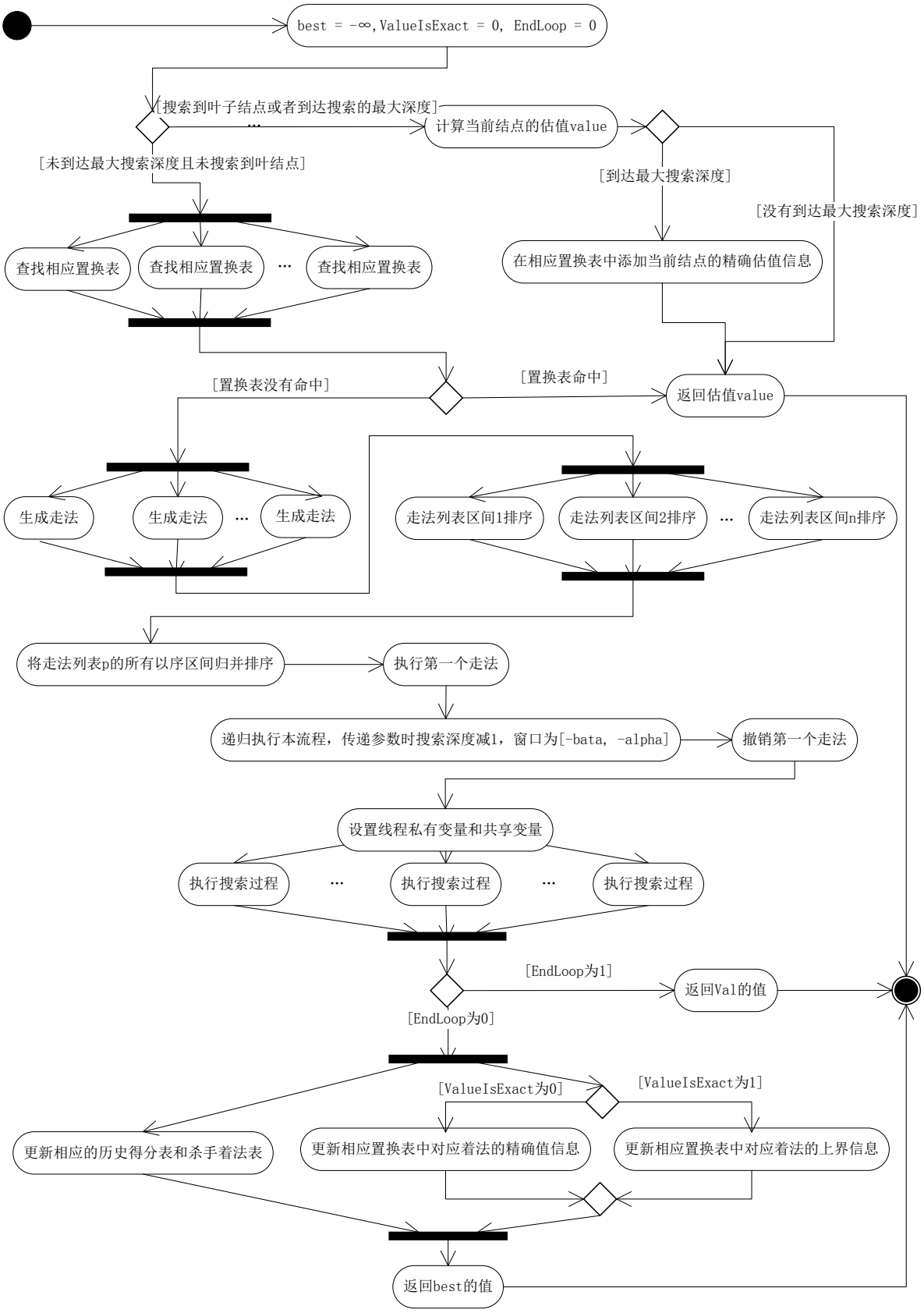


图 4.3 并行混合PVS算法PVS_Parallel的活动图

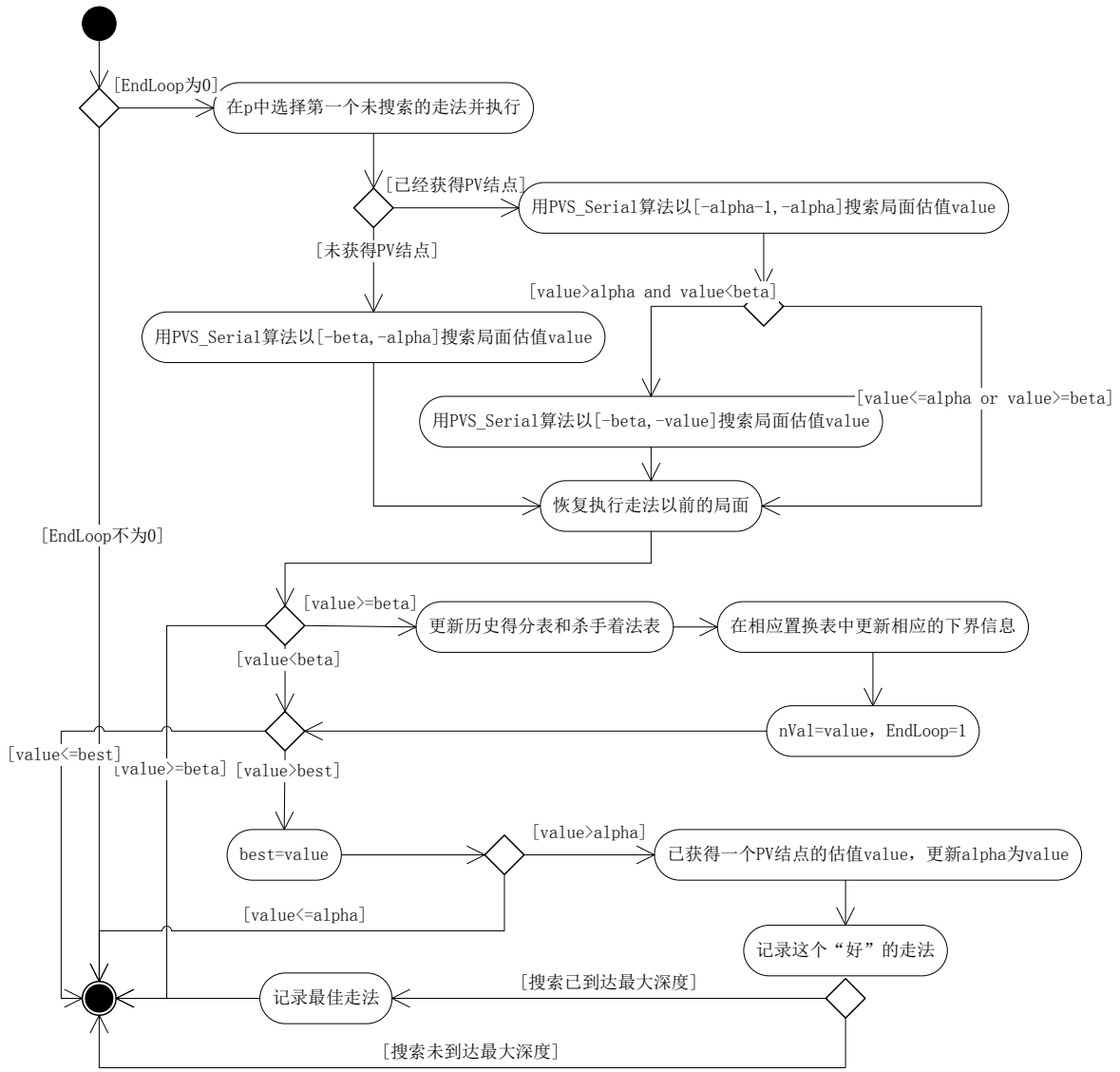


图 4.4 “执行搜索过程”的子活动图

在OpenMP下进行并行化程序设计，关键在于合理使用编译指导语句和库函数，并合理设置共享变量和私有变量。因此除了活动图外，还给出了基于OpenMP的并行混合PVS算法详细的类C/C++伪代码描述，以体现OpenMP中编译指导指令的使用。其中，算法PVS_Parallel为并行的PVS算法，是图4.3和图4.4所示活动图的具体实现。对于非整个博弈树最左侧的其他结点来说，并行搜索这些结点时，每个搜索线程调用PVS_Serial算法。

```
int PVS_Parallel(board, depth, alpha, beta)
{
    best = -∞, ValueIsExact = 0, side = (MaxDepth-depth) % 2, EndLoop = 0;
    if(GameOver(board) || depth == 0) //胜负已分或到达最大搜索深度
    {
```

value = Evaluation(board); //计算结点board的估值

if(depth == 0) InsertHashTable(board, EXACT, value, depth, NULLMOVE); /*在当前线程的置换表中添加该局面的估值信息*/

return value;

}

value = SearchALLHashTable(board, alpha, beta, depth); /*在所有的置换表中搜索当前局面对应的估值value*/

if(HashTable中的某个置换表命中) return value;

#pragma omp parallel for

for(i = 0; i < 7; i++) CreateSuccessors(board, side, i); /*生成走棋方side第i种棋子的所有着法，共车马炮兵将士相7种棋子*/

MergeToMoveList(p); //合并存入列表p, 生成着法p_0...p_{w-1}共w个

SortNode_Parallel(p); /*将p_0、p_1、...、p_{w-1}并行排序，p_i的历史得分值是所有的历史得分表中p_i得分值的累加*/

board = MakeMove(board, p_0); //在棋盘局面board下执行着法p_0

best = -PVS_Parallel(board, depth - 1, -beta, -alpha); //对着法p_0递归搜索

board = RestoreMove(board, p_0); //恢复棋盘局面

if (best > alpha)

{

GoodMove = p_0; ValueIsExact = 1; alpha = best;

if(depth == MaxDepth) BestMove = p_0;

}

$Searched_1 = Searched_2 = \ldots = Searched_{ThreadCount-1} = 0$;

#pragma omp parallel for schedule(static, 1), firstprivate(board, value, depth, p), shared(HashTable, History, KillerMoves, BestMove, GoodMove, alpha, EndLoop, val, ValueIsExact, Searched)

for(i = 1; i < w; i++)

{

#pragma omp critical

for(j = 1; j < w; j++) if($Searched_j$ == 0) { $Searched_j$ = 1; break;}

if(EndLoop) continue;

board = MakeMove(board, p_j); /*在棋盘局面board下执行着法pj*/

value = -PVS_Serial(board, depth-1, -alpha-1, -alpha, 1); //空窗探测

if(value > alpha && value < beta) value = -PVS_Serial(board, depth - 1, -beta, -value, 1); //较大窗口探测

```
            board = RestoreMove(board, pj); //恢复棋盘局面
            if(EndLoop) continue;
            if(value >= beta) //剪枝处理
            {    /*更新当前线程对应的置换表、历史史得分表和杀手着法表*/
                SaveGoodMove(pj, depth, ply);
                InsertHashTable(board, LOWERBOUND, value, depth, pj);
                val = value; EndLoop = 1; //发生了剪枝
            }
            #pragma omp critical
            if(value > best) //找到了更好的着法
            {
                best = value;
                if(value > alpha)
                {
                    GoodMove = pj; ValueIsExact = 1;
                    PVFlag = 1; alpha = value;
                    if(depth == MaxDepth) BestMove = pj;
                }
            }
        } //end of for
        if(EndLoop) return val;
        #pragma omp parallel sections
        {    /*更新当前线程对应的历史得分表、杀手着法表和置换表*/
            #pragma omp section
            SaveGoodMove(GoodMove, depth, ply);
            #pragma omp section
            if(ValueIsExact) InsertTT(board, exact, value, depth, GoodMove);
            else
            {
                if(mov != NULLMOVE) GoodMove = mov; //低出边界的修正
                InsertTT(board, UPPERBOUND, value, depth, GoodMove);
            }
        }
        return best;
    }
```

其中，每个线程执行的 PVS_Serial 算法与前述的串行模式下的混合 PVS 算法类似，所不同的是，线程 ThreadNum 针对置换表的查找操作需要使用 SearchALLHashTable 函数对所有的置换表进行查找，而置换表的更新函数 InsertHashTable 只针对该线程拥有的置换表 $HashTable_{ThreadNum}$ 进行更新；对历史得分表和杀手着法表的操作也是如此，在计算着法的历史得分并为着法排序时，查询全部的历史得分表和杀手着法表，但为一个“好”的着法更新历史得分表和杀手着法表时，调用 SaveGoodMove 函数，只更新该线程对应 CPU 处理单元拥有的历史得分表 $History_{ThreadNum}$ 和杀手着法表 $KillerMoves_{ThreadNum}$。

多个并行线程同时执行 PVS_Serial 算法，故在 PVS_Serial 算法中一旦发现有其他线程发生了剪枝，应立即返回引发剪枝的线程的估值，并终止该算法；如果某个线程更新了 alpha 值，算法也应该及时更新相应的窗口边界，提高剪枝效率。在 OpenMP 中线程与线程之间没有通信机制，因此只能通过设置共享变量实现上述功能。

由图 4.3、图 4.4 及算法伪代码可以看出，算法的串行部分只出现在参数和变量的初始化、对着法列表中的有序区间进行归并排序以及搜索博弈树最左侧的分支线路等少部分，这部分计算量极小，由阿姆达尔定律（Amdahl Law）可知，串行部分所占比例越小，并行计算的加速比性能越好，在上述算法中，只有线程选择搜索结点、更新搜索到的“好”的着法及其估值 best 和 alpha 值时才使用锁操作，这些操作瞬间即可完成，由于使用了结点排序技术，对大多数分支而言，更新“好”的着法及其估值 best 和 alpha 值这段代码都不会执行，因此锁操作带来的影响很小。

4.3 并行混合 PVS 算法的实验结果与分析

本文开发了一个真实环境的中国象棋计算机博弈系统作为实验平台，这个系统的设计方案详见第 5 章。测试所用的硬件环境是一台多核笔记本电脑，采用 Intel Core i7-2630QM（主频 2.0 GHz，4 核）的微处理器，3G DDR3 内存，操作系统使用 32 位的 Windows 7 简体中文旗舰版。

4.3.1 各种搜索策略比较的实验结果

α-β 搜索/剪枝算法、PVS 算法、MTD(f)算法等博弈树搜索算法的时间复杂度都是指数级复杂度，即假设 w 为博弈树的平均分支因子数，d 为搜索深度，α-β 搜索/剪枝算法、PVS 算法、MTD(f)算法的时间复杂度均为 $O(w^d)$。引入置换表、各种启发策略、空着裁剪等优化措施后，并不能从理论上降低算法的时间复杂度，但在真实环境的博弈树环境下，这些优化措施往往能大幅度提高剪枝率，较为可

观的降低算法的时间开销。因此，评价博弈树搜索算法的效率，应考察同一实验环境下算法的平均估值结点数，平均估值结点数越少，算法的平均效率越高。

中国象棋计算机博弈最复杂的是开局阶段的搜索，这里从文献[44]中选取 10 个有代表性的开局走法进行实验，在串行模式下，比较各种搜索算法在开局时每走一步的平均估值结点数。为确保可比性，这里均在本文第 5 章设计的中国象棋计算机博弈系统中进行比较。由于在实际中，前 3 层的搜索因深度太浅意义不大，这里列出搜索深度为 4~7 时，这种融入了置换表、启发策略（吃子启发、置换表启发、历史启发、杀手启发）和空着裁剪的混合 PVS 算法，和其他不同搜索策略的对比结果。其中开局时各种搜索算法每走一步的平均估值结点数，如表 4.1 所示；开局时各种搜索算法每走一步的平均耗时，如表 4.2 所示。

表 4.1 开局各种搜索算法每走一步的平均估值结点数（个）

搜索算法\搜索深度	4	5	6	7
基本的 α-β 算法	102313	2048359	21434885	304364607
PVS 算法	68241	793431	10296365	135284546
MTD(f)算法	59464	635140	12745914	162421713
α-β+置换表	70138	914362	7645367	99757534
α-β+启发策略（无置换表启发）	8746	189345	654218	2261055
MTD(f) +置换表+启发策略	6072	59448	407382	1927425
MTD(f) +置换表+启发策略+空着	4573	35875	112216	480674
PVS +置换表+启发策略	6134	56234	315826	1496497
PVS +置换表+启发策略+空着	4286	30132	91965	432518

表 4.2 开局各种搜索算法每走一步的平均耗时（毫秒）

搜索算法\搜索深度	4	5	6	7
基本的 α-β 算法	1276	25458	224825	2844978
PVS 算法	701	7407	99453	1174762
MTD(f)算法	663	7155	113472	1467046
α-β+置换表	807	9983	70641	928546
α-β+启发策略（无置换表启发）	120	795	8341	42176
MTD(f) +置换表+启发策略	51	342	3198	15027
MTD(f) +置换表+启发策略+空着	39	167	604	2305
PVS +置换表+启发策略	53	318	2549	11939
PVS +置换表+启发策略+空着	36	139	498	1894

从表 4.1 与表 4.2 中可以看出本文设计的结合了置换表、启发策略和空着裁

剪的混合 PVS 算法应用于中国象棋计算机博弈中的优势。

4.3.2 并行混合 PVS 算法的实验结果

第 3 章提到，衡量多核 PC 环境下并行算法的性能，需要从加速度、并行效率、可扩展性进行衡量，而可扩展性主要取决于并行效率。因此，这里在相同的实验环境下，分别测试混合 PVS 算法在串行模式、2 线程并行模式、4 线程并行模式下每走一步的平均耗时，从而计算出算法的平均加速比和平均效率。

依旧从上述 10 个有代表性的开局走法为例，分别测试串行、2 线程并行、4 线程并行模式下的混合 PVS 算法每走一步的平均耗时(毫秒),以及 2 线程并行、4 线程并行下的平均加速比和平均效率,搜索深度从 4 层到 9 层,其结果如表 4.3 所示。

表 4.3 并行线程数对并行算法性能的影响

深度\策略（开局）	串行混合 PVS 算法每走一步的平均耗时（毫秒）	2 线程并行混合 PVS 算法每走一步的平均耗时（毫秒）、平均加速比和平均并行效率	4 线程并行混合 PVS 算法每走一步的平均耗时（毫秒）、平均加速比和平均并行效率
4	36	31/1.161/0.581	34/1.059/0.265
5	139	98/1.418/0.709	81/1.716/0.429
6	498	354/1.407/0.703	295/1.688/0.422
7	1894	1183/1.601/0.801	868/2.182/0.546
8	8515	5003/1.702/0.851	3028/2.812/0.703
9	35939	20627/1.715/0.849	12965/2.772/0.693

图 4.5、图 4.6 和图 4.7 是表 4.3 对应的折线图。图 4.5 体现了开局阶段，随着搜索深度的递增，串行、2 线程并行、4 线程并行下混合 PVS 算法每走一步的平均耗时。图 4.6 和图 4.7 分别体现了开局阶段，2 线程并行、4 线程并行下混合 PVS 算法的加速比和并行效率在不同搜索深度下的变化情况。

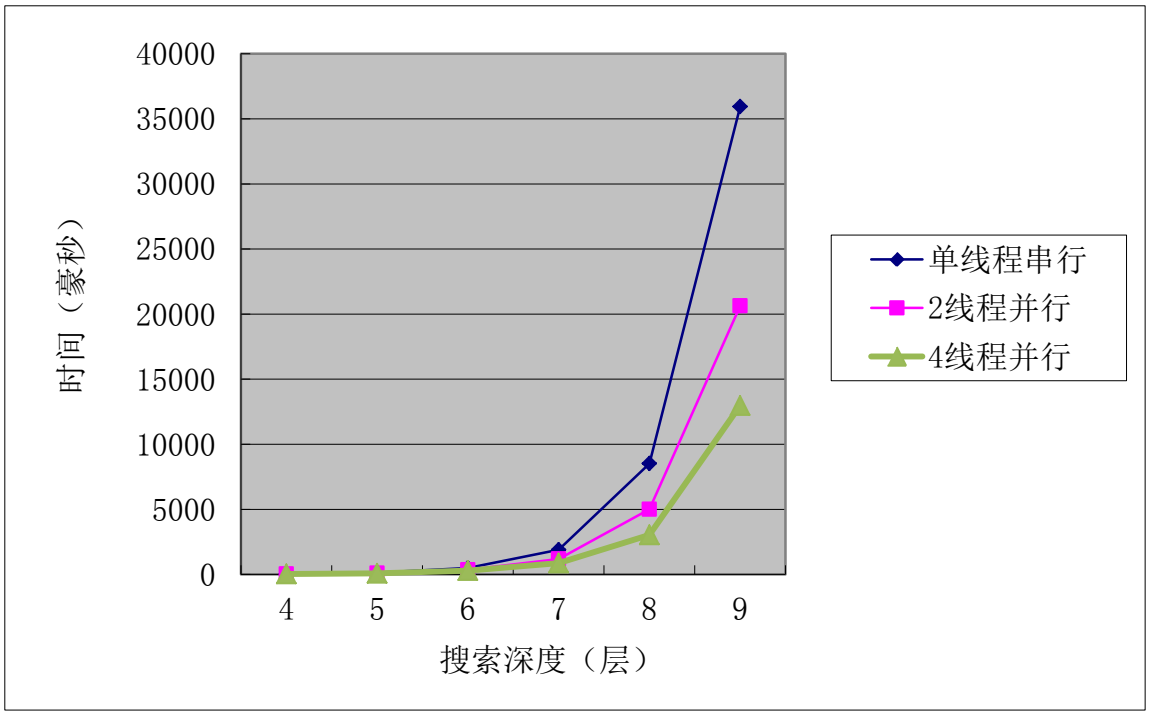


图 4.5 并行混合PVS算法每走一步的平均耗时

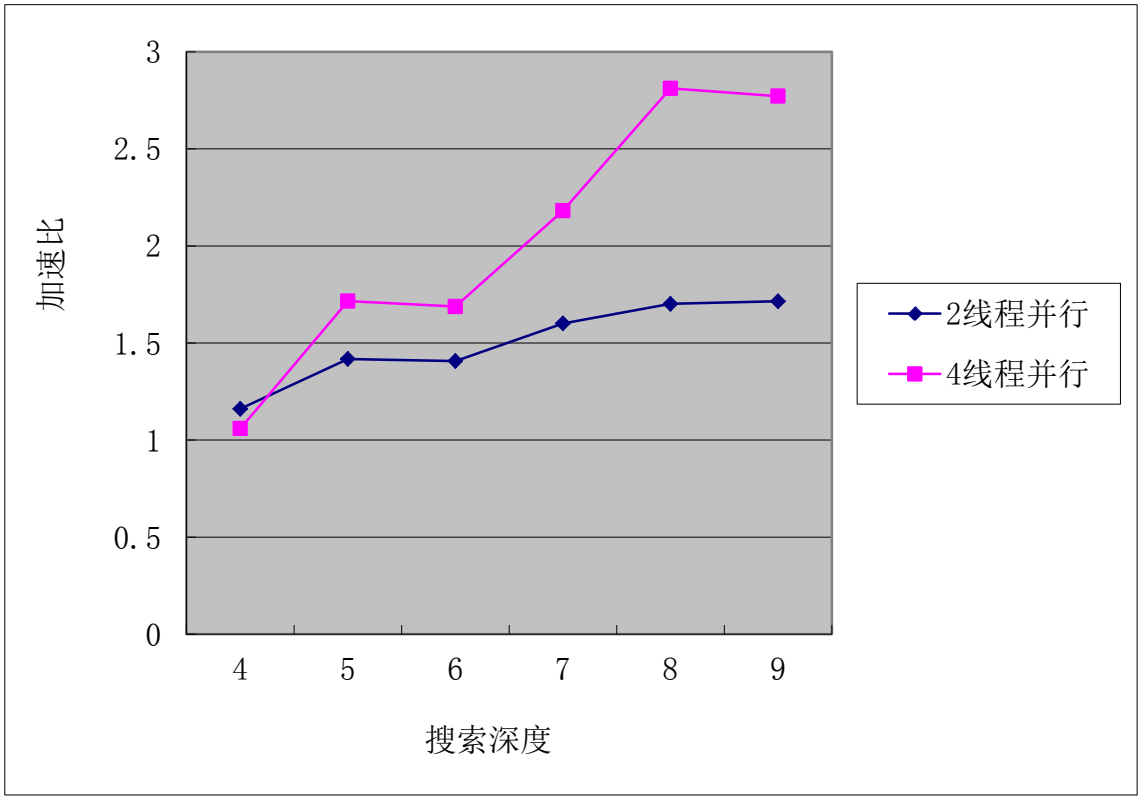


图 4.6 并行混合PVS算法的加速比

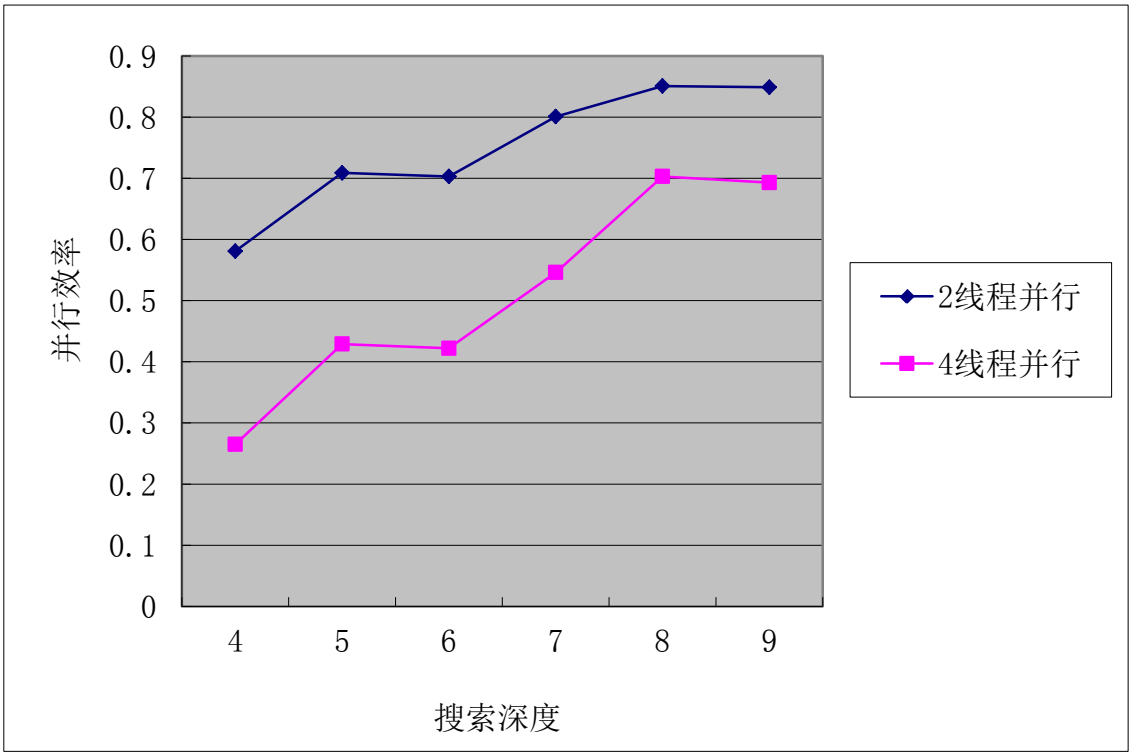


图 4.7 并行混合PVS算法的并行效率

可以看出，总体而言，与串行混合 PVS 算法相比，在并行线程数不超过 CPU 处理单元数时，并行混合 PVS 算法能有效降低搜索的耗时，加速比随着并行线程数的增加而上升，但并行效率随着并行线程数的增加而下降。

开局搜索深度从 4 层到 9 层时，串行、2 线程并行、4 线程并行模式下的混合 PVS 算法每走一步的平均估值结点数和置换表平均命中数，如表 4.4 所示。

表 4.4 并行线程数对平均估值结点数和置换表命中数的影响

深度\策略（开局）	串行混合 PVS 算法每走一步的平均估值结点数（个）和置换表命中数（次）	2 线程并行混合 PVS 算法每走一步的平均估值结点数（个）和置换表命中数（次）	4 线程并行混合 PVS 算法每走一步的平均估值结点数（个）和置换表命中数（次）
4	4286/756	5964/771	6568/851
5	30132/4186	33794/5861	38125/7088
6	91965/8047	117858/9863	140282/11973
7	432518/59027	559485/68481	792504/82706
8	2143914/195324	2695802/245813	3310346/282374
9	11382073/1041057	13053639/1296356	16134627/1379035

图 4.8 和图 4.9 是表 4.4 对应的折线图，分别体现了在开局阶段，串行、2

线程并行、4 线程并行下混合 PVS 算法每走一步的平均估值结点数和置换表平均命中数在不同搜索深度下的变化情况。

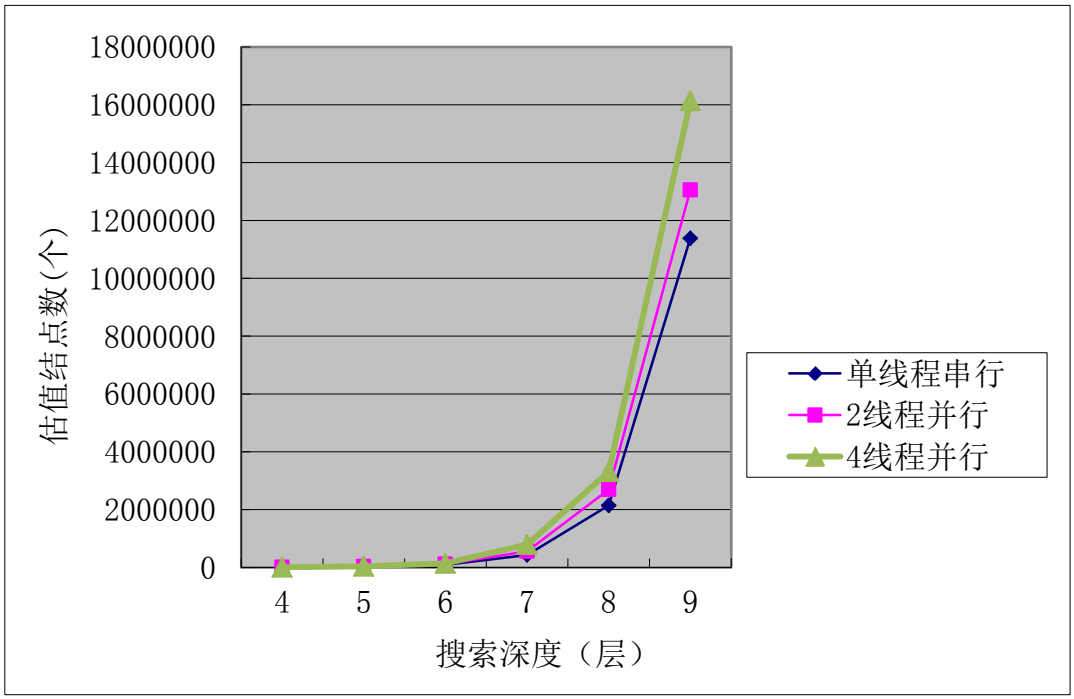


图 4.8 并行混合PVS算法每走一步的平均估值结点数

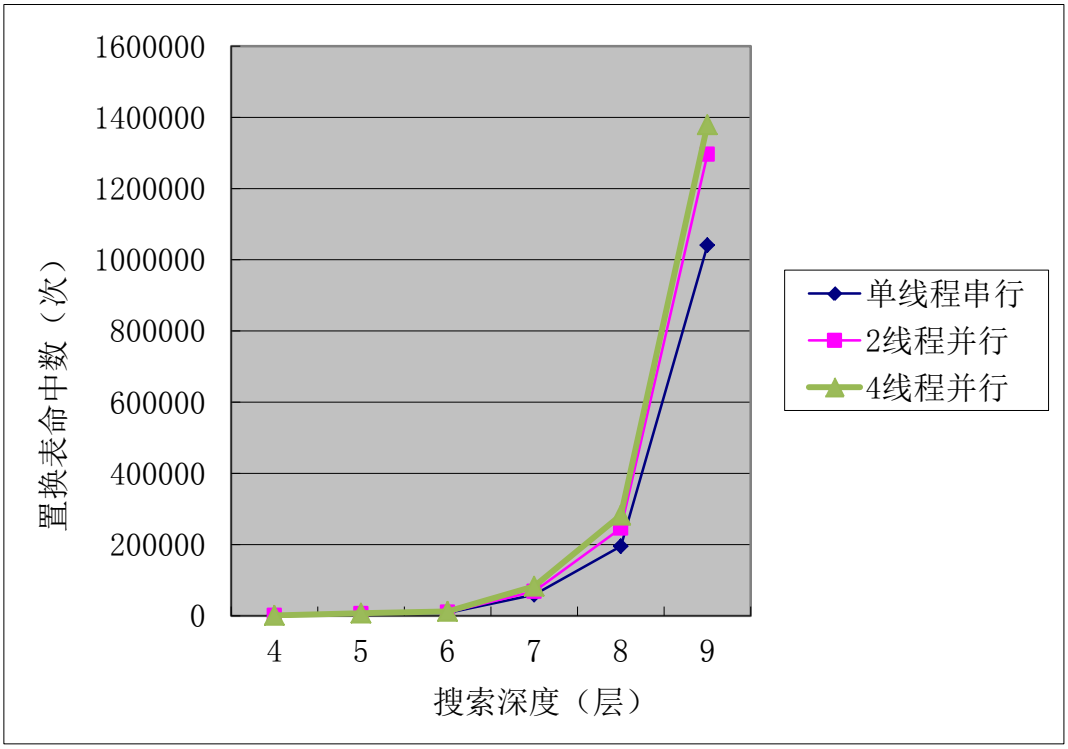


图 4.9 并行混合PVS算法每走一步的平均置换表命中数

4.3.3 实验分析

从表 4.1 和表 4.2 可以看出，串行模式下，在本文设计的中国象棋计算机博弈中，本章采用的融入了置换表、启发策略和空着裁剪的混合 PVS 算法，在众多的博弈树搜索算法和优化措施的组合中，估值结点数最少，性能是最佳的。

此外，还可以看出，在各种增强手段中，空着裁剪对减少估值结点数的作用最大，搜索深度越深，空着裁剪的效率越高，在开局的第 7 层的搜索中能裁剪将近 70%的无效结点；其次是启发策略（历史启发、吃子启发、杀手启发、置换表启发），对减少估值结点数也有很大的作用。

Plaat 认为在国际象棋计算机博弈中，MTD(f)算法略优于 PVS 算法，但在本文设计的中国象棋计算机博弈系统中发现 PVS 算法无论是单独使用还是结合各种优化措施，在搜索深度为 6 层以上的搜索时表现均优于 MTD(f)算法。

从表 4.3 和表 4.4 所示的结果可以看出，在并行线程数不超过 CPU 处理单元数的前提下，随着并行线程数的增加，平均每个线程估值结点数降低了，而置换表的全局命中数也比串行模式下有了较大幅度的提高。从表 4.3 可以看出，在第 4 层，2 线程并行的加速比很低，与串行相比几乎没什么提高，而 4 线程并行的效果甚至略逊于 2 线程并行的效果，这是因为搜索深度太浅，每个线程负担很轻，并行带来的收益和线程的创建、销毁等额外开销相比已经不足道了，因此这种并行模式在浅层搜索优势不明显。但是随着搜索深度的增加，并行模式的优势越来越明显，从第 7 层开始，在 2 线程并行模式下的加速比维持在 1.6 以上，在 4 线程并行模式下的加速比也维持在 2.1 以上。到了第 8 层和第 9 层，2 线程并行模式和 4 线程并行模式下的加速比分别达到了 1.7 和 2.7 以上。各项实验结果显示，对于多核 CPU 架构下，该并行混合 PVS 算法的性能提升是非常明显的，在线程数不超过 CPU 处理单元数时，并行线程越多加速比也越高，但总是低于 CPU 核的数目；并行线程越少效率越高，但总是小于 1。

从表 4.3、图 4.6 和图 4.7 可以看出，在 4 线程并行环境下，并行混合 PVS 算法的加速比和并行效率的绝对数字不高，这是由 PVSplitting 并行策略和 OpenMP 标准的特点决定的。一方面，多个并行线程同时搜索某个结点下的子树时，最初均以同一窗口进行搜索，而在串行模式下，同一结点下的子数，因先搜索的线程可能更新窗口边界，使后搜索的子树的窗口范围变窄。另一方面，OpenMP 中，程序员只能指定并行代码段，线程的创建、分配和销毁由编译系统自动完成，当多个 CPU 处理单元通过并行线程同时对某个结点下的子树进行搜索时，由于每个 CPU 处理单元的工作量不同，可能会出现某些 CPU 处理单元完成了分配给它的多个线程，而此时另一些 CPU 处理单元遇到了 PV 结点，依旧在进行繁重的搜索工作，已经完成搜索的 CPU 处理单元只能空闲下来等待，出

现了额外的同步开销。

下面以图 4.10 和图 4.11 同时说明这两个问题。图 4.10 演示了 4 线程并行模式下的情况，4 个 CPU 处理单元 P1、P2、P3、P4 分别同时启动线程搜索 B 结点下的孩子节点 E、F、G、H，假设空窗探测失败后均以一个较大窗口进行搜索，这里假设窗口为[-20, 50]，尽管某个 CPU 处理单元可能会更新窗口边界；而图 4.11 演示了 2 线程并行模式下的情况，2 个 CPU 处理单元 P1、P2 分别同时以窗口[-20, 50]搜索结点 E、F，假设 P1 或 P2 搜索完后将窗口下界更新为 30，则 P1 和 P2 再去搜索 G、H 时，可以以一个较小的窗口[30, 50]进行搜索，会在搜索过程中引发更多的剪枝。因此多数情况下，并行线程数越多，反而越影响剪枝效率。图 4.10 中，当 CPU 处理单元 P1、P2、P3、P4 完成对结点 E、F、G、H 的搜索后，最先完成搜索的 CPU 处理单元——假设是 P1，启动线程搜索结点 I，而剩下的 3 个 CPU 处理单元只能等待。如果在 OpenMP 中使用 nowait 子句，让空闲等待的处理单元提前启动线程搜索 B 的兄弟结点，如结点 C，此时最左侧线路上的结点 B 尚未获得估值，只能以全窗口搜索结点 C，这种消耗更大；如果采用 DTS 并行策略让空闲的处理单元协助 P1 对结点 I 进行搜索，在 OpenMP 中缺乏这种动态机制的支持，难以实现。

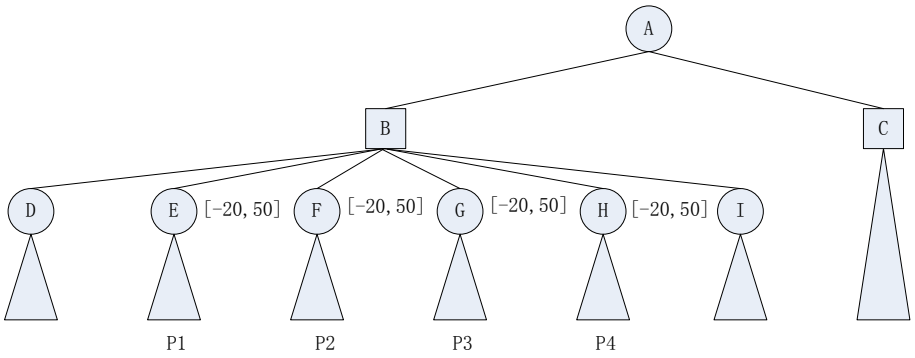


图 4.10 并行线程数为4时的搜索窗口示例

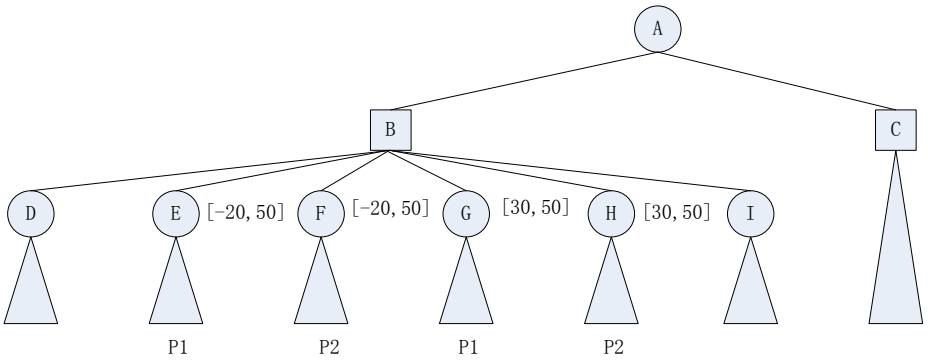


图 4.11 并行线程数为2时的搜索窗口示例

值得一提的是，实验中发现 4 线程并行模式在极少数的情况下，走出了和串行模式下不同的着法，但经过测试，发现两种着法对应局面的估值是一样的。这是因为，当出现多个最佳结点时，4 线程并行模式下可能会出现负责搜索排在较后的最佳结点的线程先完成搜索的情况，使得算法能够走出“冷招”，这是串行混合 PVS 算法做不到的。理论上来看 2 线程并行模式下也有可能出现这种情况，只是概率极低，在本次实验中并未发现，但这种概率显然会随着并行线程数的增加而增加。

4.4 小结

本章针对中国象棋计算机博弈，提出了一种融合了置换表、吃子启发、置换表启发、历史启发、杀手启发、空窗裁剪等优化措施的混合 PVS 算法，每个优化措施均针对中国象棋计算机博弈进行了合理的设计和实现，并使用 PVSplitting 并行策略，在 OpenMP 2.5 标准下对混合 PVS 算法进行了并行化设计，详细介绍了并行化的设计思路，给出了较为详细的算法描述，最后在多核 PC 环境下的真实的中国象棋计算机博弈系统中进行了实验，验证了并行混合 PVS 算法的可行性和有效性，并分析了其特点、优势和不足。

第5章 多核环境下中国象棋博弈程序设计

为了验证第 4 章设计的并行混合 PVS 算法的效果并证明该算法的可行性，本章开发了一个真实的中国象棋计算机博弈程序，作为这种并行混合 PVS 算法的实际应用。首先使用面向对象方法结合 UML 中的类图对中国象棋计算机博弈系统进行分析、设计和建模，然后使用支持 OpenMP 2.5 的 Visual C++ 2010 实现该系统。本文设计的中国象棋计算机博弈系统包含着法生成、搜索引擎、局面估值、交互界面等基本模块，暂不考虑开局库等增强模块和时间控制策略。

5.1 数据结构设计

要开发中国象棋计算机博弈程序，首先要考虑棋子和棋盘如何表示，即设计棋子和棋盘的数据结构，再考虑着法生成模块、博弈树搜索模块、局面估值模块对应的类的数据结构应该如何设计。在设计与编程时，应充分考虑满足面向对象设计与编程中的“针对接口编程”的准则。

5.1.1 棋子和棋盘的表示

（1）棋盘数组

所谓棋盘数组，就是建立棋盘位置和棋子之间的映射关系，即表明棋盘上某个位置的棋子具体是什么类型的棋子[13,30]。中国象棋红黑双方各有帅、车、马、炮、士、相、兵 7 种不同类型的棋子，这样整个棋子类型共有 14 个。最简单的方法是将每一种棋子用不同的整数表示。中国象棋的棋盘共有 10 行 9 列共 90 个位置点。最简单的表示棋盘方法，就是定义一个 10 行 9 列的二维数组如下：

```
int board[10][9];
```

其中元素 board[i][j]表示棋盘第 i 行第 j 列的棋子类型，若这个位置无棋子，则定义为所有棋子类型之外的某个值。著名的王小春的中国象棋博弈软件就是用这种方法表示棋子和棋盘的[12]。

也有一些研究者提出了一种“以空间换时间”的方法来表示棋子和棋盘。将 14 种不同类型的棋子共 32 个，用不同的无符号字符数据表示。将棋盘数组扩充为 16 行 16 列，比真实的棋盘大，将真实的棋盘放在这个矩阵的正中间，上方和下方都多出 3 行，左边多出 3 列，右边多出 4 列[30,45]。再将这个 16 行 16 列的矩阵转换成为一个包含 256 个元素的一维数组，以减少对数组元素读写的偏址运算，将棋盘定义如下：

unsigned char board[256];

其中 board[k]表示第 k/16 行，第 k%16 列的棋子（其中“/”表示整除运算，“%”表示取模运算，下同）。这样能加快运算速度，因为 $16=2^4$，计算坐标的乘除运算可以通过位运算来实现。本文用 16~31 表示红方的某个棋子，用 32~47 表示红方的某个棋子，如表 5.1 所示。

表 5.1 棋子的表示

棋子类型	对应数值
红帅	16
红仕	17, 18
红相	19, 20
红马	21, 22
红车	23, 24
红炮	25, 26
红兵	27, 28, 29, 30, 31
黑将	32
黑士	33, 34
黑象	35, 36
黑马	37, 38
黑车	39, 40
黑砲	41, 42
黑卒	43, 44, 45, 46, 47

采用 256 个元素的一维数组表示棋盘，初始局面对应的棋盘数组元素，如图 5.1 所示，第 4 行到第 13 行，第 4 列到第 12 列的中间的部分（即 k/16 从 3 到 12，k%16 从 3 到 11 的部分）为实际棋盘，红方在下，黑方在上。

这样，不仅计算坐标的乘除运算、甚至判断某个棋子属于哪一方的运算、以及判断棋子是否属于本方的运算，都可以通过位运算来实现，从而加快了运算速度。例如，假设当前走棋方为 side，side=0 表示红方，side=1 表示黑方，chess 为某个棋子对应的值，令 sidetag = (side + 1)*16，则如果条件(chess & sidetag == 1)满足，则为本方棋子，否则为对方棋子。

（2）棋子数组

所谓棋子数组，就是建立棋子与该棋子所处的棋盘位置之间的映射关系，即表明某个具体的棋子位于棋盘的哪个位置[30,45]。以表 5.1 所示的棋子编号和图 5.1 所示的 256 个元素的一维棋盘数组为例，可以定义棋子数组如下：

int piece[48];

其中，piece[k]表示棋子 k 所在的棋盘位置，例如，红帅的棋子编号为 16，在中国象棋的棋盘初始状态下，将位于棋盘数组 board 下标为 199 的位置，因此 piece[16]=199。由于这里棋子的编号为 16~47，所以本文的棋子数组也有冗余，piece[0]~piece[15]没有使用。

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	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	29	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

图 5.1 16行16列矩阵的棋盘表示

（3）双向映射数组

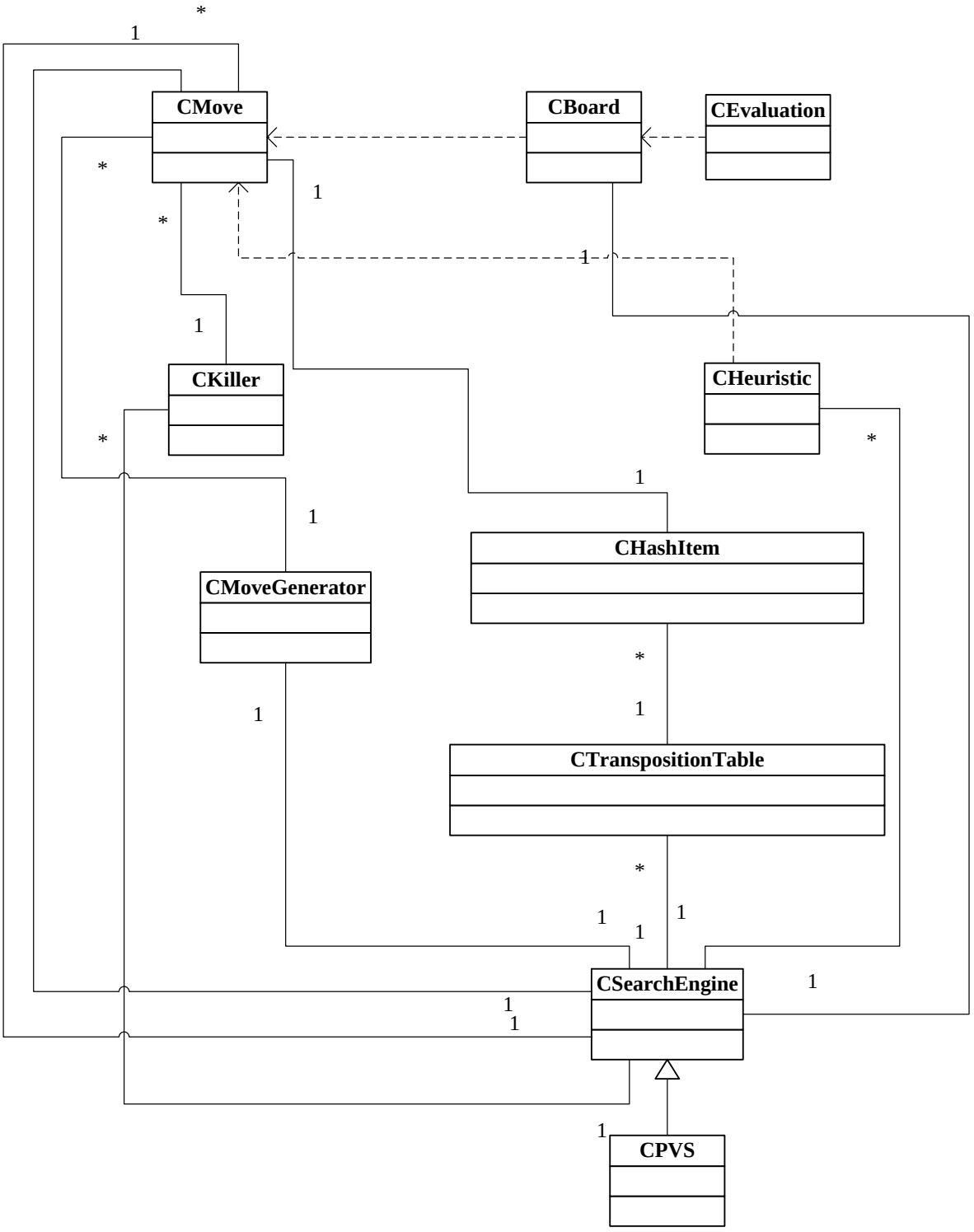
棋盘数组和棋子数组均可以对一个棋盘局面进行完整描述，二者各有优劣。棋盘数组能迅速查找棋盘上任意位置的棋子，但要搜索某一方的棋子的位置信息时需要遍历整个棋盘；棋子数组能方便地知道某个棋子在棋盘的位置，但要了解棋盘局面状况就要遍历整个棋子数组。因此一些研究者提出将棋盘数组和棋子数组同时结合使用，扬长避短，构建一种双向映射数组[30,45]。本文也采用这种双向映射数组对棋盘和棋子的信息进行描述。在棋子移动时，需要同时更新棋盘数组和棋子数组，以免二者出现数据不一致的现象。

5.1.2 类的设计与类图

本文用 UML 中的类图对程序中所使用到的类进行建模。待开发的中国象棋计算机博弈程序的类图，如图 5.2 所示，图 5.2 展示了整个系统的类和类之间的关系以及每个类的数据结构。

图 5.2 中，CBoard 类表示棋盘类，CMove 类表示着法类，CMoveGenerator

类表示着法生成类，CKiller 类表示杀手启发类，CHeuristic 类表示历史启发类，CHashItem 类表示置换表中的数据项，CTranspositionTable 类表示置换表类，CSearchEngine 类表示搜索引擎基类，而 CPVS 类表示使用了并行混合 PVS 算法的搜索引擎基类，是从 CSearchEngine 类中派生出的子类，CEvaluation 类表示局面估值类。



（a）类和类之间的结构图

CMove
+from : int +to : int +capture : int +score : int +MVV : int +LVA : int

（b）CMove 类的结构

CBoard
#board[256] : int #piece[48] : int #ply : int #side : int
+InitBoard() : void +Check() : bool +ChangeSide() : void +MakeMove(in mv : CMove) : int +RestoreMove(in mv : CMove) : void +IsProtected(in lSide : int, in dst : int) : bool

（c）CBoard 类的结构

CMoveGenerator
+MoveList[9][80] : CMove +MoveCount : int +CaptureCount : int +NoCapCount : int
+GreateAllMove(inout board : CBoard, in ply : int) : int +GreateAllCapMove(inout board : CBoard, in ply : int) : int +GreateAllNoCapMove(inout board : CBoard, in ply : int) : int +SortMoveList() : void

（d）CMoveGenerator 类的结构

CHeuristic
#History[256][256] : int
+ResetHistory() : void +GetHistoryScore(in mv : CMove) : int +EnterHistoryScore(in mv : CMove, in depth : int) : void

（e）CHeuristic 类的结构

CKiller
#KillerMoves[256][2] : CMove
+UpdateKiller(in ply : int, in mv : CMove) : void

（f）CKiller 类的结构

CHashItem
#checksum : long long #entry_type : int #eval : int #goodmove : CMove
+Match(in INT64 : long long, in alpha : int, in beta : int, out value : int, out mv : CMove, out HashDepth : int) : bool

（g）CHashItem 类的结构

CTranspositionTable
#TableSize : long #RandHash32 : long #RandHash64 : long long #HashKey32[48][256] : long #HashKey64[48][256] : long long #*pTT[2] : CHashItem
+CalculateInitHashKey() : void +SaveTTMove() : void +RestoreTTMove() : void +SearchTT(in alpha : int, in beta : int, in depth : int, in TableNo : int, in type : int, out eval : int, out mv : CMove, out HashDepth : int) : bool +InsertTT(in type : int, in eval : int, in depth : int, in TableNo : int, in mv : CMove) : void

（h）CTranspositionTable 类的结构

CSearchEngine
#CurBrd : CBoard #mg : CMoveGenerator #Evaluation : CEvaluation #HashTable[ThreadCount] : CTranspositionTable #HH : CHeuristic #Killer[ThreadCount] : CKiller #BestMove : CMove #MaxDepth : int #ply : int
+CreateAndSortMoves(inout board : CBoard, out mg : CMoveGenerator, in depth : int, in mv : CMove) : int +GetAllHistoryScore(in mv : CMove) : int +SearchAllHashTable(in alpha : int, in beta : int, in depth : int, in TableNo : int, out type : int, out eval : int, out mv : CMove) : bool +SaveGoodMove(in mv : CMove, in depth : int, in ply : int) : void +MakeMove(in mv : CMove) : void +RestoreMove(in mv : CMove) : void +MakeNullMove() : void +RestoreNullMove() : void +MakeTheGoodMove() : void +SearchTheGoodMove() : void

（i）CSearchEngine 类的结构

CPVS
+PVSWithTTAndHH_Serial(inout board : CBoard, out mg : CMoveGenerator, in nullmove : int, in depth : int, in alpha : int, in beta : int) : int +PVSWithTTAndHH_Parallel(in nullmove : int, in depth : int, in alpha : int, in beta : int) : int +SearchTheGoodMove() : bool

（j）CPVS 类的结构

CEvaluation
+Evaluate(in board : CBoard, in side : int) : int

（k）CEvaluation 类的结构

图 5.2 中国象棋计算机博弈程序的类图

棋盘和棋子的表示，采用双向映射数组，因此在CBoard类中，包含下面四个数据成员，棋盘数组board、棋子数组piece、以及当前回合数ply和当前走棋方side。

从图5.2可以看出，每个类的独立性较高，而类和类之间的耦合非常合理，这有利于实现较高的可复用性、可修改性和可扩充性。

5.2 着法生成模块设计

CMove 类表示一个着法，包含被移动的棋子的原始位置 from，移动后的位

置 to，该着法的历史得分值 score，以及 MVV 值和 LVA 值，如果是吃子着法，还要包含被吃掉的棋子 capture。

着法生成模块有两个作用，一个作用是是产生计算机在某一局面下的所有可能的着法，另一个作用是检查人类的着法（输入）是否符合规则。后者的设计十分简单，这里只讨论前者。在 CMoveGenerator 类中，定义了一个 CMove 类型的二维数组 MoveList，MoveList[depth][i]表示搜索深度为 depth 的那一层的第 i 个着法。

中国象棋的着法比较复杂，不同类型的棋子的着法规则不同。一种简单的生成着法的方法，就是针对某一方的棋子，扫描棋盘，确定棋子的关联位置，再判断相关的棋子要到达这个关联位置是否有阻碍，如果没有障碍，就是一个可行的着法。虽然按照这种方式，计算机生成一个着法不需要多少时间，但在博弈树搜索时，是以深度优先进行搜索的，对每个结点都要生成对应局面下的全部着法，这个时间消耗也较为可观了。

模板匹配法[30,45]是一种更为可行高效的着法生成方法，根据不同类型棋子的着法规则，枚举出不同类型棋子的着法方向，以及阻碍棋子走向这个方向的其他棋子有可能的位置，存储在若干个数组表示的模板中。本文采用模板匹配法，以加快着法生成的速度。

例如，马走“日”字，则有 8 个可行位置，这 8 个可行位置，任一位置，只要没有己方棋子，则马就可以到达这个位置。可以用一个一维数组保存这 8 个位置，如下：

```
char KnightDir [8] = { -0x21, -0x1f, -0x12, -0x0e, +0x0e, +0x12, +0x1f, +0x21 };
```

而每个方向，如果在相应的“马腿”处有棋子，则相应的位置就不可行，一个“马腿”可以阻碍马向两个位置落子，可以再使用一个一维数组，保存每个方向对应的“马腿”的位置，如下：

```
char KnightCheck[8] = { -0x10, -0x10, -0x01, +0x01, -0x01, +0x01, +0x10, +0x10 };
```

对于相，可行位置也是固定的，也可以用同样的方法处理。对于帅、士等，可行位置同样是固定的，没有诸如“马腿”、“象眼”一说，只需要保存方向数组即可。

对于车、炮这样可行位置不固定，沿着一些方向有很多种可能着法的情况，也可以用模板匹配法，但本文仅保存车、炮沿着水平方向和垂直方向走一步的可行位置，事实上这里只是表示一个可走方向的矢量。生成车、炮的着法时，需要沿着水平方向和垂直方向扫描，将扫描到的位置与棋子移动前的位置的距离，乘以这个矢量，得到的位置作为车、炮着法的关联位置。对于炮来说，还要考虑“爬

山”吃子的情况。

5.3 博弈树搜索模块设计

博弈树搜索模块是本文开发的中国象棋计算机博弈程序中最为核心也是最为复杂的部分，负责从某个局面开始，对博弈树进行敌对搜索，找到一个在计算机看来最好的着法。

5.3.1 并行混合 PVS 算法的应用

博弈树搜索模块涉及的类较多，从图 5.2 可以看出，搜索引擎基类 CSearchEngine 使用了历史启发类 CHeuristic、杀手启发类 CKiller、置换表类 CTranspositionTable 等多个类，这里使用了聚集关系。而并行混合 PVS 算法类 CPVS 从类 CSearchEngine 中继承。父类 CSearchEngine 提供了一个纯虚函数 SearchTheGoodMove，定义了搜索最佳着法的接口，子类 CPVS 中对 SearchTheGoodMove 函数予以具体实现。这么做的好处是，所有的搜索算法都可以用一个统一的接口来定义，如果要使用其他搜索算法，可以从类 CSearchEngine 中再派生出一个新类予以实现，其它的类几乎不用修改，这样就满足了面向对象方法中的“对扩充开放，对修改封闭”的原则（开放-封闭原则）。

从图 5.2 可以看出，第 4 章中介绍的相关算法的操作对象和操作过程，已经作为了相关的类的数据成员和成员函数。其中，类 CPVS 中的成员函数 PVS_Parallel 实现了第 4 章提出的并行混合 PVS 算法，该类的另一成员函数 SearchTheGoodMove 调用成员函数 PVS_Parallel，实现了利用并行混合 PVS 算法搜索最佳着法的过程。

为符合 OpenMP 标准，类 CSearchEngine 和类 CPVS 相关的数据成员，应设计为静态数据成员，这样在并行混合 PVS 算法中才能在 for 循环并行化时正确设置线程的私有变量和共享变量。

5.3.2 搜索延伸

第 2 章介绍的各种博弈树搜索算法和第 4 章设计的并行混合 PVS 算法，对博弈树的搜索都只是搜索到一个固定的最大搜索深度，到了叶子结点就返回估值，这个固定的深度被称为水平线（Horizon）。但有时候局面会有反复剧烈动荡的情况，这种做法会导致其后隐藏的不利局面不能被算法识别出来，这种现象被称为水平线效应（Horizon Effect）[30]。

显然，如果不为局面估值函数引入更多复杂的知识，解决水平线效应的一个简单办法就是让搜索算法在某些特殊情况下搜索得更深，直到局面相对平缓则停

止搜索，这种情况叫做搜索延伸（Search Extension）[30]，也叫做宁静搜索（Quiescence Search）[30]。引起水平线效应的原因和具体的棋类游戏规则有关。在中国象棋中，引发水平线效应的原因也很多，主要体现在[14]：（1）双方兑子，（2）被将军时，（3）单一着法，（4）遇到杀棋威胁时。

本文仅对情况（1）和（2）进行搜索延伸，遇到将军情况和兑子情况时，进行“将军延伸”和“兑子延伸”，使开发的中国象棋博弈程序更加实用。每当遇到将军局面或者最近的连续两步着法都是吃子着法，则自动将搜索深度加 1。这种加入了“将军延伸”和“兑子延伸”的搜索延伸算法的流程，可表示为如下的类 C/C++伪代码：

```
int SearchExtension (board, depth, alpha, beta, nullmove)
{
    if(depth == 0)
    {
        if(check(board)) depth++; //将军延伸
        else if(前一步着法是吃子着法 && 再前一步着法是吃子着法) depth++; //兑子延伸
    }
    if(GameOver(board) || depth == 0)
    {
        ……; //此处的代码和混合 PVS 算法的代码类似
    }
    …… //之后的代码和混合 PVS 算法的代码类似
}
```

5.4 局面估值模块设计

CEvaluation 类是局面估值类。局面估值模块使用研究人员设计的估值函数，对博弈双方的局势优劣程度做出一个客观定量的分析。对棋类游戏的计算机博弈程序来说，计算机走棋的智能，除了取决于博弈树搜索算法、搜索深度之外，另一个重要的因素就是局面估值函数。中国象棋计算机博弈程序也是如此，只有对博弈树的结点所表示的棋盘局面的优劣程度进行较为精确的估值，才能使计算机拥有更高的棋力，反之，如果估值函数不能正确反映棋盘局面的优劣程度，即使再好的博弈树搜索算法，也会走出很多“臭棋”。如果说博弈树搜索算法的目的是让计算机想的更深，那么评估函数的目的是让计算机想的更准。

中国象棋的局面优劣程度的判断非常复杂，不仅依赖于具体的游戏规则，也依赖于人类对游戏的经验常识。故在设计估值函数时，应该将人类感性的、定性

的认知，转换为计算机中理性的、定量的模型。在 α-β 搜索/剪枝算法及其衍生算法中，需要对博弈树中到达最大搜索深度的结点进行估值，配合搜索算法找到目前看来最好的着法。本节首先介绍静态估值函数，再通过自适应遗传算法对静态估值函数进行优化，最后在 OpenMP 环境下实现自适应遗传算法的并行化。

5.4.1 静态估值函数

一般而言，中国象棋红黑双方局面估值的计算，要考虑的因素包括不同类型棋子的基本价值、棋子的灵活度价值、棋子的位置价值、棋子受威胁的程度、棋子受保护的程度等[12,45]。因此，红黑双方棋子价值总和的计算，可分别表示如下：

$$Value_r(p) = PieceValue_r(p) + FlexibleValue_r(p) + PositionValue_r(p) - ThreatenValue_r(p) + GuardValue_r(p) \tag{5.1}$$

$$Value_b(p) = PieceValue_b(p) + FlexibleValue_b(p) + PositionValue_b(p) - ThreatenValue_b(p) + GuardValue_b(p) \tag{5.2}$$

其中，$Value_r(p)$和 $Value_b(p)$分别表示红黑双方棋子价值总和，$PieceValue_r(p)$和 $PieceValue_b(p)$分别表示红黑双方在局面 p 的棋子基本价值总和，$FlexibleValue_r(p)$和 $FlexibleValue_b(p)$分别表示红黑双方在局面 p 的棋子灵活度价值总和，$PositionValue_r$和 $PositionValue_b$分别表示红黑双方的棋子位置价值总和，$ThreatenValue_r(p)$和 $ThreatenValue_b(p)$分别表示红黑双方在局面 p 的棋子受威胁的分值总和，$GuardValue_r(p)$和 $GuardValue_b(p)$分别表示红黑双方在局面 p 的棋子受保护的分值总和。

对整个棋盘局面的估值，为红黑双方棋子价值总和的差值，可表示如下：

$$Eval(p) = Value_r(p) - Value_b(p) \tag{5.3}$$

其中，$Eval(p)$表示整个棋盘局面的估值，$Value_r(p)$和 $Value_b(p)$分别表示红黑双方棋子价值总和。

（1）棋子的基本价值

从直观上来说，中国象棋博弈双方，拥有较多子力的一方一般占优势，而子力较少的一方一般占劣势。而每一种不同类型的棋子，重要性或者战斗力也是不同的。根据规则，中国象棋中，帅（将）的重要性最大，某一方如果失去了帅就输了。根据经验，车的战斗力最强，接近两个炮的战斗力，而炮的战斗力和马的战斗力接近，但都小于车的战斗力，仕（士）和象（相）不能“过河”，两者的战斗力也相当，略高于两个兵（卒），但肯定弱于马或者炮。如果不考虑不同棋手的运子习惯与战术策略，不同类型的棋子的基本价值关系，可以通过下面的不等式表示：

帅 > 车 > 马 ≈ 炮 > 士 ≈ 象 > 兵

其中帅的基本价值，应大于某一方其他所有棋子的基本价值总和。这里将中国象棋的不同类型的棋子的基本价值，暂定为表 5.2 所示的基本价值。

表 5.2 每个种类的棋子的基本价值

棋子种类	将	士	相	马	车	炮	兵
基本价值	10000	250	250	350	550	350	100

对红黑双方棋子的基本价值总和的计算，可分别表示如下：

$$PieceValue_r(p) = \sum_{i \in r} V_i \cdot n_i(p) \tag{5.4}$$

$$PieceValue_b(p) = \sum_{i \in b} V_i \cdot n_i(p) \tag{5.5}$$

其中 r 和 b 表示红方和黑方，p 表示某个棋盘局面，$PieceValue_r(p)$和$PieceValue_b(p)$分别表示红、黑双方在局面 p 的棋子基本价值总和，V_i 表示类型为 i 的棋子的基本价值，$n_i(p)$表示类型为 i 的棋子在局面 p 中的数量。

（2）棋子的灵活度价值

中国象棋中，棋子的灵活度也叫做棋子的运子空间，某一方的棋子的灵活度指的是处于其棋子控制的位置的数量总合，控制的位置越多局面越有利，反之越不利。例如，车的基本价值很大，但在刚开局时，位于最边角的位置，没几个可走的空间，更无法攻击对方，开局应尽早将车移动到重要位置，让它发挥更大价值。马有八面威风，但一旦被困在某个角落，也体现不了多大价值。棋子灵活性同时也体现了该棋子可走步数的多少，当某个棋子每多一个可走位置时，灵活性相应增加，应加上相应分值。这里将中国象棋不同类型的棋子每多一个可走位置时增加的分值，暂定为表 5.3 所示的分值。

表 5.3 每个种类的棋子每多一个可走位置时加上的分值

棋子种类	将	士	相	马	车	炮	兵
每多一个可走位置时加上的分值	0	1	1	12	6	6	15

对红黑双方棋子的灵活度价值总和的计算，可分别表示如下：

$$FlexibleValue_r(p) = \sum_{i \in r} F_i \cdot m_i(p) \tag{5.6}$$

$$FlexibleValue_b(p) = \sum_{i \in b} F_i \cdot m_i(p) \tag{5.7}$$

其中 r 和 b 表示红方和黑方，p 表示某个棋盘局面，$FlexibleValue_r(p)$和$FlexibleValue_b(p)$分别表示红、黑双方在局面 p 的棋子灵活度价值总和，F_i 表示

棋子 i 每多一个可走位置时增加的分值，$m_i(p)$表示棋子 i 在局面 p 中的当前着法数。

（3）棋子的位置价值

在中国象棋中，一个棋子在棋局中的价值，还与其所处的位置有密切关系。同一个棋子，在棋盘中所处的具体位置不同，表现出的价值也有所不同。进攻类的棋子，一旦处在对方九宫附近或者处在重要线路上，就会给对方造成较大威胁。例如，兵的基本价值很小，灵活度价值也很低，但一旦逼近对方九宫就会给对方造成巨大威胁，价值大增，不过兵一旦“沉底”之后，价值又会减少。这就需要为不同类型的棋子构造一个针对不同棋盘位置的附加价值。

这里以红马、红兵、红车为例，它们在棋盘不同位置的位置值，分别暂定为如图 5.3、图 5.4、图 5.5 所示的分值，其中红方在下。可以看出，一旦某个棋子在对方的九宫附近，或者在其他重要线路上，则棋子的位置价值相应的增加。例如从图 5.3 可以看出，马处于对方“卧槽”的位置时，位置值最大，在己方的九宫中心时，因有可能对己方的帅、士的着法造成一定障碍，故位置值最小。

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	70	80	90	80	70	80	90	80	70	0	0	0	0
0	0	0	80	110	125	90	70	90	125	110	80	0	0	0	0
0	0	0	90	100	120	125	120	125	120	100	90	0	0	0	0
0	0	0	90	100	120	130	110	130	120	100	90	0	0	0	0
0	0	0	90	110	110	120	100	120	110	110	90	0	0	0	0
0	0	0	90	100	100	110	100	110	100	100	90	0	0	0	0
0	0	0	80	90	100	100	90	100	100	90	80	0	0	0	0
0	0	0	80	80	90	90	80	90	90	80	80	0	0	0	0
0	0	0	70	75	75	70	50	70	75	75	70	0	0	0	0
0	0	0	60	70	75	70	60	70	75	70	60	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 5.3 红马的位置值

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	10	10	10	20	25	20	10	10	10	0	0	0	0
0	0	0	25	30	40	50	60	50	40	30	25	0	0	0	0
0	0	0	25	30	30	40	40	40	30	30	25	0	0	0	0
0	0	0	20	25	25	30	30	30	25	25	20	0	0	0	0
0	0	0	15	20	20	20	20	20	20	20	15	0	0	0	0
0	0	0	10	0	15	0	15	0	15	0	10	0	0	0	0
0	0	0	10	0	10	0	15	0	10	0	10	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 5.4 红兵的位置值

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	160	170	160	150	150	150	160	170	160	0	0	0	0
0	0	0	170	180	170	190	250	190	170	180	170	0	0	0	0
0	0	0	170	190	200	220	240	220	200	180	170	0	0	0	0
0	0	0	180	220	210	240	250	240	210	220	180	0	0	0	0
0	0	0	180	220	210	240	250	240	210	220	180	0	0	0	0
0	0	0	180	220	210	210	250	210	210	220	180	0	0	0	0
0	0	0	170	190	180	220	240	220	200	190	170	0	0	0	0
0	0	0	170	180	170	170	160	170	170	180	170	0	0	0	0
0	0	0	160	170	160	160	150	160	160	170	160	0	0	0	0
0	0	0	150	160	150	160	150	160	150	160	150	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 5.5 红车的位置值

对红黑双方棋子的位置价值总和的计算，可分别表示如下：

$$PositionValue_r(p) = \sum_{i \in r} W_{i,j} \tag{5.8}$$

$$PositionValue_b(p) = \sum_{i \in b} W_{i,j} \tag{5.9}$$

其中 p 表示表示某个棋盘局面，r 和 b 表示红方和黑方，$PositionValue_r(p)$ 和 $PositionValue_b(p)$分别表示红、黑双方在局面 p 下的棋子位置价值总和，$W_{i,j}$ 表示棋子 i 在棋盘位置 j 处对应的位置值。

（4）棋子受威胁的程度

在中国象棋中，如果某方的一个棋子位于对方棋子一步可达的着法位置上，则认为被对方棋子威胁，因为如果下一步轮到对方走棋，这个棋子就很有可能被对方吃掉。某方受敌方威胁的棋子越多，被威胁方的局面越不利，受威胁棋子的价值应该相应地降低。

对红黑双方棋子受威胁的分值总和的计算，可分别表示如下：

$$ThreatenValue_r(p) = \sum_{\substack{i \in r \\ j \in b}} f(PieceValue_i, PieceValue_j \mid 棋子i在棋子j的着法上) \tag{5.10}$$

$$ThreatenValue_b(p) = \sum_{\substack{i \in b \\ j \in r}} f(PieceValue_i, PieceValue_j \mid 棋子i在棋子j的着法上) \tag{5.11}$$

其中 p 表示表示某个棋盘局面，$ThreatenValue_r(p)$和 $ThreatenValue_b(p)$分别表示红、黑双方在局面 p 下的棋子受威胁的分值总和，$PieceValue_i$ 和 $PieceValue_j$ 分别表示棋子 i 和棋子 j 的基本价值，其中棋子 i 在对方棋子 j 的一步可达的着法上。这表明，棋子受威胁的分值，是与威胁棋子的价值与被威胁棋子的价值相关的一个函数 f。这个函数可以自行设计，并且和具体的走棋方有关。如果是红方的帅被威胁，而下一步轮到黑方走棋，红方必输，估值函数可以返回对应的极小的负数，但如果是红方走棋，这个将军有可能被化解。如果是红方其他棋子被威胁，下一步轮到红方走棋，红方有可能化解威胁，所以此时威胁较小，但如果是轮到黑方走棋，这个威胁分值就较大了。

（5）棋子受保护的程度

在中国象棋中，如果某方的一个棋子位于己方棋子一步可达的着法位置上，则认为被自己的棋子保护，因为如果下一步轮到对方走棋，如果己方这个受保护的棋子位于对方的某个棋子一步可达的着法的位置，对方会因有所顾忌不一定会吃掉己方这个棋子。某方受到自己棋子保护的棋子越多，局面相对有利，受保护的棋子价值应该相应地升高。

经常出现某一方的两个棋子“互保”的情况，如“连环马”、“霸王车”、“担子炮”等局势。如果出现这种情况，两个“互保”的棋子的价值都会相应升高。

对红黑双方棋子受保护的分值总和的计算，可分别表示如下：

$$GuardValue_r(p)=\begin{cases}\sum_{\substack{i,j\in r\\ k\in b}} g_1(PieceValue_i \mid \text{棋子i同时在棋子j和棋子k的着法上})\\ \sum_{i,j\in r} g_2(PieceValue_i \mid \text{棋子}i\text{在棋子j的着法上})\end{cases} \tag{5.12}$$

$$GuardValue_b(p)=\begin{cases}\sum_{\substack{i,j\in b\\ k\in r}} g_1(PieceValue_i \mid \text{棋子i同时在棋子j和棋子k的着法上})\\ \sum_{i,j\in b} g_2(PieceValue_i \mid \text{棋子}i\text{在棋子j的着法上})\end{cases} \tag{5.13}$$

其中 p 表示表示某个棋盘局面，$GuardValue_r(p)$和 $GuardValue_b(p)$分别表示红、黑双方的棋子在局面 p 下受保护的分值总和，$PieceValue_i$ 表示棋子 i 的基本价值，其中棋子 i 在己方的棋子 j 的一步可达的着法上。这表明，棋子受保护的分值对应的函数，要分情况讨论。如果棋子 i 处于己方的棋子 j 保护，同时处于对方的棋子 k 的一步可达的着法上，则表示为函数 g_1，此时棋子 i 受到对方威胁，这个受保护的分值应该较高，以抵消一部分因受威胁扣除的分值；如果棋子 i 处于己方的棋子 j 保护，并没有受到对方的棋子威胁，则表示为函数 g_2，这个受保护的分值相对较低，因为此时无需抵消一部分因受威胁扣除的分值。

5.4.2 用遗传算法优化估值函数

静态评估函数中的每一项参数，在初始设定时往往依赖于研究者的棋类知识和经验。要使中国象棋博弈程序拥有更高的智能，可以采用某种智能优化算法确定相关的参数，如粒子群优化算法、神经网络等，也有研究者采用“二次估值”的方法，通过“局势因子”调整估值函数，使用遗传算法对估值函数进行优化也是一种有效的办法。

遗传算法（Genetic Algorithm，GA）[46]是 Holland 于 1975 年提出的一种模拟自然界中生物进化和淘汰的计算模型。遗传算法的基本思想是[46]：对问题的可行解进行编码，将每个可行解的编码看作个体，选择一些可行解，构成初始种群。设计评价个体优劣的适应值函数，使目标函数越好的个体的适应值越高，计算种群中每个个体的适应值，依据适值的大小确定种群中每个个体的选择概率，让适应值越大的个体被复制的几率越高，选中的个体再以一定的概率 P_c 进行交叉和一定的小概率 P_m 进行变异，生成下一代种群，反复迭代，直到满足收敛条件，就得到了问题的满意解。

遗传算法的基本流程，如图 5.6 所示。

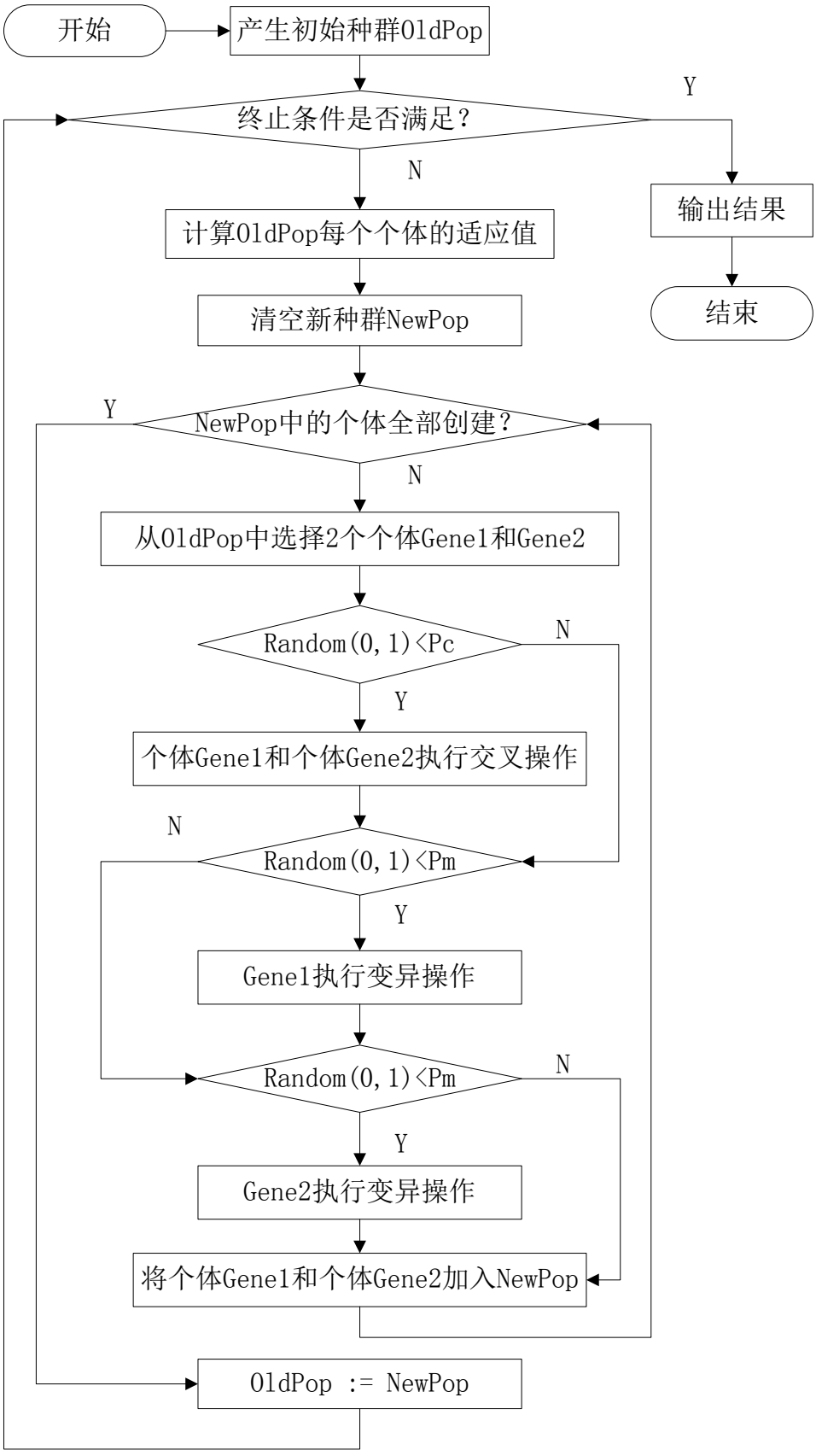


图 5.6 遗传算法流程图

使用遗传算法优化中国象棋的局面估值函数，可以把具有不同参数的估值函数看作不同的个体，把个体之间的比赛结果作为适应度函数，用来评价个体的优劣程度，不同的个体，表征的估值函数中的参数不同，但其他模块，如搜索算法、着法生成及相应的数据结构都是相同。每个参数，均采用二进制编码方式。因为帅（将）的基本价值无疑是一个非常大的数，故帅（将）的基本价值不参与进化。

（1）选择

这里对选择操作采用“正比选择”和“锦标赛选择”。为了克服传统的锦标赛选择速度太慢的缺点，国内的王骄等研究者对锦标赛选择做了改进，将共有 m 个个体的种群分成 n 组，每个组在组内进行两两对弈，每个组的冠军为选择出来的最佳个体，这 n 个最佳个体再经过交叉、变异，产生 m-n 个新个体，形成一个新种群[47]。这种方法较大幅度提高了选择的速度，但是也有不足之处。一方面，个体的分组是随机的，如果分组情况不理想，容易陷入局部最优；另一方面，遗传算法不需要知道外界信息，个体与个体之间的对弈都是封闭的，这种方法难免存在局限性。本文对此进行了改进，并将选择方法定义为两个阶段。

第一阶段：采用正比选择。选择某个已有的中国象棋博弈系统作为陪练程序，每个个体分别和这个陪练程序进行交换先后手的两次对弈，得分与对弈的回合数有关。本文选择文献[12]中王小春的中国象棋计算机博弈程序 chess 稍加修改，作为陪练程序，该程序属于学习交流用的开源软件，棋力约处于中等水平。对每个个体的优劣程度而言，如果陪练程序获胜，则回合数越多越好；如果己方获胜，则回合数越少越好。某个个体的适应值，可以看作该个体与所有陪练程序比赛的得分总和，越优良的个体，被选择的概率越高。这里采用精英主义[47]，每一次迭代都将最好的个体原封不动地复制到下一代。接着进行交叉、变异操作生成下一代种群，在这个过程中保持种群规模不变。

第二阶段：采用锦标赛选择。将种群中的 NP 个个体分成 m 组，分组时根据第一阶段的结果将个体按照适应值从好到坏排序，排序后，编号为 num 的个体分配给第 k 组，其中 k = (num % m)。每个组内的个体两两之间进行交换先后手的对弈。这里再次采用精英主义，根据积分，选出每个组积分最多的冠军个体，这 m 个冠军个体，经过交叉、变异操作再产生 NP - m 个新个体，一起再次构成规模为 NP 的新种群，继续进行下一轮迭代。

（2）交叉

这里使用最为简单的单点交叉。进行交叉操作时，根据交叉概率 P_c，某两个父代个体的对应参数随机选择一个二进制位作为交叉点，生成两个子代个体的对应参数。种群中所有个体的每一种不同的参数，都互不影响的进行这样的操作。

（3）变异

变异操作也是根据估值函数的参数进行操作，每一个参数根据变异概率 P_m，

随机将该参数的一个二进制位将其进行按位取反操作。

遗传算法在进化前期，群体中的大多数个体尚未具备很好的性能，此时交叉概率 P_c 和变异概率 P_m 可以相对稍大；但进行到进化的后期，群体中个体基本上都具备了较好的适应值，如果此时再对个体进行较大概率的交叉和变异，反而有可能会导致下一代的优良性整体不如上一代的情况。Srinvivas 等人提出了自适应遗传算法（Adaptive Genetic Algorithm，AGA）[47]，在进化的后期，使交叉概率 P_c 和变异概率 P_m 随着群体的平均适应值动态改变。

自适应遗传算法中，P_c 和 P_m 的计算，显然与群体的平均适应值和目前最优个体的适应值有关，但并没有一个固定的公式。对于一个具体的优化问题，要进行反复实验来确定 P_c 和 P_m 的计算公式。一种在棋类计算机博弈系统中常用的动态计算 P_c 和 P_m 的计算公式为[47]：

$$P_c = \begin{cases} P_{c_1} - \dfrac{(P_{c_1} - P_{c_2}) \cdot (f' - f_{avg})}{f_{\max} - f_{avg}}, & f' \geq f_{avg} \\ P_{c_1}, & f' < f_{avg} \end{cases} \tag{5.14}$$

$$P_m = \begin{cases} P_{m_1} - \dfrac{(P_{m_1} - P_{m_2}) \cdot (f_{\max} - f)}{f_{\max} - f_{avg}}, & f' \geq f_{avg} \\ P_{m_1}, & f' < f_{avg} \end{cases} \tag{5.15}$$

本文也使用这个公式动态计算 P_c 和 P_m。式中，f_{max} 为群体中最佳个体的适应值，f_{avg} 为当前代中群体平均适应值，f’为进行交叉操作的两个个体中适应值的较大值，f 为进行变异操作的个体的适应值。而 P_{c1}、P_{c2} 和 P_{m1}、P_{m2} 均表示某个取值范围的上界和下界，可以进行如下取值[47]：P_{c1}=0.9，P_{c2}=0.6，P_{m1}=0.1，P_{m2}=0.001。这样，群体的多样性和遗传算法的收敛性均有了较好的体现。

5.4.3 遗传算法在 OpenMP 下的并行化

估值函数中的参数繁多，用上述自适应遗传算法对估值函数的参数进行优化，是非常耗时的一项工作，但遗传算法具有天然的可并行性。遗传算法的并行形式分为四类[48]：（1）个体适应值评价操作内的并行；（2）种群中所有个体适应值评价的并行性；（3）算法操作内部的并行性；（4）种群分组的并行性。

针对 OpenMP 标准下的多核 CPU 环境，可以将自适应的遗传算法进行并行化处理，提高算法的时间效率。这里将（2）、（3）、（4）三种方式结合起来，优化估值函数的自适应遗传算法在 OpenMP 下的并行化过程，可用类 C/C++伪代码表示如下：

```
Procedure GA()
{
    #pragma omp parallel for
    for (i = 0; i < NP; i++) 初始化初始种群的第 i 个个体;
    while (终止条件不满足)
    {
        if (第一阶段的选择没有结束) //第一阶段的选择
        {
            #pragma omp parallel for
            for (i = 0; i < NP; i++)
            {
                第 i 个个体和陪练程序进行交换先后手的对弈,记录结果;
            }
            #pragma omp parallel for
            for (i = 0; i < NP; i++) 计算第 i 个个体的适应值;
            种群内的冠军个体直接复制到下一代;
        }
        else //第二阶段的选择
        {
            #pragma omp parallel for
            for (i = 0; i < NP; i++) 将第 i 个个体划分到第(i % m)组;
            #pragma omp parallel for
            for (k = 0; k < m; k++)
            {
                第 k 组内的个体两两之间进行交换先后手的对弈;
                计算第 k 组内个体的适应值;
                第 k 组内的冠军个体直接复制到下一代;
            }
        }
        r = 已复制的精英个体数;
        #pragma omp parallel for
        for (j = r; j < NP; j += 2)
        {
            从当前种群中选择两个个体;
            计算交叉概率 Pc;
```

```
                if(Random(0, 1) >= Pc)
                {
                    将选择的两个个体执行交叉操作后生成 gene1 和 gene2;
                }
                else 将选择的两个个体直接复制为 gene1 和 gene2;
                计算变异概率 Pm;
                if(Random(0, 1) >= Pm) 对 gene1 进行变异操作;
                计算变异概率 Pm;
                if(Random(0, 1) >= Pm) 对 gene2 进行变异操作;
            }
        }
    }
```

5.4.4 优化实验结果

以每个不同种类的棋子的基本价值的优化过程为例，迭代到 500 代时，各参数基本上没什么变化了。本文选择遗传算法优化过程（第一阶段和第二阶段整个过程）中的第 50 代、第 100 代、第 150 代、第 200 代、第 250 代、第 300 代、第 350 代、第 400 代、第 450 代、第 500 代的最优个体，以及陪练程序，进行交换先后手的对弈，每胜一次，累加 2 分，平局双方累加 1 分，每负一次，不累加分（累加 0 分）。每一代最优个体及陪练程序的对弈积分情况，如表 5.4 所示。

表 5.4 个体对弈积分表

代数	50	100	150	200	250	300	350	400	450	500	陪练
胜局场次	1	1	4	6	7	11	11	15	18	18	8
败局场次	19	18	14	13	9	6	5	4	0	0	12
平局场次	0	1	2	1	4	3	4	1	2	2	0
累计积分	2	3	10	13	18	25	26	31	38	38	16

表 5.4 中，每一代最优个体对弈积分对应的折线图，如图 5.7 所示。

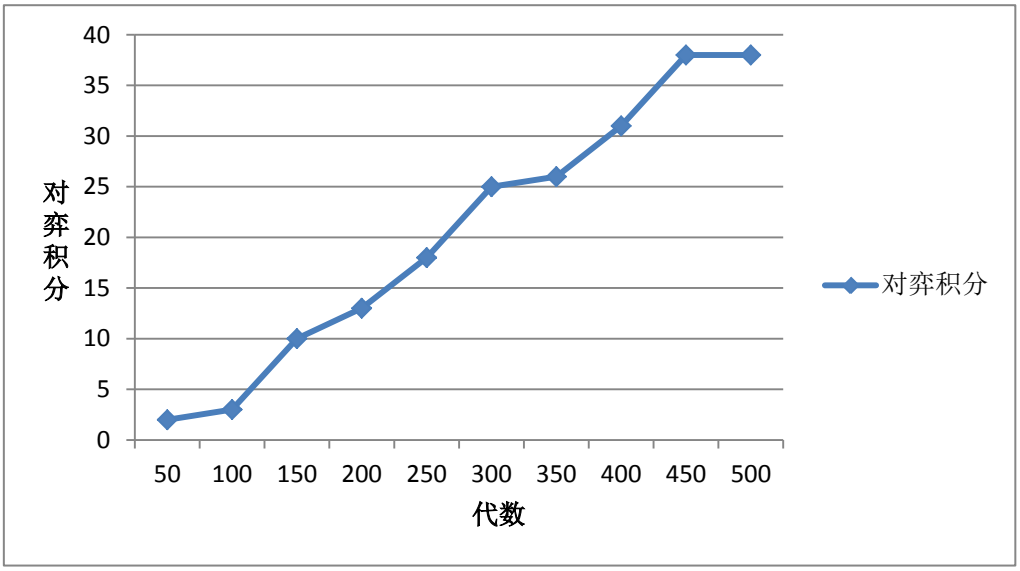


图 5.7 个体对弈积分变化图

从表 5.4 和图 5.7 可以看出，总体而言，对弈积分随着代数的增加而递增，每一代个体的智能总体处于上升状态，在后阶段，陪练程序已无取胜可能。这表明博弈系统的胜率越来越高，棋力越来越强，估值函数对局面的优劣程度的反应越来越准确。

本文仅给出每个种类的棋子的基本价值的优化结果，及其与优化前的基本价值的比较，如表 5.5 所示，其中将未参与进化。

表 5.5 棋子基本价值的优化结果

棋子种类	将	士	相	马	车	炮	兵
基本价值（优化前）	10000	250	250	350	550	350	100
基本价值（优化后）	10000	291	284	498	823	486	85

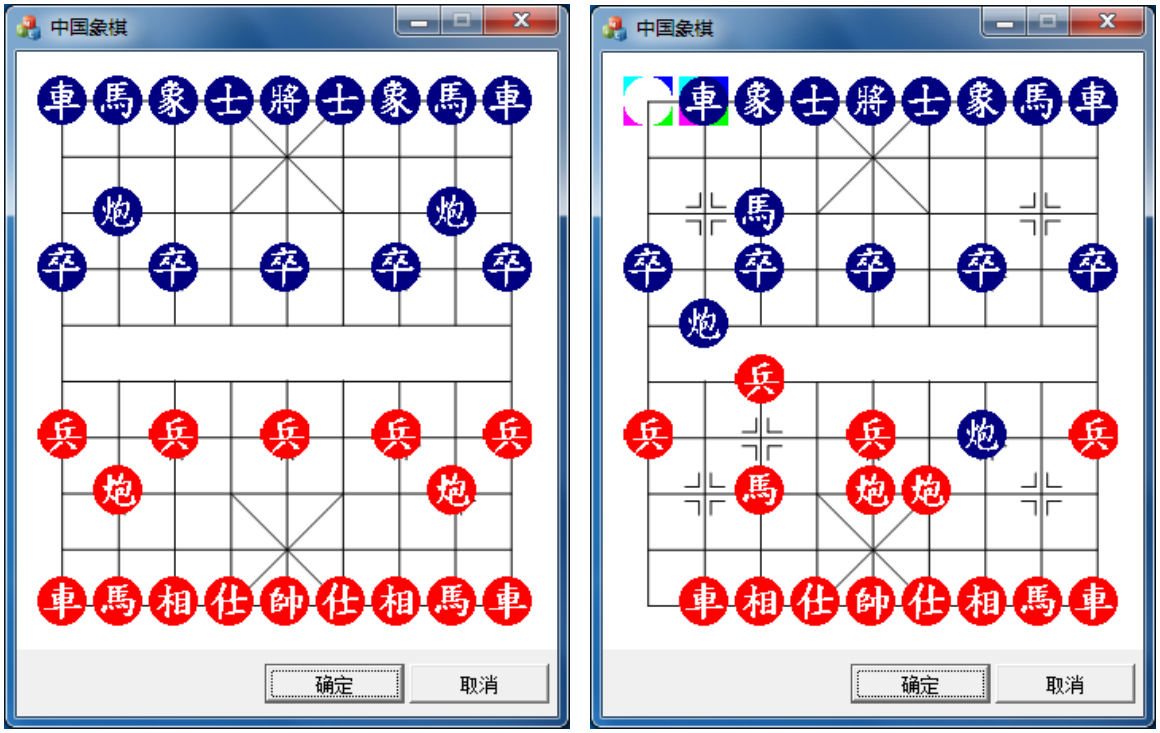
从表 5.5 可以看出，棋子的基本价值的优化结果中，一个车的基本价值约等于一个马和一个炮的基本价值之和，一个士和一个相的基本价值约相当，一个马或一个炮的基本价值约相当于一个士和一个相的基本价值之和，这与人类棋手的经验基本吻合。

5.5 界面与运行结果

界面负责和使用者的交互，界面设计并非中国象棋计算机博弈程序的核心，本文仅作简单介绍。这里在 Windows 7 操作系统环境下，使用 Visual C++ 2010

进行开发，综合使用 C++语言和 MFC 类库。最终得到程序运行的主界面，如图 5.8 所示。

系统刚运行时的界面如图 5.8（a）所示。系统运行时，设置好搜索深度后，人类棋手可以和计算机进行对弈。例如，设定搜索深度为 9 层时，人类棋手执红方走“炮八平五”以当头炮开局，则计算机执黑方以“马 2 进 3”应对，红方“兵七进一”，黑方“炮 2 进 2”，红方“马八进七”，黑方“炮 8 平 7”，红方“炮二平四”，黑方“炮 7 进 4”，红方“车九平八”，黑方“车 1 平 2”……，如图 5.8（b）所示。



（a）初始化界面　　（b）开局对弈界面

图 5.8 中国象棋计算机博弈程序的主界面

5.6 小结

本章以多核 PC 为硬件环境，严格遵循面向对象方法，设计了一个真实的中国象棋计算机博弈系统，并以支持 OpenMP 2.5 的 Visual C++ 2010 下予以实现。概括性地介绍了数据结构和类图的设计，以及着法生成模块、搜索引擎模块、局面估值模块和交互界面的设计，将 OpenMP 下的并行混合 PVS 算法运用于搜索引擎模块。重点介绍了静态估值函数的设计，以及用遗传算法优化估值函数的方法，并提出了优化估值函数的遗传算法在 OpenMP 下的并行化设计。

结论

计算机博弈是人工智能领域的重要研究方向，具有重大意义。而对中国象棋计算机博弈的研究始终是近三十年来海峡两岸乃至国际学术界研究的重点。

本文完成的主要工作如下：

（1）对博弈树搜索算法及应用现状进行总结，介绍并分析了各种基于 α-β 剪枝策略的搜索算法和各种优化措施，如 α-β 搜索/剪枝算法、PVS 算法、MTD(f) 算法、置换表、吃子启发、置换表启发、历史启发、杀手启发、空着裁剪等，以及 Tree Splitting、PVSplitting、DTS 三种并行策略，结合自己的理解，深入阐述了这些搜索算法和优化措施、并行思想的机理。

（2）对 OpenMP 并行编程模型进行了研究，介绍了 OpenMP 模型的机制、执行模式、编译指导指令、库函数与环境变量，以及多核并行算法的性能评价标准，总结了 OpenMP 编程需要注意的问题。

（3）在真实的中国象棋计算机博弈环境下，对多种博弈树搜索算法及其优化措施进行了比较。 根据中国象棋计算机博弈的特点，提出了一种融合了空着裁剪、置换表、历史启发、杀手启发、吃子启发、置换表启发等多种优化措施的混合 PVS 算法，设计了适合中国象棋计算机博弈的置换表、空着裁剪、启发策略的细节和各种优化措施的结合顺序，提高了算法的性能。并将这个混合 PVS 算法在 OpenMP 标准下进行了基于 PVSplitting 策略的并行化设计，合理对搜索任务进行分解，并通过为并行的每个线程单独设置置换表、着法历史得分表和杀手着法表，每个 CPU 处理单元查询时读取全部的置换表、着法历史得分表和杀手着法表，但更新时只写入当前 CPU 处理单元拥有的置换表、着法历史得分表和杀手着法表，以减少关键代码段的锁操作带来的影响，并设计了合理的线程调度策略，尽可能实现负载平衡。

（4）根据多核 PC 的特点，提出了一个基于面向对象方法的中国象棋计算机博弈系统的设计方案，并在 Visual C++ 2010 中予以实现，以“针对接口编程”为原则，对搜索引擎、估值函数、着法生成等模块进行了合理的设计，着重分析了中国象棋局面估值函数设计时需要考虑的各种要素，并使用自适应遗传算法对估值函数进行了优化。进一步，以这个真实的中国象棋计算机博弈程序为环境，对本文提出的基于 OpenMP 的并行混合 PVS 算法进行了实验，并对实验结果进行了分析，验证了算法的有效性、可行性和高效性。

本文的主要创新体现如下：

（1）将空着裁剪、置换表、历史启发、杀手启发、吃子启发、置换表启发

这些优化措施融入 PVS 算法，形成一种适用于中国象棋计算机博弈的混合 PVS 算法，并使用 PVSplitting 策略，在 OpenMP 模型下对混合 PVS 算法进行了并行化设计，合理设计了算法及其并行化的细节问题。

（2）使用面向对象方法开发出一个实际的中国象棋计算机博弈系统，将并行混合 PVS 算法应用于中国象棋计算机博弈系统，使用 OpenMP 下的并行自适应遗传算法优化了估值函数。

本文设计的基于 OpenMP 的混合 PVS 算法，以及基于这个算法的中国象棋计算机博弈系统具有如下特点：

（1）该算法易于实现，仅需普通的多核 PC 即可运行，对硬件和环境的要求较低。多核 PC 现已成为主流，随着多核多线程编程技术的广泛应用，该算法具有较大的应用前景。

（2）真实环境下的实验证明了该算法剪枝效率较高，并行模式在速度上明显优于串行模式，并且当出现多个最佳着法时，在 4 线程并行模式下有时能随机走出和串行模式下不同的着法，这是传统的串行算法无法做到的。

（3）该中国象棋计算机博弈系统具有一定的智能和棋力，采用面向对象开发思想，使得系统模块结构合理，模块内部内聚性高，模块和模块之间的耦合性低，系统易于维护和扩充。

但是，限于自身能力和精力，加之时间有限，本文设计的中国象棋计算机博弈系统的棋力目前只能与普通业余棋手抗衡，尚无法战胜高水平的专业棋手，和顶级中国象棋计算机博弈程序（如纵马奔流、象眼、棋天大圣等）比较也有一定的差距。对这方面的研究还有很多工作要做，进一步的工作如下：

（1）继续研究更多的适合中国象棋计算机博弈算法的优化措施。本文在研究博弈树搜索算法及其优化措施时，尚未涉及时间控制策略、开局库等增强手段，如果要设计一个具有更高智能的中国象棋计算机博弈系统，合理的时间控制策略和完备的开局库是必不可少的。以后的研究可考虑将时间控制策略、开局库等增强手段融入混合 PVS 算法，对现有算法做进一步优化。

（2）和 Unix/Linux 下的 PThread 多核编程模型相比，OpenMP 2.5 虽然对硬件和环境要求不高，编程相对简单，易于实现多核 PC 环境下的并行化，但不够灵活，这种基于 PVSplitting 并行策略的并行混合 PVS 算法，在 OpenMP 并行编程模型下出现了额外的同步开销，加速比和效率未能达到最大化。随着更高版本的 OpenMP 标准和更高版本的 Vusual C++开发工具的推出，将利用 OpenMP 的新特性继续研究混合 PVS 算法的并行化，或者使用其他的并行策略对混合 PVS 算法进行并行化设计，以期获得更好的加速比和效率。

（3）本文设计的 OpenMP 下的并行 PVS 算法，采用“以空间换时间”的思想，每个 CPU 处理单元拥有独立的置换表、着法历史得分表和杀手着法表，置

换表所占的内存空间较大，对于目前普通的双核、三核、四核 PC 和 2G、4G 的内存环境尚能承受，但在拥有更多处理单元的多核 CPU，需要更多的置换表、着法历史得分表和杀手着法表，而置换表对内存需求较大，如果内存资源不增加，置换表数量的增加可能导致内存资源耗尽。但如果让所有的 CPU 处理单元共享一个置换表，又会增加关键代码段从而影响加速比和并行效率。如何解决这个问题也有必要进行更深入的研究。

（4）对棋子间的战术与配合，如“连环马”、“霸王车”、“马后炮”等特殊配合，以及“捉双”等战术，都能给对方造成巨大威胁，或者缓解己方所处的不利局面。对弈出现这些局面时，估值函数应当给予己方相应的附加值，但本文设计的中国象棋局面估值函数，尚未考虑为棋子间的战术与配合设置附加值的问题。本文设计的 OpenMP 下优化估值函数的并行自适应遗传算法，也仅针对棋子的基本价值做了实验，对棋子灵活度、棋子位置、棋子受威胁情况、棋子受保护情况等方面未能进行更详细的实验研究，这些都导致了估值函数存在不足之处，有待进一步完善。

参考文献

[1] Luger G F. Artificial intelligence: Structures and strategies for complex problem solving. 6th Edition. New Jersey: Pearson Addison-Wesley Press, 2009, 1-22

[2] Russell S, Norvig P. Artificial Intelligence: A Modern Approach. 2rd Edition. New Jersey: Prentice Hall, 2006, 14-22

[3] 徐心和, 王骄. 中国象棋计算机博弈关键技术分析. 小型微型计算机系统, 2006,27(6): 61-69

[4] 危春波. 中国象棋博弈系统的研究与实现: [昆明理工大学硕士学位论文]. 昆明:昆明理工大学, 2008, 1-2

[5] 林建伟. 博弈树搜索技术在牌类网络游戏中的应用: [汕头大学硕士学位论文]. 汕头:汕头大学, 2009, 1-2

[6] 岳金朋, 冯速. 博弈树搜索算法概述. 计算机系统应用, 2009,9 : 203-207

[7] 高强. 一种混合博弈树算法在中国象棋人机博弈中的应用研究: [大连交通大学硕士学位论文]. 大连:大连交通大学, 2006, 1-29

[8] 陈国良, 孙广中, 徐云, 等. 并行算法研究方法学. 计算机学报, 2008,31(9): 1493-1502

[9] 郭秀丽. 中国象棋计算机博弈中搜索算法的研究与改进: [河北大学硕士学位论文]. 保定:河北大学, 2010, 2-4

[10] 王晓鹏, 王骄, 徐心和, 等. 中国象棋与国际象棋比较分析. 重庆工学院学报, 2007,21(1) : 71-76

[11] 谭膺. 基于 SMP 的并行游戏树搜索程序负载分析研究: [中国科技大学硕士学位论文]. 合肥:中国科技大学, 2006, 5-14

[12] 王小春. PC 游戏编程——人机博弈. 重庆:重庆大学出版社, 2002, 102-158

[13] 涂志坚. 电脑象棋的设计与实现: [中山大学硕士学位论文]. 中山:中山大学, 2004, 3-30

[14] 黄晨. 解剖大象的眼睛——“象眼”博弈程序分析. 象棋百科全书网. http://www.xqbase.com/resource.htm, 2007-10-24

[15] 李小舟. 基于改进博弈树的黑白棋设计与实现: [华南理工大学硕士学位论文]. 广州:华南理工大学, 2010, 2-3

[16] Quinn M J. Parallel programming in C with MPI and OpenMP. New York: McGraw-Hill Companies, 2003, 316-318

[17] Hyratt R. A high-performance parallel algorithm to search depth-first game

trees: [dissertation]. Birmingham: University of Alabama at Birmingham, 1988, 45-49

[18] 李之棠, 陈华民. 博弈树并行搜索算法. 小型微型计算机系统, 1998,19(10) : 53-56

[19] 王京辉, 乔卫民. 基于 PVM 的博弈树的网络并行搜索. 计算机工程, 2005,31(9) : 29-30

[20] OpenMP 官方网站. http://openmp.org/wp/, 2011-4-15

[21] 张聪品, 刘春红, 徐久成. 博弈树启发式搜索的 α-β 剪枝技术研究. 计算机工程与应用, 2008,44(16) : 54-55

[22] Knuth D E, Moore R W. An analysis of alpha-beta pruning. Artificial Intelligence, 1975,6(4) : 293-326

[23] Fishburn J, Finkel R. Parallel alpha-beta search on Arachne. In: Proc of International Conference on Parallel Processing. Madison, 1980, 235-243

[24] 白圣秋. DTS 演算法效能改良之研究: [国立台湾师范大学资讯工程研究所硕士论文]. 台北:国立台湾师范大学, 2007, 4-12

[25] Zobrist A. A new hashing method with application for game playing. Technical Report 88, The University of Wisconsin. Madison, 1970,13(2) : 69-73.

[26] Kishimoto A, Schaeffer J. Transposition Table Driven Work Scheduling in Distributed Game-Tree Search. In: Proc of 15th Canadian Conference on Artificial Intelligence. Calgary, 2002, 56-68

[27] 焦尚彬,刘丁. 博弈树置换表启发式算法研究. 计算机工程与应用, 2010,46(6) : 42-45

[28] 高强. 哈希技术在中国象棋机器博弈系统中的应用研究. 科学技术与工程, 2008,8(17): 4869-4871

[29] 万翼. 计算机国际象棋博弈系统的研究与实现: [西南交通大学硕士学位论文]. 成都:西南交通大学, 2006, 31-33

[30] 蒋鹏, 雷贻, 陈园园. C/C++中国象棋程序入门与提高. 北京:电子工业出版社, 2009, 214-306

[31] Schaeffer J. The history heuristic. Journal of the International Computer Chess Association, 1983,6(3):16-19

[32] Schaeffer J. The history heuristic and alpha-beta search enhancements in practice. IEEE Transactions on Pattern Analysis and Machine Intelligence, 1989,11(11) : 1203-1212

[33] Selim A, Monroe N. The principal continuation and the killer heuristic. In: Proc of ACM National Conference. Seattle, 1997, 466-478

[34] 岳金朋, 冯速. 中国象棋 Alpha-Beta 搜索算法的研究与改进. 北京师范大学学报（自然科学版）, 2009,45(2) : 156-160

[35] Reinefeld A. An improvement of the Scout tree-search algorithm. Journal of the International Computer Chess Association, 1983,6(4) : 4-14.

[36] Donninger C. Null move and deep search : Selective search heuristics for obtuse chess programs. ICCA Journal, 1993,16(3) : 137-143

[37] Heinz E A. Adaptive null-move pruning. ICCA Journal, 1999,22(3):123-132

[38] Plaat A. Research Re: Search & Re-search : [dissertation]. Rotterdam: Erasmus University, 1996, 255-293.

[39] 邹竞. 基于 MTD(f)的中国象棋人机博弈算法的设计与优化. 计算机与数字工程, 2008,36(9) : 38-43

[40] 张明亮, 吴俊, 李凡长. 极小树叶结点数定理的补充证明及有关分析. 模式识别与人工智能, 2011,12(4) : 521-526

[41] Marsland T A, Campbel M S. Parallel search of strongly ordered game trees. ACM Computing Surveys, 1982,14(4) : 533-551

[42] 周伟民. 多核计算与程序设计. 武汉:华中科技大学出版社, 2009, 7-121

[43] Intel 软件学院教材编写组. 多核多线程技术. 上海:上海交通大学出版社, 2011, 97-134

[44] 黄少龙. 象棋开局战理. 北京:人民体育出版社, 1983, 58-94

[45] 谢燕茹. 中国象棋计算机博弈数据结构与评估函数的研究和实现: [西安理工大学硕士学位论文]. 西安:西安理工大学, 2008, 7-20

[46] 王小平, 曹立明. 遗传算法——理论、应用与软件实现. 西安:西安交通大学出版社, 2006, 1-50

[47] 王骄, 王涛, 罗艳红, 等. 中国象棋计算机博弈系统评估函数的自适应遗传算法实现. 东北大学学报, 2005,26(10) : 949-952

[48] 唐天兵, 谢祥宏, 申文杰, 等. 多核 CPU 环境下的并行遗传算法的研究. 广西大学学报（自然科学版）, 2009,34(4) : 546-550

附录 A 攻读学位期间发表的学术论文

[1] 邹竞. 基于 MTD(f)的中国象棋人机博弈算法的设计与优化. 计算机与数字工程, 2008,36(9):38-43, 独作

[2] 邹竞. 基于 FP_growth 算法的课程相关性的关联规则研究. 计算机与数字工程, 2009,37(6):39-43, 独作

[3] 邹竞, 谢鲲. C4.5 算法在移动通信行业客户流失分析中的应用.计算技术与自动化, 2009,28(3):98-101, 第一作者

[4] 邹竞, 谢鲲. 基于 OpenMP 的并行混合 PVS 算法及在中国象棋中的应用. 已在“小型微型计算机系统”期刊投稿, 第一作者

附录 B 攻读学位期间参与的科研项目

[1] 多跳无线 mesh 网络中协作通信关键技术研究，国家自然科学基金(61003305)，2011.1-2013.12，主研人员

[2] 基于协作通信的无线 mesh 网络关键技术研究．博士点基金(20100161120022)，2011.1-2013.12，主研人员

[3] 无线 mesh 网络中分簇协作路由关键技术研究．湖南省自然科学基金(11JJA003)，2011.1-2013.12，主研人员

致谢

本文即将完成之际，首先谨向我的导师谢鲲博士、副教授致以衷心的谢意。我能力有限，身体也一直欠佳，在论文的选题、研究、编程、撰写过程中，总是磕磕碰碰，在学习、生活、工作中也遇到了诸多不顺。但谢博士不嫌我资质愚钝，也没有指责我做事拖拉和效率低下，始终一如既往给我关心、鼓励和帮助，让我重拾信心，才使我在博弈树搜索和 OpenMP 并行编程中小有所获。初稿完成时，谢博士更是从百忙之中抽出时间进行批阅，提出了中肯的修改意见。谢博士精湛的学术造诣、睿智的学术智慧、实事求是的工作作风、细致的工作耐心给我留下了极深刻的印象。

其次，感谢我的父母和妻子，在我编程和撰写论文的时期内，他们承担了大多数的家务来支持我，才使本来就不擅长阅读和撰写论文的我有了更多的时间进行学习和研究。我的父母和妻子太辛苦了，令我既感且愧，家人也是我的动力泉源之一。

再次，感谢我的同事蔡美玲博士和我素未蒙面的师弟李秦古，他们帮我在互联网下载了不少有一定参考价值的文献，使我在计算机博弈领域学习到了前人的研究成果。感谢我的同事曾望老师，在英文文献的阅读和翻译上给了我一定的帮助。感谢我昔日的学生、现在的朋友李琴瑜，这个善良的女孩在我去年遇到困难的时候十分耐心的安慰和激励我，才使我有了坚持到底的决心。我的同事王艳瑞老师、唐思斯老师和王凌风老师也在论文的格式细节上给了我一些合理建议，在此一并致谢。

最后，感谢各位专家和老师在百忙之中对本文进行评审并提出宝贵的修改意见。

总之，我以最真挚的感恩之心，真诚感谢激励和帮助过我的每一个人，谢谢你们！