学校代码：10425
学　　号：G0307060

中国石油大学工程硕士专业学位论文　华东

（申请工程硕士学位）

基于搜索算法的人工智能在五子棋博弈中的应用研究

工程领域：　计算机技术
培养方向：　人工智能及应用
硕 士 生：　王志水
指导教师：　刘新平(副教授)
　　孙士明（讲师）

入学日期：2004 年 3 月　　论文完成日期：2006 年 10 月

基于搜索算法的人工智能在五子棋博弈中的应用研究

王志水（计算机技术）

指导教师：刘新平(副教授) 孙士明（讲师）

摘 要

本文以计算机五子棋博弈系统作为研究课题，在对大量的相关文献进行分析研究的基础上，按照人工智能和计算机博弈的一般原理设计了一个五子棋博弈系统的基本模型，并作了简单的实现和验证，所做的工作包括三个方面：

第一 研究了五子棋在计算机中的表示问题，讨论了计算机中存贮棋局和识别下棋次序，局势状态变化及局势特征、走法产生等方法。

第二 研究了博弈树的极小极大搜索技术及在此基础上的 Alpha-Beta 剪枝过程和剪枝优化问题。实现将候选的后继节点按位置邻近顺序排序，使剪枝过程得到优化。

第三 根据五子棋的特点，提取棋局局势的若干特征，对这些特征赋加权分，并对整个棋局进行特征统计，采用线性函数求得棋局的总估计分值，从而提高了五子棋程序对弈的水平和能力。

在上述工作的基础上，本文的创新性研究主要包括以下两个方面：

第一 对五子棋博弈的专业知识进行了认真的整理，针对五子棋博弈规则简单、局势判断清楚的特点，对五子棋常见的开局、定式及其后的对局做了细致的统计分析，阐明了五子棋对弈中黑白双方优劣势并非均衡的规律，这一规律作为一个指导原则在设计五子棋博弈系统时起到重要作用。

第二 通过线性函数获得的总估计值，实现了算法的优化和加强，主要有以下两个方面：在执行常规的 Alpha-Beta 搜索和 NegaScout 算法之前，

使用置换表搜索避免相同局面的节点的搜索。通过使用威胁空间搜索，让计算机对某一局面找出全部的致胜威胁次序，避免了对静态估值函数的调用，只有计算机对当前局面无法找到致胜威胁次序时，才执行对静态估值函数的调用，由于五子棋博弈中，黑方先行者占有很强的优势，大多数局面往往都能找到致胜威胁次序，所以采用威胁空间搜索，结果表明，可以极大的提高程序的表现和对弈水平。

关键词：搜索算法，人工智能，五子棋，博弈

Application and Research Based on Searching Algorithm of AI in Five-Piece Game-Playing

Wang Zhishui(Computer Technology)
Directed by Associate Professor Liu Xinping,Instructor Sun Shiming

Abstract

This thesis studies the computer Five-Piece game-playing systems. According to the artificial intelligence and general theory of computer game-playing, a basic models of the Five-Piece game-playing systems is designed. Three aspects were done in the work:

Firstly, the question of how to express the Five-Piece in computer was studied and how to store the board position in computer as well as distinguish the sequence in playing and the change \character of situation were discussed.

Secondly, the Minimax Searching technology of Game Tree was investigated. The further research of Alpha-Beta Procedure and optimization problem of which based on it were being done. The candidate sequence nodes were sorted according to there location in order to optimize the shearing process.

Thirdly, according to the character of Five-Piece, some characters of the situation were being extracted and being endowed with coefficient. By studying the global attributes of whole board position, the total evaluation value was gotten through a linear function.

On the foundation of the above mentioned work, the creative research mainly includes following two aspectses:

At first ,the professional knowledge of Five-Piece game-playing is sorted carefully. The familiar game beginning, fixed position and the following game

play were analyzed carefully for the simpleness rule, the clarity situation estimation of the Five-Piece game-playing. A law which is not balance in superiority\inferiority position between two parts, playing an important effect as a guidance in designing the Five-Piece game-playing system, is clarified.

Secondly, aiming at overcoming the weakness and shortage in the first version procedure, the algorithm was enhanced and optimized. The first version procedure lose ours satisfaction for its slow searching speed and bad game exhibition. Two main reasons contribute to the situation. One is the application of routine Alpha- Beta search and NegaScout algorithm which can not avoid repeating search to the same position nodes. The other one is the adoption of fixed estimate method, which will lead to the lower intelligence for its unnicety and can not to improve the game power during the game procession.

There still two methods to solving the problem. One is searching the displacement form to avoid repeating search to the same position nodes before routine Alpha- Beta search and NegaScout algorithm were performed. The other is applying threaten space search which can let the computer to find out all the threaten sequence to win the game. This can be aviod tranfer the static function. Only can not the computer find the threaten sequence to current position, the static function is tranfered. Because the black part has a mighty dominance in the game, the threaten sequence can be often found using threaten space search. The results indicate that the search can increase the program exhibition and game level greatly.

Keyword: search algorithm , artificial intelligence, Five-Piece game,Game,

独 创 性 声 明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得中国石油大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

签 名：王玉水 2007年 4月 5日

关于论文使用授权的说明

本人完全了解中国石油大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件及电子版，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

（保密论文在解密后应遵守此规定）

学生签名：王玉水 2007年 4月 5日

导师签名：刘振平 2007年 4月 5日

第1章 绪论

1.1 课题来源

人工智能是综合性很强的一门边缘学科，它的中心任务是研究如何使计算机去做那些过去只能靠人的智力才能做的工作。目前，各国都把人工智能作为重点列入本国的高科技发展计划，投入巨大的人力和物力。作为一门边缘学科，它有诸多的研究领域：专家系统、决策支持系统、机器学习、机器视觉、自然语言理解等等，作为人工智能研究的一个重要分支，计算机博弈是检验人工智能发展水平的一个重要方面，它广泛应用于政治、经济、军事和生物竞争等领域。它的研究成果为人工智能带来了很多重要的方法和理论，产生了广泛的社会影响和学术影响，也广泛应用于军事指挥和经济决策中。

现在，人类已经不必为信息的匮乏而担忧了，相反，每天接触到的大量信息倒让人有些不知所措。可以预计，在不久的将来，具有人工智能的计算机将按照我们的意愿处理这些信息，从而提高人类生活的质量和工作效率。现在，人工智能已不仅仅是让计算机像人一样的听、说、读、感知外部世界并与人类交流，将来的发展方向是要计算机像人一样的思考、推理、预测，也就是说处理信息。

人工智能中大多以下棋（如：象棋、围棋、五子棋、西洋跳棋等）为例研究博弈规律。博弈成为了一个典型的问题。早期，人工智能的创始人之一 A. L. Samuel 编了一个计算机下西洋跳棋的程序。1959 年，该程序战胜了设计者本人，1962 年则击败了美国的一个州冠军。现在，国内中山大学的陈志行教授的“手谈”围棋程序已有一定“智力”。Tesauro 的

TD-GAMMON（1995 年）西洋双陆棋程序经过上百万盘的学习训练，程序达到世界水平。这些程序或者需要经过大量训练，或者“智力”有限，或者是采用大规模搜索算法实现，难以避免“组合爆炸”的危机，因此，一个真正“智能” 的，有学习能力的高效率的博弈策略还有待进一步研究。

本文以计算机五子棋博弈系统作为研究课题，在讨论了人工智能中用于计算机博弈的一般技术之后，重点对五子棋对弈的内在规律和专业知识进行分析研究，给出了解决五子棋博弈的重要搜索算法：置换表搜索算法、威胁空间搜索算法，并对算法作了一定的改进。

1.2 博弈简史

博弈思想最早产生于古代的军事活动和游戏活动。在体育游戏中，经常会出现这种情况，即甲乙双方各出三个人进行摔跤比赛。甲乙双方的领头人不是让自己的队员随意地同对方某一队员较量，而是先了解清楚对方三名成员的实力，并把对方三名成员的实力同己方成员的实力作客观对比，然后作出决定：谁打头阵，谁在中间，谁压轴，以自己的弱者去对付对方的最强者，以自己的最强者对付对方的次强者，以自己的次强者对付对方的最弱者，保证二比一稳赢对方。

计算机博弈，简单的说，就是让计算机像人一样从事高度智能的博弈活动。研究者们从事研究的计算机博弈项目主要有国际象棋，围棋，中国象棋，五子棋，西洋跳棋，桥牌，麻将，Othello，Hearts，Backgammon，Sarabble等[1-4]。其中，二人零和完备信息博弈的技术性和复杂性较强，是人们研究博弈的集中点。二人零和随机性研究的一个代表是 Backgammon，并且产生了很大的影响。桥牌是研究不完备信息下的推理的好方法。高随机性的博弈项目趣味性很强，常用于娱乐和赌博，是研究对策论和决策的好例子。

近代计算机博弈的研究是从四十年代后期开始的，国际象棋是影响最大、研究时间最长、投入研究精力最多的博弈项目，成为计算机博弈发展的主线。

1950 年 C.Shannon 发表了两篇有关计算机博弈的奠基性文章（Programming A Computer for Playing Chess 和 A Chess-playing Mechine）。1951 年 A.Turing 完成了一个叫做 Turochamp 的国际象棋程序，但这个程序还不能在已有的计算机上运行。1956 年 Los Alamos 实验室的研究小组研制了一个真正能够在 MANIAC-I 机器上运行的程序（不过这个程序对棋盘、棋子、规则都进行了简化）。1957 年 Bernstein 利用深度优先搜索策略，每层选七种走法展开对局树，搜索四层，他的程序在 IBM704 机器上操作，能在标准棋盘上下出合理的着法，是第一个完整的计算机国际象棋程序[2]。

1958.年，人工智能界的代表人物 H.A.Simon 预言[3]："计算机将在十年内赢得国际象棋比赛的世界冠军。"当然，这个预言过分乐观了。

1967 年 MIT 的 Greenblatt 等人在 PDP-6 机器上，利用软件开发工具开发的 MacHack VI 程序，参加麻省国际象棋锦标赛，创出了计算机正式击败人类选手的记录[2]。

从 1970 年起，ACM(Association for Computer Machinery)开始举办每年一度的全美计算机国际象棋大赛。从 1974 年起，三年一度的世界计算机国际象棋大赛开始举办。

1997 年，IBM 公司的超级计算机"深蓝"战胜了国际象棋世界冠军卡斯帕罗夫，成为人工智能领域的一个里程碑。

在其它的博弈项目上，1959 年 Samuel 等人利用对策理论和启发式搜索技术编制了一个西洋跳棋程序，这个程序具有学习能力，它可以在不断的对弈中改善自己的棋艺。4 年后，这个程序战胜了设计者本人。又过了 3

年，这个程序战胜了美国一个保持 8 年之久的长胜不败的冠军。这个程序向人们展示了机器学习的能力，提出了许多令人深思的社会问题和哲学问题[5]。1979 年 H. Berliner 的程序 BKG9.8 以 7 比 1 战胜了 Backgammon 游戏的世界冠军 Luigi Villa，1980 年美国西北大学 Mike Reeve 的程序 The MOOR 战胜了 Othello 世界冠军[1]。

1989 年，第一届计算机奥林匹克大赛在英国伦敦正式揭幕，计算机博弈在世界上的影响日益广泛。

1.3 研究计算机博弈的意义

人们常常把计算机博弈描述为人工智能的果蝇，即，人类在计算机博弈的研究中衍生了大量的研究成果，这些成果对更广泛的领域产生了重要影响。通过博弈问题来研究人工智能典型问题，具有以下的优点[4]：

1, 博弈问题局限在一个小的有典型意义的范围内，研究容易深入。

2, 博弈问题非常集中的体现了人类的智能，已足以为现实世界提供新的方法和新的模型。

3, 专家经验容易获取。

4, 进展可以精确的展示和表现出来，不同方法和模型的优缺点也容易比较。

用一棵树来表示棋局发展的种种可能性，这种树叫做博弈树（对局树）。根节点表示对局的开始状态，每一种可能的走法造成的结果作为其子节点，而对每一个这样的子节点，考虑另一方的各种可能应对，作为下一层的子节点，这样一直找下去就得到了对局树。

这是一个典型的指数复杂性问题，如何在这棵树上有效的搜索，找出最佳或满意的目标，是研究的主要问题。

由于机器速度和存储空间的限制，这种搜索只能进行到某一深度，得到一个 n 层的子树。对于这一子树的叶节点进行某种评价，然后用搜索的方法找出最优解或满意解，这就是计算机博弈的主要方法。

在博弈研究的早期阶段，人们就是使用这种方法来研究博弈问题的，所以主要的研究内容是[15][16]：

1，评价的效率更高，评价要花费时间和空间的代价，如何建立有效、快速的评价函数和评价方法。

2，找最佳解（在生成的子树上）的过程更为有效，由此发展了各种搜索法。

在五、六十年代，博弈研究是人工智能研究的带头领域，Alpha-Beta 剪枝、启发搜索最初就是由博弈树的研究发展而来的。

专家系统的方法出现以后，研究者在产生树的过程中引入专家知识，使得不必要的节点不再产生，并以此来改进程序的效率。

博弈研究的另一个成果是在六十年代就引入了机器学习，使机器自己不断改进自己的博弈水平：

1，最简单的学习是改进评价函数的一些参数，使之更加精确和实用。

2，记忆自己和别人的错误，不再犯已犯过的错误。

3，记录一些常见的形状和模式，以便找出局部最好的应对。

4，自己学习规则。

这些研究对于人工智能的发展起了很大的推动作用。

不完备信息博弈的研究还推动了不精确信息、模糊信息推理和复杂环境决策的研究过程。由于博弈不能避免搜索，而搜索对时间的要求较高，所以，博弈一直是研究并行算法、并行体系结构的工具。

人们通过博弈的研究还提出了认知心理学上的很多新课题，对认知心理学产生了影响。有些博弈项目为一些新模型与新方法的研究提供了帮助。

例如，Backgammon 就对神经网络的发展做出了贡献。

计算机博弈中指定决策和选择决策，与政治、军事、经济、日常生活中的决策有很多类似之处，是一种典型的决策系统，所以它的研究对于建立现实社会的决策支持系统有很强的参考价值，也因此，在国外计算机博弈研究项目经常得到军方和政府机构的支持和赞助。1990 年，美军利用超级计算机对“沙漠风暴”行动进行战略模拟，博弈也成为这场有史以来最成功战争中的高技术明星。

最后，在计算机日益普及和大众化的现代社会，高水平的博弈系统很容易获得可观的商业价值。目前，世界领先的计算机围棋程序基本上都是商业产品。事实上，个人计算机软件市场的大约 80%销售额是来自游戏软件，其中有传统的博弈游戏，而非博弈游戏中也不可缺少人工智能与博弈的成分。

1.4 计算机博弈过程

计算机五子棋对弈是一种完备信息博弈（Games of Perfect Information），意思是指参与双方在任何时候都完全清楚每一个棋子是否存在，位于何处。只要看看棋盘，就一清二楚。象棋、围棋等都属于完全知识博弈。要想实现人和计算机双方对弈，不妨假设人是甲方，计算机是乙方，人和计算机对弈的过程可以如下表述：假设首先由甲方走棋，他面对的是一个开始局面 1，从这局面可以有 M 种符合规则的下法。这 M 种下法分别形成了局面 2，3，……M+1。如图 1-1 所示。

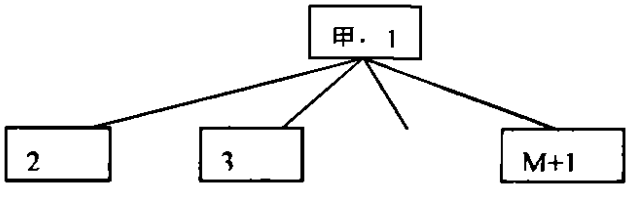

图 1-1 甲方面对的局势

假设甲选择了形成局面 2 的下法，轮到乙下棋。乙面对局面 2，又可以有 N 种可能的下法，形成 N 种新的局面 k+1,k+2, ……k+N,如图 1-2 所示。

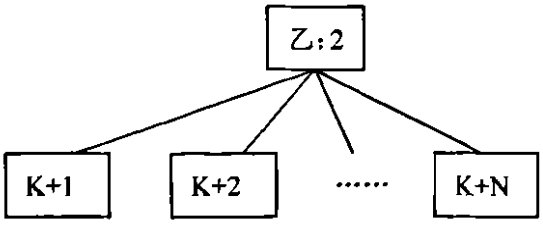

图 1-2 乙方面对的局势

如果甲选择形成局面 3，4，……N+1 下法，乙方都对应有若干种下法。这样甲乙双方轮流下棋，棋盘局面发展变化就形成如图 1-3 所示的一棵树形状，通常称为博弈树。

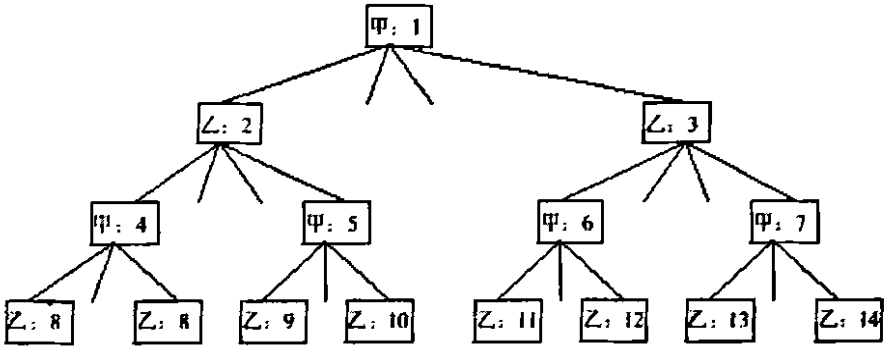

图 1-3 博弈树的例子

博弈树最终的叶节点有甲赢乙输，甲输乙赢，甲乙平手三种。下棋者总是从当前局面出发选择最有利于自己的走法下一子，如甲在局面 1，他将从乙 2、乙 3 等局面中选择最有利自己的走法；同样，乙在 2 局面时也从甲 4、甲 5 等局面中选择最有利自己的走法。为了从很多的局面中选出最有的，就需要一个搜索算法和一个对局面进行形势判断的函数。搜索算法通常使用极大极小值算法、Alpha-Beta 剪枝技术，对形势的好坏，用估值函数[6]进行判断，这些将在论文中介绍。

1.5 计算机五子棋基本知识介绍

本论文中五子棋的基本规则如下[14]：

(1)棋盘：采用国际上标准的15×15路线的正方形棋盘。

(2) 下法：两人分别执黑白两色棋子，轮流在棋盘上选择一个无子的交叉点走子，无子的交叉点又称为空点，规定由黑方先行走棋。

(3) 输赢判断：黑、白双方有一方的五个棋子在衡、竖或斜的方向上联接成无间断的一条线即为该方赢。

注：以上规则即为通常的业余的无限制下棋法，而在职业的五子棋比赛中，对于黑方有“三三”禁手，“四三”禁手，禁止“长连”等限制。

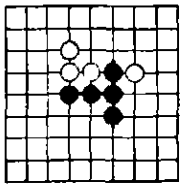

图1-4 黑子三三的禁手

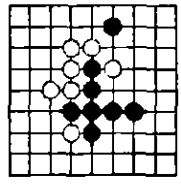

图1-5 黑子四四的禁手

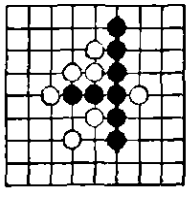

图1-6 黑子长连的禁手

1.6 本文解决的问题

本课题的主要工作是将计算机博弈理论应用在五子棋程序的设计与实现上，建立计算机下五子棋的总体模型，利用基本的极大极小搜索算法、Alpha-Beta剪枝、渴望搜索算法结合估值函数可以实现一个一般的五子棋对

程序设计，但是，它的水平与能力不能令人满意，本文在对五子棋对的内在规律和专业知识深入研究的基础上，给出了适合该棋类对的更佳的搜索算法：基于哈希表的置换表搜索算法、威胁空间（Threat Space Search）搜索算法。结果表明，以上算法在五子棋的程序设计中是成功的。

第2章 状态空间表示与走法产生

要让计算机学会下棋，首先就要把下棋问题表示成计算机可理解的形式，即把五子棋问题形式化，存储在计算机中，并能让搜索、估值等算法对这些数据进行操作。需要在计算机中表示的主要问题有棋盘局势状态、落子的顺序、局势状态的变化及局势特征的表示等。

2.1 棋盘局势状态表示

棋盘表示主要探讨的是使用什么数据结构来表示棋盘上的信息。一般说来，这与具体的棋类知识密切相关。通常，用来描述棋盘及其上棋子信息的是一个二维数组。

要让计算机知道棋盘局势状态，就是要它记住棋盘中哪个位置有黑子，哪个位置有白子以及哪个位置是空点。对于五子棋程序而言，因为棋盘是15行，15列，因此可以将棋盘状态的描述用一个15×15的二维数组表示。

本程序的数据将基于如下所示的数据表示：

```
BYTE m_RenjuBoard[Grid_num][Grid_num]=
  {
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
```

```
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
  }
```

其中，（1）棋盘状态数据由一个15×15的二维数组表示。

（2）用数字“0”和“1”来表示不同的棋子，黑色棋子用“0”表示，白色棋子用“1”表示。

（3）没有棋子的格子用0xFF表示。

在程序中定义了一个结构用来表示棋子的位置：

```
typedef   Struct _stoneposition
{
    BYTE   x;
    BYTE   y;
}STONEPOS;
```

另外，定义一个结构用以描述棋子的走法：

```
typedef struct   _stonemove
{
        STONEPOS     StonePos;           //棋子的位置
        int          Score;              //走法的得分
}STONEMOVE;
```

2.2 棋盘中下棋的顺序表示

棋盘中哪个子先下，哪个后下也需要表示出来。在程序中用堆栈的结构存储棋子及坐标，即先下的子放在栈低端，后下的放在高端，保持其顺序。通过访问栈，可以得到下棋的顺序。

例如：如图2-1，表示这样四颗棋子分别下在棋盘上：

1－黑子下在（10，10）位置；

2－白子下在（10，11）位置；

3－黑子下在（9，　11）位置；

4－白子下在（11，10）位置。

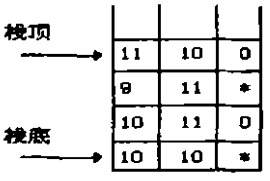

图 2－1 棋子存储栈结构

2.3 状态的变化

下棋的双方每下一子，棋盘的状态都会发生变化。由于搜索过程中有回溯发生，因此状态更新后，还要能恢复成原来的。由于程序中是采用堆栈的结构，用入栈操作就可将一方下的棋子存入栈中，得到新的棋盘状态，当搜索过程的回溯发生时用出栈操作退出一子，即可恢复到前一状态。

2.4 特征的表示

在对棋盘的局势进行判断时，估值函数必须提取棋盘状态的特征以便对局势进行判断。程序中采用字符串结构表示特征。如一个五子相连的特征可以这样表示：

```
char line[10];
```

line＝“00000”;

在某一状态的棋盘中采用模式匹配的方法，寻找是否存在一个特征。

最后，为了在使用的时候能够避免数据表示出差错，我们将上述数据描述定义成一系列便于使用的宏。定义在一个头文件里（define.h），下面是 define.h 的源代码。

```
//define.h //data structure and macros
#ifndef define_h_
#define define_h_
#define Grid_Num 19   //
#define Grid_Count 361 //
#define BLACK 0
#define WHITE 1
#define NOSTONE   0xFF
//这个宏用以检查某一坐标是否是棋盘上的有效落子点
#define   IsValidPos(x,y)   ((x>=0   &&x<Grid_Num)   &&(y>=0   &&
y<Grid_Num))
//用以表示棋子位置的结构
typedef Struct _stoneposition
{
        BYTE   x;
        BYTE   y;
}STONEPOS;
//用以表示走法的结构
typedef struct _stonemove
{
```

```
    STONEPOS    StonePos;        //棋子的位置
    int         Score;           //走法的得分
}STONEMOVE;
#endif    //define_h_
```

2.5 走法产生

走法产生是指将一个局面的所有可能的走法罗列出来的那一部分程序。也就是用来告诉其它部分下一步可以往哪里走的模块。各种棋类随规则的不同，走法产生的复杂程度也有很大的区别。一般说来，在一种棋类游戏中，双方棋子的种类越多，各种棋子走法的规则越多，则在程序中，走法产生的实现就越复杂。以中国象棋为例，比如“象”只可以走田子，就检查与这个“象”相关联的“象”位上是否有自己的棋子，并且要检查其间的“象”眼上是否有棋子；而“兵”则要注意是不可后退并以此只能前进一步。现在假定由一个轮到红方走棋的局面，要列举出红方所有合乎规则的走法。

在五子棋的对弈程序中，由于双方只有黑白各一个子，并且在走子的过程中，没有吃子、提子等规则的存在，双方轮流走子，只要有一方首先在棋盘的水平、垂直、斜线方向上使己方的五个子连成一条直线，就获得胜利。因此，对于五子棋的走法产生来说，棋盘上的任意空白点都是合法的下一步。这样在五子棋的走法产生模块里，只要扫描棋盘，寻找到所有的空白，就可以罗列出所有符合规则的下一步。

因为五子棋的走法产生非常容易实现，下面给出在走法产生器类中寻找所有合法走法的函数实现：

```
//用以产生当前局面 positon 中所有合法的走法
```

```
//position 是包含所有棋子位置信息的二维数组
//nPly 指明当前搜索的层数，每层将走法存在不同的位置，以免覆盖
//nSide 指明产生哪一方的走法，WHITE 为白方，BLACK 为黑方
 int CreatePossibleMove(BYTE position[][Grid_Num],int nPly,int nSide)
{
  int i,j;
  m_nMoveCount=0;  //此变量记录所有合法的走法总数
  for(i=0;i<Grid_num;i++)
   for(j=0;j<Grid_num;j++)
    {
     if(position[i][j]==(BYTE)NOSYONE  //当前位置是空白
     {
        addMove(j,i,nPly);  //将当前走法添加到存放走法的数组中
     }
  //使用历史启发搜索中的静态归并函数对走法队列进行排序
  //目的是为了提高剪枝效率
 MergeSort(m_Move[nPly],m_nMoveCount,0);
Return m_nMoveCount;    //返回和法走法个数
}
```

2.6 小结

本章阐述了五子棋在计算机中的表示问题、计算机中存贮棋局和识别下棋次序，局势状态变化及局势特征、走法产生等方法。

第 3 章　博弈树的搜索技术

计算机要选择有利于它的最佳下法，就要能判断哪种形势对它最有利。但往往对一个形势的判断是很难做到准确的，特别是一盘棋刚开始的时候，棋盘的形势不明朗，即使是专家也不能做出准确的判断。为了判断哪种下法最有利，我们往往需要向后面计算几步，看看在走了几步棋之后，局面的形势如何。这被称为“多算胜”，也就是说，谁看得越深远，谁就可以获胜。这种思维方式被用到了计算机上。前面我们说过，棋局的不断向后计算，形成一棵博弈树，“多算胜”的思想即是对博弈树的搜索过程，这就是博弈树的极小极大搜索算法。

3.1 极小极大树搜索算法

讨论过程中，还是假设有两个博弈者甲方和乙方。可以认为甲方即为计算机，乙方为对手。我们的任务是为甲方找到最佳的移动。在图 3－1 形成的博弈树中，假设甲方先下，然后两个博弈者轮流下棋。因此，深度为奇数的节点，对应于甲方下了一子后的局面，即轮到乙方选择下法的局面，称为极小节点；深度为偶数的节点，对应于乙方下一子后的局面，即轮到甲方选择下法的局面，称为极大节点；博弈树的顶节点深度为 0。 K 层包含深度为 2K－1 和 2K 的节点（K>0），通常用层数给出博弈树的搜索程度，它表示向后预测的甲乙两方交替下棋的回合数。

如果想穷尽博弈树的一切可能性，那是不可行的。在本课题采用的棋盘中，共有 225 个点，也就是说在空棋盘上下第一个子时，有 225 种可能的下法，当然这时候在除边线上的点外的大部分点上效果是等同的。即使是取已经下过棋子的点的周围的，可能与已有点形成五子连的点做为后继着

法，一个局面的后继节点也至少有 30 个以上。再假设博弈双方各下了 10 子，共 20 子，那么局面的可能性就有 $30^{20} \approx 10^{29}$ 个。这只是最后一深度的节点数，总的局面数就更多了。这样的数量，不要说存储器难以有足够的数量存储，就是时间也不是现实可行的。因此，对博弈树的搜索到一定深度就不再往下走了， 根据一个评估值函数 f ，对局势进行评判，用估计值代替实际的搜索。估值函数的好坏直接影响了程序的“智能”的高低，它的构造将在论文后面部分介绍，在这里我们首先明确一点： f 函数是从甲方（即计算机方）立场出发的，估计值越大对甲方越有利，估计值越小对乙方越有利。令：

f(甲赢)＝＋∞

f(甲输)＝－∞

f(其它局面)＝＋∞与－∞中间值。

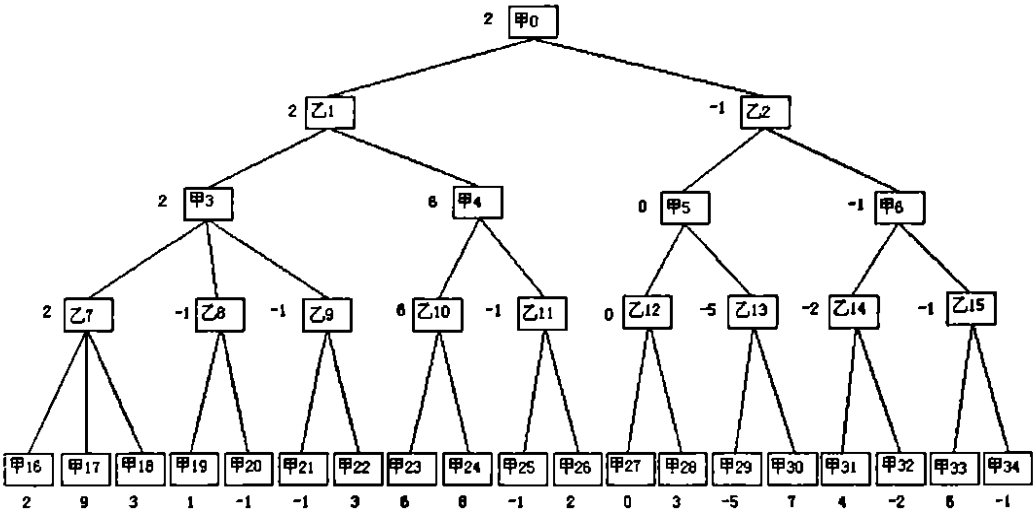

图 3－1 极小极大树搜索过程

一个最佳走步可以由一个极小极大化过程产生，甲方要从搜索树局势 0 的叶节点中选取，他肯定要选择拥有最大值的节点。因此，一个标有“甲”的节点的估计值可以由它的标有“乙”的子节点的最大值确定。另一方面，

乙方从叶节点选时，由于估计值越 小对它越有利，因此必然选取估计值值最小的节点（即最负的估计值）。因此，标有“乙”的节点估计值可由它的标有“甲”的子节点的最小值确定。综合以上两方面，可从博弈树的叶节点出发，一层层倒推得到上一层的估计值，直到得出根节点的估计值，这样就可以确定从根节点出发的最佳走步。由于这样，我们也称叶节点以上的各节点的值为倒推值。

在图3－1的博弈树中，从局面1出发，展开了两层，假设已经由估值函数计算出局面16到34的估计值，写在节点下方。我们来看看甲方在局势0如何根据博弈树选择最佳走法。

首先我们可由估值函数f计算出叶节点形势甲16到甲34的估分值，已标在节点下方。然后根据上述原则，确定深度为3的节点乙7到乙15的倒推值。如乙7局势是轮到乙方下棋，他要从子节点甲16、甲17及甲18中选择有最小估值的形势做为他的走步。在这里，f(甲16)<f(甲18)<f(甲17)，因此选 f（甲16）＝2 做为乙7的倒推值，即乙7形势下，乙选甲16做为下一走步。同样道理，可以得到乙8到乙15节点的倒推值，标在节点旁边。

再次，我们要确定深度为2的节点甲3到甲6的倒推值。如甲3局势下，轮到甲方选择最佳走步，他要从子节点乙7到乙9中选择有最大值的为他的下一走步。这里 2>-1， 因此甲3的倒推值取2，即甲方在局势3时应选乙7 做为下一走步。同理甲4到甲6也取子节点估值的最大值，它们的倒推值也标在节点旁边。

再次，乙1、乙2同理选择子节点估值的最小值作为倒推值，分别为2和－1。

最后，我们看根节点甲0，甲在局势0下选择具有最大估值的子节点乙1做为他的最佳走步。

整个过程有效性基于这样的假设[8]：开始节点的后继的倒推值比直接从

静态评估函数中得到的值更可靠。由于倒推值基于在博弈树中的预先推算，并且取决于在博弈结束时发生的一些特性，这些值往往更加切合实际。

实际上可以看出，问题转化为对某一局面下衍生出的最小最大树的遍历。这种遍历用一个深度优先搜索就可以处理了。在搜索的过程中，当前路径上的每一个结点上都保存着搜索到目前为止的当前最优值。搜索用一个递归函数实现，该函数的功能是计算某个节点的最优值。每当一个结点得到子节点的返回值时，就从当前最优值和返回值中选取一个更优的值作为当前最优值。当一个节点的子节点都搜索完毕之时，其当前最优值就是该节点的实际最优值，也就是这个节点的得分，这时，将这个值返回其父节点。下面给出极大极小搜索的伪码[9]：

```
double minimax(int depth, Position p)
{                          /* 计算局面 p 的分值 */
  int i;
  double f, t;
  if(depth==0)
    return evaluate(p);      /* 叶节点 */
  找出 p 的后继局面 p_1, …, p_w;
  if(电脑下棋)
    f=-1000;
  else
    f=1000;    /*初始化当前最优值，假设 1000 为一个不可能达到的值*/
  for(i=1;i<=w;i++)
  {
    t=minimax(depth-1,p_i);
    if(t>f && 电脑下棋)
```

```
            f=t;
        if(t<f && 对手下棋)
            f=t;
    }
    return f;
}
```

3.2 Alpha-Beta 剪枝过程

上述的极小极大值搜索过程中，遍历了整棵的博弈树，每一个节点都访问了一次，这样的搜索算法粗糙，效率低下，搜索量非常大。但如果要减少搜索量，就可能影响搜索效果。假如将叶节点的评估，计算倒推值与树的产生同时进行，就可能大量减少所需搜索的节点数目，而且保持搜索效果不变。这个技术叫 Alpha-Beta[10]剪枝过程。

假设采用深度优先算法搜索，叶节点一产生，便被静态评估，并假定一旦一个位置可给与倒推值时，这个值便被计算出来。

图 3-1 的博弈树说明 Alpha-Beta 剪枝过程。Alpha-Beta 剪枝过程原理[11]如图 3-2 所示。

首先根据甲 16、甲 17、甲 18 叶节点的静态估值算的乙 7 的倒推值，令乙 7 倒推值为 b(乙 7)=2。节点甲 3 是取极大值的节点，因此倒推值必然有 b(甲 3)≥ 2 。令 Pb（甲 3）＝ 2 为甲 3 的候选倒推值。

然后根据深度优先算法搜索到乙 8 节点，它先生成的甲 19 节点，f (甲 19)＝ 1 ，乙 8 是取极小值的节点，所以它的倒推值 b (乙 8)≤ 1 ，即使再搜索乙 8 的子节点，b (乙 8)也不可能大于 1 。再上一层，可以看出它不可能改变 Pb（甲 3）的值，所以乙 8 以下的其它子节点可以停止搜索了，因为

它们对寻找最佳解已经不能做出贡献了。同理由 f（甲 21）≈－1 可以停止对乙 9 以下其它节点的搜索。至此可以确定 b（甲 3）＝ 2，b(乙 1)≤ 2 ＝ P b （乙 1）。

由 f（甲 23）＝ 6，得 b（乙 10）＝ 6，b（甲 4）≥ 6，而已知 b（乙 1）≤ 2，可见对甲 4 以下的分枝可以停止搜索了。

b（乙 1）＝2，b 甲 0）≥2＝Pb（甲 0）。

又由 f（甲 27）＝0，所以 b（乙 12）≤ 0，而 b（乙 13）≤－ 5，得 b（甲 5）≥ 0，b（乙 2）≤ 0 。又已知 b（甲 0）≥ 2，所以乙 2 的其它分枝已经没有必要搜索了。最终，得 b（甲 0）＝2，选子节点乙 1 做为甲方的最佳走步。

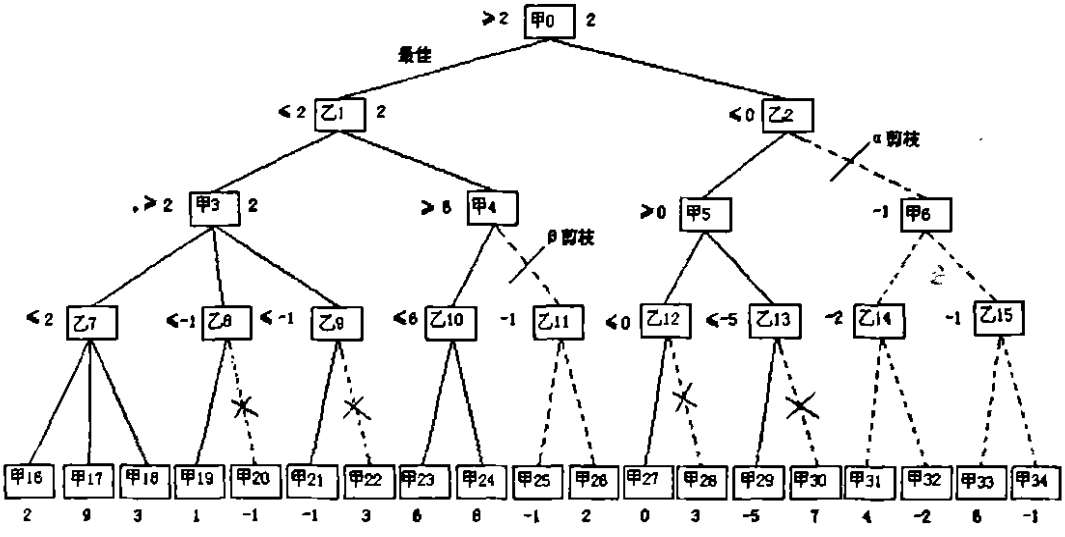

图 3－2 Alpha-Beta 剪枝过程

从上面可以看出，原来要搜索 35 个节点，现在只要搜索 21 个就可以了，搜索量大大减少，而且也不会因为搜索量的减少，使得搜索结果有误。

我们把搜索过程中得到的极大节点 X 的候选倒推值 Pb(X)叫 α 值，它是 b（X）的下界，不可能再下降了。而把极小节点 Y 的候选倒推值 Pb（Y）叫 β 值，它是 b（Y）的上界，不可能再上升了。

在搜索过程中，α和β值按以下规律计算[12]：

（1）极大节点的α值等于当前子节点中的最大倒推值；

（2）极小节点的β值等于当前子节点中的最小倒推值。

Alpha-Beta剪枝的规则如下：

规则 1 当任何极小节点的β值小于等于它的极大的祖先节点的α值，则可以中止该极小节点以下的搜索。该极小节点的最终倒推值即为它的β值。该值与完全极小化搜索得到的值不等，但同样可以选择相同的最佳移动方式。

规则 2 当任何极大节点的α值大于等于它的极小的祖先节点的β值时，则可以中止该极大节点以下的搜索，该极大节点的最终倒推值等于它的α值。

当按规则1停止搜索时，称产生Alpha剪枝；当按规则2停止搜索时，称产生Beta剪枝。保存α、β值，并且当可能时进行剪枝的整个过程通常称为 Alpha-Beta 剪枝过程。当开始节点的所有子节点都得到倒推值后，过程终止。最佳首步移动是产生有最大倒推值的子节点的移动。该过程产生的移动与简单使用极小极大过程搜索相同深度得到的方案是相似的。唯一不同是，Alpha-Beta过程通常用少得多的搜索就可以找到最佳首步移动。

Alpha-Beta过程算法伪码[13]如下：

```
double alphabeta( int depth, double alpha, double beta, Position &p );
{                                   /* 计算局面 p 的最优值 */
    int i;
    double t;
    if(depth==0)
        return evaluate(p);                    /* 叶结点 */
     找出 p 的后继局面 p_1, …, p_w;
```

```
if (极大节点)
{
 for(i=1; i<=w; i++)
 {
  t=alphabeta(depth-1,alpha,beta,p_i);
  if(t>alpha)
     if(t>beta)
        return t;
     else
        alpha=t;
  }
  return alpha;
 }
else /*极小节点*/
{
  for(i=1; i<=w; i++)
  {
     t=alphabeta(depth-1,alpha,beta,p_i);
     if(t<beta)
      if(t>alpha)
           return t;
  else
     beta=t;
  }
  return beta;
```

```
    }
  }
```

3.3 Alpha-Beta 剪枝存在的问题及优化

为了完成 Alpha-Beta 剪枝，至少部分搜索树要产生至最大深度值，因为 α、β 值需要基于叶节点的静态值。因此，使用 Alpha-Beta 剪枝过程通常用到深度优先搜索。从上述例子看，Alpha-Beta 剪枝过程搜索效率与节点的排列顺序有很大关系。

在图 3－2 的剪枝过程中，其中的如图 3－3 的一部分，产生了两处剪枝，节点甲 20 和节点甲 22 不再搜索了。

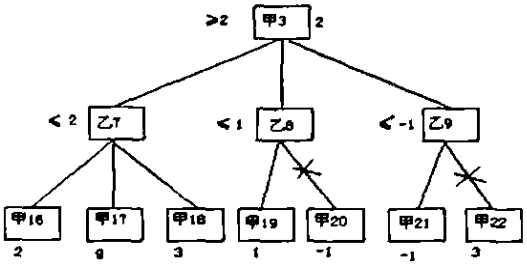

图 3-3　有利的节点排序

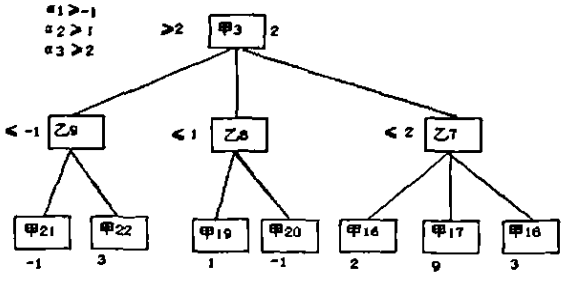

图 3-4　不利的节点排序

如果图3－3的节点排列顺序变成如图3－4的顺序，那么结果就大不一样了。在乙9节点处得到β值为－1，由此得甲3的第一次α值为$\alpha_1=-1$，它不能让乙8节点的子节点产生剪枝。访问了乙8节点后，乙8的β值为1，于是甲3的α值变为$\alpha_2=1$。同样，甲16、甲17和甲18都要被访问，才能确定甲3的α值应为$\alpha_3=2$，从而得到甲3的最优倒推值为2。

可见由于节点顺序的改变，图3－4中Alpha-Beta剪枝过程就无计可施。

假设一棵博弈树深度为d，并且每个节点（除叶节点）都由b个子节点，这样一棵树大约由b^d个叶节点。假设Alpha-Beta过程产生的子节点按其倒推值大小排－－对极小节点按从小到大顺序，对极大节点按从大到小顺序（当然，它们的倒推值是不可能在后继节点产生前知道的，因此，除非偶然，否则该顺序不可能得到）。

这种排序可以使剪枝的数目最大化，并使产生的末端节点数最小化。我们用N_d表示叶节点的最小数目，可如下表示：

$$N_d=\begin{cases}2b^{d/2}-1 & d\text{为偶数}\\ b^{(d+1)/2}+b^{(d-1)/2}-1 & d\text{为奇数}\end{cases}$$

也就是说，优化的Alpha-Beta搜索产生的深度为d的树，其叶节点的数目等同于不用Alpha-Beta搜索时深度为d/2产生的叶节点数。因此，当不用Alpha-Beta的搜索进行到d/2深度时，Alpha-Beta搜索（最佳排序）将进行到d深度。

当然，最佳节点排序是不可能实现的（假如可实现，就完全不需要搜索过程了）。

最糟糕的情况下，Alpha-Beta搜索不会产生剪枝，有效分枝因子不变化。实际上，假如将好的试探方法用于排列后继节点，Alpha-Beta过程通常可以最大程度地减少有效分枝因子。

在这里，我们用了这样的一些方法来排序：

（1）　由于五子棋以五子连为胜，因此一个已有子周围的点是较有威胁的点，而且最后下的一子的周围点威胁性更大。换而言之，这些点对极小节点来说，倒推值会相对较小，对极大节点来说，倒推值会相对大些。可将它们排在搜索队列前面，优先搜索。本程序中在存储棋子的栈中从栈顶到栈底，依次取其中有棋子位置的周围 24 个点做为搜索节点。如图 3－5 中，黑子周围的 24 个空点就是优先搜索的点。

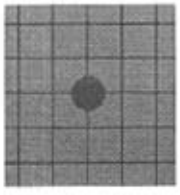

图 3－5 优先搜索空点 1

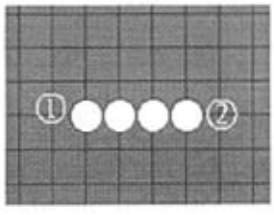

图 3－6 优先搜索空点 2

（2）对一些特殊棋型周围的点，如图 3－6 中的①和②空点，它们很可能就是当前局势的极小极大值，优先搜索它们，可以减少极小极大算法的搜索量。

3.4 NegaScout 搜索

正常情况下，在搜索树的根结点，也就是电脑棋手需要决策的局面，我们可以调用 3.2 中给出的函数 alphabeta 来得到根结点的分值，同时也可以得到应该走哪一步棋的结论。这时，调用该函数时，我们应使参数 alpha=－1000，beta=1000，这里 1000 是一个足够大的数。这是因为在根结点时，我们需要把窗口初始化成最大，在搜索过程中，随着信息的获得，这个窗口会渐渐地减小，最后收敛到根结点的最优值上。

现在我们来看一个有趣的现象：如果我们调用 alphabeta(d,b-0.1,b,p)会产生什么结果？这里 0.1 是一个较小的常数，而 b 是某一个数值。

直接从程序中分析，现在形式参数 beta 就等于 b。如果函数的返回值大于 b（也就是 beta），说明对于局面 p，至少有一个子结点的返回值比 b 大，所以其最优值必然大于 b。这时，我们知道 b 是局面 p 最优值的下界。注意，在这种情况下，后面也许还有子结点根本来不及搜索就被裁剪了，所以这时的返回值并不代表局面 p 的实际最优值，可能比实际最优值小。如果函数的返回值小于 b，说明没有一个子结点的最优值比 b 大，于是局面 p 的最优值必然小于 b（事实行，由于初始的窗口在函数递归过程中被反复传递下去，所有的返回值都只体现了是否比 b 大的信息，所以返回值不是局面 p 的实际最优值）。此时，b 是局面 p 最优值的上界。于是，我们得到结论，运用这种类型的 alphabeta 函数调用，根据返回值是否比 b 大，可以得出局面 p 的最优值是否比 b 的结论。

这结论可以表示成三种可能结果：

1．若 f(p)≤b-0.1，则 MM≤b-0.1；

2．若 f(p)≥b，则 MM≥b；

3．若 b-0.1<f(p)<b，则 MM=f(p)。

其中，MM 是 alphabeta(d,b-0.1,b,p)的返回值，f(p)是博弈树节点 p 极小极大值。

上面中的情况 1 和 2 分别称为偏高和偏低。这两种情况虽然没有准确的求出 f(p)的值，但却对 f(p)值的范围提供了有用的信息，以此为基础可以实行进一步的搜索。

注意到在函数调用过程中，alpha 和 beta 很接近，因此窗口很小，在窗口很小的情况下，搜索树的各个层次中，得到的返回值比 beta 大的可能性就大大增加了，这样剪枝的可能性也变大了。

这种类型的 alphabeta 函数调用被称为零窗口 Alpha-Beta 搜索[20]，由于产生了大量的剪枝，它的时间消耗比正规的 Alpha-Beta 裁剪少。但正规的 Alpha-Beta 裁剪可以得到一个局面的实际最优值，而零窗口 Alpha-Beta 调用只能得到实际最优值是大于或小于某一个特定值 b 的结果，也就是说，一个上界或下界的信息。

于是，可以对 Alpha-Beta 裁剪作一个改进：在每次向下搜索之前,首先用零窗口 Alpha-Beta 调用判断一下，这个搜索是否能够改善当前的最优值，即局面 p.某一个子结点的实际最优值是否比 alpha 大。如果比 alpha 大，则搜索该子结点，否则就跳过这个子结点。这就是 NegaScout[7]搜索。换个说法就是：如果 f(p)值相对窗口[b-0.1,b]来说是偏低了，则下一步搜索可以[$-\infty$,MM]为窗口，如果偏高，则可以[MM,$+\infty$]为窗口。

在这个新算法中，增加的时间消耗是零窗口 Alpha-Beta 调用，而如果零窗口 Alpha-Beta 调用的返回值比 alpha 小，那么就可以节约一个较大窗口的 Alpha-Beta 调用花费的时间。实践证明,由于零窗口 Alpha-Beta 裁剪中产生了大量的剪枝，所以其实践消耗相对于大窗口 Alpha-Beta 调用而言是很小的[18]。总的说，NegaScout 搜索较 Alpha-Beta 裁剪有效。下面依照前述的思想，给出一个 NegaScout[19]算法的伪码：

```
double negascout(int depth, double alpha, double beta, Position &p );
{          /* 计算局面 p 的最优值 */
    int i;
    double t;
    if(depth==0)
    return evaluate(p);    /* 叶结点 */
    找出 p 的后继局面 p_1, …, p_w;
```

```
    if (极大值节点)
     {
     for(i=1;i<=w;i++ )
      {
        t =negascout(depth-1,alpha,beta,p_i);
        if(t>alpha)
          if(t>beta)
            return t;
          else
            alpha=t;
      }
    return alpha;
}
else //极小值节点
{
  for(i=1;i<=w;i++)
  {
    t=negascout(depth-1,alpha,beta,p_i);
    if(t<beta)
      if(t<alpha)
          return t;
      else
          beta = t;
  }
return beta;
```

```
}
```

3.5 小结

本章给出了极大极小节点的定义，研究了博弈树的极小极大搜索技术及在此基础上的Alpha-Beta剪枝过程和剪枝优化问题。实现将候选的后继节点按位置邻近顺序排序，使剪枝过程得到优化。阐述了NegaScout搜索，并与Alpha-Beta裁剪作了比较，指出NegaScout搜索较Alpha-Beta裁剪有效。

第 4 章 静态估值函数

在前述博弈树搜索算法中，我们假定最后叶节点的静态估值函数是绝对可靠的，每次都以该估计值为依据。但实际上，这估值函数肯定是会有误差的，否则就不叫静态估值函数了。而一旦在某个下级节点处发生误差，这种错误沿着通路向上传播时，将有可能造成上级节点作决策时产生关键性失误，并且这种失误是无法纠正的。因此，估值函数设计的好坏，直接影响了计算机下棋的“智能”的高低。下面介绍本程序初版中估值函数的设计方法。

4.1 棋盘局势特征

为了给某一局势估分，提取局势中一些特殊的棋型，给它们分别估分，并累加求和，得局势的总得分。

由于五子棋规则中，横、竖及斜方向上的同色五子相连都为赢，因此在寻找特征时， 横、竖、斜方向上都要寻找。我们称棋盘上某一行、某一列或某一对角斜线为一路，即有如图 4－1 四种情况。

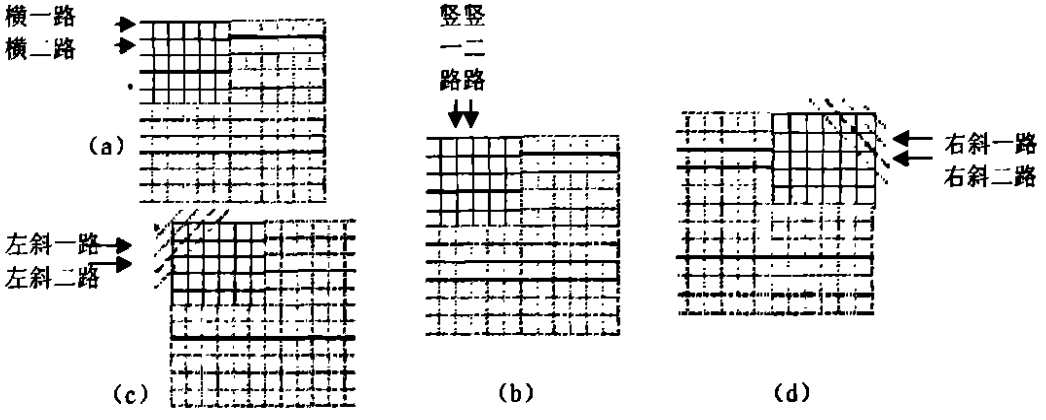

图 4－1 棋盘路示意图

使用的棋盘为15行15列，因此行和列中共有15+15=30路；图c中从右上到左下的对角斜线有29路（包括左上的顶点和右下顶点）；图d中从左上到右下的对角斜线也有29路，但对于五子棋来说必须在一条直线上有连续五个棋子才能赢，因此在对角线上就可以减少8路。所以，整个棋盘中共30+21+21=72路。

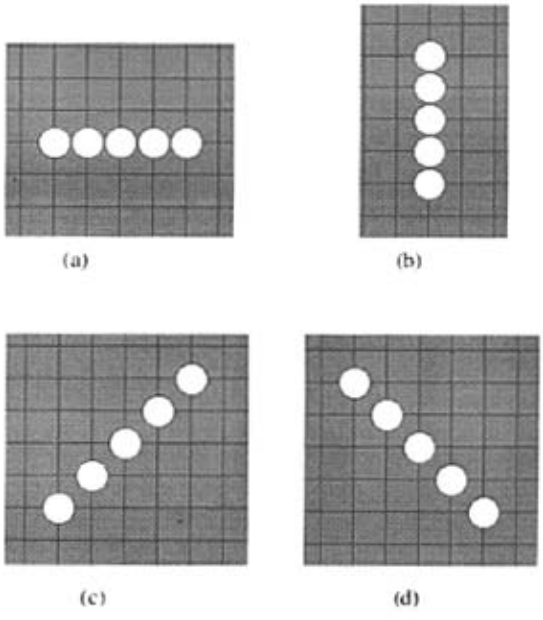

图 4−2 棋盘特征形式

对一个特征，如“五子连”，在棋盘上有四种出现形式。如图4−2。不管哪种形式，我们都称为某路上有此特征。如(a)图称为横x路上有五子连特征，（b）图称为竖x路上有五子连特征，（c）图称为左斜x路上有五子连特征，（d）图称为右斜x路上有五子连特征。其它特征也是同样。

程序使用的主要特征有：

(1) “OOOOO”　　(2) “+OOOO+”

(3) “+OOO++”　　(4) “++OOO+”

(5) “+OO+O+”　　(6) “+O+OO+”

(7) “OOOO+”　　(8) “+OOOO”

(9) “OO+OO”　　(10) “O+OOO”

(11) “OOO+O”　　(12) “++OO++”

(13) “++O+O+”　　(14) “+O+O++”

(15) “+++O++”　　(16) “++O+++”

等。其中，“O”表示有棋子，“＋”表示空点。

4.2 估值

有了特征后，就可以给特征分别赋评估分值，赋值时应考虑的规则[17]有：

1．赋值从计算机方的立场出发（假设计算机下白子），那么，出现白子的特征给正分，出现对手黑子特征给负分。以符合评分越大，对计算机方越有利；分越小，对计算机方越不利，而对对手越有利的原则。

2．同一特征可能在不同路上（甚至同一路上）会重复出现，一些威胁性小的特征即使出现多次，累加得分也不应该超过一个极有威胁性的特征得分。如特征“++OO++”威胁性小，特征“+OOOO+”威胁性非常大，不管前者出现多少个，原则上得分都不应该超过后者出现一个的得分，尽量使它们得分差距大些，让估分做到最大限度的准确。

给主要的规则评分如下：

(1) “OOOOO” →50000　　(2) “+OOOO+” →4320

(3) “+OOO++” →720　　(4) “++OOO+” →720

(5) “+OO+O+” →720　　(6) “+O+OO+” →720

(7) “OOOO+” →720　　(8) “+OOOO” →720

(9) “OO+OO” →720　　(10) “O+OOO” →720

(11) “OOO+O” →720　　(12) “++OO++” →120

(13) “++O+O+” →120　　(14) “+O+O++” →120

(15) “+++O++” →20　　(16) “++O+++” →20

上面的评分是从计算机方棋子有这些特征，给正分。如果是对手有这些特征，则得分取反，得负分。例如对手如果有“OOOOO”特征，将得－50000 分。

为了求局势总分 f，我们先求第 i 路得分 h(i)：

h(i)=hc(i)+hm(i)

其中 hc(i)为第 i 路计算机方总得分，hm(i)为第 i 路对手总得分。

如图 4－3，在横的这路上，白方出现特征（3），（我们只匹配一个规则，因此特征（4）不再统计），黑方出现特征（12），因此有

hc(i)=720 ; hm(i)=-120

h(i)=hc(i)+hm(i)=720-120=600;

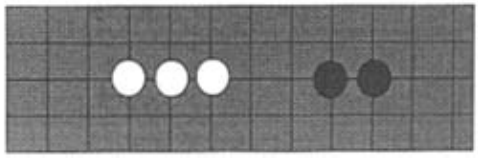

图 4－3 局势特征例

棋盘上总共有 72 路，所以局势 n 的总得分为：

$$f(n)=\sum_{i=1}^{72}h(i)$$

4.3 静态估值函数的不足及实践结果

采用极小极大搜索加静态估值技术，实践中，达到了比初学者强的水平，一些比较熟练的业余人员时常也会负于此程序。

采用固定的估值法，为设计这个估值函数，要求设计人员对下棋的方法有较多的了解，能充分判断棋局局面中的某一特征在形势判断中所起的重要程度 （即相应的分值），并给整个局面比较准确的评分。但是面对成千上万的局面，即使是大师也不可能一一做出精确的形势判断，特别是在对局的开始阶段，棋局的优劣更难以判断。而且，如果对大量的棋局状态进行存储，就要求有大的存储空间及快速的搜索算法。因此，静态的估值函数不可能有很大的准确性。估值的不准确使其“智力”较低，而且固定的赋值方式使其估值准确度不能通过学习改善，“智力”也就不能提高。因此继续研究，通过对搜索算法的优化与修正，特别是针对五子棋本身对弈的特点和规律，采用置换表搜索与威胁空间搜索相结合的搜索技术，显著的提高了五子棋程序对弈的水平和能力。

4.4 小结

本章根据五子棋的特点，提取棋局局势的若干特征，对这些特征赋加权分，并对整个棋局进行特征统计，采用一个线性函数求得棋局的总估计分值。

第 5 章 搜索算法的优化与增强

为了进一步提高五子棋程序的对弈水平，必须采用更加行之有效的搜索技术，本章首先讨论的置换表的搜索是直接在前述算法的基础上的完善和加强，然后着重介绍的是威胁空间的搜索，两者的结合，可以有效的提高五子棋对弈中最佳走法的搜索问题。

5.1 置换表（Transposition Table）的搜索

在前面第三章讨论的极大极小搜索的过程中，使用了 Alpha-beta 剪枝技术和 Negascout 算法来改进搜索的效率。改进搜索算法的目标在于将不必搜索的（冗余）分枝从搜索的过程中尽量剔除，以达到尽量搜索少的分枝来降低运算量的目的。

我们知道可以通过 Alpha-Beta 剪枝来剪除两种类型的冗余/分枝节点，但是已经搜索过的节点可以免于搜索。

现在来看一下如图 5-1 中极大极小树的片断，该片断从节点 A 开始，有两个分枝 B 和 C(为简化问题，其他节点从略)，B 有子节点 D，C 有子节点 E，再往下 D 有子节点 F，E 有子节点 G。可以看到，在极大极小树的不同分枝上，存在着完全相同的节点。在图 5-1 中，节点 F 和节点 G 完全相同。对于节点 F，在搜索节点 B 的时候，就可能已经搜索过了。那么，在搜索 G 的子节点时，我们能不能直接利用已经搜索过的节点 F 的结果，而不是重新搜索一遍节点 G 呢？

想法是可行的。如果要利用已经搜索过的节点的结果，我们就要用一张表把搜索过的节点记录下来。然后在后续的搜索中，察看记录在表中的这些结果，如果将要搜索的某个节点已有记录，就直接利用记录下来的结

果。这种方法叫做置换表（Transposition Table,简称 TT）。

如果将 Alpha-Beta 搜索过程中每一节点的结果都记录下来，则在任意节点向下搜索之前先察看这些记录，就可避免上述重复的操作。

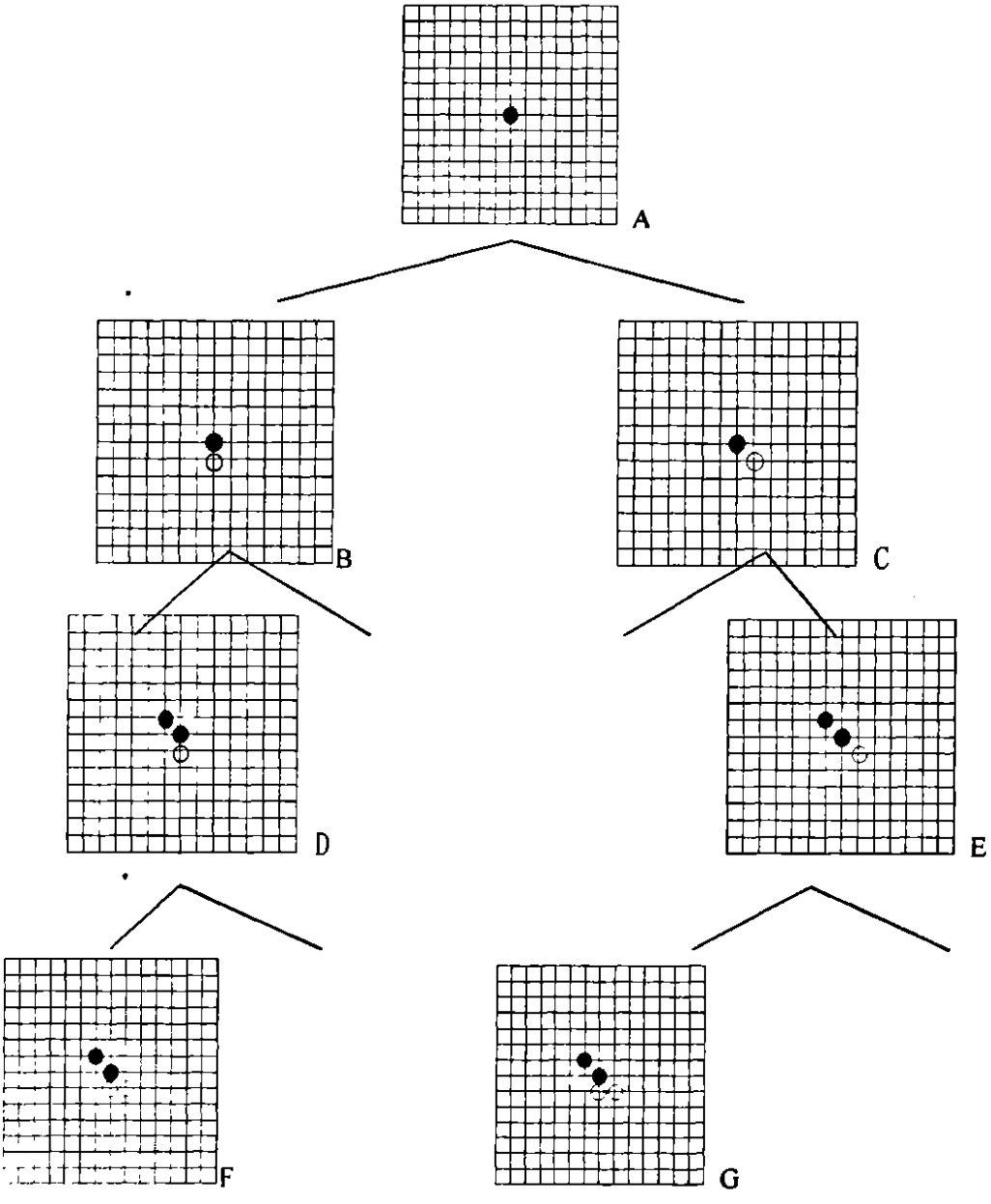

图 5-1 极大极小树存在重复节点

我们将这一过程以伪代码表述如下[19]：

```
//类 C 伪代码，查询置换表的 Alpha-beta 搜索
int alphabeta(depth,&alpha,&beta)
{
    value=lookupTT(depth,index);    //查询置换表
    if(value is valid)      //察看此节点是否已在置换表 TT 中
        return value;       //命中，已有直接返回表中记录值
      //没查到，进行 alphabeta 搜索过程，求出节点值 value
  Search with alphabeta...
  //将搜索过的值记录到 TT 当中
  StoreToTT(depth,value,Index);
  return value;
}
```

5.1.1 哈希表（Hash Table）

在置换表的搜索算法中，关键的问题是：置换表如何实现？困难集中在时间和空间复杂度上[20]。

（1）可能的节点可以认为是无穷多，将其存入一张表中要占据多少存储空间呢？

（2）即使有无穷多内存空间，一个节点要从表中找到自己对应的一项，需要花费多少时间？虽然有序表可用二分查找等快速的算法，但要维持表中的内容有序，又要进行大量排序操作，会耗费更多时间。如果花费在置换表中查找对应项的时间及进行排序的时间的开销大于使用一般的 Alphabeta 搜索的时间开销，显然是得不偿失的。

因此要想实现基于置换表的搜索必须要满足如下条件：

（1）查找记录中的节点数据时速度要非常的快，最好是类似于随机存

取。

（2）将节点数据放入记录的速度也要非常快，这就意味着数据项的插入过程不可能出数据移动排序等操作。

（3）要能在有限的存储空间内进行。

解决上述问题的方法就是哈希表，哈希方法的思想为每一个学过数据结构或算法课程的开发人员所熟知。对棋类博弈来说，定义一个巨大的数组，将局面记入其中，每一局面在数组中对应唯一的位置。这样，将数据记入和读出就类似于随机读写，不会有耗时的查找和插入过程。但是，以五子棋而论，如果为每一局面在数组中对应一个与其它局面不同的位置的话，则组合的结果是全世界的计算机内存加起来也不够这样一个数组用。

解决的方法是让每一局面在数组中对应唯一的位置，但并不保证数组中每一个数据项对应唯一的局面。比如将所有的局面对应在 10^6 单位的数组上。这样，必然有一些局面对应在相同的位置上。但是对一次搜索而言，这种情况发生的概率并不高。一旦某个搜索过的局面在该数数组中未找到。我们不过是对它进行 Alpha-beta 搜索而已。不同的局面对应在数组中存储单元上的情形，叫做冲突。

下面是利用哈希方法实现置换表的方法，定义哈希数组如下：

```
struct HASHITEM
{
  int64 checksum; //64 位哈希值，用以验证表中数据是否是要找的局面
  int depth;      //该表项求值时的搜索深度
  //表项值的类型
  enum {exact,lower_bound,upper_bound} entry_type;
  double eval;    //所代表的节点的值
```

```
}hashtable[HASH_TABLE_SIZE];
  //定义大小为 HASH_TABLE_SIZE 的哈希数组
```

对要搜索的每一节点，计算出它的一个哈希值（hashIndex，通常是一个 32 位对哈希表大小取模）以确定此局面在哈希表中的位置。计算另一个哈希值（checksum）来校验表中的数据项是否是所要的那一项。这通常是一个 64 位数，实际上 64 位数在理论上也还是会有冲突，但其发生的几率已经非常小，在程序中可以将其忽略而几乎没有影响。

在对某一局面搜索之前，先察看哈希表项 hashtable[hashIndex],如果 hashtable[hashIndex].checksum==checksum 并且 hashtable[hashIndex].Depth 大于等于最大搜索深度减去当前层数，就返回 hashtable[hashIndex].Eval 作为当前局面的估值。

当对一个局面的搜索完成之后，将 checksum、当前层数和估值结果保存到 hashTable[hashIndex]当中，已备后面的搜索使用。

下面的伪代码示意了如何使用置换表与 Alpha-beta 搜索协同工作。

```
//类 C 伪代码，alpha-beta 搜索+置换表
int alphabeta(int depth,int alpha,int beta)
{
  int value,bestvalue=-INFINITY;
  int hashvalue;
  if(Game Over)
    return evaluation; //当前节点胜负已分，返回估值
  if(depth<=0)
    return evaluation; //叶子节点返回估值
  //察看置换表中有无当前节点数据
  if(lookup(depth,&alpha,&beta,&hashvalue))
```

```
        return hashvalue;  //有，直接返回
    for(each possibly move on) //对下一步每一可能的走法 m
    {
      make move m;  //产生 m 的子节点；
      //递归搜索子节点
      value=-alphabeta(d-1,-beta,-localalpha);
      unmake move m;          //撤销子节点
      if(value>bestvalue)
        bestvalue=value;  //保留最佳值
      if(bestvalue>=beta)
        break;              // bate 剪枝
      if(bestvalue>alpha)
        alpha=bestvalue;  //修改 alpha 范围
    }
    //将搜索过的结果存入置换表
  store(depth,bestvalue,alpha,beta);
  return  bestvalue;
}
```

5.1.2 应用置换表的其它问题

基于哈希的置换表的应用，还有一些重要的问题应予以注意。

（1）清除哈希表

哈希表是否需要在每次搜索前清空？答案是不需要的。因为，前次搜索的内容对后面的搜索并无不良影响。并且，清除哈希表本身也需要耗费时间。在搜索层数较深的情形下，后继搜索过程中很多节点在前次搜索的过程中出现过。这样就可以直接使用而无须重复搜索，这还会提高搜索性

能。在层次较低的搜索中，这一现象几乎观察不到，但随着搜索深度的加深，当前搜索从前次搜索遗留的哈希数据中受益的程度会有所提高。

正是由于这个原因，当同一哈希地址上要插入第二个哈希表项时，只要64位哈希值不同，也就是说不是同一节点。如果发生冲突了，后来的数据应当覆盖先前的数据，因为先前的数据可能是以前的搜索遗留的。

（2）哈希值的层次

在前面的哈希表项定义里，有一项记录了该表的搜索深度。这一项在使用时也很重要。因为同一局面在搜索树的不同层次上出现，如果哈希表中记录了某一个节点的值，但当前要搜索的相同节点的层次不同。这时，如果哈希表中的节点层次较待搜索的节点层次低，也就是更靠近叶子节点，则应当进行搜索并将搜索结果覆盖到哈希表中；如果哈希表中的节点层次较待搜索的节点层次高，也就是更靠近根节点，则应当直接引用哈希表中的数据而免于搜索。更一般的结论是层次越高的节点其求值精度也越高。所以，置换表的搜索可在某种程度上提高搜索精度。

但因用不同层节点的哈希值时要注意其节点类型，在基于极大极小值的搜索算法里取极大值的节点对取极小值的节点的值来说是无用的，反之亦然。如果做相互引用，则有可能得到不正确的搜索结果。

也可以将置换表设计成仅有相同层次的节点才可以引用，这样取得的搜索结果将与不用置换表的搜索结果完全相同，仅仅是搜索速度有所差异。在这种情况下，可以建立多个大小不一的哈希数组以容纳不同层次上的节点，就可以避免不同层次上的节点相互覆盖。作者的一些实验表明，这样的置换表与前述可在不同层次引用的设计在性能上没有明显差异。这也表明实际上查询置换表时命中的节点绝大部分是同层的节点。

（3）子节点

在使用基于哈希表的置换表搜索技术中，是否对叶子节点进行哈希处

理？因为叶子节点数量巨大，并且哈希查找和插入也需要一定的时间耗费。所以哈希处理可能节约不了多少时间，甚至使搜索效率降低。

这个问题应该根据具体的情况而定。对于围棋等估值函数复杂，估值过程时间长的棋类来说，对叶子节点进行哈希处理对性能的提高是肯定的。而对于象棋等棋类，则应视估值函数的时间而定。搜索的最大深度越大，哈希表的命中率越高，相对每一个节点都要执行查找/插入操作的时间花费而言，省下的时间也越多。当搜索的最大深度很浅时，哈希表的命中率不高，此时对叶子节点进行哈希操作不一定能提高搜索速度。因此，在设计不同的博弈程序时最好进行试验以实际测定对叶子节点进行哈希处理给搜索性能带来的影响。而如果估值函数速度很快，这就没必要对叶子节点进行处理了。这样哈希表中存放的节点会少得多，哈希表的尺寸也可以设计的更小。

由于在最初的五子棋程序设计时，在估值函数的处理过程中，更多的是考虑程序的易实现、易读性，所以估值函数的执行速度比较慢，因此，在改进的五子棋程序中，在置换表的搜索中对叶子节点进行处理可以取得更高的搜索速度。

（4）散列度

哈希表的性能与哈希函数有极大关系，散列程度高，冲突少的哈希函数拥有更高的命中率。因此哈希表的散列程度是影响其性能的重要因素。

影响散列度的主要因素之一是求哈希值的方法。可以有多种方法来计算哈希值，这方面的方法在大量的介绍算法或数据结构的书籍中都有所介绍。本程序中使用的是Zobrist哈希技术，一种在博弈程序中广泛使用的随机哈希方法，兼顾了速度和优秀的散列度。

（5）哈希表的尺寸

在均匀程度相同的情形下，越大的哈希表有越好的散列度。哈希表越

大，冲突越少，性能越佳。因此，定义一个尽量大的哈希表可在某种程度上提高查找效率，鉴于 PC 机环境上配置的内存越来越大，使用几十兆甚至上百兆的哈希表也有可能。但是，哈希数组的定义不应超过物理内存的大小。一旦使用虚拟内存，哈希表的操作速度将大幅度降低。对于相同搜索深度而言，哈希表的大小与性能的关系并不是线性的。Jonathan scheffer 曾指出,一般哈希表的大小每增加一倍，约可使命中率提高 7%左右。但当哈希表增大到一定程度时，再增加其大小带来的性能提高可能相当小。也就是说，对于某以确定的搜索深度而言，当哈希表的尺寸大到已使其中的冲突接近零的时候，再增加哈希表的尺寸将没有任何实际意义，除非要进行更深层的搜索。在一个具体的博弈程序中，对于哈希表尺寸大小的确定，应通过具体实验来确定合适的哈希表尺寸，已达到尽可能好的效果。

（6）解决冲突

一般是通过使用链表或其他重定位的方法来解决哈希表的冲突，而不是在发生冲突的时候覆盖已有数据。但是，解决冲突有以下几方面的不利因素：

①(同全部可能的局面相比)哈希表的大小非常之小，如果解决冲突，可能几次搜索就导致哈希表的空间近乎耗尽，从而使哈希操作的效率大幅降低。

②如果动态申请内存空间，则面临内存上的大量消耗。并且申请时所需的大量时间将导致性能进一步下降。

③由于开局时的哈希数据对中局和残局几乎没有帮助，在哈希表中解决冲突将导致大量数据冗余，从而使效率降低。

④依赖高散列度的哈希方法配合足够大小的哈希表，可以取得极低的冲突率。解决冲突带来的哈希命中率的提高微不足道。

因此，通常情况下，使用置换表的搜索不会在解决冲突的问题上下功

夫。但围棋等棋类可能是个例外。在围棋种，哈希表中遗留的内容对下一次搜索几乎没有帮助。并且由于围棋的估值函数的复杂度高，运算时间长，从而不可能搜索很深的层次。实际在哈希表中存放的节点数目远较五子棋等棋类少，所以解决冲突对于提高搜索效率仍是有价值的。

5.1.3 Zobrist 哈希技术

实现基于哈希的置换表还有一个重要问题，那就是：如何快速产生哈希值？对每一节点，我们要产生一个 32 位的索引用来定位节点在哈希表中的位置，还要产生 64 位的 checksum 来验证表中记录的内容与当前局面是否是同一局面。

在散列存储中，利用散列函数（哈希函数）可以将棋盘数据经过某种运算映射到一个 32 位数和一个 64 位数上面。并且这些数值应当尽量散列，以降低冲突发生的几率。由于对每一个节点都要求出两个哈希值，所以这个求哈希值的过程要尽量快速。

Zobrist 于 1970 年提出了一种快速求哈希值的方法，至今仍为博弈程序所广泛使用，下面对此方法作一介绍。

在程序启动的时候（也可在棋局开始的时候），建立一个多维数组（在五子棋里通常是三维）Z[pieceType][boardWidth][boardHeight]。其中 pieceType 是棋子种类，在五子棋种棋子种类为 2；boardWidth 为棋盘宽度，五子棋为 15；boardHeight 为棋盘高度，五子棋种同样为 15。然后，将此数组中填满随机数。若要求某一局面的哈希值，则将棋盘上所有棋子在数组 Z 种对应的随机数相加，即可得到。如在五子棋棋盘正中有一黑子（假定类型为 1），则该黑棋子对应的随机数就是 Z[1][9][9]；如该黑子相邻位置下方有一白子（假定类型是 0），则该白棋子的随机数是 Z[0][9][10]。将棋盘上所有棋子对应的随机数相加，得到哈希值。若要得到 32 位哈希值，数组 Z 中的元素应为 32 位，要得到 64 位哈希值，数组 Z 中元素应为 64 位。

在搜索过程中，棋盘的每一局面对应一个节点，对每一节点，计算哈希值都要将当前棋盘上所有棋子的随机数加总起来，是否速度太慢而影响搜索效率，对这个问题，Zobrist 方法可以用增量式计算的方法解决，无须每次都加总所有棋子。在程序的根部，做一次加总操作求出根节点的哈希值，当搜索一个新节点时只要棋子在移动前将对应的随机数从哈希值中减去，再加上该棋子在移动后对应的随机数即算出了该子节点的哈希值。如果移动的棋子吃了别的棋子，还要减去被吃掉的棋子被吃前所对应的随机数。当对这个局面搜索完成后，再将搜索前减掉的值加上，搜索前加上的值减掉，就恢复了当前节点的哈希值。

计算哈希值的加减过程也可以用（按位）异或操作替代-----比加减简单而且快一点。但加减一样可以工作的很好。

显然，增量式计算哈希值的方法在思想上同 Alpha-beta 过程中的 makemove/Unmakemove 是一致的。

Zobrist 哈希的方法，不仅可用于置换表，而且也可用于开局库和残局库等方面，还可以在程序中建立一个小的哈希表用来检测在某些棋类程序中是否发生了循环，比如象棋里的“长将”。

5.2 威胁空间搜索（Threat Space Search）

在这部分里，我们将首先讨论三个方面的五子棋知识以及它们的应用。第一方面的内容包括一些对棋手（人和计算机）来说非常重要的术语。我们将这些术语用于“威胁次序”。第二方面的内容介绍人类五子棋棋手思考时的策略。接下来介绍的是常规的五子棋程序走法选择的处理过程。最后，我们研究人类和计算机在走法选择的方法上究竟有多大程度的不同。从而使我们发现常规的五子棋程序在哪些方面有所欠缺。

5.2.1 专业术语和“威胁次序”

五子棋对弈中，“威胁”是个重要的概念。最主要的威胁有以下四种类型：

1，冲四（图 5-2a）：在有五个点的一条直线上，其中任意四个点被攻方棋子占据，而第五个点是空白的。

2，活四（图 5-2b）:在有六个点的一条直线上，攻方棋子占据了中间的四个位置，并且外面的两个位置是空白的。

3，活三（图 5-2c,d）：可以有两种情况，一是在有七个点的一条直线上，中心的三个点被攻方棋子占据，其余的四个点是空的；或者是在有六个点的一条直线上，中心位置的四个点中有三个被攻方的三个紧密相连的棋子占据，其余三个位置是空白的。

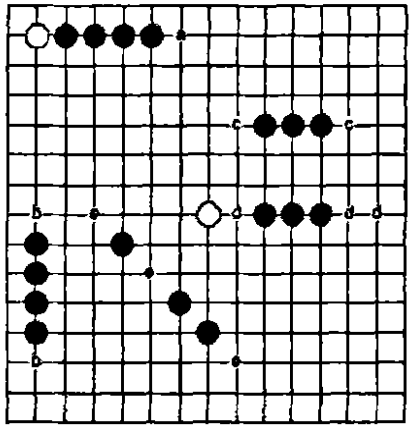

图 5-2 威胁的类型

4，断三（图 5-2e）：在有六个点的一条直线上，攻方三个并非紧密相连的棋子占据中心四个位置中的三个，而其余的三个位置是空白的。

如果棋手走出了一个“冲四”，他的下一步就会形成一个致胜的“威胁”。因此，这个“威胁”必须立即应对。如果在对弈过程中，走出了一个“活四”，那么，对手的任何应对都太晚了，因为攻击者的下一步走子有两个空

白的位置可以取胜。对于“活三”，攻击者的威胁是下一步走子可以形成一个“活四”，因此，即使“活三”这种威胁需要两步才能获胜，但是它也必须马上应对。如果“活三”的两侧都可以形成“活四”，那么就有两种应对的走法，都是直接走在攻击子的相邻位置(图 5-2c)；如果“活三”只有一侧可以形成“活四”，那么可以有三种应对的走法(图 5-2d)。另外，对于“断三”而言，也可以有三种应对的走法（图 5-2e）。

一个棋手若要在对手任何可能的应对情况下取得胜利，就必须形成双重威胁（可以是一个活四或者是两个单独的威胁）。在大多数情况下，若要形成双重威胁，通常是一系列连续进行的包含若干威胁的走子。我们将上述形成（致胜的）双重威胁的一系列威胁走子序列称之为致胜威胁次序。致胜威胁次序中每一次威胁走子都迫使防御方应对威胁。因此，对于防御者来说，机会是有限的。

在图 5-3 中，棋盘的形式表明，黑方可以通过只包含冲四的制胜威胁次序取得胜利。由于冲四必须马上应对，全部的走子次序对于白方而言是唯一的也是被迫的。黑方走黑 1 后，白必须在白 2 应，接下来的黑 3、白 4、黑 5、白 6、黑 7、白 8、黑 9、白 10、黑 11、白 12、黑 13、白 14、黑 15，以上黑方每一步走子，都形成冲四，对于白方，都只能有唯一的应对，至黑 15，黑方已经形成活四，白方下在白 16，无济于事，黑 17 赢。

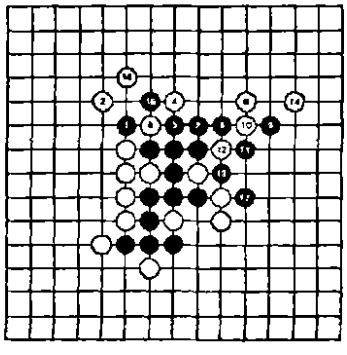

图 5-3 包含冲四的致胜威胁次序

图 5-4 中，棋盘的形式表明，黑方可以通过包含“活三”的致胜威胁次序取得胜利，尽管这一过程曾被白方的“冲四”干扰过。正如前面已经介绍过的，白方每次走子时，他可选择的机会是有限的。在此对弈过程中，黑首先走黑 1，形成“断三”，白方被迫走白 2，然后黑 3，形成“活三”，白走白 4，黑再走黑 5，至此已形成两个“断三”，尽管这时白方又走了白 6，形成冲四，黑方走黑 7，白走白 8，黑方应对黑 9，白方所能形成的威胁已经没有了，所以最后，白方的失败仍然是不可避免。我们认为黑方精确的致胜威胁次序取决于白方对于“活三”的应对。因此，在这种情况下，给出“致胜威胁树”的概念。

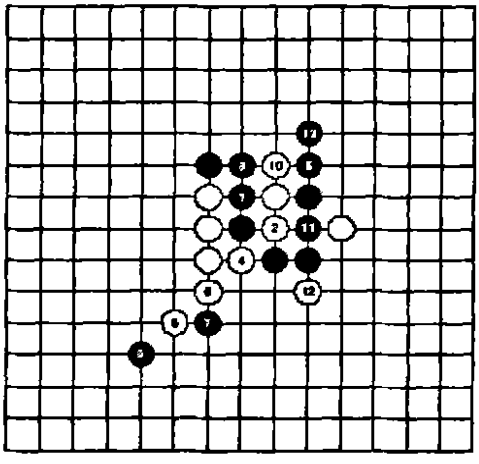

图 5-4　包含活三的致胜威胁次序

5.2.2 人类专业棋手的分析

人类专业棋手的知识和经验对于我们设计一个表现优异的五子棋系统提供了丰富而翔实的素材。当我们观察一个专业棋手对局的时候，我们会很清楚的发现，专业棋手们能够很快的在棋盘上错综复杂的形势中，找到形成致胜威胁次序的那一部分。这些致胜的次序典型的在 5 到 20 步的范围内。

人类专业棋手按照下面的步骤可以很快的发现致胜威胁次序。

(1)当棋盘上棋子的分布似乎对进攻方有利时这一部分就被挑选出来。然后判断这些进攻棋子是否能够足以形成致胜威胁次序。这一步的决策是依赖于“感觉”的，来自于长期地对棋子模式判断的经验的积累。

(2)考虑各种各样的威胁，特别是与已经在棋盘上的其他进攻棋子相关的威胁。对手的应对走法通常可以忽略。

(3)一旦在各种变化中，进攻方发现可以将己方的棋子有机的结合起来形成一个双重的威胁，就进行研究防御者怎样才能将这个潜在的致胜威胁次序瓦解。无论对手是否有一种以上的应对，都要检查上述的威胁次序是否在各种情况下都能起作用。另外，还要研究对手是否能够形成一个或更多的冲四来进行反击。

(4)只要在同一个威胁次序中，有些变化不能取得胜利，就要检查在另外的威胁次序中其余的局势变化能否取得胜利。

(5)在实战中，一个致胜的威胁次序常常包含一个简单的变化，这取决于对手的应对走法。

特别地，搜索空间的大小在一方第一次搜索时就能显著的减少。只要发现了一个潜在的致胜威胁次序，就要研究对手应对的影响。当出现一个致胜威胁次序时，就可以只考虑进攻者的走法，而不必考虑对手的应对走法。只要对手不能破坏这个威胁次序，那么对手的一些可能的冲四也可以忽略而不会有任何影响。

在没有致胜威胁次序的对局情形时，走子的优先考虑是尽可能的增加形成威胁的机会，或者无论对手如何走子，己方的应对必须是尽可能的减少对手形成威胁的机会。人类棋手对局势发展的评估基于两个方面：(1)直接计算各种可能性。比如说对手对当前局面的某一部分没有应对。(2)一种所谓的好棋形。比如说，当棋子结合得很好时就是一个好棋形。

5.2.3 计算机程序的策略

通过对国内外已有的五子棋程序的研究发现，最强的五子棋程序使用(各种)Alpha-Beta 搜索算法。在各种局面中，只评估有限数目的最可能的走法，走法选择是基于历史启发函数的。在搜索时，当发现获胜（或失败）的情形时，或者是当搜索达到预先设定的深度时搜索就结束，然后就执行静态评估函数。若一些中间节点不是希望的搜索节点时有些程序也采用向前剪枝的方法。当局势产生威胁时，防守者只有有限的应对。最强的程序可以搜索到 16 层的深度。在对局的开局阶段，程序往往使用开局库。

5.2.4 人与计算机

很显然，人和计算机都要花费时间去搜索致胜的威胁次序。人类使用乐观的搜索和验证方法。计算机使用搜索树技术。对于后者，超强的计算能力可以弥补大量的额外工作和对静态评估函数的频繁调用。而计算机对局势的评估相比人类明显的处于劣势。人工智能研究的一个挑战就是研究人类如何利用聪明的方法而变得更加智慧，而不是利用繁重的搜索来提高人类的博弈能力。这个目标可以通过将人类复杂的方法形式化然后应用到大量的计算中来实现。

在图 5-5 中，我们给出了一个极端的例子。黑方有 16 个点可以形成冲四。全部地，黑方有 $8! \times 2^8 \approx 10,000,000$ 种情况可形成一系列的 8 步走子（8 次走子必须应对），当使用置换表的搜索算法时，仍然有超过 6000 种的变化。但是，对于人类棋手，局势是非常的明显，其中没有任何一种变化会形成致胜威胁次序。人们知道形成以上全部的威胁也没有任何价值。

在图 5-6 中，给出的是与图 5-5 相似的情形。黑方现在有致胜的威胁次序。

可以想象一个五子棋程序能够在很长的搜索中发现这个致胜威胁次序。它的威胁树可能会执行棋盘上毫不相干部分的威胁而产生多余的分支。

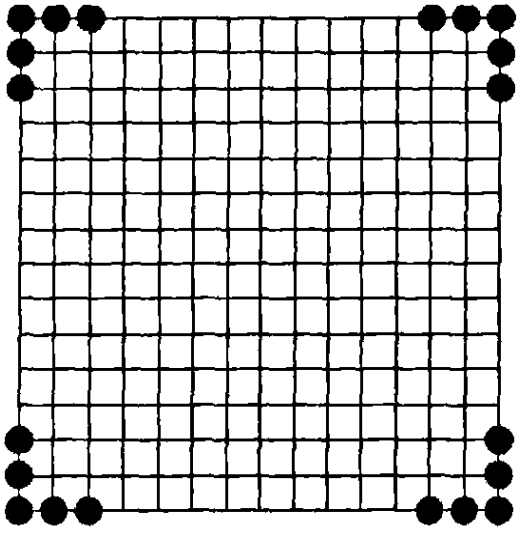

图 5-5 存在无用威胁的棋盘局势

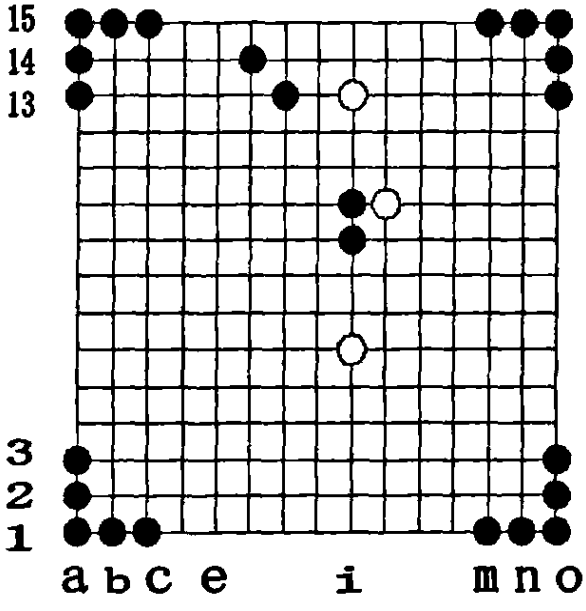

图 5-6 存在真正威胁的局势

我们断言，人类专业棋手在搜索致胜威胁次序时，常规的搜索树算法不能有效的模仿他们。下面我们提供一个模型试图实现人类的搜索策略。这个模型需要引入威胁空间搜索技术。

5.2.5 威胁空间搜索

一个致胜威胁次序包含许多威胁。因此，我们研究的重点集中在威胁空间上。威胁空间的大小经常包含数百万个节点。因此，要重点解决两个问题（1）减少空间的大小（2）尽可能高效的搜索剩余空间。

威胁空间实质性的减少依赖于防守者面对威胁时应对的决心。从 5.2.1 节，我们知道：形成冲四之后，有一种可能的应对；形成活三之后，有两种或三种可能的应对。比如说，当形成一个活三之后，我们考虑所有可能的应对，那么局面变化的数目就会迅速增加。但是，从 5.2.2 节，我们知道，若对手没有正确的应对，人类专业棋手通常能够发现致胜威胁次序。这个想法可以用来减少搜索空间。现在，我们不是在对手的应对中进行选择，而是允许对手在同一局面可以走所有可能地走法。如果我们仍然能够发现致胜威胁次序，那就可以确信对手的应对是无关紧要的。这种方法的缺点是当某一局面中存在致胜威胁次序时却不能真正发现。

总之，我们减少搜索空间到只搜索一个攻击走法空间（威胁），它们都与全部的直接防守走法相对应。我们引入 6 个概念来详细描述威胁空间搜索，下面是他们的定义：

定义 1 攻击点：一个威胁的攻击点是攻击方所走的位置。

定义 2 应对点：一个威胁的应对点是防守方所走的位置，来应对当前的威胁。

定义 3 协作点：除去攻击点之外，能够形成威胁的其它的位置。

定义 4 威胁 A 依赖于威胁 B：如果威胁 A 的协作点是威胁 B 攻击点。

定义 5 威胁 A 依赖树：是以 A 为根节点，仅包含依赖节点的一棵树。也就是说，每一个节点 J 的孩子节点是依赖于节点 J 的威胁子树。

定义 6 冲突：两棵独立的威胁树 P 和 Q 是冲突的，当包含威胁 A 的依赖树 P 和包含威胁 B 的依赖树 Q 满足下列情况之一（1）威胁 A 攻击点是

威胁 B 应对点（2）反之亦然（3）威胁 A 的应对点也是威胁 B 应对点。

例如：在图 5-6 中，黑方走子 e15 形成冲四（攻击点是 e15，应对点是 d15，协作点是 a15,b15,c15)。在走了 e15 和 d15 之后，黑方走 i11 又形成冲四（协作点是 e15，f14 和 g13)，因此这个冲四的攻击点 i11 是依赖于上一个冲四的攻击点 e15，因为 e15 是攻击点 i11 这个威胁的协作点。

威胁 i11 的依赖树是这样一棵树：它的根节点是“攻击点是 i11 的一个冲四”，根节点唯一的子节点是攻击点是 e15 的一个冲四。由于这个最后的威胁包含的协作点都已经在当前的棋盘上，它不再依赖于任何其它的威胁。

对于定义 6 的一个非常清楚的例子就是攻击点是 e15 威胁（应对点是 d15）与攻击点是 d15 威胁（应对点是 e15)，因为这两个威胁不能在同一个致胜威胁次序中被一起执行。因此，这两个威胁是冲突的。如果威胁空间有很大的依赖树，就需要做很多的工作去检查这些依赖树是否是冲突的。但是，这些额外的工作要比普通的研究更重要。

现在威胁空间搜索可以用下面两个原则来描述：

原则一 威胁 A 如果独立于威胁 B，则威胁 A 就不能出现在威胁 B 的搜索树中。

原则二 在威胁空间搜索树中，只包含攻击方的威胁。当发现一个潜在的致胜威胁次序之后，就要判断这个威胁次序是否能够应对任何的反击。

在表 5-1 种，我们归纳了图 5-6 所示局势的威胁空间搜索树

在表 5-1 的每一行中，包含深度，威胁类型，攻击点和应对点。在搜索树中，存在 18 个独立的节点：在棋盘的四个角上有 16 个冲四，在 i7 至 i12 的范围内有两个活三。其中有 16 个威胁（15 个冲四 1 个活三）不能形成任何新的威胁。因此这 16 个节点在威胁空间搜索树中是终端节点，因为威胁空间搜索的第一条原则要求搜索树中的孩子节点必须依赖于它的父亲节点。只有两个威胁在后来的威胁中能形成可利用的攻击点，下面我们讨论

这两个威胁：

表 5-1　图 5-6 的搜索树

深度	威胁类型	攻击点	应对点
1	冲四	l15	k15
1	冲四	k15	l15
1	冲四	e15	d15
2	冲四	i11	h12
3	活四	i8	i7
2	冲四	h12	i11
1	冲四	d15	e15
1	冲四	o12	o11
1	冲四	o11	012
1	冲四	a12	a11
1	冲四	a11	a12
1	活三	i11	i7,i8,i12
2	冲四	h12	e15
2	冲四	e15	h12
3	五连	d15	
1	活三	i8	i7,i11,i12
1	冲四	o5	o4
1	冲四	o4	o5
1	冲四	l1	k1
1	冲四	k1	l1
1	冲四	e1	d1
1	冲四	d1	e1
1	冲四	a5	a4
1	冲四	a4	a5

黑走攻击点 e15 后（应对点 d15），又形成两个新的威胁，也就是在 i11

和 h12。攻击点 h12（应对点 i11）不能形成新的威胁。黑走攻击点 i11（应对点 h12），能够在 i8 位置形成一个活四。由于活四能够保证取得胜利，因此一个潜在的致胜威胁次序就发现了。

黑走攻击点 i11（应对点 i7，i8，12），能够形成两个新的冲四，也就是在 h12 和 e15。攻击点 e15（应对点 h12）能够在 d15 形成五连。因此第二个潜在的致胜威胁次序发现了。这个空间搜索树只包含 24 个节点，而全部的可能威胁次序都找出来了。

图 5-6 的举例中，通过搜索，发现了两个可能的致胜威胁次序，而且每一个随后的威胁都是由当前棋盘上的棋子决定的：每一个新的攻击点起源于前面的威胁。然而，在某些局面里，两个或三个独立威胁的攻击点可以合并形成新的威胁。在这种情况下，应该连接独立的节点。合并的结果能够成为新威胁的祖先节点。我们注意到只有当那些攻击点可能形成新的威胁时连接独立的节点才有意义。也就是说它们必须在一条直线上，并且距离很近有可能形成五连。

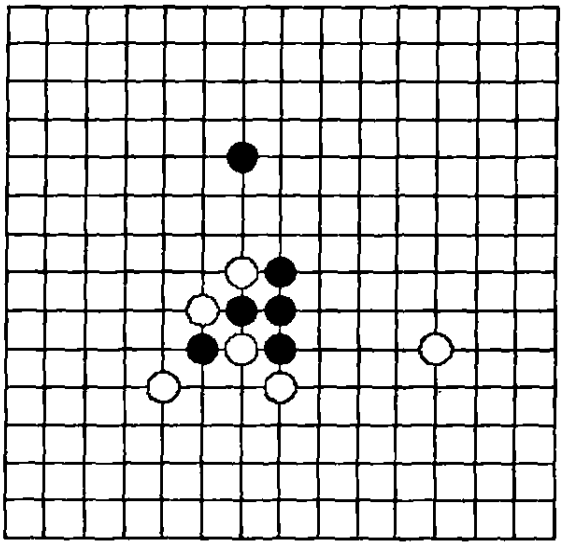

图 5-7 合并攻击点而致胜

图 5-7 的局势表明，黑方可以通过合并各自独立威胁的攻击点获得胜利。表 5-2 中，我们初始化威胁空间搜索树。搜索树并不包含活四或者五连。这意味着没有发现潜在的致胜威胁次序。

表 5-2 初始化图 5-7 的搜索树

深度	威胁类型	攻击点	应对点
1	冲四	h10	h9
1	冲四	h9	h10
1	冲四	j10	il9
1	冲四	i9	j10

对于表 5-2 中冲四的攻击点，我们可以将攻击点建立 6 个合并。我们注意到 h9 和 h10 都是攻击点并且在一条直线上，但是它们的依赖树是冲突的。因此，h9-h10 的合并不能添加到威胁空间搜索树中。基于同样的原因，i9-j10 也不能添加到搜索树中。H9-j10 的合并并不在一条直线上，其余三个攻击点的合并在一条直线上，并且它们的依赖树并不冲突。它们是：h10-i9，h10-j10 和 h9-i9。在表 5-3 中，我们给出了这 3 个潜在的有价值的合并。对

表 5-3 合并图 5-7 的威胁

深度	威胁类型	攻击点	应对点
1	冲四	h10	h9
2	与 i9 合并	i9	j10
3	活四	f12	e13
2	与 j10 合并	j10	i9
1	冲四	h9	h10
2	与 i9 合并	i9	j10
1	冲四	j10	i9
1	冲四	i9	j10

于每一个合并，如果可能，我们按照威胁空间搜索的原则建立了新的搜索

树。结果表明 h10-i9 的合并可以在 f12 建立一个活四。因此，威胁{h10,i9}的合并及随后的 f12 可以导致潜在的致胜威胁次序。

以上两个例子清楚的说明了威胁空间搜索的搜索过程。现在给出一个一般的描述：对于一个给定的对局局势，生成所有独立威胁的搜索树，无论在哪一个分支的叶子节点，若存在活四或者五连，就要检查这个潜在的致胜威胁次序能否成功。如果没有任何一种变化可以使攻击者获得胜利，就要对全部独立的节点进行合并。整个过程一直重复进行直到发现致胜的威胁次序，或者再也没有新的潜在的有价值的合并就结束。概括上述思想，威胁空间搜索就是一个线性化的搜索过程。

5.2.6 算法优化后效果的检验

通过对搜索算法的优化与增强，主要是引入了置换表的搜索和威胁空间搜索算法，程序的表现和对弈水平有了明显的提高，对弈的结果可以从两个方面得到验证。第一种情形：本程序与一般的人类棋手，通常是业余的五子棋爱好者对弈时，若让计算机执黑先行，计算机获胜的概率非常高。第二种情形，作者在网上下载了两个常见的五子棋对弈的程序（快乐五子和黑石 BlackStone），对弈的结果，本程序都获得了胜利。下面是与两个五子棋程序对弈的结果。

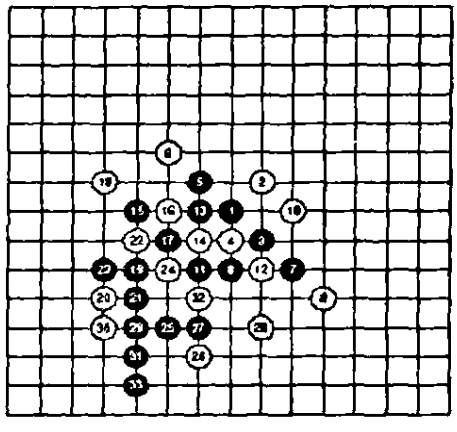

图 5-8 本程序黑先胜 对 快乐五子

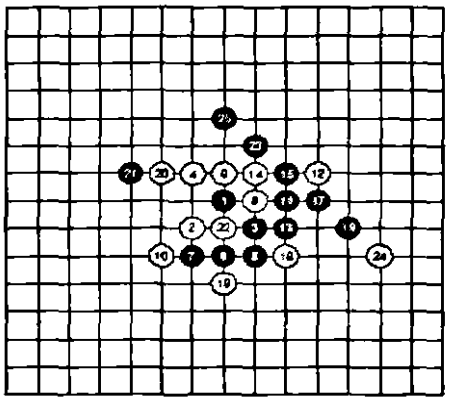

图 5-9 本程序黑先胜 对 BlackStone

5.2.7 威胁空间搜索的不足

威胁空间搜索，在对弈刚开始，还没有产生威胁时，无法发挥作用。对白子来说，五子棋规则改变后，黑子只注重攻击的话，容易而威胁空间搜索，在对弈刚开始，还没有产生威胁时，无法发挥作用，掉入白子的陷阱中而下出禁着输了该棋局，因为当有的情况下，白子做防守的可能有两种以上，例如在防守活三的情况下，若是黑子只考虑对自己有优势的情况时，不单落入陷阱中，还有可能让白子作防守后反而盘面已经对黑字不利。而且当搜索深度不够时，黑子找不到它的 Winning Sequence，而退而求其次所找出来的路径就并非最佳的路径了，为了避免这个一时之选造成对本身的不利，我将再作进一步的研究。

5.3 小结

本章讨论了基于哈希表的置换表搜索法，介绍了如何利用 Zobrist 哈希技术计算哈希值来实现置换表搜索，置换表的搜索是直接在前述算法的基础上的完善和加强；重点阐述了威胁空间搜索，讨论五子棋知识以及它们的应用，给出了致胜威胁次序概念，详细地描述了威胁空间搜索过程；

两种搜索算法的结合，有效的提高了五子棋对弈中最佳走法的搜索问题，搜索算法得到了明显的优化，较圆满的解决了五子棋人机对奕问题，程序的表现和对弈水平有了明显的提高，并对算法优化后的效果进行了检验。

第6章 结论

人工智能的中心任务是研究如何使计算机去做那些过去只能靠人的智力才能做的工作。它有诸多的研究领域：专家系统、决策支持系统、机器学习、机器视觉、自然语言理解等等，计算机博奕也是其中之一，计算机博弈是人工智能研究的一个重要分支，它的研究为人工智能带来了很多重要的方法和理论，产生了广泛的社会影响和学术影响。

本文研究了五子棋在计算机中的表示问题、计算机中存贮棋局和识别下棋次序，局势状态变化及局势特征、走法产生等方法。研究了 Alpha-Beta 剪枝的改进算法 NegaScout 算法，研究了博弈树的极小极大搜索技术及在此基础上的 Alpha-Beta 剪枝过程和剪枝优化问题。实现将候选的后继节点按位置邻近顺序排序，使剪枝过程得到优化。根据根据五子棋的特点，提取棋局局势的若干特征，对这些特征赋加权分，并对整个棋局进行特征统计，采用一个线性函数求得棋局的总估计分值。

本文对五子棋博奕的专业知识进行认真的整理，针对五子棋博奕规则简单、局势判断清楚的特点，对五子棋常见的开局、定式及其后的对局做大量细致的统计分析，利用五子棋对奕的内在规律和专业知识，对搜索算法进行了的优化与增强，给出适合该棋类对奕的更佳的搜索算法：基于哈希表的置换表搜索算法、威胁空间（Threat Space Search）搜索算法。两者的结合，可以有效的提高五子棋对弈中最佳走法的搜索问题。使用了置换表搜索算法和威胁空间搜索算法，有效的提高了五子棋对弈中最佳走法的搜索问题，搜索算法得到了明显的优化，较圆满的解决了五子棋人机对奕问题，程序的表现和对弈水平有了明显的提高。结果表明，以上算法在五子棋的程序设计中是成功的

参考文献

[1] Nils J. Nilsson，郑扣根，庄越挺译，潘云鹤校. 人工智能. 北京：机械工业出版社. 2000

[2] 孙宏伟，何丰泉. 五子连珠棋初步. 哈尔滨：黑龙江科技出版社.1999

[3] 王鲁明，戴汝为. 在计算机围棋中形象思维的研究. 自动化学报，1997，23(4)：564-566

[4]Kierulf, Anders. Smart Game Board: A Workbench for Game-Playing Programs, with Go and Othello as Case Studies: [Ph.D. Thesis No. 9135]. Switzerland: Swiss Federal Institute of Technology (ETH) Z"urich, 1990

[5] 蔡自兴，徐光祐. 人工智能及应用. 北京： 清华大学初版社，1996

[6] 肖齐英，王正志. 博弈树搜索与静态估值函数，计算机应用研究. 1997,4

[7] A.Reinefeld, An Improvement of the Scout Tree Search Algorithm, ICCA Journal, Vol. 6,No. 4, 1983

[8] J. Schaeffer, Distributed Game-tree Searching, Journal of Parallel and Distributed Computing, Vol. 6, 1989

[9] A. Plaat, J. Schaeffer, W. Pijls, A. de Bruin, Best-first Fixed-depth Minimax Algorithms, Aritificial Intelligence, 1996

[10] G. C. Stockman, A Minimax Algorithm Better than Alphabeta? Aritificial Intelligence, Vol. 12, No. 2, 1979

[11] H. J. Berliner, C. McConnell, B* Probability Based Search, Artificial Intelligence, Vol. 86, No. 1, 1996

[12]L. Victor Allis: "Searching for Solutions in Games and Artificial intelligence" Part I. PhDThesis, September 1994, ISBN 90-9007488-0. http://hoggy.virtualave.net/y.bishop/pubs/ps78.1.zip

[13]Martin Schmidt: "Temporal Difference Learning and Chess" Part I. Aarhus

University,

[14] 孙宏伟，何丰泉. 五子连珠棋初步. 哈尔滨：黑龙江科技出版社. 1999

[15] Jay Burmeister, Janet Wiles. Computer Go Tech Report. In: Proceedings of the

Second Games Programming Workshop, Japan, 1999

[16] B. Bouzy, T. Cazenave. Computer Go: An AI-Oriented Survey. Artificial Intelligence, October 2001, 132(1): 39-103

[17]Herik, H.J. van den, and Allis, L.V. (eds.) (1992). Heuristic Programming in Artificial Intelligence 3:the third computer olympiad. Ellis Horwood Ltd., Chichester, England.

[18]Uiterwijk, J.W.H.M. (1992a). Go-Moku still farfrom Optimality. Heuristic Programming in ArtificialIntelligence 3: the third computer olympiad(eds. H.J. van den Herik and L.V. Allis), pp. 47-50. Ellis Horwood Ltd, Chichester, England.

[19] 张玉志. 计算机围棋博弈系统 [博士学位论文]. 北京：中国科学院计算技术研究所 1991

[20] Hall M.R. and Loeb D.E. (1992). Thoughts on Programming aDiplomat. Heuristic Programming in Artificial Intelligence 3: thethird computer olympiad (eds. H.J. Van den Herik and L.V. Allis), pp. 123-145. Ellis Horwood Ltd, Chichester. (6)

致谢

我的研究生学习即将结束，在硕士论文完成之际，我要向所有关心、支持和帮助过我的人们表示最诚挚的谢意。

首先感谢我的导师刘新平和孙士明两位老师，在研究生学习期间，他们渊博的专业知识、严谨的治学作风和亲切和蔼的师德使我在为学为人之道上受益终身。治学严谨、待人真诚，襟怀坦荡，高屋建瓴的学术眼光、对事业孜孜不倦地追求和勤奋不辍的精神使我受益匪浅，是我终生学习的榜样，在此向导师致以最诚挚的谢意和最美好的祝愿！

在读研期间，有幸能够聆听各位任课老师的授课，他们渊博的知识极大的拓展了我的知识视野和专业水平，在此向各位老师致以最诚挚的谢意！

感谢我的同事董红安老师对我的帮助。

感谢我的家人在学习和生活上给予了无微不至的关心和鼓励，感谢我的妻子为照顾家庭和孩子而付出的心血和汗水。

最后，感谢所有曾给予我鼓励、支持和关心的亲人、朋友和同学。

个人简历、在学期间的研究成果

王志水，男，生于 1965 年 3 月 9 日，1986 年 7 月毕业于滨州师范专科学校，1992 年 7 月取得华东师范大学本科学历，1986 年 7 月至 2002 年 1 月任教于滨州卫生学校，2002 年 2 月至今在滨州职业学院计算机信息科学系任教，2003 年 10 月考入中国石油大学（华东）攻读工程硕士学位，专业为计算机技术，研究方向为人工智能及应用。

[1] 王志水．自动控制升降旗系统．《电脑知识与技术》．2007(1)