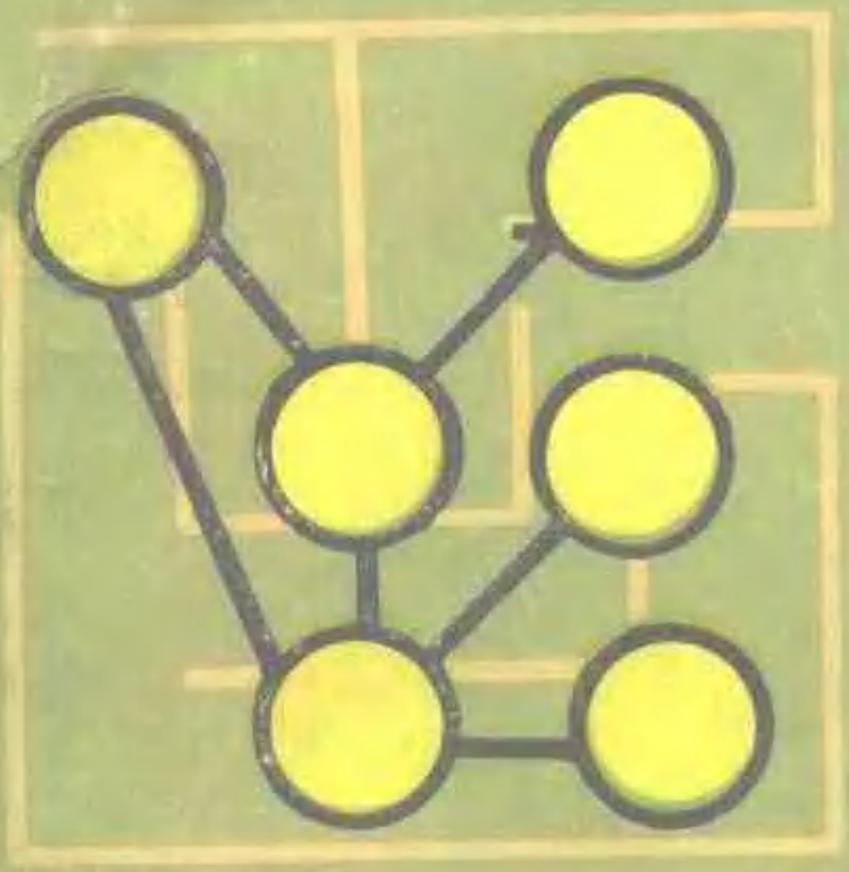


人工智能入门

〔美〕N. 格雷亨姆

机械工业出版社

人工智能入门

[美] N. 格雷亨姆 著

欧阳文道 何 川 计小玲 译

杨忠祥 校

机械工业出版社

本书是一本人工智能知识的入门性读物，叙述了人工智能的基本原理、推理思维及其理论基础－计算机逻辑。文中贯以有趣的实例，如传教士与野人、迷宫问题、梵塔、计算机下棋和机器人动作规划等。文字生动，深入浅出，为进一步阅读其它专著提供了基础。

本书可供科技人员、大专院校和中等学校师生、管理人员以及具有中等以上文化程度的人员阅读。

Artificial Intelligence

Making machines"think"

Neill Graham

TAB BOOKS, 1979

* * *

人工智能入门

[美] N. 格雷亨姆 著

欧阳文道 何 川 计小玲 译

杨忠祥 校

*

责任编辑：林佩珊

*

机械工业出版社出版（北京阜成门外百万庄南里一号）

（北京市书刊出版业营业许可证出字第117号）

中国铁道出版社印刷厂印刷

新华书店北京发行所发行·新华书店经售

*

开本 787×1092 1/32·印张 9¾·字数 211千字

1988年5月北京第一版·1988年5月北京第一次印刷

印数 0,001—5,100·定价：3.10元

*

ZSBN 7—111—00757—3/TP·54

译　　序

人工智能(或称机器智能)是近十年来发展迅速的一门新兴学科。它研究如何利用计算机或机器去代替人的一部分脑力劳动。工人智能与控制论、心理学、仿生学、自动化等学科有着密切的联系。它的研究还具有重要的哲学意义。

近几年来我国对人工智能感兴趣的人日益增多，但尚缺乏这方面的参考书籍。翻译本书的目的是填补这方面的空白。

本书原有两个标题：主题为“人工智能”，副题为“使机器思维”，但考虑到它是一本入门性的科技著作，故把书名译成“人工智能人门”。

本书的特点是结合实例来阐述人工智能的基本原理。如结合著名的“传教士与野人”、“梵塔”和“迷宫老鼠”等问题来说明问题求解的策略与方法。这些都是西方人工智能著作中常引用的例子。但要注意“传教士与野人”问题中明显地含有种族歧视的偏见，把原始部落民族看成是一群吃人野兽，因此我们不妨把这个问题改成别的名称，但问题求解的策略还是一样的。此外本书结合机器人的行动的控制问题说明规划原理和复杂动作的分级规划方法；结合博弈问题以西洋跳棋和国际象棋为例介绍各种博弈策略所使用的树状搜索法与启发式方法。值得注意的是，启发式方法可以说是现代人工智能技术的核心。

本书介绍了人工智能进行推理思维的理论基础——计算

机逻辑，并以实例说明其应用，还介绍了几种在计算机上表达知识的方法以及智能程序语言 LISP 和初步的智能程序设计技术。

作为入门性的著作，本书不可能全面深入地说明人工智能的所有问题及其在各方面的应用，但可为阅读人工智能的其它专著提供基础。作者对一些理论概念和技术性问题力求避免引用高深的数学工具和计算机理论，以利于读者理解和接受。因此本书对人工智能知识的普及无疑是有益的。

本书的翻译工作，序言部分由校阅者兼任，第一、二、四、五、九、十三章由欧阳文道同志承担，第六、七、十、十一、十二章由何川同志承担，第三、八、十四章由计小玲同志承担，最后由杨忠祥同志校阅定稿。

人工智能是一门新兴学科，所以有关术语在国内尚无统一译法，因此书末编有汉英名词对照表，以供参考。

本书全部译文虽经统一审校，修改与整理，并在译校中改正了原书中的若干错误，但因译校水平有限，可能还有错误与不当之处，敬希读者不吝指正。

序

还有不太久以前，计算机的程序设计带来的效果使人们仿佛置身于魔术表演的奇境之中，而懂魔术技巧的人只有那些令人感到神秘莫测的少数职业演员，这些人经常为了谋生才去发挥他们的表演才能。由于计算机已逐渐普及深入到人们的生活之中，许多行业的工作人员也感到他们和计算机这种奇妙机器的关系日益密切，而且还经常有程序设计人员为他们表演“魔术”。

但是，由于集成电路制造部门在显微工艺的奇迹中研制成功了功能日益繁多的微型电路，从面使计算机不久就成为放在写字台上使用的微型计算机，完全不象它们所淘汰的多机柜组合式计算机了。那么，程序设计人员的情况如何呢？由于功能强而价格便宜的新一代计算机已星罗棋布于世界各地的工作场所，因而程序设计工作已成为许多计算机使用人员所必须掌握的技能，而不单是程序设计人员的职责了。

在计算机获得气速增长的最近时期，许多业余爱好者也都在设计制造自己个人用的微型计算机并自行设计程序。一度为专职人员垄断的计算机程序设计工作，现在已拥有大批半专业和业余的程序设计人员。这批新生力量不久即发现常规的程序设计方法有其局限性。现代计算机的应用越来越需要高级的程序设计技术。这种技术采用了一些大学和新技术开发公司在人工智能方面的研究成果，利用计算机做某些工作。这些工作如果由人来完成的话，就需要用人的智能才行。

人工智能程序最早的应用之一就是让计算机做俄译英的翻译工作。1957 年 10 月苏联把第一颗人造卫星送进了环绕地球的轨道。这使许多美国人认识到：俄国科学家所掌握的英语知识已使大多数不惜俄文的美国科学家处于不利地位。俄国人能够阅读美国的科技文献，而美国却无法阅读俄文书刊。于是不久以后他们就用计算机费劲地做起俄译英的工作，但只能译出 80%。其余 20%由于太复杂，使人感到计算机翻译没有多大希望，因而放弃了当时在这方面所作的努力。但是在今天，语言学家与程序设计专家共同合作以改善自然语言的计算机翻译能力。甚至一些业余爱好者也对自然语言翻译作出了贡献。

人工智能的另一个领域——计算机下象棋已经相当普及，以致出现一年一度的国际计算机象棋锦标赛。计算机在比赛中相互“斗智”。然而目前的国际象棋大师还是比计算机这个对手更高明。他们认为计算机还缺乏长远规划的能力。但是未来很可能有更高级的人工智能技术与更高速、更大容量的计算机。它们结合在一起就会克服上述弱点。到了那时，有关的比赛就只有在计算机之间进行了。

虽然在国际象棋之类的游戏中计算机还能与人相抗衡，但在模式识别方而计算机就比人差远了。模式识别是人工智能的一个实践领域。识别物体(多半是为了形成机器人视觉)也需要有非常高级的程序，但这方而的程序设计还处在幼年时期。

促进计算机及其程序发挥最大效率的努力，已经冲破了某些清规戒律所设置的现代禁区。这种狂热的进联精神在实质上似乎就是人类生活本身的缩影。我们也许是从科学幻想小说作者那里感染到了对未来世界的热烈希望。在未来世界

里，日常事务工作由机器人来承担，而具有超凡能力的电脑将会尽力为我们解决一些连我们自己也梦想不到的问题。但愿我们在仰望满天繁星，憧憬未来美好前景的时候，还能为改善地球上目前人类的命运作出贡献。

作者在书中对这个正在繁荣成长的科学领域作了入门性的介绍。不论你的兴趣是在智能博弈游戏、自然语言处理、机器人、数理研究，还是在工程技术方面，人工智能都将证明它是程序设计知识宝库中的一种有力工具。

N. 格雷亨姆

目　录

第一章　什么是人工智能…… 1
一、人工智能的应用…… 1
二、为什么我们需要更聪明的计算机…… 7
三、算法…… 8
四、人工智能程序设计原理…… 11
第二章　问题求解的策略…… 17
一、状态、操作和目标…… 17
二、状态图…… 27
三、计算机的表达方法…… 36
第三章　状态图搜索…… 42
一、搜索树…… 42
二、宽度优先搜索…… 48
三、深度优先搜索…… 52
四、有序搜索…… 56
五、其它启发技术…… 69
第四章　子问题、子目标和计划…… 71
一、子目标和计划…… 72
二、编写计划…… 74
三、搜索与/或树…… 80
第五章　分级计划和过程网…… 87
一、分级计划…… 88
二、过程网…… 90
三、监督执行…… 98
第六章　博弈游戏程序：树状搜索…… 103
一、博弈树…… 103
二、博弈策略…… 107

三、最小最大值方法…… 109
四、终局和静态评价函数…… 111
五、深度优先的最小最大值评价方法…… 117
六、α-β 方法…… 123
第七章　博弈游戏程序：启发式方法…… 126
一、棋局的表达…… 127
二、控制博弈树的规模…… 134
三、评价函数…… 138
四、规划…… 143
第八章　模式识别与感知…… 145
一、定义…… 146
二、特征空间、区域和原型…… 152
三、图象分析…… 157
第九章　机器人…… 165
一、效应器…… 170
二、感觉器…… 172
三、控制…… 174
第十章　计算逻辑：命题和谓词…… 176
一、命题逻辑…… 178
二、谓词逻辑…… 187
第十一章　计算逻辑：归结法…… 196
一、归结原理…… 196
二、转换成子句形式…… 202
三、一致化…… 205
四、小结…… 210
第十二章　知识表达法…… 211
一、基于谓词逻辑的表达法…… 211
二、知识表达法与规则…… 221
三、特性表…… 228
四、语义网络…… 231

五、框架…… 232
六、产生式系统…… 236
第十三章　自然语言处理…… 239
一、句法、语义和转移网络…… 240
二、格语法…… 248
三、概念依赖理论…… 250
第十四章　LISP 语言…… 253
一、原子和表…… 253
二、用表来表示信息结构…… 255
三、LISP 程序设计的原理…… 260
附录一　参考文献…… 288
附录二　汉英名词对照表…… 291

第一章　什么是人工智能

人工智能是计算机科学的一个分支，它用计算机程序来做某些工作，这些工作如果让人来做的话，就需要智能。

为什么要说“如果让人来做”这句话呢？因为如果计算机用程序做了某事，人们可以完全否认它需要什么智能，至少是认为计算机这样做的方式不是什么智能。人们可以无休止地辩论：计算机的哪些技巧可以构成智能行为，哪些不能构成智能行为。我们不妨避开这些辩论，仅仅宣称我们试图让计算机用程序来做某些工作，如果让人来做就非要用人的智能不可。至于说，能不能把做这种工作的计算机也叫做“有智能”的，那就由你了。

象计算机科学中的其他一些事物一样，“人工智能”也常常用它英文名称“Artificial Intelligence”的开头两个字母“AI”来表示。

一、人工智能的应用

和人工智能有密切关系的应用领域有：

1. 问题求解
2. 自然语言处理
3. 感知和模式识别
4. 信息存储和检索
5. 机器人控制
6. 博弈游戏
7. 自动程序设计

8. 计算逻辑学

(一)问题求解

我们给予计算机的大多数任务都可以看作是"问题"。我们常常希望用计算机进行必要的操作来达到预定的目的。比方说,我们有一个文件,上面有规定的每小时工资额和公司每个职工每星期的工作时数,我们的目的是要发给每个职工一张工资单。借助于工资单程序,计算机就可以达到这个目的,解决我们所要解决的问题。

非人工智能程序设计所解决的问题,只能是相当有限的一类问题。例如工资单程序,在处理工资单问题时会稍有变化,例如职工类别不相同,扣除金额数不一致等等;但是我们绝不能期望用工资单程序来解决诸如下棋问题,为飞向月球的宇宙飞船制订计划步骤的问题等。

人工智能程序则和这一类为专门目的服务的程序相反,它的设计是尽可能为一般的目的服务的。这种一般目的程序,只要你给出问题和与解决该问题有关的知识,它就应能制定出解决问题的方法,这点可以说和人做的差不多是相同的。

我们知道,传统的计算机科学在设计解决专门问题的程序时,总是遵循程序员预先给定的一步步解决问题的步骤和方法。而人工智能的程序,则自己能制定一步步解决给定问题的步骤和方法,然后执行这些步骤。

(二)自然语言处理

现在,计算机和人直接交流还不很方便,特别是在人们想要给计算机以信息,或发布命令的时候更是如此(利用"封装语句"的办法,让计算机把信息传给人,效果稍好一些)。

给计算机的命令通常总是用一些看似模糊的编码来写成,写的时候还得按照精确的规则,一点不能马虎。许多常用

的程序语言都是这种编码的恰当例子。凡是给初学者教过程序语言的人都知道，这许多规则、约束，对于刚入门者是多么不自然和难弄！尤其使人生畏的是大型计算机上的作业控制语言，哪怕是很有经验的程序员也往往会被它弄得晕头转向。

要使计算机能够在非专业人员中间推广使用，则计算机必须要能够用自然语言(即人类语言，例如英语)和人打交道；而且我们希望将来计算机不但能够"懂"书面言语，也能"懂"口头言语。

计算机除了能和其使用者方便地相互交流外，我们还希望它能够完成一些面向语言的任务：例如把一种自然语言翻译成另一种自然语言，或者是就有关专题从一本书中找出某个问题的答案等。

(三)感知和模式识别

在计算机应用中，有时需要计算机通过电视摄象机、微音器或其他感受元件来感知它周围的环境。有时甚至还要求计算机能够识别它所感知的环境中的一些重要特征或模式。

模式识别对于许多人工智能工作来说往往是一个先决条件。比方说，在求解问题的时候，有时就需要从问题的情景中抽出一些重要的模式，来获得解答问题的启示(福尔摩斯侦探小说就是以模式识别为基础求解问题的最好例子)。

(四)信息存储和检索

二十世纪以来，在医学、法律、科学、工程和其他领域内都出现了信息爆炸的现象。现在任何一个领域的工作人员都不可能在他的脑子里保存他工作所需要的全部信息。甚至于连他所需要的信息是否存在，他都可能不知道。

计算机能够为你存储大量信息。不过，在使用现有的计算机系统时，你必须把需要的信息描述得相当详细。你必须

对你所要查找的信息知道得相当清楚，以便计算机为你查找；倘若你能够把你的题目即你所要解决的问题描述清楚（当然你希望最好是用自然语言），而且让计算机去为你寻找，那该会有多么大的帮助！

信息存储和检索对于其他人工智能工作也是很需要的。比方说，问题求解程序就需要对有关问题和情况的完整知识，它甚至还需要知道过去在解决类似问题时行之有效的技巧和经验法则。怎样把这一类信息和知识变成容易利用的形式，并存储在计算机里面，这称之为“知识表达”。

（五）机器人控制

对于有些工作，例如工厂中组装机器，计算机如果仅仅知道它的周围环境情况甚至能直接感知它们，还是不够的。计算机必须能够改变它的环境，例如拿起一个零件把它装到另一个零件上去。具有这种能力的计算机就叫做机器人（如果操作机械远离计算机时，则操作机械叫做机器人，而在远处的计算机则控制机器人。利用分时方法一台计算机可以控制许多个机器人。）

机器人的一部分问题是机械方面的：现在还没有设计出和人的手臂一样灵活的机械人手或臂。这些问题应先由机械和电气工程技术人员来解决，然后计算机科学家才有条件开始工作。纵使一个机器人已经装备了适当的电气、机械附件和必要的装置，但要研制出好的计算机程序，使具体的命令能发送到这些装置上，还有大量工作要做。

（六）博弈游戏

自从十八世纪有人搞了一个下棋机器人的骗局以来，人们对制造能下棋的机器（在今天，也就是设计能下棋的计算机程序）的想法越来越着了迷。还有不少人设计了其他一些游

戏程序，如跳棋、立方(或三维)一字棋、多米诺骨牌、十五子游戏和日本围棋等程序。

博弈游戏程序是人工智能练习的很好形式，因为：

1. 棋盘和博弈规则都很容易在计算机里面表示出来。

2. 对比起来，表现规则虽则容易，但玩一些复杂的游戏如下象棋和围棋等时解决起来却很困难。要计算出一着正确的下法，绝非易事。计算机琢磨游戏的方法，和人进行游戏时的方法是十分类似的。

3. 有很多博弈游戏专家，老爱挑计算机的毛病，他们的这些批评可以帮助指出程序中哪些地方还需要改进。

4. 在经常举行的博弈比赛中，评分是按照比赛者的赛中情况来评定的。参加比赛的计算机程序，也可进行评分。这样一来，它的性能就可以直接与博弈专家相比较。

(七)自动程序设计

编写和检查复杂的计算机程序是一项非常费时的工作。要编制出一个编译程序或一个操作系统，往往需要一支程序员队伍工作好几年的时间，其结果也还有缺点。往往还有些毛病，使用户非常苦恼；等到好不容易把这些毛病找了出来，才能消除这些毛病。这就产生了现在所谓的软件“危机”。

解决这个危机的一条出路，就是让计算机能够从待解决问题的陈述中自动产生自己的程序。这样一种“编写程序的程序”，是今天编译程序和报告产生程序的一个扩展。编译程序和报告产生程序虽然也能产生程序，但它们要求人们把要产生的程序叙述得非常详尽，所以还是不太合乎理想的。

(八)计算逻辑学

有时候，我们需要证明一些事实是另外一些事实的逻辑上必然结果。例如，一个检察官必须从他手头获得的证据中

证明被告确犯有非法行为，才能对被告提出起诉。计算逻辑学就是根据事实进行演绎推理的程序设计技术。

计算逻辑学还有一个重要的应用就是应用于数学。数学家从少数被称之为公理的叙述出发，演绎出一些有趣的被称之为定理的结果。数学家必须保证在任何情况下，只要这些公理是真的，那末这些定理也就是真的。有了这样的保证，我们就可放心地在科学和工程中应用这些定理，而不必再去重复数学家已经进行过的推导。

举例说，最近计算机帮助数学家证明了长期以来非常闻名的拓朴学中的四色定律。不过，这个证明的计划还是由数学家作出的，计算机只不过是依照计划进行了繁重的、详尽的证明工作罢了。现在，我们希望计算机不但能作出详细证明，而且还能自己制定证明的计划。

计算逻辑学的另一个应用就是证明计算机的程序是否正确。这也是自动程序设计的另一个重要方面。程序员只作设计和编制程序这类创造性工作，而计算机则用来证明所作出来的程序是否正确；也就是证明计算机程序能否解决所要解决的问题。如果计算机证明它是成功的，我们就可以很放心地使用这些程序；如果计算机证明它不成功，那么计算机也会找出失败的原因，或者找出程序中的毛病所在。

还有一些人工智能程序也需要计算逻辑学，特别是那些包含有情报检索和知识表达方面的工作。在计算机程序中存贮了一些事实，可是，当求解问题的时候，人们或者该工作程序都还需要另外一些事实，这时计算机就必须从已知的一些事实中推导出所需要的事实。

二、为什么我们需要更聪明的计算机

在上一节中，我们叙述了计算机在表现人类某方面智能上的一些应用。现在我们将进一步概括说明为什么需要这样的计算机：

1. 智能计算机能使它自己适应于使用它的人的需要。对计算机往往有很多批评，即认为是我们按照机器的方式来工作，而不是让机器按照我们自己的方式来工作。所谓用我们自己的方式来工作，包括用英语或其他自然语言来说话、书写在内。人们希望计算机也具有这样的能力。

2. 机器人能够在不适合于人类的环境中工作，例如，在海底、地下矿井、外层空间等等。

3. 机器人能够做非常沉闷的和重复性的工作，而这些工作需要具有象人类感知和操作周围事物那样的能力，许多生产线上的工作就属于这个类别。

4. 计算机能存贮在我们工作和日常生活中所需要的大量信息。根据需要计算机能起到相当于教员、咨询人员、记者或者邮递员的作用。

5. 在一个复杂的世界中，往往有许多非常复杂的问题，这就要求人们有解决各种各样问题的能力。如解决经济问题、能源问题、环境保护问题，以及外交关系问题等等，对这样一些问题。智能计算机往往可以帮助我们去解决其中一些非常棘手的问题。

6. 在我们模仿人类智能的时候，可以从要模仿的这些工作中，学到很多东西。事实上，人工智能研究人员可以分为两类：一类研究人员仔细地模仿人类智能的一些方法，他们的兴趣不但是要了解这些方法，而且还要建造具有这样功能的智

能计算机；另一类研究人员的兴趣只是在于获得人类智能所达到的结果，而并不关心必须采用什么方法。

一些科学幻想计算机和机器人，例如 2001 年的 HAL；“空间探险”和“星际战争”中的 C—3PO，这些机器人往往具有很广泛的、多方面的人类智力。可是这样一些机器人现在仍不过是科学幻想而已。因为对人类智能的许多方面，我们现在仍然是知之甚少，在这方面计算机似乎还不可能马上取得很大的突破。

不过幸而这种类似人类的智能，对前述的大多数应用是不需要的。譬如说：仓库机器人，他只需要识别和操纵那些与装卸和库存有关的零件就行了。我们不会要求它去讨论哲学问题或者去下棋。反之，一个下棋的程序并不需要具有能直接感知它周围环境的能力，因为它的现实世界不会超过棋盘的范围。

所以，在我们探索人类智能全部能力的时候，只要我们编制这样一些程序，能够显示出人类某些单项智能，哪怕是以一种非常有限的形式表现出来，那么人工智能的研究就会立即开始取得某些成果。对计算机的某些应用来说，如果仅仅是一般地增加一些程序的灵活性，可靠性和解决问题的能力那是远远不够的。因此，人工智能所研究出来的一些技术方法，具有广泛的用途，它成为每个程序人员非常重要的工具。

三、算　　法

在非人工智能的程序设计中，程序员首先定出逐步解决某一类问题的方法，或者叫作“算法”，然后，他把这些算法步骤翻译成特定的程序语言，这样就得到这样一种程序，使得计算机有可能去解决同一类的许多问题。

但这种方法存在两个困难：

1. 有相当多的一类问题，数学家已经证明没有任何算法可以解答其中任何一个问题。

2. 纵使有一种算法能够解决某一类别中的所有问题，可是这个算法只对其中一些很小的问题才有实际意义；对于一些较大的实际问题却仍然是无能为力的。

拿第一个困难来说，我们可以举出用某种程序语言来写的计算机程序为例。譬如用 BASIC 语言写的计算机程序。最好是我们有一种算法能够检验这些程序中的某一个程序，并且可以确定它是不是可以终结，或者是否会进入一个无终点的循环中去。但可惜的是数学家已经证明不存在这样一种算法。也就是说，没有任何一种可以逐步实施的方法，能够应用到任何一个 BASIC 程序中去以确定它是否会进入一个无终点的循环。任何一种运用通用程序语言写的程序，都有同样的困难。

虽然有某种算法存在，可却是不实用的。为了说明白这一点，我们先考虑在执行某一算法，它所需要的时间是怎样取决于待解决的问题的规模的大小的。

我们就举加法和乘法问题为例。对加法和乘法，只要有小学程度的人都是很熟悉的，加法乘法问题的大小，是用它相加的或相乘的数值的数字位数来衡量的。这就是说，如果我们要加或乘两个十位数字的数目，加法和乘法问题的大小就是 10。

现在我们考虑，两个数的加法问题，很清楚，用我们在小学里所学过的知识，就可以看出加法所需要的时间是与相加两数的位数成正比的。譬如说，我们作两个十位数的加法，所需的时间就是作两个五位数的加法的两倍。

让我们用字母 n 来表示问题的大小，也就是相加数字的位数，用传统加法，其所需的时间是和 n 成正比的。

在乘法中，我们必须将一个数目的各位数字乘以另一数目的各位数字，那就是说，我们进行逐位数字相乘的次数就是 $n \times n$ 或者是 n^2 进行逐位数字相乘所需的时间也就是和 n^2 成正比的。那么我们乘两个十位数所需的时间将是乘两个五位数所需的时间的 4 倍，因为 $10^2/5^2 = 100/25 = 4$

随着问题规模的增大，与加法相比，乘法所需的时间增长得更快。一般地说，我们执行一个算法所需要的时间是因问题大小即 n 值而异的，但对不同的算法所需的时间也是不同的。

包含有 n、n^2、n^3 等的数学表达式叫作多项式。与 n、n^2、n^3 等成正比的算法执行时间叫作多项式运行时间。以多项式时间来运行的算法我们认为是能够实际执行的。

表 1-1　多项式和指数增长对照表

大小	多项式			指数		
n	n	n^2	n^3	2^n	n!	n^n
1	1	1	1	2	1	1
2	2	4	8	4	2	4
3	3	9	27	8	6	27
4	4	16	64	16	24	256
5	5	25	125	32	120	3125
6	6	36	216	64	720	46656
7	7	49	343	128	5040	823543
8	8	64	512	256	40320	16777216
9	9	81	729	512	362880	3.9×10^8
10	10	100	1000	1024	3628800	10^{10}
20	20	400	8000	1048576	2.4×10^{18}	10×10^{20}

有些算法运行的时间是和表达式如 2^n，n!（n 的阶乘，即从 1 到 n 的所有连续整数之积）和 n^n 等成比例，这样的算法叫作以指数时间运行的算法。表 1-1 指出，指数表达式随 n

而增长的速度比多项式要快得多。

以指数时间运行的算法，被认为是不能实际执行的，这一类算法只能成功地用于该类问题中非常少的一些问题。随着问题规模的增大，运行时间增加得非常迅速，以致即使一些中等大小的问题，也需要数千个计算机机时，至于规模庞大的问题则需要几千年的计算机机时。

四、人工智能程序设计原理

在人类每天所遇到的许多种非常重要的问题中，其最有名的一些算法都是以指数时间来运行的。而且数学家认为，这一类问题中的多数，它的所有一切算法都是要以指数时间运行的。

这类问题甚至不存在任何一种可以解决它的一切问题的算法；而且即使存在有这样一种算法，它也是以指数时间运行的。那么人类又是怎样应付自如地对付这类问题的呢？其理由大概有三个：

1. 任何一种算法都必须保证对于某一类问题的每个问题都起作用；然而人类所用的方法则只需用于一些合理的问题，或者是一些在实际中容易遇到的问题。人类可以相当成功地分析一个计算机程序，但如果你要求任何一个人分析一千页 BASIC 程序的话，他通常是会拒绝的，因为这个要求是不合理的。

2. 对算法的要求是要绝对地正确无误，而对人类来说却允许有偶然的失败和错误。人类愿意使用这样一种方法，它只要大部分时间起作用，或者偶然能起作用就行了。当一个算法在某一个问题上遭受失败时，他可以马上转向另一方法。

3. 人类并不总是坚持要找出一个准确的或者是最好的

问题解答，而只要求这个解答足以近似地解决手头的问题就够了，而不必浪费时间去寻求比它更好的解答。

从上面看来，好象人类的算法不如计算机的算法好；可是，实际上，人类在解决大多数实际问题时，他的方法要比计算机的算法成功得多。典型的情况是，有时候一个算法在理论上说能解决某类问题中的全部问题，但当把它用于实际问题时，由于算法是以指数时间运行，所以是根本不可能实现的。

不过，通常地说，一个人虽然用的是一些可能会发生错误的、近似的、粗略的方法，可是它却能对付实际问题，使他能得到一个即使不是理论上最好的、但却是一个可行的解答。所以，人工智能科学家不打算建立能解决大量问题的算法，而是集中精力来发明一些有时候虽然可能错误，但却是人类所使用的那些往往可能更为成功的一些方法。不过人工智能也并不放弃在实际上存在着的能够解决当前问题的一些算法。在本节后面我们将要介绍作为现在人工智能研究基础的一些技术。

(一)搜索

人类解决问题的方法之一就是尝试错误的搜索法，即逐个地检验可能解决问题的每个方法，直至找到一个可行的方法为止。检验一切可能的解决问题的方法，而不事先考虑哪一个方法是可行的，我们称它为盲目的、蛮干的搜索法。盲目搜索法，不是解决大而复杂的问题的方法。这在下一章中我们还会讨论，因为盲目搜索法不可避免地要用指数时间运行。其原因在于这种搜索法通常是要把问题中所包含的元素的可能组合都搜索一遍，那么，问题的规模越大，可能组合的数量也就增长得越大，这种效应通常被称为"组合爆炸"。

(二)启发式搜索法

如果我们用某种提示和经验规则来引导搜索朝正确方向进行，就可避免组合爆炸的发生。根据提示我们就可不必检验每个可能的解答，这样也就可以把我们进行的搜索限定在一些可能是最可行的范围内。

“启发式”(heuristic)一词源出于希腊语，它的意思是“帮助发现”。一个启发式就是一种暗示或者一种经验规则，它可以帮助我们发现问题的解答，用启发式进行的搜索法，就叫作“启发式搜索”。

人类大量地应用着这种启发式，有时候，我们把这种方式看成是一种经验规则。举例来说，我们用热水去浇瓶盖，然后去开启它，这就是应用经验规则。在另一些情况下，启发式就是简单地应用我们去做某件事的正确方法的知识。我们“知道”应该怎么样去作一件事情，但并不会感觉到我们是在遵循着任何特别的规则，可是如果看到有人不按照这些不成文的规则作事，就会感到奇怪。

格言或谚语往往是一些很好的启发式的例子，试看以下几段话：

- 一针及时省九针。
- 多得不如现得。
- 看神打卦，看事说话。
- 失败为成功之母。

可以看出所有的格言(和全部启发式)都有共同的特征。

每个格言都是针对某种特定的情景。由“一针及时省九针”这句格言所唤起我们想像的情景是完全不同于“多得不如现得”这句格言所唤起的情景的。

每个格言都是一个良好的忠告，那就是说，每个格言在相

应的情景下都是值得考虑的。

可是另一方面，格言并不总是保证我们能得到想要得到的结果。引用格言的时候，我们不难理解，在某一种情景下某个格言是可能应用的，可是我们却不应当应用它。譬如，当我们正在作一项根本不可能成功的工作时，那么“失败为成功之母”这句格言，也就不能帮忙了。

所有的启发式都有以下共同特点：它们都应用于特定的情况，而且它们的忠告是值得考虑的，但是它们是不同于算法的。因为启发式并不确保成功，它们只不过是值得尝试的一些事情，它们在特定的情景下，或许起作用，也或许不起作用。

启发式搜索是人工智能中问题求解技术的一个主要方法。

(三)模式识别

因为每种启发式方法都只适用于某种特定的情景，所以首先我们就需要了解问题的情况，然后才能决定采用哪种启发式。求解问题的人先要了解问题情况的模式，再从这些模式出发，来决定应采取哪些启发式。

譬如，有不少谈“方法”一类的书里有许多规则都具有这样的形式：如果某种情景发生，那么就该怎么怎么办，所提的“某种情景”就是一种模式，而“怎么怎么办”就是当这种相应的模式被找到以后，可以使用的那种启发式方法。

一个老师傅和一个受过良好教育的新手，也许都知道同样一个启发式方法，但是老师傅马上就可以认出待解决的那个问题的模式，知道哪个方法是可行的；而新手则往往需要一个一个地进行尝试，一直到他不得不全部放弃或碰到一个可行的方法为止。

(四)规划

另外一种避免组合爆炸的方法是首先作规划，也就是制订一个解决问题的计划，然后再考虑哪些解决问题的方法适合所制订的计划。

譬如，假设你要从一个城市驱车到另一个远方的城市去，而你又不愿意在开车的时候去查看地图；另一方面，假设指路牌只能告诉你从一个城市到另外一个邻近城市，而不能指给你如何走到这个远方的城市。解决的办法就是先制订一个旅行计划。出发前你先得查看地图，然后，应用这种启发式方法：考虑从出发地到你要去的远方城市，其间所经过的城市，基本上应在一条直线上。这样，你就可以把沿途的城市列成一个表，按照这个计划表，你就可以利用指路牌引导你从一个城市驱车到表上所列的下一个城市。这样做，就可大大地减少从出发点到终点可能经过的路径的数目。开车的时候也就不必担心，因为当你离开每一个城市的时候，你只要研究一下如何走到表上所列的下一个城市的路标，而不必再去考虑从这个城市出发的每一条可能的路径了。

所以计划能大大地减少搜索中每一步所应考虑的那些可能选择的路径数目，从而也就减少了组合爆炸的可能性。

利用启发式的一个好办法便是利用它来设想计划。解答问题的人，要首先按照问题的情况找出与特定启发式有关的一些模式，这个与模式有关的启发式将会为解答该问题而提示一个特定的计划。一旦这个计划被试用时，该计划的每一个步骤都可被看成一个新的问题，并且可用同样的方法去解决。

启发式并不能保证一定成功，所以第一个计划也可能行不通。求解问题的人，就得准备放弃旧的计划来制订新的计划。可是另一方面，如果他放弃一个计划又去找新的计划，这

样改变的次数过多，那么，组合性爆炸就肯定会出现。

（五）知识表达

用计算机来解决问题之前，必须在计算机内把待解决的问题详加描述，给出表达或编码。

更重要的是，启发式和解决问题的计划以及它的模式都必须要存贮在计算机内，而信息存贮的方式必须是易于存取。计算机还必须要能很快地从和问题情景对应的模式迅速过渡到特定的计划和经验规则中去。

人类因具有经验，所以他能自动地作到这一点。一个下棋的专家，当他审视每个棋局时，马上就能看出许多种模式，这些模式会提出下这盘棋的所有可能的棋路；而一个新手对于同一个棋局，则只能看到一些最简单的关系。一个数学教师，当看到一个代数表达式的时候，马上就能想出好几种方法能将它转换成某种所需要的形式，而一个学生呢，他就会无可奈何地埋怨："不知从何下手"。

我们需要注意的是，每一种不同的人工智能的技术，都可以互相利用。问题求解往往需要模式识别和知识表达。而模式识别则又可能包含问题求解（寻求相关的模式本身就是一个求解的问题），且对于所寻求的模式的知识表达必须很方便地存贮在记忆里。从计算机的信息库里检索所需要的信息，既包含着问题求解，也包含着知识表达。看来在任何一个人工智能的领域中要前进的话，都不可能不涉及到其他的方面。

第二章　问题求解的策略

在第一章中我们已经指出人工智能的程序就是问题求解的程序。所以，我们研究人工智能就当然要从问题的研究开始。各种各样的问题都有哪些共同的特点呢？我们是否能一般地来讨论与任何特定问题细节无关的那些特征呢？我们又怎样在计算机内表达这样一些特征，使得很容易利用问题求解的程序去处理它们呢？

一、状态、操作和目标

一切问题都有三个重要的共同特征，就是状态、操作和目标。下面我们就用具体问题做例子来说明这些特征，试以“传教士和野人”问题为例。在河的一岸，有三个传教士、三个野人和一条船，传教士想用这条船将这些人全部都渡到河的对岸。可是很不幸，这里有两个难题非考虑不可。

1. 该船每次只能载两个人。传教士和野人都会划船，因此这只船每次可运送一个传教士，或者运送一个野人，也可以运送两个传教士或者两个野人，或者是一个传教士和一个野人。

2. 在河的一岸，如果野人的数目超过了传教士的数目，那么野人就会把传教士吃掉。

因此传教士必须做出一个摆渡计划，既能使所有的人渡到对岸，又不致冒传教士被吃掉的危险。我们假设野人能够同传教士合作去执行该计划，这可能因为野人不知道传教士所设计的这个计划是欺骗他们，使得他们无法吃掉传教士的。

(一)状态

为要解决这个问题，盘算着如何由两岸来回运送传教士和野人的时候，我们经常要注意下面三个问题：

1. 河的两岸各有多少个传教士？

2. 河的两岸各有多少个野人？

3. 船在河的哪一边？

这三条信息描述了求解问题过程中目前的状态，即有多少传教士和野人已经渡到对岸，还剩多少人等待渡河，以及船现在河岸的哪边，还有谁能够摆渡，等等。

我们把每个问题都看成是一个独立的"世界"，所谓状态也就是指这个世界中事物的现状；这个状态也可能纯系假设性的，因为求解问题的人在现实世界中采取任何实际行动之前，他们只能去想象解决问题的方法。

当然，对玩偶问题说来，这个世界也就是玩偶世界。对于传教士和野人问题来说，我们所遇到的只有三个传教士、三个野人、一只船和一条河。但如果问题求解者是个机器人，要他去处理现实世界的话，那么状态就是现实世界的状态，它至少是对机器人有关的现实世界的那一部分。

在求解问题时，经常给予我们的是问题的初始状态。拿传教士和野人问题来说，它的初始状态就是所有的传教士、野人和船都在河岸的一边（譬如说，都在左岸）。问题求解的目的就是要改变这种状态，使成为具有某些特征的新状态。

研究问题世界中一切可能状态，往往是很有用的办法。让我们以传教士和野人问题为例来试试看。

要描述一个特定状态，我们必须给出在河的两岸传教士(M)和野人(C)的数目，以及船(B)在河岸的位置。可将初始状态描述如下：

	左 岸	右 岸
传教士(M)	3	0
野 人(C)	3	0
船 (B)	有	无

如果我们把一个传教士和一个野人渡到了河的对岸，那么我们就得到下面的状态：

	左 岸	右 岸
传教士(M)	2	1
野 人(C)	2	1
船 (B)	无	有

可是实际上，这里我们所给出的状态已经超出需要，因为左岸和右岸相加的传教士人数必定是 3 个，野人也是 3 个，船总归只能在河的某一边。因此如果我们确定了一岸，(譬如说左岸)的情况，那么，我们也就知道了另外一岸(右岸)的情况。因此我们可以简化初始状态的描述为：

	左 岸
传教士(M)	3
野 人(C)	3
船 (B)	有

当我们运送一个传教士和一个野人到对岸以后的状态就变成了：

	左 岸
传教士(M)	2
野 人(C)	2
船 (B)	无

我们一共有多少种可能状态呢？因为左岸可能有 0 个、1 个、2 个或 3 个传教士，所以就传教士在两岸分配的情况来

说，只有四种可能性。对野人来说也同样有四种可能性。船可能在河的左岸或右岸，即有两种可能性。于是，总的可能状态有 4×4×2 种，即一共有 32 种。

表 2-1 表明了这 32 种可能状态。当然只要写出左岸的情况就足够了。但为了读者方便起见，我们还是把两岸的情形都写出来。

且慢！我们应该注意的是，在表中某些情况下，在河的一岸野人的数目超过了传教士的数目；这种情况下，传教士就要遭殃，很显然这不是合法状态。

稍加思索，我们就能看出，合法状态只有两种类型：

1. 在河的两岸传教士和野人的数目恰好相等。

2. 三个传教士都在河的一边，对岸一个传教士也没有。而野人则可在两岸之间任意分配。因为这时在河的一岸传教士的数目肯定要超过野人的数目，而在另一岸一个传教士也没有，因此也就不会有人被吃掉的危险。

我们把表 2-1 中 32 种状态都检查一遍，就会发现，其中有 20 种状态可以满足上述两个条件中的任一个，我们在表 2-2 中列出了这 20 种合法状态，但其中有 4 种仍然是有问题的，因为它们是从初始状态无论如何也达不到的状态。这四种不可能达到的状态是：

左　岸			右　岸		
传教士(M)	野人(C)	船(B)	传教士(M)	野人(C)	船(B)
3	3	无	0	0	有
0	0	有	3	3	无
3	0	有	0	3	无
0	3	无	3	0	有

表 2-1　传教士和野人问题中 32 种可能状态

左岸			右岸		
M	C	B	M	C	B
0	0	有	3	3	无
0	1	有	3	2	无
0	2	有	3	1	无
0	3	有	3	0	无
1	0	有	2	3*	无
1	1	有	2	2	无
1	2*	有	2	1	无
1	3*	有	2	0	无
2	0	有	1	3*	无
2	1	有	1	2*	无
2	2	有	1	1	无
2	3*	有	1	0	无
3	0	有	0	3	无
3	1	有	0	2	无
3	2	有	0	1	无
3	3	有	0	0	无
0	0	无	3	3	有
0	1	无	3	2	有
0	2	无	3	1	有
0	3	无	3	0	有
1	0	无	2	3*	有
1	1	无	2	2	有
1	2*	无	2	1	有
1	3*	无	2	0	有
2	0	无	1	3*	有
2	1	无	1	2*	有
2	2	无	1	1	有
2	3*	无	1	0	有
3	0	无	0	3	有
3	1	无	0	2	有
3	2	无	0	1	有
3	3	无	0	0	有

* 非法状态　M 传教士　C 野人　B 船

前面两种状态之所以是不可能的，是因为船一定得有人

表 2-2　传教士和野人问题中的 20 种合法状态

左　岸			右　岸		
M	C	B	M	C	B
0	0	有*	3	3	无
0	1	有	3	2	无
0	2	有	3	1	无
0	3	有	3	0	无
1	1	有	2	2	无
2	2	有	1	1	无
3	0	有*	0	3	无
3	1	有	0	2	无
3	2	有	0	1	无
3	3	有	0	0	无
0	0	无	3	3	有
0	1	无	3	2	有
0	2	无	3	1	有
0	3	无*	3	0	有
1	1	无	2	2	有
2	2	无	1	1	有
3	0	无	0	3	有
3	1	无	0	2	有
3	2	无	0	1	有
3	3	无*	0	0	有

* 不可能从起点状态到达

划，才能摆渡到对岸，所以不能允许船只在一个没有人的岸边。

后面两种状态也是不可能的，但情况稍为复杂，因为这只船是在传教士所在的一边，最后必须得由一个传教士划船渡到已经有三个野人的那个岸边再返回。事实上，他是再也回不来了。因为当他到达对岸的时候，那里已经有三个野人在等着，而只有他一个传教士，他可能马上被野人吃掉。

表 2-3 表示的是除去以上不可能状态以后剩下的从初始状态可能达到的 16 种合法状态。

表 2-3　传教士和野人问题中 16 种可能达到的合法状态

左　岸			右　岸		
M	C	B	M	C	B
0	1	有	3	2	无
0	2	有	3	1	无
0	3	有	3	0	无
1	1	有	2	2	无
2	2	有	1	1	无
3	1	有	0	2	无
3	2	有	0	1	无
3	3	有	0	0	无
0	0	无	3	3	有
0	1	无	3	2	有
0	2	无	3	1	有
1	1	无	2	2	有
2	2	无	1	1	有
3	0	无	0	3	有
3	1	无	0	2	有
3	2	无	0	1	有

(二)操作和操作符

我们解决问题的时候，要实行一系列的操作，使从初始状态转变为某一种我们所希望达到的状态。

一般地说，对于容许的操作，还有一些约束和限制。譬如阿西莫夫（Isaac Asimov）的“机器人三定律”就是讲机器人在解决问题和进行操作时应遵守的三条约束（比方说，杀人就是不容许的操作）。

就传教士和野人的问题来说，它的操作包括用一只船来回运送传教士和野人，最后全部渡到对岸。当然我们不能任意地改变两岸传教士和野人的数目和船的地点；当我们作任何改变时，都要受到以下四条约束的限制：

1. 传教士和野人只能用船只渡到对岸，而船又必须由一

个人去划它。因此在船只离去的此岸人数经常是要减少的，而船只到达的彼岸，人数却是增加的。

2. 因为船只能载两个人，所以每渡一次最多只能渡过去两个人。

3. 所运载的传教士和野人的数目，绝不可能超过停放船只的这一岸边的传教士和野人的数目。

4. 渡河的结果绝不容许产生一种非法状态；即在任何一岸都不容许野人的数目超过传教士的数目。

操作符就是对应于有效操作的状态改变的一个详细描述。操作符包括两部分，即条件部分和行动部分。操作符只许应用于那些条件成立的状态，条件部分约束着操作符的应用。行动部分所描述的则是当操作符应用时所产生的状态变化。

操作符的条件部分体现了对可容许操作的约束。

就传教士和野人的问题来说，我们可以对每一个操作符都给予一个叙述它的效应的描述性名称。譬如，“把一个传教士和一个野人从左岸运到右岸”就是其中的一个操作符。它也包括条件和行动两个部分：

条　　件	行　　动
船在右岸	将船从左岸划到右岸
左岸至少有一个传教士和一个野人	左岸传教士和野人的数目各减去一个
左岸的传教士和野人的数目相等	右岸传教士和野人的数目各增加一个

另一个操作符，就是只把一个野人从左岸运到右岸。这个操作符的描述名称为“从左岸运送一个野人到右岸”。它的两个部分可列出如下：

条　　件	行　　动
船在左岸 左岸有 0 个传教士或者 3 个传教士 左岸至少有一个野人	将船从左岸划到右岸 左岸野人的数目减少 1 人 右岸野人的数目增加 1 人

条件部分应保证任何操作符都不得产生出不合法状态，譬如，“将一个传教士和一个野人从左岸运到右岸”这个操作符只能用于这样一种状态，即左右两岸野人的数目都等于传教士的数目。而行动部分并不会改变这个条件，所以应用这个操作符以后这个条件仍然成立。所以结果状态也是合法的。

表 2-4 给出了传教士和野人问题的所有操作符的描述名称。

表 2-4　传教士和野人问题操作符的描述名称

1. 将 1 个传教士从左岸运到右岸
2. 将 1 个野人从左岸运到右岸
3. 将 1 个传教士和 1 个野人从左岸运到右岸
4. 将 2 个传教士从左岸运到右岸
5. 将 2 个野人从左岸运到右岸
6. 将 1 个传教士从右岸运到左岸
7. 将 1 个野人从右岸运到左岸
8. 将 1 个传教士和 1 个野人从右岸运到左岸
9. 将 2 个传教士从右岸运到左岸
10. 将 2 个野人从右岸运到左岸

这些操作符的行动已经包含在它们的描述名称里面。请你想象出每个操作符所需要的条件，那是很有趣的，这些条件应使

得：

1. 操作符能合理地应用到所提问题的状态。

2. 其结果必须是个合法状态。

这条件的第一部分容易做到，第二部分就需要好好思考一番了。

(三)目标

一个问题的目标，就是从初始状态出发，我们应用操作符想要得到的那些状态。

有时候我们只有一个目标状态。譬如传教士和野人问题中，我们想要达到的是以下状态：

	左　岸	右　岸
传教士(M)	0	3
野　人(C)	0	3
船　(B)	无	无

另一方面，一个目标状态，也可以是使某个给定条件能够成立的任何状态；如果该条件在许多状态下都是可以成立的，则这些状态中每一个状态都是可以接受的目标。譬如，如果问题是要赢得一盘棋，则能把对方"将死"的任何状态(棋局)都是一个满意的目标。

(四)问题的解

一个问题的解就是这样的一系列操作符，它使我们从初始状态出发，依次运用这些操作符以后，最终可以达到的状态为一个目标状态。在我们展示某个问题的解时，每运用一次操作符所获得的中间状态往往也应该表示出来。

表 2-5 所示就是关于传教士和野人问题的解。这个问题

还有几种其他的解，但它们和这个解稍有不同。

表 2-5　传教士和野人问题的一种解答

解　　答	状　态					
	左　岸			右　岸		
	M	C	B	M	C	B
初始位置	3	3	有	0	0	无
将 1 个传教士和 1 个野人从左岸运到右岸	2	2	无	1	1	有
将 1 个传教士从右岸运到左岸	3	2	有	0	1	无
将 2 个野人从左岸运到右岸	3	0	无	0	3	有
将 1 个野人从右岸运到左岸	3	1	有	0	2	无
将 2 个传教士从左岸运到右岸	1	1	无	2	2	有
将 1 个传教士和 1 个野人从右岸运到左岸	2	2	无	1	1	无
将 2 个传教士从左岸运到右岸	0	2	无	3	1	有
将 1 个野人从右岸运到左岸	0	3	有	3	0	无
将 2 个野人从左岸运到右岸	0	1	无	3	2	有
将 1 个传教士从右岸运到左岸	1	1	有	2	2	无
将 1 个传教士和 1 个野人从左岸运到右岸	0	0	无	3	3	有

注：M 传教士　C 野人　B 船

二、状　态　图

我们在思考一个问题的时候，这样做往往是很有帮助的：即我们只需注意状态和操作符之间的各种关系，而可忽略它的详细结构。状态图就可为我们提供这种帮助。

（一）图的术语

“图”这个词用于数学中有两重意义。我们这里所谈的图，不是指那些描绘数学函数和实验数据的图。我们所谈的图是由点及其连线所构成的图。“网络”这个词就是这里所说的“图”这个词的同义语。

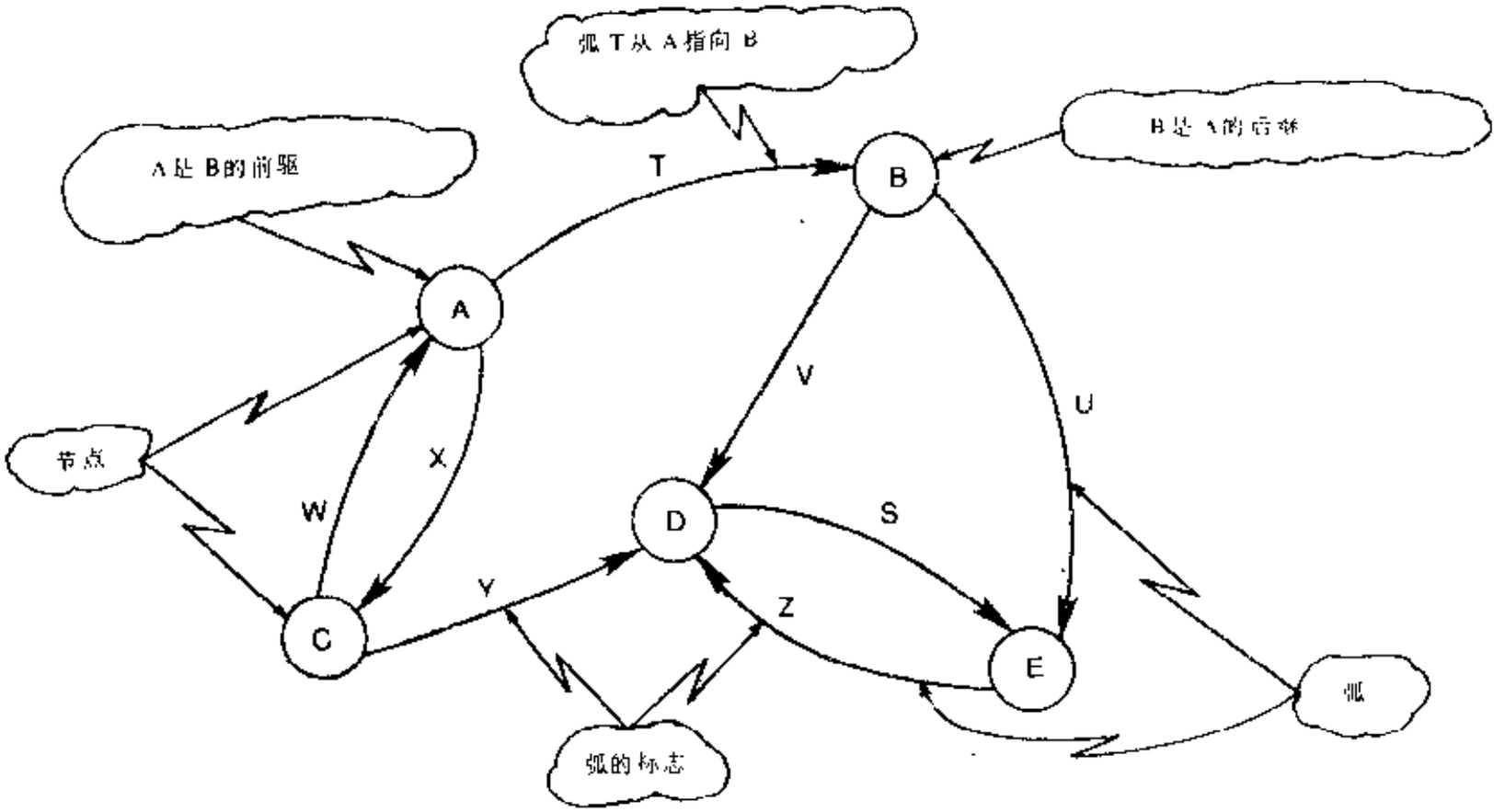


图 2-1　图的基本名词术语

在图 2-1 中我们用举例的方式说明了图的一些基本术语，这里面的各点称为“节点”，为了便于给节点加上标志，我们常常用圆圈来代替几何点。连接这个节点的联线称做“弧”。如果弧线是带箭头的，那么这个图就称之为“有向图”。如果弧线是带有标记的，那么我们就称这图为有标记的图。一个有标记的图又称做“带色图”。因为我们可以把带有同一标记的弧线看做是同一种颜色的，而带不同标记的弧线就是带有不同颜色的。

如果弧线从节点 A 联向节点 B，并且箭头指向节点 B，我们就说 A 是 B 的前驱，而 B 是 A 的后继。所谓路径指的是这样一系列的节点，其中每一个节点都是它一个节点的前驱。

(二)状态图

在状态图中，节点都对应于状态，而弧的标记则对应了操作符。其中有一个节点代表初始状态，还有一个或多个节点对应于目标状态。问题的解就是从对应于初始状态的节点连结到一个对应于目标状态的节点的路径。

我们曾经用具体例子来帮助理解什么叫状态、操作符和目标，我们也可以借助于具体例子来理解什么叫状态图。下面我们就用所谓“梵塔”问题在图 2-2 中举例说明。

在图 2-2(a)中我们见到三根柱子：1 号，2 号，3 号。在 1 号柱上有些盘子自下而上叠放着，一个比一个小。给我们的问题是要把所有的盘子从 1 号柱挪到 3 号柱上去。约束条件是：不许把较大的盘子放到较小的盘上面，也就是说，在每个柱子上最大的盘子永远要放在最底下，次大的盘子放在它的上面。依此类推。

梵塔问题的复杂程度与盘子的数目有关。我们这里用较

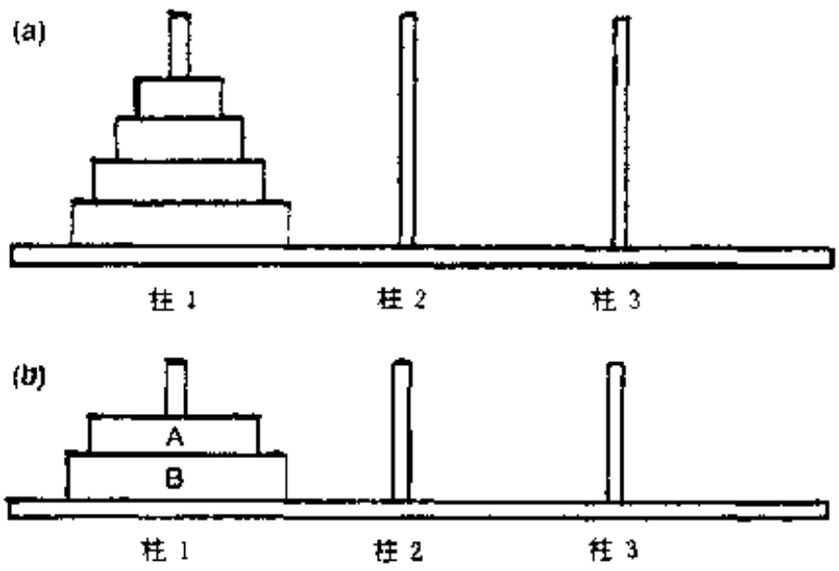


图 2-2

(a)梵塔问题 (b)只有两个盘的梵塔

简单的问题来举例,以便容易给出它的状态图,所以我们下面的例子中只有两个盘子。

在图 2-2(b)中,是两个盘子的梵塔的例子。令 A 代表较小的盘子,B 代表较大的盘子,子是我们只需要说明 A 盘在哪根柱子上,B 盘在哪根柱子上,就可以很清楚的描述梵塔的各个状态。如果 A 盘和 B 盘在同一柱子上,我们用不着担心它们的次序。因为大盘永远在底部而小盘永远在顶部。

梵塔的任何状态现在都可以用一对数字来表示,例如(32)或(13)。括号内第一个数说明 A 盘所在的柱号、第二个数说明 B 盘所在的柱号。因此(13)说明 A 盘在 1 号柱上,B 盘在 3 号柱上。在(33)状态中,A 盘和 B 盘都在 3 号柱上;当然它们总是按照所要求的顺序放置,即 B 盘一定在下面,A 盘一定在上面。

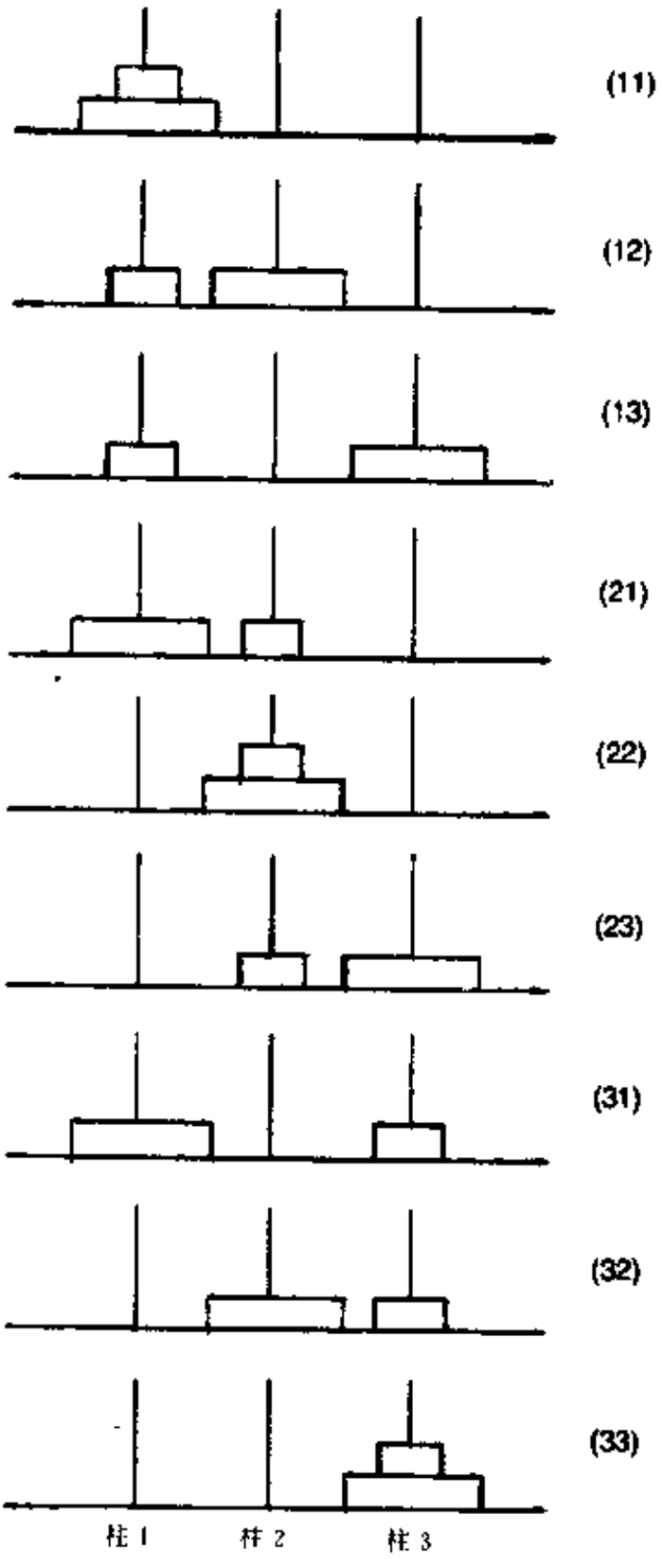


图 2-3　梵塔世界中的九种状态

因为 A 盘和 B 盘中的任一个都可以放在 1 号柱、2 号柱或 3 号柱的上面，这样就一共有 3×3 共九种状态。任何一对数，例如(31)或(22)都可以描述一种状态，但要求其中每位数字都是从 1 到 3 以内的数。所有这些容许数字的组合都是成立的，无一例外。就象在传教士和野人问题中那样。图 2-3 表示了梵塔问题中的九种状态及其描述。

其初始状态是(11)，目标状态是(33)。

每个操作都包含把一个盘子从一个柱移向另一个柱的动作。操作符的作用就是改变状态描述中的一个数字，以便反映盘子移动的情况。当然操作符并不能任意地改变这些数字，因为它要受到以下的约束：

1. 每次只许移动一个盘子。所以操作符只能改变一对数目中的第一个数字或第二个数字，但不能同时改变两个数字。

2. 如果 A 盘和 B 盘在同一号柱上，我们只能移动顶部的那个盘，也就是 A 盘。因此对于状态(11)、(22)和(33)时只能允许把第一个数字即对应于 A 盘的数加以改变。

3. B 盘绝不允许放在 A 盘的顶上，因此一对数中的第二个数字绝不允许变到和第一个数字相等。换句话说，我们不能把(21)变成(22)，因为 A 盘已在 2 号柱上面，再把 B 盘放在它上面是违反规定的(这里假定我们不许把 A 盘先拿下来，把 B 盘放上去，然后再把 A 盘放回)。

现在我们用下述方式来描写操作符。

A：3—2 代表将 A 盘从 3 号柱移到 2 号柱；

B：1—2 代表将 B 盘从 1 号柱移到 2 号柱，照此类推。

表 2-6 表示了梵塔问题中的几个不同的操作符。

表 2-6　梵塔问题中 12 个操作符

A:1—2	B:1—2
A:1—3	B:1—3
A:2—1	B:2—1
A:2—3	B:2—3
A:3—1	B:3—1
A:3—2	B:3—2

在以上的约束中，约束 1 是自动满足的，因为每个操作符每次只移动一个盘。可是约束 2 和约束 3 却限制操作符使它不能适应每一个状态。例如：B:2—3 不能应用到 (22)，因为有约束 2。而 B:3—2 不能应用于(23)，因有约束 3。

最后还有一条，比方说，B:1—2 只能应用于 B 盘在 1 号柱的所有状态。因为如果 B 盘不是已经在 1 号柱上的话，我们就不可能把 B 盘从 1 号柱移到其他柱上去。

梵塔状态图。图 2-4 表明梵塔问题的状态图。

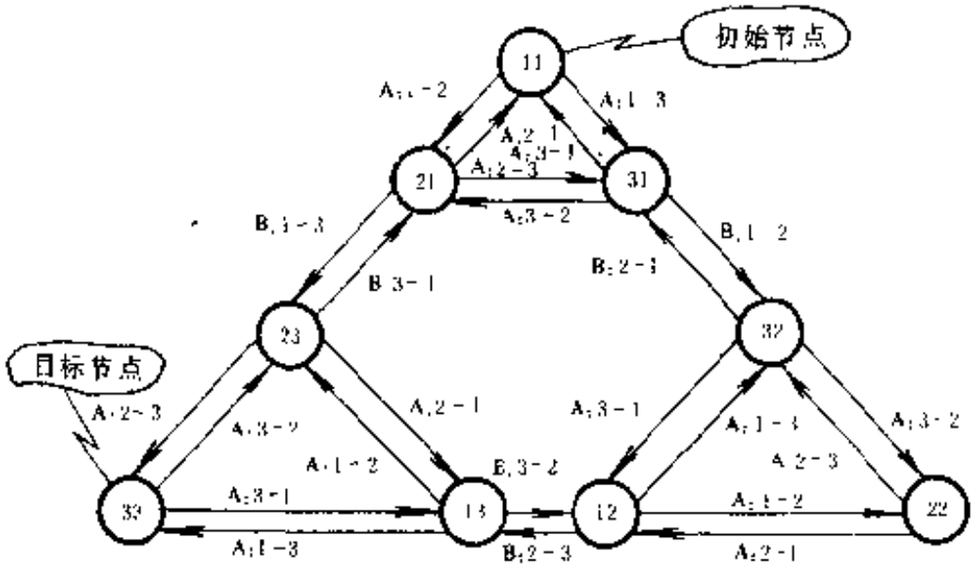


图 2-4　梵塔问题的状态图

图中九个节点中的每一个都对应于梵塔世界中九个状态中的一个。每个节点都有它对应的状态描述标志。

二十四个弧中的每一个都对应用于十二个操作符中的一个。

请注意，某些“节点对”并没有被任何弧线直接连接起来，这是因为它们受到约束的缘故。这些约束不允许某些状态直接借一个操作符转变到另一状态(如果没有这些约束，那么整个问题就都不存在了。那么我们就可以从初始状态用一个操作符直接达到目标状态了)。

图中我们标明了初始状态和目标状态。

问题的解，就是指任何一条从初始状态出发到达目标状态的路径。就梵塔问题来说，它有多个解，其中的三个解表示在图 2-5 中。如果可能的解不止一个时，我们就必须用某种判据来选择其中最好的一个。所谓最佳的解通常就是指从初始状态转变为目标状态时所作的操作为最少的一条路径。

所以求解一个问题，就相当于从图中找到一条从初始节点到目标节点的路径。这问题和从迷宫中寻找道路的问题是十分类似的。一切问题如果从图形的观点出发，都可以看作是一个迷宫问题。

暗示图。我们刚才举的是明示图，也就是说我们可以把整个图画在一张纸上，或者把它存贮在计算机的存贮器里。

可是，这并不总是可能做到的，譬如说，在原始的梵塔问题中就有 64 个盘子。这就是说，它共有 3^{64} 或大约 3×10^{30} (即 3 后面带 30 个“0”)种状态。我们绝不可能把这么多节点的图画在一张纸上，或者把它存贮在计算机内。

幸而，我们并不总是需要画出整个图来。给定任何一个节点时，我们的兴趣主要在于紧接它的后继。也就是那些由原项用弧所连结的那些节点。就状态而言，某个紧接的后继，就是从原来状态应用单个操作符所能达到的任一状态。我们只要使用所有可能允许的操作符来对一个状态进行运算，就能产生出它所有紧邻后继。

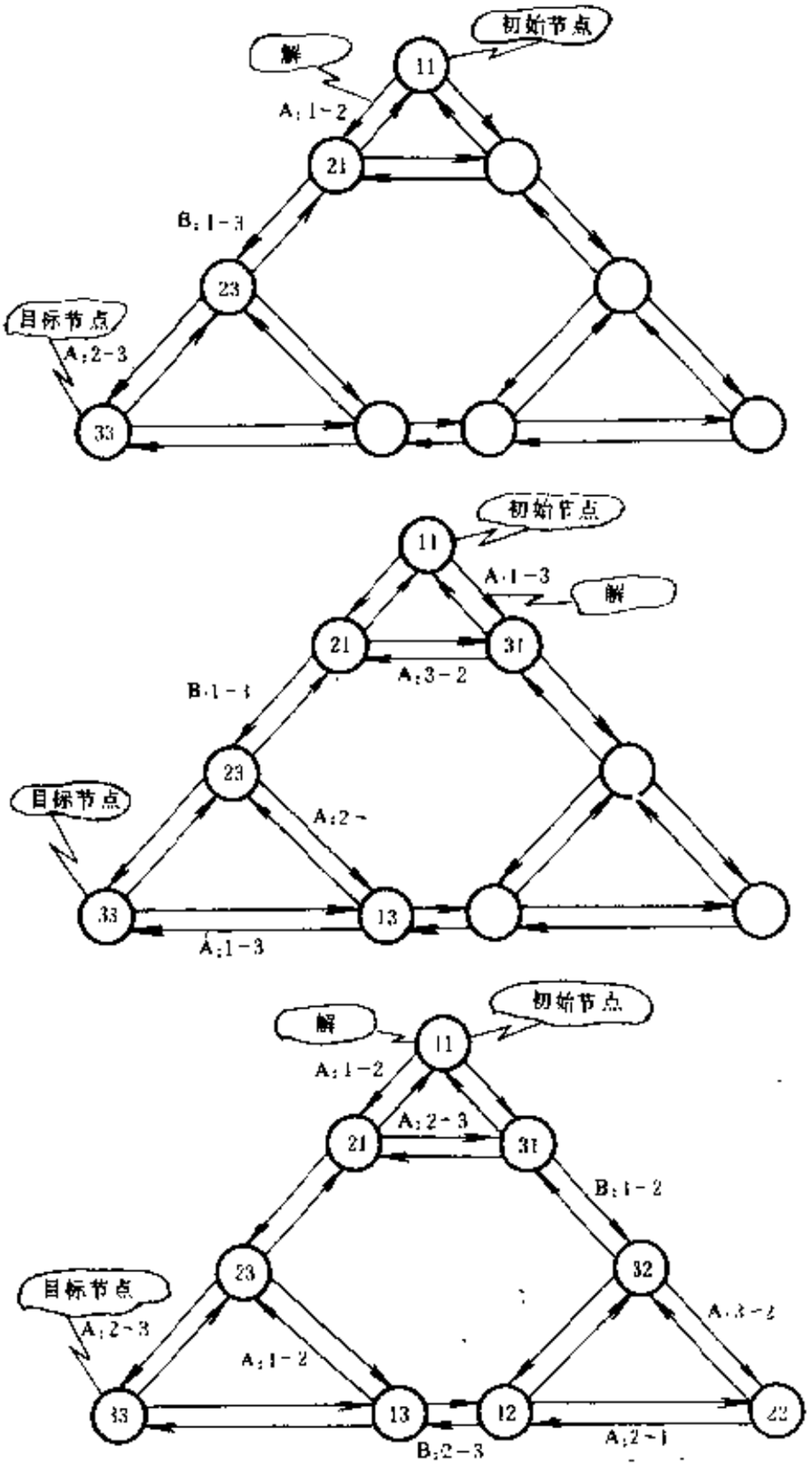


图 2-5　梵塔问题的三个解。每个解都是从初始节点到目标节点的一条路径

所谓暗示图就是照这样的规则所定义的一个图形；这个规则就是查找任何节点的紧邻后继的规则。这样，我们就能构成从初始节点出发，经过各种途径所能达到的一个局部图。只要对不同的初始节点都这样做下去，我们就能构成我们所需要的图形中的任何一个局部。

凡是与复杂问题有关的图形差不多都是暗示性的。我们一次能存贮多少这样的图形，取决于使用的计算机存贮容量的大小。

三、计算机的表达方法

当我们表达一个问题的时候，我们往往要构造一些其他的问题，使这些问题等同于原始问题。对应于每种原始状态或者操作符，都有相应状态和相应操作符的表达方式。这些操作符对状态所起的效应在原始问题及其表达方式中也是对应的。

为什么我们要使用表达方式呢，就是因为它比原始问题更容易理解和处理。以状态图为例：不管原始问题的形式如何，一旦把它表达成状态图以后，它就转换成为一个从原始节点向某个目标节点寻找路径的问题了。如果我们能研究出从图形中寻找路径的一般方法，那么，我们也就可能用这样一个方法来解决不同种类的问题，即只要把这个方法应用于这些问题的状态图表达方式就行了。

如果我们想要利用计算机来求解问题，那么，我们就必须把这些问题在计算机内表达出来。这就是说，我们必须确立所要解决的问题中的所有元素和计算机所能够处理的存贮值之间的一一对应关系。

(一)状态表达法

我们就从状态如何表达来开始吧。对于每一个问题来说，问题世界的状态，都具有某些特征或特性。这些特征对于不同的状态具有不同的值，所有这些特征的值，决定着一个特定的状态。

以传教士和野人问题为例来说，状态特征就是左岸传教士的数目和左岸野人的数目和船是否在左岸（请记住：对右岸的情况我们不必说明，因为我们可以从左岸的情况推导出右岸的状况）。

在高级语言如 BASIC 和 FORTRAN 中准备了一些变量，这些变量所具有的值，在程序执行过程中是可以改变的。我们可以用一个变量来表明某状态的一个特征。即这些变量的值的某个特别规定，可以确定一种特定的状态。

例如，我们可以令 M 代表传教士的数目，用 C 代表野人的数目，如果船在左岸，我们可以令 B 的值为 1（或 Yes），如果船在右岸则 B 值为 0（或 NO），那么下表所列的值：

变　量	值
M	3
C	1
B	1

就可以指明这样一种状态：即左岸有三个传教士（右岸一个也没有）、一个野人（右岸有两个）和船在左岸。

我们可以把传教士和野人运过来、运过去，实际上只要改变这些变量的值就行了，这是用语句来完成的。按 BASIC 语言的写法，就是用语句

```
10    LET  M=M-2
```

```
20      LET   B=0
```

就可以表示把两个传教士从左岸渡到右岸，其结果状态可以用下表中的值来表示：

变　量	值
M	1
C	1
B	0

可是，这里有个问题，就是我们也同样容易象表达一个合法状态一样，用变量的值来表达一个非法状态。下面的两个状态就都是非法的：

变　量	值
M	5
C	100
B	3

和

变　量	值
M	1
C	2
B	1

第一个状态之所以是非法，因为这些值对现在的问题是无意义的(因为传教士和野人的数目都超过 3 个，船的状态 3 毫无意义)。第二种状态之所以非法，因为它所表示的是传教士将要被吃掉的状态(野人的数目超过传教士)。

在大多数程序语言中，没有任何办法能约束一个变量的所有值，使得它能保证任何值都描述一种合法状态。所以，我们就必须另外设法确定，每当一个变量的值被改变的时候，它所产生的新值必须描述一个合法状态。换言之，我们把监督这个状态的描述是否合法的重担放在操作符身上，而不是放在状态变量身上。

最后，让我们来表达一下梵塔问题的状态。状态的两个特征是 A 盘所在的柱号以及 B 盘所在的柱号。我们可以用两个状态变量 A 和 B 来表示，它们的值分别代表 A 盘和 B 盘

所在的柱号。下面的值就描述了 A 盘在 2 号柱上，B 盘在 3 号柱上的这种状态。

变　量	值
A	2
B	3

(二)操作符表达法

我们已经知道每一个操作符都包含有条件部分和行动部分。

行动部分通过改变状态变量的值来改变这个状态。它们很容易用一些赋值语句来表达，譬如："将一个传教士和一个野人从左岸运往右岸"这个操作符，在 BASIC 语言中，它的行动部分就可以用下面三个语句来表达：

```
10    LET  M=M-1
20    LET  C=C-1
30    LET  B=0
```

但仅当执行这个行动部分于这个状态，使能产生另外一种合法状态时，这个操作符的条件部分才允许把这种运算应用到一个特定的状态。譬如说，刚才所给出的行动不允许应用于这样一种状态，即 C 值为 0 的状态，因为它的结果是一种不合法状态(因为其结果 C 的值将是负数－1)

假定我们应用的是 BASIC 语言的一个较好版本，它允许我们比较变量的值，如表 2-7 中所示。那么一些简单的条件，就可以通过"与"(AND)、"或"(OR)和"非"(NO)等逻辑运算符号来产生出一些更为复杂的条件语句。

例如式子

(B=1)　AND(C>=1)AND((M=0)OR(M=3))。

是描述这样一种状态，即其 B 值为 1 且 C 值大于或等于

1，以及 M 值可能等于 0 或 3。

表 2-7　变量值的比较

条　件	意　　义
X=Y	X 的值等于 Y 的值
X<Y	X 的值小于 Y 的值
X>Y	X 的值大于 Y 的值
X<=Y	X 的值小于或等于 Y 的值
X>=Y	X 的值大于或等于 Y 的值
X<>Y	X 的值不等于 Y 的值

在程序中，有一些条件可以用专门的条件语句来描述。例如 BASIC 语句中的“IF”语句。举例：若上述条件为真，就把 F 的值定为 1，否则 F 的值就为 0。可以记为：

```
10    LET  F=0
20    IF (B=1) AND (C>=1) AND ((M=0) OR (M=
      3))
      THEN LET F=1
```

象 F 这样的变量，它的值是反映条件检查的结果的，它常用于人工智能中。这样一些值就叫作标志(flags)。

表 2-8 表示了传教士和野人问题的所有 10 个操作符。条件和行动部分都用刚才描述的记法来表示，我们用行首缩进的写法来使条件部分更易于阅读。如果不用行首缩进写法，操作符的条件部分就可写成

```
B=0 AND((M=0 AND C=1)OR(M=2 AND C=2))
```

操作符的描述名称已见于表 2-4 中，在本表中不再重复，然而号数仍和表 2-4 同。建议读者最好检验一下每个操作符，了解为什么这些行动只有当条件成立时，才能产生合法状态。

表 2-8　传教士和野人问题中的 10 个操作符

条　件	行　动
1. B=1 AND(M=3 AND C=2) OR(M=1 AND C=1))	LET M=M−1 LET B=0
2. B=1 AND C≥1 AND(M=0 OR M=3)	LET C=C−1 LET B=0
3. B=1 AND C≥1 AND M=C	LET M=M−1 LET C=C−1 LET B=0
4. B=1 AND((M=3 AND C=1) OR(M=2 AND C=2))	LET M=M−2 LET B=0
5. B=1 AND C≥2 AND(M=0 OR M=3	LET C=C−2 LET B=0
6. B=0 AND((M=0 AND C=1 OR(M=2 AND C=2))	LET M=M+1 LET B=1
7. B=0 AND C≤2 AND(M=0 OR M=3)	LET C=C+1 LET B=1
8. B=0 AND C≤2 AND M=C	LET M=M+1 LET C=C+1 LET B=1
9. B=0 AND(M=0 AND C=2 OR(M=1 AND C=1))	LET M=M+2 LET B=1
10. B=0 ANE C≤1 AND(M=0 OR M=3)	LET C=C+2 LET B=1

第三章　状态图搜索

试图解决问题的一个方法是由初始状态出发进行试探，以期找到一条通往目标状态的路径。我们称这种试探为状态图搜索，也称之为状态空间搜索。状态图也叫状态空间。

一、搜　索　树

图 3-1 表示一个迷宫问题，其目的是要找到一条由起点到指定终点的通过迷宫的路径。

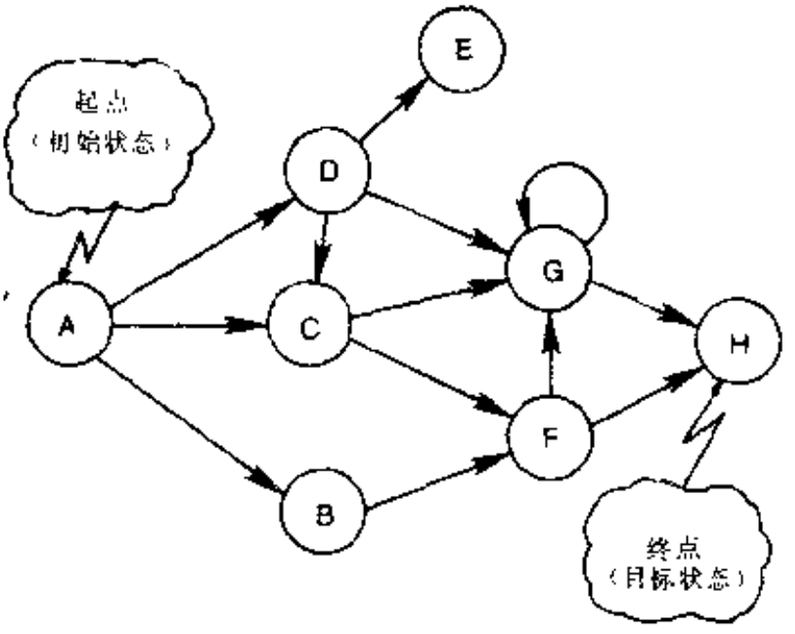


图 3-1　迷宫问题。此问题是要寻找一条由起点到指定终点的路径，按此法画的迷宫图即为其本身的状态图

迷宫当然是一个图，而且就是其本身的状态图。如图 3-2 所示，按照通常习惯画出的迷宫，很容易转换为状态图的形式。由上一章我们知道，对于任何一个问题都可以作一状

态图，所以如果我们愿意的话，可以将任何一个问题都看作是一个迷宫问题。

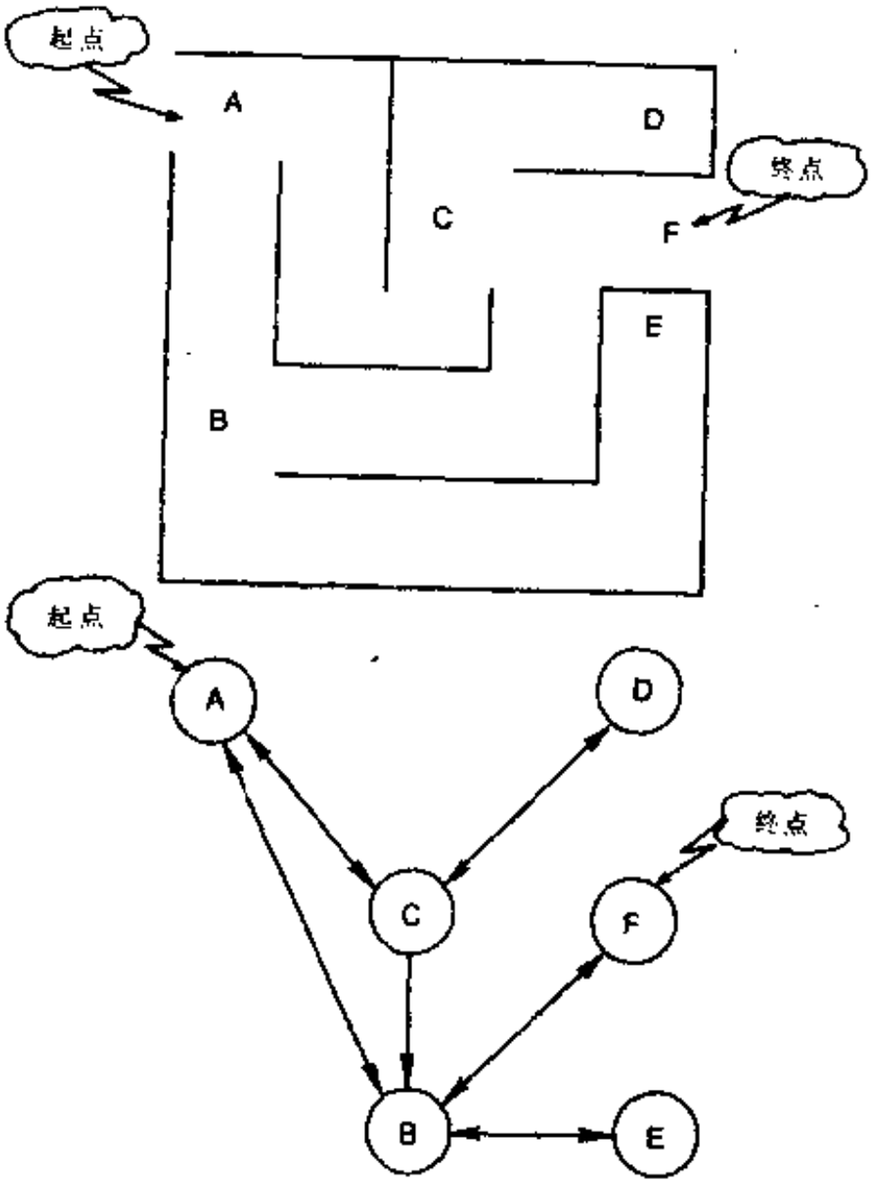


图 3-2　按照通常习惯画的迷宫也很容易转变成状态图。（一个两端带有箭头的弧线是在相同的两个节点间，方向相反的两条有向弧线的简略画法。这样的弧线表示两个方向都有通路可走）

假定现在我们由起点出发，开始寻找穿过迷宫的道路。

我们面临着什么情况呢？有好几条路径由起点通往其它节点。如果我们沿其中的一条来到下一个节点，就又会发现同样的情况：即有若干条路径通往其它节点。

简单地说，我们面临着一种重复分枝的情况，只要我们所沿着的路径一遇到节点，就又分为几条不同的路径。当这些节点的每一路依次遇到节点时，这种分枝就重复出现，如此等等。用树来表示重复分枝的情形是再好不过的了。

(一)树

如图 3-3 所示，计算机科学中的树与自然界里的树是非常类似的，只不过它是倒过来画的，根在顶上而叶在底下。

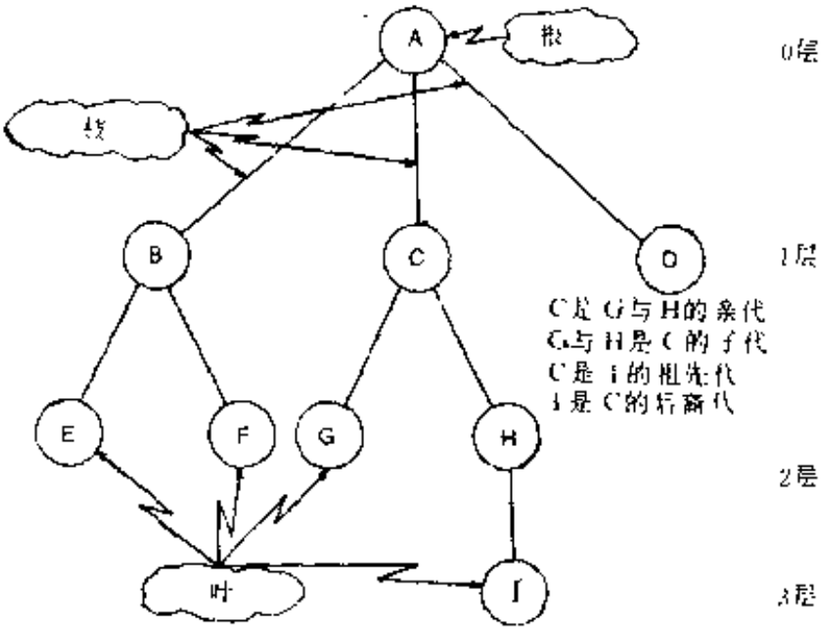


图 3-3　树的术语

从技术上讲，树是个有向图，其中每个节点最多只有一个前驱。恰好只有一个节点没有前驱，此节点称之为根。没有后继的节点称之为叶，而弧线称之为枝。虽然树是一个有向图，但其箭头常常省略，因为这些箭头总是由顶端指向底部的。

树中的每一个节点都处于某一层。根处于 0 层，根的后

继处于 1 层，根的后继的后继处于 2 层，如此等等。在人工智能中，一个节点的层数常称之为深度。树根的深度为 0，它后继的深度为 1，如此类推。当把层次看作表示深度时，我们说根是处于最顶层，根的后继处于次低层，如此等等。

一个节点的后继就是它的子代，子代共同的前驱就是它们的亲代。具有相同亲代的子代是同胞或孪生。

子树是另一个较大的树的一部分。如果树的任一节点连同它的子代，以及子代的子代在内，我们就得到一个子树，开始取的这个节点即为子树的根。

(二)搜索树

搜索树表明了状态图中哪些路径是已经试探过了的。当开始搜索时，搜索树只包括一个节点，即状态图的起始节点。在试探不同路径时，搜索树不断生长着。当搜索结束时，搜索树就表明搜索期间已被试探过的每条路径。

图 3-4 表明了图 3-1 中状态图的一种可能的搜索树。此搜索树的各节点相应于其状态图的各节点，并且相应于问题

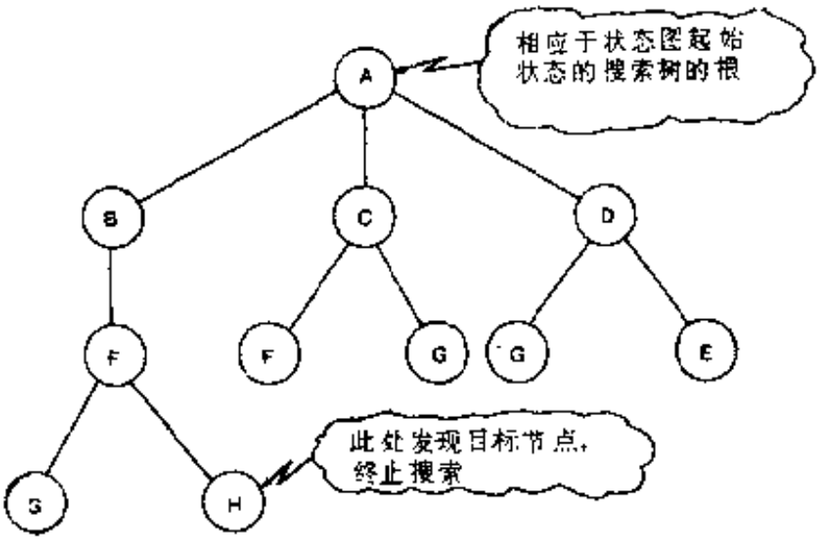


图 3-4　建立在图 3-1 状态图基础上的搜索树

世界的各状态。搜索树的枝相应于状态图中的弧线，并且相

应于问题中的操作符。

搜索树的根是状态图的起始节点，搜索树从根到叶的一条路径相应于状态图中一条已经试探过的路径。

叶是一个由此不再发出弧线的节点，一个节点成其为叶有四种原因：

1. 其状态图中相应的节点是死端，由它不再发出弧线。

2. 节点是一个目标节点。当找到一个目标时，搜索就终止。

3. 由该节点发出的弧线已经搜索过了。在状态图中，由起始节点到达某一节点，可能不止一条路径。因此，该节点在搜索树中可能不止一次出现。如果由一个节点发出的弧线在第一次遇到该节点时，已经搜索过了，那么再次遇到该节点时，就没有必要再去搜索那些已经经过了的弧线。

4. 在目前阶段，搜索还没进行到对所讨论的节点发出的弧线施行搜索。

由根到目标节点(叶节点)的一条路径构成了该问题的一个解。

对状态图的搜索有几种不同的策略。不同的策略可以试探不同的路径，即使在试探同一路径时，这些策略通常也不会以相同的顺序来试探。搜索树"生长"的确切方式取决于所采用的搜索策略。

让我们更加仔细地来看看生长过程。通常状态图仅仅是以隐式给出的。我们并没有把整个状态图预先贮备好在某处，但是对于一个给定的状态施用全部可能的操作符，就能产生一个给定节点的所有后继。在搜索树上就可以产生一个给定节点的全部子节点。当一个节点的子节点均已产生，且附加在树上时，则称此节点为扩展了的节点。搜索树是通过扩

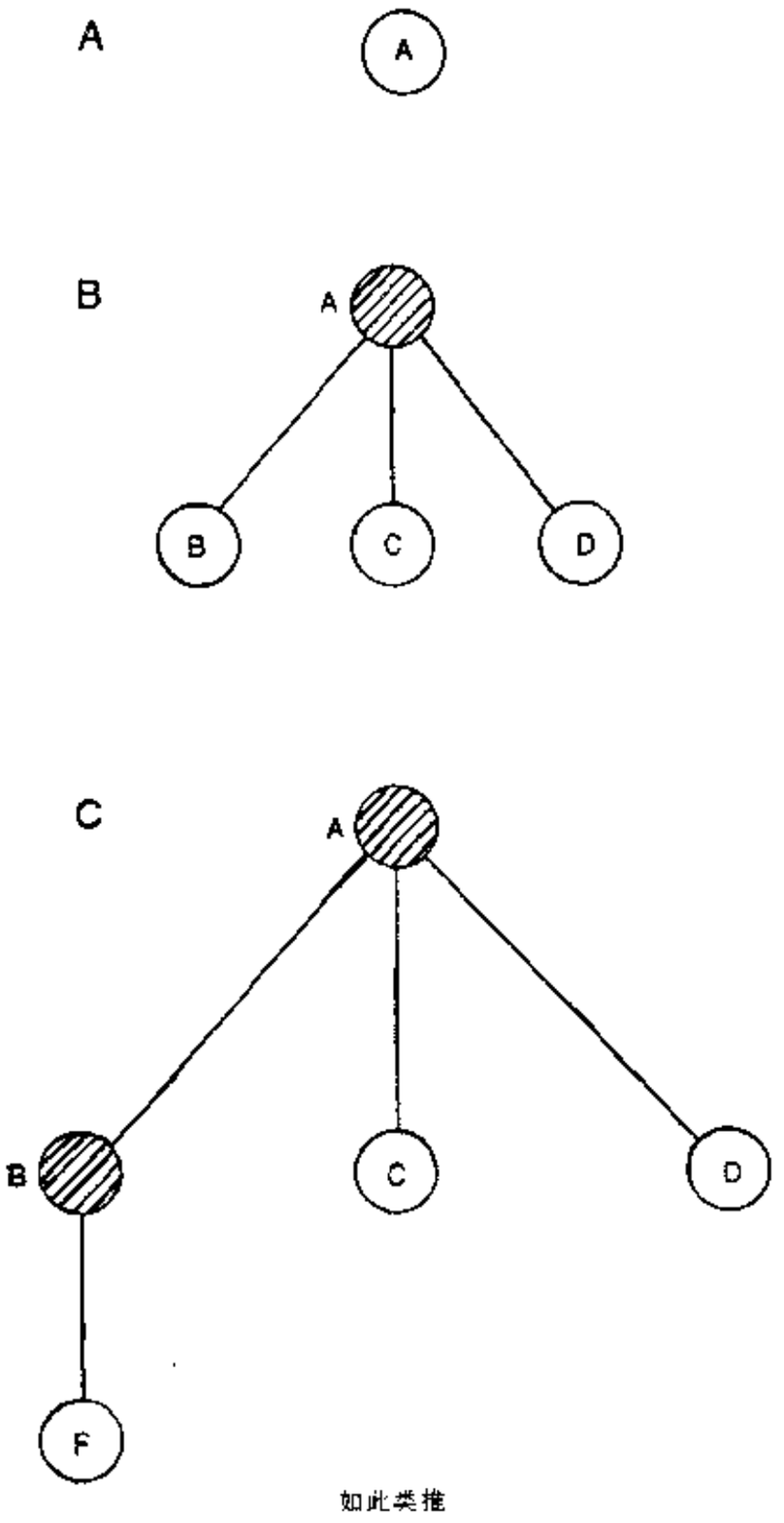


图 3-5　搜索树通过扩展节点来生长

展节点来生长的。如图 3-5 所示。

将搜索树的节点分为开的或闭的两类是很有用的。开节点是指对其扩展的可能性尚未进行检验，而闭节点则已经检查完毕。节点是可以扩展的，除非它是目标节点，或是死端，或者相应的状态图节点已在搜索树的其它地方扩展过了。图 3-5 中开节点未画阴影线，而闭节点画了阴影线。

一个开节点总是搜索树的一个叶。一个闭节点一般不是一个叶。但是，由于上述原因之一，闭节点不能进行扩展的话，它就可能是一个叶。

全部搜索都是通过扩展开节点来进行的。所采用的特定的搜索策略决定了开节点的扩展顺序。

二、宽度优先搜索

在宽度优先搜索中，搜索树的节点是一层一层地检查的。只有在上一层上的每一个节点都检查完毕之后，这一层的节点才能开始检查。

图 3-6 表示宽度优先搜索。我们由起始节点开始，它是开节点。这个节点被扩展后，产生了第一层的节点。每个第一层的节点现在已检查过了，那些已被扩展的节点产生第二层的节点。当所有的第一层节点都检查完毕，我们就移到第二层的节点上去。这个过程一直进行下去，直到找到目标节点或者直到不再有开节点为止。

宽度优先搜索可用称为队的数据结构来处理，队具有排队的特性。节点由排头移去加到排尾。

有关队，堆栈，树和图这些数据结构在计算机内部如何表达的详细内容可在我的《计算机业余爱好者的微处理机程序编制》(TAB 丛书 1977)一书中找到。

宽度优先搜索由队中单个节点，即起始节点开始，当一个节点已经扩展时，它的子代放在排尾。要检查的下一个节点总是从排头来取。因此，队就是这样一个等待次序，其中开节点轮流等待检查并转成闭节点。

宽度优先搜索最重要的优点是：它总会找到由起始节点到目标节点的最短路径。这条路径是由起始状态转变到目标状态所需要操作或运算最少的解答。

最短路径的解答往往是最理想的解答，但又不尽然。在现实世界的问题中，所付代价往往是与所选用的每一种操作有关。最短路径的解答不一定是最便宜的解答，因为它调用操作所需的代价可能要比较长路径解答的代价大得多。

如果我们设想用汽车考察一个国家的某一地区，就能看到宽度优先搜索的另一个问题。假设我们在 A 城，取某一条道路驱车前往 B 城。宽度优先搜索要求我们要继续下去就须返回 A 城，而后再由另一道路离开 A 城。但一般的常识告诉我们返回 A 城是浪费时间的，所以我们应该选取一条未走过的道路离开 B 城而继续下去。

在计算机搜索中会产生类似的问题。在现实世界的问题中，状态的描述可能是极其复杂的，以至于在内存中没有足够的单元来存贮这许多状态的描述。然而操作符可能是非常简单的，因为每种操作符可能只对复杂状态中的一小部分起作用。

对这个程序设计问题的方案是只存贮一种状态的描述。在开始时，它可能就是初始状态的描述。搜索树的存贮方法就是存贮与每个分枝相连系的操作符。

当我们通过搜索树工作时，1 们把各个分枝上遇到的操作符应用于单个状态的描述。当我们到达某一节点时，初始

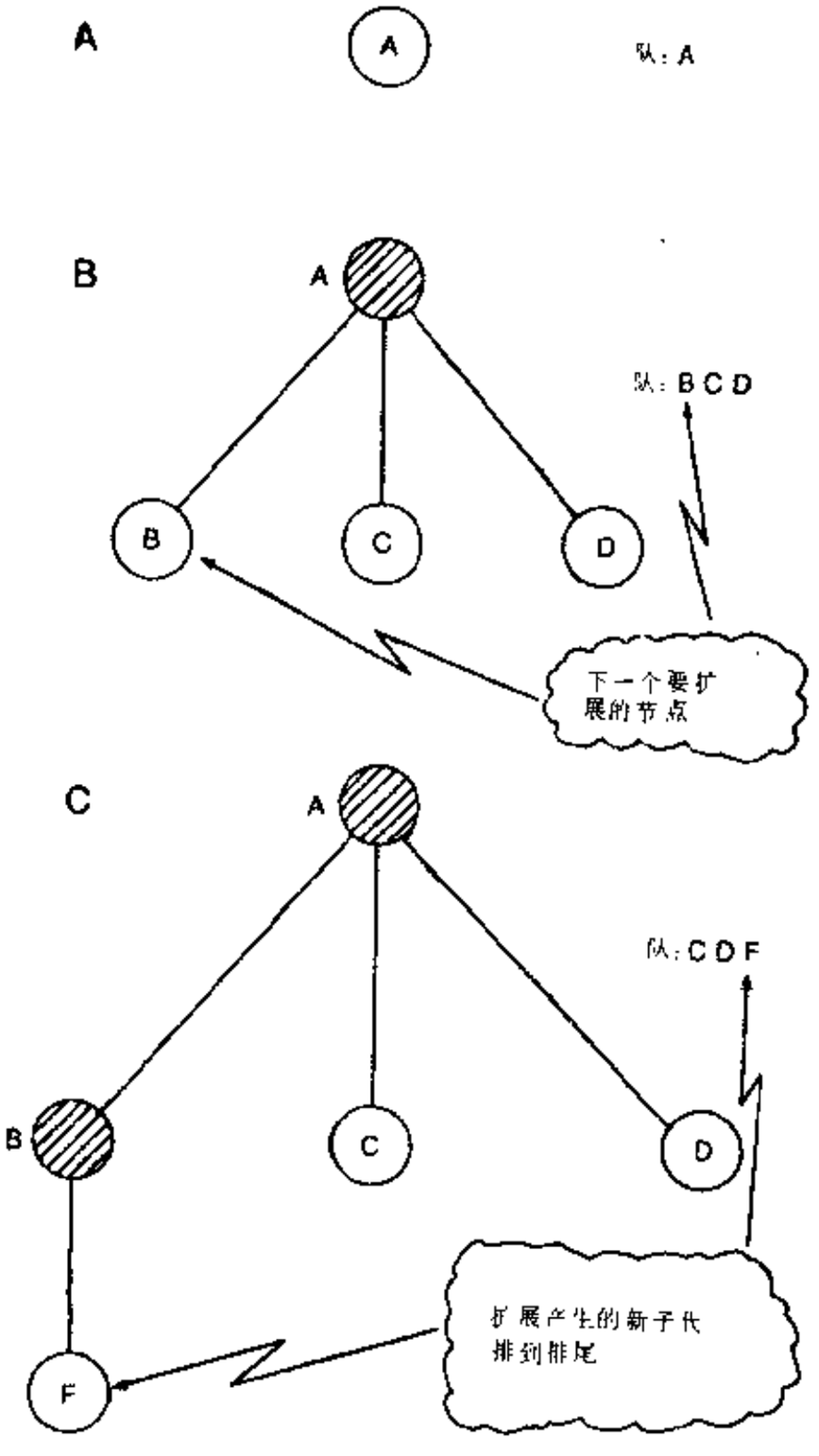
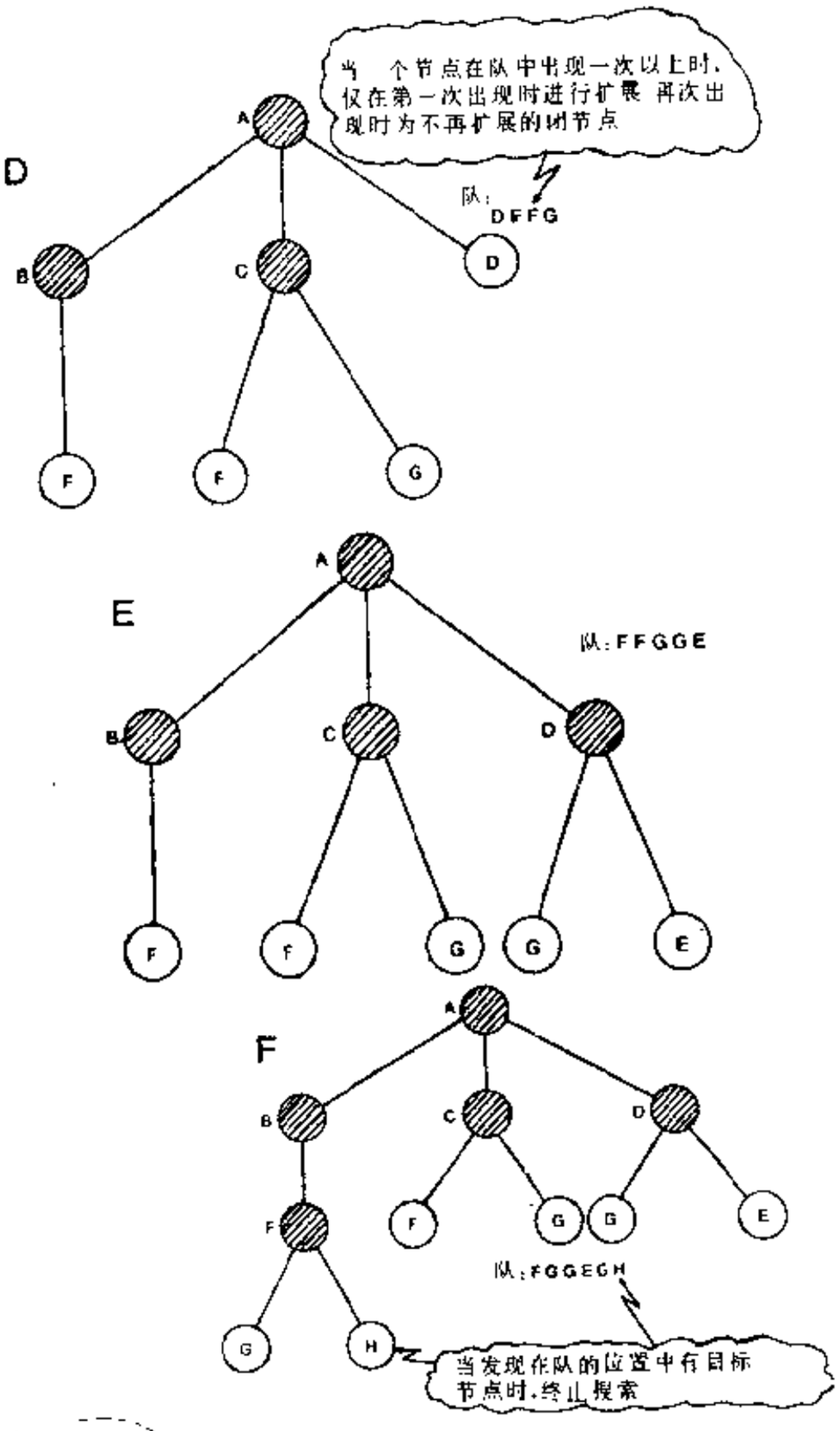


图 3-6　宽度优先搜索。


当一个节点在队中出现一次以上时，仅在第一次出现时进行扩展，再次出现时为不再扩展的闭节点
D
队：DFFG
A
B
C
D
F
F
G
E
队：FFGGE
A
B
C
D
F
F
G
G
E
F
A
B
C
D
F
F
G
G
E
队：FGGEGH
G
H
当发现在队的位置中有目标节点时，终止搜索

排头在左边，排尾在右边

状态的描述就会转变为对该节点的描述。为要从亲节点到子节点，我们在从亲节点到子节点的分枝上应用操作符。为要从子节点到亲节点，我们在反方向应用了同样的操作符。

为了进行宽度优先搜索，我们必须从亲节点到子节点应用一个操作符，在反方向再应用该操作符返回亲节点，然后又用另一个操作符到一个新的子节点等等。总是这样返回亲节点看来是浪费时间的。似乎最好是按照由亲代到子代的方向尽可能地进行下去，只有在找不到继续往下走的路径时才按原路返回。

三、深度优先搜索

总是向着由亲代到子代方向进行，直到不得不返回追踪的搜索，称之为深度优先搜索。

我们能够用堆栈代替排队的方法，将宽度优先搜索转变为深度优先搜索。堆栈是一种数据的结构，其特性好象是桌子上的一叠纸或一叠卡片。堆栈有最顶层和最底层。我们可以把一些项放到堆栈的顶层，也可以从其顶层移去一些项。而由堆栈移去的一些项总是与其放入堆栈的次序相反。假如说排队的法则是"先来的先处理"或是"先进先出"的话，那么堆栈的法则就是"后来的先处理"或是"后进先出"。

图 3-7 表明深度优先的搜索。我们由堆栈的起始节点着手。当扩展一个节点时，就将其子代放在堆栈的最顶层。下一个要检查的节点由堆栈顶层取出。

如果扩展目前正在检查的节点，它的子代就放在堆栈的顶层。下一个检查的节点将取自堆栈顶层。所以，我们按照由亲代到子代顺序进行。如果目前正检查的节点不是扩展节点，那么由堆栈顶层取出的下一个节点就是同一层的另一个节点

或是比目前这个节点更高一层的另一个节点。在移动到新节点的过程中，我们要按原路返回追踪要么停留在同一层上，要么返回到上一层，而不是在由亲代到子代的方向上移动。

在深度优先搜索中，扩展节点不能扩展的原因与宽度优先搜索相同，即节点是目标节点，节点是死端，或者是相应于同一状态的节点已在搜索树的其它地方扩展过了。

但是在深度优先搜索之中，我们还有不能扩展一个节点的另一个理由。对于许多问题，状态图包含无穷多种状态，或者状态多达 100^{100} 这样巨大的数字，实际上也可以认为是无限的。对于这样的图，深度优先搜索会永远或近乎永远在由亲代到子代的方向上进行下去而不返回追踪。然而一般若由开始状态搜索了一段相当长的距离而未发现目标节点时，我们就宁愿返回追踪，另外探索一条路径，而不是继续进行下去而离初始状态越来越远。

所以，对于深度优先搜索，我们通常对搜索树规定一个深度界限或最大深度。当检查一个节点时，如果其深度已等于深度界限，就不再扩展，而代之以返回追踪。如图 3-8 所示。这样就迫使搜索树扩展开来，而不让它只试探几条路径(也许只有一条路径)到很深的深度。

深度优先搜索使返回追踪次数减到最少，它最适用于上节所讲到的一种情况，即返回追踪包括耗费时间的复杂计算。

深度优先搜索所得到的解答可能不是最短路径的解答，但所得到的解答不会超过深度界限。小的深度界限可以迫使搜索去寻求那些短路径的解。如果所有解的路径长度均大于深度界限，那么就得不到任何解。更大的深度界限可以得到较长路径的解，而且增加得到某个解的机会。

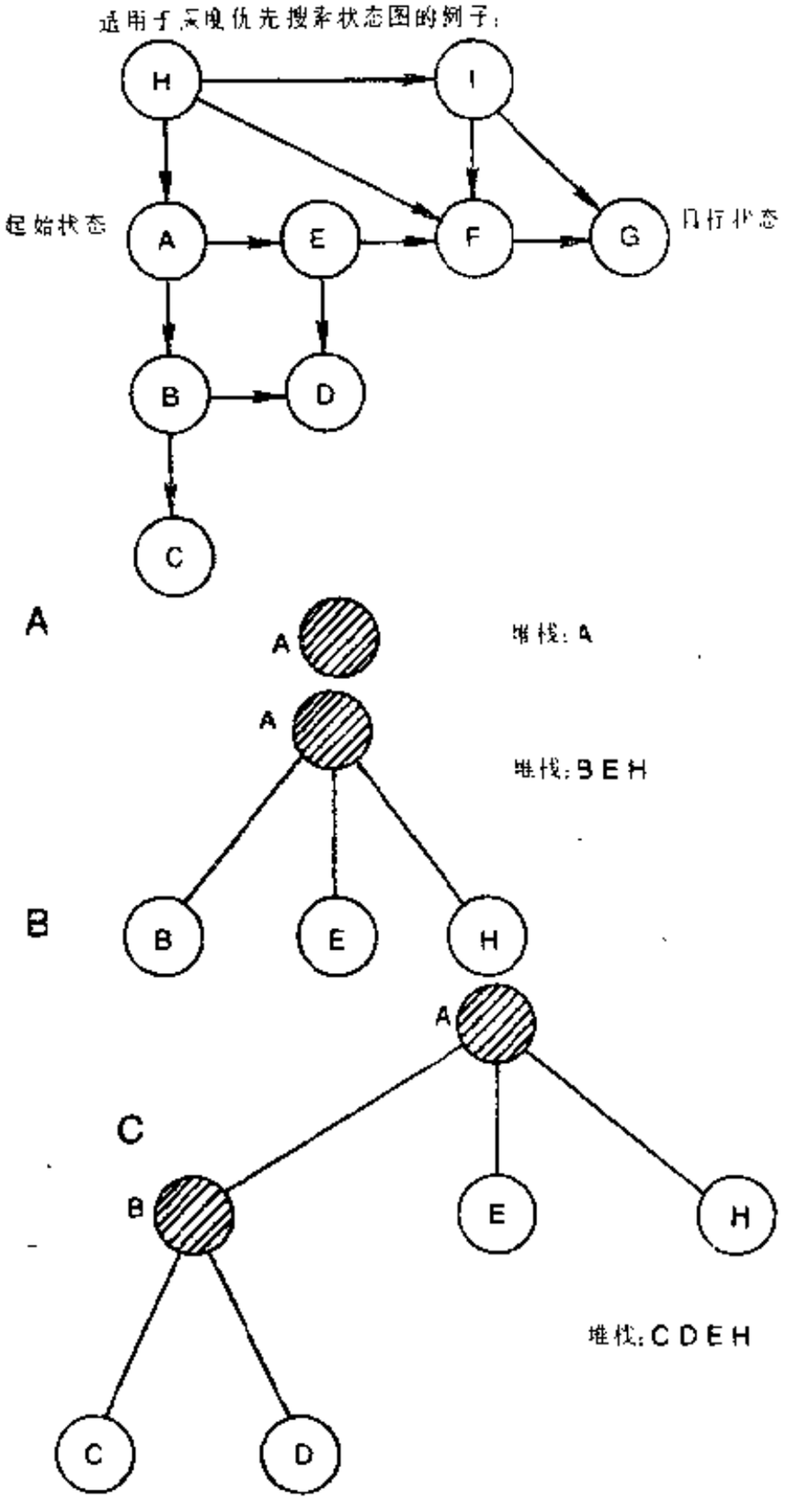
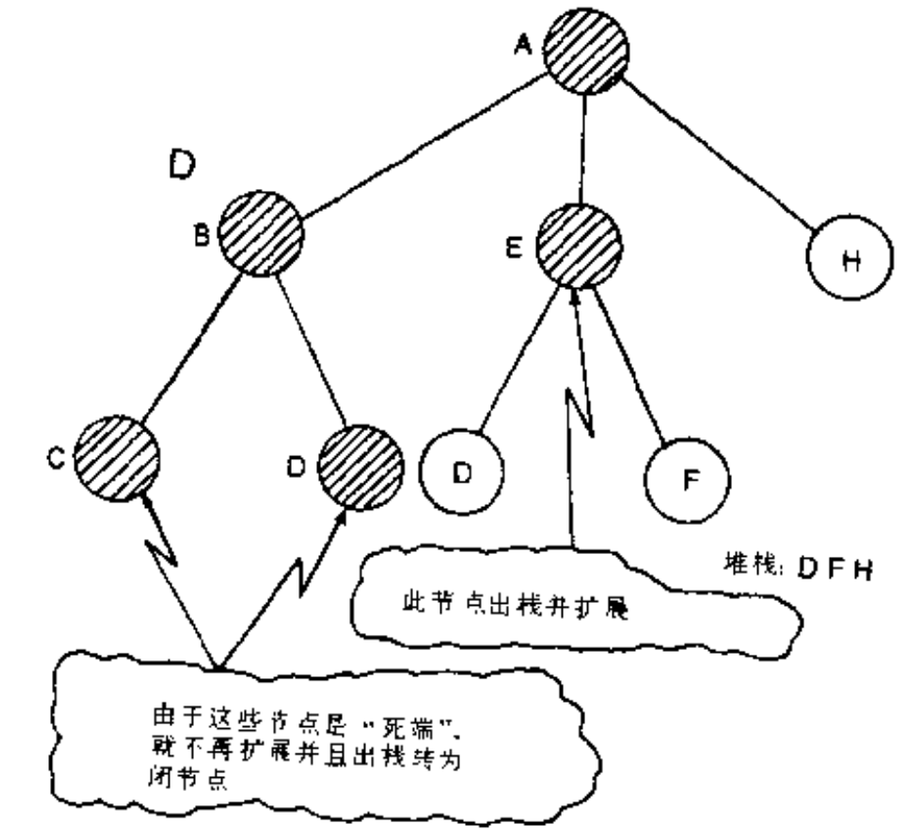
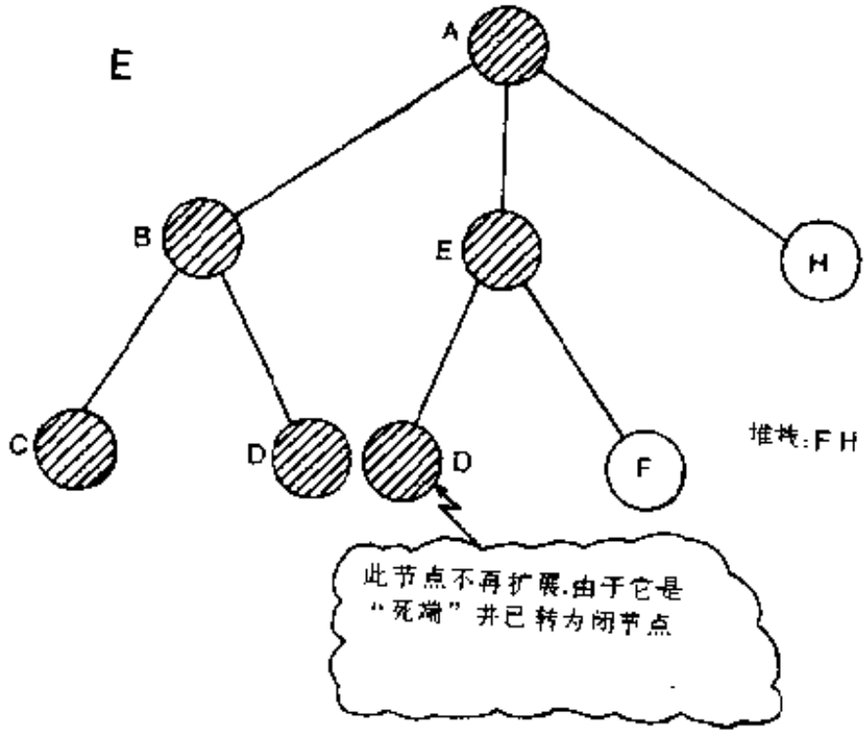


图 3-7　深度优先搜索，堆栈





顶层在左边，堆栈底层在右边。

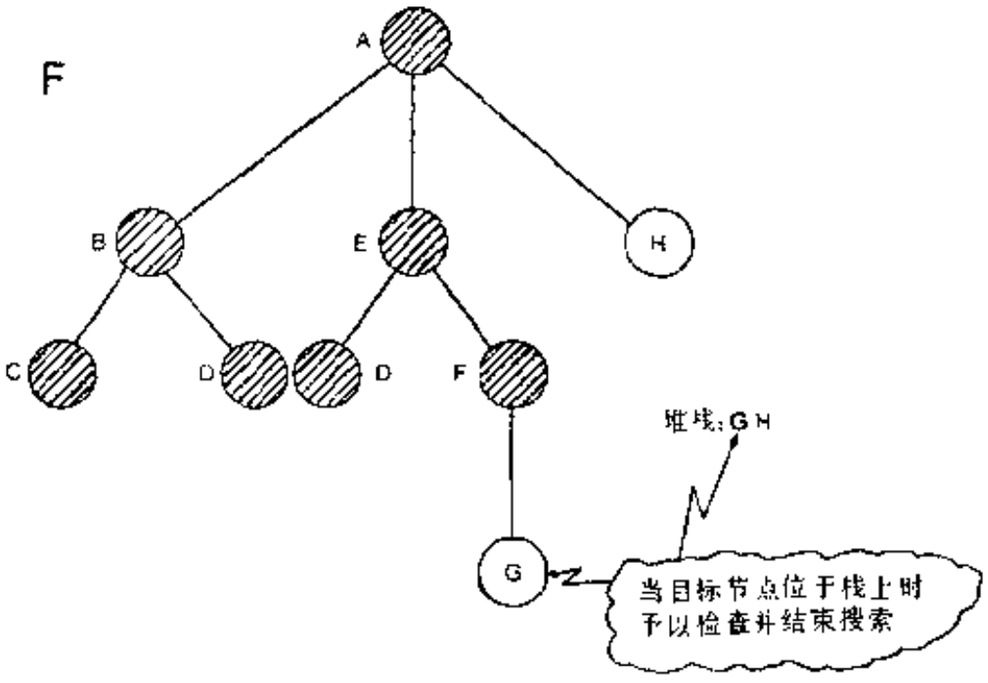


图 3-7(F 部分) 深度优先搜索的终局

深度优先搜索的程序经常可以用这样的方式来设计：必须存贮的节点仅仅是由根到目前正检查的节点路径上的那些节点。这些节点可以存放在一个堆栈中，根在堆栈的底部，而当前的节点在堆栈的顶部。堆栈中节点的最大数目为深度界限加 1。

因此，深度优先搜索可以用相当小的存贮容量来实现。在这种情况下，深度界限常常是由可利用的存贮容量来决定的。

四、有 序 搜 索

宽度优先及深度优先搜索二者均为盲目搜索之实例。因为所要检查的节点的次序是由搜索方法来决定而不是由相应状态的某个特性来决定的；而且也不打算从状态描述中抽取

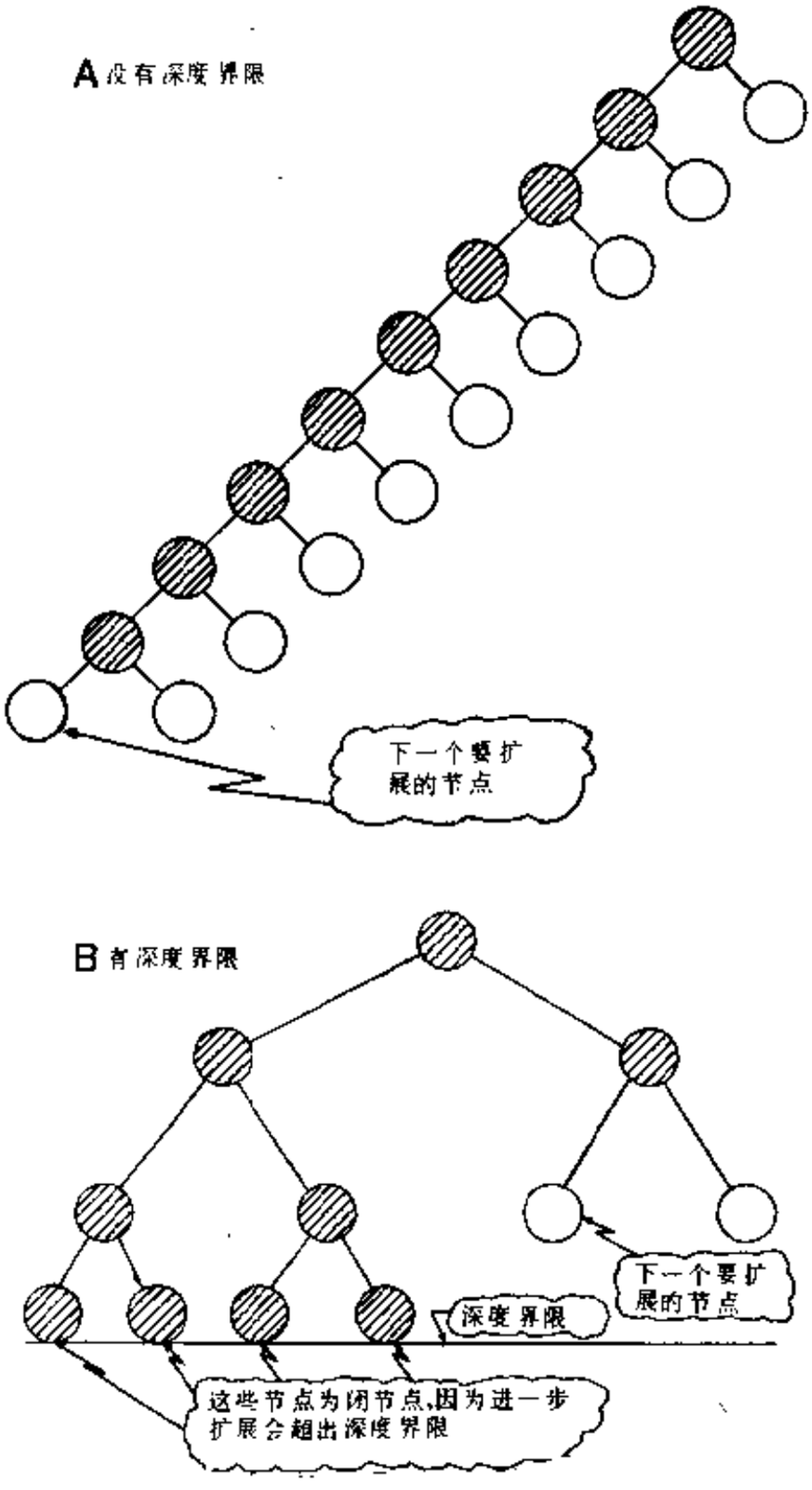


图3-8 深度界限迫使搜索树展开,而不是沿着一条路径到很深的深度

能够指出搜索正确方向的信息。在这一节中，我们将开始研究如何很好地利用这些补充的信息。

首先，我们要把状态图中一条路径的代价定义为该路径的第一个节点到其最后一个节点所需要的操作次数。这个代价就是该路径上弧线的个数。特别是，一条求解路径的代价就是由初始状态转变到目标状态所需的操作次数，也正是该求解路径弧线的个数。表 3-1 表明了图 3-1 中几条求解路径代价的计算。

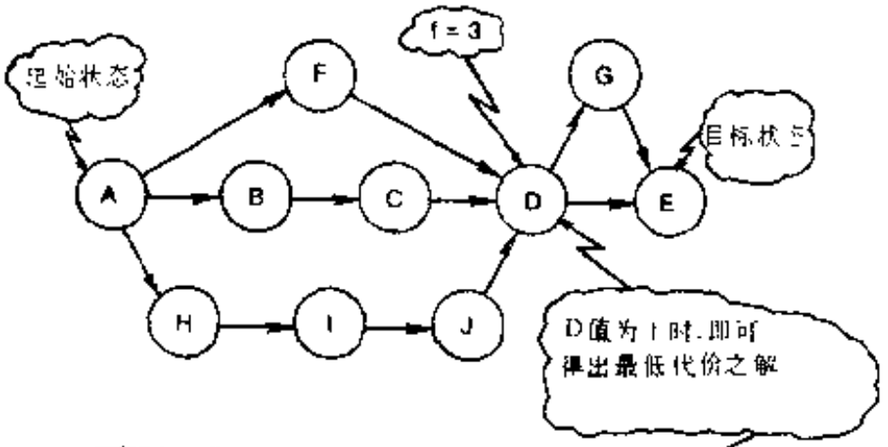
表 3-1　计算求解路径的代价

解　　答①	代　　价②
ADGH	3
ADGGH	4
ADCGH	4
ADCFGH	5
ACGH	3
ACFH	3
ABEGH	4
ABFH	3

① 顺序通过的节点。　② 所经弧线的数目。

现在来看看状态图中的一个特殊节点。如图 3-9 所示，可能有几条求解路径通过该节点。每一条求解路径都要有代价。我们感兴趣的是通过所讨论节点的代价最低的一条求解路径。用 f 来表示这个最低的代价。对任一节点，都能计算出一个 f 值。这个值一定就是代价最低的通过该节点的求解路径的代价。

假定我们现在正在状态图中搜索，而且必须从几个可能的节点中决定哪个要到下一个节点去。看起来比较合理的办法是选择位于最低求解路径上的节点。所以我们走 f 值最低的节点。根据搜索树，我们要选择 f 值最小的开节点来检查。



通过节点D的路径	路径的代价
AFDGE	4
AFDE	3
ABCDGE	5
ABCDE	4
AHIJDGE	6
AHIJDE	5

图 3-9　给定节点的 f 值为通过该节点的最低代价求解路径的代价

遗憾的是，在搜索过程中遇到一个节点时，我们不知道是否有求解路径通过该节点，更不必说最低代价路径了。尽管我们不知道 f 的准确值，但可以估计这个值并用以指导搜索。

为了有助于估计 f 值，我们把 f 分为两部分。如图 3-10 所示。通过给定节点的最低代价求解路径，本身可以分为两部分——一部分由起始节点到所讨论的节点；一部分由所讨论的节点到目标。令 g 为由起始节点到所讨论节点的路径之代价，令 h 为由所讨论节点到目标的路径之代价，整个路径的代价由下式给出：

$$f = g + h$$

在搜索树中，我们可以对每一个节点进行精确的计算，如图 3-11 所示。对于搜索中一个特定的节点，有一条唯一的路径从树的根到那个节点。搜索树中的这条路径相当于在状态

图中连结起始节点至相应的状态图节点的 g 值。这个 g 值恰好等于该节点的深度。

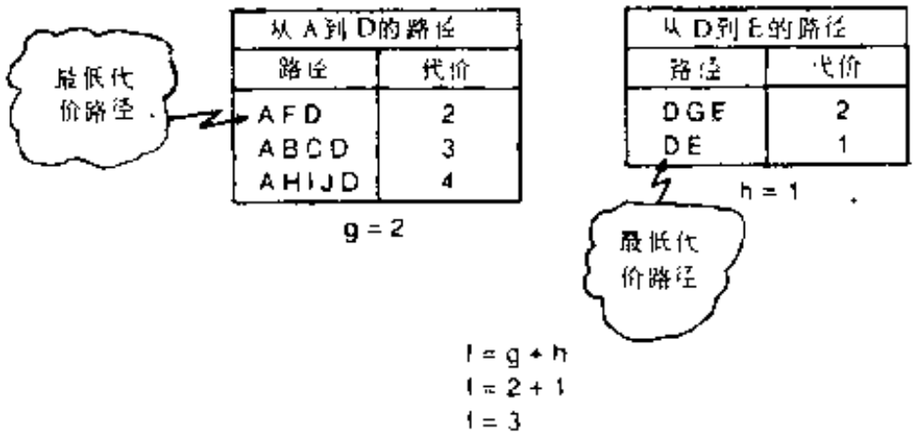


图 3-10　我们可用 g 与 h 之和来表示 f，此处 g 为由起始节点到所讨论节点最经济路径的代价，h 为由该节点到目标节点最经济路径的代价

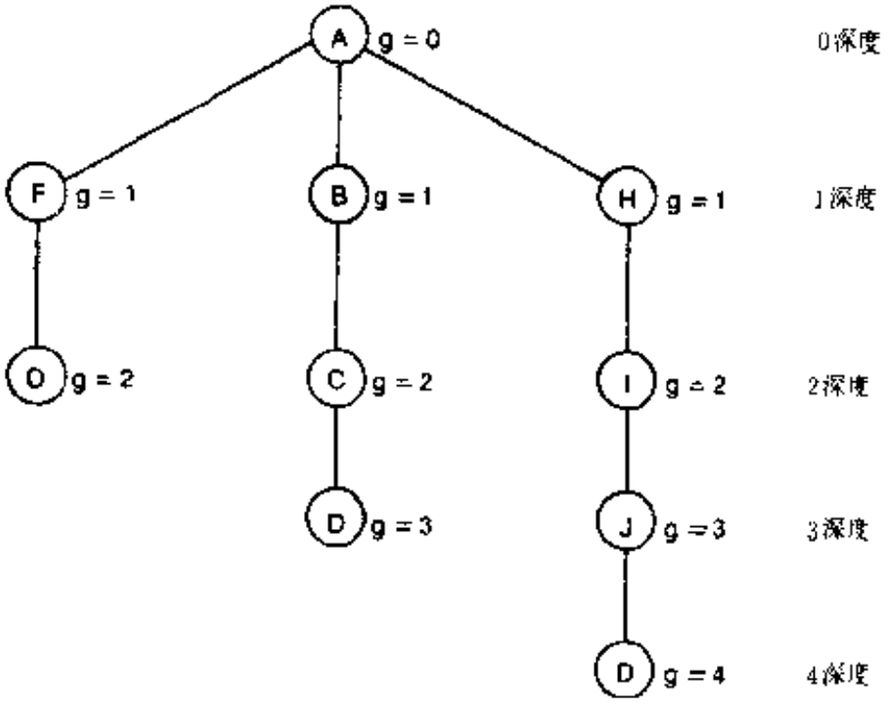


图 3-11　搜索树中每一个节的 g 值可进行精确的计算。g 的值就是该节点的深度

因为从起始节点常常可以有几条路径到达给定的状态图节点，所以相应于同一状态图节点可以有几个搜索树的节点。对每个搜索树节点，都可以计算其 g 值。我们并不认为搜索树中由根到节点的路径中无论哪一个都相应于状态图中起始节点到节点的最短路径。因此对不同搜索树的节点所算出的 g 值是对状态图节点的不同估值，其中最小值才是最佳估值。

因此，在进行搜索时，我们把由搜索图节点所算出的 g 值用来作为相应状态图节点的估值。图 3-12 表明了这点。

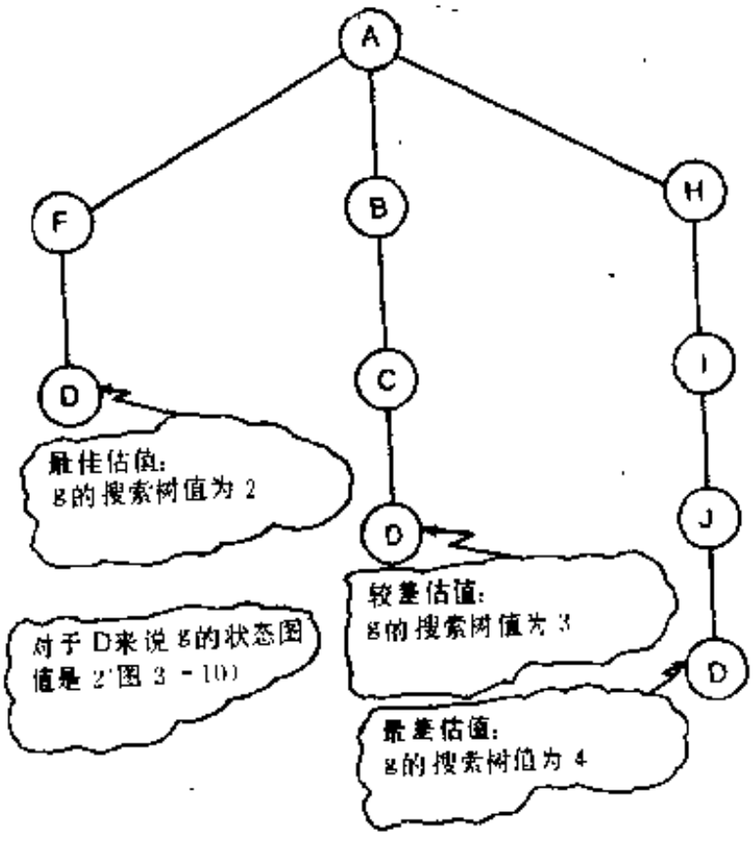


图 3-12　相应于给定状态的搜索树的节点所算出的 g 值为相应状态图节点 g 值的估值

与 g 相反，没有办法由搜索树来估计 h 值。我们最感兴趣的是对开节点 h 值的估计。因为开节点是尚未扩展过的，

所以搜索树不能告诉我们从这个开节点出发到达目标节点所走路径的任何情况。

唯一能帮助我们估计 h 值的就是其状态描述本身。我们研究状态描述并试图估计出将这一状态转变为目标状态需要多少操作。这一估计即为 h 的估值。

表 3-2 和图 3-13 说明了对上一章研究过的两个问题估计 h 的方法。

对于“传教士和野人”的问题来说，仍旧留在左岸的传教士和野人的数目就是对尚待完成工作量的一种估值。送一个野人过河然后让船返回，以便再送另一个野人过河，船的往返是两次，一次过河，一次返回。所以，左岸野人数的两倍是把这些野人送过河从而达到目标状态所必需的操作次数的估值。我们可利用这一值作为对 h 值的估值。这仅仅是一种粗略的估计，因为它忽略了为确保传教士不被野人吃掉所必须的策略与计谋，尽管如此，它还是能使搜索集中于送野人过河这一当前任务上。对于梵塔问题来说，我们的注意力集中于柱 3，而忽略在其它两柱上必须进行什么样的操作，这样我们就能够很容易地定出只涉及柱 3 到达目标状态所需的操作次数。我们取这个次数作 h 的估值。当然，这是一个很低的估值，因为我们忽略了在柱 1 和柱 2 上的全部操作。

显然，估计 h 值的方法取决于一个特定问题的规定细节。我们还无法得出适用于所有问题的 h 估值的一般方法。对于每一个特定具体的问题，我们必须研究状态描述的特征，看看哪一种特征能提供关于从目标状态到目前状态有多远的某种线索。

由状态描述所抽取的用来估计 h 值的信息称之为“启发

表 3-2　传教士和野人问题中的 h 估值

左　岸			右　岸			h 的估值
传教士	野人	船	传教士	野人	船	
0	1	有	3	2	无	2
0	2	有	3	1	无	4
0	3	有	3	0	无	6
1	1	有	2	2	无	4
2	2	有	1	1	无	8
3	1	有	0	2	无	8
3	2	有	0	1	无	10
3	3	有	0	0	无	12
0	0	无	3	3	有	0
0	1	无	3	2	有	2
0	2	无	3	1	有	4
1	1	无	2	2	有	4
2	2	无	1	1	有	8
3	0	无	0	3	有	6
3	1	无	0	2	有	8
3	2	无	0	1	有	10

式信息”。因为它能帮助我们发现问题的解答。我们前面说过，启发术就是一种解决问题的经验法则。估计 h 的方法可以看作是确定给定状态距目标状态有多远的一种经验法则。

与所有启发方法一样，对 h 的估值也受误差影响。即我们的经验法则可能对一个距目标状态较近的状态错误地估计一个较高的 h 值，而对另一个距目标状态较远的状态却估计了一个较低的 h 值。当出现这种情况时，函数 g 迟早就会起作用。如果不适当的 h 值把我们引到徒劳的搜索中，那么沿着徒劳搜索路径 g 值的增加很快会迫使我们返回并换条路径再试，即使这样做会走到并不具有最低 h 值的节点去。

梵塔	
柱 3 上的盘	h的估值
无	2
A	3
B	1
AB	0

当估值与准确值不同时，
准确值在括号内表示

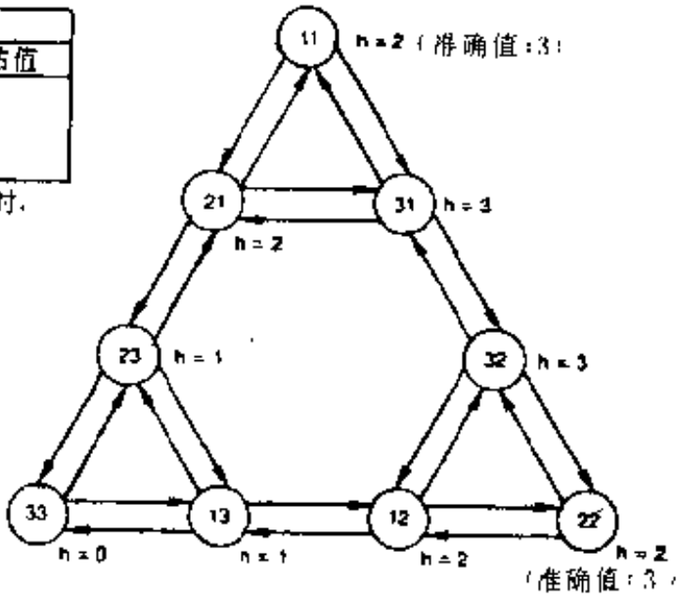


图 3-13　梵塔问题中的 h 估值

估计 h 值的函数常称之为评价函数，因为它评价给定状态的方法是估计转变该状态到目标状态所需的代价。当我们研究评价函数起着重要作用的博弈游戏程序时，再来谈评价函数的问题。

现在让我们来看看如何执行一个由 f,g,h，的估值来指导的搜索。这样一个搜索称为有序搜索，因为 f 的估值决定了检查开节点并转变成为闭节点的次序。

图 3-14 表示有序搜索。在图 3-15 和下面的讨论中字母 f,g 和 h 就代表 f,g 和 h 的估值，但我们不再予以说明。

我们由状态的起始节点开始搜索，该节点即变为搜索树的根。对根面言 g 值为 0。我们利用已发现的任一经验法则来计算 h 值。f 值为 $g+h$ 之值，由于 g 值为 0，所以 f 值就是 h 值。

当扩展一个节点时，我们对于它的每一个子节点计算 f

值。由于子节点的 g 值总是比其亲节点的 g 值大 1，所以我们能够很容易地计算出 g 值。h 值用经验法则来计算。与通常一样 f 利用公式 $f=g+h$ 来计算。

当选择一个开节点来检查时，我们总是选 f 值最小的那个节点。要是有几个节点具有相同的 f 值，就任选一个，如果其中有一个是目标节点，那么就应该选择目标节点。

除了要计算 f 值以外，有序搜索与宽度优先搜索和深度优先搜索类似。队和堆栈可用一个表来代替，在表中，开节点按照它们的 f 值来排列，f 值较小的节点靠近表头，f 值较大的节点靠近表尾。于是表中第一个节点就是 f 值最小的节点，也就是要选来进行检查的下一个节点。

当一个节点进行扩展的时候，它的每一个子节点都放入表中。每个子节点都按照其 f 值插入适当的位置，以保持表的次序。

由于下一个要取出的项总是具有最高的优先权的那一项，而与各项加入时的次序无关，所以象这样的表有时称为优先次序队。这里"最高优先权"就意味着 f 值最小。

如果从优先次序队取出待检查的节点就是目标节点，则搜索终止。如果它是个死端，则就关闭而不再扩展。如果此节点与搜索树其它任何节点所对应的状态都不同，则它就扩展。

如果被检查节点和搜索树中某个其它节点所对应的状态相同，我们所必须采取的步骤就比宽度优先搜索或深度优先搜索复杂得多。

如果被检查节点的 g 值大于或等于相应于同一状态的其它节点的 g 值，则被检查的节点关闭并不再扩展。

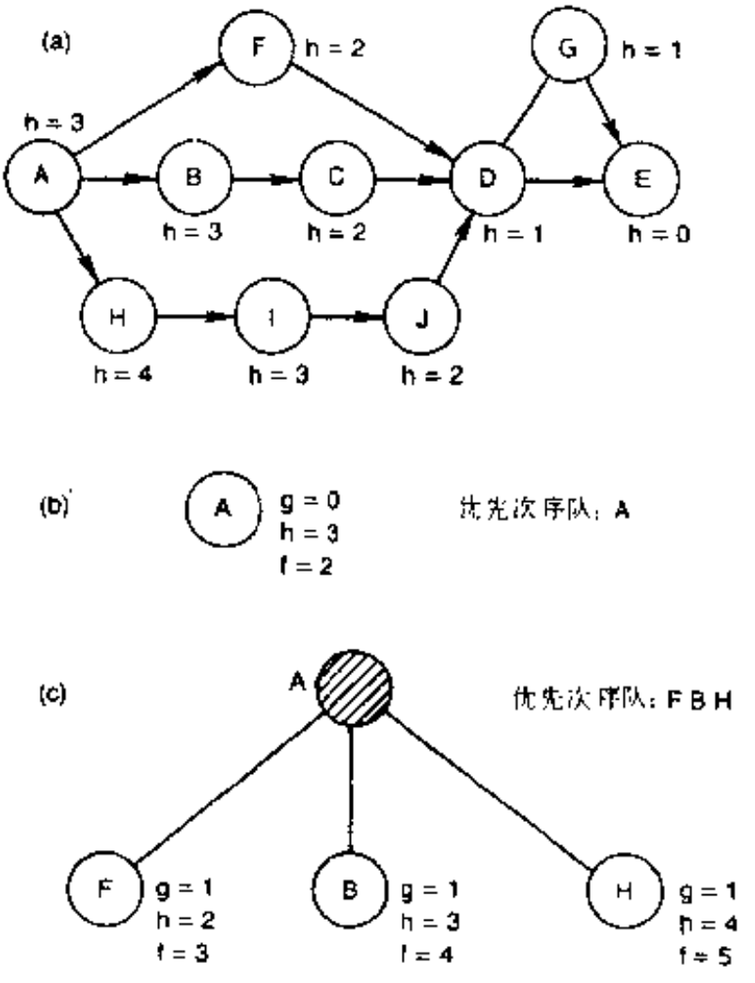
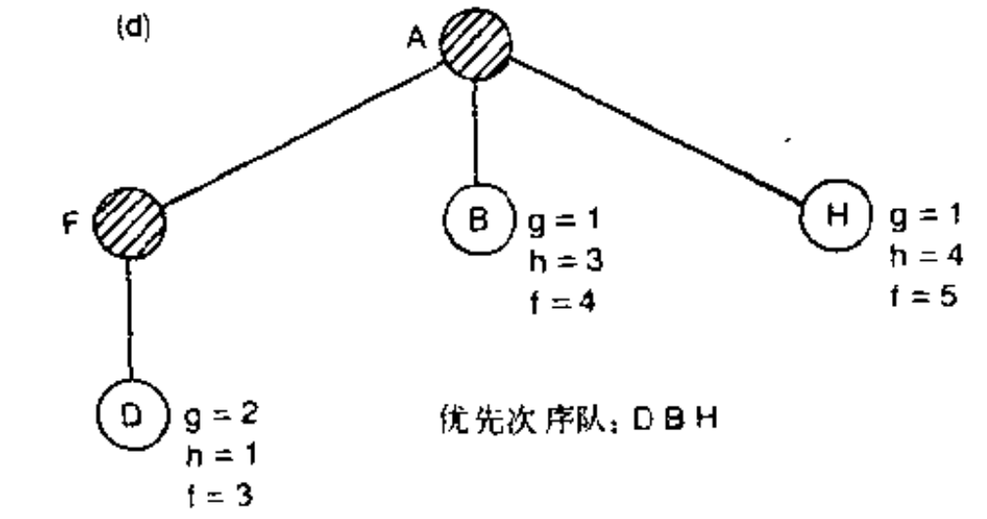
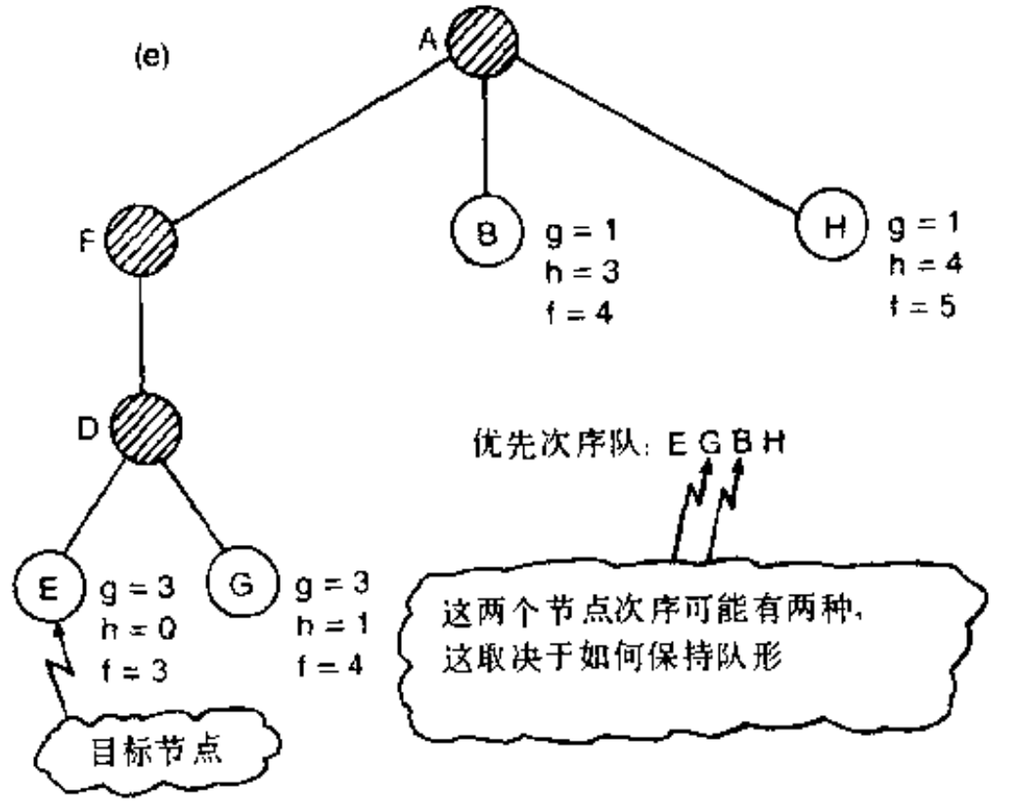


图 3-14　有序搜索。对 f 值最低的节点进行下一步扩展。(*a*)中给

在更实际的问题中，经常不知道准确值，而是利用如表 3-2 和图





出所用的 h 值。由于本问题较简单，所以这些值就是准确值。3-13 中那样的启发估值。搜索树计算出的 g 值也要用到

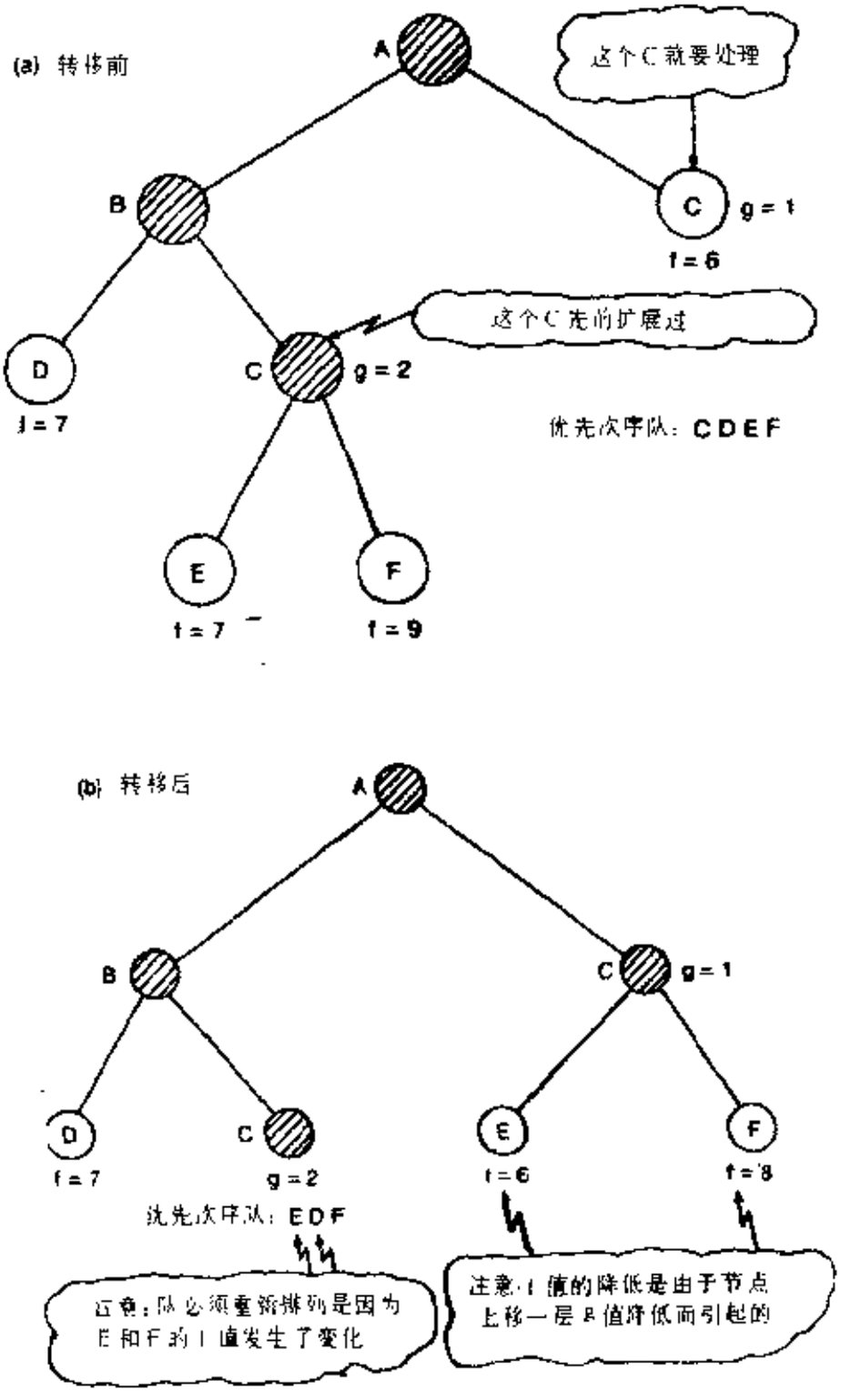


图 3-15　如果扩展了一个节点之后，发现另一个节点相应于同一状态并具有较低 g 值，则将已扩展过节点的子代转移到这个新找出的节点上

但是假设被检查节点的 g 值小于相应于同一状态的任何其它节点的 g 值，则该被检查节点就进行扩展。如果相应于同一状态的其它节点之一已扩展过，则由该节点取出其子节点并附于被检查节点之上，如图 3-15 所示。现在必须对被检查节点的所有后代重新计算 g 值。

这种处理有些复杂化，其目的是为确保相应于同一状态的所有节点中 g 值最低的节点得到扩展。（这个 g 值是由起始状态到所讨论状态的最短路径代价的最佳估值。）如果我们扩展一个节点，并且发现相应于同一状态的另一个节点具有较低 g 值，那么我们必须重新构造搜索树，以便新发现的节点得到扩展。

有序搜索的显著特点是：利用以 h 估值表示的有关状态的详细信息来指导搜索向目标前进。g 的作用类似于深度优先搜索中的深度界限的作用，它防止搜索程序沿着一条无结果的路径走得太远。

为简便起见，我们已经假定对每一种操作符而言，实行一次操作的代价是相同的。但是有序搜索很容易推广到一般情况，也就是不同的操作符具有与其有关的不同的代价。

五、其它启发技术

还有几种其它的技术采用启发式经验法则信息以改善搜索的效率。

(一)子代的排序

当我们在宽度优先或深度优先搜索中扩展一个节点时，我们将这个节点的所有子代置于队或堆栈之中。而有关它们的放置次序问题却没有提到。如果我们能够利用启发信息来估计哪一个子节点最容易转化为目标节点，那么就可以在栈

或队中将子代排序，以便对最有希望的节点先进行检查。由于这种排序仅仅是对同一亲代的子代进行的，所以这种搜索方法的基本的宽度优先或深度优先的特性是不会改变的。

(二)子代生成的限制

至今为止，当一个节点扩展的时候，其所有的子节点都加到搜索树上，我们仍然用启发信息来估计哪一个子节点最容易转化为目标节点，然后只把少数最有希望的子节点加到搜索树上而舍去其余部分。当然，不应该忘记，启发方法也难免有错误，我们可能丢掉了最好的解，甚至是唯一的解必须通过的节点。

(三)树的修剪

在搜索程序运行了一段时间而没有找出任何解之后，搜索树将充满可利用的存贮空间，此时我们可以利用启发信息来决定哪些开节点是最有希望的，然后，可以剪除那些不通向有利节点的路径。这样就可以继续搜索，而树则能够向通过剪枝而收回的存贮空间中生长。和以前一样，我们也可能无意中剪掉了可以得出解答的唯一路径。

(四)终止

当达到深度界限时，深度优先搜索会终止任一路径的探索。启发信息可用来继续探索，但在有希望路径上比在没什么希望的路径上会继续进行得更深一些。

第四章　子问题、子目标和计划

有许多问题太复杂了，不能够用上章中介绍的状态图搜索法。因为这一类问题的状态图实在太庞大（甚至无限大），就是用启发式引导搜索，也不得不研究状态图的相当大一部分，故也无济于事。何况用状态图搜索求解必需占用极大量的计算机存储空间和计算机机时。要解决这一类问题，必需另外寻求更加有效的方法。

这一类复杂问题往往可以划分为若干个子问题来进行处理。因为这样做可以使得：

1. 子问题比原问题容易求解。

2. 如果所有子问题得到了解决，那么原问题也就得到解决。

实际上，在日常生活中我们就常常不假思索地把问题划小。譬如要油漆房间，我们会自动地把问题划分为：购买油漆、取梯子、覆盖家俱、刷天花板、刷墙壁等。

应当注意，有些子问题的解决顺序是很重要的。例如，在刷天花板和墙壁之前必须买好油漆，在刷天花板之前必须先覆盖好家俱。可是，有些子问题则不是这样。如先买油漆还是先取梯子，它们的顺序是无关紧要的。

有时子问题还可以再划分为更简单的子问题。例如购买油漆，可以划分为：驱车到油漆商店，选择油漆，付款，驱车回家等。

一、子目标和计划

把一个问题划分为子问题的方法之一，是在初始状态和目标状态之间引入子目标或中间目标。这样就可以把从初始状态到目的状态的原问题变成下面一些子问题：从初始状态到第一个子目标，从第一个子目标到第二个子目标，依此类推；最后一个子问题是从最后一个子目标到达目标状态。

子目标的序列就构成了解决原问题的一个计划。计划则通过解决每一个子问题而得到实现。

我们现在用图 4-1 来加以说明。假设原问题是要从初始状态 A 达到目标状态 B。

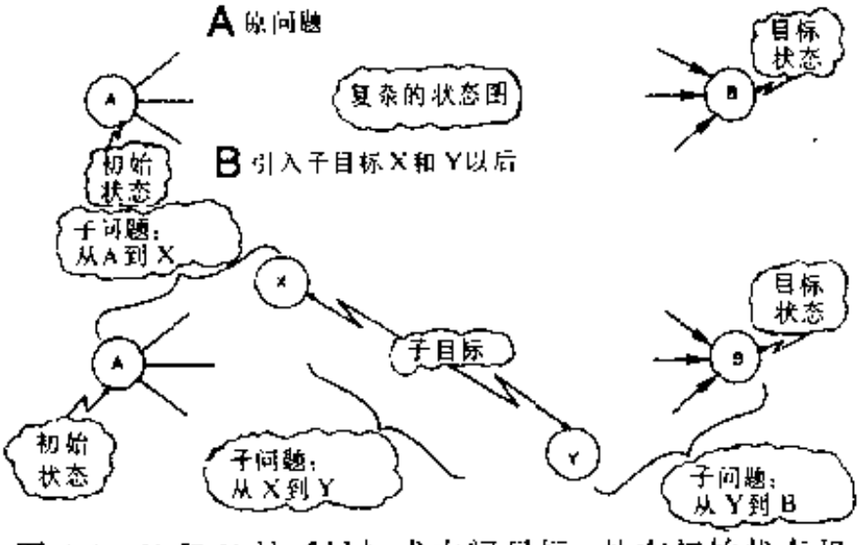


图 4-1　X 和 Y 是子目标或中间目标，处在初始状态和目标状态之间。我们把原问题变成了从 A 到 X，从 X 到 Y，从 Y 到 B 等子问题

我们筹划解决这问题时，引入子目标 X 和 Y。计划是从 A 到 X，从 X 到 Y，再从 Y 到 B，计划中的每一步都是一个子问题。当所有的子问题获得解决时，就是整个计划的实现。

为什么从 A 到 X，从 X 到 Y，从 Y 到 B 等子问题要比直接从 A 到 B 的原问题简单些呢？理由可能是多方面的，但从

状态图来看并不都能明显地看出。这些理由是：

1. A 到 X 比 A 到 B 近。从 A 出发到 X 和从 A 到 B 相比所需搜索状态图的部分较小。同理，从 X 到 Y 和从 Y 到 B 也是这样。

2. 可能找到更有效的启发式方法，引导我们更快地完成从 A 到 X、从 X 到 Y、从 Y 到 B 的搜索，而却没有直接从 A 到 B 的启发式方法。

3. 我们可能从已往的经验中知道，如何从 A 到达 X，从 X 到达 Y 和从 Y 到达 B。既然我们已经知道如何实现计划的每一步骤，所以一旦我们做出如上的解题计划，问题就迎刃而解了。退一步说，那怕是我们只知道其中的一个步骤，例如从 X 到 Y 如何实现，整个问题实质上也简化了一步。

另一种计划方法，是先确定一个关键操作符。所谓“关键操作符”就是解决问题过程中某处非用不可的操作符。比方说，问题是从城市 1 旅行到城市 2，两个城市之间有一道河，河上只有一座桥梁。显然，“过桥”就是关键操作符。

给出了关键操作符以后，我们就可以做出计划如下：

1. 将初始状态转变到使该关键操作符可能应用的状态。也就是关键操作符的条件部分取得真值的状态。

2. 应用关键操作符。

3. 将应用关键操作符所得到的结果状态转变到目标状态。

既然关键操作符已知，只需把它加以应用，则上面的子问题 2 就知道怎样解决了。子问题 1 和子问题 3 也许仍然很复杂不能直接求解，而必须继续分解为更简单的子问题。

现在仍以从城市 1 到城市 2 的问题为例，应用关键操作符可以产生如下的合理计划：

1. 从城市 1 走到桥边。

2. 过桥。

3. 从桥的另一边走到城市 2。

二、编 写 计 划

计算机程序怎样才能编写解题计划？这正是人工智能研究的一个课题，这个课题还远没有完全解决。现在我们看看某些在人工智能程序中曾用过的编制计划的方法。这些方法还都是试验性的，还需要进一步研究改进。

如果在一张纸上先画出整个问题的状态图，然后加以考察，我们将不难鉴别哪些是关键操作符，哪些是合理的子目标。可惜，(a) 计算机不能象人那样一瞥就掌握整个状态图，而且(b)大多数现实世界中的问题都很复杂，它们的状态图太大了，不可能画在一张纸上或存储在计算机里。所以状态图的整体结构对子编制计划不能有所帮助，从而使我们不得不集中注意力到状态的细部结构和操作符方面来。

(一)差异递减法

最早的人工智能技术之一，是集中注意于初始状态和目标状态的差异，找出能减少它们的差异的操作符，应用这些操作符，使初始状态逐步接近目标状态。

所谓"差异"，就是状态描述的某种特征，它对子初始状态和目的状态具有不同的值。通常，目标状态是用它们(指目标状态)所必需满足的条件来规定的；这时，差异就是初始状态的某种特征，它的值恰恰阻止了初始状态去满足目标的条件。

为了说明差异的应用，试考虑图 4-2 所示的问题，房间里有桌子 A、桌子 B，一个盒子和机器人。盒子放在桌 A 上。给机器人的问题是把盒子从桌 A 拿到桌 B。

状态的特征是机器人的所在位置、盒子的所在位置，以及机器人手中是否持有盒子。初始状态即图 4-2 中所示。目标状态则是盒子放在桌子 B 上的任何一种状态。

对这一问题的操作符，就是我们给机器人的命令。假设机器人能听从下述命令：

- GO TO（走到）。令机器人走到指定地点，即 A 桌旁或 B 桌旁。
- PICK UP BOX（拿起盒子）。令机器人从站着的桌子旁边拿起盒子来。这个操作符只有在机器人手里没有拿着东西并且正站在放着盒子的桌旁时才用得上。
- SET DOWN BOX（放下盒子）。令机器人把盒子放到身旁的桌子上去。这个操作符只有在机器人已经把盒子拿到手上并且站在空桌旁边时才用得上。

我们一开始就要问为什么初始状态不是目标状态。理由是在初始状态下盒子不在 B 桌上。我们要找到一个操作符能够把盒子放到 B 桌上去。

SET DOWN BOX（放下盒子）是这样一个操作符。为了应用它，机器人必须把盒子拿在手上并且站在一张桌子旁边。机器人不管站在那张桌子旁边都可用这个操作符，但要得到预期的结果，机器人必须站在 B 桌的旁边。

于是，我们可以把原始问题分成两个子问题（图 4-3 a）。第一个子问题是从初始状态变到让机器人站在 B 桌旁手里拿着盒子。第二个子问题是把盒子放到 B 桌上去。第二个子问题很容易应用操作符 SET DOWN BOX（放下盒子）来解决。

现在来看第一个子问题，它的目标状态是机器人站在 B

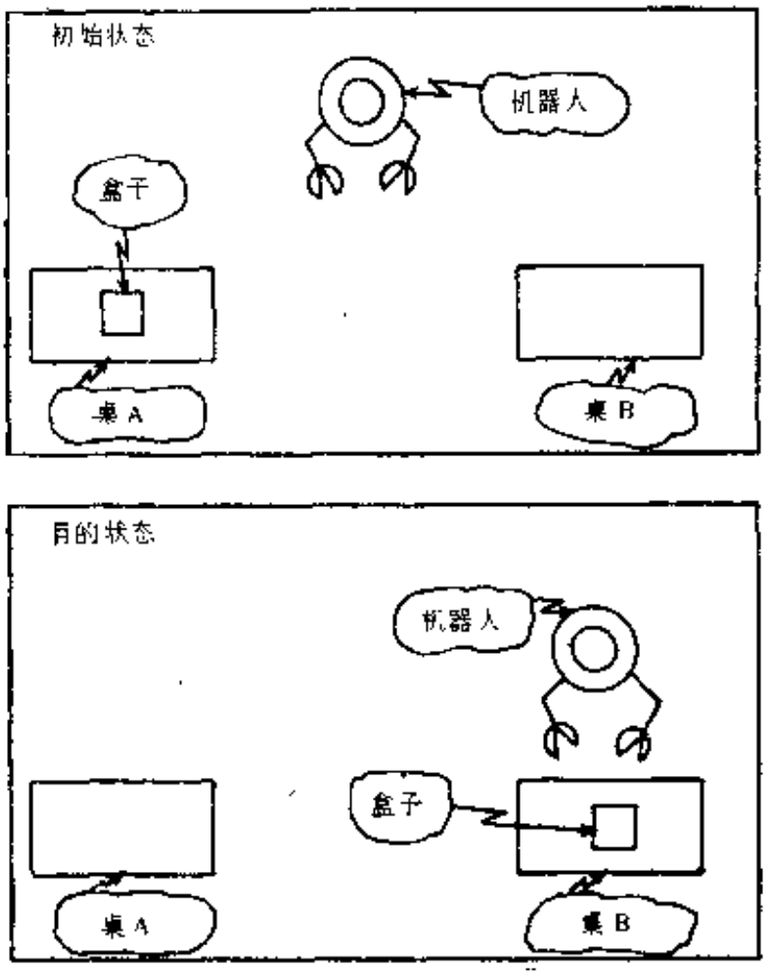


图 4-2　机器人控制问题中的初始状态和目标状态。
机器人要把盒子从 A 桌移到 B 桌

桌旁手拿着盒子。

对这个子问题来说，在初始状态和目标状态之间有两个差异。在初始状态下，(1) 机器人不在 B 桌旁，(2) 手里没有盒子。

我们着手来解决第一个差异。操作符“GO TO”(走到)B 桌能使机器人走到 B 桌旁，于是我们的计划如图 4-3b 所示。我们首先把盒子放到机器人的手中，然后应用“GO TO”(走到)B 桌和“SET DOWN BOX”(放下盒子)这两个操作符，但要注意：做计划的人要知道应用操作符“GO TO”(走到)B 桌并不会改变机器人手里拿着盒子这件事实，这点很重要。

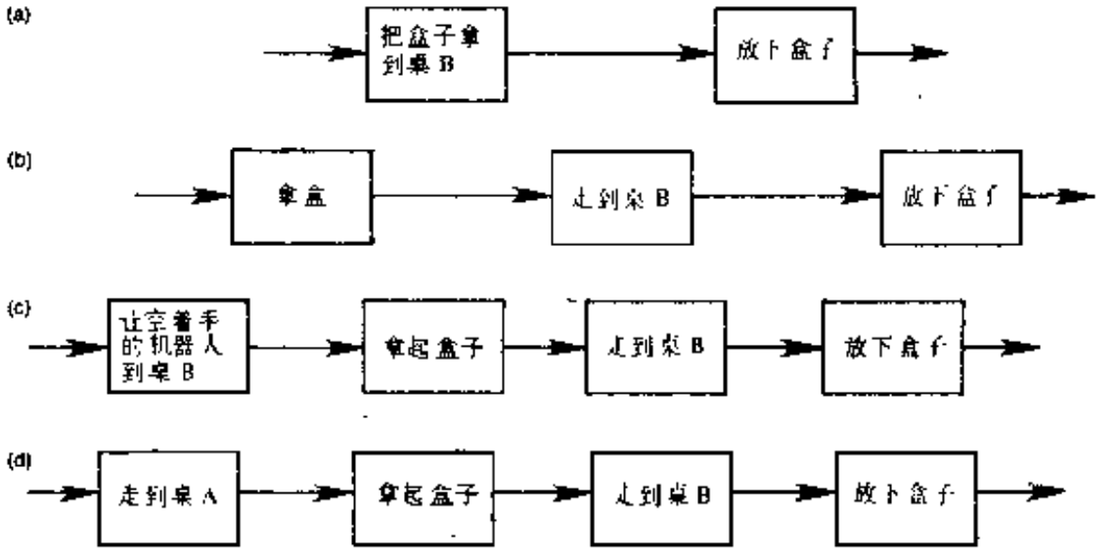


图 4-3　解决机器人控制问题的计划的逐步细化

现在待解决的子问题是：从初始状态改变到机器人手里拿着盒子的状态。操作符“PICK UP BOX”(拿起盒子)可以做到这点。但是这个操作符只能应用在当机器人空手站在 A 桌旁的条件下，我们的计划现在成了图 4-3c 的那个样子。

剩下的问题是让机器人空手站在 A 桌旁。在初始状态下，机器人是空着手的，但不在 A 桌旁。操作符“GO TO”(走到)A 桌可以消灭这个差异。

于是最后的计划将如图 4-3d 所示。我们给机器人下的命令共有四条，即：GO TO table A(走到 A 桌)；PICK UP BOX(拿起盒子)；GO TO table B(走到 B 桌)；SET DOWN BOX(放下盒子)。

请注意，在解决问题时我们是从目标状态开始，回溯到初始状态，由于明显的理由，这项技术方法叫做倒推法。

对于象这个例子那样简单的问题，差异递减法很起作用。但对于复杂问题，由于下述困难，这个方法往往失败：

1. 初始状态和目标状态距离太大，以致它们的差异不能提供任何线索，譬如问题是造房子，仅仅对一堆砖，一堆木料

和一幢建好的房子进行比较，很难推演出建筑和木工工艺来。

2. 状态的各个特征可能互相影响。就是说，找不到这样的操作符，它只改变某一特征而保持其他特征不变。因而当我们应用一个操作符去消灭某一个差异时，可能同时改变了若干其他特征，以致产生新的差异多于被消灭的差异。

3. 差异递减法程序可能浪费许多时间，因为有些操作符所减少的差异是微不足道的，根本不能解决问题。不久前在《SIGART 通讯》上刊着一幅漫画：一个机器人坐在树梢上望着月亮说："我的目标是到月亮上去。现在我爬到了这棵树上，减少了我的现状和目标之间的差异"。这是对差异递减法的一个很有风趣的评论。

4. 对于登月机器人说来，初始状态和目标状态未免太类似了。地球上的机器人和月亮上的机器人只有唯一的一个不同特征(即机器人在空间的位置)。可是绝不会有一种单一操作能够把机器人送上月球来消灭这个差别。这个差异对于事实上需要的整个复杂操作过程提供不了任何线索。

简言之，差异递减法只适用于这样一种场合，即初始状态和目标状态之间只有少数特征不同，而且有这样一些操作符能对每个特征个别地进行处理。但它并不能产生出什么主意来制订复杂的计划解决困难的问题。

(二)启发式方法：利用现有的知识

倘若我们要把登月机器人问题用差异递减法来解决，这无异于要求程序从描绘整个宇宙飞行的理论和实践开始，这是不可能的。可是，一个空间科学家，却不难凭借有关如何将不同大小和重量的物体从地球送到别的天体上去的广泛知识，迅速地做出一个月球飞行的粗略计划。

对于一些对人来说是低难度的问题，控制机器人程序所

需要的不过是一些日常活动的普通常识。譬如，一个人从来不用考虑怎样才能把一个盒子从一张桌子移到另一张桌上去。如果你要他解释为什么要这样作（比如先走到有盒子的桌旁去，拿起盒子，……等等）他一定会争辩说这不过是普通常识，根本无需解释。可是用差异递减法编制程序，却必须经过一系列繁琐的推理才能引导机器人完成同一任务。

怎样把现有的知识结合到问题求解的程序里面去，这仍然是一个有待研究的课题。我们在这里介绍一种有人提出过的似乎颇有希望的想法，虽然它仍需要进行试验和精心研究。

这个想法就是把知识和启发式结合起来。而启发式，我们还记得，是一条经验法则。每一条启发式通常都包含一个条件和一个或多个计划。

条件部分的作用就象操作符中的条件部分一样，它决定什么时候可以应用启发。除了应用于初始状态外，它还可以应用于问题的一切因素，包括：初始状态，目的状态和所有可利用的操作符。

计划部分包含一个计划表。当条件成立时，某个计划便被推荐出来。计划可以用各种形式表现：如一系列的子目标，关键操作符，或（由原计划分解而成的）若干子问题的表。

常常会碰到这种情况：一种启发式行不通，于是它所推荐的计划也不能进行——这就是说，其中有一个或几个子问题无法解决。在这种情况下，就得试验计划表中的其他计划。如果整个计划表试完都不行，就得试验其他（对于所求解的问题其条件能够成立的）启发式。如果所有启发式试完都不行，那就说明这程序不能解决这个问题。

启发式的条件部分也可以看作是：当问题中存在某种模式时，该条件就成立。这时启发式告诉我们到问题中去查找

某些模式，一旦找到了它，就可以推荐某些可供试验的计划。模式识别本身是一个相当重要的问题，我们留到下章再讲。

编写计划和实现计划，其重要性是不言而喻的。譬如在计算机下棋这个领域中，人工智能程序之所以遭到棋师的批评，常常是因为只注意到下棋的细节——特别是防备突袭的动作——而且举棋没有目的。这表明缺乏一个取胜的整体计划，是这种程序的主要弱点。

三、搜索与/或树

将问题分解为若干子问题，再将这些子问题进一步分解为其他子问题，如此这般进行下去时，用的就是所谓问题约简的求解方法。本节将对问题约简搜索法作一些详细考察。

象在第三章描述过的状态图搜索法一样，我们也可以用搜索树来帮助理解问题约简搜索法。

问题约简搜索树的节点就是问题。树的根就是原问题，所有其他节点都是子问题。一个节点的子节点就是亲节点约简而成的子问题。节点的扩展把对应于亲代的问题约简成为对应子子代的子问题。

搜索树的叶有三种类型：

1. 不需约简为子问题便可解决的问题叫做原始问题，对应的叶节点称为已解的。

2. 不能解的问题或不能再约简为子问题的问题，称之为无解的，对应的叶节点相当子状态图搜索法中的死端点。搜索程序不能沿着引向无解的节点的路径进行下去，而必须溯回并试验其他路径。

3. 在搜索过程中未经检验过是否可能扩展的叶节点。

将问题约简为子问题的办法只有两个，所以，扩展一个节

点的方法也只有两个，这里用图 4-4 来加以说明。

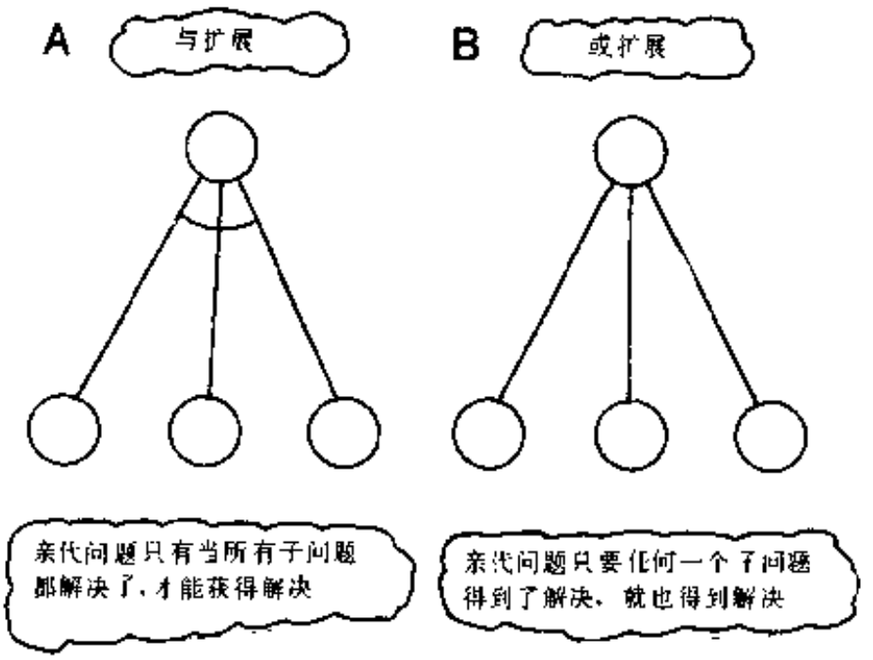


图 4-4　与扩展和或扩展。节点下面的弧指示与扩展

(一)与扩展

一个问题只有当它所有的子问题都解决后才能得到解决；这一类问题的约简一直是我们讨论的主要问题。对应于这一类问题的节点的扩展叫“与”扩展。图 4-4a 中，与扩展用子节点分枝上跨接的半园弧来表示。图 4-5a 中表示完全由与扩展构成的“与树”。在与树中对应于根上的问题只有当所有叶节点都解决了以后，它才能得到解决。

(二)或扩展

一个问题只要它的任一个子问题解决了，它也就解决了。这些子问题等于是原问题的替身，因为解决任何一个子问题也就解决了原问题。与这一类问题相应的叶节点的扩展叫做“或”扩展。图 4-4b 表示的是一个或扩展，而图 4-5b 表示的是一个完全由或扩展构成的“或树”。在或树中对应于根上的问题，只要叶节点上的任何一个得到了解决，它也就得到了解

决。

或树更接近于状态图搜索法中的树（请记住在状态图搜索法中从亲代到子代的分枝代表的是可以互相替换的待搜索的路径）。或树中已获解决的叶节点对应于状态图树中的目标节点，不能解的叶节点则对应于一个死端点。

通常我们遇到的既不是纯粹的与树也不是纯粹的或树，而是如图 4-5c 所示的与/或树。一个与扩展引入一组子问题，这些子问题解决了，才能使亲问题得到解决。所有这些子问题就构成了解决亲问题的一个计划。一个或扩展引入的是一组可供抉择的解决亲问题的计划，这些计划中任何一个得到了实现，都可以使亲问题得到解决。

图 4-6 所示的是图 4-2 中机器人控制问题的一个搜索树。这是一个单纯的与树，因为问题比较简单，可以直接求解而毋需考虑选择其他计划。

与/或树在搜索时可以用宽度优选法、深度优选法或有序搜索法。其细节和状态图的类似，不再赘述。更详细的了解可以参看文献目录中 Nilson 的著作。

同一个子问题可以出现在若干个不同问题扩展中，这意味着在一个搜索树中的若干个不同的节点可以对应于同一个子问题。这和状态图搜索树中，若干个不同的搜索树节点对应于同一个状态图节点相类似。这两种情况的处理方法也是相似的。

问题约简搜索法中有一个特点和状态图搜索法中对应的特点完全不同，这就是：确定什么时候问题解决而且搜索可以终止。

在状态图搜索中，一旦目标节点被找到，搜索就可以终止。而在问题约简搜索中，如果我们搜索的是一个单纯的或

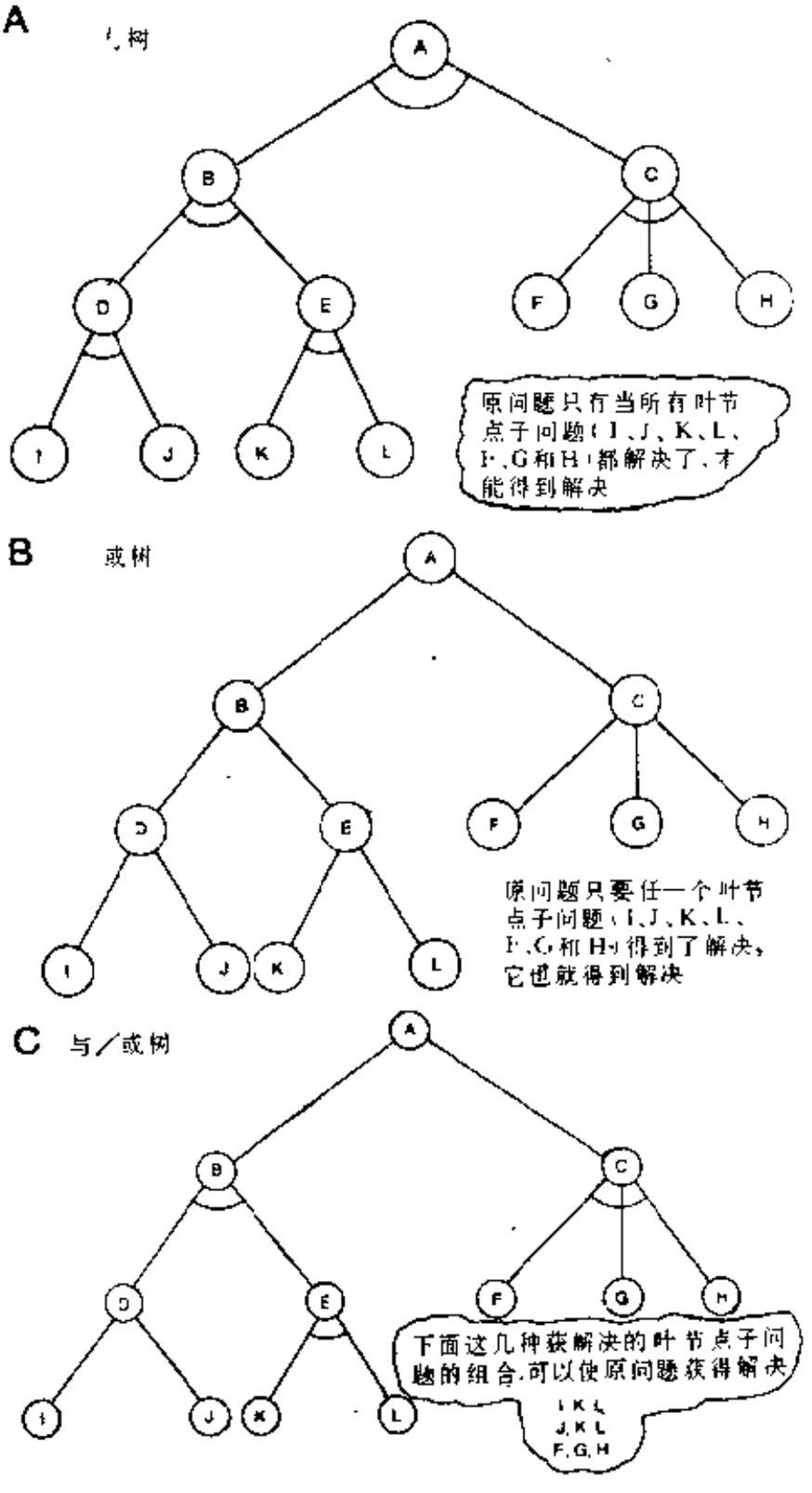


图 4-5　与树、或树和与/或树

树，只要找到了一个可解的节点，马上就可以停止搜索。不过，在与/或树中这个过程就更复杂一些。

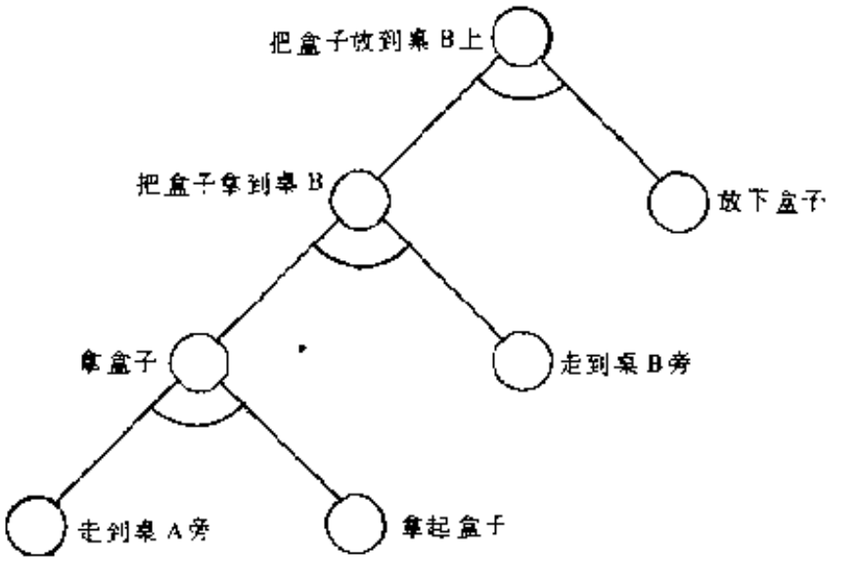


图 4-6　在图 4-2 中提出的机器人控制问题的搜索树

图 4-7 说明怎样确定一个与/或树的搜索何时可以终止。对应于已知解答的那些问题的叶节点用 s 标出，而对应于不能解答的那些问题的叶节点用 u 标出。其余没有标出记号的叶节点，则是还没有扩展的叶节点。

现在我们自下而上，一层一层地上升到根节点，同时把这些节点标上 s 或 u，或者不加标记。标记节点的规则如下：

1. 有与扩展的节点，若该节点的所有子节点标上 s，则该节点也标上 s。若有任一个子节点标上 u，则该节点也标上 u。否则，该节点不带标记。

2. 有或扩展的节点。若该节点的任何一个子节点标上 s，则该节点也标上 s。若所有子节点都标上 u，则该节点也标上 u。否则，该节点不带标记。

我们从叶节点开始上升到根，总是先标记一个节点的所有子节点，然后才标记该节点自身。若根节点标记为 s，则原问题得到解决。若根节点标记为 u，则原问题不得解决而问题求

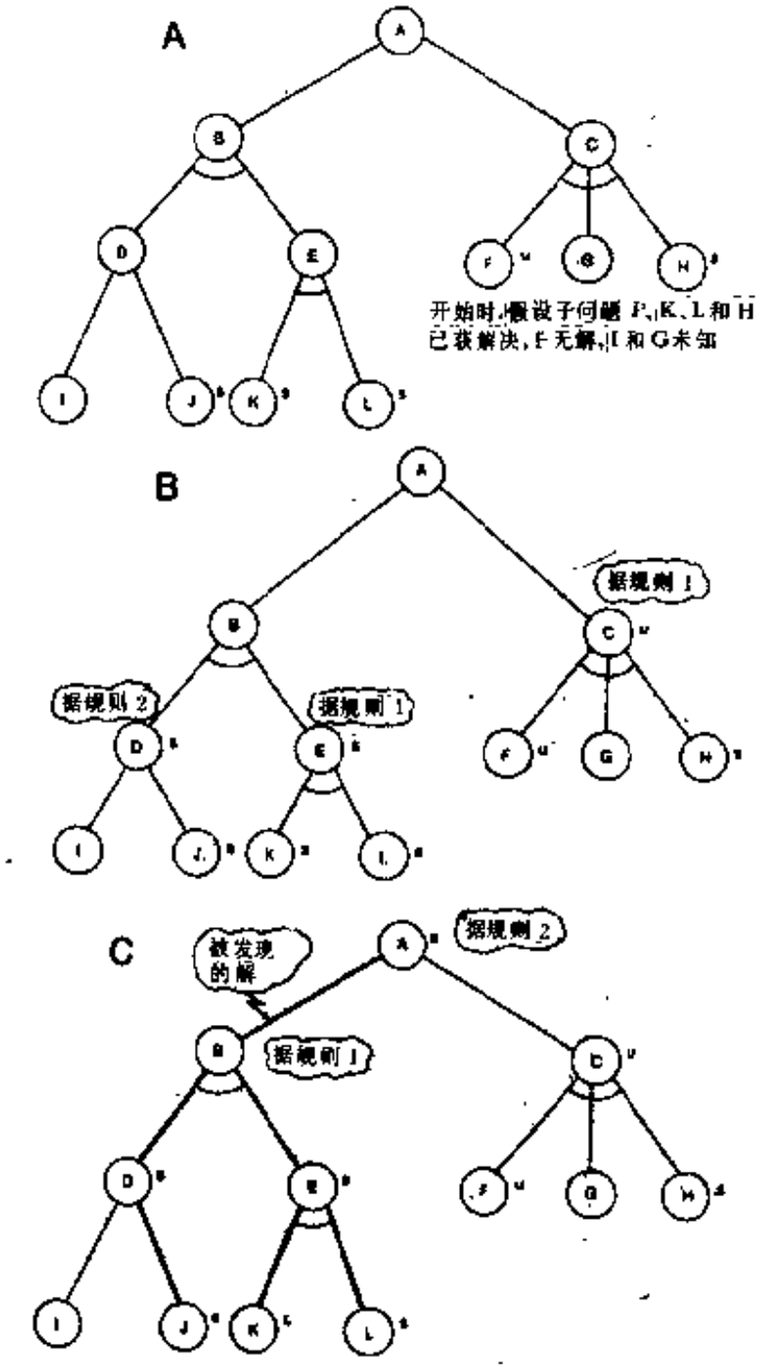


图 4-7　确定与/或树中根节点对应的问题有解或无解，标有 s 的节点有解，标有 u 的节点无解，我们假设叶节点已标记，从叶节点上推到根。在 C 图中粗线表示被发现的解

解程序归于失败。若根节点无标记，则搜索应当继续进行下

去。

在搜索过程中，每当检验一个叶节点并发现它已解决或无解时，都要实行上述过程，看看根节点有解或无解。只有从根到叶的通路上的节点才需要计算其标记。

以后在谈到博弈游戏程序时，我们还要回到与/或树的问题来加以讨论。

第五章　分级计划和过程网

在第一章中提到过，探索一个不断分枝的途径时可能发生所谓组合爆炸的问题。状态图搜索法和问题约简搜索法都可以用搜索树来加以描述，所以二者也都有组合爆炸的问题。

让我们试看一下爆炸规模的大小。假设每个节点都有七个子节点(早期下象棋程序中的分枝因数是七)。这意味着在状态图搜索法中每一个节点都有七根弧线，而在问题约简搜索法中，每一个问题或子问题都可以分解为七个子问题。

树的根部虽只有一条探索路径，但每下移一级，探索路径的数目都要用七乘一次。在表 5-1 中我们给出了从 0 级到 10 级探索路径的数目。

现在的计算机技术允许我们在合理的时间内探索多达近三十亿条路径。在可预见的将来大概还不可能超过更多的级数。

表 5-1　组合爆炸

搜索树级别	搜索路径数
0	0
1	7
2	49
3	343
4	2401
5	16807
6	117649
7	823543
8	5764801
9	40353607
10	282475249

当然，我们需要用启发式方法来限制每一个节点所产生

的子节点数目。可是，局部启发式方法（就是只考虑到一个单独节点的情况）很难起到充分的限制作用。如第一章中所举的例子，从城市到城市（局部启发式）对于指引我们到达目的地是没有什么用处的，除非我们已事先知道我们必需经过哪些城市才能到达目的地。

一、分级计划

人们在解决问题的时候避免组合爆炸的方法通常是，在深入细节之前，先做出解决问题的计划。在解决问题的每一步上，他只搜索和计划相一致的少数几条路径（也许只有一条路径）。

解决问题的详细计划并不是一次就想起来的。通常总是先有一个一般的设想，然后经过多次细化，每次都增添一些细节，直到最后才做出一个详细的计划。有些读者也许熟悉结构程序设计吧，在设计中一个程序概念是经过反复多次细化，才系统地发展成为一个程序的。

这样一种计划方法便叫做分级规划或等级规划。整个分级计划最上一级是最粗略的计划，最下一级是最详细的计划。如图 5-1 所示，计划的每一步都形成一个子树。树的每一级都相当于原计划的一次特殊细化。

一个计划细化以后，不应当马上就把产生这个计划的粗略计划抛弃。实际上，我们应当保留如图 5-1 的整个树。假设详细计划的某一步执行不通的时候，我们不应放弃整个计划。相反地，我们应当返回到发生问题的这一步的亲节点上去，设法再加以细化。这样，我们便可以围绕一个难点去寻求解决，而不必放弃整个计划，一切从头开始。

有些程序要求不仅要想到计划，而且要留意计划在现实

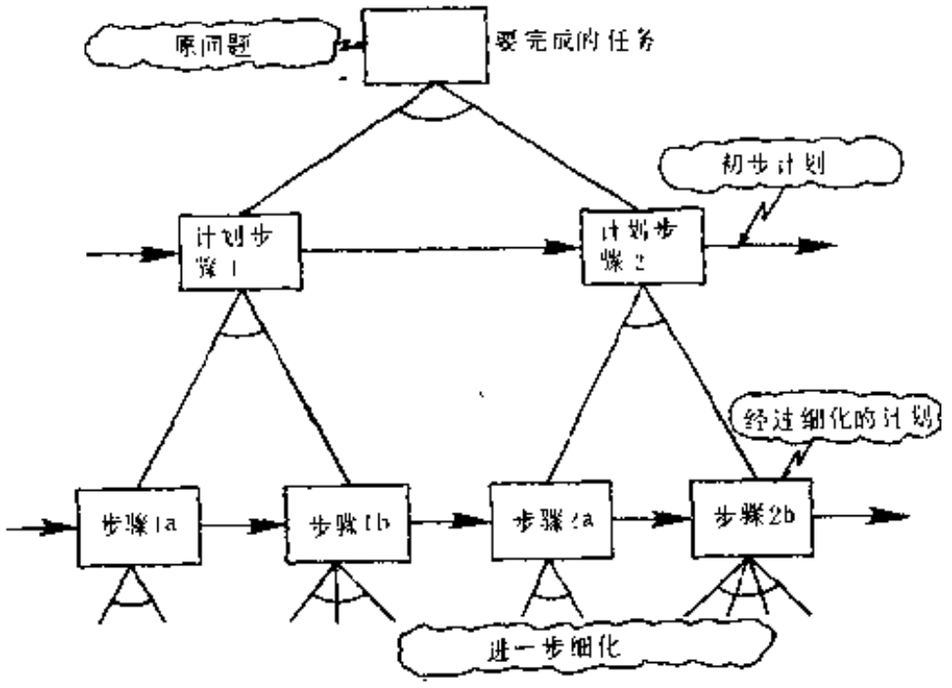


图 5-1　分级计划图。树的每一级是上一级计划的细化

世界中得到实现。这一类程序的典型例子有：

1. 机器人控制程序。

2. 为执行一定任务而逐步给人以指示的程序。

3. 博弈游戏程序。

我们称这类程序为必须监督计划执行的程序。

即使计划已经完全做出，在执行计划的过程中仍有可能发生意外的困难：机器人在路上可能发现意外的障碍；人可能发现无法执行某一特定的指示；对手可能采取出其不意的棋步，等等。这时，我们就得再次把出毛病这一步的亲代或祖代计划加以进一步细化，而不需要把整个计划放弃。

从另一方面来看，一个非常详细的计划中每一步的祖代计划中都包含一定的知识，说明选择这一步的理由。如果这一步出了问题，我们仍可以利用这些知识做其他的合理选择，而尽量避免修改计划的其余部分。

这一章中我们将介绍萨切尔多蒂(Earl D. Sacerdoti)提出

的分级规划法。有兴趣的读者可以参看他的书:《计划和行为的结构》(见参考文献目录)。这本书里包含有许多关于制订和执行计划的很有价值的见解。

二、过 程 网

萨切尔多蒂把每一级计划的细化都用过程网来表示。(这里,"网"是"图"的同义语。)

为了了解过程网是怎么一回事,我们先研究一个具体问题,仍以机器人控制为例。记得我们前面讲过,对于人来说是微不足道的问题,对于机器人来说,绝不是什么微不足道的问题。这点是必需强调的。

图 5-2a 表示一个问题的全部情景。机器人在一个房间里,房间里有三个盒子:小盒 A、中盒 B 和大盒 C。三个盒子分两处放着。现在令机器人把三个盒子堆放在一处如图 5-2b 所示。当然大盒总应放在底下,中盒放在中间,小盒放在顶上,否则它们容易颠覆下来。

我们假定机器人能够拿起盒子把它放到地板上或者别的盒子上去。我们又假定这些盒子都很重,所以机器人每次只能拿一个盒子而不会一次拿好几个。

机器人控制程序首先必须检验一下这个问题,然后做出一个解决问题的粗略计划。这里我们暂不详细讨论问题的模式如何被识别以及它如何和计划发生联系的问题,而留待以后各章再介绍一些做这些事情的办法,至于哪个办法最好,目前还弄得不很清楚。

假定求解问题的人已经知道,A 必须放在 B 上,B 必须放在 C 上,这样才稳当。于是,初步计划就是要把 A 往 B 上搁,把 B 往 C 上搁,如图 5-3 所示。现在先不考虑这两桩事的先

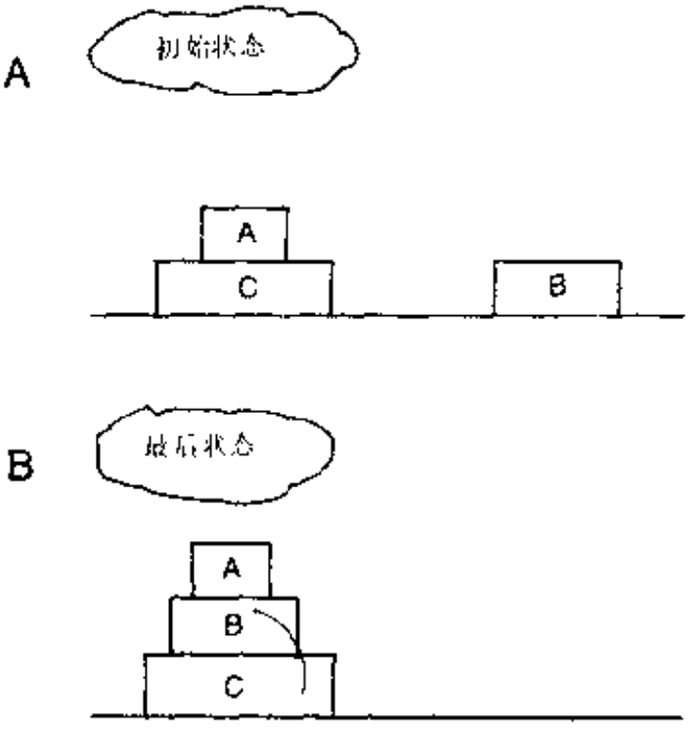


图 5-2　机器人控制问题。机器人要把三个盒子堆在一起，且 A 盒在上，B 盒在中间，C 盒在下，使它们不易倾倒

后次序。

图 5-3 是一个最简单的过程网，它只包含一个节点。这个节点规定了一个一步计划里的单个步骤。

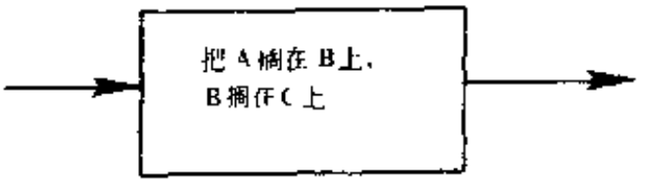


图 5-3　初步计划把 A 搁在 B 上、B 搁在 C 上，两个操作的顺序未加规定

如果有两个任务要完成，显而易见的计划就是去分别执行这两个任务。我们仍然不会考虑按某一特定的次序去执行这两个任务。

于是，我们可以把图 5-3 扩展成图 5-4 那样的过程网。

网的通路一分为二：一条路是“把 A 搁到 B 上”。一条路是“把 B 搁到 C 上”。我们用与节点弧线来指出：要完成这个计划必须把这两步都做好。可是，在计划细化的这一级水平上，并没有对这两步执行的先后顺序做出规定。这计划叫做非线性计划，因为它的各个步骤不是排列在一条直线上，来指明它们执行的先后顺序。

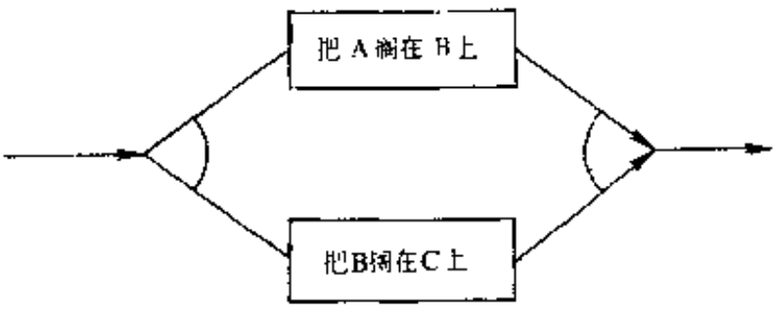


图 5 4　过程网有并联的两个分枝。两个分枝的操作都必须执行，但对执行的顺序未作规定

现在我们需要做出一个把一个盒子搁到另一个盒上的计划。假设有两个盒子 X 和 Y，要把 X 搁到 Y 上面去。那么我们需要做三件事：

1. 清除盒 X 上面放着的东西。因为机器人每次只能拿起一个盒子。

2. 清除盒 Y 上面放着的东西，以便给盒 X 腾出地方来。

3. 把盒 X 放到盒 Y 上面去。我们假定 SET XON Y（置 X 于 Y 上）这条命令能使机器人拿起盒 X 并且放到 Y 上面去。

图 5-5 表示这个计划的过程网。注意它没有管清除这两个盒子上部的先后次序，但规定了在发出 SET X ON Y（置 X 于 Y 上）这条命令之前，两个盒子必须清除好。

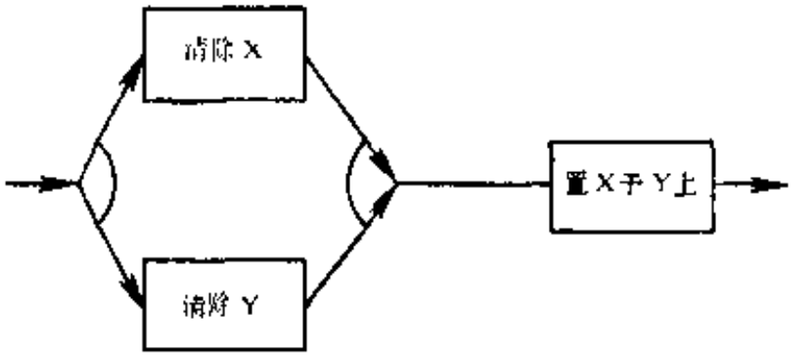


图 5-5　把盒 X 搁在 Y 上的计划。先清除 X 上部还是先清除 Y 上部无关紧要

现在我们用图 5-5 堆放盒子的计划来把图 5-4 的计划加以细化。共用两次：一次用 A 和 B 来代替 X 和 Y，一次用 B 和 C 来代替 X 和 Y。图 5-6(a)是它的结果。

注意图 5-6 (a) 中有三个节点没有画成矩形而是画成椭圆形。这叫做幻节点。所以称为幻节点的原因是，它们所要达到的子目标在初始状态中已经为真。具体地说，在初始状态中 A 和 B 上部都已经清除；所以，“清除 A”和“清除 B”都是幻节点。

幻节点在较低的级次中决不会再扩展得更细。它们只是被保留在计划里面，以表示初始状态中已做过哪些假设。除了它们不再扩展以外，在计划过程中对待幻节点和对待其他节点都一个样。

分级规划中的一个关键是：在每一级对计划进行细化时，都必须检验计划，发现问题加以更正或改进。萨切尔多蒂把这一步骤叫做评判。在我们的计划里，头两步不需要评判，现在这一步需要评判。

图 5-6 (a) 的注释中暴露一个问题：计划中出现了矛盾。计划的上半部“置 A 于 B 上”这个节点把 A 搁在 B 上面，可是

计划的下半部“清除 B”这个幻节点却告诉我们 B 的上面已经清除。如果我们先实现计划的上半部然后再实现计划的下半部，当我们进行到计划的下半部时，B 的上面因为已经有 A 并未清除，那么“置 B 于 C 上”也就不可能实现（假若“清除 B”是一个真节点而非幻节点，那么计划的下半部就会抵消掉“置 A 于 B 上”所做的工作。所以不管“清除 B”这个节点是真是幻，都存在着问题）。

为解决这个问题，我们必须保证图 5-6（a）计划中的下半部在“置 A 于 B 上”之前执行。图 5-6（b）是修改后的计划。注意我们只稍稍改变一下次序，以保证“消除 B”永远在“置 A 于 B 上”的前面。

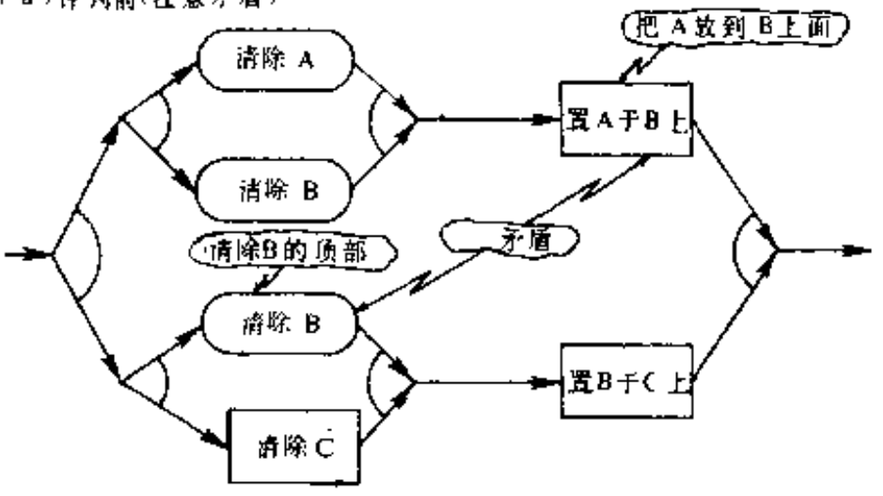
图 5-6（b）的计划还可以进一步改进。当计划的下半部执行完时，B 的上面已经清除好。事实上在执行“置 B 于 C 上”之前 B 就已经是清除好了，而“置 B 于 C 上”这个操作并不会在 B 上面加上任何东西。所以，计划上半部的“清除 B”这个幻节点也是多余的、可以取消。图 5-6（c）是这一级计划的最后修改结果。

在图 5-6（c）中，“清除 A”和“清除 B”都是幻节点，故不会再进一步扩展。“置 A 于 B 上”和“置 B 于 C 上”已假定为原始操作，可以直接下达给机器人而不需扩展。所以只剩“清除 C”是唯一需要扩展的节点。

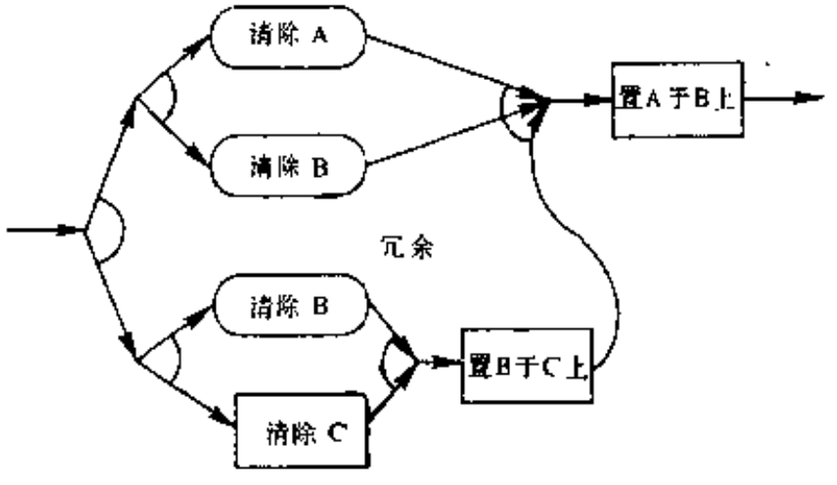
为此，我们需要做出一个一般计划：清除某个盒子的上部（譬如盒 X 上部）。问题是我们不知道 X 上面堆放着几个盒子。但假设 X 上面直接放着的是 Y。于是我们可以进行如下：

- “清除 Y”。进行必需的操作使 Y 的上面不放任何东西。
- “置 Y 于地板上”。

(a)评判前(注意矛盾)



(b) 消除矛盾后



(c) 消除冗余结点

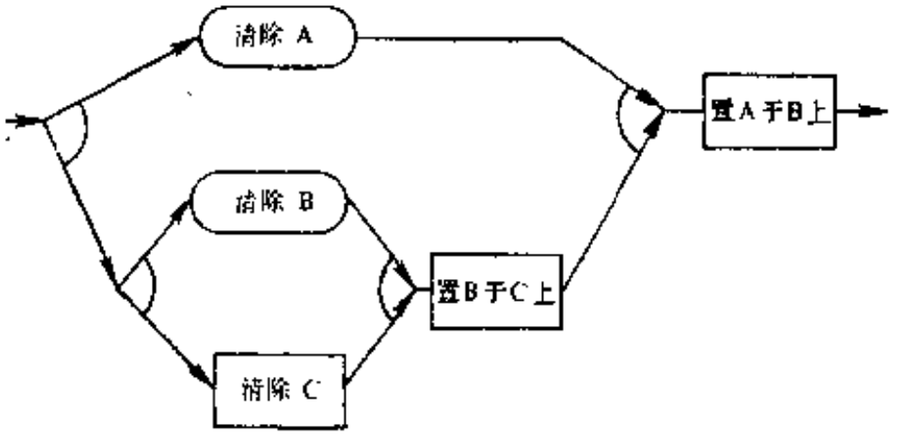


图 5-6

(a)应用图 5-5 的计划将图 5-4 扩展的初步结果

(b)评判后解决了 a 中的矛盾 (c)评判后清除了多余的节点

图 5-7 表示这个计划的过程网。请注意“清除”的定义本身包含一个清除除的操作，所以“清除”部分地是以本身来定义的。任何一个函数，如部分地以函数本身来定义叫做递归的函数。故“清除”是一个递归的计划。如图 5-8 所示，“清除”步骤的反复扩展，能将指定的盒子上面堆放着的任何数目的盒子都清除掉。

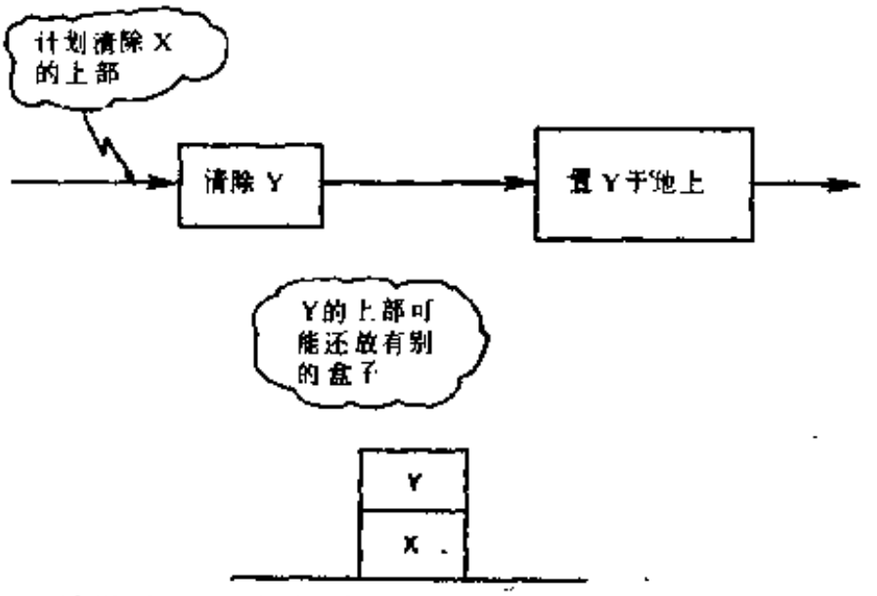


图 5-7 清除盒 X 上部的计划。假设盒 Y 直接放在盒 X 上面，盒 Y 上面还可能放着别的任何数目的盒子

图 5-9 (a) 表示出图 5-6 (c) 中“清除 C”节点扩展后的结果。该计划还可以通过评判得到改善。“置 B 于 C 上”执行以后，A 的上部仍然是清除了的，故上面分枝中的“清除 A”是多余的，把它取消以后便得到图 5-9 (b)，这是计划的最后修改结果。

幻节点“清除 A”和“清除 B”被保留在计划里面，因为当计划被拿来执行时，很可能发现 A 或 B 还根本没被清除好。这时幻节点就可以用来作出必要的补充计划，应付这种意外的情况。

如果我们假定这种意外不可能发生，且 A 和 B 在一开始

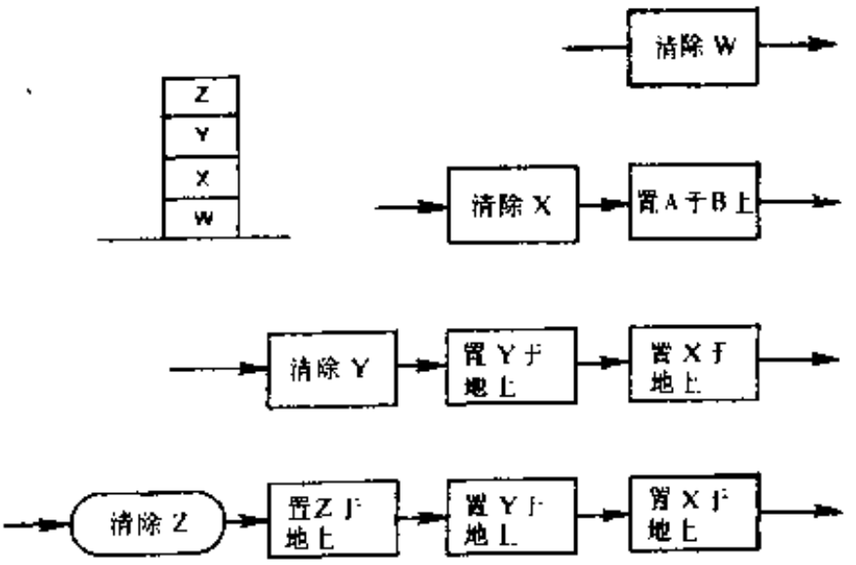


图 5-8　反复扩展“清除”节点，便可用图 5-7 的计划来清除指定的盒子上面的任何数目的盒子

确实已经清好，那么给机器人下达的命令将是：

“置 A 于地板上”；“置 B 于 C 上”；“置 A 于 B 上”。

这个例子说明了过程网的两个重要特点：

1. 当计划产生的时候，某些步骤执行的最好次序是什么还不十分清楚。过程网允许对这种次序可以不先做出规定，等到正确的次序确实弄清以后再说。而这可能要等到计划实际被执行的时候才可能知道。

2. 在每一级计划进行细化的时候，我们使用启发式，萨切尔多蒂称之为评判法；它用来减少错误、消除多余和改进计划。评判常常用来规定步骤执行的次序，而这种次序原来并没有规定。

过程网的有趣的特点还不只这些。它还有其他一些有趣的特点，我们将在“知识表达”和“自然语言处理”等章中再加以介绍。

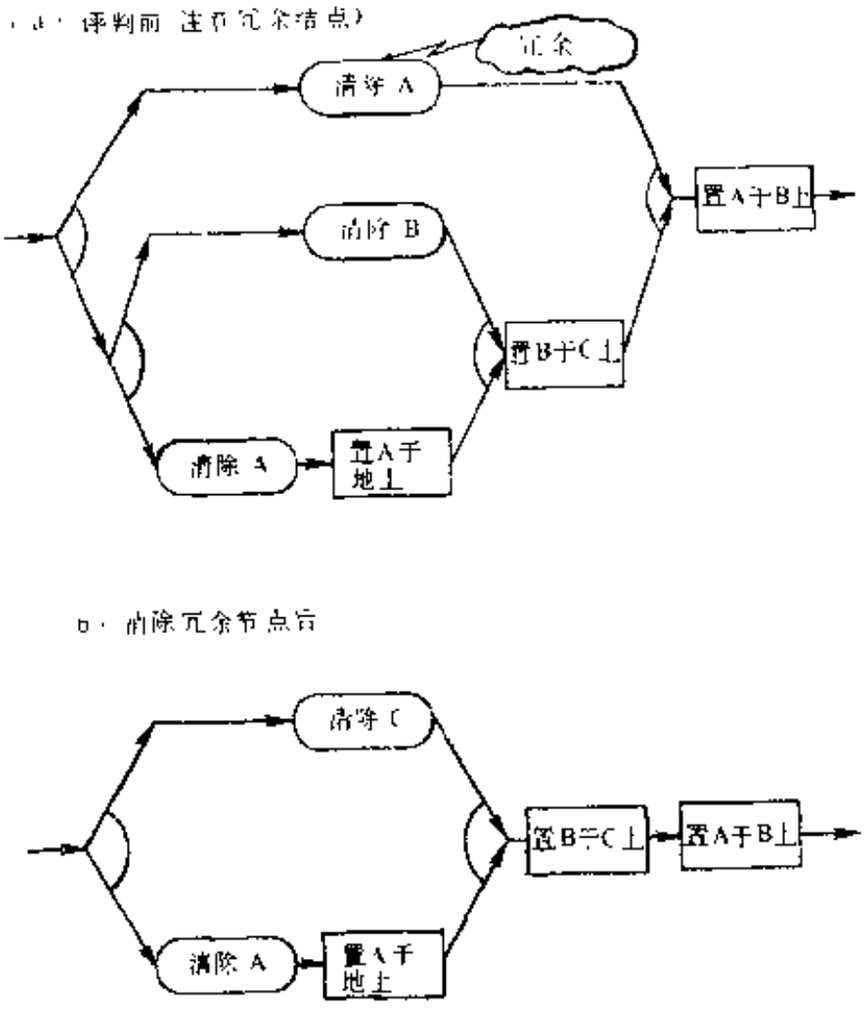


图 5-9

(a)在图 5-6c 中用图 5-7 的计划扩展"清除 C"节点的结果

(b)在(a)中经过评判消除了多余的"清除 A"节点

三、监督执行

有些程序，例如控制机器人的、教学的或玩游戏的程序，不但需要做出计划，而且需要监督计划的执行。过程网特别适合于监督计划执行，原因有二：

1. 计划和执行是可以互相交替进行的。假若计划已经制定，但在执行计划之际发现某些障碍，我们可以回到计划上来，想出绕过障碍的办法，然后再来执行修改后的过程网。如

果预计到一种频繁出现的障碍，则可以先做出一个粗略的初步计划，等到有关的障碍已知时，在执行每一步骤之前才将计划进行细化。

2. 因为计划可以用于好几个不同详细程度的级别。执行计划的各个不同部分的指示也就可以有不同的详细程度。对人下指示时这点是尤其重要的；因为人可能知道如何执行计划的某些部分，但对其他部分则需要更详细的指示。

我们下面看一看怎样把过程网应用于在本节开头提出的三个问题领域中的某些细节。

(一)机器人控制

给机器人的所有指令必须具有均匀一致的详细程度。就是说，所有指令都必须用机器人能够执行的命令给出。

极言之，给机器人的命令必须极其详细：应当告诉机器人将一只手向某一方向伸出去多少厘米，或者告诉它把两只手合拢来直到压力传感器告诉它已经抓住一只盒子为止。

预先详细计划好这些动作看来是没有用的，因为机器人要做些什么将依赖于传感器每秒每秒所收集的信息。准确地说，就是机器人相对于要抓取的下一个盒子来说处在什么位置？机器人的手相对于要抓取的盒子来说又处在什么地方？假若发现要搬动的物体形状很奇特，用怎样的方式抓取它最好？

所以，看来给机器人所做的初步计划只应详细到我们举的例子那样的程度："拿起盒子"，"放下盒子"，"置盒 A 子盒 B 上"，等等。更详细的计划最好是在机器人准备执行之前，而且计划所必需的全部信息都已从感觉器获得的时候，方才做出。这个例子说明在什么地方计划和执行可以交替进行，这对我们来说是迟早很有用的。

倘若在机器人周围环境中发现意外的障碍，或者机器人本身出了机械故障，那么我们也许需要在执行时间内做出新的计划。

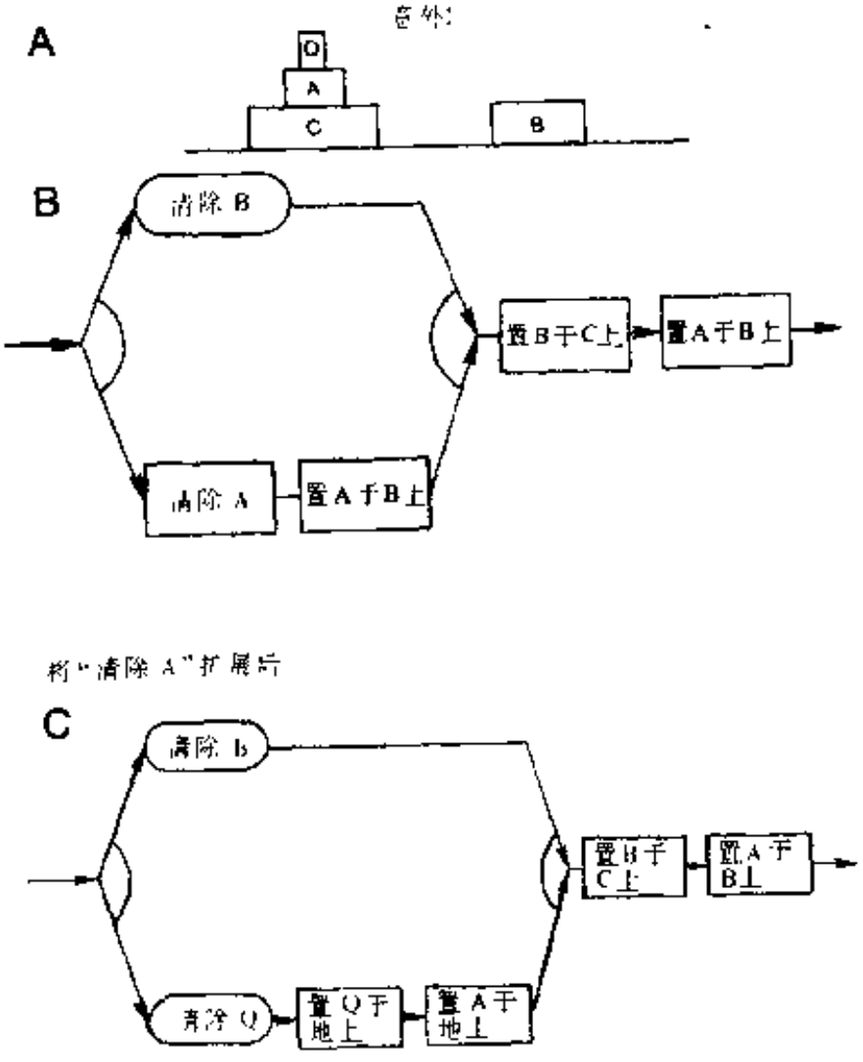


图 5-10　机器人着手执行图 5-9b 的计划时，意外地发现情况 a，即 Q 在盒 A 的上部。计划不得不改变，将幻节点"清除 A"变为真节点，并加以扩展，如 B、C 两部分所示

譬如，机器人着手执行上节提到的堆放盒子的计划时，意外地发现在盒 A 上面还有一个盒 Q，如图 5-10A 所示。它不得不重新计划，将幻节点"清除 A"换成真节点，如图 5-10B 所

示；再将该节点展开，如图 5-10C 所示。这样就把原计划按新情况修订好了。

假若机器人局部失灵，这就要看情况处理。如果是工业机器人，就可以用另一个机器人替换它，而把它送去修理。但如果是一个探索宇宙机器人，送它到宇宙空间已化费了巨资，那就只有想尽一切办法克服这个故障，让它尽可能地完成尚未完成的任务。

一台控制机器人的计算机总是在它的记忆里存储着一大批经常要执行的行动计划。需要的时候，随时都可以取出来，而不必等到需要行动的时候才去重新编制计划。机器人可以通过存储新近研究的计划进行学习。在各种情况下，存储的不只是最详细的计划，而且是把整个分级计划都存储进去。因为我们知道，在执行计划当中如需要重新修订计划时，总是需要更高一级计划的帮助。

更困难的是，有时需要设法把一个计划分成几个部分并且把它们分门别类，以便我们把从某一任务中学到的行动计划抽取出来，插入到和另一任务有关的计划中去。

(二)教人

教人和教机器人主要的差别在于，教人时需要的详细程度往往因人而异，而且计划中每一步和另一步需要的详细程度也往往有别。一个人很可能熟悉计划中某一步的内容，因为他过去已经做过多遍，所以不需要对他细讲了；但是下一步他却可能不熟悉，需要详尽的指导。

解决的办法之一是，教学程序一开始使用较高级的、未经细化的计划。受教者对每一步都可以要求细讲。要求一旦提出，教学程序就把这一步转到下一级更加详细的计划。如果这一级还觉得不够清楚，教学程序可以转到更下一级，依此类

推。

有时受教者可能不知道怎样执行某一操作，或者执行得不正确。譬如一个人装配机件的时候，可能把机件装反了。这时教学程序就得修改原计划，帮助受教者克服困难。

(三)博弈

就我所知，过程网之类还不曾用到博弈上去。不过这很可能是它应用的一个领域，尤其是下棋程序。一般对于下棋程序的批评总是说，程序明显地缺乏获胜的长远规划。

博弈计划的一个有趣的特点是，计划必须随着博弈的进行而向前发展；因为导致胜利的最后攻击的计划，必须以弈者和对手地位的强弱对比为基础。

博弈程序可能从一个较高级的一般计划开始。计划的第一步可能扩展。扩展是以开局的通用规则为基础，例如发子和控制中心的规则。在下象棋的程序中，标准的开局往往存储在计算机里面，随时可以取用。

在博弈进行中，博弈程序要研究棋盘的开局，以便找到一种模式可以提示进攻或防守的有利路线。这些可以用来修改尚未下完的部分计划。修改应尽可能在较低的级别进行，以便在计划的过程中尽可能地维持程序。这有助于防止漫无目的地试验和不断地改弦易张。当然，倘若对手的一着完全出乎意料，原来的计划可能前功尽弃，程序计划将不得不从头开始。

由于对手的下法的不确定性，计划和监督执行这两件事在博弈游戏程序中，比在我们讨论过的任何其他问题中要更加紧密地结合进行。

第六章　博弈游戏程序：树状搜索

在本章和下一章中，我们要探讨一下难度较大的棋类游戏程序，比如国际象棋和西洋跳棋等等。用这些程序来同人或其它程序对弈。然而，有些程序是把计算机精心设计成一个棋盘，人们可以在其上对弈（或者是一种单人玩的纸牌游戏）。这种程序更接近于系统模拟的领域，而不属于人工智能的范畴。

一、博　弈　树

假定有二个人或者二台机器在下棋。我们把其中一名称为棋手，另一名称为对手。而我们始终从棋手的角度来观看这场竞赛。这样一来，如果棋手赢了、对手输了的话，我们就说这盘棋赢了；如果棋手输了、对手赢了，我们就说这盘棋输了。

假设现在该轮到棋手走了。在大多数情况下，棋手对这步棋可以有若干种选择。对于棋手的每一种选择，对手也有若干可供选择的相应棋步。对于棋手的每一步棋以及对手的每一步回棋，棋手又有自己进一步的选择。显然，这里所遇到的分支情况同我们在问题求解中遇到的情形是相同的。

实际上，我们可以把一盘棋想象成为具有一个入口（起始位置）和一组出口的迷宫。有些出口标上了“赢”的记号；有些出口标上了“输”的记号；而有些出口标上了“和局”的记号（图6-1）。在入口处，棋手选择某条路径起步（第一步）。在路径的第一个叉口，对手挑选了自己的路径回步（第二步）。棋手

和对手就这样轮番选择自己的路径走下去。棋手总是力争通向胜利的出口，而对手却总是把棋路引向输的出口。有时双方各自的努力不相上下，最后在“和局”出口结束棋局。或者他们一直在这个迷宫中徘徊，直到形势变得非常明朗：双方都是循环着兜圈子，这时只好双方握手言和。

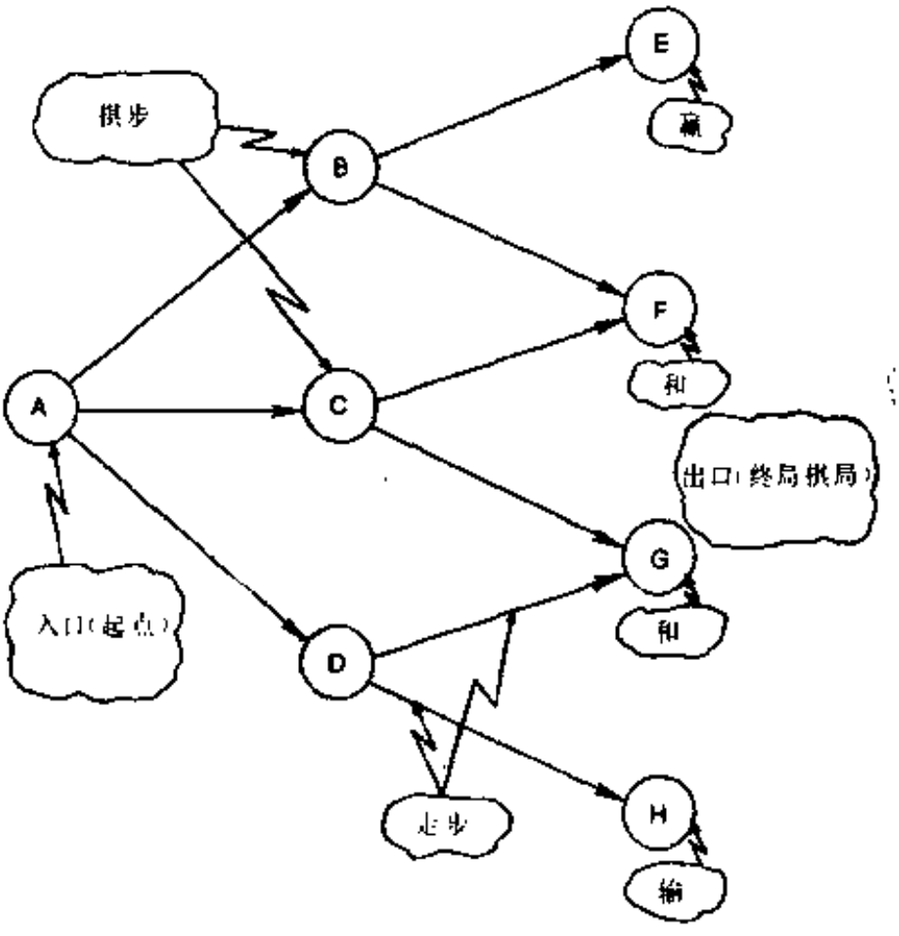


图 6-1　我们可以把一局棋赛看作一个迷宫。有一个入口对应其起始位置，还有标明“赢”、“输”、“和局”的若干出口

因此，下棋游戏同问题求解是相类似的，就是要在状态图中找出一条从初始状态到目标状态的路径。但是，它们之间却有一个很大的差别。在状态图搜索中，总是由一名选手来

选择下一步往那儿走。而在棋类的对弈中，棋手只有一半选择的权利，另一半由对手作出决定。棋手是一直朝着目标(或赢的位置)努力，而对手却是通过其每一棋步对此设置障碍。寻找机会把棋手从通往目标的路径上引开。

对于任何一种博弈竞赛，我们可以构成一个博弈树。它类似于状态图和问题求解搜索中使用的搜索树。如同图 6-2 所示。博弈树的节点对应于某一个棋局，其分支表示走一步棋；根部对应于开始位置，其叶表示对弈到此结束(无合法的棋步可走了)。在其叶对应的棋局中，竞赛的结果可以是赢、输或者和局。

所谓棋局，就是所有那些必须记录下来的信息。根据这些信息，比赛在按计划暂停以后能够得以继续进行下去。显然，这些信息包括了此时棋子在棋盘上的位置以及指出下步是轮到棋手走，还是对手走。

在人工智能的文献中，半步(half move)或者回步(ply)这种术语有时是用来表示棋手及对手走的一步。而整步 (fill move) 这一术语指的是棋手的一步及对手的回步。这些术语很可能来源于西方棋类对弈的纪录方法。在这类比赛中，白方的一步及黑方的回步都记录在同一行上，每一行只给出一个顺序号码。然而，在通常的情况下，一步就是由棋手或对手走的一步棋，而不是把双方的一步一回合在一起称之为一步。在本书中我们采用的正是这一通常的做法。

博弈树是一棵与/或树 (AND/OR tree)，不同于在状态搜索中使用的纯粹的或树。

其原因是：当轮到棋手走步时，他可以决定选择那一步。如果起码有一步可以担保棋手能达到赢的棋局，那么棋手就会选择这一步并保证能取胜。因此，对应于轮到棋手走的节

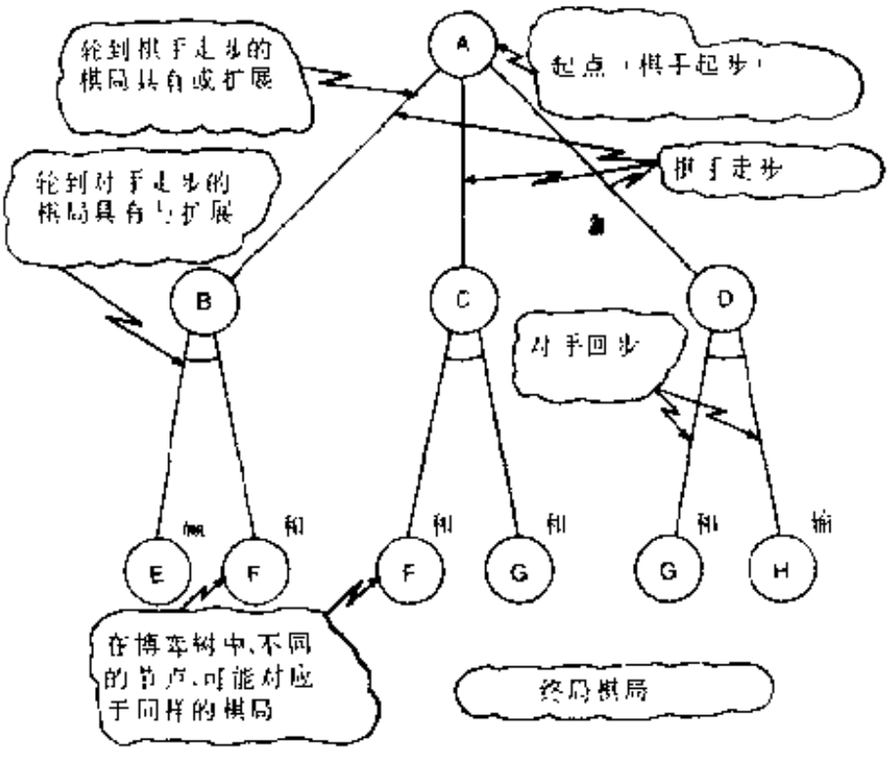


图 6-2　博弈树上的节点对应于某一棋局，根对应于起始位置，叶对应于竞赛结束时的棋局，枝对应于棋步

点是一个或扩展节点(OR)。

当轮到对手走时，选择是由对手决定的。棋手没有任何选择的权利。只有对手的所有棋步都会导致担保棋手会赢的棋局，这时棋手才一定会赢。因此，对应于轮到对手走的节点是一个与扩展节点(AND)。

对于一场经过深思熟虑的棋赛来说，其博弈树是非常之大的（对国际象棋来说有 10^{120} 个节点）。以至于不可能把这样大的博弈树存入计算机里，也不可能在任何合理的、有限的时间内进行详尽的搜索。尽管如此，首先深入地考察一下完整的博弈树，然后再看看如何来修正我们的原有想法，以便把搜索树修整到一个合理的规模。这样做还是很有意义的。

二、博 弈 策 略

假设我们对所讨论的博弈问题构造了一个完整的博弈树，我们希望能从中找出棋手应采取的策略。这种策略应能确保棋手会赢，或者起码能得到和局的结果。

首先我们把该博弈树的每一个节点标上 w（对应于赢）、d（对应于和局）或者 l（对应于输）。如果当前的棋局对应于标有 w 的结点，那么就存在一种策略可以担保棋手会获胜；如果节点标的是 d，那么除非对手失误，否则棋手最好的前景就是争取和局；如果节点标的是 l，那么棋手只好认输了，除非对手下错了棋。

对一个节点标以 w、d 和 l 的过程，同第四章中节点标以有解和无解的过程完全相同。

我们的讨论从叶节点开始。每一叶节点对应于一场棋赛结束的终局。根据博弈的规则，叶节点确定了棋手的赢、输和和局。这样，我们就把每一个叶节点相应地标上 w、l 和 d。如图 6-3 所示。

现在我们按照从叶往根的方向进行研究。按照每一节点的子节点的标号来标记该节点本身。节点标注的规则如下：

• 轮到棋手走步时（这时对应于或扩展节点），如果该节点的子节点至少有一个标有 w，那么，该节点标上 w；如果所有的子节点都标为 l，那么该节点标为 l。其它情况标上 d。

• 轮到对手走步时（这时对应于与扩展节点），如果该节点的子节点都标了 w，那么该节点标为 w；如果有一个以上的子节点标上了 l，那么该节点标为 l。其它情况标上 d。

根节点的标注表明，在对手不失误的情况下，棋手能够得到的最好结果。如果根节点的标注为 w，那么棋手稳操胜券；

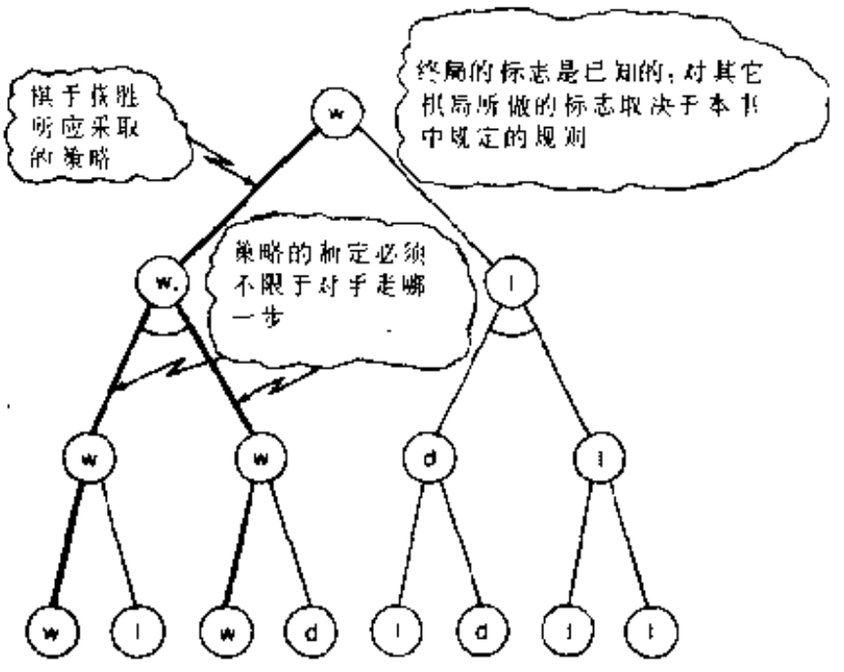


图 6-3　在一个博弈树的节点上标注 w（赢）、l（输）、d（和局）。我们先标注叶节点，然后往上一直标注到根节点

如果为 l，那么对手一定能击败棋手；如果为 d，那么在对手不失误的条件下，棋手在本盘竞赛中能得到的最好结果就是和局。

一场棋赛，若其根节点能标注上 w 或 l，并且是很简单易于分析的话，就可以成为赌徒的骗人棋局。根节点标注为 w 的话，不论谁先走，先走者都能赢；而根节点标注为 l 的话，不论谁后走，则后走者也保证会赢。当然是要采取正确的策略才行。赌徒知道那一方一定能赢，以及要赢应采取的策略。而这些，受骗者肯定是不知道的。

棋手的策略应该遵循这样的原则：如果有一步棋能走到节点为 w 的棋局，那么就应当下这一步棋；如果所有的棋步都通向节点为 l 的棋局，那只好放弃这盘棋认输。其它情况

下，就要走到节点标注为 d 的节点。

对手采取的策略正好相反：如果有一步棋能走到节点标注为 l 的棋局，那么就下这一步棋，如果所有的棋步都通向节点为 w 的棋局，那就只有放弃这盘棋认输。其它情况下，就要走到节点标注为 d 的节点。

当有二条以上的路径都能通往 l 节点，或者有二条以上的路径通往 d 节点时，棋手所采取的策略就不再是作出决定走那一步棋了。在实际对弈中，棋手总是想选择 w 节点，达到了 w 节点，就使得往后的对弈过程变得简单了。这样做就能减少棋手失误以至失去优势的机会。基于同样的理由，棋手在达不到 w 节点时，应当选择 d 节点。这样就能导致最复杂的情况产生。希望对手在这种情况下失误以便自己重新得到优势。至此，我们的讨论还是很不充分的。因为在所有的 w 节点或所有的 l 节点之间，我们并没有给出任何差别。

三、最小最大值方法

现在我们来看看博弈树节点标注的另一种方法：最小最大值方法。在整个博弈树尽管大得出奇、然而却只有一部分有用的情况下，运用最小最大值方法是有其优点的，很容易推广使用。

比方说，竞赛的结果是以钱为赌注的。为方便起见，设赌金为 1 美元。如果棋手赢了，他就获得 1 美元；如果他输了，则失去 1 美元。在和局的情况下，不存在钱的得失问题。

我们把棋手赢的钱数称之为收益。如果棋手赢了，其收益为 1，如果输了，收益为 −1。和局时，收益为 0。

现在我们来定义一个节点的值。该值即为棋手遵循上章所叙述过的策略而确保能得到的收益。因此，对于 w 节点，

其收益为1；对d节点，收益为0，对于l节点，收益为－1。

我们用不着一定要先把节点标上w、d或l，然后再去注明它们的值。可以直接计算出它们的值。

前面我们已经知道，对于每一个叶节点，比赛的结果或者是赢、或者是输、或者是和局。因此，我们先把比赛能赢的那些叶节点标上1，把和局的叶节点标上0，把输的叶节点标上－1。

如同我们前面做过的那样。从叶节点开始，其余节点按照其子节点的值来计算出它的值，一直到把根节点的值算出来。

这种计算后面所包含的想法是：棋手总是选择能得到最大收益的棋步；而对手总是选择通往最小收益的棋步（因为对手所赢的正是棋手所输的）。更确切地说，我们按下面的规则来计算每个节点的值：

1. 轮到棋手走时（此时对应于或扩展节点），节点的值是其子节点值中的最大值。

2. 轮到对手走时（此时对应于与扩展节点）节点的值为其子节点值中的最小值。

如何用最小最大值方法来计算节点的值，这在图6-4中作了说明。把它同图6-3比较，可以看出，最小最大值方法是同w—d—l标注方法等价的。

在讨论最小最大值方法时，棋手和对手常常是作为最大(Max)及最小(Min)来考虑的。Max，即棋手，总是选择能走到最大值节点的棋步，而Min，即对手，总是选择能走到最小值节点的棋步。

至此，最小最大值方法已经给我们提供了一个简明而等效的手段来标注博弈树的节点。然而，这种方法不是采用任

意的标号，如 w、d 和 l，来标注节点，而是使用数字来标注。这就使得我们能够很容易地推广这个方法，不用把节点的值局限于 1，0 和 −1。

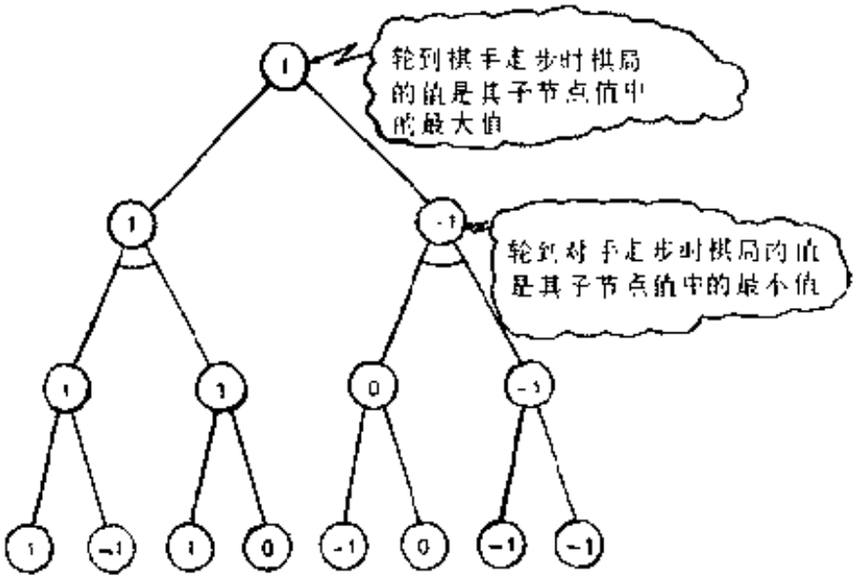


图 6-4　用最小最大值方法来标注博弈树节点的值

四、终局和静态评价函数

现在我们终于要面临这样一个不能回避的问题：即对于象国际象棋这样一类棋赛，我们不可能建造一个完整的博弈树。甚至也不可能建造该博弈树的基本部分。我们充其量所能做到的，就是围绕目前正在下的棋局，把博弈树中的很小一部分建造出来。

这样一来，我们所建造起来的博弈树的根就不再是棋赛的起始棋局了，而是轮到棋手走的一个棋局。在博弈树中，我们对那些已经下过的棋局并不感兴趣，而只关心那些将要下的棋局。

同样地，这时博弈树的叶也不再代表这场棋赛赢、输或和棋的终局，而是表示在合理的时间及允许的存贮空间内，竞赛

路径所能达到的博弈树的最远节点。

我们还是沿用以前采用的方法，把对应于叶节点的棋局叫终局。那么，什么是终局的判断标准呢？或者更确切地说，在探索一条特定的竞赛路径时，什么时候中止这种探索对竞赛才是保险的呢？

一般我们倾向于首先对终局标上它们各自的值，用以反映每个终局对棋手的价值。然后我们用最小最大值方法来处理这些值。因此，在考虑了双方所拥有棋子的数量、类型以及它们在棋局中的作用等因素以后，终局的值就可以很可靠地评价出来。

如果有一个棋局不能用这种方法来进行可靠地评价，那么该棋局就不是一个合适的终局。假设现在玩的是西洋跳棋，并且轮到棋手走了。这时只有一次连续的跳步（西洋跳棋亦因此得名），而且这一跳步一下子就能摧毁对手在棋局中的力量。显然，在这种情况下仍然要按照双方力量的布局来评价该棋局的值是没有意义的。因为在这一（连续）跳步之后，对手棋局的形势就会急骤恶化。

上述只是一个极端的例子。在一般情况下，如果轮到下棋的这方走不到好的终局，或者得不到把棋子变成王的棋局，而对方的下一步棋又对棋局的性质产生了本质的影响。这时，对该棋局进行值的评价是没有多大意义的。

简单的做法是：我们对“活”棋和“死”棋进行区别。所谓活棋，就是走了一步棋以后，双方力量的对比会产生重要变化的棋局。而所谓死棋，就是找不到这样一步棋的棋局。哪一棋局可以看成是活棋，哪一棋局可以看作为死棋，这取决于对棋局进行评价时所采用的准则。

这样的话，终局应当是死棋局。在活棋局中，有时由于时

间或存贮空间的限制，偶而也会出现强迫中断对竞赛路径探索的情况。但是，如果时间或存贮空间没有限制的话，那么对竞赛每条路径的探索会一直延伸到死棋局为止。

具体来说，在下棋程序中有这样一段：TERMINAL (POSITION, DEPTH)。它是用来判别博弈树上的具体节点是否是叶节点。正如括号中诸项所表明的那样，TERMINAL 的判定取决于相应棋局的细节以及节点的深度。

TERMINAL 通常用来衡量一个棋局对于节点深度的活性，即在下一、二步棋内棋局能够发生有意义变化的机会的大小。一种典型的策略包括以下几方面的内容：

1. 如果 DEPTH（深度）小于 3，则该棋局不是终局。每一对弈路径的深度至少应当延伸到 3。

2. 如果 DEPTH（深度）为 3 到 10，那么在一个棋局为死棋局时，该棋局就是终局；在棋局为活棋局时，该棋局就不是终局。我们在该深度范围内，可以沿着一条对弈的路径往前探索，直到发现死棋局为止，然后中止探索。在通常情况下，活棋局的确定程度是随着其深度的增加而变小的。在深度为 3 时，如果一个棋局具有在几步之内就可以出现有利局势的任何特征，那么该特征就足以使棋局变活。在深度为 10 时，仅仅只有在下一步棋能够对棋局产生急骤变化时——比方说吃掉一个重要的棋子——才可能足以把棋局变活。

3. 如果深度为 11，那么该棋局即为终局。对弈路径的延伸在深度为 11 时无论如何要中止，而不考虑棋局是死是活，从而确保剩下的时间及存贮空间能够用于其它对弈路径的探索。

对于一个具体的程序而言，其深度中止的分界点（如 3、10 及 11），通常必须经过实验之后来确定。以便程序能够得

到好的棋局而具有足够的远见，但是又不至于在探索那些无意义的对弈路径上化费过多的时间及存贮空间。大家知道，选择这些深度值的过程就是所谓程序调整。

一个终局已知的话，我们需要找到一些方法来确定该棋局对于棋手的值，这样才能运用最小最大值方法来找到棋手的最佳棋步。因此，一个下棋程序，都有一个 TERMINAL—VALUE(POSITION)（终局值）的处理方法来确定某棋局对于棋手的值。这就是所谓的静态评价函数。它之所以得到此名，是因为它是以棋子在棋盘中的静态分布为基础进行估算的，而不是基于对一系列可能棋步的动态探索。

我们仍然可以把棋局的值看成为棋手在该局中能确保得到的收益。但是在终局的情况下，对弈的结果不是赢，就是输，或者为和局。因此，这时把赌注全押在一方（棋手或对手方），或者双方均摊都不是合理的做法。取而代之的方法是，在已达到我们所指出的终局时，这场竞赛就得暂停。理由可以是棋手或对手在另外什么地方有紧急事务要去处理。随后我们让一位公证人来决定如何分配这笔赌注。分配的依据就是棋局中双方力量的对比。用这种方法棋手所得到的收益数目就是该棋局的值。TERMINAL—VALUE 这一终局值处理方法必须尽可能模拟这位公正的专家对棋局的评价（请把该确定棋手的棋局值的评价函数同第三章中确定达到目标的花费所采用的评价函数进行比较）。

TERMINAL—VALUE 确定棋局值的过程，必须以该棋局的特征为基础。对于西洋跳棋及国际象棋这样一些棋类，它们的一些重要特征有：

1. 兵力。这是指在棋子，或者说“人”数上的优势。不同棋子的相对的值都要考虑到。例如在国际象棋中，王和一般

棋子的相对值为 3 比 2（人们一般愿意用 3 个棋子去换 2 个王）。这样，在对一个棋局进行评价的过程中，我们可以把双方拥有的每一个王算作 3 点，而把每一个棋子算作 2 点。

2. 机动性。所谓机动性就是一个棋子可以移动到的方格数。此外，机动性的定义是可以变化的。有时候，任何合法的棋都属于机动性；有时候，只有棋子在走了一步以后下一步不被吃掉的情况下，该步才属于机动性；有时候，只有攻入对方领地的那些棋步才属于机动性。

3. 关键棋格的控制权。在多数的棋赛中，都会有一些关键性的棋格。对这些方格的控制具有战略上的重要性。通常情况下，这些关键性的棋格处于棋盘的中心位置。我们可以依据对关键棋格所控制的数目给双方打分。

4. 特定模式。对于每一场棋赛，都会有一些公认为是具有战略意义的棋子的布局模式。可以给双方各自棋局中的每一模式打分。

对于上述的每一个特征，我们都计算出其优势，即每一特征的优势为棋手对该特征所得到的分数减去对手对同一特征所得到的分数。

在最简单的情况下，根据每一特征的重要性，我们把每一优势乘以一个相应的数。然后把所有的结果相加得到该棋局的值。更为精确的算法是采用一张表，根据各种优势值的每一种组合，该表都给该棋局确定一个值。这种表允许考虑各种特征之间的相互作用（例如可以是只有在另一个特征存在时，某一个特征才是重要的）。当我们只是计算出每一特征的值，再简单地把它们加在一起时，并没有考虑到它们之间的相互作用。

下面我们来看一个非常简单的西洋跳棋的评价函数。它

包括三项：

1. k＝王的优势

＝棋手拥有的王的个数—对手拥有的王的个数

2. m＝棋子的优势

＝棋手拥有的棋子的个数—对手拥有的棋子的个数

3. mb＝机动性优势

＝棋手的机动性-对手的机动性

我们用“棋子”来表示除王以外的其它棋子。一个棋子可以占据它而不会被吃掉的方格数就是各方的机动性。

现在，我们要来决定王的优势、棋子的优势以及机动性优势之间的相对重要性。西洋跳棋的传统看法认为，王和棋子的相对值是 3：2。假设我们决定每一个棋子比增加一个机动性方格要重要 4 倍的话，那么其评价函数就是：

值 value＝6k＋4m＋mb

找到一个合适的数来乘以每一个特征优势也是属于调整的问题。

如果棋局已经到了赢、输、和见分晓的地步，那么该棋局就可以作为特殊情况处理。赢的棋局给一个大的正数作为它的值；输的棋局的值是上述那个正数的负值；和局的值仍然取 0。赢的棋局所得的确切值的大小是无关紧要的，只要比评价函数对任何一场非赢棋局所算出的值都大就可以了。

值得注意的是，评价函数是用这样的方法来定义的：如果棋手与对手突然相互交换位置，即棋手接管了对手的棋子、对手接管棋手的棋子。这时，棋局的值只要简单地变换一下其代数符号就可以了。换个方式来说，比如一个棋局对棋手来说其值为 100，那么对于对手来说，该棋局的值即为－100。

五、深度优先的最小最大值评价方法

现在我们回头来看看棋手对当前的棋局是如何考虑的，以及应如何决定下一步棋的走法。假设有了一种方法，它对任何棋局都可以给出所有的合法棋步，同时还能给出采用了这些合法棋步中任何一步以后所形成的新的棋局。另外我们假设已经有了上一节所谈到过的 TERMINAL 及 TERMINAL-VALUE 处理方法。我们希望能决定棋手应走的棋步。

在图 6-5 中，我们从当前的棋局开始，将其扩展，然后再扩展它的子棋局，如此类推。这样我们就得到了一个部分博弈树。但是，每一个节点在扩展之前，我们都要用 TERMINAL 来检查一下该节点，看它是否为终局。如果检查结果是，那么应该不扩展这个节点；如果不是的话，就扩展它。应当注意的是，TERMINAL 中使用的 DEPTH（深度）是指其根为当前棋局的部分博弈树的深度，而不是指根为初始棋局的整个博弈树的深度。

上述的扩展一直进行下去，直到所有的节点都不能再扩展为止。最后我们得到了这样一个博弈树，它的根对应于当前的棋局，而其叶对应于由 TERMINAL 判定为终局的棋局。

对于每一个叶节点，我们都可以用 TERMINAL-VALUE 的方法来处理其相应的棋局并且计算出节点的值。

应用最小最大值方法，我们对该博弈树的每一个节点都标注上一个值。如前所述，棋手所要走到的节点，它的值是其所有子节点值中的最大值；而对手所要走到的节点，它的值是其所有子节点值中的最小值。图 6-6 说明了这种计算方法。由于这些节点值的计算方法本身的缘故，那些非叶节点所标注的值通常称之为回溯值（back—up values）。

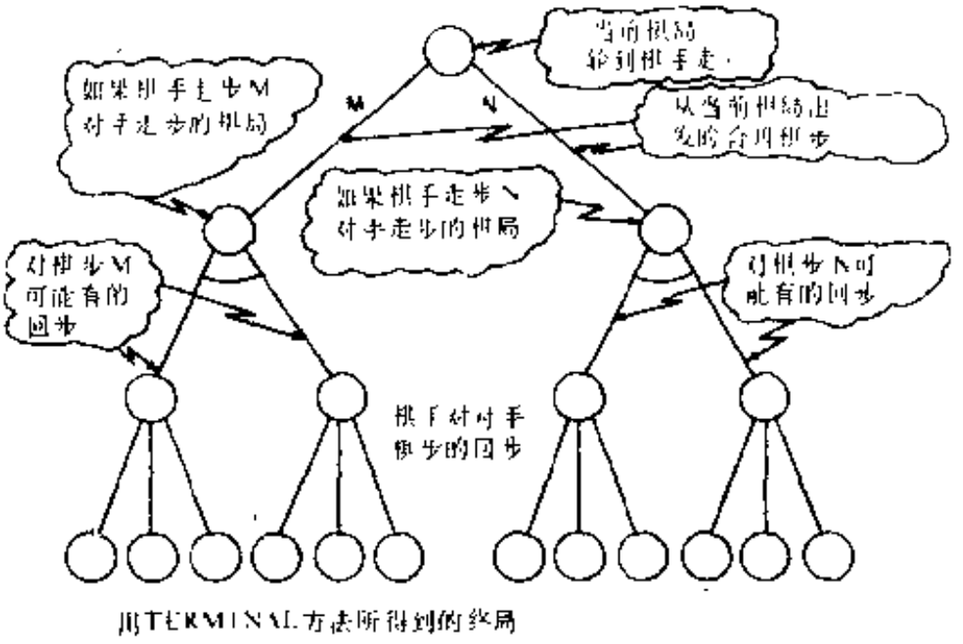


图 6-5　用来从当前棋局开始去寻找最佳棋步的部分博弈树。博弈树的根对应于当前棋局；其叶对应于由 TERMINAL 方法所得到的终局位置

对棋手来说，他所选择的最佳棋步就是能够得到具有最大回溯值棋局的棋步。

在对手的下一步棋走完以后，那么就要从这时的当前棋局开始建造一个新的博弈树。这一过程会不断重复。因此，每一个部分博弈树实际上只是用来决定一步棋。真正用到的值仅仅是根的子节点的回溯值。这些子节点所对应的棋局就是棋手自由选择的棋步所能走到的棋局。

实际上，我们的确用不着一次就把整个博弈树存贮起来。如果运用一个叫做深度优先的最小最大值方法，那么在只存贮整个博弈树的很小一部分的情况下，我们也能计算出根的子节点的值。当节点需要计算其值时，我们就产生这些节点；当节点已完成它们的使命之后，我们便删除掉这些节点。

同所有的深度优先的方法一样，深度优先的最小最大值方法是从根部开始处理，沿着从亲节点到子节点的方向进行，

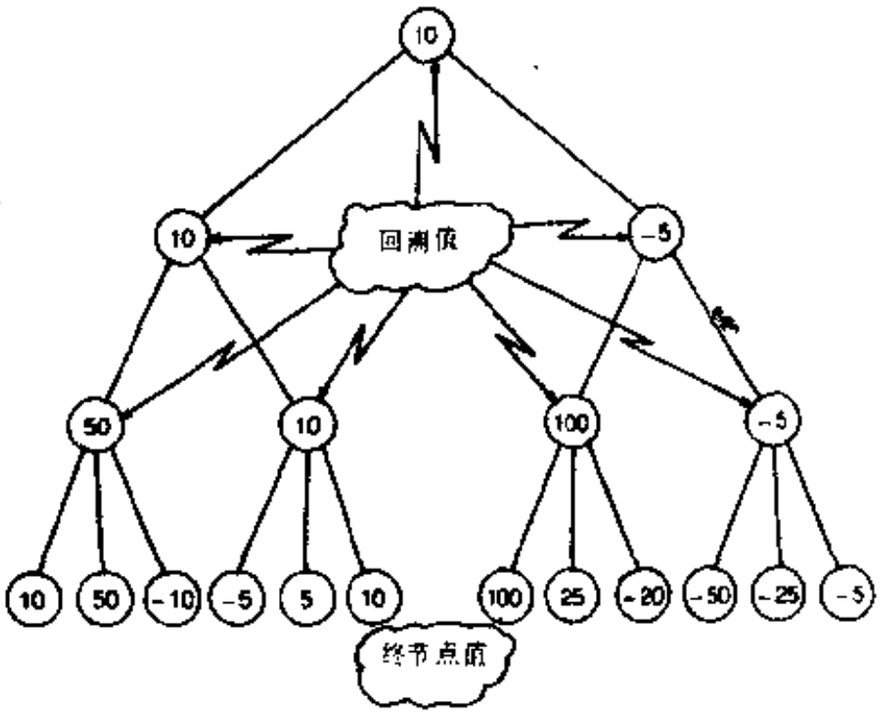


图 6-6　运用最小最大值方法对部分博弈树的节点进行标值。叶节点的值是用 TEMINAL—VALUE 计算出来的。最小最大值方法把这些值回溯到其它的节点上去

直到遇到终局为止。然后再返回，一直到找到另一条未曾探索的路径，再沿着这条路径行进到遇见终局为止。如此类推。图 6-7 表示了用深度优先的最小最大值方法对节点搜索的先后次序。

这一方法可以同时既建立博弈树，又对其节点进行估值计算。在沿着从亲节点到子节点的方向采用这一方法时，节点是扩展的。如同图 6-8 所示，当节点第一次扩展时，只新建了一个子节点。其它子节点只有在过程返回并重新访问该节点时才建立。只有那些处子当前正在扩展路径上的子节点以及那些尚未标注回溯值的子节点才被存贮起来。

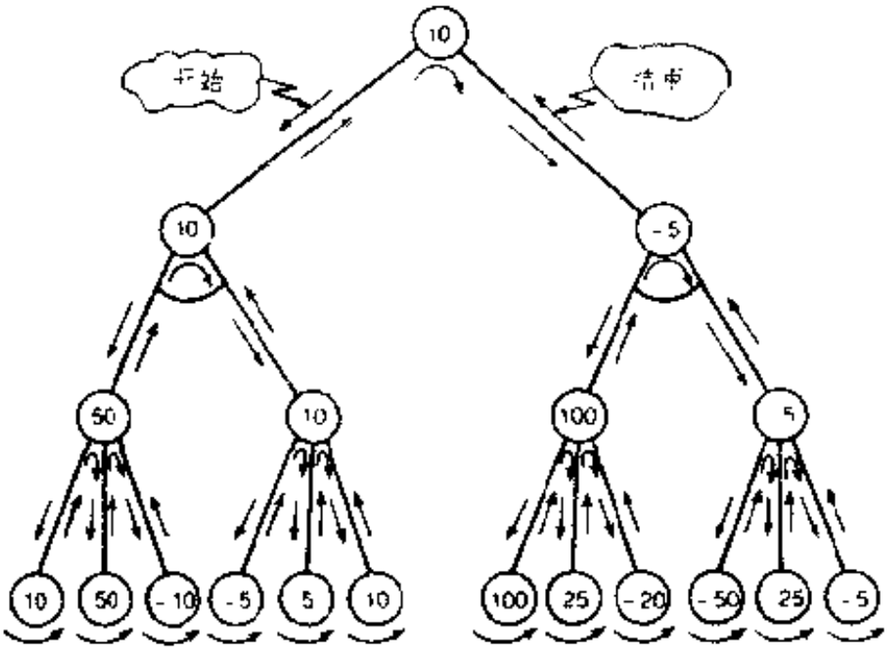


图 6-7　采用深度优先的最小最大值方法所得到的带分搜索博弈树的路径

在采用这种方法达到终节点时（可以用 TERMINAL 方法来识别终节点），就用 TERMINAL－VALUE 方法来处理该终节点所对应的棋局，并对该节点标上评价值。然后，这种过程又回溯到终节点的亲节点上去。

如前所述，在回溯期间如果访问到某个节点，通常要对该节点继续进行扩展，产生一个新的子节点，进而再对该新的子节点进行处理。实际上，该节点的所有子节点最终都会产生出来（即相应棋局的所有合法棋步被探查到）。在处理过程重新遇到该节点时，就根据其子节点的值来计算出该节点的回溯值。是取这些子节点值中的最大值还是最小值作为回溯值，这取决于轮到那一方下棋了。这时，所有的子节点就不再有用，可以删除掉。程序过程又回溯到刚刚赋值的亲节点上，再继续上述过程。

除了那个正处于扩展路径上的子节点以外，我们不需要存贮任何别的子节点。如同图 6-9 所示，我们对每一节点只

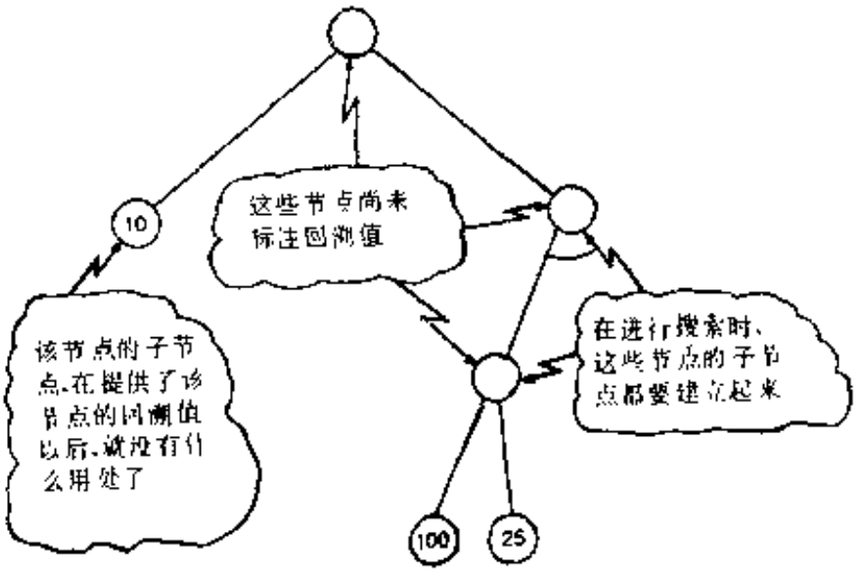


图 6-8　只有在需要时我们才产生节点，在节点不再有用时我们就删掉它们。这样，我们在采用深度优先的最小最大值方法时就可以节省存贮空间

标出了其部分回溯值。在该节点产生时，该值作为节点的初始值；在以后节点每次被重新访问时，该值都要更新。在任何时候，一个节点的部分回溯值总是已计算出来的子节点值中的最大值或者最小值(这取决于轮到谁下棋)。当一个节点的所有子节点的值都计算出来以后，其部分回溯值即为该节点的值，亦是其所有子节点值中的最大值或最小值。

对于轮到棋手下棋的一个节点来说，其部分回溯值一开始就给定为一个大的负值。当重新访问到这个节点时，就要把这个值同其刚刚计算出的子节点的值进行比较，取它们中大的那个数作为该节点的新的回溯值。

对于轮到对手下棋的一个节点来说，其部分回溯值一开始就给定为一个大的正数。当重新访问到该节点时，就要把这个值同其刚刚计算出的子节点的值进行比较，取它们中小的那个数作为该节点的新的回溯值。

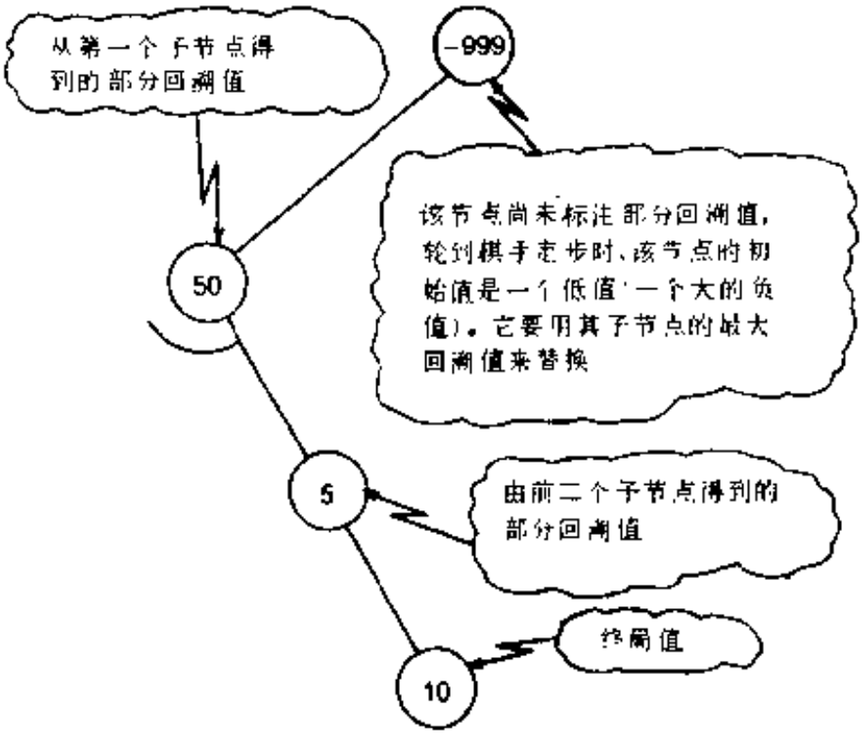


图 6-9　即使对每一个节点都保留一个回溯值，我们也可以减少需要存贮的节点数目。一个节点，只要对它的一个新的子节点计算出值，那么该节点的部分回溯值就被更新（如果有必要的话）

如同图 6-9 所示，在我们需要存贮的任何时刻，应该存贮的节点仅仅是从根节点到该时刻当前节点之间路径上的那些节点。把这些节点存贮到堆栈中去是非常方便的。在堆栈中，根节点放在底部，而当前节点放在顶部。

尽管我们能实现只存贮博弈树中从根节点到当前节点的那一部分，但是却不能保证说这一定是最佳的方法。有些程序就是把整个博弈树存贮起来，以便先前探查过的对弈路径能被重新访问或进一步地探查。这在先前的路径比之后来探查的路径更能得到好的结果时，是很有必要的。然而，在存贮空间受到限制时，关于每次只存贮尽可能少的节点这一想法却是富有吸引力的。

六、α-β 方 法

在深度优先的最小最大法中，我们可以看到，博弈树的某些部分并不会产生任何有意义的值，因而也根本用不着去扩展博弈树的这一部分。识别博弈树中这些可以忽略部分的技术，称之为α-β法。之所以叫这个名字，是由于历史原因造成的。

考虑由图 6-10a 所示的情况，我们可以看出，在轮到棋手下棋的节点上，其部分回溯值为 10。而其当前计算出来的子节点的部分回溯值为 8 *。现在，由于该子节点是轮到对手下棋的节点，而对手总是要走到那个具有最小值的棋局，故进一步探查的结果只会小于这个值。无论其子节点最后的确实值是多少，它总是小于或等于 8[⊖]。

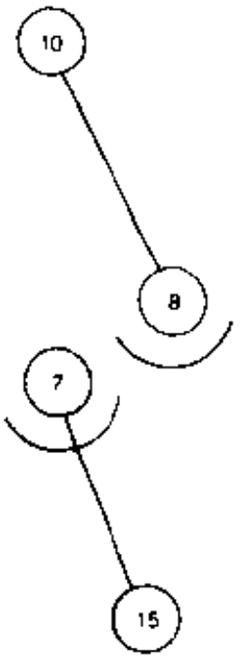
从另一方面来看，该节点本身的部分回溯值为 10。因为这时轮到棋手下棋，所以只有大于 10 的子节点值才能改变这个部分回溯值。

结论是：我们不需要去进一步扩展其子节点或其它任一后裔节点。这是因为进一步的扩展至多只能减少其子节点的部分回溯值，而其目前的值已经足够小到不能影响其亲节点的部分回溯值了。这种情况就是所谓的α剪枝。

现在，我们把一般的原则叙述如下：

在考虑轮到棋手下棋的一个亲节点及轮到对手下棋的一个子节点时，如果该子节点的回溯值已经小于或等于其亲节点的回溯值，那么就不需要对该子节点或者其后裔节点作更多的处理了。计算的过程可以返回到亲节点上。

⊖ 原文误为 5。——译者注



(a) α 剪枝。轮到对手下棋的子节点上，其部分回溯值已经小于轮到棋手走步的亲节点的部分回溯值。这时，采用 α 剪枝法，不需要进一步往下探索该子节点的后裔节点（进一步的探索只会减少该子节点的部分回溯值，而目前，它的值已经足够小，不会影响其亲节点的值）

(b) β 剪枝。轮到棋手走步的子节点上，其部分回溯值已经大于轮到对手下棋的亲节点的部分回溯值。这时采用 β 剪枝法，不需要进一步往下探索子节点的后裔节点（进一步的探索只会增大该子节点的部分回溯值，而目前，它的值已经足够大，不会影响其亲节点的值。）

图 6-10　α-β 方法可以检查出，在什么情况下对子节点的进一步探索已经不能影响其亲节点的值。既不需要产生其子节点的子节点（或其后裔），也用不着去计算它们的值

当亲节点是轮到对手下棋的一个节点时，该原则要作相应的改动（图 6-10b）：

在考虑轮到对手下棋的一个亲节点及轮到棋手下棋的一个子节点时，如果该子节点的回溯值已经大于或等于其亲节点的回溯值，那么就不需要对该子节点或者其后裔节点作更多的处理了。计算的过程可以返回到亲节点上。

这种情况就是所谓的 β 剪枝。

α-β 剪枝这一技术允许我们有时可以不去考虑某节点的某些子节点的情况到底如何。然而，由于非终节点的每一个子节点又是其后裔节点所组成的整个博弈树的根，所以，我们

忽略了那些子节点的话，不仅仅是忽略了它们本身，还忽略了它们的所有后裔节点。因此，这一技术可以删去数量相当大的节点，因而也就大大地节省了探索博弈树所需要的时间。

在下一章，我们要接触到另外一些能减少探索时间的启发式方法。

第七章　博弈游戏程序：启发式方法

上一章讲述了一些建造和搜索博弈树的方法。这些以及所有其它的树状搜索方法的致命弱点就是组合爆炸问题。除非严格控制博弈树的规模，否则它很快就会变得异常庞大，以至都找不到一个合理的存贮空间来存贮，也不可能在一个合理的时间内对该树进行搜索。

我们可以采用启发式方法来限制树的宽度和深度，从而来控制博弈树的规模。这里广度是指每一局棋中所能考虑到的棋步数(如果所有可能的棋步都考虑到了，那么我们进行的搜索就是全广度搜索)；而深度是指沿着某一路经向前探查到的下棋步数。

同样地，我们必须要对终局的值进行评价。这里，终局对应于博弈树的叶节点。这是一个模式识别的问题。如果把一盘终局摆在一位名棋手面前，他就可以对该棋局作出评价：说出那一方占有很大的优势；还是只占一定的优势；或者双方势均力敌。如果进而要求该名棋手对其中评价作出解释，那他就会指出那些棋子的布局(即模式)对这一方或那一方是有利的。我们也需要计算机能象名棋手一样，可以识别出这样的棋局(即模式)，同时还能对相应的评价函数进行运算。

如前所述，对于下国际象棋的程序，最通常的做法是并不要这种程序包含一长串的规划。由计算机作出的每一棋步选择，在不考虑前后棋局的情况下，经过检查可以认为是非常合理、非常好的。但是，综观一系列的棋步后就可以看出，就机器来说，它并没有一个完整的取胜计划。遗憾的是，在国际象

棋程序中采用长期规划的方法，实际上只取得了很少的成果。现有的这类程序主要是以树状搜索及评价函数为基础的。在本书中，我们所能做到的至多也是提出一些解决这个问题的可能方法。

对一个博弈树的搜索，其所需要的时间与贮存空间，取决于在计算机内是如何把竞赛的棋局表达出来的。我们现在开始讨论这些问题，并从棋局的表达方法入手。

一、棋局的表达

迄今为止，对于国际象棋和西洋跳棋这类棋类游戏，已做了大量的程序方面的工作。我们的讨论将限于这些棋赛的范围。当然，要把这些讨论推广到其他对弈竞赛中去，是不会有什么困难的。

(一)棋子

我们需要找到一些在计算机中表达棋子的方法。人们经常采用的方法是简单地用数来表示棋子。正数代表机器的棋子，负数代表对手的棋子。表 7-1 是国际象棋棋子的一般表达方法。

表 7-1　国际象棋棋子的表达

棋子名称	计算机方	对手方
兵	1	−1
马	2	−2
象	3	−3
车	4	−4
王后	5	−5
王	6	−6

(二)棋盘

国际象棋和西洋跳棋的棋盘是二维平面，有八列和八行。因此，我们可能首先想到用一个 8×8 维数组来表示这个棋盘

(BOARD)，如图 7-1 所示。而 BOARD(1，4)的值就代表了占据第 1 行与第 4 列相交那个方格的棋子代码。比如，在图 7-1 中，BOARD(1，4)=5，按照刚刚谈过的国际象棋棋子方法，可以看出，上面的那个方格是由机器方面的王后所占据。

自然，一个二维数组处理起来是不如一维数组那么方便。它有二个下标变量需要处理，而不是一个下标变量。此外，为了找到相应于下标变量值的地址，还必须进行计算。这方面二维数组要比一维数组复杂得多。由于这些原因，尽管棋盘的结构是二维的，通常我们还是用一维数组表示棋盘。

图 7-2 表示的一个一维数组——BRD。是一个国际象棋的棋盘。BRD 有 64 个单元，每个单元对应于棋盘上的一个方格。在图 7-2 中，标上了数字的棋盘表示 BRD 的单元同棋盘方格之间的对应关系。第一行的方格对应于 BRD(1) 至 BRD(8)，第二行的方格对应于 BRD(9) 到 BRD(16)，以此类推。在用 BRD 的形式存贮棋盘时，我们把棋盘分解成行，再一行一行地把它存贮起来。我们也可以不用这个方式，而是按列的方式把棋盘贮存起来。但总的看来，似乎以行的方式存贮棋盘更为普遍一些。

把棋盘表示为一个一维数组，会碰到这样一个问题—如何决定棋盘的边界。比如说，在 BRD(8)有一个棋子王，程序可能想把它移到 BRD(9)。然而，从 BRD(8)移到 BRD(9)并不意味着是从一个方格移到相邻的另一个方格，而是从第一行最右边的方格移到第二行最左边的方格。因此，我们需要简单的方法让程序懂得，类似这样的棋步是犯规的。

这个问题可以这样来解决。如图 7-3 所示，在棋盘的每一边都额外增加一列或一行作为边框，并且给其上的每个方格定一个值。这个值只要不曾用来表示过棋子或空格就行。

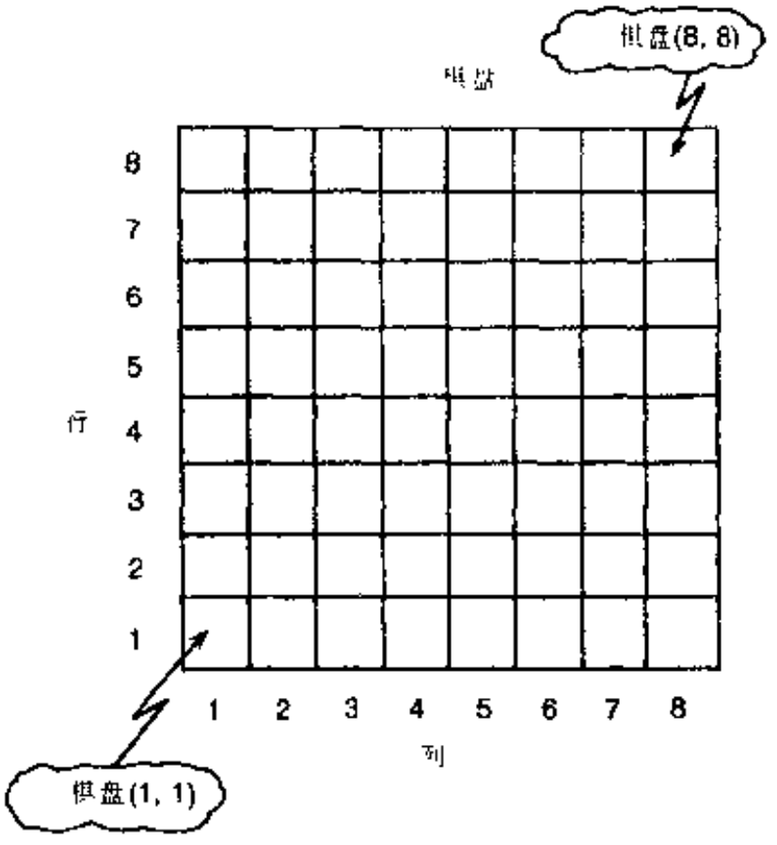


图 7-1 国际象棋的棋盘可以用一个二维数组——BOARD 来表示。这种表示方式的不足之处是：存贮一个二维数组所必需的计算要比存贮一个一维数组的要复杂得多

在图例中，我们给定为 7。在一维数组中，棋盘的每一行同下一行是由二个边框方格隔开的。如果程序移动到边界外边，它就落到了边框的某一个方格之中。这时，边框方格的值 7 就会提醒程序，它已经越过了棋盘的边界。

我们还可以在棋盘的底部加上二行边框方格，在其顶部也加上二行边框方格。这样，程序就可以用检查左右边界越线的同样方法来检查自己是否上下越界。这样一来，我们就得到了图 7-3 中所示的数组 BD。BD 数组共有 120 个单元。实际上，我们是用增加程序表达空间的代价来节省程序运算

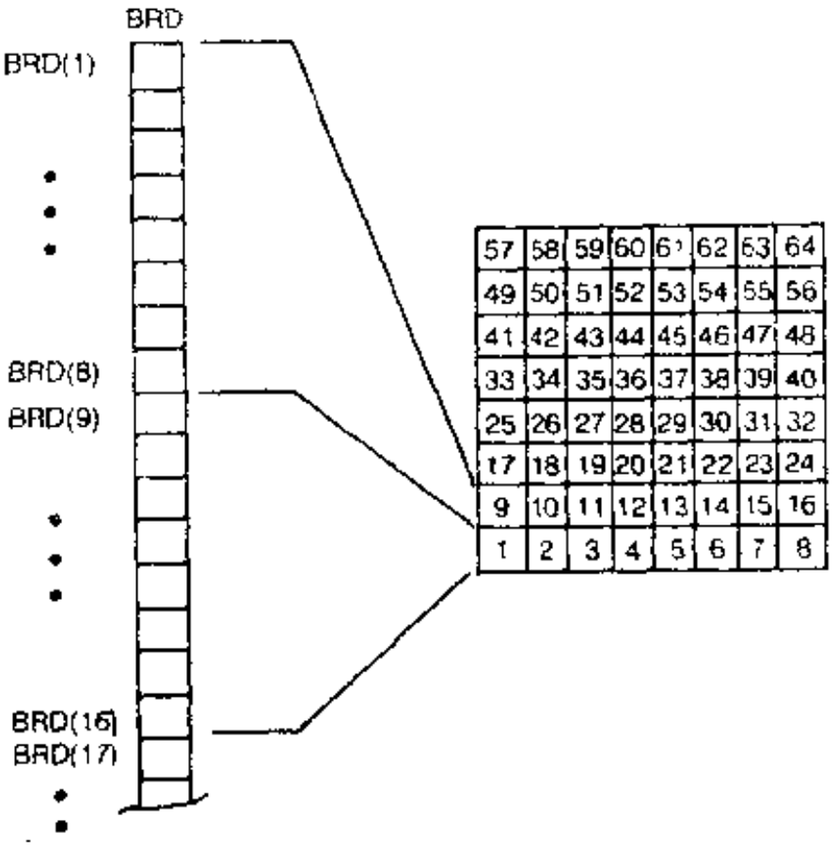


图 7-2 用一个一维数组—BRD 来表示国际象棋的棋盘。棋盘上标有数字，数字表明 BRD 上该单元对应的方格。箭头表示各行的元素是如何存贮在 BRD 中的

所需要的时间。采用 BD 的程序运行起来要比采用 BRD 的程序快。这是因为采用 BD 后，对于棋盘边界的检查要更为简单的缘故。

在 BD 数组中，我们之所以在棋盘的上下边界外加了二行边框方块，在相邻的二行之间加上二个边框方块，这是考虑到国际象棋中马这个棋子的移动规则的缘故。马每走一步是在一个方向上移动二格而在另一方向上移动一格。

(三)棋步

下棋程序必须知道所有棋子的合法棋步。我们可以这样

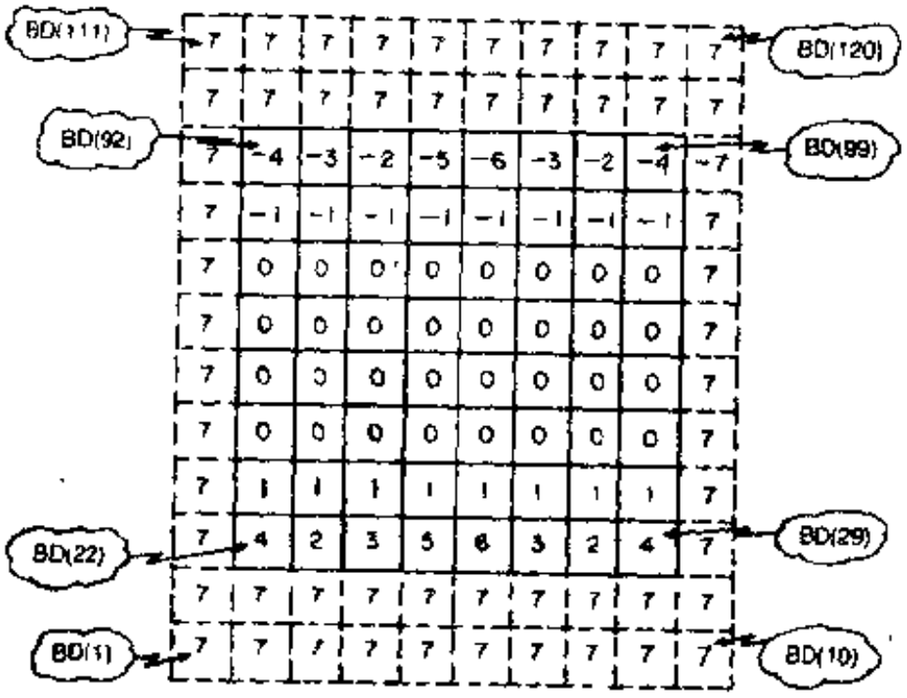


图 7-3　在实际的棋盘周围加虚线的方格。每个虚线方格给定一个特殊的值(图中为 7)。这样可以最有效地检查棋盘的边界。最后形成的 12×10 的棋盘可以用一个线性数组 BD 来表示。该数组共有 120 个单元。该图表示了棋局开始时,BD 数组的情况

来提供这方面的信息，如图 7-4 所示。棋盘上每一棋子的棋步可以用一个数字来表示，该数字同该棋子所在方格的数字之和即可为该棋子所应走到的那个方格的数。因此，在采用 BD 数组的方式来表示棋盘时，棋子所在方格的数加上 1 就等于把棋子向右移动一步，加上－1 就是等子把棋子向左移动一步；加以 10 就是把该棋子朝棋盘的上方移动一格，加以－10 就是把该棋子朝棋盘的下方移动一格。以此类推。

每一个数字都代表了一个棋步方向。例如，11 表示棋子沿对角线方向朝右上方走了一步。把某棋子所在方格的数字

加上 11，就是把这个棋子移到对角线上的右上方的方格上。再加上 11 就又把该棋子朝同一方向移动一格。等等。不断重复地加上 11，我们就可以把棋子移动到我们所希望移动到的方格，直到遇到已被别的棋子占据的方格或者碰到棋盘的边界为止。

在图 7-4 中用到的数字给出了国际象棋棋子的移动方向。有些棋子，如王、马和兵，只能在给定的方向上移动一个方格（对马来说是移到与它原来所在方块不相邻的那个新方格）。因而对这些棋子来说，每次只能加上一个移动方向的数。其它棋子在特定的方向上可以移过任意数目的方格。因而对这些棋子来说，只要不遇到被别的棋子占据的方格或者走出棋盘的边界，就可以把该棋子所占据方格的数字同一个移动的方向的数重复地相加。每次相加的和就指出了该棋子所移到的新方格。

一个棋步还可以用棋子所离开的方格及其所达到的方格来表达。因此，25—35 表示一个在方格 25 的棋子走到方格 35 的棋步。通常更为方便的做法是既包括了移动棋子的类型又包括了棋步的起步方格及终止方格。例如，6：26—36 表示机器的棋子王从方格 26 移动到方格 36；而—5：99—55 表示对手的王后从方格 99 移动到方格 55。

存贮棋步同存贮棋局比较起来，它所需要的存贮空间要小得多。因此，对于已经存贮起来的博弈树的每一个节点，大多数下棋程序并不把这些节点所对应的棋局存贮起来。而是把从一个节点到下一个节点所需要的棋步存贮起来。需要存贮起来的棋局只是当前需要进行的这一棋局。在程序往博弈树的叶节点方向深入时，它就给出相应的棋步；而在往博弈树的根方向进行回溯时，它不确定棋步。

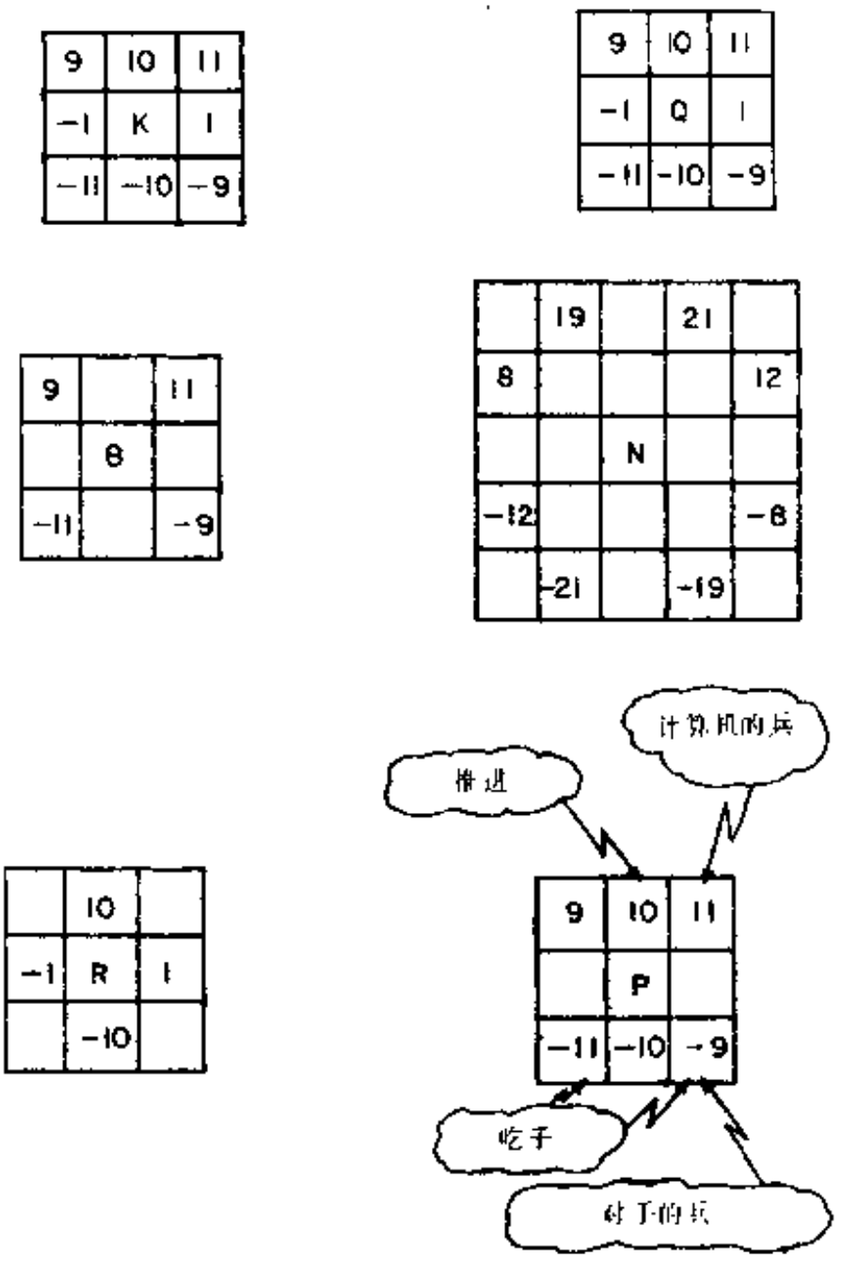


图 7-4　对每一个棋子来说，其移动方向是由一些数字给出的。这个数字同棋子所在方格的数字相加就等于把该棋子沿这一方向移动了一个方格。图中所给出的数字用于含有 120 个单元的 BD 数组。对于兵来说，允许它移动的方向取决于它属于那一方以及它是在前移还是在吃子

二、控制博弈树的规模

对于任何一种不是那么太简单的下棋游戏来说，一个完整的博弈树的节点数是异常庞大的，往往达到天文数字那样的数量级。因此，相应的下棋程序在可能的时间内，占用合理的存贮空间，只能探索整个博弈树的一小部分。如何选择要探索的这部分博弈树就是所谓的控制博弈树的规模。很自然，人们都希望自己所选择的那部分博弈树，能够包含那些具有最大取胜机会的竞赛路径。

控制博弈树的规模必须从二方入手：宽度和深度。

有一些程序—包括非常成功的 CHESS 4.6 程序—都没有控制宽度的任何措施。它们总是进行全宽度搜索，即在非终局的情况下，程序对每一个合乎棋规的棋步都进行探索。一个非终局节点的子节点数目是同相应的棋局所含有的合法棋步的数目相同的。在国际象棋中，这一数目大约是 30 左右。实行全宽度搜索的程序要探索规模甚大的博弈树，其节点数大约为 400000 到 500000 之间。进行这么大数量的搜索只有在大型、高速(亦是昂贵的)计算机上才能实现。而且，即便有了高速计算机，还必须严格限制竞赛路径所延伸的深度，因为需要检查到的竞赛路径实在太多了。

博弈树的宽度可以这样来控制，即对程序给定一系列的扇出参数：fanout-0，fanout-1，fanout-2 等等(通常大约 10 个扇出参数就足够了)。每一扇出参数对那一些同一级上需要进一步探索的棋局都限定了其合理棋步的最大数目。

因此，如果 fanout－0＝25，那么，博弈树根节点的棋局最多只能探索 25 个合法棋步(即根节点至多有 25 个子节点)。其余的合法棋步就作为没有希望的棋步而放弃掉。类似地，

如果 fanout－1＝20，那么在棋局的深度为 1 时，至多只能考虑 20 个合法棋步。换句话说，在深度为 1 的每一个节点至多只有 20 个子节点。

一般说来，扇出参数是随着深度的增加而减小的。fanout－0 的值的范围是 25 到 30。这就是说，在初始节点，除了那些明显看得出的、最没有希望的棋步外，其余的都要进行探索。在一个可能探索到的深度上，比方说 fanout－10，它的范围是 5～10 之间，甚至更小。在这样的深度上，只对那些看来是最有希望的棋步进行探索。

扇出参数的值通常经过实验来决定。这亦是程序调整的一部分。在对弈的对程中，程序还可以改变扇出参数的值。如果程序在时间上受到了限制，那么它就可以减小扇出参数的值。这样，搜索的范围变窄但速度要快得多。如果时间富裕的话，程序也可以增加扇出参数的值。这样，搜索的范围变宽，速度变慢，但却是比较全面的。

总之，我们对于每一个棋局，只去探索数目有限的棋步，而把其余的棋步舍弃掉。自然，人们愿意选择探索那些最有希望赢的棋步，而只是舍弃那些希望较小的棋步。问题在子程序如何来选择这些最有希望赢的棋步呢？

程序可以采用启发式的方法，对棋局的每一个合法棋步标上一个可行分，然后按可行分进行分类得到一张合法棋步的表。可行分最高的棋步置于表的开头，可行分最低的棋步位于表的末端。要探索的棋步从表头开始选取。如果当前棋局的扇出参数为 15，那么，置于表前而的 15 个最有希望的棋步将依次得到进一步的探索。

这一方法就是所谓的可行排序方法。可行排序方法还有另外一个优点：如果探索是按照分类表中棋步出现的先后次

序进行的话，那么首先探查的是可行分最高的棋步，接着是探索与之相邻的下一个可行分高的棋步，以此类推。这样就能提高上一章中所提出的α—β剪枝的效率。因为α—β剪枝方法最希望实现的是对当前棋局中最有希望的棋步首先进行探索。

可行分是根据对棋赛的启发式方法来给定的。每一启发式方法都阐述了一些条件，这些条件对实际的棋步来说可以成立，也可能不成立。同时它还说明了如果条件成立的话，棋步应得到或应受罚的分(如果条件正是所希望的，该棋步就得分；否则就该罚分)。根据所有的启发式规则所给出的分之和。就是可行分的值(取罚分为负值)。

当然，启发式方法的细节取决于具体棋赛的内容。下面是用于国际象棋的细则：

1. 吃子(capture)：如果一个棋步能够吃掉对方的棋子，就给该棋步打分。分的值取决于被吃棋子的值。

2. 攻击(attack)：移动后的棋子，如果它攻击对方棋子的数目增加的话，就给该棋步打分。

3. 安全(safety)：一个棋步把棋子放到了它会被吃掉的地方，该棋步就要被罚分；而棋步把棋子从它会被吃掉的位置放在一个安全的地方，该棋步就可以得分。

4. 出子(development)：把王后、车、象或马从它的初始位置移出参战，对这样的棋步要给分。

5. 推进(adevance)：对于把王后、象或马移动到对方一侧的棋步，要给分。相反，对于后撤这些棋子的棋步要罚分。

6. 用于特殊棋子的规则：在国际象棋对弈中，有许多用于特殊棋子的启发式规则。例如，当王后、象和马处于棋盘最左或最右一列时，一般认为它们的作用是比较小的。因而把

这些棋子移到棋盘的左边界或右边界的棋步会被罚分。对于王、兵和车这些棋子，也有一些附加的特别的规则。例如，对换位（王和车的棋步，它可以把王放在不受到袭击的地方）可以给一个很高的可行分。如果换位是可能的话，那末，其后的棋步一般总是要探索的。

在国际象棋中，用于确定可行序的启发式方法是很多的。但是，每一启发式方法的引用都要进行一定时间的计算，以确定选择那些合法棋步。如果引用启发式方法太多，那么花费在每一节点上的计算时间就会太长。这就使得在合理的时间内程序可以探索到的节点数要大大减少。

对于每一个启发式方法应给予或应罚多少分，一般都是在进行程序调整时，通过实验来决定。

深度控制可以通过一系列的深度剪枝来实现。在每一个深度上的剪枝都同一张特性表组合在一起，这些特性都可以在棋局上找到。如果在剪枝深度上的棋局中找不到组合表上的一个以上的特征时，那么该棋局就是终局。如果能找到组合表中的一个特征，棋赛的路径就要沿此继续推向深入。

例如可以存贮如下形式的一张特性表：

深度剪枝	特性表
depth—1	feature—list—1
depth—2	feature—list—2
⋮	⋮
depth—(n—1)	feature—list—(n—1)
depth—n	feature—list—n

如果一个节点的深度小于 depth—1，那么该节点就不是终节点。如果节点的深度大于或等于 depth—n，该节点就是

终节点。depth－n 以后的竞赛路径不需要去探索。

如果一个节点的深度是在 depth－1 到 depth－10 的范围内，那么就可以用特征表来确定该节点是否为终节点。例如，假定一个节点的深度大于或者等于 depth－1，而小于 depth－2，那末就要检查一下该棋局是否含有 feature－list－1 中列举的特性。如果有的话，该节点就不是终节点，否则，即为终节点。其余的剪枝方法及特性的使用都与此相同。

特性表的内容取决于对死局的理解。这一点在前一章中已经解释过了。特性表中的特性是指这样一类事情：它在一、二步棋以后，会使棋局的性质产生一些变化：如可能吃掉对方的棋子或者受到对方攻击等等。在深度比较浅的时候，我们可归纳出一张新内容比较多的性质表；在深度较深时，由于要对博弈树的规模进行控制，所以只能着重考虑那些看来会对棋局表示重大影响的特性。

一般常用的只是较浅的剪枝方法，比方说只到 2。深度剪枝的初始值也是在程序调整时通过实验来确定的。在对弈的过程中，程序还可以改变深度剪枝的值，改变的深浅程度取决于时间是否富裕。

三、评 价 函 数

在决定下一步走那一步棋时，程序于搜索树的 depth-1 位置上要用到节点的回溯值。回溯值是从评价函数对终节点所标注的值回溯而来的。因此，在做出下一步棋的决定时，程序所用到的全部信息归根结蒂都是由评价函数产生的。程序如果能下得出好棋，很显然，程序能够有效地区别出适当的终局位置是一件很重要的事情。

另一方面，每一个终节点都必须用到评价函数。这样

来，计算所花费的时间就成了一个不容忽视的问题。把评价函数定得复杂一些或许会提高有效性，但是计算所需要时间就会增加，这就意味着在下一步棋所允许的时间范围内，所能检查到的终节点的数量就要减少。为了在评价函数的有效性及计算所需要的时间之间得到一个令人满意的折衷选择，就需要对程序进行仔细地调整。

(一)兵力

在大多数棋类游戏中，每一方一开始就拥有一定数量的各种参战的棋子，这就是参战各方的兵力(material)。如果一方的棋子被对方吃掉了，那么该方就失去了一些兵力；如果可以把实力较小的棋子变成实力较大的棋子，那么这一方也可以增加兵力。在西洋跳棋中拥立新王，在国际象棋中兵变成了王后就是属于这一种情况。

兵力的优势是评价函数的一个组成部分。至于对于国际象棋来说，它是最重要的一个组成部分。兵力领先的一方在竞赛中也是领先的。如果一方在兵力上占有明显的优势（多一个或两个兵），那么就用不着麻烦去计算评价函数的其余部分了。

（人们在下棋时往往用牺牲兵力的代价来换取棋局形势的改善—在国际象棋中，例如有著名的弃子开局法。但是，对于程序来说，除非牺牲棋子的做法会导致迅速的赢棋，否则，程序应当是不会这样做的。因为它缺乏长期规划的能力来运用棋局布局上的优势。因此，程序的经验就是，在兵力上占优势的一方在棋局上也有优势。）

兵力的优势是根据不同棋子的不同的值来计算的。在国际象棋中，通常用到的值如下：

兵　　1

马　　3
象　　3
车　　5
王后　9
王　　一个大数，常大于200

（当然，王实际上是无价之宝，因为它是决不能被对方吃掉的。当王不能摆脱被吃的命运时，这一方比赛就输了。不过，为了进行运算，对王给定一个确定的值还是必要的。例如，在棋子受到攻击时，该攻击的重要性就可以从该棋子的值来衡量。此外，为了计算上的简单起见，人们有时也让程序去考虑一些不合理的棋局，如王被吃等等。总之，一旦给王一个确定的值，那么该值同其他棋子的比较，总是一个很大的数。）

国际象棋中兵力优势的计算方法，同前面章节中相应的西洋跳棋中的计算方法是相似的。假定p、n、b、r和g代表兵、马、象、车等的兵力优势（如果只研究合理棋局的话，那就不存在什么“王的优势”），那么：

p＝棋手的兵的数目－对手的兵的数目 n、b、r和g的计算方法也与此类似。这样，一个棋局总的兵力优势就由下式给出：

兵力优势＝9g＋5r＋3b＋3n＋p

按照这样的算法，一个棋局的优势很少超过1.5，即棋局的优势不会多于$1\frac{1}{2}$个兵。为了让棋局优势的各组成部分的值在一个合理的范围内（同时也为了使计算机处理起来更方便些，把这些数取为整数），我们可以用一些整数，比方说100，来乘以这些棋子的值。这样，我们就可以在兵的值＝100的数量级上计算兵力的优势，而棋局的布局优势，比方说，其

范围可以从－150 到 150。而后，我们就得到了一个完整的评价函数：

评价函数

＝(兵力优势)＋(布局优势)

＝(900g＋500r＋300b＋300n＋100p)＋(布局优势)

当然，"兵＝100"这一量度只是例子罢了。在实际运用的时候，它取决于棋局优势的计算方法，亦即取决于该优势值所变化的范围。

(二)布局

我们可以把评价函数中兵力部分看作为其战术部分，它是同每一棋步走后立即产生的得失情况相联系的。而棋局的布局部分可以看作为其战略部分，它所关心的是要得到兵力的非常有利的布局，以便在深一步的竞赛中占据优势。前面已经谈到过．因为目前程序还不能做出战略上所必需的规划来取得布局上的优势，因此，目前战术部分的重要性远远大于战略部分的重要性。

但是，对于双方在兵力上都不占据优势的终局，程序通常是一定要对该终局进行评价运算的。在这样的棋局中，除了根据兵力布局的情况进行评价运算之外，没有其它的什么办法来评价棋局的优势。实际的算法是这样的：对于每一方来说，如果在棋盘上找到自己一方所希望有的特征，就加分；如果找到不希望有的特征，就减分。然后，各方都能得到一个总分。布局的优势就是棋手的总分减去对手的总分。

一般经常用到的计算得分的特征如下：

1．攻击。对对手受到攻击的每一个棋子都给一个分。每种分的值的大小取决于受攻击之子的值。

2．王的保护。如果王被暴露，处于受到攻击的位置，就

要被罚分。

3. 出子。如果车、马、象和王后没有离开过它们的初始位置，就要被罚分。

4. 中心点的控制。控制住了棋盘中关键性的中心方格，就奖励分。控制中心点，可以是攻击这些中心方格，也可以是占据这些中心方格。

5. 机动性。对于棋子可以走到其上，而自己又不会被对方吃掉的方格，要奖励给分。

6. 兵的排列。对于公认是特别有效的兵的排列，要奖励给分。而对那些被公认是无效的兵的排列，要受罚扣分。

当然，这并不是一张完整的表。每一个编写国际象棋程序的人都有自己注重的特性。还没有谁可以声称自己的这张表一定就比他人的特征表更有效。

有些程序所采用的评价函数的计算方法，与上面提到的方法不同。其中有二种方法是这样的：

1. 查表法。不采取对每一特性打分，然后把这些分加起来得到的总分的办法，而是根据表中已有的特性组合，让程序找出与实际情况符合那一种组合，再从表中查出这种组合应该给的分。查表法的优点在于，表中所列的各项都能对各种特征之间的相互关系进行充分的考虑。例如，正好由于另一特性的存在使得这一特性变得有价值得多。或者相反，由于另一特性的存在使得这一特性变得没有什么价值。因此，如果不考虑其它特性是否存在的话，我们就不能对某一特性打分。亚瑟·撒梅尔（Arthur Samuels）在他的一个非常成功的西洋跳棋程序中使用了这种查表法。

2. 基于特定模式的评价法。与根据棋局的一般特性进行评价计算的做法不同，人们可以找出一些更详尽、更特殊的

模式。例如，对王的致命攻击，交叉将军等等有用的模式。就象运用一般特性那样，对有用的模式就加分；对无用的模式就减分。一种名为 USC 的国际象模程序使用的就是这个方法。这一方法的论述可以在《科学的美国人》(Scientific American)这一杂志的 1973 年 6 月号上找到。USC 程序的一个颇有意思的特点是，它创造了一种语言，使得没有任何计算机程序知识的国际象棋大师也能制定出自己的模式。

四、规　　划

象 CHESS6.4 这样一些成功的国际象棋比赛程序，它每下一步模，所要搜索的树包含了 400000 到 500000 个节点。而据心理学家说，一个下模的棋手，他每走一步棋，脑子里大概要考虑、审查 50 个棋局。很显然，在国际象棋比赛程序中，对于控制博弈树的规模还有很多工作要做。从根本上说，如果要想让程序在这方面同人类的功能接近的话，需要提出一些全新的思想才行。

其中有一个非常具有吸引力，但却基本上未经实验的想法是采用长远规划。在前面的章节中，我们已经看到，把搜索限制在一个计划的框框内，能大大减小博弈树的规模。此外，棋手对当前下棋程序最普遍的一个批评就是：程序明显地缺乏取胜的长远计划。确实，如果要利用错综复杂的棋局特征，不进行某种形式的规划是不行的。

要研究出一种完善的博弈规划技术，还需要做很多试验。在第五章中讨论过的分级式规划技术或许就是这方面的一个良好开端。在对弈进行的过程中，可以产生一个计划，也可以细化，修正甚至最后放弃这一计划。自然，程序在修正或者放弃该计划之前，总是尽可能地找出与当前计划相符合的最好

棋步。等级式规划对计划所做的修正总是在尽可能低的层次上进行。

第八章　模式识别与感知

当一台计算机从外界接收信息时，通常它所接收到的是一大堆未加处理的细节。对于一幅图画，计算机收到的是组成这幅图画的千万个点的亮度表；对于一种声音，计算机收到的是千万个数的数表，每个数给出了在某一瞬间的音波振幅；对于一局国际象棋，计算机得到的是表明每个棋盘格子中的棋子位置表。从这一大堆未加工的细节之中，计算机必须抽取为达到其目标所需的信息。对于一幅图画，计算机所感兴趣的并不在于每一点的亮度，而在于识别一些所熟悉的物体，例如门、窗户、桌子和椅子等。对于一种声音，它所感兴趣的可能就在于识别所说的话；而对于一局国际象棋，则它可能希望找到各种棋局特征，如将死棋势、交叉将棋势、杀将棋势以及一两步内将军的可能性等。

当识别一个棋式时，我们将把对一个物体的复杂而繁琐的描述代之以非常简单、概括、然而更加实用的描述。

例如，假设我们要识别图画中的一部电话机，那么我们就单用“电话”这个词来代替组成电话机图像的千万个点的亮度表就行了，并且用这一个词的描述远比原来用亮度的描述有用得多。首先，在我们的头脑里有许多包含在“电话”标题下的有用事实与行动计划，例如如何打一个长途电话。其次，我们有许多联想和行动规划涉及到电话。如果我们要与住在城镇另一边的朋友联系，我们立刻想到附近是否有电话，如果有，就给朋友打个电话，尽管在打电话之前，我们必须能够认出电话。

我们希望从电话机图象中抽取的特征正好与我们想达到的目的有关。如果我们只是想给某人打电话，只要“电话”这个词就足够了，尽管我们还必须区别这是公用电话还是私人电话。然而画家在画电话时，他感兴趣的是观众的视线方向以及电话机上的光线反射途径；而在另一方面，电话工程师则对于按照他所熟悉的电气性能把电话进行分类更感兴趣，如此等等。

人类进行模式识别是很容易的。熟悉的模式，例如电话机，我们从小就能认识而不用再加思索；其它模式，例如棋谱和乐谱，就需要训练才能理解。但是，一旦我们经过训练，模式似乎就会“跳到我们面前”，不用什么思索就能认识它们。事实上，我们很难设想如何做到视而不见。

计算机进行模式识别可就困难多了。现行程序识别模式不如我们所希望的那么好，并且经常是完全靠大量的计算来识别它们。模式识别是一个特别需要深入研究和用先进思想去填补的领域。

一、定　　义

一个模式就是一个对象的集体或类别。每一个对象是模式的一个例子，称为模式实例。确定对象属于不属于一个模式的法则，称为模式法则。

例如，集体 3，5，7，9，11，是一个模式，称为 P。（我们也可按照数学家的习惯，用大括号括进对象的集体或集合。）模式实例是 5 个数字 3，5，7，9，11；其模式法则如下：

一个对象属于模式 P，如果它是大于或等于 3 和小于或等于 11 的整数。

图 8-1 是另外一个例子，一个打印大写字母 A 的模式。

模式实例是如图示打印出来的各种不同的 A 字。删节号表示这一列实例表还没有完；仔细检查印刷刊物以后我们就会发现许多其它与已印出 A 字稍有不同的实例。所以很难最终给出这种模式一个法则。我们可能这么说："这个对象一定是大写字母 A。"但这实际上又使我们回到了实例表。例如图 8-1 给出的大写字母表或在印刷字体手册中能找到的更完善的表。我们可给出一个很长的复杂的规则，规定写一个 A 的笔画数和把它们组合起来的方法。这样一个规则会包含大多数 A。但或许还会有一些例外，例如草写和古体的英文字母 A。很难甚至不可能找到一个简单的规则，可以包罗所有的模式实例。

图 8-1　打印大写字母 A 的模式，每一个 A 是一种模式实例

在人工智能文献中，模式这一个词通常是指模式规则而不是实例的集合。在其它章节中我们将常常遵循这种说法。但从刚才的"大写字母 A"的例子中可以看出，为什么一个模式的最一般的定义是对象的一个集合，而不是一个规则。

下面是一些属于"模式识别"或"感知"范围的一些特殊的问题。

(一)模式分类

给出一个对象的详细描述以后，我们总希望将它分类成为一种或多种已知模式的例子。例如，当一个字母表通过光学扫描器的时候，我们希望所给出的输出能分成为 A，B，C 等等实例。模式分类可采用模式法则或与模式实例相比较的方

法。

(二)模式匹配

我们可能是根据已知模式去找合适的对象，而不是根据已知对象去找合适的模式。这种技术常常用来从一个事实的集合即所谓的数据库中提取有用的信息。

例如，假设我们的数据库中包含有说明有关现实世界事实的英文句子。我们可以写出如下的一个模式法则：

X AND Y ARE BROTHERS. X 和 Y 是兄弟。

X IS OLDER THAN Y. X 比 Y 年龄大。

Y MAKES MORE MONEY THAN X. Y 比 X 挣钱多。

字母 X 和 Y 是可以用词来替换的变量。按照模式法则，用词代替 X 和 Y 所得到的任何三个句子都是模式的一个实例。

我们希望找出属于数据库的模式实例，所有代表这些实例的句子在现实世界中大致是真的。例如，如果数据库包含下列句子：

BOB AND JIM ARE BROTHERS 鲍勃和吉姆是兄弟。

BOB IS OLDER THAN JIM. 鲍勃比吉姆年龄大。

JIM MAKES MORE MONEY THAN BOB. 吉姆比鲍勃挣钱多。

那么，这就是一个模式实例。因为这些句子可以在模式法则中以鲍勃代替 X，吉姆代替 Y 就可以得到。通过搜索整个数据库，我们能够找出所有可用来代替 X 和 Y 的人名对，按照模式法则替换以后的语句仍然成立。

模式匹配广泛地用于人工智能，我们还将进一步看到与自然语言理解，计算逻辑，产生式系统以及人工智能程序语言有关的许多实例。模式法则一般简称为模式。

(三)描述

我们可以用模式来描述一个对象。考虑一下你所在的房间。这个房间最详细的描述就是它的一幅图画，它可以用画面各点的一组亮度表来表示。不甚详细但又很有用的描述是用桌子、椅子、门、窗以及墙上的画等字眼来表示房间。给出已知房间一幅画，计算机就能搜索桌、椅等等模式实例，并根据这些模式构成一种新的描述，只有这种新的描述才适用机器人，比方说要求它去重新安排家具。

(四)特征抽取

通常一个对象的原始描述包含许多无关的细节，它们对直接搜索我们感兴趣的模式实例是没有用的。因而我们首先研究的是较简单的模式实例，并按照这些较简单的模式去构造对象的一个新的描述，再用这些新的但不甚详细的描述来分析我们感兴趣的模式实例。

这些较简单的模式称为特征。寻找模式特征实例的过程称之为特征抽取，有时也称为预处理或预分析。

特征抽取的实例是很多的。例如在图形分析中，在开始辨认对象本身之前，我们必须定出勾划对象的轮廓线。在分析一局国际象棋时，在开始计算评价函数之前，我们首先要系统地搜索棋局，以发现所有的将死、交叉将、杀将与将军等等。

特征抽取的结果常常是一组数字，例如给出字母表中一个要识别的字母以后，计算机首先要确定如下事项：

- 组成字母的直线笔划数。
- 线终止的端点数（例如E有三个端点，而C有两个端点）。
- 角数，例如字母A的顶部就是一个角。
- 字母的宽度。

等等。

每一个数字将对象划分为几个可能模式中的一类。例如直线笔划数的值将大写字母分类如下：

直线笔划数	模式
0	(S, C, O)
1	(I, J, G, Q)
2	(T, X)
3	(F, H, Z, A)
4	(E, M, W)

(我们不考虑字母端点以外的小笔划，例如I作为单一的一条垂直线来处理。)

这就启示我们可以扩充特征的定义，直到包含诸如"直线笔划数"这样的性质，这种扩充特征的值又把许多可能的模式分类归入某一个数字。允许一个特征对应许多模式就简化了描述。在下一节中我们将看到，可对扩充特征的值有效地进行某些数学和统计运算。

我们可以把复杂的描述，例如一幅图画，认为是用一个等级式模式识别器来进行工作的对象。在等级的最低层是处理原始数据的识别器，并抽取十分简单的特征。这些特征形成对对象的描述，然后又被上一层的识别器加以处理，这一层抽取的特征又将在比它高的一层上处理，如此类推。等级的层数根据需要而定，以便使顶层最终能识别出所感兴趣的模式来。

(五)学习

我们希望教会一部计算机程序学会一定的模式。为此，我们首先给计算机提供一些对象，一些模式的实例，还有一些不是模式的实例，并告诉计算机哪一个是，哪一个不是。这些

对象构成训练集，然后我们从试验集中取出一些对象给计算机，并且希望它告诉我们哪一个对象是所讨论的模式实例。计算机进行成功识别的多少就反映它对模式识别学习的好坏。

例如，在能学习区别印刷字符的程序上我们做了许多工作，这样一个程序在适当的训练集的帮助下，可以自己适应不同的字样。由于有几百种不同的字样，所以如果一个字符识别程序要能广泛应用的话，看来必须具有这种适应能力。另一个例子是：Arthur Samuels 的西洋跳棋程序，它用修改其评价函数的方法来学习下跳棋。所组成博弈的训练集是由棋谱中选取的。程序走的棋步要与棋手推荐的棋步相比较，程序还要调整其评价函数，试图使它走出的棋步尽可能与棋手的棋步一致。通过大量的棋谱进行博弈训练之后，程序就变成了一个出色的西洋跳棋能手。

在这些例子中，程序并不学习什么是相关的特征，也不学习如何由原始数据抽取这些特征的方法，相反地它仅仅学习根据程序人员设计的特征所建立的模式法则。例如 Samuels 跳棋程序并不学习相关实力优势，机动性，中心控制等等特性。这些特征以及从跳棋棋局中抽取特征的子程序都是由 Samuels 自己设计的。程序能学习的只是从所提供的已知形式特征中如何去构造一个评价函数。试图从原始数据中抽取相关特征的学习程序并不太成功。

(六)模式的表达

另一个问题是如何进行模式的表达和描述，以便能进行人机联系。这就是说，我们能否设计出一种方便的语言来说明模式法则？一种通用的模式语言尚未设计出来，但有几种专用的语言值得一提：

1. 判定表．条件项相应于已经从原始数据中抽取的特征。每个判定表法则就是一个模式法则。判定表通常用来规定(通过动作项)相应于被识别的模式所要采取的动作。

2. 包含变量的表达式．这在模式匹配的讨论中已经叙述过．那里所讨论的表达式是英文句子。这些表达式广泛地应用于人工智能语言以及 SNOBOL 字符串处理语言之中。在本书的其余部分，我们还会几次提到它们。

3. 面向主题的语言．这种语言面向专门主题所用的术语，以便不懂编制程序的某一门专家能把模式写成程序(以及把它对模式作出反应时应采取的动作或启发式方法写成程序)。例如，USC 国际象棋程序的作者设计了一种语言，以使一个象棋手用来描述重要的象棋模式，并建议计算机如何去评价包含有模式实例的棋局。

如果人工智能程序要在专门领域之中，例如国际象棋，化学或医学中起作用，它们必须能够从这些领域的专家中获取知识，但这些专家却对计算机程序设计一无所知。所以我们可以预料对描述模式，启发术和计划来说易于使用的语言将变得越来越重要。

二、特征空间、区域和原型

这一节所叙述的方法绝不是模式识别问题的最后解决。但它们已经成功地应用于这样一些实际问题，例如印刷字符，血球，染色体以及天气预报的分类。它们还用于市场预测(但无明显效果)。

(一)特征空间

假设已经对一个对象的原始数据进行特征抽取。特征抽取过程已经对该对象产生了一个描述，例如一组数字，这已在

上面特征抽取一节中叙述过了。我们可以把每一组数字看作是特征空间中的一个点。

例如，医生通常从他的病人身上抽取两个特征，身高与体重。图 8-2 表示了这个相应的特征空间，身高与体重为直角坐标系的两个轴。每一个病人用一个点来表示，他的坐标就是他的身高与体重。

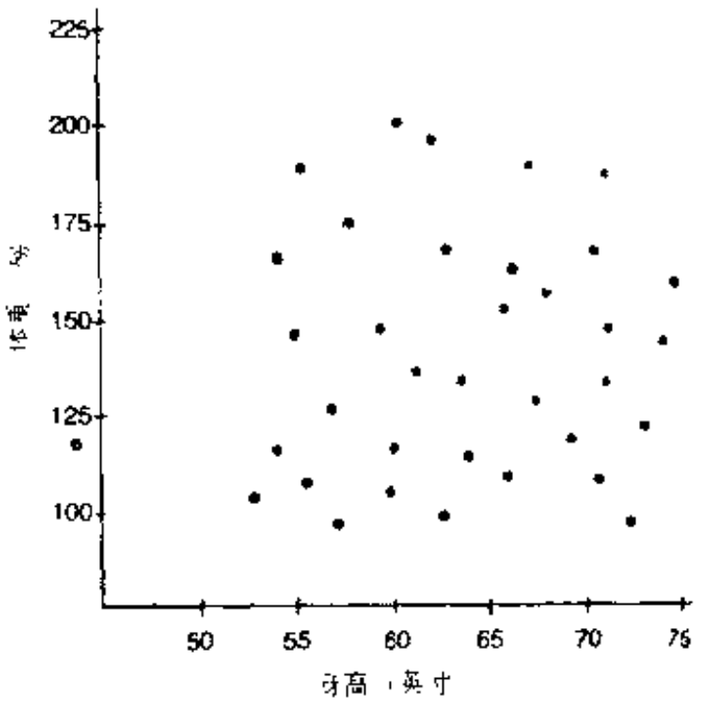


图 8-2　以身高与体重为特征的特征空间，已经测过身高与体重的人用一个点来代表

特征抽取过程可以根据若干以至成百的特征来进行对象分类。当多于两个特征时，我们实际上不能在图纸上画出点来。但我们可以把这组特征值看作是抽象的多维空间中的一个点。虽然我们很不容易想象这样的空间，但是数学家已经提出一些方法，使得在多维空间的运算就象我们在二维直角坐标系中的运算一样容易。

(二)区域

如果所抽取的特征适用于要识别的对象，那我们就能够把特征空间分为若干区域，每个区域对应于特定的模式。只有当相应于对象的点位于相应模式的区域内，对象才是模式的一个实例。

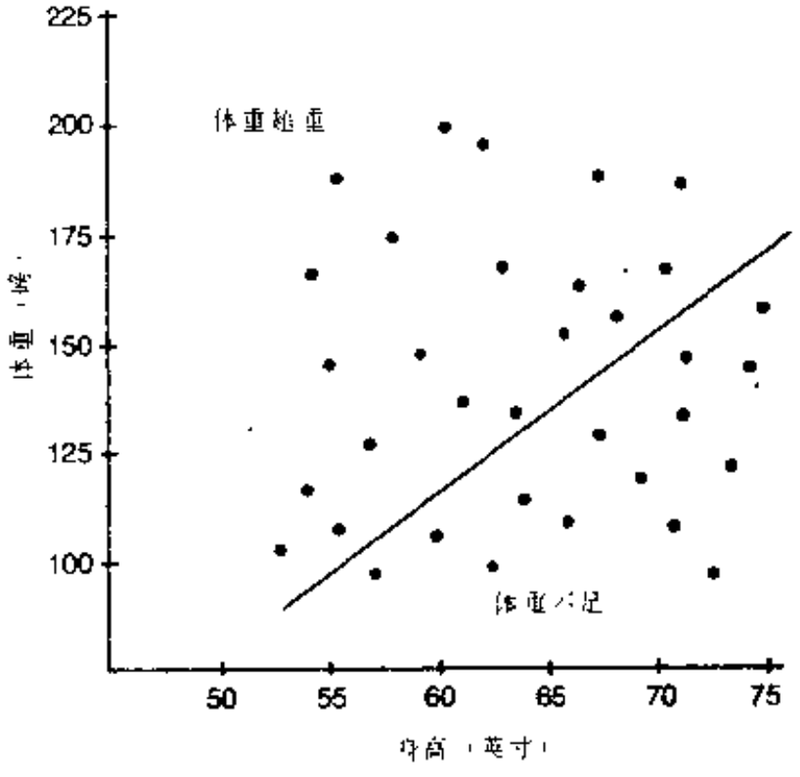


图 8-3　正常体重线将身高--体重特征空间分为体重不足与体重超重两个区域

例如，我们可以画一条身高与体重正常关系线，将身高—体重特征空间分为两个区域（图 8-3）。点在这条线以上的人体重超重，而点在这条线以下的人体重不足。点恰好在这条线的人既不是体重超重，也不是体重不足，这正是所希望的情况，但也往往造成困难，若一个点正好位于分界线上，我们就说不清它到底属于哪一个区域，因此也说不清它是哪一个模式的实例。

只要可能的话，我们希望用直线来划分区域，因为直线在数学上是最容易处理的。可用直线来划分的区域称之为线性可分。某些区域不能用直线分开，如图 8 4 所示。对这样的区域我们必须用较复杂的曲线，例如椭圆，或用一组线段来代替一条直线。

（在多维空间里，与一条直线类似的情况称之为超平面。因此我们一般按照超平面将特征空间划分为区域。）

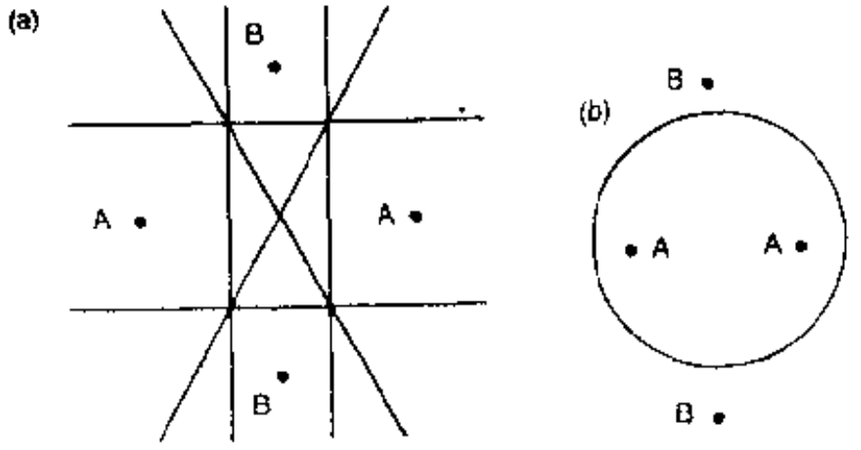


图 8 4，某些区域不是线性可分的。如在(a)中所示，找不到一条直线能将 A 型点与 B 型点分开。但用一个圆即可分开，如(b)中所示

为了找到将特征空间划分为区域的曲线或曲面，发展了许多强有力的数学技术。这些技术的成败与否取决于特征本身，如果这些特征确实适合于区别我们所要识别的模式，那么这些技术才会成功，否则就会失败。

（三）原型

代替将特征空间划分为区域的另一种方法是：画出相应于每个模式的一个典型实例—或原型—的那些点。给出相应于未知对象的一个点以后，就要找出该点与每个原型点的距离，然后将未知对象归类于最靠近原型的模式。两点间的距

离是由包含它们坐标的某一公式给出的。它不一定是在几何中常用的距离公式。定义一个好的距离（两个对象之间差异程度的度量）是这种方法的一个中心任务。

（四）样板

利用原型的另一个方法是作一个样板。假定我们要识别印刷字符。我们把每一个字符的实例印在一片透明的塑料上。这些原型字符就是样板。为了分类一个未知字符，我们把每一个样板依次放在它的上面，看看哪一个原型与未知字符匹配得最密切。

计算机可以把未知字符与样板（用点的亮度表示）存于不同的存贮区域，以便作类似的比较。为了把未知字符与样板相比较，计算机逐个按地址来比较两个存贮区域的内容。

用样板方法的困难在于被识别字符的细微变化将导致匹配的失败。如果字符稍微大一点或小一点，在一个方向或另一个方向倾斜，或者中心偏离，就不可能与正确字体的样板匹配。事实上，它可能与一个不正确的样板匹配得比正确的样板还要好。所以在一个字符与样板匹配之前，必须经过预处理来给它以标准的位置、大小与方向。

（五）小结

本节中叙述的方法已成功地得到应用，例如光学字符识别和血球识别。另一方面，它们也有本身的局限性，因此肯定不能成为模式识别问题的最终解决方法。

特征空间方法的基本要求是一组易于抽取的特征的值要确实能区别我们所要识别的模式，以便相应于不同模式实例的诸点能处于特征空间的不同部分。对于样板方法，我们必须能够消除被识别对象的细微变化，每一个对象都必须纳入一个标准形式以便与样板比较。

这些方法中没有哪一种用于这样一些对象，它们的识别更多地取决于这些对象所处的前后关系，而不是取决于对象本身的细节。例如在一幅画中，一条线的意义更多地依赖于这条线和另外的线之间的关系，而不是线本身。对于一个人说话时发出的特定的声音也可以这么讲。

最后，对某些对象进行识别时，更多地是用它们的功能，而不是它们的细节。例如瓶子就有各种尺寸和形状，我们识别一个瓶子，并不过多地根据它们的尺寸和形状，而是根据它们的功能，即用它来盛液体（或者如果它是空的，就可以用来盛液体）。

三、图象分析

分析由电视摄像机得到的图象是模式识别的主要任务。目前这一任务尚未很好完成，工作尚待进行。为了说明目前有关这个题目的想法，本节叙述目前某些实验系统所采取的图象分析步骤。

（一）图象获取

来自电视摄像机的图象在选定点进行采样，这些点的亮度转换成数字并存于计算机的存贮器中。所以对计算机而言，图象只不过是数字的阵而已。

（二）平滑

由于“噪声”—获取图象过程中的误差，使得存于计算机中的某些数字并不对应于原来景象中的点的亮度。“平滑”就涉及到用来消除和减少这些误差影响的数学运算。这些运算可应用于存贮在计算机内的图画—也就是数字的表。

例如，我们可以搜索在图中比其周围邻近的 8 点都亮的任一点。由于这个点看来可能是个误差，所以我们把它的亮

度变成相邻点中最亮点的亮度。或者换一个方法，以相邻点的平均亮度来代替其亮度。比任何相邻点的亮度都暗的点也用类似的方法处理。

还有许多其它可能的平滑技术，应用它们必须小心，以免有用的图象与噪声一起平滑掉了。这些技术已被用来增强从宇宙飞船发回地面的图象。

(三)原始草图

当我们画一景物的草图时，草图中的线相应于原始景物中不同亮度区域的分割线。分析一个景物的第一步是把它转换成草图，找出不同亮度区域的分割线。

假设我们以直线路径穿过图象，如图 8-5 所示。如果记下我们所遇到的点的亮度，就可以得到如下的数据：

1 1 2 3 4 5 6 5 4 3 2 1 1

这表示当我们通过景物时，亮度开始增强而后再次下降。当用这种方法通过景象时，我们如何来决定什么时候通过线——什么时候从一个亮度水平移到另一个亮度水平呢？

我们可以对每个数用一个操作符来进行运算，以得到表示亮度水平发生变化点的新表。假设原始表为：

1 1 1 2 3 4 4 4 4 3 2 1 1 1

我们应用如下定义的操作符

$$B_{new} - B_{old} - B_{left}$$

(新的亮度－旧的亮度－左边的亮度)

这指出每个新点值是这样计算的，将相应的旧点值减去紧挨着它左边的旧点值。下面是新旧数值表：

旧值： 1 1 1 2 3 4 4 4 4 3 2 1 1 1

新值： 0 0 1 1 1 0 0 0 −1 −1 −1 0 0

我们看到只有当亮度发生变化时，新的表列值才不为零。新

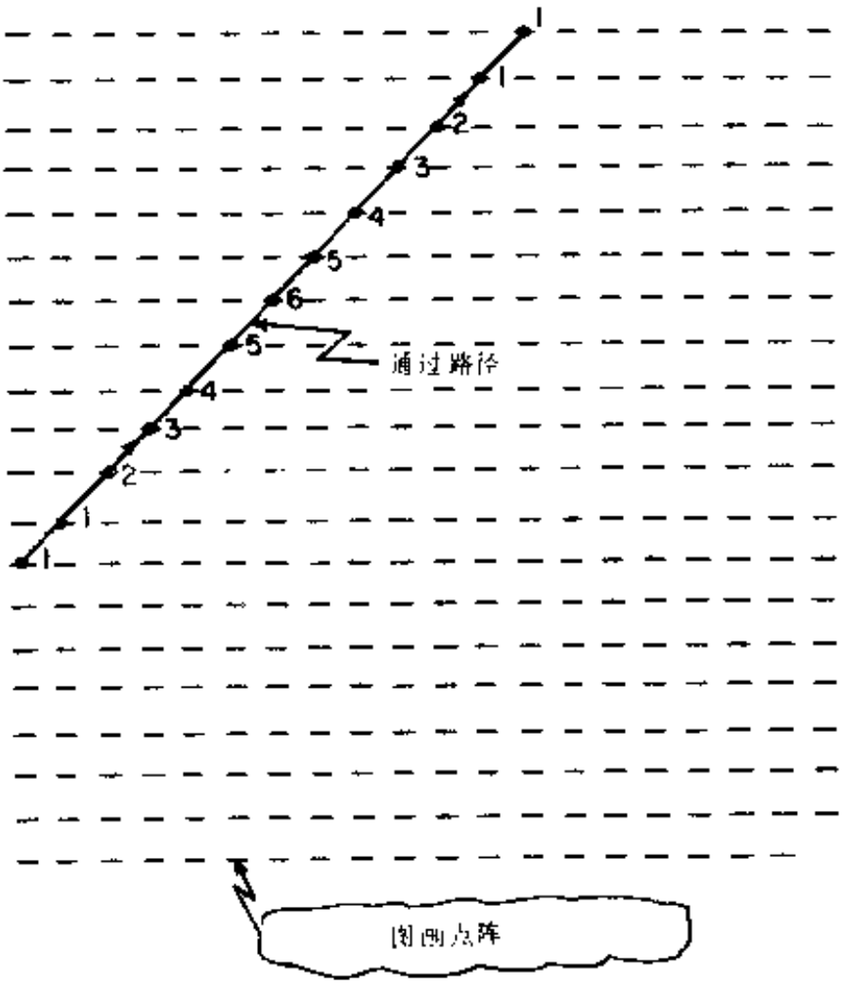


图 8-5　我们能够以直线路径通过图画点阵并记下所遇到点的亮度

表中的非零值即相应于不同亮度区域的分割线。

（读者如果学过微积分，就会知道我们正是沿着所讨论的线对亮度求一次微商。二次微商在研究亮度变化时也是很有用的。）

现在假定我们在原始图象上编排一个点阵，并通过每个点取不同方向的线，如图 8-6 所示。如果我们沿着每条线应用不同的运算符云分析亮度的变化，我们就能定出不同亮度区域的分割线，给出位置、长度和方向等性质的分割线的表称之为原始草图。

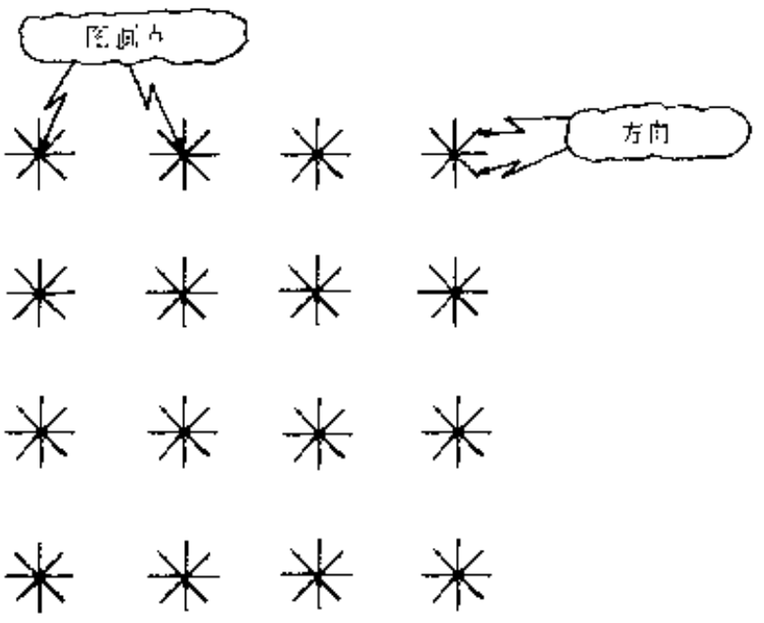


图 8-6　沿着通过每个图画点不同方向的线，我们可以分析亮度的变化

原始草图是由很多短线段组成，与画家用以素描景物的铅笔笔划一样。这些短线段必须连在一起以形成表示物体或物体某些部分的边界。

做这些工作的方法是各种各样的，而且没有发现适用于所有环境的方法。计算机要做的是找出一系列短线段，并连结在一起，形成一条直线或一条平滑的曲线。有一个寻找代表一条直线的线段的简便方法。程序把满足下列两个条件的线段收集成一组：

1)一组中的每条线段是在同一组中另一线段的几个画点之内。

2)一组中没有线段与同一组任何其它线段垂直。

每一组表示一条直线，这条直线连结这组中相距最远的两条线段。

用这种或某些其它规则填上了边之后，计算机就能努力去填充间隔，通过看这个边与另外的边所形成的棱角来判断

这些边是开始还是结束。如果边突然中止而未与其它任何边相连，则程序就推测有一个间隔，并寻找在逻辑上能与它相连的另一个边。

人们也可集中注意于区域本身，而不是集中注意分割区域的边。程序试图将景物分成一些亮度均匀的小块面积。这些面积可看成是分割的区域，并且可认为它代表原来景物的不同表面。人们画出了划分区域的边界线，也就得到了该景物的草图。

或许一个高级的景物分析程序必须对线和区域二者进行处理，即程序将努力把短线段连结在一起以形成区域，并把区域中亮度相同的小块面积识别出来。两种方法可以互相补充。区域分析能帮助分割区域的边缘，填补间隔；边缘分析能帮助区域之间建立边界线。

纹理分析也能应用于识别区域。一个纹理表面是一个在其上有某种模式的表面，如一块木头或一块布等。在原始草图中纹理用许多短线段来描绘，正象画家所画的那样。对线段进行统计分析可识别出不同的纹理。程序收集具有不同亮度，不同长度，不同方位的线段的数量统计资料。统计量相同的面积可以看成有相同的纹理。它们可能是相同区域的一部分或者至少是密切相关的区域。统计量根本不同的面积有不同的纹理，几乎可以肯定是属于不同的区域。

我们可把区域看成是代表原始景物的表面—比如一个房间的墙壁或桌子面。识别了这些表面，剩下的问题是确定每个表面相对于观察者的方位。我们是垂直往下看一个特定的表面呢，还是以一定的角度来看呢？

研究这个问题的一个方法是分析该表面的照明度。如果我们知道光线落在景物上的方向，那么物理定律告诉我们，表

面的视在亮度将如何随表面与光源和观察者的方位而变化。一个景物分析程序能够努力找出照明的方向和一组与所观察亮度一致的表面方位。这种分析的数学计算是极其困难的，特别当所讨论的表面是曲面的时候。

有了照明，有关景物的许多信息就可以从阴影分析推导出来。物体及其阴影分析能够得到比分析物体本身更多的信息。

假设(在目前技术状态下是一种大胆的假设)相应于不同表面的区域已被辨识出来，分割它们的边也已经建立。剩下的问题是把区域连在一起以形成对象，并去辨识所形成的对象。特别是程序希望把区域和边的表转换成有意义对象的表，例如桌子、椅子、电话等等。

如果景物仅仅是由积木、盒子、棱锥体、楔形体和由直线围成的平面所组成的其它物体组成，那么物体的辨识问题就能直接而简单地予以解决。

想法是这样的。在积木世界中的可见边界分为三类：

1. 一条凸边分割两个可见表面。当从前面稍稍偏上观察一积木时，该积木顶面与前面的分割线就是这种例子。

2. 一条凸边分割两个表面，仅有一个表面可见。当由前上方观察一个积木时，它的顶面与背面的分割线就是这种例子，这时背面是隐藏的。

3. 一个凹边。当从盒子内部观察盒子时，盒子两个面的分割线就是这种例子。

现在这三种边只能用有限的方式连结在一起形成棱角。按照形成棱角的边缘种类，可将可能的棱角分类。

例如，假设一确定的棱角是由三个凸边相交于一点而得到，这必定是一个积木的一个外棱角，它不会是内部看到的盒

子的一个棱角，因为内棱角的边缘必须是凹边。假设每条凹边是分割两个可见表面的的那种边，那么这必定是像盒子的顶棱角那样的一个角，只有这样组成棱角的所有表面才是可见的。

程序通过寻找景物中边和棱角的一致标记来应用这种思想。每一条边被标记成上述三种情况中的一种，每个棱角被标记成容许种类的一种。因为每一种容许棱角仅能由一确定种类的容许边缘所构成，所以容许边缘连结在一起才能形成容许的棱角的要求约束了标记范围。程序检查不同的标记直到找到一个与所有边缘和棱角相一致的标记为止（如果找不到相一致的标记，那么图画就是一种“不可能的对象”。有时我们能画出自然界中不存在的图画，这种不可能的东西常常是荷兰画家 M. C. Escher 作品中的特色）。

一旦所有的边缘和棱角分好了类，那就比较容易地把它们组合成已知的物体。应用这种技术可编出程序来分析十分复杂的景物，例如由盒子、积木、棱锥体、楔形体以及其它平面物体所组成的景物。

遗憾的是，标记技术仅适用于“积木世界”。它对曲面无效。现实世界中的对象可以整个或部分地包含有曲面，而曲面并不提供明显的边界与棱角给标记程序来处理。

在现实世界中辨认对象问题仅仅向前走了第一步。其中一些步骤将会被提到，但还不清楚这些步骤是否方向正确。

一个复杂的对象可以用一个简单的几何形状来近似。水杯可用一个圆柱体来近似，一部电话可用一个楔形体来近似，一个金鱼缸可用球形来近似。这就与艺术书为初学画的人推荐的方法恰恰相反。初学画的人被教导说，画一幅画首先画出其简单的几何图形，如方块、球形、圆柱和楔形。当景物勾

画出来后，画家再回过头来补充这个对象的细节，原来的轮廓线也就被擦掉。

圆柱体对这种分析方法特别有用，如果我们允许圆柱体的直径从顶部到底部是可以变化的，在这样推一步之后就更有用，例如可口可乐瓶可看成是一个圆柱体，其直径在顶部非常小，往下则渐渐变大，又再次变小，最后在底部变得大起来。

由于广泛应用车床和其它旋转机床来进行加工，所以很多人造的物体具有这种对称的圆柱形状。

用于一般化圆柱形的程序已设计出用以识别考古学家感兴趣的各种希腊花瓶。

对象发生的背景现在用得还不多，但必将会进一步广泛应用于物体辨识程序上。例如一个放在桌上外形粗糙的黑色楔形物就很象一部电话，这通常是我们推测性的辨识，即便是细部还很模糊。然而，如果同一个物体挂在天花板上，那么在我们进行识别之前，一定会走近前去仔细地看看，要是还认不清楚，那就根本识别不出来了。令我们为难的模糊照片之所以难以辨认，就是因为照片中所描述的事物没有背景。利用我们的背景知识和我们的预测能力与在十二和十三章中要讨论的框架概念，知识表达和自然语言处理有关。

不论最后发展用什么方法来进行景物分析，都需要极大量的计算，这是很清楚的。大概将需要有专用微处理机，它能在一幅画面中的每一个点上同时进行相同的运算。这可用来构成“智能”电视摄象机，它提供给计算机的将不是景物的原始数据，而是表面、线、或许甚至是物体本身的表。有理由认为人的光导神经在把图象传给大脑之前就进行了大量的预处理。

第九章　机　器　人

机器人，就是在本身的能源和自己控制之下工作的一种机器。因为现在本身具有能源的机器已属司空见惯，所以机器人的区别主要在于它还在自己的控制之下。机器被指派任务以后，能够执行任务的各个步骤，不再需要人的干预。

机器人就是满足上述定义的机器，它的复杂程度是千差万别的。在机器人的谱系里，一端可以称之为自动机器。自动机器可以不需人的参与实现复杂的一序列动作。不过实际上机器每一次运转，这一序列动作都是相同的。机器几乎没有什么能力根据环境条件修改自己的动作。

在机器人谱系的另一端是在人工智能实验室里制造出来的一些机器。这种机器人用电视摄象机或其他手段来感知环境。它们能够接收和执行高级命令，来处理环境，并且自己会填补较低级的细节。

比方说，告诉机器人把绿色积木放进红色盒子，机器人能首先确定绿色积木和红色盒子在房间里的位置，然后自己能移动到绿色积木附近去，并且搬开或者绕过它路上遇到的障碍。在拿起积木放到盒子里面去之前，它也许还要移去搁在绿色积木上的某些东西。最后，它也许还要搬走已经放在红色盒子里边的某些东西，以便腾空来放绿色积木。

我们熟悉的自动机器的例子首先是自动家用器具如自动洗衣机，自动换片的唱机等。这些机器能够用简单的方式对环境作出反应。例如，自动换片唱机检测到唱片放完时，就会启动换新片的动作。在唱片放完之前它会拒绝新片。当它检

测到所有唱片都已经放完时，会自动把电源关掉。不过，虽然机器能够作出这类简单的反应，但唱机仍只能放唱片，洗衣机仍只能洗衣服。我们不能命令它去做别的机器的工作。

这类自动机有过一段很有趣的历史，特别是历史上曾有过一些机器，比起家用器具来更接近于现在多数人心目中的机器人的样子。

十八世纪，欧洲曾经风行钟表自动机，这种自动机有着人形或动物形，能模仿为人们所熟悉的这些活物的动作。有的机器人演奏大键琴；有的机器人能写一封可读的信，还不时将一支羽笔浸到墨水瓶里去蘸墨水；机械鸭子能吃、饮、排粪、鸣叫，并在水中拍打翅膀。

这种钟表机械人通常用带销钉的鼓来控制，颇类似于那些在音乐匣子中的音柱。当鼓慢慢转动时，它表面上的销钉就会触动不同的机构去使人形动作。这种带销钉的鼓其实就是一种“只读存储器”，里面存储着人形（或动物形）将要实现的动作。如要改变它们的动作，就得另换一个新鼓，上面销钉的位置和原来的不同。但通常它们没有这种备用程序；每个机器人造好以后，能做一种一定的序列的动作。这些机器是为了娱乐而制造的，虽则它们总是重复一系列固定的动作，但观众总是感到新鲜的。

这些早期机器人的后代，今天仍可以在各地的游乐场中看到。例如在狄斯奈游乐场中，在有声活动电子玩具展览中，还有林肯和其他美国历史人物的形象为观众表演，不过技术上已有了很大改进：用电气或电子装置代替了钟表机构，但其设计思想和十八世纪的自动机器并没有什么两样。

也许是钟表自动机第一次引起了一场关于机器能不能思维的讨论。对于生命的模仿在人们中间引起了扰动。有人哪

啧着说这是一种巫术。有一个自动机器设计师甚至因此被捕，并受到宗教法庭审讯！

正是在这样的历史背景下，出现了有名的坎姆拍仑(Baron Von Kempelen)的下棋机器人的骗局。他把人藏在机器里边，假装成能下棋的钟表自动机器。这个假的下棋机器人模仿别的自动机器的模样，坐在棋盘前面，用手带些可笑的机械动作来移动棋子。

现在言归正传。谈到现代工业机器人，我们可以从家用器具和钟表自动机器迈出一步，用编制的程序使机器人实现不同序列的动作。不过，一旦程序编好了，这种机器和上面谈到过的机器一样，也是一遍又一遍地重复同一序列的动作。

这种"选择和放置"操作器常常是用插接板来控制，这和许多在有计算机以前的数据处理机十分相似。插接板是按行和列来安排好的。每一行代表机器人的手和臂的某个动作。每一列代表机器人程序的某一步骤。在行与列的交叉点上的插销可以使机器人做出程序中某一特定步骤中的某一特定动作。每一动作的大小程度—手伸多远，抓多紧—都决定于机械步骤，就象打字机的边缘键和制表键那样。

插接板上的插销对应于钟表自动机的鼓上的销钉。但是十八世纪的设计师大概从未使用过可以调整的启停装置；他们毕竟不是以灵活性当作设计的目标的。当启停装置一旦装好，插销放好，机器人的程序也就编好了，可以按照需要随便多少次地执行已经编好程序的动作了。

现在使用的工业机器人，在复杂程度上又迈进了两大步。

第一步，电位器盘代替了插接板，电位器代替了机械启停器。每个电位器的设定点决定了机器人某一部分应当移动的特定位置。

不用机械启停器时，机器操纵器就必须包含反馈敏感元件，使得机器人某一部分的实际位置可以和盘上电位器设定位置相比较。电位器也常常用来做反馈敏感元件。在程序的每一步骤，机器人的操作部分总是被调整到使它上面安装的电位器的设定点，和电位器盘上相应的某列电位器的设定点相同。

第二步，插接板和电位器盘都通通取消了。机器人的行动受一个便携式控制盒的"教员"的引导。机器人把引导它行动时所要求的位置都"记忆"到它的内部存储器中去。于是它也能重复它被教过的那些动作。现有的模型中，常用镀线存储器来储存所要求的运动的细节。将来这种存储器无疑地将被更新式的存储器，如磁泡存储器所代替。

这一类机器从一个记忆位置运动到另一个位置时，它的动作往往显得有些可笑（使用插接板和电位器的机器人也是这样）。对于多数用途来说这没有什么要紧。但如果运动的平滑性很重要，例如用机器人喷漆时，那就得把大量的中间位置也存储起来（通常是存储在磁带上面）。这样机器人就可以用非常微小的步子从一个位置运动到另一个位置，模拟着连续性的动作。

到目前为止，还没有一个机器人用到计算机；插接板、电位器盘和镀线存储器控制就已经够了。用计算机来控制机器人，目前价钱仍然太贵了；只有研究实验室的机器人才用计算机来控制。但微处理机的出现使一切有了改变，我们可以预期，用组装式微处理机来控制的机器人今后会越来越多。那么，这些计算机控制的机器人会增添哪些能力呢？

计算机能做的一件最重要的工作，就是能把机器人的内部坐标和外部坐标进行转换。下面让我们来解释一下。

譬如说，当机器人用电位器盘来编程序的时候，电位器的设定并不直接地规定机器人操作器在空间的位置。比方说，它并不直接规定机器人的手应当移到某一个位置：离桌子表面 6 英寸高，离桌子的一边 20 英吋，离另一边 30 英吋等等。电位器的设定只不过规定机器人手臂各关节的位置。它还要靠编程人员来保证按照程序给定的位置，它能够运动到所要求的相对于桌子的空间位置。

机器人的关节和其他部分的位置即所谓内部坐标。而机器人的手（比方说），相对于房间内其他物体的位置用数来表示时，则叫做外部坐标或工作空间坐标，我们真正感兴趣的是机器人怎样相对于环境中其他物体而运动，即它的工作空间坐标；对于机器人的每个关节转动多少度我们并不感兴趣。

计算机的一个重要作用是能够在外部坐标和内部坐标之间进行转换。它能够把一个命令，比方说，将机器人的手向上抬高 6 英吋，向左移动 4 英吋，转换成达到这一目标所要求的关节运动。

用计算机来控制，我们就能够获得连续运动，而不必在记忆中存储那么多中间位置。假若我们要使机器人的手在给定的两个工作空间位置之间平滑地运动，我们只要存储起点和终点位置就够了。计算机运用初等解析几何，就能够计算出两点之间连接的一条直线上许多中间位置的工作空间坐标。它把这些中间位置从工作空间转换成内部坐标之后，就可以向控制机器人手臂关节的电动机发出控制信号了。

计算机还能进行许多其他有用的运算。比方说，要把重物从一个地方移到另一个地方，机器人必须运用比较大的力来使物体移动，并用比较小的力来克服摩擦力使物体保持恒速运动，然后用相反方向的力使物体停止在要求的位置（试比

较一下在拥挤的道路上驾驶汽车操纵加速器和制动器的情况)。计算机能够计算出,在运动轨迹的每一点上对物体要使用什么力和多大的力,才能尽可能快地把它从一处搬到另一处。

计算机控制敞开了一扇大门,使我们能够造出过去不曾造出过的复杂得多的机器人。这些机器人(包括现在仅在实验室里出现的机器人在内)能够用复杂的方式和环境打交道,并且能够执行相当高级的命令。

制造复杂的机器人所要考虑的主要问题包括在三个领域内:

1. 效应器和操作器。就是机器人处理(外部)现实世界的机械部件。

2. 感觉器,把现实世界的信息传送给机器人的装置。机器人越是复杂,感觉器起到的作用就越重要。

3. 控制。所指的主要是计算机程序,它使机器人能够在感觉器获得信息的基础上接受命令和执行命令,使效应器以适当的方式运动。

一、效 应 器

效应器包括机器人的手臂,以及使机器人从一处运动到另一处的手段,机器人的某些感觉器也是需要运动的。譬如机器人使用电视摄象机时,它应当能够把摄象机指向任何方向,而毋需转动整个身体来面对它所要看的方向。它还应当能够向前看,也能够向上看和向下看。

工业机器人通常是沿着生产线固定安装在站台上。研究用的机器人则通常用自身的能源在房间里面运动,而且总是用轮子来移动的。轮子是用电机来驱动的。对于轻便的机器

人，用步进马达是很合适的；它每一步把轮子转一已知角度。有些科学幻想电影中的机器人用履带而不用轮子（象推土机那样），但看来履带并不比轮子好。将来除非真正需要机器人上下阶梯或者在不规则的表面上走动，大概不会有人费力去建造有人一样的双脚的机器人吧。因为用轮子要简单得多。

效应器的研究大都集中在机器人的手和臂上。早期的研究人员企图采用实验室处理放射性物质的遥控操作器来改造，但这种改造不很成功。因为遥控操作器不便于精确地进行控制。这种装置很大程度上依赖操作人员的眼睛，注视着遥控装置在做什么，并且相应地纠正它的位置。有了这种视觉反馈，操作人员才能做复杂的工作，象把容器的顶盖旋紧，拧进小灯泡，或者划燃火柴等。但是如果让他闭上眼睛，他便会一筹莫展。

现代实验机器人使用专门为它设计的手和臂，设计强调简单和精确，所谓“简单”就是尽可能采用简单的关节设计。以便简化内部和外部坐标转换的任务，减少需要同时运动的关节数而仍能产生给定的手运动。“精确”的含意是能够给电动机发出信号，让它把手运动到非常接近所需要的位置。这就免得机器人总要观察手的运动位置和不断地修改它给电动机的命令，去纠正机械误差（这正是遥控操作器不能令人满意的原因）。

模仿人的手臂并不一定是最好的方法。象旋紧或旋松螺丝的时候，就需要不停地旋转手腕，这时，一个套筒式的前臂要比模仿人的肘弯既简单又精确得多。

现代的机器人使用电动机或气动、液动装置来驱动手和臂。这样作成的操作器非常强大，能负担重荷。但如果在未经校验好的计算机程序控制下，它也是很危险的。研究室里

的机器人常常关在笼子里，以防止它的非常有力然而控制不佳的手臂偶然会打伤旁边站着的人。

机器人手臂设计已成为一件专门工作。工业机器人的手臂常常就是干活的工具。焊工机器人的手臂末端就是焊枪；漆工机器人的手臂末端就是喷漆器。这种专门化的机器手是可以互换的：机器人可以按照工作的需要陆续装上焊枪、攻锥、钻具、喷漆器，烙铁、板钳等等。

但即使是工业机器人也仍然需要通用的手，来拾取和调准手中的工件。最简单的通用手有一双爪子，象老虎钳或管钳那样。对更复杂的手也做过一些实验，不过至今还没有人试验过近似人的手和手指头那样多功能的机器手。正确地说，倒底什么样的设计最好，目前还不很清楚；最好的设计也许主要根据需要机器做什么工作而定，盲目地模仿人手并不见得是最好的设计，将来机器人多功能的手最终被设计出来时，可能仍完全不象人的手和手指。

二、感 觉 器

要使机器人超过前面所讲的简单工业操作器的水平。它就必须具有能够感觉并且对环境作出反应的能力。

最明显的方法是利用电视摄象机的眼睛，让机器人能看到它周围的事物。但很可惜，这并不是最简单的办法；事实相反，它可说是最复杂的办法。在模式识别一章里我们已经知道，图象分析和景物描述在技术上仍很不完善。目前这种分析还只能可靠地用到极其有限的景物方面，例如全部由不同大小和形状的积木组成的景物。在这种简单的情形下，图象分析所需要的程序也是很长、很复杂的。它们必须在很大的快速计算机上运行，才能迅速地得到对机器人有用的结果。

除非将来图象分析技术大有改进，否则，实用的机器人还不会用电视摄象机而宁可用更简单的感觉器。

机器人应当能够确定自己在房间里的位置。假如它在房间里遵循固定的路径运动；例如，一个卸料机器人必须通过走廊从一个房间到另一个房间去，这就可以用一条通电电缆埋在地板下面，装设在机器人上面的电感觉器可以检测到电流，于是机器人就可以使自己的位置确定在电缆的上面。

蝙蝠、海豚和潜艇用的声纳，也可以被机器人用来测定它自己相对于房间四壁以及途中的障碍物的位置，以及它所要操作的物件的位置。机器人周期地发射超声脉冲，返回来的回声就可以指示出墙壁和障碍的所在位置，从脉冲发射到回声返回之间的时间迟延可以给出墙壁和障碍的距离。

声纳的分辨率--即检测细微末节的能力—由于声波的波长很长而受到了限制。所以声纳用于监测细微的工作，远不如用它来检测障碍和确定自己相对于房间的位置。

激光束用于机器人感觉器有多种方式。举例来说，激光束可以象声纳一样使用：一束激光发送出去，它的反射光返回所需的时间可以测量出来。由于激光束很狭小而且波长又短，所以障碍物的反射方向可以测定得比用声纳更精确。可是另一方面，由于光速很快，这个方法只适用离机器人一米以外的障碍物。对于更近的障碍物，发射脉冲和返回之间的时间迟延太短，不便测量。

对较近的物体，可以利用反射激光束的强度来测它的距离。机器人可以用一只手把激光发生器发出的光束指向这只手所要抓取的东西，然后以返回光束的强度来测定手相对于物体的位置。

从激光和其他光源所发出的光线也可以用来帮助景物分

析。举例说，一束激光可以用柱状棱镜散射成一薄层光线。当这一薄层光线碰到物体表面，便可以看到一窄条光线，正好把该物体的轮廓显现出来。这一薄层光线如果扫射到景物上面，电视摄象机观察到这一层光线碰到物体时所形成的线条，便可以用这个信息来进行景物分析。另一个办法是发射一个校验板似的网格到景物上面，观察它落到不同的物体上面网格发生的畸变，便可以帮助景物分析程序确定物体的大小和形状。

机器人不但需要视觉，还需要触觉。最简单的触觉感受元件就是微型开关，装有这种开关的手臂碰到物体时，微型开关就闭合。更先进的感觉器是利用应力仪，比方说，它可以测量手握物体时所用的力的大小。

当机器人打算组装零件时，力感觉器就更为重要。因为稍有偏离，零件就不会一试就顺畅地接在一起。通过测量机器人的手在组装零件时使用的力的大小，控制计算机就可以确定偏离的性质，于是计算机就可以发出校正的命令给机器人的手，来正确地对准零件。这种力感觉器可以装在机器人的手腕上，从而测出在整个手上的力量。

三、控　　制

本书所有其他章节都和机器人的控制有关。一个机器人控制程序必须保持有一个数据库或世界模型，它包含对机器人周围环境的描述。问题求解可以用来确定在一定环境下，采取什么行动才可以成功地实现指定的任务。问题求解又包括状态图、搜索、问题约简、计划，启发式和逻辑推理。甚至和博弈有关的启发式技术也证明是很有用的。模式识别需要用来从感觉器的读数中提取信息。如果给机器人的命令是用英

语给出的，那就要用到自然语言处理技术。最后，有许多（即便不是绝大多数）实验机器人的程序都是用 LISP（一种高级程序语言，又叫表处理语言）来写的！

当然机械人控制还有一些它自己的特点。举一例来说，利用高级问题求解程序去直接处理诸如手臂或腕关节那样的细节问题通常是不实际的。相反，控制程序是分层分级来组织的，最顶层接受外部世界的高级命令。底层则送出极其详细的命令给驱动机械人效应器的机构。

中间各层则照中等详细命令工作，每一层的任务就是接收上一层的命令并把它转化为下一层可执行的细节。这种转换往往需要问题求解技术。对每一层而言，下一层就象是一部接受一组命令的机器。当由顶向底穿行时，命令也变得更有效更精细。在底层我们就可以命令机器人转动手腕到，比如说 5 度，或者命令手爪合上 10 厘米。每一层就代表一个要接受不同详细程度的命令的机器人。我们要注意，这些不同等级的细节与第五章所描写的等级计划颇为一致。

实际上每一级不仅接受上一级的命令，而且还接受下一级表示机器人实际动作的反馈信息。每一级不仅负责向低一级发出命令，而且还利用反馈信息来监督这些命令的执行，并且了解命令是否正确无误地付诸实现。如果命令被执行得不适当，那么就要有一级发出补充命令以改正这种情况。

第十章　计算逻辑：命题和谓词

一个机器人，对其周围环境的事实，需要经常进行推理。当周围环境的某种变化会引起其它部分发生变化时，更是如此。

假设在一个有窗有门的房间里有一个机器人。在门的旁边有张桌子。在桌子上有个盒子。在这种情况下，对于机器人周围环境的知识，有一部分可以用下面三个语句来描述：

1. 桌子在门旁边。

2. 盒子放在桌上。

3. 盒子在门旁边。

现在假设机器人把桌子搬走，让它靠近窗户。显然，这时第一语句必须改为：

1. 桌子在窗户旁边。

那么，对第二个语句和第三个语句应该怎么办呢？这就不太明显了，起码对机器人而言是如此。桌子移动以后，盒子仍然在桌子上吗？如果是这样的话，那么显然（对机器人而言并不"显然"）要把第三语句改为：

3'. 盒子在窗户旁边。

如果放在桌子上的东西没有发生变化的话，我们就可以说桌子是"小心地"被移动到窗户旁的。机器人可能会进行这样的推理：

· 如果盒子是放在桌子上，并且我小心地移动桌子的话，那么盒子还在桌上。

· 盒子原来就在桌子上。

·我小心地移动桌子。

·因此，盒子还在桌子上。

于是，第二条语句可以保留不变。

至于第三条语句，机器人可以这样来推理：

·如果桌子是在窗户旁边，并且盒子是在桌子上，那么盒子也在窗户旁边。

·盒子在桌子上。

·桌子在窗户旁边。

·因此，盒子在窗户旁边。

于是，第三条语句就需要改变为语句 3′ 。

读者可能会说，用不着这么复杂，机器人只要找一找，看看盒子在哪儿就行了。但是，如果机器人实施的是一系列复杂的动作的话，它在实施之前，就要在自己“头脑”中尽力规划好行动的顺序。因此，它就应当拥有一定的手段，能够了解到移动了桌子就会移动盒子。

机器人可能要使用逻辑推理的方法去推断：什么时候达到目标状态，或者首先应如何达到目标状态。如果机器人的程序是要移动桌子，使得盒子在窗户旁边。那么，它可能会这样进行推理：如果桌子在窗户旁边，盒子也就在窗户旁边了，于是目标就达到了。

在下面两个领域里，逻辑推理也是非常重要的：

1. 数学定理证明。

2. 证明计算机程序的正确性。即证明计算机程序实际上能解决它预定要解决的问题。因此，也就是证明这个程序用不着进行“修正”。

最后，用于逻辑推理而发展起来的一种形式体系，可能会提供一种在计算机内部实现的结构，用它来表达关于外部世

界的知识。

计算逻辑(用计算机来进行逻辑推理)是以传统的符号逻辑或数理逻辑为基础的。计算逻辑本身又分为两部分，即比较简单的命题逻辑与比较复杂的谓词逻辑(命题演算与谓词演算这两个术语是经常用到的。其中演算一词仅仅是指一种运算体系。对于人们较为熟悉的微积分演算，这里用不上)。

一、命题逻辑

在逻辑中，简单地说，命题就是一个陈述语句。它可以为真，亦可以为假。下面就是一些命题的例子：

"盒子在桌子上。"

"机器人在窗户旁边。"

"明天早上太阳将会升起。"

一个命题，如果同现实世界的事实相符合的话，即为真(或者是为某些具体目的而假设的事实。例如，我们可以因为某种兴趣，也可以为了弄清真实世界的真相而去研究一个想象的世界)。如果命题同真正的事实或假设的事实不符的话，该命题即为假。在符号逻辑中，我们常常把真简写为T，把假简写为F。

命题常常用单个的字母来表示，就象在代数中用字母表示数字量那样。通常用来表示命题的字母有p、q、r和s。例如，我们可以让：

p代表"盒子在桌子上。"

q代表"机器人在窗户旁边。"

r代表"明天早上太阳将会升起。"

每当我们用到p、q和r时，就相当于用到其相应的命题。如果拿语句

p

为例的话，就等于说：

盒子在桌子上。

如果语句为：

p or q

那么说的就是：

盒子在桌子上或者机器人在窗户旁边。

当然，字母与命题之间的对应关系并不是固定不变的。在代数里，对某个问题我们可以让 a 代表 5，让 b 代表 10。而对另外一个问题，就可以让 a 代表 20，b 代表 100。同样的道理，在某个问题中，p 能够代表：

“盒子在桌子上。”

而在另一个问题中能够代表：

“昨天下过雨。”

在代数里，我们经常使用 a、b、c 和 d 等字母来讨论一些这样的原理和关系，在这些原理和关系中，不论 a、b、c 和 d 的实际值为多少，这些原理和关系总是成立的。同样的道理，我们常常讨论的是 p、q、r 和 s，而不管这些字母实际上代表了什么样的命题，并且这种讨论常常是从诸如“设 p 和 q 为任意命题”的语句开始。

(一)连接词

只有单个的命题是没有什么用的，当我们用“或”(or)、“与”(and)以及“蕴含”(implies)等连接词把命题连接在一起组成新的命题时，逻辑才开始具有意义及用途。

数学家常常用外来符号表示这些连接词，如表 10-1 所示。为了使我们的逻辑表达式易于阅读(同时也为了便于打字和印刷)，我们效仿一些程序语言，使用 **or**、**and**、**not** 和

implies 来代表这些连接词。它们用黑体字印出，以提醒我们：它们是一些特殊的符号，而不是逻辑表达式中用到的英文单字。

表 10.1　典型的数理逻辑符号

连接词	汉译义	符　号
and	与	∧　&
or	或	∨　|
not	非	~　¬
implies	蕴含	⊃　→

当两个命题经由连接词连接起来后，结果所形成的表达式就表示了另外一个命题。该命题的含义是由原先的两个命题及连接词来规定的。

例如，如果 p、q 和 r 代表了上一节所给出的那几个命题，那么表达式：

not　p

代表了：

“盒子不在桌子上。”

而

p　**or**　q

代表了：

“盒子在桌子上或者机器人在窗子旁边。”

(二)真值表

假定我们有了一些含有命题和连接词的表达式，并且进一步假定我们知道表达式中每一独立命题的真假与否。我们希望能计算出由整个表达式所表示的命题的真假与否。

这个计算可以分二步来做。首先，我们对表达式中的每一个命题赋以真值。或者是 T，或者是 F。真命题的值为 T，

而假命题的值为 F。

第二步，我们把 **and**、**or**、**not** 和 **implies** 作为对 T 和 F 进行运算的操作符。就象在代数表达式中用一、－、×和÷对数字进行运算一样。换句话说，我们是用“逻辑计算”来算出整个表达式的真值。

为了进行一般的算术运算，我们需要用加法表和乘法表来确定两个数的和与积。同样地我们需要用 **or** 表、**and** 表、**not** 表和 **implies** 表来进行逻辑计算。我们从每一张表中都可以得到相应的操作符对值 T 和 F 的运算结果，这种表，我们称之为真值表。

最简单的逻辑连接词是 **not**。如果 p 为任一命题，那么 **not** p 为假的话，p 即为真。反之亦然。我们可以把这一事实用下面的真值表来表示：

p	not p
T	F
F	T

因此，尽管我们确实不知道命题 p 表示什么，但我们可以肯定，如果 p 为 T，那么 **not** p 为 F。反之亦然。

如果 p 为真，q 为真，或者说 p、q 二者均为真，那么命题

p **or** q

为真。这样就可以得到 or 的真值表：

p	q	p or q
T	T	T
T	F	T
F	T	T
F	F	F

当 p、q 二者均为真时，p **or** q 亦为真。这一事实可以把

符号逻辑中的 or 看成是具有兼而有之的性质。因此，在把语句变换为符号形式时，必须小心。我们在英语中，有时用到 or 这个词，它具有排它的意义，即指一件事为真或者另一件事为真，而不是指二者皆为真。

在下面的命题中，

p **and** q

只有当 p 和 q 皆为真时，p and q 才为真。这样就可以得到 and 的真值表：

p	q	p and q
T	T	T
T	F	F
F	T	F
F	F	F

连接词 implies 的情况稍微复杂一些。命题

p **implies** q

其相应的语句为：

"如果 p 则 q。"

即：如果说命题 p 为真，则命题 q 亦为真。另一方面，如果 p 为假，那么我们不能断定 q 为真还是为假。

于是，p implies q 为假的唯一情况是：当 p 为真而 q 为假。这样，可以写出 implies 的真值表：

p	q	p implies q
T	T	T
T	F	F
F	T	T
F	F	T

我们稍微多花一些时间来讨论一下 implies 的独特性质。

首先，尽管二个命题看来没有什么关系，但是，其中一个命题可以蕴含另外一个命题。

例如，假定

p 表示“地球围绕太阳运行。”

q 表示“光速为 186000 英里/秒。”

那么：

p **implies** q

为真，因为上面两个语句都为真（见上面的真值表第一行）。这时并不管地球围绕太阳运行与光速之间根本没有什么关系。

使人更加难以理解的是，一个假的命题蕴含着任何一个命题。假设：

p 代表“地球围绕太阳运行。”

q 代表“我有一只可爱的独角兽。”

r 代表“我有一只可爱的大鹏鸟。”

自然，p 为真，而 q 与 r 一定为假，因为独角兽及大鹏鸟都是神话中的鸟兽。但是

r **implies** p

r **implies** q

都为真。这一结论可以从 **implies** 真值表的后两行中看出。

因此，

p **implies** q

为真，只是表示当 p 为真时，q 亦为真。此外没有别的任何含义。它决不表示 p 和 q 之间有任何的因果关系。同时，如果 p 为假，则对 q 是作不出任何真假判断来的。

(三)论证

已知某些命题为真，我们希望最终能推导出其它的命题

也为真。进行这种推导的一种方法是：先把已知为真的命题赋以真值 T，把已知为假的命题赋以真值 F。然后求出待求命题的真值。但是，通常采用的更直接的方法是建立一些规则，通过这些规则可以从那些已知的命题推导出新的真命题。因此，我们也可以建立一些规则来从已知的真命题中推导出我们感兴趣的那些命题。

这样的规则称之为论证方式。它规定了一种被证明是有效的论证形式。我们只要用这些有效的形式来论证一个命题的真实性，就能够确保所得到的结论确实是从假设的前提推导而来的。

例如，下面是一个论证方式。它在传统的逻辑学中被称为合取。

p　　（前提）

q　　（前提）

---

p **and** q　　（结论）

上式中，水平直线以上的命题为前提，假设它们为真。水平直线以下的命题为结论。如果前提为真，那么结论的真实性也是确定无疑的。

合取论证形式可以保证，如果两个命题已知为真，那么用 **and** 把它们连接起来所得到的命题亦为真。下面是这一论证方式的一个例子：

书在桌子上　　（前提）

桌子在窗户旁边。　　（前提）

书在桌子上并且桌子在窗户旁边。（结论）

论证方式的有效性可以用构造一张前提与结论的真值表的方法来证明。在这种真值表中，所有的前提均为真，所以它

的每一行都为 T，如果对这样的所有行来说，结论也为 T 的话，那么该论证形式就是有效的。在这种情况下，前提总是为真，因此真值表能确保结论也为真。

合取包含了前提与结论两个部分，我们采用 **and** 的真值表可以证明合取的有效性。在其真值表中，只有在第一行两个前提都为真，则在同一行结论也为真。因此，合取是有效的。

下面是另一种论证方式，叫做假言推理(MP)：

p	(前提)
p **implies** q	(前提)
q	(结论)

MP 告诉我们，如果一个命题为真，并且该命题还蕴含着另外一个命题，那么另外那个命题也为真。

下面是一个 MP 论证的例子：

天在下雨。	(前提)
如果天在下雨，我必须拿伞。	(前提)
我必须拿伞。	(结论)

MP 的有效性可以用 **implies** 的真值表来证明。同样地，我们也可以在其真值表中只找到一行，该行上两个前提为真，结论也为真。

注意：在论证方式中出现的 p、q 等等字母是可以代表任何命题的。特别是，它也可以代表那些自身包含有连接词的命题。因此，

p **and** q	(前提)
(p **and** q)**implies** r	(前提)
r	(结论)

也是属于假言推理(MP)的例子。但是要注意，当一个表达式替换了另一个表达式中的某个字母时，被替换的表达式要用括号括起来。

现在，我们对上述机器人的论证，采取合取和假言推理(MP)的方法进行形式化。所进行的论证是：当盒子在桌子上，而桌子在窗户旁边时，则盒子在窗户旁边。设

p 代表"桌子在窗户旁边。"

q 代表"盒子在桌子上。"

r 代表"盒子在窗户旁边。"

论证的前提是：

p

q

(p and q)implies r

论证本身可以写成一系列用数字编号的步骤。论证从前提开始，以结论结束。每一步的右边是注释，说明该步是从前面的那一步以及采用什么样的论证方式(C代表合取，MP代表假言推理)得到的。论证过程如下：

1. p	(前提)
2. q	(前提)
3. (p and q)implies r	(前提)
4. p and q	(1,2,C)
5. r	(3,4,MP)

从第1步到第3步是前提，第4步是对第1步、第2步用C得到的。第5步是结论，是对第3步、第4步用MP得到的结果(注意：论证的前提以什么顺序出现是无关紧要的)。

最后，值得看一下一个无效的论证形式的例子。下面的论证方式人们幽默地称之为"白痴"推理(Modus Moron)：

q

p implies q

p

广告和政治论争中有很多这种"白痴"推理的例子：

琼斯赞成同古巴保持更密切的关系。

如果琼斯是个共产党员，他会赞成同古巴保持更密切的关系。

琼斯是个共产党员。

为了看出这种"白痴"推理是无效的，我们可过头来看 implies 的真值表。在第三行中，我们看到"白痴"推理所用的两个前提都为真，但它的结论却为假。由于前提的真实性并不能保证结论的真实性，所以"白痴"推理是一种无效的论证方式。

传统的逻辑学是以具有大量的论证方式为其特色的。所有这些论证方式，用简单的真值表就能很容易得到证明。在下一章，我们会看到，只用一种演绎规则，就可以替代所有这些传统的论证方式。因此，我们不再讨论这些传统的论证方式了。

二、谓词逻辑

(一)个体和谓词

命题逻辑只限于用来处理那些完整的语句。它不能把语句分解为下面二部分：有待肯定的事物及对这些事物要作出的判断。谓词逻辑弥补了这一局限性。

考察一下这个命题：

盒子在桌子上。

它涉及到真实世界的两件东西：

盒子　　(the box)

桌子　　(the table)

盒子与桌子之间的关系是由下面的词规定的：

在……上(is　on)

在谓词逻辑中，凡是命题要对之作出判断的那些项称为个体。为书写方便起见，个体用单名给出。我们把上述命题中的个体写成：

个　体	单　名
盒子	BOX
桌子	TABLE

命题中对个体作出判断的那一部分称之为谓词。我们对每个谓词也用一个单名表示：

谓　词	单　名
在……上	ON

这样，我们就得以用谓词和个体来写出原来的那个命题：

ON(BOX TABLE)

首先写谓词，然后在括号里给出个体。由于它与通常的英语语序不同，所以刚开始看起来很不习惯。但是，人们会很快适应这种表示方法的。

谓词后面括号中的个体称为变元(argument)这是英文单词 argument 的一种特殊用法，它与前面命题逻辑中介绍的用法—论证(argument)完全不同。

一个谓词同其变元组合在一起，就构成了一个命题。命题逻辑的任何一种算法都可以用到命题上面。

我们仍举机器人为例，考虑它对桌子、盒子和窗户所进行的推理。不过，这次不用单个字母面是用谓词和个体来表示

命题。

除了上面已有的外，我们还需一个个体，

个　体	单　名
窗　户	WINDOW

以及一个谓词。

谓　词	单　名
在……旁边	BY

机器人论证的前提是：

BY(TABLE WINDOW)

ON(BOX,TABLE)

(BY(TABLE,WINDOW)**and** ON(BOX,TABLE))

implies BY (BOX,WINDOW)

论证本身的过程如前：

1. BY(TABLE,WINDOW)　　(前提)

2. ON(BOX,TABLE)　　(前提)

3. (BY (TABLE, WINDOW) **and** ON (BOX, TABLE) **implies** BY(BOX,WINDOW)　　(前提)

4. BY(TABLE,WINDOW)**and** ON(BOX,TABLE)　　(1,2,C)

5. BY(BOX,WINDOW)　　(3,4,MP)

注意：由于一个谓词加上的变元构成一个命题，所以在命题逻辑的任一论证形式中，命题可以用来替换 p,q 等。因此

BY(TABLE,WINDOW)　　(前提)

ON(BOX,TABLE)　　(前提)

BY(TABLE,WINDOW)**and** ON(BOX,TABLE)

是进行合取的一个例子。

（二）个体变量

迄今为止，我们仅仅是找到了一种书写命题的方法。对于我们所得到的语句以及变元，我们尚未脱离命题逻辑的范畴。

个体变量（或者变量）允许我们编写语句，这在命题逻辑中是不可能的。

一个变量可以代表任一个体。对于一个为真的含有变量的命题，无论用什么个体来替换这一变量，命题都必定为真。

我们用字母 x，y 和 z 来代表变量。

现在来考察下面的命题：

(BY (TABLE, WINDOW) **and** ON (x, TABLE)) **implies** BY (x, WINDOW)

把转换成语言就读作：

“如果桌子在窗户旁边并有件东西在桌子上，那么，那件东西在窗户旁边。”

注意：句子中“有件东西”和“那件东西”的说法是很罗唆的。但是可惜在语言中找不出与变量直接等同的词。

如果我们断定上面的命题为真，那么对于任何替换 x 的个体来说，我们都能断定该命题的真实性。因此，我们一下子就推断出：

(BY (TABLE, WINDOW) **and** ON (BOOK, TABLE)) **implies** BY (BOOK, WINDOW)

(BY (TABLE, WINDOW) **and** ON (BALL, TABLE)) **implies** BY (BALL, TABLE)

(BY (TABLE, WINDOW) **and** ON (VASE, TABLE)) **implies**

BY(VASE,WINDOW)

(BY(TABLE,WINDOW)**and** ON(PEN,TABLE))**implies** BY(PEN,WINDOW)

等等命题的真实性。实际上，我们可以把其它任何物体的名字来代替变量 x。

使用两个变量，我们可以使上述语句更一般化：

(BY(TABLE,x)**and** ON(y,TABLE)**implies**

BY(y,x)

把它转换成语言就读作：

"如果桌子在什么东西旁边，那么在桌子上的任何东西都在那个东西旁边。"

在这里语言的表达同样也是相当罗唆的。

把上面这个语句具体化，用不同的个体名来替代变量 x 和 y，可以得到许多语句。其中几个如下：

(BY (TABLE, DOOR) **and** ON (GLASS, TABLE)) **implies** BY (GLASS,DOOR)

(BY (TABLE, CHAIR) **and** ON (BOOK, TABLE)) **implies** BY (BOOK,CHAIR)

(BY (TABLE, ROBOT) **and** ON (VASE, TABLE)) **implies** BY (VASE,ROBOT)

要想得到上面命题的可能的最一般化形式，我们必须要用三个变量：

(BY(x,y)**and** ON(z,x))**implies** BY(z,y)

把它转换成语言的实际方法只有一个，即在语言中使用变量：

"如果 x 是在 y 旁边并且 z 在 x 上，那么 z 也在 y 旁边。"

尽管谓词逻辑的某些表达式很难用语言来表示。但是它的其余的表达式却提供了英语语句的非常有效的表达法。当语句中包括了“所有的”或“没有一个”时，这一优点就显得更加突出。

例如，这样一个英语句子：

“所有的人都会死。”

它能表示为：

MAN(x)implies MORTAL(x)

用语言来表达，就是：

“如果什么东西是人，那它一定会死。”

语句：

“没有一种动物是独角兽。”

它能表示为：

ANIMAL(x)implies(not UNICORN(x))

用语言来表达，就是：

“如果任何一种东西是动物，它就不是独角兽。”

(三)实例化

用一个具体的个体名来替换变量，通常称为实例化。个体即为变量的一个具体“实例”，所以称为“实例化”。在正规的讨论中，我们把实例化简写为 IS。

例如，下面是一个人们常常引用的论证，主题是讲人的寿命问题：

所有的人都一定会死。

苏格拉底是个人。

因此，苏格拉底一定会死。

在谓词逻辑中，它们就变为：

1. MAN(x)implies MORTAL(x)　　(前提)

2. MAN(SOCRATES) (前提)

3. MAN(SOCRATES)**implies** MORTAL(SOCRATES)
(1,IS)

4. MORTAL(SOCRATES) (2,3,MP)

第三步是第一步中的 x 用 SOCRATES 替换后得到的。

作为实例化的另一个例子，我们用下面的前提来进行机器人的论证：

BY(TABLE,WINDOW)

ON(BOX,TABLE)

(BY(x,y)**and** ON(z,x))**implies** BY(z,y)

论证的步骤如下：

1. BY(TABLE,WINDOW) (前提)

2. ON(BOX,TABLE) (前提)

3. (BY(x,y)**and** ON(z,x))**implies** BY(z,y)
(前提)

4. (BY(TABLE,WINDOW)**and** ON(BOX,TABLE))**implies**
BY(BOX,WINDOW) (3,IS)

5. BY(TABLE,WINDOW)**and** ON(BOX TABLE)
(1,2,C)

6. BY(BOX,WINDOW) (4,5,C)

在第四步是在第三步中用 TABLE 替换 x，WINDOW 替换 y，BOX 替换 z 后得到的。

(四)存在和函数

常常有这样的情况，我们仅仅需要判定某种东西的存在而不需要进一步说明它。例如，我们或许只希望说：

桌子上有个盒子。

而不去进一步说明这个盒子如何。

我们是这样来判断一件事物的存在：对该事物命名并且把该名字用于表达式中去。我们通常用字母 a、b、c 和 d 来给事物命名，这些字母称为个体常量。

因此，通过写出的

BOX(a)and ON(a,TABLE)

可以断定：桌子上有个盒子。个体常量 a 是对恰好在桌子上的那个盒子的命名。对于 a，我们所知道的仅仅限于：(1)a 是个盒子。(2) a 在桌子上。这些都是上面的命题所蕴含的意义。

有时我们想要判断一个个体的存在，往往取决于其它个体的情况。例如，我们可能习惯于说：

每一个人都有一个母亲。

很显然，对于每一个人的母亲，我们不能使用同样的个体常量。我们需要有一种方法，来表示每个人都有自己的母亲，并且不同人的母亲通常也是不同的。

我们采用函数来实现这一点。在数学与计算机程序中，函数具有这样的性质：当它作用于一个值时，就产生另外一个值。例如，BASIC 语言中的平方根函数总是产生出它的变元的平方根：

SQR(4)－2

SQR(9)＝3

SQR(16)－4

SQR(25)＝5

等等。

当 SQR 作用于一个数时，它总是产生该数的平方根。现在我们设想一个函数，比方说是 f，当它作用于一个个体时，它能产生出该个体的母亲。这样，我们就可以有：

f(JONH)＝SUE(意为 JONH 的母亲是 SUE)

f(JACK)＝JANE(意为 JACK 的母亲是 JANE)

f(JANE)＝MARY(意为 JANE 的母亲是 MARY)

f(JOE)＝HEVEN(意为 JOE 的母亲是 HEVEN)

等等。

这个函数就是我们用以判断每个人都有一个母亲所需要的函数。我们可以这样来写：

PERSON(x)**implies** MOTHER(f(x),x)

这就是说，对任何一个个体，其个体名都可以替换 x。如果那个个体是人，那么该个体就有一个由 f(x) 所指出的母亲。x 的母亲—f(x) 取决于 x。这一事实说明，不同的个体，一般有不同的母亲。

由于采用 3 个体常数，所以我们对一个函数所知道的仅仅限于那些含有这个函数的命题所蕴含的内容。因此，我们对函数 f 所知道的全部内容是：

· f(x)仅当 x 是人时才有意义。

· f(x)是 x 的母亲。

我们通常用 f,g,h 来表示函数。

在数理逻辑中，使用了两种变量：全称变量和存在变量。全称变量，就象我们一直沿用的变量那样，对于任何个体，只要它的名字能替换这些变量，就可以确定该命题为真。存在变量规定了某些个体的存在。每一命题都给出一个特别的前缀，以指定哪些变量属于全称变量，哪些变量属于存在变量。然而，在计算逻辑中，看来只使用全称变量更为方便，而用个体常量及函数来表示存在。

第十一章　计算逻辑：归结法

现今的计算逻辑程序几乎毫无例外地采用了“归结”(resolution)这一大家所知的技术。归结是一种演绎方法，它取代了传统逻辑学中的所有的论证方式。归结的许多方面都是属于技术性的问题，因而这些问题也不是本书所要讨论的内容。尽管如此，对于基于归结的计算逻辑程序赖以运行的基本原理，我们理解起来不会碰到什么困难。

一、归 结 原 理

计算逻辑有自己使用的一套词汇。这里，我们尽量避免介绍过多的专门技术词汇，而只介绍几个对于理解其基本原理有帮助的求语。

原子(atom)。原子是一个命题，它不能再拆开成为其它的命题。用单个字母，例如 p，来表示的命题是一个原子。一个谓词同它的变元组合在一起，也是个原子。例如 ON(BOX, TABLE)。或者从另一方面来说，象

p **and** q

BY (TABLE, WINDOW) **and** ON (BOX, TABLE) 就不是原子。

文字(literal)。文字也是一个原子，或是在前面加有 **not** 的原子。因此，象

p　　ON(BOX,TABLE)

not p　**not** ON(BOX,TABLE)

都是原子。一个文字，如果在其前面没有加上 **not**，即为正文

字(positive)。如果有 not 的话,即为负文字(negative)。例如

正 文 字	负 文 字
p	not p
ON(BOX,TABLE)	not ON(BOX,TABLE)

句子(clause)。子句是用 or 连接起来的一系列文字。因此,象

p or q

(not p)or q

p or(not q)

(not p)or(not q)

ON(BOX,TABLE)or ON(BOX,DESK)

都是子句。

将连接词 or 的含意进一步引伸,用它可以连接两个以上的命题。我们定义

p or q or r or s or ……

在命题 p、q、r、s、…… 至少有一个为真时为真;如果 p、q、r、s、…… 全部为假的话即为假。

如果一个命题在一个命题序列中出现了二次以上的话,那么重复出现的命题可以删去。因此

p or q or r or q 与 p or q or r

是等价的命题。由于在 or 的定义中采用了至少有一个的提法,所以我们可以把重复的命题删去。如果 q 为真,那么 q 出现一次就足以使整个命题为真。q 的重复出现并不会使整个命题变得更真些。反之,如果 q 为假,那么不管 q 出现多少次,也不会影响到整个命题的真实性。

p、q、r、s 出现的顺序也是无关紧要的。因为根据 or 的定

义，它们出现的顺序对命题的真假也没有什么影响。

采用这种引伸了的 or 的定义，我们就可以说一个子句应该具有以下的形式：

l or m or n or o or r…

这里 l、m、n、o、r、…都是文字，或者是原子、负原子形式的命题。

归结原理是一种应用于子句上的论证方式。请看下面二个子句：

p or l or m or……

(not p)or n or o or……

子句 p 为命题，l、m、n、o、……是任意的文字。运用归结原理可以断定：如果这二个命题都为真，那么命题 l or m or n or o …亦为真。我们正式把论证方式的归结法(R)定义为：

p or l or m or…　　(前提)

(not p)or n or o or…　　(前提)

l or m or n or o…　　(结论)

正文字 p 与负文字 not p"相消"，然后所有剩下的文字组合在一个子句里。

归结法的有效性可以很容易得到证明。如果 p or l or m or…为真，那么命题 p、l、m、…中必定有一个为真。由于 p 为假，所以文字 l，m…中必定有一个为真。

如果 p 为真的话，情况又如何呢？在这种情况下 not p 为假。如果这时(not p)or n or o or…为真，那么命题(not p)、n、o、…之一必定为真。

因此，不论 p 为真为假，文字 l、m、n、o、…中必须有一个为真。再根据 or 的定义可以得出结论：l or m or n or o…必为真。

如果我们把归结法所涉及到的命题数目加以限制，那么就可以用真值表来证明归结法的有效性。下面是运用归结法的一个具体例子，表 11-1 给出了它的真值表

p or l

(not p)or m

l or m

表 11-1 检验归结法一个具体例子的真值表

p	l	m	not p	p or l	(not p)or m	m or l
T	T	T	F	T	T	T *
T	T	F	F	T	F	T
T	F	T	F	T	T	T *
T	F	F	F	T	F	F
F	T	T	T	T	T	T *
F	T	F	T	T	T	T *
F	F	T	T	F	T	T
F	F	F	T	F	T	F

* 表示一个前提均为真时，相应的结论也为真。

如果在归结法所用到的二个前提里都出现了同一个文字，那么该文字就会在结论中重复出现。但是根据上面的定义，子句中重复的文字可以删去。所以我们总是把它们删去而不加以特别的说明

下面是一些应用归结法的例子：

a) p or(not q)

(not p)or r

(not q)or r

b)(not MAN(SOCRATES))or MORTAL(SOCRATES)

MAN(SOCRATES)

MORTAL(SOCRATES)

c)　阳光灿烂或者我将带着伞
　　并非阳光灿烂
　　———————————
　　我将带着伞

现在来看看下面这个归结法的应用：

p
(not p)
□

文字 p 与 not p 相互抵消。我们得到是一个空子句 (empty clause)。里面不包含任何文字。习惯上采用一个空方格来表示一个空子句。

从另一个角度来看待空子句的话，可以把它看作一种表示矛盾的方法。在上面的归结中，p 与 (not p) 都是前提。这就是说，我们断定它们二个都为真。但实际上这是不可能的——如果 p 为真，那么(not p)为假，反之亦然。断定它们都为真是自相矛盾的。这二个矛盾前提相消的结果，得到了一个空子句。

这个结论在一般情况下都是对的。我们只要把一个命题同其否定命题归结一下，就可以得到一个空子句。因此，如果论证的结果是以空子句结束的话，则其前提中包含了矛盾，即前提中同时存在一个命题及其否定命题。

矛盾可以看成是一件应避免产生的事情。但它对于归结原理来说，实际上却是一个很关键的事情。我们采用归结的方法，按照以下的步骤，从一组前提来推导出结论。

1. 取给定的前提及所希望结论的否定作为一组新的前提。

2. 从这一组新的前提推导出空子句。

3. 由于假定原来的前提为真，所希望的结果为假，因而导致了矛盾。故只要原来的前提为真，那么所希望的结论也一定为真。或者换句话说，所希望的结论是前提的必然结果。

这一推理方法，数学家们称之为归谬法--推导出谬误来。一个人假设自己所要证明的东西为假，然后证明这将导致"谬误"—矛盾出现。

下面是用归结法进行证明的两个例子：

a)原始前提：

(not MAN(SOCRATES))or MORTAL(SOCRATES)

所希望的结论：

MORTAL(SOCRATES)

证明：

1. (not MAN(SOCRATES))or MORTAL(SOCRATES)　　(前提)

2. MAN(SOCRATES)　　(前提)

3. not MORTAL(SOCRATES)　　(否定结论)

4. MORTAL(SOCRATES)　　(1,2, R)㊀

5. □　　(3,4,R)

b)原始前提

(not ANIMAL(ROVER))or(not UNICORN(ROVER))

ANIMAL(ROVER)

所希望的结论：

not UNICORN(ROVER)

证明

1. (not ANIMAL(ROVER))or(not UNICORN

㊀ 表示归结法。

(ROVER)) (前提)

2. ANIMAL(ROVER) (前提)

3. UNICORN(ROVER) (否定结论)

4. not UNICORN(ROVER) (1,2,R)

5. □ (3,4,R)

证到第三步的时候，要注意 not (not p) 同 p 是等价的。这可以用真值表很容易就证出来。具体证明见下节。

二、转换成子句形式

归结原理仅仅应用于哪些命题为子句的场合。如果归结法证明中所用到的前提及否定结论不是子句形式的话，那么在进行证明之前，必须将它们转换成子句的形式。

即使是简单的命题，要直观上对它进行这种转换也是不容易的。例如，往往是要费脑子才能看得出：

(not ANIMAL(ROVER))or(not UNICORN(ROVER))同

ANIMAL (ROVER) implies (not UNICORN (ROVER)) 是等效的。

幸好，我们可以证明一组恒等式 (set of identities)。运用这些恒等式就可以把命题从一种形式转换为其他的许多形式。就像我们在解代数方程时运用到代数恒等式那样。

如果二个命题有着相同的真值表，那么这二个命题就是等效的。我们用等号来表示这种等效性。一个恒等式表明二个命题是等效的。一个命题总是可以由一个等效的命题来替代，因为它们二者的真值表总是相同的。

恒等式是用真值表来证明的。例如，对于恒等式

not(not p) = p

我们可以用下面的真值表来证明

p	not p	not(not p)
T	F	T
F	T	F

由于 p 及 not (not p) 这二列是恒等的，它们总是具有相同的真值，因此上式是恒等式。

采用同样的方法，我们可以用表 11-2 来证明：

表 11－2　真值表证明恒等式 p implies q＝(not p)or q

p	q	not p	p implies q	(not p)or q
T	T	F	T	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

注：等效的二个命题，其二列真值是恒等的。

下面是一张恒等式表。我们在把命题转换成子句形式时要用到这些恒等式。用真值表的方法可以证明所有这些恒等式。

I－1. **not(not** p**)** ＝p

I－2. p **implies** q＝**(not** p**)or** q

I－3. **nor(**p **and** q**)** ＝**(not** p**)or(not** q**)**

I－4. **not(**p **or** q**)** ＝**(not** p**)and(not** q**)**

I－5. **(**p **and** q**)or** r＝**(**p **or** r**)and(**q **or** r**)**

另外还需要有一条原则。不过，这条原则不可能表示为恒等式。命题

p　**and**　q

是用来判定 p 和 q 二个都为真。从另一方面看，如果我们把

p 和 q 作为各自独立的前提：

p　　（前提）

q　　（前提）

那么，我们仍然可以判定 p 和 q 二者均为真。这是由于一个论证的所有前提都被断定为真的缘故。因此

p **and** q

总可以用二个独立的命题

p

q

来代替。我们把这一原则称之为 AND 原则。

下面我们把一些命题转换成子句的形式。右边括号内的注释说明了在转换中该步运用了那个恒等式或原则。

a) 1. MAN(SOCRATES)**implies** MORTAL(SOCRATES)

2. (**not** MAN(SOCRATES))**or** MORTAL(SOCRATES)

(I－2)

b) 1. ANIMAL(x)**implies**(**not** UNICORN(x))

2. (**not** ANIMAL(x))**or**(**not** UNICORN(x))　　(I－2)

c) 1. (BY(x,y)**and** ON(z,x))**implies** BY(z,y)

2. (**not**(BY(x,y)**and** ON(z,x)))**or** BY(z,y)　　(I－2)

3. (**not** BY(x,y)**or**(**not** ON(z,x))**or** BY(z,y)　　(I－3)

d) 1. (MAN(x)**or** WOMAN(x))**implies** MORTAL(x)

2. (**not**(MAN(x)**or** WOMAN(x)))**or** MORTAL(x)

(I－2)

3. ((**not** MAN(x))**and**(**not** WOMAN(x)))

or MORTAL(x)　　(I－4)

4. ((**not** MAN(x))**or** MORTAL(x))

and((**not** WOMAN(x)**or** MORTAL(x))　　(I－5)

5a. (not MAN(x))or MORTAL(x)

(AND)

5b. (not WOMAN(x))or MORTAL(x)

(AND)

e) 1. not(not UNICORN(x))

2. UNICORN(x) (I—1)

三、一 致 化

迄今为止，在运用归结法进行证明的例子中，我们还没有涉及到变量。下面我们来研究一下前提中含有变量的证明。

原始前提：

(MAN(x)or WOMAN(x))implies MORTAL(x)

WOMAN(JANE)

所希望的结论：

MORTAL(JANE)

证明：(原始前提已转换成子句形式)

1. (not MAN(x))or MORTAL(x) (前提)

2. (not WOMAN(x))or MORTAL(x) (前提)

3. WOMAN(JANE) (前提)

4. not MORTAL(JANE) (否定结论)

5. (not WOMAN(JANE)or MORTAL(JANE)

(2,1S)

6. MORTAL(JANE) (3,5,R)

7. □ (4,6,R)

我们要特别注意第四步，式中

(not WOMAN(x))or MORTAL(x)

的 x 用 JANE 替代了以后，它就变成了实例：

(not WOMAN(JANE))or MORTAL(JANE)

这样做的目的是为了实现以下的归结过程：

(not WOMAN(JANE))or MORTAL(JANE)

WOMAN(JANE)

MORTAL(JANE)

实例实现的具体过程就是所谓的一致化（unification）。一致化就是力图找出变量的替换量来使得二个原子恒等。（我们可以回顾一下，原子是一个含有其变元的谓词，如WOMAN(x)。）之所以一致化，有二点理由：

1. 归结（resolution）。如果经过一致化（即使其恒等）的两个原子具有相反的"符号"（即一正一负），并且是在不同的子句中出现，那么这二个原子所在的二个子句可以实现归结。

2. 因子析出。如果经过一致化的两个原子符号相同，并且出现在同一个子句中，就是说，该子句包含了二个恒等的文字。重复的文字可以删去，这就是所谓的因子析出（factoring），而经过替换并且删去了重复文字的子句就是原始子句的因子（factor）。

为了在较为复杂的子句中应用一致化、归结和因子析出更方便起见，我们需要一些简明的标号来表示谓词、变量、常量以及函数。我们假定：

P,Q,R　代表谓词

x,y,z　代表变量

a,b,c　代表常量

f,g,h　代表函数

我们下面来考虑二个原子：

p(x,f(x),y)

p(a,z,g(z))

在这二个谓词中，变量是 x，y 和 z。如果可能的话，我们希望能找到这些变量的替换量，从而使得二个原子恒等。

我们从左到右来检查一下这二个原子。我们首先注意到，它们的谓词名称 p 是相同的。如果情况不是这样的话，我们就应立即放弃上面的愿望。因为没有那一种变量的替换可以把二个不同名称的谓词变得相等。

下面我们看看每个原子的第一个变元。它们是 x 和 a，如果用 a 替换 x，这二个变元就可以恒等。在第一个原子中用 a 替换 x，就得到：

p(a,f(a),y)

p(a,z,g(z))

接着我们来看每个原子的第二个变元。相应的变元是 f(a) 和 z。如果用 f(a) 来替换 z 的话，它们就能恒等。在第二个原子中用 f(a) 替换 z 后得到

p(a,f(a),y)

p(a,f(a),g(f(a)))

现在，每个原子只剩下第三个变元要考虑了。它们是 y 和 g(f(a))。用 g(f(a)) 替换 y 得到

p(a,f(a),g(f(a)))

p(a,f(a),g(f(a)))

这样就使得这二个原子恒等了。实际上，我们进行了下面三次替换：

a　替换 x

f(a)　替换 z

g(f(a))　替换 y

现在，我们来看看二种实现一致化都要用到的方法是如何运用的，假定有二个子句：

P(x,f(x),y)or Q(x,y)

(not P(a,z,g(z)))or R(z)

如果我们采用上面同样的方法来置换，把二个子句的第一个原子一致化起来，就得到：

P(a,f(a),g(f(a)))or Q(a,g(f(a)))

(not P(a,f(a),g(f(a))))or R(f(a))

现在就可以对这些子句应用归结法了。把每一个子句的第一个文字删去，就得到

Q(a,g(f(a)))or R(f(a))

在另一方面，假设二个业已一致化的原子出现在同一子句中，情况又如何呢？例如，假定有：

P(x,f(x),y)or P(a,z,g(z))or Q(x,y)

用替换的方法把前面二个原子一致化，得到：

P(a,f(a),g(f(a)))or P(a,f(a),g(f(a))) or Q(a,g(f(a)))

删去重复的文字，得到：

P(a,f(a),g(f(a)))or Q(a,g(f(a)))

这就是原始子句的因子。这说明原始的子句只有一个因子。但是，一般说来，在一个子句中，不同的替换可以把不同的文字一致化，因面得到不同的因子。因此，一个子句可以有许多因子。

现在，我们可以完整地叙述一下应用归结法如何从一组前提来推导出结论的过程：

1. 从前提与所希望结论的否定入手，首先得到与之对应的子句。

2. 采用归结或因子析出的方法进行一致化，得到新的子句。

3. 如果得到的是空子句，那么说明所希望的结论是原始前提的必然结果。但是，如果在得到空子句前中止了推导过程，那么或者可能是从前提推不出所希望的结论，或者可能是我们中止得太早，只要再多做些工作就会得到空子句。

下面是采用归结法进行证明的最后一个例子。表 11-3 是机器人采用归结法进行论证的这一部份。它证明了当盒子在桌子上并且桌子靠近窗子时，盒子靠近窗子。下面的注释说明了一致化是在那一步完成的以及进行了什么样的替换。在这一论证过程中没有用到因子析出。但是一种归结法总是需要一种一致化方法的。注释就说明了对那一个原子进行了一致化，做了什么样的替换。

表 11-3　利用归结法的一个典型证明

1. BY(TABLE, WINDOW)	(前提)
2. ON(BOX, TABLE)	(前提)
3. (not BY(x, y)or(not ON(z, x))or BY(z, y)	(前提)
4. not BY(BOX, WINDOW)	(否定结论)
5. (not BY (TABLE, WINDOW)) or (not ON (z, TABLE))or BY(z, WINDOW)	(3, IS)①
6. (not ON(z, TABLE))or BY(z, WINDOW)	(1, 5, R)
7. (not ON (BOX, TABLE)) or BY (BOX, WINDOW)	(6, IS)②
8. BY(BOX, WINDOW)	(2, 7, R)
9. □	(4, 8, R)

①BY(x, y) 和 BY (TABLE, WINDOW) 是用 TABLE 替换 x、用 WINDOW 替换 y 而实现一致化的。实现了这个一致化，第 6 步才能应用归结法。

②ON (z, TABLE) 和 ON (BOX, TABLE) 是用 BOX 替换 z 而实现一致化的。这样，在第 8 步时才能应用归结法。

一致化并不局限于在谓词逻辑中使用。在后面的章节中我们可以看到，在使表达式恒等时，不少情况下都需要采用替换变量的方法才能实现。

四、小　　结

归结法只是在下述意义上才是完备的：如果结论确能从前提中推导出来，那么重复地采用一致化，归结法及因子析出方法才会最终得出空子句。

归结法在计算机上的程序化是很容易实现的。并且同原先的计算逻辑的程序比起来，其程序效率要高得多。

归结法程序所能处理的前提与结论，比我们在本书中介绍的要复杂得多。但是，在目前，它还不能处理类似下面的一些复杂问题。如证明深奥的数学定理（用于验证复杂的计算机程序）；帮助机器人适应现实世界的复杂性（不同于实验室中有限的范围），等等。对于这些问题来说，归结法程序在得到空子句之前，就已经耗尽了计算机允许的时间与存贮空间。

困难所在就是组合爆炸问题。这在人工智能领域中是常有的现象。一致化、归结、因子析出所得到的许多子句，同推导出空子句并没有什么关系。这样，程序只是在推导的路径上浪费时间，并且导致进入死胡同。

由于存在着这样一些困难，因此有的学者认为，用计算逻辑来解决复杂的定理证明问题是不可能的，因而放弃了这方面的工作。另外一些人在寻找一些约束条件，使得在运用归结及因子析出的同时可以减少所产生的子句数目而不破坏其完备性。还有一部分人（包括作者本人在内）认为，这些问题的解决在于要使用得力有效的启发式方法及规划技术，以指导归结程序的运行，达到推导出空子句的目标。

第十二章　知识表达法

一个人工智能程序，必须对自己所要考虑的周围世界，拥有一个内在的模型。有些程序只是把问题输入，把结果打印出来。所有的推导工作都是在这个内在模型内完成的，因而程序本身根本不同现实世界打交道。另外一些同现实世界产生交互作用的程序，象机器人控制程序等，也是需要内在模型的。机器人在现实世界中实际执行动作之前，程序要用这个内在模型来预测及规划一系列动作。

在第二章中，我们探讨过对于简单问题的属性值表达法。这些问题所代表的世界是用一定的属性的集合来描述的。这些属性值的每一个可能的集合都刻画了这一世界的可能状态。在本章的后面，我们要介绍一个更加完善的属性值表达法。现在，我们先来看看如何使用谓词逻辑，通过一组命题来描述世界的。

一、基于谓词逻辑的表达法

我们先从一个例子入手。这个例子是同我们在第四章中遇到的一个例子相类似的。

图 11-1 所示的世界是我们要对它抽取模型的世界。这是一个有壁室（ALCOVE）的房间。房间内有二张桌子—A 和 B、一个盒子（BOX）和一个机器人（ROBOT）。机器人可以在壁室内，也可以在 A 桌旁或 B 桌旁。盒子可以在 A 桌上，或者在 B 桌上或者被机器人抓在手中。

我们用下面的单名来代表这个世界中的个体：

单　名	个　体
ROBOT	机器人
ALCOVE	壁　室
BOX	盒　子
A	桌　A
B	桌　B

我们还需要定义一些谓词。定义由每个谓词实际使用的例子来给出。当然，这些谓词所取的变元可以与例子中的变元不同。

例	含　义
TABLE(A)	A 是一张桌子
EMPTY HANDED(ROBOT)	机器人双手空空
AT(ROBOT,A)	机器人在桌 A 旁
HOLDS(ROBOT,BOX)	机器人拿着盒子
ON(BOX,A)	盒子在桌 A 上

注意：我们在使用这些谓词时，可能会得到相互矛盾的句子。如：

EMPTYHANDED (ROBOT) and HOLDS (ROBOT, BOX) 还可能得到一些毫无意义的句子。如：

EMPTYHANDED(ALCOVE)

必须小心避免出现上述两种情况。

我们采用数据库的方法来描述机器人目前的状态。简单来说，数据库实际上是一张语句(即命题)表。它列出了机器人世界的有关事实。

例如，我们希望描述图 12-1 所示的状态：机器人手空着，

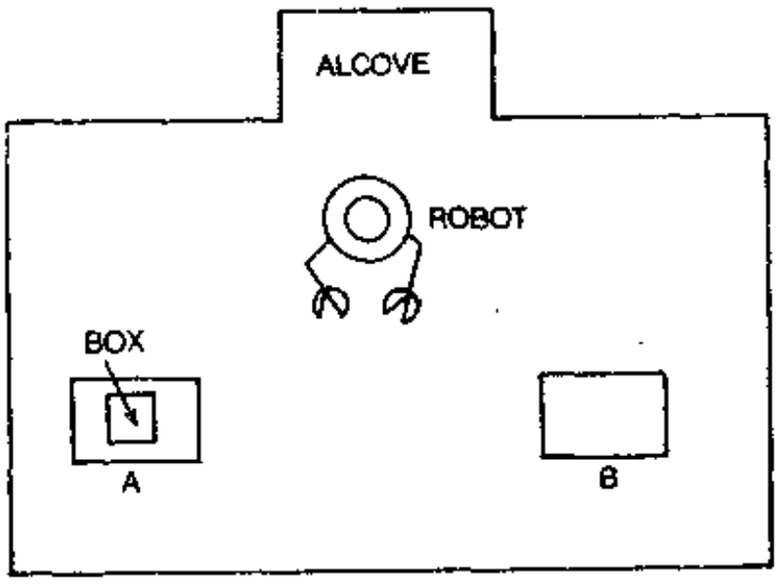


图 12-1　本章例子所采用的机器人世界

站在壁室处，盒子放在桌 A 上。对于这种状态，数据库就是：

AT(ROBOT,ALCOVE)EMPTYHANDED(ROBOT)

ON(BOX,A)

TABLE(A)

TABLE(B)

（我们必须规定 A 和 B 是桌子，这样才能运用条件"盒子只能放在桌子上。）

假设我们希望机器人把盒子从桌 A 移到桌 B 上，然后自己回到壁室。其最后状态为：

AT(ROBOT,ALCOVE)

EMPTYHANDED(ROBOT)

ON(BOX,B)

TABLE(A)

TABLE(B)

我们看到，在数据库中，大多数语句都保持不变。这一特

点是很典型的。我们在一个世界内所做出的每一个动作只是改变了数据库中少数几条语句。在更为实际与复杂的世界中，保持不变的语句数目比起这个简单的例子来要远为大得多。

所以，在一个动作实施了以后，我们集中精力注意数据库中那些由该动作所引起的变化。这样，我们就可以集中研究由动作引起的少量变化，而不是把精力放在那些数量大而又保持不变的语句上。

例如，机器人把盒子从桌 A 上移到桌 B 上，然后回到壁室，这一工作可以用

删：ON(BOX,A)

增：ON(BOX,B)

来描述。就是说，机器人动作后的总效果是在数据库中删去 ON(BOX,A)，增加 ON(BOX,B)

其它的动作也可以用同样的方法来表示。比如，假设我们希望机器人走近桌 A，然后拿起盒子。那么，这一动作的效果就是：

删：AT(ROBOT,ALCOVE)

EMPTYHANDED(ROBOT)

ON(BOX,A)

增：AT(ROBOT,A)

HOLDS(ROBOT,BOX)

因此，对于我们所研究的世界内所希望实施的任何动作，都可以用一张增表及一张删表来表示。删表表示一个动作结束后要从数据库中删掉的那些语句；增表表示该动作结束后要加入到数据库中去的语句。

我们向机器人发出的初始命令是用操作符（operator）来

表示的。如用在第二章中讲述过的那样，一个操作符是由一个条件及一个动作组成的。这个条件决定，在一个具体的状态下，是否可以使用该操作符。如果决定可以使用该操作符的话，那么动作部分就会告诉我们状态是如何改变的。

我们已经谈过动作是怎样表达的。那么条件又是如何表达的呢？

如果操作符是可以运用的话，那么对于相应的状态来说，条件就是一个必定为真的命题。就是说，如果我们把从数据库中取出的语句作为前提，那么由此得到的结论即为操作符的条件。这样的情况下才能运用该操作符。否则不行。

我们来看看下面这个操作符是如何表示的：

PICK UP BOX FROM A

该操作符要机器人拿起放在桌 A 上的盒子。很显然，运用该操作符必须满足以下条件：盒子放在桌 A 上；机器人站在桌 A 旁；机器人手是空的。我们用条件及描述动作的增、删表来表示这个操作符：

条件：　ON(BOX,A)
　　and AT(ROBOT,A)
　　and EMPTYHANDED(ROBOT)

删表：　EMPTYHANDED(ROBOT)
　　ON(BOX,A)

增表：　HOLDS(ROBOT,BOX)

条件这一项意味着，仅当 ON(BOX, A)、AT(ROBOT, A) 及 EMPTYHANDED (ROBOT) 都为真时，该操作符才能被运用。就是说，它们必须都能在数据库中找得到。

（我们假定 and 能推广运用于二个命题以上的情况，就象前一章中对 or 的推广一样。因此，只有在 p、q、r、s、…都为真

时，

p and q and r and s and…

才为真。恒等式1—4也能推广为：

not(p and q and r and s and…)

=(not p)or(not q)or(not r)or(not s)or…

这是因为如果p、q、r、s、……中只要有一个为假时，p and q and r and s and……即为假。）

现在，我们假设机器人世界的目前状态为：

AT(ROBOT,A)

EMPTYHANDED(ROBOT)

ON(BOX,A)

TABLE(A)

TABLE(B)

如果PICK UP BOX FROM A这个操作符是可以运用的话，那么其条件一定是把上述语句作为前提所推出的必然结论。利用归结法，我们很容易证明，条件是这些语句的必然结果：

1. AT(ROBOT,A) （前提）

2. EMPTYHANDED(ROBOT) （前提）

3. ON(BOX,A) （前提）

4. TABLE(A) （前提）

5. TABLE(B) （前提）

6. (not ON(BOX,A))
   or(not AT(ROBOT,A))
   or(not EMPTYHANDED(ROBOT))
   （否定结论）

7. (not AT(ROBOT,A))
   or(not EMPTYHANDED(ROBOT))

(3,6,R)

8. **not** EMPTYHANDED(ROBOT) (1,7,R)

9. □ (2,8,R)

第6步中的否定结论是这样得到的：应用恒等式1—4的一般化形式，把条件变为其否定形式。

如果条件仅仅包括了用 and 连接起来的语句，并且每一语句可以在也可以不在数据库中出现时，那么我们实际上并不一定要应用归结法。我们很容易看出，如果条件所包含的所有语句都在数据库中出现的话，那么条件即为真；反之为假。在条件相当复杂时，还是需要采用归结法(或计算逻辑的一些其它方法)的。

在条件中使用变量，如同使用独立的常量那样，通常是很方便的。由此而得到的条件更加一般化。比方说，我们并不针对具体的个体 ALCOVE，而可以在 ALCOVE 出现的地方用变量来表示。这样就可以允许其他任何名字的个体来替换这个变量。

在同一个操作符中，我们有两种使用变量的方法。在某些场合下，我们希望用某个常量来替换某个变量；而在其它的场合下，我们希望所使用的常量是原先已经用来替换过某一变量的常量。

区别这两种情况有许多不同的方法。我们沿用温斯顿(winston)在他的人工智能教科书中采用的方法。

>x　试图用一个常量替换 x，使得条件为真。

<x　使用原先已经替换过 x 的那个常量。在这种情况下，决不进行新的替换。

例如，考虑下面这个条件：

ON(BOX,>x)

and AT(ROBOT<x)

and EMPTYHANDED(ROBOT)

在 ON (BOX, >x) 中的>x 就是要我们去找到一个个体去替换它，以使 ON(BOX, >x) 能在数据库中出现。比方，在数据库中包含了 ON(BOX,A)，那么 A 就会是>x 的一个可能的替换。

在 AT (ROBOT, <x) 中的<x 就是要我们去看看，当<x 取一个先前已替换过>x 的常量时，在数据库中是否存在这样一个语句。例如，如果 A 替换过>x，那么我们必须看看数据库中是否有语句 AT(ROBOT,A)。

注意：我们不能对<x 进行新的替换。举例来说，如果我们在数据库中找到的是 AT (ROBOT, B) 而不是 AT (ROBOT, B)，那么我们就不能用 B 来替换 A。这时，我们所了解到的只是 AT (ROBOT, A) 不在数据库中。因而在用 A 替换>x 时，条件不为真。

作为另外一个例子，对于下面的数据库来说，上述的条件为真：

AT(ROBOT,B)

EMPTYHANDED(ROBOT)

ON(BOX,B)

TABLE(A)

TABLE(B)

同时，B 可以替换>x。从另一方面来看，上述条件对于数据库

AT(ROBOT,ALCOVE)

EMPTYHANDED(ROBDT)

ON(BOX,B)

TABLE(A)

TABLE(B)

不为真。B 是>x 的唯一可能的替换。但是，AT(ROBOT, B)并不在数据库中。

自然，对于这样的数据库：

AT(ROBOT, ALCOVE)

EMPTYHANDED(ROBOT)

TABLE(A)

TABLE(B)

上述的条件也不可能为真。因为没有一个>x 的替换，可以使得 ON(BOX, >x)可以在数据库中出现。

变量>x 只允许在条件中出现。但是，<x 还可在增表或删表中出现。在增，删表中出现的语句取决于哪个个体常量取代>x 后能使条件为真。

例如，考虑下面的操作符：

PICK UP BOX

条件： ON(BOX, >x)

and TABLE(<x)

and AT(ROBOT, <x)

and EMPTYHANDED(ROBOT)

删：EMPTYHANDED(ROBOT)

ON(BOX, <x)

增：HOLDS(ROBOT, BOX)

如果有一个盒子停放在什么东西(>x)上，而这件东西又是桌子，机器人又站在桌子旁，那么上面的条件即为真。操作符的动作就是删去“机器人手空着”及“盒子在桌子上”这二个语句；加上“机器人拿着盒子”这一个语句。

这样，对于下面的数据：

AT(ROBOT,A)

EMPTYHANDED(ROBOT)

ON(BOX,A)

TABLE(A)

TABLE(B)

上述条件为真。并且所完成的动作为：

删：EMPTYHANDED(ROBOT)

ON(BOX,A)

增：HOLDS(ROBOT,BOX)

对于这样的数据库：

AT(ROBOT,B)

EMPTYHANDED(ROBOT)

ON(BOX,B)

TABLE(A)

TABLE(B)

上面的条件亦为真。但是，由于是用 B 而不是用 A 来替换 $\succ_x$，所以完成的动作也不同：

删：EMPTYHANDED(ROBOT)

ON(BOX,B)

增：HOLDS(ROBOT,BOX)

我们回想一下第四章，在那里讲到了两个控制机器人的操作符：一个是 SET DOWN BOX，它要机器人把盒子放在邻近的桌子上。另一个是 GO TO a，它要机器人走到 a 点（a 可以是 A、B 或者 ALCOVE）。

SET DOWN BOX 这一操作符同 PICK UP BOX 非常相似：

SET DOWN BOX

条件：AT(ROBOT,＞x)

and TABLE(＜x)

and HOLDS(ROBOT,BOX)

删：HOLDS(ROBOT,BOX)

增：EMPTYHANDED(ROBOT)

ON(BOX,＜x)

如果机器人是站在桌子旁边并且手里拿着盒子，那么 SET DOWN BOX 就可以实现：机器人把盒子放在桌子上，手是空的。

GO TO a 要把机器人移动到地点 a：

GO TO a

条件：AT(ROBOT,＞x)

删：AT(ROBOT,＜x)

增：AT(ROBOT,a)

这个条件是非常一般化的。它的含义仅仅是说，机器人一定是在某处。该条件的真正目的是找到某个常量可以替换＞x，以便把 AT(ROBOT,＜x)删去。

二、知识表达法与规则

有关世界模型的事实，决不是一个智能程序所需要的唯一知识。程序还应具有如何执行任务所需要的启发式知识(heuristic knowledge)、经验或提示。

让我们看看，对于制定一个把盒子从桌 A 移动到桌 B，可能需要具备什么样的知识。

首先，我们注意所要解决的问题。我们回想起来，我们是用初始状态、一组目标状态和一组操作符来确定一个问题的。

我们可以把初始状态定为：

AT(ROBOT,ALCOVE)

EMPTYHANDED(ROBOT)

ON(BOX,A)

TABLE(A)

TABLE(B)

而操作符就是上一节中给出的那几个。

通常，我们对每一目标的细节并不感兴趣。而仅关心那些所期望的条件能够成立。因此，我们通过规定那些必须成立的条件来规定目标状态。任何一个状态，只要条件对它来说为真，该状态就是目标状态。

因此，如果我们只是希望确保盒子是在 B 桌上的话，那么目标状态就是：

目标： ON(BOX,B)

如果我们要求机器人在移动盒子后一定回到壁室的话，我们的目标就是

目标 ON(BOX,B)

and AT(ROBOT,ALCOVE)

等等。

为了拟定一个计划的各个步骤，我们集中注意的是每一步所能达到的目标。对于每一步来说，其初始状态就是当时间到了要实现这一步时，现实世界所处的状态。

当计划的每一步开始实施时，为了跟踪真实世界的变化，对每一步我们都提出一份增表及一份删表。这些表规定了我们所知道的语句中，那些应当删去，那些应当增加，以便达到该步预定的目标。由于达到该目标的任何细节还没有作出规定，所以这些增、删表通常是不完全的。只有在对某一步进行

细化时，才会在表中再增加新的语句。这样，同规划的步骤相联系的增、删表使得我们一步一步地跟上状态的变化，一直到当前的规划层所了解的状态为止。在对规划进行细化时，我们就会进一步了解到每一步对现实世界状态所引起的变化。

我们回头来看看上面的例子。我们希望得到的目标是：

目标：ON(BOX,B)

目前，我们所知道的达到这个目标会引起的唯一变化是

增：ON(BOX,B)

这二个语句就构成了我们规划的最高、亦是最粗略的一层。如图 12-2 所示。

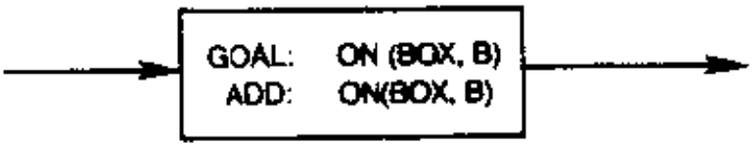


图 12-2　把盒子放在桌 B 上这一规划的最高层

为了细化每一步，我们需要制定一个规划。它应既能达到这一步骤的目标，又要适合现实世界当前的状态。因此，规划可以分成以下二部分：

1. 目标。这是规划实现后得到的结果。
2. 条件。为了使规划切实可行，条件在真实世界中必须为真。

例如，考虑按照以下分类的规划：

目标：ON(BOX,>x)

条件：AT(ROBOT,>y)

and EMPTYHANDED(ROBOT)

这一分类说明：(a) 这一规划是要把盒子放在某一物体--->x

上。(b) 只有当机器人在某处(在>y 处)而且它手是空的时候,这一规划才是有用的。在阐明这一规划的具体步骤时,要用到替换>x 和>y 的值。

我们分两步来得到一个达到既定目标的规划。首先,我们尽量把规划中的目标语句同要达到的目标进行匹配。在上面的例子中,要达到的目标是:

ON(BOX,B)

只要用 B 替换>x,ON(BOX,>x)就可以同该目标匹配。

这样,我们现在知道了,这一规划能够实现既定的目标。但是,我们还不知道,在现实世界的当前状态下,这个规划是否可用。因此,第二步就要看看在目前状态下,条件是否满足。在上面的例子中,当前状态正好是初始状态,并且我们只要用 ALCOVE 来替换>y,条件就可以满足。

规划本身包含三步:

1. 机器人拿起盒子。

2. 机器人拿着盒子走到<x 处,在本例中走到桌 B 旁。

3. 机器人把盒子放在<x(B 桌)上,同时返回到它的起始点<y(ALCOVE)。

至于要达到的目标以及现实世界所发生的变化,我们可以得到:

1. 目标: HOLDS(ROBOT,BOX)
   删: EMPTYHANDED(ROBOT)
   增: HOLDS(ROBOT,BOX)
2. 目标: AT(ROBOT,<x)
   增: AT(ROBOT,<x)
3. 目标: ON(BOX,<x)
   and AT(ROBOT,<y)

删：HOLDS(ROBOT,BOX)

增：ON(BOX,<x)

AT(ROBOT,<y)

图 12-3 是一个用 B 替换<x，用 ALCOVE 替换>y 后得到的一个实际规划。

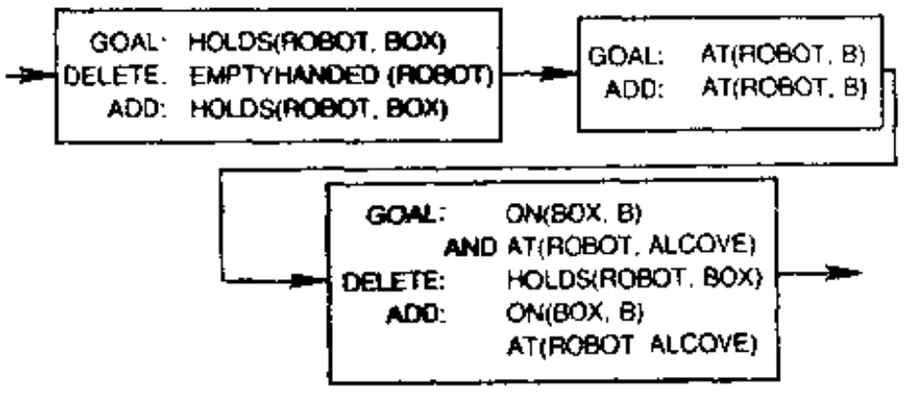


图 12-3 把盒子放在桌 B 上这一规划的最初细化

在每一步中，增、删表都给我们表示出：从一步到下一步状态是怎样改变的。因此，我们可以看出，第一步完成后，删去了 EMPTYHANDED (ROBOT)，而加上了 HOLDS (ROBOT, BOX)。第二步之后，加上了 AT(ROBOT,<x)等等。

另一方面，这里并没有提到删去 AT (ROBOT, <y)。很显然，在增加 AT(ROBOT,<x)之前，就应该把它删去。产生这一现象的原因是：在规划做到这一层时，我们的确不知道在第一步或第二步时是否应该把它删掉。机器人不用移动位置能否够得着盒子，或者机器人必须先移动到某个位置上去才行。这些问题只有在对规划作进一步细化时才能解决。因此，正如已经讲过的，在一个规划某些高层所规定的状态变化，决不是完全的。

现在，我们可以对规划的每一步作进一步细化了。例如，

我们看看如何对第三步进行细化。在第三步，我们要达到的目标是：

ON(BOX,B)and AT(ROBOT,ALCOVE)

考虑如下分类的规划：

目标： ON(BOX,>x)
and AT(ROBOT,>y)

条件： AT(ROBOT,<x)
and HOLDS(ROBOT,BOX)
and TABLE(<x)

其目标部分，在用 B 替换>x，用 ALCOVE 替换>y 后，就可以同要达到的目标匹配。

（注意：在本规划中使用的变量>x 及>y 同其它规划中用到的同名变量毫无关系。本规划中用来替换>x 和>y 的值同以前谈到的规划中的相同，这纯粹是一种巧合。）

现在，我们必须来看看，分类中的条件部分在当前的现实世界中是否成立。也就是说，条件中的所有三个语句在该步实施的一开始是否就成立。实际上，这三个语句是成立的。因为 AT (ROBOT, B) 是在第二步加入的，HOLDS (ROBOT, BOX) 是在第一步加入的，而 TABLE(B) 在初始状态即为真。这三条语句，没有任何一条在以后的步骤中删除过。

（由此我们可以看到增、删表的用途，尽管它们还是不完全的。）

规划的各个步骤如下：

1. 操作符：SET DOWN BOX
删：HOLDS(ROBOT,BOX)
增：EMPTYHANDED(ROBOT)
ON(BOX,>x)

2. 操作符　GO TO>y

删：　　AT(ROBOT,>x)

增：　　AT(ROBOT,>y)

图 12-4 是第二步细化后的实例。

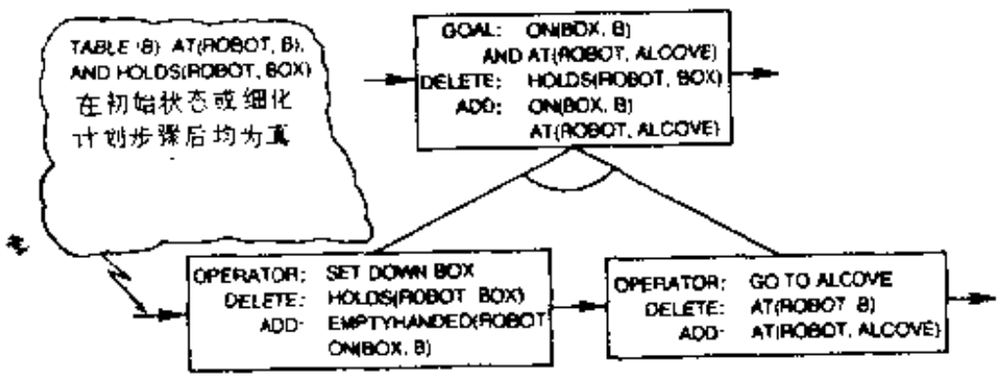


图 12-4　图 12-3 中第二步的进一步细化。现在每一步都是一个操作符——直接向机器人发布的命令

注意：本规划的每一步包含了一个要采用的操作符，而不包含所要达到的目标。不规定目标而规定了操作符的步骤是最基本的步骤。就是说，它不能进一步细化了。

注意：本规划的条件保证了每一单独操作符运用所需要的条件都能得到满足。

概括起来说，我们把每一个规划都看作为取得某一结果所需要的提示、格言、经验、忠告或启发。其目标部分规定了要得到的结果；而条件部分告诉我们什么时候规划才能被采用。在对规划这一技术运用水平比较高的时候，就象不关心以往的经验何时能指出规划有了良好的成功机会一样，我们也不怎么关心规划的每一步何时具体实现的问题。因此，条件主要是以经验得出的结果为基础，而不是主要依靠逻辑分析。所以，条件的真实性并不能保证计划一定成功，而只是指

出了这是一个值得一试的计划。

对于制定、存贮、调出和使用计划所需要的综合技术来说，本书所谈到的思想，以及在大多数其它人工智能文献中所提出的思想，都只不过是朝着这个方向刚刚迈出的该提似的几步。

三、特 性 表

迄今为止，对于所研究世界的模型，我们都是用谓词逻辑形式的语句来表达其中的各种事实。我们现在采用另外一种方法来表达这些事实。这一方法是在第二章中所讨论的简单的属性值表达法的推广。

我们把特性定义为一个属性值对。下面是一些特性的例子：

属 性	值
COLOR(颜色)	RED(红)
ENGINE(引擎)	V8
MODEL(型号)	1978
BODY(车体)	SEDAN(轿车)
MAKE(制造厂家)	FORD(福特)

所有这些特性都是同汽车的描述有关。很显然，对此还可以给出更多的特性。更确切地说，需要给出的特性取决于我们描述它的目的。车主感兴趣的特性、汽车出售商所关心的特性，以及汽车修理技师所关心的特性都是各不相同的。自然，也有一些相同的部分。

描述一个现实世界的状态，有一种方法就是把这个世界中的每一个物体同一种特性表结合起来。这个特性表包含了该物体的所有特性，而这些特性都是同状态描述的目的相联

系的。

例如，在表 12-1 中，给出了机器人、二张桌子和一个盒子的特性表。我们就用这张表来描述机器人所在房间的状态。显然，这时的情景是：机器人站在桌 A 旁并且手里拿着盒子。

表 12-1　站在桌 A 旁并且手里拿着盒子的机器人

ROBOT	
属性	值
LOCATION	A
HOLDS	BOX
EMPTYHANDED	FALSE

A	
属性	值
IS—A	TABLE

B	
属性	值
IS—A	TABLE

BOX	
属性	值
SUPPORTED—BY	ROBOT

现在假设机器人把盒子放回到桌 A 上，那么特性表就变成了表 12-2 所示的内容。注意：某此特性仅仅改变了它们的值。对机器人来说（EMPTYHANDED TRUE）替换了（EMPTYHANDED FALSE）。而对盒子来说，（SUPPORTED—BY ROBOT）变为（SUPPORTED—BY A）。另一方面，在表 12-2 中，机器人的特性表中删去了（HOLDS BOX）这一属性，而桌 A 的特性表中却增加了（SUPPORTS BOX）这一属性。

作为另外一个例子，假设机器人不是把盒子放在桌 A 上，而是把盒拿着走向桌 B，再把它放在桌 B 上。那么，特性表的内容就成了表 12-3 所示的内容。同样地，特性表的变化清楚地反映了上述情况的变化。

然而，在特性表中有两种多余的东西。首先拿机器人的特性表来说，我们有了属性 EMPTYHANDED，又可以有属性 HOLDS。如果（EMPTYHANDED FALSE）是在特性表中，那么

表 12-2　机器人在桌 B 旁；盒子在桌 B 上

ROBOT		A	
属性	值	属性	值
LOCATION	A	IS—A	TABLE
EMPTYHANDED	TRUE	SUPPORTS	BOX
BOX		B	
属性	值	属性	值
SUPPORTED—BY	A	IS—A	TABLE

表 12-3　机器人站在桌 B 旁；盒子在桌 B 上

ROBOT		A	
属性	值	属性	值
LOCATION	B	IS—A	TABLE
EMPTYHANDED	TRUE	B	
BOX		属性	值
属性	值	IS—A	TABLE
SUPPORTED—BY	B	SUPPORTS	BOX

(HOLDS BOX)（或者可能是拿着别的什么东西）也必须放入特性表中。但是，如果 (EMPTYHANDED TRUE) 在特性表中的话，那么(HOLDS BOX)就不能出现在这张特性表中。在特性表变更的过程中，必须小心，不能让这二种相互关联的属性相互冲突。

第二种情况是，盒子的特性包含了(SUPPORTED—BY B)这一属性，而同时在桌 B 的特性表中包含了 (SUPPORTS BOX) 这一属性。出现这种情况并不奇怪。因为计算机在考虑桌子时，它需要知道桌子上放着什么；而在考虑盒子时，又需要知道盒子放在什么东西上面。然而，需要注意的是：在盒子被拿开时，盒子的特性表同桌子的特性表都要产生相应的

变化。

特性表的一个优点是，每一事物的属性都同事物本身结合在一起。比方拿一个以机器人为对象的程序来说，如果它运行的话，它就用不着为了寻找那些有关机器人的语句去搜索整个庞大的数据库。它已经有了包括机器人所有特性的一张特性表。

然而，特性表的最重要的优点也许在于：它们业已成为LISP程序设计语言的一个组成部分了。在后面第14章中要讨论到的LISP语言，是在人工智能领域中应用最广泛中的一种语言。在LISP语言中，对于诸如ROBOT，BOX，A这样的符号，它都会自动给出它们的特性表。LISP系统运用其中的某些特性来定义常量、对变量赋值和命名函数。LISP用户也可以把特性放入特性表中，或者删去它们，或者改变它们的值。具有这些功能的函数，在大多数LISP语言的文本中都可以找到。

因此，由于在程序语言中已经建立起来了处理特性表的各种手段，每一个程序用不着自己去重新去搞一套，所以对于使用LISP语言的程序设计者来说，特性表用起来是非常方便的。

四、语义网络

我们可以把一个物体及其特性看作为定义一张图或一个网络。物体及值都是图的节点，而属性就是连接节点的弧的标志。比方说，（LOCATION A）在ROBOT的特性表中出现时，就把ROBOT这个节点同节点A通过弧线LOCATION连接起来了。

这一类的图就叫做语义网络。

图 12-5 给出了几种语义网络，它们对应于机器人世界的几种状态。

语义网络的一个有意义的特征是：具体状态的特征及其细节都可以存贮起来。例如，在图 12-5 中，有二条弧线通向节点 TABLE。而这些 TABLES 可以同其它节点连接起来，方法是把这些桌子的共同特征表达出来。如图 12-6 所示。

因此，在语义网络中，一些节点是用来表示一般的概念。例如桌子、椅子、房子等等。其它的用于表示具体的事物。如我坐的椅子、我的房子、我正在上面写信的桌子等等。还有一些可以用来表示一些抽象的概念，如 TRUE（真）与 FALSE（假），或者代表一些数值，如 3.14 等。所有这些事物之间的相互关系用弧线来表明。每一条弧线上的标志规定了节点之间的预定的联系。

在语义网络中，程序可以从它感兴趣的节点开始，沿着弧线到其它有关的节点，然后又从这些节点出发到更远的有关节点，如此类推。这就像人脑具备的能力一样，能够从一个思想跳到另一个相关的思想。人脑在自由联想和每日的思维中都会有这种情况产生。但是，语义网络也象个迷宫，计算机也很容易在里面迷路。就象语义网络也很容易引导计算机得出所需要的信息一样。为了使计算机能很快地得到它所需要的知识，需要提出一些更有力的组织原理。

五、框　　架

人们不必对自己的每一个行动都经过推理后才决定。并且能一天天这样生活下去。这是因为每天生活的内容大部分包含了一系列的典型情景。下面是这样一些情景的例子：

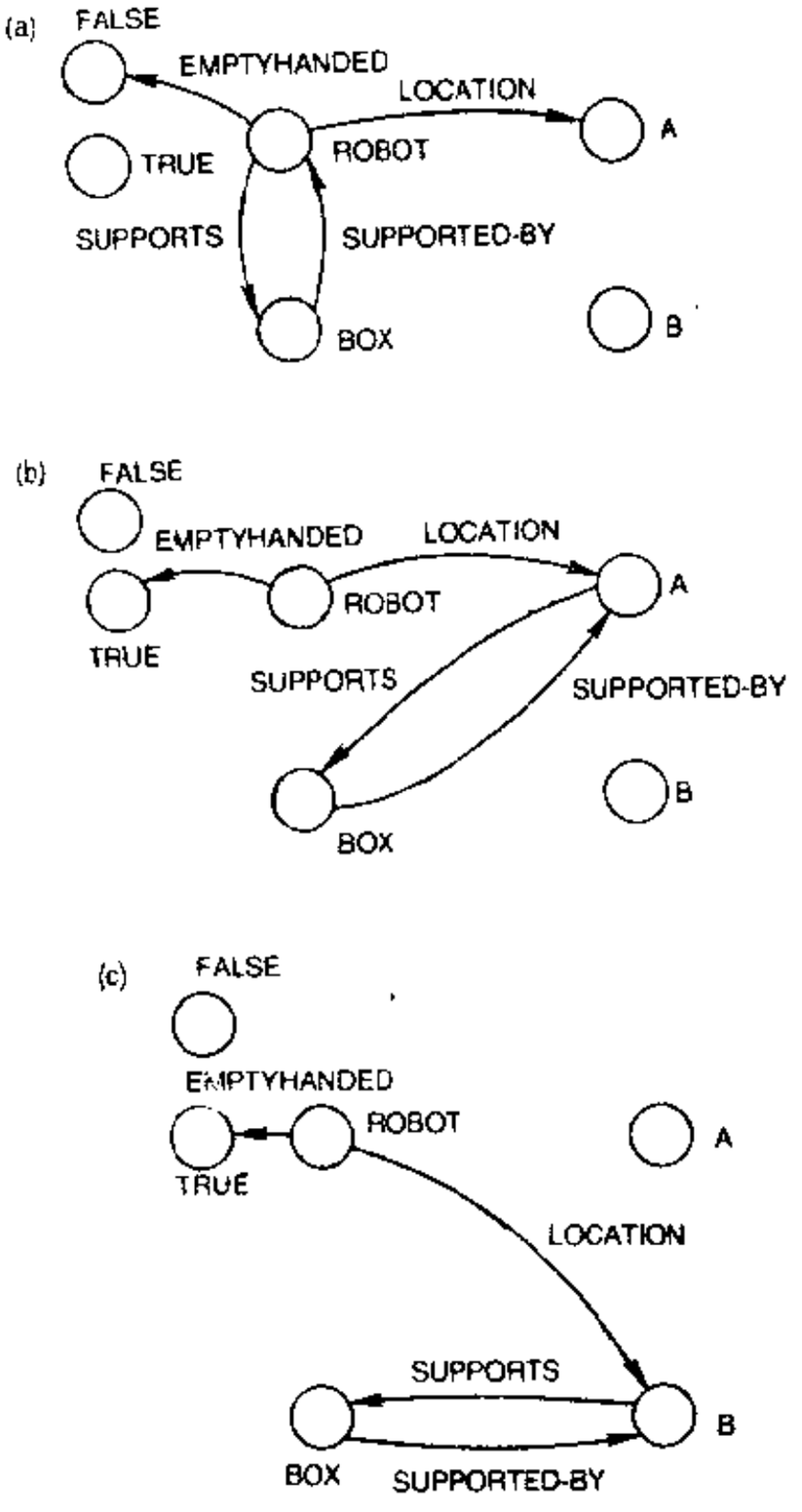


图 12-5　用语义网络表达的机器人世界的几种状态

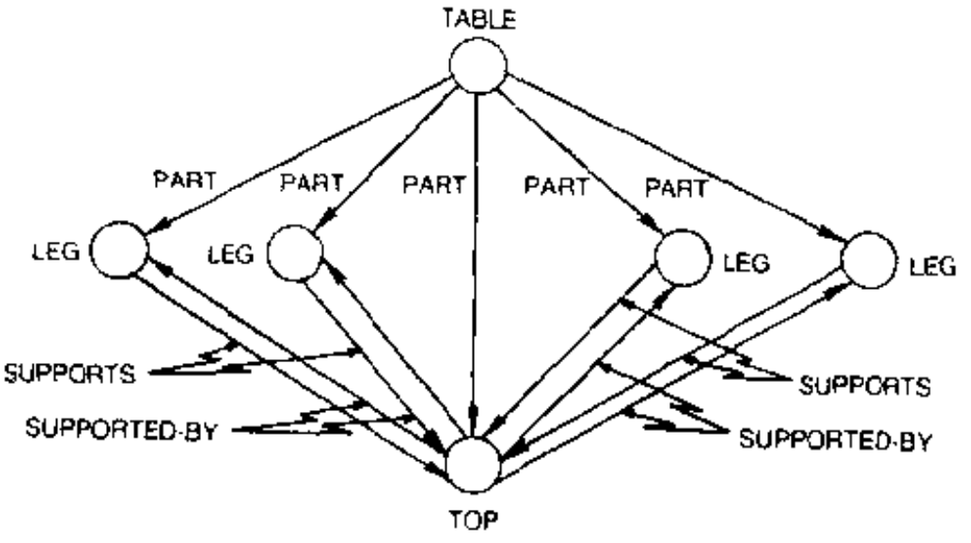


图 12-6　桌子的概念可以用语义网络来表示，这个网络表示了桌子的各个部分以及它们是如何联系起来的

1. 坐在客厅里看电视。
2. 开车去上班。
3. 在办公室里处理日常事务。
4. 上大学或高中的课。
5. 在饭馆里吃饭。
6. 饭后收拾干净桌子。

如何处理以上的情景以及许多其它的情景，人们在刚刚接触生活的时候就开始学习了。此后，人们就能不经过思考直接处理这些情景。或者，人们可以把精力集中到那些不同点或重点上，比方这里的其它人都讲了些什么，略加思索以后就能决定情景所要求采取的行动。我们称之为常识的东西而实际上就是处理日常事务的能力。

尽管我们以前没有同当前情景完全相同的经历，但是对于每一类典型的情景，我们还是能预料到会出现某些特征。

例如，在进入一个新的饭馆以前，我们可以想像得出：它有桌子、男女服务员、菜单、坐在门口的收款员，等等。当然，一旦实际上进入该饭馆后，可能会发现一些细节是与想像的不尽相同。或许是这里没有服务员，人们必须自己到柜台去领取饭菜等。在这种情况下，就要对自己预料的情况作一些小的修改，其它都照原先预料的进行处理就行。

框架是用来描述典型情景的一种数据结构。我们可以想像框架有许多空槽，这些空槽对应于我们预料在情景中能找到的各次具体内容。拿客厅为例，它的空槽就是用来表示我们预料会在客厅里找出的各种东西。如地板、墙、天花板、沙发、扶手椅、咖啡桌、电视机以及挂在墙上的画等。

当一个框架第一次被调用时，在其大部空槽中，都填有典型值，或者默认值。例如，在客厅这一框架中，电视机这一空槽的默认值可能是一架平常的彩色电视机。

自然，当我们进入一间具体的客厅时，就会发现，有一些默认值需要进行替换。或许是这间客厅只有黑白电视机而不是彩色电视机。或许是电视机没有放在客厅内，而是放在这幢房子的其它地方。或许是房子主人讨厌电视机，因此，房子里根本没有电视机。

当我们注意到这些变化时，进行默认值的改变是很容易的。但是，重要的部分是那些我们不一定非要改变的部分，是那些并非要集中精力注意的事情。例如，除非发现客厅里没有一张沙发，否则我们不见得会那么注意客厅里沙发的情况。然而，如果我们去想像客厅中的情景的话，自然而然会想到客厅中会有沙发的。

因此，一个框架，在我们对它有了切身体验之前，或者在我们对自己的经验进行分析之前，就概括了我们所能设想到

的那些事物。框架理论可以为以下几个目的服务：

1. 一个程序经常要对它当前并未体验的情景进行推理（例如，除非程序要去控制机器人，否则程序根本不会去“体验”它所要进行推理的情景）。对于这样的情景，框架会提供必要的“常识”性的设想。这些设想就是根据这些情景得到的。

2. 还有一些为例行事务而建立的框架。例如把一个盒子从一个桌子移到另一个桌子上。在这类框架中，一些空槽专门用子计划或启发式方法，或者将这些事务预以实施。一个机器人就要用到许多这类的框架。

3. 在自然语言的句子分析中，或者在对电视摄象机所摄取的景象进行分析时，计算机需要对外部世界做出某些以常识为基础的设想。在这种场合，框架也能提供所需要的设想。即使在空槽没有填入默认值的情况下，仅仅知道什么空槽必须填充也是有很大好处的。比如，一个景象分析程序，对于景象中可能存在也可能不存在的物体，可以不进行盲目地搜索，而是把主要工作放在直接填充相应框架的空槽值上就可以了。例如，在客厅中的机器人，可以从开始寻找沙发入手，然后找椅子，由此类推。

六、产生式系统

关于规划和计算逻辑，我们已经接触了几种情景。在这些情景中，模式是与数据库中的项相匹配的。一个模式通常都含有变量，并且在匹配实现的时候，这些变量就变成了实例，即用具体的值来替换这些变量。在匹配一个模式时会产生一个具体实施的计划或者是行将采取的行动。

用这种方法，可以对一个智能程序或者一个机器人提供

一种控制机构。控制是由一张产生式表规定的。每一条产生式规则都具有以下的形式：

模式→动作

计算机按照控制程序所指出的方向，搜索整个产生式表，尽量把每一条产生式规则同已知的数据库进行匹配。当匹配实现时，就"点燃"了相应的产生式规则。就是说，产生式规则中的动作开始执行了。而对数据库中点燃产生式规则的项可以赋于某种标志，使得不至于又立即用它再去点燃产生式。这就防止了一条产生式规则被同一数据库项反复引用的情况发生。

（匹配一个模式以及点燃一个产生式规则的控制程序可以看成是一个产生式系统语言的解释程序。就象 BASIC 解释程序是 BASIC 语言进行解释一样。用于编写产生式规则的系统就是定义了一种程序设计语言。）

数据库所采取的形式，很大程度上取决于所要处理的问题本身。有一种方法模仿人的行为，把数据库做成为一个短期记忆库，或者叫 STM。就象人类的短期记忆一样，STM 只记住大约七条信息。当有新的信息进入 STM 时，新的就取代了旧的。就是说，旧的信息被遗忘了。还有一种长期记忆库—LTM。通过产生式规则的动作部分可以把信息从 LTM 传送到 STM。即，某一产生式规则可以"回忆起"具体的事实，并把它送到短期记忆库中去。由感官所获得的信息可以通过自己的途径强行进入 STM。例如，当我们开车时，如果很明显看出就要撞到人的时候，就一定要中断自己原来的思路。

产生式系统并不是对所有的问题都是有效的。例如，我们如果把机器人可能有的所有产生式规则看作是在任何情景中都可能点燃的候选者，这是很成问题的。适合于一种情景

仅仅是产生式规则中的某些子集合。因此，需要有一些方法来确定那些产生式规则可以作为点燃的候选者。一个有意义的想法是：在每一具体情景所对应的框架上都加上一个标有“可用产生式规则”的空槽。当某情景被检测出来以后，其相应的框架就被调用，随后就可以得到该框架所规定的那些可用产生式规则了。

第十三章　自然语言处理

第一个受到重视并且获得资助的人工智能计划是自然语言翻译的研究。五十年代中期人们关心的一件事是，多数美国科学家和工程师不会读俄文，然而多数苏联科学家和工程师都会读英文。因为当时我们(美国)的技术力量在某些方面处于劣势，人们理所当然地担心我们的科技人员如果跟不上俄国人好象源源不断地自动产生出来的那些炫人的科学论文，就会要远远地抛在后面。

搞计算机的人们跑来支援。他们答应设计出把俄文翻译成英文的计算机程序，计划着百分之八十的翻译可以用简单的逐字替换来进行。剩下百分之二十不宜于逐字进行翻译的，就用特别设计的程序来进行处理，最后由编辑人员来润色那英语译文；这个编辑人员不一定要懂俄文。如果不嫌译文累赘的话，编辑人员甚至也可以不用。

不出所料，百分之八十的翻译都可以直接用一本字典和为数不多的有关语法功能和关系的规则来完成。不幸的是，剩下百分之二十却非常难以对付。没有人想得出好办法来翻译好这百分之二十，使得编辑人员只需要最后稍加修饰。计算机唯一能做到的最好的办法是，把有问题的词列出所有可能的英文译义，然后让一个有经验的翻译人员来选择正确的译法。

机器翻译容易错的有时是一些谚语。譬如“out of sight, out of mind”(眼不见，心不烦)，译成俄文变成了“又瞎又疯”。“The spirit is willing but the flesh is weak”(心有余而力不足)，

译成俄文变成了“酒香肉腐”，闹出了大笑话。

最后，所有的翻译计划都归于失败，资金被撤销；机器翻译得到一个坏名声，只有弃旧图新。当时的计算机虽然对翻译人员能起到一个文书的助手作用，它在技术上仍不足以直接担当这项工作。

然而，虽则机器翻译暂且被搁置起来，但自然语言处理的研究却继续进行着。无疑地，如果计算机要为公众所用，而不只是为少数专家和业余爱好者所用，计算机就必须学会用人类自己的语言进行通讯。因为大多数人大概不会都化力气去学会计算机语言，也没有道理让大家都必得学它。

研究自然语言处理的方法也和人工智能许多别的领域一样，有许多不同的途径。现在还弄不清楚哪种方法最好。本章介绍的是一些典型的方法，现在看来是比较有前途的，但不能保证这些就是最后的答案。

一、句法、语义和转移网络

自然语言的任一个句子，都可以从两种观点（句法或语义）出发去加以分析。

句法或语法，是关于句子的形式的。例如

I Seen the airplane

这个句子在句法上是错的（虽然是完全有意义的），因为英语句法认为“Seen”这个字前面必须加助词“have”，“has”或“had”。注意到句法时丝毫没有涉及到句子的意义。因为句法仅仅是说明，当一类词（例如动词的过去分词）用于一个正确的英语句子中时，必须和另一类词（例如助词）伴随在一起。

在“I seen the airplane”中，偏离句法的程度较小，所以虽然我们认出这个句子是不合语法的，却不难猜出这话的意思

是“我看见飞机”。可是，下面的句子：

Airplane I seen have the

偏离正确的英语句法太远了，它的词序太乱了，我们的眼睛简直无法找到英语句子中的主语、谓语等等，所以也就不知道它的意思。

语义学是关于句子的意义的。例如下面的句子

I drank a glass of green kindness

我 饮了 一 杯 绿色的好意

就词的通常解释来说毫无意义可言，因为“好意”既无颜色，也不能饮上一杯。可是，这个句子却是完全合乎语法的，很容易分析出主语、谓语、动词和前置词短语等等。

极言之，语义学是包罗一切的。我们感兴趣的是创造和理解具有意义的句子。一切句法规则（除了少数是学究们所拘泥，然而别人并不理会的规则以外）都是为了使我们大家用同样的方式遣词造句，以便于互相理解而存在的。一个平常的人面对这个完全不合语法的句子“Airplane I seen have the”时，也许并不会抗议这个句子不合语法，而是声言这个句子“毫无意义”。

不过，多数语言都有它的组织规则，这些规则是不依赖于词的意义的。这些规则会帮助我们区别词的模式，而不必操心词的意义。

例如，下面二者

the large red book on the table

和

a glass of green kindness

都是语言学家称之为名词短语的例子。不过，第一个是有意义的，第二个没有意义。但是两者都是名词短语而不管它们

有无意义，因为它们都是用一个方式由限定词（“a”，“the”，“this”，“that”，“these”，“those”，）、形容词、名词和前置词等构成的。我们可以认出它们都是名词短语，而不必操心它们的意义是什么。

句法可以使我们不必操心词的意义而区别词的模式，所以它是我们分析句子时非常有用的第一步。

我们现在试看一下一小部分简单的英语语法（只能是一小部分，因为整个英语语法包含的规则就有好几千条）。

我们从一个句子必须有主语、谓语的规则开始。主语和谓语描述的是句子里某一组词的功能，语言学家宁愿用“名词短语”和“动词短语”这两个术语来代替它们。这两个术语描述的是词的模式，而不是它们的功能。

在书写语法的时候，我们把“句子”、“名词短语”和“动词短语”分别用缩写 S，NP 和 VP 来表示。当我们说一个句子包含一个名词短语跟着一个动词短语的时候，我们写作：

$$S \rightarrow NP\ VP$$

这条规则告诉我们，任何一个英语句子都可以分成名词短语和动词短语这两部分。例如：

NP　　VP

The boys ran home→The boys　ran home

我们如要分析这样一个句子，显然第一件事是找出它的名词短语，找到以后，第二件事是找动词短语。

但当我们看到一个名词短语时，我们怎样识别它呢？构成名词短语的词模式是由另外一条语法规则来描述的：

$$NP \rightarrow (DET)\ (ADJ^*)\ N\ (PP^*)$$

在语法规则中，圆括号里的项是允许选择的。所以，在名词短语中，唯一必要的是名词 N。

可是，名词还可以有选择地用一个限定词（DET）（"a" "the"等）和形容词（ADJ＊）、前置词短语（PP＊）等来加以修饰。这里的＊号表示这项的个数可以是任意多个。所以，这条规则告诉我们，只能用一个限定词，但可以用任意多个形容词和前置词短语。

凡是出现的项，必须按规则规定好的次序出现在句子里面，所以，如果有限定词的话，一定出现在最前面，然后是形容词、然后是名词，最后才是前置词短语。下面的例子表示怎样用这条规则来分析一个名词短语：

	DET	ADJ	N	PP
The red book on the table→	the	red	book	on the table

假若我们要分析一个英语名词短语，我们首先就要找它的限定词，如果没有，再找形容词，如果也没有，再找名词。如果找不到名词，那末这一组词就不是一个名词短语。

其他词组也可以类似地加以定义。例如：

VP——→VTRAN　NP

这条规则是说，一个动词短语包含一个及物动词(VTRAN)跟着一个名词短语(NP)，在英语中还有好几种动词短语，不过这里我们只考虑这一种。

前置词短语由一个前置词(PREP)和一个名词短语(NP)组成。

PP——→PREP　NP

注意到名词短语是前置词短语中的一个成分，前置词短语是名词短语中的一个成分。所以前置词短语和名词短语是互相以对方来定义的。这种情况导致嵌套结构。例如

the book on the shelf over the door to the kitchen 是一个NP，它可以分析为：

DET N PP

the book on the shelf over the door to the kitchen

而 PP 又可分析为：

PREP NP

on the shelf over the door to the kitchen

这个新的 NP 又可再分解成 DET,N 和 PP：

DET N PP

the shelf over the door to the kitchen

新的 PP 还可以进一步分下去，依此类推。

这种语法分析法可以用图 13-1 中所表示的转移网络来实现。这个网络描述了一系列机器，每个机器是一个图形。机器的各个状态是带标号的圆圈。各个状态之间的连线表示从一个状态到另一状态之间的可能转移。

我们可以把每个图形所描述的机器看成是识别特定的词模式的专家。这样 S 机器就是识别句子的专家，NP 机是识别名词短语的专家等等。每部机器还要请另外一个或几个专家来帮忙。NP 机器要请 PP 机器来识别前置词短语；PP 机器要请 NP 机器来识别名词短语，如此等等。

比方说，要识别一个句子，我们就要开动 S 机器先到达状态 1。如要到达状态 2，我们得找到一个名词短语。于是 S 机器调请 NP 机器来帮忙。当 NP 机器回答说它找到了一个名词短语时，S 机器便进入状态 2，这里它发现需要的是动词短语，于是 S 机器又调用 VP 机器，当 VP 机器成功地找到了动词短语后，S 机器便进入了状态 3。状态 3 的双圈表示 S 机器的工作到此已经完成；于是它可以向我们报告说，句子已经找到了。

NP 机器要比 S 机器稍微复杂一些，NP（名词短语）可以

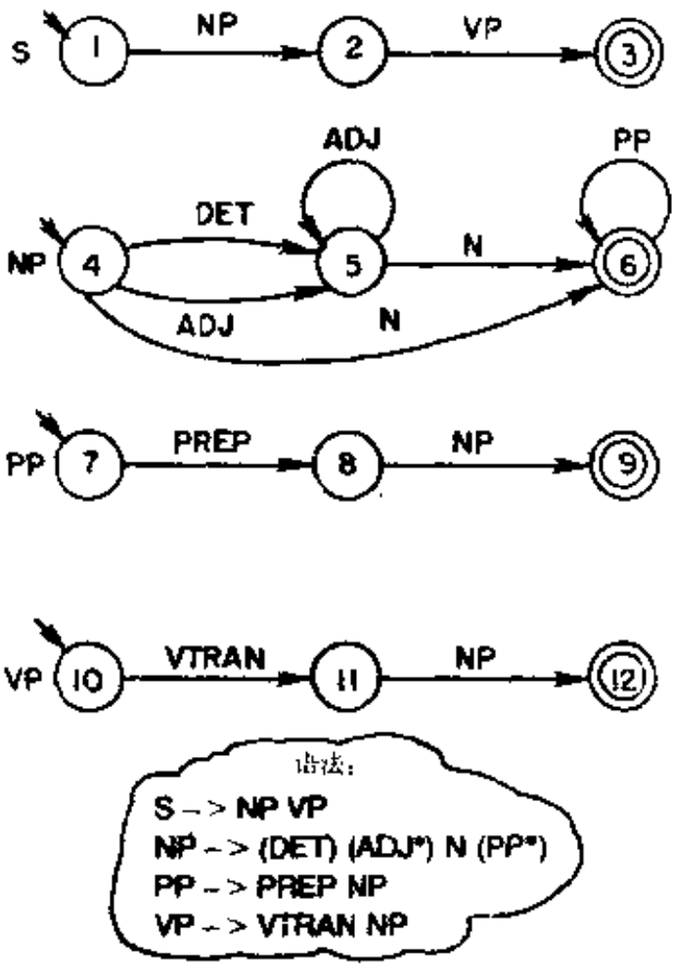


图 13-1　英语一个小子集的转移网络，每个图是一台查找词模式的机器。每台机器又可调用别的机器来查找需要的模式

译注：图中 S—句子　NP—名词短语　VP—动词短语　ADJ—形容词　DET—区别词　N—名词　PP—前置词短语　PREP—前置词　VTRAN—及物动词　NP—名词短语

从一个 DET（限定词）一个 ADJ（形容词）或一个 N（名词）开始。所以有三条转移路线从状态 4 出发。它跟着的是哪一个和下一状态是什么？依赖于找到的究竟是什么？是一个 DET（限定词），ADJ（形容词），还是 N（名词）？

因为一个名词短语可以包含任意多个形容词和前置词短

语，所以 NP 机器存在着标有 ADJ 和 PP 的回路。ADJ 回路可以让机器不断返回到状态 5 来寻找更多的 ADJ，直到它碰到一个 N 为止。然后它就开始找 PP，每次返回到状态 6 便再找一次 PP。当 PP 都已经找完，NP 机器的工作便也做完了，它便送回给调用它的机器以一个控制信号。

当然事实上我们并不用为了识别每一个词模式分别建造一个机器，所以有这些机器都不过是我们存储在计算机记忆里的表格而已。例如，下面这张表格就代表 NP 机器：

状态	词模式	下一状态
4	DET	5
4	ADJ	5
4	N	6
5	ADJ	5
5	N	6
6	PP	6
6		返回(结束)

最后一款仅当“最后再来一个”不成时—也就是再没有 PP 可找的时候才用到。

我们可以写出一个计算机程序来模拟机器的行为，通过查找表格，找出当我们在所分析的句子中寻找某一特定词模式的每一状态下，每部机器要作什么。正如我们看到过的产生式系统的控制程序的情形一样，这种程序也不过是解释程序的一种形式，而描述机器的表格也不过是程序语言的一种形式而已。

当构成转移网络的机器分析一个句子的时候，它们能够收集它们所识别的词模式的信息。我们可以想到和每一个词模式相联系的一个框架(frame)(在第十二章我们已描述过框

架)。当识别该模式的机器扫过该句子时，它收集着信息，并且用这些信息来填充框架中的空槽(slots)。例如，在识别一个名词短语时，NP 机器就填充着限定词、形容词、名词，以及前置词短语等的空槽。它还可能注意到名词短语是单数还是复数，它指的是否人，如果是人属于什么性等等。于是，名词短语“the tall skinny boy in the yard”的框架填充可能象下面的样子：

空槽	值	空槽	值
DET	the	NUMBER(数)	singular
ADJs	tall, skinny	PERSON(人)	yes
N	boy	GENDER(性)	male
PPs	in the yard		

收集这信息的一个简单的办法是安排一个特定的子程序以供每一转移来调用。(往往更方便的办法是在每一转移之前有一子程序调用，在每一转移之后又有一个子程序调用)。子程序可以收集信息并把它填入空槽。

子程序调用是很容易做到的，子程序地址直接就加在描述转移网络的表格里面。(在前面举例说明过的表格形式中，于是，每一行就对应一次转移，在转移前后要调用的子程序地址，就可以加在表格对应的行里面)。解释程序要修改后用来调用子程序，子程序的地址应在适当的时候从表中查到。具有调用子程序的转移网络就叫增广转移网络(augmented transition network)，或缩写为 ATN。

二、格 语 法

我们可以用 ATN 来识别词模式。对每一个词模式，我们可以填充一个框架的空槽，给模式以一个相当完善的描述。描述各种词模式的画面可以用来作为进一步处理的基础。

我们能不能用类似的做法，把句子当做一个整体来对待呢？

有一个办法叫做格语法(case grammar)。“格”(case)这个词来自传统语法，如主格和宾格，它们表示一个词或短语在句子中的功能。这种传统语法的格叫做表面的格，因为它们只涉及到句子的外观，而不涉及它所包藏的概念。在格语法中所用的是深部的格，它涉及到句子在通讯中包藏的思想和观念。

比方说，考虑一个描述行动的句子。它所涉及的深部的格可能有：

1. OBJECT(受事)，指行为所及的对象。

2. AGENT(施事)，指使行为产生的事物。

3. CO—AGENT(共施事)，施事可能借助以发生该行为，帮助者叫共施事。

4. INSTRUMENT(工具)，施事或与施事实现行为所使用的任何工具。

5. SOURSE(来源)。

6. DESTINATION(去处)，行为往往包括移动某物从一处到另一处。来源和去处便是指移出和移往的地方。

7. TRAJECTORY(轨迹)，从来源到去向所经的路径。

8. CONVEYANCE(载具)。将物放在里面从来源移往去处。

我们可以把一个框架和每一个句子联系起来。框架的空槽对应于各种深部的格。分析一个句子就包括着把句子框架的空槽用一组组的词填充起来。这些词组就是 ATN 找到的名词短语。还有动词的空槽，它是描述行为的。

例如，下面的句子

John gave the book to Sally

它的框架可能是这样：

VERB(动词)：	gave(给)
OBJECT(受事)：	the book(书)
AGENT(施事)：	John (约翰)
SOURCE(来源)：	John (约翰)
DESTINATION(去处)：	Sally (莎莉)

再举一个例子：

Bill built the kit using a screwdriver and a soldering iron

这个句子的框架可能是这样：

VERB(动词)：	built(造)
OBJECT(受事)：	the kit(工具)
AGENT(施事)：	Bill(比尔)
INSTRUMENT(工具)：	screwdriver(改锥)
	soldering iron(烙铁)

我们不打算详细讨论如何填充句子框架空槽的问题。这里只提到一下某些考虑，它们是：

1. 动词的含意可能暗示和它共用的词的格。例如“give(给)”，暗示转移，所以自然会有 SOURCE（来源）和 DESTINATION(去处)。我们还可以推测到 AGENT(施事)和

SOURCE(来源)是同一个。

2. 前置词往往提示跟随的名词短语可能属于什么格，例如“with”往往提示“共施事”(CO－AGENT)，如：

John fixed the computer with Jack

(约翰同杰克修理计算机)

或者提示“工具”(INSTRUMENT)，如：

John fixed the computer with soldering iron

(约翰用烙铁修理计算机)

3. 词组在句子里的位置对它们的格有重要的意义，这在英语中尤其是这样，因为英语比某些别的语言更着重词序。例如，在主动态的句子中打头的名词短语(传统的主格)大都是 AGENT(施事)，象下面的句子就是：

The boy hit the ball(男孩击球)

这样那样一些考虑对各种名词短语都加上一些约束。每种约束都相当弱——它可能暗示不只一种格。但当所有的约束加在一起时，它们就往往把每个名词短语的格紧紧地约束住了。

三、概念依赖理论

概念依赖理论(conceptual dependency theory)进行句子分析时多走了一步，它还分析句子中蕴含着的因果关系。它和格语法在许多方面很相似；它也从分析的句子中抽取出值来填充框架的空槽。许多空槽的名称我们已经从格语法中熟悉过，如 OBJECT，SOURCE，DESTINATION 等等。但格语法中的 AGENT(施事)在概念依赖理论中却叫做 ACTOR(行动者)。

概念依赖理论不但涉及到行动者，而且涉及到行动本身，

它企图把一个复杂的行动分解为一个或多个基本行动，如：iNGEST（摄取），TRANSFER-POSSESSION（转让），EXPEL（排出），MOVE-OBJECT（移物），HEAR（听），SEE（视），等等。

举例，如下面的句子

Jack gave the ball to Jane

（杰克给球与詹尼）

可以分析如下：

ACT（行动）：　　TRANS-POSSESSION（转让）

ACTOR（行动者）：　　Jack（杰克）

SOURCE（来源）：　　Jack（杰克）

DESTINATION（去处）：　　Jane（詹尼）

而句子

Jack took the ball from Jane

（杰克从詹尼那里拿走球）

则可以分析如下：

ACT　（行为）：　　TRANSFER-POSSESSION（转让）

ACTOR（行动者）：　　Jack（杰克）

SOURCE（来源）：　　Jane（詹尼）

DESTINATION（去处）：　　Jack（杰克）

动词 give（给）和 take（拿）都可用同一个基本动作 TRANSFER-POSSESSION（转让）来表示，不过它们对 SOURCE（来源）和 DESTINATION（去向）则填充以不同的内容。

问题的下一步是分析句群，并从中提取有关情境、过程、方案、情节以及其他方面的描述。这里框架方法似乎是很有前途的。当程序逐句地读入一段文字以后，它就把这段文字所描述的情境的框架的空槽一一填满。读入的文字愈多，填满的空槽也愈多。已经填满的空槽又可以用来帮助分析现在

遇到的句子。比方说，用来解决代词的参照物问题，或者用来解释那些不能脱离上下文的有岐义的句子。不过在实际上进行这样一些分析，有许多细节问题还有待进一步完善。

第十四章　LISP 语言

人工智能的程序几乎可以用任何程序语言来写。前面各章所谈到的各种方法都可以用 BASIC 或 FORTRAN 语言写成的程序来说明。然而当快速执行与有效利用存储更为重要时，例如在下国际象棋的程序中，也许还得用汇编语言。

但是，有些语言是专门为人工智能的程序而设计的，其中应用最广泛的是 LISP，它可用作表格处理器。LISP 已经在许多类型不同的计算机上付诸实现，其中包括小型计算机。所以在微型计算机上的应用也不会出现新问题。

一、原 子 和 表

LISP 处理两种对象：原子(atom)和表(list)。(LISP 的原子与谓词逻辑的原子无关。事实上，谓词逻辑中的一个“原子”在 LISP 中通常是以表来表示的。)

(一)原子

原子是用以命名对象的符号。已命名的对象可以是程序中的对象，例如变量和函数；或者是用程序处理的客观世界里的各种对象，例如机器人，桌子、箱子、房间、窗户等等。下面是原子的一些例子：

ROBOT	ALCOVE
BOX	PLUS
A	DIFFERENCE
B	SQRT

每一个原子都有一个特性表，它可以用来描述已命名的

对象并且规定该原子与其它对象的关系。

数字在 LISP 中也被看作是原子。因此

25	3.14
−100	−25
2.5	−4.5

都是原子。但是由于数字不能命名任何对象，也不能有与其数值不同的特性，所以数字是原子的非常特殊的情况。正如在其它程序语言中一样，数字只在算术运算中用来表达数值。

(二)表

表由括在括号内的一系列成员组成，并且相互之间用空格隔开。表的成员可以是原子：

```
(ROBOT  BOX  A  B  ALCOVE)
(COLOR  RED)
(LENGTH  25)
(A  B  C  D  E  F  G)
(HOLDS  ROBOT  BOX)
```

表还可以包含其它表作为其成员：

```
(A  (B  C)  D  (E  F  G))
((LENGTH  25)  (WIDTH  5)  (HEIGHT  100))
((AT  ROBOT  ALCOVE)
  (ON  BOX  A)
  (EMPTYHANDED  ROBOT)
  (TABLE  A)
  (TABLE  B))
((3(2  5)(3  4  9))
```

因此，第一个例子的成员为 A，(B C)，D 和 (E F G)。第一个和第三个成员是原子，第二个和最后一个成员是表。如

例 3 所示，表的成员可以分成几行来写。当各元素本身为表时，常常必须分行写。

当然，作为另一个表的成员的表仍然可以有其它的表作为其成员，而那些其它的表还可以有另外的表作为其成员，以此类推。

```
(((A B C)D)E((F G)H I))
```

这样复杂的表，当其成员分成几行来写时，就比较清楚了。

```
(((A B C)D)
    E
 ((F G)H I))
```

二、用表来表示信息结构

让我们来看一下前面各章所使用的一些信息结构，如何用表和原子来表示。

(一)树

我们可以把树看作是由根与一个或多个子树共同组成，如图 14-1 所示。每个子树也是一个树，其根就是原树的一个子代。

我们可以用一个表来表示一个树，其第一个成员为树的根，而其余成员是它的子树：

```
(根  子树－1  子树－2…… 子树－n)
```

只由一个根组成的子树称之为叶，它仅仅由命名它的原子来表示。

因此，在图 14-2 中，叶节点用

```
E F G H I J K
```

表示，根的子树用

```
(B E F G)(C H I)(D J K L)
```

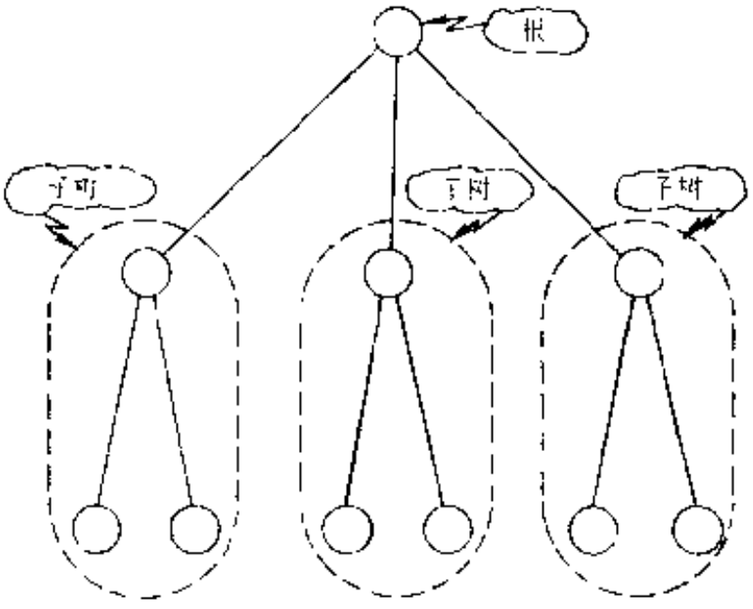


图 14.1 树可以看作是由根与一个或多个子树共同组成的

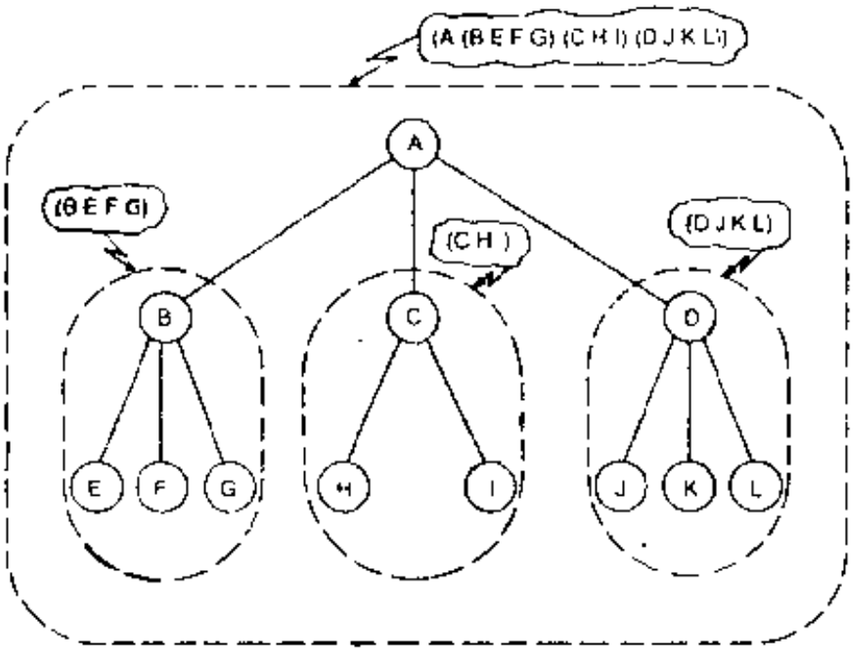


图 14.2 树能够用一个表来表示，其第一个元素表示根，其余的元素表示子树。如果子树不是叶，就用同类型的表表示

表示，并且整个树用

(A(B E F G)，(C H I) (D J K L))

表示。

树的每一个节点用一个原子命名。原子的特性表可以包含或指明我们所希望的与节点有关的全部信息（例如博弈的位置）。

(二)谓词逻辑表达式

谓词逻辑的原子用表来表示，表由谓词与其后的诸变元所组成：

(ON BOX A)　　表示　　ON(BOX,A)

(AT ROBOT ALCOVE)表示　　AT(ROBOT,ALCOVE)

(TABLE A)　　表示　　TABLE(A)

(HOLDS ROBOT BOX)表示　　HOLDS(ROBOT,BOX)

像下面那样的表达式

ON(BOX,A) or ON(BOX,B)

是用前缀形来表示的。操作符 or 首先写出，接着写操作符所连结的子表达式：

(OR(ON BOX A)(ON BOX B))

同样地，

(AND(ON BOX A)(AT ROBOT ALCOVE))

表示　ON(BOX,A)**and** AT(ROBOT ALCOVE)

并且　(NOT(ON BOX A))

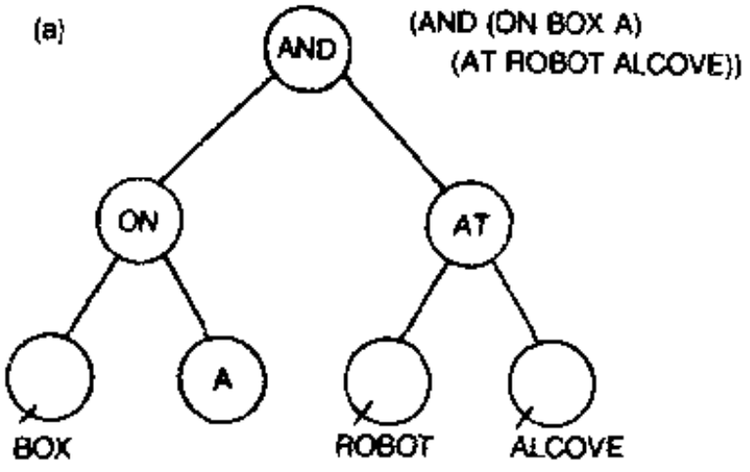
表示　**not**　ON(BOX,A)

值得注意的是谓词逻辑表达式的表示法与树表示法相同。如图 14-3a 所示，表达式

(AND(ON BOX A)(AT ROBOT ALCOVE))

与根为 AND 的树相同。根的子代是 ON 和 AT。ON 的子代是 BOX 和 A，而 AT 的子代是 ROBOT 和 ALCOVE。注意树的叶节点都是单个的，例如 BOX，A，ROBOT，和 ALCOVE；而非

叶节点是逻辑算子如 AND，或谓词例如 ON 和 AT。



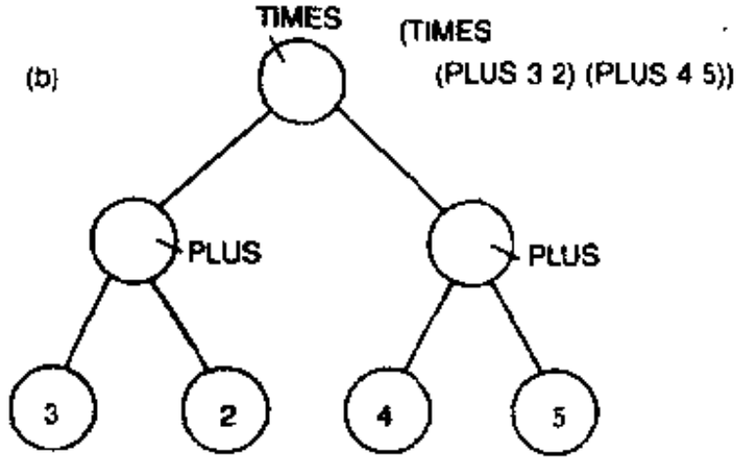


图 14-3　逻辑表达式和算术表达式都有树结构，并且能用和树相同的方式用表的形式来表达

算术表达式也以树的形式表示。因而

$(3+2)\times(4+5)$

表示为(TIMES(PLUS 3 2)(PLUS 4 5))

这也相应于树表示法，如图 14-3b 所示。

因为一个表达式可以被表示成一个树，所以处理树的程序可用于处理表达式。

(三)图

对于图的每一个节点，我们必须详细标明哪些弧线从该

节点出发和每条弧线的终止节点。

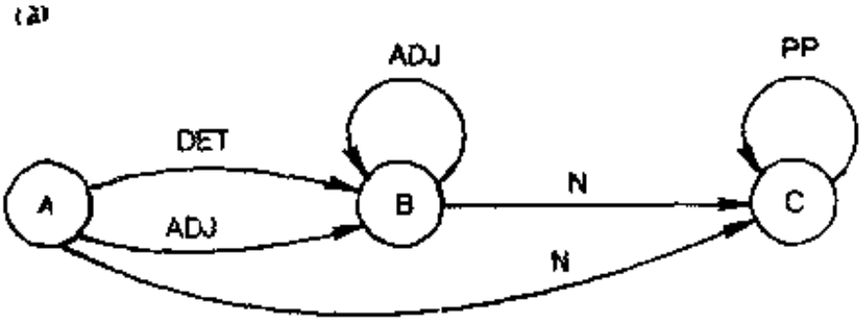
图 14-4 表明用表来表 1 图有两种方法。第一种方法是一个有

(节点　弧线　节点)

形式的三元表，每个括号表示该弧线从第一个节点连到第二个节点。第二种方法是一个有

(节点(弧线 节点)(弧线 节点)(弧线 节点)……)

形式的多层表。这个表的第一项表示一个节点，其余的各项对应于由该节点出发的各条弧线，每条弧线的弧标识符与弧终止节点都是已知的。



(b) ((A DET B)
(A ADJ B)
(A N C)
B ADJ B)
(B N C)
(C PP C))

(c) ((A (DET B) (ADJ B) (N C))
(B (ADJ B) (N C))
(C (PP C)))

图 14-4　用表来表示图有两种方法。例图是十三章中用于"名词短语机"的状态图的简化

(四)其它方法

上述表示树、表达式和图的方法已获得了广泛的应用，但它们并不是神圣不可改变的。其它方法也是可能的，尤其是当这些方法更适合于解决某一类特殊问题时更应被采用。

三、LISP 程序设计的原理

(一)函数和变元

LISP 系统是一个函数求值的机器。用户打入一个函数和它的变元，LISP 给出函数作用于变元的结果。

函数及其变元以前缀形式写出，函数写在第一项，接着写变元。因此，PLUS 是 LISP 的加函数，并且

(PLUS 3 5)

表示 3 与 5 之和。如果我们打入(PLUS 3 5)，LISP 就给出回答 8。

为了说明用户—LISP 的对话，我们把用户输入表示在一行开始的最左边，而 LISP 的回答响应退后两格。于是，上面刚提到的对话形式就可以写成：

```
(PLUS 3 5)                    用户
    8                         LISP
```

下面的对话将介绍一些其它函数：

```
(DIFFERENCE 9 4)
    5
(MINUS 2)
    -2
(TIMES 4 3)
    12
(QUOTIENT 3 2)
    1.5
(SQRT 9)
    3
```

一个函数的变元本身也可以是表达式。LISP 在求它们

的函数值之前，首先要求各变元的值：

```
(PLUS(TIMES  3 4)(DIFFERENCE  9 5))
    16
(TIMES 3(PLUS 2(TIMES 4 5)))
    66
```

第一个表达式等价于

$$(3\times 4)+(9-5)$$

而第二个表达式等价于

$$3\times(2+(4\times 5))$$

(二)表函数

在 LISP 里最感兴趣的当然不是那些算术函数，而是那些处理表格的函数。在说明它们之前，我们必须来看看 LISP 是怎样解决大多数程序中所出现的问题。

让我们暂时从 LISP 转到 BASIC，因为 BASIC 是大家比较熟悉的。考虑两个 BASIC 语句：

```
10  PRINT  3+5
20  PRINT  "3+5"
```

在第一个语句里，计算机把 3+5 作为一个要求值的表达式来处理，并打印计算结果 8。而在第二个语句里，引号标志 3+5 是一个字符串，不是一个表达式，所以计算机只打印出 3+5。

LISP 中也有同样的问题，表

```
(PLUS 3 5)
```

可能是一个要求值的表达式，或者只是一个包含元素 PLUS、3 和 5 的表。如果是后面一种解释，我们必须在表前面加上单引号：

```
'(PLUS 3 5)
```

下面的对话说明了这种区别：

```
(PLUS 3 5)
    8
'(PLUS 3 5)
    (PLUS 3 5)
(DIFFERENCE 5 9)
          -4
'(DIFFERENCE 5 9)
    (DIFFERENCE 5 9)
'(A B C)
    (A B C)
(A B C)
    ERROR
```

得出错误是因为当给 LISP 输入(A B C)时，它把 A 作为一个函数来解释，而把 B 和 C 当作 A 的变元。因而 LISP 要从 A 的特性表中找出它的函数定义，如果用户还没有对 A 下函数的定义，那么就无法找到其定义，所以 LISP 就指出错误。

不幸的是，由于历史原因下面三个表处理函数采用了没有意义的名字，人们必须硬记住它们的意义。

CAR(读作〔ka:〕)得到表的第一个成员：

```
(CAR '(A B C D))
    A
(CAR '(AT ROBOT ALCOVE))
    AT
(CAR '((A B)(C D)E))
    (A B)
```

CDR（读作〔kuder〕）得到除去表第一个成员之后所剩下

的部分：

```
(CDR '(A B C D))
    (B C D)
(CDR '(AT ROBOT ALCOVE))
    (ROBOT ALCOVE)
(CDR '((A B)(C D)E))
    ((C D)E)
```

CONS(读作[kons])在表内加上一个第一项新成员：

```
(CONS 'X '(A B C))
    (X A B C)
(CONS 'AT '(ROBOT ALCOVE))
    (AT ROBOT ALCOVE)
(CONS '(X Y) '(A B))
    ((X Y)A B)
```

要注意的是有些原子，例如 X 和 AT 也必须加上引号，如果它们仅仅代表它们自己的话。下面我们就会看到被引号引起来的原子表示什么。

APPEND 是名实比较相符的一个表处理函数，它把两个表连结在一起：

```
(APPEND '(A B) '(X Y))
    (A B X Y)
(APPEND '(1 2 3) '(4 5 6))
    (1 2 3 4 5 6)
```

像算术函数一样，表处理函数也可以被组合起来形成表达式：

```
(CONS(CAR '(X Y Z))(CDR '(A B C)))
    (X Z B C)
```

```
(APPEND(CDR '(X Y Z))(CDR '(A B C)))
    (Y Z B C)
```

(三)变量

像大多数其它程序语言一样，LISP 可以对变量赋值。一个变量不论何时出现在表达式中，都用它的值来代替。LISP 用原子作为变量。

函数 SETQ 给变量赋值。

```
(SETQ X 5)
    5
(SETQ Y 10)
    10
```

现在 X 和 Y 已经分别被赋值为 5 和 10，当它们在表达式中出现的时候，就用它们的值代入：

```
X
    5
(PLUS X Y)
    15
```

当一个原子前面用引号时，它就代表它本身；当一个原子未用引号时，它就代表它的值。

```
Y
    10
 Y
    Y
```

如果原子是带引号的表的一部分，它就代表它本身：

```
(TIMES X Y)
    50
'(TIMES X Y)
```

(TIMES X Y)

变量的值，当需要的时候可以随时改变：

```
(SETQ X 25)
    25
X
    25
(SETQ X '(A B C D))
    (A B C D)
X
    (A B C D)
(CDR X)
    (B C D)
```

SETQ 中的 Q 告诉我们第一个变元是自动地加引号的。因此，在(SETQ X 25)中，X 代表它本身，尽管我们没在 X 前面加引号(并且必须不加)。

SETQ 给出的函数值总是等于赋予变量的值，但是给变量赋值所产生的副作用要比所得到的函数值重要得多。通常，执行 SETQ 函数纯粹是为了它的副作用，而忽略函数所得的值。

(四)定义新函数

人们是用定义新函数的方法来编 LISP 程序的。这些用户定义的函数也能够与它们的变元一起打在屏幕上，正如我们前面讨论过的固有函数一样。

假定我们想定义一个函数 SQUARE，它会产生一个数的平方，也即是该数的自乘。我们希望 SQUARE 有如下性能：

```
(SQUARE 2)
    4
```

```
(SQUARE 3)
    9
(SQUARE 4)
    16
```

我们可以定义 SQUARE 如下：

```
(DEFINE(SQUARE X)(TIMES X X))
    SQUARE
```

该函数的定义为带有三个成员的表。它们是：

• DEFINE. 这是个 LISP 函数，它负责接受定义并且将它们存入存储器。

• (SQUARE X). 表中这个成员是所要定义的函数的名字 SQUARE 以及虚变量 X，后者处于变元的位置上。当函数被调用时，变元的值将暂时赋予 X。若函数变元多于一个，那么就需要相应数量的虚变量。

• (TIMES X X). 这个表达式计算所要求的函数值，变元所用的值已被赋予虚变量 X。

因此，如果我们打入

```
(SQUARE 2)
```

则 X 的值被设定为 2。所以(TIMES X X)等于(TIMES 2 2)，其值为 4，也就是函数所得到的值。

DEFINE 的变元 (SQUARE X) 和 (TIMES X X) 是自动加引号的，因而它们就代表它们自己。当定义函数时，不进行求值；只有当函数被调用时才求值。

由 DEFINE 得到的值就是所要定义函数的名字。此值我们并不感兴趣，所以通常不必为表示它而费心。DEFINE 是 LISP 用以执行其副作用的另一个函数。

让我们来列举另外几个函数定义

```
(DEFINE(CUBEX)(TIMES X(TIMES X X)))
```

函数 CUBE 得其变量的立方：

```
(CUBE 2)
    8
(CUBE 3)
    27
```

当然，我们也能够定义新的表处理函数：

```
(DEFINE(SECOND X)(CAR(CDR X)))
```

SECOND 得到表的第二个成员：

```
(SECOND '(A B C D))
    B
(SECOND '((A B)(C D)E)
    (C D)
```

下面是一个带有两个变元的函数：

```
(DEFINE(JOIN X Y)(APPEND(CDR X)(CDR Y)))
```

JOIN 从其两个变元的每一个之中移去第一个成员，并且把两个表的剩余部分连在一起：

```
(JOIN '(A B C) '(X Y Z))
    (B C Y Z)
(JOIN '(1 2 3 4) '(5 6 7 8))
    (2 3 4 6 7 8)
```

较复杂的函数定义通常不得不写成几行：

```
(DEFINE  (JOIN X Y)
(APPEND  (CDR X)
         (CDR Y)))
```

我们常常将语句分开，使得每个函数变元写在另一个变元之下，如上面表示的那样。

(五)谓词

绝大多数程序语言都有条件语句，以数据是否满足条件来决定程序的执行。现在，我们希望看到在 LISP 中这一点是怎样实现的。

首先，在 LISP 中真(true)和假(false)是分别用原子 T 与 NIL 来表示的。用 NIL 表示假，这是 LISP 的另一个令人讨厌但又不得不记住的古怪的表示方法。

T 和 NIL 就是它们自己本身的值；因此根本不需要加引号：

```
T
    T
NIL
    NIL
```

谓词是其值为 T 或 NIL 的函数。每个谓词相应于对其变元，可满足或不可满足的某一条件。若对其变元、条件满足，则谓词得 T，若条件不满足，则得 NIL。

若谓词 EQUAL（等于）的变元是相同的，则得 T。

```
(EQUAL 5 5)
    T
(EQUAL 5 7)
    NIL
(EQUAL 'A 'B)
    NIL
(EQUAL 'C 'C)
    T
(EQUAL '(A B C) '(A B C))
    T
```

```
(EQUAL '(A B C) '(A B C D))
    NIL
```

GREATERP(大于)和 LESSP(小于)比较数字的值

```
(GREATERP 5 3)
    T
(GREATERP 3 5)
    NIL
(LESSP 3 5)
    T
(LESSP 5 3)
    NIL
```

ZEROP 识别数字零。

```
(ZEROP 0)
    T
(ZEROP 25)
    NIL
```

ATOM 识别原子,而 NUMBERP 识别数字。

```
(ATOM 25)
    T
(NUMBERP 25)
    T
(ATOM 'A)
    T
(NUMBERP 'A)
    NIL
(ATOM '(A B C))
    NIL
```

```
(NUMBERP '(A B C))
    NIL
```

许多 LISP 谓词的名字以 P 结尾，但是很遗憾有些例外，例如 EQUAL 和 ATOM 就限制了这个惯例的使用。

(六)COND 函数

COND 函数是 LISP 的条件语句，它有下列形式：

```
(COND(条件 1   表达式 1)
     (条件 2   表达式 2)
               ⋮
               ⋮
     (条件 n   表达式 n)
```

每个条件是一个表达式，其正常求值为 T 或 NIL，然而不是 NIL 的任何值都可以认为是“真”。COND 对每个条件依次求值。当找到第一个其值不为 NIL 的条件时，就求相应的表达式的值，并且所得到的值即为整个 COND 表达式的值。若所有条件的值均为 NIL，则 COND 的值也得 NIL。

例如我们定义一个算术函数 SGN，若其变元大于零，则它的值为 1；若其变元等于零，则它的值为零；若其变元小于零，则它的值为－1。

```
(DEFINE(SGN X)
    (COND((GREATERP X 0)1)
          ((ZEROP X)0)
          ((LESSP X 0)－1)))
```

若 X 的值大于零，则第一个条件(GREATERP X 0)的值为 T，那么 COND 的值即为 1。若 X 的值等于零，则(GREATERP X 0)为 NIL，而(ZEROP X 0)为 T，那么 COND 的值为零。若 X 的值小于零，则(GREATERP X 0)与(ZEROP

X)二者的值均为NIL，而(LESSP X 0)为T，那么COND的值为－1。

如果X的值既不大于零也不等于零，那么它必然小于零，所以实际上不必对(LESSP X 0)求值。如果(GREATERP X 0)和(ZEROP X)二者均为NIL，那么(LESSP X 0)必然是T。这个观察使我们可以稍微简化一下SGN：

```
(DEFINE(SGN X)
    (COND((GREATERP X 0)1)
          ((ZEROP X)0)
          (T －1)))
```

第三个条件本身恰好是T，所以总是“真”。无论什么时候当(GREATERP X 0)和(ZEROP X)二者均为NIL时，第三个条件总为T，所以SGN的值为－1。然而，若(GREATERP X 0)或(ZEROP X)出现T，那么(T －1)就不起作用，因为第一个为T的条件决定了COND所得的值。

COND中最后一个条件用T的办法，在LISP程序编制中广泛采用。

无论我们采用哪一个定义，SGN都会得到所期望的值：

```
(SGN 3)
    1
(SGN 0)
    0
(SGN －25)
    －1
```

(七)递归函数

一种特别简捷的定义函数的方法是部分地依据函数本身来定义函数。这种定义称之为递归定义。

例如，让我们来定义一个函数 LENGTH，它的值是表中元素的个数：

```
(LENGTH '(X Y Z))
    3
(LENGTH '(A B C D E F))
    6
```

要计数的元素本身也可以是表：

```
(LENGTH '((A B C)(X Y)))
    2
(LENGTH '((A B)(C D)(E F)))
    3
```

原子不包含元素，所以它的长度为 0：

```
(LENGTH 'A)
    0
(LENGTH 'B)
    0
```

空表()没有任何元素，它的长度当然为 0：

```
(LENGTH '())
    0
```

LISP 还有另一种特性，在计算机内部空表()同样用原子 NIL 来表示，而 NIL 我们已用来表示“假”。就计算机而言，()与 NIL 是一回事：

```
(EQUAL '( ) NIL)
    T
```

因为 NIL 是一个原子，所以空表也是个原子：

```
(ATOM '( ))
    T
```

若我们对仅有一个元素的表取 CDR，则得到空表：

```
(CDR '(A))
    NIL
    (EQUAL(CDR '(A)) '( ))
    T
(ATOM(CDR '(A)))
    T
```

现在，我们就来对函数 LENGTH 给出递归定义：

```
(DEFINE(LENGTH X)
    (COND((ATOM X)0)
          (T(PLUS 1(LENGTH(CDR X))))))
```

仔细看一下 COND，我们看到 LENGTH 的定义分为两种情况。对于第一种情况 (ATOM X) 是 T，对于第二种情况 (ATOM X)是 NIL。

在第一种情况下，每当(ATOM X)是 T 时，LENGTH 就得 0。因此，所有原子的长度均为 0，而且由于空表( )就是原子 NIL，所以第一种情况赋予空表的长度为 0。

如果 (ATOM X) 是 NIL，那么由 COND 的第二行给出的第二种情况就适用了。为了了解这种情况下函数是如何工作的，假定 X 的值是表(A B C)，计算机工作情况如下：

· 计算(CDR X)。因为 X 的值是(A B C)，所以(CDR X)的值是(B C)。

· 计算 (LENGTH (CDR X))。因为 (CDR X) 的值是 (B C)，所以(LENGTH(CDR X))的值是 2。

· 计算(PLUS 1(LENGTH(CDR X)))。因为 (LENGTH (CDR X)) 的值是 2，所以 (PLUS 1 (LENGTH (CDR X))) 的值必定是 3。3 即作为初始表 (A B C) 的长度。当然

这是正确的。

只剩下一个问题，就是我们用 LENGTH 本身定义了 LENGTH。在计算 (LENGTH '(A B C)) 的值时，我们采用了 (LENGTH '(B C)) 的值为 2 的事实。我们是怎样巧妙地做到这点的呢？

让我们仔细看一看当求 (LENGTH '(A B C)) 的值时会发生什么情况。我们给 X 定义，并给出 X 值 (A B C)。因为 (A B C) 不是一个原子，所以第二种情况成立，并且要算出 (LENGTH (CDR X)) 的值。因为 (CDR X) 的值为 (B C)，所以上式的求值与 (LENGTH '(B C)) 的求值是等价的。

为了得到 (LENGTH '(B C)) 的值，我们必须回到定义。X 现在的值为 (B C)。（我们每利用一次定义，X 就会有一个不同的值。）因为 (B C) 不是原子，所以适用第二种情况，且又一次求 (LENGTH (CDR X)) 的值。这次，(CDR X) 的值是 (C)，所以我们这时要求的是 (LENGTH '(C)) 的值。

求 (LENGTH '(C)) 使我们又一次返回到定义。此外，(C) 不是原子，适用第二种情况，并且要求的是 (LENGTH (CDR X)) 的值。(CDR '(C)) 的值是空表 ( )，所以我们现在要求的是 (LENGTH '( ))的值。

下一步是决定性的。你可能感到奇怪，究竟是什么会防止我们永远重复引用函数 LENGTH？防止的方法就是我们每引用一次 LENGTH，所得到的表的元素就比上一次少了一个。最终我们必定会将函数用于空表 ( ) 上。但空表由第一种情况处理，就不会陷入再一次引用 LENGTH 的境地。

因此，当我们为求 (LENGTH '( )) 返回到定义时，第一种情况就成立了，并且所得到的值为 0。

既然知道了 (LENGTH '( )) 的值，我们就能回过头来求

(LENGTH '(C))的值，它是由(PLUS 1(LENGTH '( )))的值来确定的，这个值为1。

知道了(LENGTH '(C))的值，我们就能计算(LENGTH '(B C))的值。依据定义(LENGTH '(B C))的值等于(PLUS 1(LENGTH '(C)))的值，即为2。

最后，(LENGTH '(B C))的值被用来计算(LENGTH '(ABC))的值，如前所述。

为了计算(LENGTH '(A B C))的值，LENGTH函数被反复引用了4次。

表达式	值
(LENGTH '(A B C))	3
(LENGTH '(B C))	2
(LENGTH '(C))	1
(LENGTH '( ))	0

我们可以认为计算机是这样工作的：先按左边的表达式自上而下地工作，然后按右列的值自下而上地倒推求值。在完全计算出来任何值之前，必须引用LENGTH函数4次。首先算出的是(LENGTH '( ))的值，然后用这个值来计算(LENGTH '(C))的值，而(LENGTH '(C))的值又用来计算(LENGTH '(B C))，如此等等。

概括起来递归定义有以下几个基本特点：

1. 定义必须包含一个COND，其中至少有两种情况，也可以是更多的情况。

2. 必须至少有一种情况能够在不利用被定义函数的情况下求值。

3. 当引用到被定义的函数时，必须使函数变元简化，其方法是：通过反复引用该函数直到不再引用这个函数本身就

能求值的情况出现为止(例如，每利用一次 LENGTH，它的变元就减少了一个元素，最后引用到(LENGTH '( ))，而它的求值就不需要再引用 LENGTH)。

下面是递归定义的另一个例子。假定我们希望得到一个函数 SUBST，它能用一个值来代替表达式中的一个变量(这样一个函数在谓词逻辑演算中是非常有用的，这是其应用的一个方面)。

SUBST 表示如下：

```
(SUBST 'TABLE 'X '(AT ROBOT X))
    (AT ROBOT TABLE)
(SUBST 3 'X '(TIMES X X))
    (TIMES 3 3)
(SUBST '(TIMES 5 5) 'Y '(PLUS 10 Y))
    (PLUS 10(TIMES 5 5))
```

通常，我们希望定义一个函数(SUBST X Y Z)，使得 Z 值中的 Y 值由 X 值来代替。虽然在此函数的应用中，Y 常常是原子，但 Y 不一定非原子不可。

SUBST 的递归定义为：

```
(DEFINE(SUBST X Y Z)
(COND((EQUAL Y Z)X)
      ((ATOM Z)Z)
      (T(CONS(SUBST X Y(CAR Z))
          (SUBST X Y(CDR Z))))))
```

SUBST 有三种情况，前两种情况用不到 SUBST，而第三种情况要用到 SUBST。

第一种情况是被代替的值与 Z 本身相同时

```
(SUBST '(A B) 'U 'U)
```

(A B)

在这种情况下得到 X 的值。

第二种情况是 Z 的值是一个原子，它不等于所要代替值：

(SUBST '(A B) 'U 'V)

'V

在这种情况下，不可能发生代替，结果得出 Z 的值。

第三种情况是产生递归的情况。我们将 Z 的值分解为(CAR Z)和(CDR Z)。于是，调用 SUBST 在(CAR Z)和(CDR Z)中作替换，所得到的值用 CONS 重新组成一个表达式。

例如，假定我们要寻求(SUBST 'A 'U '((F U)(G U)))的值。Z 的值为((F U)(G U))，(CAR Z)的值为(F U)，而(CDR Z)的值为((G U))。

将 SUBST 加到(CAR Z)和(CDR Z)上，得到：

表达式	值
(SUBST X Y(CAR Z))	(F A)
(SUBST X Y(CDR Z))	((GA))

(记住 X 的值为 A，Y 的值为 U) CONS 将 (F A) 加到表((G A))中去，以便得到((F A)(G A))，此为(SUBST 'A 'U '((F U)(G U)))的值。

SUBST 的这个定义是有效的递归定义吗？或者，它会进入无穷地反复引用 SUBST 的循环吗？

SUBST 的定义是有效的。如果反复地对一个表达式取 CAR 和 CDR，我们最终会得到一个原子（可能是 NIL）。因此，最后调用 SUBST 时，第三变元必然是原子。于是头两种情况之一就会用到(若第一种情况未用到，则第二种情况一定会用到)。事实上，在完成求值之前，对作为第三变元 Z 值中

的每一个原子都已经引用了 SUBST。

(八)程序特编

虽然递归定义是简捷的，但它们并不总是表示计算函数值的最有效的方法。每次引用到函数本身，描述引用函数的信息不得不存到计算机里。这些信息不但占据存储空间，而且为了存储这些信息还需要时间。在任一结果作出之前，递归函数可能调用多次，甚至开始计算所希望的值之前，许多空间和时间还可能被递归调用所占用。

LISP 程序特编允许 LISP 根据类似于其它语言的程序(例如 BASIC 或者 FORTRAN)来定义函数。许多函数用程序计算比用递归计算更简便。

让我们以一个 LENGTH 的程序来说明：

```
(DEFINE(LENGTH X)
    (PROG(N)
          (SETQ N 0)
    LOOP
          (COND((ATOM X)(RETURN N)))
          (SETQ N(PLUS N 1))
          (SETQ X(CDR X))
          (GO LOOP)))
```

PROG 为 LISP 函数，它用来解释程序，其形式为：

```
  (PROG   局部变量
          程序步骤 1
            ⋮
            ⋮
          程序步骤 n)
```

紧接着原子 PROG 的是局部变量表(在 LENGTH 中，仅

有一个局部变量 N)。程序用局部变量表示“便条”式的暂时存储，一部分程序中的变元先赋予需要的值，此赋值随后即用于程序的另一部分。

局部变量的意义在于：其中一个变量的赋值对于程序以外的同名变量的值不起作用。例如，在 LENGTH 定义中对 N 的赋值仅仅在 LENGTH 的定义中会影响 N 的值。而对别处，例如在其它函数的定义中 N 的值没有影响。

局部变量表的后面就是程序步骤。有两种程序步骤：

1. 表达式。这些都是表，例如：

```
(SETQ N (PLUS N 1))
```

这些表达式的目的与其它语言中的“语句”相同，虽然这些表达式有值，但它们的值是被忽略的，表达式只用来对它们的“副作用”求值，比如对变量赋值。上述表达式的作用就是使 N 值增加 1。

2. 标号。标号是原子。在 LENGTH 中唯一的标号是 LOOP，它们与汇编语言中的标号，BASIC 和 FORTRAN 中的语句标号起同样的作用。人们可以指挥计算机从程序的其它地方“转”到标号这儿。在 LENGTH 中执行 (GO LOOP) 使得原子 LOOP 后面的表达式再次被求值。

LENGTH 程序由设置 N 的值为零开始，而后检查 X 之值是否是一个原子。如果是一个原子，(RETURN N) 就会停止执行程序，得出 N 的值作为 PROG 的值，也就是 LENGTH 的值。N 目前的值为 0，所以若 X 的值是一个原子，则 (LENGTH X) 之值为 0，这正是所期望的值。

假如 X 的值不是一个原子，则程序反复执行由 LOOP 到程序末尾的循环。每循环一次，N 的值被

```
(SETQ N(PLUS N 1))
```

加 1，并且第一个元素被

(SETQ X(CDR X))

从 X 的值中去掉。当程序转了整整一个循环以后，就从 X 的值中一个一个地去掉元素，并且用 N 计算所去掉元素的个数。最后，所有的元素都被去掉，X 的值就会是()或 NIL。因为 NIL 是一个原子。

(COND(ATOM X)(RETURN N))

就会使程序得出 N 的值。由于 N 计算已被去掉的元素的个数，并且所有的元素都被去掉了，所以 N 就是 X 初始值表中元素的数目。

例如，假定我们要求(LENGTH '(A B C D E))的值。X 初始值为(A B C D E)，每次程序到达标号 LOOP 时，X 与 N 逐次的值为：

X 值	N 值
(A B C D E)	0
(B C D E)	1
(C D E)	2
(D E)	3
(E)	4
( )	5

因为()是一个原子，(COND ((ATOM X) (RETURN N)))使 PROG 得到值 5，它正是(LENGTH '(A B C D E))的值。

什么时候我们采用递归定义？什么时候我们采用程序循环呢？当我们只不过通过表的元素来工作时，采用递归一般不是有效的。因此 LENGTH（它只是简单地计算表中元素的数目）的递归定义不是一个好定义。用程序的方法计算元素的数目是比较快和比较节省存储的，因为对表中的每一个元

素，我们都没有重新引用 LENGTH。

那么，什么时候使用递归呢？许多信息结构包含有与结构本身形式相同的某些部分。一个表可以含有作为其元素的其它的表。一个表达式可以含有若干子表达式，而子表达式本身又是表达式。树可以含有若干子树。如果我们定义一个函数去处理一个表，一个表达式或一个树，那么调用正被定义的函数去处理表的一个元素、一个子表达式或一个子树是很自然的。采用程序特编可以得到相同的结果，但是由此编成的程序比递归函数定义复杂得多。

还有另一种考虑。让我们称嵌套的深度为一项中包含类似项作为其子部分的次数。在下表中

(A B(C(D E)F)G H)

(D E)的嵌套深度是 2，因为(D E)是(C(D E)F)的一个元素，而(C(D E)F)又是(A B(C(D E)F)G H)的一个元素。一般这种嵌套深度还算是少的，比一个典型的表中的元素要少得多。因此，递归调用的次数要尽量的少，才能使递归对效率降低的影响也减到最小。

在 LISP 中，我们用 CAR 函数，从一个表中取出它的第一个元素，该元素也可以是一个表。另一方面，当我们仅仅通过表的元素来工作时，我们用 CDR 来一个一个地移去前面的元素，从而显示出表中后面的元素。所以递归调用一个表的 CAR 通常是恰当的，而有关 CDR 的那些递归调用，一般最好代之以程序循环。

例如上一节定义的函数 SUBST，进行了两次递归调用：

(SUBST X Y (CAR Z))

(SUBST X Y (CDR Z))

由刚才所说的，我们知道可以采取保留 CAR 的递归调用的办

法，而以程序循环来代替 CDR 的递归调用来改进 SUBST 的定义。这样一来，我们得到下面的程序：

```
(DEFINE(SUBST X Y Z)
        (PROG(U V)
(COND((EQUAL Y Z)(RETURN X))
      ((ATOM Z)(RETURN Z)))
    LOOP
        (COND((NULL Z)(RETURN U)))
        (SETQ V (SUBST X Y (CAR Z)))
        (SETQ U (APPEND U (LIST V)))
        (SETQ Z (CDR Z))
        (GO LOOP)))
```

程序由检查 (EQUAL Y Z) 和 (ATOM Z) 开始，恰好象递归定义所作的那样。若 (EQUAL Y Z) 为 T，所得到的 X 值作为函数的值，若(ATOM Z)为 T，所得到 Z 值作为函数的值。

反之，由 LOOP 开始的循环被重复执行，每次通过循环 (SETQ Z (CDR Z)) 消去 Z 值目前的第一个元素。因此 Z 的初始值中的每一个元素都能在某一循环中成为第一个元素。

每次通过循环

```
(SETQ V (SUBST X Y(CAR Z)))
```

递归地调用 SUBST，在 Z 的第一个元素上执行代替。所执行的代替的结果赋予局部变量 V。由于我们重复循环，原始表中的每一个元素都会变成 Z 的第一个元素。V 的序列值就是在那些元素上执行代替的结果。

在 Z 的初始值中每一个元素上进行代替而成的表就组成局部变量 U 的值。局部变量总是由 NIL 值开始，因此 U 的值最初是空表，每通过一次循环，V 的现时值被

(SETQ U (APPEND U (LIST V)))

加到表的末尾。函数(LIST V)用一对括号括入V值，使它成为单元素的表：(LIST 'A)的值是(A)。于是这个单元素的表被附加到U值上，结果是V的值被加上作为U值的一个元素。

最后所有的元素都会从Z值中被消去，Z值成为()。对一空表，谓词NULL得T，反之得NIL。因此当Z的初始值的所有元素都已经被处理，而且被处理的元素都已放入表U中以后，则

(COND(NULL Z)(RETURN U))

得U的值作为PROG的值，因而就是SUBST的值。

例如，假定要求

(SUBST 'A 'B((C B)(B D)))

的值。X值与Y值分别为'A和'B。每当计算机到达标号LOOP时，Z值，U值和V值变化如下：

Z	U	V
((C B)(B D))	( )	( )
((B D))	((C A))	(C A)
()	((C A)(A D))	(A D)

最后，在大型计算机上的LISP文本有一个MAPCAR函数，它是把某一个函数用于一个表的每一个元素上。利用MAPCAR比利用程序循环更为有效。虽然循环仍然发生，但它发生在执行MAPCAR的机器语言程序之中，而不是在LISP程序之中。

(九)性质表

正如前面叙述过的，LISP把每个原子都用一个性质表联系起来。性质表具有我们所希望的与特定原子相联系的信

息。例如，在 ROBOT 的性质表中，我们可以放入我们所要知道的有关机器人的一切——它的位置，它是否拿着某种东西，等等。

在 LISP 中，数据库也就是程序处理有关事物时所需要的信息，一般是存在性质表内。

一个原子的性质表，通常不用表处理函数例如 CAR，CDR，APPEND 来处理；而是用特殊的函数来处理性质表，其中最重要的是 GET 和 PUTPROP。

例如，假定机器人在房间中的问题里，我们希望记录机器人目前的位置是在壁间里。我们可以写：

```
(PUTPROP 'ROBOT 'ALCOVE 'LOCATION)
```

这就是给性质 LOCATION 一个 ALCOVE 的值。如果 ROBOT 的性质表已经没有 LOCATION 这个性质，我们就放入这个性质，并还带有值 ALCOVE。为了找到机器人的位置，我们利用函数 GET：

```
(GET 'ROBOT 'LOCATION)
    ALCOVE
```

为了记录机器人移动到新位置，我们可以利用 PUTPROP 来改变位置性质的值：

```
(PUTPROP 'ROBOT 'TABLE 'LOCATION)
```

而后 GET 就会产生：

```
(GET 'ROBOT 'LOCATION)
    TABLE
```

因为每个原子都有它自己的性质表，所以我们能够很容易地记录我们所希望的许多不同机器人的位置

```
    (PUTPROP 'ROBOT1 'ALCOVE 'LOCATION)
    ALCOVE
```

```
(PUTPROP 'ROBOT2 'TABLE 'LOCATION)
  TABLE
(GET 'ROBOT1 'LOCATION)
  ALCOVE
(GET 'ROBOT2 'LOCATION)
  TABLE
```

注意 PUTPROP 函数的值就是它放在性质表中的值，(PUTPROP 'ROBOT 'ALCOVE 'LOCATION) 的值是 ALCOVE。此外 PUTROP 是其"副作用"比求其值用得更多的另一个函数。重要的是 PUTPROP 在性质表中所造成的变化，而不是包含 PUTPROP 的表达式的值。

(十)LISP 的其它文本

像使用一些其它语言的情况一样，不同的人在写 LISP 解释程序的时候，使用了略有差别的 LISP 文本。通常，在这些情况下，用户必须参考特殊文本的语言手册；在使用这种手册时，用户必须仔细注意有哪些特点是可用的以及是如何使用它们的。本章所描述的 LISP 文本和其它一些文本之间的几个主要区别是：在某些 LISP 文本中，必须用 QUOTE 函数来代替单个引号，以 (QUOTE A) 代替 'A。这使得表达式非常累赘，例如：

```
(SUBST 'A 'B '((C B)(D B))
```

写成了

```
(SUBST(QUOTE A)(QUOTE B)(QUOTE((C B)(D B))))
```

为了方便用户，采用 QUOTE 函数的 LISP 文本一般都提供一种写入函数及其变元的方法，这种办法不需要对所有的变元加引号，因此就不用打入

```
(CONS(QUOTE A)(QUOTE(B C D)))
```

而可以打入

CONS(A(B C D))

即以由 CONS 函数和变元表(A(B C D))组成的对儿来打入。表中的变元是自动地加引号的，这样就可以避免使用很不灵活的函数 QUOTE。照例，计算机得出的函数值也是：

CONS(A(B C D))

(A B C D)

要注意的是简化了的文本，仅当函数及其变元为了立刻求值而打入时才工作。当有关函数的表达式在另一个函数的定义中出现时，则必须采用较繁的形式。

函数的定义还包括一个函数 LAMBDA，它对在定义中所使用的虚变量赋予函数变元的值。(LAMBDA 这个名字是由数学上采用希腊字母 λ 代表函数而来的。)例如，在许多老的 LISP 文本中，函数 SQUARE 就写作：

(DEFINE((SQUARE(LAMBDA(X)(TIMES X X)))))

而不是写作目前大多数文本所采用的比较简单的形式：

(DEFINE(SQUARE X)(TIMES X X))

(在目前的文本中，LAMBDA 函数是可以利用的，并且能够用于专门用途，如当需要定义一个还未定名的函数的时候。但不是每一个定义都必须利用 LAMBDA)。

函数 SETQ 只能用于函数定义的里面。为了直接对计算机写入给一个原子赋予特定值的请求，则采用 CSETQ 函数来代替 SETQ 函数。

(十一)LISP 解释程序

LISP 能够在小型计算机上付诸实现。适用于 PDP-8 微型计算机的执行程序已在文献目录中给出。这个解释程序的清单是值得每一个愿意在其机器上执行 LISP 的人好好研究。

PDP-8 的解释程序大约占 2K 字，相当于微处理机 4KBYTE * 左右。

* BYTE 为二进制信息组，1BYTE＝8bit。

附录… 参 考 文 献

GENERAL

1. Edward Feigenbaum and Julian Feldman, eds., *Computers and Thought*, McGraw Hill, New York, 1963.
2. Philip Jackson, Jr., *Introduction to Artificial Intelligence*, Petrocelli/Charter, New York, 1974
3. Nils J. Nilsson, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971.
4. Bertram Raphael, *The Thinking Computer*, W. H. Freeman, San Francisco, 1976.
6 James R. Slagel, *Artificial Intelligence: The Heuristic Programming Approach*, McGraw-Hill New York, 1971.
7 Patrick Henry Winston, *Artificial Intelligence*, Addison-Wesley, Reading, Mass., 1977.

PROBLEM SOLVING TECHNIQUES

8. Wayne A. Wickelgren, *How to Solve Problems*, W. H. Freeman, San Francisco, 1974

See also[3]and[6]

PLANNING

9. Earl D. Sacerdoti, *A Structure for Plans and Behavior*, Elsevier, New York 1977.

CHESS

10. Peter W. Frey, ed., *Chess Skill in Man and Machine*, Springer-Verlag, New York, 1977.

11. Edward W. Kozdrowicki and Dennis W Cooper, "COKO III: The Cooper-Koz Chess Program," *Communications of the ACM*, July, 1977p. 4 ll.

12. Monroe Newborn, *Computer Chess*, Academic Press, New York, 1975.

13. Herbert A. Simon and William G. Chase, "Skill in Chess," *American Scientist*, July-August, 1973, p. 394.

14. Albert L. Zobrist and Frederic R. Carlson, Jr., "An Advice-Taking Chess Computer," *Scientific American*, June, 1973 p. 92.

PATTERN RECOGNITION AND PERCEPTION

15. Partrick Henry Winston, ed., *The Psychology of Computer Vision*, McGraw-Hill, New York, 1975.

See also[2]and[7].

ROBOTS

16. James S. Albus and John M. Evans, Jr., "Robot Systems," *Scientific American*, February, 1976, p. 76.

17. David Heisman, *Building Your Own Working Robot*, TAB Boos, Blue Ridge Summit, Pa., 1976.

18. Ralph Hollis, "Newt: A Mobile, Cognitive Robot," *Byte*, June, 1977, p. 30.

19. James L. Nevins and Daniel E Whitney, "Computer-Controlled Assembly," *Scientific American*, February, 1978, p. 62

COMPUTATIONAL LOGIC

See[3]and[6]

NATURAL LANGUAGE PROCESSING

20 Neil M. Goldman, "Sentence Paraphrasing from a Conceptual

Base," *Communications of the ACM*, February, 1975, p. 96.

21. Roger C. Schank, *et. al.*, "Inference and Paraphrase by Computer," *Journal of the Association for Computing Machinery*, July, 1975, 309.
22. R. Simmons and J. Slocum, "Generating English Discourse from Sematic Networks," *Communications of the ACM*, October, 1972, p. 891.

See also[7].

LISP .

23. G. vander Mey and W. L. vander Poel, *A Lisp Interpreter for the PDP*. 8, DECUS (Digtal Equipment Corporanon User's Society), Maynard, Mass, 1968, DECUS No. 8-102a. Program writeup, program listing, and paper tape are available

See also[7]

附录二　汉英名词对照表

说明：本表术语按本书章次与出现前后为序，括号内为参考译名

第　一　章

问题求解	problem solving
自然语言处理	natural language processing
模式识别	pattern recongnition
感知	perception
检索(再现)	retrieval
博弈游戏	game-playing
计算逻辑	computational logic
表达(表现)	representation
搜索	search
启发式	heuristic
组合爆炸	combinatorial explosion
规划	planning
启发式搜索	heuristic search

第　二　章

问题	problem
策略	strategy
状态	state

操作(运算)	operation
目标	goal
"传教士与野人"	Missionaries and Cannibals
操作符(算子)	operator
约束	comstraint
图	graph
网络	network
节点	node
弧(弧线)	arc
标记	label
色图	colored graph
前驱(前项)	predecessor
后继(后项)	successor
梵塔	Tower of Hanoi
暗示图	implicit graph
明示图	explicit graph

第 三 章

状态图搜索	state-graph search
迷宫	maze
重复分枝	repeated branching
树	tree
根	root
叶	leaf
枝	branch
深度	depth

子代	children
亲代	parent
同胞	siblings
孪生	twins
祖代	ancestor
后代	descendant
搜索树	search tree
死端(死点)	dead end
扩展	expansion
开节点	open node
闭节点	closed node
深度优先(先纵)搜索	depth-first search
宽度优先(先横)搜索	breadth-first search
队	queue
堆栈	stack
亲节点	parent node
子节点	child node
深度界限	depth bound
有序搜索	ordered search
最低代价搜索	lowest-cost path
估值	estimate
评价函数	evaluation function
优先次序队	priority queue
排序	ordering
生成	generation
修剪	pruning
终止	termination

第 四 章

子问题	subproblem
子目标	subgoal
计划	plan
关键操作符	key operator
差异	difference
差异递减法	difference reduction
与/或树	AND/OR tree
问题约简法	problem reduction
原问题	primitive problem
已解的	solved
无解的	unsolvable
与扩展	AND expansion
或扩展	OR expansion
替身	alternation

第 五 章

分级计划	hierarchical plan
过程网	procedural net
分级规划	hierarchical planning
等级(层次)	hierarchy
步骤	step
细化	refinement
监督执行	monitoring execution

非线性计划	nonlinear plan
幻节点	phantom node
评判	criticism
评判法	critics

第 六 章

树状搜索	tree search
博弈树	game tree
最小最大值	minimax
静态评价函数	static evaluation function
回溯值	back-up value
α—β 方法	alpha—beta procedure
α 剪枝	alpha cutoff
β 剪枝	beta cutoff

第 七 章

广度	width
全广度搜索	full—width search
扇形参数	fanout parameter
可行排序	plausibility ordering
深度控制	depth cutrol
深度剪技	depth—cutoff
特性表	feature list
查表法	table lookup

第 八 章

模式	pattern
模式实例	pattern example
模式法则	pattern rule
模式匹配	pattern matching
特征抽取	feature extraction
预处理	preprocessing
预分析	preanalysis
训练集	training set
试验集	test set
判定表	decision table
线性可分	linearly seperable
超平面	hyperplane
原型	prototype
样板	template
原始草图	primal sketch
区域	region
图像获取	image acguisition
图像分析	image analysis
平滑	smoothing

第 九 章

外部坐标	external coordinates
内部坐标	internal coordinates
工作空间坐标	work space coordinates

效应器	effector
操作器	manipulator
感觉器	sensor

第 十 章

命题	proposition
谓词	predicates
符号逻辑	symbolic logic
数理逻辑	mathematical logic
谓词逻辑	predicate logic
连接词	connectives
否定	negation
真值表	truth table
命题演算	propositional calculus
谓词演算	predicate calculus
真	true
假	false
论证	argument
论证方式	argument form
有效性	validity
有效(永真)	valid
无效	invalid
合取	conjunction
假言推理	Modus Ponens
个体	individual
变元	argument

个体变量	individual variable
实例化	instantiation
个体常量	individual constsnt
全称变量	universal variable
存在变量	existantial variable

第 十一 章

归结(消解)	resolution
原子	atom
文字	literal
子句	clause
空子句	empty clause
矛盾	contradiction
归谬法	reduction and absurdum
一致化	unificaltion
因子析出	factoring
因子	factor
替换	substitution
完备性	completeness

第 十二 章

表达法	representation
属性值	attribute value
增表	add list
删表	delete list

特性(性质)表	property list
语义网络	semantic network
框架(画面)	frame
槽(空位)	slot
产生式	production
条件-动作	condition—action

第 十三 章

句法	syntax
语义学	semantics
嵌套结构	nested structure
转移网络	transition network
增广转移网络	augmented transition network ATN
概念依赖	concept dependency

第 十四 章

原子	atom
表	list
成员(元素)	member
表达	representing
表达式	expression
表达法	representation
函数	function
变元	atgument

表函数	list function
表处理函数	list-manipulating function
引号	quote
程序特编	program feature
局部变量	local variable
程序步骤	program step
标号	label
性质表	property list
解释程序	interpreter

封面页
书名页
版权页
前言页
目录页
第一章　什么是人工智能
一、人工智能的应用
二、为什么我们需要更聪明的计算机
三、算法
四、人工智能程序设计原理
第二章　问题求解的策略
一、状态、操作和目标
二、状态图
三、计算机的表达方法
第三章　状态图搜索
一、搜索树
二、宽度优先搜索
三、深度优先搜索
四、有序搜索
五、其它启发技术
第四章　子问题、子目标和计划
一、子目标和计划
二、编写计划
三、搜索与/或树
第五章　分级计划和过程网
一、分级计划
二、过程网
三、监督执行
第六章　博弈游戏程序：树状搜索
一、博弈树
二、博弈策略
三、最小最大值方法
四、终局和静态评价函数
五、深度优先的最小最大值评价方法
六、α-β方法
第七章　博弈游戏程序：启发式方法
一、棋局的表达
二、控制博弈树的规模
三、评价函数
四、规划
第八章　模式识别与感知
一、定义
二、特征空间、区域和原型
三、图象分析
第九章　机器人
一、效应器

二、感觉器
三、控制
第十章　计算逻辑：命题和谓词
一、命题逻辑
二、谓词逻辑
第十一章　计算逻辑：归结法
一、归结原理
二、转换成子句形式
三、一致化
四、小结
第十二章　知识表达法
一、基于谓词逻辑的表达法
二、知识表达法与规则
三、特性表
四、语义网络
五、框架
六、产生式系统
第十三章　自然语言处理
一、句法、语义和转移网络
二、格语法
三、概念依赖理论
第十四章　ＬＩＳＰ语言
一、原子和表
二、用表来表示信息结构
三、ＬＩＳＰ程序设计的原理
附录一　参考文献
附录二　汉英名词对照表
附录页