普通高等教育“十五”国家级规划教材

计算机算法基础

(第三版)

余祥宣　崔国华　邹海明

华中科技大学出版社

内 容 简 介

本书是教育部普通高等教育“十五”国家级规划教材。

计算机算法是计算机科学和计算机应用的核心。无论是计算机系统、系统软件的设计，还是为解决计算机的各种应用课题做的设计都可归结为算法的设计。

本书围绕算法设计的基本方法，对计算机领域中许多常用的非数值算法作了精辟的描述，并分析了这些算法所需的时间和空间。全书共分 11 章，第 1 章系统地介绍了计算机算法所涉及的数学知识，第 2 章至第 9 章介绍了递归算法、分治法、贪心法、动态规划、基本检索与周游方法、回溯法以及分枝-限界法等基本设计方法，第 10 章对当今计算机科学的前沿课题——P $\stackrel{?}{=}$ NP 问题的有关知识作了初步介绍，第 11 章则对日益兴起的并行算法的基本设计方法作了介绍。

本书可作为高等院校与计算机有关的各专业的教学用书，也可作为从事计算机科学、工程和应用的工作人员的自学教材和参考书。

书　名：计算机算法基础(第三版)

作　者：余祥宣

出版社：华中理工大学出版社

出版日期：2000

ISBN：7-5609-2126-4 / TP301.6

定 价：19.80

序

凡是学习了一种程序设计语言(不论是初级的还是高级的)课程并能编写一些实用程序的读者,也许都有这样一种体会,学会编程容易,但要想编出好程序难,因而很想系统地学习设计算法的知识。另外,一些著名计算机科学家在有关计算机科学教育的论述中认为,计算机科学是一种创造性思维活动的科学,其教育必须面向设计。计算机算法设计与分析正是面向设计的、处于核心地位的教育课程。

基于上述的认识,自 20 世纪 80 年代初,作者对计算机科学专业的学生一直坚持开设算法设计与分析课程。在这门课程的教学过程中,我们查阅了国外流行的数种教材,发现多数教材在面向设计方面不是重视不够就是处理不甚恰当,只有 E .Horowitz 和 S .Sahni 合著的“Fundamentals of Computer Algorithms”一书比较集中地反映了以上观点。不过要将该书作为一个学期的教材则嫌内容太多。为了解决此门课程教学的急需,在选用此书作为主要素材并参考、吸取了其它书籍的某些长处的基础上,根据计算机各专业学生目前需要形成的知识结构和在教学实践中的体会,于 1985 年编写了这本《计算机算法基础》,希望能对从事计算机算法教学和想设计出良好算法的人有所裨益。

本书出版以来,受到许多学校的普遍欢迎,并于 1992 年荣获机电部电子教材类一等奖,2000 年被列为教育部普通高等教育“十五”国家级规划教材。

计算机科学技术发展很快,特别是近几年来并行处理技术渐趋成熟,高性能并行计算机已投入使用。因此,算法课程除了讲授串行算法的设计方法外,必须增加并行算法的有关内容,以便将学生培养成为面向 21 世纪的计算机高级技术人才。鉴于上述原因,于 1998 年对本书进行了修订再版。再版时,删去了某些过时的内容,增加了新的内容,如第 11 章并行算法。

计算机算法设计与分析和数学密切相关,为使读者在设计和分析算法方面具有坚实的基础,本次修订把与算法密切相关的数学知识归结成一章。由于递归是算法设计与分析的重要方法,在本次再版中增加了递归算法的内容,全面地介绍此类算法相关的知识。

全书共分 11 章。在第 1 章中介绍了有关的数学概念,同时在计数方法、母函数以及级数求和等方面做了系统的介绍;在第 2 章中介绍了算法的基本概念,并对计算复杂度、算法的描述工具和本书用到的基本数据结构作了简要的阐述;第 3 章围绕递归算法的实现机制、递归算法设计以及算法复杂度分析等内容,对递归算法做了全面的介绍;然后,针对设计算法时常用的一些基本设计策略组织了第 4 章至第 9 章的内容。其中每一章都先介绍一种设计算法的基本方法,然后从解决计算机科学和应用中出现的实际问题入手,由简到繁地描述几个经典的精巧算法,同时对每个算法所需的时间和空间作出数量级的分析,使读者既能学到一些常用的精巧算法,又能通过对这些设计策略的反复应用,牢固掌握这些基本设计方法,收到融会贯通之效。细心的读者从目录中就可立即发现,一个问题往往可以用多种设计

策略求解。要指出的是,随着本书内容的逐步展开,读者将会体会到综合应用多种设计策略可以更有效地解决问题。第 10 章对当今计算机科学的前沿课题——P $\stackrel{?}{=}$ NP 问题的有关知识作了初步介绍,目的是希望读者在学习了前 9 章内容之后,在理论上能提高到一个新的高度,激发某些读者对此课题的研究兴趣。第 11 章讨论了各种通用并行计算模型上的并行算法设计和分析方法。它以并行计算模型为线索,讲述了并行算法的基础知识及其基本设计技术,着重介绍了各种常用的和典型的并行算法。

本书的内容是自成体系的,凡具有大学二年级数学基础和有使用过一种高级程序设计语言编程序经验的人,都能学懂本书的内容。对于已学过数据结构课程的读者,可以跳过 2.4 节,而对于没学过数据结构课程的读者,则必须认真学习并掌握此节的全部内容。各章后面都附有一定数量的习题,其中有些题目最好是在写出算法后,用一种语言写成程序并到机器上去运行,以检验所设计的算法的有效性。

讲授这门课程一般 70 学时左右即可完成。根据数年来对大学本科生、研究生讲授这门课程的经验,建议对本科生讲授第 1 章至第 9 章以及第 11 章的全部或大部分内容,对第 10 章只作简要的介绍;研究生则需认真学习第 10 章,其余章节的学习与本科生的相同。这门课程既可采用课堂讲授方式,也可采用讲授、自学和讨论相结合的方式。

本书第 2 章以及第 4 章到第 9 章由余祥宣编写,第 10 章由邹海明编写,第 1 章、第 3 章和第 11 章由崔国华编写。值本书再版之际,谨向在数年前就向我们建议增加并行算法的中国科学技术大学唐策善教授致以诚挚的感谢。

编　　者

2005 年 10 月

目　　录

第 1 章

数学预备知识

数学是算法设计与算法分析的重要基础,本章将介绍其后各章所涉及的数学概念和数学方法。在后面的各章节中,读者若遇到不熟悉的数学概念和数学方法,则可以学习或查阅本章的相关部分。

1.1 集　　合

数学意义上的集合概念在计算机科学上有着广泛的用途。

定义 1.1　在研究某一类对象时,可把这类对象的整体称为集合,组成一个集合的对象称为该集合的元素。

设 A 是一个集合,a 是集合 A 中的元素,可表示为 $a\in A$,读作 a 属于 A。如果 a 不是集合 A 的元素,则表示为 $a\notin A$,读作 a 不属于 A。

例如,26 个小写英文字母,可组成一个字母集合 A,每个小写字母皆属于 A,可写为 $a\in A, b\in A, \cdots, z\in A$。所有阿拉伯数字都不属于 A,则有 $2\notin A, 8\notin A$ 等。

有限个元素 $x_1, x_2, \cdots, x_n$ 组成的集合,称为有限集合。无限个元素组成的集合,称为无限集合。例如,整数构成的集合是一个无限集合。

把不含元素的集合,称为空集,记为 $\varnothing$。

集合的表示法,有列举法和描述法两种。

列举法是把集合的元素一一列举出来。例如,26 个小写英文字母组成的集合 A,可写成 $A=\{a, b, c, \cdots, z\}$;阿拉伯数字组成的集合 D,可写成 $D=\{0, 1, 2, \cdots, 9\}$ 以及集合 $C=\{a^1, a^2, a^3, \cdots\}$ 等。

描述法描述出集合中元素所符合的规则。

例如,$N=\{n\mid n$ 是自然数$\}$,表明 N 是自然数的集合。

$$A=\{x\mid x\in Z \text{ 且 } 0\leq x\leq 5\}$$

其中,Z 是整数集,则 $A=\{0,1,2,3,4,5\}$。

1.1.1 集合之间的关系

(1) 设两个集合 A, B 包含的元素完全相同,则称集合 A 和 B 相等,表示为 $A=B$。

例如,若集合 $A=\{a, b, c\}$,集合 $B=\{b, a, c\}$,则有 $A=B$。

应指出,一个集合中元素排列的顺序是无关紧要的。

对有限集合 A,其中不同元素的个数称为集合的基数,表示为 $\#A$ 或 $|A|$。

例如,$B=\{a, b, c, 4, 8\}$,其基数 $\#B=5$。

(2) 设两个集合 A, B,当 A 的元素都是 B 的元素时,称 A 包含于 B,或称 A 是 B 的子

集,表示为 $A \quad B$。当 $A \quad B$ 且 $A \neq B$ 时,称 A 是 B 的真子集,表示为 $A \quad B$。

如果所研究的集合皆为某个集合的子集,则称该集合为全集,记为 E。

(3) 由(1)和(2)可知,对于任意两个集合 A, B, $A = B$ 的充要条件是 $A \quad B$ 且 $B \quad A$。

1.1.2 幂集

设 A 是一个集合,则 A 的所有子集组成的集合称为 A 的幂集,表示为 2^A 或 $\rho(A)$。

例如:设 $A = \{a, b, c\}$,则有

$$\rho(A) = \{\quad, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$$

设 A 是有限集, $\# A = n$,则 $\rho(A)$ 的元素个数为

$$C_n^0 + C_n^1 + \cdots + C_n^n = 2^n$$

应指出,空集 是任何集合的一个子集。

1.1.3 集合的运算

(1) 设有两个集合 A, B,则由 A 和 B 的所有共同元素构成的集合,称为 A 和 B 的交集,表示为 $A \cap B$。

例如, $A = \{a, b, c\}$, $B = \{c, d, e, f\}$,则 $A \cap B = \{c\}$。

(2) 设有两个集合 A, B,则所有属于 A 或属于 B 的元素组成的集合,称为 A 和 B 的并集,表示为 $A \cup B$。

例如, $A = \{a, b\}$, $B = \{7, 8\}$,则 $A \cup B = \{a, b, 7, 8\}$。

(3) 设有两个集合 A, B,则所有属于 A 而不属于 B 的一切元素组成的集合,称为 B 对 A 的补集,表示为 $A - B$。

例如, $A = \{a, b, c, d\}$, $B = \{c, d, e\}$,则 $A - B = \{a, b\}$, $B - A = \{e\}$。

设有全集 E 和集合 A,则称 $E - A$ 是集合 A 的补集,表示为 $\overline{A}$。

(4) 设有两个集合 A, B,则所有序偶 (a, b) 组成的集合,称为 A, B 的笛卡儿乘积,表示为 $A \times B$。

$$A \times B = \{(a, b) \mid a \in A \text{ 且 } b \in B\}$$

例如, $A = \{a, b, c\}$, $B = \{0, 1\}$,则

$$A \times B = \{(a, 0), (a, 1), (b, 0), (b, 1), (c, 0), (c, 1)\}$$

序偶函数的元素排列是有顺序的,不能任意颠倒, (a, b) 和 (b, a) 是不相同的两个序偶,因此,两个序偶相等,应该是对应元素相同。例如,对于 $(a, b) = (c, d)$,应有 $a = c$ 和 $b = d$。

对任意集合 A, B, C 有如下运算律:

(1) $A \cup A = A$, $A \cap A = A$;

(2) $A \cup B = B \cup A$, $A \cap B = B \cap A$;

(3) $(A \cup B) \cup C = A \cup (B \cup C)$, $(A \cap B) \cap C = A \cap (B \cap C)$;

(4) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$, $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$;

(5) $A \cup (A \cap B) = A$, $A \cap (A \cup B) = A$;

(6) $A \cup \overline{A} = E$, $A \cap \overline{A} =$;

(7) $\overline{A \cup B} = \overline{A} \cap \overline{B}$, $\overline{A \cap B} = \overline{A} \cup \overline{B}$;

(8) $E \cup A = E$, $E \cap A = A$;

(9) $A \cup \quad = A$, $A \cap \quad = \quad$ 。

1.2 计数方法

1.2.1 加法法则及乘法法则

在研究计数时经常要用到两个最基本的法则,一个是加法法则,另一个是乘法法则。

1. 加法法则

若具有性质 A 的事件有 m 个,具有性质 B 的事件有 n 个,则当 A 和 B 是无关的二类时,具有性质 A 或性质 B 的事件有 $m+n$ 个。

2. 乘法法则

若具有性质 A 的事件有 m 个,具有性质 B 的事件有 n 个,则当 A 和 B 相互独立时,具有性质 A 与性质 B 的事件有 $m \cdot n$ 个。

例 1.1　设 A 到 B 有 3 条不同的路径, B 到 C 也有 3 条不同的路径,求从 A 经过 B 到 C 的路径数。

解　如图 1.1 所示,假定前面 3 条由 A 到 B 的路径分别为 a, b, c,后面 B 到 C 的路径分别为 1,2,3,则 A 到 C(经过 B)的路径为

$$a1, a2, a3; b1, b2, b3; c1, c2, c3$$

所以从 A 经过 B 到 C 共有 $3 \times 3 = 9$ 条不同的路径。

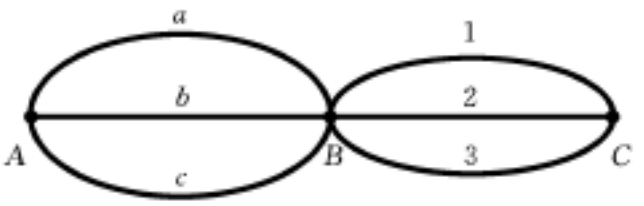


图 1.1　A 经过 B 到 C 的路径

例 1.2　求布尔函数 $f(x_1, x_2, \cdots, x_n)$ 的数目。

解　根据乘法法则,长度为 n 的 $(0,1)$ 符号串 $x_1 x_2 \cdots x_n$, $x_i = 0, 1$, $i = 1, 2, \cdots, n$, 共有 $2 \times 2 \times \cdots \times 2 = 2^n$ 个,记这些符号串分别为 $a_1, a_2, \cdots, a_{2^n}$。若以 $n = 2$ 为例,则 $a_1 = 00$, $a_2 = \underbrace{}_{n\text{个}}$ 01, $a_3 = 10$, $a_4 = 11$。$f(a_j)$ 可取值 0 或 1, $j = 1, 2, 3, 4$, 则 $n = 2$ 的布尔函数数目应为 $2^4 = 16$。

同理, n 个变元的布尔函数数目为 2^{2^n}。当 $n = 4$ 时, $2^4 = 16$, 故 4 个布尔变量 x_1, x_2, x_3, x_4 的布尔函数数目为 $2^{16} = 65536$。

例 1.3　$n = 7^3 \times 11^2 \times 13^4$, 求除尽 n 的数的个数。

解　由于除尽 n 的数可写成 $7^{a_1} \times 11^{a_2} \times 13^{a_3}$, $0 \leq a_1 \leq 3, 0 \leq a_2 \leq 2, 0 \leq a_3 \leq 4$, 故除尽 n 的数的个数为 $4 \times 3 \times 5 = 60$。

1.2.2 一一对应

一一对应是在计数过程中经常用到的方法之一。若问题 A 和问题 B 一一对应, A 不容

易求解,而 B 较容易求解,则可以通过对 B 的计数,反过来得到 A 的解。

例 1.4 在集合 $A=\{a_i \mid a_i$ 为整数,$1\leq i\leq 100\}$ 中,问求最大元素的过程中需要做多少次比较?

解 考虑到每做一次比较就淘汰一个数,则从 100 个数中找出其中最大的数 M,必须淘汰 99 个数,故须做 99 次比较。

例 1.5 证明: n 个有标号 $1,2,\cdots,n$ 的顶点的树的数目等于 n^{n-2}。

证明 对这个结论有许多不同的证明方法,下面采用一一对应的办法来证明。

假定 T 是其中一棵树,树叶中标号最小者设为 a_1,a_1 的邻接点为 b_1,消去点 a_1 和边(a_1,b_1),点 b_1 便成为余下的树的树叶。在余下的树中继续寻找标号最小的树叶,设为 a_2,a_2 以边(a_2,b_2)与 b_2 相连接,再消去 a_2 及边(a_2,b_2),继续以上的步骤 $n-2$ 次,直到最后剩下一条边为止。于是,一棵树对应一个序列

$$b_1,b_2,\cdots,b_{n-2}$$

$b_1,b_2,\cdots,b_{n-2}$ 是 1 到 n 中的数,且并不是一定不相同。反之,任给一个序列

$$b_1,b_2,\cdots,b_{n-2} \tag{1}$$

其中,$1\leq b_i\leq n$, $i=1,2,\cdots,n-2$,由此便可找到与之对应的树 T,方法如下:从序列

$$1,2,\cdots,n \tag{2}$$

中找到第 1 个不出现在 $b_1,b_2,\cdots,b_{n-2}$ 中的数,这个数显然就是 a_1,同时找出树 T 的边(a_1,b_1),从序列(1)和序列(2)中分别消去 b_1 和 a_1。在余下的序列中继续以上的步骤 $n-2$ 次,直到序列(1)为空为止。这时,将序列(2)剩下的两个数设为 a_k 和 b_k,则边(a_k,b_k)也是树 T 的最后一条边。于是,便得到树 T。

这就证明了 n 个标号顶点的树和 $n-2$ 个数

$$b_1,b_2,\cdots,b_{n-2}$$

一一对应,其中 $1\leq b_i\leq n$, $i=1,2,\cdots,n-2$。根据乘法法则,序列(1)的数目为 n^{n-2} 个。结论得证。

1.2.3 排列

设 $A=\{a_1,a_2,\cdots,a_n\}$ 是 n 个不同元素的集合,r 满足 $0\leq r\leq n$,任取 A 中 r 个元素按顺序排成一列,称为从 A 中取 r 个的一个排列。

例如,$A=\{a,b,c,d\}$,$r=3$,从 A 中取 3 个元素排列的全体如下:

$$abc,\ acb,\ bac,\ bca,\ cab,\ cba,$$
$$abd,\ aab,\ bad,\ bda,\ dab,\ dba,$$
$$acd,\ adc,\ cad,\ cda,\ dac,\ dca$$

总数为 24 个。

令 P_r^n 表示从 n 个元素中取 r 个元素排列的全体数目,也用 $P(n,r)$ 来表示。

从 n 个元素中取 r 个元素排列,可以看作是与下面的问题一一对应。

图 1.2 所示为有编号的 r 个盒子,设 A 为有标志 $1,2,\cdots,n$ 的 n 个球,从 A 中取出一个球放在第 1 个盒子中,从余下的 $n-1$ 个球中取另一个放在第 2 个盒子中,以此类推,最后,从 $n-r+1$ 个余下的球中取一个放到第 r 个盒子中。由此可得从 A 中取 r 个盒子的一个排列。

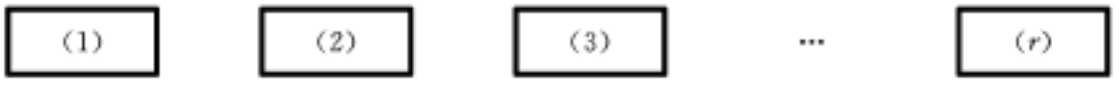


图 1.2　r 个盒子

第 1 个盒子有 n 种出现的可能,第 2 个盒子有 $n-1$ 种出现的可能,……第 r 个盒子有 $n-r+1$ 种出现的可能,根据乘法法则:

$$P_r^n = n(n-1)(n-2)\cdots(n-r+1)$$

由于 $n! = (n-1)(n-2)\cdots 2\times 1$,所以有

$$P_r^n = n!/(n-r)!$$

为了方便起见,令 $0! = 1$,所以

$$P_0^n = 1,\ P_n^n = n!$$

例 1.6　由 5 种颜色的星状物、20 种不同的花排成如下的图案:两边是星状物,中间是 3 朵花。问共有多少种这样的图案?

解　5 种颜色的星状物取两个排列的排列数为

$$P_2^5 = 5\times 4 = 20$$

20 种不同的花,取 3 种排列的排列数为

$$P_3^{20} = 20\times 19\times 18 = 6840$$

根据乘法法则,共有图案数为

$$20\times 6840 = 136800$$

例 1.7　A 单位有 7 位代表,B 单位有 3 位代表,排成一列合影,要求 B 单位 3 个人排在一起。问有多少种不同的排列方案?

解　B 单位 3 个人的某一排列为 1 个元素参加单位 A 进行排列,可得方案数为

$$(7+1)! = 8!$$

但 B 单位 3 个人共有 $3!$ 种排列。根据乘法法则,总排列方案数为

$$8!\times 3!$$

例 1.8　求 20000 到 70000 间的偶数中由不同数字组成的 5 位数的个数。

解　假定所求的数为 $abcde$ 这 5 位数,其中 $2\leq a\leq 6$, $e\in\{0,2,4,6,8\}$。

(1) 若 $a\in\{2,4,6\}$,则 e 有 4 种选择,bcd 有 $P_3^{10-2} = P_3^8$ 种选择。根据乘法法则,有 $3\times 4\times P_3^8 = 4032$ 种可能。

(2) 若 $a\in\{3,5\}$,则 e 有 5 种选择。bcd 依然有 P_3^8 种选择。根据乘法法则,有 $2\times 5\times P_3^8 = 3360$ 种可能。

根据加法法则,总个数为 $4032+3360 = 7392$。

例 1.9　随机地选择 $n(n<365)$ 个人,求其中至少有两人生日相同的概率。

解　n 个人生日不同的排列数为

$$P(365,n) = \frac{365\times 364\times\cdots\times 3\times 2\times 1}{(365-n)!}$$

n 个人生日的序列可能有 365^n 种,故 n 个人在一起生日不同的概率为

$$P(365,n)/365^n = \frac{365\times 364\times\cdots\times(365-n+1)}{365^n}$$

n 个人中有相同生日的概率为

$$1 - \frac{365 \times 364 \times \cdots \times (365 - n + 1)}{365^n}$$

若从 n 个元素中取 r 个元素沿一圆周排列,则称为周圆排列,或简称为圆排列。

从 n 个元素中取 r 个元素沿一圆周排列的排列数用 Q_r^n 表示,或表示为 $Q(n, r)$。

圆排列与排列的不同之处在于圆排列头尾相邻,比如,4 个元素 a, b, c, d 可以有 4 种不同的排列:

$$abcd, \quad dabc, \quad cdab, \quad bcda$$

但在圆排列中则是一回事,属于同一个圆排列。

从 n 个元素中取 r 个元素作圆排列的排列数 Q_r^n 与 P_r^n 的关系是

$$Q_r^n = P_r^n / r$$

这个道理很明显,将取 r 个元素作排列的结果与圆排列比较,每个排列重复了 r 次。同理,

$$Q_n^n = \frac{n!}{n} = (n - 1)!$$

例 1.10　5 颗红色珠子、3 颗蓝色珠子装在圆板的四周,试问:有多少种方案?若蓝色珠子不相邻则又有多少种排列方案?蓝色珠子在一起又如何?

解　若不加限制,则有 $Q_8^8 = 7!$ 种排列方案。

若要求蓝色珠子在一起,即把它们看作是一个元素进行圆排列,则为

$$Q_6^6 = 5!$$

最后讨论蓝色珠子不相邻的情况。5 颗红色珠子的圆排列有 4! 种。第 1 个蓝色珠子有 5 种选择;第 2 个蓝色珠子为了避免与第 1 个蓝色珠子相邻,只能有 4 种选择;第 3 个蓝色珠子只有 3 种选择,故有

$$4! \times 5 \times 4 \times 3 = 1440$$

例 1.11　5 对夫妇出席一次宴会,围一圆桌坐下,试问:有几种不同的方案?若要求每对夫妻相邻又有多少种方案?

解　若 5 对夫妻 10 个人围圆桌而坐(没有限制条件),则此问题解为

$$Q_{10}^{10} = 9! = 362880$$

若加上限制条件——夫妻相邻而坐,但可以交换座位,则可将其看成是 5 个元素的圆排列,排列数为 4!。根据乘法法则可得其总方案数为

$$2^5 \times 4! = 32 \times 24 = 768$$

1.2.4　组合

若从 n 个元素中取出 r 个而不考虑它的顺序,则称之为从 n 中取 r 的组合。其数目记为 $C(n, r)$, C_r^n 或 $\begin{pmatrix} n \\ r \end{pmatrix}$。

组合问题可以看作是:球有标志 $1, 2, \cdots, n$, 盒子则没有区别,从 n 个球中取 r 个放到 r 个盒子里,每个盒子一个,便得到 n 取 r 的组合。

若在每一种组合结果的基础上对盒子进行排列,便得到 n 取 r 的排列,则有

$$P(n, r) = C(n, r) \cdot r!$$

或

$$P_r^n = C_r^n \cdot r!$$

$$C_r^n = \frac{n!}{(n-r)!r!}$$

例 1.12　甲和乙两单位共有 11 个成员,其中甲单位 7 人,乙单位 4 人,拟从中组成一个 5 人小组:

(1) 要求必须包含乙单位 2 人;

(2) 要求至少包含乙单位 2 人;

(3) 要求乙单位某一人与甲单位特定一人不能同时在一个小组。

试分别求各有多少种方案。

解　(1)

$$n_1 = \binom{4}{2}\binom{7}{3} = \frac{4!}{2!2!} \times \frac{7!}{3!4!} = 210$$

(2) 若乙单位 2 人,甲单位 3 人,共 210 种方案;若乙单位 3 人,甲单位 2 人,则方案数为

$$\binom{4}{3}\binom{7}{2} = \frac{4!}{3!} \times \frac{7!}{2!5!} = 84$$

最后乙单位 4 人,甲单位 1 人,共 $\binom{4}{4}\binom{7}{1} = 7$ 种方案。

根据加法法则,总方案数为

$$210 + 84 + 7 = 301$$

(3) 将甲单位某人设为 A,不与乙单位的 B 分在一组,这是指从不附加其他限制条件下的组合中,去掉 A 和 B 在一起的情况。至于 A 和 B 在一起的组合,可以考虑先将 A 与 B 排除在外,余下 9 人中取 3 人组合,然后将 A 与 B 加上去,所以 A 和 B 不在一起的组合数为

$$\binom{11}{5} - \binom{9}{3} = 462 - 84 = 378$$

例 1.13　假定有 $a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8$ 八位成员,两两配对分成 4 组,试求方案 N。

解　方法一　a_1 选择其同伴有 7 种可能,余下 6 人中一人选择其同伴只有 5 种可能,余下 4 人,其中一人选择同伴有 3 种选择可能,所以共有方案数为

$$N = 7 \times 5 \times 3 = 105$$

方法二　将 8 个成员进行全排列,共有 8! 种可能。若将它分成 4 组

$$\{12\}, \{34\}, \{56\}, \{78\}$$

其中,若 1 和 2,3 和 4,5 和 6,7 和 8 互换,对于选择同伴没有影响,则全排列中选择同伴有重复,其重复数为 2^4。

同时,4 组之间的排列也不影响同伴关系,故全排列中对同伴关系也有重复,则其重复数为 4!。故得

$$N = \frac{8!}{4!2^4} = \frac{8 \times 7 \times 6 \times 5}{16} = 105$$

方法三　将 8 人分成 4 组,第 1 组有 $\binom{8}{2}$ 种选择;余下 6 人,第 2 组有 $\binom{6}{2}$ 种选择;同理,

第 3 组有$\binom{4}{2}$种选择。根据乘法法则,共有

$$N=\binom{8}{2}\binom{6}{2}\binom{4}{2}\binom{2}{2}\Big/4!=\frac{8\times7\times6\times5\times4\times3}{2^3\times4!}=105$$

式中,除以 4 !是因为其中 4 !组与顺序无关。

在组合的定义中若增加一些限制条件,则会产生一些新的计数概念和方法,最常见的有允许重复的组合和不相邻的组合。

1 . 允许重复的组合

允许重复的组合指的是从 $A=\{1,2,\cdots,n\}$ 中取 r 个元素$\{a_1,a_2,\cdots,a_r\}$,$a_i\in A$, $i=1,2,\cdots,r$,而且允许 $a_i=a_j$, $i\neq j$。

例如, $A=\{1,2,3\}$,若取 2 个作允许重复的组合,则除了不重复的组合$\{1,2\}\{1,3\}\{2,3\}$之外,还有$\{1,1\}\{2,2\}\{3,3\}$。

定理 1 .1　在 n 个不同的元素中取 r 个进行组合,若允许重复,则组合数为 $C(n+r-1,r)$。

证明　只要证明允许重复的组合与从 $n+r-1$ 个不同的元素中取 r 个作不重复的组合一一对应,定理就获得了证明。

先证明从 n 中取 r 个可重复组合和从 $n+r-1$ 中取 r 个的不允许重复的组合一一对应。

不失一般性,假定有 n 个不同元素分别为 $1,2,\cdots,n$,从中取 r 个作允许重复的组合 a_1, a_2, $\cdots$, a_r,由于存在重复,故

$$a_1\leq a_2\leq\cdots\leq a_r$$

从$(a_1,a_2,\cdots,a_r)$引出$(a_1,a_2+1,\cdots,a_k+k-1,\cdots,a_r+r-1)$,使每一允许重复的组合$(a_1,a_2,\cdots,a_r)$对应于一个不允许重复的组合$(a_1,a_2+1,\cdots,a_k+k-1,\cdots,a_r+r-1)$,令后者为$(b_1,b_2,\cdots,b_r)$,即 $b_k=a_k+(k-1)$, $k=1,2,\cdots,r$,其中 $b_1<b_2<\cdots<b_r\leq n+r-1$,则从 1 到 n 中取 r 个允许重复的组合,对应一个从 1 到 $n+r-1$ 中取 r 的不允许重复的组合。

反之,从$(1,2,\cdots,n+r-1)$中取 r 个作不重复的组合$(b_1,b_2,\cdots,b_r)$,不妨设 $b_1<b_2<\cdots<b_r\leq n+r-1$,构造序列 $b_1,b_2-1,b_3-2,\cdots,b_r-r+1$,显然有

$$b_1\leq b_2-1\leq b_3-2\leq\cdots\leq b_r-r+1$$

令 $a_k=b_k-k+1$, $k=1,2,\cdots,r$,则

$$a_1\leq a_2\leq\cdots\leq a_r\leq n$$

从 1 到 $n+r-1$ 中取 r 个作不重复的组合,和从 1 到 n 中取 r 个作允许重复的组合一一对应。这就证明了在 n 个不同元素中取 r 个进行允许重复的组合,其组合数为 $C(n+r-1,r)$。

允许重复组合的典型模型是:取 r 个无区别的球,放进 n 个有标志的盒子,每个盒子中的球数不加限制,则允许重复的组合数即为其方案数。

定理 1 .2　将 r 个无区别的球放进 n 个有标志的盒子,每个盒子中可多于一个,则共有 $C(n+r-1,r)$种方案。

例 1 .14　试问$(x+y+z)^4$有多少项?

解　$(x+y+z)^4=(x+y+z)(x+y+z)(x+y+z)(x+y+z)$的展开式是右端每一项的一个元素相乘的结果。每一项都是无区别的,可取 x、y 或 z,可以理解为将无区别的球投

进有标志的盒子,而且每盒的球数不限。例如,x^4可以理解为 4 个球都放进标志为 x 的盒子,标志为 y 和 z 的盒子为空。x^2yz 相当于两个球放进标志为 x 的盒子,另外两个盒子 y 和 z 各一球。所以,这个问题相当于 $n=3, r=4$,作允许重复组合,其组合数为

$$N = C(3 + r - 1, 4) = C(6, 4) = 6 \times 5 / 2 = 15$$

即$(x+y+z)^4$共 15 项:

$$x^4, x^3y, x^3z, x^2yz, x^2y^2, x^2y^2, x^2z^2, xy^3, xz^3, xyz^2, xy^2z, y^2z^2, y^3z, z^3y, z^4, y^4$$

2. 不相邻的组合

不相邻的组合指的是从序列 $A=\{1,2,\cdots,n\}$中取 r 个,其中不存在 $i, i+1$ 两个相邻的数同时出现于一个组合中的组合,例如,$A=\{1,2,3,4,5,6,7\}$,取 3 个作不相邻的组合,有

$$\{1,3,5\}\{1,3,6\}\{1,3,7\}\{1,4,6\}\{1,4,7\}\{1,5,7\}\{2,4,6\}\{2,4,7\}\{3,5,7\}$$

定理 1.3　从 $A=\{1,2,\cdots,n\}$中取 r 个作不相邻的组合,其组合数为

$$\binom{n-r+1}{r}$$

证明　设 $B=\{b_1,b_2,\cdots,b_r\}$是所求的不相邻的一个组合,不妨假定 $b_1<b_2<b_3<\cdots<b_r$,令 $c_1=b_1, c_2=b_2-1, c_3=b_3-2,\cdots,c_r=b_r-r+1$,则

$$\{c_1,c_2,\cdots,c_r\}$$

可以是$\overline{A}=\{1,2,\cdots,n-r+1\}$的一个组合。这证明了 A 的一个不相邻组合,对应于$\overline{A}=\{1,2,\cdots,n-r+1\}$的一个组合。

反之,对于$\overline{A}$的一个组合

$$\{d_1,d_2,\cdots,d_r\}\qquad d_1<d_2<d_3<\cdots<d_r\leq n-r+1$$

对应于一个不相邻的组合:

$$\{d_1,d_2+1,d_3+2,\cdots,d_r+r-1\}\qquad d_r+r-1\leq n$$

而且

$$(d_{i+1}+i)-(d_i+i-1)=d_{i+1}-d_i+1>1\qquad i=1,2,\cdots,r-1$$

1.3 母　函　数

母函数是计数的重要工具,它在第 3 章的解递归关系式中也有重要的应用。

1.3.1 母函数的性质及应用

定义 1.2　设 $a_0,a_1,a_2,\cdots,a_n,\cdots$是一数列,定义它的母函数为幂级数

$$f(x) = a_0 + a_1x^1 + a_2x^2 + \cdots + a_nx^n + \cdots$$

虽然上面的幂级数有收敛问题,但在这里暂不考虑收敛性。我们的目的是将任意一个数列用另一种形式表示,即将数列想象为幂级数形式。这里将 $x^0=1, x^1, x^2,\cdots,x^n,\cdots$解释为代数符号或者是区分序列中不同项 $a_0,a_1,a_2,\cdots,a_n,\cdots$的形式记号。

例如,令 m 是正整数,则二项式系数序列

$$\binom{m}{0}\binom{m}{1}\binom{m}{2},\cdots,\binom{m}{m}$$

的母函数 $f_m(x)=\binom{m}{0}+\binom{m}{1}x+\binom{m}{2}x^2+\cdots+\binom{m}{m}x^m$。根据二项式定理，$f_m(x)=(1+x)^m$。更一般地，令 a 是实数，则二项式系数序列

$$\binom{a}{0},\binom{a}{1},\binom{a}{2},\cdots,\binom{a}{n},\cdots$$

的母函数

$$f_a(x)=\binom{a}{0}+\binom{a}{1}x+\binom{a}{2}x^2+\cdots+\binom{a}{n}x^n+\cdots$$

根据广义二项式定理

$$f_a(x)=(1+x)^a$$

1．母函数的基本公式

下面给出在一定条件下母函数与母函数之间存在的一些重要关系。假定序列$\{a_i\}$的母函数为 $A(x)$；序列$\{b_i\}$的母函数为 $B(x)$，则有

(1) 若 $b_k=\begin{cases}0 & k<l\\ a_{k-l} & k\geq l\end{cases}$

则

$$B(x)=x^lA(x)$$

证明　因为序列$\{b_i\}$前面 l 项为 0，即

$$b_0=b_1=\cdots=b_{l-1}=0$$

$$b_l=a_0$$

$$b_{l+1}=a_1$$

$$\cdots$$

$$b_k=a_{k-l}$$

$$\cdots$$

$$\begin{aligned}B(x)&=b_0+b_1x+\cdots+b_{l-1}x^{l-1}+b_lx^l+x^{l+1}+\cdots\\&=0+0+\cdots+0+a_0x^l+a_1x^{l+1}+\cdots\\&=x^l(a_0+a_1x+a_2x^2+\cdots)\\&=x^lA(x)\end{aligned}$$

(2) 若 $b_k=a_{k+l}$，则

$$B(x)=\left[A(x)-\sum_{k=0}^{l-1}a_kx^k\right]\Big/x^l$$

证明　因为

$$\begin{aligned}B(x)&=b_0+b_1x+b_2x^2+\cdots\\&=a_l+a_{l+1}x+a_{l+2}x^2+\cdots\\&=\frac{1}{x^l}(a_lx^l+a_{l+1}x^{l+1}+a_{l+2}x^{l+2}+\cdots)\\&=\frac{1}{x^l}[A(x)-(a_0+a_1x+a_2x^2+\cdots+a_{l-1}x^{l-1})]\\&=\left[A(x)-\sum_{k=0}^{l-1}a_kx^k\right]\Big/x^l\end{aligned}$$

(3) 若 $b_k=\sum_{i=1}^{k}a_i$，则

$$B(x)=\frac{A(x)}{1-x}$$

证明　因为

$$
\begin{aligned}
1:\ & b_0 = a_0 \\
x:\ & b_1 = a_0 + a_1 \\
x^2:\ & b_2 = a_0 + a_1 + a_2 \\
+)\ & \cdots \\
\hline
B(x) = {} & a_0(1 + x + x^2 + \cdots) \\
& + a_1 x(1 + x + x^2 + \cdots) \\
& + a_2 x^2(1 + x + x^2 + \cdots) + \cdots \\
= {} & (1 + x + x^2 + \cdots)(a_0 + a_1 x + a_2 x^2 + \cdots) \\
= {} & \frac{A(x)}{1 - x}
\end{aligned}
$$

(4) 若 $\sum_{k=0}^{\infty} a_k$ 收敛, $b_k = \sum_{j=k}^{\infty} a_j$, 则

$$B(x) = \frac{A(1) - xA(x)}{1 - x}$$

类似于对(3)作形式演算,因而不难得到上面的结果。

$$
\begin{aligned}
1:\ & b_0 = a_0 + a_1 + a_2 + \cdots = A(1) \\
x:\ & b_1 = a_1 + a_2 + \cdots = A(1) - a_0 \\
x^2:\ & b^2 = a_2 + \cdots = A(1) - a_0 - a_1 \\
+)\ & \cdots \\
\hline
B(x) = {} & A(1)(1 + x + x^2 + \cdots) - a_0 x(1 + x + x^2 + \cdots) \\
& - a_1 x^2(1 + x + x^2 + \cdots) - \cdots \\
= {} & [A(1) - x(a_0 + a_1 x + a_2 x + \cdots)]/(1 - x) \\
= {} & \frac{A(1) - xA(x)}{1 - x}
\end{aligned}
$$

反过来,通过 $(1 - x)B(x) = A(1) - xA(x)$,比较等式两端,可得

$$b_0 = A(1) = \sum_{i=0}^{\infty} a_i$$

$$b_1 = b_0 - a_0 = \sum_{i=1}^{\infty} a_i,\ b_1 - b_0 = -a_0$$

$$b_2 = b_1 - a_1 = \sum_{i=2}^{\infty} a_i,\ b_2 - b_1 = -a_1$$

最后,用数学归纳法证明一般结果。

(5) 若 $b_k = ka_k$,则 $B(x) = xA'(x)$。结论十分明显,证明从略。

(6) 若 $b_k = \frac{a_k}{1 + k}$,则

$$B(x) = \frac{1}{x}\int_0^x A(x)\,\mathrm{d}x$$

结论很显然,证明从略。

(7) 若 $c_k = a_0 b_k + a_1 b_{k-1} + a_2 b_{k-2} + \cdots + a_k b_0$, $k = 0,1,2,\cdots$,则

$$C(x) = c_0 + c_1 x + c_2 x^2 + \cdots = A(x) B(x)$$

证明　因为

$$\begin{aligned}
&1: c_0 = a_0 b_0\\
&x: c_1 = a_0 b_1 + a_1 b_0\\
&x^2: c^2 = a_0 b_2 + a_1 b_1 + a_2 b_0\\
&+) \quad \cdots
\end{aligned}$$

$$\begin{aligned}
C(x) = {} & a_0 (b_0 + b_1 x + b_2 x^2 + \cdots)\\
& + a_1 x(b_0 + b_1 x + b_2 x^2 + \cdots)\\
& + a_2 x^2 (b_0 + b_1 x + b_2 x^2 + \cdots)\\
& + \cdots\\
= {} & (a_0 + a_1 x + a_2 x^2 + \cdots)\\
& \cdot (b_0 + b_1 x + b_2 x^2 + \cdots)\\
= {} & A(x) B(x)
\end{aligned}$$

2. 母函数的应用

$$A(x) = \frac{1}{1 - x} = 1 + x + x^2 + \cdots$$

例如

$$B(x) = \frac{1}{(1 - x)^2} = \frac{A(x)}{1 - x} = 1 + 2x + 3x + 4x + \cdots$$

$$C(x) = \frac{1}{(1 - x)^3} = \frac{B(x)}{1 - x} = 1 + 3x + 6x^2 + 10x^3 + \cdots$$

即

$$c_k = \sum_{h=1}^{k} h, \quad k = 1,2,3,\cdots$$

又如

$$\frac{1}{1 - x} = 1 + x + x^2 + \cdots$$

令

$$A(x) = \frac{1}{1 - x}$$

$$B(x) = xA'(x) = x/(1 - x)^2 = x(1 + 2x + 3x^2 + \cdots)$$

则

$$b_0 = 0, b_k = k, k \geq 1$$

例 1.15　有红球两个,白球、黄球各一个,试求有多少种不同的组合方案。

解　设 r, ω, y 分别代表红球,白球,黄球。

$$\begin{aligned}
&(1 + r + r^2)(1 + \omega)(1 + y)\\
&= (1 + r + r^2)(1 + y + \omega + y\omega)\\
&= 1 + (r + y + \omega) + (r^2 + ry + r\omega + y\omega) + (r^2 y + r^2 \omega + ry\omega) + y^2 y\omega
\end{aligned}$$

由此可见,除一个球也不取的情况外,有

(1) 取一个球组合数为 3,即分别取红、黄、白 3 种。

(2) 取两个球的组合数为 4,即两红,一黄一红,一白一红,一黄一白。

(3) 取 3 个球的组合数为 3,即两红一黄,两红一白,一红一黄一白。

(4) 取 4 个球的组合为 1,即两红一黄一白。

令取 r 个球的组合数为 C_r,则序列 C_0, C_1, C_2, C_3, C_4 的母函数为

$$G(x) = (1 + x + x^2)(1 + x)^2 = 1 + 3x + 4x^2 + 3x^3 + x^4$$

共有 1 + 3 + 4 + 3 + 1 = 12 种组合方式。

例 1.16　若有 1g、2g、3g、4g 的砝码各一枚,问能称出几种可能的重量?

解　采用如下办法计算母函数:

$$(1 + \underset{1g}{x^1})(1 + \underset{2g}{x^2})(1 + \underset{3g}{x^3})(1 + \underset{4g}{x^4})$$

$$= (1 + x + x^2 + x^3)(1 + x^3 + x^4 + x^7)$$

$$= 1 + x + x^2 + 2x^3 + 2x^4 + 2x^5 + 2x^6 + 2x^7 + x^8 + x^9 + x^{10}$$

将 1 看做 x_0,则上式各项的含意是

$$\underset{\text{取(0g 或 1g)}}{(x_0 + x)} \cdot \underset{\text{与 (0g 或 2g)}}{(x^0 + x^2)} \cdot \underset{\text{与 (0g 或 3g)}}{(x^0 + x^3)} \cdot \underset{\text{与 (0g 或 4g)}}{(x^0 + x^4)}$$

若是 + 号表达或,则乘号的意义是"与"。这里指数表示克数,系数是方案数。$2x^3$ 表示 3g 的方案有 2 个,一个是 1g 和 2g 一起,另一个是一个 3g 的砝码。

例 1.17　求 1 角、2 角、3 角的邮票可贴出不同的数值邮资的方案数的母函数。

解　由于邮票允许重复,故母函数为

$$G(x) = \underset{1\text{ 角}}{(1 + x + x^2 + \cdots)}\underset{2\text{ 角}}{(1 + x^2 + x^4 + \cdots)}\underset{3\text{ 角}}{(1 + x^3 + x^6 + \cdots)}$$

$$= \frac{1}{1 - x}\frac{1}{1 - x^2}\frac{1}{1 - x^3} = \frac{1}{1 - x - x^2 + x^4 + x^5 - x^6}$$

在 $G(x)$ 的因子 $1 + x^2 + x^4 + x^6 + \cdots$ 中,1 实际上是 $1x^0$,表示不贴 2 角邮票,x^4 表示贴两张 2 角邮票,其余类推。直接利用多项式除法可得

$$G(x) = 1 + x + 2x^2 + 3x^3 + 4x^4 + 5x^5 + 6x^6 + \cdots$$

以其中 x^4 为例,系数为 4,即 4 拆分成 1,2,3 之和的拆分数为 4,即

$$1 + 1 + 1 + 1,\quad 1 + 1 + 2,\quad 2 + 2,\quad 1 + 3$$

例 1.18　若有 1g 的砝码 3 枚,2g 的 4 枚,4g 的 2 枚,问能称出哪些重量?各有几种方案?

解　$G(x) = \underset{\text{1g 3 枚}}{(1 + x + x^2 + x^3)}\underset{\text{2g 4 枚}}{(1 + x^2 + x^4 + x^6 + x^8)}\underset{\text{4g 2 枚}}{(1 + x^4 + x^8)}$

$$= (1 + x + 2x^2 + 2x^3 + 2x^4 + 2x^5 + 2x^6 + 2x^7 + 2x^8 + 2x^9 + x^{10} + x^{11})$$
$$\cdot (1 + x^4 + x^8)$$
$$= 1 + x + 2x^2 + 2x^3 + 3x^4 + 3x^5 + 4x^6 + 4x^7 + 5x^8 + 5x^9 + 5x^{10}$$
$$+ 5x^{11} + 4x^{12} + 4x^{13} + 3x^{14} + 3x^{15} + 2x^{16} + 2x^{17} + x^{18} + x^{19}$$

上面 $1 + x^4 + x^8$ 表示 4g 的砝码有 2 枚;或不取,或取 1 枚,或取 2 枚,其他项同理说明。x^8 项系数为 5,说明称出 8g 的方案有 5 种

$$1 + 1 + 2 + 2 + 2,\quad 2 + 2 + 4,\quad 1 + 1 + 2 + 4,\quad 2 + 2 + 2 + 2,\quad 4 + 4$$

例 1.19　求整数 n 拆分成 $1, 2, \cdots, m$ 的和,并允许重复的拆分数。若其中 m 至少出现

一次,试求它的方案数及其母函数。

解　因为 n 拆分成 $1,2,\cdots,m$ 的和允许重复,故其母函数为

$$G(x)=(1+x+x^2+\cdots)(1+x^2+x^4+\cdots)\cdots(1+x^m+x^{2m}+\cdots)$$

$$=\frac{1}{1-x}\frac{1}{1-x^2}\cdots\frac{1}{1-x^m}=\frac{1}{(1-x)(1-x^2)\cdots(1-x^m)}$$

若要 m 至少出一次,则其母函数为

$$G_1(x)=(1+x+x^2+\cdots)(1+x^2+x^4+\cdots)\cdots(x^m+x^{2m}+\cdots)$$

$$=\frac{x^m}{(1-x)(1-x^2)\cdots(1-x^m)}$$

$$=\frac{1}{(1-x)(1-x^2)\cdots(1-x^{m-1})}\frac{x^m}{1-x^m}$$

$$=\frac{1}{(1-x)(1-x^2)\cdots(1-x^m)}-\frac{1}{(1-x)(1-x^2)\cdots(1-x^{m-1})}$$

上式的组合意义很明显:即整数 n 拆分成 1 到 m 的拆分数,减去 n 拆分成 1 到 $m-1$ 的拆分数,即为拆分成 1 到 m 且至少出现一个 m 的拆分数。

1.3.2　指数型母函数

定义 1.3　对于序列 $a_0,a_1,a_2,\cdots$,定义

$$G_e(x)=a_0+a_1\frac{x}{1!}+a_2\frac{x^2}{2!}+a_3\frac{x^3}{3!}+\cdots$$

为序列$\{a_i\}$的指数型母函数。

指数型母函数是计算可重复的排列问题的重要工具,例如,8 个元素中 a_1 重复 3 次,a_2 重复 2 次,a_3 重复 3 次,从中取 r 个组合,记其组合数为 C_r,则 $C_0,C_1,C_2,\cdots,C_8$ 的母函数为

$$G(x)=(1+x+x^2+x^3)(1+x+x^2)(1+x+x^2+x^3)$$

$$=(1+2x+3x^2+2x^4+x^5)(1+x+x^2+x^3)$$

$$=1+3x+6x^2+9x^3+10x^4+9x^5+6x^6+3x^7+x^8$$

由 x_4 的系数可知,从这 8 个元素中取 4 个元素组合,其组合数为 10。这 10 个组合如何构成的呢?还可以通过下列的乘积来得出:

$$(1+x_1+x_1^2+x_1^3)(1+x_2+x_2^2)(1+x_3+x_3^2+x_3^3)$$

$$=[1+(x_1+x_2)+(x_1^2+x_1x_2+x_2^2)+(x_1^3+x_1^2x_2+x_1x_2^2)+(x_1^3x_2+x_1^2x_2^2)$$

$$+x_1^3x_2^2](1+x_3+x_3^2+x_3^3)$$

其中,$x_1^3x_2+x_1^2x_2^2$ 表示这样 4 个元素的组合:3 个 a_1,1 个 a_2 和 2 个 a_1,2 个 a_2。其余以此类推。

展开式中 4 次方项有

$$x_1x_3^3+x_2x_3^3+x_1^2x_3^2+x_1x_2x_3^2+x_2^2x_3^2+x_1^3x_3+x_1^2x_2x_3+x_1x_2^2x_3+x_1^3x_2+x_1^2x_2^2$$

从 8 个元素中取 4 个进行排列:其排列数应是每一组合的排列。比如,$x_1x_3^3$ 项表示 1 个 a_1,3 个 a_3;$x_1x_2^2x_3$ 表示 1 个 a_1,两个 a_2,1 个 a_3。各自的排列数各异,$x_1x_3^3$ 对应的排列数为 $\frac{4!}{1!3!}$,而 $x_1x_2^2x_3$ 项对应的排列数则为 $\frac{4!}{1!2!1!}$,因此,得出上面所说的取 4 个元素的排列数

为

$$4!\left(\frac{1}{1!3!}+\frac{1}{1!3!}+\frac{1}{2!2!}+\frac{1}{1!1!2!}+\frac{1}{2!2!}+\frac{1}{3!1!}+\frac{1}{2!1!1!}+\frac{1}{1!2!1!}+\frac{1}{3!1!}+\frac{1}{2!2!}\right)$$

$$=4\left(\frac{4}{3!}+\frac{3}{2!2!}+\frac{3}{2!}\right)=16+18+36=70$$

利用指数型母函数可以比较方便地解决上面的问题,引入下面的函数:

$$G_e(x)=\left(1+\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}\right)\left(1+\frac{x}{1!}+\frac{x^2}{2!}\right)\left(1+\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}\right)$$

$$=1+3x+\frac{9}{2}x^2+\frac{14}{3}x^3+\frac{35}{12}x^4+\frac{17}{12}x^5+\frac{35}{72}x^6+\frac{8}{72}x^7+\frac{1}{72}x^8$$

$$=1+3\frac{x}{1!}+9\frac{x^2}{2!}+28\frac{x^3}{3!}+70\frac{x^4}{4!}+170\frac{x^5}{5!}+350\frac{x^6}{6!}+560\frac{x^7}{7!}+560\frac{x^8}{8!}$$

这是因为取 k 的排列数应是 $k!$,故取 4 个的排列数应是

$$4!\frac{35}{12}=4\times3\times2\times\frac{35}{12}=70$$

例 1.20　$a_1, a_2, \cdots, a_7$ 为 7 个有区别的球,将它们放进 4 个有标志的盒子,要求第 1,2 两盒必须含偶数个球,第 3 个盒含有奇数个球。

解　本题可以理解为从 1,2,3,4 这 4 个数字取 7 个作允许重复的排列,如图 1.3 所示。即标志为 1 的球在第 3 个盒子,标志为 2 的球在第 1 个盒子,标志为 3 的球在第 2 个盒子,等等。因此,球的标号 1,2,…,7 放进的盒子的顺序号为 3124214。

图 1.3　一种放球方法

根据上述的对应情况,可以利用指数型母函数来求解。设 r 个有标志的球的分配方案数为 a_r,则序列$\{a_r\}$的母函数为

$$G(x)=\underbrace{\left(1+\frac{1}{2!}x^2+\frac{1}{4!}x^4+\cdots\right)^2}_{\text{第 1,2 盒}}\underbrace{\left(\frac{1}{1!}x+\frac{1}{3!}x^3+\cdots\right)}_{\text{第 3 盒}}\left(1+\frac{x}{1!}+\frac{x^2}{2!}+\cdots\right)$$

$$=\left(\frac{e^x+e^{-x}}{2}\right)^2\left(\frac{e^x-e^{-x}}{2}\right)e^x$$

$$=\frac{1}{8}(e^{2x}+2+e^{-2x})(e^x-e^{-x})e^x$$

$$=\frac{1}{8}(e^{4x}+1+e^{2x}-e^{-2x})$$

$$=\frac{1}{8}\left\{-1+\sum_{i=1}^{\infty}[4^i+2^i-(-2)^i]\frac{x^i}{i!}\right\}$$

则　$a_r=\frac{1}{8}[4^r+2^r-(-2)^r]\qquad r\geq1$

例 1.21　求 1,3,5,7,9 这 5 个数字组成的 n 位数的个数,要求其中 3 和 7 出现的次数为偶数,其他数字出现的次数无限制。

解 设满足条件的 r 位数的数目为 a_r，则序列 $a_0, a_1, a_2, \cdots$ 的母函数为

$$G(x) = \underbrace{\left(1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots\right)^2}_{\text{3和7对应项}} \underbrace{\left(1 + x + \frac{x^2}{2!} + \cdots\right)^3}_{\text{1,5,9对应项}}$$

由于 $e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots$，则

$$1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots = \frac{1}{2}(e^x + e^{-x})$$

$$G_e(x) = \frac{1}{4}(e^x + e^{-x})^2 e^{3x} = \frac{1}{4}(e^{5x} + e^{3x} + e^x)$$

$$= \frac{1}{4}\sum_{n=0}^{\infty}(5^n + 2 \times 3^n + 1)\frac{x^n}{n!}$$

$$a_n = \frac{1}{4}(5^n + 2 \times 3^n + 1)$$

1.4 级数求和

1.4.1 由组合的实际意义产生的计数公式及级数求和公式

$$\binom{n}{r} = \binom{n}{n-r} = \frac{n!}{r!(n-r)!} \tag{1.1}$$

组合意义：从 n 个元素 $a_1, a_2, \cdots, a_n$ 中取走 r 个元素，余下的元素个数为 $n-r$，从 n 个中取 r 个的组合与从 n 个中取 $n-r$ 个的组合一一对应。它的特例 $\binom{m+n}{m} = \binom{m+n}{n}$。

$$C(n, r) = C(n-1, r) + C(n-r, r-1) \tag{1.2}$$

组合意义：从 n 个元素 $a_1, a_2, \cdots, a_n$ 中取 r 个组合的全体，就其中元素 a_1 来看，可分为两类，即所求的组合由以下两部分构成：

(1) 组合中含有元素 a_1。这一类组合可看做是除去 a_1，从剩下的 $n-1$ 元素中取 $r-1$ 的组合，即由从 $n-1$ 个元素 $a_2, a_3, \cdots, a_n$ 中取 $r-1$ 个组合，再加上 a_1 而形成。其组合数目为 $C(n-1, r-1)$。

(2) 不含有元素 a_1 的组合。这一类组合可看做从 $a_2, a_3, \cdots, a_n$ 中取 r 个组合，其组合数目为 $C(n-1, r)$。

$$C(n+r+1, r) = C(n+r, r) + C(n+r-1, r-1) + C(n+r-2, r-2) + \cdots + C(n+1, 1) + C(n, 0) \tag{1.3}$$

组合意义：等式左端是从 $n+r+1$ 个元素 $a_1, a_2, \cdots, a_{n+r+1}$ 中取出 r 个元素作不允许重复的组合，结果不外有以下几种情形：

(1) r 个组合元素中不含有元素 a_1 的。相当于从 $n+r$ 个元素 $a_2, a_3, \cdots, a_{n+r+1}$ 中取 r 个元素的组合，其组合数为 $C(n+r, r)$。

(2) r 个组合元素中含有 a_1 但不含有元素 a_2 的。即对含有元素 a_1 的组合中依据 a_2 元素重新分类。包含 a_1 而不包含 a_2 的组合，相当于从除去元素 a_1, a_2 后的 $n+r-1$ 个元素 a_3，

$a_4, \cdots, a_{n+r+1}$ 中取 $r-1$ 个组合，然后加上元素 a_1 而成。其组合数为 $C(n+r-1, r-1)$。

(3) r 个组合元素中含元素 a_1，a_2，但不含元素 a_3 者。相当于从除去 a_1，a_2，a_3 三元素后的 $n+r-2$ 个元素中取 $r-2$ 个元素的组合，然后加上 a_1，a_2 而成。其组合数为 $C(n+r-2, r-2)$。其余以此类推。

取出的 r 个组合元素中含有 a_1，a_2，$\cdots$，a_{r-1}，但不含有元素 a_r，相当于从 a_{r+1}，$\cdots$，a_{n+r+1} 中取 1 个元素与 a_1，a_2，$\cdots$，a_{r-1} 组合，其组合数显然为 $C(n+1, 1)$。

(4) 由 a_1，a_2，$\cdots$，a_r 组成的组合 $C(n, 0) = 1$。

从式(1.3)可得到下列级数的和：

$$S_1 = 1 + 2 + 3 + \cdots + n = \sum_{k=1}^{n} k = C(1,1) + C(2,1) + \cdots + C(n,1) \tag{1.3a}$$

由于

$$C(n,1) = C(n, n-1)$$

$$S_1 = C(1,0) + C(2,1) + C(3,2) + \cdots + C(n, n-1)$$

相当于等式中 $n=1$，$r=n-1$，所以，$S_1 = C(n+1, n-1) = \frac{1}{2}n(n+1)$。

$$S_2 = 1\times 2 + 2\times 3 + 3\times 4 + \cdots + n(n+1) = 2 + 2C(3,2) + 2C(4,2) + \cdots + 2C(n+1,2)$$

$$= 2[C(2,0) + C(3,1) + C(4,2) + \cdots + C(n+1, n-1)] \tag{1.3b}$$

相当于式(1.2)中 $n=2$，$r=n-1$，所以，$S_2 = 2C(n+2, n-1) = \frac{1}{3}n(n+1)(n+2)$。

$$S_3 = 1\times 2\times 3 + 2\times 3\times 4 + \cdots + n(n+1)(n+2)$$

$$= 3![C(3,3) + C(4,3) + \cdots + C(n+2,3)]$$

$$= 3![C(3,0) + C(4,1) + \cdots + C(n+2, n-1)]$$

$$= 3!C(n+3, n-1) = 3!C(n+3,4)$$

$$= \frac{1}{4}n(n+1)(n+2)(n+3) \tag{1.3c}$$

$$\binom{n}{l}\binom{l}{r} = \binom{n}{r}\binom{n-r}{l-r} \qquad l \geq r \tag{1.4}$$

组合意义：从 n 个元素中取 l 个组合，再在所得的每个组合的 l 个元素中取 r 个组合，由此所得组合的全体，相当于从 n 个元素中直接取 r 个组合，但有重复，其重复度为 $\binom{n-r}{l-r}$，即取重复数等于从剩下的 $n-r$ 个元素中取 $l-r$ 个组合的组合数。

$$C(m+n, 2) - C(m, 2) - C(n, 2) = mn \tag{1.5}$$

组合意义：m 个男生和 n 个女生，一男一女的组合共有 mn 种方案，其等式左端为从 $m+n$ 个人中取 2 个组合，减去两个男生的组合 $C(m, 2)$，再减去两个女生的组合 $C(n, 2)$，余下的为一男一女的组合。

$$C(m, 0) + C(m, 1) + C(m, 2) + \cdots + C(m, m) = 2^m \tag{1.6}$$

证明　根据二项式定理

$$(x+y)^m = x^m + C(m,1)x^{m-1}y + C(m,2)x^{m-2}y^2 + \cdots + y^m$$

令 $x=y=1$，式(1.6)便得到证明。

组合意义：由 m 个元素中的每一个元素“取”与“不取”构成所有状态，由乘法法则可知其

总数为 2^m。等式的左端说明这所有的状态可分解为从 m 个元素中分别取 $0,1,2,\cdots,m$ 个组合的总和。

$$C(n,0) - C(n,1) + C(n,2) - \cdots(-1)^{n-1} C(n,n) = 0 \qquad (1.7)$$

在 $(x+y)^n = x^n + C(n,1)x^{n-1}y + C(n,2)x^{n-2}y^2 + \cdots + C(n,n)y^n$ 中，令 $x=1$，$y=-1$，便得到证明。

组合意义：n 个元素中取 r 个的组合，r 为奇数的组合数之和等于 r 为偶数的组合数之和（0 作为偶数考虑在内）。

只要证明在 r 为偶数的组合和 r 为奇数的组合之间建立起一一对应关系即可。

$$\binom{m+n}{r} = \binom{m}{0}\binom{n}{r} + \binom{m}{1}\binom{n}{r-1} + \cdots + \binom{m}{r}\binom{n}{0},\ r \le \min(m,n) \qquad (1.8)$$

组合意义：设从 $m+n$ 个有标志的球中取 r 个球组合，这 $m+n$ 个球中 m 个是红的，n 个是蓝的，则一切组合不外乎以下几种可能：r 个球都是蓝的，无一红的，有 $\binom{m}{0}\binom{n}{r}$ 种方案；$r-1$ 个球是蓝的，1 个是红的，则有 $\binom{m}{1}\binom{n}{r-1}$ 种方案；……最后是 r 个红的，无一蓝的，有 $\binom{m}{r}\binom{n}{0}$ 种方案，依据加法法则得式(1.8)。

$$\binom{m+n}{m} = \binom{m}{0}\binom{n}{0} + \binom{m}{1}\binom{n}{1} + \cdots + \binom{m}{m}\binom{n}{m}, \quad m \le n \qquad (1.9)$$

若式(1.6)中 $m \le n$，$r=n$，则有

$$\binom{m+n}{m} = \binom{m}{m}\binom{n}{0} + \binom{m}{m-1}\binom{n}{1} + \cdots + \binom{m}{m}\binom{n}{m} = \binom{m}{0}\binom{n}{0} + \binom{m}{1}\binom{n}{1} + \cdots + \binom{m}{m}\binom{n}{m}$$

组合意义：设有 m 个红球，n 个蓝球，且这 $m+n$ 个球是有标志的。式(1.9)的左端是从这 $m+n$ 个球中取 m 个组合的组合数。等式右端每项形为 $\binom{m}{k}\binom{n}{k}$，表示由 k 个红球、k 个蓝球形成的组合数，即等式右端为由数目相等的红、蓝球形成的组合数。例如，$\binom{m}{1}\binom{n}{1} = mn$，为从 $m+n$ 个球中取 2 个球，一为红球，一为蓝球的组合数，等等。

从 $m+n$ 个球中取 m 个球，若其中有 k 个红球，则必有 $m-k$ 个蓝球。但 m 个红球取走 k 个，余下的红球正好也是 $m-k$ 个。所以 m 个球中的蓝色球数目和余下的红色球数目一样，于是取 m 个球，其中 k 个红色球、$m-k$ 个蓝色球的组合，和 $m-k$ 个红色球、$m-k$ 个蓝色球的组合一一对应，式(1.9)得到证明。

1.4.2 其它的一些常用求和公式

$$\frac{1}{1-x} = 1 + x + x^2 + \cdots \qquad (1.10)$$

$$\frac{1}{(1-x)^2} = \frac{1}{1-x}\cdot\frac{1}{1-x} = (1+x+x^2+\cdots)(1+x+x^2+\cdots) = 1 + 2x + 3x^2 + \cdots \qquad (1.11)$$

$$\frac{1}{(1-x)^n} = 1 + nx + \frac{n(n+1)}{2!}x^2 + \frac{n(n+1)(n+2)}{3!}x^3 + \cdots \tag{1.12}$$

其中,n 是正整数。

这些公式可以通过数学归纳法来证明。更一般的形式为广义的牛顿二项式展开:若 α 是实数,有

$$(1 \pm x)^{\alpha} = \sum_{k=0}^{\infty} \binom{\alpha}{k} (\pm x)^k$$

$$= 1 \pm \alpha x + \frac{\alpha(\alpha-1)}{2!}x^2 \pm \frac{\alpha(\alpha-1)(\alpha-2)}{3!}x^3 + \cdots$$

$$+ (-1)^k \frac{\alpha(\alpha-1)\cdots(\alpha-k+1)}{k!}x^k + \cdots$$

因此,式(1.10),(1.11),(1.12)分别是 $\alpha = -1, \alpha = -2, \alpha = -n$ 的结果。

$$e^k = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \tag{1.13}$$

证明:用台劳公式将函数 e^k 在 $x = 0$ 处展开即成,由式(1.13)可以得到下面几个公式:

$$e = 2 + \frac{1}{2!} + \frac{1}{3!} + \cdots \tag{1.13a}$$

$$\frac{e^x + e^{-x}}{2} = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots \tag{1.13b}$$

$$\frac{e^x - e^{-x}}{2} = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots \tag{1.13c}$$

$$1^2 + 2^2 + \cdots + n^2 = \frac{1}{6}n(n+1)(2n+1) \tag{1.14}$$

证明　记 $S_n = 1^2 + 2^2 + \cdots + n^2$,则有 $S_n - S_{n-1} = n^2$。

同理 $S_{n-1} - S_{n-2} = (n-1)^2$,相减得 $S_n - 2S_{n-2} + S_{n-2} = 2n - 1$。

同理 $S_{n-1} - 2S_{n-2} + S_{n-3} = 2(n-1) - 1$,相减得 $S_n - 3S_{n-1} + 3S_{n-2} - 3S_{n-3} = 2$。

同理 $S_{n-1} - 3S_{n-2} + 3S_{n-3} - 4S_{n-4} = 2$,则 $S_n - 4S_{n-1} + 6S_{n-2} - 4S_{n-3} + S_{n-4} = 0$。

$$S_0 = 0,\ S_1 = 1,\ S_2 = 5,\ S_3 = 14$$

由第 3 章的齐次递推关系式的解法可推出结论:

$$1^3 + 2^3 + \cdots + n^3 = \binom{n}{1} + 7\binom{n}{2} + 12\binom{n}{3} + 6\binom{n}{4} \tag{1.15}$$

证明　类似式(1.14)的证法,略。

习　题　一

1.1　设集合 $S = \{a, b, c\}$ 和 $R = \{0, 1, 2\}$,写出(1) $S - R, R - S$;(2) $S \times R, R \times S$。

1.2　给出下列集合的幂集:(1) $\{x, \{y\}\}$;(2) $\{\ ,\{\ \}\}$。

1.3　设集合 $S = \{0\}$,写出 S 和 $\rho(S)$ 的幂集。

1.4　设集合 A、B,全集为 E,证明(1) $A \cup B = \overline{A} \cap \overline{B}$;(2) $\overline{A \cap B} = \overline{A} \cup \overline{B}$。

1.5　设自然数集合 N 有下列子集:$A = \{1, 2, 7, 8\}$, $B = \{x \mid x^2 < 50\}$, $C = \{x \mid x$ 可被 3 整除,$0 \leq x \leq 30\}$。求下列集合:(1) $A \cup (B \cap C)$;(2) $A \cap (B \cup C)$;(3) $B - (A \cup C)$。

1.6　5 个女生,7 个男生进行排列:

(1) 若女生在一起,则有多少种不同的排列?

(2) 若女生两两不相邻,则有多少种不同的排列?

(3) 两男生 A 和 B 之间正好有 3 个女生的排列是多少?

1.7 26 个英文字母进行排列,求 x 和 y 之间有 5 个字母的排列数。

1.8 求 3000 到 8000 之间的奇数的数目,而且没有相同的数字。

1.9 计算:$1\cdot 1!+2\cdot 2!+3\cdot 3!+\cdots+n\cdot n!$

1.10 证明 $nC(n-1,r)=(r+1)C(n,r+1)$,并给予组合解释。

1.11 试证等式:$\sum_{k=1}^{n} kC(n,k)=n2^{n-1}$。

1.12 n 个完全一样的球放到 r 个有标志的盒($n\geq r$)中,无一空盒,试问:有多少种方案?

1.13 n 和 r 都是正整数,而且 $r\leq n$,试证下列等式:

(1) $P_r^n=nP_{r-1}^{n-1}$;　　(2) $P_r^n=(n-r+1)P_{r-1}^n$;

(3) $P_r^n=\frac{n}{n-r}P_r^{n-1}$, $r<n$;　　(4) $P_r^{n+1}=P_r^n+rP_{r-1}^n$;

(5) $P_r^{n+1}=r!+r(P_{r-1}^n+P_{r-1}^{n-1}+\cdots+P_{r-1}^r)$。

1.14 8 个盒子排成一列,5 个有标志的球放到盒子中,每盒最多放一个球,要求空盒不相邻,问有多少种排列方案?

1.15 甲单位有 10 个男同志、4 个女同志,乙单位有 15 个男同志、10 个女同志,由他们产生一个 7 人的代表团,要求其中甲单位占 4 人,而且 7 人中男同志有 5 个,试问:有多少种方案?

1.16 一个盒子里有 7 个无区别的白球,5 个无区别的黑球。每次从中随机取走一个球,已知前面取走 6 个,其中 3 个是白的,试求取第 6 个球是白球的概率。

1.17 6 位男宾,5 位女宾围一圆桌而坐:

(1) 女宾不相邻有多少种方案?

(2) 所有女宾在一起有多少种方案?

(3) 一位女宾 A 和两位男宾相邻又有多少种方案?

1.18 在 1 到 n 的自然数中选取不同且互不相邻的 k 个数,有多少种选取方案?

1.19 已知序列$\left\{\binom{3}{3},\binom{4}{3},\cdots,\binom{n+3}{3},\cdots\right\}$,求母函数。

1.20 已知母函数 $G(x)=\frac{3+78x}{1-3x-54x^2}$,求序列$\{a_n\}$。

1.21 已知母函数$\frac{3-9x}{1-x-56x^2}$,求对应的序列$\{a_n\}$。

1.22 求序列$\{1,0,2,0,3,0,4,0,\cdots\}$的母函数。

1.23 设 $G=1+2x^2+3x^4+4x^6+\cdots+(n+1)x^{2n}+\cdots$求$(1-x^2)G,(1-x^2)^2G$。

1.24 求下列序列的母函数:

(1) $1,0,1,0,1,0,\cdots$　　(2) $0,-1,0,-1,0,-1,\cdots$　　(3) $1,-1,1,-1,1,-1,\cdots$

1.25 已知 $a_n=\sum_{k=1}^{n+1}k^2$, $\frac{1+x}{(1-x)^3}=\sum_{n=0}^{\infty}(n+1)^2x^n$,求序列$\{a_n\}$的母函数。

1.26 已知$\{P_n\}$的母函数为$\frac{x}{1-2x-x^2}$,

(1) 求 p_0 和 p_1;

(2) 求序列$\{p_n\}$的递推关系。

1.27 已知$\{a_n\}$的母函数为$\frac{1}{1-x+x^2}$,求序列$\{a_n\}$的递推关系,并求 a_0, a_1。

第 2 章

导引与基本数据结构

凡是使用数字计算机解决过数值计算或非数值计算问题的人对于算法(algorithm)一词都不陌生,因为他们都学习和编制过一些这样或那样的算法。但是,如果要给算法下一个定义或者作稍许准确一点的描述,那么其中的大多数人都会感到是件相当棘手的事情。的确,算法和数字、计算等基本概念一样,要给它下一个严格的定义是不容易的,只能笼统地把算法定义成解一确定类问题的任意一种特殊的方法。但在计算机科学中,算法已逐渐成了用计算机解一类问题的精确、有效方法的代名词。如果对算法作稍许详细一点的非形式描述,则算法就是一组有穷的规则,它规定了解决某一特定类型问题的一系列运算。

2.1 算　　法

2.1.1 算法的重要特性

1.确定性

算法的每一种运算必须要有确切的定义,即每一种运算应该执行何种动作必须是相当清楚的、无二义性的。在算法中不允许有诸如"计算 5/0"或"将 6 或 7 与 x 相加"之类的运算,因为前者的结果是什么不清楚,而后者对于两种可能的运算应做哪一种也不知道。

2.能行性

一个算法是能行的指的是算法中有待实现的运算都是基本的运算,每种运算至少在原理上能由人用纸和笔在有限的时间内完成。整数算术运算是能行运算的一个例子,而实数算术运算则不是能行的,因为某些实数值只能由无限长的十进制数展开式来表示,像这样的两个数相加就违背能行性这一特性。

3.输入

一个算法有 0 个或多个输入,这些输入是在算法开始之前给出的量,这些输入取自特定的对象集合。

4.输出

一个算法产生一个或多个输出,这些输出是同输入有某种特定关系的量。

5.有穷性

一个算法总是在执行了有穷步的运算之后终止。

凡是算法,都必须满足以上 5 条特性。只满足前 4 条特性的一组规则不能称为算法,只能叫做计算过程。操作系统就是计算过程的一个重要例子。设计操作系统的目的是为了控制作业的运行,当没有作业时,这一计算过程并不终止,而是处于等待状态,一直等到一个新的作业进入。尽管计算过程包括这样一类重要的例子,我们还是将本书的讨论范围限制在

那些总是终止的计算过程上。

由于研究计算机算法最终的目的是有效地求出问题的解,就需要将算法投入到计算机上运行,因此,对算法的讨论不能只研究到它能在有穷步内终止就结束,而应对有穷性作进一步的研究,即对算法的效率要作出分析。例如,在象棋比赛中,对任意给定的一种棋局,都可以设计出一种算法来判断这一棋局是否可以导致赢局。这样的一个算法需要从开局起对所有棋子可能进行的移动以及相应的对策作逐一的检查。为了作出应走哪些棋着的决策,其计算步骤虽是有穷的,但实际上即使在最先进的计算机上运算也要千千万万年。由此可知,只应把那些在相当有穷步内就终止的算法投入计算机中运行,而对不能在相当有穷步内终止的算法则应另辟蹊径,免得无益耗费计算机的宝贵资源。

2.1.2 算法学习的基本内容

要制定一个算法,一般要经过设计、确认、分析、编码、检查、调试、计时等阶段,因此学习计算机算法必须涉及这些方面的内容。在这些内容中有许多都是现今重要而活跃的研究领域。为便于区别,把算法学习的内容分成以下 5 个不同的方面。

1. 如何设计算法

设计算法的工作是不可能完全自动化的。本书是要使读者学会已被实践证明是有用的一些基本设计策略。这些策略不仅在计算机科学,而且在运筹学、电气工程等多个领域都是非常有用的,利用它们已经设计出了很多精致有效的好算法。读者们一旦掌握了这些策略,也一定会设计出更多新的、有用的算法。

2. 如何表示算法

语言是交流思想的工具,设计的算法也要用语言恰当地表示出来。本书基本采用结构程序设计的方式,选择了一种名为 SPARKS 的程序设计语言来简单明了地表示算法。至于结构程序设计的内容本书并不打算具体介绍,而是将所能收集到的那些主要结构用于本书所给出的算法之中。

3. 如何确认算法

一旦设计出了算法,就应证明它对所有可能的合法输入都能给出正确的答案,这一工作称为算法确认(algorithm validation)。要指出的是,用 SPARKS 所描述的算法还不足以是一个可以立即投入机器运行的程序(关于这一点在学完了 2.2、2.3 节之后就会更清晰)。确认的目的在于使我们确信这一算法将能正确无误地工作,而与写出这一算法所用的程序语言无关。一旦证明了算法的正确性,就可将其写成程序,在将程序放到机器上运行之前,实际上还应证明程序是正确的,即证明程序对所有可能的合法输入都能得到正确的结果,这一工作称为程序证明(program proving)。这一领域是当前很多计算机科学工作者集中研究的对象,还处于相当初期的阶段。在这一领域的工作还没取得突破性的进展之前,为了增强对所编制程序的置信度,只能用对程序的测试来权宜代替。

4. 如何分析算法

在前面对有穷性的讨论中,曾提及只应把能在相当有穷步内终止的算法实际投入计算机运行。细心的读者可能当时就会觉得“相当”一词用得非常含糊,能否对有穷步给出一个数量界限呢?这实际上是我们要在这里回答的问题。执行一个算法,要使用计算机的中央

处理器(CPU)完成各种运算,还要用存储器来存放程序和数据。算法分析(analysis of algorithms)是对一个算法需要多少计算时间和存储空间作定量的分析。分析算法不仅可以预计所设计的算法能在什么样的环境中有效地运行,而且可以知道在最好、最坏和平均情况下执行得怎么样,还可以使读者对解决同一问题不同算法的有效性作出比较判断。关于算法分析更确切的表征将在 2 .2 节讨论。

5 . 如何测试程序

测试程序实际上由调试和作时空分布图两部分组成。调试(debugging)程序是在抽象数据集上执行程序,以确定是否会产生错误的结果,若有,则修改程序。但是,这一工作正如著名计算机科学家迪伊克斯特拉(E .Dijkstra)所说的那样“ 调试只能指出有错误,而不能指出它们不存在错误 ”。尽管如此,在程序正确性证明还没取得突破性进展的今天,调试仍是不可缺少且必须认真进行的一项重要工作。作时空分布图是用各种给定的数据执行调试认为是正确的程序,并测定为计算出结果所花去的时间和空间,以印证以前所做的分析是否正确和指出实现最优化的有效逻辑位置。

上述 5 个方面基本概括了学习算法所应涉及的内容。由于篇幅关系不可能面面俱到地讨论所有课题,而是将精力集中于设计和分析,对其它部分只适当述及。

最后要指出的是,本书中所介绍的算法,其绝大部分属于求解非数值计算范畴内的问题。

2 .2　分 析 算 法

分析算法是一种有趣的智力工作,它可以充分发挥人的聪明才智。更重要的是,从经济观点来看,由分析算法可以知道为完成一项任务所设计的算法的好坏,从而促使人们设计出一些更好的算法,以收到少花钱多办事、办好事的经济效果。

在讨论怎样分析算法之前,需要对算法运行的计算机类型作出假定。这对解出问题的速度影响甚大。现假定使用一台“ 通用 ”计算机,这台“ 通用 ”机就是平时所使用的顺序处理机:它每次执行程序中的一条指令,并带有容量足够的随机存取存储器,在固定的时间内可以把一个数从任一单元取出或存入。

要分析一个算法,首先就要确定使用哪些运算以及执行这些运算所用的时间。一般将这些运算分为两类。一类是基本运算,它既包括加、减、乘、除这 4 种基本的整数算术运算,也包括浮点算术、比较、对变量赋值和过程调用等。这些运算所用时间虽然不同,但一般都只花费一个固定量的时间,因此,称它们为其时间是囿界于常数的运算。另一类运算则不然,它们可能是由一些更基本的任意长序列的运算所组成。例如,两个字符串的比较运算可以是一系列字符比较指令,而字符比较指令又可使用移位和位比较指令。如果说比较一个字符的时间囿界于常数,那么比较两个字符串的时间总量就取决于它们的长度。

第二件要做的事是确定能反映出算法在各种情况下工作的数据集,即要求我们编出能产生最好、最坏和有代表性情况的数据配置,通过使用这些数据配置来运行算法,以了解算法的性能。这部分工作是算法分析最重要和最富于创造性的工作之一。有关这方面更深入的叙述将在以后讨论一些具体算法时再进行。

对一个算法要作出全面的分析可分成两个阶段来进行,即事前分析(a priori analysis)和事后测试(a posteriori testing)。由事前分析,求出该算法的一个时间限界函数(它是一些有关参数的函数);而由事后测试收集此算法的执行时间和实际占用空间的统计资料。假设在程序中的某个地方,有语句 x←x + y。在给出某种初始数据作为输入的情况下,如果要确定执行这条语句的时间总量,则基本上要求两项信息,该语句的频率计数(frequency count)(即,该语句执行的次数)和每执行一次该语句所需要的时间。这两个数的乘积就是时间总量。由于每次执行都依赖于所用的机器和程序设计语言以及它的编译程序,因此事前分析只限于确定每条语句的频率计数。频率计数与所用的机器无关,而且独立于写这算法的程序设计语言,从而可由算法直接确定。

例如,考虑下面(a)、(b)、(c)三个程序段:

```
x←x + y
```

(a)

```
for i←1 to n do
    x←x + y
repeat
```

(b)

```
for i←1 to n do
    for j←1 to n do
        x←x + y
    repeat
repeat
```

(c)

对于每个程序段,假定语句 x←x + y 仅包含在当前可以看得见的循环之中,那么,在程序段(a),此语句的频率计数为 1,在程序段(b)是 n,在程序段(c)是 n^2。这些频率计数显然具有不同的数量级。就算法分析而言,一条语句的数量级指的是执行它的频率,而一个算法的数量级则指的是它所有语句执行频率的和。假定解同一个问题的 3 个算法的数量级分别为 n,n^2 和 n^3。比较起来,自然更乐意采用第一个算法,因为它比其余两个算法要快。例如,若 n = 10,在假定所有基本运算都具有相等的工作时间的情况下,这些算法则需分别执行 10,100 和 1000 个单位时间。由以上分析可知,确定一个算法的数量级是十分重要的,它在本质上反映了一个算法所需要的计算时间。

在实际的算法分析中,往往要分析出一个算法的计算时间或频率总数,并用某种函数来表示,譬如说用一个多项式来表示。但是,由于算法本身可能相当复杂,或者受其它许多因素影响,使得在事前分析阶段根本就写不出这个多项式的完整形式,甚至连最高次项的系数都不能写出,而只能写出该项的次数并判断出这个多项式与最高次项的关系。尽管这是件令人非常遗憾的事,但幸运的是这种关系仍反映了算法在计算时间上的基本特性,关于这一点,稍后即可看出。因此,在算法的事前分析阶段,一般都致力于确定这种关系。下面给出这种关系的数学描述。

2.2.1 计算时间的渐近表示

假设某算法的计算时间是 f(n),其中变量 n 可以是输入或输出量,也可以是两者之和,还可以是其中之一的某种测度(例如,数组的维数,图的边数,等等)。g(n)是在事前分析中确定的某个形式很简单的函数,例如,n^m, $\log n$ *, 2^n, n! 等,它是独立于机器和语言的函数;

---

* 除非另有声明,本书所用的对数均以 2 为底。

而 f(n)则与机器和语言有关。

定义 2.1　如果存在两个正常数 c 和 n_0,对于所有的 $n \geq n_0$,有

$$|f(n)| \leq c|g(n)|$$

则记作 f(n) = O(g(n))。

因此,一个算法具有 O(g(n))的计算时间指的是如果此算法用 n 值不变的同一类数据在某台机器上运行,则所用的时间总是小于|g(n)|的一个常数倍。所以,g(n)是计算时间 f(n)的一个上界函数,f(n)的数量级就是 g(n)。当然,在确定 f(n)的数量级时总是试图求出最小的 g(n),使得 f(n) = O(g(n))。现在来证明一个很有用的定理。

定理 2.1　若 $A(n) = a_m n^m + \cdots + a_1 n + a_0$ 是一个 m 次多项式,则 $A(n) = O(n^m)$。

证明　取 $n_0 = 1$,当 $n \geq n_0$ 时,利用 A(n)的定义和一个简单的不等式,有

$$\begin{aligned} |A(n)| &\leq |a_m| n^m + \cdots + |a_1| n + |a_0| \\ &\leq (|a_m| + |a_{m-1}| / n + \cdots + |a_0| / n^m) n^m \\ &\leq (|a_m| + \cdots + |a_0|) n^m \end{aligned}$$

选取 $c = |a_m| + \cdots + |a_0|$,定理立即得证。

事实上,只要将 n_0 取得足够大,可以证明只要 c 是比$|a_m|$大的任意一个常数,此定理都成立。

这个定理表明,变量 n 的固定阶数为 m 的任一多项式,与此多项式的最高阶 n^m 同阶。因此,只要计算时间为 m 阶的多项式的算法,其时间都可用 $O(n^m)$来表示。例如,若一个算法有数量级为 $c_1 n^{m_1}, c_2 n^{m_2}, \cdots, c_k n^{m_k}$ 的 k 个语句,则此算法的数量级就是 $c_1 n^{m_1} + c_2 n^{m_2} + \cdots + c_k n^{m_k}$。由定理 2.1 可知,它等于 $O(n^m)$,其中 $m = \max\{m_i \mid 1 \leq i \leq k\}$。

为了说明数量级的改进对算法有效性的影响,下面举一例子:假设有解决同一个问题的两个算法,它们都有 n 个输入,分别要求 n^2 和 nlogn 次运算,那么,当 n = 1024 时,它们需要 1048576 和 10240 次运算。如果每执行一次运算所需的时间是 1μs,则在输入相同的情况下,第一个算法大约需要 1.05s,第二个算法需要 0.01s。如果将 n 增加到 2048,则运算次数就变成 4194304 和 22528,即大约需要 4.2s 和 0.02s。这表明在将 n 加倍的情况下,一个 $O(n^2)$的算法要用 4 倍长的时间来完成,而一个 O(nlogn)的算法则只要两倍多一点的时间即可完成。在一般情况下,n 值为数千是很常见的,因此,数量级的大小对算法有效性的影响是决定性的。

从计算时间上可以把算法分成两类,凡可用多项式来对其计算时间限界的算法,称为多项式时间算法(polynomial time algorithm);而计算时间用指数函数限界的算法则称为指数时间算法(exponential time algorithm)。例如,一个计算时间为 O(1)的算法,它的基本运算执行的次数是固定的,因此,总的时间由一个常数(即,零次多项式)来限界,而一个时间为 $O(n^2)$的算法则由一个二次多项式来限界。以下六种计算时间的多项式时间算法是最为常见的,其关系为

$$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3)$$

指数时间算法一般有 $O(2^n)$、O(n!)和 $O(n^n)$等,其关系为

$$O(2^n) < O(n!) < O(n^n)$$

其中,最常见的是时间为 $O(2^n)$ 的算法。当 n 取得很大时,指数时间算法和多项式时间算法在所需时间上相差非常悬殊,因为根本就找不到一个这样的 m,使得 2^n 囿界于 n^m。换言之,对于任意的 $m \geq 0$,总可以找到 n_0,当 $n \geq n_0$ 时,有 $2^n > n^m$。因此,只要有人能将现有指数时间算法中的任何一个算法化简为多项式时间算法,就取得了一个伟大的成就。

图 2.1 和表 2.1 显示了当常数为 1 时的 6 种典型计算时间函数的增长情况。从中可以看出,O(logn)、O(n)和O(nlogn)比另外 3 种时间函数的增长率慢得多。

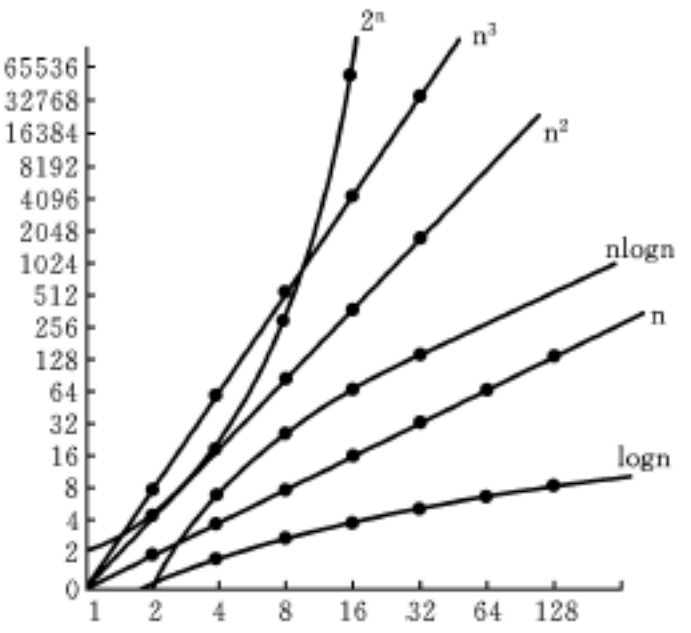


图 2.1 一般计算时间函数的曲线

由这些结果可以看出,当数据集的规模(即 n 的取值)很大时,在现代计算机上运行具有比 O(nlogn)复杂度还高的算法往往是很困难的。尤其是指数时间算法,它只有在 n 值取得非常小时才实用。因此,在顺序处理机上扩大所处理问题的规模,最有效的途径是降低算法计算复杂度的数量级,而不是提高计算机的速度。

表 2.1 计算时间函数值

logn	n	nlogn	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

符号 O 作为算法性能描述的工具,它表示计算时间的上界函数。为了进一步刻画算法的性能特性,有时也希望确定时间的下界函数,为此,引进另一个数学符号。

定义 2.2 如果存在两个正常数 c 和 n_0,对于所有的 $n > n_0$,有

$$|f(n)| \geq c|g(n)|$$

则记为 $f(n) = \Omega(g(n))$。

在某些情况下,某算法的计算时间既有 $f(n) = \Omega(g(n))$又有 $f(n) = O(g(n))$,即 g(n)既是 f(n)的上界又是它的下界。为简便起见,引进另一个数学符号来表示这种情况。

定义 2.3 如果存在正常数 c_1, c_2 和 n_0,对于所有的 $n > n_0$,有

$$c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$$

则记为 $f(n) = \Theta(g(n))$。

一个算法的 $f(n) = \Theta(g(n))$意味着该算法在最好和最坏情况下的计算时间就一个常因子范围内而言是相同的。这几种数学符号要经常使用,希望大家在此就能明确它们各自的含义。

上面仅对算法的计算时间特性作了较详细的介绍,算法计算空间的分析也可作完全类似的讨论,在此从略。

2.2.2　常用的整数求和公式

在算法分析中,在确定语句的频率时,经常会遇到以下形式的表达式:

$$\sum_{g(n)\le i\le h(n)} f(i) \tag{2.1}$$

其中,f(i)是一个带有理数系数且以 i 为变量的多项式。这个表达式最常用到的是以下几种形式:

$$\sum_{1\le i\le n} 1 \qquad \sum_{1\le i\le n} i \qquad \sum_{1\le i\le n} i^2 \tag{2.2}$$

由于它们都是有限求和,因此可列出它们的求和公式。由此可以容易地看出,第一个多项式的和数为 n。为以后使用方便,下面直接写出其余多项式的求和公式:

$$\sum_{1\le i\le n} i = n(n+1)/2 = \Theta(n^2) \tag{2.3}$$

$$\sum_{1\le i\le n} i^2 = n(n+1)(2n+1)/6 = \Theta(n^3) \tag{2.4}$$

通式是

$$\sum_{1\le i\le n} i^k = \frac{n^{k+1}}{k+1} + \frac{n^k}{2} + \text{低次项} = \Theta(n^{k+1}) \tag{2.5}$$

2.2.3　作时空性能分布图

事后测试是在对算法进行设计、确认、事前分析、编码和调试之后要做的工作,以确定程序所耗费的精确时间和空间,即作时空性能分布图。由于事后测试与所用计算机密切相关,故在此只对这一阶段所要进行的基本工作和若干注意之点概略地作一些介绍。

以作时间分布图为例,要精确地确定算法的计算时间,首先必须在所用计算机上配置一台能读出时间的时钟;还必须了解该时钟的精确程度以及计算机所用操作系统的工作方式。这是因为前者随所用计算机的不同而有相当大的差异,如果在一台时钟精确度不高的计算机上运行需时很少(譬如说比时钟的误差值还小)的程序,那么,所得的计时图只不过是一些“噪声”,其时间分布性能完全会被淹没在这些噪声之中。如果后者是以多道程序或分时方式工作的操作系统,则在取得算法工作的可靠时间上会出现困难,尤其对于那些计时中包含了换出磁盘上的用户程序要用的那部分时间的操作系统而言。由于时间随当前记入系统的用户数而变化,因此无法确定算法本身所花去的时间。

为解决因时钟误差而引起的噪声问题,下面推荐两种可供选用的方法:一是增加输入规模,直至得到算法所需的可靠的时间总量;二是取足够大的 r,将此算法反复执行 r 次,然后用 r 去除总的时间。

在解决了计时方面的具体技术问题之后,就可考虑如何作出时间性能分布图。对于事前分析为 Θ(g(n))时间的算法,应选择按输入不断增大其规模的数据集,再用这些数据集在计算机上运行程序,从而得到使用这些数据集情况下算法所耗费的时间,并画出这一数量级的时间曲线。如果这曲线与事前分析所得曲线形状基本吻合,则印证了早先分析的结论。而对于事前分析为 O(g(n))时间的算法,则应在各种规模的范围内分别按最好、最坏和平均

情况的那些数据集独立运行程序,作出各种情况的时间曲线,并由这些曲线来分析最优的有效逻辑位置。

另外,如果为了解决某一个问题,分别设计了几种具有同一数量级的不同算法,或者为加快某种算法的速度,在同一数量级情况下作了一些改进,那么,只要在输入相同数据集的情况下作出它们的时间分布图就可比较出哪一个算法的速度更快些。

2.3 用 SPARKS 语言写算法

为了便于表达算法所具有的特性,最好能用程序设计语言将算法写出来。选用语言最起码的一条要求是,由该语言所写出的每一个合法的句子都必须具有唯一的含义。程序就是用程序设计语言所表示的算法。本书中所出现的过程、子程序这样一些术语,有时也作为程序的同义词。选用何种语言来写算法呢?由于关心的是算法本身的基本思想和基本步骤,同时希望选用的语言简明、够用,写出的算法便于阅读并能容易地用人工或机器翻译成其它实际使用的程序设计语言,因此,这里选用的是 SPARKS 语言。它与 ALGOL 语言和 PASCAL 语言的形式很接近,凡是掌握了一门高级程序设计语言的人都能很快看懂并掌握 SPARKS 语言。

SPARKS 语言的基本数据类型是整型、实型、布尔型和字符型。变量只能存放单一类型的值,可以用下述形式来说明其类型:

integer x,y　　boolean a,b　　char c,d

在 SPARKS 中,有特殊含义的标识符将作为保留字来考虑,用黑体字表示。给变量命名的规则是,以字母起头,不允许使用特殊字符,且不要太长;不允许与任何保留字重复。一行可以有数条语句,但要用分号隔开。

完成对变量赋值的是赋值语句:

〈变量〉←〈表达式〉

左箭头表示把右边的值赋给左边的变量。

有两个布尔值

true　　false

为产生这两个布尔值,设置了逻辑运算符

and　　or　　not

和关系运算符

$<$　　$\leq$　　$=$　　$\neq$　　$\geq$　　$>$

SPARKS 使用带有任意整数下界和上界的多维数组。例如,一个 n 维整型数组可用以下形式说明:integer A($l_1:u_1,\cdots,l_n:u_n$),其下界是 l_i,上界是 u_i,$1\leq i\leq n$,l_i 和 u_i 都是整数或整型变量。如果某一维的下界 l_i 为 1,则在数组说明中的那个 l_i 可以不写出。例如,

integer A(5,7：20)　与　integer A(1：5,7：20)

等效。为了保持 SPARKS 语法的简明性,只使用数组作为基本结构单元来构造所有数据对象,而没有引进记录等结构类型。

条件语句具有以下形式:

if cond then S_1　**或**　if cond then S_1 endif

　　else S_2

endif

其中,cond **是一个布尔表达式**,S_1 , S_2 **是任意组** SPARKS **语句。** endif **表示条件语句的结束。条件语句的流程图由图** 2 .2 **给出。**

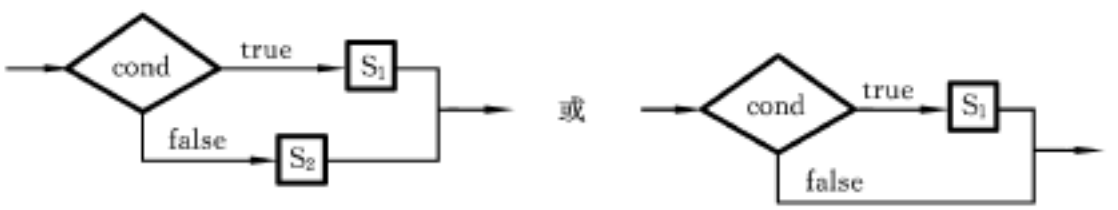


图 2 .2　if **语句**

假定布尔表达式按“短路”方式求值:对给出的布尔表达式(cond1 or cond2),**若** cond1 **为真,则不对** cond2 **求值;而对给出的布尔表达式**(cond1 and cond2),**若** cond1 **为假,则不对** cond2 **求值。**

SPARKS **中的另一种语句是** case **语句(情况语句),此语句可以很容易地把数个选择对象区别开,而无需使用多重** if—then—else **语句。它有如下形式:**

case

：cond1：S_1

：cond2：S_2

…

：cond n：S_n

：else：S_{n+1}

endcase

其中,S_i **是** SPARKS **语句组,**$1 \leq i \leq n+1$;else **子句并不是必需的。该语句的语义由下面的流程图(图** 2 .3)**所描述。**

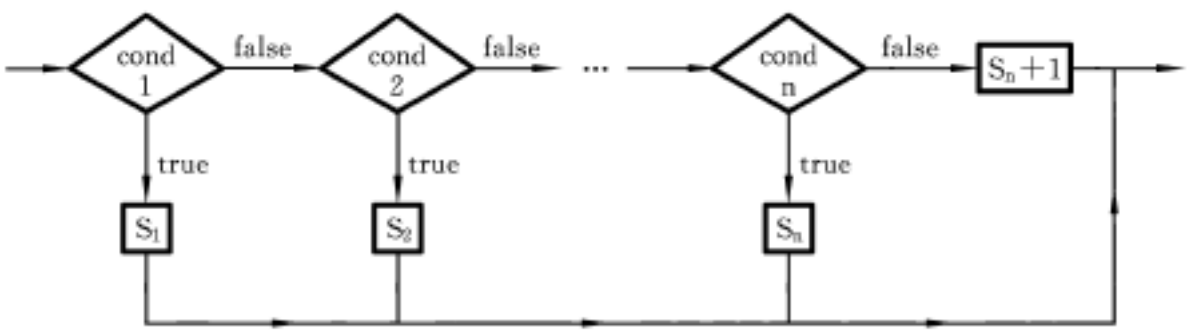


图 2 .3　case **语句**

为便于写算法,SPARKS **提供了几种可实现迭代的循环语句。**

第一种循环语句是 while **语句:**

while cond do

　　S

repeat

其中,cond **和** S **均与前面的意思相同。该语句的含义由图** 2 .4 **给出。**

第二种循环语句是 loop—until—repeat **语句:**

```
loop
S
until cond repeat
```

它有如下含义(见图 2 .5)。loop—until—repeat 语句与 while 语句相比,它保证了至少要执行一次 S 语句。

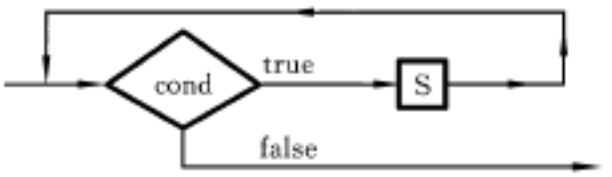


图 2 .4 while 语句

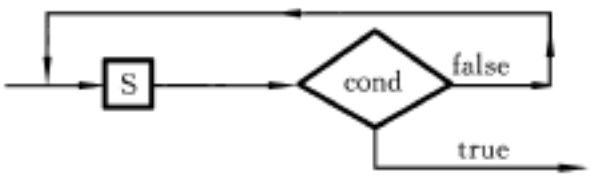


图 2 .5 loop—until—repeat 语句

第三种循环语句是 for 循环语句:

```
for vble←start to finish by increment do
    S
repeat
```

vble 是一个变量,start,finish 和 increment 是算术表达式。一个整型或实型变量或者一个常数都是算术表达式的简单形式。子句“ by increment ”不是必需的,当其没出现时就自动取 + 1。此语句的含义可以用 SPARKS 写成:

```
vble←start
fin←finish
incr←increment
while (vble-fin) * incr≤0 do
   S
   vble←vble + incr
repeat
```

注意:这些表达式只计算一次,并且将其值作为变量 vble, fin 和 incr(其中两个是新引进的变量)的值存入。这 3 个变量的类型与箭头右边表达式的类型应一致。S 代表 SPARKS 的语句序列,它不改变 vble 的值。

最后一种循环语句是 loop—until—repeat 语句的简化形式:

```
loop
  S
repeat
```

它的含义见图 2 .6。从形式上看,这语句描述了一个无限循环!然而,假定这语句和 S 中的某种检测条件一起使用,那么,这个检测条件将导致一个出口。退出这样的循环的一种方法是在 S 中使用

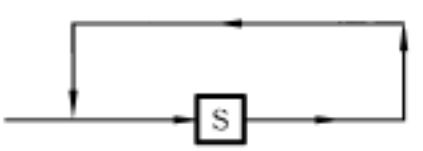


图 2 .6 loop—repeat 语句

```
go to label
```

语句,将控制转移到“ label ”。任何语句都可以附以标号,其方法是在那条语句的前面放置一个标识符和一个冒号。尽管通常并不需要 go to 语句,然而,当要将递归程序转换成迭代形式时,go to 语句则是有用的。go to 的一种受限制的形式是

exit

它的作用是将控制转移到含有 exit 的最内层循环语句后面的第一条语句。这循环语句可以是一条 while 语句,也可以是一条 loop—until—repeat 或者 for 语句。exit 能有条件地或无条件地使用。例如

loop

S_1

if cond then exit endif

S_2

repeat

其执行情况见图 2 .7。

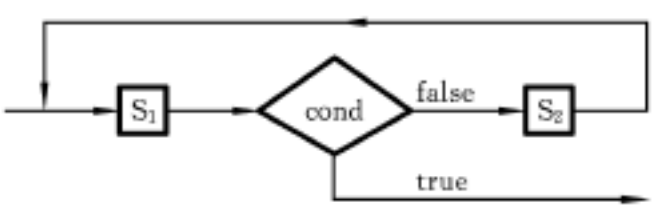


图 2 .7　带有 exit 的 loop—repeat

一个完整的 SPARKS 程序是一个或多个过程的集合,第一个过程作为主程序。执行从主程序开始。对于任一 SPARKS 过程,譬如过程 A,当到达 end 或 return 语句时,控制返回到调用过程 A 的那个 SPARKS 过程。如果过程 A 是主程序,控制则返回到操作系统。单个的 SPARKS 过程有以下形式:

procedure NAME(〈参数表〉)

〈说明部分〉

S

end NAME

一个 SPARKS 过程可以是一个纯过程(又称为子例行程序),也可以是一个函数。在这两种情况下都要对过程命名,并且把所用的形参作为一个表放在过程名后面的圆括号中。实参与形参的结合是由访问调用规则控制的,其意思是,在运行时把实参的地址送到被调用的过程中去代换与其对应的形参。对于那些是常数或表达式的实参,则假定它们存放在内部生成字中,因此将它们的地址送到被调用过程。

在函数中,返回值由放在紧接 return 的一对括号中的值来表示。例如,

return(〈表达式〉)

其中,表达式的值作为函数的值来传送。对于过程而言,end 的执行意味着执行一条没有值与其相联系的 return 语句。为了停止程序的执行,可以使用 stop 语句。

一个过程包括 3 种类型的变量:局部变量、全程变量和形参。局部变量(local variabl)是在当前过程中说明的变量。全程变量(global variabl)是在已包含当前过程的过程中说明为局部变量的变量。形参(formal parameter)是参数表中的一个标识符,由于它实际上永远不会含有值,因此它已不是一个变量,在运行时它由调用语句中对应位置的实参所代换。

对上述这 3 类变量所具有的特性是否均要给予详细的说明呢?由于需要的是能简明描述算法的基本思想与步骤的语言,故对于与此无根本损害而在计算机上付诸实现时又不可缺少的一些语法成分,则可以在写算法时采取较为“灵活”的方式。SPARKS 就具有这样的特点,它允许编制算法的人对变量说明的详略与否根据具体情况灵活处理,只要能清楚地反

映出变量的前后关系就行。下面用一个例子来加以说明。考虑求 n(n > 0) 个数的最大值的 SPARKS 过程 MAX。

算法 2 .1　求 n 项的最大元素

```
procedure MAX( A,n,j)
    ∥置 j,使 A(j)是 A(1∶n)中的最大元素,n > 0∥
    xmax←A(1) ;j←1
    for i←2 to n do
      if A(i) > xmax then xmax←A(i) ;j←i;endif
    repeat
end MAX
```

容易看出,在执行了过程 MAX 以后,对 j 换名的实参的值是最大元素在 A(1∶n)中的位置。算法中除变量 xmax 以外,其它变量中哪些是局部变量或是形参一眼就可看出。将 xmax 看成是全程变量是说得通的。假定 A(1∶n)是实数数组,那么,这个过程的完整说明就可表示如下:

```
procedure MAX( A, n, j)
    global real xmax;
    parameters integer j, n; real A(1∶n)
    local integer i;
```

正如上面说到的那样,除 xmax 外,其余变量的前后关系是很明显的,因此可以约定,在以 SPARKS 描述的算法中,除非另加说明,否则所用到的变量不是局部变量就是形参。实施不太严格说明的另一优点是提供了一个称为同质异相(polymorphism)的一般化类型。以过程 MAX 为例,由于真正关心的是算法模型的处理,因此要是不给出 A(1∶n)是整型、实型或字符型的说明,即作出同质异相的处理可能使算法模型更具有一般性。这反而是大多数程序设计语言要求必须对 A 的数据类型进行说明所不可能有的长处,使用这些程序语言,对于同一个算法模型,不得不写出 3 个独立的过程。因此,说明的这种短缺更符合要求。于是,对过程 MAX 只需作如下说明:

```
procedure MAX( A, n, j)
    global xmax
    integer i,j,n
```

过程中使用变量的分类方法,除了将变量分成局部变量、全程变量和参变量的分类方法以外,还可以有别的分类方法。为了帮助读者理解过程中各种变量的作用,下面再介绍另一种分类方法。需要指出的是,下述内容并不要求在用 SPARKS 写的算法中出现,而是在写算法过程中对所使用的变量应有的一种认识。这种分类法只与参变量和全程变量有关。从一个变量是否把值带入或带出过程这一角度出发,也可以把过程中的变量(指参变量和全程变量)分为 3 类:一类变量是只将一个值带入过程,其值在过程的整个执行期间保持不变,将其称为 in 型变量;另一类变量是在过程入口处设定义,当过程结束时给此变量赋予一个值并由其带出,这类变量称为 out 型变量;第三类变量是将一个值带入,并且将一个(可能变化了的)值带出,这类变量叫做 inout 型变量。

如果将凡是改变其参变量或全程变量的过程叫做具有边界效应(side effect) ,那么上述

分类法对于理解这一概念就很有帮助。由于纯过程没有函数值返回,且它的返回值是通过改变参变量或(和)全程变量的值来实现的,因此纯过程至少应有一个 out 或 inout 变量,即纯过程完全是通过边界效应来起作用的。函数过程也可以有边界效应,但为了避免函数的副作用,在使用 SPARKS 写算法时,约定只准写没有边界效应的函数过程或者纯过程。

那么,在什么情况下将一个算法写成函数过程或者写成纯过程呢?这与该算法在调用它的过程中的使用情况有关。如果该算法的返回值在调用它的过程的某表达式中使用,且只使用一次,则最好将其写成函数过程。例如,如果需要写一个确定两棵树是否相等的过程,就应作一个布尔函数,比方说 EQUAL(S,T),它或者返回一个真值或者返回一个假值,那么在程序中就可出现对 EQUAL 调用的下述形式:

```
if EQUAL (S,T) then ...
```

又如,要作一个计算最大公因数的函数并在赋值语句中使用它:

```
z←x * y/ gcd(x,y)
```

要是 gcd(x,y)使用的次数在一次以上,则可以将它的值赋给一个变量(t←gcd(x,y))或者将它写成一个具有边界效应的纯过程 gcd(a,b,c),然后使用语句 call gcd(x,y,t)来调用它。除了上述情况外,一般都把一个算法写成纯过程。

还要指出的是,过程对其它过程的调用是在执行了某种任务后,返回到原调用过程中的下一条语句。如果一个过程包含对自身的调用,就叫做直接递归(direct recursion)。如果一个过程调用另一过程,而这另一过程又调用原来的过程,就称为间接递归(indirect recursion)。这两种递归形式在 SPARKS 中都允许使用。

对于输入、输出,SPARKS 没给出有关格式的任何细节,因为它们对拟定算法模型是非必要的,所以输入、输出只采用两个过程:

```
read (〈参数表〉);print (〈参数表〉)
```

首尾均带有双斜线的注解可以放在一个程序中的任何地方,例如,

```
// 这是一条注解 //
```

至此,SPARKS 语言基本定义完毕,就一个程序语言而论,它还是不完整的。例如,混合算术规则、I/O 格式规则等都没提到,甚至连完整的字符集也没给出。不过这些问题对我们来说都不重要,故无需被这些问题所搅扰。最后,当用自然语言或常见的数学表示能较好地描述算法模型时,也允许这样做。带有自然语言或常见数学表示的 SPARKS 称为拟 SPARKS。

2.4　基本数据结构

要设计一个有效的算法,就必须选择或设计适合该问题的数据结构,使得算法采用这种数据结构时能对数据施行有效的运算,因此构造数据结构是改进算法的基本方法之一。一个熟练的计算机工作者必定掌握了若干种已被证明和分析过的数据结构,当他为解决某个问题而设计算法时,这些数据结构就会在他头脑中不断浮现,并从中作出选择。本节介绍了本书中最频繁使用的一些数据结构,事实上,它们在各种算法中也经常被使用。对于专门学

习过数据结构课程的读者,学习本节内容,一方面可复习一下用 SPARKS 描述的数据结构内容,另一方面或许既可弥补以前学习的不足又可对算法设计和分析取得初步的感性认识。如果还没学过数据结构,那么就请认真学习并掌握本节的全部内容。

2.4.1 栈和队列

在计算机程序中经常用到的一种最简单的数据组织形式就是有序表(又叫线性表)。它由某个集合中的 n 个元素组成,通常写成$(a_1, a_2, \cdots, a_n)$,这里 $n \geq 0$,当 $n = 0$ 时是一个空表。栈(stack)和队列(queue)是两种特殊的有序表。对栈来说,所有的插入和删除都在称为栈顶(top)的表的一端进行。而队列的所有插入只能在称为尾部(rear)的一端进行,所有的删除则只能在称为前部(front)的另一端进行。

栈的运算意味着如果依次将元素 A,B,C,D,E 插入栈,那么从栈中移出的第一个元素必定是 E,即最后进入栈的元素将首先被移出,因此栈又被称为后进先出(LIFO)表。队列的运算要求第一个插入队中的元素也第一个被移出,因此队列是一个先进先出(FIFO)表。栈和队列的一个例子见图 2.8,在该例中,栈和队都包含同样的 5 个元素并且以相同的次序插入。

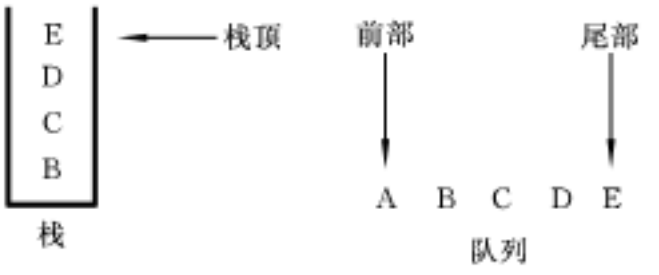


图 2.8 栈和队列的例

栈和队列的存储结构一般使用以下两种表示方法:一种是用一个一维数组表示,另一种则是使用链接表表示。现分述如下。

如果用一个一维数组 STACK(1∶n)来表示栈,则其中的 n 就是允许存入栈中元素的最大数目。于是,栈中的第一个元素(又称为存入栈底的元素)将被存放在 STACK(1),第二个元素存放在 STACK(2),第 i 个元素则存放在 STACK(i)。此时,还需要一个变量 top 作为栈顶指针,它指向该栈的顶部元素。为了测试栈是否为空,使用"if top = 0"即可。若非空,则栈顶元素在 STACK(top)。插入和删除元素是栈的两个实质性运算,可用 SPARKS 描述如下。

算法 2.2 栈运算

```
procedure ADD(item,STACK,n,top)
    // 将 item 插入规模最大为 n 的栈 STACK;top 是 STACK 中当前的元素数目 //
    if top≥n then call STACKFULL endif
    top←top + 1
    STACK(top)←item
end ADD
```

(a) 插入一个元素

```
procedure DELETE(item,STACK,top)
    // 移出 STACK 顶部元素,除非 STACK 为空,否则就将该元素存入 item //
    if top≤0 then call STACKEMPTY endif
    item←STACK(top)
    top←top - 1
```

```
end DELETE
```

(b) 删去一个元素

每执行一次 ADD 或 DELETE 过程都要花去一常量的时间,而且与栈中已存元素的个数无关。STACKFULL 和 STACKEMPTY 是两个未加详细说明的过程,这是因为它们与具体的应用有关。通常在栈满的情况下就要发出需分配更多存储单元的信号并且从过程 STACKFULL 返回;而栈空则往往是一种有意义的情况。

链接表是一些结点的集合,每个结点则由若干个信息段组成,在这些信息段中可以有一个信息段用来存放数据对象(即前面所说的元素),其余的信息段用来存放数据对象之间的顺序的链接信息。用来表示栈的链接表是一种单向链接表(又称线性链表),表中的每个结点有两个分别叫做 DATA 和 LINK 的信息段。DATA 信息段用来存放数据对象,LINK 信息段则指向在它之前刚存入的数据对象的那个结点。由于所有结点的地址都大于零,而栈底元素之前再没有元素,因此这个结点的 LINK 信息段存放一个零。例如,依次把元素 A、B、C、D、E 插入链式栈(见图 2 .9)。

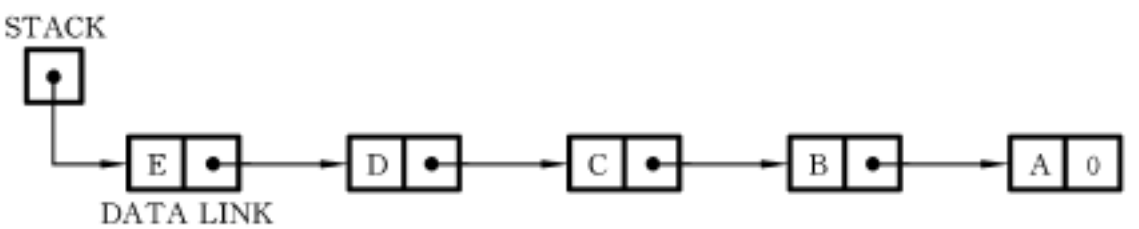


图 2 9　含有 5 个元素的链式栈

这个链式栈中还用了一个变量 STACK,它指示栈顶的那个结点(即插入的最后一项)。置 STACK = 0,表示栈为空。使用链式栈也能很容易地完成插入和删除元素的运算。插入一个元素用下面 4 条语句就可完成:

```
call GETNODE(T)
DATA(T)←item
LINK(T)←STACK
STACK←T
```

过程 GETNODE 的功能是给变量 T 分配一个可用结点,如果不再有可用结点则作相应处理,例如,终止这个程序。

删除元素可以像下述的那样处理:

```
if STACK = 0 then call STACKEMPTY endif
item←DATA(STACK)
T←STACK
STACK←LINK(STACK)
call RETNODE(T)
```

如果栈为空,则调用过程 STACKEMPTY。反之,将栈顶元素存入 item,保留栈顶指针,STACK 转而指向下一个结点,用过程 RETNODE 把刚释放的原栈顶结点存入可用结点表,准备以后提供给 GETNODE 使用。

栈的链接表示虽然比它的顺序数组表示占用较多的空间,但由于链式栈不需要一片连续单元,且多种类似的链式结构和一些别的结构可共用自由空间中的相同地方,因此使用链

接具有更大的灵活性。还要指出的是,插入和删除的计算时间与任何一种存储结构表示的栈的大小无关,且总是一个常量。

当队列用数组 Q(0：n - 1)表示时,可以把它当成一个环形来看待。插入元素是按顺时针方向将 rear 移到下一个空位置来实现的。当 rear = n - 1 且 0 位置为空时,下一个元素就存入Q(0)。front 总是指着队中第一个元素按逆时针方向转的前一个位置。只有队列为空时,才有 front = rear,开始时 front = rear = 0。图 2 .10 中描述了一个 n > 4 的数组中含有 J_1 ～J_4 4 个元素的环形队列的两种可能存放方式。

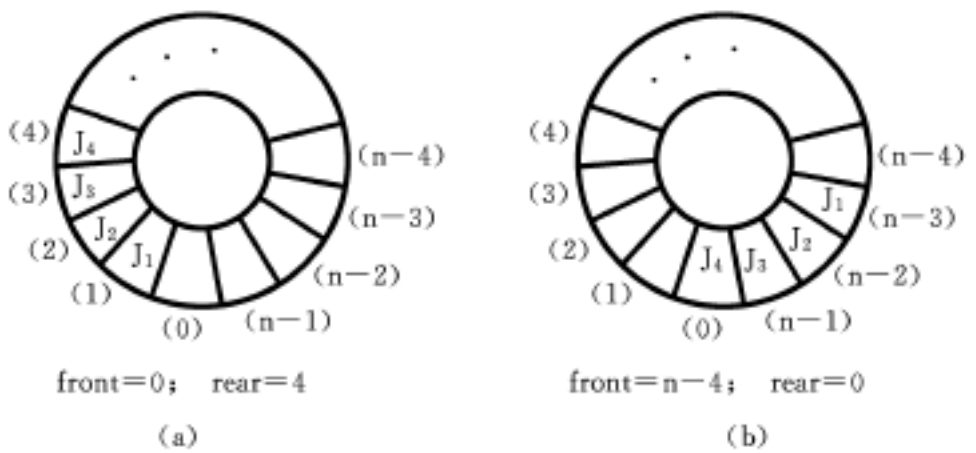


图 2 .10　含有元素 J_1 ,J_2 ,J_3 ,J_4 且容量为 n 的环形队列

对于用数组结构表示的队列,元素的插入和删除可用 SPARKS 描述成如下过程。

算法 2 .3　队列运算

```
procedure ADDQ(item, Q, n ,front ,rear)
    // 将 item 插入到存放在 Q(0：n - 1)的环形队列中 //
    // rear 指向队列的最后一项,front 指向队列中 //
    // 第一项按逆时针方向转到前一个位置 //
    rear←(rear + 1) mod n    // 模运算求余数,使 rear 顺时针前进一个位置 //
    if front = rear then call QUEUEFULL endif
    Q(rear)←item    // 将新项插入队列 //
end ADDQ
```

(a) 插入一个元素

```
procedure DELETEQ(item, Q,n ,front , rear)
    // 将队 Q(0：n)的第一个元素移出并将此元素存入 item //
    if front = rear then call QUEUEEMPTY endif
    front←(front + 1) mod n    // 顺时针向前移动 //
    item←Q(front)    // 将队中前部元素置入 item //
end DELETEQ
```

(b) 删去一个元素

容易看出,在把数组看成环形的情况下,这两个算法的计算时间都是 O(1)。特别要指出的是,这两个算法在队列满和队列空的测试条件上表面看来似乎是相同的,都是判断是否 front = rear,但由于这两种情况下队列所呈现的状态不同,因此在 front = rear 时队列是满还是空完全可以区别。在 ADDQ 情况下,即在执行了 rear←(rear + 1) mod n 语句之后,打算

插入一个元素时,front = rear 才调用 QUEUEFULL。此时,由于队列中的第一个元素不在 Q(front),而在由这点顺时针前进的第一个位置上,因此,实际上还有一个空位置 Q(rear),即 Q(front)。于是,ADDQ 在还剩一个空位置时通过调用 QUEUEFULL 发出队列满的信号,这与 DELETE 情况下,队列空的状态完全不同。这还表明,使用这种处理方法后,队列在任何时刻至多只能有 n - 1 个元素而不是 n 个元素。虽然也有将 n 个位置都用上的算法,譬如设置一个作为识别标志的变量 tag,在队列空时置 tag = 0,但这样做会使算法时间增长,而在任何包含队列的算法中 ADDQ 和 DELETEQ 都要多次使用,因此往往宁愿少用一个队列位置来节省计算时间。QUEUEFULL 和 QUEUEEMPTY 也是两个依赖于具体应用而没详加说明的过程。

队列的链接表示与栈的相应表示类似,每个结点也是由 DATA 和 LINK 两个信息段组成,前者用来保存数据对象,后者保存其链接信息。只是它使用了两个指针变量 front 和 rear 来指示前部和尾部的位置。元素在尾部插入而在前部删除。一旦 front = 0 就发出队列空的信息。图 2 .11 示出了具有 4 个元素 A、B、C、D 且按此顺序进入的链式队列。在队列的链表结构上作插入和删除运算获取可用结点和释放无用结点的工作也与 GETNODE 和 RETNODE 过程类似,至于链接队列的插入和删除过程的描述则留作一道习题。

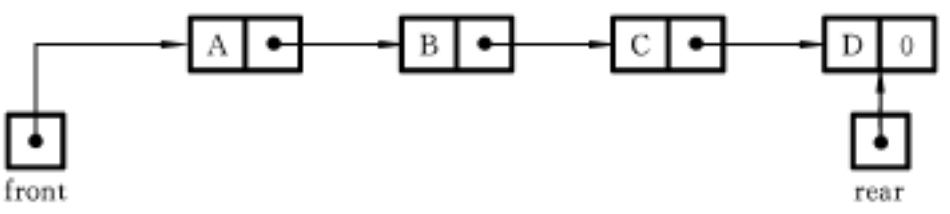


图 2 .11　具有 4 个元素的链接队列

2 .4 .2　树

定义 2 .4　树(tree)是一个或多个结点的有限集合,它使得:① 有一个特别指定的称作根(root)的结点;② 剩下的结点被分成 m≥0 个不相交的集合 $T_1, \cdots, T_m$,这些集合的每一个都是一棵树,并称 $T_1, \cdots, T_m$ 为这个根的子树(subtree)。

为讨论方便起见,引进了许多经常使用的术语。现以图 2 .12 所示的树为例来说明。这棵树有 13 个结点,每个结点的数据项是一个字母。这棵树的根为 A(有时也说成结点 A),它采用了一般将根画在顶部的图形表示方法。一个结点的子树数目称为该结点的度(degree)。A 的度是 3,C 的度是 1,F 的度是 0。度为 0 的结点称作叶子(leaf)或终端结点(terminal node),度不为 0 的结点则叫做非终端结点。图 2 .12 中 K, L, F, G, M, I, J 都是叶子,其余的结点则是非终端结点。结点 X 的子树的根是 X 的儿子(children),X 则是它儿子们的父亲(parent)。因此,D 的儿子是 H, I, J;D 的父亲是 A。同一父亲的儿子之间称为兄弟(sibling)。例如, H, I, J 是兄弟。根据需要,还可随时增加一些类似的词汇,例如,可以说 D 是 M 的祖父等。一棵树的度是这棵树中结点度的最大值。图 2 .12 中的树的度为 3。

结点的级(level)(又叫层)是这样确定的,设根是 1 级,若某结点在 p 级,则它的儿子在 p + 1级。一棵树中结点的最大级数定义为该树的高度(height)或深度(depth)。

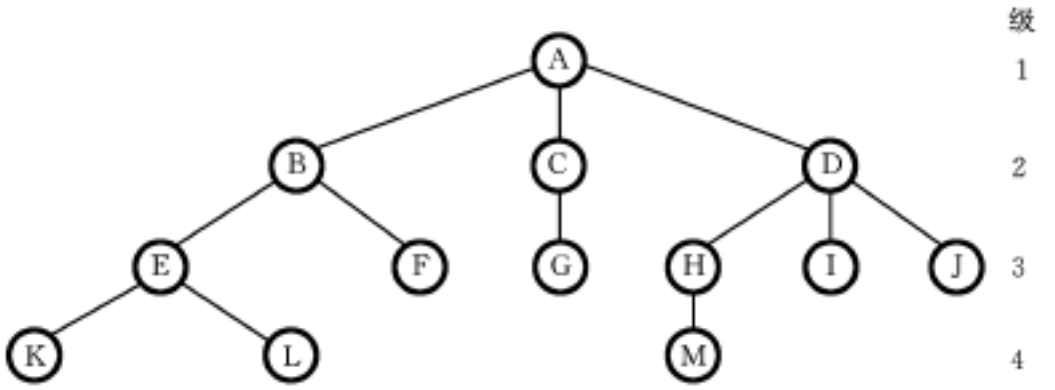


图 2 .12　一棵树

森林(forest)是 m≥0 个不相交树的集合。森林与树的概念非常接近,如果去掉树的根,就得到一个森林。例如,如果去掉图 2 .12 中的树的根 A,就得到有三棵树的森林。

怎样在计算机的存储器中表示一棵树呢?由树的构造方式很自然地会想到采用链接表存储结构,使链接表中的结点与树中的结点相对应。但是,由于树中各个结点的度可能很不一致,这就导致链接表各个结点所具有的信息段数目不相同。可以想象,使用这样的数据表示写算法是比较繁杂的。为了克服这一弱点,应使链接表结点信息段数归一。图 2 .13 提供了一种使链接表结点的信息段数固定的结构表示法,它给出了图 2 .12 中那棵树的表结构。图 2 .13 中,每个结点有 3 个信息段:TAG,DATA 和 LINK。信息段 TAG 的作用是设置标志。当 TAG = 0 时,DATA 和 LINK 的使用与以前所述相同;当 TAG = 1 时,DATA 信息段存放的不再是数据对象而是一个链接信息,即在此情况下 DATA 和 LINK 都存放表指针。这样,树就可以如下表示:表中的第一个结点中存放根并链接一些结点,这些结点指示着一些子表并且含有这根的所有子树。

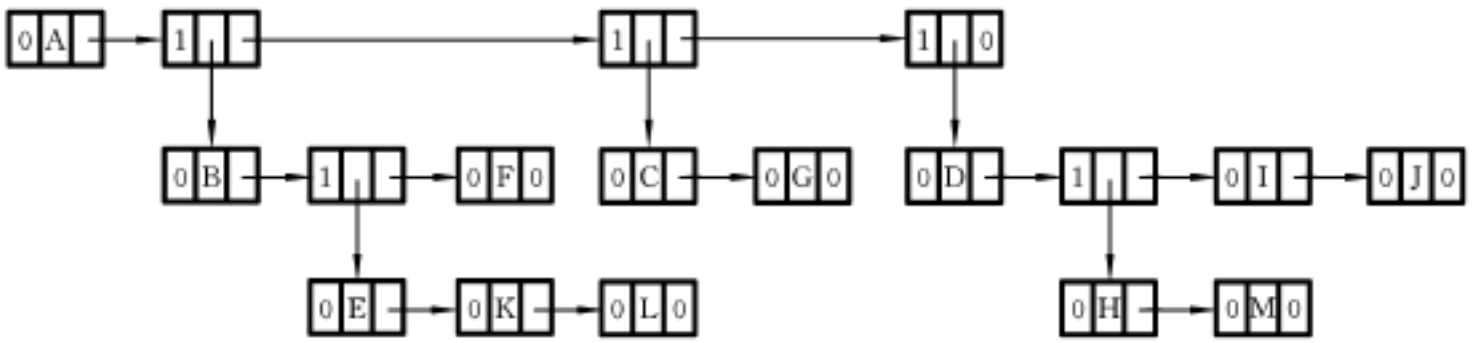


图 2 .13　图 2 .12 中树的表结构

1 . 二元树

二元树是使用得非常频繁的很重要的树结构。它具有以下特性:任何一个结点至多只能有两个儿子,即不存在度大于 2 的结点;另外,二元树的子树是有次序之分的,即分为左子树和右子树,而树的子树实际上无次序之分;再者,二元树允许有 0 个结点,而一棵树至少要有一个结点。将以上特性归纳起来,可对二元树定义如下。

定义 2 .5　二元树(binary tree)是结点的有限集合,它或者为空,或者由一个根和两棵树(称之为左子树、右子树)的不相交的二元树所组成。

作为例子,图 2 .14 显示了两棵特殊的二元树。图 2 .14(a)所示的是一棵斜树(skewed tree),它向左斜,还有一棵与之对应的右斜树。图 2 .14(b)所示的树称为完全二元树(complete binary tree)。这类树稍后定义。但要指出的是,由这棵树可以看出,它所有的叶结点

都在两个相邻的级上。另外要说明的是,树中引进的所有术语对二元树都适用。

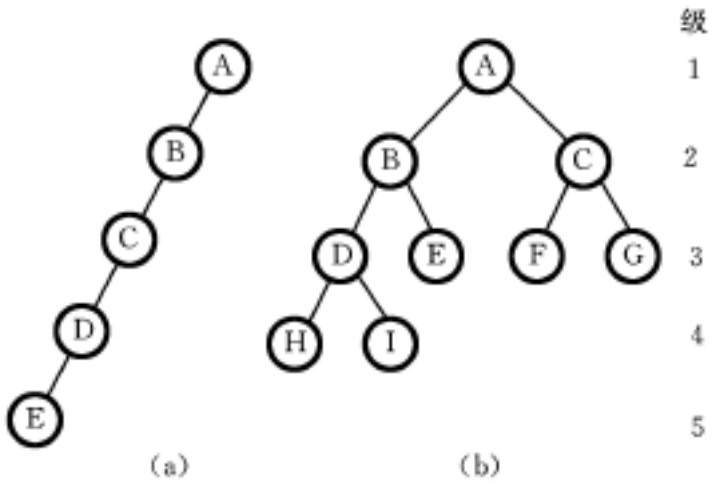


图 2 .14　两棵特殊的二元树

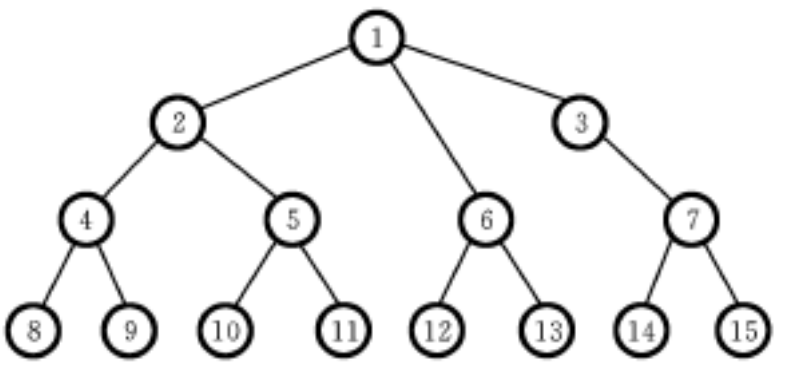


图 2 .15　深度为 4 的满二元树

引理 2 .1　一棵二元树第 i 级的最大结点数是 2^{i-1}。深度为 k 的二元树的最大结点数为 $2k - 1, k > 0$。

证明留给读者。

深度为 k 且有 $2^k - 1$ 个结点的二元树叫做深度为 k 的满(full)二元树。图2 .15给出了一棵深度为 4 的满二元树。对于满二元树有一种非常自然且有用的顺序表示,即给满二元树的结点从根开始从左到右逐级顺序编上 1,2,3,4,…的号码(见图 2 .16)。一棵有 n 个结点深度为 k 的二元树,当它的结点相当于深度为 k 的满二元树中编号为 1 到 n 的结点时,则称此二元树是完全的。由此定义可直接推出:完全二元树的叶子至多出现在相邻的两级上。利用以上的顺序编号法可以把完全树的结点紧凑地存放在一个一维数组 TREE 中,即把编号为 i 的结点存放在 TREE(i)中而无需任何链接信息。下面的引理采用了这种编号方式,因此能容易地确定任一结点 i 的父亲、左儿子和右儿子的位置。

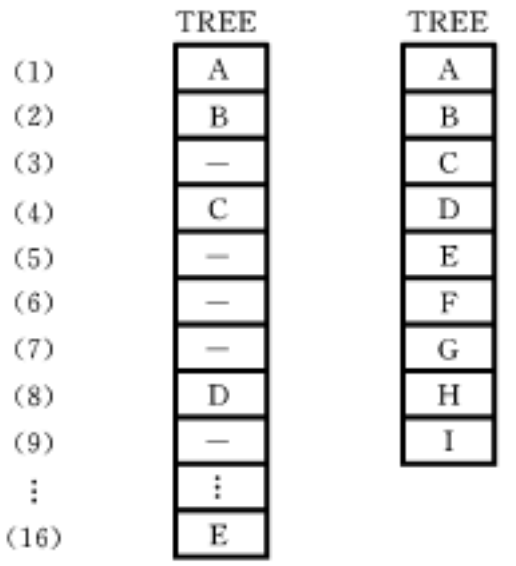


图 2 .16　图 2 .14 中两棵二元树的顺序表示

引理 2 .2　一棵有 n 个结点的完全二元树,如果它的结点按上述方法顺序编号,则对于编号为 i($1 \le i \le n$)的结点,有

(1) 若 $i \ne 1$,则 PARENT(i)在 $\lfloor i/2 \rfloor$*;若 i = 1,则 i 就是根并且 i 没有父亲。

(2) 若 $2i \le n$,则 LCHILD(i)在 2i;若 2i > n,则 i 没有左儿子。

(3) 若 $2i + 1 \le n$,则 RCHILD(i)在 2i + 1;若 2i + 1 > n,则 i 没有右儿子。

证明略。

对于完全二元树来说,这种顺序表示法从空间使用效率上看是很理想的,因为它没有浪费一点空间。但如果将这种表示法延伸到表示任何二元树,则在多数情况下要浪费不少空

---

* 如果 x 是任意实数,则定义 $\lfloor x \rfloor$ = 小于或等于 x 的最大整数,称 $\lfloor x \rfloor$ 为 x 的下限。同样,定义 $\lceil x \rceil$ = 大于或等于 x 的最小整数,称 $\lceil x \rceil$ 为 x 的上限。

间。例如,对图 2 .14(a)所示的左斜树使用顺序表示,只用了这数组空间的 1/ 3 还不到。在最坏情况下,对于深度为 k 的右斜树,它虽然需要 $2^k - 1$ 个位置的数组,但只有 k 个位置被用到。

即使对于完全二元树来说,这种顺序表示也有不足之处。例如,若要作插入或删除结点运算,则需要移动许多潜在的结点,并因此还引起剩余结点级数的改变。使用链接表示可以轻易地克服顺序表示的不足。在一般情况下链表的每个结点有 3 个信息段,它们是 LCHILD,DATA 和 RCHILD。如果需要确定结点的父亲,则可以增加一个 PARENT 信息段。图 2 .17 给出了图 2 .14 中二元树的链接表示。

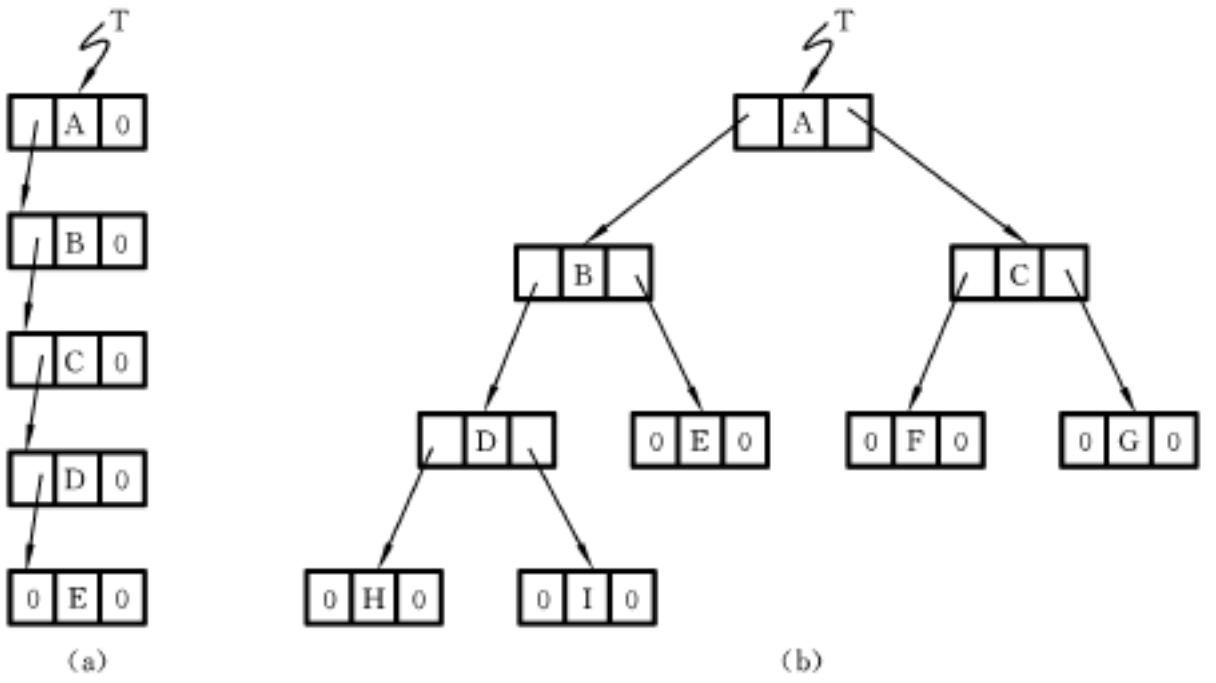


图 2 .17　图 2 .14 中二元树的链接表示

在实际生活中,有很多问题若用二元树构造其数据,则往往使所设计的算法较之使用别的数据结构简单、有效。所使用的数据放在结点的 DATA 信息段,这些数据通常是一些数或是可以比较其大小的数据对象(例如,字符串就可按字母顺序比较大小)。下面就在数据可比较大小的假设下介绍两种特殊的二元树,一种是堆的二元树,另一种是二分检索树。这是两种很有用的且在本书中要用到的结构。

定义 2 .6　堆(heap)是一棵完全二元树,它的每个结点的值至少和该结点的儿子们(如果存在的话)的值一样大。

堆是一种数据结构,它通常用来解决既要把元素插入集合,又要从集合中快速找出最大(或最小)元素的这样一类问题。如果所有元素各不相同,则由堆的定义立即可知最大元素在堆的根上。也可以按每个结点的值小于或等于它儿子的值来定义堆,在这种定义下根就含有最小元素。这样定义的堆称为 min-堆,而由定义 2 .6 所定义的堆叫做 max-堆。图 2 .18 给出了一个由数据(35, 40,45,50,80,90)所组成的堆。

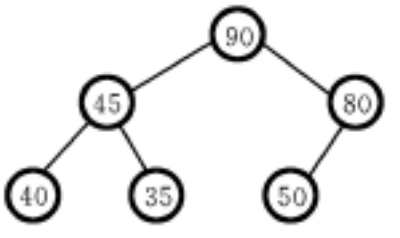


图 2 .18　由集合(35,40,45,50, 80,90)组成的堆

定义 2 .7　二分检索树(binary search tree)T 是一棵二元树,它或者为空,或者其每个结点含有一个可比较大小的数据元素,并且:

(1) T 的左子树的所有元素比根结点 T 中的元素小;

(2) 右子树的所有元素比根结点 T 中的元素大;

(3) T 的左子树和右子树也是二分检索树。

注意:二分检索树要求树中所有结点中的元素是互异的。

在编译程序中,把名字表构造成二分检索树,要确定某标识符是否在名字表中出现时,这种结构的检索算法就比很多别的算法快,为简单起见,可以造一张只含有 13 个 SPARKS 保留字的表,并把它们存入字符数组 NAME(1：13)中。

NAME:	(1)	(2)	(3)	(4)	(5)	(6)	(7)
	case	do	else	end	endcase	endif	if
NAME:	(8)	(9)	(10)	(11)	(12)	(13)	
	loop	procedure	repeat	return	then	while	

图 2 .19 所示的是一棵含有 NAME 中数据元素的二分检索树。

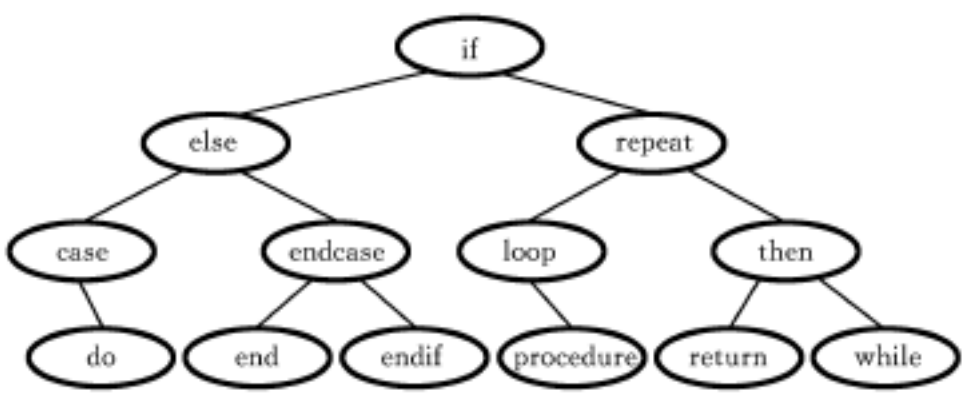


图 2 .19　一棵二分检索树

当用链接表来表示二分检索树时,当然希望结点的大小固定,但允许保留字的长短不一。为了解决这一矛盾,在实际处理时保留字并不进入结点的 DATA 信息段,而是存入该保留字所在数组元素的下标。LCHILD 和 RCHILD 信息段中则保存该保留字的左、右儿子(它们也是保留字)在树中的位置。表 2 2 给出了在计算机中用数组 D(1：13)存放这棵二分检索树的实际表示,其中每个数组元素 D(i)含有 DATA、LCHILD 和 RCHILD 三个信息段。例如,在 D(3)的 DATA 段中 10 是一个下标值,它指示 repeat 存放在数组 NAME 中的位置,而 LCHILD 和 RCHILD 中的 6 和 7 也是下标值,但它们指示 repeat 的左儿子 loop 和右儿子 then 在树中(即数组 D 中)的位置。

表 2 2　图 2 .19 的数组表示

	LCHILD	DATA	RCHILD
D(1)	2	7	3
D(2)	4	3	5
D(3)	6	10	7
D(4)	0	1	8
D(5)	9	5	10
D(6)	0	8	11
D(7)	12	12	13
D(8)	0	2	0
D(9)	0	4	0
D(10)	0	6	0
D(11)	0	9	0
D(12)	0	11	0
D(13)	0	13	0

由二元树可很自然地推广到 k(k≥2)元树。在 k 元树中一个结点至多有 k 个儿子,并且这些儿子是有序的。二元树的顺序表示和结点大小固定的链接表示都可推广到多元树。

2 .树转换成二元树

由于二元树结构的处理比较简单,因此,常常把其它树转换成二元树来处理,这样,可使需要的空间减小、算法简化。其转换方法如下:设有一棵树 T(它的根为 T_1),人为安排它的子树有序且设为 T_{11}, T_{12}, …, T_{1k}。用 T_1 做二元树的根,T_{11} 做 T_1 的左子树,然后 T_{1i} 做 $T_{1,i-1}$ 的右子树,$2 \leq i \leq k$。看起来就像图 2 .20 所示的那样。

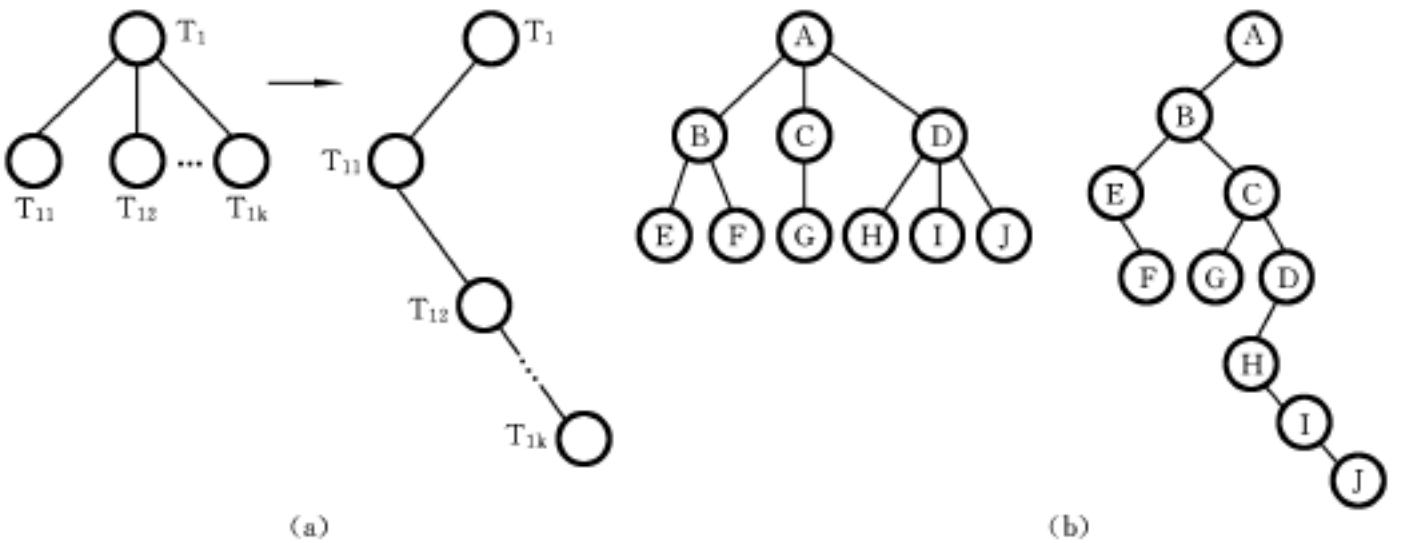


图 2 .20 将一棵树转换成二元树

(a) 一般情况;(b) 一个例子

2 .4 .3 集合的树表示和不相交集合的合并——树结构应用实例

假定有一个全集合 U,它由 n 个元素所组成,那么从 U 中可以构造出很多个集合。对于这些集合经常进行的是求取它们的交集和并集的运算。而如果这些集合是一些不相交的集合,即这些集合中的任意两个集合都没有公共元素,那么对它们大量施行的两种基本运算是合并某些集合和查找某个元素在哪一个集合之中。例如,有一个 n = 10 的全集合 U = {1, 2,3,4,5,6,7,8,9,10},由它可构造出集合 S_1 = {1,7,8,9}, S_2 = {2,5,10}和 S_3 = {3,4,6}。它们是 3 个不相交集合。如果将 S_1 和 S_2 合并成一个新的集合,就执行合并运算,$S_1 \cup S_2$ = {1,7,8,9,2,5,10}。如果要查找元素 4 包含在哪个集合之中,经过一番查找运算就可知道 4 在集合 S_3 中。

为了使上述运算有效地执行,就需要将集合中的元素构造成一定的结构形式,设计出在这些结构下执行上述运算的算法,分析这些算法的计算复杂度,并从中选出计算时间短、算法简便的数据结构。

通常,用一个位向量 SET(1 : n)来表示集合,且若 U 的第 i 个元素在这集合中,则置 SET(i) = 1,否则置零。这种表示的优点在于可以迅速地确定某个元素是否属于这个集合,而且能够很方便地利用计算机上的“逻辑加”和“逻辑乘”指令来实现两个集合的并和交之类的运算。但这种表示有很大的局限性,它只有在 n 很小(例如,n 小于等于一个机器字长)时,运算才特别有效。当 n 很大(例如,大于一个机器字长),而每个集合的大小相对于 n 来说又很小时,这种表示将使得并或交运算的执行时间不是与两个集合中的元素数目成正比,而是与 n 成正比,因此,这种表示是无效的。

集合的另一种表示方法是用集合的元素表来表示每一个集合,由于元素表中的元素是

无序的,因此要执行并或交运算,花费的时间与参加运算的两个集合长度的乘积成正比,而确定某元素属于哪一个集合的运算时间则与这些集合长度的和成正比。即使事先将参加运算的集合中的元素作某种排序,执行并或交运算的时间还是与参加运算的集合长度之和成正比。集合还可以用树结构来表示。由下面的讨论可以看到,用树来表示集合,对于不相交集合,能方便地进行集合的并和确定某元素属于哪个集合这两种基本运算(使得为这两种运算所拟定的算法能有效地实现)。前面例子中的 3 个不相交集合 S_1,S_2,S_3 可以用图 2.21 所示的 3 棵树来表示。

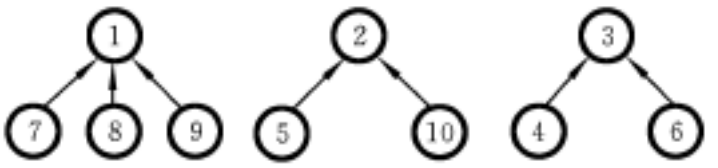


图 2.21　用树表示不相交集合

集合在用树表示的情况下,求两个不相交集合的并集,一种最直接的方法就是使一棵树变成另一棵树的子树。于是,$S_1 \cup S_2$ 就有图 2.22 所示的形式之一。

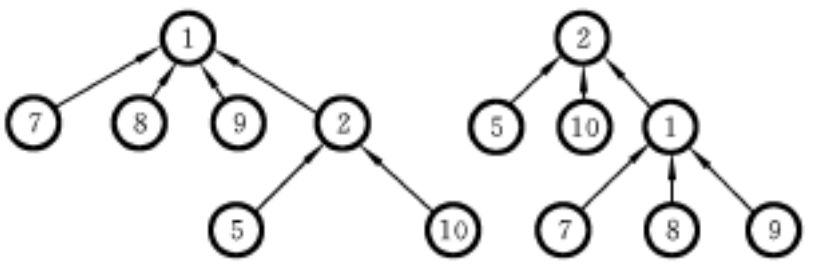


图 2.22　$S_1 \cup S_2$ 的两种树表示

在这种树表示中,值得特别指出的是结点按父亲关系相连接的情况,若树用链接表来表示,则链接表中的每个结点需要设置 PARENT 信息段,而无需设置 LCHILD 和 RCHILD 信息段。为讨论方便起见,参加运算集合中的元素都用存放这些元素数组的下标来代替,并使这些下标与表示链接表的数组的下标相对应,在这种对应关系下,链接表的数组元素的下标就代表集合中的相应元素,从而可以取消链接表中结点的 DATA 信息段。这样一来,链接表的结点只含有一个信息段,即 PARENT 信息段。因此,不妨把表示链接表的数组就记为 PARENT(1∶n)。PARENT(i)中存放着元素 i 在树中的父结点的指针。这样,根结点的 PARENT 信息段的内容就为零,而根结点就是 PARENT 数组中存放这个零值的数组元素的下标。用这个下标作为集合的名字,可以使讨论进一步简化。这样一来,两个不相交集合的合并,实质上就是把一棵树中根的 PARENT 信息段置成另一棵树的根;而生成的新树的根就代表这两个集合合并之后的并集。至于找元素 i 所属集合的运算则成了确定包含元素 i 的树的根。由以上讨论可知,为这两种运算初步设计的算法就是过程 U 和 F。

算法 2.4　简单的合并与查找运算

```
procedure U(i,j)
    // 根为 i 和 j 的两个不相交集合用它们的并来取代 //
    integer i,j
    PARENT(i)←j
end U
```

```
procedure F(i)
    //找包含元素 i 的树的根//
    integer i,j
    j←i
    while PARENT(j) > 0 do    //若此结点是根,则 PARENT(j) = 0//
    j←PARENT(j)
    repeat
    return (j)
end F
```

这两个算法虽然相当简单,但性能却不理想。例如,假设有 n 个元素 1,2,…,n,开始它们的每一个分别在只有它自己的一个集合中,即 $S_i = \{i\}, 1 \leq i \leq n$,那么最初的结构是由这 n 个结点的森林和 PARENT(i) = 0,1≤i≤n 所组成,如果要依次作下面一系列的合并和查找运算:

U(1,2),F(1),U(2,3),F(1),U(3,4)
F(1),U(4,5),…,F(1),U(n - 1,n)

则产生图 2.23 所示的退化树。

图 2.23 最坏情况树

由于每执行一次 U 算法所花的时间为常数,因此进行这 n - 1 次合并的计算时间为 O(n)。而算法 F 的执行时间与此元素在树中所处的级数成正比,即,如果元素在树中的第 i 级,则 F 的计算时间就为 O(i),因此,例中的 n - 2 次查找所用的时间为 $O(n^2)$。易于看出,这个例子给出了一系列合并、查找情况下算法 U 和 F 的最坏情况。产生这种最坏情况的症结何在呢?问题显然不在 F 而在 U,即多次的合并运算构造出一棵串行的退化树。为避免构造出退化树,在对树 i 和树 j 作合并运算时使用一条加权规则,即,如果树 i 的结点数少于树 j 的结点数,则使 j 成为 i 的父亲,反之,则使 i 成为 j 的父亲。要使用这一规则就需要知道每一棵树有多少结点,实现这一点的一种简单方法是,在每棵树的根结点增设一个 COUNT 计数信息段。如果 i 是根结点,则 COUNT(i) = 树 i 的结点数。为了保证结点大小固定,实际上不用在根结点增设 COUNT 信息段,而是将结点计数以负数形式保存在根结点的 PARENT 信息段中。由于除根结点外的所有其它结点的 PARENT 是正数,所以,这样处理等价于用一个符号位把计数器与链接指针区别开来,而不会导致混乱。

算法 2.5 使用加权规则的合并算法

```
procedure UNION(i,j)
    //使用加权规则合并根为 i 和 j 的两个集合,i≠j//
    //PARENT(i) = - COUNT(i), PARENT(j) = - COUNT(j)//
    integer i,j,x
    x←PARENT(i) + PARENT(j)
    if PARENT(i) > PARENT(j)
    then PARENT(i)←j    //i 的结点少//
      PARENT(j)←x
    else PARENT(j)←i    //j 的结点少//
      PARENT(i)←x
```

```
        endif
    end UNION
```

UNION 这个算法的计算时间虽比 U 算法的时间增加了一些,但还是以常数限界的。如果将算法 2 .5 中的一系列合并运算改为使用 UNION,就得到图 2 .24 中的树。

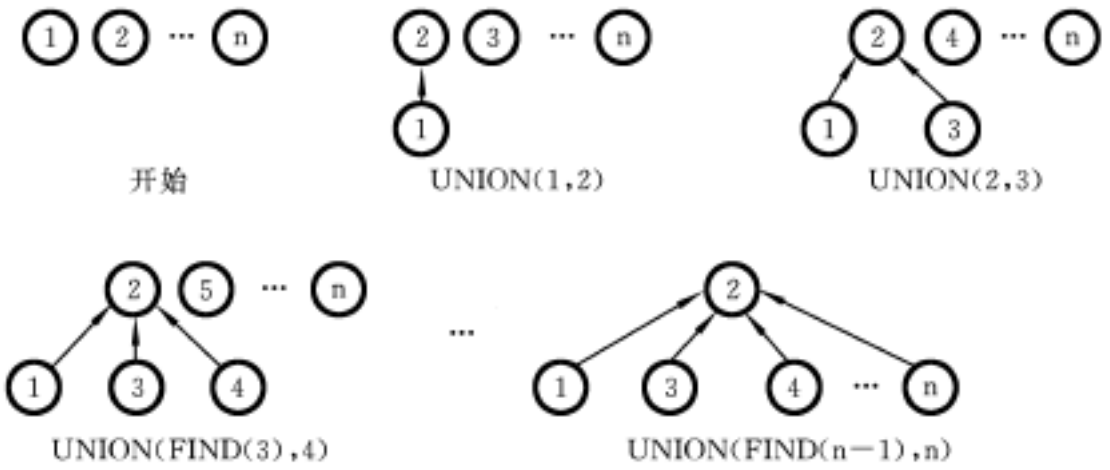


图 2 .24　利用加权规则得到的树

现在,对于结点 1 作上述 n - 2 次查找运算,由于任何结点的最大级数都不超过 2,所以 n - 2 次查找的时间是 O(n)。但是,这并不是使用加权规则作任意 n 次合并和查找的最坏情况。下面的引理 2 .3 给出了执行一次查找的最大时间。

引理 2 .3　设 T 是一棵由算法 UNION 所产生的有 n 个结点的树。在 T 中没有结点的级数会大于$\lfloor \log n \rfloor + 1$。

证明　对于 n = 1,引理显然为真。假设对于所有结点数为 i 的树,$i \leq n - 1$,引理为真。现证明对于 i = n 引理亦真。设 T 是由算法 UNION 所产生的一棵 n 个结点的树,现考虑最后一次执行的合并运算 UNION(k,j)。设 m 是树 j 的结点数,n - m 是树 k 的结点数。不失一般性,可以假设 $1 \leq m \leq n/2$,于是,T 中任一结点的最大级数或者与 k 中的最大级数相同,或者比 j 的最大级数大 1。若为前者,则 T 的最大级数$\leq \lfloor \log(n - m) \rfloor + 1 \leq \lfloor \log n \rfloor + 1$。若为后者,则 T 的最大级数$\leq \lfloor \log m \rfloor + 2 \leq \lfloor \log(n/2) \rfloor + 2 \leq \lfloor \log n \rfloor + 1$。证毕。

下例说明引理 2 .3 的界限不仅是一个上界,而且是最小上界,它对于某个合并序列是可以达到的。

例 2 .1　在由初始结构 PARENT(i) = - COUNT(i) = - 1, $1 \leq i \leq 2^3$ 所开始的下述合并序列上,考虑算法的工作情况:

UNION(1,2), UNION(3,4), UNION(5,6), UNION(7,8)
UNION(1,3), UNION(5,7), UNION(1,5)

所得到的树由图 2 .25 示出。由这个例子可以很容易推广到一般情况而得到具有$\lfloor \log m \rfloor + 1$级的 m 个结点的树。

作为引理 2 .3 的一个结果,对于算法 UNION 所产生的 n 个结点的树,执行一次查找的最长时间至多是 O(logn)。如果要处理的合并和查找序列含有 n 次合并和 m 次查找,则最坏情况时间就变成了 O(mlogn)。令人振奋的是还可作进一步的改进。这种改进是在查找运算中使用一条称之为压缩的规则。压缩规则:如果 j 是由 i 到它的根的路径上的一个结点,则置 PARENT(j)←root(i)。使用压缩规则的查找算法描述如下。

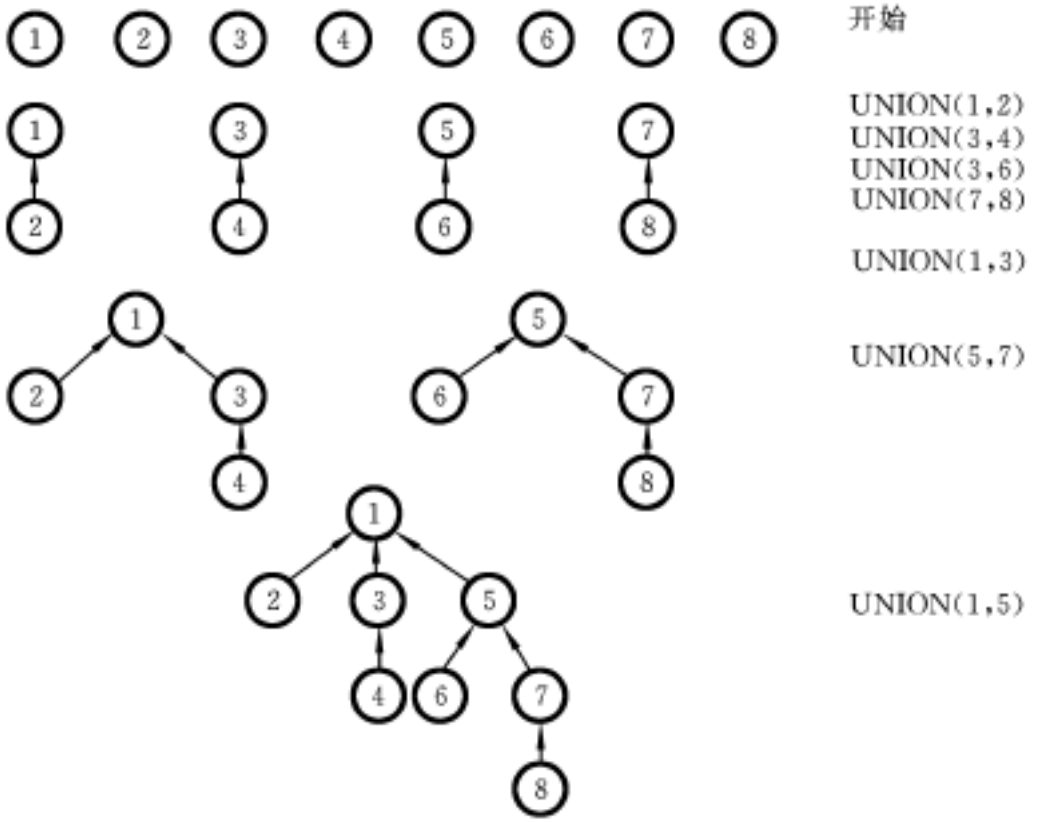


图 2 .25　使用加权规则的一棵最坏情况树

算法 2 .6　使用压缩规则的查找算法

```
procedure FIND(i)
    // 查找含有元素 i 的树根,使用压缩规则去压缩由 i 到根 j 的所有结点 //
    j←i
    while PARENT(j) > 0 do    // 找根 //
    j←PARENT(j)
    repeat
    k←i
    while k≠j do    // 压缩由 i 到根 j 的结点 //
    t←PARENT(k)
    PARENT(k)←j
    k←t
    repeat
    return(j)
end FIND
```

仔细考察此算法可以看出,对于单一的查找运算,FIND 的计算时间比 F 增加了一倍。因此,在某些情况下(例如,在大量查找和少量合并运算情况下)使用 FIND 可使整个处理时间慢下来。但在最坏情况下,相对于只是使用加权规则的处理确实是一个很大的改进。

例 2 .2　考虑例 2 .1 中的合并序列用 UNION 算法所生成的树。现要求作下列 8 次查找:

FIND(8), FIND(8), FIND(8), FIND(8), FIND(8), FIND(8), FIND(8), FIND(8)

如果使用过程 FEND(8),就要求沿 3 个 PARENT 信息段而上;处理这 8 次查找共需要移动 24 次。而在使用 FIND 算法的情况下,第一个 FIND(8)要沿 3 个 PARENT 信息段而上,然后重置这 3 个链接信息段;在剩下的 7 次查找中,每一次沿一个信息段而上且重置一次末级结点 8 的链接信息段,总共要移动 20 次。

在处理合并和查找序列时,UNION-FIND 算法的最坏情况时间如何限界呢？这由引理 2.4 给出。在叙述这个引理之前,先引进一个增长非常慢的函数 $\alpha(m,n)$,这个函数与阿克曼(Ackermann)函数 $A(p,q)$的逆函数有关,函数 $\alpha(m,n)$的定义如下:

$$\alpha(m,n)=\min\{z\geq 1 \mid A(z,4\lceil m/n\rceil)>\log n\}$$

此处所使用的 Ackermann 函数是 Ackermann 函数的变型,其定义如下:

$$A(p,q)=\begin{cases}2q & p=0\\ 0 & q=0 \text{ 且 } p\geq 1\\ 2 & p\geq 1 \text{ 且 } q=1\\ A(p-1,A(p,q-1)) & p\geq 1 \text{ 且 } q\geq 2\end{cases}$$

函数 $A(p,q)$是增长速度很高的一个函数,表 2.3 给出了该函数的部分值。

表 2.3　$A(p,q)$的部分函数值

p \ q	0	1	2	3	4	5	6	7	8	9	10	11	12	…
0	0	2	4	6	8	10	12	14	16	18	20	22	24	…
1	0	2	4	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	…
2	0	2	4	2^{2^2}	$2^{2^{2^2}}$	$2^{2^{2^{2^2}}}$	…	…	…	…	…	…	…	…
3	0	2	4	$2^{2^{2^2}}$	$2^{2^{\cdot^{\cdot^{\cdot^2}}}}$	}65536 个 2	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…

事实上,可以证明如下结论成立:

(1) $A(p,q+1)>A(p,q)$

(2) $A(p+1,q)>A(p,q)$

(3) $A(3,4)=2^{2^{\cdot^{\cdot^{\cdot^2}}}}$ }65536 个 2

对 $\alpha(m,n)$,假设 $m\neq 0$,在 $\log n<A(3,4)$的情况下,则由(a)和(b)及 $\alpha(m,n)$的定义可知 $\alpha(m,n)\leq 3$;而由(c)可看出 $A(3,4)$是一个非常大的数,所以在计算机求解的实际问题中,总可假定 $\log n<A(3,4)$。因此,在实际场合中无论 m、n 增长多快,总有 $\alpha(m,n)\leq 3$。

引理 2.4　设 $T(m,n)$是处理 $m\geq n$ 次 FIND 和 $n-1$ 次 UNION 的混合序列所要求的最坏情况时间,则对于某两个正常数 k_1 和 k_2 有

$$k_1 m\alpha(m,n)\leq T(m,n)\leq k_2 m\alpha(m,n)$$

引理 2.4 表明 UNION-FIND 序列的时间复杂度几乎与 FIND 的次数 m 成线性关系,即只比线性关系稍差。所需要的空间是每个元素一个结点。

下面来看一个 UNION 和 FIND 算法的简单应用。它是处理 FORTRAN 语言中的 EQUIVALENCE语句(等价语句)时所要做的事情的抽象化。假定输入形式为 i≡j(i 与 j 等价)的对偶集合。要求对新输入的对偶能迅速处理并对任何时候询问一个元素当前在哪一个等价类作出快速响应。为解决以上问题,可以把要生成的那些等价类看成是一些集合。因为没有变量能在一个以上的等价类之中,所以这些集合是互不相交的。假定初始状态为每个等价类仅含一个元素,即元素本身,则 PARENT(i) = -1,$1\leq i\leq n$。如果要处理的对偶是 i≡j,那就必须先找出 i 和 j 所在的集合。如果这两个集合不同,则将它们合并为一个集

合。如果这两个集合相同,则说明 i,j 已在同一等价类中。由于关系 i≡j 是多余的重复,因此无需再作处理。为了处理每一个等价对偶,至多需要作两次 FIND 和一次 UNION。如果假定有 n 个变量和 m≥n 个等价对偶,则总的处理时间至多是 O(mα(m,n))。这样处理就顺利解决了上面所提出的问题,而且在开始时并不要求引进所有的对偶,这就是所谓的按“联机”方式工作,在以后的章节中,将会看到这两个集合运算算法在其它方面的有效应用。

2.4.4 图

数据对象图是由欧拉(L.Euler)在 1736 年首先引进的一种重要数据结构,一个图(graph)G 由称之为结点(vertices)V 和边(edges)E 的两个集合所组成。V 是一个有限非空的结点集合,这些结点通常被记数为 1,2,…,n。E 则是结点对偶的集合,E 中的每一对偶就是 G 的一条边。

如果这些对偶是有序的(即对偶〈i,j〉与对偶〈j,i〉比起来是不同的),则称图是有向的(directed)。反之,则称它是无向的(undirected)。有向边用尖括号来表示,无向边用圆括号表示。于是〈i,j〉表示一条有向边,(i,j)表示一条无向边。注意,不允许形式为〈i,i〉或(i,i)的边。对于许多应用而言,往往还有一个称之为成本的正实数,它被附于每一条边上,这样的图称为网络(network)。

在一个无向图中,如果存在边(i,j),就说结点 i 与 j 相邻接。一个结点的度是它邻接点的数目。对于有向图,要将以下两点区别开:结点 i 的入度是用 i 作为边的第二个成分的边的数目,而出度则是用 i 作为边的第一个成分的边的数目。如果有向边表示成〈i,j〉,则 i 是邻接到 j 的,而 j 是由 i 所邻接的。

由结点 v_p 到 v_q 的一条路(path)是结点 v_p,v_{i1},v_{i2},…,v_{in},v_q 的一个序列,它使得(v_p,v_{i1}),(v_{i1},v_{i2}),…,(v_{in},v_q)是 E(G)中的边。一条路的长度(length)是组成这条路的边数。一条简单路(simple path)是除了第一和最后一个结点可以相同以外,所有结点都有不同的路。环(cycle)是第一个和最后一个结点相同的简单路。

图 2.26 所示的是一个有向图和一个无向图的例子,它们都有 5 个结点和 5 条边。在有向图中,结点 1 的入度为 0,出度为 3;在无向图中,结点 1 的度为 3。在无向图中,每一对结点之间都存在一条路;而在有向图中,就不存在从结点 3(或结点 5)通到任何其它一个结点的路。在无向图中,边(1,2)(2,3)构成一条简单路;而路(1,2)(2,3)(3,1)是一个环。

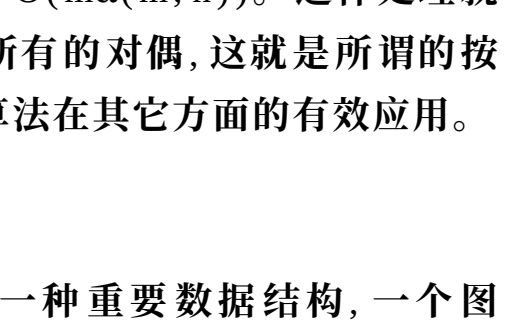


图 2.26 两个实例图

下面讨论图的连通性。在一个无向图中,如果每一对结点之间存在一条路,则称该图是连通的(connected),否则是不连通的。一个图的子图是 V 的结点子集(记为 VB)和 E 中连接 VB 中结点的边的子集。连通分图(又叫分图,连通支或支)是最大的连通子图,即,对于无向图 G=(V,E)中的连通子图 G′=(V′,E′),如果 G 中不存在另外的连通子图 G″=(V″,E″),使得 V′ V″或 E′ E″,则 G′是 G 的连通分图。在有向图中,如果对于每一对结点 i 和 j,既存在一条从 i 到 j 的路,又存在一条从 j 到 i 的路,则称该有向图是强连通的(strongly connected)。

图有两种常用的表示方法。这两种表示法可以分别看成是顺序表示法和链接表示法。

顺序表示使用 n 行和 n 列的方阵表,其中 n 是结点数,这个表称为邻接矩阵(adjacency matrix)。对于一个无向图,这个邻接矩阵 GRAPH(1：n,1：n)定义为,若边(i,j)出现,则 GRAPH(i,j) = 1,否则为 0。如果这图是一个网络,则 GRAPH(i,j) = 边(i,j)的成本。如果(i,j)不出现,则 GRAPH(i,j)的值是 + ∞。对于一个有向图,当且仅当〈i,j〉是一条边时,GRAPH(i,j) = 1。在有向网络中,GRAPH(i,j)被类似定义。

表 2.4 显示了图 2.26 中有向图和无向图的邻接矩阵。两个矩阵都是 5×5 并且矩阵元素只为 0 或 1。注意:在这两种情况下,对角线元素是 0,表示根本没有"自身的边"。第二个矩阵有一个所有无向图都具有的特殊结构,即 GRAPH(i,j) = GRAPH(j,i)。这说明矩阵是对称的。虽然邻接矩阵具有 n^2 个元素,但对于无向图而言,只保存上三角矩阵的 n(n - 1)/ 2 个元素就足够了。要指出的是,因为 GRAPH(i,i) = 0,所以主对角线不需要被存储。

表 2.4　图 2.27 的邻接矩阵

	1	2	3	4	5
(1)	0	1	1	1	0
(2)	0	0	1	0	0
(3)	0	0	0	0	0
(4)	0	0	0	0	1
(5)	0	0	0	0	0

	1	2	3	4	5
	0	1	1	1	0
	1	0	1	0	0
	1	1	0	0	0
	1	0	0	0	1
	0	0	0	1	0

在着手图上的任何计算以前,通常需要将邻接矩阵初始化,使它包含将要在其上计算的图。这一步一般至少要求进行 $O(n^2)$ 次运算。因此,几乎任何一个使用这种表示的算法,它的计算时间至少是 $O(n^2)$。即使这图只有 O(n)条边也是如此。这一事实使我们考虑另一种表示——邻接表。

给定一个图,它的邻接表(adjacency list)是由 n 个表组成的,每个结点 i 有一个表,这个表由 i 所邻接的那些结点所组成。因为经常需要存取一个随机结点的那些邻接结点,所以要把这些表的表头按顺序存放。不过,一个结点的邻接表可以链接在一起。图 2.27 显示了图 2.26 所示的两个图的邻接表。

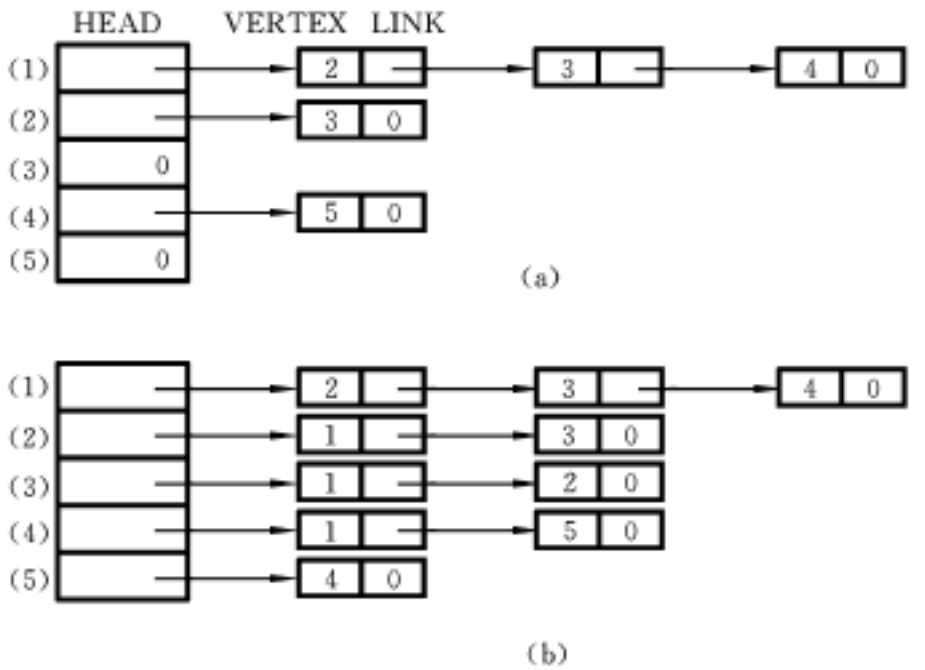


图 2.27　图 2.26 的邻接表

这两个图都有 5 个顺序位置(头结点),其中的值或者是零(如果不存在邻接点)或者是一个结点表的指针。表上每一个结点有两个信息段,一个是结点,一个是表中下一个元素的指针。这里的有向图有 5 个结点,而无向图则有 10 个结点。通常,一个具有 n 个结点和 e

条边的有向图要求 n 个位置加上 e 个结点,而一个无向图则要求 n 个位置加上 2e 个结点,这比邻接矩阵所要求的空间可能稍许小一点。

在图上既不插入又不删除边或结点时,这些邻接表可以按它们自身的形式顺序地用一个一维数组 VERTEX(1：p)来表示。其中,若这图是有向图,则 p = e;若这图是无向图,则 p = 2e。HEAD(i),1≤i≤n,给出结点 i 的邻接表的开始点。如果定义 HEAD(n + 1) = p + 1,则结点 i 的邻接表中的那些结点就存放在 VERTEX(j)中,其中 HEAD(i)≤j< HEAD(i + 1)。如果 i 的邻接表为空,则 HEAD(i) = HEAD(i + 1)。图 2.28 给出了与图 2.27 所示的链接表相对应的顺序邻接表。

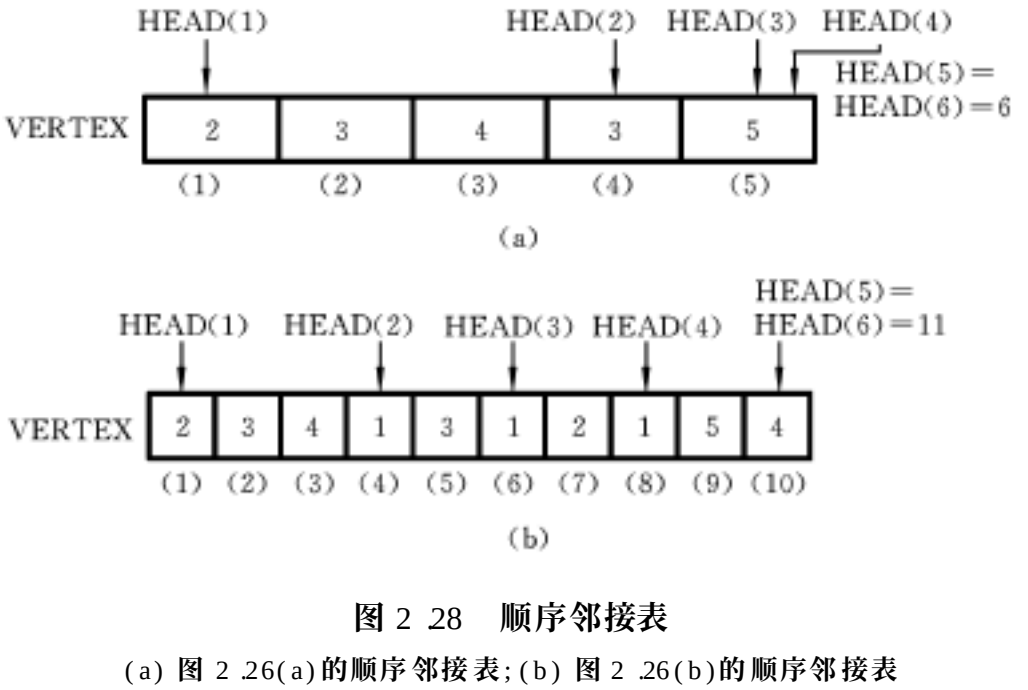


图 2.28 顺序邻接表

(a) 图 2.26(a)的顺序邻接表;(b) 图 2.26(b)的顺序邻接表

习 题 二

2.1 对于下列函数,求使得第二个函数比第一个函数小的 n 的最小值。

(1) n^2,10n;　　(2) 2^n,$2n^3$;

(3) $n^2/\log n$,$n(\log n)^2$;　　(4) $n^3/2$,$n^{2.81}$。

2.2 使用计算器(或手算)来扩展表 2.1。

(1) 增加下述的列的值:loglogn,$n^2\log n$,$n^3\log n$ 和 n^n。

(2) 增加 n 为下述值的行:64,128,256,512 和 1024(可使用近似值)。

2.3 用一种更清晰的方法重写下列程序段:

```
i←n
while i > 1 do
   y←F(x)
   i←i - 2
repeat
```

```
if a > b
then if c > d
        then if e > f then x←1
                      else x←2
             endif
     else x←3
     endif
else x←4
endif
```

2.4 写一个布尔函数,由该函数获取一个以 0 或 1 为元素的数组 A(1：n),n≥1,并要求确定每个连

续为 1 的序列的大小是否为偶数。分析所写的算法的计算时间。

2.5　如果习题 2.4 的函数不是以递归形式写出的话,就请再写出习题 2.4 的递归程序。

2.6　写出用链接表表示队列时插入和删去元素的算法 ADDQ 和 DELETEQ。

2.7　在数组 C(0：n - 1)中存放着一个以 F 和 R 作为前部和尾部的环形队列。

(1) 根据 F,R 和 n 列出求此环形队列中元素个数的关系式。

(2) 写出删去此队列中第 k 个元素的算法。

(3) 写出将元素 Y 插入紧接在 k 个元素之后的算法。

(4) 求所写的关于(2)、(3)算法的时间复杂度。

2.8　双端队列是一个可在其两端作插入和删去运算的线性表。如何在一个一维数组中表示这个队列?写出在该队列两端插入和删去元素的算法。

2.9　考虑一个假想的数据对象 X2。X2 是可在它的两端插入但只能在一端删去元素的线性表。对 X2 设计一种链接表表示,并写出插入和删去的算法。试说明所用的表示法的初始条件和边界条件。

2.10　证明引理 2.1。

2.11　写出在二分检索树 T 上检索标识符 X 的算法。假定 T 中的每一个结点是 3 个信息段: LCHILR,DATA 和 RCHILD。求所写算法的计算时间是多少?

2.12　设计一种能将图存放在穿孔卡片上的合适表示。写出输入这个图并产生它的邻接矩阵的算法。

2.13　利用习题 2.12 的外部表示,写一个输入这个图并生成它的邻接表的算法。

2.14　对于有 n 个结点和 e 条边的无向图 G,证明其所有结点度数之和为 2e。

2.15　(1) 设 G 是一个具有 n 个结点的无向连通图。证明 G 至少有 n - 1 条边,并证明凡具有 n - 1 条边的无向连通图是一棵树。

(2) 一个有 n 个结点的强连通图的最小边数是多少?它具有什么样的形状?证明你的答案。

2.16　对于一个有 n 个结点的无向图 G,证明下列命题等价:

(1) G 是一棵树;

(2) G 是连通的,但若去掉任何一条边,则所得到的图是不连通的;

(3) 对于属于 V(G)中的每对不同的结点 u 和 v,恰好存在一条由 u 到 v 的简单路;

(4) G 不包含环且有 n - 1 条边;

(5) G 是连通的且有 n - 1 条边。

第 3 章

递 归 算 法

递归是程序设计中的一个非常重要的课题。用递归技术设计的算法简单明了,常为人们所采用。递归算法的设计与分析是算法设计与分析的一大类。

3 .1　递归算法的实现机制

递归算法(包括直接递归和间接递归子程序)都是通过自己调用自己,将求解问题转化成性质相同的子问题,最终达到求解的目的的。递归算法充分地利用了计算机系统内部机能,自动实现调用过程中对相关且必要的信息的保存与恢复,从而省略了求解过程中许多细节的描述。

3 .1 .1　子程序的内部实现原理

要理解递归,首先应理解一般子程序的内部实现原理。

1 . 子程序调用的一般形式

子程序的调用一般有四种形式,分别如图 3 .1(a)～3 .1(d)所示。

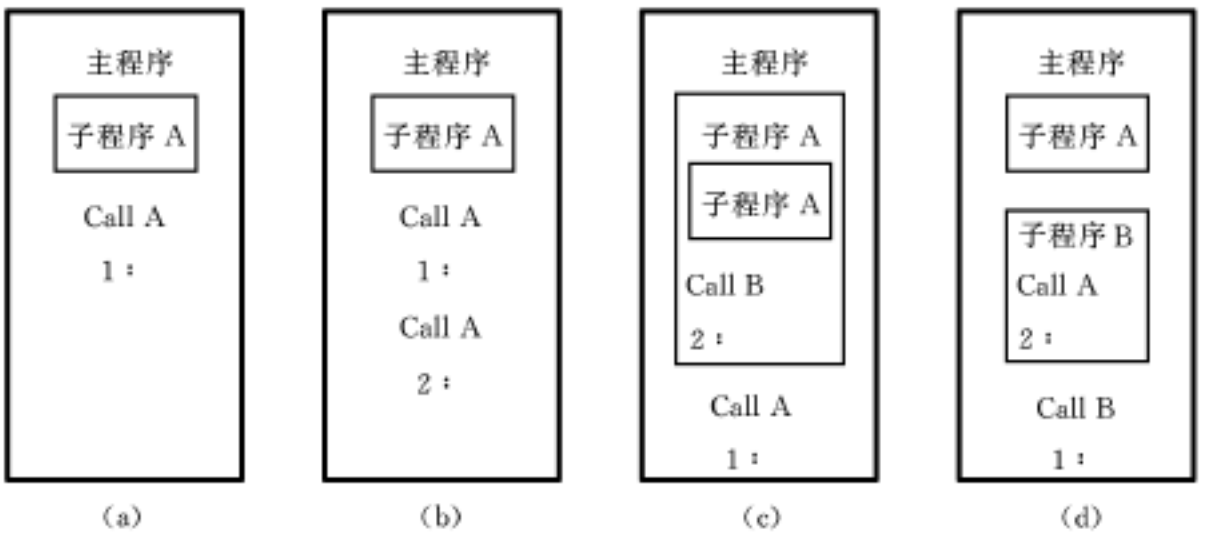


图 3 .1　子程序调用的几种形式

(a) 1 次调用;(b) n 次调用;(c) 嵌套调用;(d) 平行调用

对于图(a),当主程序执行到 call A 时,系统自动地保存好 1：语句在指令区的地址(为了叙述方便,不妨视地址为 1,下同),便于调用 A 结束后能从系统获得返回地址,按地址执行下一条指令。

对于图(b),主程序中有多次对同一子程序 A 的调用。在第 i 次重复调用子程序 A 之前,系统自动地保存好地址 i,便于第 i 次调用 A 结束后能顺利地按地址 i 返回。这种情况与图(a)所示的不一样,保存的地址有多个,但在某一时刻最多只保存一个地址,一旦获得地址返回后,保留的地址 i 被释放。

对于图(c)、(d),当主程序执行到 call A(或 call B)时,系统自动地保存好地址1,转入子程序 A(或 B),在第二次调用子程序 call B(或 call A)时,再保存好地址2,执行完子程序 B(或 A)之后,获得地址2返回,继续执行。当子程序 A(或 B)执行完之后,获得地址1并返回,接着执行到主程序完。图(c)、(d)两种情况与图(b)不同,在某一时刻可能保存多个地址,而且后保存的地址先释放。因此,对返回地址的管理,需要用栈方式实现。

由此看出,系统在实现子程序的调用时,要用栈方式管理调用子程序时的返回地址。

除了对地址的管理以外,系统要为支持程序的模块化而提供局部变量的概念及实现。在内部实现时,编译系统为每个将要执行的子程序的局部变量(包括形参)分配存储空间,并限定这些局部变量不能由该子程序以外的程序直接访问,子程序之间的数据传送通过参数或全局变量实现。这样就保证了局部变量及其特性的实现。将这些局部变量、返回地址一同放在栈顶,就能较好地实现这一要求。于是,被调过程对其局部变量的操作是对栈顶中相应变量的操作,这些局部变量随着被调过程的执行而存在于栈顶,当被调过程结束时,局部变量从栈顶撤出。

2.值的回传

在计算机的高级语言中,实参与形参的数据传送以两种方式实现,一是按值传送(比如 PASCAL 语言中的值参),二是按址传送(比如 PASCAL 语言中的变参)。由于形实结合的方式不同,在调用前后,值参对应的实参的值是不发生变化的,而变参所对应的实参的值将执行过程中对变参的修改进行回传。对于变参回传值,计算机内部实现有两种方法。

(1) 两次值传送方式。按指定类型为变参设置相应的存储空间,在执行调用时,将实参值传送给变参,在返回时将变参的值回传给实参。

(2) 地址传送方式。在内部将变参设置成一个地址,调用时首先执行地址传送,将实参的地址传送给变参,在子程序执行过程中,对变参的操作实际上变成对所对应的实参的操作。

在下面讨论递归问题时,对变参的值的回传用第一种方式,即两次值传送方式。

除了变参外,还有函数的值的回传。有两个原因使这种回传不能直接进行:一是因为要将仅能在被调用层使用的变量的值传送到调用层的变量,所以不能在调用层直接进行;二是由于各调用操作中实参的多样性,使得传送不能在被调用层直接进行。鉴于此,借用一个全局变量,通过栈实现回传,但是,这种方式会造成栈的结构上的不一致,以及调用操作的次序问题等不便之处。由于在某个时刻,最多只有一个返回操作,所以在后面讨论中,专设一个回传变量的全局变量,用于存放回传值。

3.子程序调用的内部操作

综上所述,子程序调用的内部实现为两个方面。

(1) 在执行调用时,系统至少执行的操作。

① 返回地址进栈,同时在栈顶为被调子程序的局部变量开辟空间;

② 为被调子程序准备数据:计算实参值,并赋给对应的栈顶的形参;

③ 将指令流转入被调子程序的入口处。

(2) 在执行返回操作时,系统至少执行的操作。

① 若有变参或是函数,将其值保存到回传变量中;

② 从栈顶取出返回地址;

③ 按返回地址返回;

④ 若有变参或是函数,从回传变量中取出保存的值传送给相应的变量或位置上。

3.1.2 递归过程的内部实现原理

一个递归过程的执行类似于多个子程序的嵌套调用,递归过程是自己调用自己本身代码。如果把每一次的递归调用视为调用自身代码的复制件,则递归实现过程基本上和一般子程序的实现相同。当然,在内部实现时,系统并不是去复制一份程序代码放到内存,而是采用代码共享的方式,其细节不必深究。于是,前面提出的调用前系统做的三件事,与调用结束时系统还要做的四件事,对于递归调用仍亦如此。

3.2 递归转非递归

对于那些本来就要用递归设计求解的问题,在设计其算法时能不能既发挥递归表示直观及易于证明算法正确的特点,又克服由于使用递归而带来总开销增加的不足呢?为此,建议采用如下办法:在算法设计的初期阶段使用递归,一旦所设计的递归算法被证明为正确且确信是一个好算法时,就可以消去递归,把该算法翻译成与之等价的、只使用迭代的算法。这一翻译过程可使用一组简单的转换规则来完成,也可根据具体情况将所得到的迭代算法作进一步的改进,以提高迭代过程的效率。

下面介绍的是将直接递归过程翻译成只使用迭代过程的一组规则。对于间接递归过程的处理只需把这组规则稍作修改即可。所谓翻译主要是,将递归过程中出现递归调用的地方,用等价的非递归代码来代替,并对 return 语句作适当处理。这组规则如下。

(1) 在过程的开始部分,插入说明为栈的代码并将其初始化为空。在一般情况下,这个栈用来存放参数、局部变量和函数的值,每次递归调用的返回地址也要存入栈。

(2) 将标号 L_i 附于第一条可执行语句。然后,对于每一处递归调用都用两组执行下列规则的指令来代替。

(3) 将所有参数和局部变量的值存入栈。栈顶指针可作为一个全程变量来看待。

(4) 建立第 i 个新标号 L_i,并将 i 存入栈。这个标号的 i 值将用来计算返回地址。此标号放在规则(7)所描述的程序段中。

(5) 计算这次调用的各实参(可能是表达式)的值,并把这些值赋给相应的形参。

(6) 插入一条无条件转向语句,转向过程的开始部分。

(7) 如果此过程是函数,则对递归过程中含有此次函数调用的那条语句作如下处理:将该语句的此次函数调用部分用从栈顶取回该函数值的代码来代替,其余部分的代码按原描述方式照抄,并将规则(4)中建立的标号附于这条语句上。如果此过程不是函数,则将规则(4)中建立的标号附于规则(6)所产生的转移语句后面的那条语句。

以上步骤是消去过程中各处的递归调用,下面对递归过程中出现的 return 语句进行处理(将纯过程结束处的 end 语句看成是一条没有值与其相联系的 return 语句)。在每个有 return 语句的地方,执行下述规则:

(8) 如果栈为空,则执行正常返回。

(9) 否则,将所有输出参数(即理解为 out 或 inout 型的参数)的当前值赋给栈顶上的那些对应的变量。

(10) 如果栈中有返回地址标号的下标,就插入一条此下标从栈中退出的代码,并把这个下标值赋给一个未使用的变量。

(11) 从栈中退出所有局部变量和参数的值并把它们赋给对应的变量。

(12) 如果这个过程是函数,则插入以下指令,这些指令用来计算紧接在 return 语句后面的表达式并将结果值存入栈顶。

(13) 用返回地址标号的下标实现对该标号的转向。

在一般情况下,使用上述规则都可将一个直接递归过程正确地翻译成与之等价的只使用迭代的过程。它的效率通常比原递归模型要高,进一步简化这程序可使效率再次提高。

下面举一个递归化迭代的例子,虽然例中的问题最好是使用迭代来求解,若用递归描述反而变得不很直观,但用它有助于读者对以上规则有一些感性认识。

例 3.1　写一个求数组 A(1：n)中最大元素的过程。

算法 3.1　递归求取最大值

```
procedure MAX1(i)
// 这是一个函数过程,它返回使 A(k)是 A(1：n)中最大元素的最大下标 k//
  global integer n, A (1：n),j, k ;
  integer i
  if i< n then j ← MAX1 (i+ 1)
          if A(i) > A(j) then k ← i
                  else  k ← j
          endif
          else k ← n
  endif
        return(k)
end MAX1
```

用几个简单的数据在这算法上实验一下立即就可理解这个递归模型。在运行时间上,由于过程调用和隐式栈管理方面的消费使我们自然考虑到消去递归。

算法 3.2　与算法 3.1 等价的迭代算法

```
procedure MAX2(i)
    local integer j, k; global integer n, A (1：n);
    integer i
    integer STACK (1：2 × n);                          //规则(1)//
    top ← 0                                            //规则(1)//
L1 : if i< n                                           //规则(2)//
    then top ← top+ 1; STACK (top) ← i                 //规则(3)//
         top ← top + 1; STACK (top) ← 2;               //规则(4)//
         i ← i+ 1                                      //规则(5)//
         go to L1                                      //规则(6)//
```

```
        L2：j ← STACK (top); top ← top - 1                    //规则(7)//
            if A(i) > A(j) then k←I
                    else k←j
            endif
        else k←n
        endif
    if top = 0 then return(k)                                  //规则(8)//
    else addr ← STACK (top); top ← top - 1                     //规则(10)//
        i ← STACK (top); top ← top - 1                         //规则(11)//
        top ← top + 1 ; STACK (top ) ← k                       //规则(12)//
        if addr = 2 then go to L2 endif                        //规则(13)//
    endif
end MAX2
```

这个迭代过程可以通过分析它的运算方式来简化。由于过程只返回到一个地方,所以不必重复存放返回地址。另外,因为在任何时刻只有一个函数值,即当前最大值的下标,故可以把这个值不存入栈而是存放在一个单变量中。由于过程中只有一个参变量 i,事实上,每进行一次新的递归,调用 i 的值就加 1,即退出一层递归恢复原来那层递归的 i 值,只比要退出的这层递归中的 i 值少 1,因此参变量 i 值也不需要使用栈。这样一来就可将栈完全取消。将 i 置 n 并用 k 存放当前最大值的下标还可取消由 go to L1 所产生的循环。经过这一系列简化所导出的过程就是算法 3.3。

算法 3.3　算法 3.2 的改进模型

```
procedure MAX3 (A,n)
    integer i, k, n;
    i←k←n
while i > 1 do
    i←i - 1
    if A(i) > A(k) then k←k endif
    repeat
    return (k)
end MAX3
```

消去递归的目的是为了产生效率更高的、在计算上等效的迭代程序。因为在某些场合下可能还有更简单的转换规则,所以不必在任何情况下死套前面所叙述的 13 条规则,而应具体情况具体分析。例如,若过程只有最后一条语句是递归调用,则可通过简单地计算过程调用中实参的值并转向过程开始部分来消去递归,而且不需要栈。下面给出这样一个例子。

例 3.2　求整数 a 与 b 的最大公因数。

算法 3.4　求最大公因数

```
procedure GCD (a,b)
    //假设 a > b≥0//
    if b = 0 then return (a)
        else return (GCD (b,a mod b))
```

```
        endif
    end GCD
```

消去递归后得到下面程序。

算法 3.5　与算法 3.4 等价的迭代算法

```
    procedure GCD1 (a,b)
        L1 : if b=0 then return (a)
                    else t ← b; b←a mod b; a←t; go to L1
             endif
    end GCD 1
```

稍作整理可得算法 3.6。

算法 3.6　算法 3.5 的改进模型

```
    procedure GCD2 (a,b)
        while b≠0 do
            t←b; b←a mod b; a←t
        repeat
        return (a)
    end GCD2
```

当然,如果所使用机器的有关编译程序能将递归过程编译成有效的代码,则完全不必将递归化为迭代。

3.3　递归算法设计

哪些问题可以用递归求解,这是首先应该明白的。如果同时满足以下 3 个要求:

(1) 问题 P 的描述涉及规模,即 P(size);

(2) 规模发生变化后,问题的性质不发生变化;

(3) 问题的解决有出口。

则可用递归求解,其表现形式为

```
    procedure P(参数表);
    begin
        if  递归出口
          then 简单操作
        else
          begin 简单操作;call P;简单操作  end;
    endp;
```

下面通过几个例子进行示范。

例 3.3　[简单的 0/1 背包问题]设一背包可容物品的最大质量为 m,现有 n 件物品,质量为 $m_1, m_2, \cdots, m_n, m_i$,均为正整数,要从 n 件物品中挑选若干件,使放入背包的质量之和正好为 m。

由于 0/1 背包问题中,对第 i 件物品要么取,要么舍,不许取一部分,因此,这个问题可能有解,也可能无解。于是,用布尔函数描述问题。

本题满足递归求解问题的 3 个要求。在递归描述中,涉及的必要参数是:当前背包的剩余容量,当前可选物品的质量序列(序列是固定的),序列的最大下标决定了序列的一个子序列。因此,必要的参数是 m 和 n。

具体实现时,要找准递归的出口、递推的关系。这两部分是保证递归算法的正确性的必要条件。用 knap(m,n)表示问题。

(1) 先取最后一件物品 m_n 放入包中,若 $m_n = m$,则 knap←true;

(2) 若 $m_n < m$,则 $m - m_n > 0$。如果还有可选物品,即 $n > 1$,就变为考虑

$$knap(m - m_n, n - 1)$$

是否可行的问题,也就是当选中的是 m_n 时,看子问题 $knap(m - m_n, n - 1)$ 是否有解。如果有解,则

$$knap(m,n) \xrightarrow{转化} knap(m - m_n, n - 1)$$

否则

$$knap(m,n) \xrightarrow{转化} knap(m, n - 1)$$

即放弃 m_n,在 $m_1, \cdots, m_{n-1}$ 上考虑 0/1 背包问题。

(3) 若 $m_n > m$,则第 n 件物品不能装入包中,这时如果还剩有可选物品(即 $n > 1$),那么

$$knap(m,n) \xrightarrow{转化} knap(m, n - 1)$$

根据以上分析,有算法如下:

```
function knap(m,n)
begin
case m - mn      of
     : = 0 : knap  ← true;
: > 0 : begin
          if n >1 then
          if knap (m - mn, n - 1) = true then
            knap  ← true;
          else
           knap  ← knap (m,n - 1)
          else
           knap  ← false;
          end;
: < 0: begin
        if n > 1 then
         knap  ←knap (m,n - 1)
        else
         knap  ←false;
        end;
endcase;
endp;
```

例 3.4 [n 阶 Hanoi 塔问题]有 n 个盘子依其半径大小套在柱子 A 上,其中半径大的在底下。柱子 B 和 C 没套盘子,现要将 A 上的盘子换到 C 上,要求每次只能移一个,并且不

容许将大盘子压在小盘子的上面。

为了叙述方便,在移动盘子的过程中,3 根柱子分别称为源柱、辅助柱、目标柱,初始时它们分别是 X,Y,Z;对盘子从小到大编号 1,2,3,…,n。

显然,这是一个直接递归问题。下面通过不完全归纳法,找出递归出口和递归关系。

当 n = 1 时,$X \xrightarrow{1} Z$。

当 n = 2 时,$X \xrightarrow{1} Y$, $X \xrightarrow{2} Z$, $Y \xrightarrow{1} Z$。

当 n = 3 时,它和 n = 2 时的递归关系是,将源柱为 X、目标柱为 Z 的 3 阶 Hanoi 塔问题分解为源柱为 X、目标柱为 Y 的 2 阶 Hanoi 塔问题;将源柱 X 上的 3 号盘子挂到目标柱 Z 上;源柱为 Y、目标柱为 Z 的 2 阶 Hanoi 塔问题求解等 3 个子问题。由于 2 阶 Hanoi 塔问题已找到了求解过程,源柱 X 上的 3 号盘子直接挂到目标柱 Z 是可行的。因此,3 阶和 2 阶 Hanoi 塔问题的递归关系已明确。

事实上,2 阶和 1 阶的 Hanoi 塔问题也存在这种递归关系。推而广之。如果用 Hanoi(n,x,y,z)表示 n 阶 Hanoi 塔问题,则 n 阶 Hanoi 塔问题与 n - 1 阶的 Hanoi 塔问题之间的递归关系也是由 3 个子问题合成的:

(1) Hanoi(n - 1,X,Z,Y);

(2) $X \xrightarrow{n} Z$;

(3) Hanoi(n - 1,Y,X,Z)。

这是递归关系,出口是 n - 1 时的情况。

例 3.5　[棋子移动]有 2n 个棋子(n≥4)排成一行,白子用 0 代表,黑子用 1 代表,n = 5 的初始状态为

0 0 0 0 0 1 1 1 1 1 _ _(右边至少有两个空位)

移动规则是:每次必须同时移动相邻两个棋子,颜色不限,移动方向不限;每次移动必须跳过若干棋子,不能调换两个棋子的位置。要求最后成为

0 1 0 1 0 1 0 1 0 1

下面用不完全归纳法找出口和递归关系:

n = 4	0 0 0 0 1 1 1 1 _ _	
第一步	0 0 0 _ _ 1 1 1 0 1	(4,5)→(9,10)
第二步	0 0 0 1 0 1 1 _ _ 1	(8,9)→(4,5)
第三步	0 _ _ 1 0 1 1 0 0 1	(2,3)→(8,9)
第四步	0 1 0 1 0 1 _ _ 0 1	(7,8)→(2,3)
第五步	_ _ 0 1 0 1 0 1 0 1	(1,2)→(7,8)
n = 5	0 0 0 0 0 1 1 1 1 1 _ _	
第一步	0 0 0 0 _ _ 1 1 1 1 0 1	(5,6)→(11,12)
第二步	0 0 0 0 1 1 1 1 _ _ 0 1	(9,10)→(5,6)

这是前 8 枚棋子为 n = 4 的情况,移法如 n = 4 的第一步到第五步,用同样的方法可完成 n = 5 时的移子。

n = 6　　0 0 0 0 0 0 1 1 1 1 1 1 _ _

第一步　　0 0 0 0 0 _ _ 1 1 1 1 1 0 1　　(6,7)→(13,14)

第二步　　0 0 0 0 0 1 1 1 1 1 _ _ 0 1　　(11,12)→(6,7)

前 10 枚棋子为 n = 5 的情况。由此归纳如下。

n = 4 是递归的出口,在退出时做 5 个移动操作:

```
move    (4,5)→(9,10)
move    (8,9)→(4,5)
move    (2,3)→(8,9)
move    (7,8)→(2,3)
move    (1,2)→(7,8)
```

如果 2n 个棋子的移动用 chess(n)表示,则递归关系是

```
move    (n,n+1)→(2n+1,2n+2);
move    (2n-1,2n)→(n,n+1);
call chess (n-1);
```

于是,递归过程如下:

```
procedure chess(n);
begin
if n=4 then
  begin
    move    (4,5)→(9,10)
    move    (8,9)→(4,5)
    move    (2,3)→(8,9)
    move    (7,8)→(2,3)
    move    (1,2)→(7,8)
  end
else
 begin
   move    (n,n+1)→(2n+1,2n+2);
   move    (2n-1,2n)→(n,n+1);
   call chess(n-1);
   end;
endp;
```

例 3.6　求 n 个元素的全排列。

分析: n = 1　　输出 a_1;

n = 2　　输出 $a_1\ a_2$;

$a_2\ a_1$;

n = 3　　输出

$a_1\ a_2\ a_3$;

$a_1\ a_3\ a_2$;

$a_2\ a_1\ a_3$;

$a_2\ a_3\ a_1$;

$a_3\ a_2\ a_1$;

$a_3\ a_1\ a_2$;

分析 n = 3,全部排列分成如下 3 类:

(1) a_1 类:a_1 之后跟 a_2,a_3 的所有全排列;

(2) a_2 类:a_2 之后跟 a_1,a_3 的所有全排列;

(3) a_3 类:a_3 之后跟 a_2,a_1 的所有全排列。

将(1)中 a_1,a_2 的互换位置,得到(2);将(1)中 a_1,a_3 的互换位置,得到(3)。它说明可以用循环的方式重复执行交换位置,后面跟随剩余序列的所有排列,对剩余序列再使用这个方法,这就是递归调用,当后跟的元素没有时就得到递归的边界。于是:

```
procedure  range(a,k,n)                    //求 a 的第 k 到 n 个元素的全排列//
begin
    if k = n then
     print (a)
    else
    for i ← k to n do
    begin
      a[k] ←→ a [i];
        call range (a, k + 1, n)
      end;
endp;
```

在主程序中的调用是

```
call range (a,1,n)
```

例 3.7　[自然数拆分]任何一个大于 1 的自然数 n,总可以拆分成若干个小于 n 的自然数之和,试求 n 的所有拆分(用不完全归纳法)。

n = 2　　可拆分成　　2 = 1 + 1

n = 3　　可拆分成　　3 = 1 + 2

= 1 + 1 + 1

n = 4　　可拆分成　　4 = 1 + 3

= 1 + 1 + 2

= 1 + 1 + 1 + 1

= 2 + 2

……

n = 7　　可拆分成　　7 = 1 + 6

= 1 + 1 + 5

= 1 + 1 + 1 + 4

= 1 + 1 + 1 + 1 + 3

= 1 + 1 + 1 + 1 + 1 + 2

= 1 + 1 + 1 + 1 + 1 + 1 + 1
= 1 + 1 + 1 + 2 + 2
= 1 + 1 + 2 + 3
= 1 + 2 + 4
= 1 + 2 + 2 + 2
= 1 + 3 + 3
= 2 + 5
= 2 + 2 + 3
= 3 + 4

用数组 a 存储完成 n 的一种拆分。从上面不完全归纳法的分析可知当 n = 7 时,按 a[1]分类,有 a[1] = 1, a[1] = 2, …, a[1] = n/2,共 n/2 大类拆分。在每一类拆分时,a[1]←i, a[2]←n - i,从 k = 2,继续拆分从 a[k]开始,a[k]能否再拆分取决于 a[k]/2 是否大于等于 a[k - 1]。递归过程的参数 t 指向要拆分的数 a[k],于是有算法

```
procedure  split(t:int);
  begin
  for k←1 to t do
  write (a [k]);
  j←t; L← a[j];
  for i←a[j - 1] to L/ 2 do
      begin
      a[j] ←i;      a[j + 1] ← L - i;     call split(j + 1);
      end;
  endp;
procedure main(n)
  begin
  for i←1 to n/ 2 do
   begin
      a[1] ← i;      a[2] ←n - i;      call split(2);
   end;
  endp;
```

3.4 递归关系式的计算

3.4.1 递归算法的时间复杂度分析

根据递归算法的设计思想,可以通过建立算法时间复杂度的递归关系式,然后求得算法的时间复杂度。下面给出几个具体的例子。

例 3.8 在前面求数组 A[1：n]的最大元的问题(见例 3.1)中,这里采用另一种方法编写递归算法,并用非形式化的形式,即自然语言描述此算法。

解 递归算法如下:

```
procedure M[j,j]
  // 这是一个函数过程,它将 A[i,j]中最大元赋予 A[i,j]//
  (1) 当 j,i<2 时,采用逐一比较的方法求出 A[i:j]的最大元,并给 A[i:j]赋相应值;
  //  A[i:j]中元素个数≤2 时//
  (2) 当 j - i≥2 时,
      ① L←⌊(i + j)/2⌋
      ② 若 M[i,L] ≥M[L + 1,j]则 M[j,j]←M[i,L],否则 M[j,j]←M[L + 1,j]
end M
```

下面对此算法进行时间复杂度分析。

设算法的时间复杂度为 $T(n)$,其中 $n = j - i + 1$,即为 $A[i,j]$中元素个数,则

$$T(n) = \begin{cases} C_1 & n \leq 2 \\ 2T\left(\frac{n}{2}\right) + C_2 & n > 2 \end{cases}$$

所以

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + C_2 = 2^2 T\left(\frac{n}{2^2}\right) + (2+1)C_2 \\ &= 2^{\log_2 n}\left(\frac{n}{2^{\log_2 n}}\right) + (1 + 2 + 2^2 + \cdots + 2^{\log_2 n - 1})C_2 \\ &= nC_1 + (n-1)C_2 = \Theta(n) \end{aligned}$$

例 3.9　一个楼有 n 个台阶,有一个人上楼有时一次跨一个台阶,有时一次跨两个台阶,编写一个算法,计算此人有几种不同的上楼方法,并分析算法的时间复杂度。

解　这里设计一个递归算法。

```
procedure H(n)
  // H(n)是个函数过程,算法将方法数赋予 H(n)//
  (1) 当 n = 1 时,H(1)←1
  (2) 当 n = 2 时,H(2)←1
  (3) 当 n > 2 时,H(n)←H(n - 1) + H(n - 2)
  // H(n)等于第一次跨一个台阶和第一次跨两个台阶的方法数的和//
end H
```

算法时间复杂度分析:设时间复杂度为 $T(n)$,则

$$T(n) = \begin{cases} C & n \leq 2 \\ T(n-1) + T(n-2) & n > 2 \end{cases}$$

所以

$$T(n) \leq 2T(n-1) \leq 2^2 T(n-2) \leq \cdots \leq 2^{n-1} T(1) = O(2^n)$$

递归算法的特点是思路清晰,算法的描述简洁且易理解,递归算法大多可用数学的形式表示,即表示成一个递归关系式,例如,上例中算法可表示成

$$H(n) = \begin{cases} 1 & n = 1 \\ 2 & n = 2 \\ H(n-1) + H(n-2) & n > 2 \end{cases}$$

当递归关系式比较复杂时,若直接按递归关系式实现算法,则算法的时间复杂度往往是非多项式时间,这是递归算法的一个不足。一种有效的解决方法是利用数学的方法解递归关系式,将结果或者中间结果编程实现,这样就可以大大地降低时间复杂度。下面将讨论递

归关系式的解法,重点讨论线性常系数递归关系式的解法。

3.4.2 k 阶线性齐次递归关系式的解法

定义 3.1 递归关系式

$$a_n=\begin{cases} c_1a_{n-1}+c_2a_{n-2}+\cdots+c_ka_{n-k}+d(n) & n\geq k\geq 1 \\ a_i=b_i & 0\leq i\leq k-1 \end{cases}$$

其中,$c_k\neq 0$,c_1,c_2,$\cdots$,c_k,b_0,b_1,$\cdots$,b_{k-1}是给定的常数,称为 k 阶线性常系数递归关系式。当 $d(n)=0$ 时,称此递归关系式为齐次的。

1.一阶线性齐次递归关系式

对于一阶线性齐次递归关系式

$$\begin{cases} a_n=c_1a_{n-1} & c_1\neq 0 \\ a_0=b_0 \end{cases}$$

可采用递步推的方法求得 $a_n=C_1^{n-1}b$。

2.二阶线性齐次递归关系式

对于二阶线性齐次递归关系式

$$\begin{cases} a_n+ba_{n-1}+ca_{n-2}=0 & c\neq 0 \\ a_0=c_0,a_1=c_1 \end{cases}$$

令母函数 $G(x)=a_0+a_1x+a_2x^2+\cdots$,则$(1+bx+cx^2)G(x)$ 计算如下:

$$\begin{aligned} G(x)&=a_0+a_1x+a_2x^2+\cdots \\ bxG(x)&=ba_0x+ba_1x^2+\cdots \\ +)cx^2G(x)&=ca_0x^2+\cdots \end{aligned}$$

$$(1+bx+cx^2)G(x)=a_0+(a_1+ba_0)x+(a_2+ba_1+ca_0)x^2+\cdots+(a_n+ba_{n-1}+ca_{n-2})x^n+\cdots$$

因为 $$a_n+ba_{n-1}+ca_{n-2}=0$$

所以 $$(1-bx+cx^2)G(x)=a_0+(a_1+ba_0)x$$

$$G(x)=\frac{a_0+(a_1+ba_0)x}{1+bx+cx^2}$$

和 $a_n+ba_{n-1}+ca_{n-2}=0$ 对应的分母 $1+bx+cx^2$ 用 $D(x)$表示,即 $D(x)=1+bx+cx^2$。与 $D(x)$对应的 $K(x)=x^2+bx+c$,称为与递归关系对应的特征多项式,$K(x)=0$,即 $x^2+bx+c=0$ 称为特征方程,它的根为

$$r_{1,2}=\frac{-b\pm\sqrt{b^2-4ac}}{2}$$

称为特征根,则

$$D(x)=1+bx+cx^2=(1-r_1x)(1-r_2x)$$

针对 r_1,r_2 为实根还是复根,r_1 是否等于 r_2?分别讨论如下。

(1) 当 r_1 与 r_2 为互不相同的实数时,因为

$$G(x)=\frac{a_0+(a_1+ba_0)x}{(1-r_1x)(1-r_2x)}=\frac{A}{1-r_1x}+\frac{B}{1-r_2x}=\sum_{n=0}^{\infty}[Ar_1^n+Br_2^n]x^n$$

所以,当 $n=0$ 时,$A+B=a_0$;当 $n=1$ 时,$Ar_1+Br_2=a_1$。其中,

$$A=\begin{vmatrix} a_0 & 1 \\ a_1 & r_2 \end{vmatrix} \Big/ \begin{vmatrix} 1 & 1 \\ r_1 & r_2 \end{vmatrix} = \frac{a_0 r_2 - a_1}{r_2 - r_1} = \frac{a_1 - a_0 r_2}{r_1 - r_2} \quad B = \frac{a_1 - a_0 r_1}{r_2 - r_1}$$

$$a_n = Ar_1^n + Br_2^n = \frac{a_1 - a_0 r_2}{r_1 - r_2}(r_1)^n + \frac{a_1 - a_0 r_1}{r_2 - r_1}(r_2)^n$$

(2) 当 $r_1 \neq r_2$ 时,但 r_1 和 r_2 是一对共轭复根时,设

$r_1 = \rho e^{i\theta}$, $r_2 = \rho e^{-i\theta}$

$(r_1)^n = \rho^n e^{in\theta} = \rho^n(\cos n\theta + i\sin n\theta)$

$(r_2)^n = \rho^n e^{-in\theta} = \rho^n(\cos n\theta - i\sin n\theta)$

$a_n = A_1(r_1)^n + A_2(r_2)^n = A_1\rho^n(\cos n\theta + i\sin n\theta) + A_2\rho^n(\cos n\theta - i\sin n\theta)$

$= (A_1 + A_2)\rho^n \cos n\theta + (A_1 - iA_2)\rho^n \sin n\theta$

由于 A_1 和 A_2 是待定常数,令 $k_1 = A_1 + A_2$, $k_2 = A_1 - iA_2$;k_1 和 k_2 也是待定常数,故 $a_n = k_1\rho^n \cos n\theta + k_2\rho^n \sin n\theta$。

(3) 当 $r_1 = r_2$ 时,即 $b^2 = 4c$,令 $r = r_1 = r_2 = \frac{b}{2}$。

$$G = \frac{a_0 + (a_1 + ba_0)x}{(1 - rx)^2} = \frac{A}{1 - rx} + \frac{B}{(1 - rx)^2}$$

因为

$$\frac{1}{(1 - rx)^2} = (1 + rx + r^2x^2 + \cdots)/(1 - x) = 1 + 2x + 3x^2 + \cdots$$

所以

$$G = A\sum_{h=0}^{\infty} r^h x^h + B\sum_{h=0}^{\infty}(h+1)r^h x^h = \sum_{h=0}^{\infty}[A + B(h+1)]r^h x^h$$

$$a_n = [A + B(n+1)]r^n = [(A+B) + Bn]r^n = [h + kn]r^n$$

$$r = \frac{b}{2}$$

当 $n=0$ 时,有 $a_0 = h$,当 $n=1$ 时,有 $a_1 = (h+k)r$,则

$$k = \frac{a_1}{r} - a_0$$

所以对于二重根 r,

$$a_n = \left[a_0 + \left(\frac{a_1}{r} - a_0\right)n\right]r^n$$

例 3.10　解递归关系式

$$\begin{cases} a_n - a_{n-1} - 12a_{n-2} = 0 \\ a_0 = 3 \\ a_1 = 26 \end{cases}$$

解　由特征方程 $x^2 - x - 12 = 0$,求得特征根为 $r_1 = -3$, $r_2 = 4$,所以

$$a_n = \frac{a_1 - a_0 r_2}{r_1 - r_2}(r_1)^n + \frac{a_1 - a_0 r_1}{r_2 - r_1}(r_2)^2 = 5 \times 4^n - 2 \times (-3)^n$$

例 3.11　解递归关系式 $\begin{cases} a_n = a_{n-1} - a_{n-2} \\ a_1 = 1, a_2 = 0 \end{cases}$

解　递归关系的特征方程为

$$x^2 - x + 1 = 0$$

$$x = \frac{1 + \sqrt{-3}}{2} = \frac{1}{2} \pm \frac{\sqrt{3}}{2}i = \cos\frac{\pi}{3} \pm i\sin\frac{\pi}{3} = e^{\pm\frac{\pi}{3}i}$$

所以有 $a_n = A_1\cos\frac{n}{3}\pi + A_2\sin\frac{n}{3}\pi$

……

$$a_2 = -\frac{1}{2}A_1 + \frac{\sqrt{3}}{2}A_2 = 0$$

$$a_1 = \frac{1}{2}A_1 + \frac{\sqrt{3}}{2}A_2 = 1$$

因为 $A_1 = 1, A_2 = \frac{1}{3}\sqrt{3}$

所以 $a_n = \cos\frac{n\pi}{3} + \frac{1}{\sqrt{3}}\sin\frac{n\pi}{3}$

例 3.12 **解递归关系式**

$$\begin{cases} a_n + 4a_{n-1} + 4a_{n-2} = 0 \\ a_0 = 1, a_1 = 4 \end{cases}$$

解 特征方程

$$x^2 + 4x + 4 = (x - 2)^2$$

$$D(x) = (1 - 2x)^2, r = 2$$

$$a_n = (h + kn)2^n$$

$$a_0 = h = 1, a_1 = (1 + k)2 = 4, k = 1$$

则 $a_n = (1 + n)2^n$

3. k **阶线性齐次递归关系式**

对于任意阶线性齐次递归关系式

$$\begin{cases} a_n + c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} = 0 \\ a_l = d_l, 0 \le l \le k - 1, k \ge 1, d_l \text{是已知的常数}, 1 \le k \end{cases}$$

令 $a_0, a_1, a_2, \cdots$**序列的母函数为**

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots$$

$$D(x) = a_0 + c_1 x + c_2 x^2 + \cdots + c_k x^k$$

$$D(x)A(x) = (1 + c_1 x + c_2 x^2 + \cdots + c_k x^k)(a_0 + a_1 x + \cdots)$$

$$= a_0 + (a_1 + c_1 a_0)x + (a_2 + c_1 a_1 + c_2 a_0)x^2$$

$$+ (a_3 + c_1 a_2 + c_2 a_1 + c_3 a_0)x^3 + \cdots$$

$$+ (a_{k-1} + c_1 a_{k-2} + c_2 a_{k-3} + \cdots + c_{k-1} a_0)x^{k-1} + \cdots$$

$$+ (a_m + c_1 a_{m-1} + c_2 a_{m-2} + \cdots + c_k a_{m-k})x^m + \cdots$$

由于 $m \ge k$**,有** $a_m + c_1 a_{m-1} + c_2 a_{m-2} + \cdots + c_k a_{m-k} = 0$**,则**

$$D(x) = a_0 + (a_1 + c_1 a_0)x + (a_2 + c_1 a_1 + c_2 a_0)x^2 + \cdots + (a_{k-1} + c_1 a_{k-2} + \cdots + c_{k-1} a_0)x^{k-1}$$

令 $p_{k-1}(x) = a_0 + (a_1 + c_1 a_0)x + (a_2 + c_1 a_1 + c_2 a_0)x^2 + \cdots + (a_{k-1} + c_1 a_{k-2} + \cdots + c_{k-1} a_0)x^{k-1}$

所以　$$A(x)=\frac{p_{k-1}(x)}{D(x)}$$

设特征方程

$$K(x)\equiv x^k+c_1x^{k-1}+c_2x^{k-2}+\cdots+c_k=0$$

有 s 个不同的根 $r_1,r_2,\cdots,r_s$，其中 h_i 是 r_i 的重根数，$k\le i\le s$，所以 $h_1+h_2+\cdots+h_s=k$，所以 $A(x)=\dfrac{p_{k-1}(x)}{(1-r_1x)^{h_1}(1-r_2x)^{h_2}\cdots(1-r_s)^{h_s}}$，根据部分分数法有 $A(x)=\sum_{i=1}^{s}\left[\sum_{j=1}^{h_i}\dfrac{A_j^{(i)}}{(1-r_ix)^j}\right]$，再利用广义二项式定理，有

$$\begin{aligned}a_n=&(b_0^{(1)}+b_1^{(1)}n+\cdots+b_{h_1-1}^{(1)}n^{h_1-1})r_1^n\\&+(b_0^{(2)}+b_1^{(2)}n+\cdots+b_{h_2-1}^{(2)}n^{h_2-1})r_2^n+\cdots\\&+(b_0^{(s)}+b_1^{(s)}n+\cdots+b_{h_s-1}^{(s)}n^{h_s-1})r_s^n\end{aligned}$$

其中，$b_j^{(i)}$ 是待定常数，$i=1,2,\cdots,s,j=0,1,\cdots,h_j-1$。$b_j^{(i)}$ 可通过解 k 个线性方程 $a_l=d_l$，$0\le l\le k-1$ 来确定。

例 3.13　解递归关系式

$$a_n-9a_{n-1}+26a_{n-2}-24a_{n-3}=0$$

$$a_0=6,a_1=17,a_2=53$$

解　递归关系的特征方程为 $K(x)=x^3-9x^2+26x-24=0$，因为

$$x^3-9x^2+26x-24=(x-2)(x^2-7x+12)=(x-2)(x-3)(x-4)$$

所以　$$a_n=A_12^n+A_23^n+A_34^n$$

再根据初始条件 $n=0,1,2$，分别得

$$\begin{cases}A_1+A_2+A_3=6\\2A_1+3A_2+4A_3=17\\4A_1+9A_2+16A_3=53\end{cases}$$

$$D=\begin{vmatrix}1&1&1\\2&3&4\\4&9&16\end{vmatrix}=2\qquad A_1=\frac{1}{2}\begin{vmatrix}6&1&1\\17&3&4\\53&9&16\end{vmatrix}=\frac{6}{2}=3$$

同理得

$$A_2=1,A_3=2$$

所以　$$a_n=3\times2^n+3^n+2\times4^n$$

例 3.14　解递归关系式

$$a_n+a_{n-1}-11a_{n-2}-13a_{n-3}+26a_{n-4}+20a_{n-5}-24a_{n-6}=0$$

$$a_0=7,a_1=4,a_2=37,a_3=32,a_4=163,a_5=646$$

解　因为 $K(x)=x^6+x^5-11x^4-13x^3+26x^2+20x-24=0$

$$24=2^3\times3\times1$$

所以　$$K(x)=(x+2)^3(x-3)(x-1)^2$$

$$D(x)=(1-2x)^3(1-3x)(1-x)^2$$

$$a_n=c_1n+c_2+c_33^n+c_4n^2(-2)^n+c_5n(-2)^n+c_6(-2)^n$$

根据初始条件 $a_0=7, a_1=4$ 可得

$$c_2+c_3+\cdots+c_6=7$$

$$c_1+c_2+3c_3-2c_4-2c_5-2c_6=4$$

$$2c_1+c_2+9c_3+16c_4+8c_5+4c_6=37$$

$$3c_1+c_2+27c_3-72c_4-24c_5-8c_6=32$$

$$4c_1+c_2+81c_3+256c_4+64c_5+16c_6=163$$

$$5c_1+c_2+243c_3-800c_4-160c_5-32c_6=646$$

$$c_1=-1, c_2=5, c_3=2, c_4=-1, c_5=4, c_6=0$$

即
$$a_n=-n+5+2\times 3^n+(-n^2+4n)(-2)^n$$

3.4.3 线性常系数非齐次递归关系式的解法

对于非齐次递归关系式,没有一个公式化的方法,我们仅就以下几种特殊类型进行讨论。

类型一 递归关系式为

$$\begin{cases} a_n+c_1a_{n-1}+c_2a_{n-2}+\cdots+c_ka_{n-k}=r^nb(n) & (3.1)\\ a_l=d_l & (3.2)\end{cases}$$

其中,$k\geq 1, l=0,1,2,\cdots,k-1, c_1,c_2,\cdots,c_k, r$ 和 d_l 是给定的常数,$b(n)$ 是关于 n 的一个 q 次多项式。

此类型的解题步骤是

(1) 确定 r 是与式(3.1)对应的特征方程,$x^k+c_1x^{k-1}+\cdots+c_k=0$ 的几重根,根的重数记为 m,若 r 不是特征方程的根,则记 $m=0$。

(2) 记 $\alpha_n=r^n(k_0n^m+k_1n^{m+1}+\cdots+k_gn^{m+g})$,将 α_n 作为 a_n 代入式(3.1),通过比较 n^i 项的系数确定 $k_0,k_1,\cdots,k_g$。

(3) 求式(3.1)对应的齐次递归关系式 $a_n+c_1a_{n-1}+c_2a_{n-2}+\cdots+c_ka_{n-k}=0$ 的含有 K 个参数的解 β_n。

(4) $a_n=\alpha_n+\beta_n$,其中 K 个参数由初始条件式(3.2)确定,上述方法的正确性证明从略。

例 3.15 解递归关系式 $a_n+3a_{n-1}-10a_{n-2}=(-7)^n n$

解 对应的特征方程

$$K(x)\equiv x^2+3x-10=(x+5)(x-2)=0$$

有两个特征根:2 和 -5,7 不是特征根,故其 $m=0$。

令
$$\alpha_n=(-7)^n(k_0+k_1n)$$

代入递归关系式,得

$$(-7)^n(k_0+k_1n)+3(-7)^{n-1}[k_0+k_1(n-1)]-10(-7)^{n-2}[k_0+k_1(n-2)]=(-7)^n n$$

等式两端除 $(-7)^{n-2}$,得

$$(-7)^2(k_0+k_1n)-21(k_0-k_1+k_1n)-10(k_0-2k_1+k_1n)=49n$$

$$(49k_1-21k_1-10k_1)n+(49k_0-21k_0-10k_0+21k_1+20k_1)=49n$$

$$18k_1n+(18k_0+41k_1)=49n$$

则 $k_1 = 49/18, k_0 = \frac{-41}{18} \times \frac{49}{48} = \frac{-2009}{324}$，所以，$\alpha_n = (-7)^n \left[\frac{-2009}{324} + \left[\frac{49}{18} \right] n \right]$。故解为

$$a_n = k_1 (2)^n + k_2 (-5)^n + (-7)^n \left[\frac{49}{18} n - \frac{2009}{324} \right]$$

k_1 和 k_2 是任意常数，由初始条件来确定。

例 3.16　解递归关系式　　$a_n - 3a_{n-1} + 2a_{n-2} = 6n^2$，　$a_0 = 6, a_1 = 7$

解　递归关系的特征方程为

$$x^2 - 3x + 2 = (x - 1)(x - 2) = 0$$

右端项 $6n^2$ 可以看做是 $6n^2 (1)^n$，故 $m = 1, q = 2, \alpha_n = (k_1 n + k_2 n^2 + k_3 n^3) 1^n$，代入递归关系式得

$$(k_1 n + k_2 n^2 + k_3 n^3) - 3[k_1 (n - 1) + k_2 (n - 1)^2 + k_3 (n - 1)^3]$$
$$+ 2[k_1 (n - 2) + k_2 (n - 2)^2 + k_3 (n - 2)^3] = 6n^2$$

即
$$[k_3 n^3 + k_2 n^2 + k_1 n] - 3[k_3 n^3 + (k_2 - 3k_3) n^2 + (k_1 - 2k_2 + 3k_3) n$$
$$+ (-k_1 + k_2 - k_3)] - 2[k_3 n^3 + (k_2 - 6k_3) n^2 + (k_1 - 4k_2 + 12k_3) n$$
$$- (2k_1 - 4k_2 + 8k_3)] = 6n^2$$
$$n^3 (k_3 - 3k_3 + 2k_3) + n^2 (k_2 - 3k_2 + 2k_2 + 9k_3 - 12k_3) + n(k_1 - 3k_1 + 6k_2$$
$$- 9k_3 + 2k_1 - 8k_2 + 24k_3) + (3k_1 - 3k_2 + 3k_3 - 4k_1 + 8k_2 - 16k_3) = 6n^2$$

则
$$-3k_3 n^2 + (-2k_2 + 15k_3) n + (-k_1 + 5k_2 - 13k_3) = 6n^2$$

所以
$$\begin{cases} -3k_3 = 6 \\ -2k_2 + 15k_3 = 0 \\ -k_1 + 5k_2 - 13k_3 = 0 \end{cases}$$

故
$$k_3 = -2, k_2 = -15, k_1 = -49$$

则有
$$\alpha_n = -49n - 15n^2 - 2n^3$$

因为对应的齐次式的解 $\beta_n = k_1 + k_2 2^n$，所以 $a_n = \alpha_n + \beta_n = -49n - 15n^2 - 2n^3 + k_1 + k_2 2^3$，其中 k_1 和 k_2 由初始条件来确定。

类型二　递归关系式为

$$\begin{cases} a_n + c_1 a_{n-1} + c_2 a_{n-1} + \cdots + c_k a_{n-k} = r_1^n b_1 (n) + r_2^h b_2 (n) + \cdots + r_s^n b_s (n) \\ a_l = d_l \end{cases}$$

其中，$k \geq 1, l = 0, 1, 2, \cdots, k - 1, c_1, c_2, \cdots, c_k, r_1, r_2, \cdots, r_s$ 及 d_l 是给定的常数，$b_i (n)$ 是关于 n 的 q_i 次多项式，$i = 1, 2, \cdots, S$。

此类型的解法是：按类型一的方法，分别求出 $a_n + c_1 a_{n-1} c_2 a_{n-2} + \cdots + c_k a_{n-k} = r_i^n bi(n)$ 对应的 $a_n^{(i)}$ 和 β_n，其中 β_n 中含 k 个参数，然后令 $a_n = \alpha_n^{(1)} + \alpha_n^{(2)} + \cdots + \alpha_n^{(s)} + \beta_n$，最后由初始条件确定 a_n 中的 k 个参数。

例 3.17　解递归关系式　$a_n - a_{n-1} - 6a_{n-2} = 10 \cdot 4^n - 7 \cdot 3^n$

解　因为齐次式的特征方程为

$$x^2 - x - 6 = (x - 3)(x + 2) = 0$$

利用类型一的方法求出

$$\alpha_n^{(1)} = \frac{80}{3} \times 4^n, \alpha_n^{(2)} = -\frac{21}{5} \times 3^n \times n, \beta_n = k_1(-2)^n + k_2 \times 3^n$$

所以 $a_n = \alpha_n^{(1)} + \alpha_n^{(2)} + \beta_n = \frac{80}{3} \times 4^n - \frac{21}{5} n \times 3^n + k_1(-2)^n + k_2 \times 3^n$。其中 k_1 和 k_2 可由初始条件确定。

习 题 三

3.1 求下列递归关系的一般解:

(1) $a_n - 4a_{n-1} = 5^n$

(2) $a_n + 6a_{n-1} = 5 \times 3^n$

(3) $a_n - 4a_{n-1} = 4^n$

(4) $a_n + 6a_{n-1} = 4(-6)^n$

(5) $a_n - 4a_{n-1} = 2 \times 5^n - 3 \times 4^n$

(6) $a_n - 4a_{n-1} = 7 \times 4^n - 6 \times 5^n$

(7) $a_n + 6a_{n-1} = (-6)^n(2n + 3n^2)$

(8) $a_n - 4a_{n-1} = (n - n^2)4^n$

(9) $a_n - a_{n-1} = 4n^3 - 6n^2 + 4n - 1$

(10) $a_n - 7a_{n-1} + 12a_{n-2} = 5 \times 2^n - 4 \times 3^n$

(11) $a_n + 2a_{n-1} - 8a_{n-2} = 3(-4)^n - 14 \times (3)^n$

(12) $a_n - 6a_{n-1} + 9a_{n-2} = 3^n$

(13) $a_n - 7a_{n-1} + 16a_{n-2} - 12a_{n-3} = 2^n + 3^n$

(14) $a_n - 2a_{n-1} = 2^n + 3^n + 4^n$

3.2 解下列递归关系:

(1) $a_n = na_{n-1}, a_0 = 1$

(2) $a_n - a_{n-1} = \frac{1}{2^n}, a_0 = 7$

(3) $a_n - a_{n-1} = \frac{1}{3^n}$

3.3 解递归关系式:

(1) $a_n = 3a_{n-1} + 3^n - 1, a_0 = 0$

(2) $a_n - 4a_{n-1} = 4_n, a_0 = 0$

3.4 10 个数字(0 到 9)和 4 个四则运算符(+, -, ×, ÷)组成的 14 个元素。求由其中的 n 个元素的排列构成一算术表达式的个数。

3.5 n 条直线将平面分成多少个域?假定无三线共点,且两两相交。

3.6 求 n 位二进制数中最后 3 位为 010 的数的个数。

3.7 求 n 位的二进制数中最后 3 位才第 1 次出现 010 的数的个数。

第 4 章

分　治　法

4.1　一般方法

当要求解一个输入规模为 n 且取值又相当大的问题时,直接求解往往是非常困难的,有的甚至根本没法直接求出。正确的方法是,每当遇到这类问题时,首先应仔细分析问题本身所具有的特性,然后根据这些特性选择适当的设计策略来求解。在将这 n 个输入分成 k 个不同子集合的情况下,如果能得到 k 个不同的可独立求解的子问题,其中 $1 < k \leq n$,而且在求出了这些子问题的解之后,还可找到适当的方法把它们合并成整个问题的解,那么,可考虑使用分治法来求解。这种求解的思想就是将整个问题分成若干个小问题后分而治之。通常,由分治法所得到的子问题与原问题具有相同的类型。如果得到的子问题相对来说还太大,则可反复使用分治策略将这些子问题分成更小的同类型子问题,直至产生出不用进一步细分就可求解的子问题。由此可知,分治法求解很自然地可用一个递归过程来表示。

在很多考虑使用分治法求解的问题中,往往把输入分成与原问题类型相同的两个子问题,即,取 k = 2。为了能清晰地反映出使用分治策略设计实际算法的基本步骤,下面用一个称之为抽象化控制的过程对算法的控制流向作出非形式的描述。基于以上目的,此过程中的基本运算由一些没定义其具体含义的其它过程来表示。在过程中还使用了一个全程量数组 A(1：n),用它来存放(或指示)这 n 个输入。这个过程 DANDC 是函数,它最初由 DANDC(1,n)所调用。DANDC(p,q)解算输入为 A(p：q)情况下的问题。

算法 4.1　分治策略的抽象化控制

```
procedure DANDC(p,q)
  global n, A(1：n); integer m,p,q;    //1≤p≤q≤n//
  if SMALL(p,q)
     then return(G(p,q))
     else m←DIVIDE(p,q)    //p≤m<q//
        return(COMBINE(DANDC(p,m),DANDC(m+1,q)))
  endif
end DANDC
```

SMALL(p,q)是一个布尔值函数,它可判断输入规模 q - p + 1 是否小到无需进一步细分就能算出其答案的程度。若是,则调用能直接计算此规模下的子问题解的函数 G(p,q);若非,则调用分割函数 DIVIDE(p,q),返回一个表示分割在何处进行的整数。然后,将这个整数值存入变量 m。于是,原问题被分成输入为 A(p：m)和 A(m+1,q)的两个子问题。对这两个子问题分别递归调用 DANDC 得到它们各自的解 x 和 y,再用一个合并函数

COMBINE(x,y)将这两个子问题的解合成原问题(输入为 A(p,q))的解。倘若所分成的两个子问题的输入规模大致相等,则 DANDC 总的计算时间可用下面的递归关系来表示:

$$T(n)=\begin{cases} g(n) & n\text{ 足够小} \\ 2T(n/2)+f(n) & \text{否则} \end{cases}$$

其中,T(n)是输入规模为 n 的 DANDC 时间,g(n)是对足够小的输入规模能直接计算出答案的时间,f(n)是 COMBINE 的时间。

虽然以分治法为基础,将要解算的问题分成与原问题类型相同的子问题来求解的算法用递归过程描述是很自然的,但为了提高效率,则往往需要将这一递归形式转换成迭代形式。算法 4.2 就是对算法 4.1 运用 2.5 节的转换规则并经过进一步化简后的形式。

算法 4.2 分治法抽象化控制的迭代形式

```
procedure DANDC1(p,q)
    //DANDC 的迭代模型。说明一个适当大小的栈//
    local s,t
    top←0                                          //置栈为空//
    L1:while not SMALL(p,q) do
        m←DIVIDE(p,q)                              //确定如何分割这输入//
        p,q,m,0,2  进 STACK 栈                     //处理第一次递归调用//
        q←m
    repeat
    t←G(p,q)
    while top≠0 do
        p,q,m,s,ret 从 STACK 栈退出
        if ret=2
            then p,q,m,t,3 进 STACK 栈             //处理第二次递归调用//
            p←m+1
            go to L1
        else t←COMBINE(s,t)                        //将两个子解合并成一个解//
    endif
  repeat
  return(t)
end DANDC1
```

4.2 二分检索

已知一个按非降次序排列的元素表 $a_1,a_2,\cdots,a_n$,要求判定某给定元素 x 是否在该表中出现。若是,则找出 x 在表中的位置,并将此下标值赋给变量 j;若非,则将 j 置成 0。这个检索问题就可以使用分治法来求解。设该问题用 $I=(n,a_1,\cdots,a_n,x)$ 来表示,将它分解成一些子问题,一种可能的做法是,选取一个下标 k,由此得到 3 个子问题:$I_1=(k-1,a_1,\cdots,a_{k-1},x)$,$I_2=(1,a_k,x)$ 和 $I_3=(n-k,a_{k+1},\cdots,a_n,x)$。对于 I_2,通过比较 x 和 a_k 容易得到解决。如果 $x=a_k$,则 j=k 且不需再对 I_1 和 I_3 求解;否则,在 I_2 子问题中的 j=0,此时,若 $x<a_k$,则只

有 I_1 留待求解,在 I_3 子问题中的 j = 0。若 $x > a_k$,只有 I_3 留待求解,在 I_1 子问题中的 j = 0。在与 a_k 作了比较之后,留待求解的问题(如果有的话)可以再一次使用分治方法来求解。如果求解的问题(或子问题)所选的下标 k 都是其中间元素的下标(例如,对于 I, $k = \lfloor (n+1)/2 \rfloor$),则所产生的算法就是通常所说的二分检索。

4.2.1　二分检索算法

算法 4.3 用 SPARKS 语言描述了这个二分检索方法。过程 BINSRCH 有 n + 2 个输入:A,n 和 x,一个输出 j。只要还有待检查的元素,while 循环就继续下去。case 语句是对 3 种情况的选择,如果前两个条件不为真,则自动执行"else 子句"。过程结束时,如果 x 不在表中,则 j = 0,否则 A(j) = x。

算法 4.3　二分检索

```
procedure BINSRCH(A,n,x,j)
// 给定一个按非降次序排列的元素数组 A(1:n),n≥1,判断 x 是否出现。若是,置 j,使得 x = A(j),若非,j = 0 //
  integer low,high,mid,j,n;
  low←1;high←n
  while low≤high do
        mid←⌊(low + high)/2⌋
    case
      :x < A(mid):high←mid - 1
      :x > A(mid):low←mid + 1
      :else:j←mid;return
    endcase
  repeat
 j←0
end BINSRCH
```

判断 BINSRCH 是否为一个算法,除了上段所述之外,还必须使 x 和 A(mid)的比较有恰当的定义。如果 A 的元素是整数、实数或字符串,则这些比较运算都可用适当的指令正确完成。另外,还需判断 BINSRCH 是否终止。关于这一点留待证明算法正确性时回答。

在对算法的正确性作出证明以前,为了增加对此算法的置信度,不妨用一个具体的例子来模拟算法的执行。

例 4.1　假定在 A(1:9)中顺序存放着以下 9 个元素: -15, -6,0,7,9,23,54,82,101。要求检索下列 x 的值:101, -14 和 82 是否在 A 中出现。

这是两次成功和一次不成功的检索。在模拟算法执行时只需追踪变量 low, high 和 mid,其追踪轨迹由表 4.1 列出。

关于程序正确性的证明至今还是一个尚未解决的课题。在这里仅给出 BINSRCH 正确性的一种"非形式证明"。

定理 4.1　过程 BINSRCH(A,n,x,j)能正确地运行。

证明　假定 x > A(mid)之类的比较运算能恰当地被执行,且过程中所有语句都能按所

表 4 .1 例 4 .1 的实际运行轨迹

x = 101			x = - 14			x = 82		
low	high	mid	low	high	mid	low	high	mid
1	9	5	1	9	5	1	9	5
6	9	7	1	4	2	6	9	7
8	9	8	1	1	1			
9	9	9	2	1	找不到	8	9	8
		找到						找到

要求的那样工作。最初,low = 1,high = n,n≥0,且 A(1)≤…≤A(n)。如果 n = 0,则不进入 while 循环,j 被置成 0,算法终止。否则,就会进入 while 循环去查找 x 是否是 A 中的元素,对于每一次循环,有可能被检查比较的元素是 A(low),A(low + 1),…,A(mid),…,A(high)。如果 x = A(mid),则将 mid 的值送 j,算法成功地终止。否则,若 x < A(mid),则 x 根本不可能在A(mid)到 A(high)中出现,因此可将查找范围缩小到 A(low)和 A(mid - 1)之间而不会影响检索结果。缩小检索范围的工作由 high←mid - 1 语句完成。同样,若 x > A(mid),则通过 low←mid + 1 把 low 增加到 mid + 1 来缩小检索范围且不会影响检索结果。又因为 low 和 high 都是整型变量,按上述方式缩小检索区总可在有限步内使 low 变得比 high 大。如果出现这种情况,则说明 x 不在 A 中,退出循环,j 被置 0,算法终止。证毕。

BINSRCH 需要的空间是很容易确定的,它要用 n 个位置存放数组 A,还要有存放变量 low,high,mid,x 和 j 的 5 个空间位置。因此,所需的空间位置是 n + 5。至于它的计算时间,则要分别考虑最好、平均和最坏 3 种情况。为了清楚起见,对于检索问题还需将最好情况区分为成功检索的最好情况和不成功检索的最好情况来加以分析。对于平均和最坏情况的分析也作类似的处理。显然,x 只有在取 A 中任一元素的情况下,才会出现成功的检索,所以成功的检索一共有 n 种,而为了测试所有不成功的检索,只需将 x 取 n + 1 个不同的值。因此,在算出 BINSRCH 在 x 这 2n + 1 种取值情况下的执行时间之后,求取它在最坏、平均和最好情况的计算时间就不难了。

在对算法的一般情况分析以前,不妨先对例 4 .1 的实例作出分析,看看算法在频率计数上有些什么特性。从算法中可以看到,所有的运算基本上都是在进行比较和数据传送。前两条和最末一条是赋值语句,频率计数均为 1。在 while 循环中,我们集中考虑 x 和 A 中的元素比较,而其它运算的频率计数显然与这些元素比较运算的频率计数具有相同的数量级。假定只需一次比较就可确定 case 语句控制是的 3 种情况的哪一种,进而找这 9 个元素中的每一个所需的元素比较次数:

A	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
元素	- 15	- 6	0	7	9	23	54	82	101
比较次数	3	2	3	4	1	3	2	3	4

要找到一个元素至少要进行 1 次比较,至多要进行 4 次比较。对找到的 9 项比较次数取平均值,即可得到每一次成功检索的平均比较次数为 25/ 9≈2 .77。不成功检索的终止方式取决于 x 的值,总共有 9 + 1 = 10 种可能的方式。如果 x < A(1),A(1) < x < A(2),A(2) < x < A(3),A(5) < x < A(6),A(6) < x < A(7)或 A(7) < x < A(8),为了确定 x 不在 A 中出

现,算法要作 3 次元素比较,而对 x 取值的其余情况则要作 4 次元素比较。这样,一次不成功检索的元素平均比较次数就是(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4)/ 10 = 34/ 10 = 3 .4。

在 x 的所有可能取值(2n + 1)中,不难发现,算法的每一执行过程都与一系列的 mid 值有关,即算法的执行过程实质上是 x 与一系列中间元素 A(mid)的比较过程。用一棵二元树来描述算法所有可能的执行过程是清楚的。这种用来描述算法执行过程的二元树称为二元比较树。比较树的结点由称为内结点和外结点的两种结点组成。每个内结点表示一次元素比较,它用圆形结点来表示。每一条路径表示一个元素比较序列。在以元素比较为基础的二分检索算法中,每个内结点存放一个 mid 值。外结点用一个方形结点表示,在二分检索算法中它表示不成功检索的一种情况。这样,x 如果在 A 中出现,算法就在一个圆形结点处结束,这圆形结点就指出 x 在 A 中被找到处的元素的下标。x 如果不在 A 中出现,则算法在一个方形结点处终止。图 4 .1 就是 n = 9 情况下的二元比较树,它刻画了在 x 的各种取值下 BINSRCH 所产生的比较过程。

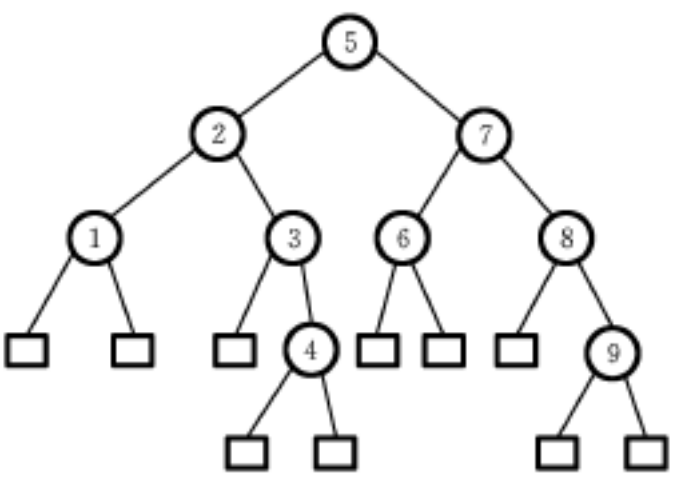


图 4 .1　n = 9 情况下二分检索的二元比较树

仍以 A(1∶9)取例 4 .1 中的数据集为例,若取 x = 23,则在执行 BINSRCH 时,首先将 x 与A(5) = 9比较,若 x > A(5),下一次则与其右结点 7 所示元素比较;若 x < A(7) = 54,则下次与 7 的左结点 6 所示元素比较;若 x = A(6) = 23,则 x 在 A 中找到,算法成功终止。

借助于二元比较树,算法 BINSRCH 的计算时间可由定理 4 .2 给出。

定理 4 .2　若 n 在区域$[2^{k-1}, 2^k)$中,则对于一次成功的检索,BINSRCH 至多作 k 次比较而对于一次不成功的检索,或者作 k - 1 次比较或者作 k 次比较。

证明　考察以 n 个结点描述 BINSRCH 执行过程的二元比较树,所有成功检索都在内部结点处结束,而所有不成功检索都在外部结点处结束。由于 $2^{k-1} \leq n < 2^k$,因此,所有的内部结点在 1,2,…,k 级,而所有的外部结点在 k 和 k + 1 级(注意,根在 1 级)。就是说,成功检索在 i 级终止所需要的元素比较数是 i,而不成功检索在 i 级外部结点终止的元素比较数是 i - 1。定理得证。

定理 4 .2 说明,最坏情况下的成功检索和不成功检索的计算时间都是 Θ(logn)。最好情况下的成功检索在 1 级结点处达到,计算时间为 Θ(1)。最好情况下的不成功检索要作 ⌊logn⌋ 次元素比较,所以计算时间是 Θ(logn)。由于外部结点都在 k 和 k + 1 级,故每种不成功检索的时间都为 Θ(logn)。因此,平均情况下不成功检索的计算时间为 Θ(logn),记为 U(n)。下面利用外部结点与内部结点到根的距离和之间的一种简单关系,由不成功检索的平均比较数求出成功检索的平均比较数。为讨论方便起见,定义两个术语:由根到所有内部结点的距离之和称为内部路径长度 I;由根到所有外部结点的距离之和称为外部路径长度 E。容易证明 I 和 E 之间有如下关系:

$$E = I + 2n$$

令 S(n)是成功检索的平均比较数。为了找一个内部结点表示的元素所需的比较数是由根到该结点的路径长度(即距离)加 1。因此,

$$S(n) = \sqrt{n} + 1$$

而到任何一个外部结点所需要的比较数是由根到该结点路径的长度,由此可得

$$U(n) = E/(n+1)$$

利用以上公式即可推得

$$S(n) = (1 + 1/n)U(n) - 1$$

由上式可知 S(n)与 U(n)是直接相关的,而 $U(n) = \Theta(\log n)$,所以成功检索的计算时间 S(n)也为 $\Theta(\log n)$。

综合所述,二分检索在各种情况下的计算时间是

成功的检索			不成功的检索
$\Theta(1)$, $\Theta(\log n)$, $\Theta(\log n)$			$\Theta(\log n)$
最好	平均	最坏	最好、平均和最坏

BINSRCH 的 case 语句可用 FORTRAN 语言的算术-if 语句来实现。而在 PL/1 或 PASCAL 之类的语言中,它可以用与下述语句等价的代码来实现:

```
if x < A(mid) then high←mid - 1
              else if x > A(mid) then low←mid + 1
                                 else j←—mid; return
                   endif                                    (*)
endif
```

在这种情况下,有时要作两次元素比较。下面介绍一种二分检索的有趣变型 BINSRCH1。在 BINSRCH1 的 while 循环中 x 和 A(mid)之间只用作一次比较。

算法 4.4　每次循环作一次比较的二分检索

```
procedure BINSRCH1 (A, n, x, j)
    // 除 n > 0 外,其余说明与 BINSRCH 同 //
    integer low, high, mid, j, n;
    low←1; high←n + 1    // high 总比可能的取值大 1 //
    while low < high - 1 do
       mid←⌊(low + high)/2⌋
       if x < A(mid)    // 在循环中只比较一次 //
          then high←mid
          else low←mid    // x≥A(mid) //
       endif
    repeat
    if x = A(low) then j←low    // x 出现 //
                  else j←0    // x 不出现 //
    endif
end BINSRCH1
```

由 BINSRCH 和 BINSRCH1 相比较可以看出,如果 BINSRCH 用(*)中语句实现 case,那么,在有些情况下(例如,当 x > A(n)时),它的元素比较数就是 BINSRCH1 的两倍。然而,对于任何一种成功的检索(例如,当 x = A(mid)时),BINSRCH1 平均要比 BINSRCH 多作$(\log n)/2$次比较。BINSRCH1 的最好、平均和最坏情况时间对于成功和不成功的检索

都是 Θ(logn)。此结论的证明留作习题。

4.2.2　以比较为基础检索的时间下界

对于在 n 个已分类元素序列上(即已按非降次序或非增次序排列的 n 个元素序列),在检索某元素是否出现问题时,能否预计还存在有以元素比较为基础的另外的检索算法,它在最坏情况下比二分检索算法在计算时间上有更低的数量级呢?下面就来讨论这个问题。

假设 n 个元素 A(1∶n)有关系 A(1) < A(2) < … < A(n),要检索一给定元素 x 是否在 A 中出现。如果只允许进行元素间的比较而不允许对它们实施运算,则在这种条件下所设计的算法都称为是以比较为基础的算法。根据二元比较树的定义,显然任何以比较为基础的检索算法在执行过程中都可以用二元比较树来描述。每个内结点表示一次元素比较,因此任何比较树中必须含有 n 个内结点,且分别与 n 个不同的 i 值相对应。每个外结点对应于一次不成功的检索。图 4.2 给出了两棵比较树,一个是模拟线性检索,一个是模拟二分检索。

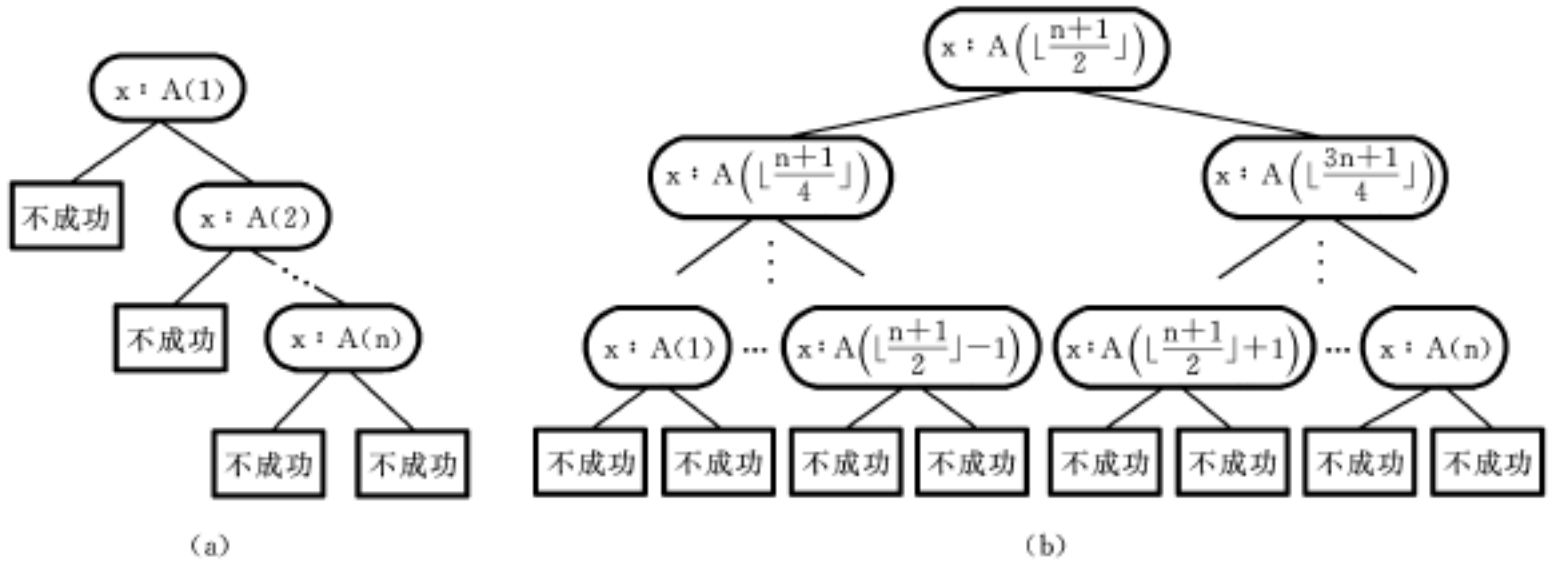


图 4.2　两个检索算法的比较树

(a) 模拟线性检索;(b) 模拟二分检索

下面的定理给出了以比较为基础的有序检索问题在最坏情况下的时间下界。

定理 4.3　设 A(1∶n)含有 n(n≥1)个不同的元素,排序为 A(1) < … < A(n);又设用以比较为基础去判断是否 x∈A(1∶n)的任何算法在最坏情况下所需的最小比较次数是 FIND(n),那么 $\mathrm{FIND}(n) \geq \lceil \log(n+1) \rceil$。

证明　通过考察模拟求解检索问题的各种可能算法的比较树可知,FIND(n)不大于树中由根到一个叶子的最长路径的距离。在这所有的树中都必定有 n 个内结点与 x 在 A 中可能的 n 种出现情况相对应。如果一棵二元树的所有内结点所在的级数小于或等于级数 k,则最多有 $2^k - 1$ 个内结点。因此,$n \leq 2^k - 1$,即 $\mathrm{FIND}(n) = k \geq \lceil \log(n+1) \rceil$。证毕。

由定理 4.3 可知,任何一种以比较为基础的算法,其最坏情况时间都不可能低于O(logn),因此,也就不可能存在其最坏情况时间比二分检索数量级还低的算法。事实上,二分检索所产生的比较树的所有外部结点都在相邻接的两个级上,而且不难证明这样的二元树使得由根到结点的最长路径减至最小,因此,二分检索是解决检索问题的最优的最坏情况算法。

4.3 找最大和最小元素

如果要在含有 n 个不同元素的集合中同时找出它的最大和最小元素,最简单的方法是将元素逐个进行比较。这种直接算法可初步描述如下。

算法 4.5 直接找最大和最小元素

```
procedure STRAITMAXMIN(A,n,max,min)
    //将 A(1:n)中的最大元素置于 max,最小元素置于 min//
    integer i,n;
    max←min←A(1)
    for i←2 to n do
      if A(i) > max
         then max←A(i) endif
      if A(i) < min
         then min←A(i) endif
    repeat
end STRAITMAXMIN
```

当要分析这个算法的时间复杂度时,只需将元素比较次数求出即可。这不仅是因为算法中的其它运算与元素比较有相同数量级的频率计数,而且更重要的是,当 A(1:n)中的元素是多项式、向量、非常大的数或字符串时,一次元素比较所用的时间比其它运算的时间多得多。

容易看出过程 STRAITMAXMIN 在最好、平均和最坏情况下均需要作 2(n-1)次元素比较。另外,只要稍许考察一下算法 4.5 就可发现,只有当 A(i) > max 为假时,才有必要比较A(i) < min,因此可用下面的语句代替 for 循环中的语句:

```
if A(i) > max then max←A(i)
        else if A(i) < min then min←A(i) endif
endif
```

在作出以上改进后,最好情况将在元素按递增次序排列时出现,元素比较数是 n-1;最坏情况将在递减次序时出现,元素比较数是 2(n-1);至于在平均情况下,A(i)将有一半的时间比 max 大,因此平均比较数是$\frac{3}{2}(n-1)$。

能否找到更好的算法呢?下面用分治策略的思想来设计一个算法与直接算法作比较。使用分治策略设计是将任一实例 I=(n,A(1),…,A(n))分成一些较小的实例来处理,例如,可以把 I 分成两个这样的实例:I1=($\lfloor n/2 \rfloor$,A(1),…,A($\lfloor n/2 \rfloor$))和 I2=(n-$\lfloor n/2 \rfloor$,A($\lfloor n/2 \rfloor$+1),…,A(n))。如果 MAX(I)和 MIN(I)是 I 中的最大和最小元素,则 MAX(I)等于 MAX(I1)和 MAX(I2)中的大者,MIN(I)等于 MIN(I1)和 MIN(I2)中的小者。如果 I 只包含一个元素,则不需要作任何分割直接就可得到其解。

算法 4.6 是以上方法所导出的过程。它是在元素集合{A(i),A(i+1),…,A(j)}中找最大和最小元素的递归过程。过程对于集合含有一个元素(i=j)和两个元素(i=j-1)的情况

分别作出处理,而对含有多于两个元素的集合,则确定其中点(正如在二分检索中那样),并且产生两个新的子问题。当分别找到这两个子问题的最大和最小值后,再比较这两个最大值和两个最小值而得到此全集合的解。max 和 min 被看成是两个内部函数,它们分别求取两个元素的大者和小者,并认为每次调用其中的一个函数都只需作一次元素比较。

算法 4.6　递归求取最大和最小元素

```
procedure MAXMIN(i,j,fmax,fmin)
    // A(1:n)是含有 n 个元素的数组,参数 i,j 是整数,1≤i,j≤n //
    // 该过程把 A(i,j)中的最大和最小元素分别赋给 fmax 和 fmin //
  integer i,j;global n,A(1:n)
  case
    :i=j:fmax←fmin←A(i)
    :i=j-1:if A(i)<A(j) then fmax←A(j);fmin←A(i)
                        else fmax←A(i);fmin←A(j)
           endif
    :else:mid←⌊(i+j)/2⌋
          call MAXMIN(i,mid,gmax,gmin)
          call MAXMIN(mid+1,j,hmax,hmin)
          fmax←max(gmax,hmax)
          fmin←min(gmin,hmin)
  endcase
end MAXMIN
```

这个过程最初由 call MAXMIN(1,n,x,y)所调用。为加深对这个过程的理解,下面来看一个例子。

例 4.2　在下述 9 个元素上模拟过程 MAXMIN

A:	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
	22	13	-5	-8	15	60	17	31	47

用一棵树来描述递归调用的轨迹是很清晰的,其中的一个结点表示一次递归调用,每作一次新的调用就增加一个结点。对于这个程序来说,每个结点有 4 项信息:i,j,fmax,fmin。根据上面的数组 A,就可作出图 4.3 所示的树。图中,根结点内的 1 和 9 是最初调用 MAXMIN 的 i,j 值。这次执行产生对 MAXMIN 的两次新的调用,其中 i 和 j 分别有值 1,5 和 6,9,这样就把集合分成了大小基本相同的两个子集。由这棵树可以直接看到递归的最大深度是 4(包括第一次调用),每个结点左上方圆圈中的数表示给 fmax 和 fmin 赋值的次序。

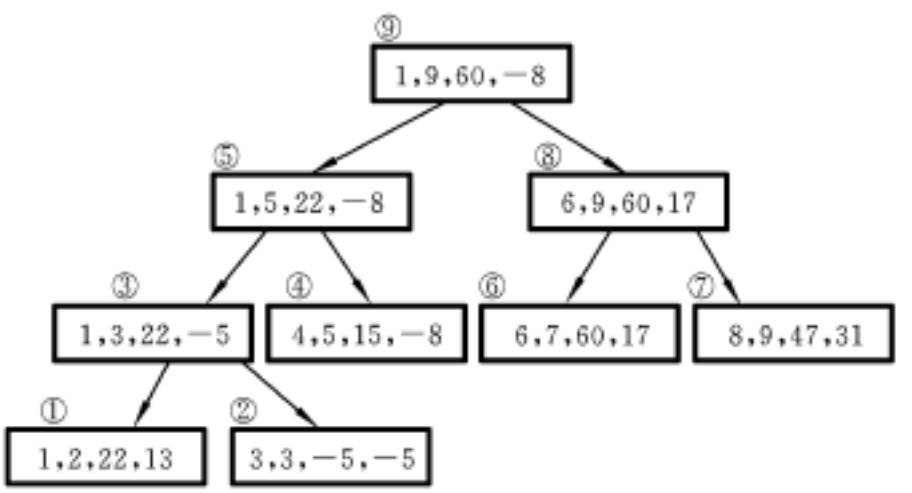


图 4.3　表示 MAXMIN 递归调用的树

MAXMIN 需要的元素比较数是多少呢?如果用 T(n)表示这个数,则所导出的递归关

系式是

$$T(n)=\begin{cases}0 & n=1\\ 1 & n=2\\ T(\lfloor n/2\rfloor)+T(\lceil n/2\rceil)+2 & n>2\end{cases}$$

当 n 是 2 的幂时,即对于某个正整数 k,$n=2^k$,有

$$\begin{aligned}T(n)&=2T(n/2)+2\\&=2(2T(n/4)+2)+2\\&=4T(n/4)+4+2\\&\cdots\\&=2^{k-1}T(2)+\sum_{1\le i\le k-1}2^i\\&=2^{k-1}+2^k-2\\&=3n/2-2\end{aligned}$$

注意:当 n 是 2 的幂时,$3n/2-2$ 是最好、平均及最坏情况的比较。此数和直接算法的比较数 $2n-2$ 相比,它少了 25%。

可以证明,任何一种以元素比较为基础的找最大和最小元素的算法,其元素比较下界均为 $\lceil 3n/2\rceil-2$ 次。因此,过程 MAXMIN 在这种意义上是最优的。那么,这是否意味着此算法确实比较好呢?不一定。其原因有如下两个。

一是 MAXMIN 要求的存储空间比直接算法多。给出 n 个元素就有 $\lfloor \log n\rfloor+1$ 级的递归,而每次递归调用需要保留到栈中的有 i,j,fmax,fmin 和返回地址五个值。虽然可用第 1 章递归化迭代的规则去掉递归,但所导出的迭代模型还需要一个其深度为 logn 数量级的栈。

二是当元素 A(i)和 A(j)的比较时间与 i 和 j 的比较时间相差不大时,过程 MAXMIN 并不可取。为说明问题,假设元素比较与 i 和 j 间的比较时间相同,又设 MAXMIN 的频率计数为 C(n),$n=2^k$,k 是某个正整数,并且对 case 语句的前两种情况用 $i\ge j-1$ 来代替 $i=j$ 和 $i=j-1$ 这两次比较,这样,只用对 i 和 $j-1$ 作一次比较就足以实现被修改过的这条 case 语句。于是 MAXMIN 的频率计数

$$C(n)=\begin{cases}2 & n=2\\ 2C(n/2)+3 & n>2\end{cases}$$

解此关系式可得

$$\begin{aligned}C(n)&=2C(n/2)+3=4C(n/4)+6+3\\&\cdots\cdots\\&=2^{k-1}C(2)+3\sum_{0\le i\le k-2}2^i=2^k+3\times 2^{k-1}-3\\&=5n/2-3。\end{aligned}$$

而 STRAITMAXMIN 的比较数是 3(n-1)(包括实现 for 循环所要的比较)。尽管它比 $5n/2-3$ 大些,但由于递归算法中 i,j,fmax,fmin 进出栈所带来的开销,因此 MAXMIN 在这种情况下反而比 STRAITMAXMIN 还要慢些。

根据以上分析可以得出结论:如果 A 的元素间的比较远比整型变量的比较代价昂贵,则分治方法产生效率较高(实际上是最优)的算法;反之,就得到一个效率较低的程序。因此,

分治策略只能看成是一个较好的然而并不总是能成功的算法设计指导。

4.4　归并分类

4.4.1　基本方法

给定一个含有 n 个元素(又叫关键字)的集合,如果要把它们按一定的次序分类(本节中自始至终假定按非降次序分类),最直接的方法就是插入法。对 A(1：n)中元素作插入分类的基本思想是:

```
for j←2 to n do
    将 A(j)放到已分类集合 A(1：j - 1)的正确位置上
repeat
```

从中可以看出,为了插入 A(j),有可能移动 A(1：j - 1)中的所有元素,因此可以预计该算法在时间特性上不会太好,算法具体描述如下:

算法 4.7　插入分类

```
procedure INSERTIONSORT (A,n)
    // 将 A(1：n)中的元素按非降次序分类,n≥1 //
    A(0)← - ∞    // 开始时生成一个虚拟值 //
    for j←2 to n do    // A(1：j - 1)已分类 //
      item←A(j);i←j - 1
      while item < A(i) do    // 0≤i < j //
        A (i + 1)←A(i);i←i - 1
          repeat
          A(i + 1)←item
        repeat
end INSERTIONSORT
```

while 循环中的语句可能执行 0～j 次(j = 2,3,…,n),因此,这过程的最坏情况限界是

$$\sum_{2\le j\le n} j = (n(n+1)/2) - 1 = \Theta(n^2)$$

如果输入数据本来就是按非降次序排列的,根本不会进入 while 的循环体,就是最好情况,计算时间是 Ω(n)。

如果用分治策略来设计分类算法,则可使最坏情况时间变为 O(n logn)。这样的算法称为归并分类算法。算法的基本思想是,将 A(1),…,A(n)分成两个集合 A(1),…,A(⌊n/2⌋)和 A(⌊n/2⌋ + 1),…,A(n)。对每个集合单独分类,然后将已分类的两个序列归并成一个含 n 个元素的分好类的序列。这种思想是典型的分治设计思想。过程 MERGESORT 通过使用递归和调用把两个已分类集合归并在一起的子过程 MERGE 非常简洁地刻画了这一处理过程。

算法 4.8　归并分类

```
procedure MERGESORT(low,high)
    // A(low：high)是一个全程数组,它含有 high - low + 1≥0 个待分类的元素 //
```

```
    integer low,high;
    if low < high;
      then mid←⌊(low + high)/ 2⌋   // 求这个集合的分割点 //
           call MERGESORT(low,mid)   // 将一个子集合分类 //
           call MERGESORT(mid + 1,high)   // 将另一个子集合分类 //
           call MERGE(low,mid,high)   // 归并两个已分类的子集合 //
    endif
end MERGESORT
```

算法 4.9　使用辅助数组归并两个已分类的集合

```
procedure MERGE(low,mid,high)
    // A(low：high)是一个全程数组,它含有两个放在 A(low：mid)和 A(mid + 1：high)中的
       已分类的子集合。目标是将这两个已分类的集合归并成一个集合,并存放到 A(low：
       high)中。
    // 使用了一个辅助数组 B(low：high) //
    integer h,i,j,k,low,mid,high;   // low≤mid < high //
    global A(low：high);local B(low：high)
    h←low;i←low;j←mid + 1;
    while h≤mid and j≤high do   // 当两个集合都没取尽时 //
       if A(h)≤A(j) then B(i)←A(h);h←h + 1
                    else B(i)←A(j);j←j + 1
       endif
       i←i + 1
    repeat
    if h > mid then for k←j to high do   // 处理剩余的元素 //
                        B(i)←A(k);i←i + 1
                        repeat
                else for k←h to mid do
                        B(i)←A(k);i←i + 1
                     repeat
       endif
    for k←low to high do      // 将已归并的集合复制到 A //
       A(k)←B(k)
    repeat
end MERGE
```

在过程 MERGESORT 开始以前,这 n 个元素应放在 A(1：n)中。使用 call MERGESORT(1,n)语句将使关键字重新排列成非降序列而存于 A 中。

例 4.3　使用归并分类算法,将含有 10 个元素的数组 A = (310,285,179,652,351,423,861,254,450,520)按非降次序分类。

过程 MERGESORT 首先把 A 分成两个各有 5 个元素的子集合,然后把 A(1：5)分成大小为 3 和 2 的两个子集合,再把 A(1：3)分成大小为 2 和 1 的子集合,最后将 A(1：2)分成各含一个元素的两个子集合,至此就开始归并。此时的状态可排成下列形式:

(310|285|179|652,351|423,861,254,450,520)

其中,直杠表示子集合的边界线。归并 A(1)和 A(2)得

(285,310|179|652,351|423,861,254,450,520)

再归并 A(1：2)和 A(3)得

(179,285,310|652,351|423,861,254,450,520)

然后将 A(4：5)分成两个各含一个元素的子集合,再将两个子集合归并得

(179,285,310|351,652|423,861,254,450,520)

接着归并 A(1：3)和 A(4：5)得

(179,285,310,351,652|423,861,254,450,520)

此时算法就返回到 MERGESORT 首次递归调用的后继语句的开始处,即准备执行第二条递归调用语句。又通过反复地递归调用和归并,将 A(6：10)分好类,其结果如下:

(179,285,310,351,652|254,423,450,520,861)

这时就有了两个各含 5 个元素的已分好类的子集合。经过最后的归并得到分好类的结果:

(179,254,285,310,351,423,450,520,652,861)

图 4 .4 所示的是在 n = 10 的情况下,由 MERGESORT 所产生的对它自己进行一系列递归调用的树表示。每个结点中的一对值都是参变量 low 和 high 的值。值得注意的是,集合的分割一直进行到产生出只含单个元素的子集合为止。图 4 .5 所示的则是一棵表示 MERGESORT 对过程 MERGE 调用的树。每个结点中的值依次是参变量 low, mid 和 high 的值。例如,含有值 1,2,3 的结点表示 A(1：2)和 A(3)中元素的归并。

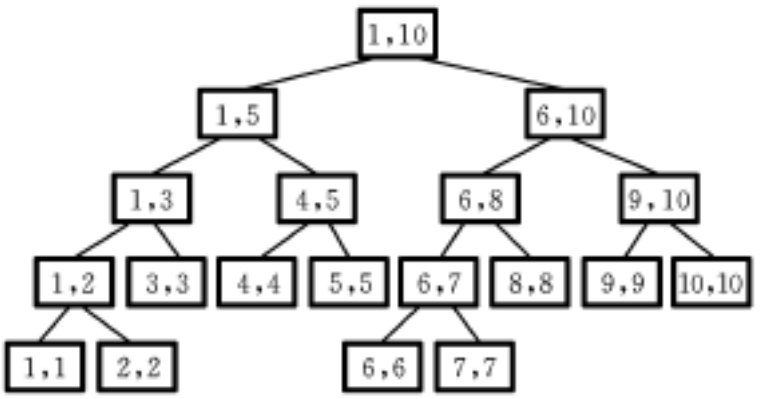


图 4 .4　用树表示 MERGESORT(1,10)的调用

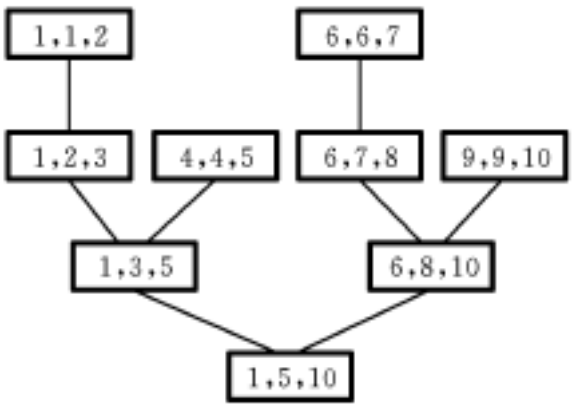


图 4 .5　用树表示对 MERGE 的调用

如果归并运算的时间与 n 成正比,则归并分类的计算时间可用递归关系式描述如下:

$$T(n)=\begin{cases} a & n=1,\ a\ \text{是常数} \\ 2T(n/2)+cn & n>1,\ c\ \text{是常数} \end{cases}$$

当 n 是 2 的幂即 $n=2^k$ 时,可以通过逐次代入求出其解:

$$\begin{aligned} T(n) &= 2(2T(n/4)+cn/2)+cn \\ &= 4T(n/4)+2cn \\ &= 4(2T(n/8)+cn/4)+2cn \\ &\cdots \\ &= 2^kT(1)+kcn \\ &= an+cn\log n \end{aligned}$$

如果 $2^k<n<2^{k+1}$,易于看出 $T(n)\le T(2^{k+1})$。因此,

$$T(n) = O(n \log n)$$

4.4.2 改进的归并分类算法

尽管算法 4.8 相当充分地反映了使用分治策略对数据对象分类的长处,但仍存在着一些明显的不足,从而限制了分类效率的进一步提高。下面逐一分析这些不足之处,并提出相应的改进措施,以期能给出一个更有效的归并分类模型。

在使用例 4.3 的数据集模拟执行 MERGESORT 的过程中,可以看到,每当集合被分成只含二个元素的子集合时,还需要使用二次递归调用将这子集合分成单个元素的集合。这表明该算法执行到将集合分成含元素相当少的子集合时,很多时间不是用在实际的分类而是消耗在处理递归上。如果不让递归一直进行到最低一级,就对这种情况作出改进。从抽象化控制过程来看,只要使 SMALL(p,q)在输入规模 q - p + 1 适当小时取真值,在这种情况下采用另一个能在小规模集合上有效工作的分类算法就能克服低层递归调用所带来消耗过大的问题。本节开始所给出的插入分类算法尽管时间复杂度为 $O(n^2)$,但当 n 很小(譬如小于 16)时却能极快地工作,因此将它作为归并分类算法中处理小规模集合情况的子过程是很适宜的。

另外,归并分类算法使用了辅助数组 B(1：n),这是一个明显的不足之处。但是,由于不可能在两个已分类集合的原来的位置上进行适当的归并,所以这 n 个位置的附加空间对于本算法是必需的。而且,算法还必须在这附加空间上竭力地工作,在每次调用 MERGE 时,把存放在 B(low：high)中的结果复制回 A(low：high)中去。不过,使用一个以整数表示的链接信息数组 LINK(1：n)来代换暂时存放元素的辅助数组 B 可以节省一些附加空间。这个链接数组在 1,2,…,n 的范围内取值。这些整数被看成是 A 的元素的指针,在分类表中它指向下一个元素所在的下标位置。因此,分类表就成了一个指针的序列,而分类表的归并则不必移动元素本身,只要改变相应的链值即可进行。用 0 表示表的结束。下面是 LINK 值的一个集合,它包含了两个已分类子集合的元素链接信息表。

LINK	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
	6	4	7	1	3	0	8	0

Q 和 R 分别表示两个表的起始处,这里 Q = 2, R = 5。Q = 2 的表是(2,4,1,6),R = 5 的表是(5,3,7,8)。这两个表分别描述了 A(1：8)中两个已分类子集合,它们有 $A(2) \leq A(4) \leq A(1) \leq A(6)$ 和 $A(5) \leq A(3) \leq A(7) \leq A(8)$。

下面是采取了以上两种改进措施的归并分类模型。

算法 4.10 使用了链接的归并分类模型

```
procedure MERGESORT1(low,high,p)
  // 利用辅助数组 LINK(low：high)将全程数组 A(low：high)按非降次序分类。LINK 中值表
     示按分类次序给出 A 下标的表,并把 p 置于指示这表的开始处 //
  global A(low:high),LINK(low:high)
  if high - low + 1 < 16
    then call INSERTIONSORT(A,LINK,low,high,p)
    else mid←⌊(low + high)/ 2⌋
```

```
        call MERGESORT1(low,mid,q)          // 返回 q 表 //
        call MERGESORT1(mid + 1,high,r)     // 返回 r 表 //
        call MERGE1(q,r,p)                  // 将表 q 和 r 归并成表 p //
    endif
end MERGESORT1
```

在初次调用 MERGESORT1 时,把要分类的元素(即关键字)先放到 A(1：n)中,并且将 LINK(1：n)置成 0。初次调用语句为 call MERGESORT1(1, n, p),p 作为按分类次序给出 A 中元素的指示表的指针返回。这里的 INSERTIONSORT 是算法 4 .7 的改型,它把对 A(low：high)的分类表变成起始点在 p 的链接信息表。每当被分类的项数小于 16 时就调用它。

修改过的归并过程如下:

算法 4 .11　使用链接表归并已分类的集合

```
procedure MERGE1(q,r,p)
    // q 和 r 是全程数组 LINK(1：n)中两个表的指针。这两个表可用来获得全程数组A(1：n)
    中已分类元素的子集合。此算法执行后构造出一个由 p 所指示的新表,利用此表可以得到
    按非降次序把 A 中元素分好类的元素表,同时 q 和 r 所指示的表随之消失。假定LINK(0)
    被定义,且假定表由 0 所结束 //
    global n,A(1：n),LINK(0：n)
    local integer i,j,k
    i←q;j←r;k←0    // 新表在 LINK(0)处开始 //
    while i≠0 and j≠0 do    // 当两表皆非空时作 //
        if A(i)≤A(j)    // 找较小的关键字 //
            then LINK(k)←i;k←i;i←LINK(i)    // 加一个新关键字到此表 //
            else LINK(k)←j;k←j;j←LINK(j)
        endif
    repeat
    if i=0 then LINK(k)←j
           else LINK(k)←i
    endif
    p←LINK(0)
end MERGE1
```

为了帮助理解这个新的归并分类模型,下面来看一个例子。

例 4 .4　假设要对下述 8 个元素序列(50,10,25,30,15,70,35,55)分类。要求使用新算法模拟执行。这里略去在少于 16 个元素时应使用 INSERTIONSORT 分类的要求。LINK 数组初始化为 0。表 4 .2 显示了在每一次调用 MERGESORT1 结束后 LINK 数组的变化情况。各行上的 p 值分别指向最近一次 MERGE1 结束时所产生的 LINK 中的那个表。右边是这些表所表示的相应的已分类元素子集合。例如,在最后一行,p = 2 表示链表(2,5,3,4,7,1,8,6)在 2 处开始,而此链表意味着 A(2)≤A(5)≤A(3)≤A(4)≤A(7)≤A(1)≤A(8)≤A(6)。

表 4.2 例 4.4 中 LINK 数组的变化过程

	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
A	-	50	10	25	30	15	70	35	55
LINK	0	0	0	0	0	0	0	0	0
q r p									
1 2 2	2	0	1	0	0	0	0	0	0(10,50)
3 4 3	3	0	1	4	0	0	0	0	0(10,50),(25,30)
2 3 2	2	0	3	4	1	0	0	0	0(10,25,30,50)
5 6 5	5	0	3	4	1	6	0	0	0(10,25,30,50),(15,70)
7 8 7	7	0	3	4	1	6	0	8	0(10,25,30,50),(15,70),(35,55)
5 7 5	5	0	3	4	1	7	0	8	6(10,25,30,50)(15,35,55,70)
2 5 2	2	8	5	4	7	3	0	1	6(10,15,25,30,35,50,55,70)

除了以上改进之外,还可采用由底向上的方式设计算法,从而取消对栈空间的需要。这留作习题。

4.4.3 以比较为基础分类的时间下界

尽管作了以上那些改进,但不难看出,所得到的新归并分类算法的时间复杂度在最坏情况下仍为 O(n logn)。事实上,任何以关键字比较为基础的分类算法,它的最坏情况的时间下界都为 Ω(n logn),因此,从数量级的角度上来看,归并分类算法是最坏情况的最优算法。

利用 4.2 节所描述的二元比较树,也可很容易给出以比较为基础的分类算法时间下界的证明。在这里,假设参加分类的 n 个关键字 A(1),…,A(n)各不相同,因此任意两个关键字 A(i)和 A(j)的比较必导致 A(i) < A(j)或者 A(i) > A(j)的结果。在描述算法各种可能执行的比较树中,每个内结点用比较对“i:j”来代表 A(i)和 A(j)的比较,当 A(i) < A(j)时进入左分枝,A(i) > A(j)时进入右分枝。各外部结点表示此算法的终止。从根到外结点的每一条路径分别与一种唯一的排列相对应。由于 n 个关键字有 n!种排列,而每种排列可以是某种特定输入下的分类结果,因此比较树必定至少有 n!个外结点,每个外结点表示一种可能的分类序列。图 4.6 给出了对 3 个关键字分类的一棵二元比较树,其边上的不等式表示此条件成立时控制的转向。

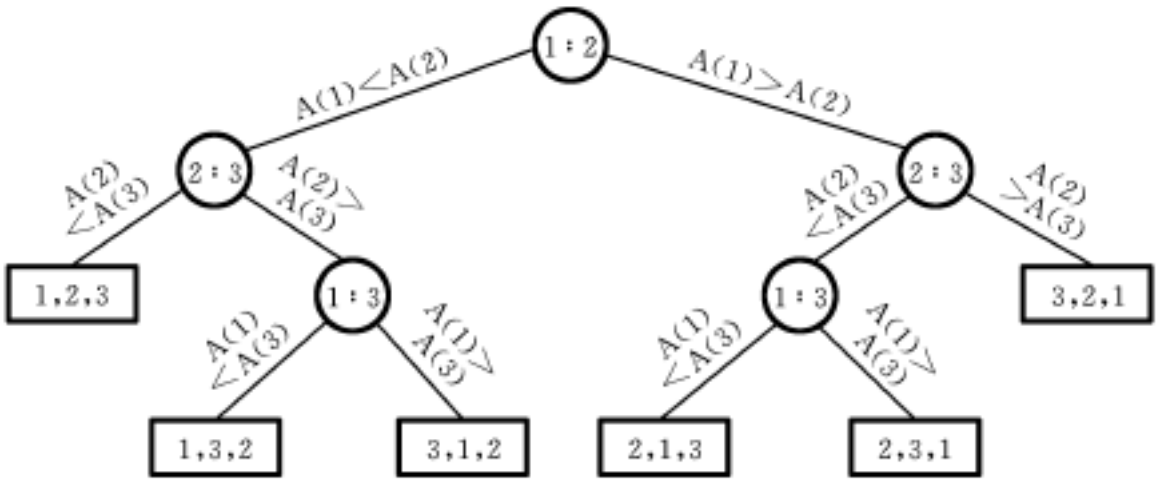


图 4.6 对 3 个关键字分类的比较树

对于任何一个以比较为基础的算法,在描述其执行的那棵比较树中,由根到某外结点的

路径长度表示生成该外结点中那个分类序列所需要的比较次数。因此,这棵树中最长路径的长度(即此树的高度)就是该算法在最坏情况下所作的比较次数。从而,要求出所有以比较为基础的对 n 个关键字分类的算法最坏情况下界,只需求出这些算法对应的比较树的最小高度,设它为 T(n)。如果一棵二元树的所有内结点的级数均小于或等于 k,对 k 施行归纳可以证得该树至多有 2^k 个外结点(比内结点数多 1)。令 T(n) = k,则

$$n! \leq 2^{T(n)}$$

而当 n > 1 时有　　$n! \geq n(n-1)(n-2)\cdots(\lceil n/2 \rceil) \geq (n/2)^{n/2}$

因此,对于 n≥4 有　　$T(n) \geq (n/2)\log(n/2) \geq (n/4)\log n$

故以比较为基础的分类算法的最坏情况的时间下界为 Ω(n logn)。

4.5　快速分类

4.5.1　快速分类算法

由著名的计算机科学家霍尔(C.A.R.Hoare)给出的快速分类算法也是根据分治策略设计的一种高效率的分类算法。它虽然也是把文件 A(1：n)分成两个子文件,但与归并分类算法有所不同:在被分成的两个子文件以后不再需要归并。于是,被分成的两个子文件至少必须满足一子文件中的所有元素都小于或等于另一子文件的任何一个元素。这是通过重新整理 A(1：n)中元素的排列顺序来达到的,其实现的基本思想如下:选取 A 的某个元素,譬如说 t = A(s),然后将其它元素重新排列,使 A(1：n)中所有在 t 以前出现的元素都小于或等于 t,而所有在 t 后面出现的元素都大于或等于 t。文件的这种重新整理叫做划分(partitioning),元素 t 称为划分元素(partition element)。因此,所谓快速分类就是通过反复对产生的文件进行划分来实现的。

过程 PARTITION 完成对文件 A(m：p - 1)的划分。在过程中,假定第一个元素 A(m)是划分元素(这种假定是非本质的,仅仅是为了方便而已,后面将会看到把其它元素选成划分元素比选第一项要好些的情况)。A(p)不属于文件 A(m：p - 1),且假定 A(p)≥A(m),引进 A(p)是为了在特殊情况下能控制程序顺利进行。因此,在对 PARTITION 初次调用,即 m = 1,p - 1 = n 时,则必须将 A(n + 1)定义成大于或等于 A(1：n)的所有元素。子过程 INTERCHANGE(x,y)执行以下赋值语句:temp←x; x←y; y←temp。

算法 4.12　用 A(m)划分集合 A(m：p - 1)

```
procedure PARTITION(m,p)
    // 在 A(m),A(m+1),…,A(p-1)中的元素按如下方式重新排列:如果最初 t = A(m),则在
       重排完成之后,对于 m 和 p-1 之间的某个 q,有 A(q) = t,并使得对于 m≤k<q,有 A(k)
       ≤t,而对于 q<k<p,有 A(k)≥t。在退出过程时,p 带着划分元素所在的下标位置,即 q
       的值返回 //
    integer m,p,i;global A(m：p-1)
    v←A(m);i←m    // A(m)是划分元素 //
```

```
        loop
          loop i←i + 1 until A(i)≥v repeat    // i 由左向右移//
          loop p←p - 1 until A(p)≤v repeat    // p 由右向左移//
          if i < p
              then call INTERCHANGE(A(i), A(p))    // A(i)和 A(p)换位//
              else exit
          endif
        repeat
        A(m)←A(p); A(p)←v    // 划分元素在位置 p//
    end PARTITION
```

为有助于理解 PARTITION 的工作情况,下面来看一个例子。

例 4.5　用过程 PARTITION 来划分下面含 9 个元素的数组。

这过程最初由 call PARTITION(1,10)所调用。用水平虚线连接的竖线指示为产生下一行而交换的两个元素。A(1) = 65 是划分元素,最终(在第六行)确定它是此数组中的第五小元素。要指出的是,在 PARTITION 执行完毕之后,除划分元素之外的所有元素相对于 A(5) = 65 也是被划分了的。

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
60	70	75	80	85	60	55	50	45	+ ∞	2	9
65	45	75	80	85	60	55	50	70	+ ∞	3	8
65	45	50	80	85	60	55	75	70	+ ∞	4	7
65	45	50	55	85	60	80	75	70	+ ∞	5	6
65	45	50	55	60	85	80	75	70	+ ∞	6	5
60	45	50	55	65	85	80	75	70	+ ∞		

有了划分集合 A(m：p - 1)的算法,使用分治策略马上就可设计出一个算法来对 n 个元素进行分类。随着对过程 PARTITION 的一次调用,就会产生出两个这样的集合 S_1 和 S_2,S_1 的所有元素小于或等于 S_2 的任何元素。因此,S_1 和 S_2 可独立分类。重复使用过程 PARTITION,每个集合都将被分类。算法 4.13 描述了这种分类的全过程。

算法 4.13　快速分类

```
procedure QUICKSORT(p,q)
    //将全程数组 A(1：n)中的元素 A(p),…,A(q)按递增的方式分类;认为 A(n + 1)已被定
      义,且大于或等于 A(p：q)的所有元素,即 A(n + 1) = + ∞//
    integer p,q;global n,A(1：n)
```

```
        if p < q
          then j←q + 1
          call PARTITION(p,j)
          call QUICKSORT(p,j - 1)    //j 是划分元素的位置//
          call QUICKSORT(j + 1,q)
        endif
    end QUICKSORT
```

4.5.2　快速分类分析

要分析 QUICKSORT,只需计算出它的元素比较数 C(n)即可。容易看出,其它运算的频率计数和 C(n)有相同的数量级。现作如下假定:

(1) 参加分类的 n 个元素各不相同;

(2) PARTITION 中的划分元素 v 是随机选取的。

假设(2)对于分析平均情况是必需的。为了能随机选出划分元素 v,可先假定有一个在区间[i,j]中生成随机整数的函数 RANDOM(i,j),然后用下列语句 i←RANDOM(m,p - 1);v←A(i);A(i)←A(m);i←m 来代换 PARTITION 中的赋值语句 v←A(m);i←m。

首先要获取 C(n)在最坏情况下的值 $C_W(n)$。容易证明:在 PARTITION 的每一次调用中元素的比较数至多是 p - m + 1。设 r 是在任一级递归上对 PARTITION 的所有调用的元素总数。在一级递归上只有一次调用,执行 PARTITION(1,n + 1),且 r = n;在二级至多作两次调用,且 r = n - 1,等等。在递归的任意一级上,所有的 PARTITION 共做 O(r)次元素比较,而每一级的 r,由于删去了前一级的划分元素,故比前一级的 r 至少要少 1。因此,$C_W(n)$是 r 由 n 变到 2 的和,即 $C_W(n) = C(n^2)$。

C(n)的平均值 $C_A(n)$比 $C_W(n)$小得多。在上面那些假定下,调用 PARTITION(m,p)时,所取划分元素 v 是 A(m∶p - 1)中第 i 小元素,1≤i≤p - m,具有相等的概率。因此,所留下待分类的两个子文件 A(m∶j - 1)和 A(j + 1∶p - 1)的概率是 1/(p - m),m≤j < p。由此,可以得到递归关系式

$$C_A(n) = n + 1 + \frac{1}{n}\sum_{1\le k\le n}[C_A(k-1) + C_A(n-k)] \qquad (4.1)$$

n + 1 是 PARTITION 第一次被调用时所需要的元素比较数。$C_A(0) = C_A(1) = 0$。用 n 乘式(4.1)两边,得

$$nC_A(n) = n(n+1) + 2[C_A(0) + C_A(1) + \cdots + C_A(n-1)] \qquad (4.2)$$

用 n - 1 代换式(4.2)中的 n,得

$$(n-1)C_A(n-1) = n(n-1) + 2[C_A(0) + \cdots + C_A(n-2)]$$

用式(4.2)减去上式,得

$$nC_A(n) - (n-1)C_A(n-1) = 2n + 2C_A(n-1)$$

即

$$C_A(n)/(n+1) = C_A(n-1)/n + 2/(n+1)$$

反复用这个等式去代换 $C_A(n-1), C_A(n-2), \cdots$,得到

$$C_A(n)/(n+1) = C_A(n-2)/(n-1) + 2/n + 2/(n+1)$$
$$= C_A(n-3)/(n-2) + 2/(n-1) + 2/n + 2/(n+1)$$

$$\cdots$$

$$= C_A(1)/2 + 2\sum_{3\le k\le n+1} 1/k \qquad (4.3)$$

由于 $$\sum_{3\le k\le n+1} 1/k \le \int_2^{n+1} \frac{dx}{x} < \log_e(n+1)$$

因此,由式(4.3)得出

$$C_A(n) < 2(n+1)\log_e(n+1) = O(n\log n)$$

由以上分析可知,快速分类算法的最坏情况时间是 $O(n^2)$,而平均情况时间是 $O(n\log n)$。

现在来考察递归所需要的栈空间,在最坏情况下,递归的最大深度可以达到 n - 1,因此所需的栈空间是 O(n)。例如,当对 PARTITION 的每一次调用,在其划分元素是 A(m：p)中的最小元素时,就会出现最坏情况。使用一个快速分类的迭代模型可以使所需的栈空间总量减至O(logn)。在这个迭代模型中,当 PARTITION 把文件 A(p：q)分成两个子文件 A(p：j - 1)和A(j + 1：q)之后,总是先对其中较小的那个子文件分类。第二条递归调用语句则用几条赋值语句和一条跳转到 loop 开始处的代码来代替,作了以上修改后,快速分类算法就呈现算法 4.14 的形式。

算法 4.14　QUICKSORT 的迭代模型

```
procedure QUICKSORT2(p,q)
  integer STACK(1：max),top   // max = 2⌊logn⌋ //
  global A(1：n);local integer j
  top←0
  loop
    while p < q do
      j←q + 1
      call PARTITION(p,j)
      if j - p < q - j then STACK(top + 1)←j + 1
                            STACK(top + 2)←q
                            q←j - 1
                       else STACK(top + 1)←p
                            STACK(top + 2)←j - 1
                            p←j + 1
      endif
      top←top + 2
    repeat //对较小的子文件分类//
    if top = 0 then return endif
    q←STACK(top);p←STACK(top - 1)
    top←top - 2
  repeat
end QUICKSORT2
```

现在来证明此算法所需的最大栈空间是 O(logn)。设所需最大栈空间是 S(n),它遵从

$$S(n) = \begin{cases} 2 + S[\lfloor (n-1)/2 \rfloor] & n > 1 \\ 0 & n \le 1 \end{cases}$$

它比 2logn 小。

如同 4.4 节所述,INSERTIONSORT 在 n 小于 16 的情况下执行是相当快的,因此,QUICKSORT2 可以在每当 q - p < 16 时用 INSERTIONSORT 来加速。

QUICKSORT 和 MERGESORT 的平均情况时间都是 O(nlogn),在平均情况下哪个更快一些呢?这里不给出其理论证明,而是给出一组在 IBM370/158 机上测试的数据来作近似的比较。这两个算法都是用 PL/1 编程的递归模型,对于 QUICKSORT,PARTITION 中划分元素采用了三者取中的规则(即划分元素是 A(m),A(m + p - 1)/2 和 A(p - 1)中值居中者)。输入数据集由(0,10000)的随机整数组成。表 4.3 记录了以毫秒为单位的实际平均计算时间。

研究这个表立即就可看出,对应于 n 的所有取值,QUICKSORT 都比 MERGESORT 快。而且还可看到,每当 n 值增加 500,QUICKSORT 的平均计算时间大致增加 250ms;MERGESORT 则不规则一些,n 每增加 500,其平均计算时间大约增加 350ms。

表 4.3　分类算法的平均计算时间/ms

n	1000	1500	2000	2500	3000	3500	4000	4500
MERGESORT	500	750	1050	1400	1650	2000	2250	2650
QUICKSORT	400	600	850	1050	1300	1550	1800	2050
n	5000	5500	6000	6500	7000	7500	8000	8500
MERGESORT	2900	3450	3500	3850	4250	4550	4950	5200
QUICKSORT	2300	2650	2800	3000	3350	3700	3900	4100

4.6　选择问题

上述 PARTITION 算法也可用来求选择问题的有效解。在这一问题中,给出 n 个元素 A(1：n),要求确定第 k 小的元素,如果划分元素 v 测定在 A(j)的位置上,则有 j - 1 个元素小于或等于 A(j),且有 n - j 个元素大于或等于 A(j)。因此,若 k < j,则第 k 小元素在 A(1：j - 1)中;若 k = j,则 A(j)就是第 k 小元素;若 k > j,则第 k 小元素是 A(j + 1：n)中第(k - j)小元素。所导出的算法是过程 SELECT(算法 4.15)。此过程把第 k 小元素放在 A(k),并划分剩余的元素,使得 A(i)≤A(k),1≤i < k 且 A(i)≥A(k),k < i≤n。

4.6.1　选择问题算法

算法 4.15　找第 k 小元素

```
procedure SELECT(A,n,k)
  //在数组 A(1),…,A(n)中找第 k 小元素 s 并把它放在位置 k,假设 1≤k≤n。将剩下的元素
    按如下方式重新排列,使 A(k) = t,对于 1≤m < t,有 A(m)≤t;对于 k < m≤n,有 A(m)≥t。
    A(n + 1) = + ∞//
  integer n,k,m,r,j;
  m←1;r←n + 1;A(n + 1)← + ∞;
  loop   //每当进入这一循环时,1≤m≤k≤r≤n + 1//
    j←r   //将剩余元素的最大下标加 1 后置给 j//
```

```
        call PARTITION(m,j)   //返回 j,它使得 A(j)是第 j 小的值//
        case
          :k = j: return
          :k < j: r←j   // j 是新的上界//
          :else: m←j + 1   // j + 1 是新的下界//
        endcase
      repeat
    end SELECT
```

例 4.6　使用 4.5 节中例 4.5 的数据集模拟 SELECT 的执行。

如果 k = 5,则 PARTITION 的首次调用就可得到解。这时,65 放在 A(5)中。倘若要找 A 的第七小元素,即 k = 7,其运算过程如下。在作了对 PARTITION 的首次调用之后,对 PARTITION 的下一次调用就是 call PARTITION(6,10):

A:	(5)	(6)	(7)	(8)	(9)	(10)	i	p
	65	85	80	75	70	+ ∞	10	9
		|——————————————|					
	65	70	80	75	85	+ ∞		

通过这次对 PARTITION 的调用,揭示出了 A 的第九小元素。接着的一次调用是 call PARTITION(6,9):

A:	(5)	(6)	(7)	(8)	(9)	(10)	i	p
	65	70	80	75	85	+ ∞	7	6
		|——|						
	65	70	80	75	85	+ ∞		

这次找到了第六小元素。由于在 SELECT 中 k≠j 仍为真,因此要再次调用 PARTITION, call PARTITION(7,9):

A:	(5)	(6)	(7)	(8)	(9)	(10)	i	p
	65	70	80	75	85	+ ∞	9	8
			|————|					
	65	70	75	80	85	+ ∞		

现在 80 是划分元素且已被正确地放在 A(8)中。然而,SELECT 还没找到第七小元素,因此还需要作一次 PARTITION 的调用,即 call PARTITION(7,8)。**这次调用只在 A(7)和 A(8)之间作了一次交换,然后返回,就找到了正确的值。**

下面分析 SELECT 的计算复杂度。首先,作与分析 QUICKSORT 时同样的假设,即:

(1) **这 n 个元素是互不相同的;**

(2) **随机选择划分元素,使得 A(m:p - 1)的每一个元素为划分元素的概率相同。**

在 SELECT 中,每次调用 PARTITION(m,j)都要花去 O(j - m + 1)的时间。当成功地执行了一次 PARTITION 之后,若还要继续划分,则或者 m 至少增加 1,或者 j 至少减小 1,

即继续划分的数组元素个数至少比上一次划分的数组元素数少 1。最初 m = 1 且 j = n + 1。因此,PARTITION 至多可被调用 n 次。所以,SELECT 的最坏情况时间至多是 $O(n^2)$。事实上,它的最坏情况时间可以达到 $O(n^2)$。例如,最坏情况是输入 A(1：n)能使得对 PARTITION 的第 i 次调用的划分元素是第 i 小元素的这样一些数据,而 k = n 时,m 随着 PARTITION 的每一次调用而增加 1,j 则保持不变。于是,作 n 次调用的时间总量是 $O\left(\sum_{i=1}^{n}(i+1)\right) = O(n^2)$,而 SELECT 的平均计算时间只是 O(n)。在证明这一结论之前,先给出平均时间的确切含义。

设 $T_A^k(n)$是找 A(1：n)的第 k 小元素的平均时间。这是对 n 个互不相同元素的所有 n! 种不同排列都求出找第 k 小元素的时间后取平均值。如果设 $T_A(n)$是 SELECT 的平均计算时间,则可对它定义如下:

$$T_A(n) = \frac{1}{n}\sum_{1\le k\le n} T_A^k(n)。$$

为证明平均计算时间方便起见,还定义

$$R(n) = \max_k\{T_A^k(n)\}$$

容易看出 $T(n)\le R(n)$。下面就来证明 $T_A(n) = O(n)$。

定理 4.4　SELECT 的平均计算时间 $T_A(n)$是 O(n)。

证明　在对 PARTITION 第一次调用时,划分元素 v 是第 i 小元素的概率为 1/n,$1\le i\le n$(这是根据 v 的随机选择所确定的)。而 PARTITION 和 SELECT 中 case 语句所要求的时间是 O(n)。因此,存在常数 c,c > 0,使得

$$T_A^k(n) \le cn + \frac{1}{n}\left(\sum_{1\le i<k} T_A^{k-i}(n-i) + \sum_{k<i\le n} T_A^k(i-1)\right) \quad n\ge 2$$

因此,

$$R(n) \le cn + \frac{1}{n}\max_k\left\{\sum_{1\le i<k} R(n-i) + \sum_{k<i\le n} R(i-1)\right\}$$

$$= cn + \frac{1}{n}\max_k\left\{\sum_{n-k+1}^{n-1} R(i) + \sum_{k}^{n-1} R(i)\right\} \quad n\ge 2 \qquad (4.4)$$

选择 $c\ge R(1)$,并施归纳法于 n,证明对于所有的 $n\ge 2$,有 $R(n)\le 4cn$。

归纳基础　对于 n = 2,式(4.4)给出:

$$R(n) \le 2c + \frac{1}{2}\max\{R(1),R(1)\} \le 2.5c < 4cn$$

归纳假设　假定 $R(n)\le 4cn$,对于所有的 $n,2\le n < m$。

归纳步骤　对于 n = m,式(4.4)给出:

$$R(m) \le cm + \frac{1}{m}\max_k\left\{\sum_{m-k+1}^{m-1} R(i) + \sum_{k}^{m-1} R(i)\right\}$$

由于 R(n)是 n 的非降函数,故可以得到当 m 为偶数而 k = m/2 时,或者当 m 为奇数而 k = (m + 1)/2 时,取

$$\sum_{m-k+1}^{m-1} R(i) + \sum_{k}^{m-1} R(i)$$

的极大值。因此,若 m 为偶数,则

$$R(m) \le cm + \frac{2}{m}\sum_{m/2}^{m-1} R(i) \le cm + \frac{8c}{m}\sum_{m/2}^{m-1} i < 4cm$$

若 m 是奇数,则

$$R(m)\leq cm+\frac{2}{m}\sum_{(m+1)/2}^{m-1}R(i)\leq cm+\frac{8c}{m}\sum_{(m+1)/2}^{m-1}i<4cm$$

由于 $T_A(n)\leq R(n)$,所以 $T_A(n)\leq 4cn$,故 $T_A(n)$是 O(n)。证毕。

选择算法所需要的附加空间是 O(1)。

4.6.2 最坏情况时间是 O(n)的选择算法

通过精细地挑选划分元素 v,可以得到一个最坏情况时间复杂度是 O(n)的选择算法。为了得到这样一个算法,元素 v 必须选择成比一部分元素小而比另一部分元素大。使用二次取中间值规则可以选出满足以上要求的元素 v。这一规则是,将参加划分的 n 个元素分成$\lfloor n/r\rfloor$组,每组有 r 个元素,这里 r 是一个大于 1 的正整数。剩余的 $n-r\lfloor n/r\rfloor$个元素忽略不计。先对这$\lfloor n/r\rfloor$组中每组的 r 个元素分类并找出其中间值元素 m_i,$1\leq i\leq\lfloor n/r\rfloor$;然后再从这$\lfloor n/r\rfloor$个 m_i 中找出它们的中间值 mm,并将 mm 作为划分元素。图 4.7 显示了 n = 35, r = 7 时的 m_i 和 mm。图中,$B_1,\cdots,B_5$ 是 5 个元素组,每组的 7 个元素沿列而下已排成一个非降序列。每列中间的元素就是 m_i。而且这些列也按 m_i 的非降次序进行了排列。因此,第 3 列的 m_i 就是 mm。

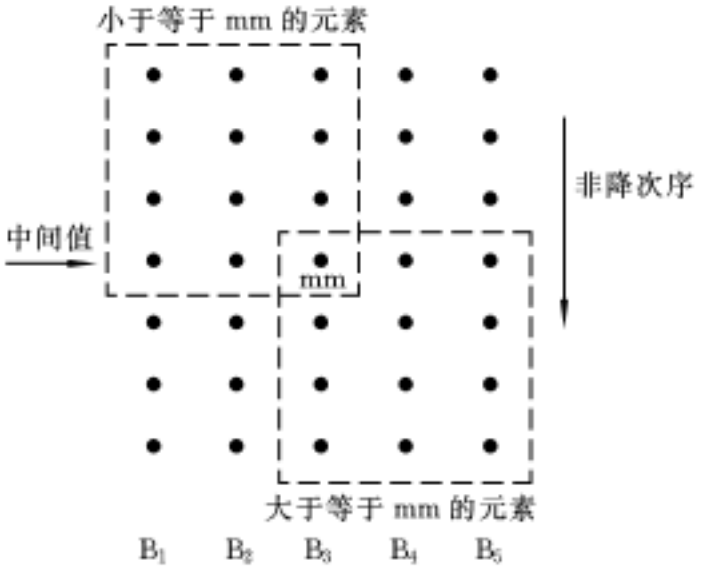


图 4.7 n = 35, r = 7 时二次取中间值

由于 r 个元素的中间值是第$\lceil r/2\rceil$小元素,因此可得(见图 4.7)至少有$\lceil\lfloor n/r\rfloor/2\rceil$个 m_i 是小于或等于 mm 的,且至少有$\lfloor n/r\rfloor-\lceil\lfloor n/r\rfloor/2\rceil+1\geq\lceil\lfloor n/r\rfloor/2\rceil$个 m_i 大于或等于 mm。所以,至少有$\lceil r/2\rceil\lceil\lfloor n/r\rfloor/2\rceil$个元素小于或等于(或者大于或等于)mm。当 r = 5 时,这个量至少是 $1.5\lfloor n/5\rfloor$。如果在 r = 5 的情况下使用二次取中值规则来选择 v = mm,那么,至少有 $1.5\lfloor n/5\rfloor$个元素小于或等于选择元素 v。这又意味着至多有 $n-1.5\lfloor n/5\rfloor\leq 0.7n+1.2$ 个元素大于 v。类似地可以推出至多有 0.7n + 1.2 个元素小于 v。于是,二次取中值规则所得到的元素满足对 v 的要求。

算法 4.16 使用二次取中规则的选择算法的说明性描述

```
procedure SELECT2(A,k,n)
    // 在集合 A 中找第 k 小元素 //
    1  若 n≤r,则采用插入法直接对 A 分类并返回第 k 小元素
    2  把 A 分成大小为 r 的⌊n/r⌋个子集合,忽略剩余的元素
    3  设 M = {m1, m2, ⋯, m⌊n/r⌋}是上面⌊n/r⌋个子集合的中间值的集合
    4  v←SELECT2(M,⌈⌊n/r⌋/2⌉,⌊n/r⌋)
    5  用 PARTITION 划分 A,v 作为划分元素
    6  假设 v 在位置 j
    7  case
        ：k = j：return (v)
```

```
        ：k < j：设 S 是 A(1：j - 1)中元素的集合
                return(SELECT2(S,k,j - 1))
        ：else：设 R 是 A(j + 1：n)中元素的集合
                return(SELECT2(R,k - j,n - j))
    endcase
end SELECT2
```

过程 SELECT2 是先采用二次取中值规则选取划分元素,然后找第 k 小元素的选择算法的说明性描述。它是一个递归过程。现在,对任一给定的 r 来分析过程 SELECT2。首先,考虑 r = 5 且 A 中各元素互不相同的情况。那么,第 7 步骤的 | S | 和 | R | 至多是 0 .7n + 1 .2,对于 n≥24,它不会大于 3n/ 4。设 T(n)是 SELECT2 所需的最坏情况时间。由于 r = 5 是固定的,因此,步骤 1 可在 O(1)时间内完成,步骤 3 的每一个 m 也可在 O(1)时间内找到。从而步骤 3 至多需要 O(n)时间。步骤 2,5 和 6 至多也只需要 O(n)时间。第 4 步的时间是 T(n/ 5),而第 7 步的时间是,当 n≥24 时至多是 T(3n/ 4)。于是,对于 n≥24,得

$$T(n) \leq T(n/5) + T(3n/4) + cn \tag{4.5}$$

对于 n≤24,T(n)显然是一线性时间,只要将上式中的 c 取得足够大,这线性时间就可表示成

$$T(n) \leq cn$$

至此,用归纳法容易证明:对于 n≥1,有

$$T(n) \leq 20cn$$

这一结论表明,在 r = 5 的情况下,求解 n 个不同元素选择问题的算法 SELECT2 的最坏情况时间是 O(n)。至于对 r 取其它值情况的讨论则留下作为习题。

当 A 中的元素不是完全不同时会出现什么问题呢?在这种情况下,当在步骤 5 调用 PARTITION 时,由于所产生的 S 和 R 这两个集合中可能有一些等于 v 的元素,因此可能导致 | S | 或 | R | 大于 0 .7n + 1 .2。处理此问题的一种方法是把 A 分成 3 个集合 U,S 和 R,使 U 由所有与 v 相同的元素组成,S 是 A 中所有比 v 小的元素的集合,A 中所有比 v 大的元素组成 R。于是将步骤 7 修改成:

```
case
    ：| S |≥k：return(SELECT2(S,k,| S |))
    ：| S | + | U |≥k：return(v)
    ：else：return(SELECT2(R,k - | S | - | U |,| R |))
endcase
```

这样处理时,由于 | S | 和 | R | ≤0 .7n + 1 .2,因此递归式(4 .5)仍然成立。所以,即使元素不是完全不同时,修改后的 SELECT2 的时间复杂度仍是 O(n)。

对于元素完全不同的情况,另一种处理方法是取 r≠5 的值,为了看出为什么要取 r≠5 的值,在 r = 5 且 A 中元素不是完全不同的情况下分析 SELECT2。考察有 0 .7n + 1 .2 个元素比 v 小,而其余元素都等于 v 的情况。通过检查 PARTITION 的执行情况可以发现,这其余元素至多有一半可能在 S 中。因此,| S | ≤0 .7n + 1 .2 + (0 .3n - 1 .2)/ 2 = 0 .85n + 0 .6。类似地,| R | ≤0 .85n + 0 .6。由于包含在算法 4 .16 步骤 4 和步骤 7 的两次递归调用的元素总数现在是1 .05n + 0 .6 > n,因此 SELECT2 的复杂度不是 O(n)。如果取 r = 9,则至少有 2 .5

(n/9)个元素小于或等于 v,且至少也有这么多个元素大于或等于 v。因此,对于 n≥90,|S|和|R|都至多是 $n-2.5(n/9)+\frac{1}{2}[2.5(n/9)]=n-1.25(n/9)\leq 31n/36+1.25\leq 63n/72$。从而可以得到下面的递归式:

$$T(n)=\begin{cases} c_1 n & n<90 \\ T(n/9)+T(63n/72)+c_1 n & n\geq 90 \end{cases}$$

其中,c_1 是一个适当的常数。

可以施归纳法于 n 证明:对于 n≥1,有

$$T(n)\leq 72c_1 n$$

SELECT2 所需要的附加空间除了几个简单变量所需要的空间外,还需要作为递归栈使用的空间。由于步骤 7 的递归调用是这次执行 SELECT2 的最后一条语句,故容易将它消去递归。因此,仅步骤 4 的递归需要栈空间。递归的最大深度是 logn,所以需要 O(logn)的栈空间。

4.6.3 SELECT2 的实现

在用 SPARKS 写实现 SELECT2 的算法以前,需要解决两个问题:①怎样找大小为 r 的集合的中间值?②将步骤 3 所找到的$\lfloor n/r \rfloor$个中间值放在什么地方?由于所使用的 r 值都不大(例如 r=5 或 9),因此,可用算法 4.7 的改进型 INSERTIONSORT(A,i,j)对每组的 r 个元素实行有效的分类。分类后的在 A(i,j)中间的那个元素就是这 r 个元素的中间值。把各组所找到的中间值存放在包含所有组全部元素的那个数组的前部,这对于处理步骤 4 是很方便的。例如,如果正在找 A(m:p)的第 k 小元素,就将这些中间值依次存放在 A(m),A(m+1),A(m+2),…中。实现 SELECT2 的 SPARKS 描述是过程 SEL(算法 4.17)。在 SEL 中,通过第 6 行和 21 行的 loop—repeat 以及 16 行到 21 行的语句等价代换了步骤 7 的递归调用。INTERCHANGE(x,y)仅是交换 x 和 y 的值。

算法 4.17 SELECT2 的 SPARKS 描述

```
line procedure SEL(A,m,p,k)
     // 返回一个 i,使得 i∈[m,p],且 A(i)是 A(m:p)中第 k 小元素,r 是一个全程变量,其取值
       为大于 1 的整数 //
1    global r
2    integer n,i,j
3    if p-m+1≤r then call INSERTIONSORT(A,m,p)
4                                return (m+k-1)
5        endif
6        loop
7            n←p-m+1   // 元素数 //
8             for i←1 to ⌊n/r⌋ do   // 计算中间值 //
9             call INSERTIONSORT (A,m+(i-1)*r,m+i*r-1)
                // 将中间值收集到 A(m:p)的前部 //
10           call INTERCHANGE(A(m+i-1),A(m+(i-1)*r+⌊n/2⌋-1))
11            repeat
```

```
12              j←SEL(A,m,m+⌊n/r⌋-1,⌈⌊n/r⌋/2⌉)   //mm//
13            call INTERCHANGE(A(m),A(j))   //产生划分元素//
14            j←p+1
15            call PARTITION(m,j)
16            case
17             :j-m+1=k: return(j)
18             :j-m+1>k: p←j-1
19             :else: k←k-(j-m+1);m←j+1
20            endcase
21          repeat
22          end SEL
```

4.7　斯特拉森矩阵乘法

二维数组无论在数值还是在非数值计算领域中都是一种相当基本而又极其重要的抽象数据结构,矩阵则是它的数学表示,因此,在研究矩阵的基本运算时,尽可能改进运算的效率无疑是件非常重要的工作。

矩阵加和矩阵乘是两种最基本的矩阵运算。设 A 和 B 是两个 $n\times n$ 矩阵,这两个矩阵相加指的是它们对应元素相加作为其和矩阵的相应元素,因此它们的和矩阵仍是一个 $n\times n$ 矩阵,记为 $C=A+B$。其时间显然为 $\Theta(n^2)$。如果将矩阵 A 和 B 的乘积记为 $C=AB$,那么 C 也是一个 $n\times n$ 矩阵,乘积 C 的第 i 行第 j 列的元素 $C(i,j)$等于 A 的第 i 行和 B 的第 j 列对应元素乘积的和,可表示为

$$C(i,j)=\sum_{1\le k\le n}A(i,k)B(k,j)\qquad 1\le i,j\le n \tag{4.6}$$

按上式计算 $C(i,j)$需要做 n 次乘法和 $n-1$ 次加法,而乘积矩阵 C 有 n^2 个元素,因此,由矩阵乘定义直接产生的矩阵乘算法的时间为 $\Theta(n^3)$。

人们长期以来对改进矩阵乘法的效率作过不少尝试,设计了一些改进算法,但在计算时间上都仍旧囿界于 n^3 这一数量级。直到 1969 年斯特拉森(V.Strassen)利用分治策略并加上一些处理技巧设计出一种矩阵乘算法以后,才在计算时间数量级上取得突破。他所设计的算法的计算时间是 $O(n^{2.81})$。此结果在第一次发表时曾震动了数学界。下面介绍斯特拉森矩阵算法的基本设计思想与主要处理技巧。

为简单起见,假定 n 是 2 的幂,即存在一非负整数 k,使得 $n=2$。在 n 不是 2 的幂的情况下,则可对 A 和 B 增加适当的全零行和全零列,使其变成级是 2 的幂的方阵。按照分治设计策略,首先可以将 A 和 B 都分成 4 个$(n/2)\times(n/2)$矩阵,于是 A 和 B 就可以看成是两个以$(n/2)\times(n/2)$矩阵为元素的 2×2 矩阵。对这两个 2×2 矩阵施以通常的矩阵乘法运算(即通过(2.6)式计算乘积矩阵元素),可得

$$\begin{bmatrix}A_{11}&A_{12}\\A_{21}&A_{22}\end{bmatrix}\begin{bmatrix}B_{11}&B_{12}\\B_{21}&B_{22}\end{bmatrix}=\begin{bmatrix}C_{11}&C_{12}\\C_{21}&C_{22}\end{bmatrix} \tag{4.7}$$

其中,

$$C_{11} = A_{11}B_{11} + A_{12}B_{21},\ C_{12} = A_{11}B_{12} + A_{12}B_{22},\ C_{21} = A_{21}B_{11} + A_{22}B_{21},\ C_{22} = A_{21}B_{12} + A_{22}B_{22} \tag{4.8}$$

使用通常的矩阵乘法和加法,计算$(n/2)\times(n/2)$矩阵 C_{11}, C_{12}, C_{21} 和 C_{22} 的各元素的值以及 $C = AB$ 各乘积元素的值,可以直接证明

$$C = AB = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

如果分块子矩阵的级(这里是 $n/2$ 级方阵)大于 2,则可以继续将这些子矩阵分成更小的方阵,直至每个子方阵只含一个元素,以至可以直接计算其乘积为止。这样的算法显然是由分治策略设计而得的。为了用式(4.8)计算 AB,需要执行以$(n/2)\times(n/2)$矩阵为元素的 8 次乘法和 4 次加法。由于每两个 $n/2$ 级方阵相加可在对于某个常数 c 而言的 cn^2 时间内完成,如果所得到的分治算法的时间用 $T(n)$表示,则可以得到下面的递归关系式:

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(n/2) + dn^2 & n > 2 \end{cases}$$

其中,b 和 d 是常数。

求解这个递归关系式得到 $T(n) = O(n^3)$,与通常的矩阵乘算法计算时间具有相同的数量级。由于矩阵乘法比矩阵加法的花费要大($O(n^3)$对 $O(n^2)$),斯特拉森发现了在分治设计的基础上使用一种减少乘法次数而让加减法次数相应增加的处理方法来计算式(4.8)中的 C_{ij}。其处理方法是,先用 7 个乘法和 10 个加(减)法来算出下面 7 个$(n/2)\times(n/2)$矩阵:

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned} \tag{4.9}$$

然后用 8 个加(减)法算出这些 C_{ij}:

$$\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned} \tag{4.10}$$

以上共用 7 次乘法和 18 次加(减)法。

由 $T(n)$所得出的递归关系式是

$$T(n) = \begin{cases} b & n \le 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases} \tag{4.11}$$

其中,a 和 b 是常数。

求解这个递归关系式,得

$$T(n) = an^2(1 + 7/4 + (7/4)^2 + \cdots + (7/4)^{k-1}) + 7^k T(1)$$

$$\leq cn^2(7/4)^{\log n} + 7^{\log n} \qquad c\text{是一个常数}$$
$$= cn^{\log 4 + \log 7 - \log 4} + n^{\log 7}$$
$$= (c+1)n^{\log 7} = O(n^{\log 7}) \approx O(n^{2.81})$$

在斯特拉森之后,有很多人继续设法改进他的结果,值得指出的是 J.E.Hopcroft 和 L.R.Kerr已经证明了两个 2×2 矩阵相乘必须要用 7 次乘法,因此要进一步获得改进,则需考虑 3×3 或 4×4 等更高级数的分块子矩阵或者用完全不同的设计策略。

最后,要提请读者注意的是,斯特拉森矩阵乘法目前还只具有理论意义,因为只有当 n 相当大时它才优于通常的矩阵乘法。经验表明,当 n 取为 120 时,斯特拉森矩阵乘法与通常的矩阵乘法在计算时间上仍无显著差别。尽管如此,它还是给出了有益的启示:即使是由定义出发所直接给出的明显算法并非总是最好的。斯特拉森矩阵乘法可能为获得更有效和在计算机上切实可行的算法奠定了基础。

关于矩阵乘法的更深入的讨论,读者可参看《线性代数与多项式的快速算法》(游兆永编,上海科学技术出版社 1980 年出版)。

习　题　四

4.1　利用 2.5 节的规则,将过程 DANDC(算法 4.1)转换成迭代形式 DANDC2,使得由 DANDC2 通过化简能够得到过程 DANDC1(算法 4.2)。

4.2　在下列情况下求解 2.1 节的递归关系式:

$$T(n) = \begin{cases} g(n) & n\text{足够小} \\ 2T(n/2) + f(n) & \text{否则} \end{cases}$$

当① $g(n) = O(1)$和 $f(n) = O(n)$;② $g(n) = O(1)$和 $f(n) = O(1)$时。

4.3　根据 4.2 节开始所给出的二分检索策略,写一个二分检索的递归过程。

4.4　作一个“二分”检索算法,它将原集合分成 1/3 和 2/3 大小的两个子集合。将这个算法与算法 4.3 相比较。

4.5　作一个“三分”检索算法,首先检查 n/3 处的元素是否等于某个 x 的值,然后检查 2n/3 处的元素。这样,或者找到 x,或者把集合缩小到原来的 1/3。分析此算法在各种情况下的计算复杂度。

4.6　对于含有 n 个内部结点的二元树,证明

$$E = I + 2n$$

其中,E、I 分别为外部和内部路径长度。

4.7　证明 BINSRCH1 的最好、平均和最坏情况的计算时间对于成功和不成功的检索都是 $\Theta(\log n)$。

4.8　将递归过程 MAXMIN 翻译成在计算上等价的非递归过程。

4.9　按以下处理思想写一个找最大最小元素的迭代程序并分析它的比较次数。它不是以分治策略为基础的,但可能比 MAXMIN 更有效。先比较相邻的两个元素,然后,将较大的元素与当前最大元素相比较,较小的元素与当前最小元素相比较。

4.10　过程 MERGESORT 的最坏情况时间是 O(n log n)。它的最好情况时间是什么?能说归并分类的时间是 $\Theta(n \log n)$吗?

4.11　写一个“由底向上”的归并分类算法,从而取消对栈空间的需要。

4.12　如果一个分类算法在结束时相同元素出现的顺序与集合没分类以前一样,则称此算法是稳定的。归并分类算法是稳定的算法吗?

4.13 QUICKSORT是一种不稳定的分类算法。但是,若把A(i)中的关键字变成A(i) * n + i - 1,那么,所有的关键字都不相同了。在分类之后,如何将关键字恢复成原来的值呢?

4.14 讨论在过程PARTITION(即算法4.12)中,将语句if i < p改成if i≤p的优缺点。在数据集(5,4,3,2,5,8,9)上模拟执行这两个算法,看看它们在执行时有何不同之处。

4.15 在下面两组关键字上模拟执行QUICKSORT(算法2.13):(1,1,1,1,1)和(5,5,8,3,4,3,2)。

4.16 在过程PARTITION中,如果将A(i)≥v改成A(i) > v,那么,在PARTITION中还要作哪些改变?写出作出修改后的划分集合的算法并将它与原PARTITION相比较。

4.17 比较MERGESORT1和QUICKSORT2这两个分类算法。设计对这两个算法的平均和最坏情况时间进行比较的数据集。

4.18 如何利用插入分类算法INSERTIONSORT来改进快速分类算法QUICKSORT2?写出这一改进算法。为此,算法4.7需作哪些修改?写出INSERTIONSORT的修改模型。

4.19 画出对4个元素分类的比较树。

4.20 (1) 假设只有在A中元素各不相同时才使用算法SELECT2。问r取下列哪些值能保证算法在最坏情况下用O(n)时间就可执行完毕?证明你的结论:r = 3,7,9,11。

(2) 如果将r选得较大(但,是适当的),那么,SELECT2的计算时间是增加还是减少?为什么?

4.21 在题4.20中,将r = 3,7,9,11改为r = 7,11,13,15并且在不限制A中元素各不相同的情况下,再完成题4.20。

4.22 用SPARKS语言写一个计算时间为$O(n^3)$的两个n×n矩阵相乘的算法,并确定作乘法和加法的精确次数。

4.23 通过手算证明由(4.9)和(4.10)式确实能得到C_{11},C_{12},C_{21}和C_{22}的正确值。

4.24 在n是3的幂的情况下,考虑n×n级矩阵的乘法。使用分治策略设计的算法可以减少3×3级矩阵乘法通常所要作的27次乘法。对于3×3级矩阵乘法必须作多少次乘法才使得计算时间比$O(n^{2.81})$小呢?对于4×4级矩阵乘法作同样的讨论。

4.25 斯特拉森算法的另一种形式是用下面的恒等式来计算式(4.8)中的C_{ij}(这样处理共用了7次乘法和15次加法):

$$S_1 = A_{21} + A_{22} \qquad M_1 = S_2 S_6 \qquad T_1 = M_1 + M_2$$
$$S_2 = S_1 - A_{11} \qquad M_2 = A_{11} B_{11} \qquad T_2 = T_1 + M_4$$
$$S_3 = A_{11} - A_{21} \qquad M_3 = A_{12} B_{21}$$
$$S_4 = A_{12} - S_2 \qquad M_4 = S_3 S_7$$
$$S_5 = B_{12} - B_{11} \qquad M_5 = S_1 S_5$$
$$S_6 = B_{22} - S_5 \qquad M_6 = S_4 B_{22}$$
$$S_7 = B_{22} - B_{12} \qquad M_7 = A_{22} S_8$$
$$S_8 = S_6 - B_{21}$$

C_{ij}是

$$C_{11} = M_2 + M_3$$
$$C_{12} = T_1 + M_5 + M_6$$
$$C_{21} = T_2 - M_7$$
$$C_{22} = T_2 + M_5$$

证明由这些恒等式确实可计算出C_{11},C_{12},C_{21}和C_{22}的正确值。

第 5 章

贪 心 方 法

5.1 一 般 方 法

在现实世界中,有这样一类问题:它有 n 个输入,而它的解就由这 n 个输入的某个子集组成,只是这个子集必须满足某些事先给定的条件。把那些必须满足的条件称为约束条件;而把满足约束条件的子集称为该问题的可行解。显然,满足约束条件的子集可能不止一个,因此,可行解一般来说是不唯一的。为了衡量可行解的优劣,事先也给出了一定的标准,这些标准一般以函数形式给出,这些函数称为目标函数。那些使目标函数取极值(极大值或极小值)的可行解,称为最优解。对这一类需求取最优解的问题,又可根据描述约束条件和目标函数的数学模型的特性或求解问题方法的不同进而细分为线性规划、整数规划、非线性规划、动态规划等问题。尽管各类规划问题都有一些相应的求解方法,但其中的某些问题,还可用一种更直接的方法来求解,这种方法就是贪心方法。

贪心方法是一种改进了的分级处理方法。它首先根据题意,选取一种量度标准;然后按这种量度标准对这 n 个输入排序,并按序一次输入一个量。如果这个输入和当前已构成在这种量度意义下的部分最优解加在一起不能产生一个可行解,则不把此输入加到这部分解中。这种能够得到某种量度意义下的最优解的分级处理方法称为贪心方法。要注意的是,对于一个给定的问题,往往可能有好几种量度标准。初看起来,这些量度标准似乎都是可取的。但实际上,用其中的大多数量度标准作贪心处理所得到该量度意义下的最优解并不是问题的最优解,而是次优解。尤其值得指出的是,把目标函数作为量度标准所得到的解也不一定是问题的最优解。因此,选择能产生问题最优解的最优量度标准是使用贪心法设计求解的核心问题。在一般情况下,要选出最优量度标准并不是一件容易的事,不过,一旦能选择出某个问题的最优量度标准,那么用贪心方法求解这个问题则特别有效。

贪心方法可以用下面的抽象化控制来描述。

算法 5.1 贪心方法的抽象化控制

```
procedure GREEDY(A,n)
    //A(1:n)包含 n 个输入//
    solution←        //将解向量 solution 初始化为空//
    for i←1 to n do
      x←SELECT(A)
      if FEASIBLE(solution,x)
          then solution←UNION(solution,x)
      endif
    repeat
```

```
        return (solution)
    end GREEDY
```

函数 SELECT 的功能是按某种最优量度标准从 A 中选择一个输入,把它的值赋给 x 并从 A 中消去它。FEASIBLE 是一个布尔函数,它判定 x 是否可以包含在解向量中。UNION将 x 与解向量结合并修改目标函数。过程 GREEDY 描述了用贪心策略设计算法的主要工作和基本控制路线。一旦给出一个特定的问题,就可将 SELECT, FEASIBLE 和 UNION具体化并付诸实现。

5.2 背包问题

本节介绍使用贪心设计策略来解决更复杂的问题——背包问题。已知有 n 种物品和一个可容纳 M 重量的背包,每种物品 i 的重量为 w_i。假定将物品 i 的一部分 x_i 放入背包就会得到 $p_i x_i$ 的效益,这里,$0 \le x_i \le 1$, $p_i > 0$。采用怎样的装包方法才会使装入背包物品的总效益最大呢?显然,由于背包容量是 M,因此,要求所有选中要装入背包的物品总重量不得超过 M。如果这 n 件物品的总重量不超过 M,则把所有物品装入背包自然获得最大效益。如果这些物品重量的和大于 M,则在这种情况下该如何装包呢?这是本节所要解决的问题。由以上叙述,可将这个问题形式描述如下:

极大化
$$\sum_{1 \le i \le n} p_i x_i \tag{5.1}$$

约束条件
$$\sum_{1 \le i \le n} w_i x_i \le M \tag{5.2}$$

$$0 \le x_i \le 1, p_i > 0, w_i > 0, 1 \le i \le n \tag{5.3}$$

其中,式(5.1)是目标函数,式(5.2)和式(5.3)是约束条件。满足约束条件的任一集合$(x_1, \cdots, x_n)$是一个可行解,使目标函数取最大值的可行解是最优解。

例 5.1 考虑下列情况下的背包问题: n = 3, M = 20, $(p_1, p_2, p_3) = (25, 24, 15)$, $(w_1, w_2, w_3) = (18, 15, 10)$。其中的 4 个可行解是

(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
① (1/2, 1/3, 1/4)	16.5	24.25
② (1, 2/15, 0)	20	28.2
③ (0, 2/3, 1)	20	31
④ (0, 1, 1/2)	20	31.5

在这 4 个可行解中,第④个解的效益值最大。下面将可看到,这个解是背包问题在这一情况下的最优解。

为了获取背包问题的最优解,必须把物品放满背包。由于可以只放物品 i 的一部分到背包中去,因此这一要求是可以达到的。如果用贪心策略来求解背包问题,则正如 5.1 节中所说的一样,首先要选出最优的量度标准。不妨先取目标函数作为量度标准,即每装入一件物品就使背包获得最大可能的效益值增量。在这种量度标准下的贪心方法就是按效益值的非增次序将物品一件件放到背包中去。如果正在考虑中的物品放不进去,则可只取其一部分来装满背包。但是,这最后一次的放法可能不符合使背包每次获得最大效益增量的量度标

准,这可以换一种能获得最大增量的物品,将它(或它的一部分)放入背包,从而使最后一次装包也符合量度标准的要求。例如,假定还剩有两个单位的空间,而在背包外还有两种物品,这两种物品有($p_i = 4, w_i = 4$)和($p_j = 3, w_j = 2$),则使用 j 就比用 i 要好些。下面对例 5.1 的数据使用这种选择策略。

物品 1 有最大的效益值($p_1 = 25$),因此首先将物品 1 放入背包,这时 $x_1 = 1$ 且获得 25 的效益。背包容量中只剩下两个单位空着。物品 2 有次大的效益值($p_2 = 24$),但 $w_2 = 15$,背包中装不下物品 2,使用 $x_2 = 2/15$ 就正好装满背包。不难看出物品 2 的 2/15 比物品 3 的 2/10 效益值高。所以,此种选择策略得到②的解,总效益值是 28.2。它是一个次优解。由此例可知,按物品效益值的非增次序装包不能得到最优解。

为什么上述贪心策略不能获得最优解呢?原因在于背包可用容量消耗过快。由此,很自然地启发我们用容量作为量度,让背包容量尽可能慢地被消耗。这就要求按物品重量的非降次序来把物品放入背包。例 5.1 的解③就是使用这种贪心策略得到的,它仍是一个次优解。这种策略也只能得到次优解,其原因在于容量虽然慢慢地被消耗,但效益值没能迅速地增加。这又启发我们采用在效益值的增长速率和容量的消耗速率之间取得平衡的量度标准。即每一次装入的物品应使它占用的每一单位容量获得当前最大的单位效益。这就需使物品的装入次序按 p_i/w_i 比值的非增次序来考虑。在这种策略下的量度是已装入物品的累计效益值与所用容量之比。其量度标准是每次装入要使累计效益值与所用容量的比值有最多的增加或最少的减小(第二次和以后的装入可能使此比值减小)。将此贪心策略应用于例 5.1 的数据,得到解④。如果将物体事先按 p_i/w_i 的非增次序分好类,则过程 GREEDY-KNAPSACK 就得出这一策略下背包问题的解。如果将物品分类的时间不算在内,则此算法所用时间为 O(n)。

算法 5.2　背包问题的贪心算法

```
procedure GREEDY-KNAPSACK(P,W,M,X,n)
    //P(1:n)和 W(1:n)分别含有按 P(i)/W(i)≥P(i+1)/W(i+1)排序的 n 件物品的效益值
      和重量。M 是背包的容量大小,而 X(1:n)是解向量//
   real P(1:n),W(1:n),X(1:n),M,cu;
   integer i,n;
   X←0    //将解向量初始化为零//
   cu←M   //cu 是背包剩余容量//
   for i←1 to n do
     if W(i) > cu then exit endif
     X(i)←1
     cu←cu - W(i)
   repeat
   if i≤n then X(i)←cu/W(i)
   endif
end GREEDY-KNAPSACK
```

值得指出的是,如果把物品事先按效益值的非增次序或重量的非降次序分好类,再使用算法 5.2 就可分别得到量度标准为最优(使每次效益增量最大或使容量消耗最慢)的

解。由背包问题量度选取的研究可知,选取最优的量度标准实为用贪心方法求解问题的核心。

下面证明用第三种策略的贪心算法所得的贪心解是一个最优解。基本思想是,把这贪心解与任一最优解相比较,如果这两个解不同,就去找开始不同的第一个 x_i,然后设法用贪心解的这个 x_i 去代换最优解的那个 x_i,并证明最优解在分量代换前后的总效益无任何变化。反复进行这种代换,直到新产生的最优解与贪心解完全一样,从而证明了贪心解是最优解。这种证明最优解的方法在本书中经常使用,因此读者从现在起就应掌握它。

定理 5.1 如果 $p_1/w_1 \geq p_2/w_2 \geq \cdots \geq p_n/w_n$,则算法 GREEDY-KNAPSACK 对于给定的背包问题实例生成一个最优解。

证明 设 $X=(x_1,\cdots,x_n)$是 GREEDY-KNAPSACK 所生成的解。如果所有的 x_i 等于 1,显然这个解就是最优解。于是,设 j 是使 $x_j \neq 1$ 的最小下标。由算法可知,对于 $1 \leq i < j$, $x_i = 1$;对于 $j < i \leq n$, $x_i = 0$;对于 j, $0 \leq x_j < 1$。如果 X 不是一个最优解,则必定存在一个可行解 $Y=(y_1,\cdots,y_n)$,使得 $\sum p_i y_i > \sum p_i x_i$。不失一般性,可以假定 $\sum w_i y_i = M$。设 k 是使得 $y_k \neq x_k$ 的最小下标。显然,这样的 k 必定存在。由上面的假设,可以推得 $y_k < x_k$。这可从 3 种可能发生的情况,即 $k<j$, $k=j$ 或 $k>j$ 分别得证明:

(1) 若 $k<j$,则 $x_k = 1$。因 $y_k \neq x_k$,从而 $y_k < x_k$。

(2) 若 $k=j$,由于 $\sum w_j x_i = M$,且对 $1 \leq i < j$,有 $x_i = y_i = 1$,而对 $j < i \leq n$,有 $x_i = 0$。若 $y_k > x_k$,显然有 $\sum w_i y_i > M$,与 Y 是可行解矛盾。若 $y_k = x_k$,与假设 $y_k \neq x_k$ 矛盾,故 $y_k < x_k$。

(3) 若 $k>j$,则 $\sum w_i y_i > M$,这是不可能的。

现在,假定把 y_k 增加到 x_k,那么必须从$(y_{k+1},\cdots,y_n)$中减去同样多的量,使得所用的总容量仍是 M。这导致一个新的解 $Z=(z_1,\cdots,z_n)$,其中,$z_i = x_i$, $1 \leq i \leq k$,并且 $\sum_{k<i\leq n} w_i(y_i - z_i) = w_k(z_k - y_k)$。因此,对于 Z 有

$$\begin{aligned}\sum_{1\leq i\leq n} p_i z_i &= \sum_{1\leq i\leq n} p_i y_i + (z_k - y_k) w_k p_k/w_k - \sum_{k<i\leq n} (y_i - z_i) w_i p_i/w_i \\ &\geq \sum_{1\leq i\leq n} p_i y_i + \left[(z_k - y_k) w_k - \sum_{k<i\leq n} (y_i - z_i) w_i\right] p_i/w_k \\ &= \sum_{1\leq i\leq n} p_i y_i\end{aligned}$$

如果 $\sum p_i z_i > \sum p_i y_i$,则 Y 不可能是最优解。如果这两个和数相等,同时 Z = X,则 X 就是最优解;若 $Z \neq X$,则需重复上面的讨论,或者证明 Y 不是最优解,或者把 Y 转换成 X,从而证明了 X 也是最优解。证毕。

5.3 带有限期的作业排序

在这一节将应用贪心设计策略来解决操作系统中单机、无资源约束且每个作业可在等量的时间内完成的作业调度问题。即,假定只能在一台机器上处理 n 个作业,每个作业均可在单位时间内完成;又假定每个作业 i 都有一个截止期限 $d_i > 0$(它是整数),当且仅当作业 i 在它的期限截止以前被完成时,方可获得 $p_j > 0$ 的效益。这个问题的一个可行解是这 n 个作业的一个子集合 J,J 中的每个作业都能在各自的截止期限之前完成。可行解的效益值是 J 中这些作业的效益之和,即 $\sum_{i\in J} p$。具有最大效益值的可行解就是最优解。

例 5.2　设 $n=4,(p_1,p_2,p_3,p_4)=(100,10,15,20)$ 和 $(d_1,d_2,d_3,d_4)=(2,1,2,1)$，则这个问题的可行解和它们的效益值如表 5.1 所示。其中，解⑦是最优的，所允许的处理次序是：先处理作业 4，再处理作业 1。于是，在时间 0 开始处理作业 4 而在时间 2 完成对作业 1 的处理。

表 5.1　例 5.2 的可行解与效益值

	可行解	处理顺序	效益值
①	(1)	1	100
②	(2)	2	10
③	(3)	3	15
④	(4)	4	20
⑤	(1,2)	2,1	110
⑥	(1,3)	1,3 或 3,1	115
⑦	(1,4)	4,1	120
⑧	(2,3)	2,3	25
⑨	(3,4)	4,3	35

5.3.1　带有限期的作业排序算法

为了拟定出一个最优解的算法，应制定如何选择下一个作业的量度标准，利用贪心策略，使得所选择的下一个作业在这种量度下达到最优。不妨首先把目标函数 $\sum_{i\in J}p_i$ 作为量度。使用这一量度，下一个要计入的作业将是在满足所产生的 J 是一个可行解的限制条件下让 $\sum_{i\in J}p_i$ 得到最大增加的作业。这就要求按 p_i 的非增次序来考虑这些作业。利用例 5.2 中的数据来应用这一准则，开始时 $J=$　，$\sum_{i\in J}p_i=0$，由于作业 1 有最大效益且 $J=\{1\}$ 是一个可行解，于是把作业 1 计入 J。下一步考虑作业 4，$J=\{1,4\}$ 也是可行解。然后，考虑作业 3，因为 $\{1,3,4\}$ 不是可行解，故作业 3 被舍弃。最后，考虑作业 2，由于 $\{1,2,4\}$ 也不可行，作业 2 被舍弃。最终所留下的是效益值为 120 的解 $J=\{1,4\}$。它是这个问题的最优解。

现在，对上面所叙述的贪心方法以算法 5.3 的形式给出其粗略的描述。

算法 5.3　作业排序算法的概略描述

```
procedure GREEDY-JOB(D,J,n)
    // 作业按 p1 ≥ p2 ≥ … ≥ pn 的次序输入，它们的期限值 D(i)≥1,1≤i≤n,n≥1。J 是在它们
       的截止期限前完成的作业的集合 //
1       J←{1}
2       for i←2 to n do
3           if J∪{i}的所有作业都能在它们的截止期限前完成
                then J←J∪{i}
4           endif
5       repeat
6       end GREEDY-JOB
```

定理 5.2　对于作业排序问题用算法 5.3 所描述的贪心方法总是得到一个最优解。

证明　设 $(p_i,d_i),1\le i\le n$，是作业排序问题的任一实例；J 是由贪心方法所选择的作业的集合；I 是一个最优解的作业集合。可证明 J 和 I 具有相同的效益值，从而 J 也是最优的。假定 $I\neq J$，因为若 $I=J$，则 J 即为最优解。容易看出，如果 I　J，则 I 就不可能是最优的。由于贪心法的工作方式也排斥了 J　I，因此至少存在着这样的两个作业 a 和 b，使得 $a\in J$ 且 $a\mid I$，$b\in I$ 且 $b\mid J$。设 a 是使得 $a\in J$ 且 $a\mid I$ 的一个具有最高效益值的作业。由于贪心方法可以得出，对于在 I 中又不在 J 中的所有作业 b，都有 $p_a\ge p_b$。这是因为若 $p_b>p_a$，则贪心方法会先于作业 a 考虑作业 b 并且把 b 计入到 J 中去。

现在，设 S_J 和 S_I 分别是 J 和 I 的可行调度表。设 i 是既属于 J 又属于 I 的一个作业；又

设 i 在 S_J 中在 t 到 t + 1 时刻被调度,而在 S_I 中则在 t′到 t′+ 1 时刻被调度。如果 t < t′,则可以将 S_I 中[t′,t′+ 1]时刻所调度的那个作业(如果有的话)与 i 相交换。如果在[t′,t′+ 1]时刻 J 中没有作业被调度,则将 i 移到[t′,t′+ 1]时刻调度,所形成的这个调度表也是可行的。如果 t′< t,则可在 S_I 中作类似的调换。用这种方法,就能得到调度表 S_J′和 S_I′,使 J 和 I 中共有的所有作业在相同的时间被调度。考虑在[t_a,t_a + 1]时刻内 S_J′中(上面所定义的)作业 a 被调度,设作业 b 在 S_I′中的这一时刻被调度,根据对 a 的选择,$p_a \geq p_b$。在 S_I′的[t_a,t_a + 1]时刻去掉作业 b 而去调度作业 a,这就给出了一张关于作业集合 I′= I - {b}∪{a}可行的调度表。显然,I′的效益值不小于 I 的效益值,并且 I′比 I 少一个与 J 不同的作业。

重复应用上述转换,使 I 在不减效益值的情况下转换成 J,因此 J 也必定是最优解。证毕。

为了便于具体实现算法 5 .3,考虑对于一个给定的 J,如何确定它是否为可行解的问题,一个明显的方法是检验 J 中作业的所有可能的排列,对于任一种次序排列的作业序列,判断这些作业是否能在其限期前完成。假定 J 中有 k 个作业,这就需要检查 k ! 个排列。对于所给出的一个排列 σ= $i_1 i_2 i_3 \cdots i_k$,由于完成作业 i_j (1≤j≤k)的最早时间是 j,因此,只要判断出 σ 排列中的每个作业的 $d_{ij} \geq j$,就可得知 σ 是一个允许的调度序列,从而 J 就是一个可行解。反之,如果 σ 排列中有一个 $d_{ij} < j$,则 σ 是一个行不通的调度序列,因为至少作业 i_i 不会在它的限期前完成,故必须去检查 J 的另外形式的排列。事实上,对于 J 的可行性可以通过只检验 J 中作业的一种特殊的排列来确定,这个排列是按期限的非降次序来完成的。

定理 5 .3 设 J 是 k 个作业的集合,σ= $i_1 i_2 \cdots i_k$ 是 J 中作业的一种排列,它使得 $d_{i1} \leq d_{i2} \leq \cdots \leq d_{ik}$。J 是一个可行解,当且仅当 J 中的作业可以按照 σ 的次序而又不违反任何一个期限的情况来处理。

证明 显然,如果 J 中的作业可以按照 σ 的次序而又不违反任何一个期限的情况来处理,则 J 就是一个可行解。若 J 是可行解,则 σ 表示可以处理这些作业的一种允许的调度序列。由于 J 可行,则必存在 σ′= $r_1 r_2 \cdots r_k$,使得 $d_{rj} \geq j, 1 \leq j \leq k$。假设 σ′≠σ,那么,令 a 是使得 $r_a \neq i_a$ 的最小下标。设 $r_b = i_a$,显然 b > a。在 σ′中将 r_a 与 r_b 相交换,因为 $d_{ra} \geq d_{rb}$,故作业可依新产生的排列 σ″= $s_1 s_2 \cdots s_k$ 的次序处理而不违反任何一个期限。连续使用这一方法,就可将 σ′转换成 σ 且不违反任何一个期限。定理得证。

即使这些作业有不同的处理时间 $t_i \geq 0$,上述定理亦真。其证明留作习题。

根据定理 5 .3,可将带限期的作业排序问题作如下处理:首先将作业 1 存入解数组 J 中,然后处理作业 2 到作业 n。假设已处理了 i - 1 个作业,其中有 k 个作业已计入 J(1),J(2),…,J(k)之中,且有 D(J(1))≤D(J(2))≤…≤D(J(k)),现在处理作业 i。为了判断 JU{i}是否可行,只需看能否找出按期限的非降次序插入作业 i 的适当位置,使得作业 i 在此处插入后有 D(J(r))≥r,1≤r≤k + 1。找作业 i 可能的插入位置可如下进行:将 D(J(k)),D(J(k - 1)),…,D(J(1))逐个与 D(i)比较,直到找到位置 q,它使得 D(i) < D(J(1));q < 1≤k 且 D(J(q))≤D(i)。此时,若 D(J(1)) > 1,q < 1≤k,则说明这 k - q 个作业均可延迟一个单位时间处理,即可将这些作业在 J 中均后移一个位置而不超过各自的期限值。在以上条件成立的情况下,只要 D(i) > q,就可将作业 i 在位置 q + 1 处插入,从而得到一个按期限的非降次序排列的含有 k + 1 个作业的可行解。以上过程可反复进行到第 n 个作业处理完毕,所得到的贪心解由定理 5 .2 可知就是一个最优解。这一处理过程可用算法 5 .3 来描述。算法中

引进了一个虚构的作业 0,它放在 J(0),且 D(J(0)) = 0。引入这一虚构作业是为了便于将作业插入位置 1。

算法 5.4　带有限期和效益的单位时间的作业排序贪心算法

```
line procedure JS(D,J,n,k)
      //D(1),…,D(n)是期限值,n≥1,作业已按 p1≥p2≥…pn 被排序。J(i)是最优解中的第 i
        个作业,1≤i≤k。终止时,D(J(i))≤D(J(i+1)),1≤i<k//
1       integer D(0:n),J(0:n),i,k,n,r
2       D(0)←J(0)←0   //初始化//
3       k←1;J(1)←1   //计入作业 1//
4       for i←2 to n do   //按 p 的非增次序考虑作业。找 i 的位置并检查插入的可行性//
5       r←k
6       while D(J(r)) > D(i) and D(J(r)) ≠ r do
7       r←r - 1
8       repeat
9       if D(J(r))≤D(i) and D(i) > r then
        //把 i 插入 J//
10            for i←k to r+1 by - 1 do
11                  J(i+1)←J(1)
12            repeat
13            J(r+1)←i;k←k+1
14          endif
15        repeat
16        end JS
```

对于 JS,有两个赖之以测量其复杂度的参数,即作业数 n 和包含在解中的作业数 s。第 6～8 行的循环至多迭代 k 次,每次迭代的时间为 O(1)。若第 9 行的条件为真,则执行 10～13 行。这些要求 O(k - r)时间去插入作业 i。因此,对于 4～15 行的循环,其每次迭代的总时间是 O(k)。该循环共迭代 n - 1 次。如果 s 是 k 的终值,即 s 是最后所得解的作业数,则算法 JS 所需要的总时间是 O(sn)。由于 s≤n,因此 JS 的最坏情况时间是 $O(n^2)$。这种情况在 $p_i = d_i = n - i + 1, 1 \le i \le n$ 时就会出现。进而可以证明最坏情况计算时间为 $\Theta(n^2)$。在计算空间方面,除了 D 所需要的空间外,为了存放解 J,还需要 Θ(s)的空间量。要指出的是,JS 并不需要具体的效益值,只要知道 $p_i \ge p_{i+1}, 1 \le i \le n$ 即可。

5.3.2　一种更快的作业排序算法

通过使用不相交集合的 UNION 与 FIND 算法(参见 2.4.3 小节)以及使用一个不同的方法来确定部分解的可行性,可以把 JS 的计算时间由 $O(n^2)$ 降低到数量级相当接近于 O(n)。如果 J 是作业的可行子集,那么可以使用下述规则来确定这些作业中的每一个作业的处理时间:若还没给作业 i 分配处理时间,则分配给它时间片[α - 1,α],其中 α 应尽量取大且时间片[α - 1,α]是空的。此规则就是尽可能推迟对作业 i 的处理。于是,在将作业一个一个地装配到 J 中时,就不必为接纳新作业而去移动 J 中那些已分配了时间片的作业。如果正被考虑的新作业不存在像上面那样定义的 α,这个作业就不能计入 J。这样处理的正确性

证明留作习题。

例 5.3 设 $n=5,(p_1,\cdots,p_5)=(20,15,10,5,1)$ 和 $(d_1,\cdots,d_5)=(2,2,1,3,3)$。使用上述可行性规则,得

J	已分配的时间片	正被考虑的作业	动作
	无	1	分配[1,2]
{1}	[1,2]	2	分配[0,1]
{1,2}	[0,1],[1,2]	3	不适合,舍弃
{1,2}	[0,1],[1,2]	4	分配[2,3]
{1,2,4}	[0,1],[1,2],[2,3]	5	舍弃

最优解是 $J=\{1,2,4\}$。

由于只有 n 个作业且每个作业花费一个单位时间,因此只需考虑这样一些时间片 $[i-1,i],1\leq i\leq b$,其中 $b=\min\{n,\max\{d_j\}\}$。为简便起见,用 i 来表示时间片 $[i-1,i]$。易于看出,这 n 个作业的期限值只能是 $\{1,2,\cdots,b\}$ 中的某些(或全部)元素。实现上述调度规则的一种方法是把这 b 个期限值分成一些集合。对于任一期限值 i,设 n_i 是使得 $n_j\leq i$ 的最大整数且是空的时间片。为避免极端情况,引进一个虚构的期限值 0 和时间片 $[-1,0]$。当且仅当 $n_j=n_j$,期限值 i 和 j 在同一个集合中,即所要处理的作业的期限值如果是 i 或 j,则当前可分配的最接近的时间片是 n_i。显然,若 $i<j$,则 $i,i+1,i+2,\cdots,j$ 都在同一个集合中。因此,上述方法就是作出一些以期限值为元素的集合,且使同一集合中的元素都有一个当前共同可用的最接近的空时间片。对于每个期限值 $i,0\leq i\leq b$,当前最接近的空时间片可用线性表元素 F(i)表示,即 $F(i)=n_i$。使用集合表示法,把每个集合表示成一棵树。根结点就认为是这个集合。最初,这 $b+1$ 个期限值最接近的空时间片是 $F(i)=i,0\leq i\leq b$,并且有 $b+1$ 个集合与 $b+1$ 个期限值相对应。用 P(i)把期限值 i 链接到它的集合树上。开始时 $P(i)=-1,0\leq i\leq b$。如果要调度具有期限 d 的作业,就需要去寻找包含期限值 $\min\{n,d\}$ 的那个根。如果这个根是 j 且只要 $F(j)\neq 0$,则F(j)是最接近的空时间片。在使用了这一时间片后,其根为 j 的集合应与包含期限值 F(j)-1 的集合合并。

过程 FJS 描述了这个更快的算法。易于看出它的计算时间是 $O(n\alpha(2n,n))$(参看2.4.3中 Ackermann 函数的逆函数)。它用于 F 和 P 的空间至多为 2n 个字节。

算法 5.5 作业排序的一个更快算法

```
line procedure FJS(D,n,b,J,k)
     //找最优解 J=J(1),…,J(k)//
     //假定 p1≥p2≥…≥pn,b=min{n,max{D(i)}}//
1    integer b,D(n),J(n),F(0:b),P(0:b)
2    for i←0 to n do   //将树置初值//
3         F(i)←i;P(i)←-1
4    repeat
5    k←0   //初始化 J//
6    for i←1 to n do   //使用贪心规则//
7         j←FIND(min(n,D(i)))
8         if F(j)≠0 then k←k+1;J(k)←i   //选择作业 i//
9                   l←FIND(F(j)-1);call UNION(l,j)
```

```
10                    F(j)←F(l)
11              endif
12        repeat
13        end FJS
```

例 5.4　设 $n=7,(p_1,\cdots,p_7)=(35,30,25,20,15,10,5)$**和**$(d_1,\cdots,d_7)=(4,2,4,3,4,8,3)$**。利用算法 FJS 求解上述作业排序问题的最优解。**

快速作业调度如图 5.1 所示。由图 5.1 可得最优解 $J=\{1,2,3,4,6\}$。

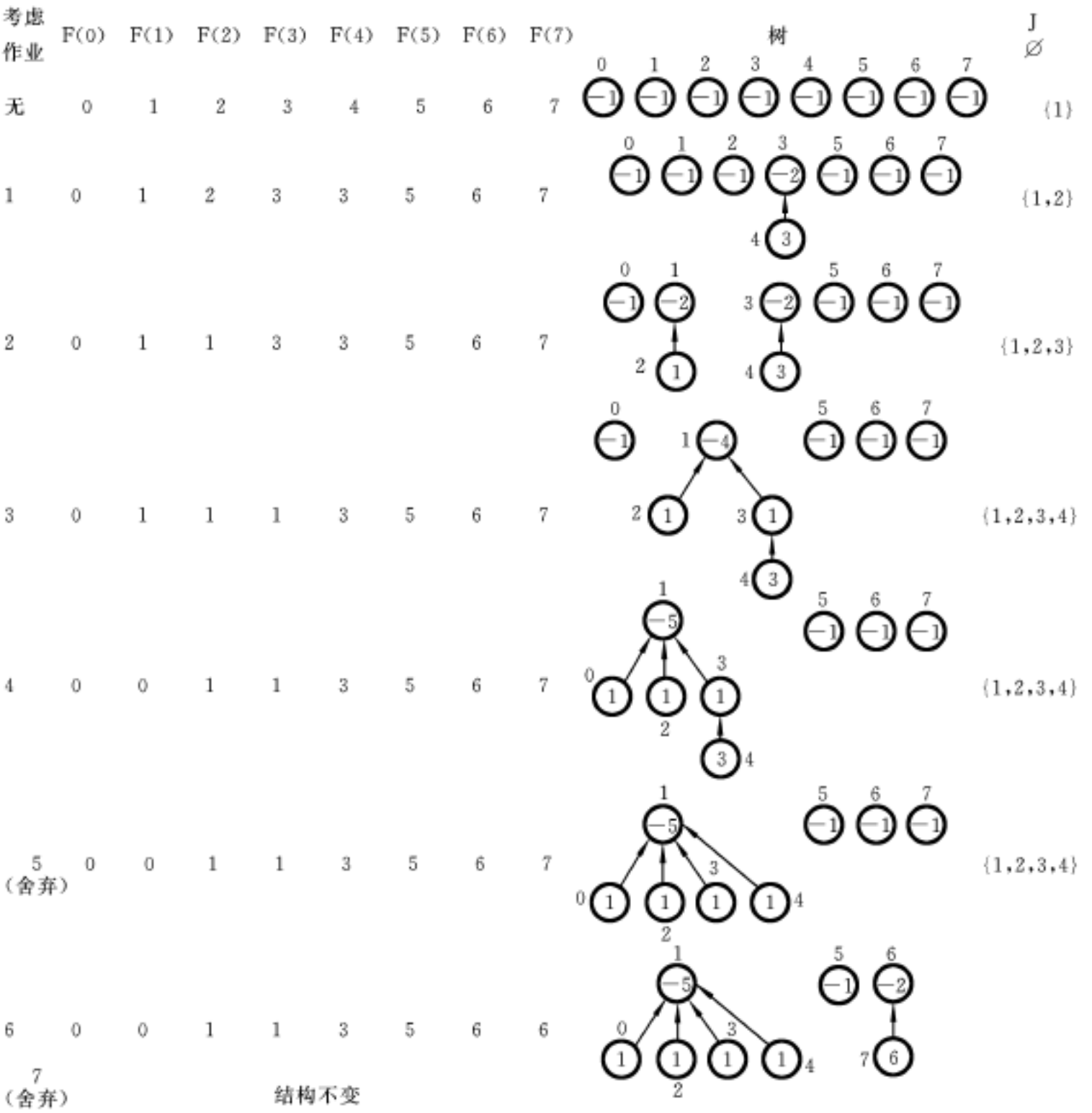


图 5.1　快速作业调度

5.4　最优归并模式

在 4.4 节已看到两个分别包含 n 个和 m 个记录的已分类文件可以在 O(n + m)时间内归并在一起而得到一个分类文件。当要把两个以上的已分类文件归并在一起时,可以通过

成对地重复归并已分类的文件来完成。例如,假定 X_1 , X_2 , X_3 , X_4 是要归并的文件,则可以首先把 X_1 和 X_2 归并成文件 Y_1 ,然后将 Y_1 和 X_3 归并成 Y_2 ,最后将 Y_2 和 X_4 归并,从而得到想要的分类文件;也可以先把 X_1 和 X_2 归并成 Y_1 ,然后把 X_3 和 X_4 归并成 Y_2 ,最后归并 Y_1 和 Y_2 而得到想要的分类文件。给出 n 个文件,则有许多种把这些文件成对地归并成一个单一分类文件的方法。不同的配对法要求不同的计算时间。现在所要论及的问题是确定一个把 n 个已分类文件归并在一起的最优方法(即需要最少比较的方法)。

例 5 .5　X_1 ,X_2 和 X_3 是分别有 30 个记录、20 个记录和 10 个记录的 3 个已分类文件,归并 X_1 和 X_2 需要 50 次记录移动,再与 X_3 归并则还要 60 次移动,其所需要的记录移动总量是 110。如果首先归并 X_2 和 X_3 (需要 30 次移动),然后归并 X_1 (需要 60 次移动),则所要作的记录移动总数仅为 90。因此,第二个归并模式比第一个要快些。

试图得到最优归并模式的贪心方法是容易表达的。由于归并一个具有 n 个记录的文件和一个具有 m 个记录的文件可能需要 n + m 次记录移动,因此对于量度标准的一种明显的选择是:每一步都归并尺寸最小的两个文件。例如,有 5 个文件(F_1 ,…, F_5),它们的尺寸为(20,30,10,5,30),由以上的贪心策略就会产生以下的归并模式: F_4 和 F_3 归并成 Z_1 ($|Z_1| = 15$);归并 Z_1 和 F_1 得到 Z_2 ($|Z_2| = 35$);把 F_2 和 F_5 归并成 Z_3 ($|Z_3| = 60$);归并 Z_2 和 Z_3 而得到答案 Z_4 。记录移动总量是 205。可以证明这是给定问题实例的最优归并模式。

像刚才所描述的归并模式称为二路归并模式(每一个归并步包含两个文件的归并)。二路归并模式可以用二元归并树来表示。图 5 .2 显示了一棵表示上面 5 个文件所得到的最优归并模式的二元归并树。叶结点被画成方块形,表示这 5 个已知的文件。这些结点称为外部结点。剩下的结点被画成圆圈,称为内部结点。每个内部结点恰好有两个儿子,它表示把它的两个儿子所表示的文件归并而得到的文件。每个结点中的数字都是那个结点所表示的文件的长度(即记录数)。

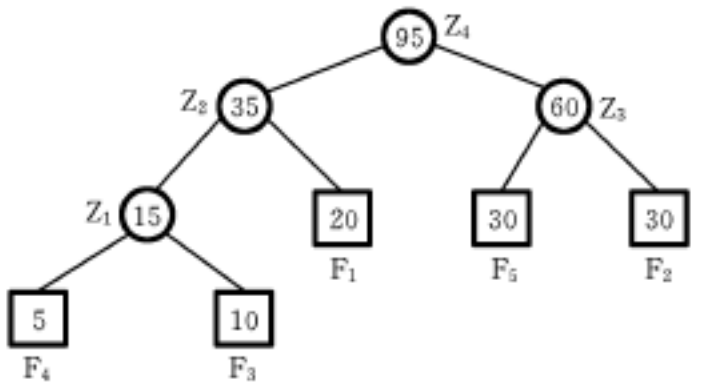


图 5 .2　表示一个归并模式的二元归并树

外部结点 F_4 在距离根结点 Z_4 为 3 的地方(一个 i 级结点在距离根为 i - 1 的地方),因此文件 F_4 的记录都要移动 3 次,一次得到 Z_1 ,一次得到 Z_2 ,最后移动一次就得到 Z_4 。如果 d_i 是由根到代表文件 F_i 的外部结点的距离, q_i 是 F_j 的长度,则这棵二元归并树的记录移动总量是 $\sum_{i=1}^{n} d_i q_i$ 。这个和数叫做这棵树的带权外部路径长度。

一个最优二路归并模式与一棵具有最小权外部路径的二元树相对应,算法 5 .6 的过程 TREE 使用上面所叙述的规则去获得 n 个文件的二元归并树。这算法把 n 个树的表 L 作为输入。树中的每一个结点有 3 个信息段, LCHILD, RCHILD 和 WEIGHT。起初, L 中的每一棵树正好有一个结点。这个结点是一个外部结点,且其 LCHILD 和 RCHILD 信息段为 0,而 WEIGHT 是要归并的 n 个文件之一的长度。在这个算法运行期间,对于 L 中的任何一棵具有根结点 T 的树, WEIGHT(T)表示要归并的文件的长度(WEIGHT(T)等于树 T 中外部结点的长度的和)。过程 TREE 用了 3 个子算法, GETNODE(T), LEAST 和 INSERT(L, T)。

子算法 GETNODE(T)为构造这棵树提供一个新结点。LEAST(L)找出 L 中一棵其根具有最小的 WEIGHT 的树,并把这棵树从 L 中删去。INSERT(L,T)把根为 T 的树插入到表 L 中。定理5 .4将证明贪心过程 TREE(算法 5 .6)产生一棵最优的二元归并树。

算法 5 .6　生成二元归并树算法

```
line procedure TREE(L,n)
     // L 是如上所述的 n 个单结点二元树的表 //
1    for i←1 to n - 1 do
2        call GETNODE(T)   // 用于归并两棵树 //
3        LCHILD(T)←LEAST(L)   // 最小的长度 //
4        RCHILD(T)←LEAST(L)
5        WEIGHT(T)←WEIGHT(LCHILD(T)) + WEIGHT(RCHILD(T))
6        call INSERT(L,T)
7    repeat
8    return(LEAST(L))
     // 留在 L 中的树是归并树 //
9    end TREE
```

例 5 .6　当 L 最初表示长度为 2,3,5,7,9,13 六个文件时,算法 TREE 是如何工作的。图 5 .3 显示出在 for 循环的每一次迭代结束时的表 L。在算法结束时所产生的二元归并树可以用来确定归并了哪些文件。归并是在这棵树中“最低”(有最大的深度)的那些文件上进行的。

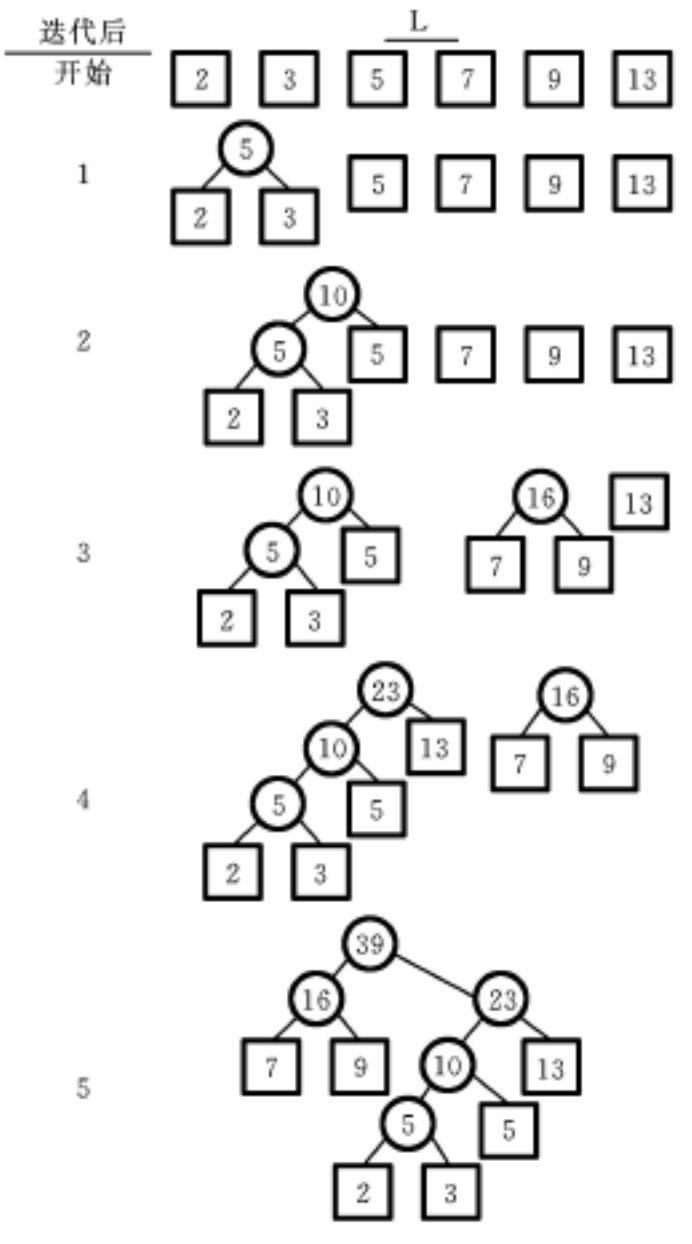


图 5 .3　例 5 .6 过程 TREE 的表 L 中的树

现在来分析算法 5 .6 需要的计算时间。主循环执行 n - 1 次。如果保持 L 按照这些根中的 WEIGHT 值的非降次序,则 LEAST(L)只需要 O(1)时间,INSERT(L,T)在 O(n)时间内被执行。因此,所花费的时间总量是 $O(n^2)$。在 L 被表示成一个 min-堆的情况下,其中根的值不超过它的儿子们的值(见 2 .4 .2 节),则 LEAST(L)和 INSERT(L,T)可在 O(logn)时间内完成(LEAST(L)和 INSERT(L,T)的算法以及其计算时间分析留作习题)。在这种情况下 TREE 的计算时间是 O(nlogn)。将第 6 行的 INSERT 和第 4 行的 LEAST 结合起来还可加快一些速度。

定理 5 .4　若 L 是最初包含 $n \geq 1$ 个单个结点的树,这些树有 WEIGHT 值为 $(q_1, q_2, \cdots, q_n)$,则算法 TREE 对于具有这些长度的 n 个文件生成一棵最优的二元归并树。

证明　通过施归纳于 n 来证明。对于 n = 1,返回一棵没有内部结点的树且这棵树显然是最

优的。假定该算法对于所有的$(q_1, q_2, \cdots, q_m)$, $1 \leq m < n$ 生成一棵最优二元归并树,现在来证明对于所有的$(q_1, q_2, \cdots, q_n)$也生成最优的树。不失一般性,假定 $q_1 \leq q_2 \leq \cdots \leq q_n$,且 q_1 和 q_2 是在 for 循环的第一次迭代期间由第 3 行和第 4 行中的算法 LEAST 所找到的两棵树的 WEIGHT 信息段的值。于是,就生成了图 5.4 的子树 T。设 T′是一棵对于$(q_1, q_2, \cdots, q_n)$的最优二元归并树。设 P 是距离根最远的一个内部结点。如果 P 的儿子不是 q_1 和 q_2,则可以用 q_1 和 q_2 来代换 P 现在的儿子而不增加 T′的带权外部路径长度。因此,T 也是一棵最优归并树中的子树。于是在 T′中如果用其权为 $q_1 + q_2$ 的一个外部结点来代换 T,则所产生的树 T″是关于$(q_1 + q_2, q_3, \cdots, q_n)$的一棵最优归并树。由归纳假设,在用其权为 $q_1 + q_2$ 的那个外部结点代换了 T 以后,过程 TREE 转化成去求取一棵关于$(q_1 + q_2, q_3, \cdots, q_n)$的最优归并树。因此,TREE 生成一棵关于$(q_1, q_2, \cdots, q_n)$的最优归并树。证毕。

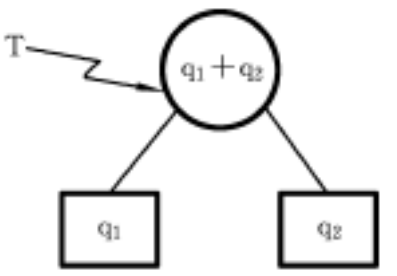


图 5.4 最简单的二元归并树

生成归并树的贪心方法也适用于 k 路归并的情况。在这种情况下,相应的归并树是一棵 k 元树。由于所有的内部结点的度数必须为 k,因此对于 n 的某些值,就不与 k 元归并树相对应。例如,当 k = 3 时,就不存在具有 n = 2 个外部结点的 k 元归并树。所以,有必要引进一定量的“虚”外部结点。每一个虚结点被赋以 0 值的 q_i。这个虚值不会影响所产生的 k 元树的带权外部路径长度。本章习题 5.11 表明了其所有内部结点都具有度数为 k 的 k 元树的存在性,只有当外部结点数 n 满足等式 n mod(k - 1) = 1 时才成立。因此,至多应增加 k - 2 个虚结点。生成最优归并树的贪心规则是:在每一步,选取 k 棵具有最小长度的子树用于归并。关于它的最优性证明,则留作习题。

5.5 最小生成树

定义 设 G = (V, E)是一个无向连通图。如果 G 的生成子图 T = (V, E′)是一棵树,则称 T 是 G 的一棵生成树(spanning tree)。

例 5.7 图 5.5 显示了 4 个结点的完全图以及它的 3 棵生成树。

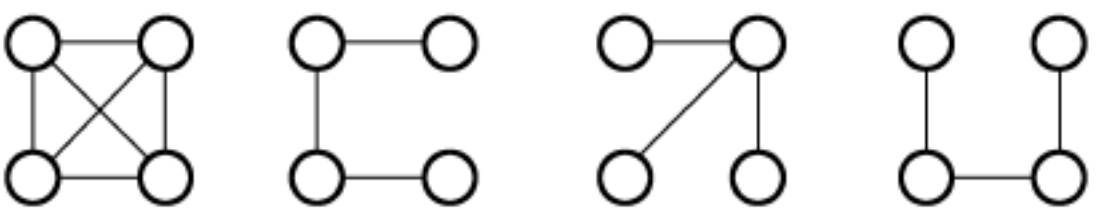


图 5.5 一个无向图和它的 3 棵生成树

应用生成树可以得到关于一个电网的一组独立的回路方程。第一步是要得到这个电网的一棵生成树。设 B 是那些不在生成树中的电网的边的集合,从 B 中取出一条边添加到这生成树上就生成一个环。从 B 中取出不同的边就生成不同的环。把克希霍夫(Kirchoff)第二定律用到每一个环上,就得到一个回路方程。用这种方法所得到的环是独立的(即这些环中没有一个可以用这些剩下的环的线性组合来得到),这是因为每一个环包含一条从 B 中取来的边,而这条边不包含在任何其它的环中。因此,这样所得的这组回路方程也是独立的。

可以证明,通过一次取 B 中的一条边放进所产生的生成树中而得到的这些环组成一个环基,从而这图中所有其它的环都能够用这个基中的这些环的线性组合构造出来。

生成树在其它方面也有广泛的应用。一种重要的应用是由生成树的性质所产生的,这一性质是,生成树是 G 的这样一个最小子图 T,它使得 V(T) = V(G)且 T 连通并具有最少的边数。任何一个具有 n 个结点的连通图都必须至少有 n - 1 条边,而所有具有 n - 1 条边的 n 结点连通图都是树。如果 G 的结点代表城市,边代表连接两个城市的可能的交通线,则连接这 n 个城市所需要的最少交通线是 n - 1 条。G 的那些生成树代表所有可行的选择。

然而,在实际情况下,这些边都有分配给它们的权。这些权可以代表建造的成本、交通线的长度及其它。假定给出这么一个带权图(假定所有的权都是正数),人们就会希望在结构上选择一组交通线,它们连接所有的城市并且有最小的总成本或者有最小的总长度。显然,在这两种情况下所选择的连线都必须构成一棵树。感兴趣的是找出 G 中具有最小成本的生成树。图 5 .6 显示了一个图和它的最小成本生成树中的一棵生成树。

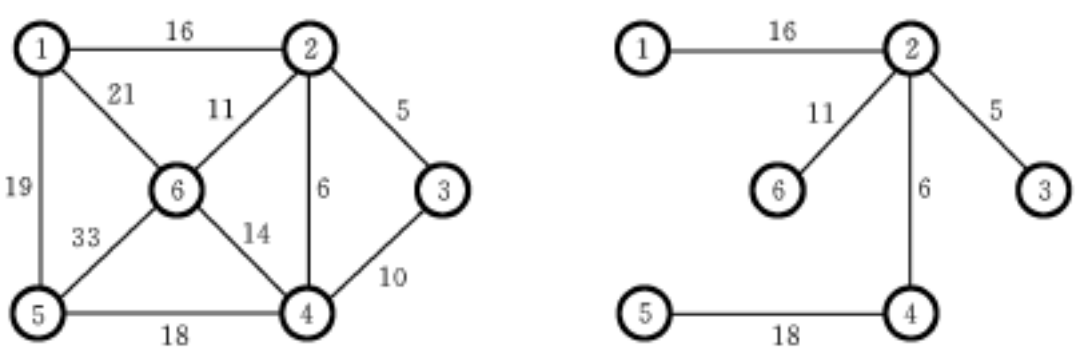


图 5 .6　一个图和它的最小成本生成树中的一棵生成树

获得最小成本生成树的贪心方法应该是一条边一条边地构造这棵树。根据某种量度标准来选择将要计入的下一条边。最简单的量度标准是选择使得迄今为止所计入的那些边的成本的和有最小增量的那条边。有两种可能的方法来解释这一量度标准。第一种方法使得迄今所选择的边的集合 A 构成一棵树,将要计入到 A 中的下一条边(u, v)是一条不在 A 中且使得 A∪{(u, v)}也是一棵树的最小成本的边。这种选择准则产生一棵最小成本生成树的证明留作习题。其相应的算法称为 Prim 算法。

例 5 .8　图 5 .7(b)显示了 Prim 方法在图 5 .7(a)所示图上的运行情况,所得到的生成树有 105 的成本。

由上述可知,Prim 方法是如何运行的,以及使用这一方法来获得求取最小生成树的 SPARKS 算法。该算法是在只计入了 G 中一条最小成本的边的这样一棵树的情况下开始的,然后一条边一条边地加进这棵树中。所要加的下一条边(i, j)是这样的一条边,其 i 是已计入到这棵树中的一个结点,j 是还没有计入的一个结点,而(i, j)的成本 COST(i, j)是所有这样的一些边(k, l)的成本最小值。这些边是其结点 k 在这棵树中,而结点 l 不在这棵树中。为了有效地求出这条边(i, j),把还没计入这树中的每一个结点 j 和值 NEAR(j)联系起来。NEAR(j)是树中的这样一个结点,它使得 COST(j, NEAR(j))是对 NEAR(j)所有选择中的最小值。而对于已经在树中的所有结点 j,定义 NEAR(j) = 0。用使 NEAR(j) ≠ 0(j 还不在树中)且 COST(j, NEAR(j))为最小值的结点来确定要计入的下一条边。

在过程 PRIM(算法 5 .7)中,第 3 行选取一条最小成本边。第 4～10 行给变量置初值以

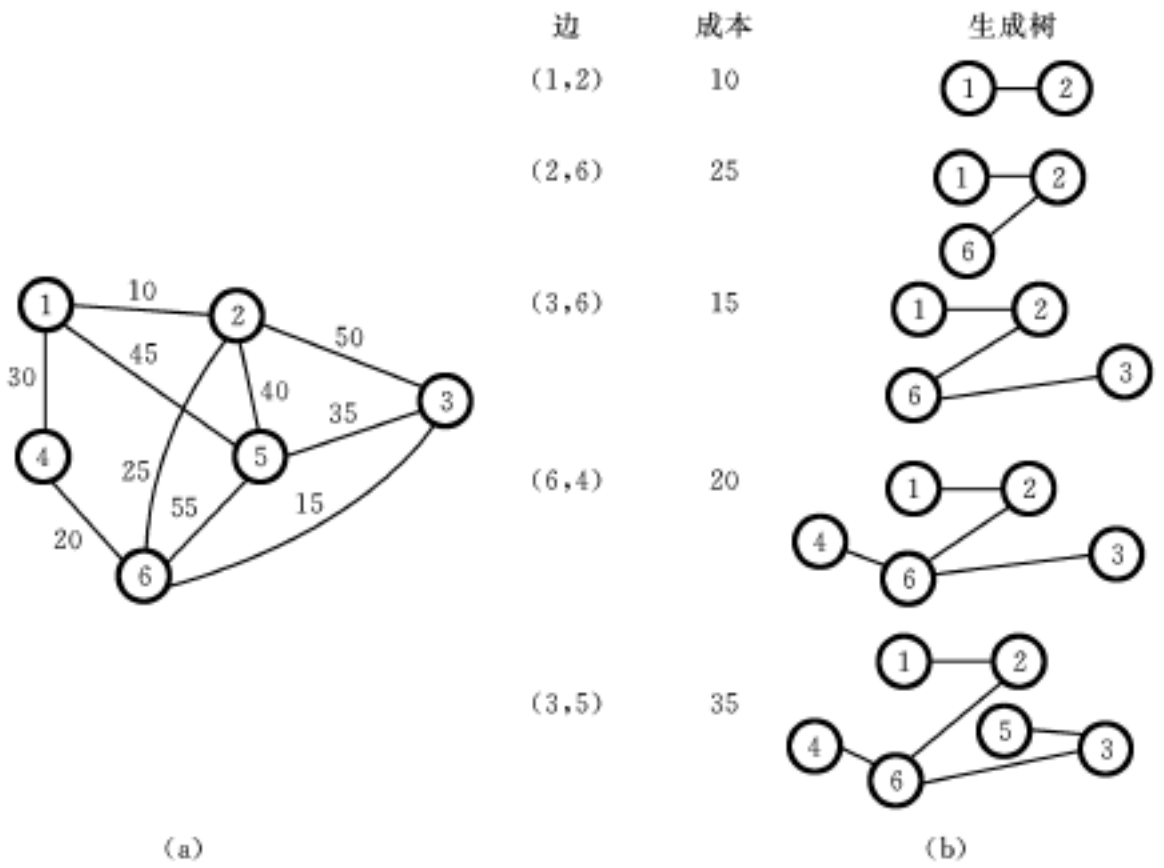


图 5.7　Prim 方法运行实例图

(a) 例 5.8 的图;(b) Prim 算法的步骤

便表示仅包含一条边(k,l)的树。在 11～21 行中,逐条边地构造生成树的剩余部分。第 12 行选取(j, NEAR(j))作为要计入的下一条边。第 16～20 行修改 NEAR(　)。

算法 5.7　Prim 最小生成树算法

```
line procedure PRIM(E,COST,n,T,mincost)
    //E 是 G 的边集。COST(n,n)是 n 结点图 G 的成本邻接矩阵,矩阵元素COST(i,j)是一个
      正实数,如果不存在边(i,j),则为 +∞。计算一棵最小生成树并把它作为一个集合存放到
      数组 T(1：n - 1,2)中(T(i,1),T(i,2))是最小成本生成树的一条边。最小成本生成树的
      总成本最后赋给 mincost//
1     real COST(n,n),mincost;
2     integer NEAR(n),n,i,j,k,l,T(1：n - 1,2);
3     (k,l)←具有最小成本的边
4     mincost←COST(k,l)
5     (T(1,1),T(1,2))←(k,l)
6     for i←1 to n do   //将 NEAR 置初值//
7       if COST(i,l) < COST(i,k) then NEAR(i)←l
8                                else NEAR(i)←k endif
9     repeat
10     NEAR(k)←NEAR(l)←0
11     for i←2 to n - 1 do   //找 T 的其余 n - 2 条边//
12     设 j 是 NEAR(j)≠0 且 COST(j,NEAR(j))最小的下标
13     (T(i,1),T(i,2))←(j,NEAR(j))
14     mincost←mincost + COST(j,NEAR(j))
15     NEAR(j)←0
16       for k←1 to n do   //修改 NEAR//
```

```
17          if NEAR(k)≠0 and COST(k,NEAR(k)) > COST(k,j)
18          then NEAR(k)←j
19        endif
20      repeat
21    repeat
22    if mincost≥∞ then print ('no spanning tree')endif
23    end PRIM
```

过程 PRIM 所需要的时间是 $\Theta(n^2)$,其中 n 是图 G 的结点数。具体分析如下:第 3 行花费$\Theta(e)$ $(e = |E|)$时间而第 4 行花费 $\Theta(1)$时间;第 6～9 行的循环花费 $\Theta(n)$时间;第 12 行和第 16～20 行的循环要求 $\Theta(n)$时间,故第 11～21 行循环的每一次迭代要花费 $\Theta(n)$时间。所以,这个循环的总时间是 $\Theta(n^2)$。因此,过程 PRIM 有 $\Theta(n^2)$的时间复杂度。

因为最小生成树包含了与每个结点 v 相关的一条最小成本边,所以还可以把这算法稍许加快一点。为此,假设 T 是图 G = (V, E)的最小成本生成树。设 v 是 T 中的任一结点。又设(v, w)是所有与 v 相关的边中具有最小成本的一条边。假定(v, w)| E(T)并且对于所有的边(v, x)∈E(T), COST(v, w) < COST(v, x)。把(v, w)包含到 T 中就生成一个唯一的环。这个环必定包括一条边(v, x), x≠w。从 E(T)∪{(v, w)}中去掉边(v, x)就破坏了这个环而且没有使图(V, E(T)∪{(v, w)} - {(v, x)})不连通,因此该图也是一棵生成树。由于 COST(v, w) < COST(v, x),所以这棵生成树比 T 有更小的成本。这与 T 是 G 中最小成本生成树相矛盾,故 T 包含如上所述的最小成本边。

实际上可以从一棵包含任何一个随意指定的结点而没有边的树开始这一算法,然后再逐条增加边。这需要改变第 3～11 行。它们可由下面这些行来代换。

```
3′  mincost←0
4′  for i←2 to n do   // 结点 1 开始就在 T 中 //
5′        NEAR(i)←1
6′  repeat
7′  NEAR(1)←0
8′～11′  for i←1 to n - 1 do
```

其总的复杂度仍然是 $\Theta(n^2)$。

对于前面所说到的量度标准有第二种解释,即图的边按成本的非降次序来考虑。这种解释是对于生成树迄今所选择的边集合 T,应使得它完成后的 T 有可能成为一棵树。因此,在算法的所有阶段 T 可以不是一棵树。当且仅当 T 中不存在环,边的集合 T 就可以最终被完成为一棵树,因此,一般说来 T 将只是一个森林。在定理 5.5 中将证明贪心方法的这样一种解释也生成一棵最小成本生成树。这一方法是 Kruskal 给出的。

例 5.9　考虑图 5.7(a)中的那个图。使用 Kruskal 方法,为了得到最小成本生成树的组成元素,按以下次序(1, 2), (3, 6), (4, 6), (2, 6), (1, 4), (3, 5), (2, 5), (1, 5), (2, 3)和(5, 6)来考虑这个图的边。其相应的成本序列是 10, 15, 20, 25, 30, 35, 40, 45, 50, 55。前 4 条边包含在 T 中。接着要考虑的边是(1, 4),这条边连接了两个在 T 中已经连通的结点,故舍弃。下一次选取边(3, 5),就完全生成了这棵生成树。图 5.8 显示了在进行这一计算的各个阶段得到的森林。最后得到的生成树有 105 的成本。

为清楚起见,Kruskal 算法在算法 5 .8 中被形式化地写出。最初,E 是 G 中所有边的集合,要在这集合上执行的运算是:① 确定具有最小成本的边(第 3 行);② 删去这条边(第 4 行)。如果 E 中的边能像一个已分类的顺序表那样被保存,则这两种运算都能有效地运行。实际上,只要能够便于确定具有最小成本的边(第 3 行),则对所有的边进行分类并不是非要不可的。如果这些边作为一个 min-堆来保存,则下一条要考虑的边就可在 O(loge)时间内得到。假定 G 有 e 条边,构造这个堆本身要花费 O(e)时间。

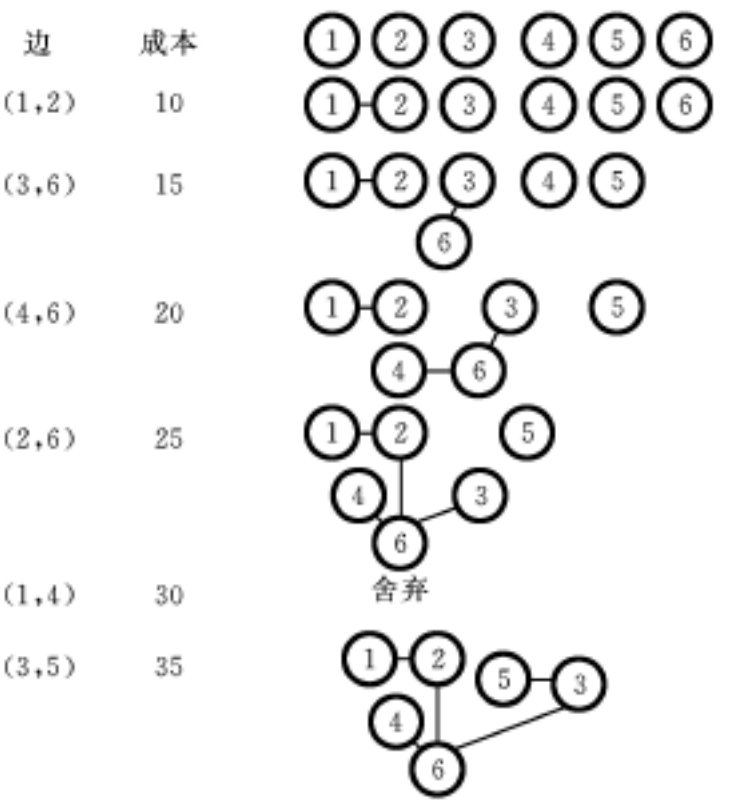


图 5 .8 Kruskal 算法中的各阶段

算法 5 .8 Kruskal 最小生成树算法的概略描述

```
1  T←
2  while T 的边少于 n - 1 条 do
3        从 E 中选取一条最小成本的边(v,w)
4        从 E 中删去(v,w)
5        if(v,w)在 T 中不生成环
6        then 将(v,w)加到 T
7        else 舍弃(v,w)
8        endif
9  repeat
```

为了有效地执行第 5 步和第 6 步,G 中的结点的组合方式应该是易于确定结点 v 和 w 是否已由早先选择的边所连通的那种。在已连通的情况下,将边(v,w)舍弃;若未连通,则把(v,w)加入到 T。一种可能的组合方法是把 T 的同一连通分图中所有结点放到一个集合中(T 的各个连通分图都是树)。那么,T 中的两个结点是连通的,当且仅当它们在同一个集合中。例如,当要考虑边(2,6)时,这些集合就是{1,2},{3,4,6}和{5}。由于结点 2 和 6 在不同的集合中,因此这些集合被合并成为{1,2,3,4,6}和{5}。要考虑的下一条边是(1,4)。由于结点 1 和 4 在同一个集合中,因此该边被舍弃,边(3 5)连接不同集合中的结点,并且产生最终的生成树。使用 2 .4 .3 节的集合表示和该节中的 UNION 和 FIND 算法,可以在几乎是线性的时间内有效地实现第 5 行和第 6 行。因此,计算时间由第 3 行和第 4 行的时间所确定,在最坏情况下第 3 行和第 4 行的计算时间是O(e loge)。

如果使用上述的表示,结果就会形成算法 5 .9 的这个过程。在第 3 行构成边的最初的一个堆,在第 4 行每一个结点被分配到一个不同的集合中(从而也是分配到一棵树中)。T 是包含在最小成本生成树中的边的集合,而 i 是 T 中的边的数目。T 可以用二维数组 T(1∶n - 1,2)表示成一个顺序表。可以用赋值语句 T(i,1)←u 和 T(i,2)←v 把边(u,v)加到 T 中去。在第 6～14 行的循环中,这些边按成本非降的次序从堆中一条一条地去掉。第 8 行求包含 u 和 v 的集合。如果 j≠k,则结点 u 和 v 在不同的集合中(从而也在不同的树中),边(u,v)就计入 T。包含 u 和 v 的集合被合并(第 12 行)。如果 j = k,则边(u,v)被舍弃,因为把它计入 T 会生成一个环。第 15 行确定是否已找到了一棵生成树。由此可知,当

且仅当图 G 是不连通的,i≠n - 1。可以证明此算法的计算时间是 O(eloge),其中 e 是 G 中的边数(e = |E|)。

算法 5.9　Kruskal 算法

```
line procedure KRUSKAL(E,COST,n,T,mincost)
    //G 有 n 个结点,E 是 G 的边集。COST(u,v)是边(u,v)的成本。T 是最小成本生成树的边
      集,mincost 是它的成本//
1     real mincost,COST(1:n,1:n)
2     integer PARENT(1:n),T(1:n - 1,2),n
3     以边成本为元素构造一个 min-堆
4     PARENT←- 1   //每个结点都在不同的集合中//
5     i←mincost←0
6     while i < n - 1 and 堆非空 do
7     从堆中删去最小成本边(u,v)并重新构造堆
8     j←FIND(u);k←FIND(v)
9     if j≠k then i←i+ 1
10        T(i,1)←u;T(i,2)←v
11        mincost←mincost + COST(u,v)
12        call UNION(j,k)
13    endif
14    repeat
15    if i≠n - 1 then print ('no spanning tree') endif
16        return
17    end KRUSKAL
```

引理 5.1　设 T 是无向连通图 G 的一棵生成树。对于任一条边 e∈E(G)而 e| E(T),有:①若将 e 加入到 T,则生成一个唯一的环;②从 E(T)∪{e}中去掉这环中的任意一条边后,剩余的边构成 G 的一棵树。

证明留作习题。

定理 5.5　Kruskal 算法对于每一个无向连通图 G 产生一棵最小成本生成树。

证明　设 G 是任一无向连通图,又设 T 是用 Kruskal 算法产生的 G 的生成树,再设 T′是 G 的最小成本生成树。现要证明 T 和 T′具有相同的成本。

设 E(T)和 E(T′)分别是 T 和 T′中的边集。若 n 是 G 中的结点数,则 T 和 T′都有 n - 1 条边。若 E(T) = E(T′),则 T 显然就是最小成本生成树。若 E(T)≠E(T′),则假设 e 是一条使得 e∈E(T)和 e| E(T′)的最小成本的边。显然,这样的 e 必定存在。由引理 5.1①,把 e 加入到 T′就产生一个唯一的环。假设 $e, e_1, e_2, \cdots, e_k$ 是这个唯一的环。这些边 $e_i, 1 \leq i \leq k$,至少有一条不在 E(T)中,如若不然,则 T 也将包含这个环 $e, e_1, e_2, \cdots, e_k$。设 e_j 是这环中使得 e_j| E(T)的一条边。若 e_j 比 e 有更小的成本,则 Kruskal 算法就会在 e 之前考虑 e_j,并把 e_j 计入 T。为了看出这一点,注意 E(T)中所有比 e 成本小的边也在 E(T′)中,并且和 e_j 一起构不成一个环。因此,$c(e_j) \geq c(e)$(c(·)是边成本函数)。

现在,重新考虑具有边集 E(T′)∪{e}的图。去掉环 $e, e_1, e_2, \cdots, e_k$ 上的任意一条边则留下一棵树 T"。特别是,如果删去边 e_j,则所产生的树 T"的成本将不比 T′的成本大(因为

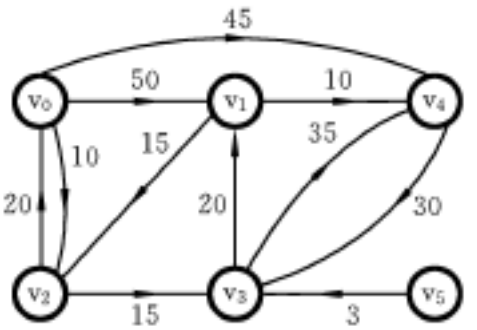
c(e′)≥c(e))。因此,T"也是一棵最小成本树。

通过反复使用上述的转换,树 T 就可以转换成 T 而在成本上没有任何增加,故 T 是一棵最小成本生成树。证毕。

5.6 单源点最短路径

要从甲地到乙地去,而甲、乙两地之间有多条交通线相连,这些交通线可以是公路、水路、铁路、航空线等,走哪条交通线才最好呢?这"最好"在不同情况下有着不同的含义,或者是距离最短,或者是时间最少,或者是旅费最省等,但抽象起来则都是在有向图中求两指定结点的最短路径问题。图论中最短路径问题可以分成很多种,例如:单源最短路径问题;每对结点之间的最短路径问题;在两个指定结点之间必须通过一个或几个其它结点的最短路径问题;找图中第一短、第二短、……的最短路径问题等等。本节只讨论单源最短路径问题。即,已知一个 n 结点有向图 G = (V,E)和边的权函数 c(e),求由 G 中某指定结点 v_0 到其它各个结点的最短路径。这里还假定所有的权都是正的。

例 5.10 考虑图 5.9(a)所示的有向图。边上的数是权,如果 v_0 是起始结点,则 v_0 到 v_1 的最短路径是 $v_0\ v_2\ v_3\ v_1$。这条路的长度是 10 + 15 + 20 = 45。虽然在这条路上有 3 条边,但它比其长度是 50 的路径 $v_0\ v_1$ 还短些。v_0 到 v_5 没有路。图 5.9(b)列出了从 v_0 到 v_1,v_2,v_3 和 v_4 的最短路径。这些路是按路径长度的非降次序列出的。



(a)

	路径	长度
(1)	$v_0\ v_2$	10
(2)	$v_0\ v_2\ v_3$	25
(3)	$v_0\ v_2\ v_3\ v_1$	45
(4)	$v_0\ v_4$	45

(b)

图 5.9 图和从 v_0 到其余各结点的最短路径

为了制定产生最短路径的贪心基础算法,对于这个问题应该想出一个多级解决办法和一种最优的量度标准。方法之一是逐条构造这些最短路径,可以使用迄今已生成的所有路径长度之和作为一种量度,为了使这一量度达到最小,其单独的每一条路径都必须具有最小长度。使用这一量度标准时,假定已经构造了 i 条最短路径,则下面要构造的路径应该是下一条最短的最小长度路径。生成从 v_0 到所有其它结点的最短路径的贪心方法就是按照路径长度的非降次序生成这些路径。首先,生成一条到最近结点的最短路径;然后生成一条到第二近结点的最短路径,等等。在图 5.9(a)中,与 v_0 最近的结点是 v_2 ($c(v_0, v_2) = 10$)。路径 $v_0\ v_2$ 是将要生成的第一条路径。与 v_0 第二近的结点是 v_3,v_0 和 v_3 之间的距离是 25。路径 $v_0\ v_2\ v_3$ 则是要生成的第二条路径。为了按这样的次序生成这些最短路径,就需要确定与其生成最短路径的下一个结点,以及到这一结点的最短路径。设 S 表示对其已经生成了最短路径的那些结点(包括 v_0)的集合。对于不在 S 中的 w,设 DIST(w)是从 v_0 开始只经过 S 中的结点而在 w 结束的那条最短路径的长度,则有以下结论。

(1) 如果下一条最短路径是到结点 u,则这条路径是从 v_0 处开始而在 u 处终止,并且只通过那些在 S 中的结点。为证明这一点,应该证明 v_0 到 u 的最短路径上的所有中间结点必定在 S 中。假定在这条路径上有一个不在 S 中的结点 w,那么,从 v_0 到 u 的路径也包含了一条从 v_0 到 w 的路径,而且这条路径的长度小于从 v_0 到 u 的路径长度。根据假设,最短路径是按照路径长度的非降次序而生成的,所以从 v_0 到 w 的最短路径应该已被生成。因此,不可能有不在 S 中的中间结点。

(2) 所生成的下一条路径的终点 u 必定是所有不在 S 内的结点中具有最小距离 DIST(u)的结点。这可由 DIST 的定义和上述(1)而得到。在存在几个不在 S 中而又有相同 DIST 的结点的情况下,则可以选择这些结点的任何一个。

(3) 在像(2)中那样选出了结点 u 并生成从 v_0 到 u 的最短路径后,结点 u 就成为 S 中的一个成员。此时,那些从 v_0 开始,只通过 S 中的结点并且在 S 外的结点 w 处结束的最短路径可能会减小,即 DIST(w)的值可能改变。如果长度改变了,则它必定是由一条从 v_0 开始,经过 u 然后到 w 的更短的路所造成的。v_0 到 u 的路径以及 u 到 w 的路径上的中间结点应全部在 S 中。而且,v_0 到 u 的路径必定是这样一条最短的路径,否则就不符合 DIST(w)的定义。u 到 w 的路径可选择成不包含任何中间结点。因此,可以得出结论,如果 DIST(w)会减少,那是由于有一条从 v_0 经 u 到 w 更短路径的缘故,其中从 v_0 到 u 的路径是这样一条最短的路,而从 u 到 w 的路径是边〈u, w〉。这条路径的长度是 DIST(u) + c(u, w)。

由上述可知,单源最短路径问题存在一个简单算法(算法 5 .10)。这个算法(通称 Dijkstra 算法)实际上只求出从 v_0 到 G 中所有其它结点的最短路径长度。这些路径的实际生成则需要对此算法作少量的补充(留下作为习题)。在过程 SHORTEST-PATHS(算法 5 .10)中,假定 G 中的 n 个结点被标记上 1 到 n,集合 S 作为一个位数组存放,如果结点 i 不在 S 中,则 S(i) = 0; 如果在 S 中,则 S(i) = 1。假定这个图用它的成本邻接矩阵来表示,COST(i, j)是边〈i, j〉的权,则在边〈i, j〉不在 E(G)中的情况下,COST(i, j)被置以某个大数 + ∞。对于 i = j, COST(i, j)可以被置成不影响这个算法结果的任一非负数。

算法 5 .10　生成最短路径的贪心算法

```
procedure SHORTEST-PATHS(v,COST,DIST,n)
    // G 是一个 n 结点有向图,它由其成本邻接矩阵 COST(n,n)表示,DIST(j)被置以结点 v 到
       结点 j 的最短路径长度,这里 1≤j≤n;DIST(v)被置成零 //
    boolean S(1：n);real COST(1：n,1：n),DIST(1：n)
    integer u,v,n,num,i,w
1   for i←1 to n do   // 将集合 S 初始化为空 //
2     S(i)←0;DIST(i)←COST(v,i)
3   repeat
4   S(v)←1;DIST(v)←0   // 结点 v 计入 S //
5   for num←2 to n - 1 do   // 确定由结点 v 出发的 n - 1 条路 //
6     选取结点 u,它使得 DIST(u) = min {DIST(w)}
                               S(w)=0
7     S(u)←1   // 结点 u 计入 S //
8     for 所有 S(w) = 0 的结点 w do   // 修改距离 //
```

```
9        DIST(w)←min (DIST(w),DIST(u) + COST(u,w))
10        repeat
11      repeat
12    end SHORTEST-PATHS
```

容易看出这个算法是正确的。在 n 结点图上,算法所花费的时间是 $O(n^2)$。为了看出这点,注意第 1 行的 for 循环需用 $\Theta(n)$时间,第 5 行的 for 循环则要执行 n - 2 次,而这个循环的每一次执行在第 6 行选择下一个结点并在第 8～10 行再一次修改 DIST 需要 O(n)时间,因此,这个循环的总时间是 $O(n^2)$。在提供一个当前不在 S 中的结点的表 T 的情况下,这个表的结点数在任何时刻都会是 n - num。这会加快第 6 行和第 8～10 行的速度,但渐近时间还是保持为$O(n^2)$。

任何最短路径算法都必须至少检查这个图中的每一条边一次,这是因为任何一条边都可能在一条最短路径中,故算法的最小时间便是 O(e)。由于用邻接矩阵来表示图,要确定哪些边在 G 中正好需用 $O(n^2)$时间,故使用这种表示法的任何最短路径算法必定花费 $O(n^2)$时间。于是,对于这种表示法,算法 SHORTEST-PATHS 在一个常因子范围内是最优的。即使换成一些邻接表,也只是使第 8～10 行的 for 循环的总时间可能降低到 O(e)(因为 DIST 只能改变那些与 u 邻近的结点)。第 6 行总的时间仍保持为 $O(n^2)$。

例 5 .11　求图 5 .10 中 v_1 到其余各个结点的最短路径。

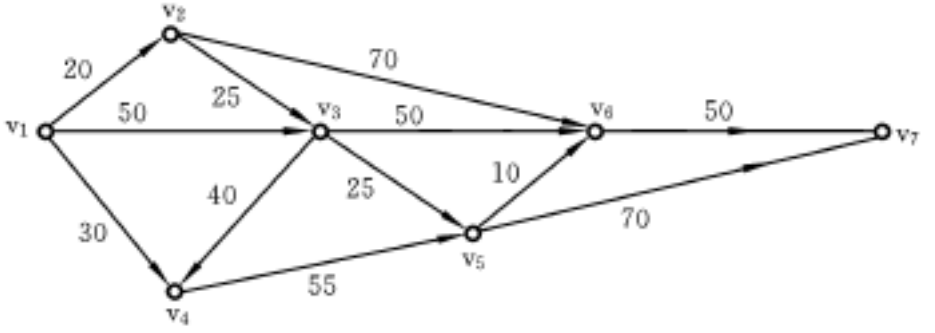


图 5 .10　一个带权的有向图

该图的成本邻接矩阵为

$$\begin{bmatrix} 0 & 20 & 50 & 30 & +\infty & +\infty & +\infty \\ +\infty & 0 & 25 & +\infty & +\infty & 70 & +\infty \\ +\infty & +\infty & 0 & 40 & 25 & 50 & +\infty \\ +\infty & +\infty & +\infty & 0 & 55 & +\infty & +\infty \\ +\infty & +\infty & +\infty & +\infty & 0 & 10 & 70 \\ +\infty & +\infty & +\infty & +\infty & +\infty & 0 & 50 \\ +\infty & +\infty & +\infty & +\infty & +\infty & +\infty & 0 \end{bmatrix}$$

以这 7 个结点有向图的有关数据为输入,执行过程 SHORTEST-PATHS。将算法第 5 行 for 循环每一次迭代所选取的结点和 DIST 的值以表 5 .1 的形式列出。可以看出,只有当这 7 个结点中的 6 个结点在 S 内时算法才会终止。

不难证明在一个连通无向图 G 中,由结点 v 到其余各结点最短路径的边构成 G 的一棵生成树(或称最短路径生成树)。显然,对于不同的根结点 v,这样的生成树可能是不同的。图5 .11显示了一个图 G,它的最小成本生成树以及一棵由结点 1 出发的最短路径生成树。

表 5.1　SHORTEST-PATHS 执行踪迹

迭　代	选　取的结点	S	DIST (1)	(2)	(3)	(4)	(5)	(6)	(7)
置初值	—	1	0	20	50	30	+∞	+∞	+∞
1	2	1,2	0	20	45	30	+∞	90	+∞
2	4	1,2,4	0	20	45	30	85	90	+∞
3	3	1,2,4,3	0	20	45	30	70	90	+∞
4	5	1,2,4,3,5	0	20	45	30	70	80	140
5	6	1,2,4,3,5,6	0	20	45	30	70	80	130

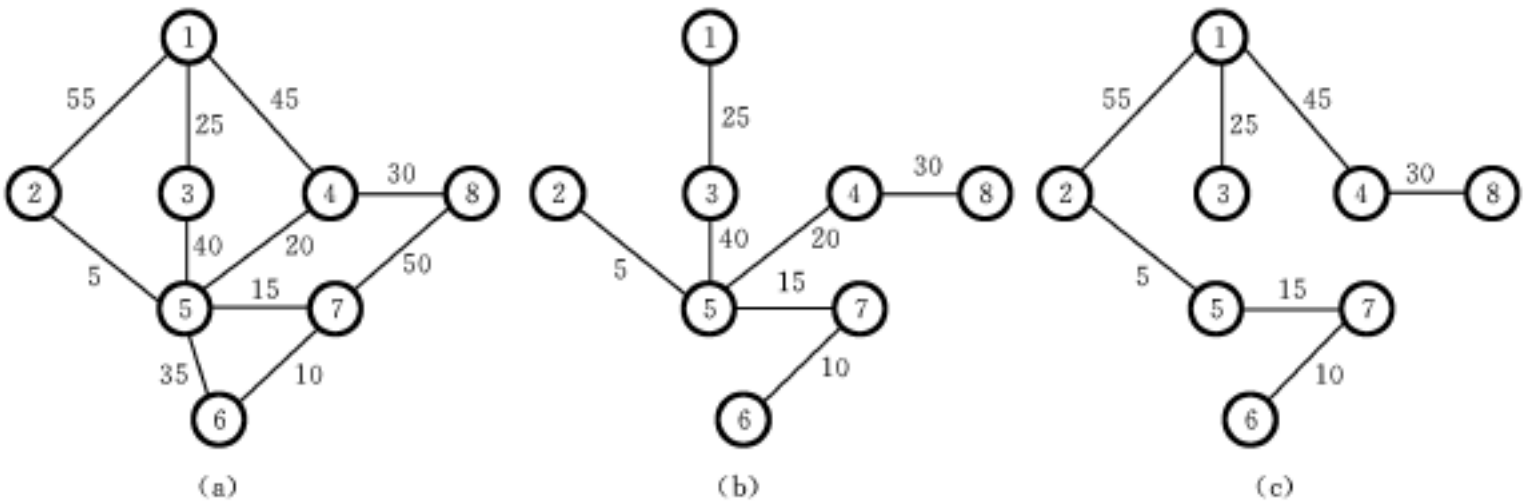


图 5.11　图和生成树

(a) 一个图;(b) 最小成本生成树;(c) 由结点 1 出发的最短路径生成树

习　题　五

5.1　设 $P_1, P_2, \cdots, P_n$ 是准备存放到长为 L 的磁带上的 n 个程序。程序 P_i 需要的带长为 a_i。如果 $\sum a_i \leq L$,显然全部程序都能存放到这条带上。现在假定 $\sum a_i > L$,要求选取一个能放在带上的程序的最大子集合 Q。最大子集指的是在其中含有最多个数的程序。构造 Q 的一种贪心策略是按 a_i 的非降次序把程序计入集合。

(1) 假设 P_i 被排成使 $a_1 \leq a_2 \leq \cdots \leq a_i$。使用上面的设计策略写一个 SPARKS 算法。要求输出一个数组 S(1：n),若 P_i 在 Q 中,则 S(i) = 1,否则 S(i) = 0。

(2) 证明这一策略总能找到最大子集 Q,使得 $\sum_{P_i \in Q} a_i \leq L$。

(3) 设 Q 是使用上述贪心策略得到的子集合,带利用率 $\sum_{P_i \in Q} a_i / L$ 可以小到什么程度?

(4) 假定现在要确定一个使带的利用率最高的程序子集合,可考虑按 a_i 的非增次序计入程序这一贪心策略。只要 P_i 带上还有够用的空间,就应将它计入 Q。假设程序已按使 $a_1 \geq a_2 \geq \cdots \geq a_n$ 的次序排列。用 SPARKS 写一个与此策略对应的算法,并分析它的时、空复杂度。

(5) 证明(1)所用的设计策略不一定得到使 $\left(\sum_{P_i \in Q} a_i\right) / L$ 取最大值的子集合。

5.2　(1)求以下情况背包问题的最优解:n = 7, M = 15, $(p_1, \cdots, p_7)$ = (10,5,15,7,6,18,3)和 $(w_1, \cdots, w_7)$ = (2,3,5,7,1,4,1)。

(2) 将以上数据情况的背包问题记为 I。设 FG(I)是物品按 p_i 的非增次序输入时由 GREEDY-KNAPSACK 所生成的解,FO(I)是一个最优解。问 FO(I)/ FG(I)是多少?

(3) 当物品按 w_i 的非降次序输入时,重复(2)的讨论。

5.3 [0/1背包问题]如果将5.3节讨论的背包问题修改成

极大化 $\sum_{1}^{n} p_i x_i$

约束条件 $\sum_{1}^{n} w_i x_i \leq M$

$x_i = 0$ 或 $1, 1 \leq i \leq n$

这种背包问题称为0/1背包问题。它要求物品或者整件装入背包或者整件不装入。求解此问题的一种贪心策略是:按 p_i/w_i 的非增次序考虑这些物品,只要正被考虑的物品能装得进就将其装入背包。证明这种策略不一定得到最优解。

5.4 [集合覆盖]已知集合族S由m个集合 $S_1, \cdots, S_m$ 组成。S的子集合 $T = \{T_1, T_2, \cdots, T_k\}$ 也是一个集合族,而且T中每一个 $T_j, 1 \leq j \leq k$,等于S中的某个 $S_i, 1 \leq i \leq m$。如果 $\bigcup_{j=1}^{k} T_j = \bigcup_{i=1}^{m} S_i$,则称T是S的一个覆盖。T的大小 $|T|$ 是T中集合的个数。S的最小覆盖是 $|T|$ 取最小值的覆盖。考虑以下的贪心策略:通过迭代构造T;在第k次迭代时,$T = \{T_1, \cdots, T_{k-1}\}$;现在将S中的一个集合 S_i 加到T,这个 S_i 中含有还不在T中最多的元素个数;当 $\bigcup_{j=1}^{k} T_j = \bigcup_{i=1}^{m} S_i$ 时停止。

(1) 假设 $\bigcup_{i=1}^{m} S_i = \{1, 2, \cdots, n\}$ 且 $m < n$。使用上面概述的策略写一个获取集合覆盖的算法,并给出该算法所需要的时间和空间。

(2) 证明上述贪心策略不一定得到最小的集合覆盖。

(3) 现在假定这样来定义最小覆盖,它是使 $\sum_{j=1}^{k} |T_j|$ 取最小值的覆盖。那么,上述策略是否总能得到这种定义下的最小覆盖呢?

5.5 [结点覆盖]设 $G = (V, E)$ 是一个无向图。G的结点覆盖是结点集V的一个子集U,它使得E中的每一条边至少与U中一个结点相关联。一个最小结点覆盖是结点数最少的结点覆盖。考虑这个问题的以下贪心算法:

```
procedure COVER(V, E)
    U←
loop
    设 v∈V 是具有最大度数的一个结点
    U←U∪{v}; V←V - {v}
    E←E - {(u, w) |或者 u = v 或者 w = v}
until E =     repeat
return (U)
and COVER
```

这个算法总是生成一个最小结点覆盖吗?

5.6 假定要将长为 $l_1, l_2, \cdots, l_n$ 的n个程序存入一盘磁带,程序i被检索的频率是 f_i。如果程序按 $i_1, i_2, \cdots, i_n$ 的次序存放,则期望检索时间(ERT)是

$$\left[\sum_{j}\left(f_{ij}\sum_{k=1}^{j} l_{ik}\right)\right] / \sum f_i$$

(1) 证明按 l_i 的非降次序存放程序不一定得到最小的ERT。

(2) 证明按 f_i 的非增次序存放程序不一定得到最小的ERT。

(3) 证明按 f_i/l_i 的非增次序来存放程序时ERT得到最小值。

5.7 假定要把长为 $l_1, l_2, \cdots, l_n$ 的n个程序分别写入到两盘磁带 T_1 和 T_2 上,并且希望按照使最大检索时间取最小值的方式存放,即,如果存放在 T_1 和 T_2 上的程序集合分别是A和B,就希望所选择的A和B使得 $\max\{\sum_{i\in A} l_i, \sum_{i\in B} l_i\}$ 取最小值。一种得到A和B的贪心方法如下:开始将A和B都初始化为空,然后一

次考虑一个程序,如果 $\sum_{i\in A} l_i = \min\{\sum_{i\in A} l_i, \sum_{i\in B} l_i\}$,则将当前正在考虑的那个程序分配给 A,否则分配给 B。证明无论是按 $l_1 \le l_2 \le \cdots \le l_n$ 或是按 $l_1 \ge l_2 \ge \cdots \ge l_n$ 的次序来考虑程序,这种方法都不能产生最优解。

5.8　(1) 当 n = 7,$(p_1, p_2, \cdots, p_7) = (3,5,20,18,1,6,30)$和$(d_1, d_2, \cdots, d_7) = (1,3,4,3,2,1,2)$时,算法 5.5 所生成的解是什么?

(2) 证明即使作业有不同的处理时间定理 5.5 亦真。这里,假定作业 i 的效益 $p_i > 0$,要用的处理时间 $t_i > 0$,限期 $d_i \ge t_i$。

5.9　(1) 对于 5.3 节的作业排序问题证明:当且仅当子集合 J 中的作业可以按下述规则处理时,J 才表示一个可行解,即如果 J 中的作业 i 还没分配处理时间,则将它分配在时间片[α - 1,α]处理,其中 α 是使得 $1 \le r \le d_i$ 的最大整数 r,且时间片[α - 1,α]是空的。

(2) 仿照例 5.4 的格式,在题 5.8 之(1)所提供的数据集上执行算法 5.5。

5.10　(1) 已知 n - 1 个元素已按 min-堆的结构形式存放在 A(1),…,A(n - 1)。现要将另一存放在 A(n)的元素和 A(1∶n - 1)中元素一起构成一个具有 n 个元素的 min-堆。对此写一个计算时间为 O(logn)的算法。

(2) 在 A(1∶n)中存放着一个 min-堆,写一个从堆顶 A(1)删去最小元素后将其余元素调整成 min-堆的算法,要求这新的堆存放在 A(1∶n - 1)中,且算法时间为 O(logn)。

(3) 利用(2)所写出的算法,写一个对 n 个元素按非增次序分类的堆分类算法。分析这个算法的计算复杂度。

5.11　(1) 证明如果一棵树的所有内部结点的度都为 k,则外部结点数 n 满足 n mod(k - 1) = 1。(2) 证明对于满足 n mod(k - 1) = 1 的正整数 n,存在一棵具有 n 个外部结点的 k 元树 T(在一棵 k 元树中,每个结点的度至多为 k)。进而证明 T 中所有内部结点的度为 k。

5.12　(1) 证明如果 n mod(k - 1) = 1,则在定理 5.4 后面所描述的贪心规则对于所有的$(q_1, q_2, \cdots, q_n)$生成一棵最优的 k 元归并树。

(2) 当$(q_1, q_2, \cdots, q_{11}) = (3,7,8,9,15,16,18,20,23,25,28)$时,画出使用这一规则所得到的最优三元归并树。

5.13　证明 5.5 节的 Prim 方法生成最小成本生成树。

5.14　在假定图用邻接表来表示的情况下重写 Prim 算法,并分析它的计算复杂度。

5.15　证明引理 5.1。

5.16　通过考察 n 结点的完全图,证明在一个 n 结点图中可能有比 $2^{n-1} - 2$ 棵还多的生成树。

5.17　在图 5.12 的有向图中,利用算法 SHORTEST-PATHS 获取按长度非降次序排列的由结点 1 到其余各结点最短路径长度。

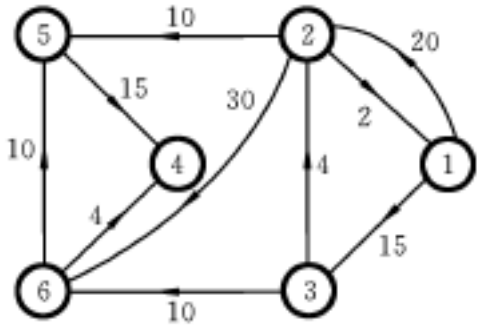


图 5.12　有向图

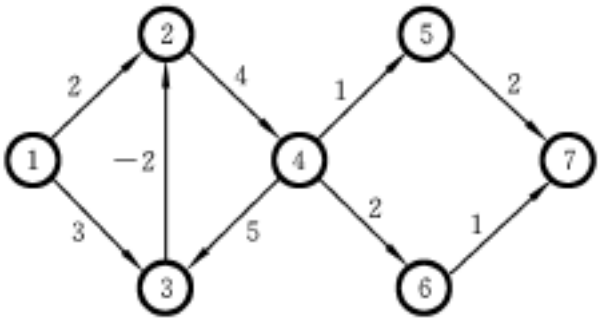


图 5.13　有向图

5.18　说明为什么将 SHORTEST-PATHS 应用于图 5.13 的有向图就不能正常地工作。结点 v_1 和 v_7 之间的最短路径是什么?

5.19　修改算法 SHORTEST-PATHS,使得它在获取最短路径的同时还得到这些最短路径。请给出修改后的算法的计算时间。

第 6 章

动 态 规 划

6.1 一 般 方 法

在实际生活中,有这么一类问题,它们的活动过程可以分为若干个阶段,而且在任一阶段后的行为都仅依赖于 i 阶段的过程状态,而与 i 阶段之前的过程如何达到这种状态的方式无关,这样的过程就构成一个多阶段决策过程。在 20 世纪 50 年代,贝尔曼(Richard Bellman)等人根据这类问题的多阶段决策的特性,提出了解决这类问题的“最优性原理”,从而创建了最优化问题的一种新的算法设计方法——动态规划。

在多阶段决策过程的每一阶段,都可能有多种可供选择的决策,必须从中选取一种决策。一旦各个阶段的决策选定之后,就构成了解决这一问题的一个决策序列。决策序列不同,所导致的问题的结果也不同。动态规划的目标就是要在所有容许选择的决策序列中选取一个会获得问题最优解的决策序列,即最优决策序列。

显然,用枚举的方法从所有可能的决策序列中选取最优决策序列是一种最笨拙的方法。贝尔曼认为,利用最优性原理(principle of optimality)以及所获得的递推关系式去求取最优决策序列可以使枚举量急剧下降。这个原理指出,过程的最优决策序列具有如下性质:无论过程的初始状态和初始决策是什么,其余的决策都必须相对于初始决策所产生的状态构成一个最优决策序列。如果所求解问题的最优性原理成立,则说明用动态规划方法有可能解决该问题;而解决问题的关键在于获取各阶段间的递推关系式。

例 6.1 [多段图问题]多段图 $G=(V,E)$ 是一个有向图。它具有如下特性:图中的结点被划分成 $k\geq 2$ 个不相交的集合 V_i,$1\leq i\leq k$,其中 V_1 和 V_k 分别只有一个结点 s(源点)和 t(汇点)。图中所有的边 $\langle u,v\rangle$ 均具有如下性质:若 $u\in V_i$,则 $v\in V_{i+1}$,$1\leq i<k-1$,且每条边 $\langle u,v\rangle$ 均附有成本 $c(u,v)$。从 s 到 t 的一条路径成本是这条路径上边的成本和。多段图问题(multistage graph problem)是求由 s 到 t 的最小成本路径。每个集合 V_i 定义图中的一段。由于 E 的约束,每条从 s 到 t 的路径都是从第 1 段开始,在第 k 段终止。图 6.1 所示的就是一个 5 段图。

对于每一条由 s 到 t 的路径,可以把它看成在 k-2 个阶段中作出的某个决策序列的相应结果。第 i 次决策就是确定 V_{i+1} 中的哪个结点在这条路径上,$1\leq i\leq k-2$。下面证明最优性原理对多段图成立。假设 $s,v_2,v_3,\cdots,v_{k-1},t$ 是一条由 s 到 t 的最短路径,还假定从源点 s(初始状态)开始,已作出了到结点 v_2 的决策(初始决策),因此 v_2 就是初始决策所产生的状态。如果把 v_2 看成是原问题的一个子问题的初始状态,则解这个子问题就是找出一条由 v_2 到 t 的最短路径。这条最短路径显然是 $v_2,v_3,\cdots,v_{k-1},t$。如若不然,设 $v_2,q_3,\cdots,q_{k-1},t$ 是一条由 v_2 到 t 的更短路径,则 $s,v_2,q_3,\cdots,q_{k-1},t$ 是一条比路径 $s,v_2,v_3,\cdots,v_{k-1},t$ 更短的

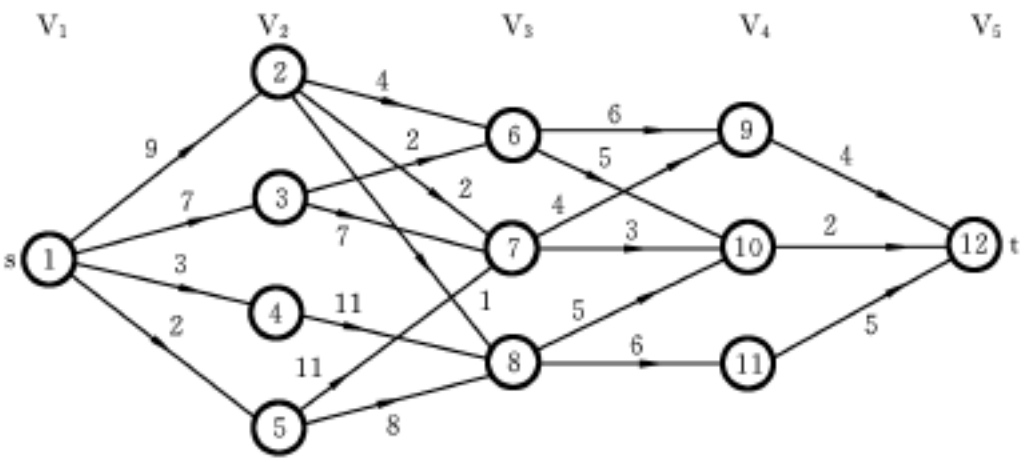


图 6.1　一个 5 段图

由 s 到 t 的路径。与假设矛盾,故最优性原理成立。因此,它为使用动态规划方法来解多段图问题提供了可能。

例 6.2　[0/1 背包问题]此问题除了限定 x_j 必须取 0 或 1 值外,其余条件及目标函数均与 5.2 节背包问题类同。用 KNAP(l, j, x)来表示这个问题。

极大化　$\sum_{l\le i\le j} p_i x_i$

约束条件　$\sum_{l\le i\le j} w_i x_i \le x$　　(6.1)

$x_i = 0$ 或 $1, l\le i\le j$

0/1 背包问题就是 KNAP(1, n, M)。设 $y_1, y_2, \cdots, y_n$ 分别是 $x_1, x_2, \cdots, x_n$ 的 0/1 值的最优序列。若 $y_1 = 0$,则 $y_2, y_3, \cdots, y_n$ 必须相对于 KNAP(2, n, M)问题构成一个最优序列。如若不然,则 $y_1, y_2, \cdots, y_n$ 就不是 KNAP(1, n, M)的最优序列。若 $y_1 = 1$,则 $y_2, y_3, \cdots, y_n$ 必须是 KNAP(2, n, M - w_1)的最优序列。如若不然,则必有另一 0/1 序列 $z_2, z_3, \cdots, z_n$ 使得 $\sum_{2\le i\le n} w_i z_i \le M - w_1$ 且 $\sum_{2\le i\le n} p_i z_i > \sum_{2\le i\le n} p_i y_i$。因此,序列 $y_1, z_2, z_3, \cdots, z_n$ 是一个对问题 KNAP(1, n, M)具有更大效益值的序列。最优性原理成立。

能用动态规划求解的问题的最优化决策序列可表示如下。设 S_0 是问题的初始状态。假定需要作 n 次决策 x_i, $1\le i\le n$。设 $X_1 = \{r_{1,1}, r_{1,2}, \cdots, r_{1,p_1}\}$ 是 x_1 的可能决策值的集合,而 S_{1,j_1} 是在选取决策值 r_{1,j_1} 以后所产生的状态,$1\le j_1 \le p_1$。又设 Γ_{1,j_1} 是相应于状态 S_{1,j_1} 的最优决策序列。那么,相应于 S_0 的最优决策序列就是$\{r_{1,j_1}\ \Gamma_{1,j_1} \mid 1\le j_1 \le p_1\}$中最优的序列,记为 $\underset{1\le j_1\le p_1}{\mathrm{OPT}}\{r_{1,j_1}\ \Gamma_{1,j_1}\} = r_1\ \Gamma_1$。如果已作了 k - 1 次决策,$1\le k-1<n$,设 $x_1, \cdots, x_{k-1}$ 的最优决策值是 $r_1, \cdots, r_{k-1}$,它们所产生的状态为 $S_1, \cdots, S_{k-1}$;又设 $X_k = \{r_{k,1}, \cdots, r_{k,p_k}\}$ 是 x_k 的可能值的集合,则 S_{k,j_k} 是选取决策值 r_{k,j_k} 后所产生的状态,$1\le j_k \le p_k$;Γ_{k,j_k} 是相应于 S_{k,j_k} 的最优决策序列。因此,相应于 S_{k-1} 的最优决策序列是 $\underset{1\le j_k\le p_k}{\mathrm{OPT}}\{r_{k,j_k}\ \Gamma_{k,j_k}\} = r_k\ \Gamma_k$。于是,相应于 S_0 的最优决策序列为 $r_1, \cdots, r_{k-1}, r_k, \Gamma_k$。

尽管上面给出了最优决策序列的表示,但由于多阶段决策问题的各个阶段是互相联系的,因此,一般不可能在每一阶段直接选出最优决策序列中属于此阶段的决策值。例如,对于 k 段图问题就不可能直接求出 V_2 中的哪个结点是最短路径上的第二个结点,V_3 中的哪个结点是该路径上的第三个结点,等等。不过,最优决策序列的表示启发我们可从最后阶段开始,以逐步向前递推的方式列出求前一阶段决策值的递推关系式,即根据 $x_{i+1}, \cdots, x_n$ 的那

些最优决策序列来列出求取 x_i 决策值的关系式,这就是动态规划的向前处理法(forward approach)。列出关系式后,由最后阶段开始,回溯求解这些关系式得出最优决策序列。由该决策序列所得到的结果就是问题的最优解。

例 6.3 在 k 段图问题中,设 $v_{2,j_2} \in V_2$, $1 \leq j_2 \leq p_2$, $|V_2| = p_2$;又设 $\Gamma_{v_{2,j_2}}$ 是由 v_2 经 j_2 到 t 的最短路径,则 s 到 t 的最短路径是 $\{s\Gamma v_{2,j_2} \mid v_{2,j_2} \in V_2, 1 \leq j_2 \leq p_2\}$ 中最短的那条路径。若设 $s, v_2, \cdots, v_i, \cdots, v_{k-1}, t$ 是 s 到 t 的一条最短路径,v 是路径上的中间结点,则 $s, v_2, \cdots, v_i$ 和 $v_i, \cdots, v_{k-1}, t$ 就应该分别是由 s 到 v_i 和由 v_i 到 t 的最短路径。

例 6.4 对于 0/1 背包问题,设 $g_j(x)$ 是 KNAP(j+1, n, x)最优解的值,显然 $g_0(M)$ 是 KNAP(1, n, M)最优解的值。由于 x_1 可能的取值是 0 或 1,因此可得

$$g_0(M) = \max\{g_1(M), g_1(M - w_1) + p_1\} \tag{6.2}$$

由于 x_j 可能的取值也是 0 或 1,因此可以得出递推关系式

$$g(x) = \max\{g_{j+1}(x), g_{j+1}(x - w_{j+1}) + p_{j+1}\} \tag{6.3}$$

显然,对于 x 大于等于零的所有取值,有 $g_n(x) = 0$,若 x 小于零,则有 $g_n(x) = -\infty$。以此为开始,利用式(6.3)可求出 $g_{n-1}(x)$,然后由 $g_{n-1}(x)$ 就可得到 $g_{n-2}(x)$。继续递推求解式(6.3),就可确定 $g_1(x)$ 和最后求出 $g_0(M)$。

在求解多阶段决策问题时,除了用向前处理法列递推关系式求解外,还可用向后处理法列递推关系式,以由前向后递推的方式求解所列出的关系式,从而得出最优决策序列。向后处理法(backward approach)就是根据 $x_1, \cdots, x_{i-1}$ 的那些最优决策序列列出求 x 的递推关系式。下面的两个例子就相当于用向后处理法来求解的。

例 6.5 在 k 段图问题中,设 $v_{k-1,j_{k-1}} \in V_{k-1}$, $1 \leq j_{k-1} \leq p_{k-1}$, $|V_{k-1}| = p_{k-1}$,又设 $\Gamma v_{k-1,j_{k-1}}$ 是由 s 到 $v_{k-1,j_{k-1}}$ 的最短路径,那么,s 到 t 的最短路径就是 $\{\Gamma v_{k-1,j_{k-1}} t \mid v_{k-1,j_{k-1}} \in V_{k-1}, 1 \leq j_{k-1} \leq p_{k-1}\}$ 中最短的那条路径。

例 6.6 在 0/1 背包问题中,设 f(x)是 KNAP(1, i, x)最优解的值,由向后处理法可列出如下的递推关系式:

$$f(x) = \max\{f_{j-1}(x), f_{j-1}(x - w_i) + p_i\} \tag{6.4}$$

设 KNAP(1, n, M)最优解的值是 $f_n(M)$。由于当所有的 $x \geq 0$ 时,有 $f_0(x) = 0$,而 $x < 0$ 时,有 $f_0(x) = -\infty$,故以此为开始来求解式(6.4)就可成功地得出 $f_1, f_2, \cdots, f_n$。

无论是使用向前处理法还是使用向后处理法,都将所有子问题的最优解保存下来。这样做的目的是为了便于逐步递推得到原问题的最优解并避免对它们的重复计算。尽管动态规划保留了所有子问题最优解的值,但在递推求解过程中,由于对包含次优子序列的决策序列不予考虑,因此大大节省了计算量。由枚举法可知,不同决策序列的总数就其所取决策值而言是指数级的(如果决策序列由 n 次决策构成,而每次决策有 p 种选择,那么可能的决策序列就有 p^n 个),而动态规划算法则可能有多项式的复杂度。

6.2 多 段 图

例 6.1 给出了多段图的定义,并且指出一个 k 段图的每一条由源点 s 到汇点 t 的路径可以看成是在 k-2 个阶段中作出的某个决策序列的相应结果。第 i 次决策就是确定 V_{i+1} 中

的哪个结点在这条路径上,$1 \le i \le k-2$。进而还证明了最优性原理对多段图成立,因此用动态规划方法有可能找到由 s 到 t 的最小成本路径。在本节首先根据上节介绍的向前处理法列出求取 s 到 t 的最小成本路径的递推式,继而给出 k 段图问题的向前处理算法。设 P(i,j)是一条从 V_i 中的结点 j 到汇点 t 的最小成本路径,COST(i,j)是这条路的成本。于是,由向前处理法立即就可得

$$\mathrm{COST}(i,j) = \min_{\substack{l \in V_{i+1} \\ \langle j,l\rangle \in E}} \{c(j,l) + \mathrm{COST}(i+1,l)\} \tag{6.5}$$

因为若 $\langle j,t\rangle \in E$,有 COST(k-1,j) = c(j,t);若 $\langle j,t\rangle \mid E$,有 COST(k-1,j) = ∞,所以可以通过如下步骤解式(6.5)并求出 COST(1,s):首先对于所有 $j \in V_{k-2}$,计算 COST(k-2,j),然后对所有的 $j \in V_{k-3}$,计算 COST(k-3,j)等,最后计算出 COST(1,s)。下面对图 6.1 的 5 段图给出具体实现这一系列计算的步骤:

COST(3,6) = min{6 + COST(4,9),5 + COST(4,10)} = 7

COST(3,7) = min{4 + COST(4,9),3 + COST(4,10)} = 5

COST(3,8) = 7

COST(2,2) = min{4 + COST(3,6),2 + COST(3,7),1 + COST(3,8)} = 7

COST(2,3) = 9

COST(2,4) = 18

COST(2,5) = 15

COST(1,1) = min{9 + COST(2,2),7 + COST(2,3),3 + COST(2,4),2 + COST(2,5)}
= 16

于是,由 s 到 t 的最小成本路径的成本为 16。如果在计算每一个 COST(i,j)的同时,记下每个状态(结点 j)所作的决策(即,使 c(j,l) + COST(i+1,l)取最小值的 l 值),设它为 D(i,j),则可容易地求出这条最小成本路径。对于图 6.1 所示的 5 段图,可得到

D(3,6) = 10　　D(3,7) = 10　　D(3,8) = 10　　D(2,2) = 7

D(2,3) = 6　　D(2,4) = 8　　D(2,5) = 8　　D(1,1) = 2

设这条最小成本路径是 $s = 1, v_2, v_3, \cdots, v_{k-1}, t = 12$。立即可知,$v_2 = D(1,1) = 2$,$v_3 = D(2,D(1,1)) = 7$ 和 $v_4 = D(3,D(2,D(1,1))) = D(3,7) = 10$。

为了使写出的算法更简单一些,可事先对结点集 V 的结点按下述方式排序:首先将 s 结点编成 1 号,然后对 V_2 中的结点编号,V_3 的结点接着 V_2 中的最后一个编号继续往下编……最后将 t 编成 n 号。经过这样编号,V_{i+1} 中结点的编号均大于 V_i 中结点的编号(见图 6.1)。于是,COST 和 D 都可按 n-1,n-2,…,1 的次序计算,而无需考虑 COST,P 和 D 中标识结点所在段数的第一个下标,因此它们的第一个下标可在算法中略去。所导出的算法是过程 FGRAPH。

算法 6.1　多段图的向前处理算法

```
line procedure FGRAPH(E,k,n,P)
    // 输入是按段的顺序给结点编号的,有 n 个结点的 k 段图。E 是边集,c(i,j)是边 <i,j> 的成
       本。P(1：k)是最小成本路径 //
1     real COST(n),integer D(n-1),P(k),r,j,k,n
```

```
2     COST(n)←0
3     for j←n - 1 to 1 by  - 1 do   // 计算 COST(j) //
4          设 r 是一个这样的结点, ⟨j,r⟩ ∈E 且使 c(j,r) + COST(r)取最小值
5          COST(j)←c(j,r) + COST(r)
6          D(j)←r
7     repeat   // 找一条最小成本路径 //
8     P(1)←1; P(k)←n
9     for j←2 to k - 1 do   // 找路径上的第 j 个结点 //
10         P(j)←D(P(j - 1))
11    repeat
12  end FGRAPH
```

过程 FGRAPH 的复杂度分析相当简单。如果 G 用邻接表表示,那么第 4 行的 r 可以在与结点 j 的度成正比的时间内算出。因此,如果 G 有 e 条边,则第 3～7 行的 for 循环的时间是 Θ(n + e),第 9～11 行的 for 循环时间是 Θ(k)。总的计算时间在Θ(n + e)以内。除了输入所需要的空间外,还需要给 COST,D 和 P 的空间。

k 段图问题也能用向后处理法求解。设 BP(i,j)是一条由源点 s 到 V_i 中结点 j 的最小成本路径,BCOST(i,j)是 BP(i,j)的成本。由向后处理法得到

$$BCOST(i,j) = \min_{\substack{l \in V_{i-1} \\ \langle l,j\rangle \in E}} \{BCOST(i-1,l) + c(l,j)\} \qquad (6.6)$$

因为,若⟨1,j⟩∈E,有 BCOST(2,j) = c(1,j),若⟨1,j⟩ E,有 BCOST(2,j) = ∞,所以,可用式(6.6)首先对 i = 3 计算 BCOST,然后对 i = 4 计算 BCOST 等,最后计算出 BCOST(k,t)。对图 6.1 的 5 段图可进行如下计算:

$$BCOST(3,6) = \min\{BCOST(2,2) + 4, BCOST(2,3) + 2\} = 9$$

$$BCOST(3,7) = 11$$

$$BCOST(3,8) = 10$$

$$BCOST(4,9) = \min\{BCOST(3,6) + 6, BCOST(3,7) + 4\} = 15$$

$$BCOST(4,10) = \min\{BCOST(3,6) + 5, BCOST(3,7) + 3, BCOST(3,8) + 5\} = 14$$

$$BCOST(4,11) = 16$$

$$BCOST(5,12) = \min\{BCOST(4,9) + 4, BCOST(4,10) + 2, BCOST(4,11) + 5\} = 16$$

获取 s 到 t 的一条最小成本路径的向后处理算法是过程 BGRAPH。它与 FGRAPH 一样略去了 BCOST,P 和 D 的第一个下标。只要 G 用它的反邻接表表示(即对于每一个结点 v,有一个使得⟨w,v⟩∈E 的结点 w 的链接表),这个算法就和 FGRAPH 有一样的计算复杂度。

算法 6.2　多段图的向后处理算法

```
procedure BGRAPH(E,k,n,P)
    // 和 FGRAPH 功能相同 //
    real BCOST(n); integer D(n - 1),P(k),r,j,k,n
    BCOST(1)←0
    for j←2 to n do   // 计算 BCOST(j) //
        设 r 是一个这样的结点, ⟨r,j⟩ ∈E 且使 BCOST(r) + c(r,j)取最小值
        BCOST(j)←BCOST(r) + c(r,j)
```

```
        D(j)←r
      repeat
      //找一条最小成本路径//
      P(1)←1;P(k)←n
      for j←k - 1 to 2 by - 1 do   //找路径上的第 j 个结点//
        P(j)←D(P(j+1))
      repeat
end BGRAPH
```

不难看出,即使对多段图的更一般形式,即图中允许有这样的边〈u,v〉,u∈V_i,v∈V_j 且 i<j,FGRAPH 和 BGRAPH 都能正确地运行。

多段图问题虽然简单,用处却很大。很多实际问题都可用多段图来表述,下面仅给出一个例子。考虑把 n 个资源分配给 r 个项目的问题。如果把 j 个资源,0≤j≤n,分配给项目 i,所获得的净利是 N(i,j),要求按使得总净利达到最大值的方法把资源分配给这 r 个项目,那么,这个问题可用一个 r+1 段图来表示。1 到 r 之间的段 i 代表项目 i。源点 s=V(1,0),汇点 t=V(r+1,n)。段 i 为 2 到 r 时,每段都有 n+1 个结点 V(i,j),0≤j≤n,其中每个结点 V(i,j)表示共把 j 个资源分配给了项目 1,2,…,i-1。图 G 中的边都具有〈V(i,j),V(i+1,l)〉的形式,这里 j≤l,1≤i≤r。当 j≤l 且 1≤i<r 时,边〈V(i,j),V(i+1,l)〉赋予 N(i,l-j)的成本值(即净利)表示给项目 i 分配 l-j 个资源。当 j≤n 且 i=r 即边具有〈V(r,j),V(r+1,n)〉的形式时,每一条这样的边被赋予 $\max_{0\le p\le n-j}\{N(r,p)\}$的成本值。图 6.2 显示出将 4 个资源分配给 3 个项目的资源分配问题所产生的 4 段图。显然,资源的最优分配方案由 s 到 t 的一条最大成本路径所确定。只要改变所有边的成本符号立即就可将此问题转换成最小成本问题,利用前面所给出的任何一个多段图算法都可算出此问题的最优解。

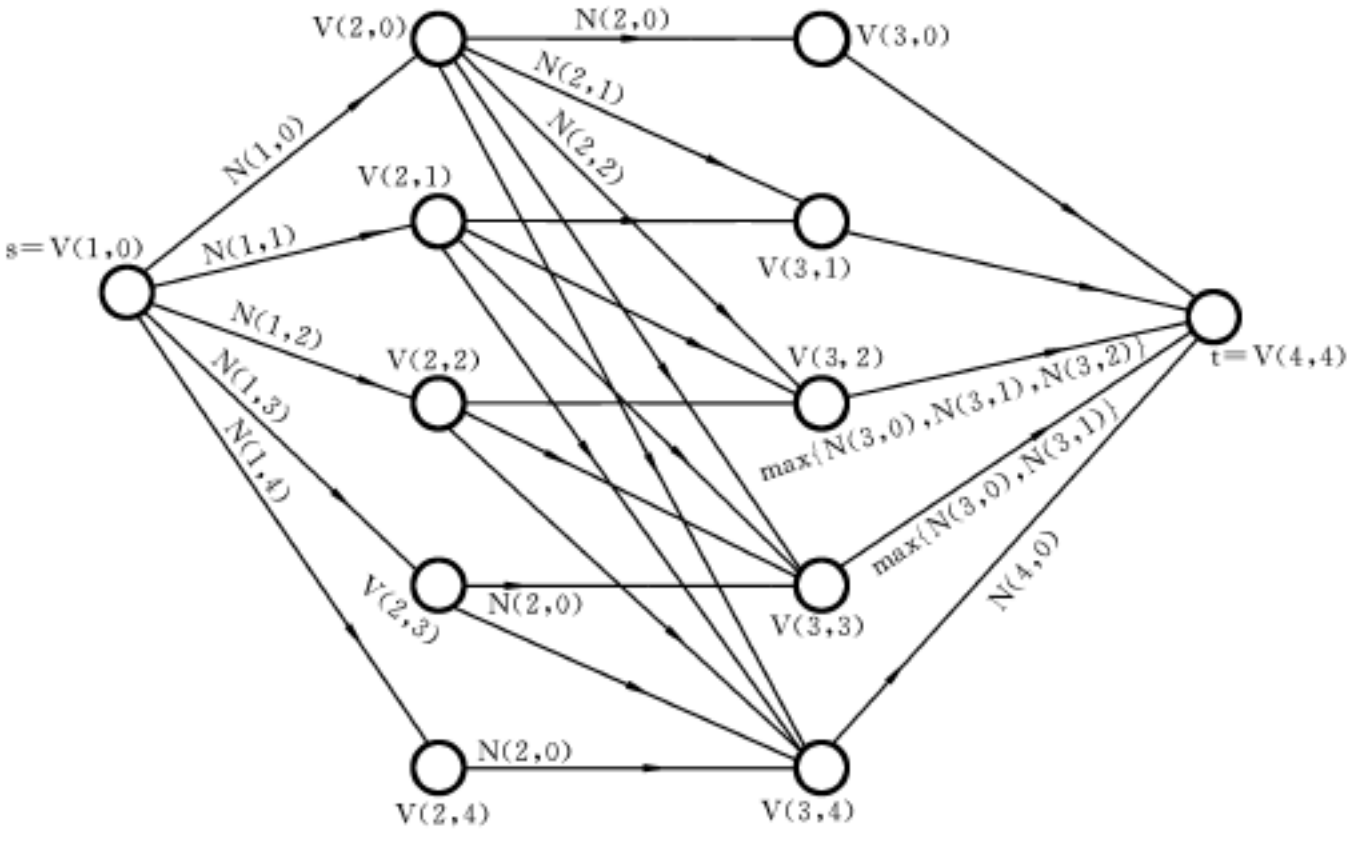


图 6.2　给 3 个项目分配资源的 4 段图

6.3 每对结点之间的最短路径

设 G=(V,E)是一个有 n 个结点的有向图。又设 C 是 G 的成本邻接矩阵,其中 C(i,i)=0,1≤i≤n;当〈i,j〉∈E(G)时,C(i,j)表示边〈i,j〉的长度(或成本);当 i≠j 且〈i,j〉| E(G)时,C(i,j)=∞。每对结点之间的最短路径问题(all pairs shortest path problem)是求满足下述条件的矩阵 A:A 中的任何元素 A(i,j)是代表结点 i 到结点 j 的最短路径的长度。这样的矩阵 A 可以通过 5.6 节中的过程 SHORTEST-PATHS 求 n 个单源点的最短路径问题而获得。由于每用一次这个过程要花 $O(n^2)$的时间,因此求出矩阵 A 所花的时间为 $O(n^3)$。下面利用动态规划方法来设计这个问题的另一种算法,虽然它要求的计算时间仍是 $O(n^3)$,但在边成本上比 SHORTEST-PATHS 要求的约束条件要弱些。它只要求 G 不含负长度的环即可,而不是要求所有的 C(i,j)≥0。要指出的是,若允许 G 包含一个负长度的环,则在此环上任意两结点间的最短长度为 -∞。

考察 G 中一条由 i 到 j 的最短路径,i≠j。这条路径由 i 出发,通过一些中间结点(也可能没有),在 j 处结束。可以假定这条路径不含有环,因为如果有环,则可去掉这个环且不增加这条路径的长度(不含有负长度的环)。如果 k 是这条最短路径上的一个中间结点,那么由 i 到 k 和由 k 到 j 的这两条子路径应分别是由 i 到 k 和由 k 到 j 的最短路径。否则,这条由 i 到 j 的路径就不是具有最小长度的路径。于是,最优性原理成立。这表明有使用动态规划的可能性。如果 k 是编号最高的中间结点,那么由 i 到 k 的这条最短路径上就不会有比编号 k-1 更大的结点通过。同样,在 k 到 j 的那条最短路径上也不会有比编号 k-1 更大的结点通过。因此,可以把求取一条由 i 到 j 的最短路径看成是如下的过程:首先需要决策哪一个结点是该路径上具有最大编号的中间结点 k,然后再去求取由 i 到 k 和由 k 到 j 这两对结点间的最短路径。当然,这两条路径都不可能有比 k-1 还大的中间结点。根据上述可以得到求解本问题的递推关系式。用 $A^k(i,j)$表示从 i 到 j 并且不经过比 k 还大的结点的最短路径长度。由于 G 中不存在编号比 n 还大的结点,因此 $A(i,j)=A^n(i,j)$,即由 i 到 j 的最短路径上不通过比 n 编号还大的结点。当然这条路径可能通过结点 n 也可能不通过结点 n。如果它通过结点 n,则 $A^n(i,j)=A^{n-1}(i,n)+A^{n-1}(n,j)$。如果它不通过结点 n,则没有编号大于 n-1 的结点,所以 $A^n(i,j)=A^{n-1}(i,j)$。组合起来就得到

$$A^n(i,j)=\min\{A^{n-1}(i,j),A^{n-1}(i,n)+A^{n-1}(n,j)\} \tag{6.7}$$

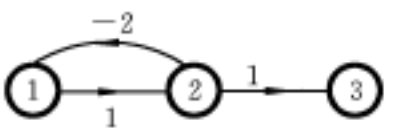
同理可得

$$A^k(i,j)=\min\{A^{k-1}(i,j),A^{k-1}(i,k)+A^{k-1}(k,j)\},k\geq 1 \tag{6.8}$$

显然,$A^0(i,j)=C(i,j)$,$1\leq i\leq n$,$1\leq j\leq n$。通过先计算 A^1,然后计算 A^2,A^3,…,A^n,就可得到每个结点对之间的最短路径长度。

下面的例子表明如果图中含有负长度的环,则式(6.8)不成立。

例 6.7 图 6.3 显示了一个有向图和它的 A^0 矩阵。



$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

图 6.3 含有负长度环的图

对于这个图,$A^2(1,3)\neq\min\{A^1(1,3),$

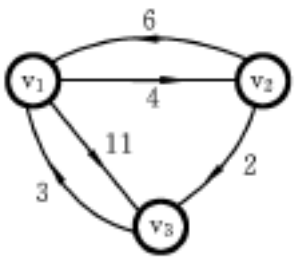
$A^1(1,2)+A^1(2,3)\}=2$。其原因在于出现了长度为 -1 的环 1,2,1,所以由 1 到 3 的最短路径 1,2,1,2,…,1,2,3 的长度可以作得任意小。

过程 ALL-PATHS 给出了求所有结点对间最短路径长度的算法。由于 $A^k(i,k)=A^{k-1}(i,k)$和 $A^k(k,j)=A^{k-1}(k,j)$,因此在构造 A^k 时,第 k 列和第 k 行的数组元素值不变。从而在 7～11 行的迭代中,当 $j>k$ 且 $i<k$ 时,执行第 9 行的语句,从形式上看是 $A^k(i,j)\leftarrow\min\{A^{k-1}(i,j),A^k(i,k)+A^{k-1}(k,j)\}$,但基于上述理由,它实质上等价于执行 $A^k(i,j)\leftarrow\min\{A^{k-1}(i,j),A^{k-1}(i,k)+A^{k-1}(k,j)\}$。同样,当 i 和 j 中至少有一个大于 k 时,第 9 行求出的 $A^k(i,j)$也一定是 $\min\{A^{k-1}(i,j),A^{k-1}(i,k)+A^{k-1}(k,j)\}$。因此,在过程中取消 A 的上标能保证进行恰当的计算。

算法 6.3　每对结点之间的最短路径长度

```
procedure ALL-PATHS(COST, A, n)
    // COST(n,n)是 n 结点图的成本邻接矩阵; A(i,j)是结点 v_i 到 v_j 的最短路径的成本;
     COST(i,i) = 0, 1≤i≤n //
    integer i,j,k,n; real COST(n,n), A(n,n)
1   for i←1 to n do
2     for j←1 to n do
3       A(i,j)←COST(i,j)    // 将 COST(i,j)复制到 A(i,j) //
4     repeat
5   repeat
6   for k←1 to n do    // 对最高下标为 k 的结点的路径//
7     for i←1 to n do    // 对于所有可能的结点对 //
8       for j←1 to n do
9         A(i,j)←min{A(i,j), A(i,k) + A(k,j)}
10      repeat
11    repeat
12  repeat
13  end ALL-PATHS
```

例 6.8　图 6.4 示出了一个有向图 G 和它的成本矩阵。A 的初始矩阵 A^0 和 3 次迭代后 A^1, A^2, A^3 的值在表 6.1 中给出。



(a)

	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b)

图 6.4　有向图 G 和它的成本矩阵

(a) 有向图 G; (b) G 的成本矩阵

表 6.1　用 ALL-PATHS 对图 6.4 的有向图产生的矩阵 A^k

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

设 M = max{COST(i, j) | ⟨i, j⟩ ∈ E(G)}。易于看出 $A^n(i,j) \leq (n-1) * M$。为便于 ALL-PATHS 能在计算机上正常执行,对于那些⟨i, j⟩ ∉ E 且 i≠j 的 COST(i, j)置以一个大于(n - 1) * M 的初值(而不是∞)。如果在结束时 A(i, j) > (n - 1) * M,则表明 G 中由 i 到 j 没有有向路径。

过程 ALL-PATHS 所需的时间很容易确定。第 9 行迭代了 n^3 次,而且整个循环与矩阵 A 中的数据无关,因此过程的计算时间是 $\Theta(n^3)$。至于求由 i 到 j 的最短路径所需增设的内容,则留作一道习题。

6.4 最优二分检索树

2.4.2 节给出了二分检索树的定义。根据定义,要求树中所有的结点是互异的。对于一个给定的标识符集合,可能有若干棵不同的二分检索树。图 6.5 给出了关于 SPARKS 保留字的一个子集的两棵二分检索树。

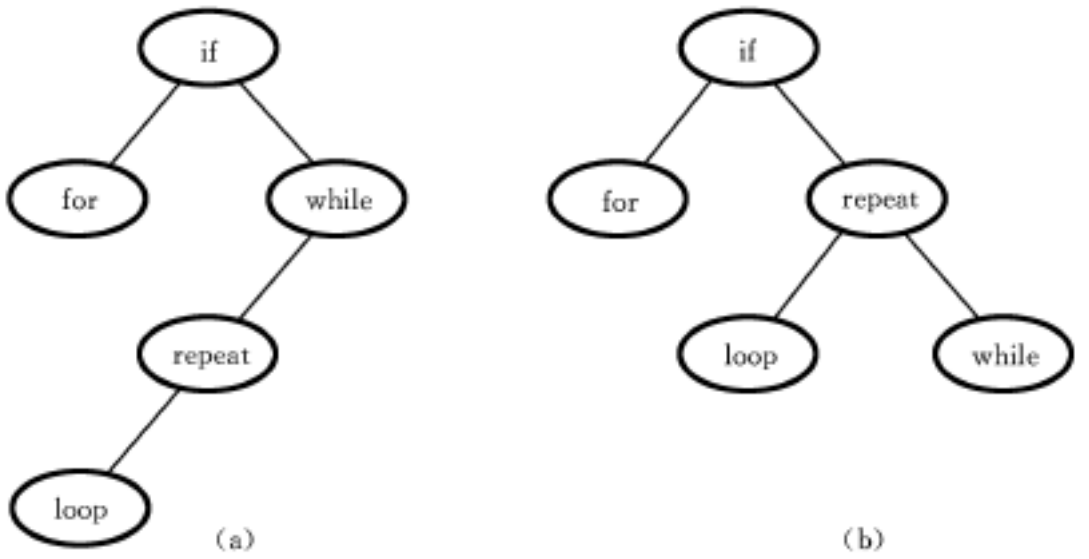


图 6.5 两棵二分检索树

为了确定标识符 X 是否在一棵二分检索树中出现,将 X 先与根比较,如果 X 比根中标识符小,则检索在左子树中继续;如果 X 等于根中标识符,则检索成功地终止;否则检索在右子树中继续下去。上述步骤可以形式化为过程 SEARCH。

算法 6.4 检索一棵二分检索树

```
        procedure SEARCH(T,X,i)
          //为 X 检索二分检索树 T,这棵树的每个结点有 3 个信息段:LCHILD,IDENT 和
            RCHILD。如果 X 不在 T 中,则置 i=0,否则将 i 置成使得 IDENT(i) = X//
1       i←T
2       while i≠0 do
3          case
4               :X < IDENT(i):i←LCHILD(i)    //检索左子树//
5               :X = IDENT(i):return
6               :X > IDENT(i):i←RCHILD(i)    //检索右子树//
7          endcase
8       repeat
9    end SEARCH
```

已知一个固定的标识符集合,希望产生一种构造二分检索树的方法。可以预料,同一个标识符集合有不同的二分检索树,而不同的二分检索树有不同的性能特征。图 6.5(a)所示的树在最坏情况下找一个标识符需要进行 4 次比较,而图 6.5(b)所示的那棵树只需要 3 次比较,在平均情况下,这两棵树各需要 12/5 和 11/5 次比较。这一计算结果是假定检索每一个标识符具有同等的概率并且在任何时候都不做不在 T 中的标识符的检索。

在一般情况下,可以预计所要检索的那些不同的标识符具有不同的频率(或概率)。另外,也要做一些不成功的检索(即对不在这棵树中标识符的检索)。假定所给出的标识符集是$\{a_1, a_2, \cdots, a_n\}$,其中 $a_1 < a_2 < \cdots < a_n$。设 P(i)是对 a_i 检索的概率,Q(i)是正被检索的标识符 X 的概率,而标识符 X 满足 $a_i < X < a_{i+1}$,$0 \leq i \leq n$(假定 $a_0 = -\infty$且 $a_{n+1} = +\infty$)。那么,$\sum_{0 \leq i \leq n} Q(i)$就是不成功检索的概率。显然,$\sum_{1 \leq i \leq n} P(i) + \sum_{0 \leq i \leq n} Q(i) = 1$。倘若已知这些数据,希望构造一棵对于$\{a_1, a_2, \cdots, a_n\}$最优的二分检索树。为此,应先说明最优二分检索树的含意。

为了得到二分检索树的成本函数,在这棵检索树的每一棵空子树的位置上加上一个虚构的结点,即外部结点,它们在图 6.6 中画成方格。所有其它的结点是内部结点。如果一棵二分检索树表示 n 个标识符,那么正好有 n 个内部结点和 n + 1 个(虚构的)外部结点。每个内部结点代表一次成功检索可能终止的位置。每个外部结点表示一次不成功检索可能终止的位置。

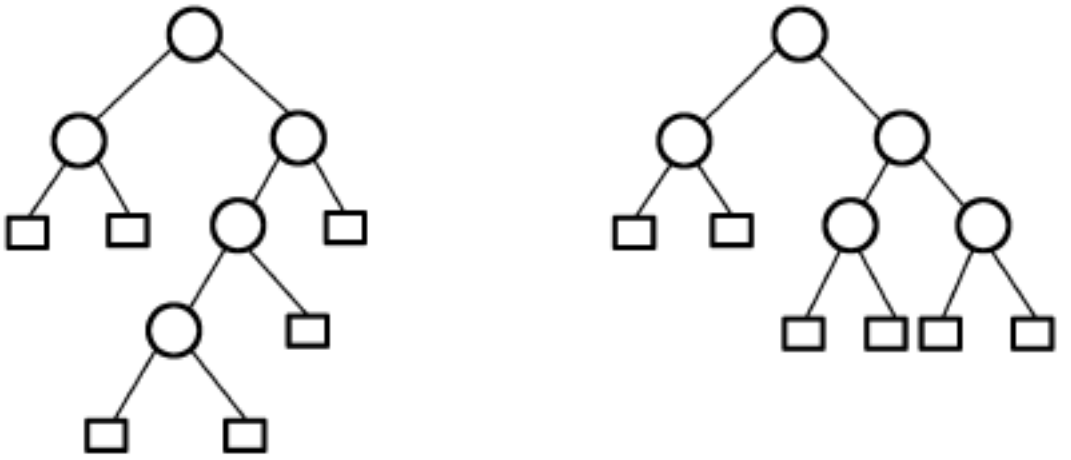


图 6.6　图 6.5 附加上外部结点后的二分检索树

如果一次成功的检索在 l 级的一个内结点处终止,则算法 6.4 的 2～8 行的循环需要作 l 次迭代。因此,内结点 a_i 所要求的成本分担额是 $P(i) * level(a_i)$。

不成功的检索在算法 SEARCH 中 i = 0(即在外结点处)的情况下终止。不在二分检索树中的标识符可以分成 n + 1 个等价类 E_i $(0 \leq i \leq n)$。E_0 包含所有使得 $X < a_0$ 的标识符 X;E_i 包含所有使得 $a_i < X < a_{i+1}$ $(1 \leq i < n)$的标识符 X;E_n 包含所有使得 $X > a_n$ 的标识符 X。易于看出,在同一类 E_i 中的所有标识符,其检索都在同一个外部结点处终止。而在不同的 E_i 中的标识符则在不同的外部结点处终止。如果 E_i 的那个失败的结点在 l 级,则只要对 while 循环作 l - 1 次迭代。于是,这个结点的成本分担额是 $Q(i) * (level(E_i) - 1)$。

上述讨论导出二分检索树的预期成本公式如下:

$$\sum_{1 \leq i \leq n} P(i) * level(a_i) + \sum_{0 \leq i \leq n} Q(i) * (level(E_i) - 1) \qquad (6.9)$$

定义标识符集$\{a_1, a_2, \cdots, a_n\}$的最优二分检索树是一棵使式(6.9)取最小值的二分检索树。

例 6.9　标识符集(a_1, a_2, a_3) = (do if stop)可能的二分检索树如图 6.7 所示。

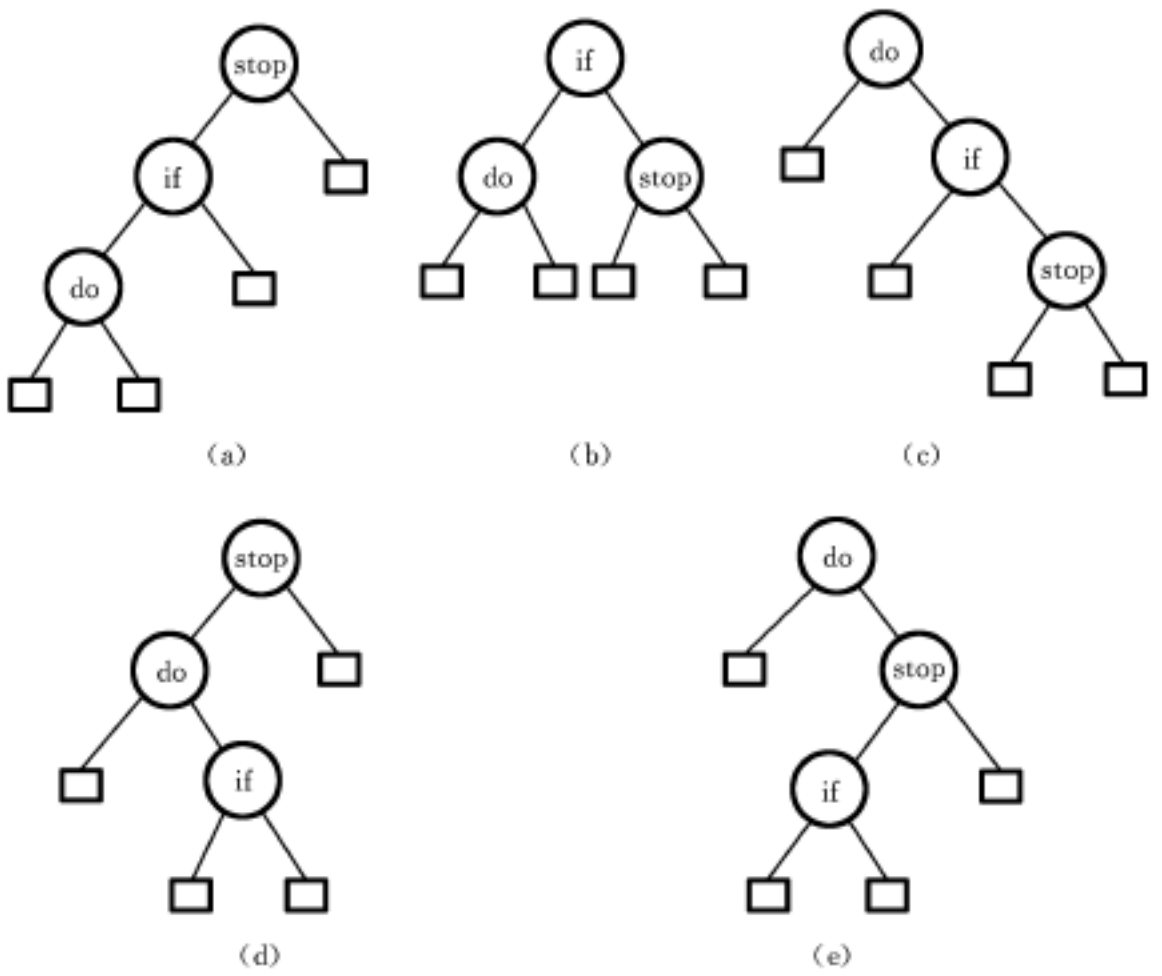


图 6.7　3 种标识符的各种二分检索树

在每个内、外结点具有相同概率 P(i) = Q(i) = 1/7 的情况下,有

cost(树 a) = 15/7　　cost(树 b) = 13/7　　cost(树 c) = 15/7

cost(树 d) = 15/7　　cost(树 e) = 15/7

正如料想的那样,树 b 是最优的。在 P(1) = 0.5, P(2) = 0.1, P(3) = 0.05, Q(0) = 0.15, Q(1) = 0.1, Q(2) = 0.05 和 Q(3) = 0.05 的情况下,则有

cost(树 a) = 2.65　　cost(树 b) = 1.9　　cost(树 c) = 1.5

cost(树 d) = 2.15　　cost(树 e) = 1.6

在以上情况下树 c 是最优的。

为了把动态规划应用于得到一棵最优二分检索树的问题,需要把构造这样的一棵树看成是一系列决策的结果,而且要能列出求取最优决策序列的递推式。解决上述问题的一种可能方法是,对于这些 a_i, $1 \le i \le n$,要决策出将其中的哪一个作为 T 的根结点。如果选择 a_k,那么,$a_1, a_2, \cdots, a_{k-1}$ 这些内部结点和 $E_0, E_1, \cdots, E_{k-1}$ 这些类的外部结点显然都将位于这个根的左子树 L 中,而其余的结点则将在右子树 R 中。定义

$$COST(L) = \sum_{1 \le i < k} P(i) * level(a_i) + \sum_{0 \le i < k} Q(i) * (level(E_i) - 1)$$

和

$$COST(R) = \sum_{k < i \le n} P(i) * level(a_i) + \sum_{k \le i \le n} Q(i) * (level(E_i) - 1)$$

在这两种情况中,级数的测定是把各个子树的根看成是第 1 级来进行的。

用 W(i,j)表示 $Q(i)+\sum_{i+1}^{l}(Q(l)+P(l))$的和,于是可以得到作为检索树 T(见图 6.8)的预期成本:

$$P(k)+COST(L)+COST(R)+W(0,k-1)+W(k,n) \tag{6.10}$$

如果 T 是最优的,则(6.10)式必定是最小值。从而,COST(L)对于包含 $a_1,a_2,\cdots,a_{k-1}$ 和 $E_0,E_1,\cdots,E_{k-1}$ 的所有二分检索树必定是最小值。同理 COST(R)也必定是最小值。如果用 C(i,j)表示包含 $a_{i+1},\cdots,a_j$ 和 $E_i,\cdots,E_j$ 的最优二分检索树的成本,那么,要 T 是最优的,就必须有 COST(L) = C(0,k-1)和 COST(R) = C(k,n)。而且 k 应该选成使得

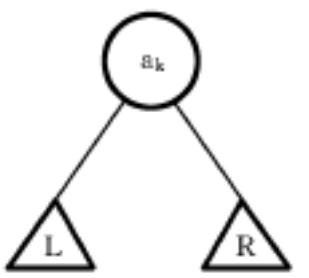


图 6.8　根为 a_k 的一棵最优二分检索树

$$P(k)+C(0,k-1)+C(k,n)+W(0,k-1)+W(k,n)$$

是最小值。因此,关于 C(0,n)有

$$C(0,n)=\min_{1\le k\le n}\{C(0,k-1)+C(k,n)+P(k)+W(0,k-1)+W(k,n)\} \tag{6.11}$$

将(6.11)一般化,对任何 C(i,j)则有

$$\begin{aligned}C(i,j)&=\min_{i<k\le j}\{C(i,k-1)+C(k,j)+P(k)+W(i,k-1)+W(k,j)\}\\&=\min_{i<k\le j}\{C(i,k-1)+C(k,j)\}+W(i,j)\end{aligned} \tag{6.12}$$

用下列步骤解式(6.12)可得 C(0,n),首先计算所有使得 j-i=1 的 C(i,j)(注意 C(i,i)=0 且 W(i,i)=Q(i),$0\le i\le n$),接着计算所有使得 j-i=2 的 C(i,j),然后计算 j-i=3 的所有C(i,j)等。如果在这一计算期间,记下每棵 T_{ij} 树的根 R(i,j),那么最优二分检索树就可以由这些 R(i,j)构造出来。要指出的是,R(i,j)是使式(6.12)取最小值的 k 值。

例 6.10　设 n=4,且(a_1,a_2,a_3,a_4)=(do if read while)。又设 P(1∶4)=(3,3,1,1)和 Q(0∶4)=(2,3,1,1,1)。为方便起见,这些 P 和 Q 都已乘了 16。最初,有 W(i,i)=Q(i),C(i,i)=0 和 R(i,i)=0,$0\le i\le 4$。使用式(6.12)和 W(i,j)=P(j)+Q(j)+W(i,j-1),得

$$W(0,1)=P(1)+Q(1)+W(0,0)=8$$
$$C(0,1)=W(0,1)+\min\{C(0,0)+C(1,1)\}=8$$
$$R(0,1)=1$$
$$W(1,2)=P(2)+Q(2)+W(1,1)=7$$
$$C(1,2)=W(1,2)+\min\{C(1,1)+C(2,2)\}=7$$
$$R(1,2)=2$$
$$W(2,3)=P(3)+Q(3)+W(2,2)=3$$
$$C(2,3)=W(2,3)+\min\{C(2,2)+C(3,3)\}=3$$
$$R(2,3)=3$$
$$W(3,4)=P(4)+Q(4)+W(3,3)=3$$
$$C(3,4)=W(3,4)+\min\{C(3,3)+C(4,4)\}=3$$
$$R(3,4)=4$$

知道了 W(i,i+1)和 C(i,i+1),$0\le i<4$,可以再一次利用式(6.12)去计算 W(i,i+2),C(i,i+2),R(i,i+2),$0\le i<3$。这一过程可以重复到得出 W(0,4),C(0,4)和 R(0,4)。表 6.2列出了这一系列计算的结果。第i行和第j列处的那一部分则分别列出了W(j,j+i),

表 6.2 计算 C(0,4),W(0,4)和 R(0,4)

行 \ 列	0	1	2	3	4
0	2, 0, 0	3, 0, 0	1, 0, 0	1, 0, 0	1, 0, 0
1	8, 8, 1	7, 7, 2	3, 3, 3	3, 3, 4	
2	12, 19, 1	9, 12, 2	5, 8, 3		
3	16, 25, 2	11, 19, 2			
4	16, 32, 2				

C(j,j+i)和 R(j,j+i)的值。按行方式由 0 到 4 行执行这一计算。从表中可以看到 C(0,4)是关于(a_1, a_2, a_3, a_4)的二分检索树的最小成本。树 T_{04} 的根是 a_2,因此,左子树是 T_{01} 而右子树是 T_{24}。T_{01} 有根 a_1 以及子树 T_{00} 和 T_{11}。T_{24} 有根 a_3,所以它的左子树是 T_{22},右子树是 T_{34}。因此,使用表中的数据,则可以绘出 T_{04} 的图形。图 6.9 显示了 T_{04}。

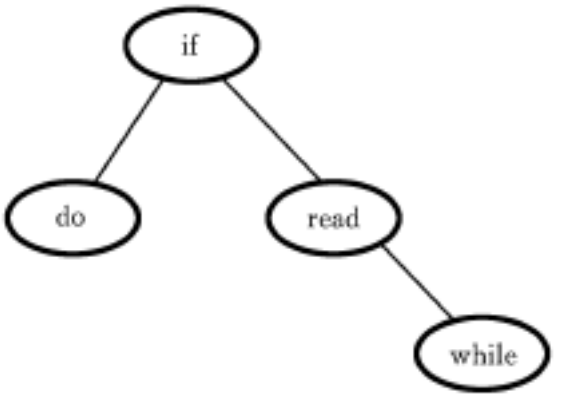


图 6.9 例 6.10 的最优检索树

上面的例子说明如何使用式(6.12)来确定这些 C 和 R,也指出了在知道 R 的情况下如何去构造 T_{0n}。现在来分析计算 C 和 R 的计算复杂度。上例中所描述的计算过程要求按(j-i)=1,2,…,n 这样的顺序去计算 C(i,j)。当 j-i=m 时有 n-m+1 个 C(i,j)要计算。每一个C(i,j)的计算要求找出 m 个量中的最小值(见式(6.12))。因此,每一个这样的C(i,j)能够在 O(m)时间内算出。所以,对于具有 j-i=m 的所有 C(i,j)总的计算时间是 $O(nm - m^2)$。计算所有的 C(i,j)和 R(i,j)的总时间

$$\sum_{1 \le m \le n} (nm - m^2) = O(n^3)$$

实际上,利用克努特(D.E.Knuth)所得到的一个结果可以比上面的更好一些。这个结果表明:最优的 k 可以通过把检索限制在区间 $R(i,j-1) \le k \le R(i+1,j)$ 求解式(6.12)而得到。在这种情况下,计算时间为 $O(n^2)$(见题 6.14)。过程 OBST(算法 6.5)使用这一结果在 $O(n^2)$时间内求得 W(i,j),R(i,j)和 C(i,j)的值,$0 \le i \le j \le n$。实际的树 T_{0n} 根据 R(i,j)的值可以在时间 O(n)内构造出来,至于构造树 T_0 的算法则留作一道习题。

算法 6.5 找最小成本二分检索树

```
procedure OBST(P,Q,n)
    // 给定几个互异的标识符 a1 < a2 < … < an。已知成功检索的概率 P(i),1≤i≤n,不成功检索
       概率 Q(i),0≤i≤n。此算法对标识符 a(i+1),…,aj 计算最优二分检索树 Tij 的成本 C(i,j),
       还计算 Tij 的根 R(i,j),Tij 的权 W(i,j) //
    real P(1:n),Q(0:n),C(0:n,0:n),W(0:n,0:n)
    integer R(0:n,0:n)
    for i←0 to n-1 do
        (W(i,i),R(i,i),C(i,i))←(Q(i),0,0)    // 置初值 //
        (W(i,i+1),R(i,i+1),C(i,i+1))←(Q(i)+Q(i+1)+P(i+1),
        i+1,Q(i)+Q(i+1)+P(i+1))    // 含一个结点的最优树 //
    repeat
(W(n,n),R(n,n),C(n,n))←(Q(n),0,0)
```

```
  for m←2 to n do   // 找有 m 个结点的最优树 //
    for i←0 to n - m do
      j←i + m
      W(i,j)←W(i,j - 1) + P(j) + Q(j)
      k←区间[R(i,j - 1),R(i+ 1,j)]中使{C(i,l - 1) + C(l,j)}取最小值的 l 值
                              // 用 Knuth 的结果解式(6 .12) //
      C(i,j)←W(i,j) + C(i,k - 1) + C(k,j)
      R(i,j)←k
    repeat
  repeat
end OBST
```

6 .5　0/ 1 背包问题

本节将使用向后处理法来求解 6 .1 节定义的 0/ 1 背包问题。对于 0/ 1 背包问题,可以通过作出变量 $x_1, x_2, \cdots, x_i$ 的一个决策序列来得到它的解。而对变量 x 的决策就是决定它们是取 0 值还是取 1 值。假定决策这些 x 的次序为 $x_n, x_{n-1}, \cdots, x_1$。在对 x_n 作出决策之后,问题处于下列两种状态之一:背包的剩余容量是 M,没产生任何效益;剩余容量是 M - w,效益值增长了 p。显然,余下来对 $x_{n-1}, x_{n-2}, \cdots, x_1$ 的决策相对于决策 x 所产生的问题状态应该是最优的,否则 $x_n, \cdots, x_1$ 就不可能是最优决策序列。如果设 $f_j(x)$ 是 KNAP(1, j, x) 最优解的值,那么,$f_n(M)$ 就可表示为

$$f_n(M) = \max\{f_{n-1}(M), f_{n-1}(M - w_n) + p_n\} \tag{6.13}$$

对于任意的 $f_i(x)$,这里 $i > 0$,则有

$$f_i(x) = \max\{f_{i-1}(x), f_{i-1}(x - w_i) + p_i\} \tag{6.14}$$

为了能由前向后递推而最后求解出 $f_n(M)$,需从 $f_0(x)$ 开始。对于所有的 $x \geq 0$,有 $f_0(x) = 0$,当 $x < 0$ 时,有 $f_0(x) = -\infty$。根据式(6 .14),马上可求解出 $0 \leq x < w_1$ 和 $x \geq w_1$ 情况下 $f_1(x)$ 的值。接着又可由式(6 .14)不断递推求出 $f_2, f_3, \cdots, f_n$ 在 x 相应取值范围内的值。

6 .5 .1　0 /1 背包问题的实例分析

例 6 .11　考虑以下情况的背包问题,n = 3,$(w_1, w_2, w_3) = (2,3,4)$,$(p_2, p_3, p_4) = (1,2,5)$ 和 M = 6。

利用式(6 .14)递推求解如下:

$$f_0(x) = \begin{cases} -\infty & x < 0 \\ 0 & x \geq 0 \end{cases}$$

$$f_1(x) = \begin{cases} -\infty & x < 0 \\ \max\{0, -\infty + 1\} = 0 & 0 \leq x < 2 \\ \max\{0, 0 + 1\} = 1 & x \geq 2 \end{cases}$$

$$f_2(x)=\begin{cases}-\infty & x<0\\ 0 & 0\leq x<2\\ 1 & 2\leq x<3\\ \max\{1,0+2\}=2 & 3\leq x<5\\ \max\{1,1+2\}=3 & x\geq 5\end{cases}$$

$$f_3(M)=\max\{3,1+5\}=6$$

因此,背包问题 KNAP(1,3,6)的最优解为 6。通过检查 f_i 的取值情况可以确定获得最优解的最优决策序列。x_3 的取值很容易确定,因为 $f_3(M)=6$ 是在 $x_3=1$ 的情况下取得的,所以 $x_3=1$。$f_3(M)-p_3=1$,$f_2(x)$和 $f_1(x)$都可取值 1,因此 $x_2=0$。f_0 不能取 1,故 $x_1=1$。于是最优决策序列$(x_1,x_2,x_3)=(1,0,1)$。

事实上,此问题用图解法求解是非常容易的。图 6.10 显示了 f_1,f_2 和 f_3 的图解过程。第一列的图给出了函数 $f_{i-1}(x-w_i)+p_i$ 的图像,将 $f_{i-1}(x)$在 x 轴上右移 w_i 个单位然后上移 p_i 个单位就得到它的图像。第二列给出由式(6.14)所得到的函数 $f_i(x)$,即它由 $f_{i-1}(x)$和 $f_{i-1}(x-w_i)+p_i$的函数曲线按 x 相同时取大值的规则归并而成。

由图 6.10 可以看出以下几点:每一个 f_i 完全由一些序偶(p_j,w_j)组成的集合所说明,其

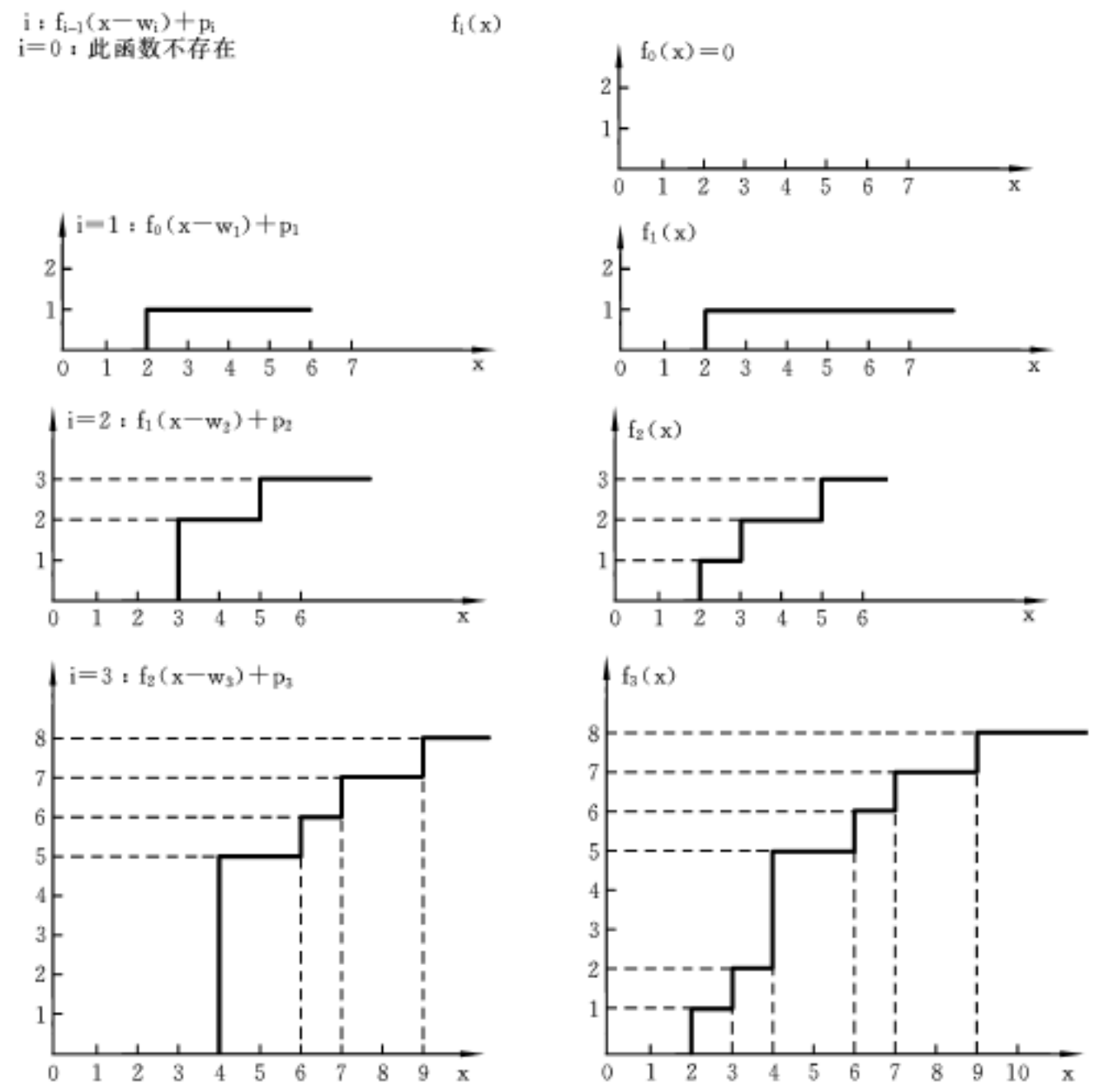


图 6.10　图解背包问题

中 w_j 是使 f_i 在其处产生一次阶跃的 x 值，$p_j = f_i(w_j)$。第一对序偶是 $(p_0, w_0) = (0,0)$。如果有 r 次阶跃，就还要知道 r 对序偶 (p_j, w_j)，$1 \leq j \leq r$。如果假定 $w_j < w_{j+1}$，$0 \leq j < r$，那么，由式(6.14)可得 $p_j < p_{j+1}$。此外，在 $0 \leq j < r$ 的情况下，对于所有使得 $w_j \leq x < w_{j+1}$ 的 x，有 $f_i(x) = f_i(w_j)$。而对于所有满足 $x \geq w_r$ 的 x，有 $f_i(x) = f_i(w_r)$。设 S^{i-1} 是 f_{i-1} 的所有序偶的集合，S_1^i 是 $f_{i-1}(x - w_i) + p_i$ 的所有序偶的集合。把序偶 (p_i, w_i) 加到 S^{i-1} 中，每一对序偶就得到 S_1^i。

$$S_1^i = \{(p,w) \mid (p - p_i, w - w_i) \in S^{i-1}\}。\tag{6.15}$$

在式(6.14)中，取 $f_{i-1}(x)$ 和 $f_{i-1}(x - w_i) + p_i$ 的最大值，在这里相当于在下述支配规则下将 S^{i-1} 和 S_1^i 归并成 S^i。如果 S^{i-1} 和 S_1^i 之一有序偶 (p_j, w_j)，另一有序偶 (p_k, w_k)，并且在 $w_j \geq w_k$ 的同时有 $p_j \leq p_k$，那么，序偶 (p_j, w_j) 被舍弃。显然，这就是(6.14)式的求最大值的运算。$f_i(w_j) = \max\{p_j, p_k\} = p_k$。

例 6.12　对于例 6.11 的数据，有

$$S^0 = \{(0,0)\} \qquad S_1^1 = \{(1,2)\}$$
$$S^1 = \{(0,0),(1,2)\} \qquad S_1^2 = \{(2,3),(3,5)\}$$
$$S^2 = \{(0,0),(1,2),(2,3),(3,5)\}$$
$$S_1^3 = \{(5,4),(6,6),(7,7),(8,9)\}$$
$$S^3 = \{(0,0),(1,2),(2,3),(5,4),(6,6),(7,7),(8,9)\}$$

根据支配规则，在 S^3 中删去了序偶(3,5)。

S^i 的上述计算过程也可由以下推理得出。在用直接枚举法求解 0/1 背包问题时，由于每个 x_i 的取值只能为 0 或 1，因此 $x_1, x_2, \cdots, x_n$ 有 2^n 个不同的 0、1 值序列。对于每一序列，若把 $\sum_1^n w_l x_l$ 记为 w_j，$\sum_1^n p_l x_l$ 记为 p_j，则此序列产生一对与之对应的序偶 (p_j, w_j)。在这 2^n 个序偶中，满足 $w_j \leq M$，且使 p_j 取最大值的序偶就是背包问题的最优解。在用动态规划的向后处理法求解 0/1 背包问题时，假定 S^{i-1} 是以下序偶所组成的集合，这些序偶是由 $x_1, x_2, \cdots, x_{i-1}$ 的 2^{i-1} 个决策序列中一些可能的序列所产生的序偶 (p_j, w_j)。那么，S^i 可按下述步骤得到。在 $x_i = 0$ 的情况下，产生的序偶集与 S^{i-1} 相同；而在 x_i 取 1 值的情况下，产生的序偶集是将 (p_i, w_i) 加到 S^{i-1} 的每一对序偶 (p_j, w_j) 得到的，这序偶集就是式(6.15)所描述的 S_1^i。然后，根据支配规则将 S^{i-1} 和 S_1^i 归并在一起就得到 S^i。这样归并的正确性证明如下：假定 S^{i-1} 和 S_1^i 之一包含 (p_j, w_j)，另一包含 (p_k, w_k)，并且有 $w_j \geq w_k$ 和 $p_j \leq p_k$。由于任何可行的子决策序列 $x_{i+1}, \cdots, x_n$ 都必须满足 $w_j + \sum_{i+1}^n w_l x_l \leq M$，当然也需满足 $w_k + \sum_{i+1}^n w_l x_l \leq M$。在这种情况下，$p_k + \sum_{i+1}^n p_l x_l \geq p_j + \sum_{i+1}^n p_l x_l$，它表明 (p_j, w_j) 导致的解比 (p_k, w_k) 能得到的最好解要差，因此根据支配规则在归并 S^{i-1} 和 S_1^i 时舍弃 (p_j, w_j) 是正确的。这里称 (p_k, w_k) 支配 (p_j, w_j)。此外，在生成序偶集 S^i 的时候，还应将 $W > M$ 的那些序偶 (p, w) 也清除掉，因为由它们不能导出满足约束条件的可行解。

这样生成的 S 的所有序偶，是背包问题 KNAP(1, i, x) 在 $0 \leq x \leq M$ 各种取值下的最优解(注意：S^i 是由一些序偶构成的有序集合)。通过计算 S^n 可以找到 KNAP(1, n, x)，$0 \leq x \leq M$ 的所有解。KNAP(1, n, M) 的最优解 $f_n(M)$ 由 S^n 的最后一对序偶的 P 值给出。

由于实际上只需要 KNAP(1, n, M)的最优解,它是由 S^n 的最末序偶(p, w) 给出的。而 S^n 的这最末序偶或者是 S^{n-1} 的最末序偶,或者是$(p_j + p_n, w_j + w_n)$,其中 $(p_j, w_j) \in S^{n-1}$ 且 w_j 是 S^{n-1} 中满足 $w_j + w_n \leq M$ 的最大值。因此,只需按上述方法求出 S^n 的最末序偶,无需计算出 S^n 也一样满足求 KNAP(1, n, M)最优解的要求。

如果已找出 S^n 的最末序偶(p_1, w_1),那么,使 $\sum p_i x_i = p_1$, $\sum w_i x_i = w_1$ 的 $x_1, \cdots, x_n$ 的决策值可以通过检索这些 S^i 来确定。若$(p_1, w_1) \in S^{n-1}$,则置 $x_n = 0$。若$(p_1, w_1) \mid S^{n-1}$,则$(p_1 - p_n, w_1 - w_n) \in S^{n-1}$,并且置 $x_n = 1$。然后,再判断留在 S^{n-1} 中的序偶(p_1, w_1)或者$(p_1 - p_n, w_1 - w_n)$ 是否属于 S^{n-2} 以确定 x_{n-1} 的取值,等等。

例 6.13　由例 6.12,在 M = 6 的情况下,$f_3(6)$的值由 S^3 中的序偶(6, 6)给出。$(6, 6) \mid S^2$,因此应置 $x_3 = 1$。序偶(6, 6)是由序偶$(6 - p_3, 6 - w_3) = (1, 2)$得到的,因此$(1, 2) \in S^2$。又,$(1, 2) \in S^1$,于是应置 $x_2 = 0$。但$(1, 2) \mid S^0$,从而得到 $x_1 = 1$。故最优解 $f_3(6) = 6$ 的最优决策序列是$(x_1, x_2, x_3) = (1, 0, 1)$。

对于以上所述的内容可以用一种非形式的算法过程 DKP 来描述。为了实现 DKP,需要给出表示序偶集 S^i 和 S_1^i 的结构形式,而且要将 MERGE-PURGE 过程具体化,使它能按要求归并 S^{i-1} 和 S_1^i,且清除一些序偶。此外,还要给出一个沿着 $S^{n-1}, \cdots, S^1$ 回溯以确定 $x_n, \cdots, x_1$ 的 0、1 值的算法。

算法 6.6　非形式化的背包算法

```
line procedure DKP(p, w, n, M)
1    S^0 ← {(0, 0)}
2    for i ← 1 to n - 1 do
3    S_1^i ← {(p_1, w_1) | (p_1 - p_i, w_1 - w_i) ∈ S^{i-1} and w_1 ≤ M}
4    S^i ← MERGE-PURGE(S^{i-1}, S_1^i)
5    repeat
6    (p_x, w_x) ← S^{n-1} 的最末序偶
7    (p_y, w_y) ← (p_1 + p_n, w_1 + w_n),其中,w_1 是 S^{n-1} 中使得 w + w_n ≤ M 的所有序偶中取最大值
     的 w // 沿 S^{n-1}, ⋯, S^1 回溯确定 x_n, x_{n-1}, ⋯, x_1 的取值 //
8    if p_x > p_y then x_n ← 0          // p_x 是 S^n 的最末序偶 //
9                  else x_n ← 1    // p_y 是 S^n 的最末序偶 //
10   endif
11   回溯确定 x_{n-1}, ⋯, x_1
12   end DKP
```

6.5.2　DKP 的实现

可以用两个一维数组 P 和 W 来存放所有的序偶(p, w)。p 的值放在 P 中,w 的值放在 W 中。序偶集 $S^0, S^1, \cdots, S^{n-1}$ 互相邻接地存放。各个集合可以用指针 F(i)来指示,这里 $0 \leq i \leq n$。对于 $0 \leq i < n$, F(i)指示 S 中第一个元素所在的位置。F(n)是 S^{n-1} 中最末元素的位置加 1。

例 6.14　使用上述表示,将例 6.12 的序偶集 S^0, S^1, S^2 以如下形式存放:

	1	2	3	4	5	6	7
P	0	0	1	0	1	2	3
W	0	0	2	0	2	3	5
	↑	↑		↑			↑
	F(0)	F(1)		F(2)			F(3)

在生成 S_1^i 的过程中可以同时将 S^{i-1} 和 S_1^i 进行归并和经过在支配规则下的序偶舍弃而生成 S^i。在 S^{i-1} 中,序偶是按 P 和 W 的递增次序排列的,因此 S^i 的序偶也将按这种次序生成。假定 S_1^i 生成的下一序偶是(p_Q, w_Q),那么 S^{i-1} 中 W 值小于 w_Q 的所有序偶都可直接记入 S^i,而(p_Q, w_Q)是否记入 S^i 则由支配规则来判断 S^{i-1} 中 W 值小于 w_Q 的序偶中是否有序偶支配(p_Q, w_Q)。若有,则(p_Q, w_Q)被舍弃;若没有,则将(p_Q, w_Q)并入 S^i。因此,并不需要直接存放 S_1^i 的附加空间。

过程 DKNAP 使用以上方法由 S^{i-1} 生成 S^i。这些 S^i 在 4～29 行的循环中生成。在每次迭代开始时,l = F(i - 1),即它是 S^{i-1} 中第一对序偶的指针,而 h 是 S^{i-1} 中最末序偶的指针。因此,h = next - 1。k 总是指向 S^{i-1} 中正在考虑是否并到 S^i 中去的序偶。第 6 行所置的 u 值提供了在 S_1^i 中不用产生的序偶指示的最小取值减 1,即它指出了对于 $u < j \leq h$ 的所有 w_j,有 $w_j + w_i > M$。因此,S_1^i 的序偶都是 $1 \leq j \leq u$ 的序偶$(P(j) + p_i, W(j) + w_i)$。这些序偶在 7～22 行的循环中生成,相应的归并工作也在这里进行。每次迭代首先生成一对序偶(pp,ww),接着将 S^{i-1} 中所有还没有被清除和归并到 S^i 中且有 W < ww 的序偶(p,w)都并入 S。要指出的是,这些序偶中没有一个可以被清除。第 13～15 行处理 S^{i-1} 中正在考虑的序偶的 W 值等于 ww 时的情况,比较 P 和 pp,将值小的对应序偶清除。第 16～18 行在 pp > P(next - 1)的情况下,把序偶(pp,ww) 加入到 S^i 中,否则(pp,ww)被清除。第 19～21 行清除 S^{i-1} 中所有能在此时清除且没有并入 S^i 的序偶。在 7～22 行将 S_1^i 归并到 S^i 以后,S^{i-1} 中可能剩下一些序偶需并入 S^i,此工作在第 23～26 行完成。要指出的是,由于第 19～21 行的处理,这些剩下的序偶没有一个能被清除。过程 PARTS(29 行)实现过程 DKP(算法6 .6)的第 6～11 行,具体实现留作一道习题。

算法 6 .7　0/ 1 背包问题的算法

```
line procedure DKNAP (p,w,n,M,m)
     real p(n),w(n),P(m),W(m),pp,ww,M
     integer F(0;n),l,h,u,i,j,p,next
1     F(0)←1; P(1)←W(1)←0    // S^0 //
2     l←h←1    // S^0 的首端与末端//
3     F(1)←next←2    // P 和 W 中第一个空位//
4     for i←1 to n - 1 do    // 生成 S^i //
5          k←l
6          u←在 l≤r≤h 中使得 W(r) + w_i ≤ M 的最大的 r
7          for j←l to u do    // 生成 S_1^i 及归并//
8               (pp,ww)←(P(j) + p_i,W(j) + w_i)    // S_1^i 中的下一个元素//
9               while k≤h and W(k) < ww do    // 从 S^{i-1} 中取元素来归并//
10                P(next)←P(k);W(next)←W(k)
```

```
11          next←next + 1;k←k + 1
12          repeat
13          if k≤h and W(k) = ww then pp←max(pp,P(k))
14                                    k←k + 1
15        endif
16        if pp > P(next - 1) then (P(next),W(next))←(pp,ww)
17                            next←next + 1
18        endif
19            while k≤h and P(k)≤P(next - 1)do   // 清除 //
20            k←k + 1
21           repeat
22        repeat
          // 将 S^(i-1) 中剩余的元素并入 S^i //
23        while k≤h do
24              (P(next),W(next))←(P(k),W(k))
25              next←next + 1;k←k + 1
26        repeat
          // 对 S^(i+1) 置初值 //
27        l←h + 1;h←next - 1;F(i + 1)←next
28        repeat
29        call PARTS
30        end DKNAP
```

6.5.3 过程 DKNAP 的分析

假设 S^i 的序偶数是 $|S^i|$。在 $i>0$ 的情况下,每个 S^i 由 S^{i-1} 和 S_1^i 归并而成,并且 $|S_1^i| \leq |S^{i-1}|$,因此 $|S^i| \leq 2|S^{i-1}|$。在最坏情况下没有序偶会被清除,所以

$$\sum_{0 \leq i \leq n-1} |S^i| = \sum_{0 \leq i \leq n-1} 2^i = 2^n - 1$$

即,DKNAP 的空间复杂度为 $O(2^n)$。

由 S^{i-1} 生成 S^i 需要 $\Theta(|S^{i-1}|)$这么多时间,因此计算 $S^0, S^1, \cdots, S^{n-1}$ 总的时间是 $\Theta(\sum |S^{i-1}|)$。由于 $|S^i| \leq 2^i$,所以计算这些 S^i 总的时间是 $O(2^n)$。

如果每件物品的重量 w_j 和所产生的效益值 p_j 都是整数,那么 S^i 中每个序偶 (p,w) 的 P 和 W 也是整数,且有 $P \leq \sum_{1 \leq j \leq i} p_j$, $W \leq M$。又由于在任一 S^i 中,它的序偶有互异的 P 值,也有互异的 W 值,因此有

$$|S^i| \leq 1 + \sum_{1 \leq j \leq i} p_j \quad \text{和} \quad |S^i| \leq 1 + \min\{\sum_{1 \leq j \leq i} w_j, M\}$$

于是,在所有 w_j 和 p_j 为整数的情况下,DKNAP 的时间和空间复杂度(除去 PARTS 的时间)是 $O(\min\{2^n, n \sum_{1 \leq i \leq n} p_i, n\ M\})$。

以上分析似乎表明,当 n 取值大时 DKNAP 的有效性是令人失望的,但这类问题在很多情况下都能在“适当的”时间内解出。这是因为在很多情况下 P 和 W 都是整数,而且 M 比 2^n 小得多。另外,支配规则在清除不应并入 S^i 的序偶上是很有效的。

使用探试方法 (heuristic) 可以加速 DKNAP 的计算。设 L 是最优解的估计值,它使得

$f_n(M) \geq L$，又设 $PLEFT(i) = \sum_{1<j\leq n} p_j$。如果序偶 $(p,w) \in S^i$，且使得 $P + PLEFT(i) < L$，那么 (p,w) 就可从 S^i 中清除掉。这是因为 (p,w) 充其量只能对 S_1^n 产生 $(P + \sum_{i<j\leq n} p_j, W + \sum_{i<j\leq n} w_j)$ 这样的序偶，而 $P + \sum_{i<j\leq n} p_j = P + PLEFT(i) < L$，因此序偶 (p,w) 不可能导致最优解 $f_n(M)$，故可从 S^i 删去。背包问题的启发性方法还要在分枝-限界这一章详细讨论。找小于等于 $f_n(M)$ 的估计值 L 的一种简单方法是取 S^i 的最末序偶 (p,w) 的 P 作为 L。此时，$P \leq f_n(M)$。更好的一种估计值是将某些剩余物品的 p 值与这个 P 值加在一起作为 L。例 6.15 用的就是这种取值方法。

例 6.15　考虑下述情况的 0/1 背包问题：$n = 6$，$(p_1, p_2, p_3, p_4, p_5, p_6) = (w_1, w_2, w_3, w_4, w_5, w_6) = (100, 50, 20, 10, 7, 3)$，$M = 165$。

如果将物品 1,2,4,6 装入背包，占用 163 的容量产生 163 的效益，取估计值 $L = 163$。由于本例中 $p_i = w_i$，所以每对属于 S^i 的序偶 (p,w) 都有 $P = W$，这里 $0 \leq i \leq 6$。于是每对序偶 (p,w) 可以用单一量 P(或 W)来代替。

$PLEFT(0) = 190$；$PLEFT(1) = 90$；$PLEFT(2) = 40$；$PLEFT(3) = 20$；$PLEFT(4) = 10$；$PLEFT(5) = 3$ 和 $PLEFT(6) = 0$。从 S^i 中删去所有使 $P + PLEFT(i) < L$ 的 P，得到

$$S^0 = \{0\} \qquad S_1^1 = \{100\}$$
$$S^1 = \{100\} \qquad S_1^2 = \{150\}$$
$$S^2 = \{150\} \qquad S_1^3 = \phi$$
$$S^3 = \{150\} \qquad S_1^4 = \{160\}$$
$$S^4 = \{160\} \qquad S_1^5 = \phi$$
$$S^5 = \{160\}$$

因 $0 + PLEFT(1) = 90 < 163$，故 S^1 中删去了 0。$100 + PLEFT(2) < 163$，故 S^2 已删去了 100。S_1^3 不包含 $150 + 20 = 170$，这是因为 $170 > M$。$f_6(165)$ 可由 S^5 求出，它等于 $160 + p_6 = 163$。

此例中 L 值一直没有改变。在一般情况下，如果计算某 S^i 会产生更好的估计值，L 值将会改变。如果不使用启发性方法，由 DKNAP 就会产生如下结果：

$$S^0 = \{0\}$$
$$S^1 = \{0, 100\}$$
$$S^2 = \{0, 50, 100, 150\}$$
$$S^3 = \{0, 20, 50, 70, 100, 120, 150\}$$
$$S^4 = \{0, 10, 20, 30, 50, 60, 70, 80, 100, 110, 120, 130, 150, 160\}$$
$$S^5 = \{0, 7, 10, 17, 20, 27, 30, 37, 50, 57, 60, 67, 70, 77, 80, 87, 100, 107, 110, 117, 120, 127, 130, 137, 150, 157, 160\}$$

$f_6(165)$ 可用已知条件 $(p_6, w_6) = (3, 3)$ 从 S^5 求出。

6.6　可靠性设计

乘积函数最优化问题，也可使用动态规划法来求解。本节讨论这类问题中的一个实例。假定要设计一个系统，这个系统由若干个以串联方式连接在一起的不同设备所组成(见图

6 .11) 。设 r_i 是设备 D_i 的可靠性(即 r_i 是设备 D_i 正常运转的概率),则整个系统的可靠性就是 $\prod r_i$。即便这些单个设备是非常可靠的(每个 r_i 都非常接近于 1),该系统的可靠性也不一定很高。例如,若 $n = 10, r_i = 0.99, 1 \leq i \leq 10$,则 $\prod r_i = 0.904$ 。为了提高系统可靠性,最好是增加一些重复设备,并通过开关线路把数个同类设备并联在一起 (见图 6 .12)。由开关线路来判明其中的设备的运行情况,并将能正常运行的某台投入使用。

图 6 .11 以串联方式连接的 n 台设备

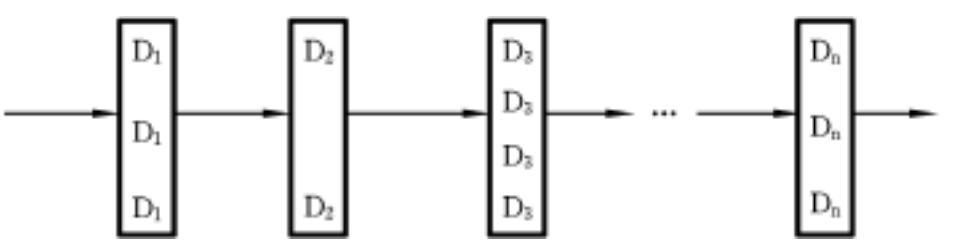


图 6 .12 每级中以并联方式连接多台设备

如果第 i 级的设备 D_i 的台数为 m_i,那么这 m_i 台设备同时出现故障的概率为 $(1 - r_i)^{m_i}$。从而第 i 级的可靠性就变成 $1 - (1 - r_i)^{m_i}$。例如,假定 $r_i = 0.99$, $m_i = 2$,于是这一级的可靠性就是 0 .9999。不过在任何实际系统中,每一级的可靠性要比 $1 - (1 - r_i)^{m_i}$ 小一些,这是由于这些开关线路本身并不是完全可靠的,而且同一类设备的失误率也不可能是完全独立的(例如,由于设计不当所造成的失误)。基于以上分析,不妨假设第 i 级的可靠性由函数 $\phi(m_i)$给定,$1 \leq i \leq n$ 。并且可以看出,开始时 $\phi(m_i)$随 m_i 值的增大而增大,在到达 m_i 的某个取值以后 $\phi(m_i)$ 的值则可能下降。这个多级系统的可靠性是 $\prod \phi(m_i)$。

诚然,增加重复的设备可提高系统的可靠性,但在实际工作中,设计一个系统都是在有成本约束的条件下实现的。因此,所谓可靠性设计最优化问题是在可容许最大成本 c 的约束下,如何使系统的可靠性达到最优的问题。假设 c_j 是一台设备 j 的成本,由于系统中每种设备至少有一台,故设备 j 允许配置的台数至多为

$$u_j = \lfloor (c + c_j - \sum_{k=1}^{n} c_k) / c_j \rfloor$$

如果用 RELI(l, i, X)表示在可容许成本 X 约束下,对第 l 种到第 i 种设备的可靠性设计问题,它就可形式描述成

极大化 $$\prod_{1 \leq j \leq i} \phi(m_i)$$

约束条件 $$\sum_{1 \leq j \leq i} c_j m_j \leq X \qquad (6.16)$$

其中,$c_j > 0, 1 \leq m_j \leq u_j = \lfloor (X + c_j - \sum_{k=1}^{i} c_k) / c_j \rfloor$ 且 m_j 为整数,$l \leq j \leq i$。

于是,整个系统的可靠性设计问题由 RELI(1, n, c)表示。它的一个最优解是对 $m_1, \cdots, m_n$ 的一系列决策的结果。每得到一个 m_i 都要对其取值进行一次决策。设 $f_i(x)$ 是在容许成本值 X 约束下对前 i 种设备组成的子系统可靠性设计的最优值,即 $f_i(x) = \max \prod_j (m_j)$,那么最优解的可靠性是 $f_n(c)$。所作的最后决策要求从 $\{1, 2, \cdots, u_n\}$中选择一个元素作为 m_n。一旦选出了 m_n 的值,则下阶段的决策就应使剩余资金 $c - c_n m_n$ 按最优的方

式使用。于是,有

$$f_n(c) = \max\{\phi(m_n) f_{n-1}(c - c_n m_n)\} \tag{6.17}$$

将(6.17)式推广到一般,对任一 $f_i(x)$, $i \geq 1$,有

$$f_i(x) = \max\{\phi(m_i) f_{i-1}(x - c_i m_i)\} \tag{6.18}$$

显然,当 $0 \leq x \leq c$ 时,对于所有的 x,有 $f_0(x) = 1$。使用类似于解 0/1 背包问题的方法可以解出(6.18)式。设 S^i 由(f,x)形式的序偶所组成,其中 $f = f_i(x)$。由 $m_1, \cdots, m_i$ 的决策序列所得出的每一个不同的 x 都至多只有一个序偶。支配规则对这个问题也适用,即当且仅当 $f_1 \geq f_2$ 而 $x_1 \leq x_2$ 时,(f_1, x_1)支配(f_2, x_2)。那些受支配的序偶可从 S^i 中舍去。

例 6.15　设计一个由设备 D_1, D_2, D_3 组成的三级系统。每台设备的成本分别为 30 元,15 元和 20 元,可靠性分别是 0.9,0.8 和 0.5,计划建立该系统的投资不得超过 105 元。假定,若 i 级有 m_i 台设备 D_i 并联,则该级的可靠性 $\phi(m_i) = 1 - (1 - r_i)^{m_i}$。上述条件可以表示为:$c = 105$; $c_1 = 30, c_2 = 15, c_3 = 20$; $r_1 = 0.9, r_2 = 0.8, r_3 = 0.5$。由此立即可得:$u_1 = 2, u_2 = 3, u_3 = 3$。

用 S^i 表示 $m_1, \cdots, m_i$ 的各种决策序列产生的所有不受支配的序偶(f,x)之集合。这里 $f = f_i(x)$。整个工作从 $S^0 = \{(1,0)\}$开始。假设已求出 S^{i-1}。S 的求取可按以下步骤进行,对于 m_i 的所有可能值,依次求出当 $m_i = j, 1 \leq j \leq u_i$ 时,由 S^{i-1} 可能得到的所有序偶的集合 S_j^i,然后将这 u_i 个 S_j^i 按支配规则归并即得 S^i。于是,由

$$S_1^1 = \{(0.9,30)\} \quad S_2^1 = \{(0.99,60)\}$$

得　$S^1 = \{(0.9,30),(0.99,60)\}$

由　$S_1^2 = \{(0.72,45),(0.792,75)\} \quad S_2^2 = \{(0.864,60)\} \quad S_3^2 = \{(0.8928,75)\}$

得　$S^2 = \{(0.72,45),(0.864,60),(0.8928,75)\}$

注意:S_2^2 中已删去了由(0.99,60)所得到的序偶(0.9504,90)。因为这只剩下 15 元,不足以让 $m_3 = 1$。

说明:归并时由于(0.792,75)受(0.864,60)支配,故舍去。

由　$S_1^3 = \{(0.36,65),(0.432,80),(0.4464,95)\}$

$S_2^3 = \{(0.54,85),(0.648,100)\}$

$S_3^3 = \{(0.63,105)\}$

得　$S^3 = \{(0.36,65),(0,432,80),(0.54,85),(0.648,100)\}$

最优设计有 0.648 的可靠性,需用资金 100 元。通过对这些 S^i 的回溯,求出 $m_1 = 1, m_2 = 2, m_3 = 2$。

对于可靠性设计问题,值得注意的是,正如上例中已作的处理那样,在 S^i 中不需要保留任何比 $c - \sum_{1 \leq j \leq n} c_j$ 还大的 x 值的序偶,这是因为任何序偶都不能超出完成这个系统所能负担的资金。此外,为了减少这些 S^i 的规模,可以像求解背包问题一样,将启发性方法引进求解可靠性问题的动态规划算法,即提供某种判定下界,如果(f,x) 小于它,则将(f,x)从 S 中删去。

6.7 货郎担问题

货郎担问题属于易于描述但难于解决的著名难题之一,至今世界上还有不少人在研究它。该问题的基本描述是:某售货员要到若干个村庄售货,各村庄之间的路程是已知的,为了提高效率,售货员决定从所在商店出发,到每个村庄售一次货然后返回商店,问他应选择一条什么路线才能使所走的总路程最短?此问题可描述如下:设 G = (V, E)是一个具有边成本 c_{ij} 的有向图,c_{ij} 的定义如下:对于所有的 i 和 j,$c_{ij} > 0$,若 $\langle i, j\rangle \mid E$,则 $c_{ij} = \infty$。令 |V| = n,并假定 n > 1。G 的一条周游路线是包含 V 中每个结点的一个有向环。周游路线的成本是此路线上所有边的成本和。货郎担问题 (traveling salesperson problem)是求取具有最小成本的周游路线问题。

有很多实际问题可归结为货郎担问题。例如,邮路问题就是一个货郎担问题。假定有一辆邮车要到 n 个不同的地点收集邮件,这种情况可以用 n + 1 个结点的图来表示。一个结点表示此邮车出发并要返回的那个邮局,其余的 n 个结点表示要收集邮件的 n 个地点。由地点 i 到地点 j 的距离则由边 $\langle i, j\rangle$ 上所赋予的成本来表示。邮车所行经的路线是一条周游路线,希望求出具有最小长度的周游路线。

第二个例子是在一条装配线上用一个机械手去紧固待装配部件上的螺帽问题。机械手由其初始位置(该位置在第一个要紧固的螺帽的上方)开始,依次移动到其余的每一个螺帽,最后返回到初始位置。机械手移动的路线就是以螺帽为结点的一个图中的一条周游路线。一条最小成本周游路线将使这机械手完成其工作所用的时间取最小值。

注意:只有机械手移动的时间总量是可变化的。

第三个例子是产品的生产安排问题。假设要在同一组机器上制造 n 种不同的产品,生产是周期性进行的,即在每一个生产周期这 n 种产品都要被制造。要生产这些产品有两种开销,一种是制造第 i 种产品时所耗费的资金($1 \le i \le n$),称为生产成本,另一种是这些机器由制造第 i 种产品变到制造第 j 种产品时所耗费的开支 c_{ij},称为转换成本。显然,生产成本与生产顺序无关。于是,我们希望找到一种制造这些产品的顺序,使得制造这 n 种产品的转换成本和为最小。由于生产是周期进行的,因此在开始下一周期生产时也要开支转换成本,它等于由最后一种产品变到制造第一种产品的转换成本。于是,可以把这个问题看成是一个具有 n 个结点,边成本为 c_{ij} 的图的货郎担问题。

货郎担问题要从图 G 的所有周游路线中求取具有最小成本的周游路线,而由始点出发的周游路线一共有(n - 1)!条,即等于除始结点外的 n - 1 个结点的排列数,因此货郎担问题是一个排列问题。排列问题比子集合的选择问题(例如,0/1 背包问题就是这类问题)通常要难于求解得多,这是因为 n 个物体有 n!种排列,只有 2^n 个子集合($n! > O(2^n)$)。通过枚举(n - 1)!条周游路线,从中找出一条具有最小成本的周游路线的算法,其计算时间显然为 O(n!)。为了改善计算时间必须考虑新的设计策略来拟制更好的算法。动态规划就是待选择的设计策略之一。但货郎担问题是否能使用动态规划设计求解呢?下面就来讨论这个问题。

不失一般性,假设周游路线是开始于结点 1 并终止于结点 1 的一条简单路径。每一条

周游路线都由一条边〈1,k〉和一条由结点 k 到结点 1 的路径所组成,其中 k∈V - {1};而这条由结点 k 到结点 1 的路径通过 V - {1,k}的每个结点各一次。容易看出,如果这条周游路线是最优的,那么这条由 k 到 1 的路径必定是通过 V - {1,k}中所有结点的由 k 到 1 的最短路径,因此最优性原理成立。设 g(i,S)是由结点 i 开始,通过 S 中的所有结点,在结点 1 终止的一条最短路径长度。g(1,V - {1})是一条最优的周游路线长度。于是,可以得出

$$g(1,V-\{1\})=\min_{2\le k\le n}\{c_{1k}+g(k,V-\{1,k\})\} \tag{6.19}$$

将式(6.19)一般化可得

$$g(i,S)=\min_{j\in S}\{c_{ij}+g(j,S-\{j\})\} \tag{6.20}$$

如果对于 k 的所有选择,知道了 g(k,V - {1,k})的值,由式(6.19)就可求解出 g(1,V - {1})。而这些 g 值则可通过式(6.20)得到。显然,g(i,) = c_{i1},1 < i≤n。于是,可应用式(6.20)去获得元素个数为 1 的所有 S 的 g(i,S)。然后,可对|S| = 2 的 S 得到 g(i,S)等等。当|S| < n - 1 时,g(i,S)所需要的 i 和 S 的值是 i≠1,1| S 且 i| S 的那些值。

例 6.17　考虑图 6.13(a),其边长由图 6.13(b)中的矩阵 C 给出:

$g(2,) = c_{21} = 5$　　$g(3,) = c_{31} = 6$　　$g(4,) = c_{41} = 8$

由式(6.20)得

$g(2,\{3\}) = c_{23} + g(3,) = 15$　　$g(2,\{4\}) = 18$　　$g(3,\{2\}) = 18$

$g(3,\{4\}) = 20$　　$g(4,\{2\}) = 13$　　$g(4,\{3\}) = 15$

接着,计算在|S| = 2 且 i≠1,1| S,i| S 情况下的 g(i,S):

$g(2,\{3,4\}) = \min\{c_{23} + g(3,\{4\}), c_{24} + g(4,\{3\})\} = 25$

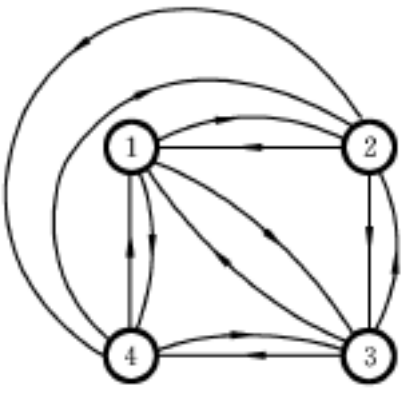
$g(3,\{2,4\}) = \min\{c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})\} = 25$

$g(4,\{2,3\}) = \min\{c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})\} = 23$

最后,由式(6.19)得

$g(1,\{2,3,4\}) = \min\{c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\})\}$

$= \min\{35,40,43\} = 35$



(a)

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

(b)

图 6.13　有向图和边长矩阵

图 6.13(a)中的这个有向图的最优周游路线长度为 35。如果对每个 g(i,S),保留使式(6.20)右边取最小值的那个 j 值,那么就可构造出这一最优周游路线。令 J(i,S)是这个值,则 J(1,{2,3,4}) = 2。它表示这条周游路线由 1 开始并通到 2。剩下的路径可由 g(2 ,{3,4})得到。因 J(2,{3,4}) = 4,故这下一条边是〈2,4〉。以后的剩余段由 g(4,{3})

来获得。J(4,{3}) = 3。因此这条最优周游路线是 1,2,4,3,1。

设 N 是用式(6 .19)计算 g(1,V - {1})以前需要计算的 g(i,S)的数目。|S|的取值在不同的决策阶段分别对应为 0,1,…,n - 2 。对于|S|的每一种取值,i 都有 n - 1 种选择。不包含 1 和 i 在内的大小为 k 的不同 S 的个数是$\binom{n-2}{k}$。因此,

$$N = \sum_{k=0}^{n-2}(n-1)\binom{n-2}{k} = (n-1)2^{n-2}$$

又因为在|S| = k 时,由式(6 .20)求 g(i,S)的值要进行 k - 1 次比较,所以用式(6 .19)和式(6 .20)求最优周游路线的算法要求的计算时间为 $\Theta(n^2 2^n)$ 。由此可知,用动态规划设计的算法比枚举法求最优周游路线在所用的时间上有所改进,但这种方法的严重缺点是空间要求要有$O(n2^n)$,这太大了,因而实际上是不可取的。在以后的章节里,将对货郎担问题作进一步的讨论。

6 .8 流水线调度问题

处理一个作业通常需要执行若干个不同的任务。对于在多道程序环境中运行的一些计算机程序,也需要执行输入、运算,然后排队打印输出等任务。假设在一条流水线上有 n 个作业,每个作业要求执行 m 个任务: T_{1i}, T_{2i}, …, T_{mi}, $1 \le i \le n$。并且,任务 T_{ji} 只能在设备 P_j 上执行,$1 \le j \le m$。除此之外,还假定对于任一作业 i,在任务 $T_{j-1,i}$ 没完成以前,不能对任务 T_{ji} 开始处理,而且同一台设备在任何时刻不能同时处理一个以上的任务。如果假设完成任务 T_{ji} 所要求的时间是 t_{ji}, $1 \le j \le m$, $1 \le i \le n$,那么如何将这 n×m 个任务分配给这 m 台设备,才能使这 n 个作业在以上要求下顺利完成呢?这就是流水线作业调度问题。

例 6 .18 考虑在 3 台设备上调度两个作业,每个作业包含 3 项任务 ,完成这些任务要求的时间由矩阵 J 给出, $J = \begin{bmatrix} 2 & 0 \\ 3 & 3 \\ 5 & 2 \end{bmatrix}$。这两个作业的两种可能调度如图 6 .14 所示。

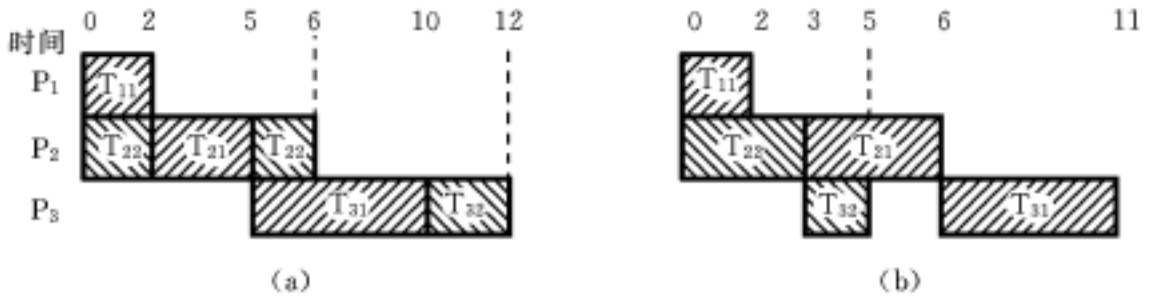


图 6 .14 例 6 .18 两种可能的调度

流水线上的作业调度,有两种基本方式:一种是非抢先调度,它要求在任何一台设备上处理一个任务一直到完成才能处理另一任务。另一种是抢先调度,它没有以上要求,图 6 .14(a)表示一种抢先调度,而图(b)则表示一种非抢先调度。作业 i 的完成时间 $f_i(S)$ 是在 S 调度方案下作业 i 的所有任务得以完成的时间。在图 6 .14(a)中 $f_1(S) = 10$, $f_2(S) = 12$。在图 6 .14(b)中, $f_1(S) = 11$, $f_2(S) = 5$。调度 S 的完成时间 F(S)由下式给出:

$$F(S) = \max_{1 \le i \le n}\{f_i(S)\} \tag{6.21}$$

平均流动时间 MFT(S)定义为

$$MFT(S) = \frac{1}{n}\sum_{1 \le i \le n} f_i(S) \tag{6.22}$$

一组给定作业的最优完成时间 OFT 调度是一种非抢先调度 S,它对所有非抢先调度而言,F(S)的值最小。可以很容易地给出抢先最优完成时间(POFT),最优平均完成时间(OMFT)和抢先最优平均完成时间(POMFT)等调度的相应定义。

当 m > 2 时,得到 OFT 和 POFT 调度的一般问题是难于计算的问题,得到 OMFT 调度的一般问题也是难于计算的问题(见第 10 章)。本节只讨论当 m = 2 时获取 OFT 调度这种特殊情况的算法。

为方便起见,用 a_j 表示 t_{1i}, b_j 表示 t_{2i}。在两台设备情况下容易证明:在两台设备上处理的任务若不按作业的排列次序处理,则在调度完成时间上不比按次序处理弱(注意:若 m > 2 则不然)。因此,调度方案的好坏完全取决于这些作业在每台设备上被处理的排列次序。当然,每个任务都应在最早的可能时间开始执行。图 6.15 所示的调度就完全由作业的排列次序 5,1,3,2,4 所确定。为讨论简单起见,假定 $a_i \neq 0, 1 \le i \le n$。事实上,如果允许有 $a_i = 0$ 的作业,那么最优调度可通过下法构造出来:首先对于所有 $a_i \neq 0$ 的作业求出一种最优调度的排列,然后把所有 $a_i = 0$ 的作业以任意次序加到这一排列的前面。

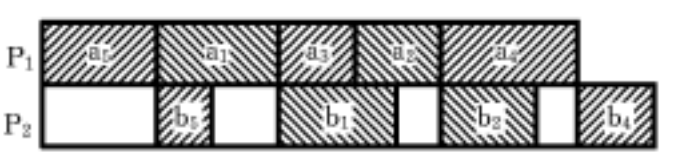


图 6.15　一种调度

容易看出,最优调度的排列具有下述性质:在给出了这个排列的第一个作业后,剩下的排列相对于这两台设备在完成第一个作业时所处的状态而言是最优的。假设对作业 1,2,…,k 的一种调度排列为 $\sigma_1, \sigma_2, \cdots, \sigma_k$。对于这种调度,设 h_1 和 h_2 分别是在设备 P_1 和 P_2 上完成任务($T_{11}, T_{12}, \cdots, T_{1k}$)和($T_{21}, T_{22}, \cdots, T_{2k}$)的时间;又设 $t = h_2 - h_1$,那么,在对作业 1,2,…,k 作了一系列决策之后,这两台设备所处的状态可以完全由 t 来表征。它表示如果要在设备 P_1 和 P_2 上处理一些别的作业,则必须在这两台设备同时处理前 k 个作业的不同的任务后,设备 P_2 还要用大小为 t 的时间段处理前 k 个作业中没处理完的任务,即在 t 这段时间及其以前,设备 P_2 不能用来处理别的作业的任务。记这些别的作业组成的集合为 S,假设 g(S,t) 是上述 t 下 S 的最优调度长度,于是作业集合{1,2,…,n}的最优长度是 g({1,2,…,n},0)。

由最优调度排列所具有的性质可得

$$g(\{1,2,\cdots,n\},0) = \min_{1 \le i \le n}\{a_i + g(\{1,2,\cdots,n\} - \{i\}, b_i)\} \tag{6.23}$$

将式(6.23)推广到一般情况,对于任意的 S 和 i 可得式(6.24)。式中,要求 g(,t) = max{t,0}且 $a_i \neq 0, 1 \le i \le n$。

$$g(S,t) = \min_{i \in S}\{a_i + g(S - \{i\}, b_i + \max\{t - a_i, 0\})\} \tag{6.24}$$

式(6.24)中 max{t - a_i,0}这一项由以下推导得出。由于任务 T_2 在 max{a_j,t}这段时间及其以前不能用设备 P_2 处理,因此,$h_2 - h_1 = b_j + \max\{a_i, t\} - a_i = b_i + \max\{t - a_i, 0\}$。

本来,可以使用一种与求解式(6.20)类似的方法求解 g(S,t),但它有这么多不同的 S,

而对这些 S 都要计算 $g(S,t)$,因此这样计算最优调度长度 $g(\{1,2,\cdots,n\},0)$ 至少要花费 $O(2^n)$的时间。下面介绍另一种代数方法来求解式(6.24),可得到一组非常简单的规则,使用这组规则可以在 $O(n\log n)$的时间内产生最优调度。

考虑这些作业的一子集 S 的任意一种调度 R。假设直到 t 这段时间 P_2 都不能用来处理 S 中的作业。如果 i 和 j 是这调度中排在前面的两个作业,则由式(6.24)就得到

$$g(S,t)=a_i+g(S-\{i\},b_i+\max\{t-a_i,0\})$$

$$=a_i+a_j+g(S-\{i,j\},b+\max\{b_i+\max\{t-a_i,0\}-a_j,0\}) \quad (6.25)$$

令

$$t_{ij}=b_j+\max\{b_i+\max\{t-a_i,0\}-a_j,0\}$$

$$=b_j+b_i-a_j+\max\{\max\{t-a_i,0\},a_j-b_i\}$$

$$=b_j+b_i-a_j+\max\{t-a_i,a_j-b_i,0\}$$

$$=b_j+b_i-a_j-a_i+\max\{t,a_i+a_j-b_i,a_i\} \quad (6.26)$$

如果作业 i 和 j 在 R 中互易其位,那么完成时间

$$g'(S,t)=a_i+a_j+g(S-\{i,j\},t_{ji})$$

其中,$t_{ji}=b_j+b_i-a_j-a_i+\max\{t,a_i+a_j-b_j,a_j\}$。

将 $g(S,t)$与 $g'(S,t)$相比较可以看出,如果下面的式(6.27)成立,则 $g(S,t)\leq g'(S,t)$。

$$\max\{t,a_i+a_j-b_i,a_i\}\leq\max\{t,a_i+a_j-b_j,a_i\} \quad (6.27)$$

为了使式(6.27)对于 t 的所有取值都成立,则需要

$$\max\{a_i+a_j-b_i,a_i\}\leq\max\{a_i+a_j-b_j,a_j\}$$

即,$a_i+a_j+\max\{-b_i,-a_j\}\leq a_i+a_j+\max\{-b_j,-a_i\}$,或

$$\min\{b_i,a_j\}\geq\min\{b_j,a_i\} \quad (6.28)$$

由式(6.28)可以断定存在一种最优调度,在这调度中,对于每一邻近的作业对 (i,j),有 $\min\{b_i,a_j\}\geq\min(b_j,a_i)$。不难证明,具有这一性质的所有调度都有相同的长度。由此足以产生对于每个邻近作业对都满足式(6.28)的任何调度。根据式(6.28)作以下的观察就能得到具有这一性质的调度。如果 $\min\{a_1,a_2,\cdots,a_n,b_1,b_2,\cdots,b_n\}$是 a_i,那么作业 i 就应是最优调度中的第一个作业。如果 $\min\{a_1,a_2,\cdots,a_n,b_1,b_2,\cdots,b_n\}$是 b_j,那么作业 j 就应是最优调度中的最后一个作业。这使我们能作出一个决策,以便决定 n 个作业中的一个作业应放的位置。然后又将式(6.28)用于其余的 n-1 个作业,再正确地决定另一作业的位置,等等。因此,由式(6.28)导出的调度规则如下。

(1) 把全部 a_i 和 b_i 分类成非降序列。

(2) 按照这一分类次序考察此序列:如果序列中下一个数是 a_j 且作业 j 还没调度,那么在还没使用的最左位置调度作业 j;如果下个数是 b_j 且作业 j 还没调度,那么在还没使用的最右位置调度作业 j;如果已经调度了作业 j,则转到该序列的下一个数。

要指出的是,上述规则也正确地把 $a_i=0$ 的那些作业放在适当位置上,因此对这样的作业不用分开考虑。

例 6.19 设 $n=4,(a_1,a_2,a_3,a_4)=(3,4,8,10)$和$(b_1,b_2,b_3,b_4)=(6,2,9,15)$,对这些 a 和 b 分类后的序列是$(b_2,a_1,a_2,b_1,a_3,b_3,a_4,b_4)=(2,3,4,6,8,9,10,15)$,设 $\sigma_1,\sigma_2,\sigma_3,\sigma_4$ 是最优调度。由于最小数是 b_2,置 $\sigma_4=2$。下一个数是 a_1,置 $\sigma_1=1$。接着的最小数是 a_2,由于作业 2 已被调度,转向再下一个数 b_1。作业 1 已被调度,再转向下一个数 a_3,置 $\sigma_2=3$。最

后剩 σ_3 是空的,而作业 4 还没调度,从而 $\sigma_3 = 4$。

习　题　六

6.1　(1) 递推关系式(6.8)对图 6.16 成立吗?为什么?

(2) 递推关系式(6.8)为什么对于含有负长度环的图不能成立?

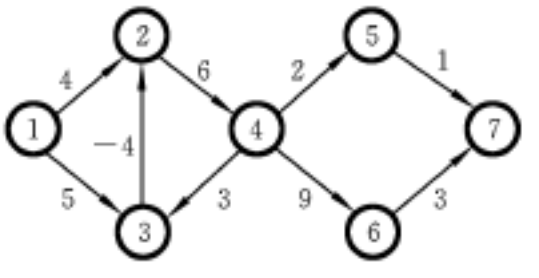


图 6.16　7 个结点的有向图

6.2　修改过程 ALL-PATHS,使其输出每对结点(i,j)间的最短路径,这个新算法的时间和空间复杂度是多少?

6.3　对于标识符集 $(a_1, a_2, a_3, a_4) = (\text{end}, \text{goto}, \text{print}, \text{stop})$,已知成功检索概率为 $P(1) = 1/20, P(2) = 1/5, P(3) = 1/10, P(4) = 1/20$,不成功检索概率为 $Q(0) = 1/5, Q(1) = 1/10, Q(2) = 1/5, Q(3) = 1/20, Q(4) = 1/20$。用算法 OBST 对其计算 $W(i,j), R(i,j)$ 和 $C(i,j)$ $(0 \le i,j \le 4)$。

6.4　(1) 证明算法 OBST 的计算时间是 $O(n^2)$。

(2) 在已知根 $R(i,j)$ $(0 \le i < j \le n)$的情况下写一个构造最优二分检索树 T 的算法。证明这样的树能在 $O(n)$时间内构造出来。

6.5　由于通常只知道成功检索和不成功检索概率的近似值,因此,若能在较短的时间内找出几乎是最优的二分检索树,也是一件很有意义的工作。所谓几乎是最优的二分检索树,就是对于给定的 P 和 Q,该树的成本(由式(6.9)计算)几乎最小。已经证明,由以下方法获得这种检索树的算法可以有 $O(n \log n)$的时间复杂度,选取这样的 k 为根,可使得 $|W(0,k-1) - W(k,n)|$ 尽可能地小。重复以上步骤去找这根的左、右子树。

(1) 对于题 6.3 的数据,用上述方法找出一棵这样的二分检索树。它的成本是什么?

(2) 用 SPAKS 写一个实现上述方法的算法。你的算法的计算时间为 $O(n \log n)$吗?

6.6　设 $(w_1, w_2, w_3, w_4) = (10, 15, 6, 9)$, $(p_1, p_2, p_3, p_4) = (2, 5, 8, 1)$。试生成每个 f_i 阶跃点的序偶集合 S^i, $0 \le i \le 4$。

6.7　假设在过程 DKNAP 已算出了 S^i, $0 \le i < n$,写一个确定 0/1 背包问题的最优决策序列 $x_1, x_2, \cdots, x_n$ 的算法 PARTS。由于已知道 F(i)和 F(i+1),因此可以使用二分检索方法去确定序偶(P,W)是否属于 S^i。所以,你的算法的时间复杂度不应大于 $O(n \max\{\log|S^i|\}) \le O(n^2)$。

6.8　给出一个使得 DKNAP (算法 6.7)出现最坏情况的例子,它使得 $|S^i| = 2^i$, $0 \le i \le n$。还要求对 n 的任意取值都适用。

6.9　(1) 证明如果所有的效益值 p_j 都是整数,那么在这样的背包问题中每个序偶集 S^i 的大小 $|S^i|$ 不大于 $1 + \sum_{1 \le j \le i} p_j / \gcd(p_1, p_2, \cdots, p_n)$,其中,$\gcd(p_1, p_2, \cdots, p_n)$是这 n 个 p_j 的最大公因数。

(2) 证明当所有的重量值 w_j 都是整数时,$|S^i| \le 1 + \min\{\sum_{1 \le j \le i} w_j, M\} / \gcd(w_1, w_2, \cdots, w_n, M)$。

6.10　把分治方法与课文中生成序偶集合的方法结合在一起,对 0/1 背包问题写一个时间复杂度为 $O(2^{n/2})$的算法。

6.11　写一个类似于 DKNAP 的算法去求解递推关系式(6.18),其算法要求的时间和空间是多少?

6.12　如果将式(6.1)的 0/1 约束用 $x_i \ge 0$ 且为整数来代替,这就是整数背包问题。

(1) 对这个问题推导出相应于式(6.14)的动态规划递推关系式。

(2) 如何将这个问题转换成 0/1 背包问题?

(提示:对于每个变量 x_i 引进一些新的 0/1 变量。如果 $0 \le x_i \le 2^j$,就引进 j 个变量,在 x_i 的二进制表示中每一位代表一个这样的变量。)

6.13 假定两个仓库 W_1 和 W_2 都存有同一种货物,其库存量分别为 r_1 和 r_2。现要将其全部发往 n 个目的地 $D_1, D_2, \cdots, D_n$。设发往 D_j 的货物为 d_j,因此 $r_1 + r_2 = \sum d_j$。如果由仓库 W_i 发送量为 x_{ij} 的货物到目的地 D_j 的花费为 C_{ij},那么仓库问题就是求各个仓库应给每个目的地发多少货才使总的花费最小。即要求出这些非负整数 x_{ij},$1 \leq i \leq 2$,$1 \leq j \leq n$,它使得 $x_{1j} + x_{2j} = d_j$,$1 \leq j \leq n$ 并且使 $\sum_{i,j} c_{ij}(x_{ij})$ 取最小值。假设当 W_1 有 x 的库存且按最优方式全部发往目的地 $D_1, D_2, \cdots, D_i$ 时所需的花费为 $g_i(x)$(W_2 的库存为 $\sum_{1 \leq j \leq i}(d_j - x)$。于是,此仓库问题的最优总花费是 $g_n(r_1)$。

(1) 求 $g_i(x)$ 的递推关系式。

(2) 写一个算法求解这个递推关系式并要能得到 x_{ij} 的决策值的最优序列,$1 \leq i \leq 2$,$1 \leq j \leq n$。

6.14 有两台机器 A、B 和 N 个要处理的作业。每台机器都能单独处理这 N 个作业。如果在机器 A 上处理作业 i 需要 a_i 个单位时间,如果在机器 B 上处理,则需要 b_i 个单位时间。由于作业和机器特性的缘故,完全可能对于某个作业 i,$a_i \geq b_i$,而对某个作业 j,$a_j < b_j$,这里 $i \neq j$。另外,不能把一个作业分由两台机器处理,而且一台机器不能同时处理两个作业。求取能确定对这 N 个作业所需最少处理时间的动态规划递推关系式。说明如何求解所得到的递推式。对所选择的例子求出最优解。说明如何确定作业的最优分配。

6.15 设 I 是在两台设备上作流水线调度的任一实例。

(1) 证明对于同一个 I,每种 POFT 调度的长度和每种 OFT 调度的长度相同。因此,6.8 节的算法也生成一种 POFT 调度。

(2) 证明对于 I 存在着一种作业在两台设备上按相同次序处理的 OFT 调度。

(3) 证明对于 I 存在着由作业的某种排列(见(2)) 所定义的 OFT 调度,而这种排列中所有 $a_i = 0$ 的作业均在这排列的前部。进而证明此排列前部那些 $a_i = 0$ 的作业出现的次序是无关紧要的。

6.16 设 I 是在两台设备上作流水线调度的任一实例。又设 $\sigma = \sigma_1, \sigma_2, \cdots, \sigma_n$ 是定义 I 的一种 OFT 调度的排列。

(1) 利用式(6.28)证明存在着一种 OFT 调度,它的 σ 对于所有相邻的 i 和 j,即使得 $i = \sigma_k$ 和 $j = \sigma_{k+1}$ 的 i 和 j,有 $\min\{b_i, a_j\} \geq \min\{b_j, a_i\}$。

(2) 对于满足(1)中条件的 σ,证明所有使得 $i = \sigma_r$ 和 $j = \sigma_r$,且 $k < r$ 的 i 和 j,$\min\{b_i, a_j\} \geq \min\{b_j, a_i\}$。

(3) 证明与满足(1)中条件的所有 σ 相应的那些调度有相同的完成时间。

(提示:使用(2),将两个满足(1)的不同调度,在不增加完成时间的情况下把一个转换成另一个)

6.17 最优性原理并不总是对可以将其解看成是一系列决策结果的所有问题成立。找两个最优性原理不成立的例子,并说明对这两个问题最优性原理为什么不成立。

第 7 章

基本检索与周游方法

7.1 一般方法

许多问题的解法涉及对二元树、树和图的处理。这种处理往往要求确定满足某一性质的那些给定数据对象中的一个结点或结点子集。例如,希望求出一棵二元树中数据值比 X 还小的所有结点或者想在一个已知的图 G 中求出某个结点 v 所能到达的所有结点。对于满足某已知性质的结点子集的判断可以通过系统地检查这些给定数据对象的那些结点来实现。这通常在数据对象中以检索的方式来进行。当这种检索必须包括对检索的数据对象的每一个结点进行检查时,就把它称之为周游。

5.4 节介绍过一个为了找标识符 X 而检索一棵二分检索树的算法。这个算法不是一个周游算法,因为它不检索这棵检索树中的每一个结点。有些问题可能要涉及周游一棵二分检索树的问题。例如,要求列出这棵树中的所有标识符的时候就是这种情况。在本章中将对这样的一些算法进行研究。

本节把所要讨论的这些方法分成 3 类。前 2 类只能分别适用于二元树和树,这些方法将检查给定数据对象中的每一个结点,因此,这些方法称为周游方法。第 3 类是适用于图的方法(也适用于树和二元树),但这些检索方法不能检查所有的结点,因此只称为检索方法。在检索(或周游)期间,一个结点的信息段可能数次被用到。因此,就有必要区别对该结点这些信息段的某些使用。在这些信息段被使用期间,这个结点就称为被访问的结点。访问一个结点可以包括打印出该结点的数据信息段;在二元树用来表示一个表达式的情况下,进行由这个结点所表示的运算;置一标志位为 1 或 0,等等。由于本节所描述的树和图的检索和周游不受具体应用所限制,因此在本节均使用术语被访问的而不指出此时在这结点所完成的特殊功能。

7.1.1 二元树周游

在处理与二元树有关的问题时,往往要求对这棵树中的每一个结点都正好访问一次。完成这一系列的访问就称为对二元树周游。完整的周游产生一棵树中信息的线性序。这个线性序可能是常见而有用的。在周游一棵二元树的时候,希望处理每一个结点并以同样的方式处理它的子树。如果设 L、D、R 分别代表在一个结点处时的左移、打印这个数据和右移,那么周游就有 6 种可能的组合:LDR、LRD、DLR、DRL、RDL 和 RLD。如果沿用先左后右的习惯进行,那就只有 3 种周游:LDR、LRD 和 DLR。这 3 种周游分别取名

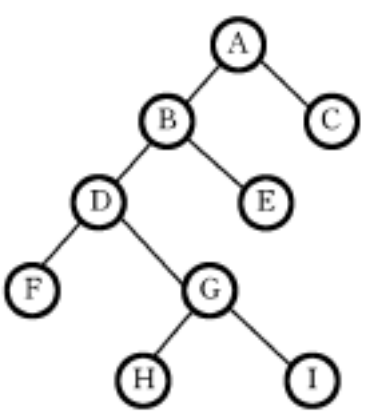


图 7.1 一棵二元树

为中根次序周游、后根次序周游和先根次序周游。下面定义这 3 种周游并在图 7.1 中的二元树上说明这些周游是如何进行的。

非形式地说,中根次序周游要求沿着这棵树向左下方移动直到不能走到更远为止,然后“访问”这个结点,转移到右儿子结点并继续上述的动作。如果不能转到右边,就返回到另一个结点。递归过程 INORDER 对这种周游作了准确而精巧的描述。子算法 VISIT 执行访问一个结点时所需要完成的任何功能。

算法 7.1　中根次序周游的递归表示

```
procedure  INORDER(T)
    // T 是一棵二元树。T 的每个结点有 3 个信息段:LCHILD,DATA,RCHILD //
    if  T≠0  then call INORDER (LCHILD(T))
                  call VISIT(T)
                  call INORDER (RCHILD(T))
    endif
end INORDER
```

表 7.1 描绘了在图 7.1 所示的那棵二元树上 INORDER 的工作情况。这里假定访问一个结点只要求印出它的 DATA 信息段,由此周游所导出的输出是 FDHGIBEAC。

表 7.1　对图 7.1 的二元树作中根次序周游,在有 call VISIT(T) 的地方用 print(DATA(T))代替

INORDER 的调用主程序	根的值	动作
	A	
1	B	
2	D	
3	F	
4	—	print('F')
4	—	print('D')
3	G	
4	H	
5	—	print('H')
5	—	print('G')
4	I	
5	—	print('I')
5	—	print('B')
2	E	
3	—	print('E')
3	—	print('A')
1	C	
2	—	print('C')
2	—	

算法 7.2 和算法 7.3 列出了先根次序和后根次序周游所对应的递归过程。

算法 7.2　先根次序周游

```
procedure  PREORDER (T)
    // T 是一棵二元树。T 中每个结点有 3 个信息段:LCHILD,DATA,RCHILD //
    if T≠0 then call VISIT (T)
                call PREORDER (LCHILD(T))
```

```
                call PREORDER (RCHILD(T))
        endif
    end PREORDER
```

算法 7.3　后根次序周游

```
    procedure POSTORDER(T)
        // T 是一棵二元树。T 中每个结点有 3 个信息段:LCHILD,DATA,RCHILD //
        if T≠0 then call POSTORDER (LCHILD(T))
                    call POSTORDER (RCHILD(T))
                    call VISIT(T)
        endif
    end POSTORDER
```

在用 print(DATA(T))代替 call VISIT 的情况下,图 7.1 的二元树用算法 7.2 和7.3分别输出 ABDFGHIEC 和 FHIGDEBCA。

定理 7.1　当输入的树 T 有 $n \geq 0$ 个结点时,设 t(n)和 s(n)分别表示这些周游算法中的任意一个算法所需要的最大时间和空间。如果访问一个结点所需要的时间和空间是 $\Theta(1)$,则 $t(n) = \Theta(n)$, $s(n) = \Theta(n)$。

证明　每个周游算法所做的工作由两部分组成:①在递归这一级所做的工作;②这一级算法的递归调用所做的工作。第一部分工作要求的时间由常数 c_1 所限界。如果 T 的左子树的结点数是 n_1,那么 t(n)就由下面的递归式给出:

$$t(n) = \max_{n_1}\{t(n_1) + t(n - n_1 - 1) + c_1\} \qquad n \geq 1$$

注意:由 $t(0) \leq c_1$,通过归纳法来证明 $t(n) \leq c_2 n + c_1$ 成立。其中 c_2 是一使得 $c_2 \geq 2c_1$ 的常数。当 $n = 0$ 时,这个不等式显然成立。假定当 $n = 0, 1, \cdots, m - 1$ 时,这个不等式成立,现在来证明当 $n = m$ 时这个不等式也成立。设 T 是一棵 m 个结点的树,又设 n_1 是 T 的左子树的结点数,于是

$$\begin{aligned} t(m) &= \max\{t(n_1) + t(n - n_1 - 1) + c_1\} \\ &\leq \max\{c_2 n_1 + c_1 + c_2 (n - n_1 - 1) + c_1 + c_1\} \\ &= \max\{c_2 n + 3c_1 - c_2\} \\ &\leq c_2 n + c_1 \end{aligned}$$

容易看出存在 c_1'和 c_2'使得 $t(n) \geq c_2' n + c_1'$。因此, $t(n) = \Theta(n)$。注意,只是为了保留递归调用时那些局部变量的值,才需要附加空间。如果 T 的深度为 d,则这个空间显然是 $\Theta(d)$。对于一棵 n 个结点的二元树,有 $d \leq n$,从而 $s(n) = \Theta(n)$。证毕。

尽管递归周游算法能够直接使用,但由于递归总开销的缘故,最好还是先把这些算法重新编成非递归形式。把一个递归算法转换成等价的非递归形式的一般规则已在第 1 章里给出。这些规则导出的算法一般并不精巧,但是,使用这些标准的转换规则,对于一个已知是正确的递归算法,可以保证其非递归形式是正确的。现在来直接写一个中根次序周游的非递归算法。假定 T 是二元树的根,它的左子树(如果非空)则必须在访问 T 以前被周游,那么,可以把 T 放到一个栈中并且开始周游它的左子树。该栈将被保持成这样的状态,当其左子树被周游完毕时,T 在这个栈的顶部。

考虑图 7.2 所示的二元树,结点 A 有左子树 B,因此将 A 存入栈,然后开始周游 B。结点 B 有左子树 D,于是 B 被存入栈并且去周游 D。D 的左子树为空,所以 D 可以被访问。周游 D 的右子树,这就要求访问结点 G,此刻就完成了对 B 的左子树的周游,结点 B 位于栈顶。从栈中移去 B 并访问之。下面着手周游 B 的右子树,由于 B 的右子树为空,因此,作为 A 的左子树的树 B 已被周游完毕,A 位于栈顶。通常,这个栈只包含那些还没有完成周游其左子树的结点。每当某个结点 Q 的左子树被周游完毕,Q 就肯定在这个栈的顶部。例如,根为 D 的子树被周游后,B 是这个栈顶部的结点,B 的周游被完成时,A 就在栈顶,当完成了对 A 的周游时,栈为空。

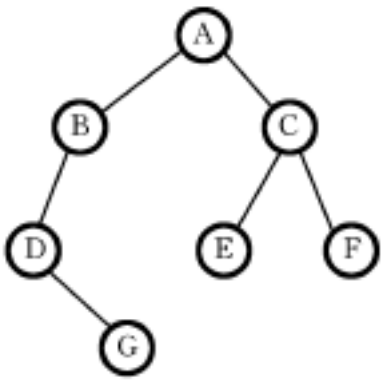


图 7.2 一棵二元树

过程 INORDER1(算法 7.4)就是这种形式的算法。变量 P 周游二元树 T,在第 4～19 行的循环开始处,P 指向要周游的子树的根。在第 5～11 行,把由 P 开始的所有左子树的根存入栈,在这个循环的出口处,P 指向其左子树为空的一个结点,因此 P 马上就要被访问。在第 12～18 行的循环开始处,P 指向现在就要访问的结点(即,如果它有非空的左子树,那么其左子树已周游完毕)。在访问了 P 结点之后,它的右子树(如果非空的话)即将被周游。在 P 有一棵空的右子树的情况下,就完成了一棵左子树的周游,此时应该移到这棵已完成的左子树的父亲,这个父亲是栈中最顶部的那个结点(第 16～17 行)。容易看出,若 Q 是 R 的左子树的根,则当 Q 的周游完成时 R 就是栈顶的那个结点。每当访问了一个结点,就把它从栈中去掉。在对 Q 的周游完成以前,Q 中的所有结点都应被访问。因此,在 R 之后存入栈中的所有结点,在完成对 Q 的周游以前都必须被删除。

算法 7.4 中根次序周游的非递归算法

```
        procedure INORDER1 (T)
            // 使用大小为 m 的栈的一个非递归算法 //
1           integer STACK(m) ,i, m
2           if T = 0 then return endif    // T 为空 //
3           P←T;i←0    // 周游 T;i 为栈顶 //
4           loop
5             while LCHILD(P)≠0 do    // 周游左子树 //
6               i←i + 1
7               if i > m then print ('stack overflow')
8                            stop
9               endif
10              STACK(i) ←P;P←LCHILD(P)
11            repeat
12            loop
13              call VISIT(P)    // P 的左子树周游完 //
14              P←RCHILD(P)
15              if P≠0 then exit endif    // 周游右子树 //
16              if i = 0 then return endif
17              P←STACK (i);i←i - 1
18            repeat    // 访问父结点 //
```

```
19          repeat
        end INORDER1
```

通过二元树 T 中的结点数 n 来分析 INORDER1 的计算时间。在第 5～11 行的 while 循环的每一次迭代中,把一个结点存入栈(第 10 行),存入栈中的每一个结点都要被访问(第 13 行)。由于没有结点被访问一次以上,因此,在此算法的整个执行过程中第 5～11 行的循环就不可能迭代 n 次以上。事实上,至多有 n - 1 个结点可能存入栈,这是因为叶子结点都不存入栈(第 5 行)并且 n≥1 的每一棵树至少有一个叶子结点。所以第 5～11 行的总的时间是 O(n)。在第 12～18 行的循环的每次迭代中,有一个结点被访问。由于 T 中的每一个结点实际上被访问一次,而且在这算法中任何别的地方都不对结点进行访问,因此该算法中的这一循环迭代 n 次。故这循环所需要的总的时间是 Θ(n)。所以,INORDER1 的时间复杂度是 Θ(n)。

就栈空间而论,只有那些具有非空左子树的结点可能存入栈。当 T 是一棵左斜二元树时,其最坏情况就会发生(见图 7 .3(b))。在一棵左斜二元树中,除了叶子结点外的每一个结点都有一棵非空左子树和一棵空右子树。在这种情况下,需要大小为 n - 1 的栈,如果每一个结点都有一棵空左子树并且除了叶子结点之外的所有结点都有一棵非空的右子树就会发生最好的情况。这样的一棵二元树是右斜二元树(见图 7 .3(a))。在这种情况下,没有结点进入栈。用 T 的深度来表示所需的栈空间则更为有效。可以证明,若 T 的深度为 d,则所需的栈空间为O(d)。

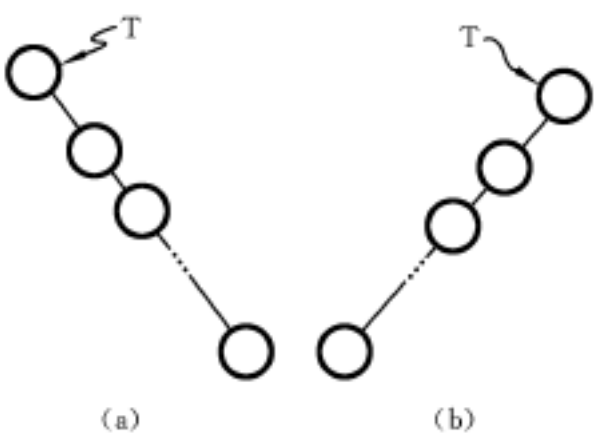


图 7 .3　斜二元树

(a)右斜树;　(b)左斜树

由以上分析可知,所有的周游算法都必须访问每一个结点,因此计算时间应该至少是 Θ(n)。对所使用的附加空间(即栈空间)作一些改进是唯一可探讨之处。下面介绍在 Θ(n)时间和 Θ(1)空间内周游二元树的一种方法。

如果每个结点有一个 PARENT 信息段与它的父结点相连接,则周游可以在 Θ(n)时间和 Θ(1)空间内完成,对此的算法设计与分析留作习题。在这里,将在没有 PARENT 信息段的情况下去获取一个具有类似特性的算法。PARENT 信息段的存在使我们能从任何一个结点 P 到达根结点。为了得到一个空间为 Θ(1)的算法,可通过颠倒由根结点到当前正被检查的结点的链的方向来完成这一效能。譬如,若假定 P 指向树 T 中当前正在检查的那个结点,Q 指向它的父亲,那么将保留一条由 Q 到根 T 的路径,这条路径叫做 Q-T 路径,它由 T 到 Q 路径上的所有结点链接到一起而组成。如果 U、V 和 W 是这条路径上的这么 3 个结点——U 是 V 的父亲而 V 是 W 的父亲,那么在 W 是 V 的右儿子的情况下就将 V 通过它的 RCHILD 信息段链接到 U。否则就通过它的 LCHILD 信息段把 V 链接到 U。

但是,使用颠倒结点链接方向的方法,在访问了 P 所指结点以后,要将 Q 所指结点与 P 所指结点再链接在一起就会出现如下问题:设 P 所指结点为 X,Q 所指结点为 Y,如果已周游了以 X 为根的子树的所有结点,此时要是 Y 只有这一个儿子结点,就可以通过 LCHILD(Y)或 RCHILD(Y)是否为零来判断 X 是 Y 的右儿子还是左儿子。因此,在这种

情况下,将 X 与 Y 重新链接起来没有什么问题。但是,如果 Y 有两个儿子,则此时的 LCHILD(Y)和 RCHILD(Y)一个是 Q-T 路径上的一个倒挂链,一个指向 Y 的另一个儿子,于是,判断 X 是左儿子还是右儿子就出现了困难。为了解决这一问题,根据中根次序周游的定义,可以在访问 Y 的时候,将 Y 存入一临时工作单元 LR,这样在周游了以 X 为根的子树后,如果 LR 中的内容是 Y,则说明 X 是 Y 的右儿子,反之,则是左儿子。但是,Q-T 路径上有两个儿子的结点可能有多个,而在访问它们时又都应将它们保存起来,因此不可避免地要引进栈。为了节省空间,可利用已访问过的叶子结点的 LCHILD 和 RCHILD 信息段来建立一个链接表结构的栈,用 LCHILD 信息段存放一个有两个儿子的结点,用 RCHILD 信息段作为链接指针。

算法 7.5 用以上方法对中根次序周游进行了具体描述。除上面引进的临时工作单元 P、Q 和 LR 外,还引进了临时工作单元 AV、TOP 和 R。其中,AV 用来指示当前可作栈使用的叶子结点,TOP 和 R 的作用在算法中一眼就可看出。

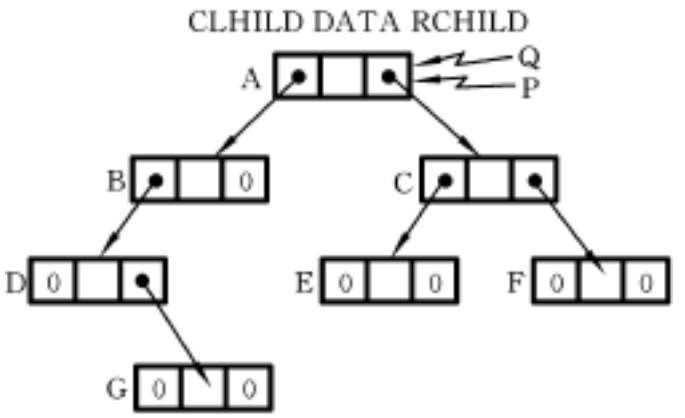
算法 7.5　在 Θ(n)的时间和 Θ(1)的空间内周游二元树的算法

```
line procedure INORDER2(T)
      //使用固定量的附加空间,二元树 T 作中根次序周游//
1     if T = 0 then return endif    //二元树为空//
2     TOP←LR←0; Q←P←T    //置初值//
3     loop
4       loop    //尽可能地往下移//
5         case
6           :LCHILD(P) = 0 and RCHILD(P) = 0:
                    //不能往下移了//
7           call VISIT(P);exit
8         :LCHILD(P) = 0:    //移到 RCHILD(P)//
9           call VISIT(P)
10            R←RCHILD(P);RCHILD(P)←Q
              Q←P;P←R
11        :else:    //移到 LCHILD(P)//
12        R←LCHILD(P);LCHILD(P)←Q;Q←P;
          P←R
13      endcase
14    repeat
      //P 是叶结点,向上移到其右子树还没检查的结点//
15    AV←P    //作为栈使用的叶结点//
16    loop    //由 P 向上移//
17      case
18      :P = T:return    //回退到了根,返回//
19      :LCHILD(Q) = 0:    //Q 由 RCHILD 链接//
20        R←RCHILD(Q);RCHILD(Q)←P;P←Q;Q←R
21      :RCHILD(Q) = 0:    //Q 由 LCHILD 链接//
22        R←LCHILD(Q);LCHILD(Q)←P;P←Q;Q←R;
```

```
            call VISIT(P)
23          :else:   //检查 P 是否为 Q 的右儿子//
24          if Q = LR then   // P 是 Q 的右儿子//
25            R←TOP;LR←LCHILD(R)   //修改 LR//
26            TOP←RCHILD(R)   //退栈//
27            LCHILD(R)←RCHILD(R)←0   //释放作栈用的叶结点//
28            R←RCHILD(Q);RCHILD(Q)←P;P←Q   Q←R
29                    else   // P 是 Q 的左儿子//
30          call VISIT(Q)
31          LCHILD(AV)←LR;RCHILD(AV)←TOP
32          TOP←AV;LR←Q
33          R←LCHILD(Q);LCHILD(Q)←P   //恢复到 P 的链接//
34          P←RCHILD(Q);RCHILD(Q)←R;exit   //移到右边//
35          endif
36          endcase
37        repeat
38        repeat
39      end INORDER2
```

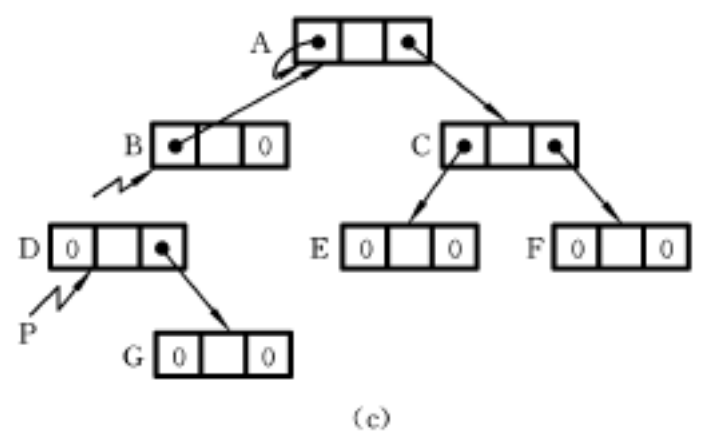
为便于理解 INORDER2,图 7 .4 描述了用 INORDER2 周游图 7 .2 所示树的基本过程。

2 行:置初值,Q-T 路径为空。



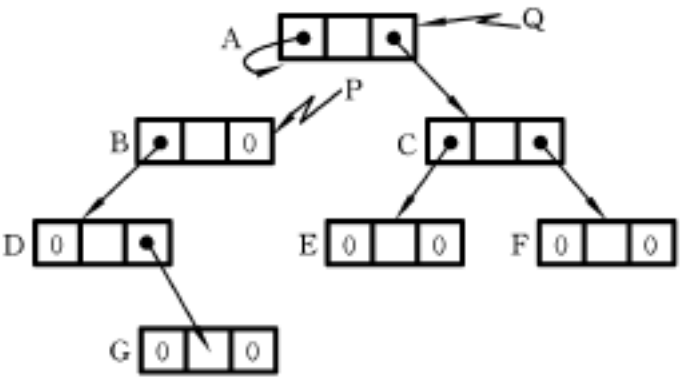
(a)

11～13 行:P 移到 D,结点 B 由 LCHILD 链接到 Q-T



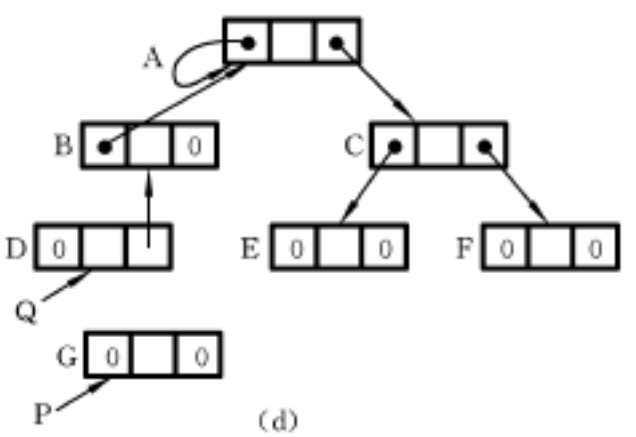
(c)

11～13 行:P 移到 B,Q-T 路径只含根结点 A,LCHILD(A) = A表示 Q-T 路径末端。



(b)

8～10 行:访问 D,P 移到 B 的右子树(结点 G),D 由 RCHILD 链接到 Q-T



(d)

图 7 .4　用 Θ(1)的附加空间周游图 7 .2 的二元树

6～7 行: 再不能下移,访问 G。

15 行: 叶结点 G 将作栈使用,存入 AV。

19～20 行:沿 Q-T 路径回退,Q 指向 B,P 指向 D 并恢复 RCHILD(D)指向 G。

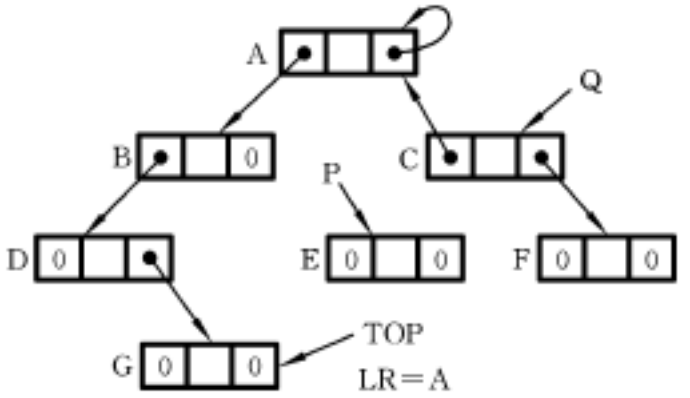
图示同(c)

(e)

23,24,29～35 行:由于 Q(即 A)的左、右儿子均不为 0,且 LR≠Q,说明 P 是 Q 的左儿子,故访问 A。然后将 LR 中的值进栈(即存入 LCHILD(G)),原栈顶地址置于 RCHILD(G),现栈顶地址变为结点 G 的地址。再将 A 存入 LR。

由 A 的 RCHILD 链接到 Q-T,并恢复 LCHILD(A)指向 B,P 移到 C。

11～13 行:P 移到 E,结点 C 由 LCHILD 链接到 Q-T。



(h)

6～7 行:访问 F

15 行: F 存入 AV

23～28 行:沿 Q-T 路径回退,由于 Q = LR,说明 F 是 C 的右儿子。修改 LR。退栈,释放作栈用的叶结点 E。P 指向 C,Q 指向 A,恢复 RCHILD(C) = F。

图示同(g)

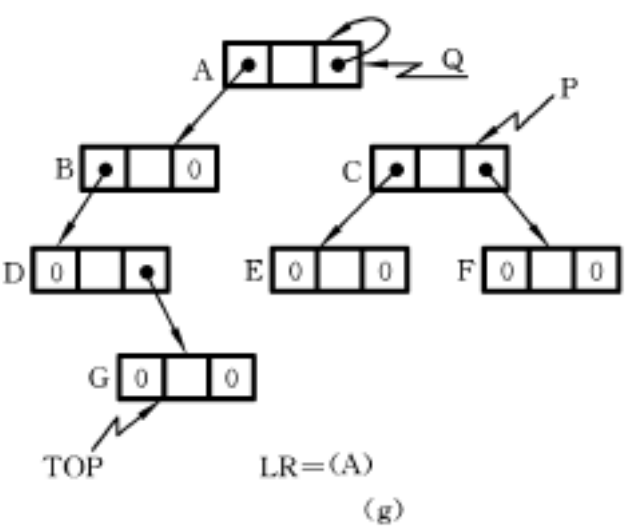
(j)

21～22 行:继续回退,Q 指向 A,P 指向 B 并恢复 LCHILD(B)指向 D。

由于从 B 的左子树返回(注意:这是根据 RCHILD(B) = 0),故访问 B 且回退到 A。

图示同(d)

(f)

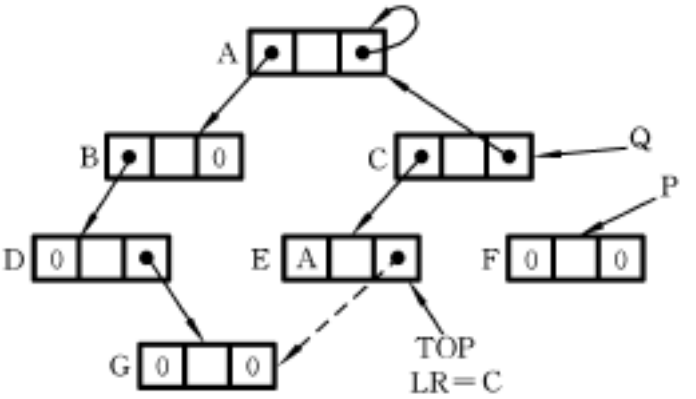


(g)

6～7 行:再不能下移,访问 E。

15 行: 叶结点 E 将作栈使用,存入 AV。

23,24,29～35 行:说明与(g)类似。访问 C。



(i)

23～28 行:继续回退,说明与(j)类似。

18 行:由于 P = T,表示已回退到根,返回。

图示同(a)

(k)

续图 7.4 用 Θ(1)的附加空间周游图 7.2 的二元树

现在,分析算法 INORDER2 需要的计算时间和辅助空间。设二元树的结点数为 n,又设度为 0,1 和 2 的结点数分别为 n_0,n_1 和 n_2,于是 $n = n_0 + n_1 + n_2$。显然,P 指向一个 0 度结点的次数为 1,这种情况出现在第 4～14 行的循环中 P 下移到达这个 0 度结点之时。而 P 将两次指向只有一个儿子的结点,一次是在下移的时候,另一次则在从它的儿子向上移动的

时候(第 16～37 行)。对于有两个儿子的结点,P 也将两次指向该结点,一次在下移时(第 4～14 行),另一次是在从它的右儿子处上移时(第 16～37 行),而当从它的左儿子处上移时(第 29 行),虽然此时访问该结点,但 P 被修改成指向该结点的右儿子。由此 P 值改变的总次数是 $n_0 + 2n_1 + 2n_2$。在第 4～14 行的循环的每次迭代中,如果 P 不是叶子,则 P 的值就改变;如果 P 是叶子,则转出这个循环且在第 16～37 行的循环中改变 P 的值,即 P 是叶子时既进入第 4～14 行的循环,又进入第 16～37 行的循环。另外,第 16～37 行的循环的每一次迭代都改变 P 的值。所以,第 4～14 行和第 16～37 行这两个循环的迭代总数为 $2n_0 + 2n_1 + 2n_2$。而这两个循环中的任何一次迭代所需的时间是 Θ(1),故第 3～38 行循环的总时间是 $\Theta(2n_0 + 2n_1 + 2n_2) = \Theta(n)$。第 1 行和第 2 行都只花 Θ(1)的时间,所以总共需要的时间是Θ(n)。

由于算法中只有一些像 P、Q、AV、LR、TOP 和 R 之类的简单变量,因此所需的辅助空间为 Θ(1)。值得指出的是,在所用辅助空间上,INORDER2 虽比 INORDER1 节省,但这种节省是以增加计算时间为代价的,因此,只有在 INORDER1 所要求的辅助空间 O(d)无法满足的情况下才考虑使用 INORDER2。

7.1.2　树周游

可以用类似于定义二元树的周游方法对树的周游进行定义。由于树的子树一般无顺序可言,而对二元树的各种周游算法又是建立在它的子树有严格顺序(左、右子树)这一基础之上的,因此,为便于对树周游作出定义,不妨假定树的子树也有某种顺序。这样一来,说一个结点的第一、第二、第三棵子树等就有了意义。由树和森林之间的关系知道,一棵树恰好是具有一棵树的森林,而一个森林可由去掉一棵树的根而得到,因此,利用森林的周游来递归定义树周游是很方便的。树周游一般也有 3 种方法,分别叫做树先根次序周游,树中根次序周游和树后根次序周游。设 F 是一个森林,这 3 种周游方法可描述如下。

1. 树先根次序周游(F)

(1) 若 F 为空,则返回;

(2) 访问 F 的第一棵树的根;

(3) 按树先根次序周游 F 的第一棵树的子树;

(4) 按树先根次序周游 F 其余的树。

2. 树中根次序周游(F)

(1) 若 F 为空,则返回;

(2) 按树中根次序周游 F 的第一棵树的子树;

(3) 访问 F 的第一棵树的根;

(4) 按树中根次序周游 F 其余的树。

3. 树后根次序周游(F)

(1) 若 F 为空,则返回;

(2) 按树后根次序周游 F 的第一棵树的子树;

(3) 按树后根次序周游 F 其余的树;

(4) 访问 F 的第一棵树的根。

由于树在一般情况下都是用其相应的二元树来表示,因此,这里就不再给出树周游的详

细算法。在后几节中,可以看到树后根次序周游的一些例子。

假定 T 是由森林 F 转换成的二元树,由转换方法可知,对 T 的先根次序和中根次序周游与 F 上的这些周游有一个自然的对应,即 T 的先根次序周游相当于按树先根次序周游访问 F 的结点,T 的中根次序周游相当于按树中根次序周游访问 F 的结点。但对 T 的后根次序周游则没有类似的自然对应。

7.1.3 图的检索和周游

与图有关的一个基本问题是路径问题。就其最简单的形式而论,它要求确定在一个给定的图 G = (V, E)中是否存在一条起自结点 v 而终于结点 u 的路径。一种更一般的形式则是确定与某已知起始结点 v∈V 有路相通的所有结点。后面这个问题可以从结点 v 开始,通过有次序地检索这个图 G 上由 v 可以到达的结点而得到解决,对此问题介绍下面两种检索方法。

1.宽度优先检索和周游

在宽度优先检索中,从结点 v 开始,并给 v 标上已到达(即访问)的标记(此时,称结点 v 还没检测。当一个算法访问了邻接于某结点的所有结点时,就说该结点已由此算法检测了)。下一步,访问邻接于 v 的所有未被访问的结点,这些结点是新的没检测的结点。而结点 v 现在已检测过了。由于这些新近已访问的结点还没有检测,于是将它们放置到未检测结点表的末端。这表上的第一个结点是下一个要检测的结点。这个未检测结点表起一个队列的作用并且可以用任何一种标准的队列表示形式来表示它。过程 BFS 描述了这一检索的细节。这个过程还用了两个算法 DELETEQ(v, Q)和 ADDQ(v, Q)。算法DELETEQ(v, Q)从队列 Q 删除一个结点并带着这个被删除的结点返回。算法ADDQ (v, Q)将结点 v 加到队列 Q 的尾部。

在图 7.5(a)所示的无向图上检验这个算法。假定这个图用邻接表(图 7.5(b))来表示,这些结点就按 1,2,3,4,5,6,7,8 这样的次序被访问。而对图 7.5(c)所示的有向图,其起始结点为 1 的宽度优先检索只访问结点 1,2,3。由结点 1 不能到达结点 4。

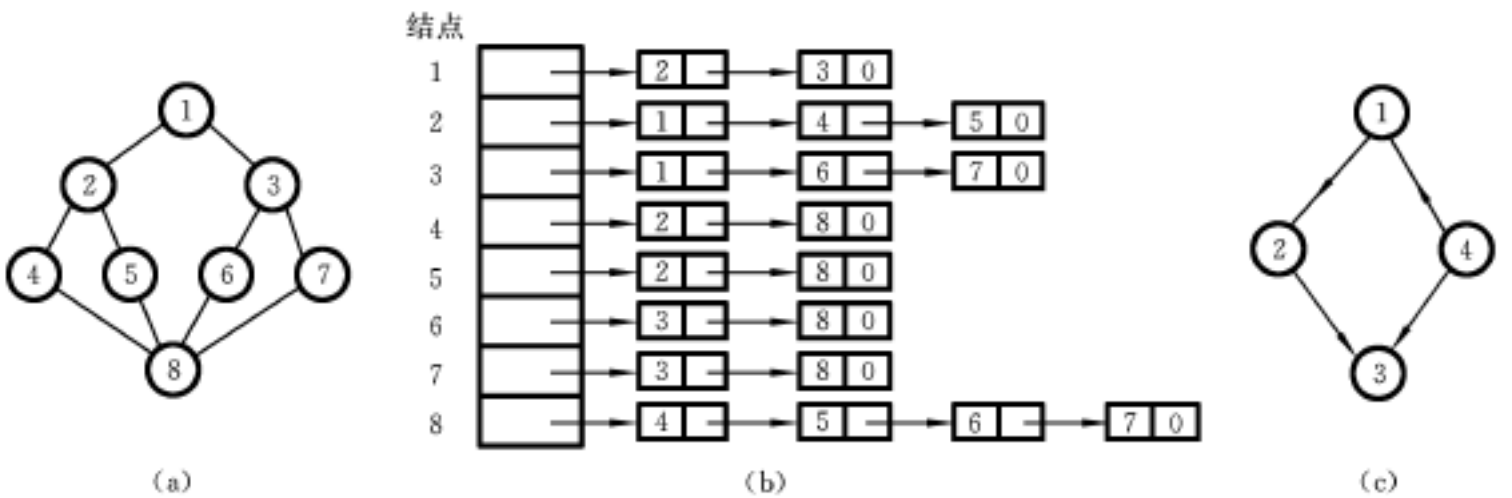


图 7.5 例图和邻接表

(a)无向图 G;(b)G 的邻接表;(c)有向图

算法 7.6 宽度优先检索算法

```
line procedure BFS(v)
    //宽度优先检索 G,它在结点 v 开始执行。所有已访问结点都标上 VISITED(i) = 1。图 G
```

```
        和数组 VISITED 是全程量。VISITED 开始时都已置成 0 //
1     VISITED(v)←1;u←v
2       将 Q 初始化为空　// Q 是未检测结点的队列//
3       loop
4         for 邻接于 u 的所有结点 w do
5           if VISITED(w) = 0 then call ADDQ(w,Q)　// w 未检测//
6                                VISITED(w)←1
7           endif
8         repeat
9       if Q 为空 then return endif // 不再有还未检测的结点//
10          call DELETEQ(u,Q)　// 从队中取一个未检测结点//
11      repeat
12    end BFS
```

定理 7.2　算法 BFS 访问由 v 可到达的所有结点。

证明　设 G = (V, E)是一个(有向或无向)图,且 v∈V。通过施归纳法于由 v 到所有可达结点 w∈V 的最短路径的长度来对这个定理进行证明。用 d(v,w)来表示由 v 到某一可达结点 w 的最短路径的长度(即边数)。显然,d(v,w)≤1 的所有结点 w 都要被访问。现在假定所有 d(v,w)≤r 的结点都要被访问,进而证明 d(v,w) = r + 1 的所有结点都要被访问。设 w 是 V 中一个 d(v,w) = r + 1 的结点,又设 u 是一个在 v 到 w 的最短路径上紧挨 w 的前一结点,于是 d(v,u) = r,因此 u 通过 BFS 被访问。假定 u≠v 且 r≥1。于是,u 在临访问之前被放到未检测结点队列 Q 上,而这个算法直到 Q 变成空时才会终止,因此,u 必定在某个时刻从 Q 中移出,而且所有邻接于 u 的未访问结点在第 4～8 行的循环中被访问,所以 w 也被访问。证毕。

定理 7.3　设 t(n,e)和 S(n,e)是算法 BFS 在任一具有 n 个结点和 e 条边的图 G 上所花的最大时间和最大附加空间。若 G 由邻接表表示,则 $t(n,e)=\Theta(n+e)$ 和 $S(n,e)=\Theta(n)$。若 G 由邻接矩阵表示,则 $t(n,e)=\Theta(n^2)$ 和 $S(n,e)=\Theta(n)$。

证明　只有在第 5 行的结点才被加到队列。仅当结点 w 有 VISITED(w) = 0 时,它才可以加到队列上。在将 w 加到队列之后,紧接着就把 VISITED(w)置成 1(第 6 行)。因此,每个结点都至多只有一次被放在队列上。结点 v 不会放在队列上,所以至多做 n - 1 次加入队列的动作,需要的队列空间至多是 n - 1。其余的变量所用的空间为 O(1)。所以 S(n,e) = O(n)。如果 G 是一个具有 v 与其余的 n - 1 个结点相连接的图,那么邻接于 v 的全部 n - 1个结点都将在同一时刻被放在队列上。此外,数组 VISITED 需要 Θ(n)的空间。因此,$S(n,e)=\Theta(n)$。这结果与使用的是邻接矩阵还是邻接表无关。

如果使用邻接表,那么对邻接于 u 的所有结点可以在时间 d(u)内作出判断。若 G 是无向图,则 d(u)是 u 的度数;若 G 是有向图,则 d(u)是 u 的出度。因此,当结点 u 被检测时,第 4～8 行循环的时间是 $\Theta(d(u))$。因为 G 中的每一个结点至多可以被检测一次,所以第 3～11行的循环的总时间是 $O(\Sigma d(u)) = O(e)$。VISITED(i)应初始化为 0,1≤i≤n。这要花费 O(n)的时间。从而总的时间是 O(n + e)。如果使用邻接矩阵,那么判断所有邻接于 u 的结点要花 Θ(n)的时间,因此总的时间就变成为 $O(n^2)$。如果 G 是一个由 v 可到达所有结

点的图,那么就要检测所有的结点,因此总的时间至少分别是 $O(n+e)$ 和 $O(n^2)$。故使用邻接表时 $t(n,e)=\Theta(n+e)$,使用邻接矩阵时 $t(n,e)=\Theta(n^2)$。证毕。

如果在一个连通无向图 G 上使用 BFS,则 G 中的所有结点都要被访问,因此该图被周游。但是,若 G 是非连通的,则 G 中至少有一个结点没被访问。通过每一次用一个新的未访问的起始结点来反复调用 BFS,就可以作出对这个图的一次完全周游。所导出的这个周游算法叫做宽度优先周游算法(BFT),定理 7.3 的证明也可以用来证明在一个 n 个结点 e 条边的图上,若使用邻接表,BFT 所要求的时间和附加空间分别是 $\Theta(n+e)$ 和 $\Theta(n)$。若使用邻接矩阵,则分别囿界于 $\Theta(n^2)$ 和 $\Theta(n)$。

算法 7.7 宽度优先图周游算法

```
procedure BFT(G,n)
    //G 的宽度优先周游//
      declare VISITED(n)
      for i←1 to n do   //将所有结点标记为未访问//
        VISITED(i)←0
      repeat
      for i←1 to n do   //反复调用 BFS//
        if VISITED(i) = 0 then call BFS(i) endif
      repeat
    end BFT
```

如果 G 是一个连通无向图,则在第一次调用 BFS 时就会访问 G 的所有结点。如果 G 是非连通的,则至少需要调用 BFS 两次,因此,BFS 可以用来判断 G 是否连通。而且在 BFT 对 BFS 的一次调用时,新近访问的所有结点表明这些结点在 G 的一个连通分图之中,所以一个图的连通分图可以用 BFT 获得。为此,BFS 可以作如下的修改:将新近访问的所有结点都放在一个表中,于是由这个表中的结点所构成的子图和这些结点的邻接表合在一起就构成一个连通分图。因此,若使用邻接表,宽度优先周游算法将在 $\Theta(n+e)$ 的时间内得到这些连通分图。BFT 也可用来得到一个无向图 G 的自反传递闭包矩阵。设 A^* 是这个矩阵,当且仅当 $i=j$ 或者 $i\neq j$ 而 i 和 j 在同一个连通分图中时,$A^*(i,j)=1$。为了判断在 $i\neq j$ 的情况下 $A^*(i,j)$ 是 1 还是 0,可以构造一个数组 CONNEC(1∶n),数组元素 CONNEC(i) 表示结点 i 所在的那个连通分图的标记数,当 CONNEC(i) = CONNEC(j) 时,$A^*(i,j)=1$,反之为 0。这个 CONNEC 数组可以在 $O(n)$ 时间内构造出来。因此,有 n 个结点 e 条边的无向图 G 的自反传递闭包矩阵,不管是使用邻接表还是使用邻接矩阵,都可以在 $\Theta(n^2)$ 的时间和 $\Theta(n)$ 空间内算出(此空间计算不包括 A^* 本身所需要的空间)。

作为宽度优先检索的最后一个应用,考虑获取无向图 G 的生成树问题。当且仅当 G 连通时,它有一棵生成树。从而 BFS 易于判断生成树的存在。此外,考虑在算法 BFS 的第 4～8 行中为到达那些未访问结点 w 所使用的边 (u,w) 的集合。这些边称为前向边 (forward edges)。设 T 表示前向边的集合。我们断言,如果 G 是连通的,则 T 是 G 的一棵生成树。对于图 7.5(a) 所示的图,边集 T 将是 G 中除了 (5,8),(6,8),(7,8) 以外的所有的边的集合 (图 7.6(a))。使用宽度优先检索所得到的生成树称为宽度优先生成树 (breadth first spanning tree)。

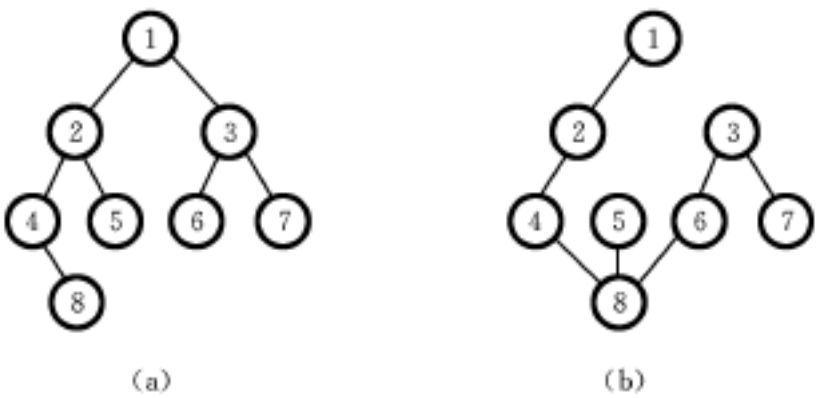


图 7 .6　图 7 .5(a)中的图的 BFS 和 DFS 生成树

定理 7 .4　修改算法 BFS,在第 1 行和第 6 行分别增加语句 T←　和 T←T∪{(u,w)}。修改后的算法叫做算法 BFS* 。如果在 v 是连通无向图中任一结点的情况下调用 BFS* ,那么在算法终止时,T 中的边组成 G 的一棵生成树。

证明　若 G 是 n 个结点的连通图,则这 n 个结点都要被访问。而且除了起始结点 v 外,所有其它的结点都要放到队列上一次(第 5 行),从而 T 将正好包含 n - 1 条边。而且这些边都是各不相同的。因此,T 中的这 n - 1 条边将定义一个关于这 n 个结点的无向图。由于这个图包含由起始结点 v 到所有其它结点的路径(因此在每个结点对之间存在一条路径),所以这个图是连通的。用归纳法容易证明对于有 n 个结点且正好有 n - 1 条边的连通图是一棵树。因此 T 是 G 的一棵生成树。证毕。

宽度优先检索有广泛的应用,其中解决最优化问题的一种重要方法:分枝-限界法(第七章)就是在宽度优先检索基础上建立起来的。

2 .深度优先检索和周游

一个图的深度优先检索与宽度优先检索的差别在于一有新的结点到达就中止对原来结点 v 的检测,且同时开始对新结点 u 的检测。在此新结点被检测后,再恢复对 v 的检测。当所有可达结点全部被检测完毕时,就结束这一检索过程。这一检索过程最好用递归描述成算法 5 .8 那样的形式。

算法 7 .8　图的深度优先检索

```
line procedure DFS(v)
    //已知一个 n 结点的无向(或有向)图 G = (V, E)以及初值已置为零的数组
      VISITED(1：n)。这个算法访问由 v 可以到达的所有结点。G 和 VISITED 是全程量//
1       VISITED(v)←1
2       for 邻接于 v 的每个结点 w do
3          if VISITED(w) = 0 then call DFS(w) endif
4       repeat
5    end DFS
```

图 7 .6(a)起始于结点 1 并且使用图 7 .5(b)的邻接表,对于这个图的深度优先检索导致按 1,2,4,8,5,6,3,7 的次序去访问这些结点。对于 DFS 的非递归算法要使用栈来保留已部分检测的所有结点。容易证明 DFS 访问由 v 可达的所有结点。如果 t(n,e)和 S(n,e)表示 DFS 对一 n 个结点 e 条边的图所花的最大时间和最大附加空间,那么 S(n,e) = Θ(n),在使用邻接表的情况下,t(n,e) = Θ(n + e),而在使用邻接矩阵的情况下,t(n,e) = Θ(n^2)。

一个图的深度优先周游是通过每次用一个新的未访问的起始结点来反复调用 DFS 实

现的。这个 DFT 算法与 BFT 的区别仅在于用对 DFS(i)的调用去代替对 BFS(i)的调用。和在 BFT 的情况下一样,使用 DFT 可以得到一个图的那些连通分图。同样,也可以用 DFT 求出一个无向图的自反传递闭包矩阵。只要对 DFS 作如下修改:把 T← 和 T← T∪{(v,w)}分别加到第 1 行和第 3 行 then 的子句,那么,DFS 终止时,在无向图 G 是连通的条件下,T 中的这些边就定义了 G 的一棵生成树。用这种方法所得到的生成树叫深度优先生成树(depth first spanning tree)。图 7 .5(a)所得到的生成树包含除(2,5),(8,7)和(1,3)以外所有的边(见图 7 .6(b))。因此,DFS 和 BFS 对迄今所讨论的检索问题是等效的。

由以上讨论可知,BFS 和 DFS 是两种根本不同的检索方法。在 BFS 中,一个结点在任何其它结点开始检测之前要完全被检测,这下一个要检测的结点是剩下未检测的第一个结点。习题中还讨论了一种检索方法(D-search),它与 BFS 的区别仅在于下一个要检测的结点是最新到达的未检测结点。在 DFS 中,一旦到达一个新的未检测结点,则中止原来那个结点的检测,并立即开始这个新结点的检测。显然 DFS 和 D-search 的实现都需要一个栈,但这两种检索方法是不同的。本节中所介绍的检索方法可用于各种各样的问题。

7 .2 代码最优化

编译程序的作用是,把按某种源语言写成的程序翻译成一个等效的汇编语言程序或机器语言程序。考察这么一个问题,把按某种语言(例如 PASCAL)写成的算术表达式翻译成汇编语言代码。这种翻译显然依赖于正使用的那种特定的汇编语言(因此,依赖于所用的机器)。开始,假定有一台非常简单的机器模型,把这模型叫做模型机 A。这台机器只有一个称为累加器的寄存器。所有的算术运算都必须在这寄存器中进行。如果⊙代表像 + 、- 、* 、/ 这样的一个双目运算符,则⊙的左运算量必须在这累加器中。为简单起见,将讨论只局限于这 4 个运算符。这一讨论易于推广到其它的运算符。相应的汇编语言指令是

LOAD X…将内存单元 X 的内容装入累加器;

STORE X…将累加器的内容存入内存 X 单元;

OP X…OP 可以是 ADD, SUB, MPY 或 DIV。

指令 OPX 用累加器的内容作为左运算量,X 存储单元的内容作为右运算量进行操作符 OP 的计算。作为一个例子,考虑算术表达式:(a + b)/ (c + d)。这表达式的两个可能的汇编语言模式在表 7 .2 中给出。T1 和 T2 是内存中的临时存储单元。在这两种情况下,结果都留在累加器中。代码段(a)比代码段(b)长两条指令。如果每条指令用的时间量相同,则代码段(b)将比代码段(a)所用的时间少 25%。对于表达式(a + b)/ (c + d)和给定的机器 A,代码段(b)显然是最优的。

定义 7 .1 表达式 E 翻译成某给定机器的机器语言或汇编语言是最优的,当且仅当这一翻译有最少的指令条数。

下面再来看 3 个例子。考虑表达式 a + b * c。表 7 .3 显示了两种可能的翻译。代码段(b)虽不符合 + 的左运算量应放在累加器、右运算量应放在内存储器中的要求,但由于 x + y = y + x,因此代码段(b)与(a)是等效的。

表 7.2　(a+b)/(c+d)两种可能的代码段

LOAD	a	LOAD	c
ADD	b	ADD	d
STORE	T_1	STORE	T1
LOAD	c	LOAD	a
ADD	d	ADD	b
STORE	T_2	DIV	T_1
LOAD	T_1		
DIV	T_2		
(a)		(b)	

表 7.3　a+b*c 的代码段

LOAD	b	LOAD	b
MPY	c	MPY	c
STORE	T_1	ADD	a
LOAD	a		
ADD	T_1		
(a)		(b)	

定义 7.2　双目运算符⊙在定义域 D 中是可交换的,当且仅当对于 D 中所有的 a 和 b,有 a⊙b = b⊙a。

+ 和 * 运算符对于整数和实数是可交换的,而 - 和/ 运算则不是。利用某些运算符的可交换性可能得到较短的代码段。下面考虑表达式 a * b + c * d。表 7.4 显示了两种可能的代码段。代码段(b)实际上计算(a + c) * b,它和 a * b + c * b 相等。

表 7.4　a*b+c*b 的代码段

LOAD	c	LOAD	a
MPY	b	ADD	c
STORE	T_1	MPY	b
LOAD	a		
MPY	b		
ADD	T_1		
(a)		(b)	

表 7.5　a*(b*c)+d*c

LOAD	b	LOAD	a
MPY	c	MPY	b
STORE	T_1	ADD	d
LOAD	a		
MPY	T_1		
STORE	T_1		
LOAD	d		
MPY	c		
STORE	T_2		
LOAD	T_1		
ADD	T_2		
(a)		(b)	

定义 7.3　双目运算符⊛相对于双目运算符　在定义域 D 中是左分配的,当且仅当对于 D 中的每一个 a,b,c,有 a⊛(b　c) = (a⊛b)　(a⊛c)。⊛相对于　是右分配的,当且仅当对于 D 中的每一个 a,b,c,有(a　b)⊛c = (a⊛c)　(b⊛c)。

在实数范围内,* 相对于 + 和 - 是左、右分配的,这是因为 a * (b + c) = (a * b) + (a * c),a * (b - c) = (a * b) - (a * c),(a + b) * c = (a * c) + (b * c)和(a - b) * c = (a * c) - (b * c)。/ 相对于 + 和 - 不是左分配的,因为 a/(b + c) ≠ (a/b) + (a/c)。但是在实数范围内,/ 是右分配的。不过,在整数范围内/ 相对于 + 和 - 不是右分配的。例如,(2 + 3)/5 = 1,而在整数算术中,由于 2/5 = 0,3/5 = 0,得(2/5) + (3/5) = 0。

最后一个例子,考虑表达式 a * (b * c) + d * c。表 7.5 描述了两种可能的代码段。表 7.5(b)的代码段使用了(a * b) * c = a * (b * c)。

定义 7.4　双目运算符⊙在定义域 D 中是可结合的,当且仅当对于 D 中的所有的 a,b,c 有 a⊙(b⊙c) = (a⊙b)⊙c。

对于整数范围和实数范围，* 运算是可结合的，/ 运算是不可结合的。

利用运算符的可结合、分配和交换的性质，可能作出较短的代码段。注意，虽然在实数范围内，有(a + c) * b = (a * b) + (c * b)，但是表 7 .4(a)和(b)的这两个代码段可能算出不同的答案。这种情况的出现是由于计算机有限长的算术运算在计算中会产生一些误差的缘故。在下面的讨论中将略去这一因素，并且假定在可应用结合律、交换律和分配律的时候都可以对它们自由地使用。

对于一个给定的表达式可能给出不同的代码段，下面讨论怎样获取最优代码段的问题。开始，将讨论局限于这台简单的机器 A。然后，再考察更一般的机器模型。运算符出现在它们的运算量之间的表达式的形式称为中缀形式。这是通常写算术表达式的方法。为了产生最优代码段，用二元树来表示算术表达式是方便的。二元树的每一个非叶子结点表示一个运算符。非叶子结点称为内部结点。某内部结点 P 的左子树表示由 P 所代表的运算符的左运算量的二元树形式。而右子树则表示其右运算量的二元树形式。一个叶子结点或者表示一个变量或者表示一个常数。图 7 .7 给出了一些表达式的二元树形式。这些表示算术表达式的二元树称为表达式树。

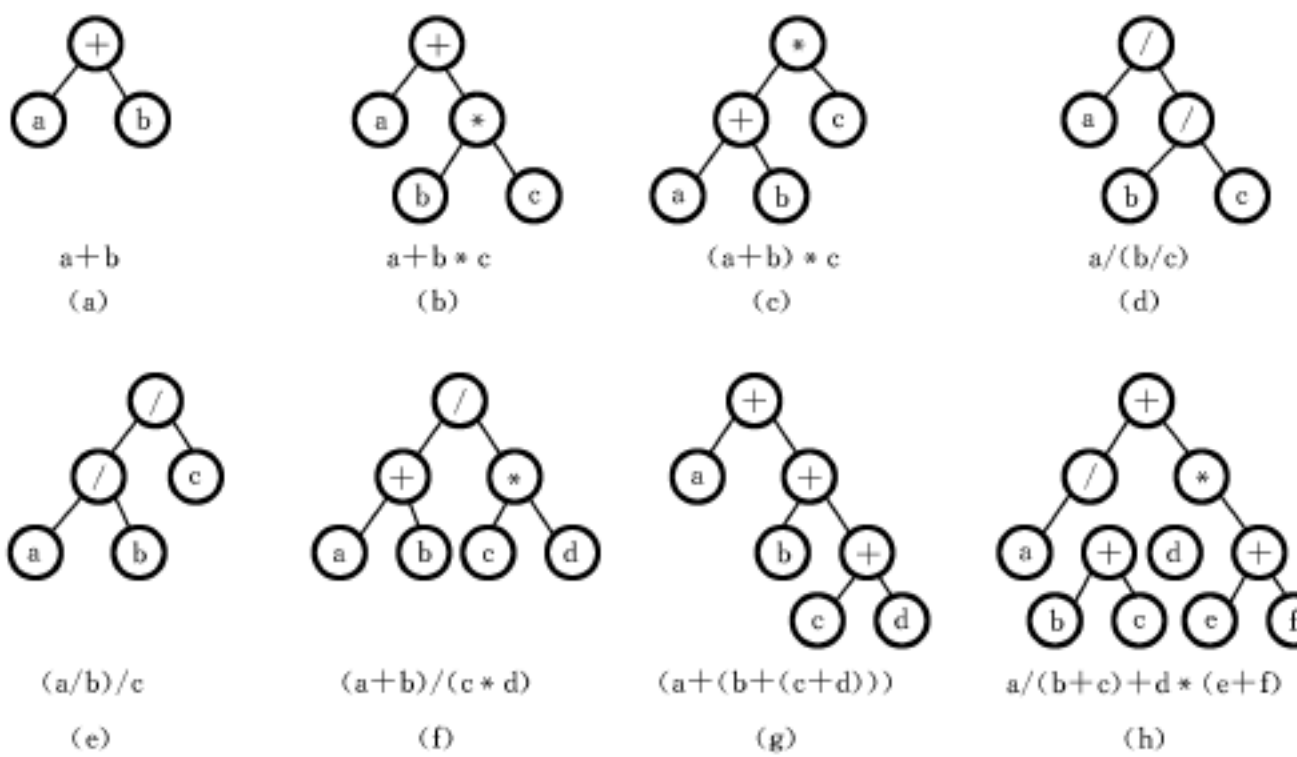


图 7 .7　某些中缀表达式的二元树形式

为了从表达式树得到产生最优代码段的算法，开始，假定所有的运算符既不可交换，也不可分配或结合，也不允许使用代数变换去化简表达式。例如，尽管 a + b - a - b 有值为 0，但在上述假定下，最优代码段将是 LOAD a; ADD b; SUB a; SUB b。我们也不涉及对公共子表达式的处理，所有的子表达式都假定为互不相关的。因此，a * b * (a * b - d)的最优代码段与 a * b * (c * e - d)的最优代码段相同。在这些假定下，易于看出若一个表达式有 n 个运算符，则它的代码段恰好有 n 条 ADD, SUB, MPY, DIV 这种类型的指令。将这类指令称为运算符指令。只有累加器的装入和存放指令条数会变化。例如，表 7 .2(a)和(b)的代码段都有 3 条运算符指令。代码段(a)有 3 条装入和两条存放指令，而代码段(b)只有两条装入和一条存放指令。易于证明在任何没有冗余指令的代码段中，除了第一条以外的每条装入指令都必须紧接在一条存放指令之后。因此，装入指令数总比存放指令数多 1。所以只要产

生使装入指令数或存放指令数为最小值的代码段就是最优的代码。

设 P 是任一表达式树的一个内结点;又设 L 和 R 分别是 P 的左、右子树;再设⊙是在结点 P 的运算符。根据对运算符所作出的假定,计算 L⊙R 的唯一方法是先独立地计算 L 和 R,然后计算 L⊙R,L 和 R 的代码段也应是最优的。假定已给出 L 和 R 的最优代码段 C_L 和 C_P,那么计算 L⊙R 就有表 7.6 所列出的那几种可能性。表中,"条件"这一列揭示了 L 和 R 的各种可能性以及在 L 和 R 不是叶子结点的情况下计算 L 和 R 的次序。在写出的代码中,⊙a 表示一条运算指令。如果⊙是 +,则指的是 ADD a。表 7.6 揭示出,在产生 L⊙R 的代码段时,只有 L 和 R 都是内部结点才有选择的机会。当 L 和 R 中的任何一个为叶子结点时,则(在①,②和③种条件下的)代码段是唯一的(禁止引进无用的代码)。当 L 和 R 都是内部结点时,条件⑤的代码段比条件④的要少些,所以应被采用。由此产生以下的观察结果,如果 R 是内部结点,在最优化码段中 C_R 在 C_L 的前面,否则,C_L 就在 C_R 的前面。

表 7.6　计算 L⊙R 的各种可能性

条　件	相应的代码段
① L 和 R 都是叶子,变量分别是 a 和 b	LOAD a;⊙b
② L 是具有变量 a 的叶子,R 不是叶子	C_R;STORE T_1;LOAD a;⊙T_1
③ R 是具有变量 a 的叶子,L 不是叶子	C_L;⊙a
④ L 和 R 都不是叶子,L 在 R 之前计算	C_L;STORE T_1;C_R;STORE T_2;LOADT_1;⊙T_2
⑤ L 和 R 都不是叶子,R 在 L 之前计算	C_R;STORE T_1;C_L;⊙T_1

以上论述的与用分治方法或动态规划法是类似的。利用分治方法,可以先获得 L 和 R 的最优代码段,然后以某种方法将这些最优代码段结合起来而得出 L⊙R 的最优代码段。用动态规划法可以将代码段看成是一系列决策的结果。其每一步都作出对某一个子表达式接着编码的决策。只有在已经产生了 L 和 R 的代码段后才可以对表达式 L⊙R 编码。

表 7.6 导致代码段的递归生成过程为 CODE1(算法 7.9)。这个算法使用过程 TEMP(i)和 RETEMP(i)。TEMP(i)得到临时存储器的一个存储单元,而 RETEMP(i)则释放临时存储单元 i。假定表达式树有一个由 T 所指示的根结点,并且每一个结点都有 3 个信息段 LCHILD,RCHILD 和 DATA。内部结点的 DATA 信息段是运算符。叶子结点的此信息段是运算量地址。另外,该算法还假定 T≠0。注意,这算法实质上执行对二元树 T 的周游,但是这一周游方法与 7.1.1 节中所讨论的 3 种方法都不相同的是,只有内部结点被访问。当访问一个结点时,就生成该结点的代码段。对一个结点的访问只有在它的两棵子树的代码段生成以后才能进行。这类似于后根次序周游。但是,在算法 CODE1 中,一棵非平凡的右子树在其相应的左子树之前被周游(一棵平凡的子树是只有一个根结点的子树)。如果把临时存储器当成一个栈来处理,而 TEMP 和 RETEMP 分别对应于从这个栈删去和插入的操作,则表 7.7 可显示由 CODE1 对图 7.7 中的某些例子所生成的代码段。根据前面的讨论,可以得出由 CODE1 所生成的代码段相对于机器 A 是最优的。在研究机器 A 的推广形式时,将给出较严密的证明。

定理 7.5　用 CODE1 所产生的代码段能正确计算由表达式树 T 所表示的算术表达式的值。

该定理的证明(对 T 的深度施行简单的归纳),留作习题。

算法 7 .9　生成代码段的算法

```
procedure CODE1(T)
    //假设 T≠0,T 是一棵表达式树,生成 T 的代码段//
    if T 是叶子 then print ("LOAD",DATA(T))
        return
    endif
    F←0   //如果 RCHILD(T)不是叶子,则将 F 置成 1//
    if RCHILD(T)不是叶子 then
        call CODE1(RCHILD(T))   //生成 C_R//
        call TEMP(i)
        print ("STORE ",i)
        F←1
    endif
    call CODE1(LCHILD(T))   //生成 C_L//
    if F = 1 then print (DATA(T),i)
                 call RETEMP(i)
             else print (DATA(T),DATA(RCHILD(T)))
    endif
end CODE1
```

如果允许使用运算符的可交换性,CODE1 在机器 A 上就不能产生出最优代码段。为了看清这一点,考察图7 .7和表7 .7的(b):当 + 可交换时,最优代码段是LOADb,MPYc,

表 7 .7　由 CODE1 对图 7 .7 的一些例子所生成的代码段

LOAD	a	LOAD	b	LOAD	a
ADD	b	MPY	c	ADD	b
		STORE	T_1	MPY	c
		LOAD	a		
		ADD	T_1		
(a)		(b)		(c)	
LOAD	c	LOAD	e		
ADD	d	ADD	f		
STORE	T_1	STORE	T_1		
LOAD	b	LOAD	d		
ADD	T_1	MPY	T_1		
STORE	T_1	STORE	T_1		
LOAD	a	LOAD	b		
ADD	T_1	ADD	c		
		STORE	T_2		
		LOAD	a		
		DIV	T_2		
		ADD	T_1		
(g)		(h)			

ADD a。如果⊙是可交换的,表 7 .6 中的可能性就要增加。为了在可交换运算符的情况下能产生最优代码段,需要修改 CODE1。所需要的修改留下作为习题。

现在,将机器 A 推广到另一种机器 B。B 有 N≥1 个可以执行算术运算的寄存器。对于 B,有 4 种类型的机器指令:

指　令	功能
(1) LOAD M,R	将内存单元 M 的内容装入寄存器 R,1≤R≤N。
(2) STORE R,M	将寄存器 R 的内容存到内存单元 M,1≤R≤N。
(3) OP R_1,M,R_2	执行(R_1)OP(M)的计算,结果放在 R_2 寄存器中。OP 是任何一种双目运算符(例如,+,-,*,/),R_1 和 R_2 是寄存器,M 是一个内存单元。R_1 可等于 R_2。
(4) OP R_1,R_2,R_3	与(3)类似。R_1,R_2 和 R_3 是寄存器。这些寄存器的某两个或全体可以相同。

比较 A 和 B 这两种机器模型,可以看出,当 N = 1 时模型 B 的(1),(2),(3)型指令与模型 A 相应的指令是相同的;(4)型指令只允许在没有附加存储存取装置的情况下执行像 a + a,a - a,a * a 和 a/ a 这样一些普通的运算。这不会改变在 A 和 B 上产生最优代码段指令的条数。因此,当 N = 1 时模型 A 在某种意义上与模型 B 是一样的。就模型 B 而言,一个给定的表达式 E 的最优代码段可以因 N 值的不同而不同。表 7 .8 列出了图 7 .7 所示的表达式(f)在 N = 1 和 N = 2 两种情况下的最优代码段。要指出的是,当 N = 1 时,必须生成一条存放指令,而当 N = 2 时,则不需要生成存放指令。寄存器标记成 R_1 和 R_2。T_1 是存储器中的临时存储单元。并且注意,LOAD 的条数不再需要正好比 STORE 的条数多 1。因此,为了最优化而只考虑 LOAD 指令的条数或者只考虑 STORE 指令的条数就不够了。而是要使它们的和取最小值。为了简化这一讨论,首先假定没有运算符是可结合、可交换或可分配的。并且假定一个运算符的左、右两个运算量即使是相同的子表达式也必须分别进行计算,这一限制被扩大到表达式之类情况,例如 a⊙a,也要求对左、右运算量都作一次内存访问。

表 7 .8　N = 1 和 N = 2 的最优代码段

N = 1		N = 2	
LOAD	c,R_1	LOAD	c,R_1
MPY	R_1,d,R_1	MPY	R_1,d,R_1
STORE	R_1,T_1	LOAD	a,R_2
LOAD	a,R_1	ADD	R_2,b,R_2
ADD	R_1,b,R_1	DIV	R_2,R_1,R_1
DIV	R_1,T_1,R_1		

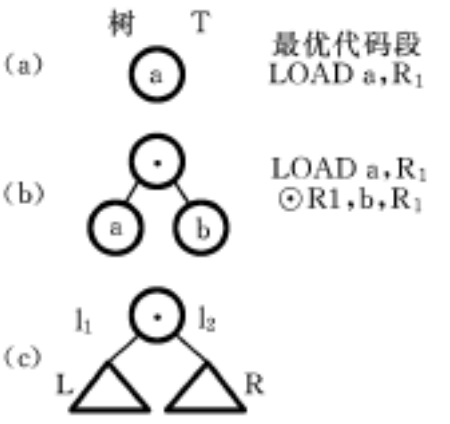


图 7 .8　计算最小的寄存器数

给定一个表达式 E,可能要问的第一个问题是:不用任何 STORE 指令可以算出 E 的值吗?第二个问题是:不用任何 STORE 指令而计算 E 的值所需要寄存器的最小数量是多少?在作了上面的那些假定的条件下来回答这些问题,并且还假定 E 的值将保留在这 N 个寄存器的一个之中。设 E 用一棵表达式树 T 来表示,若 T 只有一个结点,则这个结点必定是一个叶子,显然,所需要的全部工作就是把相应变量的值或常数装入到一个寄存器中。这种情况只需要一个寄存器,如图 7 .8(a)所示。若表达式 E 只有一个运算符,则表达式就是 a⊙b

这种形式。这种情况也只需要一个寄存器 R_1 来装 a,然后可以使用⊙R_1,b,R_1 指令(见图 7.8(b))。当出现一个以上运算符时,则有图 7.8(c)所示的情况。设 l_1 和 l_2 分别是对根运算符的左运算量(L)和右运算量(R)进行单独计算所需要寄存器的最小数。设 l 是计算 L⊙R所需要寄存器的最小数。在作了上述这些假定的情况下就应单独计算 L 和 R 的值,因此,$l \geq \max\{l_1, l_2\}$。如果 $l_1 > l_2$,可以先用 l_1 个寄存器计算 L,并将 L 的值保存在一个指定的寄存器中,然后可用其余的 $l_1 - 1 \geq l_2$ 个寄存器来计算 R。最后,用一条(4)型指令就可计算出 L⊙R。因此,当 $l_1 > l_2$ 时,$l = l_1$。类似地,当 $l_1 < l_2$ 时,$l = l_2$。所以,当 $l_1 \neq l_2$ 时,$l = \max\{l_1, l_2\}$。当 $l_1 = l_2$ 时,有 $l = l_1 + 1$,这是因为无论是先计算 L 或是先计算 R,必须将保存第一个被计算的运算量值的那个寄存器搁置起来,并且需要用另外的 l_1 个寄存器去计算第二个运算量。上述的讨论可导出定理 7.6。

定理 7.6 设 P 是一个深度至少为 2 的表达式树 T 中的结点,函数 MR(P)定义如下:

$$MR(P)=\begin{cases}0 & P\text{ 是一个叶子结点,且是它父亲的右儿子}\\ 1 & P\text{ 是一个叶子结点,且是它父亲的左儿子}\\ \max\{l_1,l_2\} & l_1=MR(LCHILD(P))\\ & l_2=MR(RCHILD(P))\\ & l_1\neq l_2\\ l_1+1 & l_1,l_2\text{ 同上}\\ & l_1=l_2\end{cases}$$

如果不允许使用 STORE 这种指令,那么一个内部结点 P 的 MR(P)是计算其根为 P 的表达式子树所需要的最小寄存器数。

上面的定理仅在对运算符所规定的那些假设下才为真。对于任何一棵表达式树 T,所有结点的 MR 值都可以用 T 的后根次序周游算法计算出来。图 7.9 给出了某些表达式树的所有结点的 MR 值。如果可用的寄存器数 N 大于或等于表达式树 T 的根 T 的 MR 值,那么不用任何 STORE 型指令就能对 T 求值。在这种情况下,最优代码段只需使 LOAD 型指令数极小化即可。根据已作的那些假设,(3)型和(4)型指令数等于内部结点数。当 MR(T) > N 时,代码段必须包含某些 STORE 型的指令,最优代码段将使得(1)和(2)型指令数之和最小。定理 7.6 的证明提供了一个代码段生成算法(算法 7.10)。后面将证明 CODE2 在规定的这些假设条件下生成最优代码段。现在先来理解这个算法。

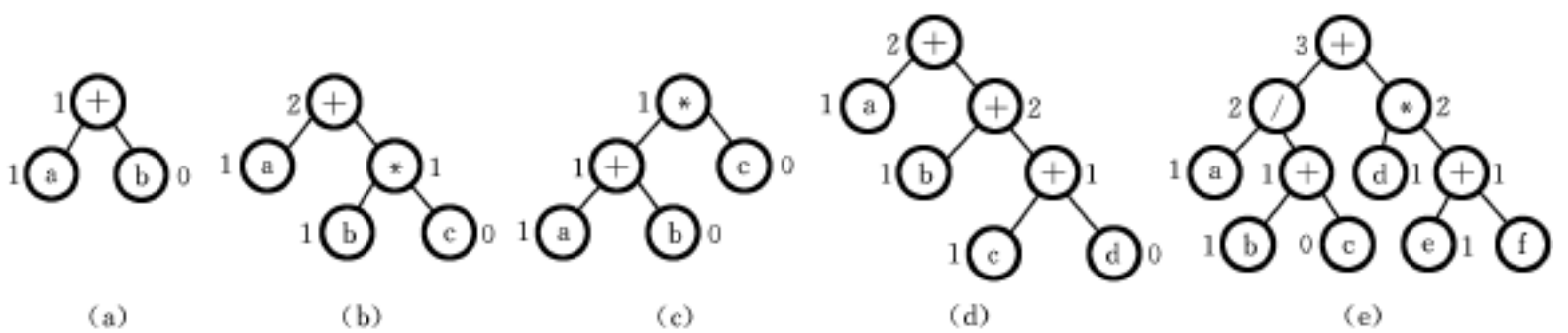


图 7.9 结点的 MR 值(即结点上方的数)

算法假定表达式树 T 中的每个结点有 4 个信息段:LCHILD,RCHILD,DATA 和 MR。MR 的值已像定理 7.6 定义的那样被算出。CODE2 用了两个与 CODE1 中相同的子程序 TEMP 和 RETEMP。为了生成表达式树 T 的代码段,CODE2 以 call CODE2(T,1)的形式

被调用。寄存器总数 N 是一个全程变量。假定 T ≠ 0,即表达式不是空的。对于 CODE2(T,i)的调用,只使用寄存器 $R_i, \cdots, R_N$ 来生成表达式 T 的代码段。结果保留在 R_i 中。在对 CODE2 初次调用的情况下,若 T 是一个叶子,则只产生一条装入指令。若 T 是一个内结点,则进入 case 语句(6～24 行)。L 和 R 分别指向 T 的左、右儿子。设 T 的 DATA 信息段中的运算符为⊙,若 R 是一个叶子,则 MR(R) = 0。因此,L⊙R 的最优代码段是 L 的最优代码段加上这条⊙运算指令。这在 7～9 行生成。而对 L 是一个叶子的情况,只有在第 8 行和第 18 行递归调用 CODE2 时,才可能出现。当在这两处 L 是叶子时,只要在此处再产生一条装入指令。在初次调用 CODE2 时,如果 MR(T) > N,则在此表达式树中必定至少有一个内结点,它的左、右儿子所需的最小寄存器数 MR(L)≥N 且 MR(R)≥N。由定理 7 .6可知,在这种情况下至少要产生一条存放指令。L⊙R 的最优代码段是 R 的最优代码段后跟随一条存放 R 的结果的指令再加上 L 的最优代码段,这在第 10～15 行生成。要指出的是,在 MR(L)≥N 且 MR(R)≥N 的情况下,第 10 行和第 13 行的两次递归调用都允许 CODE2 使用寄存器 $R_i, \cdots, R_N$。对递归深度施以简单的归纳可知,总有 i = 1。不管对 CODE2 是初次调用还是递归调用,如果其实在参数 T 的 MR(L)和 MR(R)至少有一个小于 N 且MR(R)≠0,则在第 16～23 行处理。若 MR(L) < MR(R),则 MR(L) < N。在此情况下,L⊙R 的最优代码段是使用 $R_i, \cdots, R_N$ 这 N - i + 1 个寄存器在 17 行生成 R 的最优代码,然后使用 $R_i + 1, \cdots, R_N$ 这 N - i 个寄存器在 18 行生成 L 的最优代码,再后随一条⊙运算指令。这样处理的根据是定理 7 .7。定理 7 .7 指出,在进入第 16～19 行时,总有 MR(L)≤N - i。而按上述计算了 R 以后有 N - i 个寄存器被释放(即 $R_i + 1, \cdots, R_N$),因此不用 STORE 型指令就可以计算 L。注意,若 MR(R)不小于 N,则 i = 1。当 MR(L)≥MR(R)时,在第 20～23 行处理。由定理 7 .7 知 MR(R)≤N - i,因此,在生成了 L 的最优代码后,不用 STORE 型指令,只用寄存器 $R_i + 1, \cdots, R_N$ 就可计算 R。定理 7 .8 和 7 .9 证明了用算法 CODE2 所生成的代码段的正确性和最优性。若 T 有 n 个结点,则 CODE2 所要求的时间是 Θ(n)(见习题)。

算法 7 .10　机器 B 的代码生成器

```
line procedure CODE2(T,i)
//对有 N 个寄存器的机器 B 只使用寄存器 Ri,…,RN 来生成代码。结果保存在 Ri,N 是全程变量//
1     if T 是叶子 then   //父亲的左儿子//
2        print ('LOAD',DATA(T),'R',i)
3        return
4     endif
     //T 是一个内结点//
5     L←LCHILD(T);R←RCHILD(T)
6     case
7        :MR(R) = 0:   //R 是叶子//
8           call CODE2(L,i)
9           print (DATA(T),'R',i,',',DATA(R),', R',i)
10       :MR(L)≥N and MR(R)≥N:call CODE2(R,i)
11                                  call TEMP(S)
```

```
12                                    print ('STORE', 'R',i, ', ',S)
13                                    call CODE2(L,i)
14                                    print (DATA(T), 'R',i, ', ',S, 'R',i)
15                                    call RETEMP(S)
16        :MR(L) < MR(R):   // MR(L) < N,先计算 R//
17          call CODE2(R,i)
18          call CODE2(L,i + 1)
19          print (DATA(T), ',R',i + 1, ',R',i, ',R',i)
20        :else:   // MR(L)≥MR(R)且 MR(R) < N,先计算 L//
21          call CODE2(L,i)
22          call CODE2(R,i + 1)
23          print (DATA(T), ',R',i, ',R',i + 1, ',R',i)
24        endcase
25    end CODE2
```

定理 7.7　对于 CODE2,下述结论为真:

(1) 每当计算 10～15 行时, $i = 1$;

(2) 每当计算 16～19 行时, $MR(L) \leq N - i$;

(3) 每当计算 20～23 行时, $MR(R) \leq N - i$;

(4) 每当 $MR(T) \geq N$ **时,** $i = 1$。

该证明是对递归的深度施加简单的归纳,留下作习题。

定理 7.8　对每一棵表达式树 T,CODE2 都生成正确的代码段。

证明　对 T 的深度施加简单的归纳。

表 7.9 列出了对于图 7.9 所示的某些表达式生成树用 CODE2 所生成的代码段。 R_1, R_2 **和** R_3 **是寄存器,而** T_1 **是由 TEMP( )所产生出的临时存储单元。**

现在来着手证明 CODE2 生成最优代码段。在证明前有必要区别表达式树中两种类型的结点。

定义 7.5　已知寄存器数目为 N,如果一个结点的两个儿子的 MR 值都至少为 N,则称该结点为大结点(major)。如果一个结点是没有父亲的叶子,或者是它父亲的左儿子叶子,则称该结点为小结点(minor)。

引理 7.1　设 n 是表达式树 T 中的大结点数。当表达式树 T 没有可交换的运算符且在运算符和运算量之间不存在任何关系(即不允许可结合和可分配的运算符以及公共子表达式)时,为了计算 T 的值至少需要 n 条 STORE 型指令。

该引理可以施归纳于表达式树 T 中的结点数来证明。

引理 7.2　对于任何一棵表达式树 T,由 CODE2 所生成的代码段中的 STORE 型指令条数等于表达式树 T 中的大结点数。

证明　这定理是根据观察得出的,12 行是 CODE2 中生成一条 STORE 指令的唯一地方。对于 T 中的每一个大结点,12 行恰好执行一次。

引理 7.3　设 m 是 T 中的小结点数,在引理 7.1 的假设下,计算 T 的代码段必须至少有 m 条 LOAD 指令。

表 7 .9　用 CODE2 对图 7 .9 所示的(a),(b),(e)树所生成的代码段

LOAD a, R_1	LOAD b, R_1	LOAD a, R_1
ADD R_1, b, R_1	MPY R_1, c, R_1	LOAD b, R_2
	STORE R_1, T_1	MPY R_2, c, R_2
	LOAD R_1, a	ADD R_1, R_2, R_1
	ADD R_1, T_1, R_1	
N = 1	N = 1	N = 2
(a)	(b) Ⅰ	(b) Ⅱ

LOAD d, R_1	LOAD a, R_1
LOAD e, R_2	LOAD b, R_2
ADD R_2, f, R_2	ADD R_2, c, R_2
MPY R_1, R_2, R_1	DIV R_1, R_2, R_1
STORE R_1, T_1	LOAD d, R_2
LOAD a, R_1	LOAD e, R_3
LOAD b, R_2	ADD R_3, f, R_3
ADD R_2, c, R_2	MPY R_2, R_3, R_2
DIV R_1, R_2, R_1	ADD R_1, R_2, R_1
ADD R_1, T_1, R_1	
N = 2	N = 3
(e) Ⅰ	(e) Ⅱ

该引理可以通过施归纳于任一表达式树 T 中的小结点数来证明。

引理 7 .4　对于任一表达式树 T,由 CODE2 所生成的代码段中的 LOAD 指令条数等于 T 中的小结点数。

证明　只有第 2 行产生一条 LOAD 指令。对于 T 中的每一个小结点,恰好访问第 2 行一次。

定理 7 .9　在引理 7 .1 的条件下,算法 CODE 2 生成最优代码段。

证明　定理是根据引理 7 .1～7 .4 和下面的观察得出的,在已给的这些假设的条件下,在表达式 T 的所有正确的代码段中,(3)型和(4)型的指令数都等于这棵树 T 的内部结点(即运算符)数。

如果允许可交换和可结合的运算符,表达式 E 就可能变换成一些与之等值的表达式。它们可分别用一些不同的等效表达式树来计算。设 ψ 代表等效表达式树的类,表达式 E 的代码最优化是指用某给定机器或汇编语言,对 ψ 中能生成指令条数最少的那棵表达式树生成具有最少指令数的代码段。在去掉运算符不可交换与不可结合的限制后,定理 7 .9 的结论就不再为真。例如,对于图 7 .9(b)中的那棵表达式树,由于 + 运算是可交换的,因此它的一棵等效的表达式树可表示成图 7 .10(a)所示的形式。而对于图 7 .9(d)所示的那棵树,由于 + 运算是可结合的,它的一棵等效表达式树由图 7 .10(b)给出。在 + 运算是可交换和可结合的情况下,用 CODE2 去生成图 7 .9(b)和(d)所示的代码段就不再是最优代码段了。因为它们不仅需要两个寄存器,而且生成的指令条数比用 CODE2 对图 7 .10 相应的树在只用一个寄存器生成代码段的指令条数还多一些。下面来讨论在运算符可交换和可结合情况下如何得到最优代码段。通过观察可以看出,交换和结合的变换都不会减少表达式中运算符

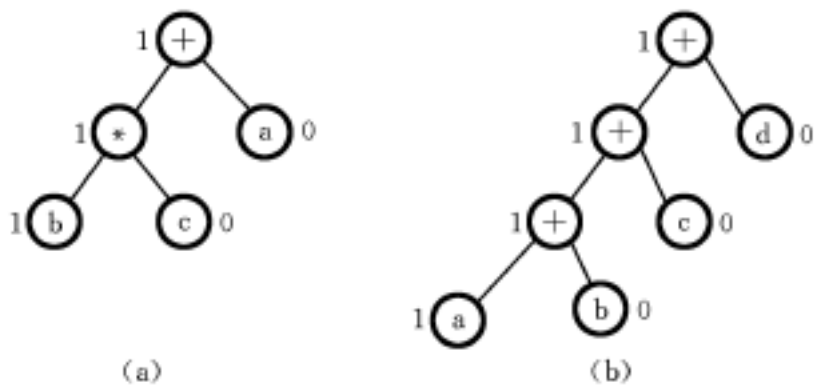


图 7 .10　与图 7 .9 的(b)和(d)等效的树

的数目,因此ψ中所有树的内结点数(即运算符数)相同。由引理 7 .1～7 .4 可以得出,E 的最优代码段应是以ψ中大、小结点和数最小的树作为输入,由 CODE2 所生成的那个代码段。当 E 有可交换的运算符而没有可结合的运算符时,这样的一棵树可以容易地由ψ中任何一棵表达式树 T 得到。交换性只允许交换可交换运算符的左、右两个运算量。如果每一个可交换运算符的左儿子都是一个内结点(除了两个儿子都是叶子的可交换符以外),则大结点和小结点之和就达到最小。于是,对于任意给定的一棵树 T,可以简单地检查出 T 中所有只有一个儿子是叶子的内结点,并在其父结点运算符是可交换的情况下,使这儿子成为此内结点(可交换的运算符)的右儿子,这样就得到一棵可供 CODE2 使用的最优树 T′(见习题)。对于 E 可结合运算符的情况,将它的表达式树作些适当的变换,亦可生成最优代码段,具体讨论留作一道习题。

如果去掉运算符左、右运算量必须单独计算的假定,由于表达式可以有公共子表达式,因此与一个表达式对应的‘表达式’树就变成为一个图。对于有公共子表达式的表达式,要获得最优代码段在计算上是非常困难的。实际上,要确定 MR(E)也非常困难,通常一些在树中能够有效解决的问题在图中就变得非常难于解决了。

7 .3　双连通分图和深度优先检索

假设有两个通信网,一个如图 7 .11 所示,另一个如图 7 .12 所示。图中结点代表通信站,边代表通信线路。这两个图显然都是无向连通图,但却有不同的特性。在图 7 .11 中,如果结点 2 的通信站发生故障,除它本身不能与任何通信站联系外还会导致 1、3、4、9、10 和 5、6、7、8 这些站间的通信中断。结点 3 和结点 5 的通信站若出故障也会发生类似的情况。图

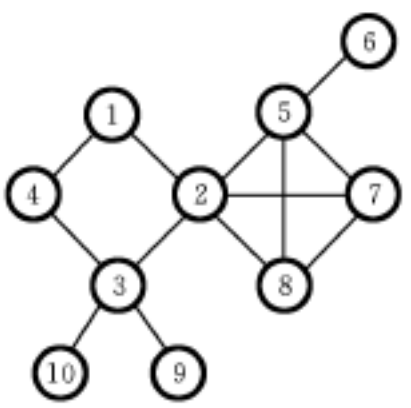


图 7 .11　一个连通图

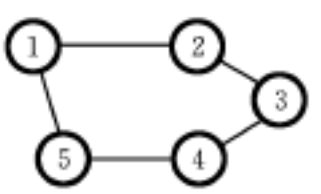


图 7 .12　一个双连通图

7 .12则不然,不管哪一个站发生故障,其余站之间仍可正常通信。出现这种差异的原因在于这两个图的连通程度不同。图 7 .11 是一个含有称之为关节点,即含结点 2、3、5 的连通图,而图 7 .12 是不含关节点的连通图。如果把无向连通图 G 中某结点 a 以及与 a 相关联的所有边删去,得到两个或两个以上的非空分图,那么结点 a 就称为 G 的关节点。将图 7 .11 中的结点 2 和边(1,2),(2,3),(2,5),(2,7)及(2,8)删去后,则留下两个彼此不连通的非空分图,如图 7 .13 所示。如果无向连通图 G 根本不包含关节点,则称 G 为双连通图。

就通信网而言,当然不希望有关节点。如果有关节点就要设法增设一些通信线路,以使整个网成为一个双连通图。本节将设计一个有效算法来测试某个连通图是否双连通;对于不是双连通的图,算法将识别出所有的关节点。在找出所有关节点的情况下,只要确定一个适当的边集增加到原连通图 G 上,就可将其变成一个双连通图。为了确定这个边集,找出图 G 的最大双连通子图是很必要的。G′= (V′, E′)是 G 的最大双连通子图指的是 G 中再没有这样的双连通子图 G″= (V″, E″)存在,使得 V′ V″且 E′ E″。最大双连通子图称为双连通分图(又称双连通支)。图 7 .12 所示的只有一个双连通分图,即这个图本身。图 7 .11 所示的双连通分图在图7 .14中列出。

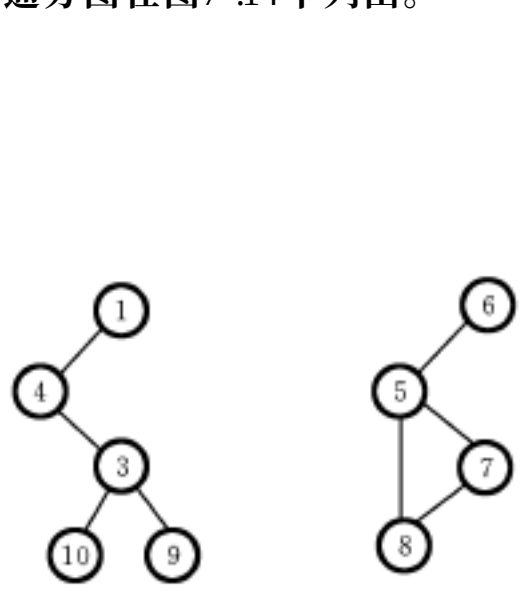


图 7 .13　图 7 .11 删去关节点 2 的结果

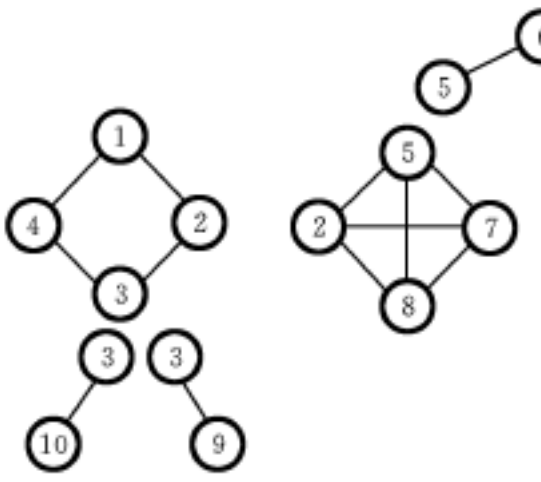


图 7 .14　图 7 .11 所示的图所含有的双连通分图

不难证明两个双连通分图至多有一个公共结点,且这个结点是关节点。因此可以推出任何一条边不可能同时在两个不同的连通分图中(因为这需要两个公共结点)。由此,采用以下的加边方法可以把图 G 变成一个双连通图:

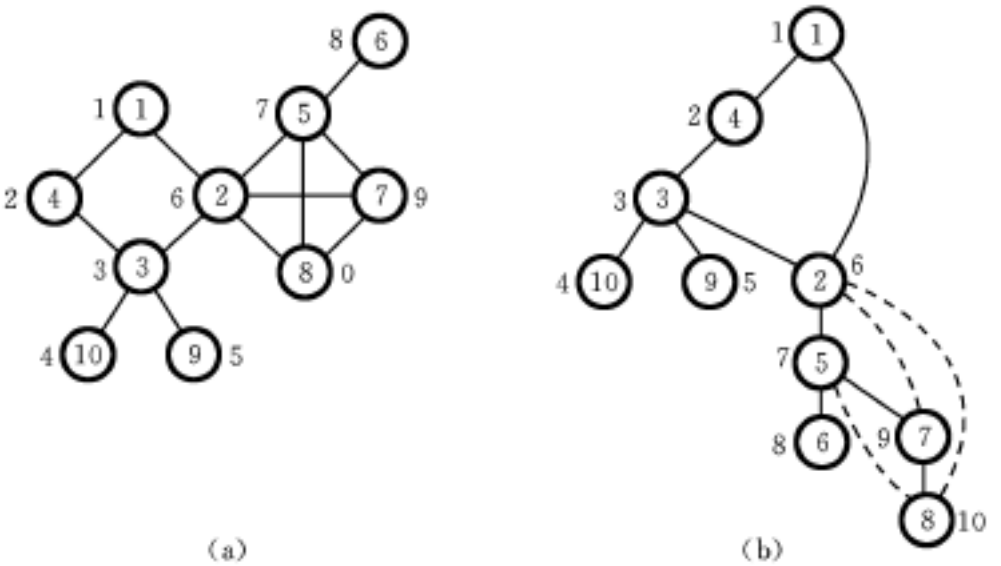
E1　for 每一个关节点 a do
E2　　设 $B_1, B_2, \cdots, B_k$ 是包含结点 a 的双连通分图
E3　　设 v_i 是 B_i 的一个结点 ,且 $v_i \neq a, 1 \leq i \leq k$
E4　　将 $(v_i, v_{i+1}), 1 \leq i < k$,加到 G
E5　repeat

由于 G 的每一个双连通分图至少包含两个结点(除非 G 本身只有一个结点),因此可得出步骤 E3 中确实存在 v_i。使用这个方法将图 7 .11 变成双连通图则需对应于关节点 3 增加边(4,10)和(10,9);对应关节点 2 增加边(1,5);对应关节点 5 增加边(6,7)。由上述方法可以看出,在步骤 E4 增加了那些边(v_i, v_{i+1})之后,结点 a 就不再是关节点了。因此在增加了与所有关节点相对应的那些边后,所产生的图是一个双连通图。如果设图 G 有 p 个关节点,

而与每个关节点对应的双连通分图数为 k_i, $1 \leq i \leq p$。那么,此方法增加的边数总共为 $\sum_{1 \leq i \leq p} (k_i - 1) = \sum_{1 \leq i \leq p} k_i - p$。可以证明此方法增加的总边数比将 G 变成双连通图所需增加的最小边数大。

现在来解决连通图 G 的关节点和双连通分图的识别问题。以下假设 G 有 $n \geq 2$ 个结点。通过考察 G 的深度优先生成树可以使这识别问题得到有效的解决。

图 7.15(a)和(b)显示了图 7.11 所示的深度优先生成树。图中,每个结点的外面都有一个数,它表示按深度优先检索算法访问这个结点的次序,这个数叫做该结点的深度优先数(DFN)。例如,DFN(1) = 1,DFN(4) = 2,DFN(6) = 8 等。图 7.15(b)中的实线边构成这棵深度优先生成树,这些边称为树边。虚线边(即所有剩下的边)称为逆边。



(a) (b)

图 7.15 图 7.11 中的一棵深度优先生成树

识别关节点和双连通分图的算法是依据深度优先生成树的两条非常有用的性质得出的。

性质一 若(u,v)是 G 中任一条边,则相对于深度优先生成树 T,或者 u 是 v 的祖先,或者 v 是 u 的祖先。换言之,深度优先生成树中没有交叉边((u,v)是一条相对于生成树 T 的交叉边指的是 u 不是 v 的祖先,而 v 也不是 u 的祖先)。为了证明这一性质成立,假定(u,v) $\in$ E(G)且(u,v)是一条交叉边。(u,v)不可能是一条树边,如若不然,必有 u 是 v 的父亲或者 v 是 u 的父亲。因此(u,v)是一条逆边。不失一般性,假定 DFN(u) < DFN(v)。于是先访问 u,而对 u 的检测至少要到访问结点 v 后才可能完成。由深度优先检索的定义可得,在 u 的检测完成之前,所有已访问结点都是 u 的子孙。因此,在 T 中 u 是 v 的祖先,从而(u,v)不可能是交叉边。

性质二 当且仅当一棵深度优先生成树的根结点至少有两个儿子时,此根结点是关节点;如果 u 是除根外的任一结点,那么,当且仅当由 u 的每一个儿子 w 出发,若只通过 w 的子孙组成的一条路径和一条逆边就可到达 u 的某个祖先时,则 u 就不是关节点。注意,如果对于 u 的某个儿子 w 不能做到这一点,那么删去结点 u 至少剩下两个非空分图(一个包含根,另一个包含结点 w)。以上性质可导致一条识别关节点的简单规则。对于每个结点 u,定义 L(u)如下:

$$L(u) = \min\{DFN(u), \min\{L(w) \mid w \text{ 是 } u \text{ 的儿子}\}\min\{DFN(w) \mid (u,w) \text{是一条逆边}\}\}$$

显然,L(u)是 u 通过一条子孙路径且至多后随一条逆边所可能到达的最低深度优先数。

由上述讨论可知,如果 u 不是根,那么当且仅当 u 有一个使得 L(w)≥DFN(u)的儿子 w 时,u 是一个关节点。对于图 5.15(b)所示的那棵生成树,各结点的最低深度优先数是:L(1∶10) = (1,1,1,1,6,8,6,6,5,4)。结点 3 是关节点,因为它的儿子结点 10 有 L(10) = 4 而 DFN(3) = 3。结点 2 也是关节点,因为儿子结点 5 有 L(5) = 6 而 DFN(2) = 6。最后一个关节点是结点 5,因为儿子结点 6 有 L(6) = 8 而 DFN(5) = 7。

按后根次序访问深度优先生成树的结点,可以很容易地算出 L(u)。于是,为了确定图 G 的关节点,必须既完成对 G 的深度优先检索,产生 G 的深度优先生成树 T,又要按后根次序访问树 T 的结点。不过这两件工作可以同时完成。过程 ART 实现 G 的深度优先检索;在检索期间,对每个新访问的结点赋予深度优先数;同时对这棵树中每个结点的 L(i)值也进行计算。这个算法假定连通图 G 和数组 DFN、L 是全程量,num 也是全程变量。显然,由算法可以看出,在结点 u 检测完毕从第 9 行返回时,L(u)已正确地算出。要指出的是,在第 5 行中,如果 w≠v,则或者(u,w)是一条逆边,或者 DFN(w) > DFN(u)≥L(u)。在这两种情况下 L(u)都能得到正确的修正。对于 ART 的初次调用是 call ART(1,0)。在调用 ART 之前 DFN 初始化为 0。

算法 7.11　计算 DFN 和 L 的算法

```
line procedure ART(u,v)
     // u 是深度优先检索的开始结点。在深度优先生成树中,u 若有父亲,那么 v 就是它的父亲。假设
        数组 DFN 是全程量,并将其初始化为 0。num 是全程变量,被初始化为 1。n 是 G 的结点数 //
     global DFN(n), L(n), num, n
1    DFN(u)←num; L(u)←num; num←num + 1
2    for 每个邻接于 u 的结点 w do
3       if DFN(w) = 0 then call ART(w,u)    // 还没访问 w //
4                          L(u)←min (L(u), L(w))
5                       else if w≠v then L(u)←min (L(u),DFN(w))
6                       endif
7       endif
8    repeat
9    end ART
```

如果连通图 G 有 n 个结点 e 条边,且 G 由邻接表表示,那么 ART 的计算时间为 O(n + e)。因此,L(1∶n)可在时间 O(n + e)内算出。一旦算出 L(1∶n),G 的关节点就能在 O(n)时间内识别出来。因此,识别关节点的总时间不超过 O(n + e)。

如何判断 G 的双连通分图呢?要是在第 3 行调用 ART 之后有 L(w)≥L(u),就可断定 u 或者是根,或者是关节点。不管 u 是否为根,也不管 u 有一个或是多个儿子,将边(u,w)和对 ART 的这次调用期间遇到的所有树边和逆边加在一起(除了包含在子树 w 中其它双连通分图的边以外),构成一个双连通分图(它的形式证明将在定理 7.10 的证明中给出)。因此,为了得到双连通分图,对 ART 需作以下修改。

(1) 引进一个用来存放边的全程栈 S。

(2) 在 2 到 3 行间增加一行:

```
2.1   if v≠w and DFN(w) < DFN(u) then 将(u,w)加到 S 的顶部
```

```
        endif
```

注意:当且仅当 v = w 或者 DFN(w) > DFN(u)时,(u,w)早已存在栈中。

(3) 在 3 到 4 行间增加下列行:

```
3.1   if L(w)≥DFN(u) then print ('new biconnected component')
3.2     loop
3.3       从栈 S 的顶部删去一条边
3.4       设这条边是(x,y)
3.5       print ('(',x,',',y,')')
3.6     until ((x,y) = (u,w) or (x,y) = (w,u)) repeat
3.7     endif
```

可以证明算法 ART 在增加了这些内容之后,其计算时间仍然是 O(n + e)。下面来证明这算法的正确性。

定理 7.10　当连通图 G 至少有两个结点时,增加了 2.1 和 3.1～3.7 行的算法,ART 能正确地生成 G 的双连通分图。

证明　当 G 只有一个结点时,它没有边,因此这过程不产生输出。G 的双连通分图就是这单个结点。对此情况可作单独处理。

当 G 的结点数 n≥2 时,算法能正确运行,这可通过施归纳于 G 的双连通分图数来证明。如果 G 只有一个双连通分图,即 G 是双连通图,显然,它的深度优先生成树的根 u 只有一个儿子 w,而且 w 是 3.1 行中使得 L(w)≥DFN(u)的唯一结点。到 w 被检测完时,G 中所有的边已作为一个双连通分图输出。

现假定该算法对至多有 m 个双连通分图的所有连通图 G 都能正确执行。下面证明对于有 m + 1 个双连通分图的所有连通图这个算法也能正确执行。考虑 3.1 行中第一次出现 L(w)≥DFN(u)的情况。此时还没有任何边被输出,因此 G 中与 w 子孙相关联的所有边都在栈中,且在边(u,w)的上面。由于 u 的子孙都不是关节点而 u 是一个关节点,因此 S 栈中(u,w)上面的边集和边(u,w)一起构成一个双连通分图。一旦将这些边从栈 S 中删除并输出,此算法基本上相当于在一个从 G 中删去这个双连通分图后所剩下的图 G′上运行。算法在 G 和 G′上运行的差别仅在于以下一点,即,在完成对结点 u 的检测期间,可能要考虑刚输出的分图中的那些边(u,r)。然而,对于这些边都有 DFN(r) ≠0 和 DFN(r) > DFN(u)≥L(u),因此,这些边只会使第 2～8 行的循环作些无意义的迭代而并不会影响这个算法。

容易证明 G′至少有两个结点。又由于 G′正好有 m 个双连通分图,因此,由归纳假设可以得出这 m 个双连通分图能正确地生成。证毕。

要特别指出的是,上面描述的算法要在生成树满足以下条件的环境中工作,相对于这棵生成树,所给定的图没有交叉边。而相对于宽度优先生成树,一些图可能有交叉边,因此算法 ART 对 BFS 不适用。

7.4　与/或图

很多复杂问题很难或没法直接求解,但可以分解成一系列(类型不同)的子问题,而这些子问题又可反复细分成一些更小的子问题,一直到分成一些可普通求解的、相当基本的问题

为止。然后,由这些分解成的子问题的全部或部分解再导出原问题的解。这种将一个问题分解成若干个子问题,又由子问题的解导出原问题解的方法称为问题化简。问题化简已在工业调度分析、定理证明等方面得到应用。

把复杂问题分解成一系列子问题的过程可以用如下结构的有向图来表示:在有向图中,结点代表问题,一个结点的子孙代表与其相联系的子问题。为了暗示父结点的解可由哪些子问题联合导出,则用一条弧将那些能联合导出其解的子结点连接在一起。例如,图 7 .16(a)表示问题 A 可以通过求解子问题 B 和 C 来解出,或者可由单个求解子问题 D 或 E 来解出。边〈A,B〉和〈A,C〉则用一条弧连在一起。为了使图中的每个结点含义单一化,即它的解或者需要求解它所有的子孙得到,或者求解它的一个子孙就可得到,通过引进像图 7 .16(b)那样的虚结点可达到此目的。这前一类结点称为与结点,后一类结点称为或结点。由与结点出发的所有边用一条穿过它们的弧相连结。图 7 .16(b)的 A 和 A″是或结点,A′是与结点。没有子孙的结点是终结点,它代表基本问题并标记上可解或不可解。可解的终结点用方框表示。

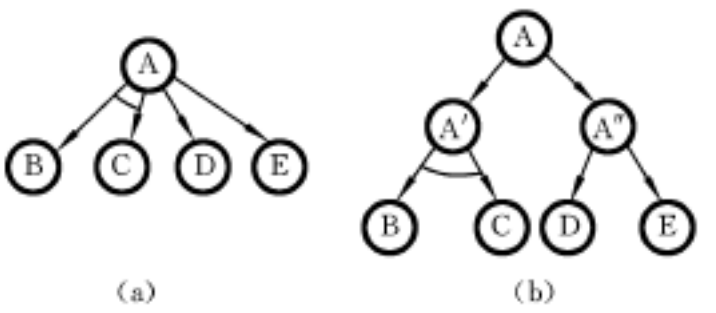


图 7 .16　表示问题的图

下面来看一个例子,某人一星期洗一次衣服,所要做的事有:收集脏衣服、洗衣服、把衣服弄干、熨平、叠好并归堆。其中,某些事可采用不同的方法,如洗衣服一项可以是手洗也可以是机器洗。对于这个问题可以构造出图 7 .17 所示的那样的一棵与/ 或树。图中,手洗的那个结点没有子孙也不是方形结点,它表示此人不采用手洗的方法。当然,洗衣服问题是个非常简单的问题,而实际应用中很多问题远非如此简单,因此使用问题化简就有了现实的意义。

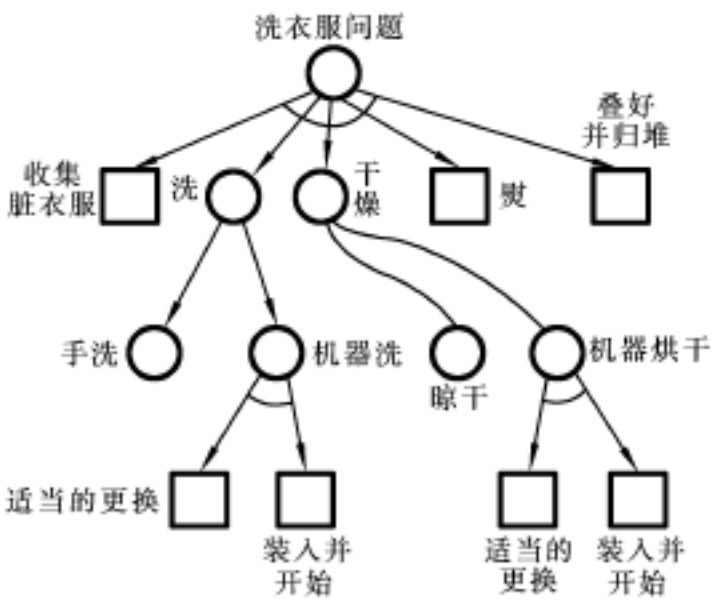


图 7 .17　洗衣服问题对应的与/ 或图

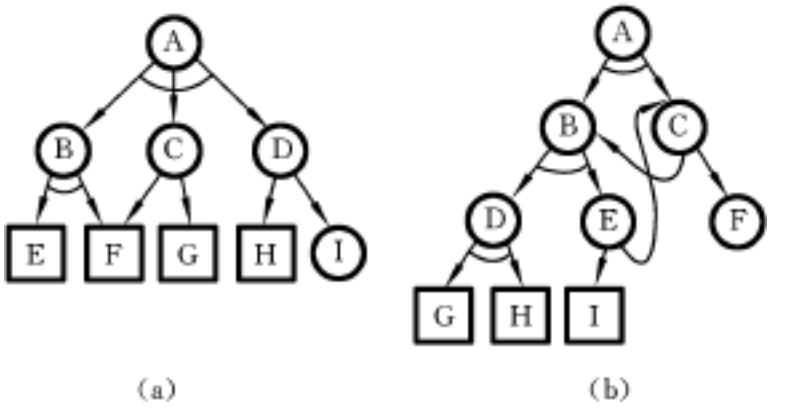


图 7 .18　两个不是树的与/ 或图

在对问题化简时,如果两个子问题在分解成的更小子问题中有一个公共的更小子问题,而这个更小的子问题只需求解一次,则在该问题的与/ 或图上可用一个结点来表示这个更小的公共子问题。图 7 .18 显示了两个出现这种情况的与/ 或图。这样的与/ 或图就不再是树了,而且可能出现像图 7 .18(b)所示的有向环。要指出的是,有向环的出现并不意味着该问

题不可解。事实上,图 7.18(b)所示的问题 A 可通过求解基本问题 G,H 和 I 来导出其解。即通过求解 G,H 和 I 导出 D 和 E 的解,因此也就能导出 B 和 C 的解。下面再引进一个概念:解图是由与/或图中一些可解结点组成的子图,它表示对问题求解。图 7.18 所示的两个图的解图由粗线条示出。

下面,只考虑问题的分解过程可以用与/或树来表示的情况,在这种情况下,如何根据问题的与/或树来判断该问题是否可解呢?这只需对这棵与/或树作后根次序周游就可得出答案。算法 7.12 对这一判断过程作了具体的描述。在算法执行过程中,一旦发现某与结点的一个儿子结点不可解(第 6 行),或者发现某或结点的一个儿子结点可解(第 11 行),就立即终止该算法,这可减少算法的工作量且对结果无任何影响。

算法 7.12　判断与/或树是否可解算法

```
line procedure SOLVE(T)
        //T 是一棵其根为 T 的与/或树,T≠0。如果问题可解则返回 1,否则返回 0//
1       case
2          :T 是终结点:ifT 可解   then return(1)
3                                 else return(0)
4                    endif
5          :T 是与结点: for T 的每个儿子 S do
6                          if SOLVE(S) = 0 then return (0)
7                            endif
8                          repeat
9                          return (1)
10      :else:for T 的每个儿子 S do   //或结点//
11              if SOLVE(S) = 1 then return (1) endif
12           repeat
13           return(0)
14      endcase
15      end SOLVE
```

对于一个给定的复杂问题,不仅需要知道此问题是否可解,而且希望知道如果问题可解,那么此问题的解是由哪些基本问题、沿着什么样的途径所导出的,即希望求出问题的解树。由于解树是与/或树的全体或一部分,因此求解树的算法可在生成与/或树算法的基础上加上一些对结点可解性的判断和删除措施而获得。因为与/或树结点的生成取决于问题的分解方法,假定问题的分解方法可用函数 F 来表示,所以对于一个已经生成的结点,可用函数 F 去生成它的所有儿子。而生成结点的次序既可按宽度优先也可按深度优先的次序来生成。因此,解树的结点也需用函数 F 并可按宽度优先或深度优先的次序来生成。不过要指出的是,一棵与/或树可能有无穷的深度,在使用解树的深度优先生成算法的情况下,即使已知解树存在,算法也可能导致所有生成的结点都在一条由根出发的无穷深度的路径上,从而根本就不能确定出一棵解树,这一点可通过对生成深度作出某种限制获得解决。譬如生成的深度只准达到某个 d,凡在深度 d 处的非终止结点都标记为不可解。这样只要有一处的深度不大于 d,就可保证生成一棵解树。宽度生成算法没有这样的缺点。因为每个结点都只有有限个儿子,所以与解树相对应的与/或树中任何一级都不可能有无穷多个儿子。于是,

只要存在解树,宽度优先生成算法就一定可以将其找出,而且找出的还是一棵具有最小深度的解树。不过,如果与/ 或树中由根出发的所有路径都有无穷的深度,宽度优先生成算法也会出现不终止的情况。这可通过限制所希望得到的解树的深度获得解决。

过程 BFGEN 是一个解树的宽度优先生成算法。如果解树存在,则算法生成与/ 或树的一棵宽度优先解树,而与/ 或树则是在结点 T 开始,应用儿子生成函数所得到的。BFGEN 使用了一个与 SOLVE 类似的子算法 ASOLVE(T)。该子算法对部分生成的与/ 或树 T 作一次后根次序周游,并且将结点标上可解、不可解或可能可解的标记。由于 T 不是一棵完整的与/ 或树,因此它有 3 类叶子结点:第一类结点是非终止叶子结点。由于非终止叶子结点还没检测,因此对其可解性暂时没法判定,故将其标记为可能可解。其它两类叶子结点是完整与/ 或树的叶子,故根据叶子结点所代表问题的可解性标上可解或不可解。如果一个非叶子结点是与结点,则只要它有一个儿子不可解它就不可解。而对于一个非叶子结点的或结点,若它至少有一个儿子可解,则该结点就是可解的。所求得的任何不可解的结点都可从 T 中删去(第 7 行)。对于任何不可解结点 P 的子孙,也没有必要检测,因为,即使某些子孙可解,P 也不能解出,所以第 9 行从队列中删去 P 的所有还没检测的子孙。如果已求出某结点可解,则没有必要进一步去检测那些还没检测的子孙,这一工作也在第 9 行完成。容易证明,如果存在一棵对应于(T,F)的解树,那么 BFGEN 就一定会找到这棵树。

注意:如果找到了这样的一棵树,那么 T 就指向它的根并且这棵树可能有某些为了求解整个问题并不需要求出的可能结点,对 T 另外作一次扫描可以消去这些多余的结点。

算法 7 .13　宽度优先生成解树

```
line procedure BFGEN(T,F)
    //F 生成 T 中的儿子结点;T 是根结点。终止时,如果存在解树,则 T 是这解树的根//
1       将队列 Q 初始化为空;        V←T
2       loop
3         用 F 生成 V 的那些儿子
          //检测 V//
4         if V 没有儿子 then 标记 V 为不可解
                        else ①将 V 的所有不是叶子结点的儿子放入队列 Q,将那些叶子结点
                               分别标上可解或不可解
                             ②把 V 的所有儿子加入树 T
5       endif
6       call ASOLVE(T)
7       从树 T 删去所有标记为不可解的结点
8       if 根结点 T 标记为可解 then return (T) endif
9       从队列 Q 中删去以下的所有结点:它们在 T 中曾有一个祖先被标记为不可解或者在 T
        中有一个标记为可解的祖先
10        if Q 为空 then print ('no solution');stop endif
11        删去队列 Q 的第一个元素;设此结点是 V
12        repeat
13    end BFGEN
```

7.5 对 策 树

本节讨论树在博弈游戏中的应用。在一盘棋赛中,对弈各方都要根据当前的局势,分析和预见以后可能出现的局面,决定自己要采取的各种对策,以争取获得最好的结果。博弈是一种竞争,而竞争现象广泛存在于社会活动的许多方面,因此本节的内容可以很自然地引申并应用于含有竞争现象的政治、经济、军事、外交等各个领域。

首先,来看一个非常简单的拾火柴棍游戏。假定盘上放有 n 支火柴,由弈者 A 和 B 两个人参加比赛。比赛规则是:两名弈者轮流从盘中取走火柴,每次从盘中取走 1,2 或 3 支火柴均为合法着。否则,为非法着;拿走盘中最后一支火柴的弈者则负了这一局,当然另一名弈者则胜这一局。任何时刻盘中剩下的火柴数都表示此时刻的棋局。拾火柴棍游戏在任一时刻的状态则由此时的棋局和轮到走下一着的弈者一起所决定。终局是表示胜局、负局或和局情况的棋局。其它棋局都是非终止棋局。在拾火柴棍游戏中只有一种终局形式,即盘中没火柴棍了。因此不是 A 胜就是 B 胜,不可能出现和局。

在以下条件成立时,棋局序列 $C_1, C_2, \cdots, C_m$ 称为有效(棋局)序列。

(1) C_1 是开始棋局。

(2) C_i $(0 < i < m)$是非终止棋局。

(3) 由 C_i 得到 C_{i+1} 是走下述棋着实现的。若 i 是奇数,则 A 者走一合法着;若 i 是偶数,则 B 者走一合法着。假定只存在有穷的合法着。

C_m 为终局的一个有效棋局序列 $C_1, C_2, \cdots, C_m$ 是此游戏的一盘实际战例。序列 $C_1, C_2, \cdots, C_m$ 的长度是 m。一种有限次的博弈游戏不存在无限长的有效序列。有限次博弈游戏所有可能的实际战例可以用一棵对策树来表示。图 7.19 描述了在 n = 6 情况下拾火柴棍游戏的对策树。树中,每个结点表示一种棋局,根结点表示开始棋局 C_1。通过 A 或 B 的走子作出由上一级到下一级的变换。奇数级所作的变换表示 A 走的棋着,偶数级的变换则是 B 走的棋着。在图 7.19 中,方形结点表示轮到 A 走子时的棋局,圆形结点表示轮到 B 走子时的棋局。由 1 级结点到 2 级结点的边上标出了 A 所走的棋着(例如,边上标出数为 1 表示拿走一支火柴),2 级结点到 3 级结点的边上则标出了 B 所走的棋着,其余各级结点间边上标出数的含意依此类推。终局用叶结点表示,叶结点中标出在此终局获胜者的名字,由拾火柴棍游戏的特点可以看出,弈者 A 只有在奇数级的叶结点处才能获胜,B 只有在偶数级的叶结点处才能获胜。对策树中任一结点的度数至多等于不同的合法棋着数。而根据定义,任一种棋局的合法着数是有穷的,因此对策树中所有结点的度数是有穷的。拾火柴棍游戏中任一棋局至多有 3 种合法棋着,因此它的对策树中所有结点的(出)度不大于 3。一棵对策树的深度是它所表示的博弈游戏中最长实际战例的长度。图 7.19 所示的对策树的深度是 7。因此,从开始到结束,这个游戏至多包含 6 次走子。根据以上引入的各种概念和术语,不难将中国象棋、国际象棋、围棋等有限次博弈游戏的对策树构造出来。但要指出的是,像中国象棋、国际象棋这类博弈游戏,严格说来,它们并不是有限次博弈游戏,因为在对弈过程中一些棋局可能重复出现。不过只要规定不允许出现重复棋局,例如把棋局的重复看成是出现和局,就可把这类博弈看成是有限次博弈,从而能构造出它们的对策树。

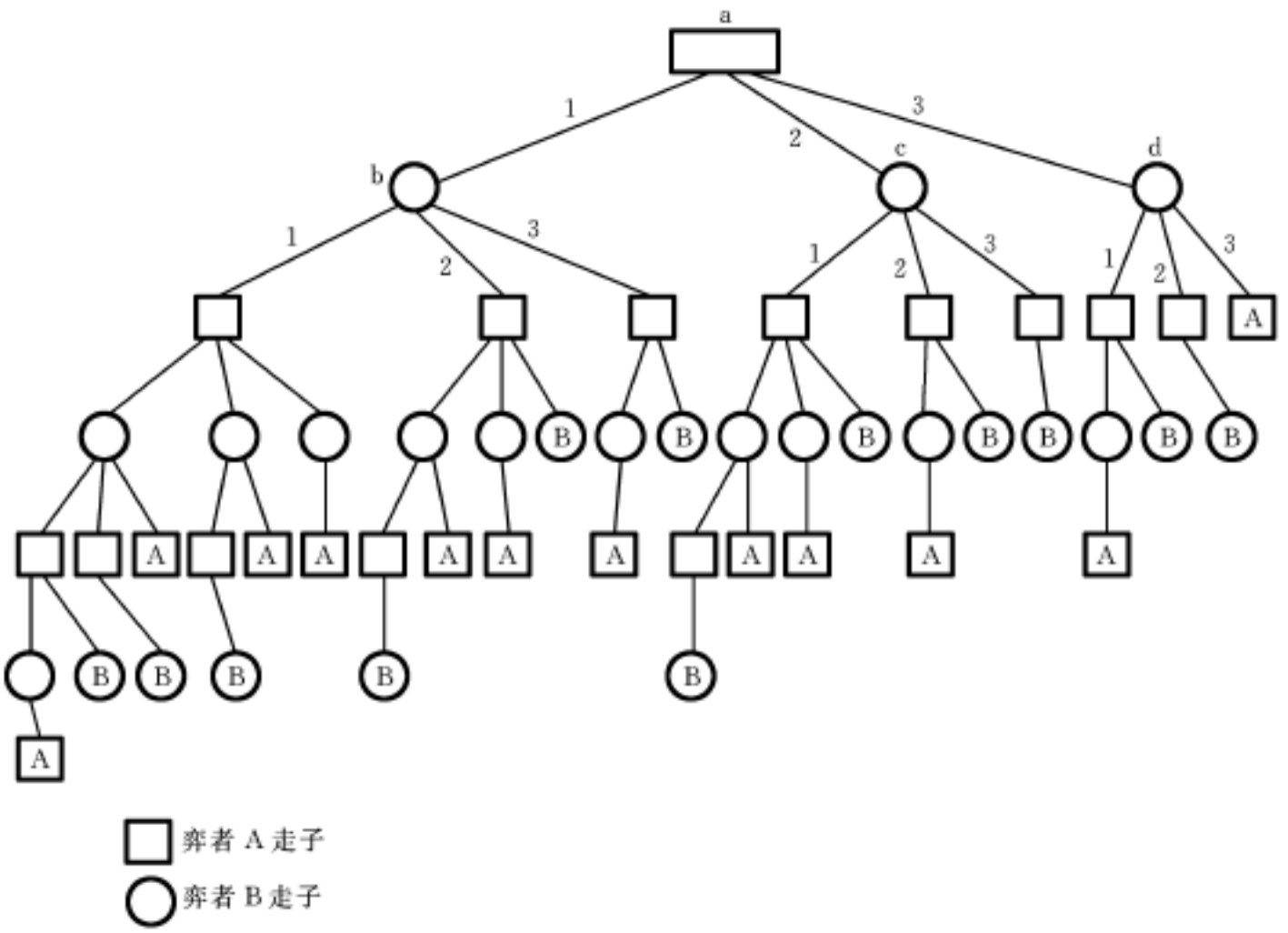


图 7 .19　n = 6 的拾火柴棍游戏的完整对策树

对策树在决定选取什么对策即确定弈者下一步应走哪着棋上是很有用的。例如,在图 7 .19 中,弈者 A 面临由根所表示的初始棋局,他应取哪一种走法才能使自己获胜的机会最大呢?棋着的选择可以用一个估价函数 E(X)来决定。E(X)以数值形式表示弈者在棋局 X 下获胜机会的大小,即它是棋局 X 对弈者价值大小的量度。设 E(X)是弈者 A 的估价函数,那么棋局 X 若使 A 有相当的获胜机会,则 E(X)的取值就高;若 X 使 A 有较多失败的可能,则 E(X)值就低。那些使 A 获胜的终止棋局或不管 B 如何应着都保证 A 能获胜的棋局,E(X)取最大值;而对于能保证 B 取胜的棋局,E(X)则取最小值。

对于其对策树结点数非常少的博弈游戏,例如,n = 6 的拾火柴棍游戏,只要对终局定义 E(X)就足以判断弈者 A 在各步应选择的棋着。可将 E(X)定义为

$$E(X)=\begin{cases}1 & \text{若 X 对 A 是胜局}\\ -1 & \text{若 X 对 A 是负局}\end{cases}$$

我们希望利用刚才所定义的这个估价函数来确定弈者 A 在棋局 a 下应走哪一着棋,即:将比赛导向 b、c、d 三种棋赛的哪一种棋局。设 V(X)是棋局 X 的价值值,那么所作的选择就应使其价值值为 max{V(b),V(c),V(d)}。对于叶结点 X,V(X)取成 E(X)。对于其余的结点,若设 d≥1 是 X 的度,$c_1, c_2, \cdots, c_d$ 是 X 的儿子们所表示的棋局,那么 V(X)可表示为

$$V(X)=\begin{cases}\max\limits_{1\le i\le d}\{V(c_i)\} & \text{若 X 是方形结点}\\ \min\limits_{1\le i\le d}\{V(c_i)\} & \text{若 X 是圆形结点}\end{cases}\qquad(7.1)$$

这种表示要证明其正确是相当简单的。若 X 是方形结点,那么它在奇数级上,只要比赛

到达这个结点就轮到 A 从此处走子。由于 A 想获胜,因此他当然应走到有最大值的那个儿子结点。若 X 是圆形结点,则 X 就在偶数级上,只要比赛到达这个结点就轮到 B 从此处走子。由于 B 也希望取胜,因此他应走一着使 A 取胜机会最小的棋,即走到对 A 有最小价值值的儿子结点。由以上所述可知,式(7 .1)定义了确定棋局 X 价值值的最大最小过程。

图 7 .20 所示的是对一种假想的博弈游戏图解的最大最小过程。$P_{1,1}$ 表示应由 A 从此处走一着随便一种什么棋局。叶结点的价值值由估价函数 E(X)来获得。从第 4 级开始,各

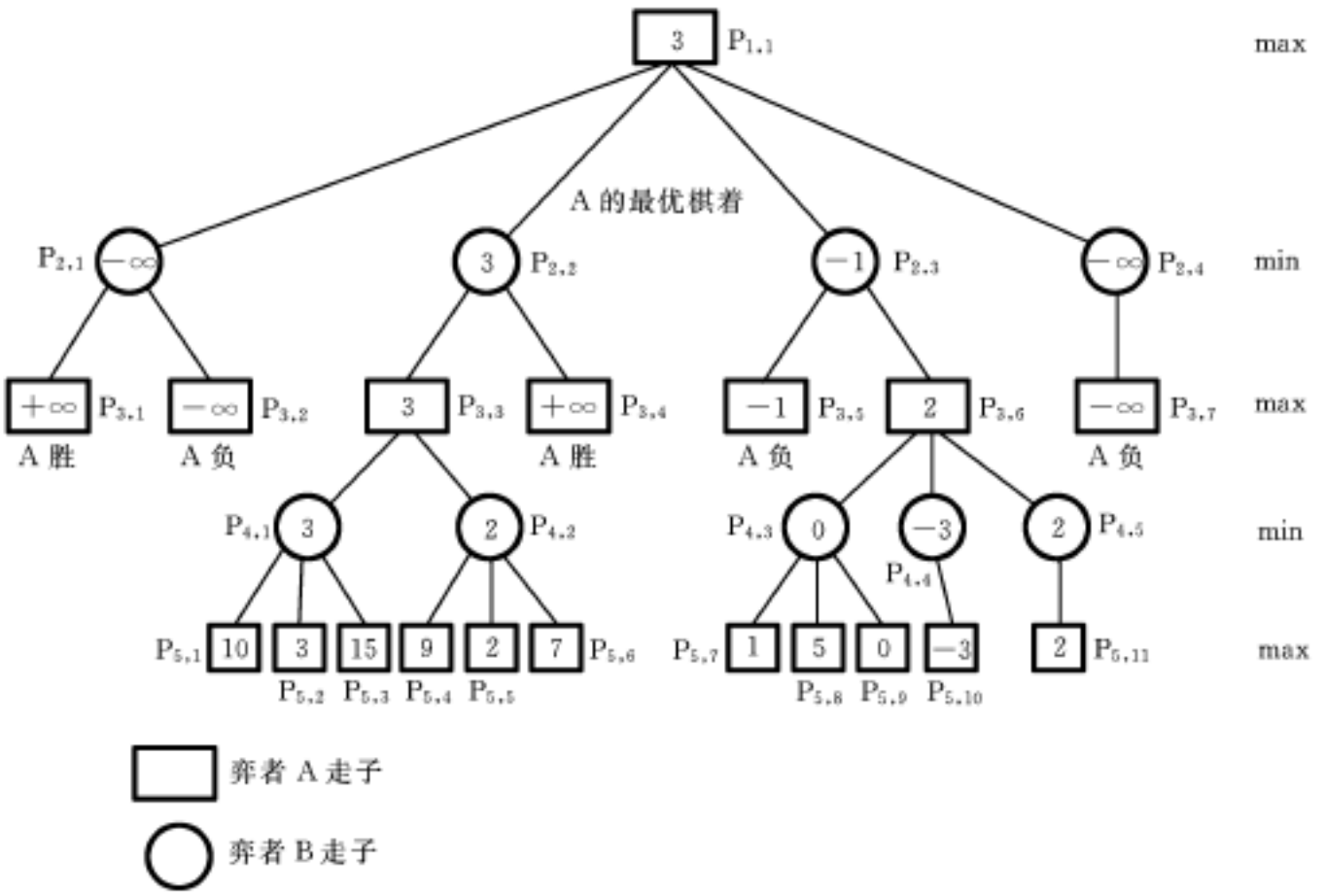


图 7 .20 一种假想博弈游戏的部分对策树

(终止结点的值是由弈者 A 的估价函数 E(X)得出的)

个结点的价值值可由式(7 .1)算得,直到算出 $P_{1,1}$ 的值为止。由于第 4 级都是圆形结点,因此这级上的所有未知值可以通过取相应结点的儿子们的最小值来得到,接着 3,2,1 级上的值也依其顺序算出。$P_{1,1}$ 的结果值是 3,这意味着从 $P_{1,1}$ 开始,A 可以希望的最好结果是到达值为 3 的棋局。虽然有些结点的值比 3 大,但只要在 B 不失着的情况下都不可能到达这些结点(这里还假定 A 的估价函数对 B 来说也是最优的,即 A、B 取类似的估价函数)。例如,A 企图在 $P_{3,1}$ 处获胜而走了一着到 $P_{2,1}$,但出乎 A 的意料,B 应着使棋局变为 $P_{3,2}$,从而导致 A 的惨败。因此,A 应走的最佳着是到 $P_{2,2}$。要指出的是,即使 A 走到 $P_{2,2}$,也可能仍然不能导致棋局 $P_{5,2}$,这是因为在一般情况下,B 会采用与 A 不同的估价函数,从而对于各种棋局产生与 A 完全不同的价值值。总之,最大最小过程可以在给定估价函数的情况下确定一个弈者可能走的最佳棋着。在拾火柴棍游戏的对策树(见图 7 .19)上应用最大最小过程,可以得出 V(a) = 1,这是因为此种游戏的 E(X)只有在保证 A 会取胜时定义取值为 1,所以这意味着如果 A 从 a 走最佳的一着,那么不管 B 怎么应着 A 都会获胜。到结点 b 是最佳着,易于证明,在结点 b,不管 B 如何应着 A 总能取胜。

对于其对策树相当小的一类博弈游戏,例如 n = 6 的拾火柴棍游戏,要将整棵树构造出来并不困难,因此在博弈过程中可以自始至终通过当时所处棋局预先看出可能导致的终局来确定下一着棋的走法。只要对弈双方都不失着,这类博弈的结果预先就确定了,因此没多大意思。还要提及的是,这类博弈游戏的对弈双方都应采用类似的估价函数,即对于弈者 A,X 为得胜棋局,则 $E_A(X)=1$;X 是失败棋局,则 $E_A(X)=-1$。于是,弈者 B 需采用 $E_B(X)=-E_A(X)$。

有较大意义的是像国际象棋一类的博弈,其完整的对策树相当大。有人估计国际象棋的对策树的结点数超过 10^{100} 个。为了全部生成国际象棋的对策树,即使用一台每秒钟能生成 10^{11} 个结点的计算机,也需要 10^{80} 年以上的时间。因此,企图通过看出由某棋局可能导致的终局来确定当前的对策实际上是不可能的。在具有大对策树的博弈游戏中,确定当前的对策时,采用了弈者通常所使用的办法,即预先向前看几着棋。在对策树中就是通过考察这一棋局下面几级(譬如 6 级)的这一部分对策树,用估价函数 E(X)来求取这棵子对策树叶子结点的值,然后可用式(7.1)来得到其余结点的值,从而确定出下一步要走的那着棋。使用产生数级子对策树确定下一步棋着的方法所导致的棋局质量和结局的好坏将取决于这两名弈者所使用的估价函数的功能和通过最大最小过程来确定当前棋局的价值值 V(X)所使用算法的好坏。算法的效能之所以会影响结局的好坏是因为它会限制可能生成的检索树的结点数。

假定弈者 A 是一台计算机,那么就应写一个可以用 A 来计算 V(X)的算法。显然,这个计算 V(X)的算法可用来确定 A 应走的下着棋。如果将最大最小过程的定义改写成如下形式,就可以写出用最大最小过程计算 V(X)的一个相当简单的递归过程。

$$V'(X)=\begin{cases} e(X) & \text{若 X 是所生成子树的叶子结点} \\ \max\limits_{1\le i\le d}\{-V'(c_i)\} & \text{若 X 不是所生成子树的叶子结点} \\ & \text{且 } c_i\ (1\le i\le d) \text{ 是 X 的儿子} \end{cases} \tag{7.2}$$

式中,若 X 是由 A 走子的位置,则 e(X) = E(X);否则,e(X) = - E(X)。

由 A 走子的棋局开始,易于证明式(7.2)的 V′(X)等于式(7.1)的 V(X)。事实上,由 A 走子的那些级上所有结点的值与式(7.1)所给出的值相同,而其它级上结点的值与式(7.1)所给出的值符号相反。

以式(7.2)为基础求取 V′(X)值的递归过程是 VE(X,l)。这个算法通过只生成以 X 为根结点的对策树的 l 级来计算 V′(X)值。因为一个结点的值只有在它的儿子们的值求出后才能确定,所以这个算法按后根次序周游对策树中以 X 为根的 l 级子树。

算法 7.14　对策树的后根次序求值

```
procedure VE(X,l)
    // 通过至多向前看 l 着棋计算 V′(X),弈者 A 使用的估价函数是 e(X)。为方便起见,假定由
       任一不是终局的棋局 X 开始,此棋局的合法棋着只允许将棋局转换成棋局 C1,C2,…,Cd //
    if X 是终局 or l=0 then return(e(X))endif
    ans← - VE(C1,l - 1)    // 周游第一棵子树 //
    for i←2 to d do    // 周游其余的子树 //
        ans←max (ans, - VE(Ci,l - 1))
```

```
        repeat
        return(ans)
    end VE
```

用 $X = P_{11}$ 和 $l = 4$ 来对算法 VE 作初次调用,就会产生出图 7.20 所示的那棵树。各棋局的值按下列次序确定:$P_{3,1}$, $P_{3,2}$, $P_{2,1}$, $P_{5,1}$, $P_{5,2}$, $P_{5,3}$, $P_{4,1}$, $P_{5,4}$, $P_{5,5}$, $P_{5,6}$, $P_{4,2}$, $P_{3,3}$, …, $P_{3,7}$, $P_{2,4}$, $P_{1,1}$。算法 VE 的目的在于求取在棋局 $P_{1,1}$ 情况下对于弈者 A 的 $V'(P_{1,1})$值,以决定 A 下一步要采取的对策。通过对此例的进一步考察将会发现即使不生成图 7.20 中的所有的结点仍可精确计算出$V'(P_{1,1})$。以下讨论所得出的结果实际上是引进一些启发性方法来对 VE 作出改进。

考察图 7.20 所示的对策树,在算出 $V(P_{4,1})$之后,就知道 $V(P_{3,3})$至少是 $V(P_{4,1}) = 3$。接着当确定 $V(P_{5,5})$是 2 时,就知道 $V(P_{4,2})$至多是 2。由于 $P_{3,3}$是一个求最大值的位置,因此 $V(P_{4,2})$不可能影响 $V(P_{3,3})$的取值。无论 $P_{4,2}$的其它儿子是什么值,由于 $V(P_{4,2})$已不可能大于 $V(P_{4,1})$,因此 $P_{3,3}$ 的值不由 $V(P_{4,2})$所确定。这一观察结果可以抽象成一条规则,在叙述这条规则之前,首先定义一个求最大值位置的 α 值,它是该位置迄今最大的可能值。于是这条规则是:如果一个求最小值位置的值被判断为小于或等于它父亲的 α 值,那么,可以停止生成这个求最小值位置其余儿子的值。在这条规则下终止生成结点值的行动称为 α 截断。图 7.20 所示的 $V(P_{4,1})$一旦确定,$P_{3,3}$ 的 α 值就变成 3。$V(P_{5,5}) \leq P_{3,3}$ 的 α 值,这意味着再不用去生成 $P_{5,6}$ 的值。

如果定义一个求最小值位置的 β 值是该位置迄今最小的可能值,于是也可相应给出一条规则:如果一个求最大值位置的值被判断为大于或等于它父亲的 β 值,那么,可以停止生成这个求最大值位置其余儿子的值。在这条规则下终止生成结点值的行动称为 β 截断。图 7.20 所示的 $V(P_{3,5})$一旦确定,$P_{2,3}$ 的 β 值就变成 -1。通过生成 $P_{5,7}$, $P_{5,8}$, $P_{5,9}$ 的值得到 $V(P_{4,3}) = 0$。由于$V(P_{4,3})$大于 $P_{2,3}$ 的 β 值,因此可以终止生成 $P_{3,6}$ 其余儿子的值。将上述两条规则合并在一起得到一条称为 α-β 截断的规则。当把 α-β 截断规则用于图 7.20 时,以 $P_{3,6}$ 为根的子树根本不生成!这是因为在确定 $P_{2,3}$ 的值时,$P_{1,1}$ 的 α 值是 3,$V(P_{3,5}) = -1$,它小于 $P_{1,1}$ 的 α 值,因此发生一次 α 截断。应着重指出的是,一个结点的 α 或 β 值是一个不断变化着的量。在生成对策树期间,任何时候它们都依赖于迄今已生成并算出了其值的那些结点。

为了将 α-β 截断规则引入算法 VE 中去,有必要按式(7.2)所定义的值来重新叙述这一规则。在式(7.2)中,所有的位置都是求最大值的位置,之所以如此是因为式(7.1)中求最小值位置的那些值都被乘了 -1。于是,在定义一个位置的 B 值是该位置迄今最大的可能值的情况下,α-β 截断规则是:对于任一结点 X,设 B 是该结点父亲的 B 值且 D = -B,那么,如果 X 的值判断为大于或等于 D,则可以停止生成 X 的其它儿子。将这一规则直接引入 VE 所导出的算法是过程 VEB。这个过程添加了一个参数 D,它是 X 的父亲的 B 值取负。

算法 7.15　使用 α-β 截断规则对一棵对策树作后根次序求值

```
procedure VEB (X, l, D)
// 使用 α-β 截断规则并只向前看一步,像式(7.2)那样确定 V'(X)。其余的假定与注释与算法 VE
   相同 //
```

```
if X 是终结点 or l = 0 then return (e(X)) endif
ans← - VEB (C1 ,l - 1,∞)   // V′(X)迄今可能的最大值//
for i←2 to d do
  if ans≥D then return (ans) endif   //使用 α-β 截断规则//
  ans←max (ans, - VEB(Ci ,l - 1, - ans))
repeat
return (ans)
end VEB
```

如果 A 从 Y 处走子,那么通过初次调用 VEB(Y,l,∞)就可以预先看 l 着棋来正确地计算出 V′(Y)。

算法 VEB 还可进一步改进,即对这棵对策树作更大的截断。其改进的基本思想是,为了影响结点 X 的值,可以利用 X 的 B 值估计 X 的孙子们的值所应有的下界。考虑图 7.21(a)所示的那棵子树,如果 V′(GC(X))≤B,那么 V′(C(X))≥ - B。在求出 C(X)后,由于 V′(C(X))≥ - B,因此 X 的 B 值是 max{B, - V′(C(X))} = B。所以,除非 V′(GC(X)) > B,否则它就不能影响 V′(X)。故 B 是 GC(X)值应有的下界。把这个下界加入到算法 VEB 就得到算法 AB。所添加的参数 LB 是 X 值应有的下界。

算法 7.16　纵深 α-β 截断算法

```
procedure AB (X,l,LB,D)
    // LB 是 V′(X)的一个下界。其余注释与 VEB 同//
    if X 是终结点 or l = 0 then return (e(X)) endif
    ans←LB   // V′(X)的当前下界//
    for i←1 to d do
      if ans≥D then return (ans) endif
      ans←max(ans, - AB(Ci ,l - 1, - D, - ans))
    repeat
    return (ans)
end AB
```

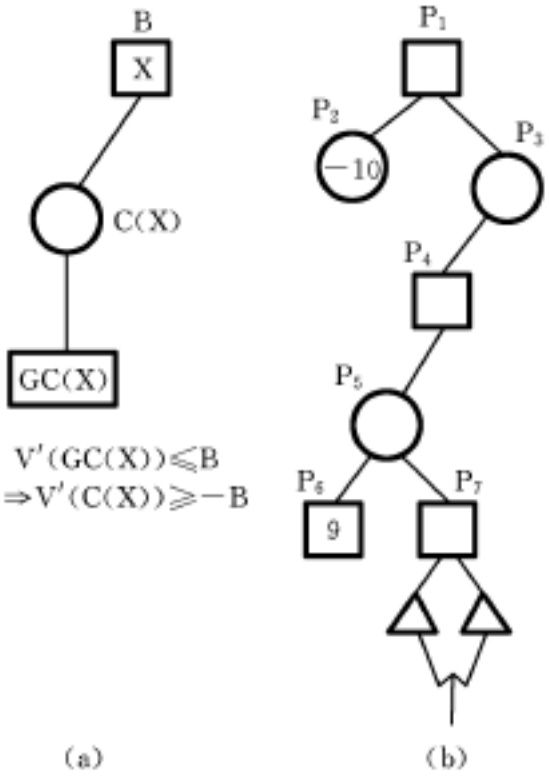


图 7.21　说明下界的对策树

不难证明初次调用 AB(Y,l, - ∞,∞)与调用 VB(Y,l)得到的结果是相同的。

图 7.21(b)显示了一棵假想的对策树,在这棵树中,使用算法 AB 比使用算法 VEB 产生更大的截断。先在图 7.21(b)所示的这棵树上执行 VEB。假定最初的调用为 VEB(P_1,l,∞),其中 l 是这棵树的深度。在检查了 P_1 的左子树之后,P_1 的 B 值被置成 10,并且生成 P_3,P_4,P_5 和 P_6 这些结点。此后,V′(P_6)确定为 9,进而 P_5 的 B 值变成 - 9。使用这一算法继续算出结点 P_7 的值。然而,在使用 AB 的情况下,由于 P_1 的 B 值是 10,P_4 的下界也就是 10,因此 P_4 实际的 B 值变为 10。因为结点 P_7 的值无论是什么都无关紧要,所以结点 P_7 不生成,于是 V′(P_5)≥ - 9 且不可能使 V′(P_4)达到它的下界。

在分析过程 AB 时,确定一棵树中会生成哪一部分结点是极其困难的。而对于 VEB 的分析,至今也只对某些类的对策树作出了证明。有兴趣的读者可以参阅 D.Knuth 于 1975 年发表在《Artificial Intelligence》第 6 期的论文:“An analysis of alpha-beta cutoffs”。

习 题 七

本章习题中的二元树,除非特别声明,其结点都由 3 个信息段表示,即有 LCHILD,DATA 和 RCHILD。

7.1 写一个统计二元树 T 的叶结点数的算法并分析它的计算时间。

7.2 使用 7.1.1 节所讨论的 3 种周游方法之一,写一个求二元树的镜像树的算法 SWAPTREE(T)。图 7.22 给出了一棵二元树 T 和它的镜像树的例子。

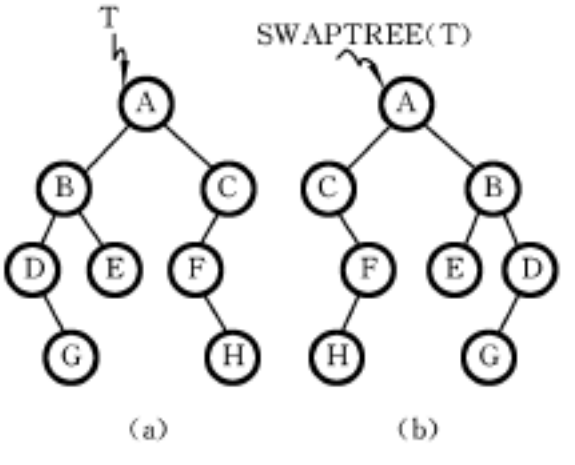


图 7.22 二元树 T 和它的镜像树

7.3 (1) 证明一棵二元树可由它的中根顺序和后根顺序所唯一定义。

(2) 证明一棵二元树可由它的中根顺序和先根顺序所唯一定义。

(3) 证明一棵二元树不能由它的先根顺序和后根顺序所唯一定义。

7.4 已知一棵二元树的中根序列为 I,后根序列为 P,写一个构造该二元树的算法。可直接使用子过程 GETNODE 去获取一个新结点。其算法的计算复杂度是什么?

7.5 给出一个例子,使用例中的数据运行你在习题 7.4 作出的算法。

7.6 如果二元树 T 有 n 个结点,证明定理 7.1 对算法 INORDER1 成立。

7.7 写一个对二元树 T 作先根次序周游的非递归算法(可以使用栈),并分析其时、空复杂度。

7.8 写一个对二元树 T 作后根次序周游的非递归算法(可以使用栈),并分析其时、空复杂度。

7.9 如果 n 结点二元树 T 的每个结点有 4 个信息段:LCHILD,DATA,PARENT,RCHILD。要求以下写的算法所用附加空间都不超过 O(1),时间都不超过 O(n)。并证明其确实达到了这些要求。

(1) 写对 T 的中根次序周游算法。

(2) 写对 T 的先根次序周游算法。

(3) 写对 T 的后根次序周游算法。

7.10 写一个时间为 $\Theta(n)$,附加空间为 $\Theta(1)$的二元树中根次序周游算法。树中的每个结点除了信息段:LCHILD,DATA 和 RCHILD 外还有一个位信息段 TAG。(提示:使用 INORDER2 链倒挂的思想,但不用 LR 的方法。用 TAG 位区别向左还是向右子树移动)

7.11 证明按树先根次序周游一棵树与按先根次序周游此树对应的二元树所给出的结果相同(即按相同的次序访问这些结点)。

7.12 证明按树中根次序周游一棵树与按中根次序周游此树对应的二元树所给出的结果相同(即按相同的次序访问这些结点)。

7.13 证明如果按树后根次序周游一棵树,那么访问这树中结点的次序与按后根次序周游对应二元树的访问这些结点的次序可能不同。

7.14 设树 T 的度为 k,而且结点 P 有 k 个儿子信息段 CHILD(P,i),$1 \le i \le k$。写出下列算法并分析它们的时、空复杂度。

(1) 写一个树中根次序周游的非递归算法 TI(T,k)。

(2) 写一个树先根次序周游的非递归算法 TPRE(T,k)。

(3) 写一个树后根次序周游的非递归算法 TPOST(T,k)。

7.15 证明对于任一无向图 G=(V,E),$v \in V$。对 BFS(v)的一次调用就会访问含结点 v 的连通分图的全部结点。

7.16　重写 BFS 和 BFT,使它能打印出无向图 G 的所有连通分图。假定 G 是按邻接表方式输入的,每个结点 i 的邻接表有头结点 HEAD(i)。

7.17　利用 BFS 的思想写一个找包含已知结点 v 的最短(有向)环算法。证明你的算法能找出最短环,分析算法的时、空复杂度。

7.18　证明 DFS 访问 G 中由 v 可到达的所有结点。

7.19　证明定理 7.3 中给出的时、空限界对 DFS 也成立。

7.20　对图的另一种检索方法是 D-search。此方法与 BFS 的不同之处在于,下一个要检测的结点是最新加到未检测结点表的那个结点。因此这个表应作成一个栈而不是一个队。

(1) 写一个 D-search 算法。

(2) 证明由结点 v 开始的 D-search 访问 v 可到达的所有结点。

(3) 你的算法的时、空复杂度是什么?

(4) 修改你的算法,使它能对无向连通图产生一棵生成树。

7.21　判断 n 结点的无向图 G 是否有环?若有,就尽可能多地写出对它的算法,分析这些算法的时、空复杂度;从而对这些算法的有效性作出评价。

7.22　写一个计算以二元树 T 表示的算术表达式的算法。假定表达式只使用双目运算符 +、-、*、/。此二元树中的每个结点有 3 个信息段:LCHILD,DATA 和 RCHILD。如果 P 是叶结点,则 DATA(P)是 P 所代表的变量或常数在存储器中的地址。VAL(DATA(P))是此变量或常数的当前值。你的算法需要多少计算时间?

7.23　证明定理 7.5。

7.24　在表达式包含某些可交换运算符的情况下完善表 7.6。

7.25　修改算法 CODE1,使其在表达式树 T 含有可交换运算符的情况下也能生成最优代码。证明你的算法确能达到这一要求。

7.26　在 T 含有可结合运算符情况下,做与题 7.25 相同的工作。

7.27　获取下述表达式的表达式树,并标出每个结点的 MR 值。然后在 N = 1 和 N = 2 的情况下,由 CODE2 生成它们的最优代码。假定所有的运算符都是不可交换和不可结合的。

(1) (a + b) * (c + d * (e + f)/ (g + h))

(2) a * b * d/ (e - f + g * (h - k) * (l + m))

(3) a * (b - c) * (d + f)/ (g * (h + j) - k * l)

7.28　参看定理 7.6 对 MR(P)的定义,写一个对二元表达式树 T 的每个结点计算其 MR(P)值的算法。假设每个结点 P 有 4 个信息段 LCHILD、DATA、MR 和 RCHILD。

7.29　证明定理 7.7。

7.30　证明定理 7.8。

7.31　证明 CODE2 的时间复杂度是 $\Theta(n)$,其中 n 是 T 的结点数。

7.32　如果 MR(T)≤n,在不允许使用装入指令的情况下,证明 CODE2 用最少的寄存器生成代码。

7.33　证明引理 7.1。

7.34　写一个算法 FLIP(T),用它来交换表达式树 T 中那些表示可交换运算符结点的左、右子树,使生成的树对每个给定的寄存器数 N,大、小结点数之和最小。FLIP 的计算复杂度是多少?

7.35　将 CODE2 扩展到具有可结合运算符表达式的计算。

7.36　识别图 7.23 的关节点并画出它们的双连通分图。

7.37　如果 G 是一个无向连通图,证明 G 中任何一条边都不可能在两个不同的双连通分图中。

7.38　假设无向连通图 G 的双连通分图是 $G_i = (V_i, E_i)$, $1 \le i \le k$。证明:

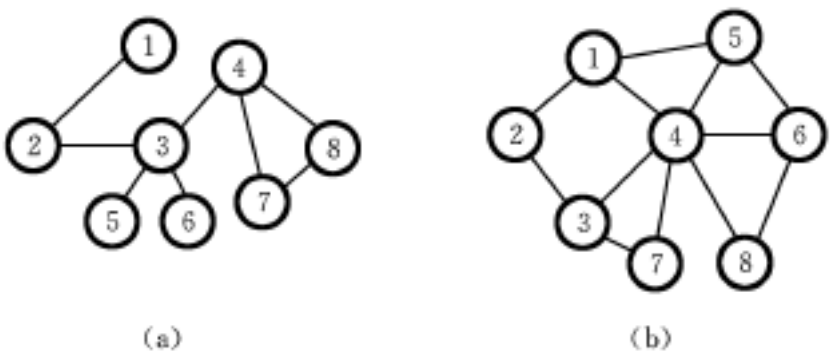


图 7.23 两个连通图

(1) 如果 $i \neq j$,那么 $V_i \cap V_j$ 至多包含一个结点。

(2) 结点 v 是 G 的关节点,当且仅当对于某 $i \neq j$, $\{v\} = V_i \cap V_j$。

7.39 设 G 是一个无向连通图,写一个算法以求出将 G 变成双连通图所需要增加的最小边数并要求输出这些边。分析你的算法需要的时间和空间。

7.40 证明如果 T 是无向连通图 G 的宽度优先生成树,那么,相对于 T,G 可能有交叉边。

7.41 假设 u 不是根,证明 u 是一个关节点,当且仅当对于 u 的某个儿子 w,$L(w) \geq DFN(u)$。

7.42 证明在算法 ART 加了 2.1 和 3.1～3.7 行后,如果 v = w 或者 DFN(w) > DFN(u),那么边(u,w)或者已在栈 S 的顶部,或者已作为双连通图的一部分输出。

7.43 修改算法 SOLVE,使它能识别 T 的一棵解子树。

7.44 写出算法 BFGEN 中使用的子算法 ASOLVE。

7.45 写一个算法 PRUNE,用它去消除由算法 BFGEN 生成的解树 T 中所有不需要求解的结点,即,使输出的树是一棵为了求解整个问题必须求解它的每个结点的解子树。

7.46 考虑下面这棵假想对策树(见图 7.24):

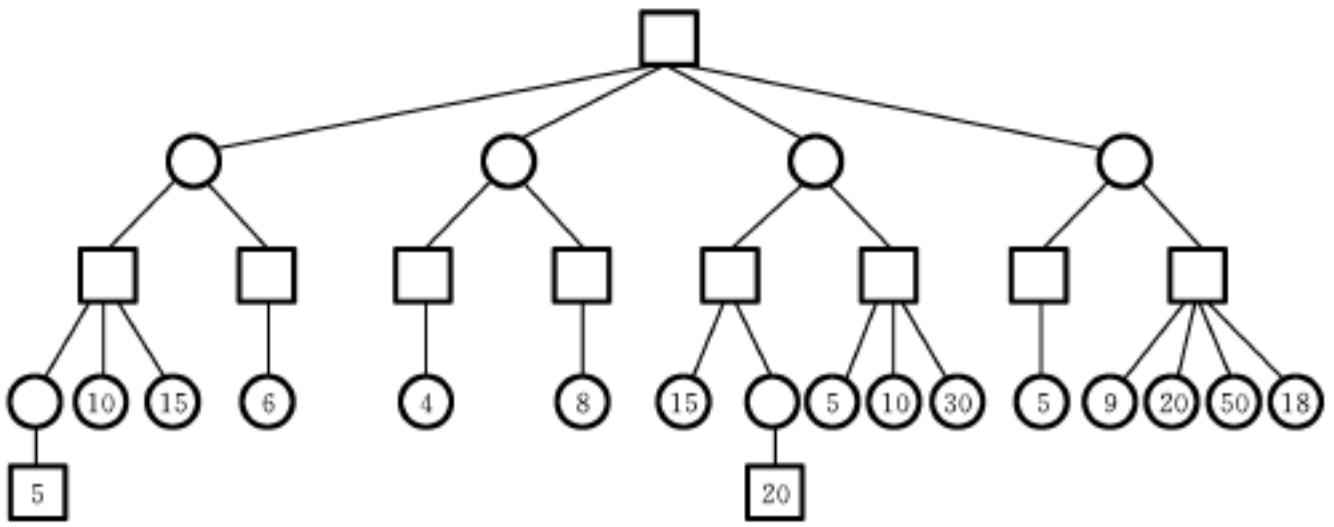


图 7.24 假想对策树

(1) 使用最大最小方法式(7.1)去获取根结点的值。

(2) 弈者 A 应采用什么棋着?

(3) 列出用算法 VE 计算这棵对策树结点的值时结点的计算顺序。

(4) 对树中的每个结点 X,用式(7.2)计算 V′(X)。

(5) 在取 X = 根,l = ∞, LB = - ∞,D = ∞的情况下,用算法 AB 计算此树的根的值期间,这树的哪些结点没有计算?

7.47 证明对于那些由 A 走子的级上的每个结点用式(7.2)计算的 V′(X)与用式(7.1)计算的 V(X)有相同的值,而对于其它级上的结点,用式(7.1)计算的 V(X)为用式(7.2)计算的 V′(X)取负。

7.48 证明初次调用 AB(X,l, - ∞,∞)与初次调用 VB(Y,l)所得的结果相同。

第 8 章

回　溯　法

8.1　一般方法

在算法设计的基本方法中,回溯法是最一般的方法之一。在那些涉及寻找一组解的问题或者求满足某些约束条件的最优解的问题中,有许多可以用回溯法来求解。

8.1.1　回溯的一般方法

为了应用回溯法,所要求的解必须能表示成一个 n-元组$(x_1,\cdots,x_n)$,其中 x_i 是取自某个有穷集 S_i。通常,所求解的问题需要求取一个使某一规范函数 $P(x_1,\cdots,x_n)$ 取极大值(或取极小值或满足该规范函数条件)的向量。有时还要找出满足规范函数 P 的所有向量。例如,将A(1：n)中的整数分类就是可用一个 n-元组表示其解的问题,其中 x_i 是 A 中第 i 小元素的下标。规范函数 P 是不等式 $A(x_i)\leq A(x_{i+1})$,其中 $1\leq i<n$。这里,S_i 是一个包含整数 1～n 的有穷集合。虽然分类问题通常是不必用回溯法求解的问题,但它是可用 n-元组列出其解的常见问题的一个例子。在这一章中,将研究一批一般认为最好是用回溯法来求解的问题。

假定集合 S_i 的大小是 m_i,于是就有 $m=m_1m_2\cdots m_n$ 个 n-元组可能满足函数 P。所谓硬性处理是构造出这 m 个 n-元组并逐一测试它们是否满足 P,从而找出该问题的所有最优解。而回溯法的基本思想是,不断地用修改过的规范函数(有时称为限界函数)$P_i(x_1,\cdots,x_i)$去测试正在构造中的 n-元组的部分向量$(x_1,\cdots,x_i)$,看其是否可能导致最优解。如果判定$(x_1,\cdots,x_i)$不可能导致最优解,那么就将可能要测试的 $m_{i+1}\cdots m_n$ 个向量一概略去。因此,回溯算法的测试次数比硬性处理的测试次数(m 次)要少得多。

用回溯法求解的许多问题都要求所有的解满足一组综合的约束条件。这些约束条件可以分成两种类型:显式约束和隐式约束。显示约束条件是限定每个 x 只从一个给定的集合上取值。显式约束条件常见的例子是:

$x_i\geq 0$　　即　　$S_i=\{$所有非负实数$\}$

$x_i=0$ 或 $x_i=1$　　即　　$S_i=\{0,1\}$

$l_i\leq x_i\leq u_i$　　即　　$S_i=\{a:l_i\leq a\leq u_i\}$

这些显式约束条件可以与所求解的问题的实例 I 有关,也可以无关。满足显式约束的所有元组确定 I 的一个可能的解空间。隐式约束条件则规定 I 的解空间中那些实际上满足规范函数的元组。因此,隐式约束描述了 x_i 必须彼此相关的情况。

例 8.1　[8-皇后问题]一个经典的组合问题是在一个 8×8 棋盘上放置 8 个皇后,且使得每两个之间都不能互相“攻击”,也就是使得每两个都不在同一行、同一列及同一条斜角线

上。给棋盘的行和列都编上 1 到 8 的号码(见图 8 .1)。这些皇后也可给以 1 到 8 的编号。由于一个皇后应在不同的行上,为不失一般性,故可以假定皇后 i 将放在行 i 上。因此,8-皇后问题可以表示成 8-元组(x_1 , x_2 ,…, x_8),其中 x_i 是放置皇后 i 所在的列号。使用这种表示的显式约束条件是 S_i = {1,2,3,4,5,6,7,8},1≤i≤8。

于是,解空间由 8^8 个 8-元组所组成。这个问题的隐式约束条件是,没有两个 x_i 可以相同(即,所有皇后都必须在不同的列上)而且没有两个皇后可以在同一条斜角线上。这两个约束条件的前者意味着所有的解都是 8-元组(1,2,3,4,5,6,7,8)的置换。于是解空间的大小由 8^8 个元组减少到 8 ! 个元组。至于如何根据 x_i 来列出第二个约束条件将在 8 .2 节介绍。把图 8 .1 中的解表示为一个 8-元组就是(4,6,8,2,7,1,3,5)。

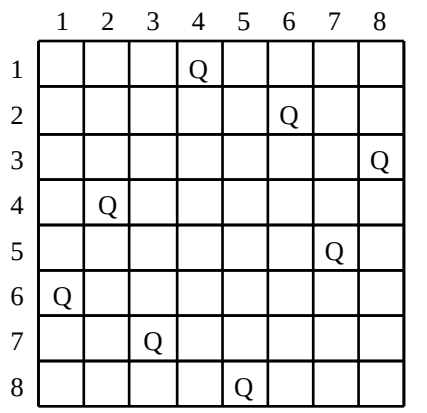


图 8 .1 8-皇后问题的一个解

例 8 .2 [子集和数问题]已知 n + 1 个正数: w_i ,1≤i≤n,和 M。要求找出 w_i 的和数是 M 的所有子集。例如,若 n = 4,(w_1 , w_2 , w_3 , w_4) = (11,13,24,7),M = 31,则满足要求的子集是(11,13,7)和(24,7)。值得指出的是,通过给出其和数为 M 的那些 w_i 的下标来表示解向量比直接用这些 w_i 表示解向量更为方便。因此,这两个解就由向量(1,2,4)和(3,4)所描述。在一般情况下,所有的解都是 k-元组(x_1 , x_2 ,…, x_k),1≤k≤n,并且不同的解可以是大小不同的元组。显式约束条件要求 x_i ∈{j | j 是一个整数且 1≤j≤n}。隐式约束条件则要求没有两个 x_i 是相同的且相应的 w_i 的和数是 M。为了避免产生同一个子集的重复情况(例如,(1,2,4)和(1,4,2)表示同一个子集),附加另一个隐式约束条件: $x_i < x_{i+1}$,1≤i < n。

子集和数问题的另一种列式表示是,每一个解的子集由这样一个 n-元组(x_1 , x_2 ,…, x_n)所表示,它使得 x_i ∈{0,1},1≤i≤n。如果没有选择 w_i ,则 x_i = 0;如果选择了 w_i ,则 x_i = 1。于是,上例的解可表示为(1,1,0,1)和(0,0,1,1)。这种列式表示使用大小固定的元组表示所有的解。因此可得出如下结论,一个问题的解可以有数种表示形式,而这些表示形式都使得所有的解是满足某些约束条件的多元组。用上面的两种表示,可以证明解空间由 2^n 个不同的元组所组成。

回溯算法通过系统地检索给定问题的解空间来确定问题的解。这检索可以用这个解空间的树结构来简化。对于一个给定的解空间,可能有多种树结构。下面的两个例子给出了组织解空间成为一棵树的一些方法。

例 8 .3 [n-皇后问题]n-皇后问题是例 8 .1 的 8-皇后问题的推广。n 个皇后将被放置在一个 n×n 的棋盘上且使得没有两个皇后可以互相攻击。推广前面所作的讨论,解空间由 n-元组(1,2,…,n)的 n ! 种排列所组成。图 8 .2 显示了当 n = 4 时一种可能的树结构。像这样的一棵树称为排列树。树的边由 x_i 的可能值所标记。由 1 级到 2 级结点的边给出 x_1 的值。因此,最左子树包含着 x_1 = 1 的所有解;这棵子树的最左子树则包含着 x_1 = 1 且 x_2 = 2 的所有解,等等。由 i 级到 i + 1 级的边用 x_i 的值标记。解空间则由从根结点到叶结点的所有路径所定义。在图 8 .2 的这棵树中有 4 ! = 24 个叶结点。

例 8 .4 [子集和数问题]在例 8 .2 中,对子集和数问题给出了解空间的两种可能的列式

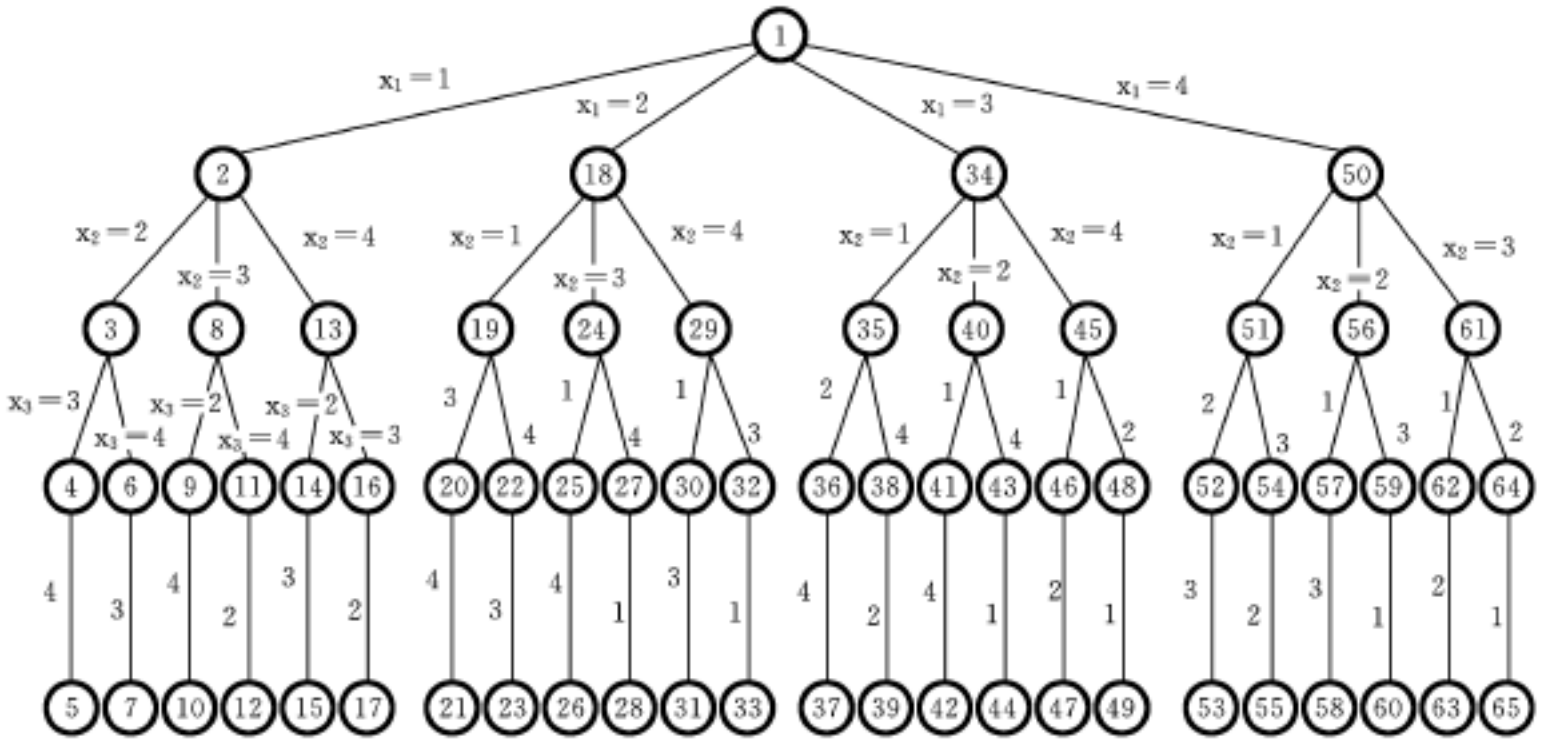


图 8 .2　4-皇后问题解空间的树结构,结点按深度优先检索编号

表示。图 8 .3 和图 8 .4 显示了在 n = 4 的情况下这两种表示的树结构形式,图 8 .3 所示的那棵树对应于元组大小可变的列式表示,树边的标记方法是,由 i 级结点到 i + 1 级结点的一条边用 x_i 来表示,在每一个结点处,解空间被分成一些子解空间,解空间则由树中的根结点到任何结点的所有路径所确定,这些可能的路径是( )(这对应于由根结点到它自身的那条空路径);(1);(1,2);(1,2,3);(1,2,3,4);(1,2,4);(1,3,4);(1,4);(2);(2,3);等等。于是,最左子树确定了包含 w_1 的所有子集,下一棵子树则确定了包含 w_2 但不包含 w_1 的所有子集,等等。

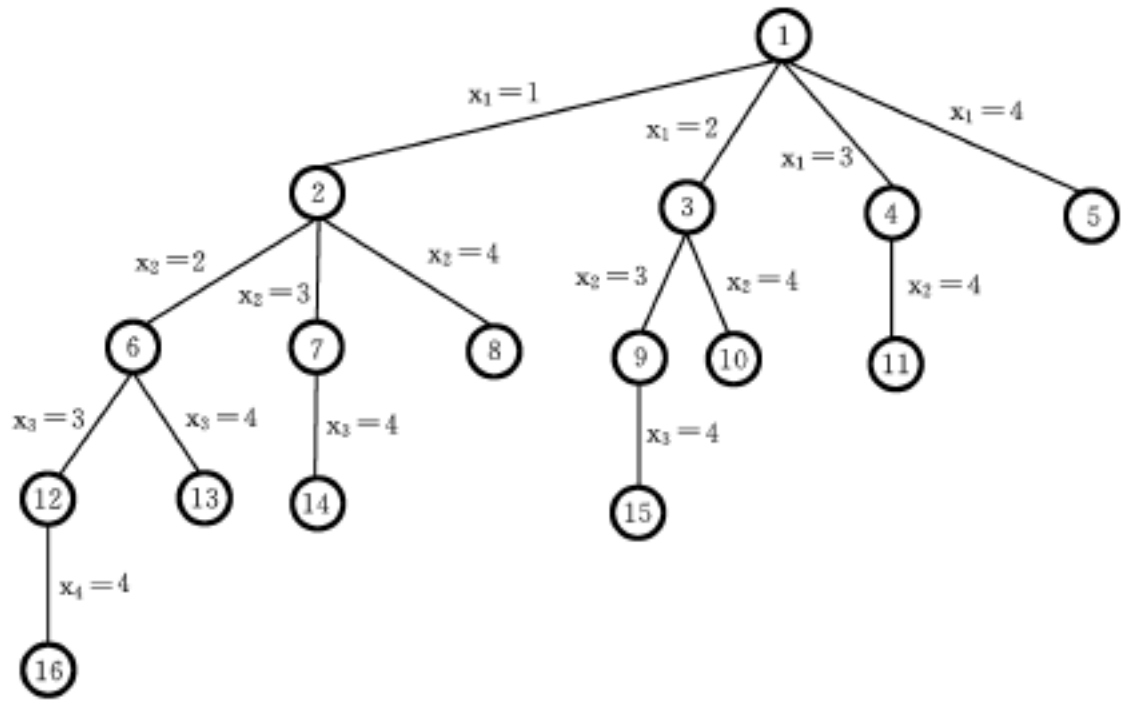


图 8 .3　子集和数问题的一种解空间结构,结点按宽度优先检索方式编号

图 8 .4 所示的树对应于元组大小固定的列式表示。由 i 级结点到 i + 1 级结点的那些边用 x_i 的值来标记,x_i 的值或者为 1 或者为 0。由根到叶结点的所有路径确定了这个解空间,

根的左子树确定包含 w_1 的所有子集,而根的右子树则确定不包含 w_1 的所有子集等等。于是有 2^4 个叶结点,表示 16 个可能的元组。

为便于讨论,引进一些关于解空间树结构的术语。树中的每一个结点确定所求解问题的一个问题状态(problem state)。由根结点到其它结点的所有路径则确定了这个问题的状态空间(state space)。解状态(solution states)是这样一些问题状态 S,对于这些问题状态,由根到 S 的那条路径确定了这解空间中的一个元组。在图 8 .3 所示的树中,所有的结点都是解状态,而在图 8 .4 所示的树中,只有叶结点才是解状态。答案状态(answer states)是这样的一些解状态 S,对于这些解状态而言,由根到 S 的这条路径确定了这问题的一个解(即,它满足隐式约束条件)。解空间的树结构称为状态空间树(state space tree)。

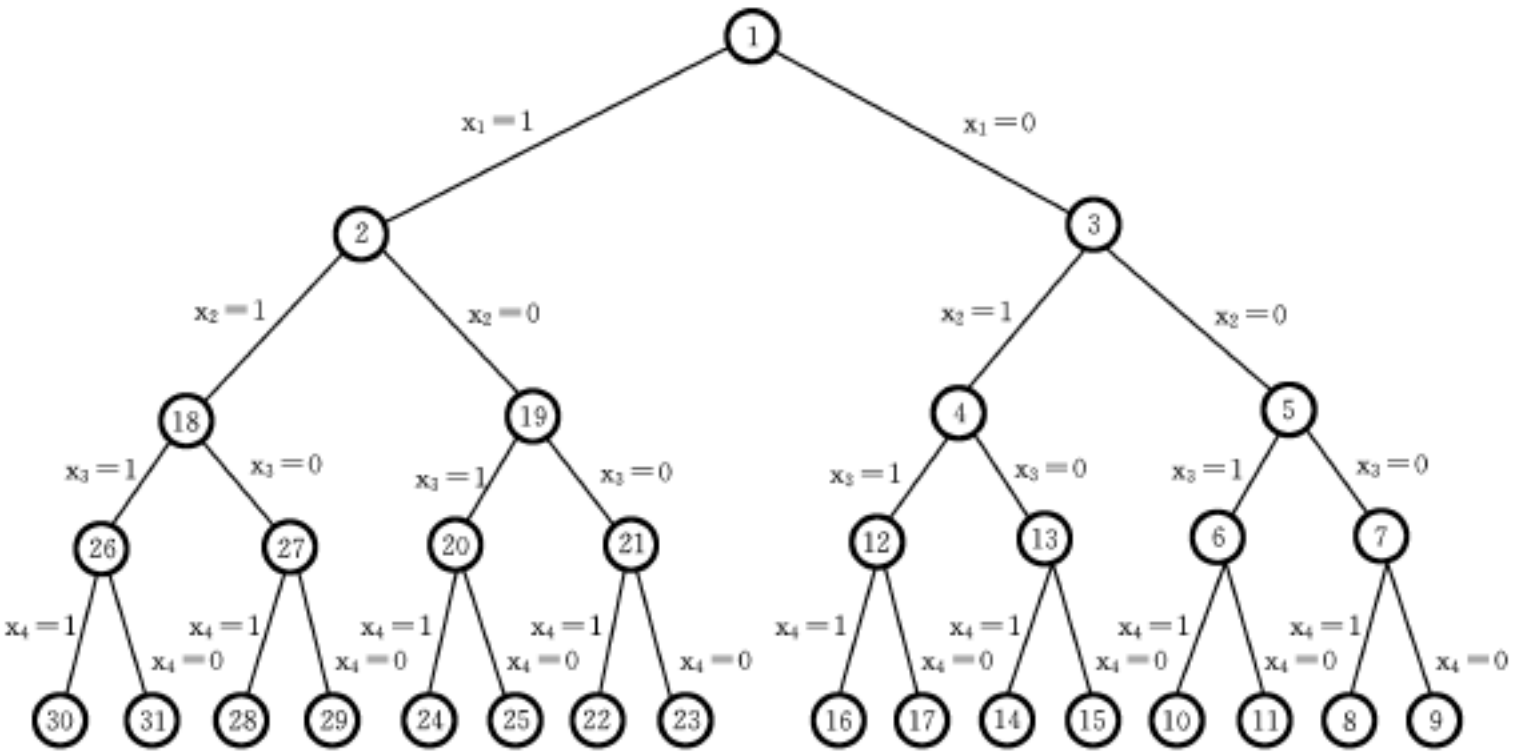


图 8 .4　子集和数问题的另一种结构,结点按 D-检索方式编号

在例 8 .3 和例 8 .4 的状态空间树中每一个内部结点处,这解空间被分成不相交的子解空间。例如,在图 8 .2 所示的结点 1 处,解空间被分成 4 个不相交的集合,子树 2,18,34 和 50 分别代表在 $x_1 = 1,2,3$ 和 4 的情况下解空间的所有元素。在结点 2 处,在 $x_1 = 1$ 的情况下的子解空间进一步被分成 3 个不相交的集合。子树 3 代表在 $x_1 = 1$ 且 $x_2 = 2$ 的情况下解空间的所有元素。

本章所研究的都是一些将每个内部结点处分成若干不相交子解空间的状态空间树。但要指出的是,这并不是构造状态空间树的必要条件,构造状态空间树只要求解空间的每个元素至少由状态空间树的一个结点所表示。

例 8 .4 中所描述的这种状态空间树结构称为静态树(static trees)。这种树结构与所要解决的问题的实例无关,对于某些问题,根据不同实例而使用不同的树结构则更为有利。在这种情况下,由于要检索解空间,因此树结构是动态确定的,与实例相关的树结构称为动态树(dynamic trees)。作为一个例子,对子集和数问题(例 8 .4),我们来研究其元组大小固定的列式表示方法。当使用动态树结构时,n = 4 的一种实例可以用图 8 .4 中所给出的树结构

来分解,而 n = 4 的另一种实例则可以用下述的一棵树来求解,在这棵树中,在第一级,其划分相当于 $x_2 = 1$ 和 $x_2 = 0$,在第 2 级,其划分可相当于 $x_1 = 1$ 和 $x_1 = 0$,而在第 3 级则相当于 $x_3 = 1$ 和 $x_3 = 0$等等。

对于任何一个问题,一旦设想出一种状态空间树,那么就可以先系统地生成问题状态,接着确定这些问题状态中的哪些状态是解状态,最后确定哪些解状态是答案状态,从而将这问题解出。为了生成问题状态,采用两种根本不同的方法。虽然,这两种方法都是从根结点开始然后生成其它结点。如果已生成一个结点而它的所有儿子结点还没有全部生成,则这个结点叫做活结点。当前正在生成其儿子结点的活结点叫 E-结点(正在扩展的结点)。不再进一步扩展或者其儿子结点已全部生成的结点就是死结点。在生成问题状态的两种方法中,都要有一张活结点表。在第一种方法中,当前的 E-结点 R 一旦生成一个新的儿子 C,这个儿子结点就变成一个新的 E-结点,当完全检测了子树 C 之后,R 结点就再次成为 E-结点。这相当于问题状态的深度优先生成。在第二种状态生成方法中,一个 E-结点一直保持到变成死结点为止。在这两种方法中,将用限界函数去杀死还没有全部生成其儿子结点的那些活结点。这样做要非常小心,以使得在处理结束时至少能生成一个答案结点;如果这个问题要求找出全部解,则要能生成所有的答案结点。使用限界函数的深度优先结点生成方法称为回溯法(backtracking)。E-结点一直保持到死为止的状态生成方法导致分枝-限界方法(branch-and-bound),分枝-限界法在第 7 章讨论。

图 8.2 中的结点就是根据深度优先生成的次序来标记的。图 8.3 和图 8.4 中的结点则是按照其 E-结点一直保持到死的两种生成方法来标记的。在图 8.3 中,每个新结点都被放入一个队列。当这个当前 E-结点已生成了它的所有儿子时,在队列前面的下一个结点就变成新的 E-结点。在图 8.4 中,那些新结点被放入栈而不是放入队列。当涉及这两种方案之一时,当前的术语就不能保持一致了。队列方法一般叫做宽度优先生成,而栈方法则称为 D-检索(深度检索)。

例 8.5　[4-皇后问题]考虑用回溯法来处理例 8.3 的 n-皇后问题。可用一些显而易见的标准来作为限界函数,即如果$(x_1, x_2, \cdots, x_i)$是到当前 E-结点的路径,那么具有父-子标记 x_{i+1}的所有儿子结点是一些这样的结点,它们使得$(x_1, \cdots, x_{i+1})$表示没有两个皇后正在互相攻击的一种棋盘格局。开始,把根结点作为唯一的活结点,这个根结点就成为 E-结点而且路径为(　)。接着生成一个儿子结点,如果假定按自然数递增的次序来生成儿子,那么,图 8.2 所示的结点 2 被生成,这条路径为(1)。这相当于把皇后 1 放在第 1 列上。结点 2 变成 E-结点,生成结点 3,但立即被杀死。下一个生成的结点是 8 且路径变成(1,3)。结点 8 成为 E-结点,由于它的所有儿子表示的是一些不可能导致答案结点的棋盘格局,因此结点 8 也被杀死。于是回溯到结点 2 并生成它的另一个儿子结点 13。现在这条路径是(1,4)。图 8.5 生动地显示了应用回溯算法求一个解所经过的步骤。图中的点表示曾试图放置一个皇后但由于受到另一皇后的攻击而放弃了的位置。在(b)中,第二个皇后被放在第 1,第 2 列而最后放置在第 3 列上。在(c)中,算法把四列都试验了,但没法将下一个皇后放在一个方格中,此时就进行回溯。在(d)中,第二个皇后被移到下一个可能的列,即第 4 列,然后将第三个皇后放在第 2 列上。图 8.5 的(e)、(f)、(g)和(h)则显示了用这个算法一直找到一个解所进行的其

余步骤。

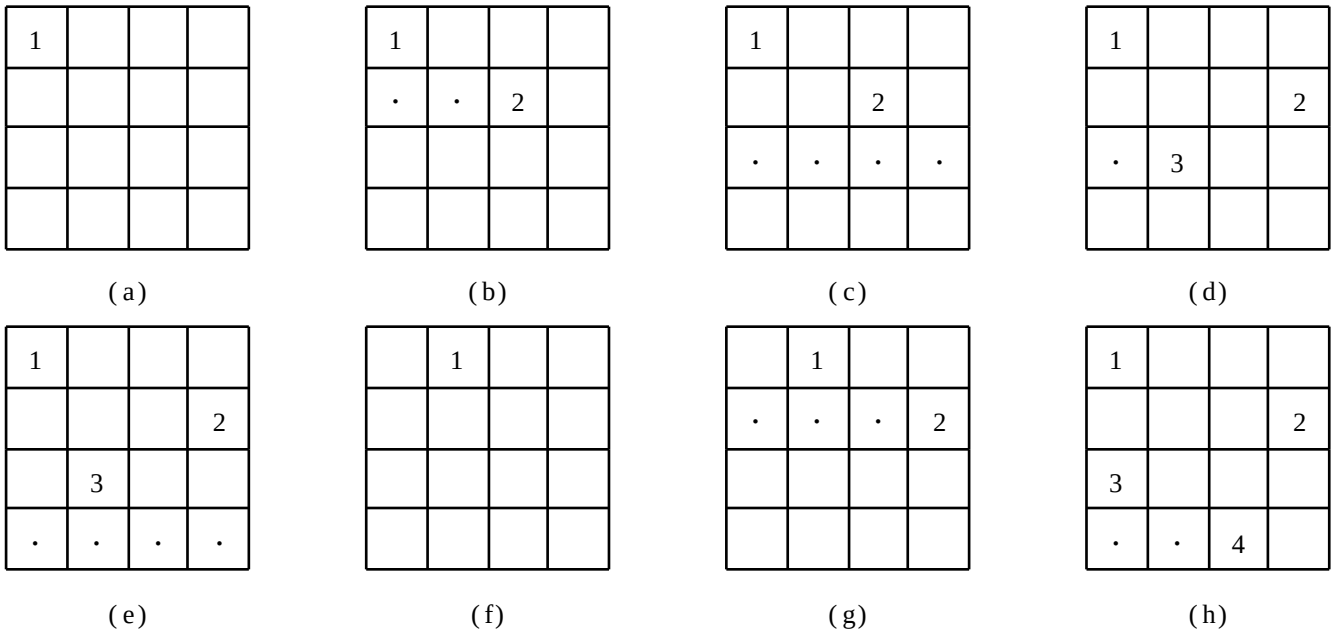


图 8.5 4-皇后问题的一个回溯解的例子

图 8.6 显示了图 8.2 所示的树的实际生成部分。结点按照它们生成的次序被标号。由限界函数所杀死的结点则在其下方写上 B。请将这棵树与包含 31 个结点的树(图 8.2)相比较。

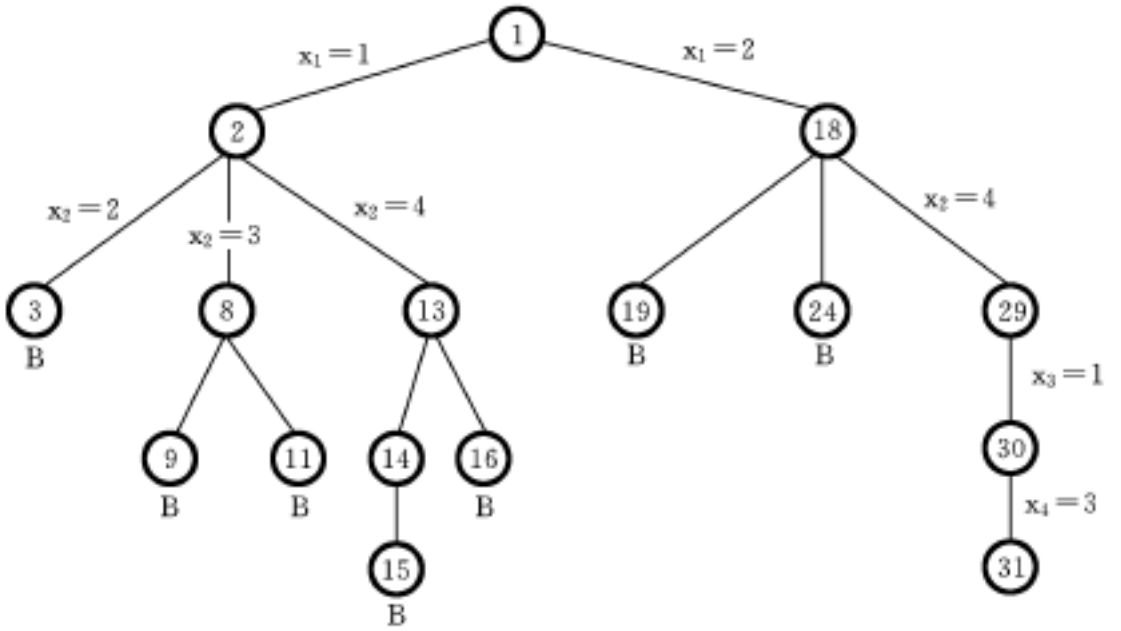


图 8.6 在回溯期间所生成的图 8.2 所示树的一部分

下面讨论回溯算法的形式描述,假定回溯法要找出所有的答案结点而不是仅仅只找出一个。设$(x_1, x_2, \cdots, x_{i-1})$是状态空间树中由根到一个结点(即问题状态)的路径,而 $T(x_1, x_2, \cdots, x_{i-1})$是下述所有结点 x_i 的集合,它使得对于每一个 x_i,$(x_1, x_2, \cdots, x_i)$是一条由根到结点 x_i 的路径;还假定存在着一些限界函数 B_i(可以表示成一些谓词),如果路径$(x_1, x_2, \cdots, x_i)$不可能延伸到一个答案结点,则 $B_i(x_1, x_2, \cdots, x_i)$取假值,否则取真值。于是,解向量 X(1:n)中第 i 个分量就是那些选自集合 $T(x_1, x_2, \cdots, x_{i-1})$且使 B_i 为真的 x_i。算法 BACKTRACK 是使用 T 和 B_i 的回溯方法的控制抽象化描述。

算法 8.1 回溯的一般方法

```
procedure BACKTRACK (n)
    // 这是对回溯法控制流程的抽象描述。每个解都在 X(1:n)中生成,一个解一经确定就立即
       印出。在 X(1),…,X(k-1)已被选定的情况下,T(X(1),…,X(k-1))给出 X(k)的所有可
```

```
    能的取值。限界函数 B(X(1),…,X(k))判断哪些元素 X(k)满足隐式约束条件 //
  integer k,n; local X(1:n)
  k←1
  while k > 0 do
    if 还剩有没检验过的 X(k)使得
      X(k)∈T(X(1),…,X(k-1)) and B (X(1),…,X(k)) = true
    then if (X(1),…,X(k)) 是一条已抵达一答案结点的路径
        then print (X(1),…,X(k)) endif
            k←k+1    // 考虑下一个集合 //
        else k←k-1    // 回溯到先前的集合 //
        endif
    repeat
end BACKTRACK
```

需要注意的是,集合 T(　)将提供作为解向量的第一个分量 X(1)的所有可能值,解向量则取使限界函数 B_1 (X(1))为真的那些 X(1)的值。还要注意这些元素是如何按深度优先方式生成的。随着 k 值的增加,解向量的各分量不断生成,直到找到一个解或者不再剩有没经检验的 X(k)为止。当 k 值减少时,算法必须重新开始生成那些可能在以前剩下而没经检验的元素。因此,还需拟定一个子算法,使它按某种次序来生成这些元素 X(k)。如果只想要一个解,则可在 print 后面设置一条 return 语句。

算法 8.2 提供了回溯算法的一种递归表示。由于它基本上是一棵树的后根次序周游,因此按照这种方法描述回溯法是自然的。这个递归模型最初由

```
call RBACKTRACK(1)
```

所调用。

算法 8.2　递归回溯算法

```
procedure RBACKTRACK(k)
    // 此算法是对回溯法抽象地递归描述。进入算法时,解向量 X(1:n)的前 k-1 个分量
      X(1),…,X(k-1)已赋值 //
    global n, X(1:n)
    for 满足下式的每个 X(k)
        X(k)∈T(X(1),…,X(k-1)) and B (X(1),…,X(k)) = true do
     if (X (1),…,X(k))是一条已抵达一答案结点的路径
         then print (X(1),…,X(k)) endif
     call RBACKTRACK(k+1)
    repeat
end RBACKTRACK
```

将解向量(x_1,…,x_n)作为一个全程数组 X(1:n)来处理。这个元组第 k 个位置上满足 B 的所有元素逐个被生成,并被连接到当前的向量(X(1),…,X(k-1)),每次 X(k)都要附之以这样的一种检查,即判断一个解是否已被找到。因此,这个算法被递归调用。当退出 for 循环时,不再剩有 X(k)的值,从而结束此层递归并继续上次没解决的调用。

要指出的是,当 k 大于 n 时,T(X(1),…,X(k-1))返回一个空集,因此根本不进入 for

循环。还要指出的是,这个程序印出所有的解,而且组成解的元组的大小是可变的。如果只想要一个解,则可加上一个标志作为一个参数,以指明首次成功的情况。

8.1.2 效率估计

上面所给出的两个回溯程序的效率主要取决于以下4种因素:①生成下一个X(k)的时间;②满足显式约束条件的X(k)的数目;③限界函数B_i的计算时间;④对于所有的i,满足B_i的X(k)的数目。如果这些限界函数大量地减少了所生成的结点数,则认为它们是好的。不过,一些好的限界函数往往需要较多的计算时间,而所希望的不只是减少所生成的结点而是要减少总的计算时间。因此,在选择限界函数时,通常在好坏与时间消费上采取折中的方案。有许多问题,其状态空间树的规模太大,要想生成其全部结点实际上是不允许的,因此应该使用限界函数并且希望在一段适当的时间内至少会找出一个解。不过对于许多问题(例如,n-皇后问题)至今还没听说有完善的限界方法。

对于许多问题,可以按任意次序使用包含各个解分量x_i可能取值的那些有穷集S_i。为了提高有效检索的效率,一般可采用一种称之为重新排列的方法。其基本思想是,在其它因素相同的情况下,从具有最少元素的集合中作下一次选择。这种策略虽已证明对n-皇后问题无效,而且还可构造出一些证明用此方法无效的例子,但从信息论的观点看,从最小集合中作下一次选择,平均来说更为有效。在图8.7中,对同一个问题用两棵回溯检索树显示了这一方法的潜在能力。如果能去掉图8.7(a)中那棵树的第一级的一个结点,那么实际上就从所考虑的情况中去掉了12个元组。如果从图8.7(b)中那棵树第一级上去掉一个结点,则只删去8个元组。更完善的重新排列策略将在后面和动态状态空间树一起研究。

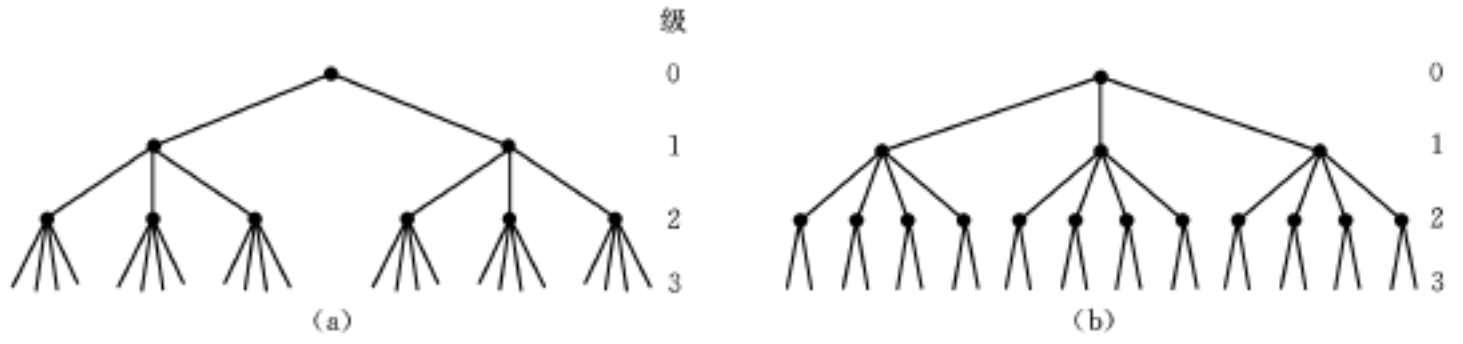


图8.7 重新排列

如上所述,有4种因素决定回溯算法所需要的时间。一旦选定了一种状态空间树结构,这4种因素的前3种因素相对来说就与所要解决问题的实例无关,只有第4种因素,即所生成的结点数因问题的实例不同而异。对于某一实例,回溯算法可能只生成O(n)个结点,而对于另一不同的实例,由于它与原实例密切相关,故也可能生成这棵状态空间树的几乎全部结点。如果解空间的结点数是2^n或n!,则回溯算法的最坏情况时间一般将分别是$O(p(n)2^n)$或O(q(n)n!)。p(n)和q(n)都是n的多项式。尽管回溯法对同一问题的不同实例在计算时间上可能出现极大差异,但当n很大时,对于某些实例而言,回溯算法确实可在很短时间内求出其解,因此回溯法仍不失为一种有效的算法设计策略,只是在决定采用回溯算法正式计算某实例之前,应预先估算出回溯算法在此实例情况下的工作效能。

用回溯算法去处理一棵树所要生成的结点数,可以用蒙特卡罗(Monte Carlo)方法估算出来。这种估计方法的一般思想是,在状态空间树中生成一条随机路径。设 X 是这条随机路径上的一个结点,而且 X 在状态空间树的第 i 级。在结点 X 处用限界函数确定没受限界的儿子结点的数目 m_i,在这 m_i 个没受限儿子结点中随机地选择一个结点作为这条路径上的下一个结点。这条路径的生成在以下结点处结束,或者它是一个叶子结点,或者该结点的所有儿子结点都已被限界。用这些 m_i 可以估算出这棵状态空间树中不受限界结点的总数 m。此数在准备检索出所有答案结点时非常有用。在这种情况下,需要生成所有的不受限结点。当只想求一个解时,由于只需生成 m 个结点的很少一部分,回溯算法就可以得到一个解,因此 m 不是一个理想的估计值。由 m_i 来估算 m,需要假定这些限界函数是固定的,即在算法执行期间当其信息逐渐增加时限界函数不变,而且同一个函数正好用于这棵状态空间树同一级的所有结点。这一假定对于大多数回溯算法并不适用。在大多数情况下,随着检索的进行限界函数会变得更强一些,在这些情况中,对 m 的估计值将大于考虑了限界函数的变化后所能得到的值。

沿用限界函数固定的假定,可以看到第 2 级没受限的结点数为 m_1。如果这棵检索树是同一级上结点有相同的度的树,那么就可预计到每一个 2 级结点平均有 m_2 个没限界的儿子,从而得出在第 3 级上有 $m_1 m_2$ 个结点。第 4 级预计没受限的结点数是 $m_1 m_2 m_3$。一般,在 i + 1 级上预计的结点数是 $m_1 m_2 \cdots m_i$。于是,在求解给定问题的实例 8.1 中所要生成的不受限界结点的估计数 $m = 1 + m_1 + m_1 m_2 + m_1 m_2 m_3 + \cdots$。

过程 ESTIMATE 是一个确定 m 值的算法。它从状态空间树的根出发选择一条随机路径。函数 SIZE 返回集合 T_k 的大小。函数 CHOOSE 从 T_k 中随机地挑选一个元素。m 和 r 是产生不受限结点估计数所用的临时工作单元。

算法 8.3　估计回溯法的效率

```
procedure ESTIMATE
    //程序沿着状态空间树中一条随机路径产生这棵树中不受限界结点的估计数//
    m←1;r←1;k←1
    loop
        T(k)←{X(k):X(k)∈T(X(1),…,X(k-1)) and B(k)(X(1),…,X(k))}
        if SIZE(T(k))=0 then exit endif
        r←r*SIZE(T(k))
        m←m+r
        X(k)←CHOOSE(T(k))
        k←k+1
    repeat
    return(m)
end ESTIMATE
```

对算法 ESTIMATE 稍加修改就可得到更准确的结点估计数。这只需增加一个 for 循环语句,选取数条不同的随机路径(一般可取 20 条),在求出沿每条路径的估计值后取平均值即得。

8.2 8-皇后问题

8-皇后问题实际上很容易一般化为 n-皇后问题,即要求找出在一个 n×n 棋盘上放置 n 个皇后并使其不能互相攻击的所有方案。令$(x_1,\cdots,x_n)$表示一个解,其中 x^n 是把第 i 个皇后放在第 i 行的列数。由于没有两个皇后可以放入同一列,因此这所有的 x_i 将是截然不同的。那么,应如何去测试两个皇后是否在同一条斜角线上呢?

如果设想棋盘的方格像二维数组 A(1∶n,1∶n)的下标那样标记,那么可以看到,对于在同一条斜角线上的由左上方到右下方的每一个元素有相同的"行-列"值,同样,在同一条斜角线上的由右上方到左下方的每一个元素则有相同的"行+列"值。假设有两个皇后被放置在(i,j)和(k,l)位置上,那么根据以上所述,仅当

$$i-j=k-l \quad \text{或} \quad i+j=k+l$$

时,它们才在同一条斜角线上。将这两个等式分别变换成

$$j-l=i-k \quad \text{与} \quad j-l=k-i$$

因此,当且仅当|j-1|=|i-k|时,两个皇后在同一条斜角线上。

过程 PLACE(k)返回一个布尔值,当第 k 个皇后能放置于 X(k)的当前值处时,这个返回值为真。这个过程测试两种情况,即 X(k)是否不同于前面 X(1),…,X(k-1)的值以及在同一条斜角线上是否根本就没有别的皇后。该过程的计算时间是 O(k-1)。

算法 8.4 可以放置一个新皇后吗

```
procedure PLACE(k)
    //如果一个皇后能放在第 k 行和 X(k)列,则返回 true;否则返回 false。X 是一个全程数组,
      进入此过程时已置了 k 个值。ABS(r)过程返回 r 的绝对值//
    global X(1∶k); integer i,k
    i←1
    while i< k do
       if X (i) = X(k)   //在同一列有两个皇后//
            or ABS (X(i) - X(k)) = ABS(i - k)   //在同一条斜角线上//
                 then return (false)
            endif
            i←i + 1
       repeat
       return (true)
end PLACE
```

使用过程 PLACE 来改进算法 8.1 给出的一般回溯方法,可得出下面求 n-皇后问题正确解的算法。

算法 8.5 n-皇后问题的所有解

```
procedure NQUEENS(n)
    //此过程使用回溯法求出在一个 n×n 棋盘上放置 n 个皇后,使其不能互相攻击的所有可能
      位置//
    integer k,n,X(1∶n)
```

```
    X(1)←0;k←1    // k 是当前行;X(k)是当前列 //
    while k > 0 do    // 对所有的行执行以下语句 //
       X(k)←X(k) + 1    // 移到下一列 //
       while X(k)≤n and not PLACE(k) do    // 此处能放这个皇后吗 //
          X(k)←X(k) + 1
       repeat
       if X(k)≤n    // 找到一个位置 //
         then if k = n    // 是一个完整的解吗 //
                    then print (X)    // 是,打印这个数组 //
                    else k←k + 1;X(k)←0    // 转向下一行 //
                    endif
       else k←k - 1    // 回溯 //
       endif
       repeat
end NQUEENS
```

此时,读者可能对于过程 NQUEENS 怎么会优于硬性处理感奇怪。原因是这样的,如果硬性要求一个 8×8 的棋盘安排出 8 块位置,就有$\binom{64}{8}$种可能的方式,即要检查将近 4 .4 $\times 10^9$ 个 8-元组。然而,过程 NQUEENS 只允许把皇后放置在不同的行和列上,因此至多需要作 8！次检查,即至多只检查 40320 个 8-元组。

可以用过程 ESTIMATE 来估算 NQUEENS 所生成的结点数。要指出的是,过程 ESTIMATE所需要的假定也适用于 NQUEENS,即,使用固定的限界函数且在检索进行时函数不改变。另外,在状态空间树的同一级的所有结点都有相同的度。图 8 .8 显示了由过程 ESTIMATE 求结点估计数所用的 5 个 8×8 棋盘。如同所要求的那样,棋盘上每一个皇后的位置是随机选取的。对于每种选择方案,都记下了可以将一个皇后合法地放在各行中列的数目(即状态树的每一级没受限的结点数)。它们都列在每个棋盘下方的向量中。向量后面的数字表示过程 ESTIMATE 由这些量值所产生的值。这 5 次试验的平均值是 1625。8-皇后状态空间树的结点总数是

$$1 + \sum_{j=0}^{7}\left(\prod_{i=0}^{j}(8 - i)\right) = 69281$$

因此,不受限结点的估计数大约只是 8-皇后状态空间树的结点总数的 2 .34%。

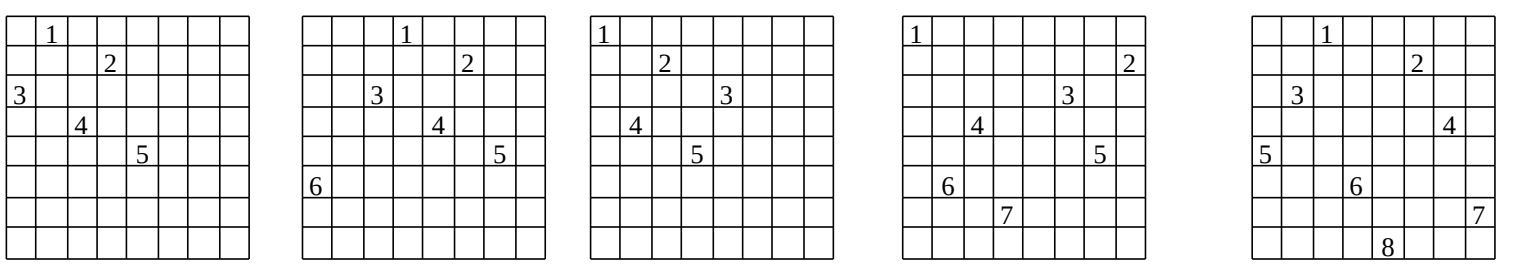


图 8 .8　8-皇后问题的 5 种方案及不受限结点的估计值

8.3 子集和数问题

子集和数问题是假定有 n 个不同的正数(通常称为权),要求找出这些数中所有使得某和数为 M 的组合。例 8.2 和例 8.4 说明了如何用大小固定或变化的元组来表示这个问题。本节将利用大小固定的元组来研究一种回溯解法,在此情况下,解向量的元素 X(i)取 1 或 0 值,它表示是否包含了权数 W(i)。

生成图 8.4 中任一结点的儿子是很容易的。对于 i 级上的一个结点,其左儿子对应于 X(i) = 1,右儿子对应于 X(i) = 0。对于限界函数的一种简单选择是,当且仅当

$$\sum_{i=1}^{k} W(i)X(i) + \sum_{i=k+1}^{n} W(i) \geq M$$

时,B(X(1),…,X(k)) = true。显然,如果这个条件不满足,X(1),…,X(k)就不能导致一个答案结点。如果假定这些 W(i)一开始就是按非降次序排列的,那么这些限界函数可以被强化。在这种情况下,如果

$$\sum_{i=1}^{k} W(i)X(i) + W(k+1) > M$$

则 X(1),…,X(k)就不能导致一个答案结点。因此,将要使用的限界函数是

$$B_k(X(1),\cdots,X(k)) = \text{true}$$

当且仅当

$$\sum_{i=1}^{k} W(i)X(i) + \sum_{i=k+1}^{n} W(i) \geq M \quad \text{且} \quad \sum_{i=1}^{k} W(i)X(i) + W(k+1) \leq M \qquad (8.1)$$

由于在即将拟制的算法中不会使用 B_n,因此不必担心这个函数中会出现 W(n + 1)。至此已说明了直接使用 8.1 节介绍的两种回溯方案中任何一种方案所需要的一切。为简单起见,将算法 8.2 修改成适应求子集和数的需要便得到递归算法 SUMOFSUB。

算法 8.6 子集和数问题的递归回溯算法

```
procedure SUMOFSUB(s,k,r)
//找 W(1:n)中和数为 M 的所有子集。进入此过程时 X(1),…,X(k-1)的值已确定。s = Σ_{j=1}^{k-1} W(j)X(j)且 r = Σ_{j=k}^{n} W(j)。这些 W(j)按非降次序排列。假定 W(1)≤M, Σ_{i=1}^{n} W(i)≥M//
1     global integer M, n; global real W(1:n); global boolean X(1:n)
2     real r,s; integer k,j
      //生成左儿子。注意,由于 B_{k-1} = true,因此 s + W(k)≤M//
3     X(k)←1
4     if s + W(k) = M then   //子集找到//
5           print (X(j),j←1 to k)
            //此处由于 W(j) > 0,1≤j≤n,因此不存在递归调用//
6                 else
7           if s + W(k) + W(k+1)≤M then   //B_k = true//
8             call SUMOFSUB (s + W(k),k + 1,r - W(k))
9           endif
10    endif
      //生成右儿子和计算 B_k 的值//
```

```
11      if s+r-W(k)≥M and s+W(k+1)≤M    //Bk = true//
12         then X(k)←0
13            call SUMOFSUB (s,k+1,r-W(k))
14      endif
15      end SUMOFSUB
```

过程 SUMOFSUB 将$\sum_{i=1}^{k} W(i)X(i)$和$\sum_{i=k+1}^{n} W(i)$分别保存在变量 s 和 r 中以避免每次都要计算这些值。此算法假定 $W(1)\leq M$ 和$\sum_{i=1}^{n} W(i)\geq M$。初次调用是 call SUMOFSUB(0,1,$\sum_{i=1}^{n} W(i)$)。着重要指出的是,算法没有明显地使用测试条件 k > n 去终止递归。之所以不需要这一测试条件,是因为在算法入口处 s≠M 且 s + r≥M,因此 r≠0,从而 k 也不可能大于 n。在第 7 行由于 s + W(k) < M 并且 s + r≥M,因此可以得出 r≠W(k),从而 k + 1≤n。还要注意,如果 s + W(k) = M(第 4 行),则 X(k + 1),…,X(n)应为 0。这些 0 在第 5 行的输出中被略去。在第 7 行没作$\sum_{i=1}^{k} W(i)X(i) + \sum_{i=k+1}^{n} W(i)\geq M$ 测试,这是因为已经知道s + r≥M 和 X(k) = 1。

例 8.6　图 8.9 示出了过程 SUMOFSUB 对 n = 6, M = 30 和 W(1∶6) = (5,10,12,13,15,18)这种情况处理时所生成的一部分状态空间树。矩形结点列出了对 SUMOFSUB 的每次调用时 s,k,r 的值。圆形结点表示其和为 M 的一个子集被打印出来的处所。在结点 A,B 和 C 处,输出分别为(1,1,0,0,1),(1,0,1,1)和(0,0,1,0,0,1)。要指出的是,图 8.9 所示的这棵树只包含 20 个矩形结点,而 n = 6 的满状态空间树中对 SUMOFSUB 调用的结点可以有 2^6 - 1 = 63 个(由于根本不需要从一个叶结点作出调用,因此这一计数排除了 64 个叶结点)。

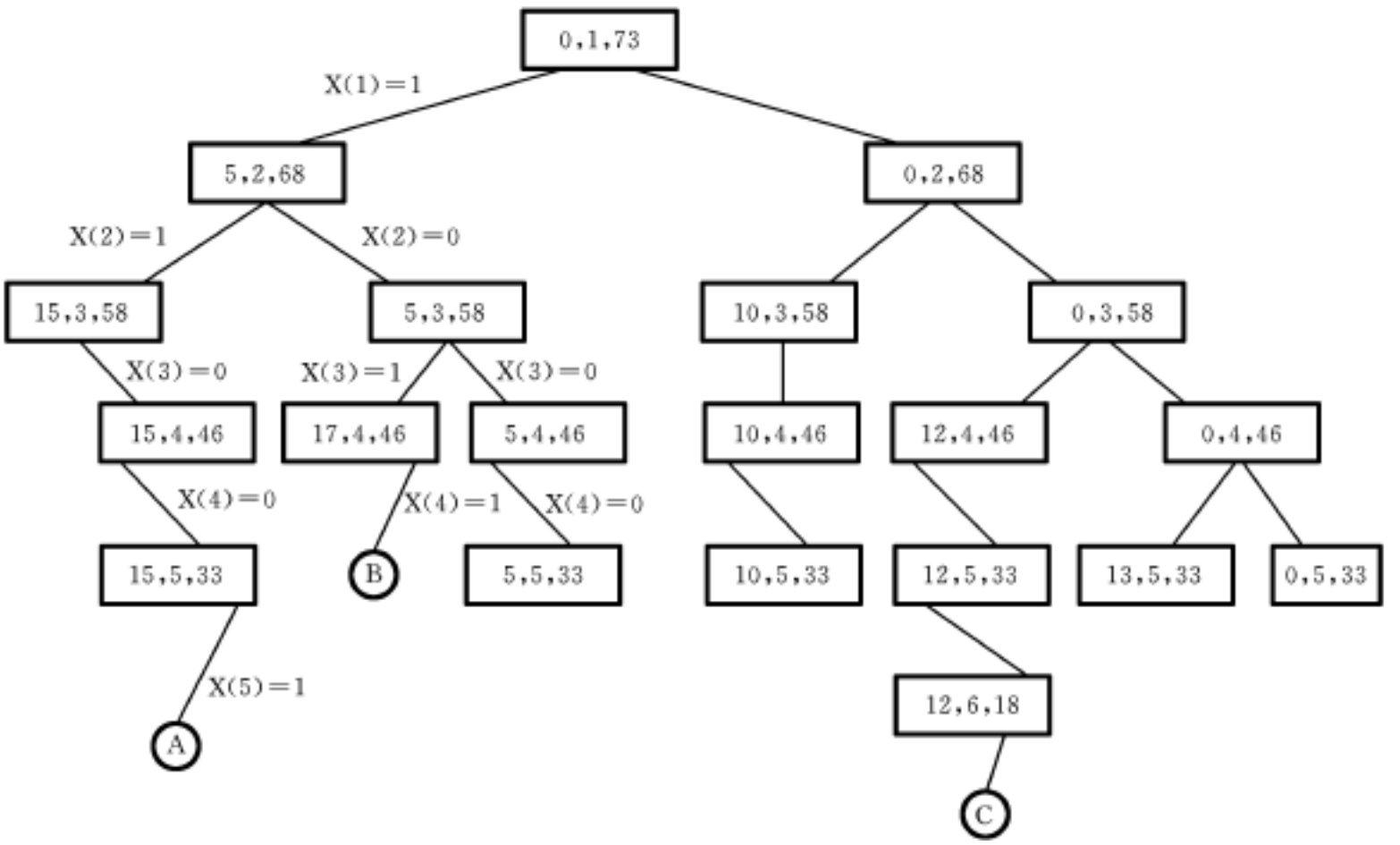


图 8.9　由 SUMOFSUB 所生成状态空间树的一部分

8.4 图的着色

已知一个图 G 和 m > 0 种颜色,在只准使用这 m 种颜色对 G 的结点着色的情况下,是否能使图中任何相邻的两个结点都具有不同的颜色呢?这个问题称为 m-着色判定问题。在 m 着色最优化问题则是求可对图 G 着色的最小整数 m。这个整数称为图 G 的色数。

对图着色的研究是从 m-可着色性问题的著名特例——4 色问题开始的。这个问题要求证明平面或球面上的任何地图的所有区域都至多可用 4 种颜色来着色,并使任何两个有一段公共边界的相邻区域没有相同的颜色。这个问题可转换成对一平面图的 4-着色判定问题(平面图是一个能画于平面上而边无任何交叉的图)。将地图的每个区域变成一个结点,若两个区域相邻,则相应的结点用一条边连接起来。图 8.10 显示了一幅有 5 个区域的地图以及与该地图对应的平面图。多年来,虽然已证明用 5 种颜色足以对任一幅地图着色,但是一直找不到一定要求多于 4 种颜色的地图。直到 1976 年这个问题才由爱普尔(K.I.Apple),黑肯(W.Haken)和考西(J.Koch)利用电子计算机的帮助得以解决。他们证明了 4 种颜色足以对任何地图着色。在这一节,不是只考虑那些由地图产生出来的图,而是所有的图。讨论在至多使用 m 种颜色的情况下,可对一给定的图着色的所有不同方法。

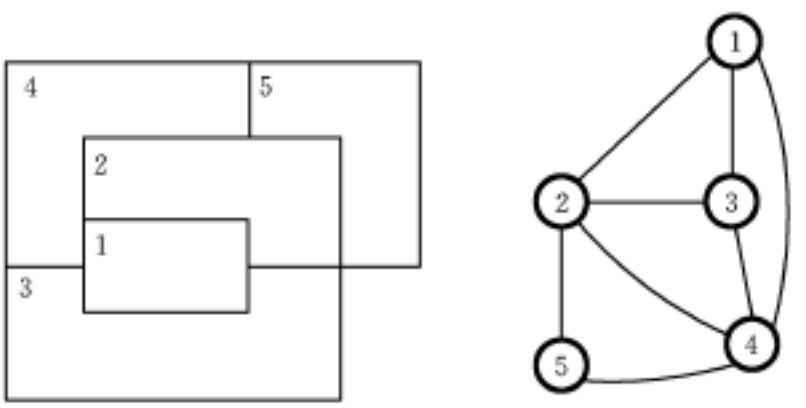


图 8.10 一幅地图和它的平面图表示

假定用图的邻接矩阵 GRAPH(1:n,1:n)来表示一个图 G,其中若(i,j)是 G 的一条边,则 GRAPH(i,j) = true,否则 GRAPH(i,j) = false。因为要拟制的算法只关心一条边是否存在,所以使用布尔值。颜色用整数 1,2,…,m 表示,解则用 n-元组(X(1),…,X(n))来给出,其中 X(i)是结点 i 的颜色。使用和算法 8.2 相同的递归回溯表示,得到算法 MCOLORING。此算法使用的基本状态空间树是一棵度数为 m,高为 n+1 的树。在 i 级上的每一个结点有 m 个儿子,它们与 X(i)的 m 种可能的赋值相对应,1≤i≤n。在 n+1 级上的结点都是叶结点。图 8.11 给出了 n=3 且 m=3 时的状态空间树。

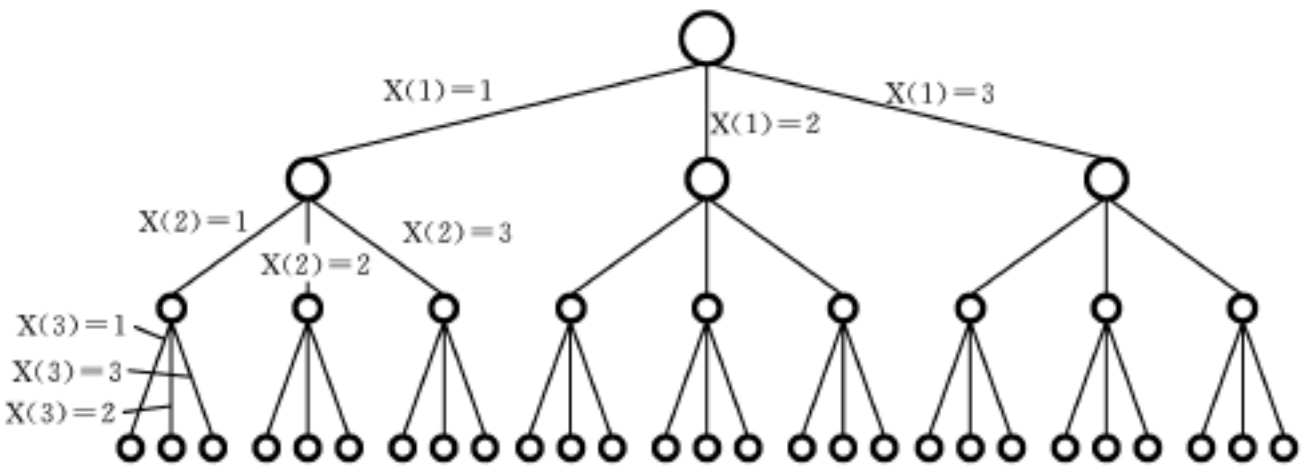


图 8.11 当 n=3,m=3 时 MCOLORING 的状态空间树

算法 8.7　找一个图的所有 m-着色方案

```
procedure MCOLORING(k)
    //这是图着色的一个递归回溯算法。图 G 用它的布尔邻接矩阵 GRAPH(1：n,1：n)表示//
    //它计算并打印出符合以下要求的全部解,把整数 1,2,…,m 分配给图中//
    //各个结点且使相邻近的结点的有不同的整数。k 是下一个要着色结点的下标//
    global integer m,n,X(1：n) boolean GRAPH(1：n,1：n)
    integer k
    loop   //产生对 X(k)所有的合法赋值//
      call NEXTVALUE(k)   //将一种合法的颜色分配给 X(k)//
      if X(k) = 0 then exit endif   //没有可用的颜色了//
      if k = n
        then print (X)   //至多用了 m 种颜色分配给 n 个结点//
        else call MCOLORING(k + 1)   //所有 m-着色方案均在此反复递归调用中产生//
      endif
      repeat
    end MCOLORING
```

在最初调用 call MCOLORING(1)之前,应对图的邻接矩阵置初值并对数组 X 置 0 值。

在确定了 X(1)到 X(k - 1)的颜色之后,过程 NEXTVALUE 从这 m 种颜色中挑选一种符合要求的颜色,并把它分配给 X(k),若无可用的颜色,则返回 X(k) = 0。

算法 8.8　生成下一种颜色

```
procedure NEXTVALUE(k)
    //进入此过程前 X(1),…,X(k - 1)已分得了区域[1,m]中的整数且相邻近的结点有不同的
      整数。本过程在区域[0,m]中给 X(k)确定一个值:如果还剩下一些颜色,它们与结点 k 邻
      接的结点分配的颜色不同,就将其中最高标值的颜色分配给结点 k;如果没剩下可用的颜
      色,则置 X(k)为 0//
    global integer m,n, X(1：n) boolean GRAPH(1：n,1：n)
    integer j,k
    loop
    X(k)←(X(k) + 1) mod (m + 1)   //试验下一个最高标值的颜色//
    if X(k) = 0 then return endif   //全部颜色用完//
    for j←1 to n do   //检查此颜色是否与邻近结点的那些颜色不同//
      if GRAPH(k,j) and   //如果(k,j)是一条边//
         X(k) = X(j)   //并且邻近的结点有相同的颜色//
         then exit endif
    repeat
    if j = n + 1 then return endif   //找到一种新颜色//
    repeat   //否则试着找另一种颜色//
end NEXTVALUE
```

算法 8.7 的计算时间上界可以由状态空间树的内部结点数 $\sum_{i=0}^{n-1} m^i$ 得到。在每个内部结点处,为了确定它的儿子们所对应的合法着色,由 NEXTVALUE 所花费的时间是 O(mn)。因此,总的时间由 $\sum_{i=1}^{n} m^i n = n(m^{n+1} - m)/(m-1) = O(nm^n)$ 所限界。

图 8 .12 显示了一个包含 4 个结点的简单图。下面是一棵由过程 MCOLORING 生成的树。到叶子结点的每一条路径表示一种至多使用 3 种颜色的着色法。

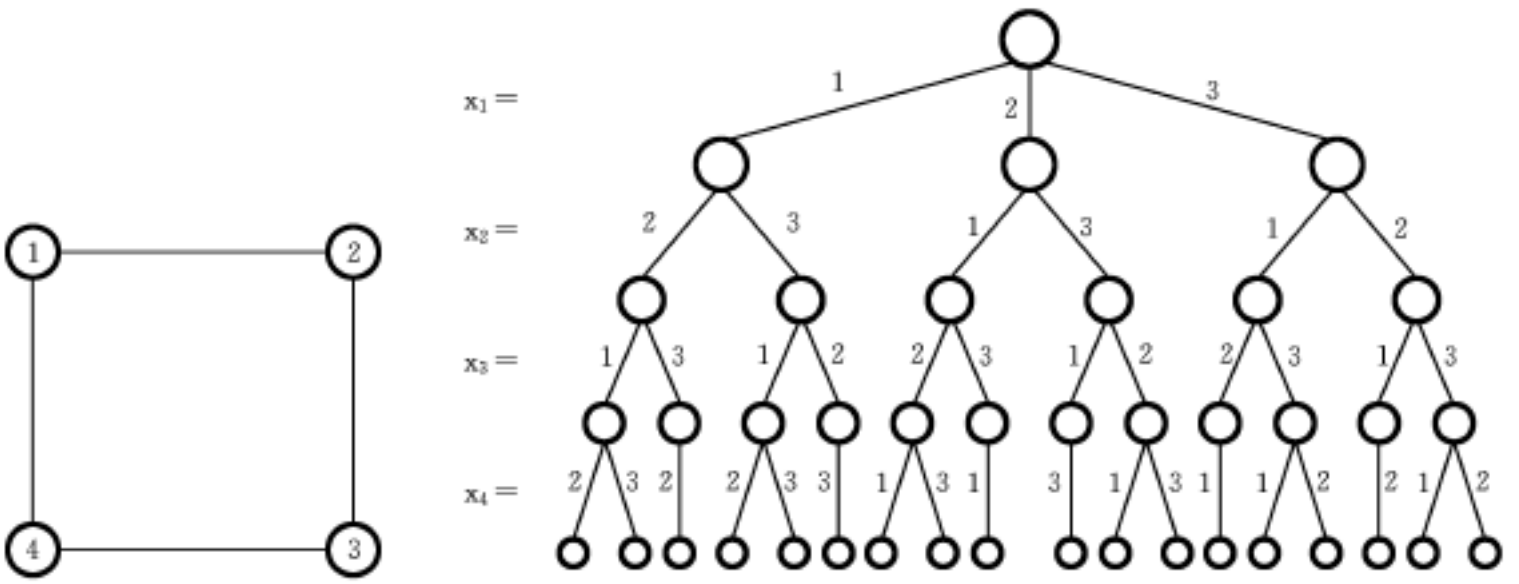


图 8 .12　一个 4 结点图和所有可能的 3 着色

注意:正好用 3 种颜色的解只有 12 种。

8 .5　哈 密 顿 环

设 G = (V, E)是一个 n 结点的连通图。一个哈密顿环是一条沿着图 G 的 n 条边环行的路径,它访问每个结点一次并且返回到它的开始位置。换言之,如果一个哈密顿环在某个结点 $v_1 \in V$ 处开始,且 G 中结点按照 $v_1, v_2, \cdots, v_{n+1}$ 的次序被访问,则边 (v_i, v_{i+1}), $1 \leq i \leq n$, 均在 G 中,且除了 v_1 和 v_{n+1} 是同一个结点外,其余的结点均各不相同。

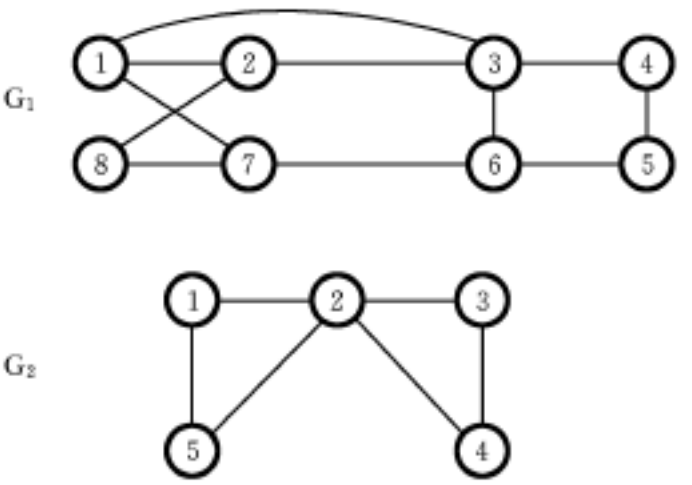


图 8 .13　两个图,第一个包含一个哈密顿环

在图 8 .13 中,图 G_1 含有一个哈密顿环 1, 2, 8, 7, 6, 5, 4, 3, 1。图 G_2 不包含哈密顿环。似乎没有一种容易的方法能确定一个已知图是否包含哈密顿环。本节考察找一个图中所有哈密顿环的回溯算法。这个图可以是有向图也可以是无向图。只有不同的环才会被输出。

用向量 $(x_1, \cdots, x_n)$ 表示用回溯法求得的解,其中,x_i 是找到的环中第 i 个被访问的结点。如果已选定 $x_1, \cdots, x_{k-1}$,那么下一步要做的工作是如何找出可能作 x_k 的结点集合。若 k = 1,则 X(1)可以是这 n 个结点中的任一结点,但为了避免将同一个环重复打印 n 次,可事先指定 X(1) = 1。若 $1 < k < n$,则 X(k)可以是不同于 X(1), X(2), ⋯, X(k - 1)且和 X(k - 1)有边相连的任一结点 v。X(n)只能是唯一剩下的且必须与 X(n - 1)和 X(1)皆有边相连的结点。过程 NEXTVALUE 给出了在求哈密顿环的过程中如何找下一个结点的算法。

算法 8 .9　生成下一个结点

```
procedure NEXTVALUE(k)
    // X(1),…,X(k - 1)是一条有 k - 1 个不同结点的路径。若 X(k) = 0,则表示再无结点可分
       配给 X(k)。若还有与 X(1),…,X(k - 1)不同且与 X(k - 1)有边相连接的结点,则将其中
       标数最高的结点置于 X(k)。若 k = n,则还需与 X(1)相连接 //
    global integer n,X(1：n), boolean GRAPH(1：n,1：n)
    integer k,j
    loop
      X(k)←(X(k) + 1) mod (n + 1)    // 下一个结点//
      if X(k) = 0 then return endif
      if GRAPH (X(k - 1),X(k))    // 有边相连吗//
        then for j←1 to k - 1 do    // 检查与前 k - 1 个结点是否相同//
              if X(j) = X(k)
                then exit    // 有相同结点,出此循环//
              endif
      repeat
      if j = k    // 若为真,则是一个不同结点//
        then if k < n or (k = n and GRAPH(X(n),1)) then return
              endif
      endif
        endif
      repeat
end NEXTVALUE
```

使用过程 NEXTVALUE 和将递归回溯算法 8 .2 细化得到算法 HAMILTONIAN。此算法可找出所有的哈密顿环。

算法 8 .10　找所有的哈密顿环

```
procedure HAMILTONIAN(k)
    // 这是找出图 G 中所有哈密顿环的递归算法。图 G 用它的布尔邻接矩阵 GRAPH(1：n,1
       ：n)表示。每个环都从结点 1 开始//
    global integer X(1：n)
    local integer k,n
    loop    // 生成 X(k)的值//
          call NEXTVALUE(k)    // 下一个合法结点分配给 X(k)//
          if X(k) = 0 then return endif
          if k = n
            then print (X,‘1’)    // 打印一个环//
            else call HAMILTONIAN(k + 1)
          endif
        repeat
end HAMILTONIAN
```

这个过程首先初始化邻接矩阵 GRAPH(1：n,1：n),然后置 X(2：n)←0,X(1)←1,再执行 call HAMILTONIAN(2)。

回想 6.7 节的货郎担问题,该问题要求找一条成本最小的"周游路线"。这条周游路线是一个哈密顿环。如果一个图每条边的成本都相同且存在周游路线,则用过程 HAMILTONIAN就可找出全部周游路线,且它们都是最小成本周游路线。

8.6 背包问题

在这一节,将重新考虑曾在第 6 章定义和求解的 0/1 背包问题。假定 n 个物品的重量 w_i,效益值 p_i 和背包容量 M 均为已知的正数,这个问题要求选择出重量的一个子集,使得

$$\sum_{1\le i\le n} w_i x_i \le M \qquad \text{且极大化} \quad \sum_{1\le i\le n} p_i x_i \tag{8.2}$$

这 n 个 x 构成一个取 0/1 值的向量。

这个问题的解空间由各分量 x_i 分别取 0 或 1 值的 2^n 个不同的 n 元向量组成,因此这个解空间与子集和数问题的解空间相同。它有两种可能的树结构,一种对应于元组大小固定的表示(图 8.4),另一种对应于元组大小可变的表示(图 8.3)。用其中任一种树结构都可导出背包问题的回溯算法。在选择限界函数上,有一条需遵循的基本原则,即无论使用哪种限界函数,都应有助于杀死某些活结点,使这些活结点实际上不再扩展。背包问题的一个好的限界函数可以取成能产生某些值的上界的函数。如果扩展给定活结点和它的任一子孙所导致最好可行解值的上界不大于迄今所确定的最好解之值,就可杀死此活结点。

本节仍使用大小固定的元组表示。如果在结点 Z 处已确定了 x_i 的值,$1\le i\le k$,则 Z 的上界可由下法获得,对 $k+1\le i\le n$,将 $x_i=0$ 或 1 的要求放宽为 $0\le x_i\le 1$,然后用 5.3 节的贪心算法求解这个放宽了要求的问题。过程 BOUND(p, w, k, M)确定通过扩展这棵状态空间树在 k+1级上的任一结点 Z 所能得到的最好解的上界。这些物品的重量和效益值是 W(i)和 P(i)。$p=\sum_{i=1}^{k} P(i)X(i)$, $w=\sum_{i=1}^{k} W(i)X(i)$,并且假定 $P(i)/W(i)\ge P(i+1)/W(i+1)$, $1\le i<n$。

算法 8.11 限界函数

```
procedure BOUND(p, w, k, M)
    // p 为当前效益总量; w 为当前背包重量; k 为上次去掉的物品; M 为背包容量; 返回一个新
       效益值 //
    global n, P(1:n), W(1:n)
    integer k, i; real b, c, p, w, M
    b←p; c←w
    for i←k+1 to n do
      c←c+W(i)
      if c<M then b←b+P(i)
              else return (b+(1-(c-M)/W(i))*P(i))
       endif
    repeat
    return (b)
end BOUND
```

由算法 8.11 可以得出,状态空间树中结点 Z 的可行左儿子的上界与 Z 的上界是一样的。因此,回溯算法无论什么时候作一次向某结点左儿子的移动,均不需要再调用限界函数

BOUND。由于每当出现可在左儿子和右儿子之间作一次移动的机会时,回溯算法总是试图移到其左儿子,因此可以看出,只有在一系列的左儿子移动之后(即可行的左儿子的移动),才需要调用 BOUND。由迭代回溯算法(算法 8 .1)得出算法 BKNAP1。

算法 8 .12　0/ 1 背包问题的回溯法求解

```
procedure BKNAP1(M,n,W,P,fw,fp,X)
    // M 是背包容量。有 n 种物品,其重量与效益分别存在数组 W(1 : n)和 P(1 : n)中;P(i)/
      W(i)≥P(i+1)/ W(i+1)。fw 是背包的最后重量,fp 是背包取得的最大效益。X(1 : n)中
      每个元素取 0 或 1 值。若物品 k 没放入背包,则 X(k) = 0;否则 X(k) = 1 //
1   integer n,k, Y(1 : n) ,i, X(1 : n) ;real M,W(1 : n) ,P(1 : n) ,fw,fp, cw,cp;
2   cw←cp←0;k←1;fp← - 1   // cw 是背包当前重量;cp 是背包当前取得的效益值//
3   loop
4     while k≤n and cw + W(k)≤M do   // 将物品 k 放入背包//
5       cw←cw + W(k) ;cp←cp + P(k) ;Y(k)←1;k←k + 1
6     repeat
7     if k > n then fp←cp;fw←cw;k←n;X←Y   // 修改解//
8       else Y(k)←0   // 超出 M,物品 K 不适合//
9     endif
10    while BOUND(cp、cw,k, M)≤fp do   // 上面置了 fp 后,BOUND = fp//
11    while k≠0 and Y(k)≠1 do
12      k←k - 1   // 找最后放入背包的物品//
13    repeat
14    if k = 0 then return endif   // 算法在此处结束//
15    Y(k)←0;cw←cw - W(k) ;cp←cp - P(k)   // 移掉第 k 项//
16    repeat
17    k←k + 1
18  repeat
19  end BKNAP1
```

当 $fp \neq -1$ 时,$X(i), 1 \leq i \leq n$,是这样的一些元素,它们使得 $\sum_{i=1}^{n} P(i)X(i) = fp$。在 4～6 行的 while 循环中,回溯算法作一连串到可行左儿子的移动。$Y(i), 1 \leq i \leq k$,是到当前结点的路径。$cw = \sum_{i=1}^{k-1} W(i)Y(i)$且 $cp = \sum_{i=1}^{k-1} P(i)Y(i)$。在第 7 行,如果 k > n,则必有 cp > fp,因为若 cp≤fp,则在上一次使用限界函数时(第 10～16 行)就会终止到此叶结点的路径。如果 k≤n,则W(k)不适合,从而必须作一次到右儿子的移动。所以在第 8 行,Y(k)被置成 0。如果在第 10 行中 BOUND≤fp,由于现今的这条路径不能导出比迄今所找到的最好解还要好的解,因此该路径可终止。在第 11～13 行,沿着到最近结点的路径回溯,而由这最近结果可以作迄今尚未试验过的移动。如果不存在这样的结点,则算法在第 14 行终止。反之,Y(k),cw 和 cp 则对应于一次右儿子移动作出相应的修改。计算这个新结点的界。继续倒转去处理第 10～16 行,一直到作出有可能得到一个大于 fp 值的解的右儿子结点为止,否则 fp 就是背包问题的最优效益值。注意,第 10 行的限界函数并不是固定的,这是因为当检索这棵树的更多结点时,fp 就改变。因此限界函数动态地被强化。

例 8.7　考虑以下情况的背包问题: P = (11, 21, 31, 33, 43, 53, 55, 65), W = (1, 11, 21, 23, 33, 43, 45, 55), M = 110, n = 8。

图 8.14 显示了对向量 Y 作出各种选择的情况下所生成的树,这棵树的第 i 级对应于将 1 或 0 赋值给 Y(i),表示或者含有重量 W(i)或者拒绝接纳重量 W(i)。一个结点内所载的两个数是重量(cw)和效益(cp),给出了该结点的下一级的两个赋值。注意,若结点不含有重量和效益值则表示此两类值与它们父亲的相同。每个右儿子以及根结点外面的数是对应于那个结点的上界。左儿子的上界与它父亲的相同。算法 8.12 的变量 fp 在结点 A、B、C 和 D 的每一处被修改。每次修改 fp 时也修改 X。终止时, fp = 159 和 X = (1, 1, 1, 0, 1, 1, 0, 0)。在状态空间树的 $2^9 - 1 = 511$ 个结点中只生成了其中的 35 个结点。注意到由于所有的 P(i) 都是整数而且所有可行解的值也是整数,因此 BOUND(p, w, k, M) 是一个更好的限界函数,使用此限界函数结点 E 和 F 就不必再扩展,从而生成的结点数可减少到 28。

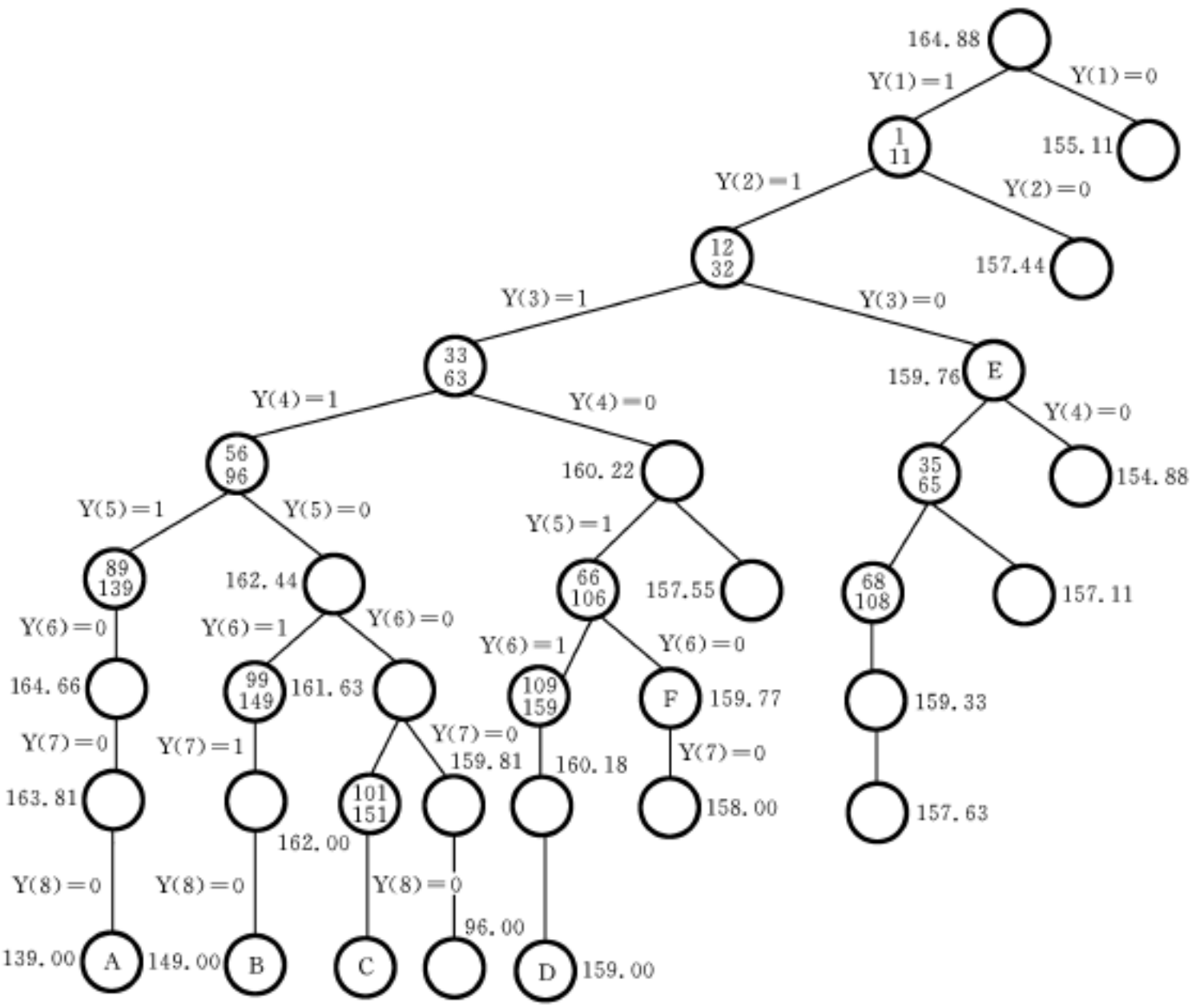


图 8.14　算法 8.12 所生成的树

每次在第 10 行调用 BOUND 时,在过程 BOUND 中基本上重复了第 4～6 行的循环,因此可对算法 BKNAP1 作进一步的改进。为了取消 BKNAP1 第 4～6 行所做的工作,需要把 BOUND 改变成一个具有边界效应的函数。这两个新算法 BOUND1 和 BKNAP2 以算法 8.13和8.14的形式写出。这两个算法与算法 8.11 和 8.12 中同名的变量含意完全一样。

算法 8 .13　有边界效应的限界函数

```
procedure BOUND1 (P,W,k,M,pp,ww,i)
    //新近移到的左儿子所对应的效益为 pp,重量为 ww。i 是上一个不适合的物品。如果所有
      物品都试验过了,则 i 的值是 n + 1//
    global n,P(1：n),W(1：n),Y(1：n)
    integer k,i; real p,w,pp,ww,M,b
    pp←p;ww←w
    for i←k + 1 to n do
      if ww + W(i)≤M then ww←ww + W(i);pp←pp + P(i);Y(i)←1
                      else return (pp + (M - ww) * P(i)/ W(i))
      endif
    repeat
    return (pp)
end BOUND1
```

算法 8 .14　改进的背包算法

```
procedure BKNAP2(M,n,W,P,fw,fp,X)
    //与 BKNAP1 同//
    integer n,k,Y(1：n),i,j,X(1：n)
    real W(1：n),P(1：n),M,fw,fp,pp,ww,cw,cp
    cw←cp←k←0;fp← - 1
    loop
      while BOUND1 (cp,cw,k,M,pp,ww,j)≤fp do
      while k≠0 and Y(k)≠1 do
      k←k - 1
      repeat
      if k = 0 then return endif
      Y(k)←0;cw←cw - W(k);cp←cp - P(k)
      repeat
      cp←pp;cw←ww;k←j   //等价于 BKNAP1 中 4～6 行的循环//
      if k > n then fp←cp;fw←cw;k←n;X←Y
               else Y(k)←0
      endif
    repeat
end BKNAP2
```

到目前为止,所讨论的都是在静态状态空间树环境下工作的回溯算法,现在讨论如何利用动态状态空间树来设计背包问题的回溯算法。下面介绍的一种回溯算法的核心思想是以 5 .3 节贪心算法所得的贪心解为基础来动态地划分解空间,并且力图去得到 0/ 1 背包问题的最优解。首先用约束条件 $0\leq x_i\leq 1$ 来代换 $x_i = 0$ 或 1 的整数约束条件,于是得到一个放宽了条件的背包问题:

$$\max \sum_{1\leq i\leq n} p_i x_i$$

约束条件

$$\sum_{1\leq i\leq n} w_i x_i \leq M, 0\leq x_i \leq 1, 1\leq i\leq n \qquad (8.3)$$

用贪心方法解式(8.3)这个背包问题,如果所得贪心解的所有 x_i 都等于 0 或 1,显然这个解也是相应 0/1 背包问题的最优解。如若不然,则必有某 x_i 使得 $0 < x_i < 1$。利用这个 x_i 把式(8.2)的解空间划分成两个子空间,使 $x_i = 0$ 在一个子空间,$x_i = 1$ 在另一子空间。于是,表示此解空间的状态空间树的左子树是以 $x_i = 0$ 为根的左分枝的子树,右子树是以 $x_i = 1$ 为右分枝的子树。一般说来,在状态空间树的每个结点 Z 处用贪心方法求解式(8.3)是在有附加限制的情况下进行的,这附加限制是已对由根到结点 Z 的那条路径进行了赋值。换言之,是在给出了根到结点 Z 那条路径各边所对应的那些 x_i 的值(它们各自取 0 或 1 值)之后,再用贪心方法求解式(8.3)。如果贪心解全是 0 或 1,则找到了此情况下的最优解。如若不然,则必有 x_i 使得 $0 < x_i < 1$。那么取 $x_i = 0$ 为结点 Z 的左分枝,$x_i = 1$ 为 Z 的右分枝。这种划分式(8.2)解空间方法之所以合理是因为在贪心解含有非整数 x_i 的情况下不仅防止了把这个贪心解作为 0/1 背包问题的可行解,并且通过强迫这个 x_i 取 0 或 1 值可以快速得到 0/1 背包问题的一个可行的贪心解。

由于贪心算法要求 $p_j/w_j \geq p_{j+1}/w_{j+1}$,因此可以预料大多数标记数较小的物品(即代表某物品的 j 值小因此密度高)在最优装法中会装入背包。当 x_i 置为 0 时,不需防止贪心算法装入任何 $j < i$ 的物品(除非 x_j 已被置成了 0),但当 x_i 置为 1 时,可能有某些 $j < i$ 的物品将不能放入背包,因此预计可能在 $x_i = 0$ 的情况下达到最优解。以上分析促使我们在设计回溯算法时考虑先试验 $x_i = 0$ 的方案,于是在构造动态状态空间树时就应选取左分枝与 $x_i = 0$ 相对应,右分枝与 $x_i = 1$ 对应。

例 8.8　利用例 8.7 的数据试验上述动态划分下回溯算法的执行。

与根结点对应的贪心解,即式(8.3)的贪心解是 $x = (1,1,1,1,1,1,21/43,0,0)$,效益值为 164.88。根结点的两棵子树分别对应于 $x_6 = 0$ 和 $x_6 = 1$(见图 8.15)。结点 2 处的贪心解为 $x = (1,1,1,1,1,0,21/45,0)$,效益值为 164.66。结点 2 处的解空间用 $x_7 = 0$ 和 $x_7 = 1$ 来划分。下一个 E-结点是结点 3,此处的解有 $x_8 = 21/55$。此时划分具有 $x_8 = 0$ 和 $x_8 = 1$。在结点 4 处的解全是整数,因此没有必要进一步扩展这一结点。至此,这个 0/1 背包问题所找到的最好解是 $X = (1,1,1,1,1,0,0,0)$,效益值为 139。结点 5 是下一个 E-结点,这个结点的贪心解是 $X = (1,1,1,22/23,0,0,0,1)$。效益值为 159.56。这个划分现在具有 $x_4 = 0$ 和 $x_4 = 1$。在结点 6 处的贪心解有值 158.66 且 $x_5 = 2/3$。紧接着结点 7 就变成为 E-结点,此处的解是(1,1,1,0,0,0,0,1),效益值是 128。在这里贪心解全是整数,结点 7 不再扩展。在结点 8 处,贪心解有值 157.71 且 $x_3 = 4/7$。结点 9 处的解全是整数且有值 140。结点 10 处的贪心解是(1,0,1,0,1,0,0,1)。它的值是 150。下一个 E-结点是结点 11。它的值是 159.52 且 $x_3 = 20/21$。这划分现在是在 $x_3 = 0$ 和 $x_3 = 1$ 上。此背包问题回溯处理的剩余部分留作习题。

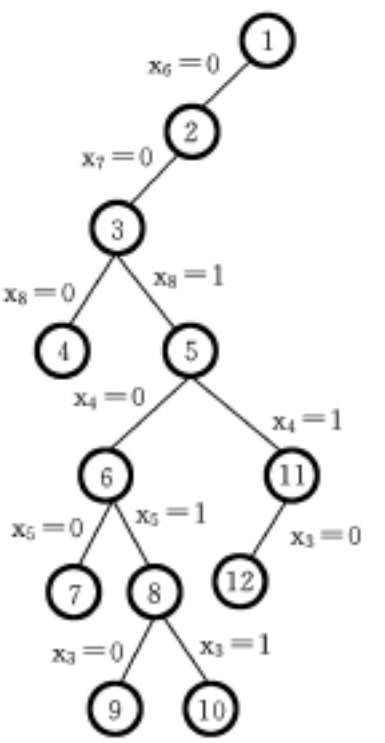


图 8.15　用例 8.7 的数据生成的部分动态状态空间树

许多实验数据结果表明,在使用静态树的情况下,运行背包问题的回溯算法所用的时间一般比用一棵动态树时要少。然而,动态划分方法在求解整数规划问题上是非常有用的。一般,整数

规划的数学表示为

$$\text{极小化} \quad \sum_{1\le j\le n} c_j x_j ,$$

$$\text{约束条件} \quad \sum_{1\le j\le n} a_{ij} x_j \le b_i , 1\le i\le m \tag{8.4}$$

且这些 x_j 是非负整数。

如果将式(8.4)中的那些 x_i 的整数约束条件用约束条件 $x_i \ge 0$ 来代替，则得到一个至少有一个解值和式(8.4)最优解的值一样大的线性规划问题。线性规划问题可用单纯形法求解(线性规划的单纯形法可在任何一本有关最优化方法的书中找到)。如果解不全是整数，则选择一个非整数 x_i 划分这个解空间。假定在状态空间树任一结点 Z 处，与此线性规划相应的最优解中 x_i 的值是 v，且 v 不是一个整数。Z 的左儿子与 $x_i \le \lfloor v \rfloor$ 相对应，而 Z 的右儿子则与 $x_i \le \lceil v \rceil$ 相对应。由于所导出的状态空间树可能有无限的深度(注意：在由根到结点 Z 的路径上，解空间能够在一个 x_i 处被划分多次，这是因为每一个 x_i 都可以有任一非负整数值)，因此，几乎总是用分枝-限界法来进行检索。

习　题　八

8.1　修改算法 8.1 和 8.2，使它们只求出问题的一个解而不用求出全部解。

8.2　使用 3.5 节给出的规则将递归算法 8.2 转换成等价的迭代模型，然后尽量将此模型简化并与算法 8.1 相比较。

8.3　重新定义过程 PLACE(k)，使它的返回值或者是第 k 个皇后可以放于其上的合法列号，或者是一个非法值，这样可提高一些过程 NQUEENS 的效率。按以上策略重写这两个过程。

8.4　对于 n-皇后问题，可以发现一些解是另一些解的简单反射或旋转的结果。例如，n = 4 时图 8.16 的两个解在反射意义下是等效的。为了找出所有不等效的解，此算法只需置 X(1) = 2,3,…,$\lceil n/2 \rceil$。修改过程 NQUEENS，使其只计算不等效的解。

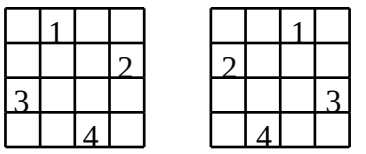


图 8.16　4-皇后问题的等效解

8.5　对 8.4 题所得到的 n-皇后问题算法在 n = 8,9,10 情况下投入运行，将每个 n 值下所找出解的数目列成表。

8.6　在一个 n×n 棋盘的任一坐标为(x,y)的方格上放有一只(国际象棋的)马，它走到下列坐标方格之一都是一合法着(当然不准走离棋盘)：(x - 2,y + 1)；(x - 1,y + 2)；(x + 1,y + 2)；(x + 2,y + 1)；(x + 2,y - 1)；(x + 1,y - 2)；(x - 1,y - 2)；(x - 2,y - 1)。要求写一算法判定马从(x,y)方格出发，在只准访问每个方格一次的情况下，是否能用 $n^2 - 1$ 步走完棋盘的其余 $n^2 - 1$ 个方格。若能，输出这条访问路线；否则输出不存在此路线的答案。

8.7　分派问题一般陈述如下：给 n 个人分派 n 件工作，把工作 j 分配给第 i 个人的成本为 COST(i,j)。设计一个回溯算法，在给每个人分派一件不同工作的情况下使得总成本最小。

8.8　设 W = (5,7,10,12,15,18,20)和 M = 35，使用过程 SUMOFSUB 找出 W 中使得和数等于 M 的全部子集并画出所生成的部分状态空间树。

8.9　用以下数据运行过程 SUMOFSUB：M = 35 和

① W = (5,7,10,12,15,18,20)，

② W = (20,18,15,12,10,7,5)，

③ W = (15,7,20,5,18,10,12)。

这 3 种情况的计算时间有明显的差别吗？

8.10 用与元组大小可变的表示相对应的状态空间树，写一个子集和数问题的回溯算法。

8.11 以大小为 n = 2,3,4,5,6,7 的这些完全图为数据运行算法 MCOLORING。假定要求用 k = n 和 k = n/2种颜色，对每个 n 和 k 值列出计算时间表。

8.12 分析回溯算法 HAMILTONIAN 的最坏情况计算时间。

8.13 用算法 8.10 画出图 8.13 中 G1 所生成的部分状态空间树。

8.14 用元组大小可变的表示方法写一个求解 0/1 背包问题的回溯算法。

8.15 利用上题所写的算法求解例 8.6 的背包问题并画出由此生成的部分状态空间树。

8.16 补全图 8.15 的状态空间树。

8.17 用 8.6 节讨论的动态状态空间树写一个背包问题的回溯算法。

8.18 [邮票问题]设想一个国家发行 n 种不同面值的邮票，并假定每封信上至多只允许贴 m 张邮票。对于给定的 m 和 n 值，写一个算法求出从邮资 1 开始在增量为 1 的情况下可能获得的邮资值的最大连续区域以及获得此区域的各种可能面值的集合。例如，对于 n = 4 和 m = 5，若有面值为(1,4,12,21)的四种邮票，则邮资最大连续区域为 1 到 71。还有其它面值的四种邮票可组成同样大小的区域吗？

8.19 假定有 32 张牌像图 8.17 那样摆在有 33 格的盘上，只有中心格空着。现规定当一张牌跳过邻近的一张牌到空格时，就将这张邻近的牌从盘上拿掉。写一个算法来找出一系列跳步，使除了最后留在中心格一张牌外其余的牌均被拿掉。

8.20 设想有 12 个平面图形，每个图形由 5 个大小相同的正方形组成，每个图形的形状均与别的图形不同。图 8.18 中用 12 个上述的图形拼成了一个 6×10 的长方形。写一个算法找出将这些图形拼成 6×10 长方形的全部摆法。

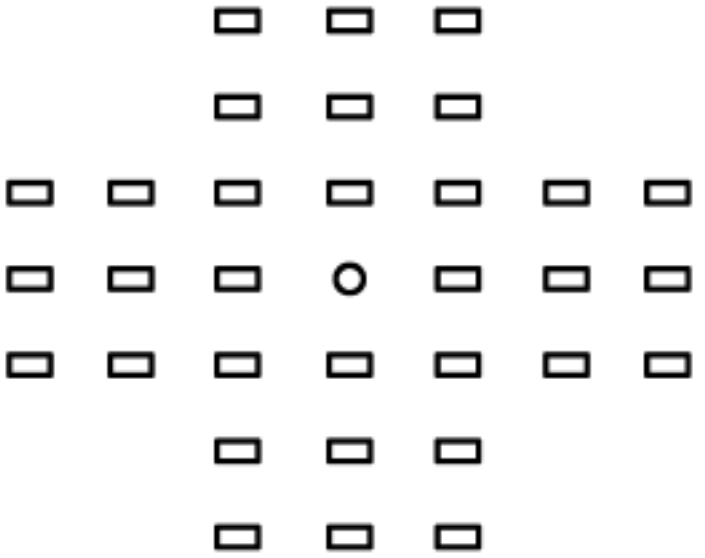


图 8.17 33 格上牌的一种摆法

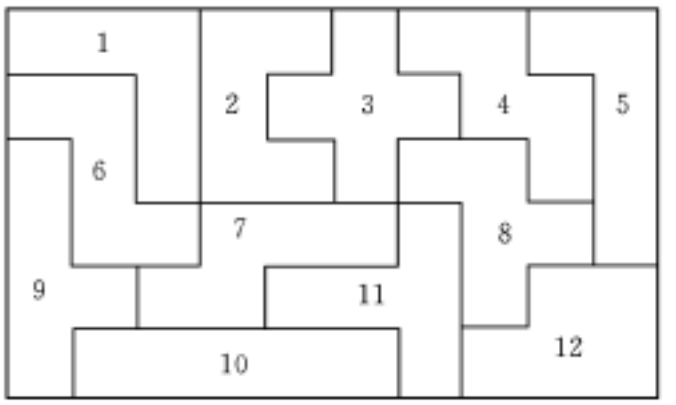


图 8.18 6×10 长方形的一种摆法

8.21 假定要将一组电子元件安装在线路板上，给定一个连线矩阵 CONN 和一个位置距离矩阵 DIST，其中，CONN(i,j)等于元件 i 和元件 j 间的连线数目，DIST(r,s)是此线路板上位置 r 和位置 s 间的距离。将这 n 个元件各自放在线路板的某位置上就构成一种布线方案，布线成本是 CONN(i,j) * DIST(r,s)乘积之和，其中元件 i 放在位置 r，元件 j 放在位置 s。设计一个算法找出使布线总成本取最小值的一种布线方案。

8.22 假设有 n 个要执行的作业但只有 k 个可以并行工作的处理器，作业 i 用 t_i 时间即可完成。写一个算法确定哪些作业按什么次序在哪些处理器上运行使完成全部作业的最后时间取最小值。

8.23 定义下列术语：状态空间、显式约束、隐式约束、问题状态、解状态、答案状态、静态树、动态树、活结点、E-结点、死结点和限界函数。

第 9 章

分枝-限界法

9.1 一般方法

本章要经常用 8.1 节所定义的术语,因此要求读者在阅读本章内容前复习 8.1 节的内容。

在图的检索方法中,BFS 和 D-检索这两种方法都是在对当前 E-结点检测完毕之后才检测以队或栈的形式存放在活结点表中的其它结点。将这两种方法一般化就成为分枝-限界策略。分枝-限界法是在生成当前 E-结点全部儿子之后再生成其它活结点的儿子且用限界函数帮助避免生成不包含答案结点子树的状态空间的检索方法。在这总的原则下,根据对状态空间树中结点检索次序的不同又可将分枝-限界设计策略分为数种不同的检索方法,其中与 BFS 类似的状态空间检索称为 FIFO(first in first out)检索,它的活结点表采用一张先进先出表(即队);类似于 D-检索的状态空间检索称为 LIFO(last in first out)检索,它的活结点表是一张后进先出表(即栈)。

例 9.1 [4-皇后问题]本例考察用一个 FIFO 分枝-限界算法检索 4-皇后问题的状态空间树(图 8.2)的基本过程。起初,只有一个活结点,即结点 1。这表示没有皇后被放在棋盘上。扩展这个结点,生成它的儿子结点 2,18,34 和 50。这些结点分别表示皇后 1 在第 1 行的 1,2,3,4 列情况下的棋盘。现在仅有的活结点是 2,18,34 和 50。如果按这样的次序生成了这些结点,则下一个 E-结点就是结点 2。扩展结点 2,生成结点 3,8 和 13。利用例 8.5 的限界函数,结点 3 立即被杀死,将结点 8 和 13 加到活结点队列。结点 18 变成下一个 E-结点,生成结点 19,24 和 29,限界函数杀死结点 19 和 24,结点 29 被加到活结点队列。下一个 E-结点是 34。图 9.1 显示了由 FIFO 分枝-限界检索生成图 8.2 中那棵树的一部分。由限界

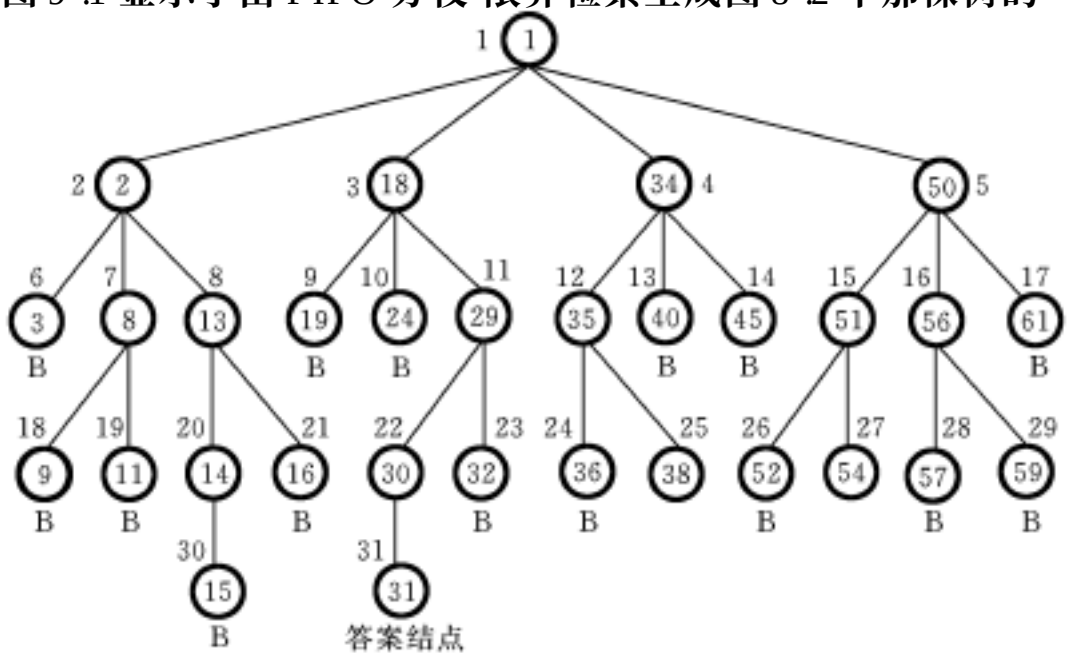


图 9.1 由 FIFO 分枝-限界法生成的一部分 4-皇后状态空间树

函数杀死的那些结点的下方有一个 B 字。结点内的数与图 8 .2 所示的结点内的数对应。结点外的数给出了用 FIFO 分枝-限界法生成结点的次序。在到达答案结点 31 时,仅剩下活结点 38(它可导致另一答案结点 39)和 54。比较图 8 .6 和图 9 .1 可以看出,对于这个问题而言回溯法占优势。

9 .1 .1 LC-检索

在 LIFO 和 FIFO 分枝-限界法中,对下一个 E-结点的选择规则相当死板,而且在某种意义上是“盲目的”。这种选择规则对于有可能快速检索到一个答案结点的结点没有给出任何优先权。因此,在例 9 .1 中,尽管在生成结点 30 后,这个结点显然只要走一步就可到达答案结点 31,但死板的 FIFO 规则却要求先扩展已生成的所有其它活结点。

对活结点使用一个“有智力的”排序函数 c(·)来选取下一个 E-结点往往可以加快到达一答案结点的检索速度。在 4-皇后的例子中,如果用一个能使结点 30 比所有其它活结点分得更优次序的排序函数,那么结点 30 就会在结点 29 之后成为 E-结点;接着扩展 E-结点就生成答案结点 31。因此,那些剩下的活结点则无需再变成 E-结点。

要给可能导致答案结点的活结点赋以优先次序,必然要附加若干计算工作,即要付出代价。对于任一结点 X,要付出的代价可以使用两种标准来度量:

(1) 在生成一个答案结点之前,子树 X 需要生成的结点数;

(2) 在子树 X 中离 X 最近的那个答案结点到 X 的路径长度。

使用后一种度量,图 9 .1 中树的根结点付出的代价是 4(结点 31 在树的第 5 级)。结点(18 和 34),(29 和 35)以及(30 和 38)的代价分别是 3,2 和 1。所有在 2,3 和 4 级上剩余结点的代价应分别大于 3,2 和 1。以这些代价作为选择下一个 E-结点的依据,则 E-结点依次为 1,18,29 和 30。得以生成的其它结点仅是 2,34,50,19,24,32 和 31。易于看出,如果使用度量(1),则对于每一种分枝-限界算法,总是生成最小数目的结点。如果使用度量(2),则要成为 E-结点的结点只是由根到最近的那个答案结点路径上的那些结点。以后用 c(·)表示“有智力的”排序函数,又称为结点成本函数。它的定义如下:如果 X 是答案结点,则 c(X)是由状态空间树的根结点到 X 的成本(即所用的代价,它可以是级数、计算复杂度等);如果 X 不是答案结点且子树 X 不包含任何答案结点,则 c(X) = ∞;否则 c(X)等于子树 X 中具有最小成本的答案结点的成本。但要指出的是,要得到结点成本函数 c(·)所用的计算工作量与解原问题具有相同的复杂度,这是因为计算一个结点的代价通常要检索包含一个答案结点的子树 X 才能确定,而这正是解决此问题所要作的检索工作,因此要得到精确的成本函数一般是不现实的。在算法中检测活结点的次序通常根据能大致估计结点成本的函数$\hat{g}$(·)来排出。

设$\hat{g}$(X)是由 X 到达一个答案结点所需做的附加工作的估计函数。只用$\hat{g}$(X)来给结点排序是否合适呢?只单纯使用函数$\hat{g}$(X)并不合适,因为这样会导致算法偏向于作纵深检查。为了看出这一点,不妨设结点 X 是当前的 E-结点且它的儿子为 Y,由于通常要求$\hat{g}$(Y) ≤$\hat{g}$(X),因此,活结点表中其它结点的成本估计值均大于$\hat{g}$(Y),于是 Y 将在 X 之后变成 E-结点;然后在 Y 的儿子们中有一个变成 E-结点;接着是 Y 的一个孙子变成 E-结点等等,直到子树 X 全部检索完毕才可能生成那些除 X 子树以外的子树结点。如果$\hat{g}$(X)就是c(X),

这种纵深检索正是所希望的,因为这样可以用最小成本到达离根最近的答案结点,其它子树的结点无需生成。但遗憾的是$\hat{g}(X)$仅是精确成本的一个估计值,因此偏向于纵深检查可能导致不能很快找到更靠近根的答案结点。例如,对于结点 W 和 Z 完全可能有以下情况,$\hat{g}(W) < \hat{g}(Z)$且 Z 比 W 更接近答案结点。此时,若单纯使用$\hat{g}(\cdot)$给结点排序,必然导致对 W 子树作纵深检查,结果显然是不理想的。为了不使算法过分偏向于作纵深检查,需改进成本估计函数,使其不只是考虑结点 X 到一个答案结点的估计成本,还应考虑由根结点到结点 X 的成本 h(X)。令这新的成本估计函数为$\hat{c}(\cdot)$,若用$\hat{c}(\cdot)$给结点 X 分配次序,则该函数使得$\hat{c}(X) = f(h(X)) + \hat{g}(X)$。显然,要求 f(·)是一个非降函数。用 f(·)á 0 可以减少算法作偏向于纵深检查的可能性,它强使算法在结点 W 和 Z 之间优先检索更靠近答案结点但又离根较近的结点 Z。

用成本估计函数$\hat{c}(X) = f(h(X)) + \hat{g}(X)$选择下一个 E-结点的检索策略总是选取$\hat{c}(\cdot)$值最小的活结点作为下一个 E-结点。因此,这种检索策略称之为最小成本检索,简称 LC-检索(Least Cost search)。值得指出的是,BFS 和 D-检索都是 LC-检索的特殊情况。如果使用$\hat{g}(X) \equiv 0$和 f(h(X)) = 结点 X 的级数,则 LC-检索依据级数来生成结点。这实质上与 BFS 检索一样。如果 f(h(X))≡0 且每当 Y 是 X 的一个儿子时总有$\hat{g}(X) \geq \hat{g}(Y)$,则这检索实质上是 D-检索。伴之有限界函数的 LC-检索称为 LC 分枝-限界检索。

9.1.2　15-谜问题——一个例子

15-谜问题叙述如下:在一个分成 16 格的方形棋盘上放有 15 块编了号码的牌(见图 9.2)。对这些牌给定一种初始排列图 9.2(a),要求通过一系列的合法移动将这一初始排列转换成图 9.2(b)所示的那样的目标排列(只有将邻接于空格的牌移到空格才是合法的移动)。于是,像图 9.2(a)所示的那样的初始排列有 4 种可能的移动,可以将编号为 2,3,5 或 6 的任何一块牌移到空格。在作了这次移动之后,可作其它的移动。每移动一次,产生一种

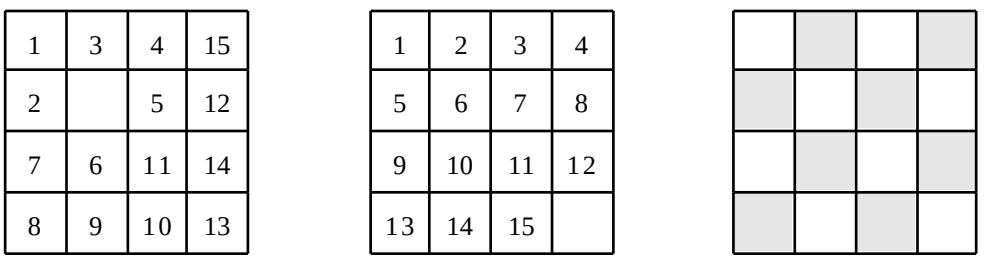


图 9.2　15-谜的一些排列

新的排列。这些排列称作这个谜问题的状态。初始排列和目标排列叫做初始状态和目标状态。若由初始状态到某状态存在一系列合法的移动,则称该状态可由初始状态到达。一种初始状态的状态空间由所有可从初始状态到达的状态构成。易于看出在棋盘上这些牌有 16！≈20.9×10^{12}种不同的排列。对于任一给定的初始状态,它可到达的状态为这些排列中的一半,由此可知这个问题的状态空间是相当庞大的,因此有必要在具体求解谜问题之前判定目标状态是否在这个初始状态的状态空间中。对此有一种非常简单的判定方法。我们先给棋盘的方格位置编上 1～16 的号码。位置 i 是在图 9.2(b)所示的目标排列中放 i 号牌的

方格位置,位置 16 是空格的位置。假设 POSITION(i)是编号为 i 的那块牌在初始状态下的位置号,1≤i<16;POSITION(16)表示空格的位置。对于任意一种状态,设 LESS(i)是使牌 j 小于牌 i 且 POSITION(j) > POSITION(i)的数目。例如,对于图 9.2(a)所示的状态,有 LESS(1) = 0,LESS(4) = 1 和 LESS(12) = 6。在初始状态下,如果空格在图 9.2(c)的阴影位置中的某一格处,则令 X = 1;否则令 X = 0。于是有定理 9.1。

定理 9.1 当且仅当 $\sum_{i=1}^{16}$ LESS(i) + X 是偶数时,图 9.2(b)所示的目标状态可由此初始状态到达。

证明留作习题。

定理 9.1 用来判定目标状态是否在这个初始状态的状态空间之中。若在,就可着手确定导致目标状态的一系列移动。为了实现这一检索,可以将此状态空间构造成一棵树。在这棵树中,每个结点的儿子表示由状态 X 通过一次合法的移动可到达的状态。不难看出,移动牌与移动空格实质上是等效的,而且在作实际移动时更为直观,因此以后都将父状态到子状态的一次转换看成是空格的一次合法移动。图 9.3(a)中树的根结点表示 15-谜问题一个实例的初始状态,该图给出了此实例所构造状态空间树的前三级和第四级的一部分。图中已对这棵树作了一些修剪,即如果结点 P 的儿子中有和 P 的父亲状态重复的,则将这一枝剪去。对此实例作 FIFO 检索表现为依图 9.3(a)中结点编号的顺序来生成这些结点。可以看出第四级有一个答案结点,由于是宽度优先检索,因此它还是离根最近的答案结点。图 9.3(b)给出了对此实例按深度优先生成其状态空间树结点的一部分,从图中一连串的棋盘格局可以看出,这种检索方法不管开始格局如何(即不管问题的具体实例),总是采取由根开始的那条最左路径,而在这条最左路径上的每一次移动不是离目标更近了而是更远了。这种检索是呆板而盲目的。尽管上面使用的宽度优先检索可以找到离根最近的答案结点,但从处理方式看也是不管开始格局如何总是按千篇一律的顺序移动,因此在这种意义下它也是呆板和盲目的。

所希望的是,一种能按不同具体实例作不同处理的有一定“智能”的检索方法。这种检索方法需给状态空间树的每个结点 X 赋予一定的成本 c(X)。如果具体实例有解,则将由根出发到最近目标结点路径上的每个结点赋以这条路径的长度作为它们的成本。于是,在图 9.3(a)所示实例中,c(1) = c(4) = c(10) = c(23) = 3,其余结点均赋以∞的成本。如果能做到这一点,在使用宽度优先检索的情况下必然会实现非常有效的检索。把根作为 E-结点开始,在生成它的儿子结点时,可以将成本为∞的结点统统杀掉,只有与根具有相同成本值的儿子结点 4 成为活结点,而且它立即成为下一个 E-结点。按这种检索策略继续处理很快就可到达目标结点 23。但这是一种很不实际的策略,因为要想简单地给出能得到像上面那样成本值的函数 c(·)是不可能的。

切合实际的做法是给出一个便于计算成本估计值的函数 $\hat{c}(X) = f(X) + \hat{g}(X)$,其中 f(X) 是由根到结点 X 路径的长度,$\hat{g}(X)$ 是以 X 为根的子树中由 X 到目标状态的一条最短路径长度的估计值。为此,这个 $\hat{g}(X)$ 至少应是能把状态 X 转换成目标状态所需的最小移动数。对它的一种可能的选择是

$$\hat{g}(X) = \text{不在其目标位置的非空白牌数目}$$

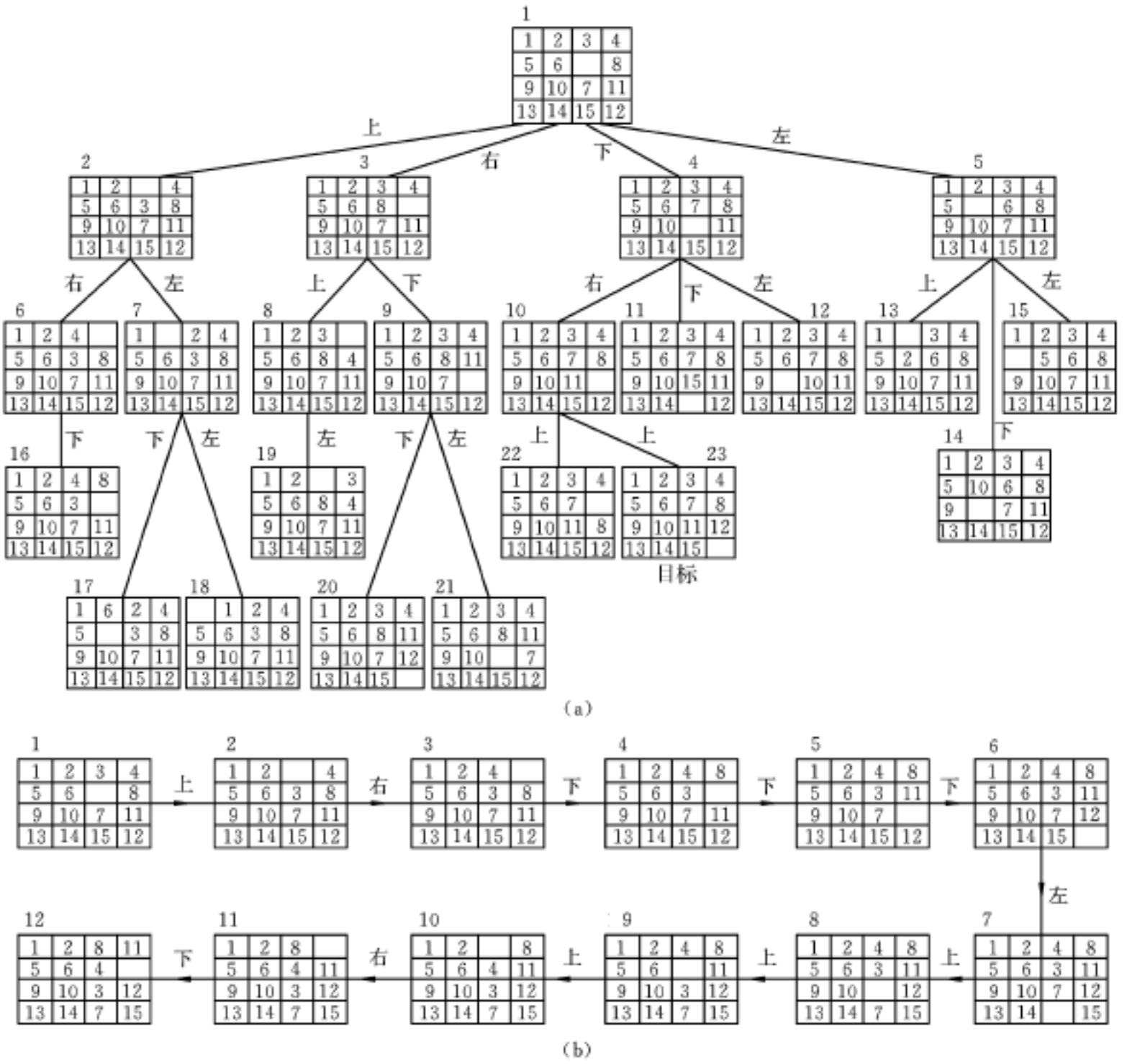


图 9 .3　15-谜问题的实例及深度优先检索

(a) 15-谜问题的一部分状态空间树;(b) 一种深度优先检索的前十步

这样定义的$\hat{g}(X)$是符合以上要求的。不难看出,为达到目标状态所需要的移动数可能大于$\hat{g}(X)$。例如对图 9 .4 的问题状态,由于只有 7 号牌不在其目标位置上,因此$\hat{g}(X) = 1$($\hat{g}(X)$的计数排除了空白牌)。然而,为了达到目标状态所需要的移动数比$\hat{g}(X)$多得多。由此可以看出,$\hat{c}(X)$是 c(X)的下界。

使用$\hat{c}(X)$图 9 .3(a)的 LC-检索将结点 1 作为 E-结点的开始。结点 1 在生成它的所有儿子结点 2,3,4 和 5 之后死去。变成 E-结点的下一个结点是具有最小$\hat{c}(X)$的活结点,$\hat{c}(2) = 1 + 4$,$\hat{c}(3) = 1 + 4$,$\hat{c}(4) = 1 + 2$ 和$\hat{c}(5) = 1 + 4$,结点 4 成为 E-结点。生成它的儿子结点,此时的活结点是 2,3,5,10,11 和 12。$\hat{c}(10) = 2 + 1$,$\hat{c}(11) = 2 + 3$,$\hat{c}(12) = 2 + 3$,具有最小$\hat{c}$值的活结点 10 成为下一个 E-结点。接着生成结点 22 和 23,结点 23 被判定是目标结点,此次检索结束。在这种情况下,LC-检索几乎

1	2	3	4
5	6		8
9	10	11	12
13	14	15	7

图 9 .4　问题状态

和使用精确函数 c(·)一样有效。由此可以看出,通过对 c(·)的适当选择,LC-检索的选择性将远比已讨论过的其它检索方法强得多。

9.1.3 LC-检索的抽象化控制

设 T 是一棵状态空间树,c(·)是 T 中结点的成本函数。如果 X 是 T 中的一个结点,则 c(X)是其根为 X 的子树中任一答案结点的最小成本。从而,c(T)是 T 中最小成本答案结点的成本。如前所述,要找到一个如上定义且易于计算的函数 c(·)通常是不可能的,为此使用一个对 c(·)估值的启发性函数$\hat{c}$(·)来代替。这个启发函数应是易于计算的并且一般有如下性质,如果 X 是一个答案结点或者是一个叶结点,则 c(X) = $\hat{c}$(X)。过程 LC(算法 9.1)用$\hat{c}$去找寻一个答案结点。这个算法用了两个子算法 LEAST(X)和 ADD(X),它们分别将一个活结点从活结点表中删去或加入。LEAST(X)找一个具有最小的$\hat{c}$值的活结点,从活结点表中删除这个结点,并将此结点放在变量 X 中返回。ADD(X)将新的活结点 X 加到活结点表。通常把这个活结点表作成一个 min-堆来使用。过程 LC 输出找到的答案结点到根结点 T 的那条路径。如果使用 PARENT 信息段将活结点 X 与它的父亲相链接,这条路径就很容易输出了。

算法 9.1 LC-检索

```
line procedure LC(T,ĉ)
     //为找一个答案结点检索 T//
0    if T 是答案结点 then 输出 T;return endif
1     E←T //E-结点//
2    将活结点表初始化为空
3    loop
4          for E 的每个儿子 X do
5            if X 是答案结点 then 输出从 X 到 T 的那条路径
6                  return
7            endif
8            call ADD(X)   //X 是新的活结点//
9            PARENT(X)←E   //指示到根的路径//
10            repeat
11    if 不再有活结点  then
        print ('no answer node')
12      stop
13    endif
14    call LEAST(E)
15    repeat
16    end LC
```

下面证明算法 LC 的正确性。变量 E 总是指着当前的 E-结点。由 LC-检索的定义,根结点是第一个 E-结点(第 1 行)。第 2 行将活结点表置初值。在执行 LC 的任何时刻,这个表含有除了 E-结点以外的所有活结点,因此这个表最初为空(第 2 行)。第 4～10 行的 for 循环检查 E-结点的所有儿子。如果有一个儿子是答案结点,则算法输出由 X 到 T 的那条路

径并且终止。如果 E 的某个儿子不是答案结点,则成为一个活结点,将它加到活结点表(第 8 行)中且将其 PARENT 信息段置 E。当生成了 E 的全部儿子时,E 变成死结点,控制到达第 11 行。这种情况只有在 E 的所有儿子都不是答案结点时才会发生,于是检索应更深入地继续进行。在没有活结点剩下的情况下,这整棵状态空间树就被检索完毕,且没有找到答案结点,算法在第 12 行结束。反之,则通过 LEAST(X)按规定去正确地选择下一个 E-结点,并从这里继续进行检索。

根据以上讨论,显然 LC 只有在找到一个答案结点或者在生成并检索了这整棵状态空间树时才会终止。因此,只有在有限状态空间树下才能保证 LC 终止。对于无限状态空间树,在其至少有一个答案结点并假定对成本估计函数$\hat{c}(\cdot)$能作出"适当"的选择时也能保证算法 LC 终止。例如,对于如下的每一对结点 X 和 Y,在 X 的级数"足够地"大于 Y 的级数时,有$\hat{c}(X) > \hat{c}(Y)$,这样的成本估计函数就能对结点作适当的选择。对于没有答案结点的无限状态空间树,LC 不会终止。因此,将检索局限在寻找估计成本不大于某个给定的限界 C 的答案结点则是可取的。

实际上 LC 算法与状态空间树的宽度优先检索算法和 D-检索算法基本相同。如果活结点表作为一个队列来实现,用 LEASL(X)和 ADD(X)算法从队列中删去或加入元素,则 LC 就转换成 FIFO 检索。如果活结点表作为一个栈来实现,用 LEAST(X)和 ADD(X)算法从栈中删去或加入元素,则 LC 就转换成 LIFO 检索。唯一的不同之处在于活结点表的构造上,即仅在于得到下一个 E-结点所使用的选择规则不同。

9.1.4　LC-检索的特性

在许多应用中,希望在所有的答案结点中找到一个最小成本的答案结点。LC 是否一定找得到具有最小成本 c(G)的答案结点 G 呢?回答是否定的。考虑图 9.5 所示的状态空间树,方形叶子结点是答案结点。每个结点内有两个数,上面的数是 c 的值,下面的数是估计值$\hat{c}$。于是 c(根结点) = 10 而$\hat{c}$(根结点) = 0。LC 首先生成根的两个儿子,然后$\hat{c}$(　) = 2 的那个结点成为 E-结点。扩展这一结点就得到答案结点 G,它有$\hat{c}(G) = c(G) = 20$,算法终止。但是,具有最小成本的答案结点是 c(G) = 10 的结点。LC 没能达到这个最小成本答案结点的原因在于有两个这样的结点 X 和 Y,当其 c(X) > c(Y)时,$\hat{c}(X) < \hat{c}(Y)$。如果使每一对 c(X) < c(Y)的 X、Y 结点都有$\hat{c}(X) < \hat{c}(Y)$,那么就可以证明在一棵至少有一个答案结点的有限状态树中 LC 总会找到一个最小成本答案结点。

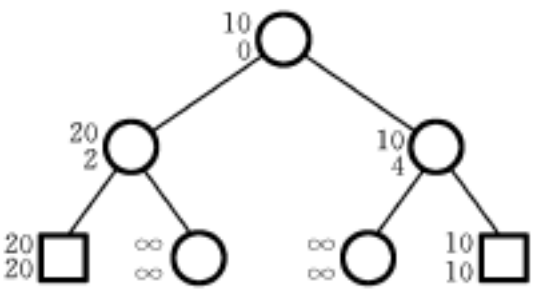


图 9.5　LC-检索

定理 9.2　在有限状态空间树 T 中,对于每一个结点 X,令$\hat{c}(X)$是 c(X)的估计值且具有以下性质:对于每一对结点 Y、Z,当且仅当 c(Y) < c(Z)时有$\hat{c}(Y) < \hat{c}(Z)$。那么在使$\hat{c}$(　)作为 c(　)的估计值时,算法 LC 到达一个最小的成本答案结点且终止。

证明　当 T 有限时,上面已经阐明如果 T 有一个答案结点,LC 就会找到它。因此,假定 LC 在一个 c(G) > c(G′)的答案结点 G 处终止,而 G′是一个最小成本答案结点。令 R 是 G 的这样一个最近的祖先,它使得子树 R 含有一个最小成本答案结点(见图 9.6)。假设 R,

$\alpha_1, \alpha_2, \cdots, \alpha_k, G'$是由 R 到 G′的路径，$R, \beta_1, \beta_2, \cdots, \beta, G$是由 R 到 G 的路径。由 R 的定义，$\alpha_1 \neq \beta_1$，且子树 β_1 不具有成本为 $c(G')$的答案结点。为使检索到达结点 G，R 必须在某一时刻变成 E-结点。此时，它的儿子(包括 α_1 和 β_1)成为活结点。由 $c(\cdot)$的定义，可以得出 $c(R) = c(\alpha_1) = c(\alpha_2) = \cdots = c(G')$和 $c(\beta_1), c(\beta_2), \cdots, c(G) > c(R)$。于是，由 $\hat{c}(\cdot)$的条件，可以得出 $\hat{c}(\alpha_1), \hat{c}(\alpha_2), \cdots, \hat{c}(\alpha_k) < \hat{c}(\beta_1)$，因此在 $\alpha_i, 1 \leq i \leq k$，变成 E-结点并到达 G′以前 β_1 不能变成 E-结点。证毕。

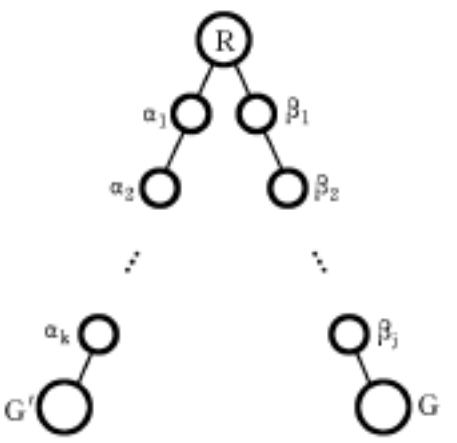


图 9.6 状态空间树

这个定理易于推广到其每个结点的度都是有限的无限状态空间树的情况。但是要得到满足定理 9.2 的要求又易于计算的 $\hat{c}(\cdot)$通常是不可能的。一般只可能找到一个易于计算且具有如下特性的 $\hat{c}(\cdot)$，对于每一个结点 X，$\hat{c}(X) \leq c(X)$。在这种情况下，算法 LC 不一定能找到最小成本答案结点(见图 9.5)。如果对于每一个结点 X 有 $\hat{c}(X) \leq c(X)$且对于答案结点 X 有 $\hat{c}(X) = c(X)$，只要对 LC 稍作修改就可得到一个在达到某个最小成本答案结点时终止的检索算法。在这个改进型的算法中，检索一直继续到一个答案结点变成 E-结点为止。这个新算法是 LC1(算法 9.2)。

算法 9.2 找最小成本答案结点的 LC-检索

```
line procedure LC1(T, ĉ)
    // 为找出最小成本答案结点检索 T //
1     E←T   // 第一个 E-结点 //
2     置活结点表为空
3     loop
4       if E 是答案结点  then 输出从 E 到 T 的路径
5                             return
6     endif
7     for E 的每个儿子 X do
8       call ADD(X); PARENT(X)←E
9   repeat
10   if 不再有活结点 then print ('no answer node')
11                          stop
12   endif
13   call LEAST(E)
14  repeat
15 end LC1
```

定理 9.3 令 $\hat{c}(\cdot)$是满足如下条件的函数，在状态空间树 T 中，对于每一个结点 X，有 $\hat{c}(X) \leq c(X)$，而对于 T 中的每一个答案结点 X，有 $\hat{c}(X) = c(X)$。如果算法在第 5 行终止，则所找到的答案结点是具有最小成本的答案结点。

证明 此时，E-结点 E 是答案结点，对于活结点表中的每一个结点 L，$\hat{c}(E) \leq \hat{c}(L)$。由假设 $\hat{c}(E) = c(E)$且对于每一个活结点 L，$\hat{c}(L) \leq c(L)$。因此 $c(E) \leq c(L)$，从而 E 是一个最小成本答案结点。证毕。

9.1.5　分枝-限界算法

检索状态空间树的各种分枝-限界方法都是在生成当前 E-结点的所有儿子之后再将另一结点变成 E-结点。假定每个答案结点 X 有一个与其相联系的 c(X),并且假定会找到最小成本的答案结点。使用一个使得$\hat{c}(X)\leq c(X)$的成本估计函数$\hat{c}(\cdot)$来给出可由任一结点 X 得出的解的下界。采用下界函数使算法具有一定的智能,减少了盲目性,另外还可通过设置最小成本的上界使算法进一步加速。如果 U 是最小成本解的成本上界,则具有$\hat{c}(X) > U$ 的所有活结点 X 可以被杀死,这是因为由 X 可以到达的所有答案结点有$c(X)\geq\hat{c}(X) > U$。在已经到达一个具有成本 U 的答案结点的情况下,那些有$\hat{c}(X)\geq U$ 的所有活结点都可以被杀死。U 的初始值可以用某种启发性方法得到,也可置成∞。显然,只要 U 的初始值不小于最小成本答案结点的成本,上述杀死活结点的规则不会去杀死可以到达最小成本答案结点的活结点。每当找到一个新的答案结点就可以修改 U 的值。

现在讨论如何根据上述思想来得到解最优化问题的分枝-限界算法。以下只考虑极小化问题,极大化问题可通过改变目标函数的符号很容易地转换成极小化问题。为了找到最优解需要将最优解的检索表示成对状态空间树答案结点的检索,这就要求定义的成本函数满足代表最优解的答案结点的 c(X)是所有结点成本的最小值。最简单的方法是直接把目标函数作为成本函数 c(·)。在这种定义下代表可行解的结点 c(X)就是那个可行解的目标函数值;代表不可行解的结点有 c(X) = ∞;而代表部分解的结点 c(X)是根为 X 的子树中最小成本结点的成本。由此可知在状态空间树中每个答案结点(当然它必定是解结点)都对应一个可行解,只有成本最小的答案结点才与最优解对应,因而,在这类问题的状态空间树中答案结点与解结点是不相区别的。另外,由于计算 c(X)一般和求解最优化问题一样困难,因此,分枝-限界算法采用一个对于所有的 X 有$\hat{c}(X)\leq c(X)$的估值函数$\hat{c}(\cdot)$。要特别指出的是,在解最优化问题时所使用的$\hat{c}(\cdot)$不是用来估计到达一个答案结点在计算方面的难易程度,而是对目标函数进行估计。

作为最优化问题的一个例子,考虑 5.4 节所引入的带限期的作业排序问题。将此问题一般化,允许作业有不同的处理时间。假定有 n 个作业和一台处理机,每个作业 i 与一个三元组(p_i, d_i, t_i)相联系,它要求 t_i 个单位处理时间,如果在期限 d_j 内没处理完则要招致 p_i 的罚款。问题的目标是从这 n 个作业中选取一个子集合 J,要求在 J 中的作业都能在相应的期限内完成且使不在 J 中的作业招致的罚款总额最小。这样的 J 就是最优解。

考虑下面的实例:n = 4;$(p_1, d_1, t_1) = (5,1,1)$;$(p_2, d_2, t_2) = (10,3,2)$;$(p_3, d_3, t_3) = (6,2,1)$;$(p_4, d_4, t_4) = (3,1,1)$。此实例的解空间由作业指标集{1,2,3,4}的所有可能的子集合组成。使用子集和数问题(例 8.2)两种表示的任何一种都可以将这解空间构造成一棵树。图 9.7 所示为元组大小可变的状态空间树,而图 9.8 所示为元组大小固定的状态空间树。在这两棵树中,方形结点代表不可行的子集合。图 9.7 中所有的圆形结点都是答案结点。结点 9 代表最优解并且是仅有的最小成本答案结点。对于这个结点,J = {2,3}罚款值(即成本)为 8。在图 9.8 中,只有圆形叶结点才是答案结点。结点 25 代表最优解,它也是仅有的最小成本答案结点。这个结点对应于 J = {2,3}和 8 这么大的罚款。图 9.8 中各答案结点的罚款数标在这些结点的下面。

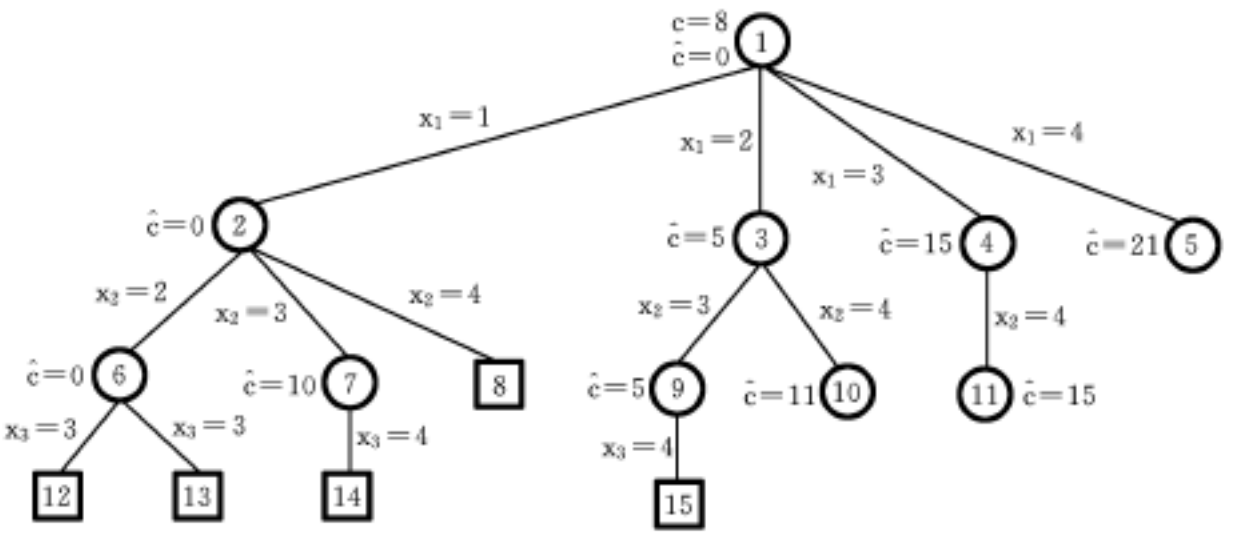


图 9.7　大小可变的元组表示的状态空间树

对图 9.7 和图 9.8 这两种状态空间表示可将成本函数 c(·)定义为,对于圆形结点 X,c(X)是根为 X 的子树中结点的最小罚款;对于方形结点,c(X) = ∞。在图 9.7 的树中,c(3) = 8,c(2) = 9,c(1) = 8。在图 9.8 的树中,c(1) = 8,c(2) = 9,c(5) = 13,c(6) = 8。显然,c(1)是最优解 J 对应的罚款。

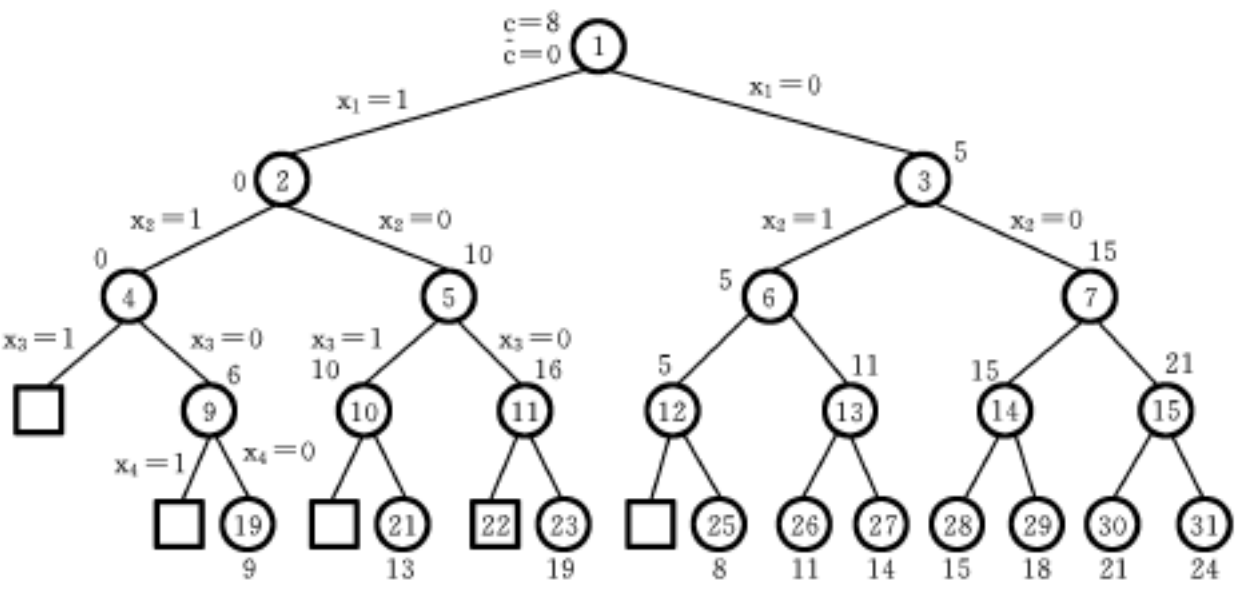


图 9.8　大小固定的元组表示的状态空间树

对于所有的 X 容易得出这样一个下界函数$\hat{c}(\cdot)$,它使得$\hat{c}(X) \le c(X)$。设 S_X 是在结点 X 对 J 所选择的作业的子集,如果 $m = \max\{i \mid i \in S_X\}$,则$\hat{c}(X) = \sum_{i<m,\ i \mid S_X} p_i$ 是使 c(X)有$\hat{c}(X) \le c(X)$的估计值。图 9.7 和图 9.8 中每个圆形结点 X 外面的数是该结点的$\hat{c}(X)$值,而方形结点$\hat{c}(X) = \infty$子树 X 中最小成本答案结点的成本的一个简单上界 u(X)可定义为$u(X) = \sum_{i \mid S_X} p_i$。注意,对带限期的作业排序问题而言,u(X)是对应结点 X 的解 S_X 的成本值。

作业排序问题的一个 FIFO 分枝-限界算法开始时可以将最小成本答案结点的成本上界取为 U = ∞或 $U = \sum_{1 \le i \le m} p_j$。假定使用图 9.7 这种大小可变的元组表示,开始时结点 1 作为 E-结点,于是依次生成结点 2,3,4 和 5。由于 u(2) = 19,u(3) = 14,u(4) = 18,u(5) = 21,因此在生成结点 3 时就将上界 U 修改成 14。因为$\hat{c}(4)$和$\hat{c}(5)$大于 U,所以结点 4 和 5 被杀死(即限界)。结点 2 成为下一个 E-结点,生成它的儿子 6,7,8。u(6) = 9,因此 U 修改成 9;

$\hat{c}(7)=10>U$,所以 7 被杀死;结点 8 是不可行结点,也被杀死。接着,结点 3 成为 E-结点,生成其儿子结点 9 和 10。$u(9)=8$,因此 U 变成 8;$\hat{c}(10)=11>U$,所以 10 被杀死。下一个 E-结点是 6,它的两个儿子均不可行。结点 9 只有一个儿子且不可行,因此 9 是最小成本答案结点,它的成本值为 8。

在实现 FIFO 分枝-限界算法时,每修改一次 U,在活结点队中那些有$\hat{c}(X)>U$或者在 U 是已找到的一个成本值情况下有$\hat{c}(X)\geq U$的结点应被杀死,但由于活结点按其生成次序放在活结点队中,因此,队中符合可杀条件的结点是随机分布的,所以每修改一次 U 就从活结点表中找出并杀死这些结点是很不经济的。一种比较经济的方法是直到可杀结点要变成 E-结点时才将其杀掉。但不管采用哪种方法,都必须识别出这个修改了的 U 是一个已找到的解的成本还是一个不是解成本的单纯的上界(这个单纯上界 U 由以下任意一种情况得到,一是还没找到一个答案结点,U 由处理一可行结点时修改而得;一是虽然找到一答案结点,但它的成本值大于它的上界值,说明这答案结点的子孙中还有成本更小的答案结点,U 取这上界值)。这样就可以决定在$\hat{c}(X)=U$的情况下是否杀死结点 X,即若 U 为前者则杀死 X,若为后者,那么 X 是有希望导致成本值等于 U 的解的结点,于是应将 X 变成 E-结点。在算法实现时,可引进一个很小的正常数 ε 来进行这一识别。此 ε 要取得足够小,使得对于任意两个可行结点 X 和 Y,如果 $u(X)<u(Y)$,则 $u(X)<u(X)+\varepsilon<u(Y)$。当 U 是由一答案结点的成本值而得时,U 就是这个成本值,而当 U 是由一单纯上界得到时,U 等于此上界值 u(X)加 ε。

过程 FIFOBB 是采用了以上处理措施的 FIFO 分枝-限界算法的粗略描述。它用了两个子算法 ADDQ(X)和 DELETEQ(X)分别将一个结点加入队或从队中删去。对于状态空间树中的每一个答案结点 X,cost(X)是结点 X 对应的解的成本。FIFOBB 假设不可行结点的估计值$\hat{c}(X)=\infty$,可行结点的估计值$\hat{c}(X)\leq c(X)\leq u(X)$。

算法 9.3　找最小成本答案结点的 FIFO 分枝-限界算法

```
line procedure FIFOBB(T,ĉ,u,ε,cost)
    //为找出最小成本答案结点检索 T。假定 T 至少包含一个解结点(即答案结点)且
      ĉ(X)≤c(X)≤u(X)//
1     E←T;PARENT(E)←0;
2     if T 是解结点 then U←min (cost(T),u(T)+ε);ans←T
3                    else U←u(T)+ε;ans←0
4     endif
5     将队置初值为空
6     loop
7       for E 的每个儿子 X do
8         if ĉ(X)<U then call ADDQ(X);PARENT(X)←E
9                        case
10                         :X 是解结点 and cost(X)<U:
11                           U←min (cost(X),u(X)+ε)
12                           ans←X
13                         :u(X)+ε<U:U←u(X)+ε
14                       endcase
```

```
15            endif
16          repeat
17          loop   //得到下一个 E-结点//
18            if 队为空 then print ('least cost = ',U)
19              while ans≠0 do
20                print (ans)
21                ans←PARENT(ans)
22              repeat
23              return
24            endif
25            call DELETEQ(X)
26            if ĉ(X) < U then exit   //杀死ĉ(X)≥U 的结点//
27            repeat
28          repeat
29      end FIFOBB
```

LC 分枝-限界的抽象化控制是 LCBB,它与 FIFOBB 有相同的假设。ADD 和 LEAST 分别加一个结点到 min-堆或从 min-堆中删去。LCBB 在堆中没有活结点或下一个 E-结点 E 有$\hat{c}(E)\geq U$的情况下终止。

算法 9.4　找最小成本结点的 LC 分枝-限界算法

```
line procedure LCBB(T,ĉ,u,ε,cost)
    //为找出最小成本结点检索 T。假定 T 至少包含一个解结点且ĉ(X)≤c(X)≤u(X)//
1   E←T;PARENT(E)←0
2   if T 是解结点 then U←min(cost(T),u(T)+ε);ans←T
3                  else U←u(T)+ε;ans←0
4   endif
5   将活结点表初始化为空
6   loop
7     for E 的每个儿子 X do
8       if ĉ(X) < U then call ADD(X)
9                         PARENT(X)←E
10                        case
11                        :X 是解结点 and cost(X) < U:
12                        U←min(cost(X),u(X)+ε)
13                        ans←X
14                        :u(X)+ε< U:U←u(X)+ε
15                        endcase
16        endif
17      repeat
18      if 不再有活结点 or 下一个 E-结点有ĉ≥U
19              then print ('least cost = ',U)
20                   while ans≠0 do
21                     print (ans)
```

```
22                    ans←PARENT(ans)
23                repeat
24                return
25          endif
26          call LEAST(E)
27       repeat
28    end LCBB
```

9.1.6　效率分析

由以上讨论可知,上、下界函数选择的好坏是决定极小化问题的各种分枝-限界算法效率的主要因素,但它们对算法效率究竟有多大影响呢?这一点正是本节最后所要讨论的问题。关于上、下界函数的选择一般可提出以下问题:

(1) 对 U 选择一个更好的初值是否能减少所生成的结点数?

(2) 扩展一些$\hat{c}(\quad) > U$ 的结点是否能减少所生成的结点数?

(3) 假定有两个成本估计函数$\hat{c}_1(\cdot)$和$\hat{c}_2(\cdot)$,对于状态空间树的每一个结点 X,若有$\hat{c}_1(X) \leq \hat{c}_2(X) \leq c(X)$,则称$\hat{c}_2(\cdot)$比$\hat{c}_1(\cdot)$好。是否用较好的成本估计函数$\hat{c}_2(\cdot)$比用$\hat{c}_1(\cdot)$生成的结点数要少呢?

对于以上问题,读者可凭直觉立即得出一些“是”或“非”的答案,但由下述定理可以看出有的结论可能刚好与你的直觉相反。假定下面出现的分枝-限界算法均用来求最小成本答案结点,$c(X)$是 X 子树中最小成本答案结点的成本。

定理 9.4　设 U_1 和 U_2 是状态空间树 T 中最小成本答案结点的两个初始上界且 $U_1 < U_2$。那么 FIFO, LIFO 和 LC 分枝-限界算法在以 U_1 为上界初始值时所生成的结点数不会多于以 U_2 为上界初始值时所生成的结点数。

证明留作习题。

定理 9.5　设 U 是状态空间树 T 最小成本答案结点的当前上界。由 FIFO, LIFO 和 LC 分枝-限界算法所生成的结点数不因扩展$\hat{c}(X) > U$ 的结点 X 而减少。

证明　因为$\hat{c}(X) > U$,所以扩展 X 不可能使 U 值减小,故扩展 X 不会影响算法在这棵树上其余部分的运算。证毕。

定理 9.6　在 FIFO 和 LIFO 分枝-限界算法中使用一个更好的成本估计函数$\hat{c}(\cdot)$不会增加其生成的结点数。

证明留作习题。

定理 9.7　在 LC 分枝-限界算法中使用一个更好的成本估计函数$\hat{c}(\cdot)$可能增加所生成的结点的个数。

证明　考虑图 9.9 所示的状态空间树,所有叶结点都是答案结点,叶结点下面的数是其成本值,从这些值中可以得出$c(1) = c(3) = 3, c(2) = 4$。结点 1,2 和 3 外面的数是对应的$\begin{bmatrix} \hat{c}_1 \\ \hat{c}_2 \end{bmatrix}$值,显然$\hat{c}_2$ 是比$\hat{c}_1$ 更好的成本估

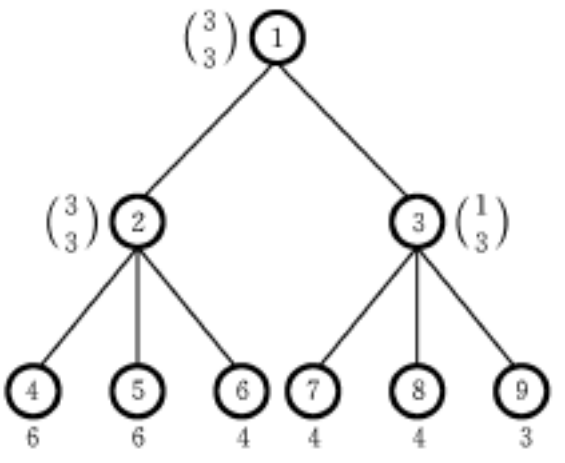


图 9.9　定理 9.7 的一个例子

计函数。如果使用$\hat{c}_2$,由于$\hat{c}_2(2)=\hat{c}_2(3)$,因此结点 2 会在结点 3 之前变成 E-结点,于是所有 9 个结点都将被生成,而使用$\hat{c}_1$将不会生成结点 4,5 和 6。证毕。

9.2 0/1 背包问题

为了用上一节所讨论的分枝-限界方法来求解 0/1 背包问题,可用函数 $-\sum p_i x_i$ 来代替目标函数 $\sum p_i x_i$,从而将背包问题由一个极大化问题转换成一个极小化问题。显然,当且仅当 $-\sum p_i x_i$ 取极小值时 $\sum p_i x_i$ 取极大值。这个修改后的背包问题可描述如下:

$$\begin{aligned} &\text{极小化} && -\sum_{i=1}^{n} p_i x_i \\ &\text{约束条件} && \sum_{i=1}^{n} w_i x_i \le M \\ & && x_i = 0 \text{ 或 } x_i = 1, 1 \le i \le n \end{aligned} \tag{9.1}$$

在 8.6 节曾介绍过 0/1 背包问题的两种状态空间树结构,这里只讨论在大小固定的元组表示下如何求解 0/1 背包问题,至于在元组大小可变时,如何求解背包问题,在此基础上是不难解决的。状态空间树中那些表示 $\sum_{1\le i\le n} w_i x_i \le M$ 的装包方案的每一个叶结点是答案结点,其它的叶结点均不可行。为了使最小成本答案结点与最优解相对应,需要对每一个答案结点 X 定义 $c(X) = -\sum_{1\le i\le n} p_i x_i$;对不可行的叶结点则定义 $c(X)=\infty$;对于非叶结点则将 c(X) 递归定义成 $\min\{c(\text{LCHILD}(X)), c(\text{RCHILD}(X))\}$。

还需要两个函数 $\hat{c}(\cdot)$ 和 $u(\cdot)$,使它们对于每个结点 X,有 $\hat{c}(X) \le c(X) \le u(X)$。这样的两个函数可由下法得到:设 X 是 j 级上的一个结点,$1 \le j \le n+1$。在结点 X 处已对前 j-1 种物品装包,这 j-1 种物品的装入情况为 $x_i, 1 \le i < j$。此时装包的成本为 $-\sum_{1\le i<j} p_i x_i$。因此,$c(X) \le -\sum_{1\le i<j} p_i x_i$,故可以取 $u(X) = -\sum_{1\le i<j} p_i x_i$。算法 9.5 是一个改进了的上界函数,对于 X 有 $u(X) = \text{UBOUND}\left(-\sum_{1\le i<j} p_i x_i, \sum_{1\le i<j} w_i x_i, j-1, M\right)$。显然,由于有 $-\text{BOUND}\left(\sum_{1\le i<j} p_i x_i, \sum_{1\le i<j} w_i x_i, j-1, M\right) \le c(X)$,这里 BOUND 是算法 8.11,故可得 $\hat{c}(X) = -\text{BOUND}\left(\sum_{1\le i<j} p_i x_i, \sum_{1\le i<j} w_i x_i, j-1, M\right)$。

算法 9.5 背包问题的上界函数 u(·)

```
procedure UBOUND(p,w,k,M)
    // p,w,k 和 M 与算法 8.11 的含意相同,W(i)和 P(i)分别是第 i 种物品的重量和效益值 //
    global W(1:n),P(1:n);integer i,k,n
    b←p;c←w
    for i←k+1 to n do
      if c+W(i)≤M then c←c+W(i);b←b-P(i) endif
    repeat
    return (b)
end UBOUND
```

9.2.1 LC 分枝-限界求解

例 9.2 [LCBB]考虑背包问题:n=4;$(p_1,p_2,p_3,p_4)=(10,10,12,18)$;$(w_1,w_2,w_3,$

w_4) = (2,4,6,9)和 M = 15。

利用上面所定义的 u(·)和$\hat{c}$(·),通过 LCBB 分枝-限界检索此问题的状态空间树的工作情况如下:开始将根(即图 9.10 的结点 1)作为 E-结点,$\hat{c}$(1) = -38,u(1) = -32。由于它不是答案结点,因此过程 LCBB 置 ans = 0 和 U = -32 + ε。扩展此 E-结点,生成它的两个儿子结点 2 和 3,$\hat{c}$(2) = -38,$\hat{c}$(3) = -32,u(2) = -32,u(3) = -22。结点 2,3 放入活结点表,结点 2 变成下一个 E-结点,它生成结点 4 和 5,这两个结点也放入活结点表。活结点表中结点 4 的$\hat{c}$值最小,所以结点 4 变成下一个 E-结点,它生成结点 6 和 7。假定先生成结点 6,它放入活结点表,接着放入结点 7 并将 U 修改成 -38 + ε。下一个 E-结点将是 6 或 7,假定是 7,则它的两个儿子是 8 和 9。将 8 放入活结点表,因为 8 是答案结点,所以将 U 修改成 -38;而结点 9 的$\hat{c}$(9) = -20 > U = 38,因此,它立即被杀死。现在活结点表中具有最小$\hat{c}$值的结点是 6 和 8,无论哪个结点变成下一个 E-结点都有$\hat{c}$(E)≥U,因此,在找到答案结点 8 的情况下终止检索,这时打印出值 -38 和路径 8,7,4,2,1,算法结束。要指出的是,由路径 8,7,4,2,1 并不能弄清是由哪些物品装入背包才得到 -$\sum p_i x_i$ = U,即看不出这些 x_i 的取值情况,因此在实现过程 LCBB 时应保留一些能反映 x_i 取值情况的附加信息。一种解决办法是每一个结点增设一个位信息段 TAG,由答案结点到根结点的这一系列 TAG 位给出这些 x_i 的值。于是,对此问题将有 TAG(2) = TAG(4) = TAG(6) = TAG(8) = 1 和 TAG(3) = TAG(5) = TAG(7) = TAG(9) = 0。路径 8,7,4,2,1 的一系列 TAG 是 1011,因此 x_4 = 1, x_3 = 0, x_2 = 1, x_1 = 1。

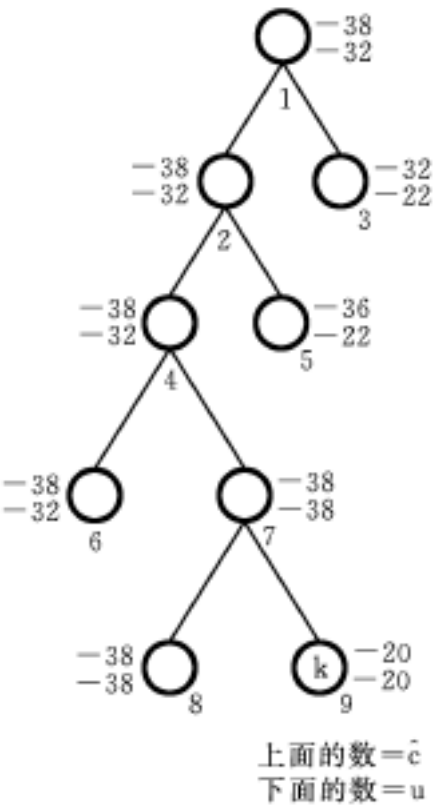


图 9.10　例 9.2 的 LC 分枝-限界树

为了用过程 LCBB(算法 9.4)求解背包问题,需要确定:①被检索的状态空间树中结点的结构;②如何生成一给定结点的儿子;③如何识别答案结点;④如何表示活结点表。

所需的结点结构取决于开始时采用哪一种状态空间树表示,这里仍采用大小固定的元组表示。要生成并放在活结点表上的每一个结点应有 6 个信息段:PARENT、LEVEL、TAG、CU、PE 和 UB 信息段。其中,PARENT 信息段是结点 X 的父结点链接指针;LEVEL 信息段标志出结点 X 在状态空间树中结点 X 的级数,在生成结点 X 的儿子时使用,通过置 $X_{LEVEL(X)}$ = 1 表示生成 X 的左儿子,$X_{LEVEL(X)}$ = 0 表示生成 X 的右儿子;位信息段 TAG 正如例 9.2 所描述的那样,用来输出最优解的 x_i 值;CU 信息段用来保存在结点 X 处背包的剩余空间,该信息段在确定 X 左儿子的可行性时使用;PE 信息段用来保存在结点 X 处已装入物品相应的效益值的和,即 $\sum_{1\le i<LEVEL(X)} p_i x_i$,它在计算$\hat{c}$(X)和 u(X)时使用;UB 信息段用来存放结点 X 的$\hat{c}$(X)值,它在确定活结点表中具有最小$\hat{c}$值的结点和将 X 插入活结点表的适当位置时使用。

使用这 6 个信息段的结点结构可以很容易地确定出任一活结点 X 的两个儿子。当且仅当 CU(X)≥$W_{LEVEL(X)}$,X 的左儿子 Y 可行。在这种情况下,PARENT(Y) = X;LEVEL(Y)

= LEVEL(X) + 1; CU(Y) = CU(X) - $W_{LEVEL(X)}$; PE(Y) = PE(X) + $P_{LEVEL(X)}$; TAG(Y) = 1 以及 UB(Y) = UB(X)。可以类似地生成 X 的右儿子。答案结点也很好识别,对于结点 X,当且仅当 LEVEL(X) = n + 1, X 是答案结点。

在活结点表上需要完成如下 3 项任务:①测试活结点表是否为空;②将结点加入表;③从表中删去具有最小 UB 值的结点。为了能有效地完成以上 3 项任务,显然应将活结点表构造成一个 min-堆。如果有 m 个结点,则第①项任务可在 Θ(1)时间内完成,第②和③项任务要用的时间是Θ(log n)。

将以上讨论和过程 LCBB 加在一起就得到 0/1 背包问题的一个完整的 LC 分枝-限界算法。如果适当修改 LCBB,还可进一步提高算法的效率。首先所要作的修改是,把计算负值的$\hat{c}$和 u 改为计算正值的 - $\hat{c}$和 - u;将保留最小上界 U 改为保留 L = - U;对于任一活结点 X,使 UB(X) = - $\hat{c}$(X)。这些改动只需在过程 LCBB 中作稍许修改即可实现。不过,这些改动对于算法的运行时间并无实质性的影响,只是将极小化问题的算法变换成了极大化问题的算法。于是 L 是最优装入的下界,而 UB(X)是由以 X 为根的子树中可能得到的答案结点的最大装入上界。再增加一些减少运行时间的措施,最后就得到求解 0/1 背包问题的 LC 分枝-限界算法 LCKNAP。

LCKNAP 用了 6 个子算法:LUBOUND(算法 9.6)、NEWNODE(算法 9.7(a))、FINISH(算法 9.7(b))、INIT、GETNODE 和 LARGEST。子算法 LUBOUND 计算 - $\hat{c}$(·)和 - $\hat{u}$(·)。NEWNODE 生成一个具有 6 个信息段的结点,给各信息段置入适当的值并将此结点加入活结点表。过程 FINISH 打印出最优解的值和此最优解中 x_i = 1 的那些物品。INIT 对可用结点表和活结点表置初值,由于这些算法都不释放结点,因此可以依次使用这些结点,即对结点 1 到 m 按 1,2,…,m 的次序赋值。GETNODE 取一个可用结点。LARGEST 在活结点表中取一个具有最大 UB 值的结点作为下一个 E-结点。

算法 9.6 计算下界和上界的算法

```
procedure LUBOUND(P,W,rw,cp,N,k,LBB,UBB)
    // rw 是背包的剩余容量,cp 为已获得的效益。还有物品 k,…,N 要考虑 //
    // LBB = - u(X),UBB = - ĉ(X) //
    LBB←cp;c←rw
    for i←k to N do
      if c < W(i) then UBB←LBB + c * P(i)/ W(i)
                  for j←i + 1 to N do
                    if c≥W(j) then c←c - W(j)
                                   LBB←LBB + P(j)
                    endif
                  repeat
                  return
      endif
      c←c - W(i);LBB←LBB + P(i)
    repeat
    UBB←LBB
end LUBOUND
```

算法 9 .7(a)　生成一个新结点

```
procedure NEWNODE(par,lev,t,cap,prof,ub)
    //生成一个新结点 I 并将它加到活结点表//
    call GETNODE(I)
    PARENT(I)←par; LEVEL(I)←lev; TAG(I)←t
    CU(I)←cap; PE(I)←prof; UB(I)←ub
    call ADD(I)
end NEWNODE
```

算法 9 .7(b)　打印答案

```
procedure FINISH(L,ANS,N)
    //输出解//
    real L; global TAG,PARENT
    print('VALUE OF OPTIMAL FILLING IS',L)
    print('OBJECTS IN KNAPSACK ARE')
    for j←N to 1 by -1 do
      if TAG(ANS)=1 then print (j) endif
      ANS←PARENT(ANS)
    repeat
end FINISH
```

LCKNAP 的参量是 P, W, M, N 和 ε。N 是物品数; P(i)和 W(i)分别是物品 i 的效益和重量,物品排列的次序满足 P(i)/ W(i)≥P(i+1)/ W(i+1), 1≤i≤N; M 是背包容量; 与 LCBB 一样,此算法也用了一个很小的正常数,ε 是它的形参。算法中局部量 L 是在迄今所找到的最好解的值与 LUBOUND 算出的最大下界减去 ε 后的值中取大值。第 1～5 行对可用结点表、活结点表和检索树的根结点置初值。根结点 E 是第一个 E-结点。第 6～24 行的循环依次检查所生成的每个活结点。此循环在以下两种情况下终止,或者不再剩有活结点(第 22 行),或者为了扩展(即取下一个 E-结点)而选择的结点 E 有 UB(E)≤L(第 24 行)。在后一种情况下,终止之所以正确是由于选作下一个 E-结点的结点具有最大 UB(E)值,因此其它的任何活结点 X 都有 UB(X)≤UB(E)≤L,这样的每一个都不可能导致其值比 L 还大的解。在此循环中,新的 E-结点 E 得到检查,有两种可能的结果:①它是一个叶结点(LEVEL(E)=n+1),则在第 9～11 行确定它是否是一个答案结点,若是,它还有可能代表最优解;②它不是一个叶结点,那么它就会生成两个儿子,左儿子 X 对应于 $x_i=1$,右儿子 Y 对应于 $x_i=0$,这里 i=LEVEL(E)。当且仅当背包有足够的空间能放下物品 i,即 cap≥W(i)时,这个左儿子是可行的,它可能导致一个答案结点。在左儿子可行的情况下,由 LUBOUND 算出它的上界并由此可得 UB(X)=U(E)。因为 UB(E)>L(第 24 行)或者 L=UBB-ε<UBB(第 5 行),所以将 X 加入活结点表。注意,由于左儿子和 E 的下界和上界相同,因此不用再计算它的下、上界。对于 E 的右儿子 Y,因为 E 是可行的,所以它总是可行的。不过它的下界和上界值可能和 E 的不同,因此要在第 16 行调用一次 LUBOUND 来获取 UB(Y)=UBB。如果 UBB≤L,则杀死结点 Y;反之则在第 18 行将其加入活结点表并在第 19 行修改 L 的值。

算法 9 .8　背包问题的 LC 分枝-限界算法

```
line procedure LCKNAP(P,W,M,N,ε)
    //这是0/1背包问题的最小成本分枝-限界算法,它使用大小固定的元组表示。假设P(1)/
      W(1)≥P(2)/W(2)≥…≥P(N)/W(N)//
    real P(N),W(N),M,L,LBB,UBB,cap,prof
    integer ANS,X,N
1    call INIT    //初始化可用结点表及活结点表//
2    call GETNODE(E)    //根结点//
3    PARENT(E)←0;LEVEL(E)←1;CU(E)←M;PE(E)←0
4    call LUBOUND (P,W,M,N,0,1,LBB,UBB)
5    L←LBB-ε;UB(E)←UBB
6    loop
7      i←LEVEL(E);cap←CU(E);prof←PE(E)
8      case
9        :i=N+1:    //解结点//
10         if prof>L then L←prof;ANS←E
11         endif
12         :else:    //E有两个儿子//
13         if cap≥W(i) then    //左儿子可行//
14         call NEWNODE(E,i+1,1,cap-W(i),prof+P(1)UB(E))
15         endif
                  //看右儿子是否会活//
16         call LUBOUND(P,W,cap,prof,N,i+1,LBB,UBB)
17         if UBB>L then    //右儿子会活//
18           call NEWNODE(E,i+1,0,cap,prof,UBB)
19           L←max (L,LBB-ε)
20         endif
21       endcase
22       if 不再有活结点 then exit endif
23       call LARGEST(E)    //下一个E-结点是UB值最大的结点//
24     until UB(E)≤L repeat
25     call FINISH(L,ANS,N)
26  end LCKNAP
```

9.2.2 FIFO分枝-限界求解

例9.3 [FIFOBB]背包问题采用式(9.1)表示,考虑用FIFOBB(算法9.3)来求解例9.2的背包实例,其工作情况如下:最开始根结点(即图9.11中结点1)是E-结点,活结点队为空。由于结点1不是答案结点,因此U置初值为$u(1)+\varepsilon=-32+\varepsilon$。扩展结点1,生成它的两个儿子2和3并将它们依次加入活结点队,结点2成为下一个E-结点,生成它的儿子4和5并将它们加入队。结点了成为下一个E-结点,生成它的儿子6和7,结点6加入队,由于结点7的$\hat{c}(7)=-30>U$,因此立即被杀死。下次扩展结点4,生成它的儿子8和9并将它们加入队,修改$U=u(9)+\varepsilon=-38+\varepsilon$。接着要成为E-结点的结点是5和6,由于它们的$\hat{c}$值均大于U,因此都被杀死。结点8是下一个E-结点,生成结点10和11,结点10不可行,于是

被杀死;结点 11 的$\hat{c}(11) = -32 > U$,因此也被杀死。接着扩展结点 9,当生成结点 12 时将 U 和 ans 的内容分别修改成 - 38 和 12,结点 12 加入活结点队,生成结点 13,但$\hat{c}(13) > U$,因此 13 立即被杀死。此时,队中只剩下一个活结点 12,结点 12 是叶结点,不可能有儿子,所以终止检索,输出 U 值和由结点 12 到根的路径。为了能知道在这条路径上 x 的取值情况,和例 9.2 一样,各个结点还需附上能反映 x_i 取值的信息。

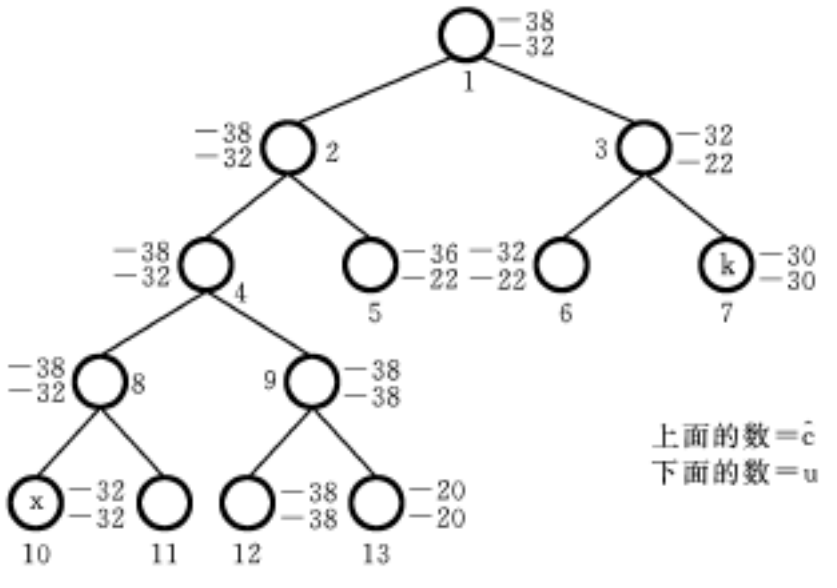


图 9.11　例 9.3 的 FIFO 分枝-限界树

在用 FIFO 分枝-限界算法处理背包问题时,由于结点的生成和确定其是否变为 E-结点是逐级进行的,因此无需对每个结点专设一个 LEVEL 信息段,而只要用标志‘#’来标出活结点队中哪些结点属于同一级即可。于是,状态空间树中每个结点可用 CU、PE、TAG、UB 和 PARENT 这 5 个信息段构成。过程 NNODE(算法 9.9)取一个可用结点并给此结点各信息段置值,然后将其加入活结点队。和 LCKNAP 一样,通过适当修改该问题的 FIFO 分枝-限界算法可将其变换成一个处理极大化问题的算法,过程 FIFOKNAP 描述了经过修改后的算法。

算法 9.9　生成一个新结点

```
procedure NNODE(par, t cap, prof, ub)
    // 生成一个新结点 I 并将它加入活结点队 //
    call GETNODE(I)
    PARENT(I) ←par; TAG(I)←t
    CU(I)←cap; PE(I)←prof; UB(E)←ub
    call ADDQ(X)
end NNODE
```

在算法 FIFOKNAP 中,L 表示最优解值的下界。由于只有生成了 N + 1 级的结点才可能到达解结点,因此可以不用 LCKNAP 中的 ε。第 3～6 行对可用结点表、根结点 E、下界 L 和活结点队置初值。活结点队最初有根结点 E 和级结束标志'# '。i 是级计数器,它的初始值是 1。在算法执行期间 i 的值总是对应于当前 E-结点的级数。在第 7～26 行 while 循环的每一次迭代中,取出 i 级上所有的活结点,它们是由第 8～23 行的循环从队中逐个被取出的。一旦取出级结束标志,则从第 11 行跳出循环;否则仅在 UB(E)≥L 时扩展 E,在第 13～21 行生成 E-结点的左、右儿子,这部分的代码与过程 LCKNAP 相应部分的代码类似。当控制从 while 循环转出时,活结点队上所剩下的结点都是 N + 1 级上的结点。其中,具有最大 PE

值的结点是最优解对应的答案结点,它可通过逐个检查这些剩下的活结点来找到。过程 FINISH(算法 9 .7)打印最优解的值和为了得到这个值应装入背包的那些物品。

算法 9 .10 背包问题的 FIFO 分枝-限界算法

```
procedure FIFOKNAP(P,W,M,N)
     //功能和假设均与 LCKNAP 相同//
1     real P(N),W(N),M,L,LBB,UBB,E,prof,cap
2     integer ANS,X,N
3     call INIT;i←1
4     call LUBOUND(P,W,M,0,N,1,L,UBB)
5     call NNODE(0,0,M,0,UBB)   //根结点//
6     call ADDQ('# ')   //级标志//
7     while i≤N do   //对于 i 级上的所有活结点//
8       loop
9         call DELETEQ(E)
10          case
11          :E = '# ':exit   //i 级结束,转到 24 行//
12          :UB(E)≥L:   //E 是活结点//
13               cap←CU(E);prof←PE(E)
14               if cap≥W(i) then   //可行左儿子//
15                  call NNODE(E,1,cap - W(i),prof + P(i),UF(E))
16               endif
17               call LUBOUND(P,W,cap,prof,N,i + 1,LBB,UBB)
18               if UBB≥L then   //右儿子是活结点//
19                  call NNODE(E,0,cap,prof,UBB)
20                  L←max (L,LBB)
21            endif
22         endcase
23      repeat
24      call ADDQ('# ')   //级的末端//
25      i←i + 1
26    repeat
27    ANS←PE(X) = L 的活结点 X
28    call FINISH(L,ANS,N)
29  end FIFOKNAP
```

9 .3 货郎担问题

6 .7 节介绍了货郎担问题的一个动态规划算法,它的计算复杂度为 $O(n^2 2^n)$。本节讨论货郎担问题的分枝-限界算法,它的最坏情况时间虽然也为 $O(n^2 2^n)$,但对于许多具体实例而言,却比动态规划算法所用的时间要少得多。

设 G = (V, E)是代表货郎担问题的某个实例的有向图,| V | = n, c_{ij} 表示边〈i, j〉的成本。

若〈i,j〉｜ E,则有 $c_{ij}=\infty$。为不失一般性,假定每一次周游均从结点 1 开始并在结点 1 结束,于是解空间 S 可表示成:S = {1,π,1|π 是{2,3,…,n}的一种排列},|S| = (n - 1) !。为减小 S 的大小,可将 S 限制为:只有在 $0\leq j\leq n-1$,$\langle i_j,i_{j+1}\rangle\in E$ 且 $i_0=i_n=1$ 的情况下,$(1,i_1,i_2,\cdots,i_{n-1})\in S$。可以将这样的 S 构造成一棵类似于 n-皇后问题的状态空间树(见图 8 .2)。图9 .12给出了|V| = 4 的一个完全图的一种状态空间树。树中每个叶结点 L 是一个解结点,它代表由根到 L 路径所确定的一次周游。结点 14 表示 $i_0=1,i_1=3,i_2=4,i_3=2$ 和 $i_4=1$ 的一次周游。

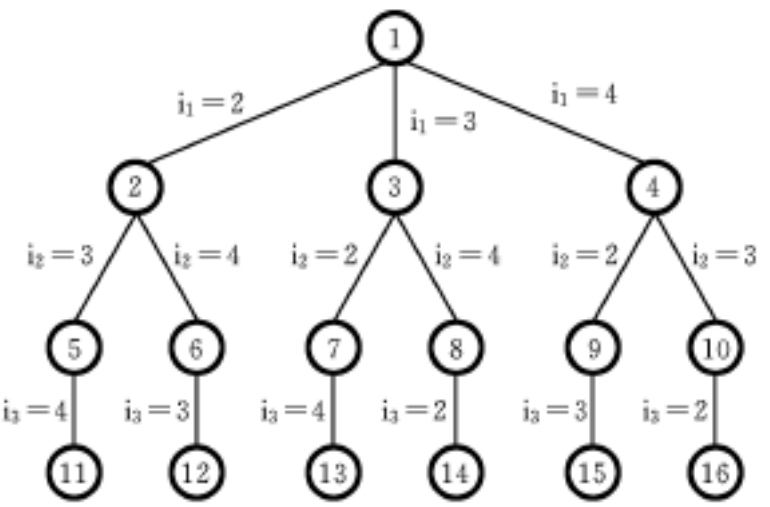


图 9 .12　n = 4,$i_0=i_4=1$ 的货郎担问题的状态空间树

为了用 LC 分枝-限界法检索货郎担问题的状态空间树,需要定义成本函数 c(·),成本估计函数 $\hat{c}(\cdot)$ 和上界函数 u(·),使它们对于每个结点 X,有 $\hat{c}(X)\leq c(X)\leq u(X)$。c(·)可以定义为

$$c(X)=\begin{cases}\text{由根到 X 的路径确定的周游路线成本} & \text{X 是叶结点}\\ \text{子树 X 中最小成本叶结点的成本} & \text{X 不是叶结点}\end{cases}$$

在 c(·)作了以上定义的情况下,对于每个结点 X 均满足 $\hat{c}(X)\leq c(X)$ 的函数 $\hat{c}(\cdot)$ 可简单地定义成:$\hat{c}(X)$ 是由根到结点 X 那条路径确定的(部分)周游路线的成本。例如,在图 9 .12 的结点 6 处所确定的部分周游路线为 $i_0=1,i_1=2,i_2=4$。它包含边 〈1,2〉 和 〈2,4〉。不过一般并不采用以上定义的 $\hat{c}(\cdot)$,而是采用一个更好的 $\hat{c}(\cdot)$。在这个 $\hat{c}(\cdot)$ 的定义中使用了图 G 的归约成本矩阵。下面先给出归约矩阵的定义。如果矩阵的一行(列)至少包含一个零且其余元素均非负,则此行(列)称为已归约行(列)。所有行和列均为已归约行和列的矩阵称为归约矩阵。可以通过对一行(列)中每个元素都减去同一个常数 t(称为约数)将该行(列)变成已归约行(列)。逐行逐列施行归约就可得到原矩阵的归约矩阵。假设第 i 行的约数为 t_i,第 j 列的约数为 r_j,$1\leq i,j\leq n$,那么各行、列的约数之和 $L=\sum_{i=1}^{n}t_i+\sum_{j=1}^{n}r_j$ 称为矩阵约数。作为一个例子,考虑一个有 5 个结点的图 G,它的成本矩阵 C 为

$$C=\begin{bmatrix}\infty & 20 & 30 & 10 & 11\\ 15 & \infty & 16 & 4 & 2\\ 3 & 5 & \infty & 2 & 4\\ 19 & 6 & 18 & \infty & 3\\ 16 & 4 & 7 & 16 & \infty\end{bmatrix}\qquad(9.2)$$

因为对 G 的每次周游只含有由 i($1\leq i\leq 5$)出发的 5 条边 〈i,1〉,〈i,2〉,〈i,3〉,〈i,4〉,〈i,5〉 中的一条边 〈i,j〉,同样也只含有进入 j($1\leq j\leq 5$)的 5 条边 〈1,j〉,〈2,j〉,〈3,j〉,〈4,j〉,〈5,j〉 中的一条边 〈i,j〉,所以在此成本矩阵中,若对 i 行(或 j 列)施行归约,即将此行(列)的每个元素减去该行(列)的最小元素 t,则此次周游成本减少 t。这表明原矩阵中各条周游路线的成本分别

是与其归约成本矩阵相应的周游路线成本与矩阵约数之和,因此矩阵约数 L 显然是此问题的最小周游成本的一个下界值,于是可以将它取作状态空间树根结点的$\hat{c}$值。对矩阵 C 的 1,2,3,4,5 行和 1,3 列施行归约得 C 的归约成本矩阵 C′,矩阵约数 L = 25。因此,图 G 的周游路线成本最少是 25。

$$C'=\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix} \tag{9.3}$$

为了定义函数$\hat{c}(\cdot)$,在货郎担问题的状态空间树中对每个结点都附以一个归约成本矩阵。设 A 是结点 R 的归约成本矩阵,S 是 R 的儿子且树边〈R,S〉对应这条周游路线中的边〈i,j〉。在 S 是非叶结点的情况下,S 的归约成本矩阵可按以下步骤求得:①为保证这条周游路线采用边〈j,j〉而不采用其它由 i 出发或者进入 j 的边,将 A 中 i 行和 j 列的元素置为∞;②为防止采用边〈i,1〉(因为在已选定的路线上加入边〈i,j〉之后若再采用边〈j,1〉就会构成一个环从而得不到这条周游路线),将 A(j,1)置为∞;③对于那些不全为∞的行和列施行归约则得到 S 的归约成本矩阵,令其为 B,矩阵约数为 r。非叶结点 S 的$\hat{c}$值可定义为

$$\hat{c}(S) = \hat{c}(R) + A(i,j) + r \tag{9.4}$$

如果 S 是叶结点,由于一个叶结点确定一条唯一的周游路线,因此可用这条周游路线的成本作为 S 的$\hat{c}$值,即$\hat{c}(S) = c(S)$。

至于上界函数 u(·)可将其定义为,对于树中每个结点 R,$u(R) = \infty$。

现在用 LC 分枝-限界算法 LCBB 求解式(9.2)的货郎担问题实例的最小成本周游路线。LCBB 使用了上面定义的$\hat{c}(\cdot)$和 u(·)。图 9.13 给出了 LCBB 所产生的那一部分状态空间树,结点外的数是该结点的$\hat{c}$值。

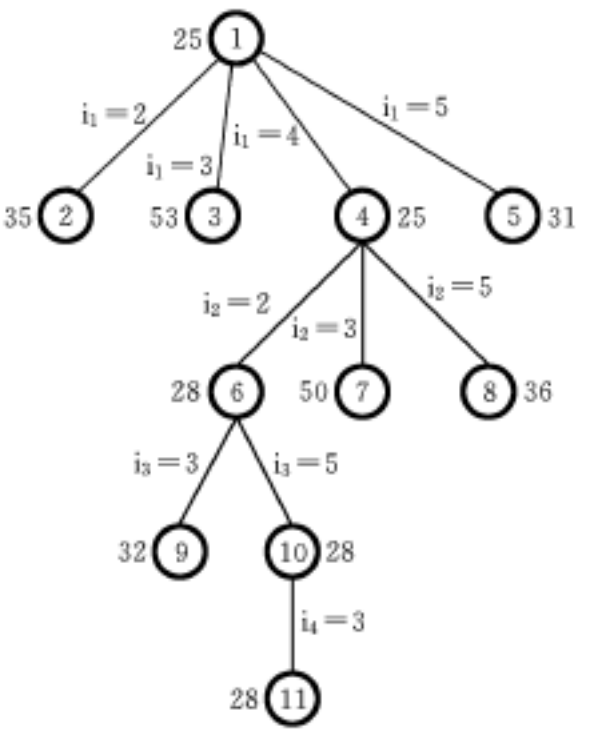


图 9.13 算法 LCBB 生成的状态空间树

根结点 1 是第一个 E-结点,它的归纳成本矩阵为式(9.3)的矩阵 C′,此时 U = ∞。扩展结点 1,依次生成结点 2,3,4 和 5。它们对应的归约成本矩阵为

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

(a)结点 2

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

(b)结点 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

(c)结点 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(d)结点 5

以结点 3 为例,它的归约成本矩阵由以下运算得到:先将矩阵 C′的 1 行和 3 列所有元素置成∞;再将 C′(3,1)置成∞;然后归约第 1 列,将该列的每个元素减去 11 即得。由式(9.4)得

$$\hat{c}(3) = 25 + 17(\text{即 } C'(1,3)\text{的值}) + 11 = 53$$

结点 2,4 和 5 的归约成本矩阵和$\hat{c}$值也可类似得到。U 的值不变。结点 4 变成下一个 E-结点,它的儿子结点 6,7 和 8 被依次生成,与它们对应的归约成本矩阵为

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

(e)结点 6

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

(f)结点 7

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

(g)结点 8

此时的活结点有 2,3,5,6,7 和 8,其中$\hat{c}(6)$最小,所以结点 6 成为下一个 E-结点。扩展结点 6,生成结点 9 和 10,它们的归约成本矩阵为

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

(h)结点 9

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

(i)结点 10

结点 10 是下一个 E-结点,由它生成结点 11。结点 11 是一个答案结点,它对应的周游路线的成本是$\hat{c}(11) = 28$, U 被修改成 28。下一个 E-结点应是结点 5,由于$\hat{c}(5) = 31 > 28$,故算法 LCBB 结束并得出最小成本周游路线是 1,4,2,5,3,1。

如果把一条周游路线看成是 n 条边的集合,则可以得到解的另一种表示形式。设图 G = (V,E)有 e 条边,于是,一条周游路线均含有这 e 条边中的 n 条不同的边。由此可以将货郎担问题的状态空间树构造成一棵二元树,树中结点的左分枝表示周游路线中包含一条指定的边,右分枝表示不包含这条边。例如,图 9.14(b)和(c)就表示图 9.14(a)所示的那个三结点图中的两棵可能的状态空间树的前三级。就一般情况而言,一个给定的问题可能有多棵不同的状态空间树,它们的差别在于对图中边的取舍的决策次序不同。图 9.14(b)所示的首先确定边〈1,3〉的取舍,图 9.14(c)所示的则首先确定边〈1,2〉的取舍。应采用哪种状态空间树进行检索随货郎担问题的具体实例而定,因此所考虑的是动态状态空间树。

为了根据问题的具体实例构造出便于检索的二元状态空间树,应确定图中边的取舍次

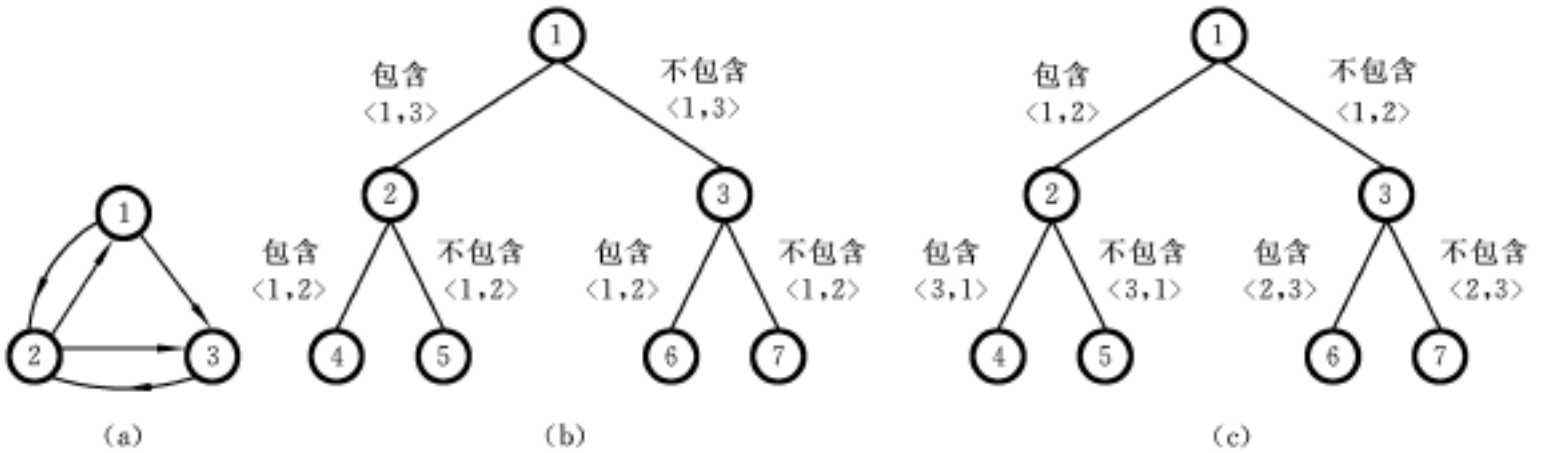


图 9.14 一个例子

(a) 圆;(b) 部分状态空间树;(c) 部分状态空间树

序。如果选取边〈i,j〉,则这条边将解空间划分成两个子集合,即在将要构造出的状态空间树中,根的左子树表示含有边〈i,j〉的所有周游,根的右子树表示不含边〈i,j〉的所有周游。此时,如果左子树中包含一条最小成本周游路线,则只需再选取 n - 1 条边;如果所有的最小成本周游路线都在右子树中,则还需对边作 n 次选择。由此可知在左子树中找最优解比在右子树中找最优解容易,所以我们希望选取一条最有可能在最小成本周游路线中的边〈i,j〉作为这条"分割"边。一般所采用的选择规则是:选取一条使其右子树具有最大$\hat{c}$值的边。使用这种选择规则可以尽快得到那些$\hat{c}$值大于最小周游成本的右子树。还有一些其它的选择规则,例如选取使左、右子树$\hat{c}$值相差最大的边等等。本节只使用前一种选择方法。

在二元状态空间树中,根结点的归约成本矩阵由该实例的成本矩阵归约而得,根的$\hat{c}$值是矩阵约数。如果非叶结点 S 是结点 R 的左儿子则 S 的归约成本矩阵的求取与前面所述步骤相同,它的$\hat{c}$值仍由式(9.4)获得。如果非叶结点 S 是结点 R 的右儿子,由于 R 的右分枝代表不包含边〈i,j〉的周游,因此应将 R 的归约成本矩阵 A 中的元素 A(i,j)置成∞后,再归约此矩阵不全为∞的行和列(实际上只需重新归约第 i 行和第 j 列),即得 S 的归约成本矩阵 B 和矩阵约数 $\Delta = \min_{k\neq j}\{A(i,k)\} + \min_{k\neq i}\{A(k,j)\}$。在计算$\hat{c}(S)$时,由于周游路线不包含边〈i,j〉,所以 A(i,j)不必加入,即

$$\hat{c}(S) = \hat{c}(R) + \Delta \qquad (9.5)$$

如果 S 是叶结点,则与前面一样有$\hat{c}(S) = c(S)$。如果选取的边〈i,j〉在 R 的归约矩阵 A 中对应的元素 A(i,j)为正数,则该边显然不可能使 R 右子树的$\hat{c}$值为最大,因为$\hat{c}(S) = \hat{c}(R)$。所以,为了使 R 右子树的$\hat{c}$值最大,应从 R 的归约成本矩阵元素为 0 的对应边中选取有最大 Δ 值的边。

仍以式(9.2)的货郎担问题为例,用算法 LCBB 在动态二元树上进行检索。开始根结点 1 是 E-结点(见图9.15),$\hat{c}(1) = 25$,归约矩阵 C′中零元素对应的边为〈1,4〉,〈2,5〉,〈3,1〉,〈3,4〉,〈4,5〉,〈5,2〉和〈5,3〉,各边的 Δ 值分别为 1, 2, 11, 0, 3, 3 和 11,因此选取边〈3,1〉或〈5,3〉作为"分割"边可以使结点 1 右分枝的$\hat{c}$值最大。假

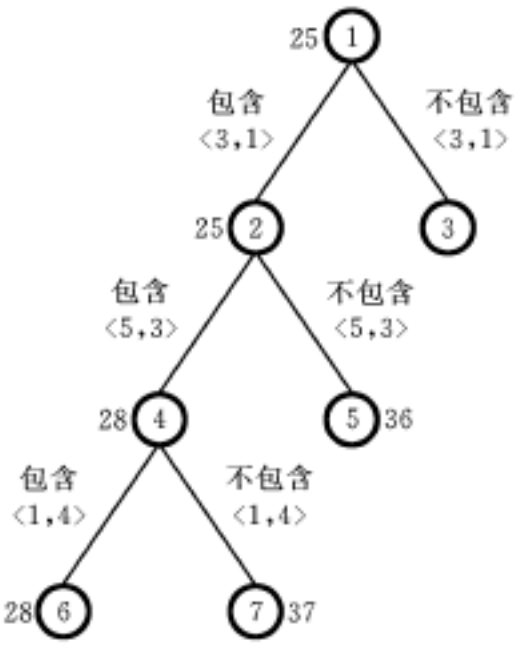


图 9.15 式(9.2)的状态空间树

定 LCBB 选取 〈3,1〉。结点 1 生成结点 2 和 3,其中$\hat{c}(2)=25$,$\hat{c}(3)=36$,与它们对应的归约成本矩阵为

$$\begin{bmatrix} \infty & 10 & \infty & 0 & 1 \\ \infty & \infty & 11 & 2 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 3 & 12 & \infty & 0 \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(a)结点 2

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 1 & \infty & 11 & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & 12 & \infty & 0 \\ 0 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b)结点 3

结点 2 成为下一个 E-结点,边 〈1,4〉,〈2,5〉,〈3,5〉,〈5,2〉和 〈5,3〉的 Δ 分别是 3,2,3,3 和 11,选取边 〈5,3〉为“分割”边,生成结点 4 和 5,$\hat{c}(4)=28$,$\hat{c}(5)=36$,与它们对应的归约成本矩阵为

$$\begin{bmatrix} \infty & 7 & \infty & 0 & \infty \\ \infty & \infty & \infty & 2 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(c)结点 4

$$\begin{bmatrix} \infty & 10 & \infty & 0 & 1 \\ \infty & \infty & 0 & 2 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 3 & 1 & \infty & 0 \\ \infty & 0 & \infty & 12 & \infty \end{bmatrix}$$

(d)结点 5

结点 4 成为下一个 E-结点,边 〈1,4〉,〈2,5〉,〈4,2〉和 〈4,5〉的 Δ 分别是 9,2,7 和 0,选取边 〈1,4〉为下一条“分割”边,生成结点 6 和 7,$\hat{c}(6)=28$,$\hat{c}(7)=37$,与它们对应的归约成本矩阵为

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(e)结点 6

$$\begin{bmatrix} \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(f)结点 7

结点 6 是下一个 E-结点,此时需求的 5 条边已求出了 3 条,即{〈3,1〉,〈5,3〉,〈1,4〉},再求两条边就可得到一条周游路线,由矩阵(e)可知此时只剩下两条边{〈2,5〉,〈4,2〉},故它们就是这条周游路线所需要的边。至此 LCBB 求出了一条成本为 28 的周游路线 5,3,1,4,2,5。U 被修改成 28,下一个应成为 E-结点的结点是 3,由于$\hat{c}(3)=36>U$,故 LCBB 结束。

此例对 LCBB 作了一点修改,即在“靠近”解结点时作的处理与“没靠近”时的不同。如在结点 6 处,由于距离解结点只有两级,此时就不再采用分枝-限界方法求结点 6 的儿子和孙子结点的$\hat{c}$值,而是对结点 6 为根的子树中结点逐个检索来找出答案结点。这种对多结点子树采用分枝-限界法而对结点数少的子树采用完全检索的处理方法可使算法效率提高一些。这种处理方法对于图 9.13 同样适用。

关于货郎担问题还可用另外一些分枝-限界方法求解,有兴趣的读者可参阅 E.Horowitz 和 S.Sahni 所著的“Fundamentals of Computer Algorithms”(1978 年)以及 S.E.Goodman 和 S.T.Hedetniemi 所著的“Introduction to the Design and Analysis of Algorithms”(1977 年)。

习 题 九

9.1 证明定理 9.1。

9.2 写一个用 LIFO 分枝-限界方法检索一个最小成本答案结点的程序概要 LIFOBB。

9.3 给定一个带有限期的作业排序问题: $n=5,(p_1,p_2,\cdots,p_5)=(6,3,4,8,5),(t_1,t_2,\cdots,t_5)=(2,1,2,1,1),(d_1,d_2,\cdots,d_5)=(3,1,4,2,4)$。采用 9.1 节关于作业排序问题的 $\hat{c}(\cdot)$ 和 $u(\cdot)$ 的定义,画出 FIFOBB,LIFOBB 和 LCBB 对上述问题所生成的、大小可变的元组表示的部分状态空间树,求出对应于最优解的罚款值。

9.4 使用大小固定的元组表示写一个求解带限期作业排序问题的完整的 LC 分枝-限界算法。

9.5 证明定理 9.4。

9.6 证明定理 9.6。

9.7 在大小可变的元组表示下解算例 9.2。

9.8 在大小可变的元组表示下解算例 9.3。

9.9 画出 LCKNAP 对下列背包问题实例所生成的部分状态空间树:

(1) $n=5,(p_1,p_2,\cdots,p_5)=(10,15,6,8,4),(w_1,w_2,\cdots,w_5)=(4,6,3,4,2),M=12$;

(2) $n=5,(p_1,p_2,\cdots,p_5)=(w_1,w_2,\cdots,w_5)=(4,4,5,8,9),M=15$。

9.10 用 LC 分枝-限界法在动态状态空间树上做 9.9 题。使用大小固定的元组表示。

9.11 使用大小固定的元组表示下的动态状态空间树,写出背包问题的 LC 分枝-限界算法。

9.12 已知一货郎担问题的实例由下面成本矩阵所定义:

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

(1) 求它的归约成本矩阵。

(2) 用与图 9.12 类似的状态空间树结构和 9.3 节定义的 $\hat{c}(\cdot)$ 去获取 LCBB 所生成的部分状态空间树。标出每个结点的 $\hat{c}$ 值并写出其对应的归约矩阵。

(3) 用 9.3 节介绍的动态状态空间树方法做第(2)题。

9.13 使用下面的货郎担成本矩阵做 9.12 题:

$$\begin{bmatrix} \infty & 11 & 10 & 9 & 6 \\ 8 & \infty & 7 & 3 & 4 \\ 8 & 4 & \infty & 4 & 8 \\ 11 & 10 & 5 & \infty & 5 \\ 6 & 9 & 5 & 5 & \infty \end{bmatrix}$$

9.14 使用像图 9.12 那样的静态状态空间树和归约成本矩阵方法写出实现货郎担问题的 LC 分枝-限界算法的有效程序。

9.15 使用动态状态空间树和归约成本矩阵方法写出实现货郎担问题的 LC 分枝-限界算法的有效程序。

9.16 对于任何货郎担问题实例,使用静态树的 LC 分枝-限界算法所生成的结点是否比使用动态树的 LC 分枝-限界算法生成的结点少?证明所下的结论。

第 10 章

NP-难度和 NP-完全的问题

10.1 基本概念

本章的内容包括了在算法研究方面的最重要的基本理论。这些基本理论对于计算机科学家、电气工程师和从事运筹学等方面的工作者都是十分有用的。因此,凡是在这些领域里的工作者建议读本章的内容。

不过,在阅读本章之前,读者应熟悉以下基本概念:一是算法的事先分析计算时间,它是在所给定的不同数据集下,通过研究算法中语句的执行频率而得到的;二是算法时间复杂度的数量级以及它们的渐近表示。如果一个算法在输入量为 n 的情况下计算时间为 T(n),则记作T(n) = O(f(n)),它表示时间以函数 f(n)为上界。T(n) = Ω(g(n))表示时间以函数 g(n)为下界。这些概念在第 2 章都作过详细的阐述。

另一个重要概念是关于两类问题的区别,其中第一类的求解只需多项式时间的算法(即 g(n)大于任何多项式)。对于已遇到和作过研究的许多问题,可按求解它们的最好算法所用计算时间分为两类。第一类问题的求解只需低次多项式时间。例如,本书前面讲过的有序检索的计算时间为 O(logn),分类为 O(nlogn),矩阵乘法为 $O(n^{2.81})$等。第二类问题则包括那些迄今已知的最好算法所需时间为非多项式时间的问题,例如货郎担问题和背包问题的时间复杂度分别为 $O(n^2 2^n)$和 $O(2^{n/2})$。对于第二类问题,人们一直在寻求更有效的算法,但至今还没有谁开发出一个具有多项式时间复杂度的算法。指出这一点是十分重要的,因为算法的时间复杂度一旦大于多项式时间 (典型的时间复杂度是指数时间),算法的执行时间就会随 n 的增大而急剧增加,以致即使是中等规模的问题也不能解出。

本章所讨论的 NP-完全性理论,对于第二类问题,既不能给出使其获得多项式时间的方法,也不说明这样的算法不存在。取而代之的是证明了许多尚不知其有多项式时间算法的问题在计算上是相关的。实际上,我们建立了分别叫做 NP-难度的和 NP-完全的两类问题。一个 NP-完全的问题具有如下性质:它可以在多项式时间内求解,当且仅当所有其它的 NP-完全问题也可在多项式时间内求解。假如有朝一日某个 NP-难度的问题可以被一个多项式时间的算法求解,那么所有的 NP-完全问题就都可以在多项式时间内求解。下面将会看到,一切 NP-完全的问题都是 NP-难度的问题,但一切 NP-难度的问题并不都是 NP-完全的。

10.1.1 不确定的算法

到目前为止在已用过的算法中,每种运算的结果都是唯一确定的,这样的算法叫做确定的算法(deterministic algorithm) 。这种算法和在计算机上执行程序的方式是一致的。从理

论的角度看,对于每种运算的结果“唯一确定”这一限制可以取消。即允许算法每种运算的结果不是唯一确定的,而是受限于某个特定的可能性集合。执行这些运算的机器可以根据稍后定义的终止条件选择可能性集合中的一个作为结果。这就引出了所谓不确定的算法(nondeterministic algorithm)。为了详细说明这种算法,在 SPARKS 中引进一个新函数和两条新语句:

(1) choice (S) ------ 任意选取集合 S 中的一个元素。

(2) failure ------ 发出不成功完成的信号。

(3) success ------ 发出成功完成的信号。

赋值语句 X←choice (1：n)使 X 内的结果是区域[1, n]中的任一整数(没有规则限定这种选择是如何作出的)。failure 和 success 的信号用来定义此算法的一种计算,这两条语句等价于 stop 语句,但不能起 return 语句的作用。每当有一组选择导致成功完成时,总能作出这样的一组选择并使算法成功地终止。当且仅当不存在任何一组选择会导致成功的信号,那么不确定的算法不成功地终止。choice, success 和 failure 的计算时间取为 O(1) 。能按这种方式执行不确定算法的机器称为不确定机(nondeterministic machine)。然而,这里所定义的不确定机实际上是不存在的,因此通过直觉可以感到这类问题不可能用“快速的”确定算法求解。

例 10 .1　考察给定元素集 A(1：n), n≥1 中,检索元素 x 的检索问题。需确定下标 j,使得 A(j) = x,或者当 x 不在 A 中时有 j = 0。此问题的一个不确定算法为

```
j←choice (1：n)
if A(j) = x then print(j); success endif
print ( '0'); failure
```

由上述定义的不确定算法当且仅当不存在一个 j 使得 A(j) = x 时输出“0”。此算法有不确定的复杂度 O(1)。

注意:由于 A 是无序的,因此确定的检索算法的复杂度为 Ω(n)。

例 10 .2　[分类]设 A(i), 1≤i≤n, 是一个尚未分类的正整数集。不确定的算法 NSORT(A, n)将这些数按非降次序分类并输出。为方便起见,采用一辅助数组 B(1：n)。第 1 行将 B 初始化为零。在第 2～6 行的循环中,每个 A(i)的值都赋给 B 中的某个位置,第 3 行不确定地定出这个位置,第 4 行弄清 B(j) 是否还没用过。因此,B 中整数的次序是 A 中初始次序的某种排列。第 7～9 行验证 B 是否已按非降次序分类。当且仅当整数以非降次序输出时,算法成功地完成。由于第 3 行对于这种输出次序总存在一组选择,因此算法 NSORT 是一个分类算法,它的复杂度为 O(n)。回忆前面讲过的各种确定的分类算法可知它们的复杂度应为 Ω(nlogn)。

算法 10 .1　不确定的分类算法

```
procedure NSORT (A, n)
    // 对 n 个正整数分类 //
    integer A(n), B(n), n, i, j
1     B←0 // B 初始化为零 //
2     for i←1 to n do
```

```
3        i←choice (1：n)
4        if B(j)≠0 then failure endif
5        B(j)←A(i)
6      repeat
7      for i←1 to n - 1 do  //验证 B 的次序//
8        if B(i) > B(i+1) then failure endif
9      repeat
10      print(B)
11      success
12      end NSORT
```

通过允许作不受限制的并行计算,可以对不确定的算法作出确定的解释。每当要作某种选择时,算法就好像给自己复制了若干副本,每种可能的选择有一个副本,于是许多副本同时被执行。第一个获得成功完成的副本,将引起其它所有副本的计算终止。如果一个副本获得不成功的完成则只该副本终止。前面说过 success 和 failure 信号相当于确定算法中的 stop 语句,但它们不能用来取代 return 语句。上述解释是为了便于读者理解不确定算法。

注意:对一台不确定机来说,当算法每次作某种选择时,它实际上是什么副本都不作,只是在每次作某种选择时,具有从可选集合中选择出一个“正确的”元素的能力(如果这样的元素存在的话)。一个“正确的”元素是相对于导致一成功终止的最短选择序列而定义的。在不存在导致成功终止的选择序列的情况下,则假定算法是在一个单位时间内终止并且输出“计算不成功”。只要有成功终止的可能,一台不确定机就会以最短的选择序列导致成功的终止。因为这种不确定机本来就是虚构和假想的,所以没有必要去注意机器在每一步是如何作出正确选择的。

完全有可能构造出一些这样的不确定算法,它们的多种不同的选择序列都会导致成功完成。例 10.2 的过程 NSORT 就是这样的一个算法。如果整数 A(i)有许多是相同的,那么在一个分类序列中将出现许多不同的排列。如果不是按已分好类的次序输出这些 A(i),而是输出所用的排列,那么这样的输出将不是唯一确定的。今后只关心那些产生唯一输出的不确定算法,特别是只研究那些不确定的判定算法(nondeterministic decision algorithm)。这些算法只产生“0”或“1”作为输出,即作二值决策,前者当且仅当没有一种选择序列可导致一个成功完成,后者当且仅当一个成功完成被产生。输出语句隐含于 success 和 failure 之中。在判定算法中不允许有明显的输出语句。显然,早先对不确定计算的定义意味着一个判定算法的输出由输入参数和算法本身的规范唯一地确定。

虽然上面所述的判定算法的概念看来似乎限制过严,但事实上许多最优化问题都可以改写成判定问题并使其具有如下性质:该判定问题可以在多项式时间内求解,当且仅当与它相应的最优化问题可以在多项式时间内求解。从另一方面说,如果判定问题不能在多项式时间内求解,那么与它相应的最优化问题也不能在多项式时间内求解。

例 10.3　[最大集团]图 G = (V, E)的最大完全子图叫作 G 的一个集团(clique)。集团的大小用所含的结点数来量度。最大集团问题即为确定 G 内最大集团的大小问题。与之对应的判定问题是,对于某个给定的 k,确定 G 是否有一个大小至少为 k 的集团。令 DCLIQUE(G,k)是此集团判定问题的一个确定的判定算法。假设 G 的结点数为 n,G 内最大集团

的大小可在多次应用 DCLIQUE(G, k)而求得。对于每个 k, k = n, n - 1, n - 2,… 直到 DCLIQUE 输出 1 为止,过程 DCLIQUE 都要被引用一次。如果 DCLIQUE 的时间复杂度为 f(n),则最大集团的大小可在 n * f(n)时间内求出。假如最大集团的大小可以在 g(n)时间内确定,于是其判定问题也可在 g(n)时间内解出。因此最大集团问题可在多项式时间内求解,当且仅当集团的判定问题可在多项式时间内求解。

例 10.4 [0/1 背包]背包的判定问题是确定对 x_i, $1 \le i \le n$, 是否存在一组 0/1 的赋值,使得 $\sum p_i x_i \ge R$ 和 $\sum w_i x_i \le M$。R 是一个给定的数,而这些 p_i 及 w_i 都是非负的数。显然,如果背包的判定问题不能在确定的多项式时间内求解,则它的最优化问题也同样不能在确定的多项式时间内求解。

在进一步讨论之前,为了测量复杂度必须先统一参数 n。假设 n 是算法的输入长度;还假定所有的输入都是整数。有理数输入可以视为一对整数。一般说输入量的长度是以该量的二进制表示度量的,即如果输入量是十进制的 10,相应的二进制表示就是 1010,因此它的长度就是 4。对于正整数 k,用二进制形式表示时的长度就是 $\lfloor \log_2 k \rfloor + 1$。0 的长度规定为 1。算法输入的大小或长度 n 是指输入的各个数长度的总和。因此,用不同的进位制时的长度是不同的,以 r 为基的正整数 k,其长度为 $\lfloor \log_r k \rfloor + 1$。在十进制中(r = 10)数 100 的长度为 $\log_{10} 100 + 1 = 3$。因为 $\log_r k = \log_2 k / \log_2 r$,于是以 r(r > 1)为基输入的长度为 c(r) · n,其中 n 是以二进制表示时的长度,c(r)是对于给定 r 后的一个常数。

当使用基 r = 1 给定输入量时,则称此输入是一进制形式的。在一进制形式中,数 5 的输入形式是 11111。于是正整数 k 的长度即为 k。

注意:一进制形式输入的长度与其相应的基为 r(r > 1)的输入的长度间具有指数关系。

例 10.5 [最大集团]最大集团判定问题的输入可以看作是一个边的序列和一个整数 k。E(G)内的每条边又是一对数值(i, j)。对于每条边(i, j),如果采用二进制表示,则其输入的大小为 $\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2$。于是任一实例的输入大小为

$$n = \sum_{\substack{(i,j) \in E(G) \\ i < j}} (\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2) + \lfloor \log_2 k \rfloor + 1$$

注意:如果 G 只有一个连通分图,则 $n \ge |V|$。于是,对于某个多项式 P( ),若这个判定问题不能由一个复杂度为 P(n)的算法求解,则它也不能被复杂度为 P(|V|)的算法求解。

例 10.6 [0/1 背包]假设 p_i, w_i, M 和 R 均为整数,背包判定问题输入量的大小为

$$m = \sum_{1 \le i \le n} (\lfloor \log_2 p_i \rfloor + \lfloor \log_2 w_i \rfloor) + \lfloor \log_2 M \rfloor + \lfloor \log_2 R \rfloor + 2n + 2$$

其中,$m \ge n$。如果输入量用一进制形式给出,则输入大小 s 为 $\sum p_i + \sum w_i + M + R$。对于某个多项式 P( ),背包的判定问题和最优化问题均可在 P(s)的时间内求解;但对于某个多项式 P( )还不知道有复杂度为 O(P(n))的算法。

下面给出不确定算法复杂度的形式定义。

定义 10.1 一个不确定算法所需的时间是指当存在一选择序列导致一成功完成时,为了完成任意给定的一个输入而达到成功完成所需步骤的最小值,在不可能成功完成的情况下所需的时间是 O(1)。假如对于所有的大小为 $n(n \ge n_0)$的输入,导致成功完成需要的时间至多为 c · f(n),其中 c 和 n_0 是两个常数,则这个不确定算法的复杂度为 O(f(n))。

在上述定义中,假定每个计算步骤的开销是固定的。在面向字处理的计算机中,每个字

的有限性确保了固定开销。当每步开销不固定时,则应考虑各条指令的开销。例如,两个 m 位的数相加花费的时间为 O(m),而按经典方法将这两数相乘,则花费的时间为 $O(m^2)$,等等。为了弄清注意这一问题的必要性,下面来考察子集和数判定问题的确定算法 SUM。它用了一个 M + 1 位的字 S。当且仅当整数 A(j),1≤j≤n,不存在和数为 i 的子集时 S 的第 i 位为 0。S 的位数按从右到左的次序编码为 0,1,2,…,M,且 S 的第 0 位总取值为 1。函数 SHIFT 把 S 中的各位均向左移动 A(i)位。这个算法的总步数只有 O(n)。然而,每步都要移动数据 M + 1 位,因此在一台传统的计算机上每步实际上所需的时间为 O(M)。假设对于一个大小固定的字,每个基本操作都需要一个单位的时间,那么该算法的真实复杂度是 O(nM)而不是 O(n)。

算法 10.2　子集和数判定的确定算法

```
procedure SUM(A,n,M)
    integer A(n),S,n,M
    S←1 //S 是一个有 M+1 位的字,第 0 位是 1//
    for i←1 to n do
S←S or SHIFT(S,A(i))
repeat
if Mth bit in S=0 then print ('no subset sums to M')
                  else print ('a subset sums to M')
endif
end SUM
```

不确定算法的优点是对于那些用确定算法写起来非常复杂的问题可以很容易地写出它们的不确定算法。事实上,对于许多可以通过系统检索指数大小解空间来确定地求解的问题,要得到它们的多项式时间的不确定算法是很容易的。

例 10.7　[背包判定问题]过程 DKP (算法 10.3)是背包问题的一个不确定多项式时间算法。第 1～3 行将 0/1 值赋给 X(i),1≤i≤n。第 4 行检查赋值的可行性并且查看所导致的效益值是否至少为 R。当且仅当判定问题的答案为"是",才可能是一个成功的终止。时间复杂度是 O(n)。如果 m 是用二进制表示的输入长度,则时间是 O(m)。

算法 10.3　背包问题的不确定算法

```
procedure DKP(P,W,n,M,R,X)
    integer P(n),W(n),R,X(n),n,M,i
1   for i←1 to n do
2       X(i)←choice(0,1)
3   repeat
4   if Σ(1≤i≤n) (W(i) * X(i)) > M or Σ(1≤i≤n) (P(i) * X(i)) < R then failure
                                                        else success
5   endif
end DKP
```

例 10.8　[最大集团]过程 DCK (算法 10.4)是此集团判定问题的一个不确定算法。算法开始时试探形成一个具有 k 个不同结点的集合,然后测试这些结点是否构成一个完全子图。若 G 是由其邻接矩阵和|V| = n 所给出,则输入长度 m 为 $n^2 + \lfloor \log_2 k \rfloor + \lfloor \log_2 n \rfloor + 2$。

易于看出第2～6行的不确定执行时间为 O(n)。第 7～10 行的时间为 $O(k^2)$。于是总时间为 $O(n+k^2)=O(n^2)=O(m)$。对于这个问题,已知它有多项式时间的确定算法。

算法 10.4　集团的不确定算法

```
    procedure DCK(G,n,k)
1     S←     // S 的初值为空集合//
2     for i←1 to k do // 选择 k 个不同的结点//
3       t←choice (1∶n)
4       if t∈S then failure endif
5       S←S∪t // 将 t 加到集合 S//
6     repeat
        // 此处 S 含有 k 个不同结点的下标//
7     for 使得 i∈S,j∈S 的每一对(i,j) and i≠j do
8     if (i,j)不是此图的边
9       then failure endif
10    repeat
11    success
      end DCK
```

例 10.9　[可满足性]令 $x_1, x_2, \cdots, x_n$,表示布尔变量(它们的值既可为真也可为假)。令 $\bar{x}_i$ 表示 x_i 的非。一个文字既可是一个变量也可是它的非。命题演算中的一个公式是一个可由文字及运算符 and 和 or 构成的表达式。例如:$(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$, $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$,其中∨表示 or,∧表示 and。如果一个公式具有 $\wedge_{i=1}^{k} c_{ij}$ 的形式,其中每个 c_i 为 $\vee l_{ji}$ 形式的子式,l_{ij} 均为文字,则此公式称作合取范式(conjunctive normal form),简写为 CNF。如果一个公式具有 $\vee_{i=1}^{k} c_i$ 的形式且其中每个 c_i 是 $\wedge l_{ij}$, l_{ij} 均为文字,则该公式称作析取范式(disjunctive nomal form),简写为 DNF。因此 $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ 是一个 DNF,而 $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$ 是一个 CNF。可满足性问题(satisfiability problem)是对于变量的任意一组真值指派确定公式是否为真。CNF-可满足性是指对于 CNF 公式的可满足性问题。

可以容易地得到一个多项式时间的不确定算法,当且仅当给定的命题公式 $E(x_1, \cdots, x_n)$ 是可满足的,则该算法成功地终止。这样的一个算法可以如下进行,即简单地从 2^n 组可能的真值指派中不确定地选取一组赋给 $(x_1, \cdots, x_n)$ 并且证明对于这组指派 $E(x_1, \cdots, x_n)$ 为真。

过程 EVAL(算法 10.5)完成上述工作。这个算法要求的不确定时间为 O(n),其中包括选取 $(x_1, \cdots, x_n)$ 的值和对这组指派计算 E 值所需的时间。此时间与 E 的长度成正比。

算法 10.5　可满足性的不确定算法

```
procedure EVAL(E,n)
// 判定命题 E 是否为可满足的。变量为 xi,1≤i≤n//
boolean x(n)
for i←1 to n do // 选取一组真值指派//
  x1←choice (true, false)
repeat
if E(x1,⋯,xn) = true then success// 可满足的//
```

```
                else failure
    endif
    end EVAL
```

10.1.2　NP-难度和 NP-完全类

为了度量算法的复杂度,下面将使用输入的长度作为参数。对于算法 A,如果存在一多项式 P( ),使得对 A 的每个大小为 n 的输入,A 的计算时间为 O(P(n)),则称 A 具有多项式复杂度 (polynomial complexity)。

定义 10.2　P 是所有可在多项式时间内用确定算法求解的判定问题的集合。NP 是所有可在多项式时间内用不确定算法求解的判定问题的集合。

因为确定的算法只是不确定算法的一种特例,所以可以断定 P　NP 。计算机科学还没解决的一个著名问题是不知是否有 P = NP 或者 P≠NP。

对于 NP 中所有的问题,是否存在具有多项式时间的确定算法,至今还未被人们发现呢?这点看来不像,因为有很多人对此曾作过十分艰巨的努力。而证明 P≠NP 看起来似乎也很难捉摸且在技术上还未曾有突破性的发现。不过 P = NP 如同许多未获解决的著名问题一样,对它进行研究仍然可以产生一些有用的结果。

为了研究这个问题,S .COOK 从形式上归纳出如下问题:在 NP 中是否存在单一的某个问题,使得如果我们能证明它在 P 内,那么就意味着 P = NP 。COOK 对此问题的回答是肯定的,这就是如下的 COOK 定理。

定理 10.1　[COOK 定理]可满足性在 P 内,当且仅当 P = NP。

证明　见 10.2 节。

至此,已为定义 NP-难度和 NP-完全两类问题作好了必要的准备。下面先定义可约性记号。

定义 10.3　令 L_1 和 L_2 是两个问题,如果有一确定的多项式时间算法求解 L_1,而这个算法使用了一个在多项式时间内求解 L_2 的确定算法,则称 L_1 约化为 L_2(也可以写作 $L_1 \propto L_2$)。

这个定义意味着如果对于 L_2 有一个多项式时间算法,那么就可以在多项式时间内求解 L_1。读者可以自行验证,记号∝是一传递关系(亦即若 $L_1 \propto L_2$, $L_2 \propto L_3$,则 $L_1 \propto L_3$)。

定义 10.4　如果可满足性约化为一个问题 L,则称此问题 L 是 NP-难度的。如果 L 是 NP 难度的且 L∈NP,则称问题 L 是 NP-完全的。

容易看出有些问题属于 NP-难度的而不是 NP-完全的。只有判定问题可以是 NP-完全的,其它问题,如最优化问题则可能是 NP-难度的。更进一步说,假如 L_1 是一个判定问题而 L_2 是一个最优化问题,非常可能 $L_1 \propto L_2$。人们可以轻易地证明:背包的判定问题可约化为背包的最优化问题;集团的判定问题可约化为集团的最优化问题。事实上也可以证明:这些最优化问题可约化为它们的判定问题(见习题)。然而当判定问题是 NP-完全时,最优化问题可能并不是 NP-完全的,也存在具有 NP-难度的问题却不是 NP-完全的。

例 10.10　作为具有 NP-难度但不是 NP-完全的判定问题的一个特例 ,考虑确定算法的停机问题。所谓停机问题是对于一个任意的确定算法 A 和输入 I,确定算法 A 对于 I 是

否会停机(或者进入一个无穷的循环)。这个问题是不可判定的,因此不存在任何(不管是什么样的复杂度)求解这个问题的算法,所以此问题显然不在 NP 内。为了证明可满足性正比于停机问题,只要简单地构造一个算法 A,它的输入是一个命题公式 X:如果 X 有 n 个变量,则 A 测试出所有的 2^n 组可能的真值,指派并验证 X 是否可满足。如果是可满足的则 A 停止;如果是不可满足的,则 A 进入无穷循环。因此对于输入 X,A 停止当且仅当 X 是可满足的。假如对于停机问题有一个多项式时间算法,那么就可以使用 A 并以 X 作为停机问题算法的输入在多项式时间内求解出可满足性问题。所以,停机问题是一个 NP-难度的问题而它又不在 NP 中。

定义 10.5 设有两个问题 L_1 和 L_2,如果 $L_1 \propto L_2$ 且 $L_2 \propto L_1$ 则称 L_1 和 L_2 为多项式等价。

为了证明问题 L_2 是 NP-难度的,只需证明 $L_1 \propto L_2$,其中 L_1 是某个已知其为 NP-难度的问题。因为$\propto$是一种传递关系,于是有可满足性$\propto L_1$ 和 $L_1 \propto L_2$,则可满足性$\propto L_2$。为了证明一个 NP-难度的判定问题是 NP-完全的,只要指出它存在多项式时间的不确定算法。下面几节将证明许多问题是 NP-难度的。尽管把讨论限制在判定问题上,然而十分明显的是它们相应的最优化问题也是 NP-难度的。对于那些确是 NP-完全问题的 NP-完全性证明则留作习题。

10.2 COOK 定理

COOK 定理(定理 10.1)指出,可满足性在 P 中当且仅当 P = NP 。本节来证明这个重要的定理。从例 10.9 中已经知道可满足性在 NP 中,因此若 P = NP 则可满足性就在 P 中。剩下来要证明的是,如果可满足性在 P 中则 P = NP。为了证明上述命题,现在来研究如何由任一多项式时间的不确定判定算法 A 和输入 I 来获得一个公式 Q(A,I),使得 Q 是可满足的当且仅当在输入为 I 的情况下 A 有一个成功的终止。如果 I 的长度为 n 以及对于某个多项式 P( ),A 的时间复杂度为 P(n),则 Q 的长度将是 $O(P^3(n)\log n) = O(P^4(n))$,构造 Q 需要的时间也将是 $O(P^3(n)\log n)$。这样就可以得到在任一输入 I 下确定 A 的结果的一个确定算法 Z。算法 Z 先计算 Q ,然后用一个对于可满足性问题的确定的算法来确定 Q 是否是可满足的。如果确定长度为 m 的公式是否可满足所需要的时间为 O(Q(m)),那么 Z 的复杂度为 $O(P^3(n)\log n + Q(P^3(n)\log n))$。如果可满足性在 P 中,则 Q(m)是 m 的一个多项式函数而且对于某个多项式 r( ),Z 的复杂度就变成 O(r(n))。因此,如果可满足性在 P 中,则对于在 NP 中的不确定算法 A 可以得到 P 中的一个确定算法 Z。于是由上面的解释可见,若可满足性在 P 中则 P = NP。

在进入由 A 和 I 构造 Q 的讨论之前,先对不确定机器模型和 A 的形式作某些简化的假设,这些假设丝毫不会改变判定问题在 NP 之中或者在 P 中的类别。这些简化假设如下:

(1) 执行算法 A 的机器是面向字的。每个字长为 w 位。在那些一个字长的数之间进行加、减、乘等运算占用一个单位时间。如果数的长度大于一个字,则对这些数进行相应的运算所需的时间一律按最长的那个数所需的时间单位来计算。

(2) 简单表达式是最多包含一个运算符并且所有操作数都是简单变量(即不能使用数

组变量)的表达式。简单表达式的一些例子为: - B, B + C, D or E, F。同时假定算法 A 中所有的赋值语句是如下形式中的一种:

(a) 〈简单变量〉←〈简单表达式〉;

(b) 〈数组变量〉←〈简单变量〉;

(c) 〈简单变量〉←〈数组变量〉;

(d) 〈简单变量〉←choice 〈S〉。

其中,S 可以是一个有穷集合$\{S_1, S_2, \cdots, S_k\}$,也可以是 l：u。在后一情况下 S 选择「l：u」范围内的一个整数。

用整型简单变量作数组的下标,并且规定:下标都是正的,只允许一维数组。显然,凡不属于上述各类型的赋值语句都可以用上述各类型的一组赋值语句来取代,因此这种限制并不改变 NP 的类别。

(3) A 中所有的变量都是整型或布尔型。

(4) A 不包含 read 和 print 语句。对 A 的输入只能通过它的参数进行。在调用 A 时,所有的变量(不是参数)的值为零(如果是布尔量则为 false)。

(5) A 不包含常数。显然任何算法中的常数都可以用一些新变量来取代。这些新变量可以加入到 A 的参数表中去,这样,有关的常数就可以看作输入的组成部分。

(6) 除了简单赋值语句外,A 只允许含有下列几种语句:

① go to k,其 k 是一个指令号。

② if c then go to a endif。c 是一个简单布尔变量(即不能是一个数组),a 是一个指令号。

③ success, failure, end。

④ A 可以含有类型说明和维数语句。这些语句在执行 A 时是不用的,所以不必把它们翻译到 Q 之中。维数信息用来给数组空间定位。假定数组中相继的元素被分配到存储器的依次相连的字中。

如果 A 有 l 条指令,则假定 A 中指令从 1 到 l 被顺序编号。A 中的每条语句都有一个编号。

①和②中的 go to 指令利用这些编号来实现转移。如何根据 go to k 和 if c then go to a endif 语句去改写 while - repeat ,loop - until - repeat, case - endcase, for - repeat 等语句是读者所熟知的。同样也应当注意到 go to k 也可以用 if true then go to k endif 来改写,由此可见 go to k 语句实际上也可以删去。

(7) 令 P(n)是一个多项式,使得 A 在长度为 n 的任一输入下所花的时间不超过 P(n)个时间单位。根据(1)中对复杂度的假设, A 不能改变或使用多于 P(n)个字的存储单元。可以假设 A 使用其下标为 1,2,3,…,P(n)的这些字的某个子集。这一假定并不会限制 NP 中判定问题的类别。为了弄清这一点,令 f(1), f(2),…, f(k), $1 \leq k \leq P(n)$,是 A 在输入 I 情况下工作时所用的那些不同的字。也可以构造另一个多项式时间的不确定算法 A′,它使用其下标为 1,2,3,…,2P(n)的 2P(n)个字并解决与 A 相同的判定问题。A′模拟 A 的动作,但 A′将地址 f(1), f(2),…, f(k)映射到集合{1,2,…,k} 上。所用的这个映射函数动态地被确定,并作为一个表存放在 P(n) + 1 到 2P(n)的字中。如果在字 P(n) + i 处的表值是 i 则 A′

用字 i 来存放的值是 A 用字 j 所存放的那个值。A 的模拟过程如下：令 k 是迄今为止 A 所访问的不同字的数目，j 是当前这一步 A 所访问的字。A′检索它的表以找到这样的一个字 P(n) + i, 1≤i≤k，使得该字的内容为 j。如果这个 i 不存在，则 A′置 k←k + 1，i←k 并将值 j 存到字 P(n) + k 中。A′对字 i 所做的一切正好就是 A 对字 j 所做的一切。显然 A′和 A 求解同一个判定问题。由于 A′为了模拟 A 的一步要花费 P(n)的时间来检索它的表，因此 A′的复杂度为 $O(P^2(n))$。因为 $P^2(n)$仍然是 n 的一个多项式，所以把算法限制在只能使用相连的字并不改变 P 和 NP 的类别。

公式 Q 要使用若干布尔变量。下面叙述在 Q 中使用的两组变量的语义：

(1) B(i,j,t),1≤i≤P(n),1≤j≤w,0≤t < P(n)。

B(i,j,t)表示 t 步计算(或 t 单位时间)之后字 i 的第 j 位的状态。一个字的位由右至左被编号，最右面的位被编号为 1。Q 将如此构造：在使 Q 为真的任一真值指派中，当且仅当在输入 I 的情况下，t 步之后 A 的某一成功计算使相应的位为 1，则 B(i,j,t)为真。

(2) S(j,t),1≤j≤l,1≤t≤P(n)。

前面对 l 曾作过假定，它是 A 的指令数。S(j,t)表示在时间 t 要执行的指令。Q 将如此构造：在使 Q 为真的任一真值指派中，当且仅当在 t 时刻 A 所执行的指令是 j，则 S(j,t)为真。

Q 将由 C,D,E,F,G 和 H 六个子公式组成，Q = C∧E∧F∧G∧H。这些子公式产生如下断言：

C　P(n)个字的初始状态表示输入 I。所有非输入变量均为 0。

D　指令 1 是要执行的第一条指令。

E　在第 i 步的末端，下一条要执行的指令只能有一条。因此，对于任何固定的 i，S(j,i)1≤j≤l，之中至多一个为真。

F　如果 S(j,i)为真且 j 是一条 success,failure 或 end 语句，那么 S(j,i + 1)也为真。如果 j 是一条赋值语句，则 S(j + 1,i + 1)为真。如果 j 是一条 go to k 语句，则 S(k,i + 1)为真。j 的最后一种可能性是 if c then a endif 语句。在这种情况下，若 c 为真则 S(a,i + 1) 为真；若 c 为假则 S(j + 1,i + 1) 为真。

G　若在第 t 步所执行的指令不是一条赋值语句，则 B(i,j,t)不变；若这条指令是赋值语句且左部变量为 X，则只有 X 可以变化。这一变化由这条指令的右部确定。

H　在时间 P(n)执行的指令是一条 success 指令，于是计算成功地终止。

显然如果 C 到 H 产生上述断言，当且仅当在输入 I 的情况下 A 存在一个成功的计算时，Q = C∧D∧E∧F∧G∧H 是可满足的。现在来给出从 C 到 H 的公式，同时还指出可以怎样把它们转换成 CNF。这种转换将使 Q 的长度增加一个不依赖于 n 的总量(但依赖于 w 和 l)，这就能使我们证明 CNF-可满足性是 NP-完全的。

(1) 公式 C 描述输入 I。有

$$C = \bigwedge_{\substack{1\le j\le P(n)\\ 1\le j\le w}} T(i,j,0)$$

如果输入要求位 B(i,j,0)(即字 i 的第 j 位)为 1，则 T(i,j,0)为 B(i,j,0)；否则 T(i,j,0)

为 $\overline{B}(i,j,0)$。于是,如果没有输入,则

$$C = \bigwedge_{\substack{1\le i\le P(n)\\ 1\le j\le w}} \overline{B}(i,j,0)$$

显然,C 由 I 唯一地确定并且是 CNF 的形式。同时,只有用一组真值指派表示 A 中所有变量的初值时 C 才是可满足的。

(2) $D = S(1,1) \wedge \overline{S}(2,1) \wedge \overline{S}(3,1) \wedge \cdots \wedge \overline{S}(l,1)$

显然,只有在指派 $S(1,1)$ = 真且 $S(i,1)$ = 假,$2 \le i \le l$ 时,D 是可满足的。使用对于 $S(i,1)$的解释,这意味着当且仅当编号为 1 的指令最先被执行时 D 为真。注意 D 是以 CNF 表示的。

(3) $E = \bigwedge_{1 < t \le P(n)} E_t$

每个 E_t 将断言在第 t 步只存在唯一的指令。可把 E_t 定义为

$$E_t = (S(1,t) \vee S(2,t) \vee \cdots \vee S(l,t)) \wedge \left(\bigwedge_{\substack{1\le j\le l\\ 1\le k\le l\\ j\ne k}} (\overline{S}(j,t) \vee \overline{S}(k,t))\right)$$

可以证明当且仅当 $S(j,t), 1 \le j \le l$,之中恰好有一个为真时 E_t 为真。注意 E 也是 CNF 形式。

(4) $F = \bigwedge_{\substack{1\le i\le l\\ 1\le t< P(n)}} F_{i,t}$

每个 $F_{i,t}$ 都断言,或者指令 i 不是一条在时刻 t 要执行的指令;若是,则由指令 i 正确地确定在时刻 t + 1 要执行的那条指令。形式上有

$$F_{i,t} = \overline{S}(i,t) \vee L$$

其中,L 定义如下:

① 如果指令 i 是 success, failure 或 end 时,则 L 为 $S(i,t+1)$。可见程序离不开这样的一条指令。

② 如果指令 i 是 go to k,则 L 为 $S(k,t+1)$。

③ 如果指令 i 是 if X then go to k endif 并且变量 X 用字 j 所表示,则 L 为$((B(j,1,t-1) \wedge S(k,t+1)) \vee (\overline{B}(j,1,-1) \wedge S(i+1,t+1)))$。这就是假设当且仅当 X 为真时 X 的第 1 位为 1。

④ 如果指令 i 不是上述的任何一种形式,则 L 为 $S(i+1,t+1)$。

按照①,②和④所定义的 $F_{i,t}$ 都是 CNF 形式。在③中的 $F_{i,t}$ 可以运用布尔恒等式 $a \vee (b \wedge c) \vee (d \wedge e) \equiv (a \vee b \vee d) \wedge (a \vee c \vee d) \wedge (a \vee b \vee e) \wedge (a \vee c \vee e)$转换成 CNF 形式。

(5) $G = \bigwedge_{\substack{1\le i\le l\\ 1\le t\le P(n)}} G_{i,t}$

每个 $G_{i,t}$ 断言:在 t 时刻,或者① 指令 i 不被执行;或者② 指令 i 被执行,并且在第 t 步后这 P(n)个字的状态相对于第 t 步前的状态和由执行指令 i 所引起的状态变化来说是正确的。形式上有

$$G_{i,t} = \overline{S}(i,t) \vee M$$

其中,M 定义如下:

① 若指令 i 是一条 go to, if—then go to—endif, success, failure 或 end 语句,则 M 断言这 P(n)个字的状态不变。即 $B(k,j,t-1) = B(k,j,t)$, $1 \le k \le P(n)$ 和 $1 \le j \le w$。

$$M=\bigwedge_{\substack{1\le k\le P(n)\\1\le j\le w}}((B(k,j,t-1)\wedge B(k,j,t))\vee(\overline{B}(k,j,t-1)\wedge\overline{B}(k,j,t)))$$

在这种情况下,$G_{i,t}$可以改写成

$$G_{i,t}=\bigwedge_{\substack{1\le k\le P(n)\\1\le j\le w}}(\overline{S}(i,t)\vee(B(k,j,t-1)\wedge B(k,j,t))\vee(\overline{B}(k,j,t-1)\wedge\overline{B}(k,j,t)))$$

$G_{i,t}$中每个子句都是这样的形式:$z\vee(x\wedge s)\vee(\overline{x}\wedge\overline{s})$,其中 z 是 $\overline{S}(i,t)$,x 代表 B(, ,t-1)而 s 代表 B(, ,t)。注意,$z\vee(x\wedge s)\vee(\overline{x}\wedge\overline{s})$等价于$(x\vee\overline{s}\vee z)\wedge(\overline{x}\vee s\vee z)$,因此 $G_{i,t}$可以很容易地转换为 CNF 的形式。

② 如果 i 是一条(a)型的赋值语句,则 M 由右部的运算符(如果有的话)决定。首先来描述当 i 是 Y←V+Z 型的指令时 M 的形式。令 Y,V 和 Z 分别出现在字 y、v 和 z 中。为简化起见,假设所有的编号都是非负的数。为了获得一个公式,它断言位 B(y,j,t),1≤j≤w,表示 B(v,j,t-1)及 B(z,j,t-1),1≤j≤w237 中部的和,则必须使用 w 个附加位 C(j,t),1≤j≤w。C(j,t)表示由位 B(v,j,t-1),B(z,j,t-1)和 C(j-1,t)相加而产生的进位,1<j≤w。C(1,t)是由 B(v,1,t-1)和 B(z,1,t-1)相加而产生的进位。回顾前面的叙述,当且仅当与某位相应的变量为真时该位为 1。在对 V 和 Z 施行按位加时,得到 C(1,t)=B(v,1,t-1)∧B(z,1,t-1)和 B(y,1,t)=B(v,1,t-1)⊕B(z,1,t-1),其中⊕是异或运算(a⊕b为真,当且仅当 a 和 b 中仅有一个为真)。注意到 $a\oplus b\equiv(a\vee b)\wedge\overline{(a\wedge b)}\equiv(a\vee b)\wedge(\overline{a}\vee\overline{b})$,因此 B(y,1,t)表达式的右边可以用这个恒等式转换成 CNF。对于 Y 的其它位,可以证明

$$B(y,j,t)=B(v,j,t-1)\oplus(B(z,j,t-1)\oplus C(j-1,t))$$

以及

$$C(j,t)=(B(v,j,t-1)\wedge B(z,j,t-1))\vee(B(v,j,t-1)\wedge C(j-1,t))\vee(B(z,j,t-1)\wedge C(j-1,t))$$

最后,还要求 C(w,t)=假(亦即没有溢出)。令 M′是对于 B(y,j,t)和 C(j,t),1≤j≤w,所有式子的 and,M 由下式给出:

$$M=(\bigwedge_{\substack{1\le k\le P(n)\\k\ne y\\1\le j\le w}}((B(k,j,t-1)\wedge B(k,j,t))\vee(\overline{B}(k,j,t-1)\wedge\overline{B}(k,j,t)))\wedge M'$$

采用 5①的思想可以把 $G_{i,t}$转换成 CNF。这一转换将使 $G_{i,t}$的长度增加一个不依赖于 n 的常数倍,下列问题则留给读者考虑:当指令 i 是 Y←V;Y←V⊛Z(其中⊛是如下算符中的一个:—,/ ,* ,<,>,≤,=等)这几种形式之一时,M 是什么?

当 i 是一条(b)或(c)型的赋值语句时,则它必须选择正确的数组元素,考察一条(b)型指令:R(m)←X。此时公式 M 可以写成:

$$M=W\wedge(\bigwedge_{1\le j\le u}M_j)$$

其中,u 是 R 的维数。注意,由于对算法 A 的限制⑦,因此 u≤P(n)。W 断言 1≤m≤u 。对 W 的详细说明则留作习题,每个 M_j 断言,或者 m≠j,或者 m=j 且只有 R 的第 j 个元素改变。假设 X 和 M 的值分别存放在字 x 和 m 中,并且 R(1∶m)存放在字 α,α+1,…,α+u-1 中,M_j 由下式给出:

$$M_j=\bigvee_{1\le k\le w}T(m,k,t-1)\vee Z$$

其中,若 j 的二进制表示中的第 k 位为 0,则 T 是 B,否则 T 是 $\overline{B}$。Z 定义为

$$Z=\bigwedge_{\substack{1\le k\le w\\ 1\le r\ne P(n)\\ r\ne \alpha+j+1}}((B(r,k,t-1)\wedge B(r,k,t))\vee\overline{B}(r,k,t-1)\wedge\overline{B}(r,k,t)))$$

$$\bigwedge_{1\le k\le w}((B(\alpha+j-1,k,t)\wedge B(x,k,t-1))\vee(\overline{B}(\alpha+j-1,k,t)\wedge\overline{B}(x,k,t-1)))$$

注意:M 中的文字数目是 $O(P^2(n))$,由于 j 是 w 位长,因此它只能表示小于 2^w 的数。于是对于 $u\ge 2^w$ 的情况需使用别的下标方案,一种简单的推广是允许多精度运算,下标变量 j 需要多少字就用多少字,所用字数取决于 u,它至多需要 log(P(n))个字,这会在 M_j 内引起少许的变化,但 M 中文字的数目仍然保持在 $O(P^2(n))$。由于程序在存取多精度下标 j 中单个的字时可以要求此程序模拟多精度运算,因此不需要明显地引入多精度运算。

当 i 是一条(c)型指令时,M 的形式与(b)型指令所得到的类似。下面讨论在如下情况时如何来构造 M,这里 i 的形式为 Y←choice(S),其中 S 既可是形如 $S=\{S_1,S_2,\cdots,S_k\}$的集合,也可是 r：u 这样的形式,假设 Y 由字 y 来提供。如果 S 是一个集合,则定义

$$M=\bigvee_{1\le j\le k}M_j$$

M_j 断言 Y 是 S_j。通过选择 $M_j=a_1\wedge a_2\wedge\cdots\wedge a_w$,这一点是很容易做到的,其中若 S_j 中的第 l 位是 1 则 $a_i=B(y,l,t)$,若 S_j 的第 l 位是 0 则 $a_i=\overline{B}(y,l,t)$。如果 S 的形式为 r：u,那么 M 是断言 $r\le Y\le u$ 的公式(这一点留作习题)。在上述两种情况下 $G_{j,t}$ 可以转换成 CNF,而 $G_{j,t}$ 的长度至多增加一个常量。

(6)令 $i_1,i_2,\cdots,i_k$ 为对应于 A 中成功语句的语句编号,则 H 由下式给出:

$$H=S(i_1,P(n))\vee S(i_2,P(n))\vee\cdots\vee S(i_k,P(n))$$

易证,当且仅当在输入 I 的情况下算法 A 的计算成功地终止时,$Q=C\wedge D\wedge E\wedge F\wedge G\wedge H$ 是可满足的。此外,根据前面所述,Q 可转换成 CNF 形式。公式 C 含有 wP(n)个文字,D 含有 l 个文字,E 含有 $O(l^2P(n))$个文字,F 含有 O(lP(n))个文字,G 含有 $O(lwP^3(n))$个文字,H 至多含有 l 个文字。当 lw 为常数时,Q 中出现的文字的总和为 $O(lwP^3(n))=O(P^3(n))$。由于 Q 中有 $O(wP^2(n)+lP(n))$个不同的文字,因此每个文字可用 $O(\log(wP^2(n)+lP(n)))=O(\log n)$位写出来。因为 P(n)至少为 n,所以 Q 的长度为 $O(P^3(n)\log n)=O(n^4)$。由 A 和 I 构造 Q 的时间也是 $O(P^3(n)\log n)$。

上面的结构表明,NP 中的每一个问题可约化为可满足性问题,也可约化为 CNF-可满足性问题。因此若这两个问题的任何一个在 P 中,则 NP　P,从而 P = NP。另外,可满足性是在 NP 中,因此由构造公式 Q 的 CNF 表明可满足性∝CNF-可满足性。将此和 CNF-可满足性在 NP 中加在一起则意味着 CNF-可满足性是 NP-完全的。

注意:由于可满足性∝可满足性且可满足性在 NP 之中,故可满足性也是 NP-完全的。

10.3　NP-难度的图问题

用来证明一个问题 L_2 具有 NP-难度的策略如下:

(1) 挑选一个已知其具有 NP-难度的问题 L_1。

(2) 证明如何从 L_1 的任一实例 I(在多项式确定时间内)获得 L_2 的一个实例 I′,使得从

I 的解能(在多项式确定时间内) 确定 L_1 实例 I 的解。

(3) 从(2)得出结论 $L_1 \propto L_2$。

(4) 由(1),(3)及∝的传递性得出结论 L_2 是 NP-难度的。

在下面的叙述中,对于头几个证明将完全按上述 4 步进行,以后的证明则只进行(1)和(2)两步,一个具有 NP-难度的判定问题 L_2 可以通过展示一个求解 L_2 的具有多项式时间的不确定算法来证明它是 NP-完全的。下面所涉及的 NP-难度的判定问题都是 NP-完全的。对其中一些问题构造多项式时间的不确定算法则留作习题。

10.3.1 集团判定问题(CDP)

在 10.1 节已介绍过集团判定问题,在定理 10.2 中将证明 CNF-可满足性∝CDP。利用可满足性∝CNF-可满足性及∝的可传递性,就可容易地建立关系:可满足性∝CDP,于是 CDP 是 NP-难度的。由于 CDP∈NP,所以 CDP 也是 NP-完全的。

定理 10.2 CNF-可满足性∝集团判定问题(CDP)。

证明 令 $F = \bigwedge_{1 \le i \le k} C_j$ 是一个 CNF 形式的命题公式,$x_j, 1 \le i \le n$,是 F 中的变量。现证如何由 F 构造图 G = (V,E),使得当且仅当 F 是可满足的,则 G 就有一个大小至少为 k 的集团。如果 F 的长度为 m,则在 O(m)时间内可由 F 获得 G。因此,若对于 CDP 有一个多项式时间算法,则使用这一构造法就能得到一个对于 CNF-可满足性的多项式时间算法。

对于任意的 F,G = (V,E)被定义如下:$V = \{\langle \sigma, i\rangle \mid \sigma$ 是子句 C_j 的一个文字$\}$;$E = \{(\langle \sigma, i\rangle, \langle \delta, j\rangle) \mid i \neq j$ 且 $\sigma \neq \delta\}$。例 10.11 给出了这种构造的一个样本。

如果 F 是可满足的,则对于 $x_i, 1 \le i \le n$,存在一组真值指派,使得在这组指派下每个子句为真。因此,在这组指派下,在每个 C_i 中至少存在一个文字 σ 为真,令 $S = \{\langle \sigma, i\rangle \mid \sigma$ 在 C 中为真$\}$ 是对于每个 i 恰好只含有一个 $\langle \sigma, i\rangle$ 的集合。S 形成 G 内一个大小为 k 的集团。类似地,如果 G 有一个大小至少为 k 的集团 K = (V′,E′),则令 $S = \{\langle \sigma, i\rangle \mid \langle \sigma, i\rangle \in V'\}$。显然,当 G 不再有大于 k 的集团时 |S| = k。更进一步,若 $S' = \{\sigma \mid \langle \sigma, i\rangle \in S$, 对于某些 i$\}$,则由于 G 中 $\langle \delta, i\rangle$ 和 $\langle \bar{\delta}, j\rangle$ 之间无边相连,故 S′不能同时含有文字 δ 和它的补 $\bar{\delta}$。于是,若 $x_j \in S'$ 就置 x_i = 真,若 $\bar{x}_i \in S'$ 就置 x = 假,而对不在 S′中的变量则选取任意的真值就可使 F 中的所有子句得到满足。因此,当且仅当 G 有一个大小至少为 k 的集团时 F 是可满足的,证毕。

例 10.11 考察 $F = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$。根据定理 10.2 的构造法生成的图如图 10.1 所示。

这个图含有 6 个大小为 2 的集团。考虑具有结点$\{\langle x_1, 1\rangle, \langle \bar{x}_2, 2\rangle\}$的那个集团,通过置 x_1 = 真和 $\bar{x}_2$ = 真(即 x_2 = 假) 使 F 被满足。x_3 既可置为真也可置为假而对 F 的可满足性无任何影响。

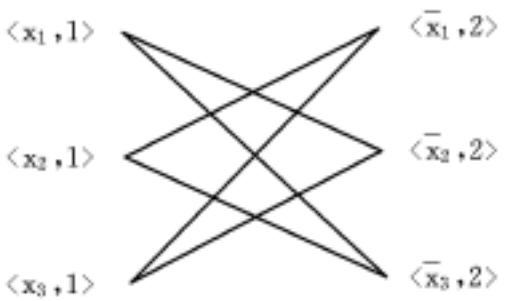


图 10.1 图和可满足性的一个样本

10.3.2 结点覆盖的判定问题

对于图 G = (V,E),集合 S V,如果 E 中所有的边都至少有一个结点在 S 中,则称 S 是图 G 的一个结点覆盖,覆盖的大小 |S| 是 S 中的结点数。

例 10 .12　考察图 10 2 中的图: S = {2,4}是大小为 2 的一个结点覆盖, S = {1,3,5}是大小为 3 的一个结点覆盖。

结点覆盖判定问题(node cover decision problem)简记为 NCDP,它是对给定的图 G 和一个整数 k,判定 G 是否有大小至多为 k 的结点覆盖。

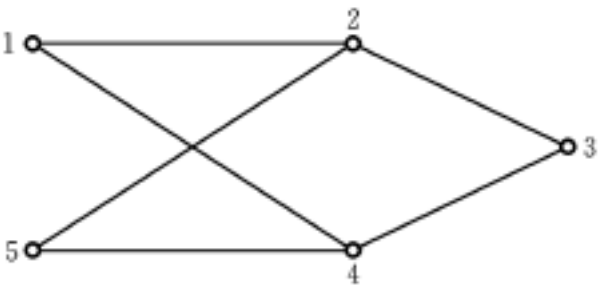


图 10 .2　图与结点覆盖的一个样本

定理 10 .3　集团判定问题(CDP)∝结点覆盖判定问题(NCDP)。

证明　令 G = (V, E)和 k 定义一个 CDP 的实例,假设|V| = n。下面来构造一个图 G′,使得当且仅当 G 有一个大小至少为 k 的集团时 G′有一个大小至多为 n - k 的结点覆盖。图 G′给出如下: $G' = (V, \overline{E})$,其中 $\overline{E} = \{(u,v) \mid u \in V, v \in V \text{ 且 } (u,v) \text{ 殛} E\}$。

现证当且仅当 G 有一个大小至少为 k 的集团时, G′有一个大小至多为 n - k 的结点覆盖。令 K 是 G 中任一大小为 k 的集团。由于不存在 $\overline{E}$ 中的边会连接 K 中的结点,因此剩在 G 中的 n - k 个结点必定覆盖 $\overline{E}$ 中所有的边,即 G′有一个大小至多为 n - k 的结点覆盖。可以类似地证明,若 S 是 G′的一个结点覆盖,则 V - S 必定形成 G 中的一个完全子图。

由于 G′可以在多项式时间内从 G 获得,因此,如果对 NCDP 有一个多项式时间的确定算法,则 CDP 可以在多项式确定时间内解出。证毕。

注意:由于 CNF-可满足性∝CDP, CDP∝NCDP 且∝是传递的,因此可得 NCDP 是 NP-难度的。

10 .3 .3　着色数判定问题(CN)

图 G = (V, E)的着色是对于所有的 $i \in V$ 所定义的一个函数 $f: V \to \{1, 2, \cdots, k\}$。如果 $(u,v) \in E$,则 $f(u) \neq f(v)$。着色数判定问题(chromatic number decision problem)简记为 CN ,它是对于某个给定的 k,确定是否能对 G 着色。

例 10 .13　图 10 2 所示的要成为二色图的可能方案是 f(1) = f(3) = f(5) = 1,而 f(2) = f(4) = 2。显然这个图不可能是 1-着色的。

为证明 CN 是 NP-难度的要用到另一个 NP-难度的问题 SATY。SATY 问题是带有以下限制的 CNF-可满足性问题,它只允许 CNF 的命题公式中每个子句至多有 3 个文字。证明 CNF-可满足性∝SATY 则留下作为习题。

定理 10 .4　每个子句至多有 3 个文字的可满足性 (SATY)∝着色数判定问题(CN)。

证明　令 F 是一个有 r 个子句且每个子句内至多有 3 个文字的 CNF 形式的命题公式。令 x_i, $1 \leq i \leq n$,是 F 中的 n 个变量。可以假定 $n \geq 4$(否则若 $n < 4$,那么可以通过对 x_1, x_2 和 x_3 的八组可能的真值指派做彻底的试验来确定 F 是否可满足),当且仅当 F 是可满足的,则可在多项式时间内构造出一个 n + 1 可着色的图 G。图 G = (V, E)定义为

$$V = \{x_1, x_2, \cdots, x_n\} \cup \{\overline{x_1}, \overline{x_2}, \cdots, \overline{x_n}\} \cup \{y_1, y_2, \cdots, y_n\} \cup \{C_1, C_2, \cdots, C_r\}$$

$$E = \{(x_i, \overline{x_i}), 1 \leq i \leq n\} \cup \{(y_i, y_j) \mid i \neq j\} \cup \{(y_i, x_j) \mid i \neq j\}$$
$$\cup \{(y_i, \overline{x_j}) \mid i \neq j\} \cup \{(x_i, C_j) \mid x_i \text{ 殛} C_j\} \cup \{(\overline{x_j}, C_j) \mid \overline{x_i} \text{ 殛} C_j\}$$

为了弄清当且仅当 F 是可满足的,则 G 是 n + 1 可着色的,首先要看到所有的 y_i 形成一个有 n 个结点的完全子图,因此,每个 y 必须分配一种不同的颜色。不失一般性,可以假设 G 中 y_i 的着色就用颜色 i,因为 y_i 也与除了 x_i 和 $\overline{x}_i$ 以外的所有 x_j 和 $\overline{x}_j$ 相连接,所以颜色 i 只能再分给 x_i 和 $\overline{x}_i$。然而$(x_i, \overline{x}_i) \in E$,因此,这两个结点中有一个需要分配一种新的颜色 n + 1,不妨将分配新颜色 n + 1 的那个结点称为“假”结点,其它结点称为“真”结点。用 n + 1 种颜色对 G 着色的唯一方法是对于每一个 i,将新颜色 n + 1 着在$\{x_i, \overline{x}_i\}$的一个之上,$1 \leq i \leq n$。

在这种情况下,其余的结点是否可不再用更多的颜色来着色呢?回答是肯定的,因为 $n \geq 4$ 且每个子句至多有 3 个文字,所以每个 C_i 至少与一对结点 x_j 和 $\overline{x}_i$ 相连接。从而没有哪个 C_i 可以着以颜色 n + 1,也没有哪个 C_i 可以着以不在 C_i 中的 x_j 或 $\overline{x}_j$ 所着的颜色。上述两句话意味着仅能给 C_i 着的颜色是在子句 C_i 中那些被称为“真”结点的 x_j 或 $\overline{x}_j$ 所着的颜色。因此,当且仅当对应于每个 C_j 存在一个“真”结点,G 是 n + 1 可着色的。故当且仅当 F 是可满足的,G 是 n + 1 可着色的。证毕。

10.3.4 有向哈密顿环(DHC)

有向图 G = (V, E)中的一个哈密顿环是一个长度为 n = |V| 的有向环,它恰好经过每个结点一次,然后回到起始的结点,DHC 问题是确定 G 是否有一个有向哈密顿环。

例 10.14 图 10.3 中 1,2,3,4,5,1 是一个有向哈密顿环。

若把边〈5,1〉从图中删去,则它就没有有向哈密顿环。

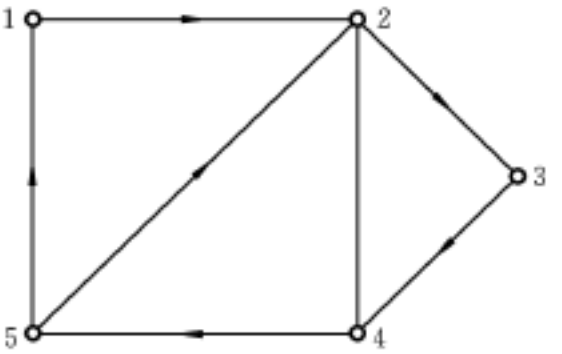


图 10.3 图和哈密顿环的样本

定理 10.5 CNF-可满足性∝有向哈密顿环(DHC)。

证明 令 F 是 CNF 形式的命题公式。下面将设法构造出一有向图 G,使得当且仅当 G 存在一向哈密顿环时,F 可满足。由于这一构造可在 F 大小的多项式时间内完成,因此 CNF-可满足性∝DHC。用一个例子来理解 G 的构造可带来极大的方便。所采用的例子是

$$F = C_1 \wedge C_2 \wedge C_3 \wedge C_4。$$

其中, $C_1 = x_1 \vee \overline{x}_2 \vee x_4 \vee \overline{x}_5$, $C_2 = \overline{x}_1 \vee x_2 \vee x_3$, $C_3 = \overline{x}_1 \vee \overline{x}_3 \vee x_5$, $C_4 = \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3 \vee \overline{x}_4 \vee \overline{x}_5$。

假设 F 有 r 个子句 $C_1, C_2, \cdots, C_r$ 和 n 个变量 $x_1, x_2, \cdots, x_n$。画一个有 r 行和 2n 列的数组,第 i 行表示子句 C_i,每个变量 x_i 由两个相邻的列表示,其中一列表示文字 x_i,另一列表示文字$\overline{x}_i$。图 10.4 列出了这一数组。当且仅当 x_i 是 C_j 中的一个文字,就在列$\overline{x}_i$和行 C_j 处插入一个⊛。同样,当且仅当$\overline{x}_i$是 C_j 中的一个文字,就在列$\overline{x}_i$和行 C_j 处插入一个⊛。在 x_i 和$\overline{x}_i$的每对列之间引入两个结点 u_i 和 v_i,u_i 在列的顶端而 v_i 在列的底部。对于每个 i,从 u_i 向上到 v_j,画两条由边组成的链,一条把列 x_i 的所有⊛连接起来,另一条则连接起 $\overline{x}_i$ 的所有⊛(参看图 10.4)。然后再画边$\langle u_i, v_{i+1} \rangle$,$1 \leq i \leq n$。在每一行 C_i 的右端引入一个方框 $\boxed{i}$,$1 \leq i \leq r$,画边$\langle u_r, \boxed{1} \rangle$ 和$\langle \boxed{r}, v_1 \rangle$,再画边$\langle \boxed{i}, \boxed{i+1} \rangle$,$1 \leq i < r$。

为了最终完成这个图,还要将每个⊛和 $\boxed{i}$ 各用一个子图来取代。其中,每个⊛用图 10.5

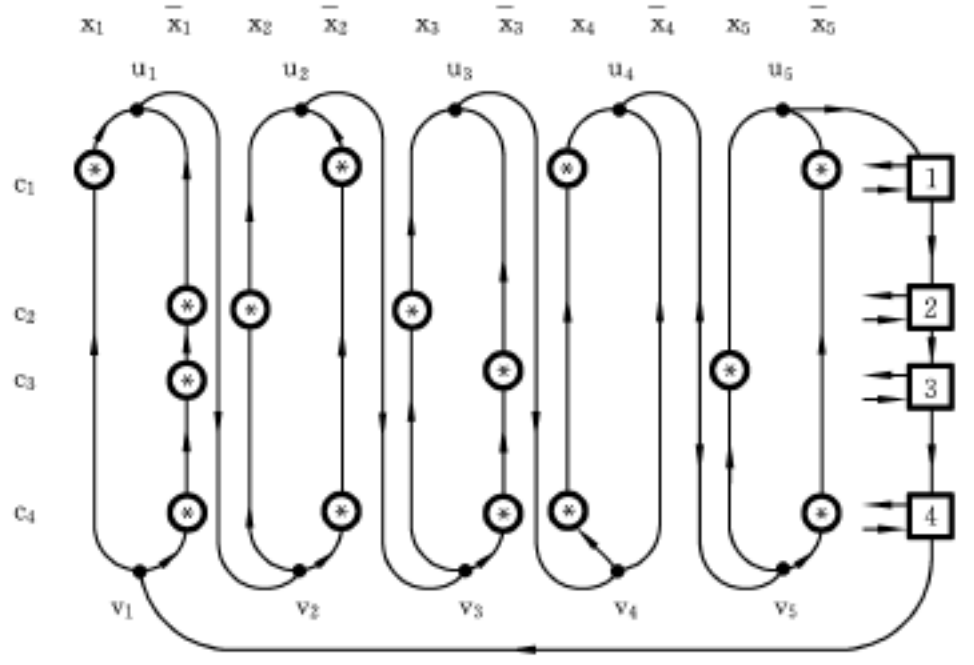


图 10.4　定理 10.5 中公式的数组结构

左边所示的子图取代(当然,对于这个子图的所有副本每个结点的标号应各不相同)。每个 i 用图 10.6 所示的子图取代之,在这个子图中 A_i 是入口结点,B_i 是出口结点。前面提到的边 〈i, i + 1〉 实际上就是 〈B_i, A_{i+1}〉。边 〈u_r, 1〉 是 〈u_r, A_1〉, 〈r, v_1〉 是 〈B_r, v_1〉。i_j 是子句 C_i 中文字的数目。图 10.6 中所示子图的一条边

$$R_{i\,a} \rightarrow ⊛ \rightarrow R_{i_{a+1}}$$

是指与 C_i 行中一个子图⊛相连接,R_{i_a} 连接到这个⊛的"1"结点,而 $R_{i_{a+1}}$ (或者,若 a = j 则 R_{i_1})从"3"结点引入。因此,在图 10.5 右边所示的⊛子图中,w_1 和 w_3 分别是"1"和"3"结点。入边是 〈R_{i1}, w_1〉,出边是 〈w_3, R_{i2}〉。这样就完成了图 G 的构造。

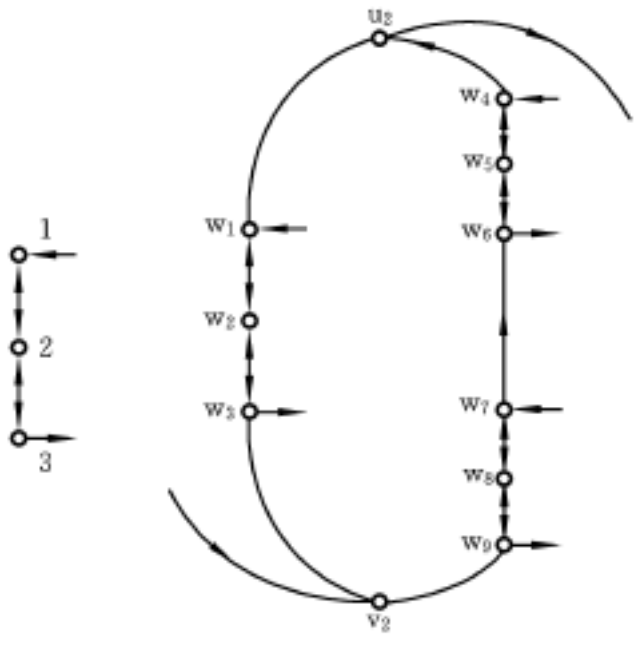


图 10.5　⊛子图及其向第 2 列的插入

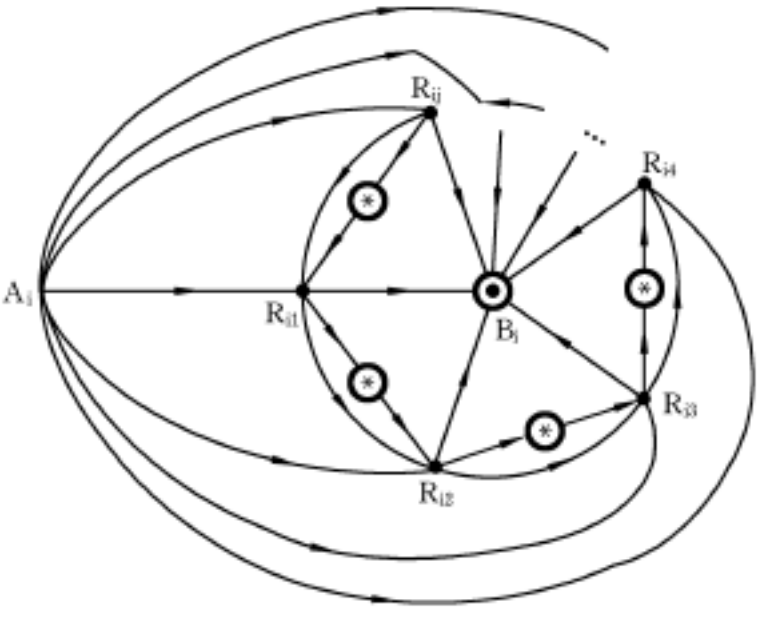


图 10.6　i 子图

若 F 是可满足的则令 S 是使 F 为真的一组真值指派。G 的一个哈密顿环可从 v_1 开始,行进到 u_1,然后 v_2;再 u_2, v_3;再 u_3, …, u_r。在由 v_i 向上行进到 u_i 时,若 x_i 在 S 中为真,则这个环采用相应于 $\bar{x}_i$ 的那一列;否则就沿相应于 $\bar{x}_i$ 的列向上行进。这个环从 u_r 将行进到 A_1,

然后经过 R_{1_1}, R_{1_2}, R_{1_3}, …, R_{1_j}, B_1 到 A_2, …, 到 v_1。在由任一子图 i 的 R_{i_a} 行进到 $R_{i_{a+1}}$ 的过程中,当且仅当某个⊛子图的结点还不在 v_1 到 R_{i_a} 的路径上时,则转移到第 i 行的⊛子图。注意,若 C_i 有 i_j 个文字,则 i 的结构至多允许转移到 ij - 1 个⊛子图,这是因为在 C_i 行中必须至少已通过了一个⊛子图(由于至少应有这样的一个子图与一个取真值的文字相对应)。从而,若 F 是可满足的,则 G 有一个有向哈密顿环。下面证明,若 G 有一个有向哈密顿环则 F 是可满足的。从 G 的任一哈密顿环上的结点 v_1 开始,由于子图⊛及 i 的结构,因此这样的环必须向上恰好沿每一对 $(x_i, \bar{x}_i)$ 中的一列行进。另外,环的这一部分在每一行必须至少经过一个⊛子图。因此,用于从 v_i 行进到 u_i $(1 \leq i \leq n)$ 的那些列定义了一组使 F 为真的真值指派。

由上述证明可得出结论,当且仅当 G 有一哈密顿环,则 F 是可满足的。同时从观察可知,在多项式时间内从 F 可以构造出 G。证毕。

10.3.5 货郎担判定问题(TSP)

在第 6 章介绍了货郎担问题,其对应的判定问题是确定一个带有边成本为 c(u,v) 的完全有向图 G = (V, E) 是否有一条总成本至多为 M 的周游路线。

定理 10.6 有向哈密顿环(DHC)∝货郎担判定问题(TSP)。

证明 从有向图 G = (V, E) 构造一完全有向图 G′ = (V, E′), E′ = {⟨i, j⟩ | i ≠ j} 而且若 ⟨i, j⟩ ∈ E 则 c(i, j) = 1, 若 i ≠ j 且 ⟨i, j⟩ 爫E 则 c(i, j) = 2。显然,当且仅当 G 有一个有向哈密顿环则 G 有一条成本至多为 n 的周游路线。证毕。

10.3.6 与/或图的判定问题(AOG)

与/或图曾在 7.4 节作过介绍。假设图中的每条边都有一个成本。与/或图 G 的解图 H 的成本是 H 中各边成本的总和。与/或图判定问题(AOG)是对于给定的输入 k,确定 G 是否有成本至多为 k 的解图。

例 10.15 考察图 10.7 所示的有向图。P_1 是所要求解的问题。因为 P_1 是一个或结点,所以可通过求解结点 P_2 或者 P_3 或者 P_7 来求解,由此承担的成本(即除了求解 P_2, P_3 和 P_7 之一所花成本之外的成本)分别为 2, 2, 8。P_2 是一个与结点,要求解它必须求解 P_4 和 P_5,完成这一工作的总成本为 2。为了求解 P_3,可以求解 P_5 或 P_6,完成此工作的最小成本为 1。于是求解 P_1 的最优方法是先求解 P_6 然后 P_3 最后 P_1。这个解的总成本是 3。

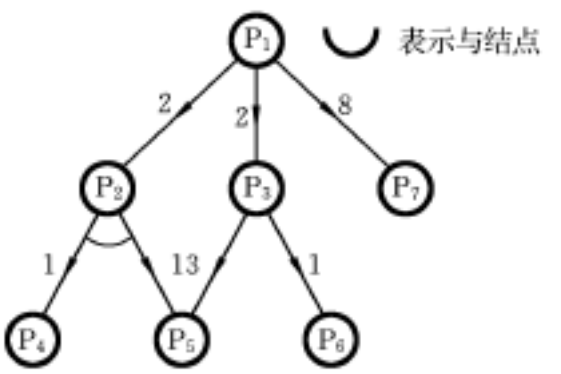


图 10.7 与/或图

定理 10.7 CNF-可满足性∝与/或图判定问题(AOG)。

证明 令 P 是一 CNF 形式的命题公式,现证如何把一个 CNF 形式的公式转换成一个与/或图,使得当且仅当 P 可满足时,此与/或图有一确定的最小成本解。令

$$P = \bigwedge_{i=1}^{k} C_i \qquad C_i = \bigvee l_j$$

其中,所有的 l_j 都是文字。P 的变量 V(P) 是 $x_1, x_2, \cdots, x_n$。此与/或图的结点情况如下:

(1) 有一个特殊的结点 S,它没有入弧。这个结点代表要求解的问题。

(2) S 是一个与结点,它具有子孙结点 P, x_1, x_2, …, x_n。

(3) 每个结点 x_i 表示公式 P 中对应的变量 x_i。每个 x_i 是一个或结点,它带有分别以 Tx_i 和 Fx_i 表示的两个子孙。如果求解 Tx_i 则相当于给变量 x_i 赋一个真值"真"。求解结点 Fx_j 相当于给变量赋一个真值"假"。

(4) 结点 P 代表公式 P,它是一个与结点。P 有 k 个子孙 C_1, C_2, …, C_k。结点 C 相当于公式 P 中的子句 C_i。所有的 C_i 结点都是或结点。

(5) 形如 Tx_i 或 Fx_i 的每个结点恰好只有一个子孙结点,那就是终结点。这些终结点用 v_1, v_2, …, v_{2n} 表示。

为了完成与/或图的构造,还要增补以下的一些边和成本:

(1) 如果 $\bar{x}_j$ 在子句 C_j 中出现,则从每个结点 C_i 增补一条边(C_i, Tx_j)。如果 $\bar{x}_j$ 在子句 C_j 中出现则增补一条边〈C_i, Fx_j〉。对于出现于子句 C_i 的所有变量 x 都要完成上述工作。C_i 指定为或结点。

(2) 规定从 Tx_i 或 Fx_i 类型的结点到它们各自的终结点的边的权或成本都为 1。

(3) 其它所有边的成本为 0。

为了求解 S,必须求解 P, x_1, x_2, …, x_n 的每一个结点。求解结点 x_1, x_2, …, x 的总成本是 n。为了求解 P,必须解出所有的结点 C_1, C_2, …, C_k。求解一个 C_j 结点的成本至多为 1。然而,若 C_i 的子孙之一已在求解结点 x_1, x_2, …, x_n 时解出,则求解 C_j 所添加的成本为 0,这是因为到它子孙结点的那些边的成本为 0 且其子孙之一已被求解。换言之,如果出现于子句 C_i 的文字之一已被赋予"真"值时结点 C 就可以无成本地求解。由此可得:只要对这些 x_i 存在某组真值指派,使得在每个子句中至少有一个文字的值为真,即公式 P 是可满足的,那么整个图(即结点 S)能在成本为 n 的情况下求解。如果 P 不可满足,则成本就会大于 n。

至此,已经揭示了如何从公式 P 出发构造一个与/或图,使得当且仅当 P 是可满足的时候所构造的与/或图有成本为 n 的解;否则成本就大于 n。显然构造这样的与/或图只需多项式时间。证毕。

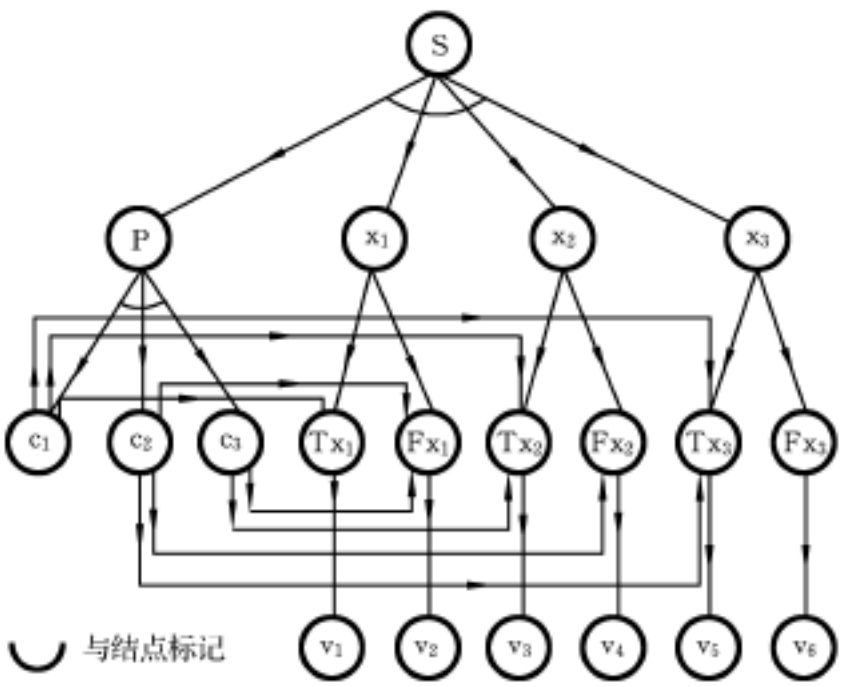


图 10.8　例 10.16 的与/或图

例 10.16　考察公式 $P = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2)$; $V(P) = x_1, x_2, x_3$; $n = 3$。图 10.8 表示运用定理 10.7 的构造法所得到的与/或图。

结点 Tx_1, Tx_2, Tx_3 可在总成本 3 的情况下求解。结点 P 不再需要额外的开销。因此结点 S 可通过求解它的所有子孙结点以及结点 Tx_1, Tx_2 和 Tx_3 来解出。这个解的总成本是 3(亦即是 n)。给 P 的变量赋真值"真",则 P 的结果也为"真"。

10.4 NP-难度的调度问题

这个问题是,对于给定的具有 n 个整数的多重集合 $A = \{a_1, a_2, \cdots, a_n\}$,判定其是否存在一个分划 P,使得 $\sum_{i \in P} a_i = \sum_{i \notin P} a_j$。这可以通过证明子集和数问题(第 8 章)是 NP-难度的问题来证明分划问题也是 NP-难度的问题。子集和数问题是确定 $A = \{a_1, a_2, \cdots, a_n\}$ 是否有一子集 S,使 S 的和数等于给定的整数 M。

定理 10.8 恰切覆盖∝子集和数。

证明 恰切覆盖问题是,对于给定的集合族 $F = \{S_1, S_2, \cdots, S_k\}$,确定是否存在由互不相交的集合所组成的子集合 $T \subseteq F$,使得下式成立:

$$\bigcup_{S_i \in T} S_i = \bigcup_{S_i \in F} S_i = \{u_1, u_2, \cdots, u_n\}$$

恰切覆盖问题是 NP-难度的,其证明留作习题。

从恰切覆盖问题的任一给定实例构造成这样的子集和数问题: $A = \{a_1, \cdots, a_k\}$,其中 $a_j = \sum_{1 \le i \le n} \varepsilon_{ji} (k+1)^{i-1}$,这里,若 $u_i \in S_j$。则 $\varepsilon_{ji} = 1$,否则 $\varepsilon_{ji} = 0$。而 $M = \sum_{0 \le i \le n} (k+1)^i = ((k+1)^n - 1)/k$。显然,当且仅当 $A = \{a_1, \cdots, a_k\}$ 有一个和数为 M 的子集合,则 F 有一恰切覆盖。因为从 F 可以在多项式时间内构造出 A 和 M,所以恰切覆盖∝子集和数。证毕。

定理 10.9 子集和数∝分划问题。

证明 令 $A = \{a_1, \cdots, a_n\}$和 M 定义子集和数问题的一个实例。构造集合 $B = \{b_1, b_2, \cdots, b_{n+2}\}$,具有 $b_i = a_i, 1 \le i \le n$, $b_{n+1} = M + 1$ 和 $b_{n+2} = (\sum_{1 \le i \le n} a_i) + 1 - M$。当且仅当 A 有一和数为 M 的子集时,B 有一分划。因为 B 可在多项式时间内由 A 和 M 获得,所以子集和数∝分划问题。证毕。

容易证明分划问题∝0/1 背包问题以及分划问题∝带限期的作业排序问题,因此这些问题也都是 NP-难度的问题。

10.4.1 相同处理器调度

令 $P_i, 1 \le i \le m$,是 m 个同样的处理器(或处理机)。例如, P_i 可以是计算机机房内的行式打印机。令 $J_i, 1 \le i \le n$,是 n 个作业,作业 J_i 要求的处理时间为 t_i。调度表 S 是将这 n 个作业分配给这些处理器的一种方案,对于每个作业 J_i, S 规定了处理它的时区和处理器。在任何时刻一个作业不能由数台处理器同时处理。令 f_i 是完成作业 J_i 处理的时间。调度表 S 的平均完成时间(MFT)是

$$MFT(S) = \frac{1}{n} \sum_{1 \le i \le n} f_i$$

令 w_i 是每个作业 J_i 的权,调度表 S 的加权平均完成时间(WMFT)是

$$WMFT(S) = \frac{1}{n} \sum_{1 \le i \le n} w_i f_i$$

令 T_1 是 P_i 完成分配给它的所有作业(或作业段)的时间, S 的完成时间是

$$FT(S) = \max_{1 \le i \le m} \{T_i\}$$

如果任一作业 J_i 从开始到结束都是在同一台处理器上连续处理的,则称 S 为不抢先调度。在抢先调度中,每个作业不一定要在一台处理器上连续处理到完成。

在这里指出 5.2 节磁带上的最优存储问题与不抢先调度的相似性是十分重要的。平均检索时间、加权平均检索时间和最大检索时间分别对应于平均完成时间、加权平均完成时间和完成时间。因此最小平均完成时间的调度可由 5.2 节开发的算法得到。要获得最小加权平均完成时间和最小完成时间的不抢先调度表则是 NP-难度的问题。

定理 10.10　分划问题$\propto$最小完成时间的不抢先调度。

证明　这里只对 $m = 2$ 的情况加以证明,将其扩展到 $m > 2$ 是很容易的。令 $a_i, 1 \leq i \leq n$ 是分划问题的一个实例。定义 n 个处理时间为 $t_i = a_i, 1 \leq i \leq n$ 的作业。当且仅当对这些 a_i 存在一个分划时,则对于这个作业集,在两台处理器上就有一个其完成时间至多为 $\Sigma t_i/2$ 的不抢先调度。证毕。

定理 10.11　分划问题$\propto$最小 WMFT 不抢先调度。

证明　这里仍只对 $m = 2$ 的情况加以证明,同样,将其扩展到 $m > 2$ 是很简单的事。令 $a_i, 1 \leq i \leq n$,定义分划问题的一个实例。对于 n 个作业且 $w_1 = t_i = a_i, 1 \leq i \leq n$,构造一个双处理器问题。对于这个作业集,当且仅当所有的 a_i 有一分划,那么就存在一种其加权平均完成时间至多为 $\frac{1}{2}\sum a_i^2 + \frac{1}{4}(\sum a_i)^2$ 的不抢先调度 S。为了证实这一点,令在处理器 P_1 上的作业的权和时间是 $(\overline{w}_1, \overline{t}_1), \cdots, (\overline{w}_k, \overline{t}_k)$,而在 P_2 上的作业则有 $(\overline{w}_1, \overline{\overline{t}}_1), \cdots, (\overline{w}_l, \overline{\overline{t}}_l)$。假定这就是作业分别在它们的处理器上处理时的次序。于是对这个调度 S 有:

$$\begin{aligned}\text{WMFT} &= \overline{w}_1\overline{t}_1 + \overline{w}_2(\overline{t}_1 + \overline{t}_2) + \cdots + \overline{w}_k(\overline{t}_1 + \cdots + \overline{t}_k) \\ &\quad + \overline{w}_1\overline{\overline{t}}_1 + \overline{w}_2(\overline{\overline{t}}_1 + \overline{\overline{t}}_2) + \cdots + \overline{w}_l(\overline{\overline{t}}_1 + \cdots + \overline{\overline{t}}_l) \\ &= \frac{1}{2}\sum w_i^2 + \frac{1}{2}(\sum \overline{w}_i)^2 + \frac{1}{2}(\sum w_i - \sum \overline{w}_i)^2\end{aligned}$$

由此可见,$\text{WMFT} \geq \frac{1}{2}\sum w_i^2 + \frac{1}{4}(\sum w_i)^2$。这个值当且仅当所有的 w_i(也就是所有的 a_i)有一分划时才能获得。证毕。

10.4.2　流水线调度

这里沿用 6.8 节所定义的术语。如果有 n 个要调度的作业,当 $m = 2$ 时其最小完成时间的调度可以在 O(nlogn)时间内做到;但当 $m = 3$ 时,无论是抢先调度或是不抢先调度,要得到其最小完成时间的调度都是具有 NP-难度的。对于不抢先调度,这一点是容易证明的。下面用抢先调度来证明这一结论。这证明对于不抢先情况也适用。

定理 10.12　分划问题$\propto$最小完成时间的抢先流水调度问题($m > 2$)。

证明　只对三处理器的情况作出证明。令 $A = \{a_1, a_2, \cdots, a_n\}$ 定义分划问题的一个实例。构造如下的抢先调度实例 FS,有 $n + 2$ 个作业和 $m = 3$ 台处理器,每个作业至多有 2 个非零的任务:

$$t_{1,i} = a_i, t_{2,i} = 0, t_{3,i} = a_i \qquad 1 \leq i \leq n$$

$$t_{1,n+1} = T/2, t_{2,n+1} = T, t_{3,n+1} = 0$$

$$t_{1,n+2} = 0, t_{2,n+2} = T, t_{3,n+2} = T/2$$

其中，$$T = \sum_{1}^{n} a_i$$

现在来证明，当且仅当 A 有一个分划，上述流水线调度实例有一个完成时间至多为 2T 的抢先调度。

(1) 如果 A 有一个分划 u，则存在一个完成时间为 2T 的不抢先调度。图 10.9 给出了一种这样的调度。

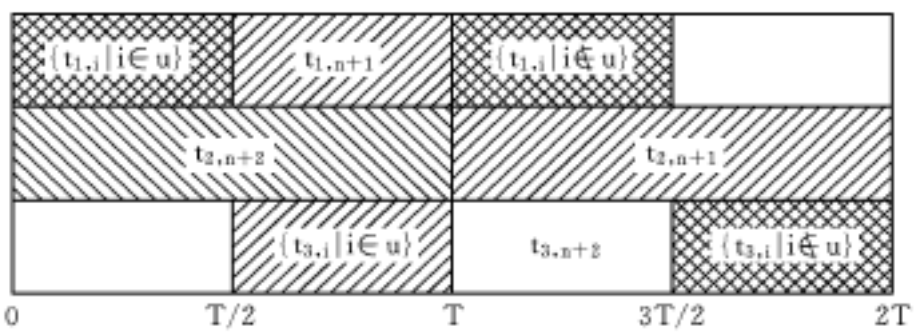


图 10.9 一种可能的调度

(2) 如果 A 不存在任何分划，则 FS 的所有抢先调度的完成时间必定大于 2T。这一点可用反证法加以证明。假设对于 FS 存在一种完成时间不大于 2T 的抢先调度，于是可得以下观察结论：

① 任务 $t_{1,n+1}$ 必须在时间 T 之前完成(因 $t_{2,n+1} = T$，而且在 $t_{1,n+1}$ 完成之前不能开始)。

② 任务 $t_{3,n+1}$ 不能在时间 T 之前开始，因为 $t_{2,n+2} = T$。

结论①意味着，在处理器 1 上最先开始的 T 个单位时间只有 T/2 是空的。令 V 是在时间 T 以前在处理器 1 上所完成任务的下标的集合(不包括任务 $t_{1,n+1}$)，由于 A 不存在任何分划，所以

$$\sum_{i \in V} t_{1,i} < T/2$$

于是 $$\sum_{\substack{i \notin V \\ 1 \le i \le n}} t_{3,i} > T/2$$

没包括在 V 中的作业在时间 T 之前不能在处理器 3 上着手处理，这是因为直到时间 T 为止在处理器 1 上对它们的处理还没完成。这一点与结论②结合起来则意味着，在时间 T，留待处理器 3 处理的工作的时间总量是：

$$t_{3,n+2} + \sum_{\substack{i \notin V \\ 1 \le i \le n}} t_{3,i} > T$$

所以，调度长度必定大于 2T。证毕。

10.4.3 作业加工调度

与流水线调度一样，作业加工调度问题中也有 m 台不同的处理器。被调度的 n 个作业都要求完成若干个任务。作业 J_i 的第 j 个任务要求 $t_{k,i,j}$ 的时间，它将在处理器 P_k 上执行。任一作业 J_i 的各任务将依 1,2,3,…的次序执行，在任务 j - 1(假若 j > 1)完成以前不能执行任务 j。值得注意的是，对于一个作业而言，它可能有许多任务在同一台处理器上被执行。在不抢先的调度中，任务一旦开始处理就一直进行到完成为止而不得被中断。FT(S) 和

MFT(S)的定义可以很自然地推广到这个问题。即使 m = 2 ,得到最小完成时间的抢先调度或最小完成时间的不抢先调度方案都是 NP-难度问题。对于不抢先的情况,利用分划来证明是很容易的。下面对抢先的情况给出证明。这个证明对不抢先情况也成立,不过对该情况不是一种最简单的证明而已。

定理 10 .13　分划问题∝最小完成时间的作业加工调度问题(m > 1)。

证明　只对使用两台处理器的情况进行证明。令 $A = \{a_1, a_2, \cdots, a_n\}$ 定义分划问题的一个实例。构造具有 n + 1 个作业和 m = 2 台处理器的作业加工问题的实例 JS。

作业 $1, \cdots, n$　$t_{1,i,1} = t_{2,i,2} = a_i$　　$1 \leq i \leq n$

作业 $n+1$　　$t_{2,n+1,1} = t_{1,n+1,2} = t_{2,n+1,3} = t_{1,n+1,4} = T/2$

其中,$T = \sum_{1}^{n} a_i$。

现证明当且仅当 A 有一分划时,上述作业加工问题有一个完成时间至多为 2T 的抢先调度。

(1) 如果 A 有一分划 u ,则存在一完成时间为 2T 的调度(见图 10 .10)。

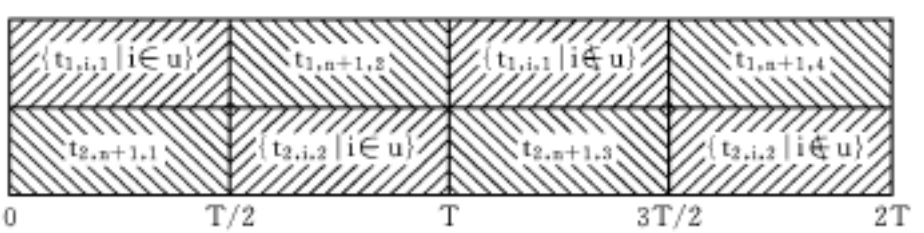


图 10 .10　另一种调度

(2) 如果 A 不存在任何分划,则对于 JS 的所有调度,其完成时间必定大于 2T 。为看出这一点,假设对于 JS 有一个完成时间至多为 2T 的调度。于是作业 n + 1 必须像图 10 .10 那样被调度。此外,在 P_1 和 P_2 上还不可能有空闲时间。令 R 是在[0, T/ 2]时间内调度到 P_1 上的作业集合。令 R′是 R 的一个子集,它包含了在这段时间内已在 P_1 上完成了其第一个任务的所有作业。由于这些 a_i 不存在分划,故 $\sum_{j \in R'} t_{i,j,1} < T/2$。因此,$\sum_{j \in R'} t_{2,,j,2} < T/2$。因为只有 R′中作业的第二个任务可以在时间[T/ 2, T]内调度到 P_2 上,由此可见在这段时间内 P_2 有某些空闲时间,所以 JS 的完成时间必定大于 2T。证毕。

10 .5　NP-难度的代码生成问题

10 .5 .1　有公共子表达式的代码生成

当算术表达式有公共子表达式时,可用一个有向无环图 dag(directed acyclic graph) 来表示。dag 中每一个内结点(即出度不为零的结点)代表一个运算符。假定表达式只含有双目运算符,则每个内结点 P 的出度为 2 。由 P 所邻接的两个内结点分别称作 P 的左和右儿子。P 的左、右儿子是 P 的左、右运算量的 dag 之根。在表达式不含公共子表达式的情况下,它的 dag 表示与 7 .2 节的树表示相同。图 10 .11 示出了一些表达式和它们的 dag 表示。

定义 10 .6　叶子是出度为 0 的结点。第一层结点是其两个儿子均为叶子的结点。共享

结点是有多于一个父亲的结点。叶子 dag 是一个 dag,它的所有共享结点都是叶子。第一层 dag 是一个 dag,它的所有共享结点都是第一层结点。

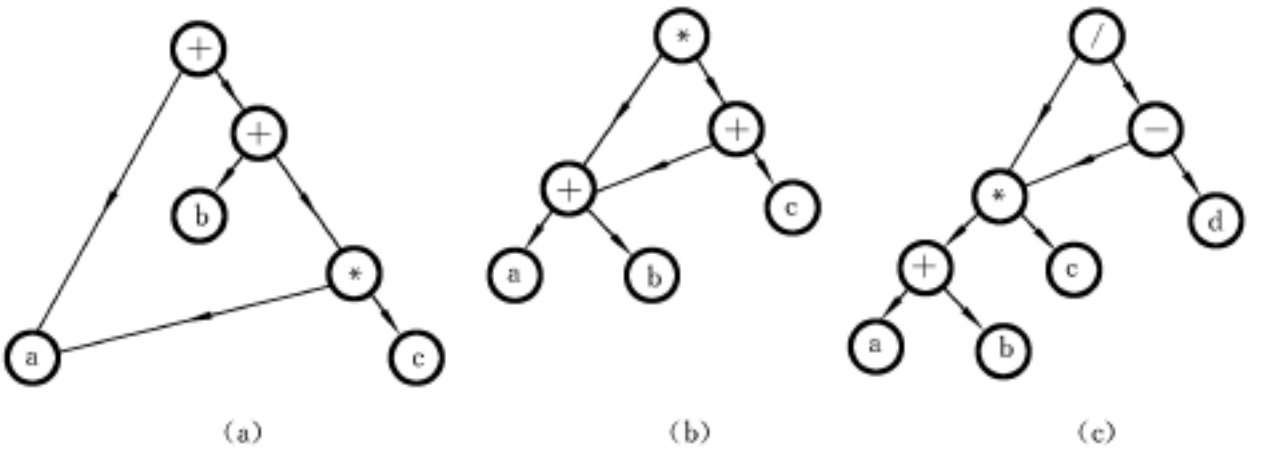


图 10 .11　表达式及其 dag

(a) a + (b + a * c); (b) (a + b) * (a + b + c); (c) (a + b) * ¢ ((a + b) * c - d)

例 10 .17　图 10 .11(a)的 dag 是一个叶子 dag。图 10 .11(b)的 dag 是一个第一层 dag。图 10 .11(c)既不是叶子 dag 也不是第一层 dag。

叶子 dag 由公共子表达式只是一些简单变量或常数的表达式所产生。第一层 dag 由只具有以下形式的公共子表达式的表达式所产生,这些公共子表达式具有 a⊕b 的形式,其中 a 和 b 是简单变量或常数,⊕是一个运算符。

对第一层 dag 生成最优代码的问题是一个具有 NP-难度的问题,即使用来生成代码的机器只有一个寄存器也是如此。在不用 STORE 指令计算 dag 的值的情况下,确定所需最小寄存器数也是 NP-难度的问题。要指出的是,在没有公共子表达式的情况下,这两个问题都可在线性时间内求解(参见 7 .2 节)。

例 10 .18　对于具有一个或两个寄存器的机器而言,图 10 .11(b)所示 dag 的最优代码给出如下:

一寄存器机器的最优代码

```
LOAD    a, R1
ADD     R1, b, R1
STORE   R1, T1
ADD     R1, c, R1
STORE   R1,T2
LOAD    T1, R1
MUL     R1, T2, R1
```

二寄存器机器的最优代码

```
LOAD    a, R1
ADD     R1, b, R1
ADD     R1, c, R2
MUL     R1, R2, R1
```

为了证明上面所述的结论,需要使用在习题中证明其为 NP-难度的反馈结点集(FNS)问题。所谓 FNS 问题是:给定一有向图 G = (V,E)和整数 k,确定是否存在结点的子集 V′,V′ ⊆ V 且|V′|≤k,使得图 H = (V - V′,E - {〈u,v〉|u∈V′或者 v∈V′})不包含有向环,这里,图 H 是从 G 中删去 V′中所有结点以及与其相关联的边而得。

下面只给出生成最优代码是具有 NP-难度的证明。使用这一证明的构造法也可证明在不用 STORE 指令计算 dag 的值的情况下确定所需最小寄存器数也是具有 NP-难度的。证明中假定表达式可含有可交换的运算符而且共享结点可以只计算一次。该证明易于推广到

允许重新计算共享结点。

定理 10.14　FNS∝在一寄存器机器上第一层 dag 的最优代码生成。

证明　令 G, k 是 FNS 的一个实例。令 n 是 G 的结点数。构造具有以下性质的 dagA: 当且仅当 G 有大小至多为 k 的反馈结点集时,对应于 A 的表达式的最优代码至多有 n + k 条 LOAD 指令。

这一 dag A 由三类结点组成: 叶子结点、链结点和树结点。所有的链结点和树结点都是对应于可交换运算符(例如,“ + ”)的内结点。叶子结点表示不同的变量。这里用 d_v 代表 G 中结点 v 的出度。对于 G 中每个结点 v, A 中就有一条链结点 $v_1, v_2, \cdots, v_{d_{v+1}}$ 的有向链与之相对应。结点 $v_{d_{v+1}}$ 是对应于 v 的那条链的头结点并且是叶结点 v_L 和 v_R 的父亲 (参见例 10.19 和图 10.12) 。v_1 是此链的链尾。对于 G 中的任一条边 〈v, w〉 ,在 A 中,对应于 v 的链就有一个除了头结点外的链结点用一条有向边连接到对应于 w 的链的头结点。在 v 对应的链中,除了头结点外不同的链结点要用有向边连接到不同的链的头结点。由于增加了这些有向边,因此每个链结点的出度为 2。又由于每个链结点代表一个可交换运算符, 因此它的两个儿子中哪个是左儿子都无关紧要。

至此,所得 dag 的每条链的链尾入度为 0。现在引进一些树结点将所有的链尾结合起来,从而得到只有一个结点(根结点)入度为 0 的 dag。由于 G 有 n 个结点,因此需要 n - 1 个树结点(每棵有 n - 1 个内部结点的二元树有 n 个外部结点)。将这 n - 1 个结点连接在一起构成一棵二元树 (任何有 n - 1 个结点的二元树皆行)。在外部结点的位置上连上这 n 条链的链尾,就得到一个对应于一算术表达式的 dagA(参见图 10.12(b))。

易于看出, A 的每个最优代码段恰好有叶结点的 n 条 LOAD 指令。每个链结点或树结点(假定一个共享结点只计算一次)则恰好有一条ⓞp 型指令。因此,唯一可变的是链和树结点的 LOAD 和 STORE 的指令条数。如果 G 中没有有向环,那么它的结点可按拓扑次序排列(G 中,只有当 u 到 v 无有向路径时,按拓扑排序结点 u 就在 v 之前)。令 $v_1, v_2, \cdots, v_n$ 是 G 中结点的拓扑次序。表达式 A 可以在不使用链和树结点的 LOAD 指令情况下进行计算。其计算步骤如下:首先计算对应于 v_n 的那条链上所有的结点并存放尾结点的结果;接着计算对应于 v_{n-1} 的链上所有的结点并计算对应于 v_{n-1} 的链的尾结点到根的那条路径上两个运算量都可得到的任何结点;最后需要存放一个结果。类似地,可计算对应于 v_{n-2} 的那条链;还可以计算从这条链尾到根的路径上两个运算量都可得到的所有结点。按这种方式继续下去就可计算整个表达式。

如果 G 至少包含一个环: $v_1, v_2, \cdots, v_i, v_1$,那么对于 A 的每个代码段必须至少含有一条对应于 $v_1, v_2, \cdots, v_i$ 之一的链上一个链结点的 LOAD 指令。进而若这些结点中没有结点在其它环上时,则对应于它们的所有链的链结点可以只用一个链结点的一条 LOAD 指令来计算。这一论证易于推广到:如果最小反馈结点集的大小为 p,则对于 A 的每个最优代码段恰好有 n + p 条 LOAD 指令。这 p 条 LOAD 指令对应于与最小反馈结点集相应的那些尾结点和那些尾结点的兄弟的一个组合。在对链结点使用的是不可交换运算符且使得链上的每个后继是它父亲的左儿子的情况下,这 p 条 LOAD 指令将对应于任一最小反馈集的那些链的链尾。此外,若最优代码段含有链结点的 p 条 LOAD 指令,则 G 有一个大小为 p 的反馈结点集。证毕。

例 10.19　图 10.12(b)表示对应于图 10.12(a)的一个 dagA。{r,s}是 G 的一个最小反馈结点集。每个链结点和树结点上的运算符可以假设为“+”。A 的每个代码段有一条对应于$(p_L,p_R),(q_L,q_R),\cdots,(u_L,u_R)$中之一的 LOAD 指令。按照 r_4，s_2，q_2，q_1，p_2，p_1，c，u_3，u_2，u_1，t_2，t_1，e，s_1，r_3，r_2，r_1，d，b，a 这样的次序来计算这些结点，就可在只使用两条附加的 LOAD 指令的情况下计算出表达式 A。为了计算 s_1 和 r_3 各需一条 LOAD 指令。

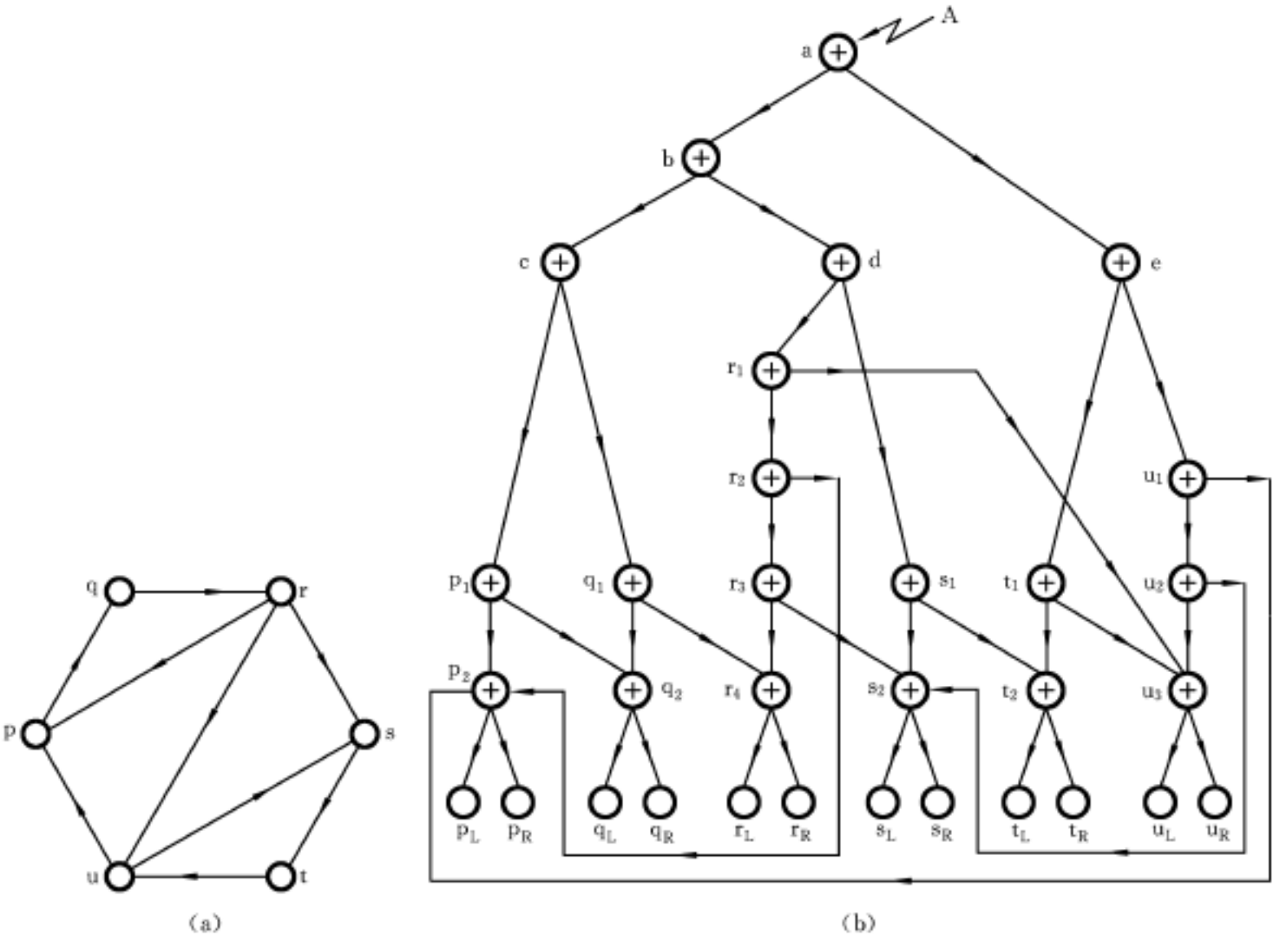


图 10.12　一个图及其对应的 dag

(a) 图 G;(b) 对应的 dag A

10.5.2　并行赋值指令的实现

SPARKS 的并行指令的格式为:$(v_1,v_2,\cdots,v_n)\leftarrow(e_1,e_2,\cdots,e_n)$,其中,这些 v_i 是不同的变量名,e_i 是表达式。这个语句的语义是,v_i 的值由表达式 e_i 的值所更新,$1\le i\le n$。表达式 e_i 的值是用 e_i 中那些变量在执行本指令之前所具有的值计算出来的。

例 10.20　(1) (A,B)←(B,C)等价于 A←B;B←C。

(2) (A,B)←(B,A)等价于 T←A;A←B;B←T。

(3) (A,B)←(A+B,A-B)等价于 $T_1\leftarrow A;T_2\leftarrow B;A\leftarrow T_1+T_2;B\leftarrow T_1-T_2$ 也等价于 $T_1\leftarrow A;A\leftarrow A+B;B\leftarrow T_1-B$。

正如例 10.20 所指出的那样,在执行一条并行赋值语句时可能需要把某些 v_i 存放到临时工作单元中去。需要存储的是出现在表达式 e_j 中的那些 v_i,这里 $1\le j\le n$。只有在变量 v_i

出现于 e_i 之中时,则称 v_i 由表达式 e_j 所引用。显然,只有那些被引用的变量才需要复制到临时工作单元。然而,例 10.20 的(2)和(3)则表明并不是所有被引用的变量都需要复制。

实现一条并行赋值语句是执行一系列以下类型的指令:$T_j \leftarrow v_i$ 和 $v_i \leftarrow e'_i$。其中,e'_i 通过用临时工作单元 T_j 取代 e_i 中出现的所有 v_i 而得;T_j 中保留着 v_i 的旧值而 v_i 则由 e'_i 所修改。令 $R = (r(1), \cdots, r(n))$ 是 $(1, 2, \cdots, n)$ 的一种排列。R 是赋值语句的一种实行(realization),它规定了并行赋值语句在实现中 $v_i \leftarrow e'_i$ 型语句出现的次序。这个次序是 $v_{r(1)} \leftarrow e'_{r(1)}$;$v_{r(2)} \leftarrow e'_{r(2)}$;等。在这一实现中还夹杂着一些 $T_j \leftarrow v_i$ 型的语句。不失一般性,可以假设语句 $T_j \leftarrow u_i$(如果它在实现中出现)总是紧接在语句 $v_i \leftarrow e'_i$ 的前面。因此一种实行就完全刻画了一种实现的特性。对于任一给定的实行,确定 $T_j \leftarrow v_i$ 型指令的最小数目是很容易的。这个数目就是此种实行的成本。一种实行 R 的成本 C(R) 是 e_j 引用 v_i 的数目,而 e_j 则与指令 $v_j \leftarrow e'_i$ 之后所出现的指令 $v_i \leftarrow e'_j$ 相对应。

例 10.21　考虑语句:$(A, B, C) \leftarrow (D, A + B, A - B)$。它有 3! = 6 种不同的实行,这些实行的成本是

R	C(R)
1,2,3	2
1,3,2	2
2,1,3	2
2,3,1	1
3,1,2	1
3,2,1	0

实行 3,2,1 对应于实现方案:$C \leftarrow A - B; B \leftarrow A + B; A \leftarrow D$。根本不需要临时存储($C(R) = 0$)。

并行赋值语句的最优实行是具有最小成本的一种实行。当这些表达式 e_i 都是些变量名或常数时,可以在线性时间 O(n) 内找到最优实行。当这些 e_i 是允许带有运算符的表达式时,则寻找最优实行是 NP-难度的问题,这可利用反馈结点集来证明。

定理 10.15　FNS$\propto$最小成本的实行。

证明　令 $G = (V, E)$ 是任一 n 个结点的有向图。构造并行赋值语句 $P: (v_1, v_2, \cdots, v_n) \leftarrow (e_1, e_2, \cdots, e_n)$,其中,这些 v_i 对应于 V 中 n 个结点,e_i 是表达式 $v_{i1} + v_{i2} + \cdots + v_{ij}$。$\{v_{i1}, v_{i2}, \cdots, v_{ij}\}$ 是由 v_i 所邻接的结点集(即,$\langle v_i, v_{il}\rangle \in E(G), 1 \leq l \leq j$)。这一构造至多需要 $O(n^2)$ 的时间。

令 U 是 G 的任一反馈结点集。又令 $G' = (V', E') = (V - U, E - \{\langle x, y\rangle \mid x \in U$ 或 $y \in U\})$ 是删除了结点集 U 和与 U 中结点相关联的所有边之后的一个图。从反馈结点集的定义可知 G′ 是一个无环图。所以 V - U 中的结点可排列成序列 $s_1, s_2, \cdots, s_m$,其中 $m = |V - U|$ 且 E′ 不含有这样的边 $\langle s_i, s_i\rangle$,对于任意的 i 和 j,$1 \leq i < j \leq m$。因此,P 的一种实现是,首先将对应于 U 中的结点的变量存入临时工作单元,然后将对应于 $v_i \in U$ 的指令 $v_i \leftarrow e'_i$ 随于其后,再继之以与 $s_1, s_2, \cdots, s_m$(并按这样的次序)相对应的指令。这将是 P 的一种正确实现。(e'_i 是用相应的临时工作单元取代了 e_i 中出现的所有 v_i 后得到的,而这些 $v_i \in U$)。与这一实现相对应的实行 R 有 $C(R) = |U|$。于是,如果 G 有一个大小至多为 k 的反馈结点集,那么 P 就有一个成本至多为 k 的最优实行。

假定 P 有一个成本为 k 的实行 R。令 U 是必须存入临时工作单元的 k 个变量的集合，又令 $R=(q_1,q_2,\cdots,q_n)$。从 C(R)的定义可知，对于 $j<i$，不会有 eq_i 引用 vq_j，除非 $vq_j\in U$。因此从 G 中删去 U 内的结点之后剩下的 G′是无环图。从而，U 定义了 G 的一个大小为 k 的反馈结点集。

总起来说，当且仅当 P 有一成本至多为 k 的实行，G 才有一大小至多为 k 的反馈结点集。因此，如果有一个在多项式时间内确定最小成本的算法，那么就能在多项式时间内求解反馈结点集问题。证毕。

10.6 若干简化了的 NP-难度问题

一旦证得问题 L 是 NP-难度的，则往往倾向于不再考虑 L 可以在确定的多项式时间内求解的可能性。但是，此时人们会自然提出一个问题：能否对一个 NP-难度的问题加以一些适当的限制，由此产生该问题的某些子类问题，那么这些子问题是否能在确定的多项式时间内求解呢？十分明显，对于任何一个 NP-难度问题，只要加上足够的限制(或者定义一个足够小的子类问题)，就可使之成为一个多项式可解问题。作为例子，考虑如下问题：

(1) 每个子句至多有 3 个文字的 CNF-可满足性问题是具有 NP-难度的。如果限制每个子句至多只有两个文字，则 CNF-可满足性是多项式可解的。

(2) 将并行赋值语句生成最优代码是 NP-难度问题。然而，如果将表达式 e_j 限制为简单变量，则最优代码可在多项式时间内生成。

(3) 在第一层 dag 生成最优代码是具有 NP-难度的，但树的最优代码可在多项式时间内生成。

(4) 确定一个平面型图是否可三着色为具有 NP-难度的。要确定该图是否可二着色，只需考察它是否为二部图即可。

因为 NP-难度问题是多项式可解的可能性极小，所以确定一个问题可在多项式时间内求解的最弱限制条件就成为一件十分重要的工作。

为了使已知有多项式时间算法的子类问题和尚不知其有这样算法的子类问题间的差距缩小，因此希望得到一组强限制条件，使得在这组限制下的问题仍然是 NP-难度的或者是 NP-完全的。

在不经证明情况下给出了最严格的限制，在这种限制下，这些问题都已知是 NP-难度的或者是 NP-完全的。这些被简化或加了限制的问题都是判定问题，对于每个问题，下面都只说明其输入和要做的判定。

定理 10.16 下列判定问题都是 NP-完全的：

1. 结点覆盖

输入：结点度至多为 3 的无向图 G 和整数 k。

判定：G 有一个大小至多为 k 的结点覆盖吗？

2. 平面结点覆盖

输入：结点度至多为 6 的平面无向图 G 和整数 k。

判定：G 有一个大小至多为 k 的结点覆盖吗？

3. 可着色性

输入:结点度至多为 4 的平面无向图 G。

判定:G 可三着色吗?

4. 无向图的哈密顿环

输入:结点度至多为 3 的无向图 G。

判定:G 是否有一哈密顿环?

5. 平面无向图的哈密顿环

输入:平面无向图 G。

判定:G 是否有一哈密顿环?

6. 平面有向图的哈密顿路

输入:一个入度至多为 3 和出度至多为 4 的平面有向图 G。

判定:G 有一条有向哈密顿路吗?

7. 一进制输入分划

输入:正整数 $a_i, 1 \leq i \leq m$, n 和 B,并有

$$\sum_{1 \leq i \leq m} a_j = nB, \frac{B}{4} < a < \frac{B}{2}, 1 \leq i \leq m \text{ 且 } m = 3n \text{ 。以一进制形式输入。}$$

判定:是否存在 a_i 的一个分划$\{A_1, \cdots, A_n\}$,使得每个 A_i 包含三个元素且

$$\sum_{a \in A_i} a = B, 1 \leq i \leq n \text{ ?}$$

8 .一进制流水线调度

输入: 用一进制表示的任务时间和整数 T。

判定: 是否存在一个双处理器的不抢先调度,使其平均完成时间至多为 T?

9. 简单最大割

输入: 图 G = (V,E)和整数 k。

判定: V 是否有一子集 V_1,使得至少有 k 条边$(u,v) \in E$,而 $u \in V_1$,且 $v \mid V_1$?

10. 可满足性 2

输入: CNF 中一命题公式 F,它的每个子句至多有两个文字;整数 k。

判定: F 是否至少有 k 个子句可满足?

11. 删除最少的边求二部子图

输入: 无向图 G 和整数 k。

判定: 能否至多删除 G 的 k 条边而得到一个二部子图?

12. 删除最少的结点求二部子图

输入: 无向图 G 和整数 k。

判定: 能否至多删除 G 的 k 个结点而得到一个二部子图?

13. 使之分成两个大小相等子集的最小割

输入: 无向图 G = (V,E),两个不同的结点 s 和 t,正整数 W。

判定: 是否有分划 $V = V_1 \cup V_2, V_1 \cap V_2 = \quad, |V_1| = |V_2|, s \in V_1, t \in V_2$ 且$|\{(u,v) \mid u \in V_1, v \in V_2$ 和$(u,v) \in E\}| \leq W$?

14. 简单的最优线性排列

输入: 无向图 $G=(V,E)$, $|V|=n$, 整数 k。

判定: 是否有一一对应的函数 f, 其定义为 $V\rightarrow\{1,2,\cdots,n\}$, 使得 $\sum_{(u,v)\in E}|f(u)-f(v)|\leq k$?

习 题 十

10.1 求一个复杂度为 $O(n)$的不确定算法,决定对于 n 个数 a_i, $1\leq i\leq n$ 是否存在一个其和数为 M 的子集。

10.2 (1)当所有的 p、w 和 M 为正整数,并且以输入长度作为衡量复杂度的函数。证明背包的最优化问题可约化为背包的判定问题。

提示:设输入长度为 m, 于是有 $\sum p_i\leq n2^m$, 其中 n 是物品数。用二分检索法确定最优解的值。

(2) 设 DK 是背包判定问题的一个算法, R 是背包最优化问题中一个最优解的值。求应如何获得 0/1 赋值,对于 $1\leq i\leq n$ 的 x_i, 方能借助于使用 DK 的 n, 使得 $\sum p_i x_i=R$ 且 $\sum w_i x_i\leq M$。

10.3 联系 COOK 定理证明中的公式 G(见 10.2 节), 对于 i 求下述各种情况下的 M。注意 M 可以含有至多 $O(P(n))$ 个文字(作为 n 的函数)。在负数时,则假设用其补码来求出 M。找出相应 $G_{i,t}$ 是如何变换为 CNF 的。在这变换过程中, $G_{i,t}$ 的长度增加必须不大于一个常数因子(例如 W^2)。

(1) $Y\leftarrow Z$。

(2) $Y\leftarrow V-Z$。

(3) $Y\leftarrow V+Z$。

(4) $Y\leftarrow V*Z$。

(5) $Y\leftarrow$choise(0,1)。

(6) $Y\leftarrow$choise(r:u), 其中 r 及 u 是变量。

10.4 证明集团的最优化问题可以约化为集团的判定问题。

10.5 设 SAT(E) 是确定命题公式 E 是否在 CNF 中的一个算法,它是可满足的。证明当 E 有 n 个变量 $x_1, x_2, \cdots, x_n$ 且可满足时,使用算法 SAT(E) n 次就可以确定对于 x_i 的一个真值指派,使得 E 为真。

10.6 [反馈结点集](1) 设 $G=(V,E)$ 为一有向图, $S\subseteq V$ 为结点的子集,使得删除 S 和涉及 S 的结点的边之后,得一没有有向环的图 G'。这样的一个 S 即为反馈结点集。S 的大小为 S 中的结点数。所谓反馈结点集的判定问题(FNS),就是对于给定的输入 k,确定 G 是否有一大小至多为 k 的反馈结点集。证明结点覆盖的判定问题$\propto$FNS。

(2) 为 FNS 写一多项式时间的不确定算法。

10.7 反馈结点集的最优化问题即为求出它的最小反馈结点集(参阅题 10.6)。证明这一问题可约化为 FNS。

10.8 [哈密顿环]在一个给定的无向图 G 内,如果存在一无向环,经过环中每一结点正好一次后回到起始点,则此无向环是 G 的哈密顿环。设 UHC 是确定这样的环是否存在的问题。证明 DHC$\propto$UHC (DHC 的定义可在 10.3 节中找到)。

10.9 证明 UHC$\propto$CNF 可满足性。

10.10 证明 DHC$\propto$CNF 可满足性。

10.11 设 SATY 是确定每个子句至多有三个文字的 CNF 中命题公式是否可满足问题。证明 CNF 可满足性$\propto$SATY。

提示:证明如何写具有多于 3 个文字的子句作为若干子句的 and,其中每个含有至多 3 个文字。为此必

须引入若干新变量。满足原始子句的任何指派必须同时满足所建的全部新子句。

10.12　[最小等价图]一有向图 $G=(V,E)$ 是有向图 $G'=(V,E')$ 的等价图,当且仅当 $E^* = E'^*$ 且 G 和 G′ 的传递闭包相同。G 是最小等价图,当且仅当在 G 的所有等价图中 $|E|$ 为最小。最小等价图判定问题(MEG)是确定 G 是否有一个最小等价图 $|E| \leq k$,其中 k 是给定的某个输入。

(1) 证明 DHC∝MFG。

(2) 为 MFG 写一个具有多项式时间的不确定算法。

10.13　[集合覆盖]设 $F=\{S_j\}$ 是一个有穷的集合族,$T \subseteq F$ 是 F 的一个子集。T 是 F 的一个覆盖当且仅当

$$\bigcup_{S_j \in T} S_j = \bigcup_{S_j \in F} S_j$$

集合覆盖的判定问题是确定 F 是否有一个不多于 k 个集合的覆盖 T。证明结点覆盖的判定问题可以约化本问题。

10.14　[恰切覆盖]设 $F=\{S_j\}$ 和习题 10.13 意义相同。$T \subseteq F$ 是 F 的恰切覆盖,当且仅当 T 是 F 的一个覆盖且 F 中的所有集合是两两不相交的。证明着色数判定问题可约化为确定 F 是否有一恰切覆盖问题。

10.15　证明 SAT(3)∝恰切覆盖。

10.16　[电路构造]设 C 是由与、或、非门组成的电路,$x_1,\cdots,x_n$ 是输入,f 是输出。证明判定 $f(x_1,x_2,\cdots,x_n)=F(x_1,x_2,\cdots,x_n)$ 是否成立是 NP-难度的,式中 F 是一命题公式。

10.17　设 $J_1,\cdots,J_n$ 是 n 个作业,作业 i 的处理时间为 t_i,截止期为 d_i。若直到时间 r_i 作业 i 还不能进行处理,证明判定 n 个作业在不破坏截止期情况下用一台机器处理是 NP-难度的。(提示:使用分划。)

10.18　设:J_i $(1\leq i\leq n)$ 是 n 个作业,$r_i=0$,$1\leq i\leq n$(r_i 的意义见题 10.17);f_i 是 J_i 在单处理器调度表 S 上的完成时间;J_i 的延迟度(tardiness) T_i 是 $\max\{0,f_i-d_i\}$;w_i $(1\leq i\leq n)$ 是非负的 J_i 权,加权延迟度的总和是 $\sum w_i T_i$。证明求 $\sum w_i T_i$ 最小化的调度是 NP-难度的。

10.19　假定 P 是一并行赋值语句 $(v_1,\cdots,v_n)<(e_1,\cdots,e_n)$,其中每个 e_i 是一简单变量,且 v_i 是各不相同的。为了简便起见,假定 P 中不同的变量为 $v_1,\cdots,v_m$,其中 $m\geq n$,且 $E=(i_1,\cdots,i_n)$ 是使得 $e_{ij}=v_{ij}$ 的一组下标。写一 O(n) 算法对 P 求最优实行。

第 11 章

并 行 算 法

11.1 并行计算机与并行计算模型

传统的计算机是串行结构的,每一时刻只能按一条指令对一个数据进行操作。为了克服这种传统结构对提高运算速度的限制,从 20 世纪 60 年代起开始将并行处理技术引入计算机的结构设计中,利用其对计算机系统结构进行改进。所谓并行处理就是把一个传统串行处理的任务分解开来,并将其分配给多个处理器同时处理,即在同一时间间隔内增加计算机的操作数量。为并行处理所设计的计算机称为并行计算机。

对于计算机,存在着几种不同的分类方法,目前大家普遍遵循的是 Flynn 分类法,它首先按指令流的重数将机器分为二类:SI (Single Instruction Stream)单指令流;MI (Multiple Instruction Stream)多指令流。其次,按操作数流的重数加以区分:SD (Single Data Stream)单数据流;MD (Multiple Data Stream)多数据流。这样,就有 4 种可能的组合,即 SISD, SIMD, MISD, MIMD。

SISD 计算机代表如今使用的大多数串行计算机,是单指令流对单数据流进行操作。

SIMD 计算机是所谓的阵列机,它有许多个处理单元 (PE),由同一个控制部件管理,所有 PE 都接收控制部件发送的相同指令,对来自不同数据流的数据集合序列进行操作。

MISD 计算机从概念上讲,则有多个 PE,接收不同的指令,对相同数据进行操作,一般认为 MISD 机目前尚无实际代表,此类结构很少受到人们的注意。

MIMD 计算机包括多处理机和多计算机两类,它们都由可各自执行自己程序的多处理器组成。其中,多处理机以各处理器共享公共存储器为特征,而多计算机以各处理器经通信链路传递信息为特征。它们与 SIMD 计算机的根本区别在于 :SIMD 机中每台处理器只能执行中央处理器的指令,而 MIMD 机中每台处理器仅接受中央处理器分给它的任务,它执行自己的指令,所以可达到指令、任务并行。

根据 Flynn 分类法,通用的并行计算机分为 SIMD 机和 MIMD 机两大类,它们是并行算法的物质基础。对于并行算法的设计者而言,不能仅局限于某种具体的并行机而设计并行算法,而必须从算法的角度,将各种并行机的基本特征加以理想化,抽象出所谓的并行计算机模型,然后在此基础上研究和设计各种有效的并行算法。由 Flynn 分类法,可将并行计算机分为两大类,这两大类也确定了两大类并行计算模型,即 SIMD 和 MIMD 两类并行计算模型。这两大类还可进一步细分,SIMD 模型可细分为基于共享存储的 SIMD 模型和基于互连网络的 SIMD 模型;MIMD 模型主要可细分为基于共享存储的 MIMD 模型和基于异步通信的互连网络模型。

SIMD 共享存储模型是假定有有限或无限个功能相同的处理器,每个处理器拥有简单

的算术运算和逻辑判断能力,在理想的情况下假定存在一个容量无限大的共享存储器,在任何时刻,任意一个处理器均可通过共享存储器的共享单元同其它任何处理器互相交换数据。由于实际情况是共享存储器的容量是有限的,因此在同一时刻,当多个处理器访问同一单元时就会发生冲突。根据模型解决冲突的能力,SIMD 共享存储模型又可进一步分为

(1) 不容许同时读和同时写。即每次只允许一个处理器读和写一个共享单元,这种模型简记为 SIMD-EREW PRAM。

(2) 容许同时读,但不容许同时写。即每次允许多个处理器同时读一个共享单元的内容,但每次只允许一个处理器向某个共享单元写内容,这种计算模型简记为 SIMD-CREW PRAM。

(3) 允许同时读和写。即每次容许任意多个处理器同时读和同时写同一个共享存储单元,这种计算模型简记为 SIMD-CRCW PRAM。

对于同一求解问题,在以上 3 种计算模型上设计的并行算法通常是不同的。算法的运算时间也不相同,记 3 种算法的计算时间为 T_1,T_2,T_3,它们满足下面关系:

$$T_1 \geq T_2 \geq T_3$$

在 SIMD 互连网络模型中,每个处理器在控制器控制下或处于活动状态,或处于不活动状态。活动状态的处理器都执行相同的指令,处理器之间的数据交换是通过互联网络进行的。根据互连网络的连接方式,SIMD 互连网络模型又分为两类。一类是处理器-处理器之间直接互连 (称为闺房式),如图 11.1 所示;另一类是处理器-存储器之间直接互连 (称为舞厅式),如图 11.2 所示。从算法设计者来讲,这二类无本质差别。因此,以后仅讨论闺房式这一类。

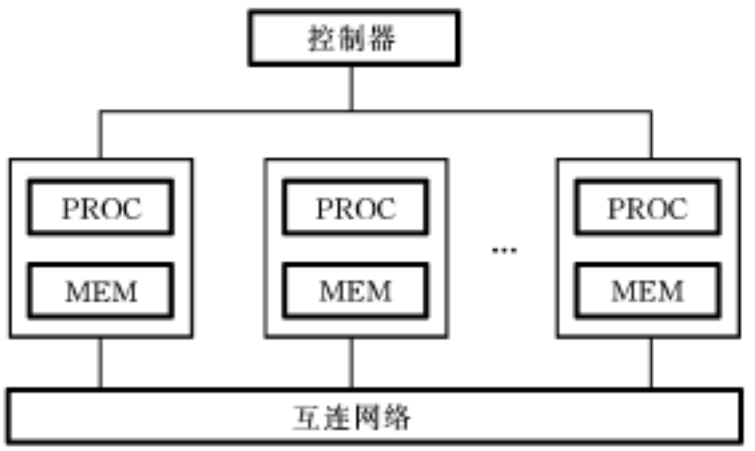


图 11.1　闺房式 SIMD 模型

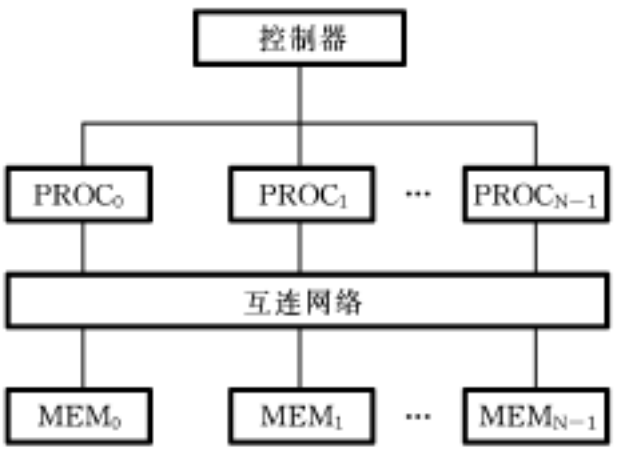


图 11.2　舞厅式 SIMD 模型

在共享存储的 MIMD 计算模型中,所有的处理器共享一个公共的存储器,每个处理器各自完成自己的任务,各处理器之间的通信是通过共享存储器中的全局变量来实现的。在这种模型上开发的算法称之为异步并行算法。

在基于互连网络的 MIMD 计算模型中,处理器之间不存在共享存储器,每个处理器从各自存储器中存取指令和数据。各处理器之间用通信网络以信息方式交换数据。在此模型上设计的算法称为分布式算法。

在基于互连网络的模型中,由于数据分布存储,信息通过互连网络进行传递,因此算法与处理器互联的拓扑结构紧密相关,下面列举一些常用的互连网络的拓扑结构。

1. 一维线性连接

此连接方式是所有并行机中处理器之间一种最简单的互连方式。其中每个处理器只与其左右近邻相连 (头尾处理器除外),如图 11 .3 所示。

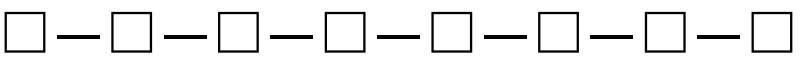


图 11 .3　一维线性连接

2. 二维网孔连接

在此连接方式中处理器之间按二维阵列形式排列,每个处理器仅与 4 个相邻处理器(若有的话)互连,如图 11 .4 所示。二维网孔连接有两个重要的变种,在这两个变种中,边界处理器也有线相连接。

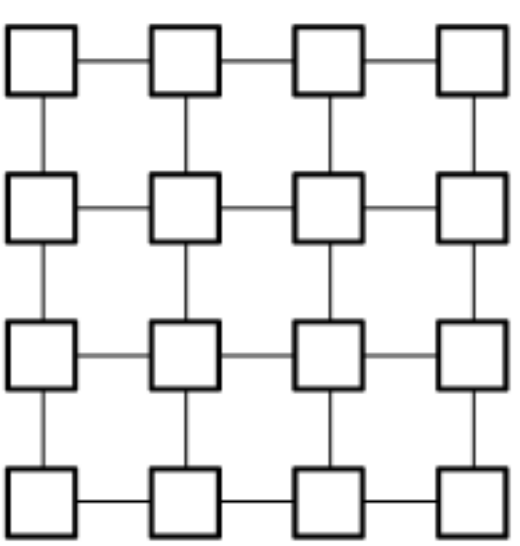


图 11 .4　二维网孔连接

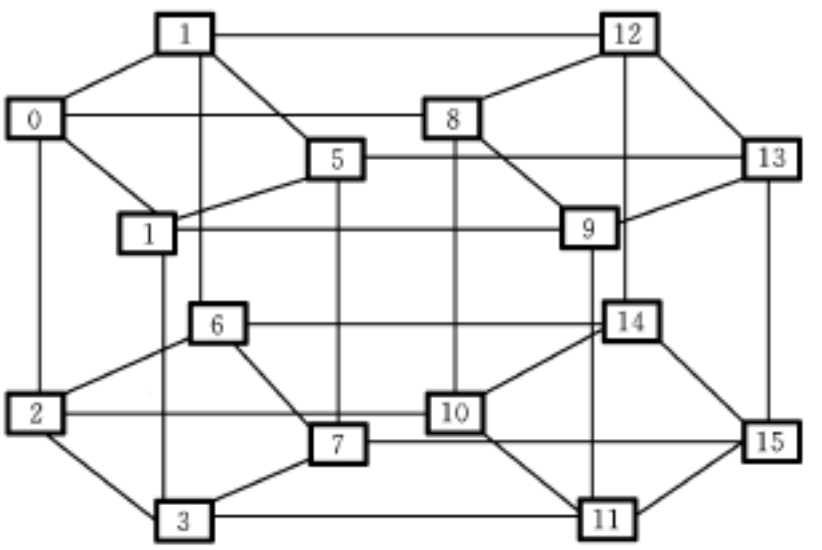


图 11 .5　四维超立方网络

3. 超立方连接

对于 $N = 2^k$ 个处理器,可将其组织成一个 k-维超立方连接。首先,处理器按 0,1,2,…,$2^k - 1$ 依次编号,然后,处理器之间按下述方式连接:处理器 i 与处理器 j 有线连接当且仅当 i 与 j 的二进制表示中仅一位不同。图 11 .5 给出了 k = 4 的四维超立方连接方式。

4. 树形连接方式

树形连接方式是利用二叉树这种常用的数据结构组织而成的。对于一棵有 d 级 (编号由根至叶为 0 到 d - 1) 的满二叉树,每个结点表示一个处理器,因此,对于一个具有 d 级的树形连接方式,共有 $n = 2^d - 1$ 个处理器组成。在此结构中,处理器的工作方式通常是:叶子结点对数据进行计算,而内部结点仅负责叶子结点间的通信及简单的逻辑运算。图 11 .6 给出了 d = 4 的树形连接结构。

5. 洗牌-交换连接方式

洗牌-交换是一类非常有用的互连结构。对于 $n = 2^k$ 个处理器,将它们按 0,1,2,…,$2^k - 1$编号,设处理器 i 的二进制表示为 $i_{k-1}, i_{k-2}, \cdots, i_1, i_0$。

下面定义洗牌与交换二个连接函数:

$$SH(i_{k-1} i_{k-2} \cdots i_0) = i_{k-2} i_{k-3} \cdots i_0 i_{k-1}$$

$$EX(i_{k-1} i_{k-2} \cdots i_1 i_0) = i_{k-1} i_{k-2} \cdots i_1 \bar{i}_0$$

其中,$\bar{i}_0 = 1 - i_0$。在此连接方式中,处理器 i 与二进制表示为 $i_{k-2} i_{k-3} \cdots i_0 i_{k-1}$ 或 $i_{k-1} i_{k-2} \cdots i_1 \bar{i}_0$

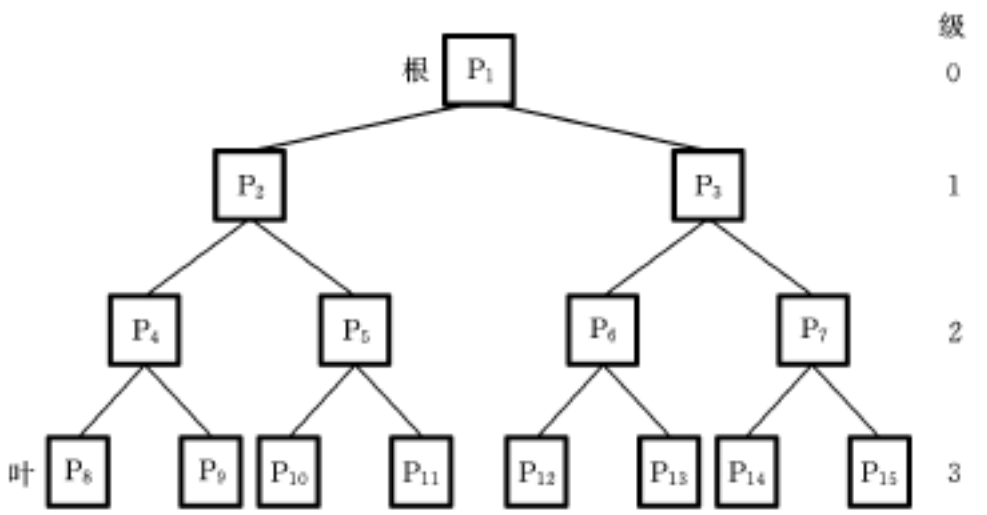


图 11 .6　4 级树形结构

的处理器相连。图 11 .7 示出了 n = 2^3 的洗牌-交换连接结构,其中实线表示 EX 函数,虚线表示 SH 函数。

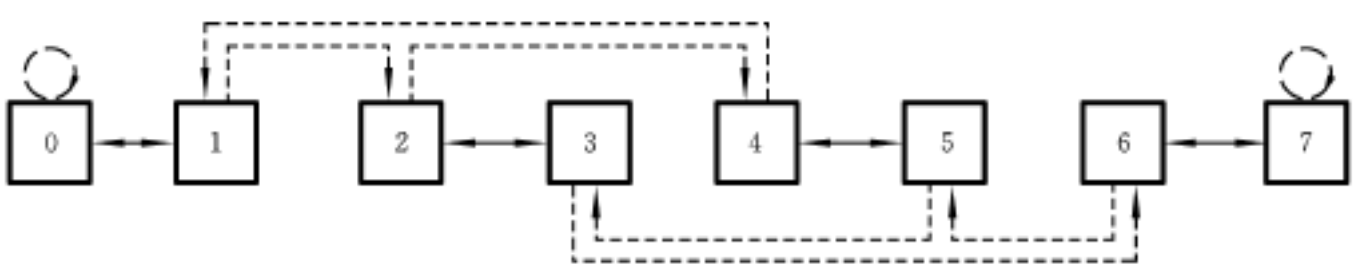


图 11 .7　n = 8 的洗牌-交换网络

11 .2　并行算法的基本概念

对并行算法的研究,可以追溯到 20 世纪 60 年代初期,它与并行计算机的研制是同时进行的。并行计算机系统的研制与应用推动了并行算法的研究和发展;同时,有效的并行算法研究又给并行计算机系统的研究与发展予以有力的支持。

什么是并行算法 ?它可理解为:适合于在某类并行计算机上求解问题和处理数据的算法,是一些可同时执行的诸进程的集合,这些进程相互作用和协调作用,从而达到对给定问题的求解。

11 .2 .1　并行算法的复杂性度量

并行算法可用不同的标准度量,但主要要关心的是算法与求解问题规模之间的关系。对于并行算法的某一属性,本章主要是指运算时间 T。设 T 与求解问题的规模 n 之间的关系是 T = T(n) ,则对于此函数的度量就是并行算法 (时间) 复杂性度量。

对于一个求解问题,其输入的情况通常是千变万化的,这就使得函数 T(n) 非常复杂和不易确定。为了度量并行算法的运算时间的属性,与串行算法一样,采用渐进的方法度量函数T(n)。首先假定问题的规模 n 趋向无穷大,在此假定下确定 T(n) 的上界、下界和精确界,并分别用符号 O、Ω 和Ⓗ表示。例如,若 T(n) = O(g(n)) ,则 g(n)是 T(n)的上界;若 T(n) = Ω (g(n)) 则 g(n) 是 T(n) 的下界;若 T(n) = Θ(g(n)) 则 g(n) 是 T(n) 的精确界。

在 SIMD 计算模型上的并行算法,其复杂性是指,在最坏情况下,算法的运行时间 T(n) 和所需的处理机的数目 P(n) ,其中 n 为问题的规模,其复杂性度量是确定 T(n) 和 P(n) 的上界。

在 MIMD 计算模型上,对于基于共享存储的并行算法,其复杂性度量与 SIMD 模型上的并行算法基本一致。而对于异步通信的分布式计算模型,其算法复杂性衡量标准主要有两个,一个是在最坏情况下算法的运算时间,另一个是在最坏情况下算法的通信复杂性,即在最坏情况下算法在整个执行期间所传送的消息总数目。

在分析算法的运算时间时,必须考虑通信时间,由于处理机之间传递的消息到达目的地的时间通常不易确定。例如,一则消息可能因通信链路被占用而需先等待。因此要想精确地分析算法的运算时间就变得非常困难。目前,估计算法的运算时间都是假定邻接处理机之间的通信可在 O(1) 时间内完成这一假定基础上得出的,不考虑信息的等待时间。在基于异步通信的分布式计算机模型中,并行算法复杂性是以通信复杂性为主要衡量标准,其次才是算法的运算时间。

11.2.2 并行算法的性能评价

在求解一个给定问题的众多的并行算法中,如何评价孰优孰劣呢?下面介绍 3 种公认的评价并行算法性能的标准。

1. 并行算法的成本 C(n)

并行算法的成本 C(n) 定义为并行算法的运算时间 T(n) 与并行算法所需处理机数目 P(n) 的乘积,即

$$C(n) = T(n) \cdot P(n) \tag{11.1}$$

它相当于在最坏的情况下求解一问题的总的执行步数。如果求解一问题的并行算法的成本在数量级上等于最坏情况下串行求解此问题所需的执行步数,则称这样的并行算法是成本最优的。

2. 加速 (又称加速比) $S_p(n)$

对一个求解问题,令 $T_s(n)$是最快的串行算法在最坏的情况下的运算时间,$T_p(n)$是求解此问题的某一并行算法在最坏情况下的运算时间,则此并行算法的加速 $S_p(n)$ 可定义为

$$S_p(n) = T_s(n) / T_p(n) \tag{11.2}$$

由定义可看出,并行算法的加速反映了该算法的并行性对运算时间的改善程度。$S_p(n)$越大,则并行算法越好。由于任何并行算法都能在一台串行机上模拟实现,因此 $T_p(n) \cdot P(n) \geq T_s(n)$,从而

$$1 \leq S_p(n) \leq P(n)$$

当 $S_p(n) = P(n)$ 时,则具有这样加速比的并行算法为最优的并行算法。由于在通常情况下,一个问题不可能分解成 P(n) 个具有相同执行步数并且可以并行执行的子问题,因此,要达到$S_p(n) = P(n)$几乎是不可能的。

3. 并行算法的效率 $E_p(n)$

并行算法的效率可定义为算法的加速与处理机数目之比,即

$$E_p(n) = S_p(n) / P(n) \tag{11.3}$$

并行算法的加速不能反应处理机的利用率,一个并行算法的加速可能很大,但处理机的利用率却可能很低。并行算法的效率反映了在执行算法时处理机的利用情况。由于 $1 \leq S_p(n) \leq P(n)$,故

$$0 < E_p(n) \leq 1$$

若一个并行算法的效率等于 1 ,则说明在执行此算法的过程中每台处理机的能力得到了充分发挥。此时,$S_p(n) = P(n)$,因此,此并行算法的串行模拟是求解问题的最佳串行算法。当然,要达到 $E_p(n) = 1$ 几乎是不可能的。

11.2.3　并行算法的设计

对于某一求解问题,在设计并行算法时,一般可采用 3 种途径:

(1) 把已有的串行求解算法进行改造,挖掘开发串行算法中固有的并行性,使之适合于在并行计算机上运行。

(2) 从求解问题本身出发,分析和改造求解问题中运算操作之间的相互关系和数据的相关性,将问题划分成若干个能彼此并行执行的子问题。在此途径中,应尽量采用一些像分治策略那样具有并行思想的设计策略。

(3) 借鉴和改造求解类似问题的并行算法,产生求解本问题的并行算法。

特别需要注意的是在采用途径(1) 设计算法时,不要认为一个好的串行算法一定能转换成一个好的并行算法。对于一个好的串行算法,其内在并行性可能并不好。同时,在有些计算模型上,并行算法的复杂性度量必须考虑通信成本,因此,有时一个性能不算好的串行算法有可能转换成一个性能好、效率高的并行算法。

在设计并行算法时,要注意以下几点。

(1) 并行算法依赖于计算模型:一个并行算法的性能在不同的计算模型上差别很大。这有时候是由于通信开销所致,但有时候是由于别的因素所致。例如,为了实现同步,在 MIMD 计算模型上需要通过很费时间的软件来实现。因此,小粒度(同步之间执行很少步运算)算法不适合在 MIMD 共享存储模型上运行。

(2) 在存储分布模型上必须考虑通信成本:在基于存储共享的计算模型上,由于处理机之间的通信是通过全局共享存储变量完成的,因此通信开销可忽略不计。在存储分布计算模型上,数据通过互连网络从一个处理机传送到另一个处理机的开销可能很大(须经很多中间结点),因此,有时候算法的通信复杂性比计算复杂性还要高,也就是说,通信所花费的时间比实际处理、变换所花费的时间还要长。

(3) 算法与数据的存储分布有关:在互连网络计算模型中,算法与数据分布密切相关,例如,在单指令流的互连网络模型中,数据存储分布的好坏直接影响算法的并行性;在多指令流的互连网络模型中,数据存储分布的好坏直接影响算法的通信开销。

11.2.4　并行算法的描述

和串行算法的描述一样,描述一个并行算法,通常可以用非形式化描述或形式化描述两种方式。所谓非形式化描述就是用通常的自然语言来表达其内容,所谓形式化描述是使用某种程序设计语言来将算法书写出来。为了便于表达算法所具有的特性,最好采用形式化

描述。选用语言时,其基本要求是用该语言所写的语句不能有二义性,同时强调直观、易理解,而不苛求严格的语法格式。本章采用 SPARKS 语言书写并行算法。

在并行算法的形式化描述中,所有串行算法的语句和过程调用等均可使用,与串行算法不同之处是在描述并行操作时须采用并行语句。下面介绍 3 条常用的并行语句:

(1) for each $i_1, i_2, \cdots, i_k$: P pardo

…

endfor

此语句表示编号为 $i_1, i_2, \cdots, i_k$ 的处理机,若满足谓词 P 为真,则执行循环体的指令序列。要用于描述 SIMD 模型中的并行算法。

(2) upon receiving M message from u do

…

此语句表示执行结点一旦收到来自结点 u 的消息 M 后,就执行相应的操作。

(3) send M message to k

此语句表示执行结点把信息 M 传送给 k。

语句 (2)、(3) 是描述互连网络模型中并行算法的通信功能。

以上对并行算法的一般概念做了简单的介绍,有关这方面的知识,读者要通过以下各节进一步巩固和加深。

11.3 SIMD 共享存储模型上的并行算法

在 SIMD 共享存储模型中设计算法颇受算法研究者欢迎,因为它抛开了各种具体的体系结构,使人们能集中精力从问题本身出发,研究和挖掘求解问题本身固有的并行性,简化算法的设计和分析,易于算法间的相互比较,使得并行算法的研究成为一项独立的活动。下面讨论几个在该模型上的并行算法。

11.3.1 广播算法

广播算法能解决 SIMD-EREW 模型上读冲突问题,因此,在此模型的许多求解问题中都要用到它。假定有 N 个处理器(编号从 1 到 N),都要读取共享存储器中的数据 m,对于这个简单问题,在 SIMD-CREW 模型上,显然只需一次并行读操作就可以完成。在 SIMD-EREW 模型上,由于不允许读冲突,故若采用处理器依次读取数据 m,则是一种串行操作,没有发挥并行处理机的并行处理能力,其运算时间为 O(N)。下面给出解决此问题的并行算法。假定 N 为 2 的幂,则其基本思想是使用一个长度为 N 的共享数组 B(开始为空)。第一步由处理器 P_1 把 m 写入 B(1)。第二步由处理器 P_1 把 B(1)写入 B(2);由 P_1, P_2 把 B(1),B(2)并行写入B(3),B(4);……由 $P_1, P_2, \cdots, P_i$ 把 B(1),B(2),…,B(i) 并行写入 B(i+1),B(i+2),…,B(2i);……最后由 $P_1, P_2, \cdots, P_{N/2}$ 把 B(1),B(2),…,B(N/2) 并行写入 B(N/2+1),B(N/2+2),…,B(N)。第三步处理器 P_i (i=1,2,…,N) 从 B(i)中并行读取数据 m。其算法如下:

```
procedure BROADCAST (m, N, B)
```

```
(1) 由处理器 P1 执行:B(1)←m
(2) for i = 0 to logN - 1 do
        for each j: 1≤j≤2^i  pardo
          处理机 Pj 执行:B(2^i + j)←B(j)
        endfor
    repeat
(3) for i: 1≤i≤N pardo
      处理机 Pi 从 B(i)中读取数据 m
    endfor
  end BROADCAST
```

由算法不难看出,第(1)、第(2)、第(3)步的运算时间分别为 O(1)、O(logN)和 O(1)。因此,广播算法的运算时间为 O(logN)。

下面评价广播算法的性能:

(1) 成本。由(11.1)式直接推出该算法成本为 O(NlogN)。

(2) 加速。由于此问题的最佳串行算法的运算时间为 O(N),因此,广播算法的加速为

$$S_p(N) = O(N/\log N)$$

(3) 效率。该算法的效率为

$$E_p(N) = S_p(n)/N = O(1/\log N)$$

在 BROADCAST 算法中是假定了处理机个数 N 为 2 的幂,若此条件不满足,则将算法第(2)步中 for i = 0 to logN - 1 do 改成 for i = 0 to $\lceil \log N - 1 \rceil$ do 即可,修改后的算法的运算时间和性能都保持不变。

11.3.2　求和算法

假定共享存储器中有 N 个数据 $S_1, S_2, \cdots, S_N$。利用 N - 1 个处理器 $P_2, \cdots, P_N$ 求出局部与整体和 $S_1 + S_2 + \cdots + S_j (1 < j \leq N)$。

此问题求解可以通过并行执行以下操作来实现:

第 1 步　处理器 $P_i (2 \leq i \leq N)$完成 $S_i \leftarrow S_i + S_{i-1}$;

第 2 步　处理器 $P_i (3 \leq i \leq N)$完成 $S_i \leftarrow S_i + S_{i-2}$;

第 3 步　处理器 $P_i (5 \leq i \leq N)$完成 $S_i \leftarrow S_i + S_{i-4}$;

……

第 J 步　处理器 $P_i (2^{j-1} + 1 \leq i \leq N)$完成 $S_i \leftarrow S_i + S_{i-2^{j-1}}$,其中 $j = \lceil \log N \rceil$。

图 11.8 描述了当 N = 7 时并行求和的执行过程。

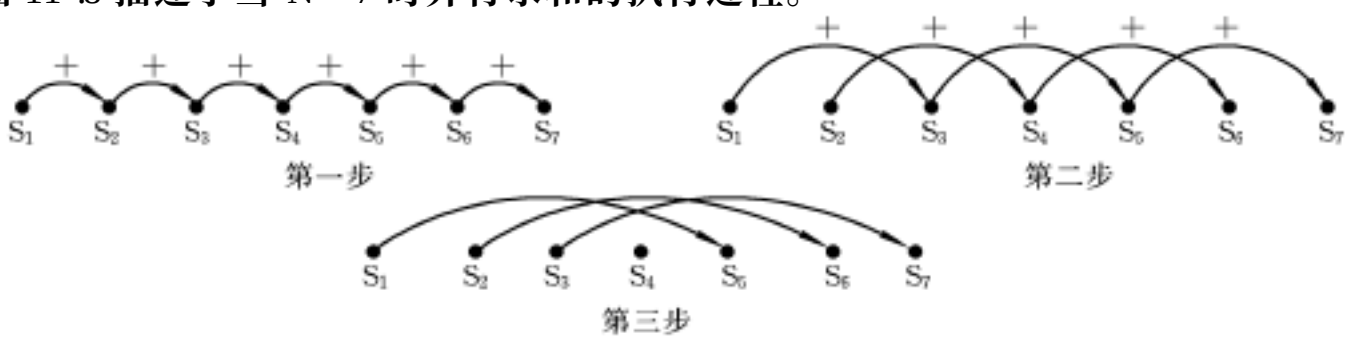


图 11.8　求和示意图

并行求和算法如下:

```
procedure AllSUMS (S1, S2, ⋯, SN)
    for j = 1 to ⌈logN⌉ do
        for each i: 2^(j-1) + 1 ≤ i ≤ N pardo
            处理器 Pi 从共享存储器中取 Si;
            处理器 Pi 从共享存储器中取 S(i-2^(j-1));
        Si ← Si + S(i-2^(j-1))
        endfor
    repeat
    end ALLSUMS
```

不难看出,在算法执行过程中不存在读、写冲突,因此该算法可在 SIMD-EREW 模型上实现。由于每一并行步仅包含三个操作,故算法的运算时间为 O(logN)。因为求和问题的最佳串行算法的运算时间为 O(N),故此并行算法的加速为 O[N/logN],其效率为 O[1/logN]。

对 ALLSUMS 算法稍加修改,就可解决其它一些问题。例如,将算法中的语句 $S_i \leftarrow S_i + S_{i-2^{j-1}}$ 改成 $S_i \leftarrow S_i * S_{i-2^{j-1}}$ 则变成了并行求积算法;将 $S_i \leftarrow S_i + S_{i-1^{j-2}}$ 改成 $S_i \leftarrow \max(S_i S_{i-2^{j-1}})$,则变成了并行求最大值算法。修改后,算法的运算时间和性能都保持不变。

11.3.3 并行归并分类算法

和串行归并分类算法一样,在设计并行归并分类算法之前,也必须首先设计出并行归并算法。

假定有两个长度分别为 p 和 q 的已分类序列,并行归并算法是用 k(k > 1)个处理器将它们合并成长度为 p + q 的分类序列。处理器个数 k 的取值不同可能产生复杂度不同的并行归类算法。这里仅就处理器个数 $k = \lfloor\sqrt{pq}\rfloor$ 的情况构造算法。算法的基本思想是,利用分治策略对序列进行分组,并且这种分组方式是动态地和递归地进行。归并过程可简述如下。

首先,在两序列中选定一些特定元素,并加以标记,这些标记元素将两序列分成了若干组,接着将第一序列中诸标记元素与第二序列中的诸标记元素进行比较,以确定第一序列中的每一标记元素应插入第二序列中的哪一组中;然后将第一序列中的诸标记元素与第二序列中它所插入的那一组中的其余元素进行比较,以确定第一序列中的每一个标记元素应插入第二序列中的哪一个位置上才能使序列保持有序。插入后,撇开原第二序列中的特定元素不管,则第一序列中的特定元素将第二序列分成若干个组,并且两个序列中的组构成组对。注意,我们规定组中元素不包含特殊元素,因此对于每个组对均存在 3 种情况:① 第一组为空;② 第二组为空;③第一、二组都不为空(这时第一组和第二组中元素分别来自第一和第二序列)。称情况③中的组对为非空组对。下面的工作是针对每一组对将第一组中元素插入第二组中去。对于情况①,不需插;对于情况②,是直接搬移;对于剩下的非空组对(若有的话),它们各自构成了独立的归并问题,因此,可并行地对各非空组对递归调用原算法,如此下去,直到不存在非空组对为止。下面给出并行归并算法的非形式化描述:

```
procedure PARA-MERGE (p,q)
// 将两个长度分别为 p、q 的已分类序列 A 和 B 用 ⌊p·q⌋ 个处理器归并成一个有序 // 序列,约定 p≤q //
```

(1) 将 A、B 序列中位置分别是 $i\lceil\sqrt{p}\rceil$ 和 $i\lceil\sqrt{q}\rceil$ $(i=1,2,\cdots)$ 的一些元素打上 * 号;

(2) 将 A 中每个带 * 号的元素与 B 中每个带 * 号的元素同时进行比较;

(3) 将 A 中带 * 号的元素与其所插入的 B 组中的每一个元素进行比较,并把它们插入 B 中;

(4) 若有情况②的组对,则将组对中第一组元素搬入 B 中;

(5) 若有情况③的组对,则按处理器的个数 $=\lfloor\sqrt{pq}\rfloor$ 的分配原则给每个子问题配置处理器,然后对各个子问题并行调用 PARA-MERGE 算法。

end PARA-MERGE

现在来分析一下算法中第(5)步递归归并时,原来的 $\lfloor\sqrt{pq}\rfloor$ 台处理器够不够用?当执行到第(5)步时,设序列 A 和 B 中各组中的元素个数分别为 p_i 和 q_i,显然,序列 A 的各组元素个数之和 $\sum p_i \le p$;序列 B 的各组元素之和 $\sum q_i \le q$。根据柯栖不等式

$$\sum \sqrt{p_i q_i} \le \sqrt{\sum p_i \sum q_i}$$

所以,$\sum\lfloor\sqrt{p_i q_i}\rfloor \le \lfloor\sum \sqrt{p_i q_i}\rfloor \le \lfloor\sqrt{\sum p_i \sum q_i}\rfloor \le \lfloor\sqrt{pq}\rfloor$。由此可见,在对各组对的并行递归调用时,处理器是够用的。

下面分析算法的运算时间:

在 SIMD-CREW 模型上,因为序列 A 和 B 中要打 * 号的元素至多有 $\lfloor\sqrt{p}\rfloor$ 和 $\lfloor\sqrt{q}\rfloor$ 个,所以至多用 $\lfloor\sqrt{p}\rfloor+\lfloor\sqrt{q}\rfloor$ 个处理器在常数时间内完成算法中的第(1)步;在算法的第(2)步中,至多比较 $\lfloor\sqrt{p}\rfloor\lfloor\sqrt{q}\rfloor$ 次,因此,至多用 $\lfloor\sqrt{p}\rfloor\lfloor\sqrt{q}\rfloor$ 个处理器可在常数时间内完成算法的第(2)步;同样可证明至多用 $\lfloor\sqrt{p}\rfloor\lfloor\sqrt{q}\rfloor$ 个处理器在常数时间内完成算法的第(3)步和第(4)步。在算法的第(5) 步中因为任意一个非空组对的第一组元素个数小于等于 $\lceil\sqrt{p}\rceil - 1 \le \lfloor\sqrt{q}\rfloor$,若设 PARA-MERGE 算法的运算时间为 $T(p)$,则第(5)步的运算时间 $\le T(\sqrt{p})$。因此,$T(p)$ 满足关系式

$$T(p)\le\begin{cases} C_1 & p \text{ 足够小} \\ T(\sqrt{p})+C_2 & \text{否则}\end{cases}$$

其中,C_1、C_2 是常数。解此关系式,可得

$$\begin{aligned} T(p) &\le T(p^{1/2})+C_2 \\ &\le T(p^{1/4})+2C_2 \\ &\cdots \\ &\le T(2)+(\log\log p)C_2 \\ &= O(\log\log p)\end{aligned}$$

由此得到 PARA-MERGE 算法在 SIMD-CREW 模型上的运算时间为 $O(\log\log p)$。

对于 SIMD-EREW 模型,由于算法的第(2)和第(3)步中存在读冲突,因此,需要调用广播算法,这时 PARA-MERGE 算法的运算还与 q 有关,若记为 $T(p,q)$,则其满足关系式

$$T(p,q)\le\begin{cases} C_1\log\sqrt{q} & p \text{ 足够小} \\ T(\sqrt{p})+C_2\log\sqrt{q} & \text{否则}\end{cases}$$

解此关系式,得到 $T(p,q)\le C\log q+\log q\cdot\log\log p = O(\log q\log\log p)$。因此,在 SIMD-EREW 模型上,PARA-MERGE 算法的运算时间为 $O(\log q\cdot\log\log p)$。

有了并行归并算法后,并行归并分类算法就可通过并行归并来构造。设序列的长度为 $n = 2^k$, $k \geq 1$ 并行归并分类算法描述如下:

第一并行步　进行$(2^0, 2^0)$并行归并;

第二并行步　进行$(2^1, 2^1)$并行归并;

……

第 k 并行步:　进行$(2^{k-1}, 2^{k-1})$并行归并。

下面分析此算法在 SIMD-CREW 模型上的运算时间和性能。

设运算时间为 sort (n) ,当 $i > 1$ 时$(2^i, 2^i)$并行归并的运算时间为 $O(\log\log 2^i)$,所以

$$\text{sort}(n) \leq \sum_{i=1}^{k} (C\log\log 2^i) = C\sum_{i=1}^{k} \log i \leq Ck\log\log n$$

即

$$\text{sort}(n) = O(\log n\log\log n) \tag{11.4}$$

设算法所需处理机个数为 P(n),所以

$$P(n) = \max(2^{k-1}2^0, 2^{k-2}2^1, \cdots, 2^0 2^{k-1}) = O(n) \tag{11.5}$$

由于最佳的串行分类算法的运算时间为 $O(n\log n)$,因此,并行归并分类算法的加速和效率分别是

$$S_p(n) = O(n/\log\log n) \tag{11.6}$$

$$E_p(n) = O(1/\log\log n) \tag{11.7}$$

对于 n 不是 2 的幂的情况,只需在序列中增加若干元素,这些元素都大于原序列中的每个元素,并且使得序列长度为 2 的幂。可以证明,经过这样的预处理后,结论式(11.4)、(11.5)、(11.6)和(11.7)都仍然正确。

在 SIMD-EREW 模型上,并行归并分类算法的运算时间和性能评价留作习题,不在这里赘述。

11.3.4　求图的连通分支的并行算法

无向图 G(V, E)的一个连通分支是 G 的一个最大连通子图。在 SIMD 共享存储模型上,求图的连通分支的方法主要有 3 种:采用某种形式的搜索方法、利用图的邻接矩阵的传递闭包法和顶点倒塌法。这里只介绍传递闭包法,顶点倒塌法将在 SIMD 互连网络中介绍。

定义 11.1　设无向图 G(V, E) 的邻接矩阵是 A,若矩阵 B 的元素定义如下:

$$b_{ij} = \begin{cases} 1 & \text{顶点 i 与顶点 j 之间有路径存在} \\ 0 & \text{顶点 i 与顶点 j 之间没有路径存在} \end{cases}$$

则称 B 是 A 的自反传递闭包。

邻接矩阵 A 与其自反传递闭包 B 存在下列关系:

$$B = (I + A)^n$$

其中,I 为单位矩阵;符号“ + ”定义为逻辑加;n 为图结点个数。下面根据这一关系来设计求连通分支的并行算法。

假定共享存储器中保存了图 G(V, E) 的邻接矩阵 A,定义 G 中任一个分支号为此分支中结点号的最小值。算法首先计算出 A 的自反传递闭包 B ,然后计算出矩阵 C,其中,

$$C(i,j) = \begin{cases} \infty & b_{ij} = 0 \\ j & b_{ij} = 1 \end{cases}$$

最后由 $d(i) = \min(c(i,1), c(i,2), \cdots, c(i,n))$ 计算出 G 中任意结点 $i(1 \leq i \leq n)$ 所在的分支号 D(i)。

其算法如下:

```
procedure connected-components G(V、E)
(1) for each p_ij :1≤i,j≤n pardo // 初始化 //
      if i = j then a(i,j) ←a(i,j) VI endif
      b(i,j)←a(i,j)
      d(i)←i
    endfor
(2) for l←1 to ⌈logn⌉ do // 计算传递闭包 B //
    for each p_ijk :1≤i,j,k≤n pardo
      b′(i,j,k) = b(i,k) ∧ b(k,j);
      b(i,j) = ∨_{k=1}^{n} b′(i,j,k)
    endfor
repeat
(3) for each p_ij :1≤i,j≤n pardo // 计算 C //
      if b(i,j) = 1 then c(i,j)←j
                    else c(i,j)←∞
      endif
    endfor;
(4) for each p_ij :1≤i,j≤n pardo // 求每个结点所在的连通分支标号 //
      d(i)←min {c(i,j) | 1≤j≤n}
      endfor
end connected-components
```

算法的第(2)步是依次并行求出 $(I + A)^2, (I + A)^4, \cdots, (I + A)^{2^{\lceil \log n \rceil}}$，由于自反传递闭包具有性质: $(I + A)^n = (I + A)^{n+T}$，其中 T 为任意正整数，因此第(2)步可求出 $(I + A)^n$。在第(2)步中，语句 $b(i,j) = \vee_{k=1}^{n} b'(i,j,k)$ 是用 n 个处理器求 $b'(i,j,1) \vee b'(i,j,2) \vee \cdots \vee b'(i,j,n)$，因此，可用求和并行算法，用 n - 1 个处理器在 O(logn) 时间内完成。在第(4)步中，语句 $d(i) \leftarrow \min\{c(i,j) \mid 1 \leq j \leq n\}$ 也可利用求和算法，用 n - 1 个处理器在 O(logn) 时间内完成。

由于存在读冲突，因此，此算法是 SIMD-CREW 模型上的算法。

定理 11.1　在 SIMD-CREW 模型上，计算无向图 $G(V,E), |V| = n$ 的连通分支 connected-components 算法需要 $O(\log^2 n)$ 时间和 $O(n^3)$ 个处理器。

证明　因为算法的第(1)步需 O(1)时间和 O(n)个处理器；第(2)步由 $\lceil \log n \rceil$ 次串行循环组成，对于每一次串行循环操作需 O(logn)时间和 $O(n^3)$ 个处理器，所以第(2)步需 $O(\log^2 n)$时间和 $O(n^3)$个处理器；第(3)步需 O(1)时间和$O(n^2)$个处理器；第(4)步需 O(logn)时间和$O(n^2)$个处理器。因此，算法需 $O(\log^2 n)$时间和 $O(n^3)$个处理器。证毕。

如果计算模型容许写冲突，那么对算法进行适当的修改可使运动时间降为 O(logn)。这项工作的关键是:在 SIMD-CRCW 模型上设计使用 $O(n^2)$个处理器，能在 O(1)时间求出 n 个元素中最大元素的并行算法。具体做法如下:

假定我们要找 $c_1, c_2, \cdots, c_n$ 这几个元素的最大值,则第(1) 步用 n^2 个处理器,在 O(1)时间内求出 t_{ij} $(1 \leq i, j \leq n)$,其中

$$t_{ij} = \begin{cases} 1 & c_i \geq c_j \\ 0 & \text{否则} \end{cases}$$

在第一步中存在读冲突,这一步的工作使得矩阵 $T = [t_{ij}]_{n \times n}$ 具有如下性质: c_k 是 $c_1, c_2, \cdots, c_n$ 中的最大元素的充要条件是矩阵 T 的第 k 行中 n 个元素皆为 1。

证明由矩阵 T 中元素 t_{ij} $(1 \leq i, j \leq n)$ 的定义可直接推出上述性质。详证略。

第(2)步用 $O(n^2)$ 个处理器,在 O(1)时间内求矩阵 T 中 n 个元素皆为 1 的某一行,并将其行号赋予 k,此步可由下列语句实现:

```
for each p_ij :1≤i,j≤n pardo
        if t_ij = 0 then t_ij ←0 endif
endfor;
for each p_i :1≤i≤n pardo
        if t_ij = 1 then k←i endif
endfor
```

显然,第(2)步中存在写冲突,在 SIMD-CRCW 模型上通过上面两步可确定 $c_1, c_2, \cdots c_n$ 中的最大元素是 c_k。

下面对 connected-components 算法中第(2)步进行修改。在矩阵 $B, B^2, B^4, \cdots, B^{2^{\lceil \log n \rceil}}$ 中,每个矩阵的矩阵元素都是 0 或 1,因此 $B^{2^l} = [c_{ij}]$,其中 $c_{ij} = \max\{b_{i1} \wedge b_{1j}, b_{i2} \wedge b_{2j}, \cdots, b_{in} \wedge b_{nj}\}$, b_{ij} 为矩阵 B 中的元素, $1 \leq i, j \leq n$, $1 \leq l \leq \lceil \log n \rceil$。这样,connected-components 算法的第(2)步可改写成:

```
for l←1 to ⌈logn⌉
    for each p_ijk :1≤i,j,k≤n pardo
        b′(i,j,k) = b(i,k)∧b(k,j)
    endfor
    for each i,j:1≤i,j≤n pardo
        b(i,j) = max{b′(i,j,1),b′(i,j,2),⋯,b′(i,j,n)}
    endfor
repeat
```

由于求 n 个元素的最大值可用 $O(n^2)$ 个处理器在 O(1)时间实现,因此,修改后算法的第(2)步的运算时间为 O(logn),需 $O(n^4)$ 个处理器。对于整个算法有下面结论。

定理 11.2 在 SIMD-CRCW 模型上,求无向图 $G(V, E)$, $|V| = n$ 的所有连通分支的自反传递闭包算法需 O(logn) 时间和 $O(n^4)$ 个处理器。

由上面的讨论不难证明本定理。详证略。

11.4 SIMD 互连网络模型上的并行算法

11.4.1 超立方模型上的求和算法

假定在 m-维超立方模型上, $n = 2^m$, n 个待加的数的集合表示为: $A = (a_0, a_1, \cdots, a_{n-1})$;

且对于所有的 $0 \leq i \leq n-1$，处理器 P_i 存有局部变量 a_i。下面在此模型上构造一个并行求和算法，使得算法结束时 a_0 就是总和 $\sum_{i=0}^{n-1} a_i$。

对于处理器 P_i，$0 \leq i \leq n-1$，设其号码 i 的二进制表示是

$$i_{m-1} i_{m-2} \cdots i_0$$

其中，$i_j = 0$ 或 1，$0 \leq j \leq m-1$。这样，可按处理器号码 i 的第 m - 1 位值 i_{m-1} 将 n 个处理器分成两类：第一类中 $i_{m-1} = 0$，第二类中 $i_{m-1} = 1$。利用超立方模型中结点相邻关系可建立二类处理器集合中元素的一一对应关系。根据这种对应关系，即可构造并行算法。

算法的非形式化描述如下：

第一并行步　对于 n 个处理器，以 i_{m-1} 的值分类，第一类中的处理器取其在第二类中所对应的处理器中的局部变量，然后与其本身的局部变量相加作为它的局部变量；

第二并行步　对于 n/2 个处理器，(即上一步中第一类的处理器)，以 i_{m-2} 的值分类，第一类中的处理器取其在第二类中所对应的处理器的局部变量，然后与其本身的局部变量相加作为它的局部变量；

……

第 m 并行步　对于 2 个处理器，(即上一步中第一类中的处理器)，以 i_0 的值分类，第一类中的处理器取其在第二类中所对应的处理器的局部变量，然后与其本身的局部变量相加作为它的局部变量。

由于每一步并行操作后，第一类处理器以“ + ”的形式保存了 n 个局部变量 $a_0, a_1, \cdots, a_{n-1}$ 的信息，故当执行完第 m 步后第一类中仅有处理器 P_0，它的局部变量 a_0 就是 n 个数的总和。算法的形式化描述如下：

```
procedure SUMMATION (a0, a1, ⋯, an-1)
  for j = (logn) - 1 to 0 do
    d←2^j;
    for each Pi: 0≤i≤d - 1 pardo
    ti    ai+d;
    ai ← ai + ti
    endfor
  repeat
end SUMMATION
```

说明：符号　表示数据从其相邻的处理器的局存中传往一个活动的处理器中的局存中。

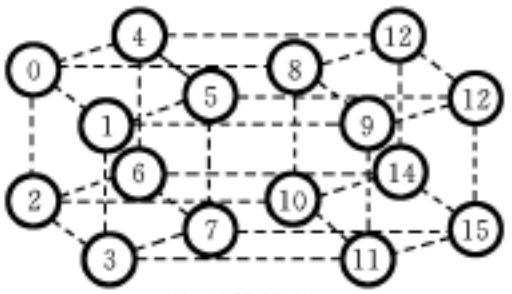
由于算法的外循环执行 logn 次，而每次循环的时间均为常数，所以以上算法的运算时间为 $\Theta(\log n)$。因为求和问题的最佳串行算法的运算时间为 $\Theta(n)$，所以，SUMMATION 算法的加速比为

$$S_p(n) = \Theta(n/\log n)$$

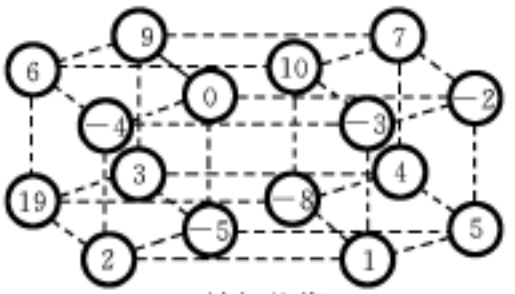
效率为

$$E_p(n) = S_p(n)/n = \Theta(1/\log n)$$

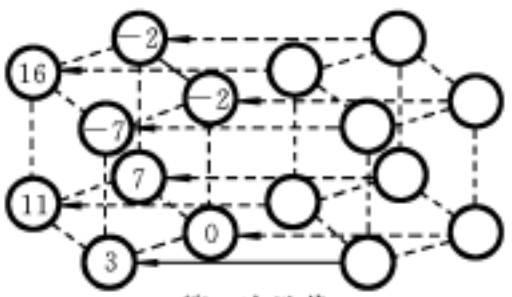
例 11.1　在超立方模型上求 16 个数的和，其过程如图 11.9 所示。



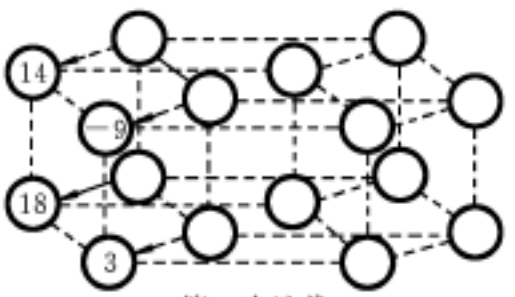
处理器号码



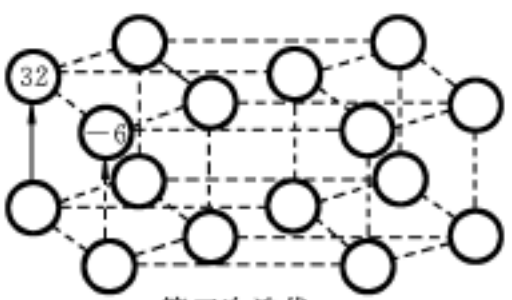
被加的值



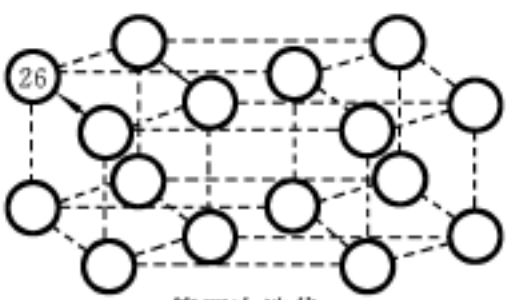
第一次迭代



第二次迭代



第三次迭代



第四次迭代

图 11.9 超立方模型上 16 个数的求和

11.4.2 一维线性模型上的并行排序算法

一维线性模型是 SIMD 互连网络模型中最简单和最基本的互连形式,系统中有 n 个处理器,编号从 1 到 n,每个处理器 P_i 均与其左、右近邻 P_{i-1}、P_{i+1} 相连,(P_1 和 P_n 除外)。我们将在此模型上构造 n 个数的并行排序算法。

假定待排序的 n 个数的输入序列 $S=\{x_1, x_2, \cdots, x_n\}$,处理器 P_i $(1 \leq i \leq n)$ 存有输入数据 x_i,算法步骤如下。

第一并行步 所有奇数编号的处理器 P_i 接收来自 P_{i+1} 中的 x_{i+1}。如果 $x_i > x_{i+1}$ 则 P_i 和 P_{i+1} 彼此交换其内容。

第二并行步 所有偶数编号的处理器 P_i 接收来自 P_{i+1} 中的 x_{i+1}。如果 $x_i > x_{i+1}$ 则 P_i 和 P_{i+1} 彼此交换其内容。

交替重复上述两并行步,经 $\lceil n/2 \rceil$ 次迭代后算法结束。

算法的形式化描述如下:

```
procedure OETS (x1, x2, ⋯, xn)
        for k = 1 to ⌈n/2⌉ do
    for each Pi : i = 1, 3, ⋯, 2⌈n/2⌉ - 1
        pardo
    if xi > xi+1 then xi → xi+1 endif
```

```
        endfor
        for each Pi : i = 2,4,⋯,2⌊(n - 1)/ 2⌋ pardo
          if xi > xi+1 then xi ←→ xi+1 endif
        endfor
      repeat
    end OETS
```

例 11 .2　对输入序列 S = {7, 6, 5, 4, 3, 2, 1}, 上述算法的排序过程见图 11 .10。

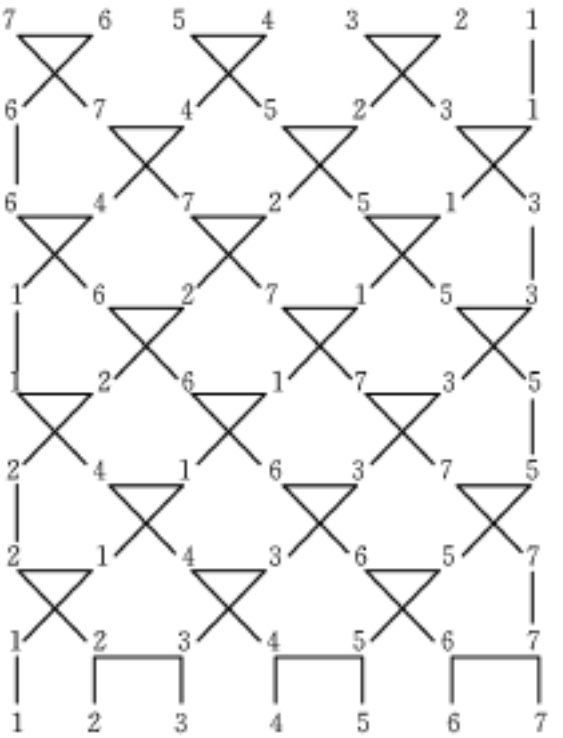


图 11 .10　OETS 算法排序示例

OETS 算法的正确性由定理 11 .3 给予保证。

定理 11 .3　OETS 算法至多经过 n 步可完成排序。

证明　此定理可通过对 n 施行归纳法来证明。

(1) **归纳基础**: 当 n≤3 时, 可用穷举法验证定理是对的。

(2) **归纳假定**: 假定排序 m 个数算法 OETS 至多需要 m 步。

(3) **归纳步骤**: 试图证明排序 m + 1 个数, 算法至多需要 m + 1 步。

为此, 可借助排序图(Sort Graph)来研究算法 OETS 对 $S = \{x_1, x_2, \cdots, x_{m+1}\}$ 的操作情况。在此图中, 追踪 S 中最大元素 M 的轨迹。根据 M 起始时是位于奇数编号和偶数编号的处理器中, 可有两种不同情况, 分别示于图 11 .11(a)和(b)中, 但每种情况都可认为在 k 步内可完成排序 m + 1 个数, 同时 M 的轨迹将排序图分成 A 和 B 两部分。假定现在 M 不存在了, 如图 11 .12(a)所示擦去 M 的轨迹, 将 A 和 B 如同图 11 .12(b)那样合并之, 这样从第 2 行向下就是一个完整的 m 个数的排序图。根据归纳假定可知: k - 1 = m, 所以 k = m + 1。证毕。

下面进行算法分析, 很容易求出 OETS 算法的运算时间 T(n) = O(n), 进而推出算法的成本 $c(n) = T(n)p(n) = O(n^2)$, 加速 $S_p(n) = O(\log n)$。

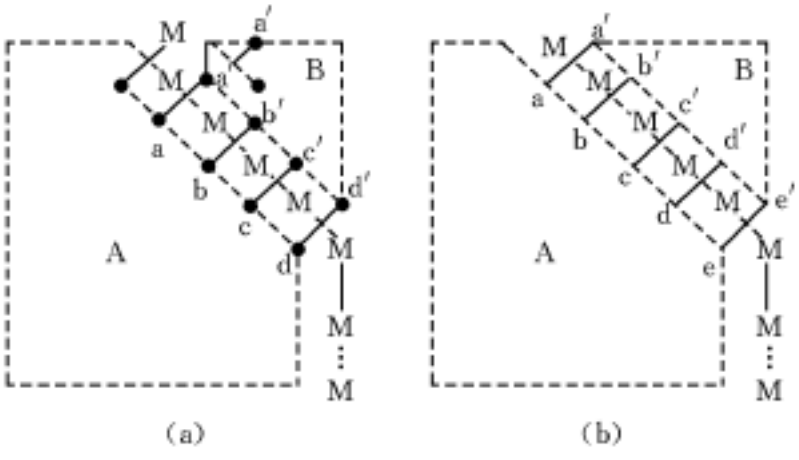


图 11 .11　在排序图中最大元素的轨迹

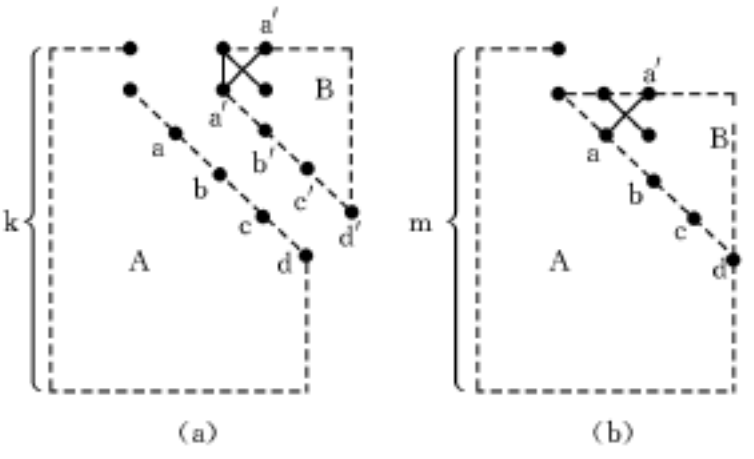


图 11 .12　证明定理 11 .3 的排序图

11 .4 .3　树形模型上求最小值算法

在求集合 $S = \{x_1, x_2, \cdots, x_n\}$ 的最小元素问题中, 首先假定 $n = 2^d (d > 0)$, 因此, 树形模型共有 2^d 个叶子, 总共有 $2^{d+1} - 1$ 个结点, 即有 $P(n) = 2^{d+1} - 1$ 个处理器。其中每个叶处理

器能存放一个数,非叶处理器能存放两个数并可决定何者为小。算法的基本思想是:首先将 S 中的 n 个元素加载到各叶处理器中,然后从 d 级开始(根为第一级)每个非叶处理器从左、右儿子中取出两个数进行比较,并保存较小者,最后使根结点保存集合 S 中的最小元素。其算法如下:

```
procedure MIN(x1,…,x2^d)
1   for each Pi:2^d ≤ i ≤ 2^(d+1) - 1 pardo
      读取 x(i-2^d+1)
      endfor
2   for j = d to 1 do
          for each Pi:2^(j-1) ≤ i ≤ 2^j - 1 pardo
          从左、右儿子中取出两个数进行比较,并保存最小者;
          endfor
    repeat
end MIN
```

算法分析如下:

很显然,算法的第 1 步运算时间为 O(1);因为 d = logn,所以算法的第 2 步的运算时间为 O(logn),因此算法的运算时间 T(n) = O(logn)。由于求极值的最佳串行算法的运算时间为 O(n),所以 MIN 算法的加速 $S_p(n) = O(n/\log n)$。算法的成本 $c(n) = T(n) \times P(n) = O(n\log n)$,显然该算法不是成本最佳的。

11.4.4 二维网孔模型上的连通分支算法

这里采用顶点倒塌法在二维网孔模型上求无向图 G(V,E)的连通分支。

定义 11.2 图 G 的一个超顶点集 A 是 G 的一个连通子图,此集由 A 中最小标号结点标识,这个结点称为 A 的根。并且对 A 中任一结点 i 赋一个指针信息:D(i) = k,其中 k 为 A 的根的标号。

顶点倒塌法的基本思想是:一开始 G 中每个结点都当成一个超顶点集,然后是超顶点集的合并、扩大过程,此过程由一系列循环完成。每次循环分为三步:

第一步 对于 G 中每个超顶点集 i,给其每个结点 i_j 赋个 0 值,使其满足当存在与 i_j 相邻的结点且此结点不在集 i 中时,$c(i_j) = \min\{D(s) / s$ 与 i_j 相邻,且 s 不在超顶点集 i 中$\}$,否则 $c(i_j) = \infty$;

第二步 对于 G 中每个超顶点集 i,修改其根结点 i 的 D 值,使其满足

$$D(i) = \begin{cases} \min\{c(t) \mid D(t) = i, c(t) \neq i\} & \text{存在与 } i \text{ 集相邻的超顶点集} \\ i & \text{否则} \end{cases}$$

到此步为止,每个超顶点集 i 的根结点的 D 指针或指向与集 i 相邻的标号最小的超顶点集,或指向其本身(此时 i 是 G 的一个连通分支);

第三步 将由第二步连接起来的所有超顶点集扩大成一个超顶点集。关于循环的次数,由定理 11.4 给出。

定理 11.4 在无向图 G(V,E),|V| = n 中,用顶点倒塌法求其连通分支的循环过程至多须执行 $\lceil \log_2 n \rceil$ 次,就可保证每个超顶点集是 G 的一个连通分支。

证明　由顶点倒塌法可知,对于任意一个超顶点集 i,当且仅当没有其它的超顶点集与它相邻时超顶点集 i 才不能扩大,因此当超顶点集 i 不能扩大时,它就是图 G 的一个连通分支。设执行完 k 次循环后,G 中可扩大超顶点集的个数为 N(k),因此,有

$$N(k)=\begin{cases} n & k=0 \\ N(k-1)/2 & k>0 \end{cases}$$

当 $k=\log n$ 时,

$$\begin{aligned} N(\log n) &\le N(\lceil \log n \rceil - 1)/2 \\ &\le N(\lceil \log n \rceil - 2)/2^2 \\ &\cdots \\ &\le N(0)/2^{\lceil \log n \rceil} \\ &\le 1 \end{aligned}$$

因为可扩大的超顶点集的个数不会等于 1,所以 $N(\lceil \log n \rceil)=0$,即至多进行 $\lceil \log n \rceil$ 次循环后,每个超顶点集是 G 的一个连通分支。

在叙述连通分支算法之前,首先引入算法所须执行的几个子过程和函数:

(1) 子过程 RAR (a(i),b(c(i)))的功能是将未屏蔽的处理器 i 中局部变量 a(i)的值等于处理器 c(i)中的局部变量 b(c(i))的值;

(2) 子过程 RAW (a(i),b(c(i)))的功能是将未屏蔽的处理器 i 中局部变量 a(i)的值写到处理器 c(i)中的局部变量 b(c(i))中;

函数 BITS(i,j,k)的功能是返回一个数,此数由整数 i(i > 0)的二进制表示第 j 位到第 k 位组成(0 位是最低位),如

BITS (17,3,1) = 0,　BITS (10,3,2) = 2

BITS (16,4,4) = 1,　BITS (15,2,3) = 3

(3) 子过程 Reduce(D,n)的功能是,对于一串由根结点的 D 指针链接的超顶点集,将此串中每个超顶点集的所有结点的 D 指针赋一个值,此值是串中最后一个超顶点集的根的标号。

子过程具体算法如下:

```
Procedure Reduce (D,n)
for b←1 to ⌈logn⌉ do
    for each i:1≤i≤n pardo
        if BITS (D(i),logn - 1,b) = BITS (i,logn - 1,b)
          then RAR(D(i),D(D(i)))
        endif
      endfor
  repeat
end Reduce
```

下面给出在二维网孔模型上用顶点倒塌法求图的连通分支的并行算法。假定无向图 G(V,E)有 $n=2^k$ 个结点,图 G 的度数为 d,令 adj(i,j)是结点 i 的邻接表($1\le i\le n, 1\le j\le n$)。如果结点 i 有 d_i ($d_i\le d$) 条关联边,那么对于 $d_i+1\le j\le d$,有 adj (i,j)←∞,对于 $1\le j\le d$,adj (i,j)是与结点 i 相邻的某一结点标号,每个顶点的邻接表存放在对应编号处理器的局

部存储器中。其算法如下:

```
procedure CCOMA
```

输入:每个处理器 i 存放着邻接表 adj(i,j),1≤j≤d

输出:每个处理器 i 有一局部变量 D(i),它是结点 i 所在连通分支的标识,D(i) 等于 i 所在连通分支的最小标号结点(1≤i≤n)。

```
1   for each i:1≤i≤n pardo // 初始化,每个结点是个超顶点集 //
        D(i)←i
    endfor
2   for b←0 to ⌈logn⌉ - 1 do // 循环进行 logn 次顶点倒塌法 //
2.1   for each i:1≤i≤n pardo // C 指针初始化 //
      c(i)←∞
      endfor
2.2   for j←1 to d do
          // 每个结点 i 找邻接结点的最小超顶点集,并将此集的标号赋予 i 结点的 C 指针 //
            for each i:1≤i≤n pardo
            RAR(temp(i),D(adj(i)));
            if temp(i) = D(i) then temp(i)←∞ endif;
            c(i)←min{c(i),temp(i)}
          endfor
      repeat
2.3   for each i:1≤i≤n pardo // 用 D 指针将某些相邻超顶点集连成串 //
          RAW(c(i),D(D(i)));   // D(i)是根结点 //
          if D(i) = ∞   then D(i)←i endif
          if D(i) > i then RAR(D(i),D(D(i)))   endif
      endfor;
2.4   Reduce(D,n)
    repeat
end CCOMA
```

对于上述算法作如下说明:

(1) 经过执行算法的第 2.3 步中的子过程 RAW(c(i),D(D(i)))后,对于任意一个超顶点集 i,其相邻的标号最小的超顶点集 D(i)的标号 D(i)不一定小于 i,但 D(i)的相邻的标号最小的超顶点集的标号 D(d(i))一定小于或等于 i。因此,经过执行算法的第 2.3 步后,对于由 D 指针链接的两个超顶点集 i 和 D(i),有 i≥D(i)。

(2) 在算法的第 2.3 步中执行子过程 RAW(c(i),D(D(i)))时可能存在写冲突,解决冲突的策略是把最小的 c(i)值写入 D(D(i))中。在给出算法的复杂性之前,首先引进一个引理,在分析复杂性时要用到它。

引理 11.1　在 n 个处理器组成的二维网孔上,实现过程 RAR 及过程 RAW 需 O($\sqrt{n}$)时间。

证明　由 11.1 中给出的二维网孔阵列图(见图 11.4),当二维网孔由 n 个处理器组成时,最坏情况下两结点信息交换步数为 2($\sqrt{n}$ - 1),(即对顶拐角上两结点的信息交换),即为

$O(\sqrt{n})$。同样可证明对于二维网孔连接的两个变种在最坏情况下同一行上两结点信息交换步数为$\sqrt{n}-2$,所以在最坏情况下两结点信息交换的步数也为 $O(\sqrt{n})$,因此在二维网孔上实现过程 RAR 及过程 RAW 需 $O(\sqrt{n})$时间。

定理 11.5　已知无向图 G(V, E),其中$|V|=2^k=n$, G 的度数为 d,那么在 n 个处理器组成的二维网孔上求 G(V, E) 的所有连通分支的 CCOMA 算法需 O(dnlogn)时间。

证明　因为算法中第 2.2 步需 O(dn) 时间,而第 2.2 步需执行$\lceil \log n \rceil$次,故整个算法需 O(dnlogn)时间。

11.5　MIMD 共享存储模型上的并行算法

11.5.1　并行求和算法

假定有 p 个处理器,其标记为 $P_0, P_1, \cdots, P_{p-1}$,全局存储器有变量 $a_0, a_1, \cdots, a_{n-1}$,它们包含有待加的各数的值,全局变量 g 存放最终结果。算法的思路是:对于每个处理器 P_i $(0 \leq i \leq p)$,它们各自利用其局部变量 l_i 计算 $a_i + a_{i+p} + a_{i+2p} + \cdots + a_{i+k_j p}$ $(n-p \leq i+k_i p \leq n)$,然后,将求得的子和加到全局变量 g 中。显然,此算法存在存储冲突,解决的办法是当一个处理器访问全局变量 g 时对其加锁,访问完毕后立即开锁,其算法如下:

```
procedure Summation-MIMD
        g←0;
        for each Pi :0≤i < p pardo
        li←0
          for j = i to n step p do
                li←li + aj
          repeat
          lock (g)
                g←g + li
          unlock (g)
        endfor
end Summation-MIMD
```

下面分析上述算法在最坏情况下的运算时间。若不考虑生成进程所需的时间,那么算法的时间开销是

(1) 局部求和所需的时间 $\Theta(n/p)$;

(2) p 个处理器串行地对全局变量 g 进行加锁、修改其值、开锁操作所需的时间 $\Theta(p)$;

(3) 同步开销 $\Theta(p)$。

因此,上述算法在最坏情况下的运算时间为$(n/p+p)$。

11.5.2　矩阵乘法的并行算法

这里介绍的在 MIMD 共享存储模型上的矩阵求积并行算法是将串行的矩阵求积算法

并行化。为叙述方便,先给出单机上的矩阵乘法算法。假定矩阵 $A=(a_{ij})_{n\times n}$,矩阵 $B=(b_{ij})_{n\times n}$,乘积矩阵 $C=(c_{ij})_{n\times n}$。单机上矩阵乘法的算法如下:

```
procedure MM(A,B,C) //C=A*B//
    for i←1 to n do
        for j←1 to n do
            t←0;
            for k←1 to n do
                t←t + aik bkj
                repeat;
                cij←t
            repeat
    repeat
end MM
```

此算法有 3 个 for 循环可以并行化,由于小粒度算法在 MIMD 共享存储模型上的效率是很差的,故这里选择最外层循环进行并行化。其算法如下:

```
procedure MMOMSM
    输入:矩阵 A 和矩阵 B,它们都是 n 阶矩阵;
    输出:矩阵 C=A*B
1     for each l 1≤l≤p pardo //p 是处理器个数//
2     for i←l to n step p do
3         for j←1 to n do
4             to←0;
5         for k←1 do n do
6             tl←tl + aik bkj
          repeat;
7             cij←tl
              repeat
          repeat
        endfor
end MMOMSM
```

定理 11.6 在 MIMD 共享存储模型上,计算两个 n 阶矩阵乘积的 MMOMSM 算法需 $O(n^3/p+p)$时间,$O(p)$个处理器$(1\leq p\leq n)$。

证明 由算法可知,每个进程计算了矩阵 C 的 n/p 行。而计算矩阵一行需 $O(n^2)$时间,因此计算矩阵 C 需 $O(n^3/p)$时间。因为进程间同步仅一次,因此同步开销是 $O(p)$时间,所以整个并行算法需 $O(n^3/p+p)$时间和 $O(p)$个处理器$(1\leq p\leq n)$。证毕。

在 MMOMSM 算法中存在读冲突,由定理 11.6 的证明过程可看出,在估计算法运算时间时,忽略了读冲突。因此,准确地说,MMOMSM 算法是建立在 MIMD-CREW 模型上的算法。

由于 MMOMSM 算法的运算时间为 $O(n^3/p+p)$,因此可证明当处理器的个数 p 为 $O(\sqrt{n^3})$ 时算法的运算时间最小$(O(n^{3/2}))$,这时算法的加速比最大。

11.5.3　枚举分类算法

本算法的基本构思是:对于在公共存储器中的待排序数列 $S=(x_1, x_2, \cdots, x_n)$,利用 $p(1\leq p\leq n)$个处理器对它进行排序。其中 $i(1\leq i\leq p)$号处理器依次确定元素 x_i, x_i+p, x_i+2p, …,排列后的位置,然后把它们赋予公共存储器中的数组 T 中相应的分量,使得 T 是 S 的排序结果。其算法如下:

```
Procedure ENUERSORT
输入:S={x(1),…,x(n)}置于共享存储器的 x 数组中
输出:已排序序列置于共享存储器的 T 数组中
处理器数:P(n)=p (1≤p≤n)
for each i:1≤i≤p pardo
    for j=i to n step p do
        k←1
        for l=1 to n do
            if x(j)>x(l) then k←k+1
            else if x(j) = x (l) and j>l then k←k+1 endif
            endif
            t(k)←x(j)
      repeat
    repeat
    endfor
end ENUERSORT
```

由于每个处理机至多确定数组 X 中的$\lceil n/p\rceil$个元素位置,而每确定一个元素的位置需 O(n)时间,因此对 X 进行排序需$\lceil n/p\rceil$* O(n)时间。因为进程同步开销为 O(p),因此整个并行算法的时间开销 $t(n)=n/p * O(n)+O(p)=O(n^2/p)$。由于最佳串行分类算法需 O(nlogn) 时间,因此此并行算法的加速比为 O(plogn/n) 。

注意:ENUERSORT 算法存在读冲突,因此它是 MIMD-CREW 模型上开发的算法。

11.5.4　二次取中的并行选择算法

首先引进在 MIMD 共享存储模型上求 m 个数的部分和算法。算法输入为序列 $A=\{a_1, a_2, \cdots, a_m\}$,输出为 $T=\{t_1, t_2, \cdots, t_m\}$,其中 $t_i=a_1+a_2+\cdots+a_i$, $1\leq i\leq m$。处理机个数为 m。其算法如下:

```
procedure PS-MIMD (A,T,m)
    for j=0 to logm - 1 do
        for i = 2^j + 1 to m pardo
            t_i ← a_ij
            t_i ← t_i + a_{i-2^j}
    endfor
    for i= 2^j + 1 to m pardo
        a_i ← t_i
    endfor
```

repeat

end PS-MIMD

引理 11.2 PS-SIMD 算法的复杂度为 O(mlogm)。

证明 显然此算法求部分和操作需 O(logm) 时间。对于 i 的每个定值有 $m-2^j$ 个处理器并行操作,且同步二次,开销为 $O(m-2^j)$时间,因为 $\sum_{j=0}^{\log m-1}(m-2^j)=O(m\log m)$所以算法在同步上的开销为 O(mlogm),由此推出算法的时间复杂度为 O(mlogm)。

假定输入系列 $S=\{x_1,x_2,\cdots,x_n\}$,在 MIMD 共享存储器模型上利用 $N=\lceil n^{1/2}\rceil$个处理器求 S 中第 $k(1\leq k\leq n)$小元素的并行算法,其非形式化描述如下:

(1) 将 S 分成 N 段,每段至多$\lceil n^{1/2}\rceil$个元素。每段指派一个处理器,各段同时并行求出各自的中值(使用串行二次取中算法);

(2) 递归调用并行选择算法求出中值的中值 m;

(3) 以 m 为划分元,并行地对各段进行划分。记 $i(1\leq i\leq N)$段分成 S^i,U^i,和 R^i 三个子段,它们分别由段中小于、等于和大于 m 的元素组成;

(4) 并行求出 $S^i(1\leq i\leq N)$中元素个数 k_s^i,U^i 中元素个数 k_u^i 和 R^i 中元素个数 k_r^i;

(5) 利用 PS-MIMD 算法,并行求出 k_s^1 $k_s^1+k_s^2$ $\cdots$, $k_s^1+k_s^2+\cdots+k_s^N$,并分别赋予 $t_s^1,t_s^2,\cdots,t_s^N$;

(6) 利用 PS-MIMD 算法并行求出 k_u^1 $k_u^1+k_u^2,\cdots,+k_u^1+k_u^2+\cdots+k_u^N$ 并分别赋予 $t_u^1,t_u^2,\cdots,t_u^N$;

(7) 利用 PS-MIMD 算法并行求出 k_r^1 $k_r^1+k_r^2,\cdots,k_r^1+k_r^2\cdots+k_r^N$ 并分别赋予 $t_r^1,t_r^2,\cdots,t_r^N$;

(8) 根据 $S^i(1\leq i\leq N)$的末地址 t_s^i 并行地将 N 个 S^i 子段中元素写入 S 的相应位置中,记这些元素组成子段 S_1;

(9) 根据 $R^i(1\leq i\leq N)$的末地址 t_r^i 并行地将 N 个 R^i 子段中元素写入 S 的相应位置中,记这些元素组成子段 S_2;

(10) 利用 t_s^N 和 t_u^N 和 t_r^N,判断第 k 个元素是等于 m 还是在 S_1 或 S_2 中,对于后两种情况则可对 S_1 或 S_2 递归调用并行选择算法。

下面对二次取中并行选择算法进行算法分析。

引理 11.3 在二次取中并行选择算法中,大于划分元素 m 的元素个数至多为(3/4)n;小于划分元素的元素个数至多为(3/4)n。

证明 因为 $m_1,m_2,\cdots,m_N$ 中至少有 N/2 个元素大于等于 m,又因为在第 i 段中至少有[N/2]个元素大于等于 m_i;因此 S 中至少有$(N^2/4)$个元素大于等于 m。

由此推出 S 中至多有 $n-N^2/4$ 个元素小于 m,即至多有(3/4)n 个元素小于划分元素。同理可证明,S 中至多有(3/4)n 个元素大于划分元素。

定理 11.7 在 MIMD 共享存储模型上,二次取中的并行选择算法在最坏情况下需 $O(\sqrt{n}\log n)$时间。

证明 记算法的运算时间为 T(n),不难验证算法中第(1)、(2)、(3)、(4)、(8)、(9)步操作时间为 $O(\sqrt{n})$或 $T(\sqrt{n})$。由于 PS-MIMD 算法的运算时间为 O(mlogm)(其中 m 为加数

个数),因此算法中第(5)、(6)、(7)步操作时间都为 $O(\sqrt{n}\log n)$。由引理 11.3 知,算法中第(10)步的操作时间为 $T((3/4)n)$。下面计算算法的同步开销。算法中需要同步的是第(1)、(3)、(4)、(8)、(9)步,其时间开销都为 $O(\sqrt{n})$。因此,算法的运算时间 $T(n)$满足

$$T(n)\leq c\sqrt{n}\log n + T(\sqrt{n}) + T((3/4)n) \qquad (11.8)$$

下面用数学归纳法证明 $T(n)\leq 40c\sqrt{n}\log n$。

当 $n\leq 5$ 时,可选定适当大的 c 使不等式 $T(n)\leq 40c\sqrt{n}\log n$ 成立。设 $5^4\leq n\leq m-1$ 时结论成立。当 $n=m$ 时,由式(11.8)得 $T(m)\leq c\sqrt{m}\log m + T(\sqrt{m}) + T(3/4m)$。由归纳法的假设得: $T(m)\leq c\sqrt{m}\log m + \frac{40c(m)^{1/4}\log m}{2} + 40c\sqrt{\frac{3}{4}}m\log(3/4m)$。因为 $40c\sqrt{\frac{3m}{4}}\log(3/4)m < 0.87\times 40c\sqrt{m}\log m$,又可证:当 $m\geq 5^4$ 时,

$$40c(m)^{\frac{1}{4}}\log m/2\leq 0.1\times 40c\sqrt{m}\log m$$

所以, $T(m) < c\sqrt{m}\log m + 4c\sqrt{m}\log m + 34.8c\sqrt{m}\log m < 40c\sqrt{m}\log m$。这样就证明了 $T(n)\leq 40c\sqrt{n}\log n$。故 $T(n)=O(\sqrt{n}\log n)$。

因为选择问题的最佳串行算法的运算时间为 $O(n)$,所以,这里介绍的二次取中并行选择算法的加速比 $Sp(n)=O(\sqrt{n}/\log n)$,同样可求出此并行算法的成本 $c(n)=T(n)P(n)=O(n\log n)$,显然此算法不是成本最佳的算法。

11.5.5　并行快速排序算法

在 MIMD 共享存储模型上,并行快速排序算法的基本思想是将串行快速排序算法并行化。在 4.5 节中介绍的串行快速排序算法即 QUICKSORT 算法,此算法在最坏情况下的运算时间为 $O(n^2)$,这一性能指标不够理想。这里对此算法作如下改进:将以第一个元素为划分元改成利用选择算法选取序列 S 中第 $\lceil n/2\rceil$ 小元素 m,以此元素作为划分元,将序列 S 分为 S_1, S_2 和 S_3 三个集合,其中 $S_1=\{x_i \mid x_i < m, x_i\in S\}$, $S_2=\{x_i \mid x_i = m, x_i\in S\}$, $S_3=\{x_i \mid x_i > m, x_i\in S\}$,然后递归调用原算法对 S_1 和 S_3 中元素进行排序。不难证明改进后的快速分类算法在最坏、最好和平均情况下的运算时间都是 $O(n\log n)$。

假定处理器个数 $N=\lceil\sqrt{n}\rceil$,在对改进的串行快速排序并行化时,可利用前面介绍的二次取中并行选择算法确定划分元 m。在对二个子排序问题的递归调用上有两种方案。第一种方案是二个递归调用并行执行。由于此方案在最坏情况下所需的处理器个数 $P(n)=\lceil\sqrt{n/2}\rceil+\lceil\sqrt{n/2}\rceil>\lceil\sqrt{n}\rceil$,所以不可行。第二种方案是二个递归调用串行执行。由于每次递归调用只需要一半的处理器进行工作,使得大量的处理器无事可做,从而直接影响了算法的并行度,所以第二种方案也不理想。为了提高算法的并行度,可按下述方式解决:3 次调用二次取中并行选择算法。在 S 中选择 3 个元素 m_1, m_2 和 m_3,它们分别是序列 S 中第 $\lceil n/4\rceil$ 小、第 $\lceil n/2\rceil$ 小和第 $\lceil 3n/4\rceil$ 小元素。以它们为划分元素,将 S 分成 S_1、S_2、S_3、S_4、U_1、U_2 和 U_3 共 7 个序列,其中, $S_1=\{x_i \mid x_i < m_1, x_i\in S\}$; $S_2=\{x_i \mid m_1 < x_i < m_2, x_i\in S\}$; $S_3=\{x_i \mid m_2 < x_i < m_3, x_i\in S\}$; $S_4=\{x_i \mid x_i > m_3, x_i\in S\}$ $U_1=\{x_i \mid x_i = m_1, x_i\in S\}$; $U_2=\{x_i \mid x_i = m_2, x_i\in S\}$; $U_3=\{x_i \mid x_i = m_3, x_i\in S\}$。然后并行地递归调用原算法对 S_1 和 S_2 同时进行排序,接

着并行地递归调用原算法对 S_3 和 S_4 同时进行排序,从而完成对序列 S 的排序。由于此方式在最坏情况下所需的处理器个数 $P(n)=\max\{\lceil\sqrt{n}\rceil, 2\lceil\sqrt{n/4}\rceil\}=\lceil\sqrt{n}\rceil$,所以是可行的。按照上述方式,在 MIMD 共享存储模型上的并行快速排序算法描述如下:

procedure PQS (S)

输入:$S=\{x_1, x_2, \cdots, x_n\}$, $|S|=n$

输出:非降有序序列

处理器数:$N=\lceil\sqrt{n}\rceil$

(1) 若 $|S|<3$,则调用串行排序算法对 S 进行排序,算法结束。

(2) 调用二次取中并行选择算法,确定 S 中第 $\lceil n/4\rceil$ 小元素 m_1。

(3) 调用二次取中并行选择算法,确定 S 中第 $\lceil n/2\rceil$ 小元素 m_2。

(4) 调用二次取中并行选择算法,确定 S 中第 $\lceil 3n/4\rceil$ 小元素 m_3。

(5) 将 S 分成 N 段,每段指定一个处理器,以 m_1, m_2 和 m_3 为划分元素,并行地对各段进行划分。记第 $i(1\leq i\leq N)$ 段分成 S_1^i、S_2^i、S_3^i、S_4^i、U_1^i、U_2^i 和 U_3^i 7 个子段,其中

$S_1^i=\{x_i \mid x_i<m_1, x_i$ 属于第 i 段$\}$; $S_2^i=\{x_i \mid m_1<x_i<m_2, x_i$ 属于第 i 段$\}$;

$S_3^i=\{x_i \mid m_2<x_i<m_3, x_i$ 属于第 i 段$\}$; $S_4^i=\{x_i \mid x_i>m_3, x_i$ 属于第 i 段$\}$;

$U_1^i=\{x_i \mid x_i=m_1, x_i$ 属于第 i 段$\}$; $U_2^i=\{x_i \mid x_i=m_2, x_i$ 属于第 i 段$\}$;

$U_3^i=\{x_i \mid x_i=m_3, x_i$ 属于第 i 段$\}$。

(6) 并行计算出每段中 7 个子段的元素个数 $|S_1^i|$、$|S_2^i|$、$|S_3^i|$、$|S_4^i|$、$|U_1^i|$、$|U_2^i|$ 和 $|U_3^i|$, $1\leq i\leq N$。

(7) 利用 PS-MIMD 算法并行求出 $|S_1^1|$、$|S_1^1|+|S_1^2|, \cdots, |S_1^1|+|S_1^2|+\cdots+|S_1^N|$。

(8) 类似于(7),分别求出 $|S_2^i|$、$|S_3^i|$、$|S_4^i|$、$|U_1^i|$、$|U_2^i|$ 和 $|U_3^i|$, $1\leq i\leq N$ 的部分和。

(9) 根据(7)的计算结果,并行地将 N 个 S_1^i $(1\leq i\leq N)$ 中元素写入 S 相应的位置中,记这些元素组成子段 S_1。

(10) 类似于(9),根据(8)的计算结果,分别把 U_1^i、S_2^i、U_2^i、S_3^i、U_3^i 和 S_4^i 中元素并行写入 S 的相应位置中,记这些元素分别组成子段 U_1、S_2、U_2、S_3、U_3 和 S_4。

(11) 对子段 S_1 和 S_2 并行调用 PQS 算法。

(12) 对子段 S_3 和 S_4 并行调用 PQS 算法。

end PQS

记 PQS 算法的运行时间为 $T(n)$,不难证明当 $|S|\geq 3$ 时 $T(n)$ 满足关系式:

$$T(n)\leq c\sqrt{n}\log n+2T(1/4n) \qquad (11.10)$$

由此关系式可推出下面结论。

定理 11.8 在 MIMD 共享存储模型上,当输入规模为 n,处理器个数 $N=\lceil\sqrt{n}\rceil$ 时,PQS 算法的运算时间为 $O(\sqrt{n}\log^2 n)$。

证明 记算法的运算时间为 $T(n)$,因此,当 $n<3$ 时定理显然成立。当 $n\geq 3$ 时,不妨设 n 为 4 的幂,设 $n=4^k$,因此

$$\begin{aligned}T(n)&\leq c\sqrt{n}\log n+2T(1/4n)\\&\leq c\sqrt{n}\log n+c\sqrt{n}\log(n/4)+2^2T(1/4^2n)\\&\cdots\cdots\\&\leq c\sqrt{n}(\log n+\log(n/4)+\cdots+\log 4)+2^{\log_4 n}T(1)\end{aligned}$$

$$= 2c\sqrt{n}(k + (k - 1) + \cdots + 1) + \sqrt{n}T(1)$$

$$= O(\sqrt{n}\log n)$$

证毕。

11.5.6　求最小元的并行算法

这里将在 MIMD 共享存储模型上利用 p 个处理器建立求序列 $I = \{A(1), A(2), \cdots, A(n)\}$的最小元的并行算法。算法的基本思想是，当 $n > p$ 时，将序列中的元素分成元素个数基本相同的 p 个组，从处理器 P_i $(1 \leq i \leq p)$求出第 i 组中的最小元素，这样把求 n 个元素的最小问题转化为求 p 个元素的最小问题。当 $n \leq p$ 时，进行 logn 次迭代，每次迭代时，将元素按中心点对称的方法两两分组，求出每组的最小元，使得经过一次迭代后元素个数减少一半，最终求出最小元素。其算法如下：

```
procedure MIMDMIN(A(1), A(2), ···, A(n), min)
    输入: I = {A(1), A(2), ···, A(n)}
    输出: I 中最小元 min
处理器个数: p
if n > p then
    for each Pi: 1≤i≤p pardo
        for j = i + p to n step p do
            if A(j) < A(i) then
            A(i)←A(j)
        endif
    repeat
  endfor
  k←p
else
  k←n
endif
while k > 1 do
    for each Pi: 1≤i≤⌈k/2⌉ pardo
        if A(k - i + 1) < A(i)
            then A(i)←A(k - i + 1)
        endif
    endfor
    k←⌈k/2⌉
  repeat
  min←A(1)
end MIMDMIN
```

下面分析 MIMDMIN 算法的时间复杂度和成本。算法由 if 语句和 while 语句二部分组成，对于第一部分，其操作时间为 $O(n/p)$。对于第二部分，其操作时间为 $O(\log p)$第一部分中仅出现一次同步，因此同步开销为 $O(p)$。在第二部分中出现 logp 次同步，同步开销为

$O(p/2 + p/4 + \cdots + 1) = O(p)$。因此算法的同步总开销为 $O(p)$。由此推出 MIMDMIN 算法的时间复杂度为 $O(p + n/p)$。由于算法中处理机个数为 p,因此算法的成本为 $O(p^2 + n)$。

由上面的分析可以看出,此算法的性能与处理器的个数 p 密切相关。求最小元的最佳串行算法的运算时间为 O(n),在 MIMDMIN 算法中,当处理器个数等于 $\sqrt{n}$ 时,算法的成本为 O(n),即等于最佳串行算法的运算时间,因此,此时的并行算法是最佳的。当处理器个数 p = n 时,并行算法的运算时间为 O(n),即等于串行算法的运算时间,显然此时算法的性能是极差的。

11.5.7 求单源点最短路径的并行算法

5.6 节讨论了求单源点最短路径的串行算法,即 SHORTEST-PATHS 算法,下面将在 MIMD 共享存储模型上把 SHORTEST-PATHS 算法并行化,产生在该模型上的求单源点最短路径的并行算法。

从直观上看,5.6 节中的 SHORTEST-PATHS 算法有 4 处可并行化,它们分别是算法的第 1 到第 3 步的 for 循环,第 5 到第 11 步的 for 循环,第 6 步的求最小值和第 8 到第 10 步的 for 循环。SHORTEST-PATHS 算法的设计策略是贪心方法,其最优量度标准是按路径长度的非减次序产生最短路径。如果对算法中第 5 到第 11 步的 for 循环进行并行化,势必破坏最优量度标准,因此,在 SHORTEST-PATHS 算法的基础上不能对此循环并行化,实现并行求出源点到所有 n - 1 个结点上的最短路径。这样,并行化工作只能在其它 3 处进行。下面给出经过并行化后的求单源点最短路径算法,其中处理器个数为 p,其算法如下:

```
procedure MSMSP
    输入:图 G(V,E)的成本邻接矩阵 cost (n,n)
    输出:从源点 v 到所有顶点 v_i 的最短路径长度 dist(i),i∈v
1   for each P_i :1≤i≤p pardo   //初始化//
        for j←i to n step p do
            S(j)←0_1 ;
            dist(j)←cost (v,j)
        repeat
    endfor
2   S(v)←1;dist(v)←0   //结点 v 计入 S//
3   for num←2 to n - 1 do   //确定由源点 v 出发的 n - 1 条路径长度//
    3.1  利用 MIMDMIN 算法,选取结点 u,它使得 dist(u) = min_{s(w)=0} {dist (w)}
    3.2  S(u)←1   //结点 u 计入 S//
    3.3  for each i: 1≤i≤p pardo   //修改 S 值为 O 的点的 dist 值//
                for j←i to n step p do
                    if S(j) = 0
                then dist(j)←min (dist(j),dist(u) + cost(u,j))
                        endif
                        repeat
    endfor
```

```
        repeat
    end MSMSP
```

在 MSMSP 算法的第 3 .1 步中,因为是求达到最小值的点,所以不能直接利用前面介绍的 MIMDMIN 算法,必须将该算法进行适当的修改,可以证明 ,修改后的 MIMDMIN 算法的时间复杂度保持不变。算法的修改以及修改后算法的时间复杂度分析留作习题。

定理 11 .9　在 MIMD 共享存储模型上,求单源点最短路径的 MSMSP 算法需 $O(np + n^2/p)$时间。

证明　显然算法中的第 1 步并行操作的时间开销为 $O(n/p)$。在第 1 步中,p 个处理器只需同步一次,因此第 1 步中同步开销为 O(p)。在第 3 步的循环体内,由前面 MIMDMIN 算法的时间复杂度分析可知,第 3 .1 步的运算时间为 $O(p + (n - num + 1)/p)$,第 3 .3 步操作需 $O(n/p)$时间,由于在第 3 .3 步中 p 个处理器同步一次,所以其同步开销为 O(p)。由此推出在第 3 步中每执行一次循环需 $O(p + n/p)$时间,因此,第 3 步的运算时间为 $O(np + n^2/p)$,这样,整个算法的运算时间为

$$O(p) + O(n/p) + O(np + n^2/p) = O(np + n^2/p)$$

证毕。

由 MSMSP 并行算法可以看出,在对串行 SHORTEST-PATHS 算法并行化时,采用的是小粒度并行,即在同步之间执行很少的运算。由于在 MIMD 模型上同步要通过很费时间的软件来实现,所以这种小粒度的并行算法在 MIMD 机器上的运行效率肯定是不佳的。如果对 SHORTEST-PATHS 算法中第 5 到第 11 步的 for 循环进行并行化可以加大算法的粒度,由前面的分析可知,SHORTEST-PATHS 算法的设计基础使得这项工作难以实现。因此,要想通过将串行算法并行化的方法得到大粒度的并行算法,必须设计出一个易于大粒度并行化的串行算法,然后对新的串行算法并行化。Moore[10] 对求单源点最短路径问题设计了一个新的串行算法,Deo 基于 MIMD 共享存储模型,实现了该算法的并行化,得到的新的并行算法的粒度比 MSMSP 算法的粒度要大得多。新的并行算法的设计与分析,可参阅文献[11]。

11 .6　MIMD 异步通信模型上的并行算法

MIMD 异步通信计算模型可以抽象为一个无向图 G(V,E),其中顶点集 V 对应处理机集合,边集 E 对应处理机间的双向通信链集合。处理机之间不存在共享存储器。处理机之间的通信是通过发送和接收信息完成的,每个处理机只能和邻接处理机(G 中邻接顶点) 进行通信。在设计和分析算法时,假定在一条通信线的同一方向上的消息到达目标的次序服从“先进先出”的次序,并假定处理机发送和接收消息操作所需的时间同处理机之间通信时间比较起来可以忽略不计。在此模型上设计的并行算法的衡量标准是以算法的通信复杂度和通信所花费的时间为主。

在 MIMD 异步通信模型上,由于每个处理机均是一台完整的计算机,它们都能在各自的指令流控制下对各自的数据流进行操作,因此,在模型上编程较为灵活,能更好地挖掘开发求解问题所固有的并行性。但是必须看到,在该模型上开发并行算法一般会涉及到进程

的通信、同步等待问题,这就使得在该模型上的算法设计和算法分析工作比在 SIMD 模型上复杂得多。

11.6.1 选择问题的并行算法

第 4 章介绍了求一数组中第 k 小元素的串行算法,即 SELECT 算法,假定数组 $A = [a_1, a_2, \cdots, a_n]$,则可将此算法非形式化描述如下:

procedure SELECT (A, n, k)

(1) 如果 $|A| = 1$,则返回此元素作为结果,否则执行以下各步。

(2) 随机地从 A 中挑选一个元素 m 作为划分元素。

(3) 将 A 划分成 AL, AE 和 AG 三个子集,它们分别包含小于、等于或大于 m 的那些元素。令

$$A' = \begin{cases} AL & k \leq |AL| \\ AE & |AL| < k \leq |AL| + |AG| \\ AG & k > |AL| + |AE| \end{cases}$$

如果 A = AE,则返回元素 m,否则按下式计算 k′值:

$$k' = \begin{cases} k & \text{若 } A' = AL \\ k - |AL| - |AE| & \text{若 } A' = AG \end{cases}$$

(4) 递归调用本算法,以求出 A′中第 k′小元素。

end SELECT

下面把上述串行算法并行化,产生在 MIMD-CL 模型上选择问题的并行算法。

首先假定处理机个数 $p = 2^t - 1$,处理机按树形结构连接。在执行算法之前,数组 A 中 n 个元素已平均分配到 p 个处理机的局部内存中,在处理机 $P_i\{1 \leq i \leq p\}$ 的局部存储器中,存储单元 m_i 存放局部存储器所存放的数组 A 中元素的个数。并行算法的非形式化描述如下:

procedure MIMD-CL-PS

(1) 根结点通知其余节点将其所保存的元素个数送往根;其余节点向根发送各自所保存的元素个数;根结点计算 $|A| = \sum_{i=1}^{p} |m_i|$,如果 $|A| = 1$,根结点通知该元素所在结点将此元素送往根结点,算法结束;否则就执行以下各步。

(2) 随机地从 |A| 个元素中选出一个元素 m 作为划分元素并送往根结点。其过程是

① 根结点在区间 [1, |A|] 中随机选择一整数 j。

② 按先根周游次序循环地进行以下操作:结点 i 判定 $m_i - j$ 是否大于等于 0,若 $m_i - j \geq 0$,则结点 i 将其保存的第 j 个元素送往根作为划分元素;否则 j: = j - m_i 并将 j 发送给下一个结点。

③ 根结点将划分元素 m 发送给其它所有结点。

(3) 每个结点 $i(1 \leq i \leq p)$ 将其局部存储器中的元素按 m 划分成 AL_i, AE_i 和 AG_i 三个子集合,它们分别包含小于、等于和大于 m 的那些元素,并将 |AL|, |AE| 和 |AG| 发送给根结点。

(4) 根结点计算出:

$$|AL| = \sum_{i=1}^{p} |AL| \qquad |AE| = \sum_{i=1}^{p} |AE_i| \qquad |AG| = \sum_{i=1}^{p} |AG_i|$$

若 $|AL| < k \leq |AL| + |AE|$,则划分元素 m 就是第 k 小元素,算法结束;

若 $k \leq |AL|$,则根结点通知每个结点 $i(1 \leq i \leq p)$ 保存集合 AL_i 中元素,并且 $m_i \leftarrow |AL_i|$;

若 $k > |AL| + |AE|$,则 $k \leftarrow k - |AL| - |AE|$,并且根结点通知每个结点 i 保存集合 AG_i 中元素,并且 $m_i \leftarrow |AG_i|$。

(5) 递归调用本算法。

end MTMD-CL-PS

定理 11.10　在串行 SELECT 算法中,至多进行了 O(n) 次递归调用,平均递归调用 O(logn)次。

证明　在最坏情况下,每进行一次递归调用元素个数减少一个,因此,在 SELECT 算法中至多递归调用 O(n)次。设在 SELECT 算法中平均调用 T(n)次。因为经过一次划分选择后,下一次递归调用所涉及的元素个数可能为 0,1,2,…,n - 1 个,并且各种情况出现的概率相同,所以,平均涉及(n - 1)/ 2 个元素。由此建立 T(n)的递推关系式:

$$T(n)=\begin{cases}1+T((n-1)/2) & n>1\\ 1 & n=1\end{cases}$$

由此不难推出 T(n) = O(logn)。证毕。

并行 MIMD-CL-PS 算法是利用并行划分来加快划分速度的,而每次划分的结果与串行 SELECT 算法的划分结果是一致的。因此,MIMD-CL-PS 算法至多递归调用 O(n) 次,平均递归调用 O(logn)次,由此可推出下面结论:

定理 11.11　MIMD-CL-PS 算法的通信复杂度在最坏情况下为 O(np),在平均情况下为O(plogn)。

证明　在算法的第(1)步骤中,非根结点向根结点发送各自所保存的元素个数的过程是:首先是由叶子结点向其父结点发送,当一个结点 i 收到其左、右儿子发送来的值后将此二值与 m_i 相加然后再发送给它的父结点。因此第(1)步骤的信息交换数为 O(p)。类似地可推出第(2)、第(3)、第(4)步骤的信息交换数都为 O(p)。由于 MIMD-CL-PS 算法在最坏情况下要运行 O(n)遍,在平均情况下要运行 O(logn)遍,因此算法的通信复杂度在最坏情况下为 $O(p_n)$,在平均情况下为O(plogn)。证毕。

定理 11.12　在最坏情况下 MIMD-CL-PS 算法的运算时间为 $O(n^2/p+n\log p)$。

证明　不难验证,算法的最坏情况是:每递归调用一次,数组 A 中元素个数仅减少一个,并且每连续 n/ p 次递归调用后,仅使一个处理机保存的元素个数为零。因此,在最坏情况下算法要运行 O(n)遍,在每一遍中划分操作时间为O(n/ p)。由此推出整个算法的操作时间为$O(n^2/p)$。因为二元树的深度为 O(logp),所以在最坏情况下算法的运算时间为 $O(n^2/p+n\log p)$。证毕。

11.6.2　求极值问题的并行算法

令 $A=\{a_1,a_2,\cdots,a_n\}$,求集合 A 中元素的极值分为求极大值和最小值。不失一般性,仅考虑求极大值。在 MIMD 异步通信模型上求极值的并行算法与网络的拓扑结构密切相关,这里首先介绍在树形网络结构上的求极大值并行算法。假定有 p 个处理器,它们以二叉树的方式彼此相连,处理器 P_1 为树的根。集合 A 中 n 个元素被分成大小大致相等的 p 个集合:$A_1,A_2,\cdots,A_p$ 并假定子集合 $A_i(1\leq i\leq p)$ 中的元素已存入处理器 P_i 中的局部存储器中。算法的基本思想是,首先根结点 P_1 通知其余结点对各自局部存储器中的元素求极大值,然后根结点 P_1 求出 A_1 中的极大值。当其余结点求出它们各自保存的元素的极大值后,由叶子结点开始向父结点送其极值,当父结点收到左、右儿子的极值后与其本身极值进行比较,取最大者为它的极值,并将此值发送给它们的父结点,当根结点收到左、右儿子的极值

后,通过比较,就产生了集合 A 中的极值。其算法如下:

```
procedure TCMAX
    根结点 P1 的算法:
      send(m) message to L(P1);        //向其左儿子发送消息 m//
      send(m) message to R(P1);        //向其右儿子发送消息 m//
      call Max (A1 ,t(1));             //求集合 A1 中的极大值并赋予 t(1)//
      upon receiving (M) from son do:
          if M > t(1) then t(1) ←M
          endif
      upon receiving (M) from son do:
      if M > t(1) then t(1) ←M
      endif
    非根结点 Pi 的算法:
      upon receiving (m) from fathen do:
      if Pi不是叶子
        then send (m) message to L(Pi);
          send (m) message to R(Pi);
      endif
      Call Max (Ai , t(i));
      if Pi是叶子 then send (M) message to farther;   //向其父发极大值信息 M( = t(i))//
          else upon receiving (M) from son do:
              if M > t(i) then t(i)←M endif;
              upon receiving (M) from son do;
              if M > t(i) then t(i)←M endif;
          send (M) message to father      //向其父发极大值信息 M( = t(i))//
    endif;
    TCMAX
    end
```

对于 TCMAX 算法,必须假定处理器的个数为 2 的幂减 1 ,因为,此假定保证了树中每个非叶子结点都有两个儿子。当处理器个数不满足上述假定时,须对算法作适当的修改。如何修改留作习题。

下面对 TCMAX 算法进行复杂性分析。

定理 11 .12 在具有 p 个处理机器的树形网络结构上,对于集合 $A = \{a_1, a_2, \cdots, a_n\}$,求其元素极大值的 TCMAX 算法的通信复杂性为 O(p),时间复杂性为 $O(n/p + \log p)$。

证明 在整个算法中,结点间进行通信的信息有两类,即通知儿子进行求极大值操作的信息 m 和传送子树中元素极大值的信息 M。这两类信息的信息交换次数都等于树的边数,因此算法的通信复杂性为 O(p)。

对于每个处理器,它们求出各自局部存储器中元素极大值的操作时间为 $O(n/p)$。因为通信树的深度为 $O(\log p)$,因此不难推出通信时间为 $O(\log p)$,最后得到算法的时间复杂性为$O(n/p) + O(\log p) = O(n/p + \log p)$。

下面介绍在环形网络拓扑结构中求极大值的并行算法。假定在 MIMD-CL 计算模型

中，p 个处理器松散耦合成一个环，集合 $A = \{a_1, a_2, \cdots, a_n\}$ 中的元素已分散存放在 p 个处理器的局部存储器中，各个处理器存储的元素个数大致相等，每个处理器只能和其近邻交换信息，没有中心控制器存在。

在环形结构中求集合 A 中元素的极大值算法可分为两大类：在第一类算法中，每个处理器可以向其左、右近邻发送和接收信息，在第二类算法中，每个处理器只能向其左近邻发送信息和从其右近邻接收信息。这里，我们仅介绍第二类算法。

下面给出的 RLMAX 算法是基于 Chang 和 Roberts[12] 提出的一种改进的单向算法的设计思想，但必须增加一个限制条件，限定集合 A 中无相同元素。算法的非形式化描述如下。

procedure RLMAX

(1) 每个结点 P_i $(1 \leq i \leq p)$ 求出其局部存储器中所保存的元素的极大值 M_i。

(2) 每个结点 P_i 将 M_i 传给其左邻结点。

(3) 每个结点 P_i 收到来自其右邻结点的信息后进行下列操作：

① 当 M_i 小于收到的信息时，P_i 将收到的信息传给其左邻结点；

② 当 M_i 大于收到的信息时，则将收到的信息丢弃；

③ 当 M_i 等于收到的信息时，则 M_i 就是集合 A 中元素的极大值。

end RLMAX

根据环形网的结构性质、单向传递的约定和对集合 A 的限制条件，任何信息在返回到产生它的结点之前，必须经过其它所有结点，因此，当且仅当 A 中元素的最大值作为信息时，此信息才能返回到产生它的结点。由此可推出算法是正确的。

关于 RLMAX 算法的时间复杂性和通信复杂性，有如下结论：

定理 11.13　RLMAX 算法的运算时间为 $O(p + n/p)$。

证明　因为此算法的运算时间由通信时间和每个处理器求部分元素的极大值操作时间所决定，又由于 p 个处理器同时启动，所以通信时间是 A 中极大值通过环网一周的时间，即为 $O(p)$，由于求部分元素极大值操作的时间为 $O(n/p)$，所以整个算法的运算时间为 $O(p + n/p)$。证毕。

定理 11.14　RLMAX 算法的通信复杂性在最好情况下为 $O(p)$，在最坏情况下为 $O(p^2)$，在平均情况下为 $O(p\log p)$。

证明　因为最好情况是，除了极大值外，每个信息只传递一次，即沿传递方向各结点的信息值按由小到大的次序排列，所以总的信息传递数为：$p + p - 1 = 2p - 1$，即为 $O(p)$。

最坏情况是沿传递方向各结点的信息值按由大到小的次序排列。不妨设 $M_1 > M_2 > \cdots > M_p$，因此信息 M_i 传递的次数为 $p - i + 1$，所以总的信息传递次数为 $\sum_{i=1}^{p}(p - i + 1) = O(p^2)$。

最后求在平均情况下信息传递次数。令 $P(g_i, k)$ 为第 i 小信息 g_i 须传递 k 次的概率，因此它就是沿传递方向有连续 k - 1 个近邻的部分极大值小于 g_i，同时 g_i 的第 k 个近邻的部分极大值大于 g_i 的概率。因为部分极大值小于 g_i 的结点数为 i - 1，大于 g_i 的节结点数为 p - i，所以

$$P(g_i, k) = \frac{\begin{pmatrix} i-1 \\ k-1 \end{pmatrix}}{\begin{pmatrix} p-1 \\ k-1 \end{pmatrix}} \cdot \frac{p-i}{p-k}$$

因为信息为极大值时,它传递 p 次,所以只考虑 p - 1 个信息。又由于它们每一个至多传递 p - 1 次,所以第 i 小信息 g_i 被传递的次数的数学期望为

$$E_{gi} = \sum_{k=1}^{p-1} k \cdot P(g_i, k)$$

对于所有的信息,其期望的传递数为

$$E = p + \sum_{i=1}^{p-1}\sum_{k=1}^{p-1} k \cdot P(g_i, k)$$

$$= p(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{p})$$

其中,调和级数的部分和为 c + logp,所以信息平均传递数为 O(plogp)。证毕。

如果去掉对集合 A 的限制条件,显然 RLMAX 算法是不正确的。对于集合 A 中存在相同元素的情况,如何求 A 中元素的极大值,则留作习题。

11.6.3 网络生成树的并行算法

对于 MIMD 异步通信计算模型,其通信网络可以看作是一个无向连通图。在此模型中,从某个结点把一个消息广播到整个网络中,有两种常见的方法:一种称之为洪水淹没法,这种方法的通信开销非常大;另一种是基于网络的一棵生成树进行广播。显然,它的信息交换次数等于网络中结点个数减去 1。不难证明,这是网络中进行广播的最小通信开销,因此,该方法是网络中进行信息传递的好方法。利用网络的生成树进行广播,必须首先在网络中建立一棵网络生成树。由此可见,建立网络生成树的问题是 MIMD 异步通信计算模型中的一个极其重要而又非常基本的问题。

求网络生成树的并行算法的基本思想是:首先在网络中任意选定一结点 s 作为树的根,然后 s 向它的邻接结点发送访问消息“V”。当一个结点 m 首次收到消息“V”后,把发送者当作自己的父结点 F(m);若一个已访问过的结点收到消息“V”后,就给发送者回送一个应答消息“A”。同时,若 m 除父结点外还有其它邻接结点,则向这些邻接结点发送消息“V”。若 m 没有其它的邻接结点,则向父结点回送应答消息“R”。当根结点收到它所有邻接结点的应答消息后,向所有儿子结点发送消息“T”并终止其算法的执行。一个结点收到消息“T”后,若其是非叶子结点,则向儿子结点发送消息“T”并终止其算法;若是叶子结点,则终止其算法。

算法的非形式化描述如下:

procedure NST

根结点 s 的算法:

(1) 向它的所有邻接结点发送消息“V”;

(2) 当收到消息“V”后,向发送者发应答消息“A”;

(3) 当收到来自某一邻接结点 m 的消息“R”后,则把结点 m 当其子结点即:$S(s) \cup \{m\} \rightarrow S(s)$;
// S(s)是结点 s 的一个信息域,用以保存其子结点名//

(4) 当收到全部邻接结点回送的消息“A”或消息“R”后,则向全部邻接结点发送消息“T”并且结束算法。

非根结点 t 的算法:

(1) 当收到某一邻接结点 m 发送的消息“V”后,则

① 若结点 t 首次收到消息“V”,则认结点 m 为其父结点,并且在结点 t 的邻接表中去掉结点 m。然后判定邻接表是否为空,若为空则向父结点 m 发消息“R”;若不为空,则向表中结点发消息“V”;

② 若结点 t 不是首次收到消息“V”,则向结点 m 发消息“A”。

(2) 当收到某一邻接结点 m 发送的消息“R”后,则

① 认 m 为其子结点;

② 若收到了它的全部邻接结点(除父结点外)发送的消息“A”或消息“R”,则向父结点发送消息“R”。

(3) 当收到某一邻接结点 m 发送的消息“A”后,结点 t 检查它的所有邻接结点(除父结点外)是否都已向 t 发送了消息“A”或“R”,若是,则向其父结点发送消息“R”。

(4) 当收到某一邻接结点 m 发送的消息“T”后,则结点 t 向其子结点发送消息“T”然后结束其算法。

end NST

算法正确性的简要说明:证明 NST 算法的正确性,关键是要验证下面 4 点:

(1) 对于除根结点 s 外,所有结点有且仅有唯一父结点;

(2) 算法能判定出叶子结点;

(3) 每个结点上的算法都能结束;

(4) 算法的认子结点过程正确。

对于第(1)点,由于消息“V”从结点 s 开始,传遍所有结点,并且任一结点(不等于)只有当它首次收到消息“V”后才认父,所以除结点 s 外,每个结点有且仅有唯一父结点。对于第(2)点,算法是根据一结点 t 能否收到其全部邻接结点(除父结点外) 发送的“A”信息来判别结点 t 是否是叶子,不难验证,这一条件正是判定一结点是否是叶子结点的充要条件,因此,算法能正确判定出叶子结点。对于第(3)点,任意结点 t 上的算法是否结束是根据以 t 为根的子树是否已完全形成来判定的。在算法中,一旦判定一结点是叶子,就可结束此结点上的算法,当一结点 t 为根的子树形成后,就向其父发送消息“R”,因此,任意一个结点,当收到它的全部邻接结点(除父结点外)发送来的消息“R”或消息“A”后,说明此结点为根的子树已完全形成。这里,消息“R”是由底向上,一级级传递的,当根结点 s 判定出树已形成后,再由顶向下发送消息“T”,通知各个结点结束其算法,因此每个结点上的算法都能结束; 对于第(4)点,算法是根据消息“R”来认儿子的,由第(3) 点的验证过程,不难看出算法的认儿过程是正确的。

关于 NST 算法的通信复杂性和时间复杂性,有如下结论。

定理 11.15　NST 算法的通信复杂性为 O(m),其中 m 为网络中通信链的条数。

证明　对于网络中每条通信链,不难验证消息“V”至多通过二次;消息“R”至多通过一次;消息“A”至多通过二次;消息“T”至多通过一次。因此,整个算法至多发送 6m 条消息,即算法的通信复杂性为 O(m)。证毕。

定理 11.16　NST 算法的时间复杂性为 O(n),其中 n 为网络中结点的个数。

证明　由算法本身可以看出,其操作时间由发送消息“V”,“R”,“A”和“T”的时间和判别邻接表中所有结点(除父结点外)是否都发送来消息“R”和“A”的时间开销所决定。因为网络中每个结点的度数有限,因此算法的操作时间为 O(1)。设从起点到最远距离的结点间的长度为 l,则算法的通信时间为 O(l),因为在最坏情况下 l = n,因此算法的时间复杂度为 O(n)。证毕。

习 题 十 一

11.1 当处理机个数 N 不为 2 的幂时须对 BROADCAST 算法进行修改,证明修改后算法的运算时间和性能保持不变。

11.2 用形式化描述方式设计在 SIMD-CREW 模型上的并行归并分类算法。

11.3 分析在 SIMD-EREW 模型上的并行归并分类算法的运算时间及算法性能。

11.4 在 SIMD 共享存储模型上将串行插入分类算法并行化,并进行算法分析。

11.5 指出在 Connected-Components 算法中哪些语句存在读冲突。

11.6 在 SIMD-CREW 模型上设计连通图的宽度优先并行搜索算法,并进行算法分析。

11.7 在 SIMD 一维线性模型上设计几个数的并行求和算法,并进行算法分析。

11.8 在 SIMD 超立方模型上设计并行求极值算法,并分析算法的运算时间和性能。

11.9 设 A,B 分别是 $m\times k$ 和 $k\times n$ 矩阵,在 MIMD 共享存储模型上设计求矩阵 $C=A\times B$ 的并行算法,并对算法进行算法分析。

11.10 对于 S={18,35,21,24,29,13,33,17,31,27,15,28,11,22,19,25,34,32,16,12,23,30,26,14,20},假定 N=5,k=6,请用图表示二次取中并行选择算法的执行过程。

11.11 在 PQS 算法中,证明当 $n\geq 3$ 时,其运算时间 T(n)满足:

$$T(n)\leq C\sqrt{n}\log n+2T\left(\frac{n}{4}\right)$$

其中 n 是元素个数,C 是一个常数。

11.12 对于序列 S={20,15,24,11,17,22,13,19,16,25,12,21,26,18,23,14},模拟 PQS 算法的执行过程。

11.13 修改 TCMAX 算法,使之适合于一般的树形网络结构模型,并对修改后的算法进行性能分析。

11.14 当集合 A 中容许有相同元素时,编写在环形结构中求集合 A 中元素极大值的单向传递算法,并分析算法的时间复杂性和通信复杂性。

参 考 文 献

1 E Horowitz,S Sahni .Fundamentals of Computer Algorithms .New York: Computer Science Press,1978

2 D E Knuth .The Art of Computer Programming,Vo13 . London: Addison-Wesley Publishing Company, 1973

3 S E Goodman . S .T .Hedetniemi .Introduction to The Design and Analysis of Algorithms . New York: McGraw-Hill Publishing Company 1977

4 A .V . Aho, J E Hopcroft, J D Ullman .The Design and Analysis of Computer Algorithms .London: Addison-Wesley Publishing Company,1974

5 游兆永 线性代数与多项式的快速算法 .上海:上海科学技术出版社,1980

6 洪帆 .离散数学基础 武汉:华中工学院出版社,1983

7 马仲蕃,魏权龄,赖炎连 数学规划讲义 .北京:中国人民大学出版社,1981

8 E . Horowitz .S .Sahni . Fundamentals of Data Structures . New York: Computer Science Press, Pitman, 1976

9 Sara Baase . Computer Algorithms . Introduction to Design and Analysis . London: Addison-Wesley Publishing Company, 1978

10 K . Hwang, F .A . Briggs .Computer Architecture and Parallel Processing New York: McGraw- Hill, Publishing Company ,1984

11 N .Deo,C .Y . Pang, R .E . Lord .Two Parallel Algorithms for Shortest Path Problems .Int'l Conf . on Parallel Processing .1980 .244～253

12 E .F . Moore . The Shortest Path Through a Maze .Proc . Int 'l Symp .on Theory of Switching,2, 1959, 285～292

13 E . Chang, R . Roberts .An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes, Cornm .ACM,22(5),1979,281～283

14 陈国良 .并行算法——排序和选择 .合肥:中国科学技术大学出版社,1990

15 唐策善,梁维发 .并行图论算法 .合肥:中国科学技术大学出版社,1991