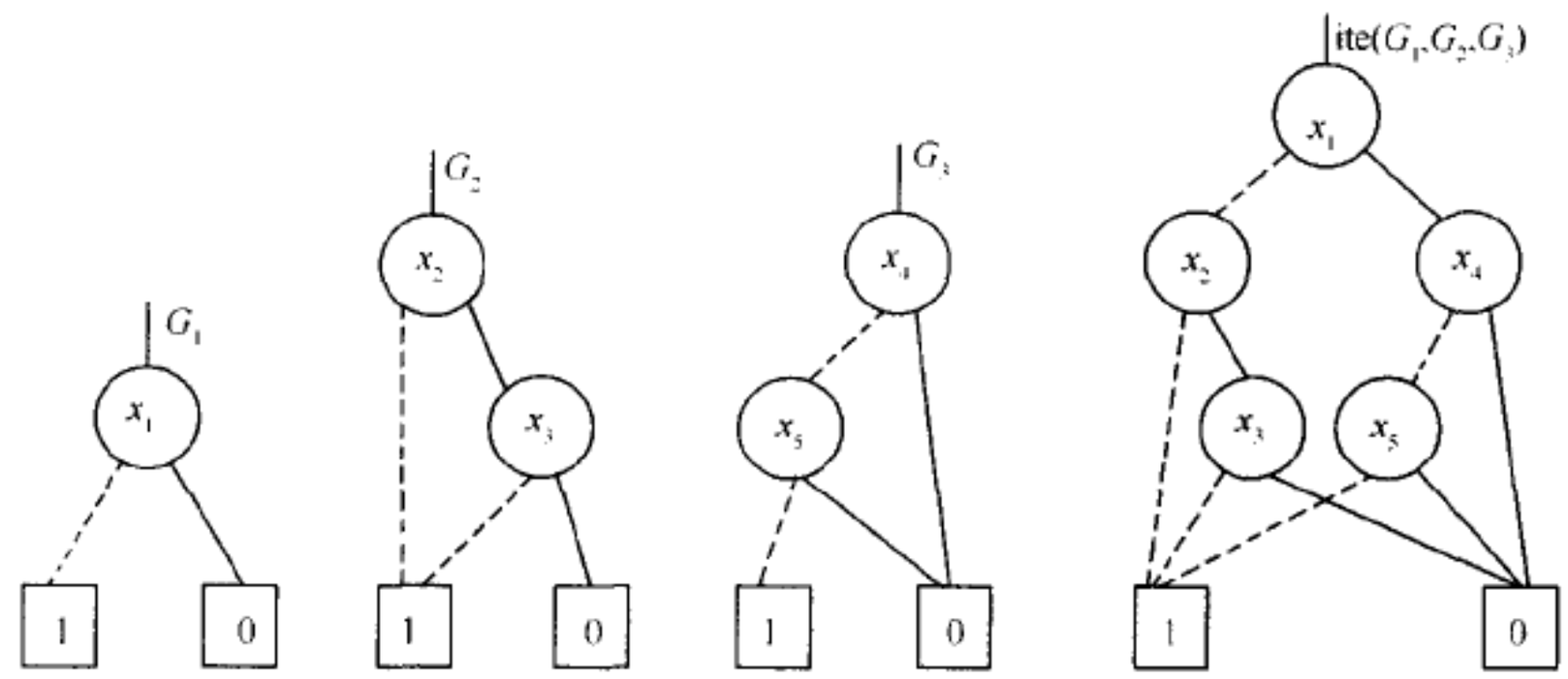



图 7.12　变量不交叉时的 BDD 构造

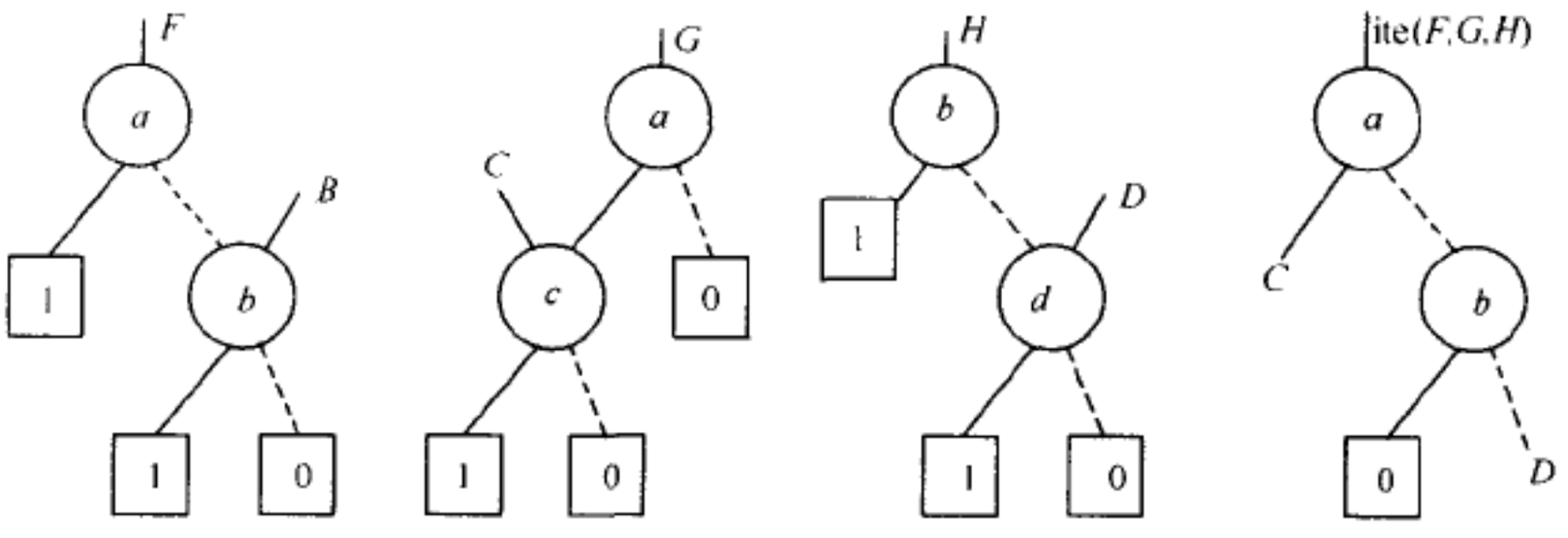



图 7.13　一般情况下的 BDD 构造

ite 迭代公式是整个 BDD 运算的核心，它能直接对 BDD 进行操作，并且能够实现所有的一元和二元逻辑运算，其他对 BDD 的操作也可以有效地建立在 ite 算符的基础上。用高级语言（如 $C$ 语言）可容易地编程实现 ite 迭代公式，这只需先定义 BDD 图中节点的数据结构，然后对图中节点的一些相关操作进行编程实现即可。

## 7.4　二元判定图的变量编序

在变量的编序给定的情况下，布尔函数的 BDD 表示是一种范式表示。由于 BDD 的结构与所给定的变量编序有关，因此如何使得 BDD 的节点数较少，也就是使得 BDD 的规模较小就成为 BDD 应用中的最重要的研究课题。

### 7.4.1　变量编序对二元判定图的影响

二元判定图的大小和形状对变量的编序很敏感。一般地，在两种不同的变量编序下，对应 BDD 的结构是不相同的。例如，对一个三位加法器函数 $f = x_1x_2 + x_3x_4 + x_5x_6$，在编序为 $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$ 时，它的二元判定图表示如图 7.14(a) 所示，在此种编序下，只要 8 个节点就可以表示。但是，若编序为 $x_1 < x_3 < x_5 < x_2 < x_4 < x_6$ 时，它的二元判定图表示如图 7.14(b) 所示，此时则需要 16 个节点才能表示。

一般地，对 $n$ 位加法器函数 $f = \sum_{i=1}^{n} x_{2i-1}x_{2i}$，在编序 $x_1 < x_2 < x_3 < \cdots < x_{2n}$ 时，$f$ 可以用 $2n+2$ 个节点的 BDD 来表示；而在编序 $x_1 < x_3 < x_5 < \cdots < x_{2n-1} < x_2 < x_4 < x_5$

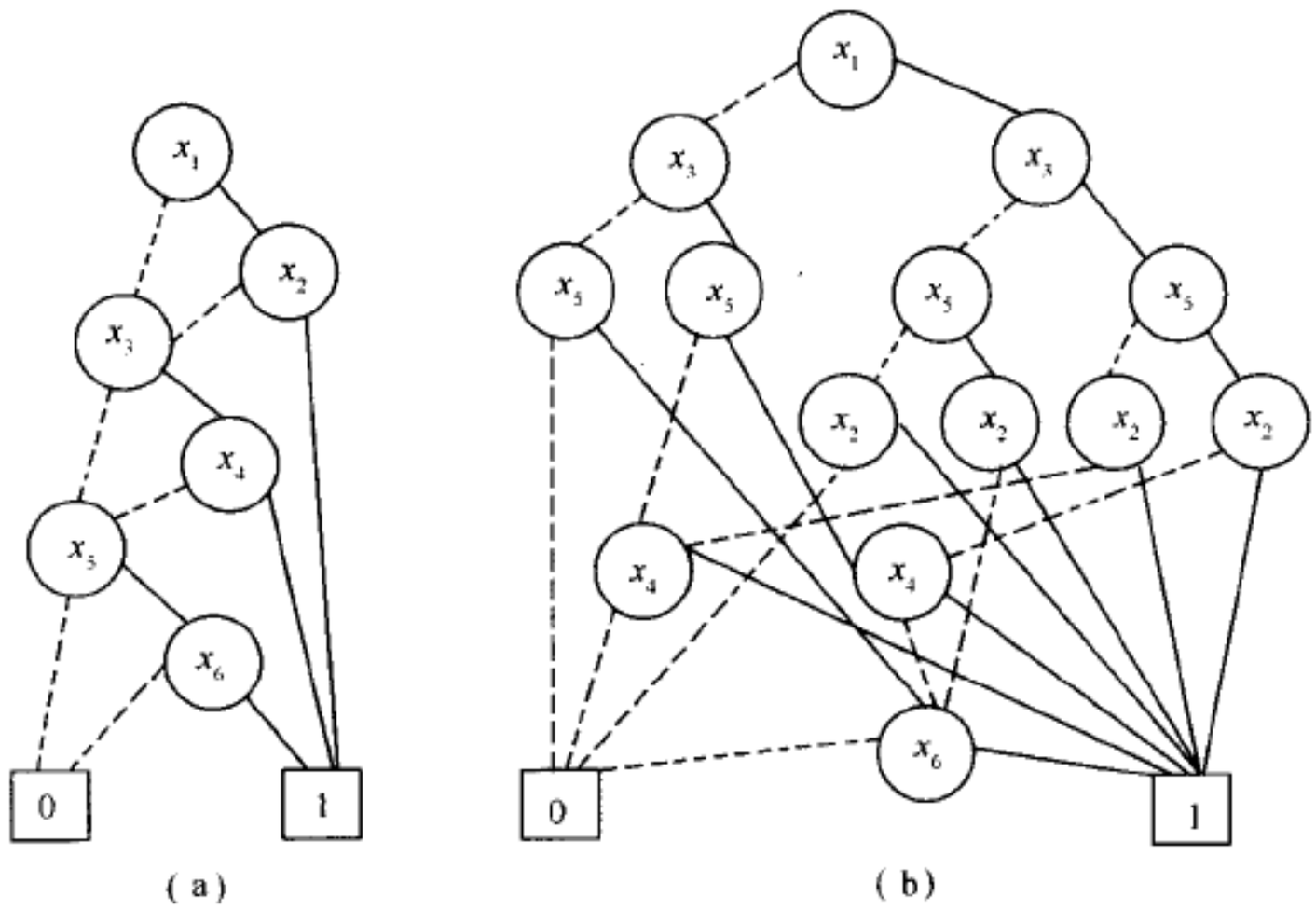



图 7.14　三位加法器在两种编序下的 BDD

$< \cdots < x_{2n}$ 时，$f$ 需要 $2^{n+1}$ 个节点构成的 BDD 才能表示。

从图 7.14 中这两个 BDD 的结构和相应的变量编序可以分析出造成这种差异的原因。在图 7.14(a) 中，变量的编序与变量在函数表达式中的出现顺序是相同的，所以 BDD 的每两级(例如 $x_1$ 和 $x_2$ 这两级) 只需要两个节点，一个节点为终节点 1(在 $x_1 = x_2 = 1$ 时)，另一个节点为下两级的 BDD 的根节点。在图 7.14(b) 中，按给定的编序，由于前 3 个变量中任意两个的取值都不能决定布尔函数的最终取值，这 3 级节点就构成了一个完全二叉树，因此比图 7.14(a) 需要更多的节点。

对变量编序问题，当布尔函数为对称函数时，则对任何编序，对应的 BDD 具有相似的结构。因此，对这类函数通过变量编序方法不能降低函数所对应 BDD 的节点数。

目前，人们对变量编序进行了广泛研究。从整体上来看，可以将变量编序方法分为两类：静态变量编序和动态变量编序。

### 7.4.2　静态变量编序

静态变量编序的原理是：通过对布尔函数的结构进行分析，找出函数的特点，进而进行变量编序。所采用的分析方法通常有深度优先搜索、广度优先搜索、对布尔函数对应的逻辑电路结构进行分析等。

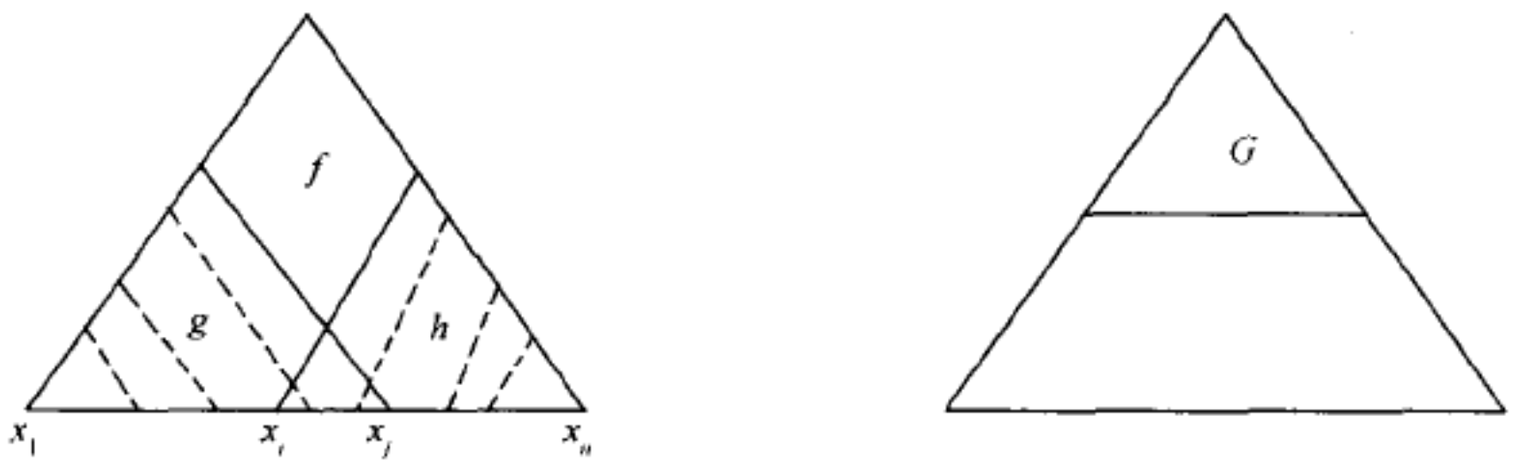
下面定性地讨论 BDD 节点数与变量编序的关系，并从中得出可用于静态变量编序的经验性方法。

对一般的布尔函数而言，它可以由许多子函数通过运算来得到。为简便起见，这里假定函数 $f$ 是经过子函数 $g$ 和 $h$ 之间的运算而得到的。子函数 $g$ 依赖于变量 $x_1 \sim x_j$，子函数 $h$ 依赖于变量 $x_i \sim x_n$。

假定函数 $f$ 的 BDD 可以分成上下两个部分，如图 7.15(b) 所示。上半部分实现了 $g$ 的全部功能，这时至少包含了 $x_1 \sim x_j$ 的所有 $j$ 个变量。

如果在编序时没有根据子函数 $g$ 和 $h$ 所依赖的变量进行编序，而将一些在 $x_j \sim x_n$ 的变量插入到 $x_1$ 和 $x_j$ 之间，则图 7.15(b) 的下半部分实现的功能还取决于被插入到上半部

分的这些变量。这就会增多下半部分的节点，使总节点数也增多。




(a)　　(b)

图 7.15　变量交叉的例子

(a) 变量交叉；(b) 函数的 BDD。

因此，为减少节点数，可以得到如下规则：

**规则 1**　被子函数所依赖的变量尽量放在一起。

该规则在使用时又可对子函数进行递归应用，一直到达原始变量为止。这就类似于深度优先搜索遍历的过程。

例如，对图 7.14 的加法器，使用**规则 1**，在对变量编序时使各子函数所依赖的变量没有交叉，例如 $x_1 < x_2 < x_3 < \cdots < x_{2n}$ 时，这样的编序是最佳的。

对一类特殊的组合电路的无扇出电路建立 BDD 时，对变量编序若使用**规则 1**，也能得到较好的编序。在无扇出电路中，所有门的输出都直接作为其他门的输入，任何信号线都没有扇出分支。因此在这种电路中没有子电路的交叉。在编序时，可以从原始输出开始，按深度优先遍历来产生输入变量的编序。例如，对图 7.16 的无扇出电路，所得到的编序 $x_1 < x_2 < x_5 < x_3 < x_4 < x_6$，就是一种最佳编序。

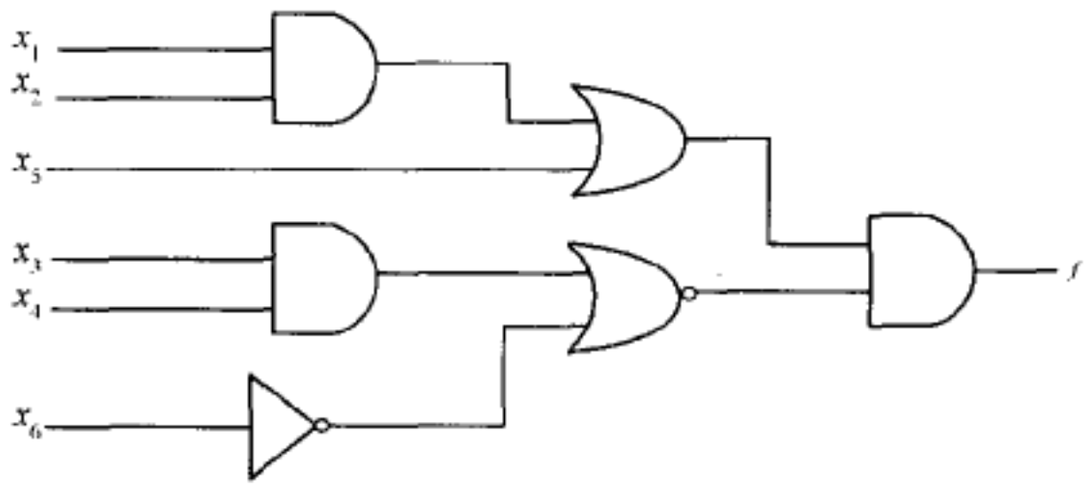



图 7.16　无扇出电路的例子

在使用**规则 1** 时，会遇到在同一子函数时如何考虑变量的编序问题，以及若各个子函数依赖的变量本身就存在交叉，例如图 7.15 中 $i < j$ 的情况。这时除了优先考虑较重要的子函数之外，还需使用如下的**规则 2**。

**规则 2**　对逻辑函数输出贡献大的输入变量在编序中优先考虑，应靠前。

这里关键是确定每个输入变量的“贡献值”。这可以根据变量在函数中出现的次数以及在子函数中的作用来确定；也可以根据一个逻辑函数所对应的电路拓扑结构信息来计算。例如在后面将介绍的“动态加权赋值法”就是基于此原理。

前面的**规则 1** 和**规则 2** 主要是针对只有一个输出的逻辑函数的情况来考虑的。若对多输出逻辑函数的情况，则需要特别处理。这是因为：一个特定的变量编序，对某个输出而言可能是较好的编序，但对另一个输出而言，就可能是一种差的编序。如何在兼顾所有输出的情况下，从全局的角度来获得一个最佳的编序，是具有现实意义的。对此，可以使用如

下的**规则 3**。

**规则 3**　对多个输出有影响的输入变量在编序中应尽量靠后。

对多输出函数(或多输出电路),通常采用共享二元判定图 SBDD 来表示,以共享同构子图,减少总节点数。图 7.17 给出了一个这方面的例子。这里函数 $f$ 有两个分量 $f_1$ 和 $f_2$,即 $f=(f_1,f_2)$,$f_1=x_1+x_2$,$f_2=x_1x_3$。图 7.17(a) 和图 7.17(b) 分别是函数 $f_1$ 和 $f_2$ 各自的 ROBDD 表示。图 7.17(c) 是在编序 $x_1<x_2<x_3$ 下,$f_1$ 和 $f_2$ 的 SBDD 表示,即是函数 $f$ 的 BDD 表示。图 7.18(c) 是在编序 $x_2<x_3<x_1$ 下,函数 $f$ 的 BDD 表示。

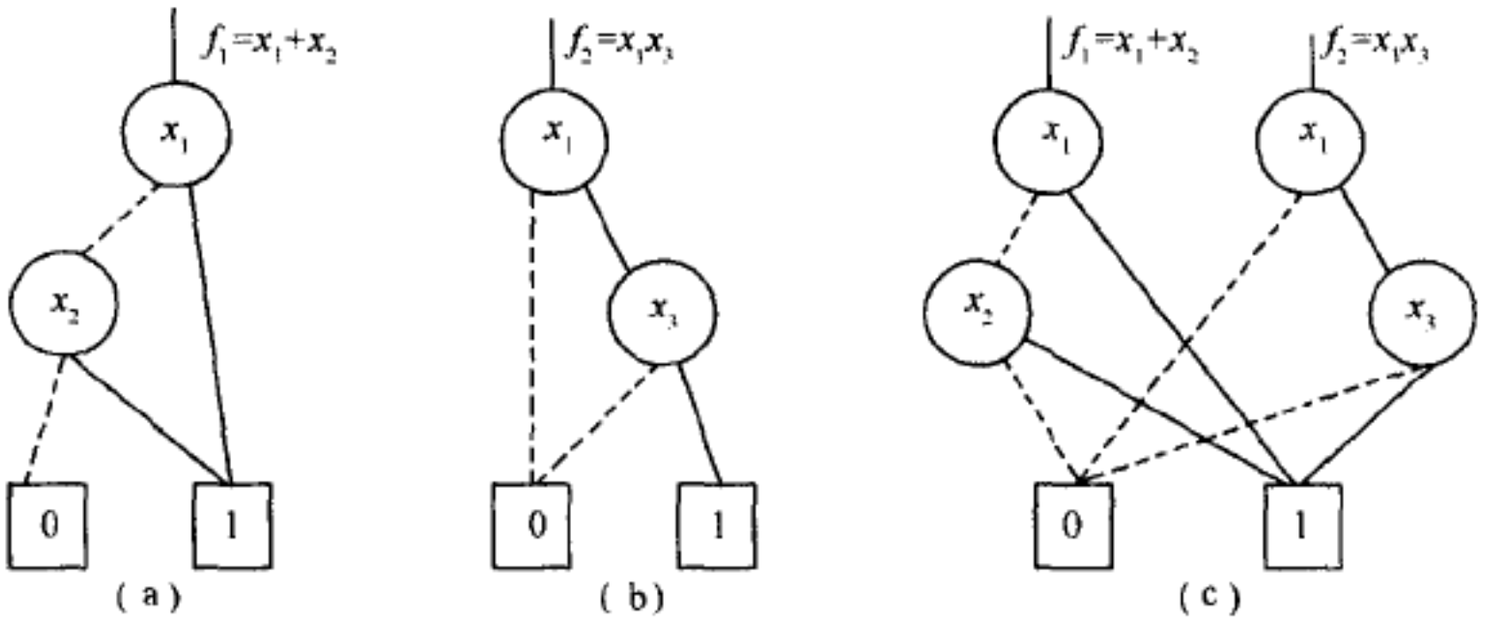



图 7.17　多输出函数$(x_1+x_2,x_1x_3)$在编序 $x_1<x_2<x_3$ 下的 BDD

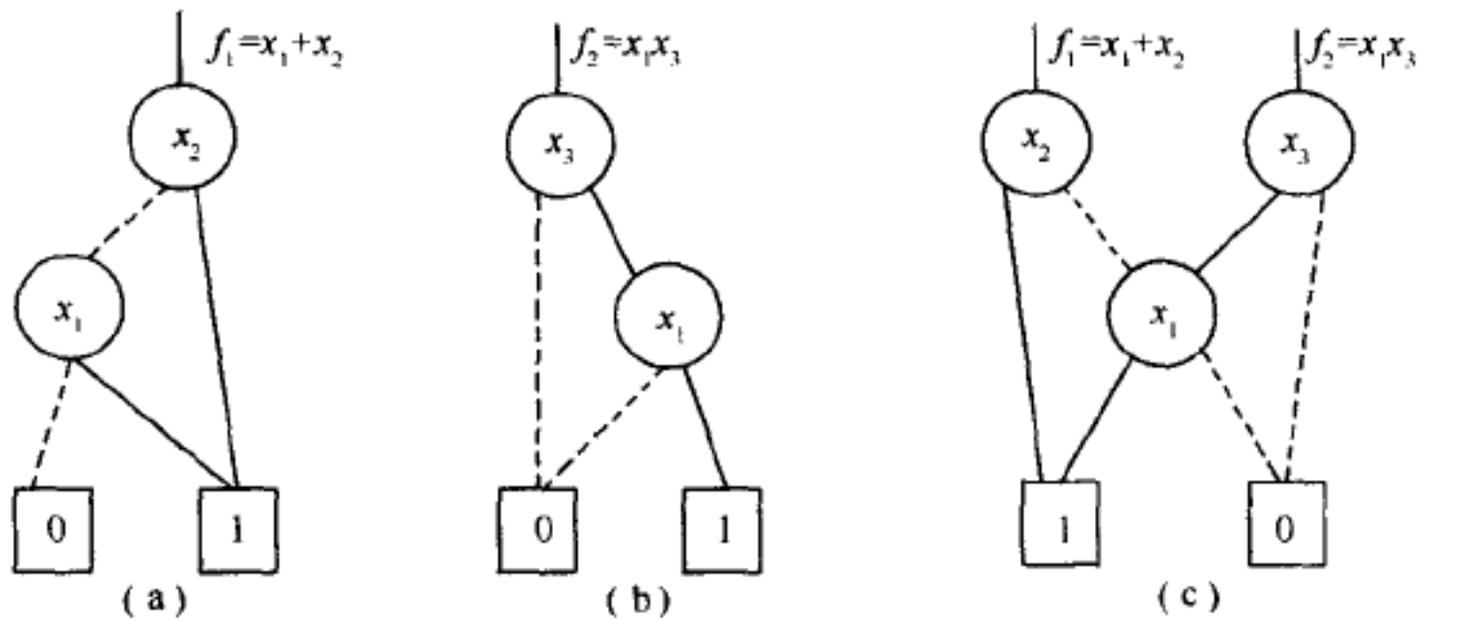



图 7.18　多输出函数$(x_1+x_2,x_1x_3)$在编序 $x_2<x_3<x_1$ 下的 BDD

由图 7.17 和图 7.18 可知,变量 $x_1$ 对函数 $f$ 的两个输出 $f_1$ 和 $f_2$ 都有影响。在编序 $x_1<x_2<x_3$ 下,$f$ 的 BDD 有 6 个节点;当把 $x_1$ 放在编序的最后,即使用编序 $x_2<x_3<x_1$ 时,$f$ 的 BDD 有 5 个节点,这比在前一种编序下,减少了一个节点。

由于实际的布尔函数多种多样,在使用上述 3 种规则时,有时可能是相互矛盾的,即 3 种规则不能同时使用。大量的实验表明,没有一种规则对所有的布尔函数都有效。因此对 BDD 节点数与变量编序所存在的内在规律的研究,还将不断进行下去。

下面介绍静态变量编序的一种具有代表性的方法:动态加权赋值法。

这种方法主要用于与布尔函数相对应的组合电路进行变量编序。它的基本出发点是:对电路的输出功能有较大影响的原始输入应有较高的“贡献值”,在编序中应靠前;在电路结构中处于同一区域的临近节点的“贡献值”的大小应相近。

对单输出电路,动态加权赋值法的实现过程如下。

首先对原始输出赋以权值 1.0,此权值按如下的规则向原始输入端进行传播:

(1) 在每个门,输出节点的权值被均等地分配到它的各个输入节点。

(2) 在每一个扇出点,所有扇出分支的权值的总和等于扇出源的权值。

由于节点的权在拓扑意义上反映了对电路原始输出的贡献,对于具有较大权的原始输入就可预知其对电路的输出功能有较大的影响,因此把最高的"贡献值"赋予具有最大权的原始输入。通过上述(1) 步和(2) 步,可以为每一个原始输入分配一个权值,之后选出具有最大权值的一个原始输入。

随后,为了挑选下一个原始输入,可删除电路中的一部分,该部分是与已选定(选出)的具有最大权值的这个原始输入有关的一些信号线组成。具体地说,是从具有最大权值的这个原始输入出发至原始输出的所有信号线中,除去未选定的其他原始输入至原始输出的信号线之后,剩余的信号线所组成的电路部分。这样,反复地权赋值和子电路的删除就可以获得原始输入的一种适当编序。下面以图 7.19 为例进行说明。

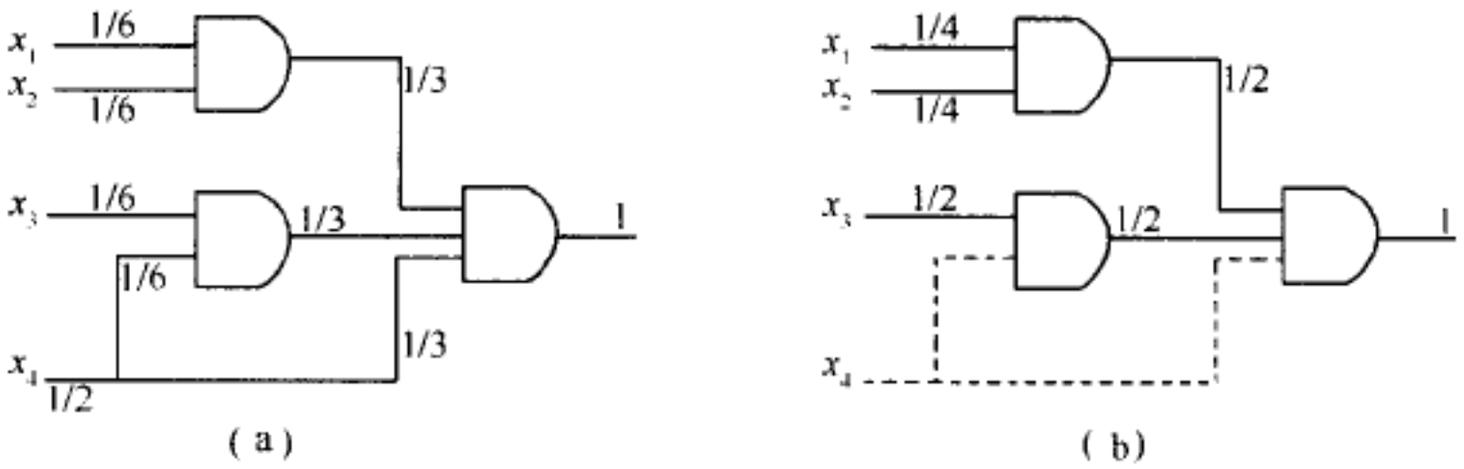



图 7.19　动态加权赋值法

(a) 第一次赋值;(b) 第二次赋值。

在第一次赋值之后,可以求得:原始输入 $x_4$ 的权值是 1/2,比其他原始输入 $x_1$、$x_2$ 和 $x_3$ 的权值都大,因此 $x_4$ 应排在编序的最前面。由于其他输入 $x_1$、$x_2$ 和 $x_3$ 的权值相等,都为 1/6,为进一步确定这 3 个输入每一个对原始输出的贡献大小,随后进行第二次赋值。此时删除了与输入 $x_4$ 相关的部分电路,如图 7.19(b) 所示,求得到 $x_3$ 的权为 1/2,在这 3 个输入 $x_1$、$x_2$ 和 $x_3$ 中是最大的,故 $x_3$ 应排在编序的前面。由于对剩下的两个输入 $x_1$ 和 $x_2$ 进行第三次赋值时,它们两个的权值相等都为 1/2,故其顺序可以随意安排。因此,得到的编序为 $x_4 < x_3 < x_2 < x_1$ 或 $x_4 < x_3 < x_1 < x_2$。

如上讨论的动态加权赋值法仅针对单输出电路的情况。当电路有多个原始输出时,可首先选择到原始输入端最深的路径的原始输出来操作,并进行权赋值。在这之后,如果还有原始输入没有编序,则选择下一个次深的路径对余下的原始输入进行编序,直到对整个电路的所有原始输入都进行了编序为止。

在很多情形下,用动态加权赋值法可以获得较好的编序。该算法的计算复杂度为 $O(m \cdot n)$。这里 $m$ 和 $n$ 分别是电路中的基本门的个数和电路中的原始输入个数。在实际应用中,这种复杂性是完全可以接受的。

### 7.4.3　动态变量编序

由于布尔函数和电路功能的多样性,上节所讨论的静态变量编序方法不一定适合于任何情况,为此人们提出了动态变量编序方法。这种方法是从某个初始编序开始,在操作过程中阶段性地不断调整变量的编序,使得各个阶段的 BDD 节点总数保持最少。随着

BDD 在一些复杂的领域应用，这种动态变量编序显得尤为必要。下面介绍几种典型的动态变量编序方法，主要有局部变量交换、窗口置换编序、过滤法、分布式编序法。局部变量交换是直接在编序中交换第 $i$ 和第 $j$ 个位置的变量 $x_i$ 和 $x_j$，因此实现过程较简单，而其余的 3 种方法则需要进行多个位置或变量的交换。

1. 窗口置换编序

在进行窗口置换之前，要先将当前的 BDD 图划分为多级。级的划分是按判决变量从根节点到终节点的次序来进行的。根节点的级为最高。若判决变量有 $n$ 个 $x_1, x_2, \cdots, x_n$，则 BDD 图共有 $n+1$ 级，其中第 0 级的节点只有终节点 0 或 1，第 $n$ 级为根节点。

窗口置换方法对于一个选定的级 $i(1<i\leqslant n)$，通过如下的变量交换方式来获得当前条件下的一种编序。选定窗口大小 $k$，这里 $k$ 是一个整数，它的值较小，且满足 $k<(n-i)$。该方法穷举搜索从第 $i$ 级开始的 $k$ 个相邻变量的所有排序，以找出在级 $i$ 的一种较优编序。在具体实现时，将先进行 $k!-1$ 次每对变量之间的位置交换；然后再进行 $k(k-1)/2$ 次的变量交换，以求出较优的编序。

对 BDD 图的每一级 $i(i=1,2,\cdots,n)$ 都要进行上述过程，直到 BDD 图中的节点没有再减少为止。此时所得到的变量编序，即是所求得的编序。

表 7.3 是这种方法的一个例子。

表 7.3　窗口置换方法的例子

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | 初始编序 | |
|---|---|---|---|---|---|---|---|---|
| $x_1$ | $x_3$ | $x_2$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | 交换 $x_2, x_3$ | |
| $x_1$ | $x_3$ | $x_4$ | $x_2$ | $x_5$ | $x_6$ | $x_7$ | 交换 $x_2, x_4$ | * |
| $x_1$ | $x_4$ | $x_3$ | $x_2$ | $x_5$ | $x_6$ | $x_7$ | 交换 $x_3, x_4$ | |
| $x_1$ | $x_4$ | $x_2$ | $x_3$ | $x_5$ | $x_6$ | $x_7$ | 交换 $x_3, x_2$ | |
| $x_1$ | $x_2$ | $x_4$ | $x_3$ | $x_5$ | $x_6$ | $x_7$ | 交换 $x_4, x_2$ | |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | 交换 $x_4, x_3$ | |
| $x_1$ | $x_3$ | $x_2$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | 交换 $x_2, x_3$ | |
| $x_1$ | $x_3$ | $x_4$ | $x_2$ | $x_5$ | $x_6$ | $x_7$ | 交换 $x_2, x_4$ | |

在表 7.3 中，设 BDD 有 7 个判决变量 $x_1, x_2, \cdots, x_7$。在对图的第一级进行窗口置换时，假定变量的初始编序为 $x_1<x_2<x_3<x_4<x_5<x_6<x_7$，见表 7.3 的第一行。当所选定的窗口大小为 $k=3$ 时，将从初始编序的 $x_2$ 开始的依次 3 个变量 $x_2$、$x_3$、$x_4$ 进行穷举搜索，而 $x_1$、$x_5$、$x_6$、$x_7$ 在编序中的位置不发生变化。

这时需要进行 $3!-1=5$ 次变量交换，这在表 7.3 的中间部分说明。此外，还需进行 $3(3-1)/2=3$ 次附加的变量交换，这在表 7.3 的最下面部分说明。

在窗口值换算法中进行附加的 $k(k-1)/2$ 次变量交换的目的是使得算法的在进行过程中恢复到在 $k!-1$ 次置换中所得到的较好编序。例如，在表 7.3 的例子中，前 5 次的最好编序假定是 $x_1<x_3<x_4<x_2<x_5<x_6<x_7$，即表 7.3 中的第三行，末尾有“*”标注。

在窗口置换算法中，当 $k$ 选得较大时，则置换次数 $k!-1$ 较大。因此，一般选取 $k=4$ 或 $k=5$ 较合适。

总之，窗口置换算法虽然在特定的某一级进行的是局部的变量排序，但由于它对图的每一级都进行处理，因此在整体上它的行为涉及所有的判决变量。如果窗口大小选得恰当，这种算法可得到一种适当的变量编序。

2. 过滤法

在窗口置换法中，为避免较大的计算复杂性，窗口大小 $k$ 一般都选得较小。如果要交换两个在编序中隔得较远的变量，例如这两个变量的在编序中相差的位置数目大于 $k$，则在对这一级的窗口置换过程中，就不能交换这两个变量的位置。这时需要进行多级的窗口置换处理才能完成。

为更好地处理这种情况，人们提出了过滤法(Sifting Algorithm)。它的基本思想是：对某一个变量而言，假定其他的所有变量在编序中的位置固定，通过算法的操作去找到该变量在编序中的最佳位置。若在 BDD 中有 $n$ 个变量，对一个变量来说，就存在 $n$ 个潜在的编序位置。过滤法的目标就是要在这 $n$ 个编序位置中找到能使 BDD 规模最小的那个位置。

从表面上看，在所有其他变量的编序位置保持固定的条件下，可以用低的计算复杂度完成对 BDD 的操作，进而求出变量的最佳位置。但在实际上，变量的最佳位置需要通过如下的复杂枚举过程来确定。

对于选定的一个变量，首先，在当前编序中，把它与其相邻的后继变量进行交换。这样重复进行多次，直到该变量成为 BDD 中的最后一个变量(当然也是编序中的最后一个变量)，也就是，把该变量移动到 BDD 的最底端。然后，把该变量依次与它的前驱变量交换，直到使它成为 BDD 的根节点的判决变量(即成为编序中的第一个变量)。在这一过程中最佳的 BDD 规模被记住，设此时该变量处于第 $\alpha$ 级。然后从 BDD 的顶部(根节点) 开始移动该变量到 BDD 的第 $\alpha$ 级。

表 7.4 是过滤法的一个例子。假定 BDD 中有 7 个判决变量。这里移动的变量为 $x_4$。使用 9 次相邻变量的交换，可以使 $x_4$ 处于编序中 7 个位置中的每一个；再通过附加的 6 次交换，使得 $x_4$ 重新到达其在编序中的最佳位置。

表 7.4　过滤法的例子

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | 初始编序 |
|---|---|---|---|---|---|---|---|
| $x_1$ | $x_2$ | $x_3$ | $x_5$ | $x_4$ | $x_6$ | $x_7$ | 交换 $x_4$，$x_5$ |
| $x_1$ | $x_2$ | $x_3$ | $x_5$ | $x_6$ | $x_4$ | $x_7$ | 交换 $x_4$，$x_6$ |
| $x_1$ | $x_2$ | $x_3$ | $x_5$ | $x_6$ | $x_7$ | $x_4$ | 交换 $x_4$，$x_7$　* |
| $x_1$ | $x_2$ | $x_3$ | $x_5$ | $x_6$ | $x_4$ | $x_7$ | 交换 $x_7$，$x_4$ |
| $x_1$ | $x_2$ | $x_3$ | $x_5$ | $x_4$ | $x_6$ | $x_7$ | 交换 $x_6$，$x_4$ |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | 交换 $x_5$，$x_4$ |
| $x_1$ | $x_2$ | $x_4$ | $x_3$ | $x_5$ | $x_6$ | $x_7$ | 交换 $x_3$，$x_4$ |
| $x_1$ | $x_4$ | $x_2$ | $x_3$ | $x_5$ | $x_6$ | $x_7$ | 交换 $x_2$，$x_4$ |
| $x_4$ | $x_1$ | $x_2$ | $x_3$ | $x_5$ | $x_6$ | $x_7$ | 交换 $x_1$，$x_4$ |
| $x_1$ | $x_4$ | $x_2$ | $x_3$ | $x_5$ | $x_6$ | $x_7$ | 交换 $x_4$，$x_1$ |
| $x_1$ | $x_2$ | $x_4$ | $x_3$ | $x_5$ | $x_6$ | $x_7$ | 交换 $x_4$，$x_2$ |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | 交换 $x_4$，$x_3$ |
| $x_1$ | $x_2$ | $x_3$ | $x_5$ | $x_4$ | $x_6$ | $x_7$ | 交换 $x_4$，$x_5$ |
| $x_1$ | $x_2$ | $x_3$ | $x_5$ | $x_6$ | $x_4$ | $x_7$ | 交换 $x_4$，$x_6$ |
| $x_1$ | $x_2$ | $x_3$ | $x_5$ | $x_6$ | $x_7$ | $x_4$ | 交换 $x_4$，$x_7$ |

过滤法在总体上按如下方式进行。首先将 BDD 按判决变量从根节点到终节点的次序来分级。根据在每一级的节点数大小将这些判决变量进行排列，节点数多的排在前，节点数少的排在后，即将判决变量按它们各自所对应的节点数大小进行降序排列。然后对一个变量而言，假定其他的变量不移动，通过移动该变量，使其达到它的局部最佳位置。在整个

过程中，每个变量的移动过程只进行一遍。

在过滤法的进行过程中，刚开始的几次变量移动可能会使 BDD 的规模有较大的增加，但随着处理过程的进行，最终所得到的 BDD 的规模会小于初始编序时的规模。

过滤法需要在 BDD 的相邻级之间进行 $O(n^2)$ 次变量交换，这里 $n$ 为判决变量数。每一次这种变量的交换所需的时间与 BDD 的宽度成正比。为了控制整个过程的总的复杂性，当在对一个选定的变量用过滤法进行处理时，若出现的 BDD 的规模为初始编序时 BDD 规模的两倍以上时，则终止对该变量的后续操作，然后选其他另一变量进行过滤处理。

此外，在如上讨论的窗口置换方法和过滤法中，可以把多个变量看成一组，把一组作为一个整体，每次对这一组变量来进行类似的置换与移动操作。

3. 分布式编序法

这种分布式编序方法是对过滤法的扩展，它只允许对 BDD 的一部分判决变量重新进行编序，而不是针对全部变量来进行。

首先定义变量间的一个距离参数 $h$，通过距离 $h$ 把 BDD 的 $n$ 个判决变量 $x_1, x_2, \cdots, x_n$ 所组成的集合分成多个子集，变量的交换仅局限于在同一个子集中的变量之间进行。当距离 $h = 1$ 时，这种方法就成为过滤法。因此过滤法是分布式编序法的特殊情况。

例如，设变量数为 15，距离 $h$ 定义为 4。BDD 的判决变量被划分成 4 个子集 $S_1$、$S_2$、$S_3$ 和 $S_4$，这里 $S_1 = \{x_1, x_5, x_9, x_{13}\}$，$S_2 = \{x_2, x_6, x_{10}, x_{14}\}$，$S_3 = \{x_3, x_7, x_{11}, x_{15}\}$，$S_4 = \{x_4, x_8, x_{12}\}$。对每个集合 $S_i (1 < i \leqslant 4)$ 中的变量，变量的交换只能与该集合中的其他变量进行，不能与该集合外的变量进行。

表 7.5 是一个分布式编序法的例子。这里的变量数为 15，距离 $h$ 定义为 2。针对子集 $S_1 = \{x_2, x_4, x_6, x_8, x_{10}, x_{12}, x_{14}\}$ 中的变量进行重新编序。该子集 $S_1$ 共有 7 个变量，它们在 BDD 的整个 15 个变量所形成的编序中占有 7 个位置。这里选变量 $x_8$ 来与子集 $S_1$ 中的其他变量进行交换，这时 $x_8$ 共有 7 种可能的（候选）位置。使用了 9 次变量交换之后，再使用附加的 6 次交换，使得 $x_8$ 重新到达了它在编序中的最佳位置。

表 7.5　分布式编序法的例子

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | 初始编序 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_{10}$ | $x_9$ | $x_8$ | $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | 交换 $x_8, x_{10}$ |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_{10}$ | $x_9$ | $x_{12}$ | $x_{11}$ | $x_8$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | 交换 $x_8, x_{12}$ |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_{10}$ | $x_9$ | $x_{12}$ | $x_{11}$ | $x_{14}$ | $x_{13}$ | $x_8$ | $x_{15}$ | 交换 $x_8, x_{14}$ * |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_{10}$ | $x_9$ | $x_{12}$ | $x_{11}$ | $x_8$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | 交换 $x_{14}, x_8$ |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_{10}$ | $x_9$ | $x_8$ | $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | 交换 $x_{12}, x_8$ |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | 交换 $x_{10}, x_8$ |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_8$ | $x_7$ | $x_6$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | 交换 $x_6, x_8$ |
| $x_1$ | $x_2$ | $x_3$ | $x_8$ | $x_5$ | $x_4$ | $x_7$ | $x_6$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | 交换 $x_4, x_8$ |
| $x_1$ | $x_8$ | $x_3$ | $x_2$ | $x_5$ | $x_4$ | $x_7$ | $x_6$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | 交换 $x_2, x_8$ |
| $x_1$ | $x_2$ | $x_3$ | $x_8$ | $x_5$ | $x_4$ | $x_7$ | $x_6$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | 交换 $x_8, x_2$ |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_8$ | $x_7$ | $x_6$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | 交换 $x_8, x_4$ |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | 交换 $x_8, x_6$ |
| $x_1$ | $x_2$ | $x_3$ | $x_1$ | $x_5$ | $x_6$ | $x_7$ | $x_{10}$ | $x_9$ | $x_8$ | $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | 交换 $x_8, x_{10}$ |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_{10}$ | $x_9$ | $x_{12}$ | $x_{11}$ | $x_8$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | 交换 $x_8, x_{12}$ |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_{10}$ | $x_9$ | $x_{12}$ | $x_{11}$ | $x_{14}$ | $x_{13}$ | $x_8$ | $x_{15}$ | 交换 $x_8, x_{14}$ |

在上面对分布式编序法的讨论中，使用了变量间的距离参数 $h$，其实可以不使用 $h$，而从 BDD 的判决变量集 $\{x_1, x_2, \cdots, x_n\}$ 中按一定的规则选出多个变量例如 $m$ 个（$m \leqslant n$）来组成子集，使变量的交换仅在该子集中进行。因此分布式编序法适合于变量交换的最一般情况。

# 第 8 章　二元判定图的性质与应用

本章讨论降低 BDD 规模的局部变量交换和最优编序以及 BDD 在电路设计中的具体应用。主要内容有：二元判定图的变量编序中局部变量交换的效果分析、变量的最优编序及其复杂性；二元判定图的一个类型 ZBDD 及应用；二元判定图在电路的逻辑综合和在电路测试中的应用。

## 8.1　变量编序中变量交换的效果分析

### 8.1.1　局部变量交换的效果

考虑有 $n$ 个变量 $x_1, x_2, \cdots, x_n$ 的布尔函数 $f(x_1, x_2, \cdots, x_n)$。对变量 $x_i$，其香农展开式为

$$f = \overline{x}_i \cdot f|_{x_i=0} + x_i \cdot f|_{x_i=1}$$

令

$$g = f|_{x_i=0}, h = f|_{x_i=1}$$

对变量的交换，下面首先考虑使 $x_i$ 迁移到编序中的第 $j$ 个位置，而原来的变量 $x_j$ 及之后的变量都后移一个位置，把这种情形记为 jump$(i,j)$。若 $j<i$，则称 jump$(i,j)$ 为向前迁移，记为 jump-up$(i,j)$；若 $j>i$，则 jump$(i,j)$ 被称为向后迁移，记为 jump-down$(i,j)$。

下面的算法说明了对向前或向后迁移的实现过程。

输入：布尔函数 $f$ 在编序 $\pi$ 下的一种 BDD 表示 $G$。不失一般性，设编序 $\pi$ 为 $x_1, x_2, \cdots, x_n$。

输出：函数 $f$ 在进行变量 $i$ 与 $j$ 的交换之后所得到的 BDD 表示 $G'$。此时，经过变量交换之后新的变量编序设为 $\pi'$。

**步骤 1**：构建 $f|_{x_i=0}$ 的 BDD，设为 $G_0$。这可通过重定向指向 $x_i$ 节点的所有边到它们的 0 边后继节点来实现。

**步骤 2**：构建 $f|_{x_i=1}$ 的 BDD，设为 $G_1$。这可通过重定向指向 $x_i$ 节点的所有边到它们的 1 边后继节点来实现。

**步骤 3**：用对 $x_i$ 的分析来构建二元判决图 $G'$。它是由 $G_1$ 和 $G_0$ 的 ite 操作来实现的，即 $\text{ite}(x_i, g, h) = x_i \cdot g + \overline{x}_i \cdot h$。该步是在编序 $\pi'$ 下来进行的。

此算法的时间复杂度为 $O(|G|^2)$。

对变量的交换过程，下面讨论 BDD 规模增大或减小的范围。

对布尔函数 $f$，由香农展开公式，可以得到一个完全二叉树。在此树中，判决变量为 $x_i$ 的节点有 $2^{i-1}$ 个。

对每一个 $s \subseteq \{1, 2, \cdots, i-1\}$，存在与 $s$ 有关的节点（记为 $v_s$）和与 $S$ 有关的路径（记

为 $P_s$)，使得由路径 $P_s$ 可到达节点 $v_s$。这种路径 $P_s$ 的长度为 $i-1$；路径 $P_s$ 的各个组成节点按如下方式定义：在判决变量为 $x_j$ 的节点，若 $j \notin s$，则选择 0 边的后继；若 $j \in s$，则选择 1 边的后继。

由 BDD 的 $v_s$ 节点所表示的函数称为 $f^s_{\mathrm{BDD}}$。由香农展开，该函数等于

$$f\mid_{x_1=a_1,x_2=a_2,\cdots,x_{i-1}=a_{i-1}}$$

这里，若 $j \notin S$，则 $a_j = 0$；若 $j \in S$，则 $a_j = 1$。

**定理 8.1** 在变量编序 $x_1, x_2, \cdots, x_n$ 下，布尔函数 $f$ 的 BDD 中判决变量为 $x_i$ 的节点数等于函数 $f^s_{\mathrm{BDD}}$ 的个数，这里 $s \subseteq \{1,2,\cdots,i-1\}$，并且这些函数 $f^s_{\mathrm{BDD}}$ 必须依赖于变量 $x_i$。

这里，一个函数 $g$ 必须依赖于变量 $x_j$，是指 $g\mid_{x_j=0}$ 与 $g\mid_{x_j=1}$ 不相同。

**证明**：固定一些判决变量 $i$。若对一些集合 $s$ 和 $s' \subseteq \{1,2,\cdots,i-1\}$，有 $f^s_{\mathrm{BDD}} = f^{s'}_{\mathrm{BDD}}$ 成立，则这些函数可由同一个节点来表示。若 $g = f^s_{\mathrm{BDD}}$ 不依赖于变量 $x_i$，则 $g\mid_{x_i=0} = g\mid_{x_i=1}$ 且 $g\mid_{x_i=0} \oplus g\mid_{x_i=1} = 0$，这时由 $g$ 表示的具有判决变量 $x_i$ 的节点可以被删去(这使用了 ROBDD 中冗余节点的删除规则)。因此，这时并不需要更多的节点。故**定理 8.1** 的结论成立。

**定理 8.2** 设在编序 $x_1, x_2, \cdots, x_n$ 下，布尔函数 $f$ 的 BDD 中判决变量为 $x_k$ ($k = 1,2,\cdots,n$) 的节点数为 $s_k$。在进行 jump-up$(i,j)$，$j < i$，即把 $x_i$ 向前迁移到位置 $j$ 之后，结果 BDD 规模的上界为：

$$s_1 + \cdots + s_{j-1} + 3(s_j + \cdots + s_{i-1}) + s_i + s_{i+1} + \cdots + s_n + 2$$

或

$$2(s_1 + s_2 + \cdots + s_{i-1}) + 1 + s_{i+1} + s_{i+2} + \cdots + s_n + 2$$

**证明**：设布尔函数 $f$ 的 BDD 表示为 $G$，把 $x_i$ 向前迁移到位置 $j$ 之后的 BDD 为 $G^*$。则 $G^*$ 所使用的编序为 $x_1, x_2, \cdots, x_{j-1}, x_i, x_j, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n$。设 $s_k{}^*$ 为 $G^*$ 中判决变量为 $x_k$ 的节点数。定理 8.1 说明，若 $k \leqslant j-1$ 或 $k \geqslant i+1$，则 $s_k{}^* = s_k$。而且，对 $k \in \{j,\cdots,i-1\}$，$s_k{}^* \leqslant 2s_k$ 成立。这时由于集合 $s' \subseteq \{1,2,\cdots,k-1,i\}$ 的数目是集合 $s \subseteq \{1,2,\cdots,k-1\}$ 的两倍。并且，若 $f^{s'-\{i\}}_{\mathrm{BDD}}$ 不必依赖于 $x_k$ 时，则 $f^{s'}_{\mathrm{BDD}}$ 也不必依赖于 $x_k$。

对 $s_i{}^*$ 这里不存在仅依赖于 $s_i$ 的较好的上界。若 $i = n$ 且 $s_i \leqslant 2$，这时 $s_i{}^*$ 可能成指数增长变得很大，例如对 $j = n/2$。下面对 $s_i{}^*$ 去证明有两个上界存在，从而说明定理中的对 $G^*$ 节点规模的两个上界。

在 $G^*$ 中判决变量为 $x_i$ 的每一个节点至少可从判决变量为 $x_k$ 的节点到达(这里 $k \leqslant j-1$)，或者从判决变量为 $x_i$ 的另一个节点到达。从判决变量为 $x_k$ ($k \leqslant j-1$) 的节点经过并到达一些层 $L$ ($L \geqslant j$) 的边最多有 $s_1 + s_2 + \cdots + s_{j-1} + 1$ 条。因此

$$s_i{}^* \leqslant s_1 + s_2 + \cdots + s_{j-1} + 1$$

$s_i{}^*$ 的另一个上界是 $s_j + \cdots + s_i$。为说明此上界的正确性，定义一个如下的一一映射。该映射将 $G^*$ 中判决变量为 $x_i$ 的 $s_i{}^*$ 个节点 $v$ 映射到在 $G$ 中由一些判决变量为 $x_k$ ($j \leqslant k \leqslant i$) 的 $s_j + \cdots + s_i$ 个节点 $w$。由**定理 8.1**，每一个节点 $v$ 对应于一个函数 $f^s_{\mathrm{BDD}}$，这里 $s \subseteq \{1,2,\cdots,j-1\}$，且 $f^s_{\mathrm{BDD}}$ 必须依赖于 $x_i$。

选取最小的 $k \in \{j,\cdots,i\}$，使得 $f^s_{\mathrm{BDD}}$ 必须依赖于 $x_k$，这时若 $s$ 是 $\{1,2,\cdots,k-1\}$ 的一个子集，则 $f^S_{\mathrm{BDD}}$ 不发生改变。

由**定理 8.1**，这个函数 $f^S_{\mathrm{BDD}}$ 可由 $G$ 中一个判决变量为 $x_k$ 的节点 $w(v)$ 来表示。下面证

明当 $v \neq v'$ 时有 $w(v) \neq w(v')$ 成立。若 $v \neq v'$，则 $f^s_{\mathrm{BDD}} \neq f^{s'}_{\mathrm{BDD}}$，否则 $v$ 和 $v'$ 在 $G^*$ 中可以合并。若 $k \neq k'$，则 $w(v) \neq w(v')$，这是由于 $w(v)$ 和 $w(v')$ 具有不同的判决变量。若 $k = k'$，则 $f^s_{\mathrm{BDD}}$ 和 $f^{s'}_{\mathrm{BDD}}$ 不必须依赖于 $x_j + \cdots + x_{k-1}$。因此这些函数也是不相同的，并且它们是由隐含 $w(v) \neq w(v')$ 的不同节点来表示的。

定理证毕。

此外，可以对变量 $x_{i-1}$ 和 $x_i$ 的交换操作进行如下的估计。由于变量 $x_{i-1}$ 和 $x_i$ 的交换等价于 $x_i$ 迁移到 $i-1$ 的位置，即 jump-up$(i, i-1)$。因此由**定理 8.2**，可得如下结论：在交换变量 $x_{i-1}$ 和 $x_i$ 之后，新的 BDD 的规模的上界为

$$2(s_1 + s_2 + \cdots + s_{i-1}) + 1 + s_{i+1} + s_{i+2} + \cdots + s_n + 2$$

或

$$s_1 + s_2 + \cdots + s_{i-2} + 3s_{i-1} + s_i + s_{i+1} + \cdots + s_n + 2。$$

**定理 8.3** 在编序 $x_1, x_2, \cdots, x_n$ 下，布尔函数 $f$ 对应的 BDD 中判决变量为 $x_k$ 的节点数为 $s_k$。在进行 $x_i$ 向后迁移到位置 $j$ 之后，即 jump-down$(i, j)$，且 $j > i$，则所得的结果 BDD 的规模的上界为

$$s_1 + \cdots + s_{i-1} + (s_{i+1} + \cdots + s_n + 2)^2$$

或

$$s_1 + \cdots + s_n + 2 + (2^{1+j-i} - 2) \cdot s_i$$

特别地，由一个向后迁移所引起的新的 BDD 的规模最多是原来的平方。

**证明**：设布尔函数 $f$ 的 BDD 表示为 $G$，把 $x_i$ 向后迁移到位置 $j$ 之后的 BDD 为 $G^*$。这时 $G^*$ 所使用的变量编序为：$x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_{j-1}, x_j, x_i, x_{j+1}, \cdots, x_n$。

设 $s_k$ 是 $G^*$ 中判决变量为 $x_k$ 的节点数。由**定理 8.1**，若 $k \leqslant i-1$ 或 $k \geqslant j+1$，则 $s_k{}^* = s_k$。对 $k \in \{i+1, \cdots, j\}$，$s_k{}^*$ 等于函数 $f^s_{\mathrm{BDD}}$ 的个数，这里 $s \subseteq \{1, \cdots, i-1, i+1, \cdots, k-1\}$，且函数 $f^s_{\mathrm{BDD}}$ 必须依赖于 $x_k$。

由香农定理，有 $f^s_{\mathrm{BDD}} = \overline{x}_i \cdot f^s_{\mathrm{BDD}}|_{x_i=0} + x_i \cdot f^s_{\mathrm{BDD}}|_{x_i=1}$，函数 $f^s_{\mathrm{BDD}}$ 必须依赖于 $x_k$，当且仅当函数 $f^s_{\mathrm{BDD}}|_{x_i=0}$ 和 $f^s_{\mathrm{BDD}}|_{x_i=1}$ 中至少有一个必须依赖于 $x_k$。这些函数都在 $G$ 中被表示。因此，存在 $s_k$ 个这种候选函数必须依赖于 $x_k$ 和存在 $s_{k+1} + \cdots + s_n + 2$ 个这种候选函数不必须依赖于 $x_k$。

考虑必须依赖于 $x_k$ 的这些函数的 3 种可能情况，可以得

$$s_k{}^* \leqslant s_k{}^2 + 2s_k(s_{k+1} + \cdots + s_n + 2)$$

下面仍去估计 $s_i{}^*$，它是必须依赖于 $x_k$ 的函数 $f^s_{\mathrm{BDD}}$ 的个数。现在 $f^s_{\mathrm{BDD}}$ 必须依赖于 $x_i$，当且仅当 $f^s_{\mathrm{BDD}}|_{x_i=0}$ 和 $f^s_{\mathrm{BDD}}|_{x_i=1}$ 不相同。这样，对 $f^s_{\mathrm{BDD}}|_{x_i=0}$ 有 $s_{j+1} + \cdots + s_n + 2$ 个候选函数；对 $f^s_{\mathrm{BDD}}|_{x_i=1}$ 有 $s_{j+1} + \cdots + s_n + 1$ 个候选函数。此外，这里有两个终端节点。因此，在 $x_i$ 向后迁移到位置 $j$ 之后，BDD 的规模的上界为

$$\sum_{1 \leqslant k \leqslant i-1} s_k + \sum_{i+1 \leqslant k \leqslant j} (s_k{}^2 + 2s_k(s_{k+1} + \cdots + s_n + 2)) + (s_{j+1} + \cdots + s_n + 2) \cdot (s_{j+1} + \cdots + s_n + 1) + \sum_{j+1 \leqslant k \leqslant n} s_k + 2$$

$$= s_1 + \cdots + s_{i-1} + (s_{i+1} + \cdots + s_n + 2)^2$$

对 $G^*$ 的第二个上界，通过一系列的变量交换：swap$(i, i+1)$，swap$(i+1, i+2)$，$\cdots$，swap$(j-1, j)$ 来实现 $x_i$ 向后迁移到位置 $j$，即 jump-down$(i, j)$。这里 swap$(i, i+1)$ 表示

交换变量 $x_i$ 和 $x_{i+1}$。

在**定理 8.2** 的第二个上界的证明中，对 $x_i$ 向前迁移时，在进行交换 $x_i$ 和 $x_{i+1}$ 之后，成立 $s_{i+1}{}^* \leqslant s_i + s_{i+1}$ 和 $s_i{}^* \leqslant 2 \cdot s_i$。这里隐含层 $i$（现在的判决变量为 $x_{i+1}$）至多含有 $s_i + s_{i+1}$ 个节点，层 $i+1$ 至多含有 $2s_i$ 个节点。在进行 $x_{k-1}$ 和 $x_k$ 的交换之后，层 $k$ 至多含有判决变量为 $x_i$ 的 $2^{k-i}$ 个节点，层 $k-1$ 至多含有判决变量为 $x_k$ 的 $s_k + 2^{k-i+1} s_i$ 个节点。因此 $G^*$ 的节点数的上界为

$$\begin{aligned} & s_1 + \cdots + s_{i-1} + s_{i+1} + \cdots + s_n + 2 + s_i(1 + 2 + \cdots + 2^{j-i}) \\ = \ & s_1 + \cdots + s_n + 2 + (2^{1+j-i} - 2)s_i \end{aligned}$$

定理证毕。

由**定理 8.3**，可以容易地获得交换操作的上界。交换操作是指直接交换 $x_i$ 和 $x_j$ 的在编序中的位置，可以将交换操作分解为一系列的向前迁移或向后迁移。此外，相邻两变量的交换是交换操作的特例。因此，可以获得如下结论：

对相邻两变量的交换，有

$$\frac{|G|}{2} \leqslant |G^*| \leqslant 2 \cdot |G|$$

对任意两变量的交换，有

$$(|G|)^{\frac{1}{2}} \leqslant |G^*| \leqslant |G^2|$$

对向前迁移，有

$$(|G|)^{\frac{1}{2}} \leqslant |G^*| \leqslant 2 \cdot |G|$$

对向后迁移，有

$$\frac{|G|}{2} \leqslant |G^*| \leqslant |G^2|$$

这里 $G$ 为变换前的 BDD，而 $G^*$ 为相关变换之后的 BDD。这里的下界是考虑相关操作的逆操作来获得的，例如向前迁移的逆操作是向后迁移。

前面的**定理 8.2** 和**定理 8.3** 说明了经过变量的交换和迁移之后，整个 BDD 图的节点的变化。下面将讨论：在经过变量的交换之后，相关层的节点数如何变化。

引入如下的表示。用 $\mathrm{Label}(x_i)$ 表示在 BDD 图中具有判决变量 $x_i$ 的节点数。对 $\{x_1, x_2, \cdots, x_n\}$ 的一个子集 $\varphi$，定义 $\mathrm{Label}(\varphi) = \sum_{x_i \in \varphi} \mathrm{Label}(x_i)$。用 $\mathrm{Label}'(x_i)$ 表示在对 BDD 进行变量交换操作之后，在结果 BDD 图中具有判决变量为 $x_i$ 的节点数。

**定理 8.4**　设 $1 \leqslant i \leqslant n$，若对级 $i$ 和级 $i+1$ 进行交换，即交换变量 $x_i$ 和 $x_{i+1}$，则成立

$$1 \leqslant \mathrm{Label}'(x_i) \leqslant 2 \cdot \mathrm{Label}(x_i)$$

和

$$\frac{1}{2}\mathrm{Label}(x_i) \leqslant \mathrm{Label}'(x_{i+1}) \leqslant \mathrm{Label}(x_i) + \mathrm{Label}(x_{i+1})$$

此外，对所有其他的级保持不变。

**证明**：考虑如下 4 种情况：

(1) $1 \leqslant \mathrm{Label}'(x_i)$ 成立。由假定，BDD 所表示的布尔函数 $f$ 依赖于变量 $x_i$，因此在判决变量为 $x_i$ 的这一级，至少存在一个节点。

(2) $\mathrm{Label}'(x_i) \leqslant 2 \cdot \mathrm{Label}(x_i)$ 成立。这是因为：在级 $i$ 的节点依赖于 $x_{i+1}$ 的每一个节

点最后通向两个节点，这两个节点中的每一个对应于 $x_{i+1}$ 的两种可能的取值。如果在级 $i$ 的节点都不依赖于 $x_{i+1}$，则节点总数不变。

(3) $\frac{1}{2}\text{Label}(x_i) \leqslant \text{Label}'(x_{i+1})$ 这个不等式是在上面不等式 $\text{Label}'(x_i) \leqslant 2 \cdot \text{Label}(x_i)$ 的逆叙述。即对两级分别进行两次交换可导致得到原来的图表示。

(4) $\text{Label}'(x_{i-1}) \leqslant \text{Label}(x_i) + \text{Label}(x_{i+1})$ 成立。这是因为在级 $i$ 中所表示的展开因子的数目不能大于已经在级 $i$ 和级 $i+1$ 中所表示的展开因子的数目。对在 BDD 的上面部分的所有变量而言，在 BDD 中的每一个节点代表了一个展开因子，这个数目不由于一个局部操作而改变。

定理证毕。

通过重复使用**定理 8.4** 以及前面的**定理 8.2** 和**定理 8.3**，可以获得如下的两个结论：

**定理 8.5** 设 $1 \leqslant i < j \leqslant n$，若从级 $i$ 向级 $j$ 迁移，即 jump-down$(i,j)$，得到如下的编序 $x_1, x_2, \cdots, x_{i-1}, x_{i+1}, \cdots, x_{j-1}, x_j, x_i, x_{j+1}, \cdots, x_n$。把结果 BDD 命名为 $G'$。令 $A = \{x_1, x_2, \cdots, x_{i-1}\}$，$B = \{x_{i+1}, \cdots, x_{j-1}, x_j\}$，$C = \{x_j, \cdots, x_n\}$。则有

$$|G'| \geqslant \text{Label}(A) + 1 + (1/2)\text{Label}(B) + \text{Label}(C)$$

$$|G'| \geqslant \text{Label}(A) + 1 + \text{Label}(x_i) + \text{Label}(C)$$

**定理 8.6** 设 $1 \leqslant j \leqslant i \leqslant n$，若级 $i$ 向级 $j$ 迁移，即 jump-up$(i,j)$，得到如下的编序 $x_1, x_2, \cdots, x_{j-1}, x_i, x_j, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n$。把结果 BDD 命名为 $G'$。令 $A = \{x_1, x_2, \cdots, x_{j-1}\}$，$B = \{x_{j-1}, \cdots, x_{i-1}\}$，$C = \{x_{i+1}, \cdots, x_n\}$，则有

$$|G'| \geqslant \text{Label}(A) + (i-j) + \frac{1}{2^{i-j}}\text{Label}(x_i) + \text{Label}(C)$$

### 8.1.2 使用低界值的过滤法

在过滤法的进行过程中，为了找到变量的最佳位置，变量移动时穿过了整个 BDD。在变量的移动过程中，如果一个变量在某一方向的移动，不能使 BDD 的规模减少，是否可以尽可能早地停止该变量在这一方向上的移动以降低算法的时间复杂性？

若使用低界信息[34]，就可以做到这一点。可以获得如下的 BDD 规模的低界值信息：若 BDD 的上半部分保持不变，但 BDD 的下半部分的编序可以任意改变，这时 BDD 的规模或 BDD 的某几层的节点数都会大于一个值，该值就称为低界值。

为了决定变量在某方向的移动是否仍是需要的，即这种移动可以降低 BDD 的规模，使用在上节中关于低界的一些结果。

对一个变量向下移动的情况，jump-down$(i,j)$，有

$$L_b(x_i) = \min_{j=1,\cdots,i-1} |G'_j| \geqslant \text{Label}(A) + \max\{\text{Label}(x_i), 1 + (1/2)\text{Label}(B)\}$$

式中：$A = \{x_1, x_2, \cdots, x_{i-1}\}$，$B = \{x_{i+1}, \cdots, x_n\}$。$L_b$ 表示低界(Lower Bound) 的意思，$G_j'$ 表示将变量 $x_i$ 移动到位置 $j$ 之后的 BDD。若这个低界值是大于以前获得的 BDD 的最小规模，则在此方向的移动不会对 BDD 规模的减少有改变。因此可以停止在此方向的移动。

这种低界值的计算可以局部进行，不必访问 BDD 的全部节点，其实现的流程如下。

```
// 在过滤法中使用低界值信息，设有 n 个变量。
    i=i0;                                   // 对 i 赋初始值 i0;
```

```
Lb=Compute_lower_bound();
best=Size_of_BDD();
while(i<n and Lb≤best)
{   level_exchange(i,i+1);          // 进行级 i 和级 i+1 间的交换
  Lb=Compute_lower_bound();
    If(Size_of_BDD()<best)   best=Size_of_BDD();
    i=i+1;   }
```

这种在过滤法中使用低界技术的原理和前面所述的变量过滤法是相同的，只是它使用了低界信息而使整个算法的速度加快，可以只进行更少次数的变量交换。

在低界 $L_b(x_i)$ 的表达式中，可以将 $1/2$ 一般化为 $1/b,(b \geqslant 2)$。实验结果表明，$b$ 的范围取为 $2 \leqslant b \leqslant 10$ 较合适。在这个范围内，在加快了算法执行的同时，BDD 的规模增加不大。

## 8.2 最优编序

从前面的讨论可知，对于一个布尔函数，变量的编序会影响它对应的 BDD 规模。对一个确定的布尔函数来说，一定存在一种最优编序，使得 BDD 的节点数最少。因此这种最优编序是人们所关心的。如何去寻找这种最优编序以及所需的时间复杂性怎样？如果用试探的方法，穷举 $n$ 个变量的每一种可能的编序，则时间复杂度与 $n!$ 有关，计算复杂性太大。下面介绍一种求最优编序的方法，其时间复杂度为 $O(n^2 \cdot 3^n)$。

在这里，为方便讨论，BDD 的变量编序采用从终节点到根节点的方式。这与通常的从根节点到终节点的方式相反。

对一个布尔函数 $f$，在变量编序 $\pi$ 之下，它的 BDD 表示可记为 BDD($f,\pi$)。把编序 $\pi$ 的分量记为 $\pi[i]$。例如，图 8.1 为函数 $f=x_1x_2+x_3x_4$ 在编序 $x_2<x_1<x_4<x_3$ 下的 BDD。

在图 8.1 中说明了节点的级。根节点 $x_3$ 是处于第四级；节点 $x_4$、节点 $x_1$ 和节点 $x_2$ 分别处于第三级、第二级和第一级。可以把图 8.1 中的判决变量 $x_i$ 直接用整数 $i$ 来表示；并将编序记为 $\pi=<2,1,4,3>$，它的分量分别为 $\pi[1]=2,\pi[2]=1,\pi[3]=4,\pi[4]=3$。

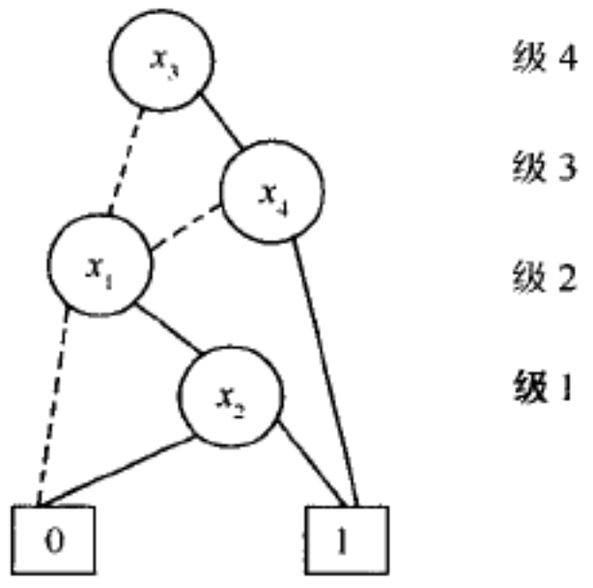



图 8.1 一个 BDD 的例子

下面引入如下的 3 种表示：

(1) 令 $f$ 是一个具有 $n$ 个变量 $x_1,x_2,\cdots,x_n$ 的布尔函数。对任意 $b \in \{0,1\}$ 和任意 $i \in \{1,2,\cdots,n\}$，定义

$$f(x_1,x_2,\cdots,x_n)\mid_{x_i=b} = f(x_1,\cdots,x_{i-1},b,x_{i+1},\cdots,x_n)$$

类似地，对任意多个 $b_1,b_2,\cdots,b_r \in \{0,1\}$，和不同的 $i_1,i_2,\cdots,i_r \in \{1,2,\cdots,n\}$，表达式

$$f\mid_{x_{i1}=b_1,x_{i2}=b_2,\cdots,x_{ir}=b_r}$$

是将 $f$ 中的 $r$ 个变量 $x_{ij}$ 由常数 $b_j$ 所替换而得到，其中 $1 \leqslant j \leqslant r$。

(2) 定义 $I$ 是 $\{1,2,\cdots,n\}$ 的一个子集，即 $I \subseteq \{1,2,\cdots,n\}$。用 $|I|$ 表示集合 $I$ 的元素个数。定义集合 $M(I)=\{\pi \mid \pi$ 是在集合 $\{1,2,\cdots,n\}$ 上的一个编序，且 $\{\pi[1],\pi[2],\cdots,\pi[|I|]\}=I\}$。

这里$M(I)$是$\{1,2,\cdots,n\}$上所有编序的一个子集，$M(I)$中每一编序的前$|I|$个成员构成了集合$I$。

(3) 对一个正整数$v \in \{1,2,\cdots,n\}$和在$\{1,2,\cdots,n\}$上的一个编序$\pi$，下面用$B(f,\pi)$表示逻辑函数$f$在编序$\pi$时所对应的二元判定图；用$C_v(f,\pi)$表示在二元判定图$B(f,\pi)$中判决变量为$v$的节点数。

因此，最优编序的计算归结为：寻找一个编序$\pi$，使$\sum_{v=1}^{n} C_v(f,\pi)$的值最小，即

$$\min \sum_{v=1}^{n} C_v(f,\pi)$$

在讨论求最优编序的算法之前，先给出如下引理。

**引理 8.1** 令$I \in \{1,2,\cdots,n\}$，$v \in I$，$k = |I|$，则存在一个常数$C$，对$M(I)$中满足$\pi[k] = v$的编序$\pi$成立$C_v(f,\pi) = C$。

**证明**：设$\pi \in M(I)$且$\pi[k] = v$成立。假定$J = \{i_1, i_2, \cdots, i_{n-k}\} = \{1,2,\cdots,n\} - I$，则对每一个矢量$\bar{b} = (b_1, b_2, \cdots, b_{n-k}) \in \{0,1\}^{n-k}$，在$B(f,\pi)$中必定出现一个节点，该节点表示了约束函数，即

$$f_{\bar{b}} = f|_{x_{i_1}=b_1, x_{i_2}=b_2, \cdots, x_{i_r}=b_r}$$

由于集合$J$依赖于集合$I$，因此集合$S = \{f_{\bar{b}} \mid \bar{b} \in (\{0,1\}^{n-k}\}$对所有$\pi \in M(I)$都为常数。

而且，在$B(f,\pi)$中存在唯一的一个节点与集合$S$中的每一个元素对应。这是因为在约简 BDD 中使用了“同构子图的合并规则”。这样，与给定的一个$f_b$对应的节点是$B(f_b, \pi')$的根节点，这里$\pi' = <\pi[1], \pi[2], \cdots, \pi[k]>$。显然，如果一个判决变量为$v$的节点在二元判定图中出现，它必定是根节点。特别地，由于在约简图 BDD 中使用了“删除冗余节点”的规则，因此仅仅是判决变量为$v$的这些节点才与$S$中的与$x_v$有关的函数对应。

因此，对任意$\pi \in M(I)$，判决变量为$v$的节点数等于$S$中与$x_v$有关的函数的个数。由于这些函数的个数为常数$C$，这样$C_v(f,\pi) = C$。

引理证毕。

这个**引理 8.1** 的意义在于：在进行变量编序时，不必进行所有$n!$次交换。例如，若已知$M(\{1,2,3\}) = <2,3,1>$，$M(\{1,2,4\}) = <4,1,2>$，$M(\{1,3,4\}) = <1,4,3>$，$M(\{2,3,4\}) = <4,3,2>$，则$M(\{1,2,3,4\})$必是$<2,3,1,4>$、$<4,1,2,3>$、$<1,4,3,2>$或$<4,3,2,1>$之一。也就是说，可以使用在 3 个变量时最优编序的知识去指导在 4 个变量时求最佳编序的过程，而不必进行$4! = 24$种变量变换。

下面是在**引理 8.1** 的基础上设计的一个求最优编序的算法。

```
// 计算最优编序的方法
for  k:=1  to  n  do
for each I do          // 这里 I 是有 k 个元素的{1,2,…,n}的子集 I,I⊆{1,2,…,n}。
                       // 计算 π_I,TABLE_I 和 MinCost_I。
  begin
    MinCost_I:=∞;
    for  each  v∈I  do
      begin
```

```
                    // 在编序<π_{I-{v}} , v>  下计算 Cost_I的值。
对每一对(t_0 , t_1),使 id(t_0 , t_1)=0;
count:=MinCost_{I-{v}} ;
用 i_1 , i_2 , …, i_{n-k}表示集合{1,2,…,n}-I 中的元素;
for each b̄∈{0,1}^{n-k}  do
  begin
      t_0 :=Table_{I-{v}} (x_{i_1} =b_1 , …, x_{i_{n-k}} =b_{n-k} , x_v =0);
      t_1 :=Table_{I-{v}} (x_{i_1} =b_1 , …, x_{i_{n-k}} =b_{n-k} , x_v =1);
      IF  t_0=t_1  THEN  TempTable(x_{i_1} =b_1 , …, x_{i_{n-k}} =b_{n-k})=t_0 ;
      else begin
          IF id(t_0 , t_1)=0  then begin      // 这一对(t_0 , t_1)是新的值。
              count:=count+1;
              id (t_0 , t_1)=count;  end;
          TempTable(x_{i_1} =b_1 , …, x_{i_{n-K}} =b_{n-k})=id(t_0 , t_1); end;
      end;
    if count<MinCost_I then begin
      MinCost_I=count;  π_I=<v, π_{I-{v}}>;
 Table_I=TempTable;  end;
 end;
end;
Return π_{{1,2,…,n}}.
```

在算法中，对每一个判决变量集合 $I \in \{1,2,\cdots,n\}$，按这些集合中元素个数 $k = |I|$ 递增的次序进行处理。对每个集合 $I$，在算法中计算了如下 3 种参数的值：

(1) 参数 $\mathrm{MinCost}_I$ 用于表示对所有 $\pi \in M(I)$，表达式 $\sum_{v \in I} C_v(f,\pi)$ 的最小值。算法开始时，设置 $\mathrm{MinCost}_{\varnothing} = 0$，这里 $\varnothing$ 代表空集。

(2) $\pi_I$ 是 $M(I)$ 中达到最小值 $\mathrm{MinCost}_I$ 的这种编序中的一种编序。

从如上的**引理 8.1** 不难得到：BDD 图的前 $k$ 级的节点仅仅依赖于这 $k$ 级的变量编序，而不依赖于其他 $n-k$ 个变量的编序。特别地，使得 $\pi[i] = \pi_I[i]$（这里 $1 \leqslant i \leqslant k$）的每一个编序 $\pi$，都满足等式 $\sum_{v \in I} C_v(f,\pi) = \mathrm{MinCost}_I$。

开始时，$\pi_{\varnothing}$ 为空序列，即 $\pi_{\varnothing} = <>$。这样，最优编序问题就成为：寻找编序 $\pi_{\{1,2,\cdots,n\}}$，使得在此编序下的二元判定图有 $\mathrm{MinCost}_{\{1,2,\cdots,n\}}$ 个非终节点。

(3) $\mathrm{Table}_I$ 是一个真值表，该真值表说明从 $\{0,1\}^{n-k}$ 到 $B(f,\pi)$ 的所有节点的映射关系，这些节点是终端节点和由集合 $I$ 中的元素构成的判决变量所对应的非终端节点。注意，由于这时存在 $\mathrm{MinCost}_I$ 个这种非终节点，因此在此真值表中一定有 $\mathrm{MinCost}_I + 2$ 个不同的值。用 1 至 $\mathrm{MinCost}_I$ 之间的整数来标识这些非终节点。下面对这个映射的详细实现作如下说明：

在 $\{0,1\}^{n-k}$ 中的每一个元素 $\bar{b} = (b_1, b_2, \cdots, b_{n-k})$ 代表了对具有下标 $i_1, i_2, \cdots, i_{n-k} \in I$

的变量的一种真值分配。在映射的作用下，使元素 $\bar{b}$ 与函数 $f\mid_{x_{i_1}=b_1,x_{i_2}=b_2,\cdots,x_{i_{n-k}}=b_{n-k}}$ 的 $B(f,\pi_I)$ 的这个节点相对应。这样，初始化时，$\text{Table}_{\varnothing}$ 是从 $\{0,1\}^n$ 到 $\{0,1\}$ 的 $f$ 的真值表。

为了计算参数 $\text{MinCost}_I$、$\pi_I$ 和 $\text{Table}_I$，考察每个 $v\in I$，并对满足 $\pi[k]=v$ 的一些编序 $\pi\in M(I)$ 计算 $C_v(f,\pi)$。这里即是考察在位置 $k$ 上出现变量 $v$，而在较低位置出现 $I$ 的其他变量的这样一些编序。由**引理 8.1** 的结论可知，不论使用这种编序中的哪一种编序 $\pi$，这时 $C_v(f,\pi)$ 将是相同的。由于已计算出了 $\text{Table}_{I-\{v\}}$，因此，为方便，使用编序 $<v,\pi_{I-\{v\}}>$ 作为这里的 $\pi$。

特别地，为了计算 $C_v(f,<v,\pi_{I-\{v\}}>)$，这里对真值表 $\text{Table}_{I-\{v\}}$ 进行了一次“对折”操作。“对折”的意思是把真值表划分为长度相等的两部分，然后比较这两部分相关的对应元素，而获得另外一个只有当前真值表的行数 1/2 的真值表。获得的这种真值表在存储时取名为 TempTable。

如果 $i_1,i_2,\cdots,i_{n-k}$ 是集合 $\{1,2,\cdots,n\}-I$ 中的元素，则对每一个长度为 $n-k$ 的矢量 $\bar{b}$，先计算 $\text{Table}_{I-\{v\}}(x_{i_1}=b_1,\cdots,x_{i_{n-k}}=b_{n-k},x_v=0)$，把此值以参数 $t_0$ 来存储。然后计算 $\text{Table}_{I-\{v\}}(x_{i_1}=b_1,\cdots,x_{i_{n-k}}=b_{n-k},x_v=1)$，把此值以参数 $t_1$ 来存储。通过用这种表示，就可以说明通过对变量 $x_{i_j}$ 分配值 $b_j(j=1,2,\cdots,n-k)$ 和对变量 $x_v$ 分配值 0，能够获得 $\text{Table}_{I-\{v\}}$ 的值。换句话说，这给了约束函数的一种表示即

$$f\mid x_{i_1}=b_1,\cdots,x_{i_{n-k}}=b_{n-k},x_v=0$$

对每一个这种对 $(t_0,t_1)$，使用如下的 3 条规则去决定在 $B(f,<v,\pi_{I-\{v\}}>)$ 中是否需要生成一个判决变量为 $v$ 的新节点：

(1) 若 $t_0=t_1$，则不生成新节点。因为它的左孩子和右孩子指针指向同一个节点。

(2) 设在BDD图中由以 $t_0$ 作为左孩子和以 $t_1$ 作为右孩子的节点为 $\text{id}(t_0,t_1)$。若 $\text{id}(t_0,t_1)$ 存在，则说明在图中已有以判决变量为 $v$ 的节点，这时将不生成新节点。因为若生成新节点的话，该节点的左右孩子将与图中已有的判决变量为 $v$ 的节点的左右孩子相同。

(3) 若 $\text{id}(t_0,t_1)$ 不存在，这时将产生一个新节点。

这里，对如上的第一种情况，把 $t_0$ 分配给 $\text{TempTable}(x_{i1}=b_1,\cdots,x_{i_{n-k}}=b_{n-k})$；对第二、第三种情况则把 $\text{id}(t_0,t_1)$ 分配给它。

在算法进行了“对折”操作之后，即是在对每一个矢量 $\bar{b}$ 的循环处理完成之后，就计算

$$\text{count}=\sum_{v\in I}\text{Cost}_v(f,<v,\pi_{I-\{v\}}>)$$

若这个 count 的新值小于以前所获得的 count 的最小值，则将这个 count 的新值作为 $\text{MinCost}_I$ 的值保存。此外，也将 $\pi_I$ 的值替换为 $<v,\pi_{I-\{v\}}>$；将 $\text{Table}_I$ 的值替换为 TempTable。

在算法中，对每一个 $v$ 用如上方法进行处理，并把 $\min\limits_{v\in I}(\text{Cost}_v+\text{MinCost}_{I-\{v\}})$ 的值赋给 $\text{MinCost}_I$，并且让 $\pi_I$ 是取得这种最小值的编序 $<v,\pi_{I-\{v\}}>$；让 $\text{Table}_I$ 是与二元判定图 $B(f,\pi_I)$ 所对应的真值表。

下面分析如上的最优编序方法的时间复杂性与空间复杂性。首先分析时间复杂性。

对每一次 $k$ 循环 $(k=1,2,\cdots,n)$ 处理 $C_n^k$ 个集合 $I$。对集合 $I$ 的每一个 $v$，就对 $\text{Table}_{I-\{v\}}$ 进行一次“对折”操作。处理 $2^{n-k}$ 个真值表的表项中的每一项需要对 $\text{Table}_{I-\{v\}}$ 进行两次查找和对 id 的一次查找。在 $\text{Table}_{I-\{v\}}$ 中查找的时间为 $O(n-k+1)$。

可以将 id 作为平衡树来完成，且进行插入和查找的时间与真值表的最大项数的对数成正比。这样，简单地初始化 id 为空集，以取代将 id 初始化为 0，这由算法中注有（*）号的这一行来完成。由于对每一个 $\bar{b} \in \{0,1\}^{n-k}$，最多有一次会生成 id，因此对 id 的插入和查找操作能在 $O(\log_2(2^{n-k+1})) = O(n-k+1)$ 内完成。故算法的总的时间复杂性的阶为

$$\sum_{k=1}^{n} C_n^k \cdot k \cdot 2^{n-k}(n-k+1)$$

对上式进行化简，可得时间复杂性为 $O(n^2 3^n)$。化简的具体过程如下：

$$\begin{aligned}
\sum_{k=1}^{n} C_n^k \cdot k \cdot 2^{n-k+1}(n-k+1) &= \sum_{k=1}^{n} C_n^k \cdot k \cdot 2^{n-k} \cdot (n-k) + \sum_{k=1}^{n} C_n^k \cdot k \cdot 2^{n-k} \\
&= \sum_{k=1}^{n} n(n-1) \cdot C_{n-2}^{k-1} \cdot 2^n \cdot \left(\frac{1}{2}\right)^k + \sum_{k=1}^{n} n \cdot C_{n-1}^{k-1} \cdot 2^n \cdot \left(\frac{1}{2}\right)^k \\
&= n(n-1) \cdot 2^{n-1} \sum_{k=0}^{n-1} C_{n-2}^k \cdot \left(\frac{1}{2}\right)^k + n \cdot 2^{n-1} \cdot \sum_{k=0}^{n-1} C_{n-1}^k \cdot \left(\frac{1}{2}\right)^k \\
&= n(n-1) \cdot 2^{n-1} \cdot \left(1+\frac{1}{2}\right)^{n-2} + n \cdot 2^{n-1} \cdot \left(1+\frac{1}{2}\right)^{n-1} \\
&= 2(n^2-n) \cdot 3^{n-2} + n \cdot 3^{n-1} = O(n^2 \cdot 3^n)
\end{aligned}$$

下面分析最优编序算法的空间复杂性。首先，在主循环对 $k$ 的迭代中，对 Table 表的查找是在以前（上一次 $k-1$ 时）所得的结果中进行。因此，在任何时刻，仅需要存储相继的两次迭代的 Table 的值。实际上，由于采用的是有序的表示，在每一次迭代中实质上只需要存储一次，因此存储空间要求为

$$\max_{0 \leqslant k \leqslant n}(C_n^k \cdot 2^{n-k})$$

下面说明当 $k = \lfloor n/3 \rfloor$ 时值为最大，因此算法的空间复杂性为 $O(3^n/\sqrt{n})$。

令 $f(k) = C_n^k \cdot 2^{n-k}$。对所有满足 $1 \leqslant k \leqslant n-1$ 的整数 $k$，如下不等式成立，即

$$\frac{f(k+1)}{f(k)} < \frac{f(k)}{f(k-1)}$$

这是因为

$$\begin{gathered}
0 < n+1 \quad \Leftrightarrow \quad (n-k) \cdot k < (k+1) \cdot (n-k+1) \quad \Leftrightarrow \\
\frac{n! \cdot 2^{n-k-1}}{(k+1)! \cdot (n-k-1)!} \cdot \frac{n! \cdot 2^{n-k+1}}{(k-1)! \cdot (n-k+1)!} < \\
\frac{n! \cdot 2^{n-k}}{k! \cdot (n-k)!} \cdot \frac{n! \cdot 2^{n-k}}{k! \cdot (n-k)!} \quad \Leftrightarrow \\
f(k+1) \cdot f(k-1) < f(k) \cdot f(k) \quad \Leftrightarrow \\
\frac{f(k+1)}{f(k)} < \frac{f(k)}{f(k-1)}
\end{gathered}$$

这个不等式说明，随着 $k$ 的增加，$f(k)$ 和 $f(k-1)$ 之比会减少。而且有 $\frac{f(1)}{f(0)} = \frac{n}{2}$，$\frac{f(n)}{f(n-2)} = \frac{1}{2n}$。因此随着 $k$ 从 1 到 $n$ 变化，$\frac{f(k)}{f(k-1)}$ 开始时大于 1，但随之逐渐减小，最后小于 1，即随着 $f(k)$ 的增加，当 $\frac{f(k)}{f(k-1)} > 1 \geqslant \frac{f(k+1)}{f(k)}$ 时可达到最大值。因此，要找出最大的 $k$，使得 $f(k) > f(k-1)$，即使得

$$\frac{n! \cdot 2^{n-k}}{k! \cdot (n-k)!} > \frac{n! \cdot 2^{n-k+1}}{(k-1)! \cdot (n-k+1)!}$$

可得$\frac{1}{k} > \frac{2}{n-k+1}$，有 $n \geqslant 3k$。

因此，$f(k)$ 的最大值在 $k = \lfloor n/3 \rfloor$ 处取得。

下面计算 $f(\lfloor n/3 \rfloor)$。为简单起见，假定 $n$ 是 3 的倍数。实质上，若 $n$ 不是 3 的倍数时，可以增加 1 个或 2 个变量到所考虑的布尔函数中。这并不会影响整个算法的复杂性。因此有

$$f(\frac{n}{3}) = C_n^{n/3} \cdot 2^{2n/3} = \frac{n! \cdot 2^{2n/3}}{(n/3)! \cdot (2n/3)!}$$

对 $f(\frac{n}{3})$ 中的 $n!$ 用公式 $n! = O(\sqrt{2\pi n} \cdot (\frac{n}{e})^n)$ 进行化简，可得 $f(n/3) = O(3^n / \sqrt{n})$。

对最优编序算法的时间复杂性，若用其他更有效的方法来处理 id 集合，例如将它用哈希表来存储，则算法的时间复杂性可降低到 $O(n \cdot 3^n)$；空间复杂性也会有降低，但复杂性的阶不变。

## 8.3 ZBDD 的定义与操作

ZBDD(Zero-suppressed BDD)是一种 BDD 的类型，它特别适合于对集合进行表示与操作。

### 8.3.1 ZBDD 的定义与性质

在实际应用中，经常会遇到对集合的处理，例如，集合中的元素为某个组合问题的解。对集合的每一个元素，可以用 $n$ 位二元矢量来表示。这种二元矢量的形式是 $x_1 x_2 \cdots x_n$，每个 $x_k \in \{0,1\}$，$1 \leqslant k \leqslant n$。可以用 $n$ 元布尔函数来表示集合中的元素(二元矢量)。这种函数当输入取集合中的矢量时，其输出为 1，否则输出为 0，称这种函数为特征函数。

可以将特征函数用 BDD 来表示。在这种 BDD 中，称从根节点到终节点 1 的路径为 1- 路径，它表示了属于集合中的所有二元矢量，或者说它表示了集合中的所有元素。

用 ZBDD，可以将多个集合放一个图中来表示。例如图 8.2 给出了集合{1000，0100}和{100，010} 的表示。

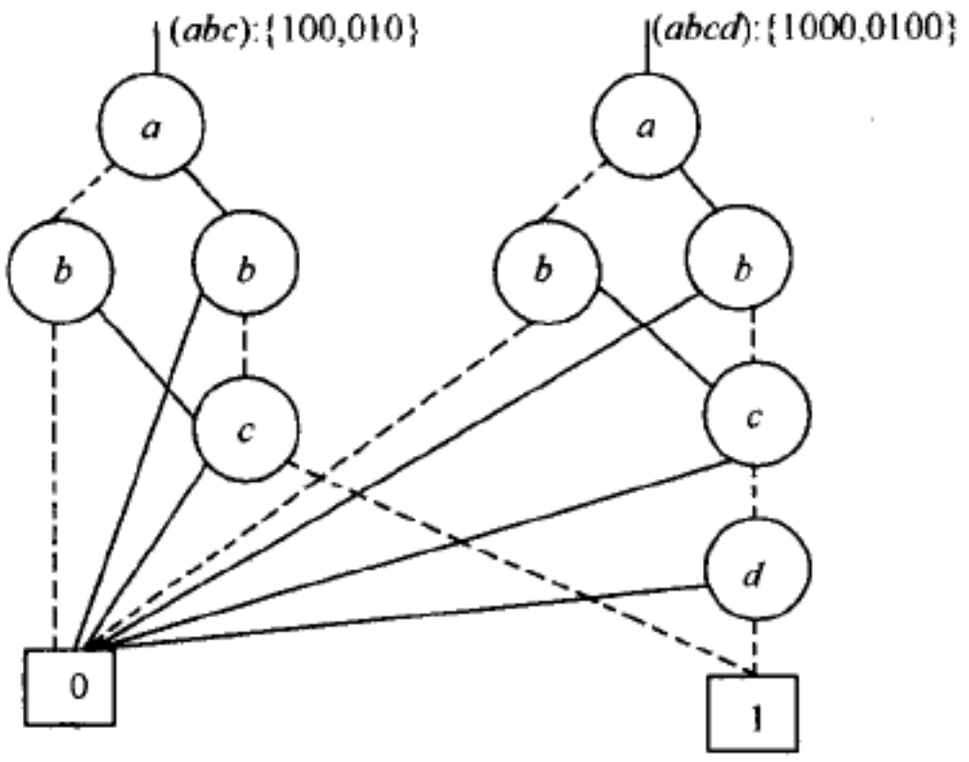



图 8.2　用 BDD 表示多个集合

在实际中，由于各个集合中二元矢量的分量可能代表不同的变量，例如 $(a\ b\ c) = (0$

1 0)，($b\ c\ d$) = (1 1 0)，($a\ b\ d$) = (1 1 1)。这时在表示时就必须先固定变量的编序，然后再构造 BDD 图。从图 5.1 可以看出，可以对图中的一些节点进行合并与约简，得到节点数更少的 BDD 图。

约简时使用如下的两条规则：

**规则 1**　删去 1 边指向 0 终节点的所有节点；使用 0 边的子图，如图 8.3 所示。

**规则 2**　共享所有同构的子图。

在**规则 1** 中，不删去 0 边指向一个终节点的这些节点，同时，也不删去两个边(0 边和 1 边)都指向同一节点的这种节点。这是它和常规 BDD 约简时的不同之处。称采用规则 1 的 BDD 为 ZBDD。如果变量的编序是固定的，则 ZBDD 可唯一地代表一个布尔函数。这是因为一个非约简的二元树在经过应用如上的约简规则之后，可以形成一个 ZBDD。图 8.4 是一种 ZBDD 表示，它代表了图 8.2 中所表示的集合。

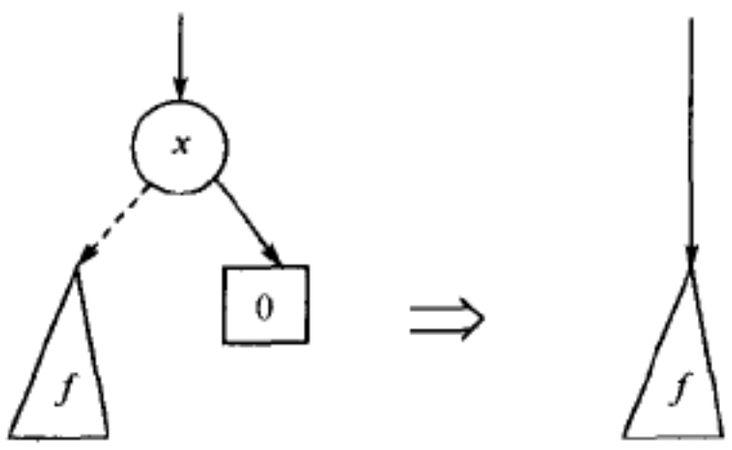



图 8.3　ZBDD 的约简规则 1

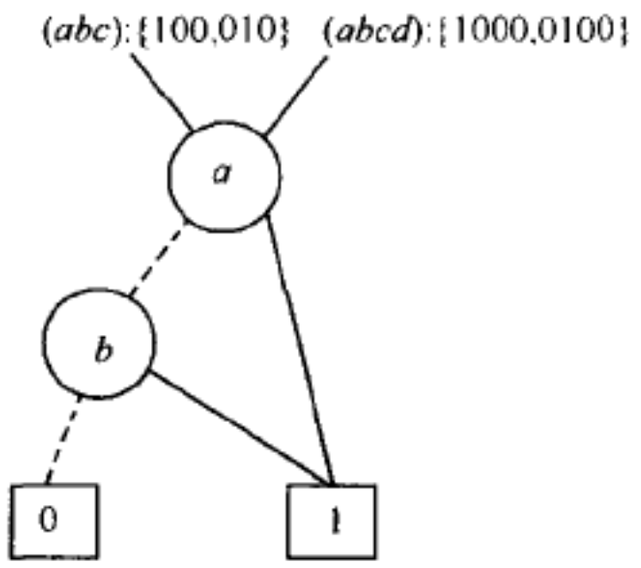



图 8.4　集合的 ZBDD 表示

ZBDD 的一个特征是：只要多个集合的输入变量的顺序一致，则 ZBDD 的结构形式独立于输入变量的个数。因此在生成 ZBDD 之前，不必固定输入变量数。

此外，在 ZBDD 中 1—路径的数目等于在每个集合中的元素个数(二元矢量数)。顺便指出，若用常规的 ROBDD 来表示集合，ROBDD 的约简规则会破坏这个性质，因此 ZBDD比 ROBDD 更适合于表示多个集合。

### 8.3.2　ZBDD 的操作

为了构造复杂集合的 ZBDD，首先定义如下的一些操作，然后把它们应用于一些基本的 ZBDD 图，这些 ZBDD 的操作有：

Empty( )　　返回空集∅；在 ZBDD 中是指返回 0 终节点。

Base( )　　返回全集 $E$；在 ZBDD 中是指返回 1 终节点。

Subset1(P, var)　返回集合 $P$ 的子集，并使得 var = 1。

Subset0(P, var)　返回集合 $P$ 的子集，并使得 var = 0。

Change(P, var)　把 var 的值取反(即 0 变为 1，1 变为 0) 之后，返回集合 $P$。

Union($P$, $Q$)　　求两个集合 $P$ 与 $Q$ 的并，返回($P \cup Q$)。

Intsec($P$, $Q$)　　求两个集合 $P$ 与 $Q$ 的交，返回($P \cap Q$)。

Diff($P$,$Q$)　　求两个集合 $P$ 与 $Q$ 的差，返回 ($P-Q$)。

Count($P$)　　计算集合 $P$ 的元素个数，返回 $|P|$。

这里，使用 Intsec( ) 操作来求两个集合的交，它可以用来说明多个元素的组合是否

包含在一个集合中。此外，在 Base( ) 操作之后使用 Change( ) 操作，可以生成一些元素的任一组合。图 8.5 是使用上面这些操作的一个例子。

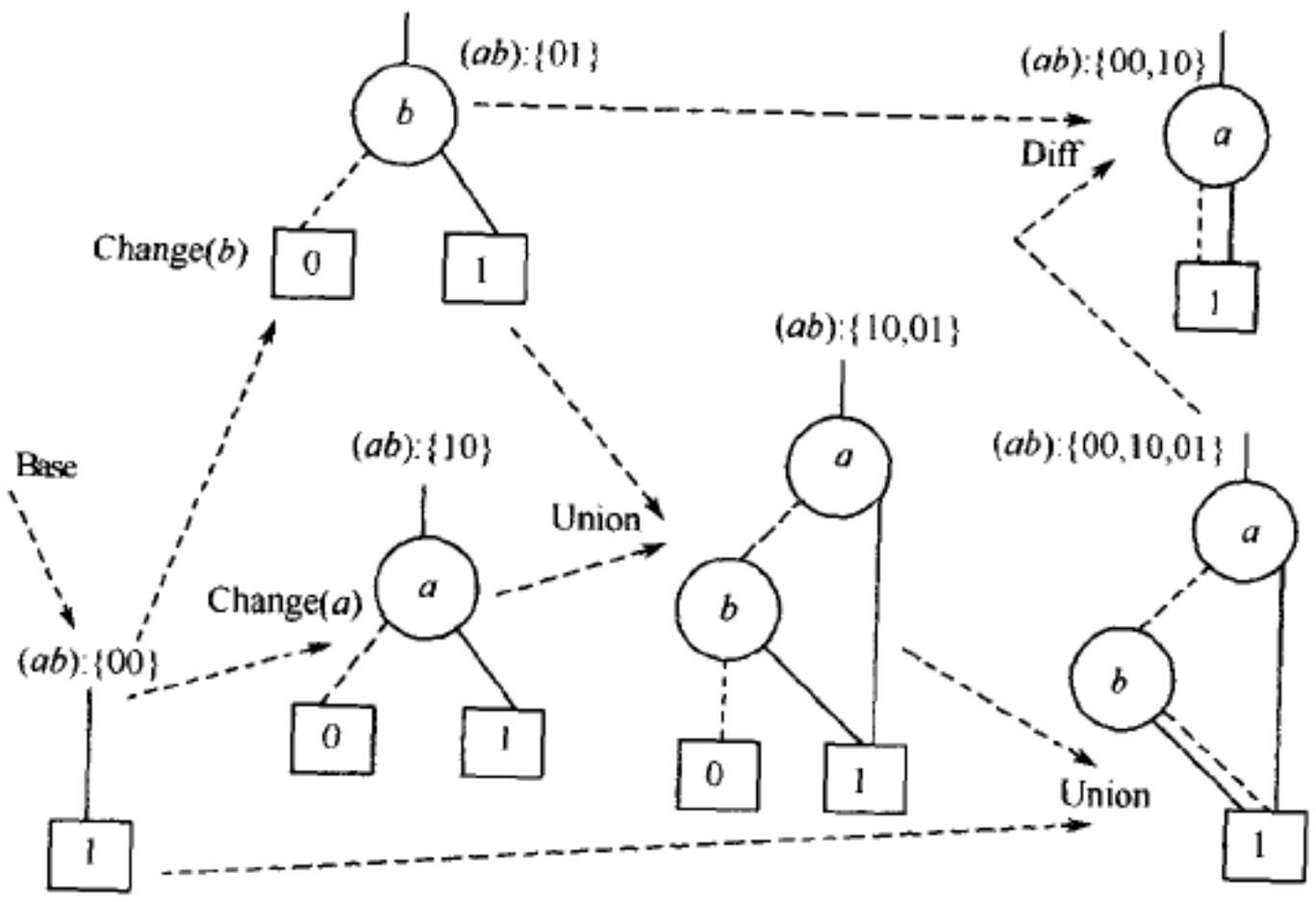



图 8.5　生成 ZBDD 的例子

下面论述 ZBDD 基本操作的实现，这里先定义一个过程 Getnode (top, $P_0$, $P_1$)，ZBDD 的其他操作可通过调用此过程来完成。过程 Getnode( ) 的功能是：对变量 top 和两个子图 $P_0$ 和 $P_1$，生成或复制一个节点。在该过程的实现时，使用了一种哈希表，称为 unique－table，用于维持每个节点的唯一性。节点的删去和共享都由 Getknode( ) 来进行。

```
Getnode (top, P0, P1)
{
    若 P1等于∅,则返回 P0;  // 节点删去
    用(top, P0, P1)在哈希表(unique-table)中查找,看是否有相应节点存在;
    若已存在,则返回该节点;  // 节点共享
        若不存在,则用(top, P0, P1)生成一个新节点 P;
    将新节点 P 增加到哈希表 unique-table 中;
    返回 P;
}
```

下面用过程 Getnode( ) 去描述 ZBDD 的一些基本操作，如 Count( ), Subset0( ), Subset1( ), Change( ), Union( ), Intsec( ), Diff( )等。这里用 P. top 代表具有最高编序的变量；而 $P_0$ 和 $P_1$ 是它的两个子图。

```
Count(P)
{   若 P= =∅,则返回 0;
    若 P= ={0},则返回 1;
    若 P= ={1},则返回 1;
    返回 Count(P0)+Count(P1);}
Subset1(P, var)
{   若 P. top<var,则返回∅;
```

若 P. top= = var,则返回 $P_1$;

若 P. top>var,则返回 Getnode( P. top, Subset1($P_0$, var), Subset1($P_1$, var)); }

Subset0(P, var)

{ 若 P. top<var,则返回 P;

若 P. top = =var,则返回 $P_0$;

若 P. top>var,则返回 Getnode(P. top,Subset0($P_0$,var),Subset0($P_1$,var)); }

Change(P,var)

{ 若 P. top<var,则返回 Getnode(var,∅,P);

若 P. top = = var,则返回 Getnode(var, $P_1$, $P_0$);

若 P. top>var,则返回 Getnode( P. top,Change($P_0$,var),Change($P_1$,var)); }

Union(P,Q)

{ 若 P = =∅,则返回 Q;

若 Q= =∅,则返回 P;

若 P= =Q,则返回 P;

若 P. top>Q. top,则返回 Getnode(P. top,Union($P_0$,Q), $P_1$);

若 P. top<Q. top,则返回 Getnode(Q. top,Union(P,$Q_0$),$Q_1$);

若 P. top= =Q. top,则返回 Getnode(P. top,Union($P_0$,$Q_0$),Union($P_1$,$Q_1$)); }

Intsec(P, Q)

{ 若 P= =∅,则返回∅;

若 Q= =∅,则返回∅;

若 P= =Q,则返回 P;

若 P. top>Q. top,则返回 Intsec($P_0$, Q);

若 P. top<Q. top,则返回 Intsec(P, $Q_0$);

若 P. top= =Q. top,则返回 Getnode(P. top, Intsec($P_0$, $Q_0$), Intsec($P_1$, $Q_1$)); }

Diff(P, Q)

{ 若 P= =∅,则返回∅;

若 Q= =∅,则返回 P;

若 P= =Q,则返回∅;

若 P. top>Q. top,则返回 Getnode( P. top, Diff($P_0$, Q), $P_1$);

若 P. top<Q. top,则返回 Diff( P, $Q_0$);

若 P. top= =Q. top,则返回 Getnode( P. top, Diff($P_0$, $Q_0$), Diff($P_1$, $Q_1$)); }

在最差的情况下,这些算法的时间复杂度与变量的数目成指数关系,但可以通过使用高速缓冲区来存储最近操作的计算结果,以降低算法的复杂性。在每一次递归调用之前,通过对缓冲区的查询,能避免对同构子图的重复操作。使用了这种措施之后,可以使得执行如上操作的时间粗略地与图的规模(节点数)成正比。

### 8.3.3 ZBDD 在布尔立方集代数中的应用

在本 8.3.3 小节中所论述的布尔立方集,只允许出现正变量,如 $x_1$、$x_2$、$x_3$,不允许出

现补变量，如 $\bar{x}_1$、$\bar{x}_2$ 等，因此这种立方集也称为非补立方集。

对通常的二元立方集，$x$ 和 $\bar{x}$ 分别表示 $x=1$ 和 $x=0$。当变量 $x$ 不出现时，则意味着 $x$ 的取值处于不关心(随意)。即 $x$ 可以取 3 种值：0，1 或 *，这里 * 代表不关心。

而对非补立方集，变量 $x$ 在立方集中出现则意味着 $x=1$。$x$ 不出现则意味着 $x=0$。因此 $x$ 的取值只有 0，1 这两种情况。

例如，二元立方集表达($a+bc$)代表了如下 5 个矢量 {111，110，101，100，011}。而非补立方集($a+bc$)则只代表两个矢量{100，011}。这里的矢量的分量形式都为 ($a\ b\ c$)。

1. 非补立方集的基本操作

在非补立方集中有一些基本集，如空集(用 0 表示)；单位集(用 1 表示)；单变量集(用 $x_k$ 表示)。单位集“1”仅由一个元素组成，该元素是进行积操作时的单位元，它不是任意变量的立方集；单变量集是仅由一个变量构成的立方所组成的集。在本节的后面，用大写字母代表一个表达式，用小写字母来代表变量。非补立方集的主要操作有：&(交)、+(并)、—(差)、*(积)、/(商)、%(求余)。其中积“*”生成所有与给定的两个立方集有关的子立方集。例如：

$$\{ab,\ b,\ c\}\&\{ab,\ 1\} = \{\ ab\ \}$$

$$\{ab,\ b,\ c\}+\{ab,\ 1\} = \{ab,\ b,\ c,\ 1\}$$

$$\{ab,\ b,\ c\}-\{ab,1\} = \{b,\ c\}$$

$$\{ab,\ b,\ c\} * \{ab\ ,\ 1\} = (ab * ab) + (ab * 1) + (b * ab) + (b * 1) + (c * ab) + (c * 1)$$

$$= \{ab,\ abc,\ b,\ c\}$$

对非补立方集的操作，有如下的一些计算公式：

$$P + P = P$$

$a * a = a$($a$ 指一个变量，但一般地 $P * P = P$ 不成立)

$(P-Q) = (Q-P)$ 的充要条件是 $P = Q$

$$P * (Q+R) = (P * Q) + (P * R\ )$$

对求余操作 % 做如下讨论。对 $P/Q$(求商) 和 $P\%Q$(求余数)，这时成立：$P = Q * (P/Q) + (P\%Q)$。一般地，满足此等式的商和余数并不唯一，在此可以使用如下的方法使它们唯一。

(1) 当 $Q$ 仅包含一个立方集时，则 $P/Q$ 是通过选取 $P$ 的一个子集来得到，该子集是由包含 $Q$ 的所有变量的一些立方集，并从这些立方集中删去 $Q$ 的变量构成的。例如

$$\{abc,\ bc,\ ac\ \}\ /\ \{bc\} = \{a,1\}$$

(2) 当 $Q$ 包含多个立方集时，则 $P/Q$ 是 $P$ 被 $Q$ 中每一个立方集除之后所得商的交。例如：

$$\{abd,\ abe,\ abg,\ cd,\ ce,\ ch\}\ /\ \{ab,\ c\}$$

$$= (\{abd,\ abe,\ abg,\ cd,\ ce,\ ch\ \}\ /\ \{ab\}\ )\ \&$$

$$(\{abd,\ abe,\ abg,\ cd,\ ce,\ ch\ \}\ /\ \{c\}\ )$$

$$= \{d, e, g\} \& \{d, e, h\} = \{d, e\}$$

(3) $P\%Q$ 可以由如下公式计算：$P\%Q = P - Q * P/Q$

2. 非补立方集操作的 ZBDD 实现

使用 ZBDD 可以有效地完成对非补立方集的操作，方法如下。首先对 3 种基本的立方集，用结构简单的 ZBDD 来表示。空集“0”对应为 0 终节点，单位集“1”对应为 1 终节点，单变量集 $x_k$ 对应为直接指向 0 终节点或 1 终节点的单节点图。

非补立方集操作中的“交”、“并”、“差”操作的实现与 8.3.2 节的方法类似。这里主要论述对“积”、“商”、“求余数”操作的实现。

对一个表达式，若通过逐项处理每一个立方集的方法来计算“积”和“商”，则计算时间会依赖于表达式的长度。这种方法在处理非常多的立方集时是不实用的，为此，可使用如下的递归算法。

该算法是基于分而治之的原理。设 $x$ 是编序最高的变量，将 $P$ 和 $Q$ 分解成如下两部分：

$$P = x * P_1 + P_0, \qquad Q = x * Q_1 + Q_0$$

积 $P * Q$ 可写为

$$P * Q = x * (P_1 * Q_1 + P_1 * Q_0 + P_0 * Q_1) + P_0 * Q_0$$

对每一个子积项可以进行递归调用，直至达到了基本集，最后可得到 $P * Q$ 的结果。

在最坏的情况下，该方法与递归调用次数的指数次幂有关，因此复杂性较大。但若采用哈希表和用缓冲来存储最近的操作，且在每次递归调用之前查询哈希表，则计算时间仅依赖于 ZBDD 的规模，而不依赖于立方集和变量的数目。

“积”($P * Q$) 操作的实现步骤为：

(1) 若 $P.top < Q.top$，则返回 $Q * P$。

(2) 若 $Q = 0$，则返回 0。

(3) 若 $Q = 1$，则返回 $P$。

(4) 若 $R$ 已存在，这时 $R$ 为该缓存中的 $P * Q$，则返回 $R$；否则 $x \leftarrow P.top$。这里 $P.top$ 为编序最高的变量。

(5) 由 $P$ 和变量 $x$，计算 $P$ 的展开因子($P_0$，$P_1$)。

(6) 由 $Q$ 和变量 $x$，计算 $Q$ 的展开因子($Q_0$，$Q_1$)。

(7) $R \leftarrow x(P_1 * Q_1 + P_1 * Q_0 + P_0 * Q_1) + P_0 * Q_0$。

(8) 将 $R$ 的值赋给缓存区，作为 $P * Q$ 的值。

(9) 返回 $R$。

“商”($P/Q$) 的操作与“积”操作的实现类似，其过程如下：

(1) 若 $Q = 1$，则返回 $P$。

(2) 若 $P = 0$ 或 $P = 1$，则返回 0。

(3) 若 $P = Q$，则返回 1。

(4) 若 $R$ 已存在，这时 $R$ 为缓存中的 $P/Q$，则返回 $R$；否则，$x \leftarrow Q.top$。这里 $Q.top$ 为编序最高的变量。

(5) 由 $P$ 和变量 $x$，计算 $P$ 的展开因子($P_0$，$P_1$)。

(6) 由 $Q$ 和变量 $x$，计算 $Q$ 的展开因子($Q_0$，$Q_1$)。

(7)$R \leftarrow P_1/Q_1$。

(8) 若 $R \neq 0$ 且 $Q_0 \neq 0$，则 $R \leftarrow R \,\&\, (P_0/Q_0)$。

(9) 将 $R$ 的值赋给缓存，作为 $P/Q$ 的值。

(10) 返回 $R$。

这里 $x$ 是 $Q$ 的根节点对应的变量，$P_0$、$P_1$、$Q_0$ 和 $Q_1$ 分别是 $P$、$Q$ 与 $x$ 所展开的子立方。这里由于 $x$ 在 $Q$ 中出现，因此 $Q_1 \neq 0$，商 $P/Q$ 可以写成：

若 $Q_0 = 0$，则 $P/Q = P_1/Q_1$；否则 $P/Q = (P_1/Q_1) \,\&\, (P_0/Q_0)$。

每一个子“商”可以被递归计算。在进行过程中，若出现子“商”($P_1/Q_1$) 或($P_0/Q_0$) 等于 0，则($P/Q$) $= 0$，计算过程被停止。

在商 $P/Q$ 计算出来之后，利用 $P\%Q = P - P * (P/Q)$，可以计算出余数($P\%Q$)。

下面以“八皇后”问题为例说明 ZBDD 在非补立方集的应用。“八皇后”问题是一个典型的组合优化问题。对此，先将该问题模型化。

首先，为棋盘上的 64 个方格分配 64 个逻辑布尔变量 $x_{ij}$，每一个变量表示皇后是否在该方格，$i$，$j = 1,2,\cdots,8$。八皇后问题可以描述为：

在一个特定的列，仅有一个变量的值为 1；

在一个特定的行，仅有一个变量的值为 1；

在一个特定的对角线，有一个变量的值为 1 或所有变量的值不为 1。

由非补立方集的操作，可以有效地解这个问题。其算法可以描述为

$$s_1 = x_{11} + x_{12} + \cdots + x_{18}$$

$$s_2 = x_{21}(s_1 \%_0 x_{11} \%_0 x_{12}) + x_{22}(s_1 \%_0 x_{11} \%_0 x_{12} \%_0 x_{13}) + \cdots + x_{28}(s_1 \%_0 x_{17} \%_0 x_{18})$$

$$s_3 = x_{31}(s_2 \%_0 x_{11} \%_0 x_{13} \%_0 x_{21} \%_0 x_{22}) + x_{32}(s_2 \%_0 x_{12} \%_0 x_{14} \%_0 x_{21} \%_0 x_{22} \%_0 x_{23}) + \cdots + x_{38}(s_2 \%_0 x_{16} \%_0 x_{18} \%_0 x_{27} \%_0 x_{28})$$

$$s_4 = \cdots\cdots$$

$$\cdots\cdots$$

$$s_8 = \cdots\cdots$$

这些表达式的含义是：

$s_1$：搜索所有的选择，去放第一个皇后。

$s_2$：考虑第一个皇后的位置，搜索所有的选择，去放第二个皇后。

$s_3$：考虑第一个和第二个皇后的位置，搜索所有的选择，去放第三个皇后。

……

$s_8$：考虑前面 7 个皇后的位置，搜索所有的选择，去放第八个皇后。

用 ZBDD 计算如上的表达式 $s_i(1 \leqslant i \leqslant 8)$，能够得到八皇后问题的解。表 8.1 给出了对 $N$ 皇后问题的实验结果。在表中，“BDD 节点数”这一列是直接使用布尔代数来表示 $N$ 皇后问题，并用 BDD 来表示时，所用节点数。“ZBDD”这一列是使用非补立方集和ZBDD来进行处理时所用节点数。“BDD 节点数/ZBDD 节点数”这一列是 BDD 节点数与 ZBDD 节点数之比，从此列可以看出 ZBDD 所需节点数与 BDD 的节点数相比大幅度减少。

表 8.1　$N$ 皇后问题的部分实验结果

| $N$ | 解的个数 | BDD 节点数 | ZBDD 节点数 | BDD 节点数/ ZBDD 节点数 |
|---|---|---|---|---|
| 4 | 2 | 29 | 8 | 3.6 |
| 5 | 10 | 166 | 40 | 4.2 |
| 6 | 4 | 129 | 24 | 5.4 |
| 7 | 40 | 1098 | 186 | 5.9 |
| 8 | 92 | 2450 | 373 | 6.6 |
| 9 | 352 | 9556 | 1309 | 7.3 |
| 10 | 724 | 25944 | 3120 | 8.3 |
| 11 | 2680 | 94821 | 10503 | 9.0 |
| 12 | 14200 | 435169 | 45833 | 9.5 |
| 13 | 73712 | 2044393 | 204781 | 10.0 |

### 8.3.4　用 ZBDD 进行多项式的表示与操作

多项式在数学、电子信息等领域有多种应用。本 8.3.4 小节用 ZBDD 来表示多项式，提供了一种压缩的和唯一的表示，可应用于算术表达式的等价性检查等多个方面。

1. 多项式的 ZBDD 表示

以变量 $x$ 的如下次方：$x^1$，$x^2$，$x^4$，$x^8$，… 为基础，对任意的 $x^k$，将它写成如下形式：

$$x^k = x^{(k_1+k_2+k_3+\cdots+k_m)} = x^{k_1} \cdot x^{k_2} \cdot x^{k_3} \cdots x^{k_m}$$

式中：$k_1$，$k_2$，…，$k_m$ 都是 2 的某次方。例如，多项式 $f = x^1 + x^5 + x^{10} + x^{20}$ 可以写成如下的等价形式：$f = x^1 + x^1x^4 + x^2x^8 + x^4x^{16}$，此式是作为 5 项 $x^1$，$x^2$，$x^4$，$x^8$，$x^{16}$ 的组合。分别为这 5 项各建立一个节点，每个节点有两条边，0 边和 1 边。因此多项式 $f$ 可用如图 8.6(a) 所示的 ZBDD 表示。图中各节点的编序，$x^1$ 的编序在最前面。

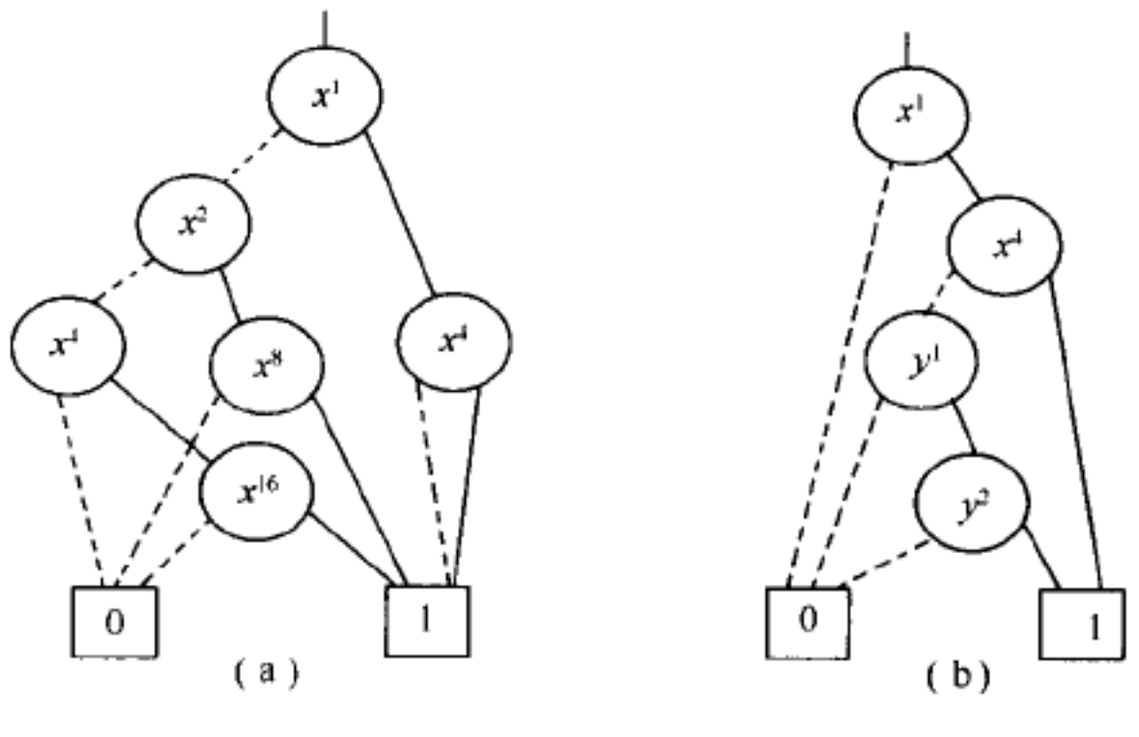



图 8.6　多项式的 ZBDD 表示

(a) 多项式 $f$ 的 ZBDD；(b) 多项式 $g$ 的 ZBDD。

若一个多项式中有多种变量，例如 $x$，$y$，$z$，则类似地，以如下次方 $x^1$，$x^2$，$x^4$，…；$y^1$，$y^2$，$y^4$，…；$z^1$，$z^2$，$z^4$，… 等为基础，在编序时，先确定变量的顺序，如 $x$，$y$，$z$，然后对每一种确定的变量，其对应节点的顺序是次数越低，则编序就越高。

图 8.6(b) 是多项式 $g = x^5 + xy^3 = x^1x^4 + x^1y^1y^2$ 的 ZBDD 表示，变量 $x$ 和 $y$ 的编序就是 $x$，$y$，即 $x$ 在前，$y$ 在后。

另外，对一般形式的多项式，它的每一项中往往有系数，这里仅讨论系数为整数的情况。

设 $C$ 是一个常数($C>1$)，可以把它写成如下的多个 2 的次方之和：

$$C = 2^{C_1} + 2^{C_2} + \cdots + 2^{C_m}$$

式中：$C_1$，$C_2$，…，$C_m$ 是互不相同的正整数。

以 $2^1$，$2^2$，$2^4$，…，等为基础，则任何一个常数 $C$ 可写成它们的组合。例如，常数 300 可写成：$300 = 2^8 + 2^5 + 2^3 + 2^2 = 2^8 + 2^1 \cdot 2^4 + 2^1 \cdot 2^2 + 2^2$。它的 ZBDD 表示如图 8.7(a) 所示。

例如，对多项式 $h = 5x^2 + 2xy$，先将常数 5 写成 $5 = 1 + 2^2$，因此 $h$ 可写成 $h = (1 + 2^2)x^2 + 2xy = x^2 + 2^2 \cdot x^2 + 2^1 \cdot x^1 \cdot y^1$，它的 ZBDD 表示如图 8.7(b) 所示。

对于负常数的情况，以 $(-2)^i$ 为基础，即 1，−2，4，−8，16，…。将该负常数进行展开。例如，(− 12) 可写成 $(-12) = (-2)^5 + (-2)^4 + (-2)^2 = -2 \cdot 2^4 + 2^4 + 2^2$。它的 ZBDD 表示如图 8.7(c) 所示。

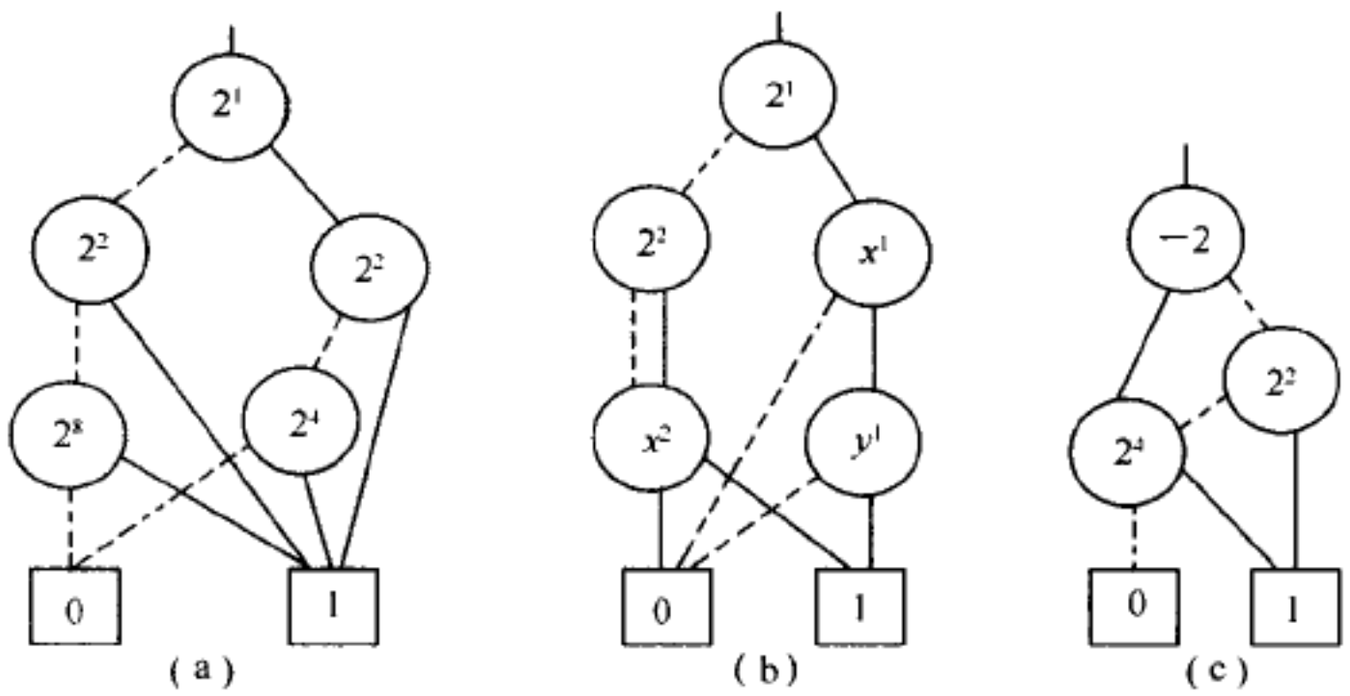



图 8.7　具有系数的多项式的 ZBDD 表示

当然，对于负常数的表示，还有其他的方法。这里的这种方法的优点是避免产生很多的值为负的节点。

用 ZBDD 表示多项式的一个特点是：它给出了多项式的一个范式表示形式，这是因为变量的次方和每一项的系数都可以进行唯一的分解。

2. 多项式的算术操作的 ZBDD 实现

对多项式进行的操作有加、减、乘等，若已经知道了两个多项式的 ZBDD 表示，如何直接求出它的经操作之后的 ZBDD 呢?这里就讨论此问题。

可以先生成一些基本多项式(如单变量、常数等) 的 ZBDD，然后使用算术操作去构建更复杂的多项式。图 8.8 说明了生成多项式($x^2 + 4xy$) 对应的 ZBDD。这里首先生成 $x$、$y$ 和 4 的 ZBDD，在这 3 个图的基础上，依次生成 $4y$、$x + 4y$ 的 ZBDD，最后由 $x$ 和 $x + 4y$ 的 ZBDD 来生成 $x(x + 4y)$ 的 ZBDD。

1) 一个变量乘多项式

一个变量 $v$ 乘以一个多项式 $F$，这种操作是多项式其他算术操作的基础。将 $F$ 分成两部分 $F_1$ 和 $F_0$。$F_1$ 包含变量 $v$，$F_0$ 不包含 $v$，即 $F = F_0 \cup (v \cdot F_1)$。这样 $v \cdot F = (v \cdot F_0) \cup (F_1 \cdot v^2)$，这时 $F$ 所对应 ZBDD 的变化如图 8.9 所示。

当变量的编序为 $x^1$，$x^2$，$x^4$，$x^8$，…，时，这种方法可以有效地进行。此外，当一个多项式被常数 $2^k$ 或 $(-2)^k$ 相乘时，这时可以通过对多项式的各个系数进行移位操作来实现。

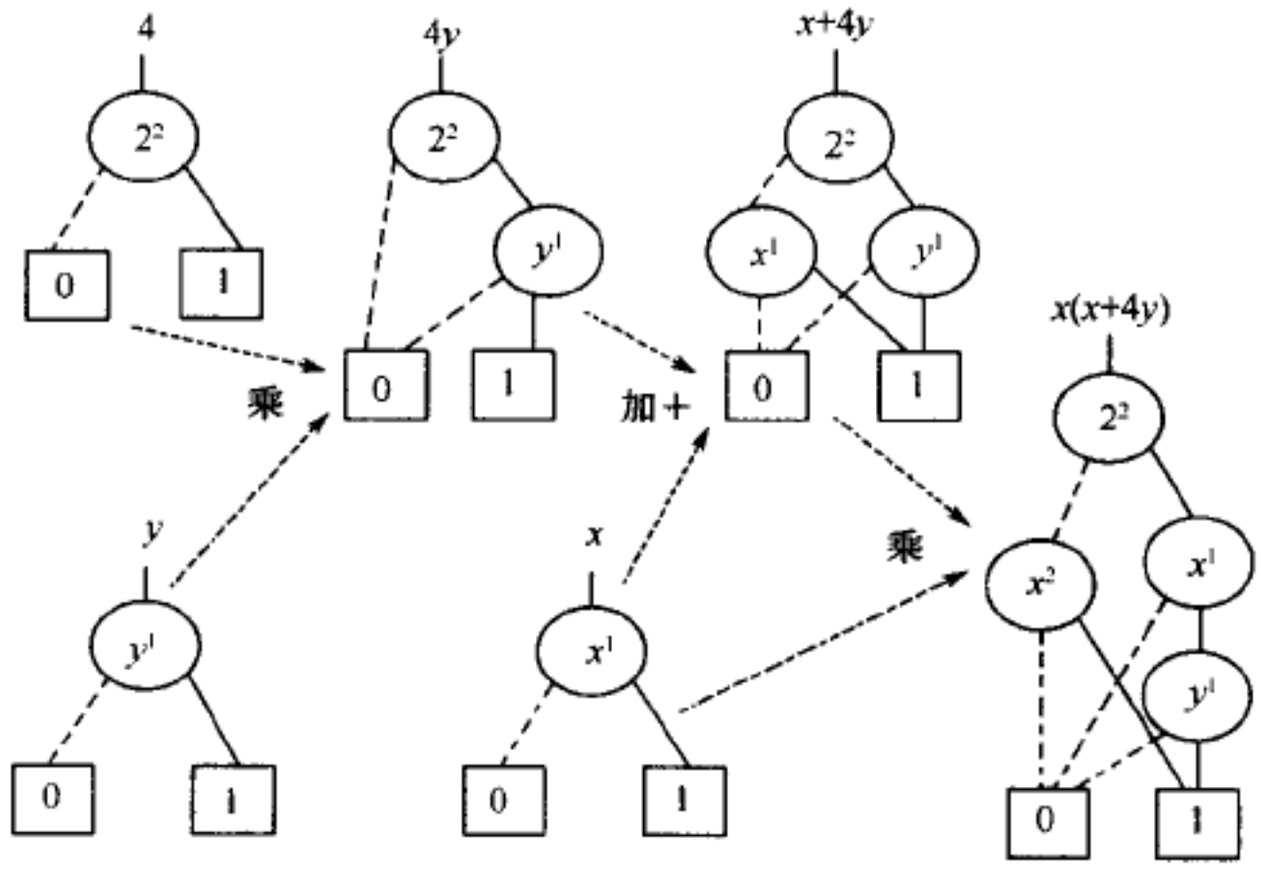



图 8.8　生成多项式($x^2+4xy$)对应的 ZBDD

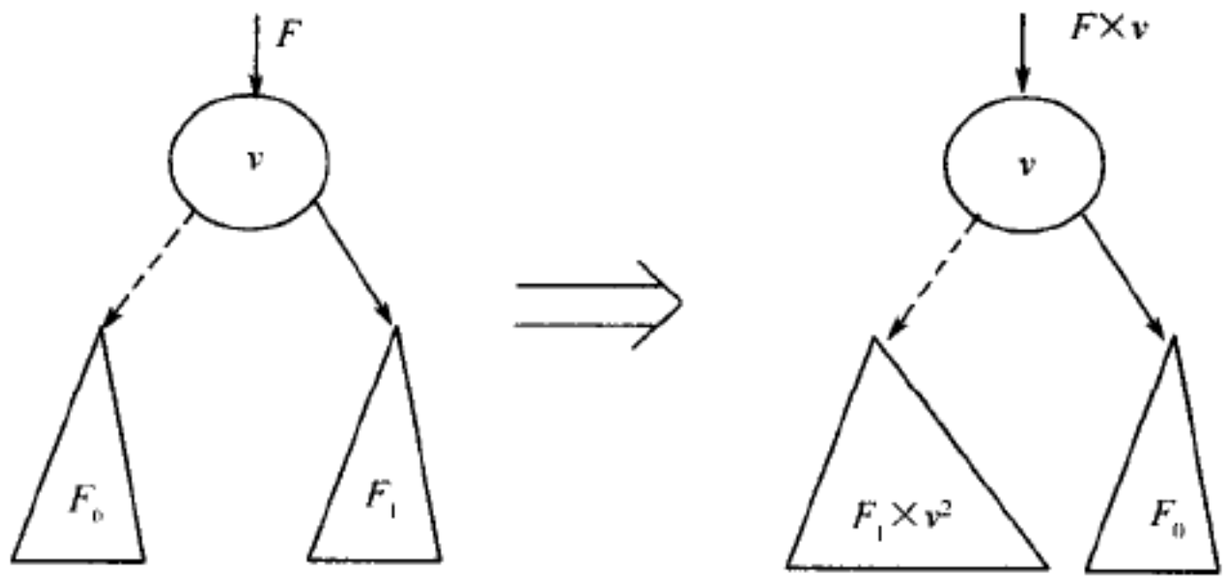



图 8.9　一个变量乘多项式的 ZBDD

2) 多项式相加

现在讨论两个多项式 $F$ 和 $G$ 的相加。若多项式 $F$ 和 $G$ 没有相同的部分，则 $F+G$ 就是将它们组合到一起。若 $F$ 和 $G$ 有相同的部分，这时可以把 $F+G$ 写成 $F+G=S+(C\cdot 2)$，这里 $C=F\cap G, S=(F\cup G)-C$。不断重复进行，直到两多项式没有相同的部分。

例如，设 $F=x+z, G=3x+y=2^1x+x+y$，现在计算 $F+G$。在第一次执行时，$C\leftarrow x, S\leftarrow 2^1x+x+y$。由于此时 $C\neq 0$，因此令 $F=2^1x+y+z(=S), G=2^1x\ (=C\times 2)$。在第二次执行时，$C\leftarrow 2^1x, S\leftarrow y+z$，由于此时 $C$ 仍不为 0，因此令 $F=y+z, G=2^2x$。在第三次执行时，$C=0$，且此时 $F+G$ 的结果就是 $2^2x+y+z$。

如果多项式中项的系数是基于 $(-2)^k$，即 $1,-2,4,-8,16,\cdots$ 等来表达时，则多项式的“加”(+) 和“减”(−) 操作可按如下方式执行：

$$(F+G)=S-(C\cdot(-2))$$

$$(F-G)=D+(B\cdot(-2))$$

这里 $D=F\cap\overline{G}, B=\overline{F}\cap G$。

3) 多项式相乘

使用前面的两个多项式相加、一个变量乘一个多项式这两种操作，来进行两个多项式 $F$ 和 $G$ 的相乘。这里所采用的是分而治之的思想。

设 $v$ 是 ZBDD 的根节点所对应的变量，首先将 $F$ 和 $G$ 分成如下两部分：

$$F = (v \cdot F_1) \cup F_0, \quad G = (v \cdot G_1) \cup G_0$$

则

$$F \cdot G = (F_0 \cdot G_0) + (F_1 \cdot G_1) \cdot v^2 + ((F_1 \cdot G_0) + (F_0 \cdot G_1)) \cdot v$$

$F \cdot G$ 的每一个子项可以用递归的方式来进行，当递归结束时，就可以得到 $F \cdot G$ 的结果。在最坏的情况下，这种方法的递归调用次数与变量数成指数增长关系。但是可以通过使用哈希表来存放最近的操作结果，而使方法的速度提高。同时，在每一次递归调用之前，通过对哈希表的查询，可以避免对相同多项式的重复操作。这样，该方法的执行时间主要依赖于 ZBDD 的规模。下面是该方法的详细实现过程。

多项式 $F \cdot G$ 的实现：

(1) 若 $F.\text{top} < G.\text{top}$，则返回 $G \cdot F$；

(2) 若 $G = 0$，则返回 0；

(3) 若 $G = 1$，则返回 $F$；

(4) 若 $H$ 已存在，这时 $H$ 为缓存中的 $F \cdot G$，则返回 $H$；

(5) $v \leftarrow F.\text{top}$；这里编序最大的在 $F$ 中；

(6) 由 $F$ 和变量 $v$，计算 $(F_0, F_1)$；

(7) 由 $G$ 和变量 $v$，计算 $(G_0, G_1)$；

(8) $H \leftarrow (F_1 \cdot G_1) \cdot v^2 + (F_0 \cdot G_0) + (F_1 \cdot G_1)) \cdot v + (F_0 \cdot G_0)$

(9) 将 $H$ 的值赋给缓存，作为 $F \cdot G$ 的值；

(10) 返回 $H$。

4) 多项式相除

两个多项式 $F$ 和 $G$ 相除，得到的结果是商 $(F/G)$ 和余数 $(F\%G)$。也可以把相除的方法进行递归描述。设 $x$ 是 ZBDD 的根节点所对应的变量，$s$ 和 $t$ 分别是 $x$ 在 $F$ 和 $G$ 中的最高次方，则把 $F$ 和 $G$ 分为如下两部分：

$$F = (x^s \cdot F_1) \cup F_0, \quad G = (x^t \cdot G_1) \cup G_0$$

因此商 $(F/G)$ 可写为

$$(F/G) = (F_1/G_1)x^{(s-t)} + (F - (F_1/G_1) \cdot x^{(s-t)} \cdot G)/G$$

这个表达式中的子项可以递归计算。由于上式中有多项式乘的操作，因此通过递归直到遇到一些基本多项式（对应的 ZBDD）时，则递归结束。例如当 $s < t$ 时，$F/G = 0$。

余数 $(F\%G)$ 可由下式得到，即

$$F\%G = F - (F/G) \cdot G$$

当使用如上的多项式相除的方法处理有多种变量的多项式时，例如有变量 $x$、$y$、$z$，该方法所得到的“商”与 ZBDD 的变量编序有关。但是，当“余数”为 0 时，所得“商”对任何变量编序都是唯一的。因此，使用这种多项式相除的方法，可以容易地检查一个多项式是否是另一个多项式的因子。

5) 替换

利用多项式的“除”，可以进行如下的替换操作。设多项式 $F$ 中含有变量 $x$，$G$ 为另一多项式，则 $F[x = G]$ 是把多项式中的 $x$ 用多项式 $G$ 来替换之后得到的多项式。这可以写成

$$F[x=G]=(F/x)[x=G]\cdot G+(F\%x)$$

对$(F/x)[x=G]$可以递归地计算。若多项式$F$中没有变量$x$出现，则$F[x=G]=F$。

由于$G$是任意的，当$G$取不同的值时，可以进行多种类型的替换。这种操作在实际应用中是非常有用的，例如$F[x=x+1]$、$F[x=z^2]$、$F[x=1]$等。

3. 多项式操作的部分实验结果

S. Minato 等在 SPARC 2 工作站上对多项式的 ZBDD 表示进行了实验[35,36]，可以处理 8000 种变量，变量的次方可达 255，每项系数的值可达 $2^{255}$。

首先对常数生成 ZBDD。在实验中，对 $n=10, 20, 30, 40, 50$，计算了 $n!$ 所对应的 ZBDD。只用 15 个节点可以表示数 1000000000。对 56!（这是超过 256 位的数）只用 62 个节点就可以表示。实验结果如表 8.2 所列。

表 8.2　$n!$ 对应的 ZBDD 的节点数

| $n$ | ZBDD 节点数 | 所需要的执行时间 /s |
|---|---|---|
| 10 | 8 | 0.1 |
| 20 | 22 | 0.3 |
| 30 | 32 | 0.9 |
| 40 | 43 | 1.7 |
| 50 | 64 | 2.8 |
| 56 | 62 | 3.2 |

对一些特殊类型的多项式，如 $x^n$、$(x+1)^n$、$\prod_{k=1}^{n}(x_k+1)$ 等，它们的 ZBDD 表示的节点数如表 8.3 和表 8.4 所列。结果表明，可以为相当大规模的多项式生成其 ZBDD 表示，其中的一些多项式具有的项数为几百万。这反过来也说明，ZBDD 可以表示大规模的多项式，而这对常规方法来说很难进行，因为它们需要的存储空间与多项式的项数成正比。

表 8.3　ZBDD 对一些常规多项式的表示

| $n$ | $x^n$ | | $n^2\cdot x^n$ | | $\sum_{k=0}^{n}x^k$ | | $\sum_{k=1}^{n}(k\times x^k)$ | | $(x+1)^n$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 节点数 | 时间 /s | 节点数 | 时间 /s | 节点数 | 时间 /s | 节点数 | 时间 /s | 节点数 | 时间 /s |
| 1 | 1 | 0.1 | 1 | 0.1 | 1 | 0.1 | 1 | 0.1 | 1 | 0.1 |
| 2 | 1 | 0.1 | 2 | 0.1 | 2 | 0.1 | 4 | 0.1 | 4 | 0.1 |
| 3 | 2 | 0.1 | 5 | 0.1 | 2 | 0.1 | 5 | 0.1 | 5 | 0.1 |
| 5 | 2 | 0.1 | 6 | 0.1 | 3 | 0.1 | 8 | 0.1 | 10 | 0.1 |
| 10 | 2 | 0.1 | 7 | 0.1 | 6 | 0.1 | 18 | 0.1 | 23 | 0.2 |
| 20 | 2 | 0.1 | 7 | 0.1 | 8 | 0.1 | 26 | 0.1 | 69 | 0.5 |
| 30 | 4 | 0.1 | 9 | 0.1 | 8 | 0.1 | 37 | 0.2 | 150 | 2.0 |
| 50 | 3 | 0.1 | 11 | 0.1 | 10 | 0.1 | 48 | 0.3 | 346 | 7.1 |
| 100 | 3 | 0.1 | 11 | 0.1 | 12 | 0.1 | 70 | 0.5 | 1209 | 39.7 |
| 200 | 3 | 0.1 | 12 | 0.1 | 14 | 0.1 | 84 | 1.0 | 4231 | 267.7 |
| 255 | 8 | 0.1 | 11 | 0.1 | 8 | 0.2 | 75 | 1.3 | 6690 | 528.8 |

表 8.4　ZBDD 对一些大规模多项式的表示

| $n$ | $\prod_{k=1}^{n}(x_k+1)$ | | | $\prod_{k=1}^{n}(x_k+k)$ | | | $\prod_{k=1}^{n}(x_k+1)^4$ | | |
|---|---|---|---|---|---|---|---|---|---|
| | 项数 | 节点数 | 时间 /s | 项数 | 节点数 | 时间 /s | 项数 | 节点数 | 时间 /s |
| 1 | 2 | 1 | 0.1 | 2 | 1 | 0.1 | 5 | 7 | 0.1 |
| 2 | 4 | 2 | 0.1 | 4 | 4 | 0.1 | 25 | 25 | 0.1 |
| 3 | 8 | 3 | 0.1 | 8 | 10 | 0.1 | 125 | 70 | 0.1 |
| 5 | 32 | 5 | 0.1 | 32 | 32 | 0.1 | 3125 | 264 | 0.3 |
| 10 | 1024 | 10 | 0.1 | 1024 | 619 | 1.1 | 9765625 | 2053 | 2.8 |
| 11 | 2048 | 11 | 0.1 | 2048 | 1131 | 3.5 | 48828125 | 2730 | 4.9 |
| 12 | 4096 | 12 | 0.1 | 4096 | 1866 | 4.8 | 244140625 | 3575 | 6.7 |
| 13 | 8192 | 13 | 0.1 | 8192 | 3334 | 8.4 | 1220703125 | 4586 | 8.8 |
| 15 | 32768 | 15 | 0.1 | 32768 | 9338 | 22.8 | $5^{15}$ | 7151 | 19.7 |

在表 8.3 和表 8.4 中，节点数是指 ZBDD 节点数。

4. 多项式的 ZBDD 表示在电路设计中的应用

由于多项式是数学的基础部分之一，因此多项式的 ZBDD 表示可应用于电路设计中以解决相关的问题。例如，一个应用是计算逻辑电路中的信号概率(图 8－10)。

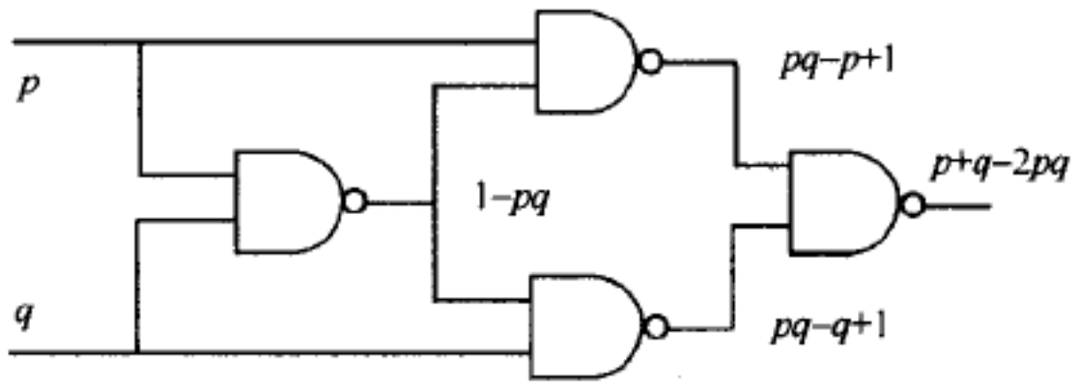



图 8.10　电路中的信号概率

对电路的每一个原始输入，给它分配一个变量代表该信号线的信号概率。电路中的内部信号线和电路原始输出的信号概率可由原始输入线的这些变量用多项式来进行表达。当电路中的节点较多时，信号概率对应多项式的规模会很大。例如，对 8 位加法器电路，相应多项式中约有 9841 项。这若用 ZBDD 来处理，只需要 125 个节点即可完成。

ZBDD 在电路设计中的另一个应用是算术级描述的形式验证。例如，假定两个算术表达式为

$$F_1 = (z-2x)\cdot(6xy-15xz+2y^2-5yz)$$

$$F_2 = (3x+y)\cdot(10xz-4xy+2yz-5z^2)$$

现在需要验证 $F_1$ 和 $F_2$ 是否相同。

由于 ZBDD 是多项式表示的范式表达，因此通过首先生成 $F_1$ 和 $F_2$ 各自对应的 ZBDD，然后检查这两个 ZBDD 是否等价，就可以判定 $F_1$ 和 $F_2$ 是否相同。此外，通过对这两个 ZBDD 进行操作，可容易地计算出这两个多项式 $F_1$ 和 $F_2$ 之差，若 $F_1$ 和 $F_2$ 不相同时，则可以找出电路设计时存在的差错。

ZBDD 在电路设计中，除了可应用于如上的信号概率计算和形式验证之外，也可用于故障仿真、功耗估计和时序分析等。

## 8.4 多级逻辑综合与 ZBDD

在 VLSI 电路设计中，逻辑综合和优化技术有着重要的地位。其中，最典型的是代数逻辑最小化方法，例如逻辑优化系统 MIS[37]，它是基于立方集（或两级逻辑）最小化，并通过使用弱除技术来生成多级逻辑。这种方法当逻辑函数表示成可行的立方集规模大小时，是有效的，但有时面对所处理的函数它的立方集表示的项数规模与输入数成指数增长关系时，就难于处理。奇偶函数和加法器就是这种函数的例子。这是基于立方集的逻辑综合方法所存在的一个问题之一。而使用 BDD，可有效地解决此问题。通过将一个立方集映射到布尔空间，该立方集可被表示为布尔函数并用 BDD 表示。使用此方法，可以用很小的存储空间来储存所表示的很大数目的立方集，这是用常规方法所不能处理的。在本节中讨论用 ZBDD 来表示立方集，以及相关的逻辑综合。

### 8.4.1 二元立方集的 ZBDD 表示

一个二元立方集，它是由一些正变量 $x_i$ 和取反变量 $\overline{x_i}$ 所组成。使用 ZBDD，能够有效地表示任何的二元立方集，且表示是唯一的。图 8.11 是立方集 $(a\ \overline{a}\ b\ \overline{b}\ c\ \overline{c}) = \{101000, 000001\} = ab + \overline{c}$ 的 ZBDD 表示。

图 8.11 的立方集可看成是多个变量 $x_k$ 和 $\overline{x_k}$ 的组合，且 $x_k$ 和 $\overline{x_k}$ 在同一个立方集中不同时出现，并且至少有一个是 0。立方集中的这些 0 在 ZBDD 中是容易压缩的。在 ZBDD 中，立方集的个数是等于图 8.11 中 1- 路径的个数。

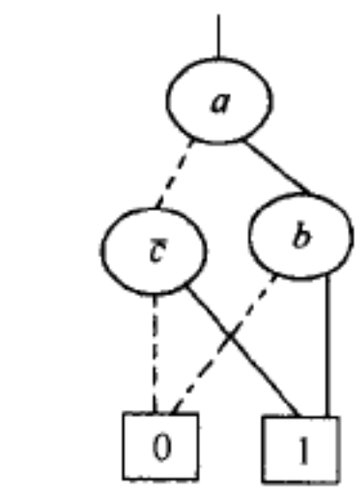


图 8.11 二元立方集的 ZBDD

在基于 ZBDD 的立方集表示中，所用的基本操作如下：

| | |
|---|---|
| "0" | 返回 ∅（无立方集） |
| "1" | 返回 1（多个立方集） |
| And0($P$,var) | 返回($\overline{\text{var}} \cdot p$) |
| And1($P$,var) | 返回(var · $p$) |
| Factor0($P$,var) | 返回 $P$ 对$\overline{\text{var}}$ 的展开因子 |
| Factor1($P$,var) | 返回 $P$ 对$\overline{\text{var}}$ 的展开因子 |
| FactorX($P$,var) | 返回 $P$ 中不包含 var 和$\overline{\text{var}}$ 的展开因子 |
| Union($P$,$Q$) | 返回($P+Q$) |
| Intsec($P$,$Q$) | 返回($P \cap Q$) |
| Diff($P$,$Q$) | 返回($P-Q$) |
| CountCubes ($P$) | 返回立方集的个数 |
| CountLists ($P$) | 返回变量的个数 |

这里，"0" 对应于 ZBDD 中的 0 终节点，"1" 对应于 1 终节点；$P$ 和 $Q$ 是立方集，var 是变量。通过使用一定数量的 And0( ) 和 And1( ) 操作，能生成任何一个立方集。

如上 3 种展开因子操作的含义是：

$$P = \overline{\text{var}} \cdot \text{Factor0} + \text{var} \cdot \text{Factor1} + \text{Factor}X$$

Intsec( ) 它是不同于逻辑上的与"AND" 操作，它获得两个立方集的公有的部分所组

成的立方集。

如上这些操作可由 ZBDD 的基本操作来完成，且它们的执行时间是略正比于图的节点数。

此外，在二元立方集的表示方面，生成压缩的立方集表示也是人们关心的问题。例如，如下的非冗余和之积式生成 ISOP 方法（Irredundant Sum-Of-Products generation）就是在此方面的一种方法。

这种算法的核心是使用如下展开式：

$$\text{isop} = \bar{v} \cdot \text{isop}_0 + v \cdot \text{isop}_1 + \text{isop}_d$$

isop 代表原始非冗余立方集，$v$ 是一个输入变量。此展开式说明 isop 能被划分为包含 $\bar{v}$、$v$ 和不包含 $\bar{v}$ 和 $v$ 的 3 种子集。当 $\bar{v}$ 和 $v$ 从每一立方集中去掉时，这 3 种子集 $\text{isop}_1$、$\text{isop}_0$ 和 $\text{isop}_d$ 也应是原始非冗余的。基于这种展开公式，ISOP 方法可以递归地生成一个原始非冗余立方集。

通过使用基于 ZBDD 的立方集表示，可以使 ISOP 方法加速。可以使用一个哈希表去存储每次递归调用的结果。在表中的每一项是由两个域 $f$ 和 $s$ 组成，$f$ 是一个指向给定 BDD 的指针，$s$ 是指向结果 ZBDD 的指针。在每一次递归调用时，检查缓存中是否有同样的子函数 $f$ 存在。若存在，就直接返回结果 $s$，这样可以避免重复操作。通过使用这种技术，可以以略正比于图的节点数的时间复杂性来执行 ISOP 方法，而独立于立方集和变量的个数。

### 8.4.2 立方集表示的分解

1. *弱除方法*

一般地，两级逻辑可被分解成多级逻辑。初始的两级逻辑对原始输出函数而言可用较大的立方集表示，如图 8.12(a) 所示。

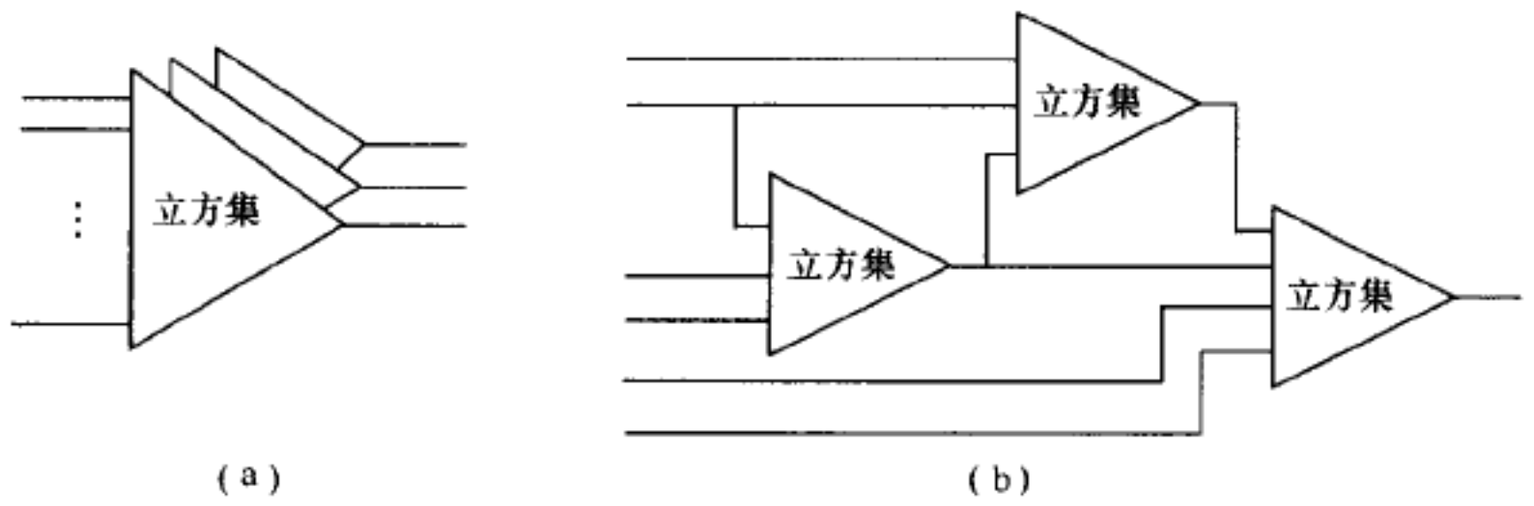



图 8.12　立方集的分解

(a) 初始立方集；(b) 立方集网络。

当决定选用一个好的中间逻辑时，则为此中间逻辑生成一个立方集，并使用一个新的中间变量去约简其他已有的立方集。最后，构建了一个由多个小的立方集组成的多级逻辑，如图 8.11(b) 所示。

这种多级逻辑由上百个立方集组成，且其中的每一个立方集都非常小（在这种稀疏组合的场合，使用 ZBDD 相当有效）。对从立方集生成多项逻辑，弱除（或称代数除）方法是一种普遍使用和成功的方法。例如，对立方集表示

$$f = abd + ab\bar{e} + ab\bar{g} + cd + c\bar{e} + ch$$

对 $f$ 用 $(ab + c)$ 来除。通过使用一个中间变量 $p$，可将 $f$ 写成

$$f = pd + p\bar{e} + ab\bar{g} + ch, \quad p = ab + c$$

对此时的 $f$，可以再用$(d+\bar{e})$ 来除，从而可对它作进一步的分解。

一般地，在弱除方法中，需要计算在除数中的每一个立方集与给定立方集相除的商之间的公共部分。例如，设 $f=abd+ab\bar{e}+ab\bar{g}+cd+c\bar{e}+ch$，$p=ab+c$。这时，$f$ 也可以写为 $f=ab(d+\bar{e}+\bar{g})+c(d+\bar{e}+h)$。商$(f/p)$ 可以表示为

$$(f/p)=(f/(ab))\cap(f/c)=(d+\bar{e}+\bar{g})\cap(d+\bar{e}+h)=d+\bar{e}$$

使用商的计算结果，则余数 $(f\%p)$ 为

$$(f\%p)=f-p(f/p)=ab\bar{g}+ch$$

用商和余数，则 $f$ 可简化为

$$f=p(f/p)+(f\%p)=p\bar{d}+pe+ab\bar{g}+ch$$

至此，使用中间变量 $p$，对 $f$ 进行分解的这一步就完成了。

如上的这种传统弱除法的执行时间与表达式的长度有关，这是因为它必须计算所有立方集与除数的商。这种方法对处理含有非常多的立方集，比如奇偶函数和加法器等，由于复杂性大，就出现了困难。对此，下面讨论用 ZBDD 来表示立方集，并进行处理。

算法的基本思想是不计算每一个立方集对除数的商，而是通过对立方集因子的展开分解。设 $v$ 是在 $p$ 中的编序为最高的输入变量，把立方集 $f$ 和 $p$ 分解成如下 3 部分：

$$f=\bar{v}f_0+vf_1+f_d,\quad p=\bar{v}p_0+v\,p_1+p_d$$

商$(f/p)$ 可为

$$(f/p)=(f_0/p_0)\cap(\ f_1/p_1\ \ )\cap(f_d/p_d)$$

商中的每一个子项被递归地计算。这一过程一直进行到递归的终止。之后就可以得到结果。若 $p_0$、$p_1$ 或 $p_d$ 中的一个值为 0，则可以跳过这一项。例如，若 $p_1=0$，则 $(f/p)=(f_0/p_0)\cap(f_d/p_d)$。每当$(f_0/p_0)$、$(f_1/p_1)$和 $(f_d/p_d)$ 中的一个变为0时，则 $(f/p)=0$。因此就不再继续进行计算。算法的详细过程如下。

计算商 $(f/p)$：

(1) 若 $p=1$，则返回 $f$。

(2) 若 $f=0$ 或 $f=1$，则返回 0。

(3) 若 $f=p$，则返回 1。

(4) 若 $q$ 已存在，这时 $q$ 为缓存中的 $f/p$，则返回 $q$；否则，$v\leftarrow p.\text{top}$。这里 $p.\text{top}$ 为 $p$ 中编序为最高的变量。

(5) 由 $f$ 和变量 $v$，计算 $f$ 的展开因子$(f_0,f_1,f_d)$。

(6) 由 $p$ 和变量 $v$，计算 $p$ 的展开因子$(p_0,p_1,p_d)$。

(7) $q\leftarrow p$。

(8) 若 $p_0\neq 0$，则 $q\leftarrow f_0/p_0$。

(9) 若 $q=0$，则返回 0。

(10) 若 $p_1\neq 0$ 且 $q=p$，则 $q\leftarrow f_1/\ p_1$。

(11) 若 $p_1\neq 0$ 但 $q\neq p$，则 $q\leftarrow q\cap(f_1/p_1)$。

(12) 若 $q=0$，则返回 0。

(13) 若 $p_d\neq 0$ 且 $q=p$，则 $q\leftarrow f_d/p_d$。

(14) 若 $p_d\neq 0$ 且 $q\neq p$，则 $q\leftarrow q\cap(f_d/p_d)$。

(15) 将 $q$ 的值赋给缓存，作为$(f/p)$ 的值。

(16) 返回 $q$。

下面用一个例子来说明上述方法的实现过程。

$$
\begin{aligned}
&(abd + ab\bar{e} + ab\bar{g} + cd + c\bar{e} + ch)/(ab + c)\\
&= (bd + b\bar{e} + b\bar{g})/b \cap (cd + c\bar{e} + ch)/c\\
&= (d + \bar{e} + \bar{g})/1 \cap (d + \bar{e} + h)/1\\
&= (d + \bar{e} + \bar{g}) \cap (d + \bar{e} + h)\\
&= d + \bar{e}
\end{aligned}
$$

为了得到余数$(f\%p) = f - p(f/p)$,需要计算在两个立方集之间的代数乘。这一过程也能递归描述,并在使用缓冲的情况下被快速执行。详细过程如下。

计算乘 $(f \cdot g)$:

(1) 若 $f.\text{top} < g.\text{top}$,则返回$(g \cdot f)$。

(2) 若 $g = 0$,则返回 0。

(3) 若 $g = 1$,则返回 $f$。

(4) 若 $h$ 已存在,这时 $h$ 为缓存中的 $f \cdot g$,则返回 $h$。$v \leftarrow f.\text{top}$。这里 $f.\text{top}$ 为 $f$ 中编序为最高的变量。

(5) 由 $f$ 和变量 $v$,计算 $f$ 的展开因子$(f_0, f_1, f_d)$。

(6) 由 $g$ 和变量 $v$,计算 $g$ 的展开因子 $(g_0, g_1, g_d)$。

(7)$h \leftarrow \bar{v}(f_0 \cdot g_0 + f_0 \cdot g_d + f_d \cdot g_0) + v(f_1 \cdot g_1 + f_1 \cdot g_d + f_d \cdot g_1) + f_d \cdot g_d$。

(8) 将 $h$ 的值赋给缓存,作为$(f \cdot g)$ 的值。

(9) 返回 $h$。

2. 除数的精选

对基于弱除方法的多级逻辑综合,其结果的质量主要依赖于除数的选取。核抽取[37]是最常见和较复杂的一种方法,它能提取出好的除数,并已成功应用于像 MIS－Ⅱ等实际的系统中。这里介绍一种找出二元立方集的除数(因子) 的简单方法。该方法的描述如下。

Divisor $(f)$

(1) 将在 $f$ 中出现两次的一个变量赋给 $v$;

(2) 若 $v$ 存在,则返回 Divisor$(f/v)$;否则返回 $f$。

若一个变量在立方集中出现二次以上,就对该变量计算立方集的展开因子,重复这一过程,可以最终获得一个除数。这里,对一个变量是否出现二次的检查,可以通过查找图的分支来进行。

对变量使用不同的编序,可以得到多个不同的除数。在这多个除数中,可以选择所具有的变量更靠近原始输入的那个除数,这样做的好处是可以使得电路具有较小的深度。

此外,使用多个立方集的公共除数可以获得较好的结果。但要获得这种公用的除数对规模大的立方集而言则是复杂和费时的。一般的做法,只是提取出一个单输出所对应的除数,然后把它应用于所有其他的立方集。

使用前两节所述的方法,可以完成一个多级逻辑综合器,它的基本流程如图 8.13 所示。开始于未优化的多级逻辑,在输入变量的一种编序下,首先生成原始输出对应布尔函数的 BDD,然后使用 ISOP 算法转换 BDD 成为原始非冗余立方集。使用基于 ZBDD 的弱除方法,将立方集展开为优化的多级逻辑。

对逻辑综合，下面给出一些相关的实验数据，以说明使用ZBDD的效果，如表8.5所列。表中的电路是8位和16位奇偶函数“XOR8”和“XOR16”，(16 +16 )位加法器“add16”，(6×6)位乘法器“mult16”。C432为ISCAS85电路。其他的一些电路选取MCNC'90国际标准电路。列“时间”是优化时花费的总CPU时间(s)；列“使用MIS”是使用MIS系统[37]将多级逻辑转换成立方集表示，并使用常规的弱除方法所得的结果。列“使用ZBDD”是使用ZBDD来表示二元立方集时对应方法所获得的结果。这些结果是在SPARC工作站(具有128MB内存)上完成的[35,38]。在表中“—”表示未得到结果。“s”表示秒，“h”表示小时。

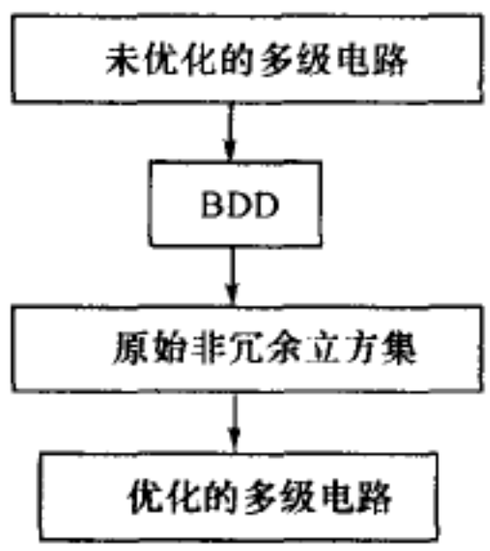



图8.13　一种多级逻辑综合的流程

表8.5　多级逻辑综合的结果

| 电路名 | 两级逻辑(使用ZBDD) | | 多级逻辑(使用MIS) | | 多级逻辑(使用ZBDD) | |
|---|---|---|---|---|---|---|
| | 信号线数 | ZBDD节点数 | 信号线数 | 时间/s | 信号线数 | 时间/s |
| X()R8 | 1152 | 28 | 28 | 38.3 | 28 | 0.3 |
| X()R16 | 557056 | 60 | — | >20h | 60 | 0.7 |
| add16 | 11468595 | 176 | — | >20h | 257 | 6.9 |
| mult6 | 22273 | 3315 | — | >20h | 6802 | 2900.7 |
| 9sym | 1036 | 42 | 83 | 29.8s | 117 | 1.8 |
| vg2 | 914 | 102 | 97 | 33.9s | 102 | 1.7 |
| alu4 | 5539 | 1129 | 1319 | 3751.6s | 1148 | 64.5 |
| apex1 | 4155 | 1768 | 2863 | 10945.1s | 2521 | 209.6 |
| apex2 | 15530 | 1144 | — | >20h | 253 | 29.5 |
| apex3 | 4679 | 1539 | 2132 | 1926.6s | 2221 | 158.2 |
| apex4 | 8055 | 1545 | 3509 | 1345.9s | 3473 | 462.4 |
| apex5 | 7603 | 2387 | 1206 | 156.9s | 1185 | 58.7 |
| C432 | 969028 | 14407 | — | >20h | 1510 | 692.3 |

实验结果表明，ZBDD表示的方法能快速地分解电路，包括奇偶函数和加法器，而对这两种电路而言其他的方法无法处理。这种方法不但使逻辑综合系统的运行效率得到了提高，而且扩大了能处理的电路类型范围。在这种逻辑综合方法中，除数的选择是一个重要问题，如何用简单的策略并考虑到立方集的特性来进行提取，以获得较优的除数，是值得进一步研究的。

## 8.5　使用BDD进行电路测试

在数字电路的生产过程中，为了检测电路功能的正确性，必须进行测试；同时在电路的使用过程中，为了检测电路中是否出现故障，也需要进行测试。为此，如果电路中存在一个故障g，为了测试该故障，应构造电路的一个输入矢量(输入序列)，在该输入矢量的作用下，使电路至少有一个输出值与正常电路(无故障电路)不同。电路测试的一个主要任务就是：对给定的电路，寻找能检测电路中所有故障的这种输入矢量。

使用二元判定图进行电路的测试生成，分为如下 4 个步骤：

(1)根据给定电路的逻辑功能，首先构造电路无故障时对应的二元判定图。

(2)对无故障电路注入一个故障，并构造故障电路对应的二元判定图。

(3)把无故障电路与故障电路所对应的这两种二元判定图作异或操作可得到一个 BDD，称该 BDD 为测试 BDD。

(4)计算测试 BDD 中从根节点到属性值为 1 的终节点的所有路径。每一路径上的边对应的变量取值就为故障的测试矢量。

下面以图 8.14 所示 C17 电路中的信号线 e4 发生 s-a-0 故障为例，说明生成该故障的测试矢量的过程。

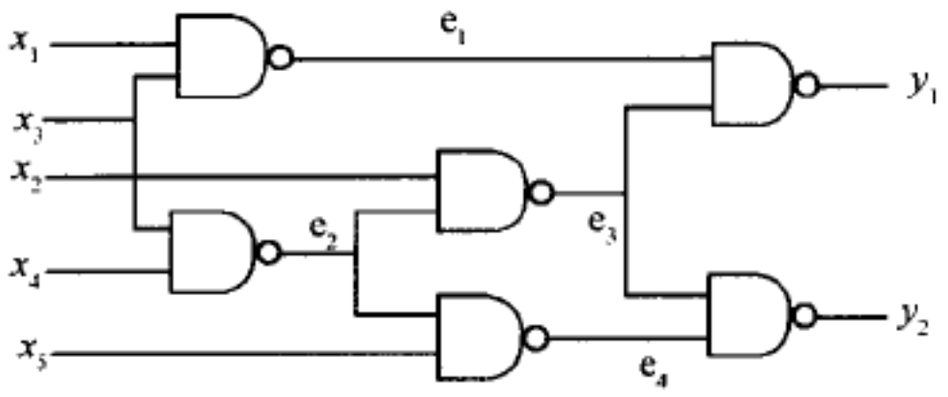



图 8.14　C17 电路

由 C17 电路的结构可知，信号线 e4 的 s-a-0 故障只对电路的输出 $y2$ 产生影响。因此首先分别建立 e4 和 e3 的无故障 BDD，然后对它们进行布尔 NAND 运算而获得 $y2$ 的无故障 BDD，如图 8.15 所示。

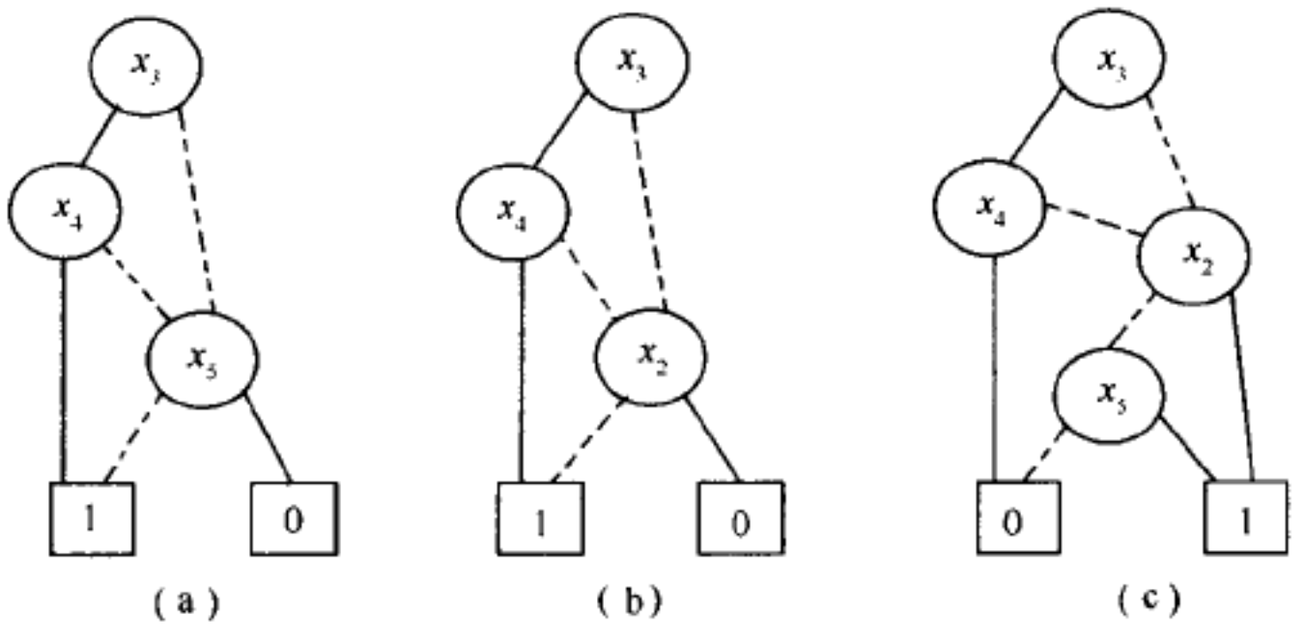



图 8.15　无故障电路对应 BDD 的建立

(a) $e_4$ 的无故障 BDD；(b) $e_3$ 的无故障 BDD；(c) $y_2$ 的无故障 BDD。

对 e4 的故障 BDD，由于 e4 发生 s-a-0 故障，因此它的故障 BDD 中只有一个属性值为 0 的终节点。由于 e3 不在 e4 的扇出路径上，因此 e3 的故障 BDD 与它的无故障 BDD 相同。将 e4 的故障 BDD 和 e3 的故障 BDD 进行布尔 NAND 运算而获得 $y2$ 的故障 BDD，它仅由一个属性值为 1 的终节点组成，如图 8.16 所示。

把 $y2$ 的故障 BDD 与它的无故障 BDD 进行布尔 XOR 运算而获得一个测试 BDD，如图 8.17 所示。

在图 8.17(c) 的测试 BDD 中，从根节点到属性值为 1 的终节点的路径有 3 条，分别为：(1) $x_3 = 0$，$x_2 = 0$，$x_5 = 0$；(2) $x_3 = 1$，$x_4 = 0$，$x_2 = 0$，$x_5 = 0$；(3) $x_3 = 1$，$x_4 = 1$。在这 3 条路径中没有涉及到的变量，它们的值可任取为 0 或 1。因此信号线 e4 的 s-a-0 故障的测试矢量如表 8.6 所列。在表 8.6 中“*”表示对应输入信号线的值可任取为 0 或 1。

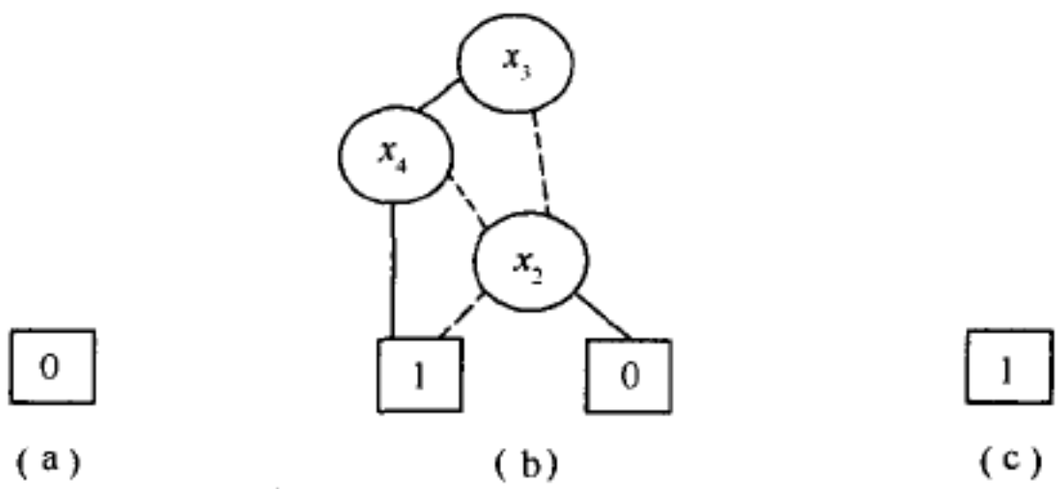



图 8.16　故障 BDD 的建立

(a)e4 的故障 BDD；(b)e3 的故障 BDD；(c)y2 的故障 BDD。

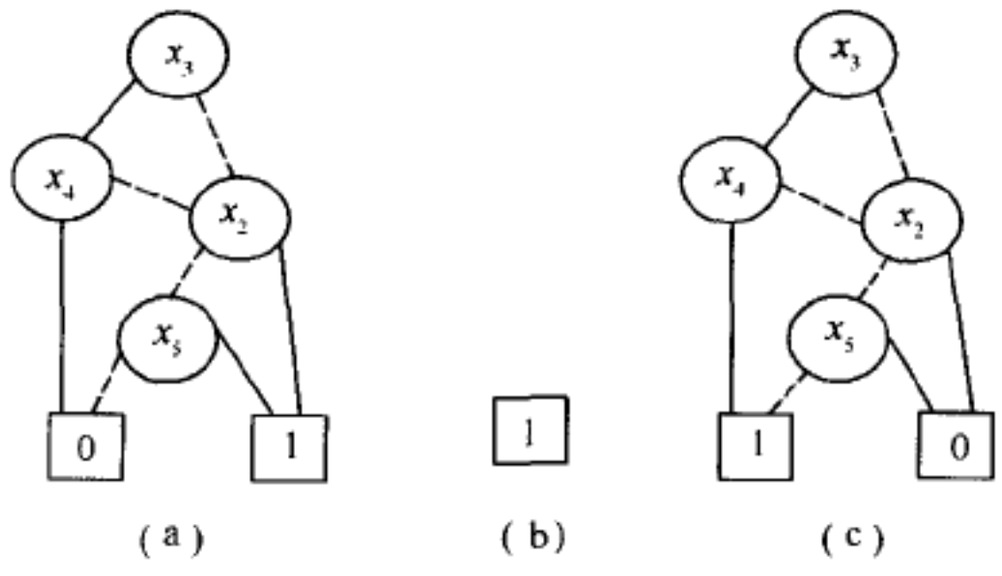



图 8.17　测试 BDD 的建立

(a)y2 的无故障 BDD；(b)y2 的故障 BDD；(c) 测试 BDD。

表 8.6　测试矢量

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|
| * | 0 | 0 | * | 0 |
| * | 0 | 1 | 0 | 0 |
| * | * | 1 | 1 | * |

表 8.6 实质上给出了信号线 e4 的 s-a-0 故障的所有测试矢量。同理可以求出 C17 电路的其他故障的测试矢量。之后，通过合并相同的矢量，可以获得电路的最小规模测试集。

与其他的基于电路结构的测试方法比较，基于 BDD 的电路测试方法的优点主要有：

(1)可以直接判断故障的可测性(即是否存在测试矢量)。这可通过比较无故障 BDD 与故障 BDD 是否相同(同构)来进行。若相同，则说明对应故障不可测；若不相同，则说明故障可测。

(2)可以求出一个故障的全部测试矢量。因此通过测试矢量的合并，可以方便地求得电路的最小规模测试集。

## 8.6　小　结

在第 7 章和第 8 章，较详细讨论了二元判定图 BDD 的结构、性质以及一些应用。二元判定图作为逻辑函数或数字电路的一种有效表示和操作方法，受到了人们的广泛重视。

在 BDD 的早期研究方面，C. Y. Lee 在 1959 年最早提出了数字电路的二元判定图表示[39]；在 1978 年 S. B. Akers 对二元判定图的简化做了进一步研究[40]；在 1986 年 R.

E. Bryant 用图论方法从理论上说明了 BDD 中子图的同构、求约简二元判定图 ROBDD 的方法，说明了一个逻辑函数其对应的 ROBDD 在同构的意义下是唯一的[41]。R. E. Bryant 的这一开创性工作在 BDD 的发展方面起到了非常关键和重要的作用。

在 BDD 的程序实现方面，ITE 算法[42]的效率较高。在 BDD 中使用节点共享，即使用共享二元判定图 SBDD[43,44]，可以使二元判定图的规模得到很大程度的降低。

多年来，人们对如何降低 BDD 的规模进行了不断研究。研究了变量编序对 BDD 规模的影响[35,45~51]和最优编序[52~55]等。同时根据不同的应用领域，提出了 BDD 的一些变形或扩展，例如 ZBDD[36~38,56]、用于多值逻辑函数操作与表示的多值判定图[57,58]等。在 BDD 的应用方面，人们也做了大量工作，除了本章前面介绍的内容之外，在组合电路与时序电路的验证[59~72]、电路的测试[73~77]、电路的可靠性等多方面都得到了应用。