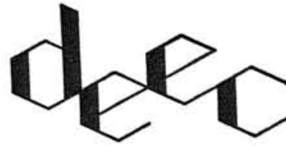
Maria I. Garcia M. Abreu S. Fernandes

ESTUDO de ALGORITMOS

DEEC
FEUP
1991



DEPARTAMENTO DE

ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

ESTUDO DE ALGORITMOS DE MINIMIZAÇÃO DE FUNÇÕES BOOLEANAS UTILIZANDO DIAGRAMAS DE DECISÃO BINARIA

FACULDADE DE ENGENHARIA

UNIVERSIDADE DO PORTO

Rua dos Bragas, 4099 Porto Codex – PORTUGAL

encadernar

Faculdade de Engenharia da Universidade do Porto

ESTUDO DE ALGORITMOS DE MINIMIZAÇÃO DE FUNÇÕES BOOLEANAS UTILIZANDO DIAGRAMAS DE DECISÃO BINARIA

681.3(043) FER/EST
UNIVERSIDADE DO PORTO
Faculdade de Engenharia
BIBLIOTECA
N.º 27066
CDU
Data 24 / 09 / 1998

catalogada

MARIA ISABEL GARCIA MARQUES ABREU SIMÕES FERNANDES

DEEC
FEUP

JULHO DE 1991

Tese submetida para satisfação parcial
dos requisitos do programa de Mestrado em
Engenharia Electrotécnica e de Computadores
(Informática Industrial)

Tese realizada sob a supervisão do
Professor Doutor José Alfredo Ribeiro da Silva Matos
Professor Associado do
Departamento de Engenharia Electrotécnica e de Computadores
da
Faculdade de Engenharia da Universidade do Porto

AGRADECIMENTOS

Ao Professor Silva Matos, meu orientador científico, por toda a atenção e apoio que dispensou à orientação do trabalho realizado e à revisão da tese.

Ao Instituto Superior de Engenharia do Porto, que me facultou as condições necessárias à frequência do Mestrado.

A todos os meus amigos e colegas que sempre me incentivaram e de alguma forma contribuiram para a realização deste trabalho.

Ao meu marido e filhos por todo o carinho com que sempre me acompanharam.

INDICE

Introdução 1

1. A Minimização Booleana 4

1.1 Enquadramento 4
1.2 Métodos de minimização 6

2. Definições Matemáticas Básicas 9

2.1 Introdução 9
2.2 Operações com funções lógicas 11
2.3 Representação algébrica de uma função lógica 12
2.4 Cubos e coberturas 14

3. Método de Quine-McCluskey 19

3.1 Introdução 19
3.2 Representação espacial de funções booleanas 19
3.3 Descrição do método 20
3.4 Circuitos de saídas múltiplas 31
3.5 Conclusão 36

4. O Programa EXPRESSO-II 37

4.1 Introdução 37
4.2 O Procedimento COMPLEMENT 41
4.3 O Procedimento TAUTOLOGIA 42
4.4 O Procedimento EXPAND 42
4.5 O Procedimento ESSENTIAL_PRIMES 43
4.6 O Procedimento IRREDUNDANT_COVER 44
4.7 O Procedimento REDUCE 46
4.8 O Procedimento LAST_GAP 47
4.9 O Procedimento MAKE_SPARSE 48
4.10 Conclusões 50

5. Método de Simplificação usando BDD's 51

5.1 Introdução 51
5.1 Alguns conceitos básicos 53
5.3 Construção de um BDD 58
5.4 Identificação ou descrição de um BDD 61
5.5 Simplificação de um BDD 64
5.6 Construção e simplificação de BDD's correspondentes a funções de saídas múltiplas 65

6. O Programa de Minimização Booleana 68

6.1 Introdução 68
6.2 Avaliação de expressões booleans 68
6.3 Simplificação de um BDD correspondente a uma função de saída única 69
6.4 Simplificação de um BDD correspondente a uma função de saídas múltiplas 83
6.5 Resultados obtidos 84

7. Conclusões 90

Referências 92

Apêndice A. Decomposição de Funções A.1

A.1 Cofactores e expansão de Shannon A.1
A.2 Fusão A.4

Apêndice B. Funções Monótonas B.1

B.1 Definições e conceitos básicos B.1
B.2 Escolha da variável de partição B.2
B.3 Complementar de uma cobertura monótona B.3

Introdução

Nesta tese pretende abordar-se o problema da minimização de funções booleanas ou lógicas e apresentar algumas soluções baseadas na representação dessas funções a partir de diagramas de decisão binária, normalmente referidos por **BDD**'s (Binary Decision Diagram), que se admite ser um método eficiente para posterior obtenção de funções simplificadas [AKE78].

Dada a complexidade que estes problemas facilmente podem atingir, a obtenção de soluções exactas é, na prática, impossível de conseguir para casos reais; sendo assim, têm-se desenvolvido métodos heurísticos que procuram contornar esta dificuldade [BRA84, BRO81, HON74].

A importância e actualidade do tema advém do facto de que a obtenção de soluções eficientes e económicas para problemas complexos que surgem em múltiplas áreas científicas só ser possível após uma simplificação das funções lógicas e portanto dos circuitos que constituem a base da respectiva implementação.

Assim, é fundamental obterem-se circuitos cujo desempenho seja cada vez melhor; portanto, circuitos lógicos a que corresponda, por exemplo, a menor área possível, a menor potência dissipada ou a maior velocidade de execução. As funções lógicas correspondentes a esses circuitos deverão ser simplificadas não só com o objectivo de se obter a função lógica mínima, mas também um tempo de execução aceitável; portanto, a minimização lógica deve conduzir globalmente à optimização dos resultados finais.

Neste trabalho são apresentados algoritmos de minimização de funções booleanas baseados em BDD's e um conjunto de programas que os implementam. O objectivo do trabalho, mais do que apresentar um programa de minimização, é o de investigar diferentes estratégias comparando entre si os méritos relativos das técnicas heurísticas que as suportam. Com este fim foi desenvolvido um programa destinado a funcionar como "banco de testes" de algumas estratégias consideradas, permitindo a obtenção de resultados que são objecto de posterior discussão.

Deste modo, ao longo deste trabalho, a minimização booleana é encarada independentemente de qualquer futura implementação, não se tendo em mente a execução de uma simplificação específica para ser usada por uma tecnologia bem definida, como por exemplo, PLA's ou elementos da chamada "lógica aleatória".

Esta tese está dividida em sete capítulos, cujo conteúdo se descreve a seguir.

No primeiro capítulo é feito o enquadramento dos problemas a analisar e apresentam-se sumariamente aspectos essenciais de alguns métodos utilizados em minimização lógica.

O segundo capítulo introduz a notação e fornece as definições e os fundamentos teóricos necessários à posterior manipulacão das funções lógicas e à descrição dos algoritmos apresentados.

No terceiro capítulo descreve-se e exemplifica-se um método clássico e exaustivo de simplificação de funções booleanas, o método tabular de Quine-McCluskey.

No quarto capítulo é apresentado resumidamente um método heurístico de minimização lógica de que foi publicada uma primeira versão em 1984, posteriormente melhorada, e que constitui a base de um programa de minimização booleana muito divulgado - EXPRESSO-II.

Com estes dois capítulos pretende-se apenas situar o problema da simplificação das funções lógicas e referir algumas das soluções já estudadas.

No quinto capítulo é apresentada uma panorâmica geral sobre BDD's, a sua construção e identificação. Introduz-se assim uma diferente abordagem ao problema da minimização booleana, baseada na proposta de Akers que, em finais da década de 70, recorreu a diagramas de decisão binária (BDD) para representar funções lógicas.

O sexto capítulo é constituido pela descrição do programa de minimização implementado a partir de BDD's e baseado em heurísticas que foram definidas de acordo com estratégias de simplificação escolhidas. Estudaram-se funções de saída única, assim como funções de saídas múltiplas. Os resultados obtidos a

partir do referido programa de minimização booleana são apresentados e discutidos no final do capítulo.

No sétimo e último capítulo é feita uma apreciação do trabalho realizado e apresentam-se algumas conclusões.

No Apêndice A é referida a decomposição de funções baseada na expansão de Shannon de uma função lógica, enquanto que no Apêndice B se introduz o problema das funções monótonas, são dadas definições e são apresentados alguns resultados conhecidos.

1. A Minimização Booleana

1.1 Enquadramento

O aperfeiçoamento das técnicas de minimização de funções lógicas ou booleanas tem sido baseado, e de certo modo condicionado, pelo desenvolvimento de conceitos e processos que conduzam a implementações que minimizem um determinado critério (número de literais, número de portas lógicas, número de transistores, área ocupada) [AKE78].

Quando nos anos 50 se iniciou o estudo do projecto digital, as gates lógicas eram muito dispendiosas. Assim, tornou-se importante que se pudesse reduzir o número de elementos usados na implementação das diversas funções lógicas.

Nos finais dos anos 60 e início dos anos 70, o custo das gates lógicas foi reduzido e portanto uma minimização lógica óptima deixou de ser tão essencial como o era até aí. Posteriormente, nos finais dos anos 70, com a utilização das PLA's (Programable Logic Arrays) na implementação dessas funções, voltou a ter grande impacto a necessidade de obtenção de uma boa minimização, já que cada termo produto obtido seria implementado como uma linha na PLA. Logo, a simplificação das funções lógicas transformou-se numa área de investigação de grande importância [BRA84].

Contudo, essa minimização é um problema de difícil abordagem, na medida em que, para se poder obter a solução óptima teriam de utilizar-se métodos exactos e exaustivos [MCC65, QUI52] que garantissem a cobertura de todas as possíveis hipóteses de simplificação. Sendo assim e para funções com elevado número de variáveis de entrada, seriam necessários não só tempos de execução muito elevados mas também a utilização de enormes espaços de memória, que tornam o problema intratável de um ponto de vista prático.

Como resultado, é normalmente aceite a utilização de soluções "menos óptimas" desde que dentro de certos limites, a que correspondam tempos de execução aceitáveis e que podem ser

conseguidas, por exemplo, pelo desenvolvivento de métodos heurísticos [BRA84, BRO81, HON74].

Portanto, desde que se defina um critério que permita determinar a qualidade de uma dada representação simplificada da função relativamente a outras, é possível dizer-se que existe uma representação que é óptima, pelo menos em relação às restantes consideradas.

Esse critério, classicamente considerado, aponta como solução óptima aquela a que corresponda:

1º) um número mínimo de produtos lógicos;

(Um **produto lógico** é uma expressão constituida pelo produto de variáveis de entrada e/ou dos seus complementos)

2º) no caso de existirem várias representações da função com o mesmo número mínimo de produtos lógicos, será óptima aquela que contiver o menor número de literais;

(Um **literal** é uma das possíveis ocorrências de uma variável de entrada)

Sempre que possível, convém caracterizar o problema em estudo, em termos da sua complexidade de modo a poder avaliar-se o comportamento de cada algoritmo utilizado para a sua resolução. A complexidade do problema da minimização lógica, considerando funções com **n** variáveis de entrada, pode ser caracterizada pelos seguintes parâmetros [BRA84]:

número máximo de mintermos	2^n
número máximo de primos implicantes	$3^n/n$

Em face destes valores é pouco provável poder encontrar-se um algoritmo exacto que seja eficiente [BRA84], uma vez que se está a trabalhar com problemas cuja resolução envolve o cálculo de um número de elementos que cresce exponencialmente com o número de variáveis de entrada.

1.2 Métodos de minimização

a) O método álgebrico

Quando as funções booleanas a minimizar são funções simples, então é fácil aplicar os teoremas da álgebra de Boole e efectuar a simplificação das expressões booleanas correspondentes. Este método aplica-se apenas a funções com poucas variáveis de entrada, para evitar que se cometam erros.

b) O método dos mapas de Karnaugh

O método clássico de Karnaugh, apresentado em todos os livros de Circuitos Lógicos e Sistema Digitais [HIL81, MAR67], é um processo intuitivo e manual e permite a simplificação de funções booleanas desde que o número de variáveis de entrada não exceda 5 ou 6. Quando o número de variáveis de entrada aumenta, o excessivo número de quadrículas do mapa dificulta a selecção dos mintermos adjacentes. A grande desvantagem deste método é a sua dependência da intuição ou habilidade humana para reconhecer determinadas simplificações. Assim, para funções de 6 ou mais variáveis é difícil ter-se a certeza de se ter feito a escolha ideal e portanto se obter a melhor simplificação.

c) O método tabular de Quine-McCluskey

Este método clássico surge após o método de Karnaugh e é um processo mais sofisticado. É um método exaustivo e exacto e garante a obtenção de uma expressão simplificada que é de facto a solução óptima. Tem a vantagem de poder ser aplicado a problemas com dimensão apreciável e poder traduzir-se num conjunto de passos susceptível de realização como um programa de computador. Todavia, e porque o método é exaustivo, os tempos de execução do programa são elevados assim como o espaço de memória utilizado. Este facto limita a dimensão das funções a simplificar por este processo a funções de cerca de 16 variáveis de entrada [FAB90, HIL81, MCC65, QUI52].

d) Os métodos heurísticos

Para tentar resolver os problemas que surgem com a execução do método de Quine-McCluskey, apareceram mais recentemente métodos heurísticos que produzem boas soluções, em tempos de

computação razoáveis e usando um espaço de memória aceitável. Refira-se o programa MINI [HON74] que foi o primeiro a surgir, desenvolvido pela IBM em meados da década de 70, e posteriormente o programa PRESTO [BRO81]. Pode-se mencionar ainda o programa SPAM [KAN81] que é uma versão mais recente de MINI, o programa POP [DEM84] que é baseado nas ideias expostas em PRESTO e os programas EXPRESSO-I [BRA82] e EXPRESSO-II [BRA84] criados durante os anos de 1982 e 1984.

As simplificações introduzidas por estes métodos baseiam-se fundamentalmente na expansão dos implicantes que representam a função lógica, um de cada vez, e na posterior remoção de todos aqueles que sejam cobertos pelo implicante expandido. Deste modo, nos métodos heurísticos, os implicantes são expandidos e alguns deles são removidos após se verificar serem cobertos pelos termos expandidos, eliminando assim o problema base dos métodos anteriormente mencionados, ou seja, a geração exaustiva e armazenamento de todos os implicantes. Por este processo obtém-se uma cobertura para a função dada que é formada por primos implicantes, mas que não é necessariamente a cobertura mínima; posteriormente essa cobertura é transformada na mínima cobertura possível. Esta sequência de expansões-reduções é executada iterativamente até não serem possíveis mais simplificações.

O programa EXPRESSO-II [BRA84] segue basicamente a sequência de transformações expansão-redução sugerida e com a sua utilização pretende obter-se uma ferramenta de minimização lógica que, na maioria dos casos, resolva os problemas recorrendo a recursos computacionais limitados e produza resultados o mais próximo possível da solução óptima. Os métodos heurísticos são desenvolvidos de modo a permitirem a obtenção de soluções fiáveis, isto é, em utilizações "normais" encontrar-se-á uma solução que pode não ser a óptima mas que se pretende seja próxima das melhores.

e) O método baseado em BDD's

A utilização do conceito de BDD, sugerido por Akers [AKE78] nos finais dos anos 70, conduz a uma abordagem própria do problema da minimização booleana. Assim, a ideia base é transformar a lista de mintermos ou a equação que define a função a simplificar numa árvore binária, em que os nós são os elementos de decisão ou

controlo e os ramos seus filhos são associados aos valores lógicos zero (falso) ou um (verdadeiro) dependendo da decisão tomada.

O valor lógico à saída de um dado nó será o valor da sua entrada esquerda se a variável de controlo do nó for igual a **0**, ou o valor da sua entrada direita se a variável de controlo for igual a **1**.

A árvore ou diagrama de decisão binária (**BDD**) assim obtida é posteriormente simplificada mediante a utilização de técnicas próprias e depois novamente convertida em equações lógicas.

Associados a este método existem alguns problemas delicados como, nomeadamente, a dependência da ordem por que se consideram as variáveis de entrada na construção do diagrama binário, em relação à eventual obtenção ou não da árvore simplificada ideal [BRY86, MOR82]. Portanto existe a possibilidade de não se encontrarem, após simplificação, as equações mais compactas possível dado a árvore minimizada correspondente não ser a óptima. Há, contudo, algoritmos que permitem encontrar a ordenação ideal para se construir a árvore binária cuja simplificação conduz ao diagrama óptimo [FRI87].

Note-se que a descrição dos vários métodos referidos utiliza os mesmos conceitos base; todavia, e porque cada método parte de diferentes suportes matemáticos, podem conduzir à ideia de se tratar de definições distintas. Ao longo deste trabalho é feita a tentativa de uniformizar e relacionar a terminologia usada pelos diversos algoritmos, com o objectivo de se ultrapassar essa aparente diferenciação.

2. Definições Matemáticas Básicas

2.1 Introdução

Neste capítulo serão introduzidas algumas notações e definições básicas, além de conceitos teóricos fundamentais sobre lógica e funções lógicas, necessários à posterior manipulação das correspondentes funções.

Seja **X** = {0, 1} e **Y** = {0, 1, 2}; então pode definir-se uma **função lógica** (ou booleana) **ff** com **n** variáveis de entrada (x_1, x_2, ..., x_n) e **m** variáveis de saída (y_1, y_2, ..., y_m), como sendo uma aplicação de $\mathbf{X}^n$ em $\mathbf{Y}^m$, onde **x** = [x_1, x_2, ..., x_n] pertencente a $\mathbf{X}^n$ são as entradas de **ff** e **y** = [y_1, y_2, ..., y_n] pertencente a $\mathbf{Y}^m$ as correspondentes saídas.

Usar-se-á o símbolo **ff** para representar uma **função lógica não completamente especificada**, isto é, uma função cujas saídas pertencem ao conjunto formado pelos valores lógicos (0, 1, 2). Refira-se que o 2 é usado para representar um don't care, isto é, um valor da função não especificado. Por outro lado uma **função lógica completamente especificada f** é uma função cujas saídas são apenas 0's ou 1's.

Para cada componente $\mathbf{ff}_i$ (i = 1, 2, ..., m) de **ff** podemos definir três conjuntos de valores de entradas de acordo com os valores obtidos na saída. Assim $\mathbf{X}_i^{ON}$ que está contido em ou é igual a $\mathbf{X}^n$, é o conjunto dos valores de entrada cujas saídas são iguais a 1, isto é, ($ff_i(x) = 1$); $\mathbf{X}_i^{ON}$ é denominado ON-set. Analogamente, $\mathbf{X}_i^{OFF}$ que está contido em ou é igual a $\mathbf{X}^n$ é o OFF-set, pois é o conjunto formado pelos valores de entrada cujas saídas são 0, isto é, ($ff_i(x) = 0$). E ainda $\mathbf{X}_i^{DC}$ que está contido em ou é igual a $\mathbf{X}^n$ e que é chamado DON'T CARE-set, a que correspondem os valores de entrada cujas saídas são iguais a 2, isto é, ($ff_i(x) = 2$), ou por outras palavras, a $\mathbf{X}_i^{DC}$ correspondem os valores de entrada para os quais a função não tem as saídas especificadas.

Uma função lógica com **m** = 1 é chamada função lógica de uma só saída; enquanto que se **m** >1 teremos uma função lógica de múltiplas saídas.

As funções lógicas podem ser representadas de diversas maneiras, sendo uma das formas mais simples a tabela de verdade. Assim, o valor de saída é especificado para cada conjunto de possíveis valores de entrada. Seja por exemplo, **ff** : $\mathbf{X^3}$ -> $\mathbf{Y^2}$ representada do seguinte modo:

x_1	x_2	x_3	y_1	y_2
0	0	0	1	1
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	2
1	1	0	1	1
1	1	1	2	1

Para esta função temos:

X_1^{ON} = { [000]; [001]; [100]; [101]; [110] }

X_1^{OFF} = { [010]; [011] }

X_1^{DC} = { [111] }

X_2^{ON} = { [000]; [010]; [011]; [110]; [111] }

X_2^{OFF} = { [001]; [100] }

X_2^{DC} = { [101] }.

A esta representação tabular pode facilmente fazer-se corresponder uma representação geométrica. Genericamente, diremos que se podem representar as várias possíveis combinações das **n** variáveis de entrada, como pontos de um espaço a **n** dimensões, onde o conjunto de todos os $\mathbf{2^n}$ possíveis pontos formará os vértices de um n-cubo booleano (hipercubo

booleano) [BRA84, FAB90]. Reportando-nos ao exemplo atrás referido, podemos então representar ff_1 e ff_2 construindo dois cubos geométricos (m = 2) nos dois espaços booleanos a três dimensões (n = 3).

Assim e generalizando para uma função lógica com **m** variáveis de saída e **n** variáveis de entrada, poderemos considerar a sua representação geométrica como sendo n-cubos representados em **m** espaços booleanos a **n** dimensões. Logo existe uma correspondência de um para um, entre o conjunto X_i^{ON}, X_i^{OFF} e X_i^{DC} e o conjunto dos vértices V_i^{ON}, V_i^{OFF} e V_i^{DC} dos n-cubos representados nos **m** espaços booleanos a **n** dimensões.

2.2 Operações com funções lógicas

a) O **complemento** de uma função lógica completamente especificada $f: X^n \to Y^m$ é uma outra função lógica completamente especificada $\bar{f}: X^n \to Y^m$, tal que as suas componentes $(\bar{f}_1, ..., \bar{f}_m)$ têm o ON-set igual ao OFF-set de **f** e o OFF-set igual ao ON-set de **f**.

b) A **intersecção** ou **produto** de duas funções lógicas completamente especificadas **f** e **g** é uma função lógica completamente especificada $h = f \cap g$ ou $h = f.g$ cujas componentes h_i têm o ON-set igual à intersecção dos ON-sets das correspondentes componentes de **f** e **g**.

c) A **diferença** entre duas funções lógicas completanente especificadas **f** e **g** é uma função lógica completamente especificada $h = f - g$ que se obtém pela intersecção de **f** com o complementar de **g**; logo $h = f \cap \bar{g}$. Então o ON-set das componentes de **h** é formado pelos elementos do ON-set de **f** que não pertencem ao ON-set de **g**.

d) A **reunião** ou **soma** de duas funções lógicas completamente especificadas **f** e **g** é uma função lógica completamente especificada $h = f \cup g$ ou $h = f + g$ cujas componentes h_i têm o ON-set igual à reunião dos ON-sets das correspondentes componentes de **f** e **g**.

e) Uma função lógica completamente especificada é dita uma **tautologia**, $f = 1$, se os OFF-sets de todas as suas componentes

são vazios; isto é, se todas as saídas de **f** são iguais a 1, quaisquer que sejam as entradas.

No caso das funções lógicas não completamente especificadas **ff**, podem construir-se três grupos de funções lógicas completatente especificadas ff^{ON}, ff^{OFF} e ff^{DC} (usualmente representadas por **f**, **d** e **r**) que determinam univocamente a função **ff**.

Define-se ff^{ON} de modo a que os ON-sets das suas componentes igualem os ON-sets das correspondentes componentes de **ff** e os OFF-sets das suas componentes igualem a reunião dos DON'T CARE-sets com os OFF-sets das correspondentes componentes de **ff**.

A função lógica ff^{DC} tem os ON-sets das suas componentes iguais aos DON'T CARE-sets das correspondentes componentes de **ff** e os OFF-sets das suas componentes iguais à reunião dos ON-sets com os OFF-sets das correspondentes componentes de **ff**.

Finalmente ff^{OFF} tem os ON-sets das suas componentes iguais aos OFF-sets das correspondentes componentes de **ff** e os OFF-sets das suas componentes iguais à reunião dos ON-sets com os DON'T CARE-sets das correspondentes componentes de **ff**.

Notar que a reunião de **f** com **d** e com **r** cobre os **m** espaços booleanos associados a **ff**. Isto é, para cada componente **i** $(1 \leq i \leq m)$ o ON-set (f_i), o ON-set (d_i) e o ON-set (r_i) são conjuntos mutuamente disjuntos e portanto a sua reunião é o conjunto de todos os vértices do cubo booleano de dimensão **n**. Então, para cada **i**, os ON-sets de f_i, d_i e r_i dividem o conjunto dos vértices do cubo booleano de dimensão **n** em três grupos. Portanto, a reunião de **f** com **d** e com **r** é uma tautologia.

2.3 Representação algébrica de uma função lógica

A **representação algébrica** de **ff**, $\mathcal{f}$, é um conjunto de **m** expressões booleanas que definem algebricamente as **m** componentes de **ff**. Assim a representação algébrica da componente ff_i, de **ff**, é uma expressão booleana $\mathcal{f}_i$ que vale 1 para todas as entradas pertencentes a X_i^{ON}, que vale 0 para todas as entradas pertencentes a X_i^{OFF} e que vale 0 ou 1 para todas as entradas pertencentes a X_i^{DC} [BRA84].

A representação algébrica de **ff** pode obter-se a partir da tabela de verdade ou da representação geométrica de **ff** (que é a representação que utiliza **m** cubos booleanos num espaço a **n** dimensões).

Genericamente a representação algébrica da componente **i** de **ff** pode ser assim construida:

a) considerar cada linha da tabela de verdade cuja saída é 1 na posição **i**;

b) para cada uma dessas linhas, escrever o produto booleano correspondente às **n** variáveis de entrada, aparecendo a variável complementada se na tabela de verdade lhe corresponder um 0 e não complementada caso lhe corresponda um 1;

c) formar a soma booleana de todos os termos produto obtidos em no passo anterior.

No caso do exemplo descrito pela tabela de verdade atrás referida, obteremos

$$\mathcal{f}_1 = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 + x_1x_2\bar{x}_3$$

$$\mathcal{f}_2 = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 + \bar{x}_1x_2x_3 + x_1x_2\bar{x}_3 + x_1x_2x_3$$

Note-se que seria possível obter outras representações algébricas de uma função lógica. Em particular, temos liberdade de escolha no modo como, na entrada, associamos os DON'T CARE a 1's ou a 0's. E para cada decisão tomada teremos uma diferente representação. Se no nosso exemplo, fizermos **d** de $\mathbf{ff_2}$ igual a 1, a representação algébrica de $\mathbf{ff_2}$, $\mathcal{f}_2$, terá mais um termo, $x_1\bar{x}_2x_3$.

Qualquer representação algébrica de uma função lógica pode ser simplificada usando as regras da álgebra de Boole, de modo a obter-se uma representação mais compacta de **ff**, mas equivalente à inicial. Por exemplo, podemos simplificar $\mathcal{f}_1$ e $\mathcal{f}_2$ e obter

$$\mathcal{f}_1 = \bar{x}_2 + x_1\bar{x}_3$$

$$\mathcal{f}_2 = x_2 + \bar{x}_1\bar{x}_3$$

Notar que, embora estas equações formem um sistema diferente do obtido anteriormente, os dois sistemas são equivalentes pois f_1 e f_2 no segundo só valerão 1 quando f_1 e f_2 no primeiro também forem 1. Então diremos que duas **representações algébricas** são **equivalentes** quando definem a mesma função.

Podemos ainda referir que existe sempre uma correspondência de um para um, entre os possíveis termos produto (que fazem parte da soma de produtos que define **ff**) e os conjuntos de vértices do cubo definido no espaço a **n** dimensões que representa **ff**.

Do nosso exemplo, $f_1 = \bar{x}_2 + x_1\bar{x}_3$, vê-se que:

- $\bar{x}_2$ é uma função lógica que vale 1 nos vértices com coordenadas (0 0 0), (1 0 1), (0 0 1) e (1 0 0) do cubo no espaço a três dimensões; notar que esses vértices formam um plano;

- $x_1\bar{x}_3$ é uma função lógica que vale 1 nos vértices com coordenadas (1 0 0) e (1 1 0) do cubo no espaço a três dimensões; notar que esses vértices formam uma recta.

2.4 Cubos e coberturas

Um **cubo** (num espaço booleano a **n** dimensões) associado a uma função **f** com saídas múltiplas pode ser especificado pelas coordenadas dos seus vértices e por um índice que indica a qual componente de **f** se refere. De um forma compacta, um cubo é um vector.

Consideremos uma função lógica com **n** entradas e **m** saídas representada por uma soma de produtos e refiramo-nos a uma dessas parcelas **p** (produto). Então diremos que um cubo **p** é especificado por um vector $\mathbf{c} = [c_1, c_2, ..., c_n, c_{n+1}, ..., c_{n+m}]$ tal que $\mathbf{c}_i$ toma um dos seguintes valores:

$$c_i = \begin{cases} 0 \text{ se } x_i \text{ aparece complementado em } p \quad (i=1, 2, ..., n) \\ 1 \text{ se } x_i \text{ não aparece complementado em } p \quad (i=1, 2, ..., n) \\ 2 \text{ se } x_i \text{ não aparece em } p \quad (i=1, 2, ..., n) \\ 3 \text{ se } p \text{ não aparece na representação algébrica de } f_{i-n} \\ \quad (i=n+1, ..., n+m) \\ 4 \text{ se } p \text{ aparece na representação algébrica de } f_{i-n} \\ \quad (i=n+1, ..., n+m) \end{cases}$$

Relativamente ao nosso exemplo, se considerarmos $p = x_1\bar{x}_3$, que existe na expressão algébrica de f_1, teremos **c** = [1 2 0 4 3]. Podemos subdividir o vector **c** em dois subvectores; um contendo os **n** primeiros valores de **c** é chamado o **cubo de entradas** ou **parte de entradas** de **c**, **I(c)**; outro contendo os **m** últimos valores é chamado o **cubo de saídas** ou **parte de saídas** de **c**, **O(c)**.

Note-se que o cubo de entradas representa, de uma forma compacta, as coordenadas dos vértices do cubo correspondente ao termo produto considerado. Assim **I(c)** = [1 2 0] identifica os vértices (1 0 0) e (1 1 0) e **O(c)** = [4 3] refere a qual dos **m** espaços o cubo pertence, neste caso f_1.

Uma **cobertura** de uma função lógica **ff** com **n** entradas e **m** saídas é um conjunto de cubos **C** que, relativamente a cada saída **j** (j = 1, 2, ..., m) a que corresponde um 4, contém (nas suas partes de entrada) todos os vértices correspondentes ao ON-set de ff_j e nenhum dos vértices do OFF-set de ff_j. Uma cobertura é portanto uma matriz $C = \{c^1, c^2, ..., c^k\}$ em que **k** representa o número total de parcelas diferentes que formam uma das possíveis representações algébricas de **ff** [BRA84].

Convém notar que existe uma correspondência de um para um, entre um conjunto de cubos e um conjunto de expressões algébricas, assim como entre uma cobertura de **ff** e a representação algébrica de **ff** expressa como soma de produtos. Assim usaremos os termos cobertura de **ff** e representação algébrica de **ff** indistintamente.

Sejam **c** = $\{c_1, c_2, ..., c_{n+m}\}$ e **d** = $\{d_1, d_2, ..., d_{n+m}\}$ dois cubos; então, diz-se que o cubo **c** cobre (ou contém) o cubo **d** ($c \supseteq d$) se cada entrada de **c** contiver a correspondente entrada de **d**; por outro lado, **c** contém estritamente **d** ($c \supset d$) se **c** contém **d** e para pelo menos um **j**, c_j contem d_j.

Geometricamente, **c** conter **d** significa que o cubo de entradas de **c** deve conter todos os vértices que constituem o cubo de entradas de **d**; e **d** apenas pode estar definido em alguns ou eventualmente em todos os espaços booleanos em que **c** existe.

Um **mintermo** é um cubo, portanto um vector, cuja parte de entradas não contém nenhum 2 e cuja parte de saídas só contém um 4 e (m-1) 3's. Notar que para o mintermo **i**, o valor 4 estará na posição **i**. Assim um mintermo pode estar contido num outro cubo, mas um mintermo não contém nenhum outro cubo que não ele próprio. Logo, um mintermo é um elemento de um cubo. Então, cada cubo pode ser decomposto num conjunto de mintermos que são elementos do cubo. Por outras palavras, um mintermo é o produto de todas as variáveis de entrada da função, ou dos seus complementos.

Geometricamente um mintermo é um vértice de um cubo booleano, especificado pela parte de entradas do cubo que representa o mintermo; esse vértice só pode existir em uma das **m** representações das saídas, portanto só pertence a um dos **m** espaços booleanos em que a função é definida.

Assim, uma função de **n** variáveis de entrada é representável por um subconjunto dos 2^n possíveis mintermos da função. E os 2^n possíveis mintermos correspondem aos 2^n vértices do n-cubo booleano definido num espaço a **n** dimensões. Os mintermos são também chamados 0-cubos da função. Então, existe uma correspondência de um para um entre os mintermos de uma função de **n** variáveis de entrada e os vértices do n-cubo booleano correspondente. Refira-se que se dois 0-cubos diferem só em uma coordenada (literal), então formam um 1-cubo. Analogamente, um 2-cubo é a combinação de quatro 0-cubos cujas coordenadas diferem apenas em dois literais.

A **intersecção** ou **produto** de dois cubos **c** e **d** é ainda um cubo **e**, (**e** = **c** ∩ **d** = **c**.**d**), que se obtém de acordo com as seguintes tabelas:

	∩	d_i: 0	1	2
c_i	0	0	ø	0
	1	ø	1	1
	2	0	1	2

$(1 \leq i \leq n)$

	∩	d_i: 3	4
c_i	3	3	3
	4	3	4

$(n < i \leq n+m)$

Note-se que, se qualquer que seja **i**, a intersecção de c_i com d_i é um conjunto vazio (ø), então a intersecção de **c** com **d** também é um cubo vazio. A intersecção de **c** com **d** também será um cubo vazio, se a parte de saídas da intersecção de **c** com **d** só contiver 3's. Por outras palavras, a intersecção de dois cubos é vazia, se as partes de entradas dos dois cubos não têm vértices comuns ou se eles não estão ambos presentes em nenhum dos **m** espaços booleanos a **n** dimensões.

Em termos geométricos, a intersecção de dois cubos é um cubo cuja parte de entradas contém os vértices comuns a **c** e a **d** e cuja parte de saídas especifica que o cubo intersecção está presente nos **m** espaços booleanos a **n** dimensões em que ambos **c** e **d** estão definidos. Dois **cubos** são **ortogonais** se a sua intersecção é vazia.

A intersecção de dois conjuntos de cubos é um conjunto de cubos, que se obtém pela intersecção dois a dois de todos os cubos dos dois conjuntos iniciais. Dois conjuntos de cubos são ortogonais, se a sua intersecção for formada por cubos vazios.

A **reunião** ou **soma** de dois cubos **c** e **d** representa-se por **e**, (**e** = **c** U **d** = **c** + **d**) e é o conjunto dos vértices cobertos pela parte de entradas de **c** e de **d** no espaço booleano a **n** dimensões em que os cubos estão definidos. Logo, a reunião de **c** e **d** é uma matriz formada pelos dois vectores **c** e **d**.

Um cubo **p** é um **implicante** de **ff** = (**f**, **d**, **r**) se a intersecção de **p** com todos os cubos que representam **r** for vazia; ou seja, se a intersecção de **p** com todos os cubos que formam o OFF-set de **ff** for vazia. Voltando ao exemplo que temos vindo a referir, vê-se que o cubo [1 2 0 4 3] que representa a segunda parcela de $\mathcal{f}_1$, é um implicante de **ff**, pois a sua intersecção com **r** = [0 1 2 4 3] é vazia. Contudo o cubo [0 2 1 3 4] já não é implicante de **ff**, pois a sua intersecção com **r** = [0 0 1 3 4] e **r** = [1 0 0 3 4] já não é sempre vazia.

Diz-se que um implicante é **primo** ou é um **primo implicante** se e só se ele não estiver contido em nenhum outro implicante, isto é, se ele for o maior implicante que é possível definir. Portanto, um primo é um implicante que contém o maior número de 2's possível e portanto o menor número de literais. Por exemplo, seja [0 1 2 4 3] o conjunto dos OFF-set de uma dada função, então o cubo [2 0 2 4 3] é primo, enquanto que o cubo [0 0 2 4 3] é um implicante mas não é primo, visto que está contido em [2 0 2 4 3].

Um primo é um **primo essencial** se e só se contém um mintermo de **ff** que não está contido em nenhum outro primo. Recorde-se que um mintermo é um cubo que não contém nenhum 2 na sua parte de entradas. Então, [2 0 2 4 3] é um primo essencial pois contém o mintermo [0 0 1 4 3] que não pertence a nenhum outro primo [BRA84].

Uma **cobertura C** da função **ff** é **mínima** se nenhum subconjunto de **C** for ainda uma cobertura de **ff**, isto é, se **C** for a menor cobertura possível de **ff**. Uma **cobertura** é **prima** se e só se todos os seus cubos são primos.

3. Método de Quine-McCluskey

3.1 Introdução

Este método clássico [MCC65] é basicamente um processo exaustivo e rigoroso que conduz a uma solução óptima para a função a simplificar. Como ponto de partida para a sua utilização é necessária a lista de mintermos que define a função. Caso a função não esteja assim definida deve ser feita uma conversão, por exemplo, de uma tabela de verdade para a correspondente lista de mintermos.

Começa então por se procurar os termos produto que devem fazer parte da função simplificada. Esses termos são chamados primos implicantes. Para determinar os primos implicantes usa-se um processo que compara cada **mintermo** com todos os outros que definem a função. Se dois mintermos diferem numa só variável, remove-se essa variável e obtém-se um termo produto com um literal a menos. Repete-se este processo para todos os mintermos, até que se complete uma procura exaustiva. Os novos termos obtidos por essa simplificação serão agora submetidos ao processo de comparação anteriormente descrito. Este procedimento é efectuado sucessivamente até não se conseguir nenhuma nova eliminação de literais nos termos considerados.

Os termos produto finalmente obtidos, assim como todos os termos que não foi possível simplificar, constituem os **primos implicantes**, que portanto serão formados por um número mínimo de literais. A soma desses primos implicantes define a função minimizada, representada sob a forma de soma de produtos.

3.2 Representação espacial de funções booleanas

Como já foi referido no Capítulo 2, é possível fazer uma representação geométrica das diferentes possíveis combinações de **n** variáveis booleanas, associando essas combinações a pontos num espaço a **n** dimensões. Esses 2^n pontos serão os vértices de um cubo booleano a **n** dimensões (hipercubo booleano).

Para representar funções de **n** variáveis como vértices de um cubo a **n** dimensões, estabelece-se uma relação de um para um, entre os mintermos da função de **n** variáveis e os vértices desse cubo.

Repare-se que estamos apenas a tratar funções de saída única, portanto consideramos cubos a **n** dimensões todos no mesmo e único espaço booleano. A generalização foi apresentada no Capítulo 2. Recorde-se também que os vértices correspondem aos mintermos e podem ser referidos como 0-cubos. Se dois 0-cubos de uma função apenas diferem numa coordenada (um literal), então diz-se que formam um 1-cubo. Analogamente se pode definir o conjunto de quatro 0-cubos como sendo um 2-cubo da função, desde que os 0-cubos difiram entre si apenas em dois literais. Assim e enquanto um 0-cubo é um vértice do cubo, um 1-cubo é uma aresta e um 2-cubo é uma face.

Recorde-se que um n-cubo maior contém um n-cubo menor; ou inversamente, que um n-cubo menor é coberto por um n-cubo maior, como foi referido no Capítulo 2. Por exemplo, o 0-cubo (1 0 0) está contido nos 1-cubos (X 0 0), (1 0 X) e (1 X 0) e no 2-cubo (1 X X); analogamente os 1-cubos (1 X 0), (1 0 X), (1 1 X) e (1 X 1) estão contidos no 2-cubo (1 X X).

3.3 Descrição do método

O método de Quine-McCluskey que, como já foi referido, utiliza como ponto de partida a lista de mintermos que define a função, pode descrever-se resumidamente em seis passos [HIL81]. Consideremos, como exemplificação, a função assim definida:

$$f\ (w, x, y, z) = \sum m(0, 1, 2, 8, 10, 11, 14, 15).$$

1º) Convertem-se os mintermos para a correspondente forma binária e ordenam-se segundo o número de 1's que essa representação binária contém, construindo-se assim uma tabela em que se representam os mintermos pelo seu número em decimal e em binário.

2º) Separam-se os mintermos em grupos de acordo com o número de 1's que possuem, isto para reduzir o número de comparações que é necessário fazer para determinar os 1-cubos a partir dos

mintermos (0-cubos). Se dois mintermos diferem um do outro apenas em uma variável, diz-se que eles combinam e então pode remover-se dos mintermos essa variável. Isto é, dois mintermos combinam se e só se a sua representação binária é idêntica em todos os bits excepto um (esse bit será 0 num dos mintermos e 1 no outro). Assim esses dois mintermos que diferem um do outro apenas num bit, existem na tabela em grupos adjacentes, como se exemplifica na Fig. 3.1.

mintermos (0-cubos)	wxyz	
m0	0000	√
m1	0001	√
m2	0010	√
m8	1000	√
m10	1010	√
m11	1011	√
m14	1110	√
m15	1111	√

(1-cubos)		
0,1	000X	*
0,2	00X0	√
0,8	X000	√
2,10	X010	√
8,10	10X0	√
10,11	101X	√
10,14	1X10	√
11,15	1X11	√
14,15	111X	√

(2-cubos)		
0,2,8,10	X0X0	*
0,8,2,10	X0X0	
10,11,14,15	1X1X	*
10,14,11,15	1X1X	

Fig. 3.1 - Determinação dos primos implicantes.

3º) Partindo da tabela com os mintermos agrupados pelo modo descrito, procura-se agora formar os 1-cubos. Começa então por comparar-se cada mintermo do primeiro grupo com todos os mintermos do grupo imediatamente abaixo. Sempre que dois mintermos são iguais em todas as posições menos numa, devem assinalar-se com um √ e construir-se, na tabela, uma nova coluna onde se coloque o 1-cubo correspondente (que necessariamente terá um X na posição em que os bits não coincidem).

Terminada a comparação de todos os mintermos do primeiro grupo com os do segundo, começa a construir-se outro grupo comparando os mintermos do segundo com os do terceiro grupo pelo processo descrito. Note-se que cada mintermo de um grupo deve ser comparado com todos os mintermos do grupo seguinte, mesmo que um deles ou ambos já tenham sido marcados com um √ e portanto já tenham sido usados na construção de outro 1-cubo. Contudo, não é necessário voltar a assinalar esses mintermos com √.

Se após estas operações, algum ou alguns dos mintermos não tiver sido assinalado com um √, isso significa que ele não foi usado na

construção de nenhum 1-cubo e portanto deve fazer parte da representação final da função como soma de produtos. Esse termo é um primo implicante. Um primo implicante é portanto um cubo que não está contido em nenhum outro cubo da mesma função.

4º) O próximo passo é a obtenção de possíveis pares de 1-cubos para formarem 2-cubos. Repare-se que, de igual modo, os elementos de cada grupo apenas precisam de ser comparados com os elementos do grupo imediatamente abaixo. Contudo dois cubos só são comparáveis se tiverem um X na mesma posição, isto é, se em ambos tiver sido eliminada a mesma variável, durante o passo anterior. Completada a comparação dos 1-cubos verifica-se se existe ou não algum primo implicante nessa coluna, isto é, se algum 1-cubo não foi assinalado com um √. No exemplo em estudo existe um primo implicante (0 0 0 X) = $\bar{w}\bar{x}\bar{y}$.

5º) Finalmente procuram-se as possíveis combinações de 2-cubos para formarem 3-cubos. No nosso caso, como nos dois cubos os X estão em diferentes posições, não há 3-cubos e portanto os 2-cubos são primos implicantes. Assim a função minimizada será a soma dos primos implicantes, isto é, $\mathbf{f = \bar{w}\bar{x}\bar{y} + \bar{x}\bar{z} + wy}$. Contudo, na maioria dos casos, a soma dos primos implicantes não é necessariamente a representação da função com o menor número de termos.

6º) Caso fosse possível formarem-se 3-cubos o processo continuaria de modo análogo ao já descrito. Repare-se que este método de comparação, utilizando a representação binária, é adquado para um tratamento computacional. Como processo manual é moroso e facilmente susceptível de erros, sobretudo quando se comparam 0's e 1's em longas listas de cubos. No entanto, esta dificuldade é reduzida se se usar como método alternativo a notação decimal em vez da binária e depois se proceder de modo análogo ao descrito.

Note-se que cada 1 em binário representa o coeficiente 1 multiplicado por uma potência de dois. Assim, quando dois mintermos diferem apenas num bit, o mintermo que contém um 1 nessa posição é maior, por uma potência de dois, do que o mintermo que contém um 0. Então para que dois mintermos possam ser combinados é preciso que o número do primeiro difira de uma potência de dois do número do segundo que, necessariamente, tem de pertencer ao grupo adjacente e abaixo na tabela.

Começa-se, então, por construir uma tabela com todos os mintermos, representados na notação decimal, agrupados de modo idêntico ao anterior, como se mostra na Fig. 3.2. Em seguida compara-se cada mintermo de um grupo com todos os mintermos do grupo seguinte; se eles diferirem de uma potência de dois, então esses mintermos combinam e formam um 1-cubo. Devem marcar-se ambos esses mintermos com um √ e escrever-se o 1-cubo na coluna correspondente da tabela.

Para representar o 1-cubo, usam-se os números que definem os dois mintermos e entre parêntesis indica-se a potência de dois pela qual eles diferem. Assim, o número entre parêntesis indica a posição do X na notação binária. Referindo-nos ao exemplo anterior poderíamos construir a seguinte tabela:

mintermos (0-cubos)	
0	√
1	√
2	√
8	√
10	√
11	√
14	√
15	√

(1-cubos)		
0,1	(1)	*
0,2	(2)	√
0,8	(8)	√
2,10	(8)	√
8,10	(2)	√
10,11	(1)	√
10,14	(4)	√
11,15	(4)	√
14,15	(1)	√

(2-cubos)		
0,2,8,10	(2,8)	*
0,8,2,10	(2,8)	
10,11,14,15	(1,4)	*
10,14,11,15	(1,4)	

Fig. 3.2 - Determinação dos primos implicantes (notação decimal).

Note-se que a comparação entre os 1-cubos de grupos adjacentes só pode ser feita relativamente aos termos que têm igual número entre parêntesis, isto é, que têm o X em igual posição. Também o par de números que representa o 1-cubo de um grupo deve diferir de uma potência de dois do par de números que representa o 1-cubo do grupo adjacente e além disso os números correspondentes ao grupo inferior devem ser maiores do que os números do grupo superior. Só assim os dois 1-cubos poderão combinar, formando os 2-cubos. Estes serão representados na próxima coluna da tabela pelos quatro números que correspondem aos dois 1-cubos, acrescidos dos dois números que indicam as posições dos X, entre parêntesis.

Como já vimos, todos os elementos da tabela que não têm um √ são primos implicantes que, como é evidente, são os mesmos que anteriormente tínhamos obtido, só que agora estão representados em notação decimal.

Assim, [0, 1 (1)] corresponde a (0 0 0 0) e a (0 0 0 1) ou seja ao 1-cubo (0 0 0 X) que é equivalente a $\bar{w}\bar{x}\bar{y}$; [0, 2, 8, 10 (2, 8)] corresponde a (0 0 0 0), a (0 0 1 0), a (1 0 0 0) e a (1 0 1 0) ou seja aos 1-cubos (0 0 X 0) e (1 0 X 0) e portanto ao 2-cubo (X 0 X 0) que é equivalente a $\bar{x}\bar{z}$; analogamente se obteria o terceiro primo implicante wy; logo $f = \bar{w}\bar{x}\bar{y} + \bar{x}\bar{z} + wy$.

Mais uma vez se faz notar que esta expressão, que é a soma dos primos implicantes que cobrem todos os mintermos de **f**, não é necessariamente a representação mínima de **f** como uma soma de produtos. Embora a representação mínima deva ser uma soma de primos implicantes, eventualmente só de alguns, os **primos implicantes essenciais**. Recorde-se que um primo implicante é essencial se e só se contém um mintermo da função que não está contido em nenhum outro primo.

Seja a função

$$f\ (w, x, y, z) = \sum m(1, 4, 6, 7, 8, 9, 10, 11, 15)$$

cujos primos implicantes se representam na tabela da Fig. 3.3.

Representação decimal	Representação simplificada	Representação binária (w x y z)	Termo produto
1,9 (8)	a	x 0 0 1	$\bar{x}\bar{y}z$
4,6 (2)	b	0 1 x 0	$\bar{w}x\bar{z}$
6,7 (1)	c	0 1 1 x	$\bar{w}xy$
7,15 (8)	d	x 1 1 1	xyz
11,15 (4)	e	1 x 1 1	wyz
8,9,10,11,(1,2)	f	1 0 x x	$w\bar{x}$

Fig. 3.3 - Determinação dos primos implicantes.

Para se encontrarem os primos implicantes que devem, necessariamente, fazer parte da representação da função na sua forma simplificada, começa por se construir uma tabela que permite fazer a selecção do conjunto óptimo de primos implicantes.

Cada coluna dessa tabela, como se mostra na Fig. 3.4, representa um mintermo e cada linha um primo implicante, sendo estes agrupados segundo o número de literais do produto que representam.

Primos Implicantes	Mintermos 1	4	6	7	8	9	10	11	15
* a	√					√			
* b		√	√						
c			√	√					
d				√					√
e								√	√
* f					√	√	√	√	
	√	√	√		√	√	√	√	

Fig. 3.4 - Tabela de primos implicantes.

Para evidenciar a composição de cada primo implicante, coloca-se um √ nas colunas correspondentes aos mintermos contidos nesse primo. Depois procede-se à selecção do número mínimo de primos implicantes que cubram todos os mintermos da função. De seguida procuram-se colunas com uma única marca √. Isso significa que o correspondente primo implicante é o único a conter o mintermo referido.

No nosso exemplo, o mintermo $\mathbf{m_1}$ é coberto pelo primo implicante $\bar{x}\bar{y}z$, isto é, a selecção do primo implicante $\mathbf{\bar{x}\bar{y}z}$ garante que $\mathbf{m_1}$ está incluido na representação da função. Analogamente $\mathbf{m_4}$ é coberto por $\mathbf{\bar{w}x\bar{z}}$ e $\mathbf{m_8}$ e $\mathbf{m_{10}}$ por $\mathbf{w\bar{x}}$. Os primos implicantes que cobrem os mintermos cuja coluna contém uma única marca √ são chamados primos implicantes essenciais. Esses devem

necessariamente ser incluidos na representação final da função para garantir que todos os mintermos nela estão contidos.

Então esses primos essenciais são marcados com um * numa nova coluna à esquerda na tabela; é também criada uma última linha onde se indica com um √ quais os mintermos que são cobertos pelos primos implicantes essenciais. Por exemplo, o primo $\bar{x}\bar{y}z$ cobre m_1 e m_9; enquanto que $\bar{w}x\bar{z}$ cobre m_4 e m_6 e $w\bar{x}$ cobre m_8, m_9, m_{10} e m_{11}. Inspeccionando a última linha da tabela pode verificar-se que os primos implicantes essenciais cobrem todos os mintermos excepto m_7 e m_{15}.

Se todos os mintermos estivessem contidos nos primos implicantes essenciais, então a soma desses termos seria a representação mínima da função como soma de produtos. No nosso exemplo teremos também de incluir, na soma de produtos final, os mintermos m_7 e m_{15}. É evidente, que neste caso o primo implicante xyz cobre m_7 e m_{15} e então este primo também tem de ser seleccionado. Mas caso não fosse tão óbvia a sua detecção, construia-se uma nova tabela, agora reduzida, como se ilustra na Fig. 3.5.

	Primos Implicantes	Mintermos 7	15
	$\bar{w}xy$	√	
**	xyz	√	√
	wyz		√

Fig. 3.5 - Tabela reduzida de primos implicantes.

Saliente-se que não haveria qualquer vantagem em usar $\bar{w}xy$ pois só cobre m_7, ou usar wyz que só cobre m_{15}, enquanto que xyz cobre m_7 e m_{15}. Sendo assim, removem-se da tabela os primos wxy e $\bar{w}xy$, pois isso não nos impedirá de encontrar a soma de produtos mínima.

O primo xyz é, então, assinalado na tabela reduzida com ** para indicar que é um primo implicante secundariamente essencial ou

necessário; pode dizer-se que o primo implicante **xyz** é dominante relativamente aos outros dois (**wyz** e $\overline{\mathbf{w}}$**xy**).

Repare-se que se na tabela reduzida aparecessem dois diferentes primos implicantes cobrindo os mesmos mintermos, bastaria usar um deles indiferentemente, podendo o outro ser suprimido da tabela.

Se, após a obtenção dos primos implicantes secundariamente essenciais através da tabela reduzida, ainda existissem vários mintermos por cobrir, poder-se-ia aplicar novamente o processo, isto é, construir uma nova tabela reduzida de onde se removeriam agora os primos implicantes secundariamente essenciais e proceder de modo semelhante ao já descrito.

Pode ainda acontecer que a tabela reduzida seja como a representada na Fig. 3.6.

Primos Implicantes	Mintermos 6	7	15	38	46	47
a	√	√				
b			√			√
c		√	√			
d					√	√
e				√	√	
f	√			√		

Fig. 3.6 - Tabela reduzida de primos implicantes.

Esta tabela já não pode ser mais simplificada, pois nenhuma coluna tem uma única marca √, nem nenhuma linha é dominante relativamente a qualquer outra. Então utilizando o **Método de Petrick** [PET59] pode obter-se uma selecção final (mínima) de primos implicantes como se descreve a seguir.

As letras que representam os primos implicantes desta tabela reduzida são interpretadas como variáveis booleanas, que portanto tomarão o valor 1 se o correspondente primo implicante for

seleccionado e o valor 0 caso contrário. Assim relativamente à primeira coluna, ou **a** ou **f** tem de ser escolhido para que $\mathbf{m}_6$ seja coberto, logo escreve-se (**a** + **f**) = 1. Analogamente, ou **a** ou **c** terá de ser seleccionado para cobrir $\mathbf{m}_7$; portanto obtém-se (**a** + **f**) (**a** + **c**) = 1.

Procedendo deste modo, constroi-se um produto de somas, cada soma sendo formada pelos primos implicantes que contêm um determinado mintermo. A expressão final será

$$(a + f)\ (a + c)\ (b + c)\ (e + f)\ (d + e)\ (b + d) = 1.$$

Aplicando as propriedades da álgebra de Boole obtém-se

$$abe + abdf + acde + bcef + cdf = 1.$$

Para que esta última equação seja verdade, pelo menos um dos cinco produtos tem de ser igual a 1. Como se pretende a solução mínima, opta-se por escolher um dos produtos **abe** ou **cdf**, pois correspondem apenas a três primos implicantes. Destes dois produtos selecciona-se o que corresponder ao menor número de variáveis de entrada. Então, os primos implicantes contidos nesse produto seleccionado são incluidos (juntamente com os primos implicantes essenciais) na soma de produtos que define a função na sua forma simplificada.

O método de Quine-McCluskey pode ser ligeiramente modificado para se utilizar na simplificação de funções com termos don't care. Assim, os don't care são incluidos na lista de mintermos usada para se fazer a determinação dos primos implicantes, que deste modo são determinados com o menor número de literais possível.

Contudo, ao construir-se a tabela de primos implicantes, apenas os mintermos são usados (e não os mintermos mais os don't care), pois os termos don't care não têm de ser cobertos pelos primos implicantes seleccionados. Veja-se o seguinte exemplo, em que se quer determinar a representação mínima de

$$f\ (A, B, C, D, E) = \Sigma\ m(1, 4, 7, 14, 17, 20, 21, 22, 23) + \Sigma\ d(0, 3, 6, 19, 30)$$

Para a obtenção dos n-cubos, listam-se então os mintermos e os don't care, como se exemplifica na Fig. 3.7.

mintermos (0-cubos)	
0	√
1	√
4	√
3	√
6	√
17	√
20	√
7	√
14	√
19	√
21	√
22	√
23	√
30	√

	(1-cubos)		
*	0,1	(1)	h
*	0,4	(4)	i
	1,3	(2)	√
	1,17	(16)	√
	4,6	(2)	√
	4,20	(16)	√
	3,7	(4)	√
	3,19	(16)	√
	6,7	(1)	√
	6,14	(8)	√
	6,22	(16)	√
	17,19	(2)	√
	17,21	(4)	√
	20,21	(1)	√
	20,22	(2)	√
	7,23	(16)	√
	14,30	(16)	√
	19,23	(4)	√
	21,23	(2)	√
	22,23	(1)	√
	22,30	(8)	√

	(2-cubos)		
*	1,3,7,19	(2,16)	a
*	4,6,20,22	(2,16)	b
*	3,7,19,23	(4,16)	c
*	6,7,22,23	(1,16)	d
*	6,14,22,30	(8,16)	e
*	17,19,21,23	(2,4)	f
*	20,21,22,23	(1,2)	g

Fig. 3.7 - Determinação dos primos implicantes.

Contudo, na tabela dos primos implicantes, mostrada na Fig. 3.8, representam-se apenas os mintermos.

Primos Implicantes	1	4	7	14	17	20	21	22	23
* * a	√				√				
* * b		√				√		√	
c			√						√
* * d			√					√	√
* e				√				√	
* * { f					√		√		√
{ g						√	√	√	√
h	√								
i		√							
	√	√	√	√	√	√	√	√	√

(Mintermos)

Fig.3.8 - Tabela de primos implicantes.

Notar que para cobrir m_{21} basta usar **f** ou **g**. Da tabela obtém-se

$$f(A, B, C, D, E) = e + a + b + d + (f \text{ ou } g),$$

onde o primo implicante **e** é essencial, **a**, **b** e **d** secundariamente essenciais e **f** e **g** opcionais; ou seja

$$f(A, B, C, D, E) = C D \bar{E} + \bar{B} \bar{C} E + \bar{B} C \bar{E} + \bar{B} C D + (A \bar{B} E \text{ ou } A \bar{B} C)$$

Para se obter esta expressão a partir da anterior, procede-se do seguinte modo. Por exemplo, para determinar a expressão correspondente ao primo implicante **e**, que é representado por [6, 14, 22, 30 (8, 16)], escolhe-se um dos mintermos que o definem, seja o 6, e converte-se 6 em binário (0 0 1 1 0); não tomando em consideração as posições correspondentes aos números entre parêntesis, portanto 8 e 16, pois correspondem a variáveis que foram eliminadas, obtém-se (- - 1 1 0), logo $\mathbf{C D \bar{E}}$.

Uma outra adaptação do método pode ser efectuada de modo a possibilitar a simplificação de funções definidas como **produtos de somas**. Apenas são necessárias duas modificações. São

listados os maxtermos (e não os mintermos), portanto os 0's da função e também os don't care, caso existam. Depois procede-se do modo já descrito e finalmente obtêm-se as somas dos complementares das variáveis de entrada, de que em seguida se calcula o complementar.

Refira-se que, como se viu, o método de Quine-McCluskey é aplicável apenas a funções representadas numa das formas standard de somas de produtos ou de produtos de somas, o que poderia ser interpretado como um condicionamento à sua utilização. Contudo, como na maioria das aplicações é usada a forma standard, não se torna relevante esse problema.

3.4 Circuitos de saídas múltiplas

Até aqui estivemos a estudar um processo de minimizar uma função de saída única, mas frequentemente o que é necessário realizar é a simplificação de funções de saídas múltiplas.

Então, pode considerar-se uma função de saídas múltiplas como um conjunto de várias funções de saída única e tratá-las pelas técnicas já referidas. Contudo, verifica-se que se obtém considerável economia a nível da implementação, se for possível efectuar-se uma partilha de elementos entre as várias funções.

Como é lógico, será de todo o interesse detectar produtos comuns às várias funções e utilizar as simplificações que conduzam às realizações óptimas das funções, isto é, às implementações de todas as funções recorrendo ao menor número possível de elementos.

Suponhamos que queríamos simplificar as três funções seguintes:

$$f_{\alpha}(A, B, C, D) = \sum m(2, 4, 10, 11, 12, 13)$$

$$f_{\beta}(A, B, C, D) = \sum m(4, 5, 10, 11, 13)$$

$$f_{\gamma}(A, B, C, D) = \sum m(1, 2, 3, 10, 11, 12)$$

Para se determinarem os primos implicantes correspondentes às três funções, procede-se essencialmente de modo análogo ao anterior e acrescentam-se três importantes condições [HIL81].

Começa por se listar todos os mintermos das várias funções, ordenados pelo número de 1's da sua representação binária, independentemente da função a que pertencem. A cada mintermo acrescenta-se uma etiqueta com a indicação de qual ou quais as funções a que correspondem. O processo é ilustrado na Fig. 3.9.

	mintermos (0-cubos)		
	1	γ	√
	2	αγ	√
•	4	αβ	i
	3	γ	√
	5	β	√
	10	αβγ	√
•	12	αγ	j
	11	αβγ	√
•	13	αβ	k

	(1-cubos)		
	1,3(2)	γ	f
	2,3(1)	γ	√
•	2,10(8)	αγ	g
•	4,5(1)	β	d
•	4,12(8)	α	b
	3,11(8)	γ	√
	5,13(8)	β	e
•	10,11(1)	αβγ	h
•	12,13(1)	α	c

	(2-cubos)		
•	2,3,10,11(1,8)	γ	a

Fig. 3.9 - Determinação dos primos implicantes de uma função de saídas múltiplas.

Para se poderem formar os 1-cubos é necessário, além das condições habituais, verificar-se que:

1º) só se podem combinar dois cubos, se existir pelo menos uma letra comum nas suas etiquetas; isto é, não se pode combinar um cubo de uma função com um cubo de uma outra função. Assim $\mathbf{m}_1$ de $\mathbf{f}_\gamma$ e $\mathbf{m}_5$ de $\mathbf{f}_\beta$, embora possuam as condições gerais para poderem combinar, não pertencem à mesma função e portanto não combinam;

2º) quando os dois cubos combinam, a etiqueta do cubo resultante é formada apenas pelas letras comuns às etiquetas dos dois cubos originais. No exemplo considerado, quando se forma o 1-cubo [5, 13 (8)], a etiqueta é β, pois é a única letra comum na etiqueta de $\mathbf{m}_5$ e de $\mathbf{m}_{13}$;

3º) após os dois cubos terem sido combinados, coloca-se uma marca √ naquele ou naqueles cujas etiquetas aparecem totalmente na etiqueta do cubo maior. Assim referindo-nos à combinação de

m_5 e m_{13} para formarem o 1-cubo [5, 13 (8) β], apenas marcamos m_5 como já tendo combinado e não m_{13}, pois embora o mintermo 13 tenha combinado para formar um 1-cubo de f_β, não combinou em f_α e é portanto um primo implicante de (αβ).

Com base nestas regras obtém-se o conjunto de primos implicantes assinalados com um • na tabela da Fig. 3.9. Agora é necessário construir a tabela de primos implicantes e obter um forma simplificada para as três funções. As colunas correspondem a todos os mintermos de cada função. As linhas são formadas pelos primos implicantes ordenados segundo o número do n-cubo a que pertencem, isto é, segundo o número de literais que os definem. Na tabela, representada na Fig. 3.10, é também indicada a função ou funções em que cada primo implicante aparece.

		f_α						f_β					f_γ					
Primos Implicantes		2	4	10	11	12	13	4	5	10	11	13	1	2	3	10	11	12
γ	a													√	√	√	√	
α	b		√			√												
α	c					√	√											
β	d							√	√									
β	e								√			√						
* γ	f												√		√			
* αγ	g	√		√										√		√		
* αβγ	h			√	√					√	√					√	√	
αβ	i		√					√										
* αγ	j					√												√
αβ	k						√					√						
		√		√	√	√				√	√		√	√	√	√	√	√

Fig. 3.10 - Tabela de primos implicantes.

Em cada linha assinalam-se os mintermos que constituem cada primo implicante, por exemplo, o primo implicante a = [2, 3, 10, 11 (18) γ] só irá ter marcas √ na função γ e nos mintermos 2, 3, 10 e 11. Depois procuram-se os primos implicantes essenciais, pesquisando quais as colunas com um único √. Neste caso os primos implicantes essenciais são (f, g, h, j), assinalados na Fig. 3.10 com um *. Note-se, contudo, que estes não cobrem todos os mintermos das três funções,

portanto é necessário construir-se uma tabela de primos implicantes reduzidos para se tentar obter os primos implicantes secundariamente essenciais ou necessários, como se mostra na Fig. 3.11.

Primos Implicantes		f_α		f_β		
		4	13	4	5	13
α	b	√				
α	c		√			
β	d			√	√	
β	e				√	√
$\alpha\beta$	i	√		√		
$\alpha\beta$	k		√			√

Fig. 3.11 - Tabela reduzida de primos implicantes.

Agora detectam-se as linhas dominantes no intuito de reduzir a tabela. Veja-se que, ao procurar-se uma linha dominante, ela deve ser dominante para toda a tabela e não apenas para uma função. Isto quer dizer que, por exemplo, a linha **d** que seria dominante relativamente à linha **i** se só se considerasse $\mathbf{f}_\beta$, deixa de o ser ao pensar-se em toda a tabela. Portanto neste caso não podemos eliminar nenhuma linha por este processo, logo não há primos implicantes secundariamente essenciais.

Aplique-se, então, o método de Petrick e obtenha-se um produto de somas. Cada soma define os primos implicantes que cobrem cada um dos mintermos que não foram cobertos pelos primos implicantes essenciais já detectados. Neste caso encontra-se

$$(b + i)\ (c + k)\ (d + i)\ (d + e)\ (e + k) = 1$$

e aplicando as leis da álgebra de Boole, obtém-se

$$c\,e\,i + b\,c\,d\,e + e\,i\,k + d\,i\,k + b\,d\,k = 1.$$

Cada produto representa um conjunto de primos implicantes suficiente para cobrir todos os mintermos que não foram cobertos pelos primos implicantes essenciais. O que se pretende agora é

seleccionar o conjunto a que corresponda o menor número de literais.

Como na soma de produtos obtida cada letra representa um primo implicante, convém considerar apenas os produtos com o menor número de elementos, ou seja, (cei), (bdk), (dik) e (eik). Os dois primeiros conjuntos representam dois 1-cubos e um 0-cubo, enquanto que os dois últimos formam um 1-cubo e dois 0-cubos.

Então, como (cei) e (bdk) contêm menor número de literais, dado que correspondem a dois 1-cubos e um 0-cubo, escolhe-se arbitrariamente um deles, por exemplo (cei). Assim o conjunto total de primos implicantes será: $c(\alpha)$, $e(\beta)$, $f(\gamma)$, $g(\alpha\partial)$, $h(\alpha\beta\gamma)$, $i(\alpha\beta)$, $j(\alpha\gamma)$ e destes determina-se o conjunto requerido para definir cada função.

Note-se que se um determinado primo implicante só se refere a uma dada função, deve necessariamente ser usado na representação dessa função. Se por outro lado, o primo implicante pertence a mais do que uma função, deve ser analisado cuidadosamente.

Assim, veja-se o caso do primo implicante $\mathbf{j} = \mathbf{m}_{12}$ que pertence a $\mathbf{f}_\alpha$ e $\mathbf{f}_\gamma$. Relativamente a $\mathbf{f}_\gamma$, o primo implicante **j** é essencial e portanto tem de pertencer à forma simplificada de $\mathbf{f}_\gamma$. Quanto a $\mathbf{f}_\alpha$, repare-se que $\mathbf{m}_{12}$ é coberto, por exemplo, por c = [12, 13 (1)]; logo **j** é redundante relativamente a $\mathbf{f}_\alpha$ e portanto não pertence à forma simplificada da função α.

Com base nestas considerações, obtém-se

$$f_\alpha = c + g + h + i$$

$$f_\beta = e + h + i$$

$$f_\gamma = f + g + h + j$$

ou, procedendo do modo anteriormente descrito

$$f_\alpha = A\,B\,\overline{C} + \overline{B}\,C\,\overline{D} + A\,B\,C + \overline{A}\,B\,C\,\overline{D}$$

$$f_\beta = A\,\overline{B}\,C + B\,\overline{C}\,D + \overline{A}\,B\,\overline{C}\,\overline{D}$$

$$f_\gamma = \overline{A}\,\overline{B}\,D + \overline{B}\,C\,\overline{D} + A\,\overline{B}\,C + A\,B\,\overline{C}\,\overline{D}$$

Repare-se que o termo $\mathbf{\overline{B}C\overline{D}}$ é comum a α e γ; $\mathbf{A\overline{B}C}$ é comum a α, β e γ; e $\mathbf{\overline{A}B\overline{C}\overline{D}}$ é comum a α e β. Assim ao ser realizada a implementação destas funções é possivel efectuar-se uma partilha de elementos entre elas, o que conduz a uma considerável economia.

3.5 Conclusão

Como já foi referido, este método é rigoroso e exaustivo e garante a obtenção de uma representação mínima da função booleana considerada. Contudo, quando o número de variáveis de entrada é elevado (cerca de 16), mesmo utilizando um bom programa de computador para realizar a simplificação da função surgem problemas e isto devido aos elevados tempos de execução do programa e ao espaço de memória que é necessário utilizar, já que o método é exaustivo. Assim, é pouco provável a obtenção de um algoritmo exacto que seja eficiente.

Uma outra eventual desvantagem do método tabular de Quine-McCluskey é a possível ocorrência de erros quando se comparam duas longas listas de mintermos. Então, o método dos mapas de Karnaugh poderia parecer preferível; mas como já foi indicado, para funções de mais de 5 variáveis não se pode ter a certeza de se estar a obter a expressão simplificada ideal.

4. O Programa EXPRESSO-II

4.1 Introdução

O programa Expresso-II [BRA84] é uma ferramenta de minimização de funções lógicas cujos principais objectivos são a simplificação de funções usando o menor número possível de recursos computacionais, bem como a optimização dos resultados finais mediante a utilização de heurísticas eficientes.

As entradas do programa são as coberturas do ON-set e do DON'T CARE-set, respectivamente **F** e **D**, de uma função booleana não completamente especificada **ff**. Como opção, as entradas também poderão ser **F** e **R**, as coberturas do ON-set e do OFF-set. A saída será uma cobertura minimizada.

Fundamentalmente este programa minimiza:

NPT - o número de termos produto, isto é, o número de cubos que formam a cobertura da função (portanto minimiza o número de linhas da matriz);

NLI - o número de literais (que não 2's) existentes no cubo de entradas da cobertura;

NLO - o número de literais no cubo de saídas.

O processo de minimização utilizado pelo programa Expresso-II, define uma função vector objectivo ϕ = (NPT, NLI, NLO) e um ciclo de simplificações que serão executadas até que nenhuma das três componentes de ϕ seja minimizada em duas sucessivas passagens no ciclo.

Antes de se iniciar a execução do programa de minimização é usado o preprocessador UNWRAP. Este procedimento é aplicado aos dados e procura quais os cubos de entrada que correspondem a mais do que uma saída (isto é, cuja parte de saída contém mais do que um 4); se um desses cubos alimentar, por exemplo, **k** saídas, será substituido por **k** cubos, cada um alimentando uma saída. Embora possa parecer que a cobertura resultante é pior do que a inicial do ponto de vista da minimização, de facto o que obtemos é

um ponto de partida para a futura simplificação que é menos rígido; em particular, e deste modo, é possivel no procedimento EXPAND decidir-se quais as melhores opções em termos de associações de saídas.

Vejamos como actuam e interactuam as várias rotinas que compõem o programa Expresso-II.

COMPLEMENT

Calcula **R** (o OFF-set da função) se **F** e **D** forem dados como entrada do programa ou calcula **D** caso **F** e **R** sejam os dados. O complemento **R** é usado posteriormente para facilmente se detectar se um dado cubo é ou não um implicante. Recorde-se que um implicante é um cubo cuja intersecção com o OFF-set (**R**) da função é o conjunto vazio. A selecção dos primos implicantes é feita no procedimento EXPAND.

EXPAND

Substitui os cubos de **F** por primos implicantes, expandindo-os (isto é, passando os 0's e os 1's para 2's sem que contudo haja intersecções com o OFF-set). Como aumenta o número de 2's na parte de entrada de **F**, diminui aí o número de literais. De seguida reduz a cobertura obtida a uma cobertura mínima (portanto reduz o número de cubos em **F**). Assim, atingem-se simultaneamente dois dos objectivos de minimização propostos. Terminado este procedimento substitui-se a cobertura **F** de **ff** por uma cobertura prima.

ESSENTIAL_PRIMES

Localiza os primos essenciais, isto é, os primos que têm necessariamente de aparecer na cobertura **F** porque contêm mintermos que apenas existem nesses cubos. Uma vez que esses primos têm de aparecer na cobertura **F**, logo que são localizados são retirados de **F** e adicionados a **D**. Deste modo os primos essenciais não são alterados desnecessariamente durante o ciclo de minimização. Notar que este procedimento apenas é executado na primeira vez que o programa percorre o ciclo de minimização.

IRREDUNDANT_COVER

Subdivide a cobertura obtida em três grupos: subcobertura relativamente essencial, subcobertura parcialmente redundante e subcobertura totalmente redundante. Os cubos pertencentes a este último grupo são extraídos da cobertura inicial. Então, o subconjunto dos cubos relativamente essenciais, juntamente com uma parte do subconjunto formado pelos cubos parcialmente redundantes e com o conjunto **D** dos DON'T CARES, formam uma cobertura para todos os mintermos de **F**. Após a execução do procedimento IRREDUNDANT_COVER, a cobertura **F** é uma cobertura mínima de primos implicantes.

REDUCE

Considera, (em sequência) cada cubo **c** pertencente a **F** e redu-lo ao menor cubo $\underline{c}$ que contém todos os mintermos de **c** que não pertencem a [(**F** - {**c**}) U **D**] e em seguida faz **F** = [(**F** - {**c**}) U {$\underline{c}$}]. Obtém-se assim uma cobertura não prima, na qual muitos dos cubos originais têm agora tamanho menor. Esses cubos, ao serem novamente expandidos (na próxima passagem no ciclo de minimização) originam dois tipos de vantagens. Primeiro, um cubo menor pode, geralmente, ser expandido em mais direcções do que um cubo maior. Segundo, quanto menores forem os cubos, mais facilmente serão cobertos pelas expansões de outros cubos. Notar que uma vez que este procedimento considera os cubos sequencialmente, as maiores ou menores reduções obtidas dependem de uma prévia ordenação heurística desses cubos. No fim de executado REDUCE, cada implicante foi reduzido a um primo implicante essencial mínimo.

Os procedimentos EXPAND, IRREDUNDANT_COVER e REDUCE fazem parte de um ciclo de minimização que só termina quando, entre uma passagem e a anterior, não tiver ocorrido nenhuma simplificação.

LAST_GAP

Este procedimento é semelhante ao REDUCE, mas o processo de redução usa uma estratégia diferente, uma vez que a ordem pela qual são tratados os cubos não é sequencial. Para cada cubo $\mathbf{c}^i$ pertencente à cobertura **F**, calcula-se $\underline{c}^i$, isto é, o menor cubo contendo a parte de **c** que não é coberta por [(**F** - {$\mathbf{c}^i$}) U **D**]. Como

os cubos são reduzidos individualmente e não em sequência, o resultado não depende do modo como os cubos estão ordenados. Deve notar-se que os cubos {$\underline{c}^i$} podem não formar uma cobertura de **ff**. Usa-se agora uma variante de EXPAND na tentativa de encontrar pelo menos um primo que cubra no mínimo um dos cubos reduzidos {$\underline{c}^i$}. O primo ou primos que obedecem a esta regra são agrupados numa cobertura **H** e então **F** é substituido por IRREDUNDANT_COVER (**F** U **H**). Verifica-se que, sempre que **H** não é vazio, após esta simplificação se obtém uma diminuição do | **F** |.

MAKE_SPARSE

Começa por retirar os primos essenciais do DON'T CARE-set **D** e colocá-los novamente na cobertura **F**. O número de cubos que formam a cobertura **F** é agora o mínimo possível; resta agora reduzir o número de literais. Obtém-se assim uma cobertura final mínima, no sentido de que mais nenhum termo produto ou literal de entrada ou de saída pode dela ser removido sem que **F** deixe de ser cobertura de **ff**.

O fluxo de execução do Expresso-II pode ser sintetizado do seguinte modo:

UNWRAP e COMPLEMENT constroem as coberturas **F**, **R** e **D** (do ON-set, OFF-set e DON'T CARE-set de **ff**).

São usados quatro vectores ϕ_1, ϕ_2, ϕ_3 e ϕ_4 para se verificar a qualidade da cobertura, após a execução, respectivamente, de cada um dos seguintes procedimentos: EXPAND, IRREDUNDANT_COVER, REDUCE e LAST_GAP. Estes vectores são inicialmente carregados com o número de termos produto, o número de literais de entrada e o número de literais de saída correspondentes à cobertura **F** inicial.

O ciclo principal do programa executa EXPAND, seguido de IRREDUNDANT_COVER e REDUCE e no final de cada procedimento verifica-se se a qualidade da cobertura melhorou (usando os vectores ϕ_1, ϕ_2, ϕ_3); se não ocorreu nenhuma redução da cobertura, inicia-se então LAST_GAP que procura encontrar algum primo que possa ainda usar-se para reduzir a cobertura. Se for encontrado algum cubo nessas condições, volta-se ao ciclo principal e processa-se a simplificação da cobertura; caso

contrário executa-se MAKE_SPARSE e termina o programa fornecendo para o exterior uma cobertura mínima de **ff**.

4.2 O Procedimento COMPLEMENT

Este procedimento calcula o complementar de funções com saídas múltiplas, com base no cálculo do complementar de funções de saída única.

No Apêndice A é referido como calcular o complementar de uma função booleana usando recursivamente a expansão de Shannon

$$\overline{f} = x_j . \overline{f}_{x_j} + \overline{x_j} . \overline{f}_{\bar{x}_j}$$

Para calcular o complementar de uma função monótona de saídas múltiplas optou-se então por concatenar os complementares das várias funções de saída única em que se decompôs a função inicial, fundamentalmente por duas razões.

Primeiro, verificou-se que a complementação de uma função de saídas múltiplas raramente poupa tempo de cálculo e muitas vezes conduz a enormes necessidades de utilização do CPU.

Segundo, o cálculo feito a partir das funções com uma única saída permite mais eficiência e uniformidade na utilização do procedimento EXPAND, que usa a complementação. Contudo é evidente que a utilização de funções de saída única conduz a maiores necessidades de armazenamento, pois haverá termos produtos que aparecerão repetidas vezes.

O procedimento COMPLEMENT (**F**, **D**) calcula o complementar de uma função de saídas múltiplas, começando por extrair as várias funções de saída única que são equivalentes à inicial, assim como o correspondente DON'T CARE-set para cada uma delas. Posteriormente calcula o complementar de cada uma dessas funções de saída única. No Apêndice B é indicado o processo de cálculo do complementar.

Assim, no final de COMPLEMENT, obtêm-se os OFF-sets (**R**) de todas as funções de saída única que, após concatenação, formarão o complementar da função de saídas múltiplas.

4.3 O Procedimento TAUTOLOGIA

A classificação de uma dada função booleana como sendo ou não uma tautologia (f = 1) é uma questão fundamental requerida pelo programa Expresso-II. O cálculo da tautologia é parte essencial de vários procedimentos (ESSENTIAL_PRIMES, IRREDUNDANT_COVER, REDUCE e LAST_GAP), pelo que é necessário que esse algoritmo seja eficiente.

O procedimento TAUTOLOGIA começa por testar a existência de alguns casos especiais que permitam conclusões imediatas. Se não existirem casos especiais, é seleccionada uma variável de participação de x_j (definida no Apêndice B) e são determinadas as matrizes que representam os cofactores F_{xj} e $F_{\bar{x}j}$ da função booleana (definidos no Apêndice A). Notar que **F** só será tautologia se F_{xj} e $F_{\bar{x}j}$ o forem. Então o procedimento TAUTOLOGIA é recursivamente chamado com argumentos F_{xj} e $F_{\bar{x}j}$ e só termina quando é detectado um dos casos especiais (isto é, quando se atinge uma folha da árvore binária correspondente).

Note-se que cada nó dessa árvore binária é representado pela matriz de um cofactor; cada ramo da árvore é um conjunto de variáveis de partição que corresponde ao caminho desde o primeiro nó até ao último dessa sequência; cada próximo nó ou é uma folha (nó terminal) e então é verificado se a função é uma tautologia ou tem dois descendentes F_{xj} e $F_{\bar{x}j}$ aos quais o procedimento TAUTOLOGIA é aplicado recursivamente.

No caso da função booleana ser de múltiplas saídas, esta só será uma tautologia se cada um das funções booleanas componentes também for uma tautologia.

4.4 O Procedimento EXPAND

Este procedimento transforma uma dada cobertura **F** de **ff** numa cobertura prima e simultaneamente o menor possível.

Para atingir tal objectivo optou-se por tratar os cubos de **F** um a um sequencialmente e maximizar o número de cubos que, em cada passo, podem ser cobertos por uma dada expansão (substituição de cada cubo **c** de **F** pelo correspondente cubo primo c^+ que contém ou

é igual a **c**). Em seguida são removidos de **F** todos os cubos cobertos por $\mathbf{c}^+$. Assim a cobertura final, além de prima, será mínima.

Contudo e logicamente, o resultado de EXPAND vai depender da ordem pela qual os cubos vão ser expandidos. Verificou-se uma melhoria nos resultados quando se optou por ordenar os cubos de **F** por ordem decrescente de tamanho, uma vez que quanto maior for um cubo, maior é a probabilidade de cobrir outros e menor a probabilidade de ser coberto por outros. Esta ordenação é muito simples mas, na maioria dos casos, verificou-se ser de facto eficaz.

O principal subprocedimento de EXPAND transforma um cubo **c** da cobertura **F** num implicante primo $\mathbf{c}^+$ e lista os cubos de **F** (outros que não **c**) que são cobertos por $\mathbf{c}^+$; como já se referiu o cubo $\mathbf{c}^+$ é escolhido de modo a que cubra o maior número possível de cubos de **F**.

Dado um cubo **c** que deva ser expandido, a sua expansão óptima (máxima) corresponde a um cubo primo **d** contido ou igual a **c** que:

- minimize o número de cubos na cobertura expandida;

- transforme o cubo **d** no maior cubo possível (isto é, dependente do menor número possível de variáveis, portanto com o maior número possível de don't care).

Portanto, a operação EXPAND minimiza não só o número de cubos em **F**, mas também o número de literais.

Para o caso de funções de saídas múltiplas, os cubos que correspondam, por exemplo, a **k** saídas serão desmultiplicados obtendo-se (a partir de cada cubo original) **k** cubos, todos com a mesma parte de entradas e em cuja parte de saídas apenas existe um 4.

4.5 O Procedimento ESSENTIAL_PRIMES

Dado que os primos essenciais têm de aparecer em todas as coberturas primas de **ff** é preferível detectá-los no início do

programa e retirá-los da cobertura em estudo durante as operações de EXPAND, REDUCE e IRREDUNDANT_COVER.

Relembre-se que um dado primo é essencial se e só se existir um mintermo no ON-set de **ff** que esteja contido apenas nesse primo. Assim, começa por determinar-se o conjunto dos primos essenciais de **F**, testando sucessivamente todos os cubos.

Para que, em termos computacionais, seja compensador fazer esse teste, é necessário que o procedimento ESSENTIAL_PRIMES seja suficientemente rápido. Seja **E** o conjunto dos primos essenciais da função **ff**; então, retirando os primos essenciais da cobertura **F** e adicionando-os a **D** (DON'T CARE-set), reduzimos a dimensão do problema de | **E** |, ou seja, o número de cubos do ON-set **F** diminui de | **E** | e o número de cubos do DON'T CARE-set aumenta de igual quantidade.

A ideia base do procedimento ESSENTIAL_PRIMES é testar todos os cubos de **F** com o cubo $\mathbf{c}^i$ para a determinação da sua essencialidade e então verificar se o conjunto dos cubos assim obtido contém ou não $\mathbf{c}^i$; se sim, é porque $\mathbf{c}^i$ não é essencial, doutro modo $\mathbf{c}^i$ é essencial.

Este teste usa fundamentalmente o algoritmo TAUTOLOGIA que é um algoritmo rápido, como já foi referido. Deste modo a extracção dos primos essenciais de **F** é uma operação compensadora relativamente a subsequentes economias de tempo.

4.6 O Procedimento IRREDUNDANT_COVER

Ao iniciar-se este procedimento partimos de uma cobertura prima **F** de **ff**, tal que nenhum cubo de **F** contém qualquer outro. Contudo, isso não garante que **F** seja uma cobertura mínima, pois pode acontecer que exista um subconjunto de **F** que também seja uma cobertura de **ff**. Dados **F** e **D**, o procedimento IRREDUNDANT_COVER produz uma cobertura mínima $\tilde{F}$ formada por um subconjunto de cubos de **F** (o menor número de cubos possível).

O procedimento IRREDUNDANT-COVER é formado basicamente por três subprocedimentos.

REDUNDANT divide **F** em dois conjuntos **E** e **R**. **E** é o conjunto dos cubos **c** pertencentes a **F** tais que (**F** - {**c**}) já não é uma cobertura de **ff**. Logicamente esses cubos têm de aparecer em qualquer subcobertura $\tilde{F}$ contida em **F** e são chamados cubos relativamente essenciais (ou necessários) de **F**. Os restantes cubos, **R**, podem ser extraídos, individualmente, de **F** sem destruir a cobertura de **ff** e são portanto chamados cubos redundantes de **F**.

O subprocedimento REDUNDANT testa cada cubo **c** pertencente a **F** para verificar se $(D \cup F - \{c\})_c$ é uma tautologia, isto é, se o cofactor de (**D** U **F** - {**c**}), relativamente a **c**, é igual a 1. Então, como $(D \cup F - \{c\})_c = 1$ se e só se **c** for coberto por (**D** U **F** - {**c**}), teremos que se (**D** U **F** - {**c**}) cobre **c**, então **c** é redundante; caso contrário **c** pertence a **E** (conjunto dos cubos relativamente essenciais).

O conjunto de cubos redundantes **R** pode ser subdividido em dois subconjuntos: o conjunto dos cubos totalmente redundantes R_t e o conjunto dos cubos parcialmente redundantes R_p. O conjunto R_t é formado pelos cubos **c** pertencentes a **R** que são cobertos por (**D** U **E**); logo, como **E** faz necessariamente parte de qualquer subcobertura $\tilde{F}$ contida em **F**, os cubos de R_t podem ser retirados de **F**. Os restantes cubos de **R** formam então o conjunto dos cubos parcialmente redundantes R_p. Este teste é feito usando o subprocedimento PARTIALLY_REDUNDANT.

Utilizando de novo o procedimento TAUTOLOGIA, verifica-se se cada cubo **c** pertencente a **F** é ou não coberto por (**D** U **E**). Então, se o cofactor de (**D** U **E**) relativamente a **c** é diferente de 1, isto é, se o cubo **c** não é coberto por (**D** U **E**) então **c** é parcialmente redundante; doutro modo **c** é totalmente redundante.

O passo mais difícil de resolver neste algoritmo consiste na extracção, a partir de R_p, de um conjunto mínimo R_c, tal que $\tilde{F} = (E \cup R_c)$ seja ainda uma cobertura de **ff**. Essa selecção de cubos é feita pelo subprocedimento MINIMAL_IRREDUNDANT, a partir do qual se obtém o conjunto R_c. O desejável seria produzir um conjunto de cubos R_c que fosse o mínimo, mas isso nem sempre pode ser garantido. O conjunto R_c obtido pode não ser o mínimo possível; contudo, se R_c for o mínimo, então $\tilde{F} = (E \cup R_c)$ é uma subcobertura mínima de **ff**.

4.7 O Procedimento REDUCE

O procedimento REDUCE transforma uma cobertura prima **F** numa cobertura, em geral, não prima $\tilde{F}$, por substituição de cada cubo de **F** por um outro cubo, normalmente menor, contido no inicial. Como alguns cubos de $\tilde{F}$ já não são primos, na próxima passagem no ciclo de minimização, EXPAND pode ser aplicado a $\tilde{F}$ de modo a obter-se uma cobertura prima diferente da inicial e que pode ter um menor número de cubos do que **F**, mas nunca mais, pois $| \tilde{F} | \leq | F |$.

Como a escolha dos cubos a serem reduzidos, assim como o método a utilizar têm definitiva importância no melhoramento da solução obtida, optou-se por utilizar uma heurística que processe os cubos de **F** sequencialmente reduzindo cada um deles ao máximo (sem destruir a cobertura).

Note-se que, se REDUCE for aplicado a uma cobertura que não seja mínima, este procedimento gerará uma cobertura mínima, uma vez que os cubos redundantes serão reduzidos a cubos nulos e portanto removidos de **F**.

Consideremos **F** e **D** e seja c^i um cubo de **F**. Como se pretende reduzir ao máximo c^i, mas mantendo **F** como uma cobertura de **ff**, terá de obter-se o menor cubo possível $\underline{c}^i$ que contenha todos os mintermos de c^i que não sejam cobertos por $[(F - \{c^i\}) \cup D]$. Então, se definirmos

$$F(i) = [(F - \{c^i\}) \cup D] \text{ e } \overline{F}(i) \text{ o seu complementar,}$$

teremos $\underline{c}^i$ = menor cubo contendo $(c^i \cap \overline{F}(i))$ = MCC $(c^i \cap \overline{F}(i))$. Obviamente, MCC $(c^i \cap \overline{F}(i))$ existe sempre, porque é a intersecção de todos os cubos contendo $(c^i \cap \overline{F}(i))$. A operação efectuada para se obter $\underline{c}^i$ é chamada redução de **c** relativamente a **F**. Note-se que $[(F - \{c^i\}) \cup \{\underline{c}^i\}]$ é ainda uma cobertura de **ff**.

Uma vez que o evoluir deste procedimento é dependente da ordem pela qual os cubos são processados, optou-se por uma heurística que permitisse uma ordenação prévia dos cubos. Assim e com base na ideia de que quanto maiores forem os cubos, mais facilmente serão reduzidos, detecta-se o maior cubo e ordenam-se os restantes, relativamente a ele, por ordem crescente de "pseudo-distância". A "pseudo-distância" é medida pelo número de

diferenças entre dois cubos, por exemplo: [0 1 2 1 3 4] e [0 2 1 1 4 4] têm "pseudo-distância" igual a três.

Então e após se reduzir o maior cubo, tenta-se reduzir os que estão mais próximos dele. Deste modo, quando posteriormente se efectuar a expansão da cobertura, ter-se-á aumentada a probabilidade de ver a expansão do maior cubo cobrir os cubos seus vizinhos.

Uma vez que o algoritmo EXPAND tem como principal finalidade maximizar o número de cubos que podem ser cobertos por uma dada expansão, é provável que o maior cubo que entretanto foi reduzido por REDUCE, seja agora expandido de modo a garantir a cobertura de um grande número de cubos seus vizinhos.

No final de REDUCE, obtém-se uma cobertura **F** em que cada cubo **c** foi substituido por $\underline{c}$.

4.8 O Procedimento LAST_GAP

Com este procedimento pretende fazer-se uma última tentativa de reduzir o número de cubos da cobertura. LAST_GAP é um procedimento formado por uma modificação de REDUCE, seguida de uma modificação de EXPAND e baseia-se no facto de, após se reduzir um cubo **c**, surgirem duas hipóteses de diminuir a dimensão da cobertura a que ele pertence. Por um lado o cubo reduzido **c*** pode ser coberto por um cubo vizinho após a expansão; e por outro o cubo reduzido **c*** pode ser expandido em diferentes direcções de modo a cobrir alguns cubos vizinhos. De qualquer modo, quanto menor for o cubo reduzido **c***, melhores resultados se obtêm.

Relembre-se que REDUCE é um procedimento dependente da ordem pela qual são tratados os cubos que formam a cobertura. Assim os últimos cubos de uma cobertura têm sempre menos hipóteses de serem reduzidos do que os primeiros; já que os cubos que os precedem já foram minimizados e a capacidade de estes virem a ser cobertos ficou diminuida.

Já se sabe que o procedimento REDUCE não garante que a dimensão da cobertura diminua quando EXPAND é aplicado. Contudo, como LAST_GAP reduz cada cubo individualmente e ao máximo e depois

aplica a expansão só aos cubos que foram reduzidos, verifica-se que, se após essa expansão se encontrarem cubos que cubram pelo menos um dos cubos reduzidos, esses possibilitarão uma diminuição da dimensão de cobertura.

Sendo $\mathbf{F} = \{\mathbf{c}^1, \mathbf{c}^2, ..., \mathbf{c}^p\}$ uma cobertura prima de $\mathbf{ff} = \{\mathbf{f}, \mathbf{d}, \mathbf{r}\}$, define-se $\mathbf{F}(\mathbf{i}) = (\mathbf{F} - \{\mathbf{c}^\mathbf{i}\})$ e $\mathbf{f}(\mathbf{i})$ como sendo a função booleana correspondente. Então a redução máxima do cubo $\mathbf{c}^\mathbf{i}$, que se representa por $\underline{\mathbf{c}}^\mathbf{i}$, é definida como sendo o menor cubo contendo $[\mathbf{c}^\mathbf{i} \cap (\mathbf{f}(\mathbf{i}) \cup \mathbf{d})]$. Observe-se que $\underline{\mathbf{F}} = \{\underline{\mathbf{c}}^1, \underline{\mathbf{c}}^2, ..., \underline{\mathbf{c}}^p\}$ não é necessariamente uma cobertura, uma vez que os cubos foram reduzidos independentemente e não em sequência.

Em LAST_GAP, a primeira parte do procedimento calcula o conjunto dos cubos maximamente reduzidos $\underline{\mathbf{F}}$ e em seguida restringe $\underline{\mathbf{F}}$ ao conjunto dos cubos que foram de facto reduzidos (isto é, excluem-se de $\underline{\mathbf{F}}$ os cubos $\underline{\mathbf{c}}^\mathbf{i} = \mathbf{c}^\mathbf{i}$). Depois faz-se a expansão desse conjunto de cubos e só desse. Note-se que os cubos que não foram reduzidos continuam sendo primos e portanto não poderiam ser cobertos.

Assim, a dimensão da cobertura só diminui se se tiver encontrado pelo menos um primo que após a expansão cubra um dos cubos reduzidos. Verifica-se (experimentalmente) que, se após esta expansão se encontar pelo menos um primo nas condições descritas, em geral ele cobrirá não um, mas dois dos cubos reduzidos.

Sendo **H** o conjunto dos cubos primos que após serem expandidos cobrem pelo menos um dos cubos reduzidos $\{\underline{\mathbf{c}}^\mathbf{i}\}$, se **H** não for vazio, então LAST_GAP acrescenta os novos primos de **H** à cobertura **F**. Deste modo, quando posteriormente se fizer IRREDUNDANT_COVER (**F** U **H**, **D**) muito provavelmente reduzir-se-á a cardinalidade de **F**. Portanto, este procedimento garante que não existe nenhuma melhor solução do que a obtida após a sua execução.

4.9 O Procedimento MAKE_SPARSE

MAKE_SPARSE é executado em dois passos e parte de uma cobertura prima e mínima **F**. O primeiro passo consiste em LOWER_OUTS que reduz, sempre que possível, todos os 4's para 3's

na tentativa de transformar a parte de saídas da cobertura numa matriz esparsa. O segundo passo (subprocedimento RAISE_IN) torna a parte de entradas da cobertura tão esparsa quanto possível, aumentando os 1's e os 0's para 2's.

Repare-se que se se aplicasse RAISE_IN antes de LOWER_OUTS, não se conseguiria aumentar nenhuma entrada, uma vez que **F** já é uma cobertura prima. Contudo, se primeiro se reduzirem as saídas (usando LOWER_OUTS), então os cubos de **F** que forem alterados deixam de ser primos e pode acontecer que seja possível expandi-los (aumentá-los) na sua parte de entradas.

Convém não esquecer que antes de se executar o procedimento MAKE_SPARSE é necessário adicionar os primos essenciais à cobertura **F**, pois mesmo os primos essenciais podem ser tornados esparsos.

Em LOWER_OUTS é construido um vector **P** que indica se cada cubo $\mathbf{c}^i$ pertencente a **F** permanece ou não primo (isto é, se é ou não alterado). Sendo $\tilde{\mathbf{F}}$ e $\tilde{\mathbf{D}}$ as restrições de **F** e **D** a uma única saída **k** (**k** = 0, ..., m, onde **m** é o número de saídas da função), este procedimento usa IRREDUNDANT_COVER e retorna um vector cujos elementos formam um subconjunto mínimo de $\tilde{\mathbf{F}}$ que é uma cobertura para a saída **k**.

O subprocedimento RAISE_IN é uma modificação de EXPAND, restringida apenas à parte de entradas de cada cubo de **F**. Note-se que os cubos que são primos e que foram referenciados pelo vector **P**, construido em LOWER_OUTS, não são modificados por este procedimento. Então, os cubos não primos $\mathbf{c}^i$ pertencentes a **F** são transformados em primos nas suas partes de entradas, isto é, a parte de entradas de cada cubo $\mathbf{c}^i$ é aumentada tornando-se tão esparsa quanto possível.

Observe-se que no final de MAKE_SPARSE a nova cobertura não é necessariamente prima, mas é esparsa, isto é, não é possível transformar mais 0's ou 1's (da parte de entradas de cada cubo) em 2's, ou transformar mais 4's (da parte de saídas) em 3's sem que **F** deixe de ser uma cobertura de **ff**.

4.10 Conclusão

Uma primeira versão do programa EXPRESSO-II, posteriormente melhorada, foi publicada em 1982; em 1984 foi apresentada uma implementação em linguagem C.

Os algoritmos que formam o programa EXPRESSO-II constituem uma ferramenta de minimização lógica que atinge dois objectivos fundamentais: uma execução robusta e eficaz e a optimização dos resultados finais.

A elevada qualidade da minimização obtida com o programa EXPRESSO-II é o resultado da eficiência dos procedimentos EXPAND, IRREDUNDANT_COVER, REDUCE e LAST_GAP. Assim, e em média, pode verificar-se experimentalmente a eficiência relativa de cada procedimento. Em percentagem do tempo total de execução, foram obtidos os seguintes valores para os tempos correspondentes aos procedimentos indicados:

COMPLEMENT	14%
EXPAND	29%
IRREDUNDANT_COVER	12%
ESSENTIAL_PRIMES	13%
REDUCE	8%
LAST_GAP	12%
MAKE_SPARSE	10%

Fazendo-se a comparação dos resultados obtidos pelo programa EXPRESSO-II com os resultados encontrados com outros programas, nomeadamente com uma versão modificada do método exaustivo de Quine-McCluskey, verifica-se que se encontram soluções tão boas quanto as anteriores, mas muito menos dispendiosas. Análogas observações se podem fazer quando se compara a execução do EXPRESSO-II com a de outros programas heurísticos, como POP [DEM84] ou MINI [HON74].

Estas tentativas heurísticas da resolução do problema da minimização de funções lógicas têm como componente fundamental um procedimento que faz a expansão de cada implicante e a posterior remoção de todos os outros implicantes por ele cobertos. Deste modo evita-se a geração exaustiva e o armazenamento de todos os implicantes que definem a função lógica.

5. Método de simplificação usando BDD's

5.1 Introdução

Uma diferente abordagem ao problema da minimização de funções booleanas é feita neste capítulo e baseia-se numa proposta de Akers [Ake78] que recorre a árvores ou diagramas de decisão binária (**BDD** - Binary Decision Diagram) para representar as funções booleanas. Estas serão reduzidas posteriormente usando métodos específicos de simplificação [AKE78].

Neste capítulo começar-se-á por fazer uma descrição do conceito de BDD e em seguida indicar-se-á como construir e identificar um BDD.

Os BDD's são grafos directos e acíclicos que fornecem uma eficiente descrição da função booleana de uma forma completa e concisa, mas independente da sua posterior implementação, e que são facilmente armazenáveis e manipuláveis em computador.

Um diagrama de decisão binária (BDD) pode ser descrito como uma técnica de representação de funções booleanas baseada em árvores binárias e formado por elementos de decisão, os nós, e por ramos de ligação entre eles [AKE78].

Assim, um nó, que é o elemento de decisão, funciona de um modo semelhante ao de uma condição IF... THEN ... ELSE ... das linguagens de programação.

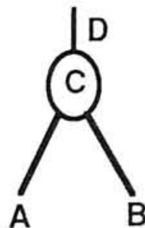


Fig. 5.1 - Representação de um nó de um BDD.

Por exemplo, o nó representado na Fig. 5.1 traduz a seguinte sequência de instruções:

```
if C=0  then D:=A
        else if C=1 then D:=B
```

Os nós são controlados pelas chamadas variáveis de controlo e, dependendo do valor associado a essa variável, o valor lógico da saída do nó é o valor da sua entrada esquerda (ramo esquerdo), se a variável de controlo for igual a **0**, ou da sua entrada direita (ramo direito) se a variável de controlo for igual a **1**.

Assim, cada um desses ramos pode ser o valor lógico **0** ou **1** ou ainda o valor de um outro nó que, nesse caso, terá também de ser calculado.

Na representação de uma função booleana os nós são controlados pelas variáveis de entrada da função. Então, a representação de uma dada função lógica **f** obtém-se atribuindo a **f** o valor à saída do nó do topo do diagrama, isto é, da raiz da árvore. Note-se que as variáveis de entrada que controlam os nós permitem definir um único percurso entre um dado nó e um **0** lógico ou um **1** lógico que estejam presentes num dos ramos de entrada. Estes poderão corresponder a saídas de nós pertencentes a níveis inferiores.

Na Fig. 5.2 mostra-se, como exemplo, a representação de uma função lógica de 3 variáveis, $\mathbf{f} = \overline{A}\,\overline{C} + A\,\overline{B}\,\overline{C} + A\,B\,C$.

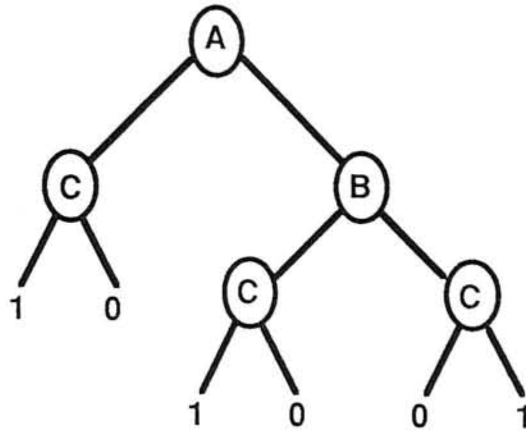


Fig. 5.2 - Representação de uma função lógica de 3 variáveis.

Assim, analisando o diagrama considerado, veremos que para o conjunto de variáveis de entrada (A = 1; B = 1; C = 1) teremos definido um dado percurso terminado em **1**; para (A = 1; B = 0; C = 0) teremos um outro; e para (A = 0; C = 0) encontraremos ainda outro. Então, a função **f** pode ser definida pela soma das três parcelas obtidas a partir das combinações das variáveis de entrada cujo caminho termina em **1**. Todas as outras possíveis combinações dessas variáveis de entrada conduzem a situações em que o valor da função é igual a **0**, ou seja, o caminho correspondente termina em **0**. Note-se que a um percurso definido ao longo do diagrama e que termine em **1** corresponde um mintermo da função.

Repare-se, no entanto, que a representação de uma dada função booleana como soma de produtos não é unívoca, pois existem sempre várias possibilidades de especificar uma função booleana, todas logicamente equivalentes. Nomeadamente, como exemplo da afirmação, veja-se o caso da representação de uma função antes e após a simplificação.

5.2 Alguns conceitos básicos

Vejamos melhor como utilizar os BDD's como técnica de representação de funções lógicas, através de alguns exemplos simples.

Consideremos a função **AND** de quatro variáveis de entrada, representada na Fig. 5.3. Como seria de esperar o único percurso terminado em **1** corresponde ao caso de todas as variáveis de entrada valerem **1**; isto é, (A = 1; B = 1; C = 1; D = 1). Note-se que para todas as outras combinações de variáveis de entrada o valor da função é **0**.

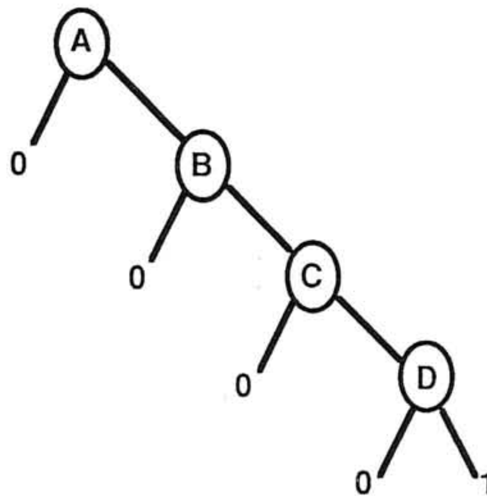


Fig. 5.3 - Representação da função AND de 4 variáveis.

Note-se que, para uma dada combinação dos valores das variáveis de entrada, existe um único caminho entre a saída de um nó, neste caso da raíz da árvore, e os valores lógicos **0** ou **1**. Repare-se ainda que o valor da saída do nó raiz é o valor da função para as combinações das variáveis de entrada cujos percursos terminam em **1**. Outros exemplos são também facilmente representados. Nas Fig. 5.4, 5.5 e 5.6 apresentam-se as representações das funções OR, NAND e NOR de quatro variáveis, respectivamente.

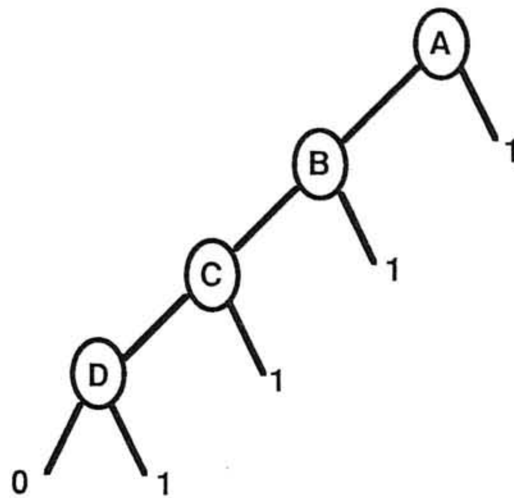


Fig. 5.4 - Representação da função OR de 4 variáveis.

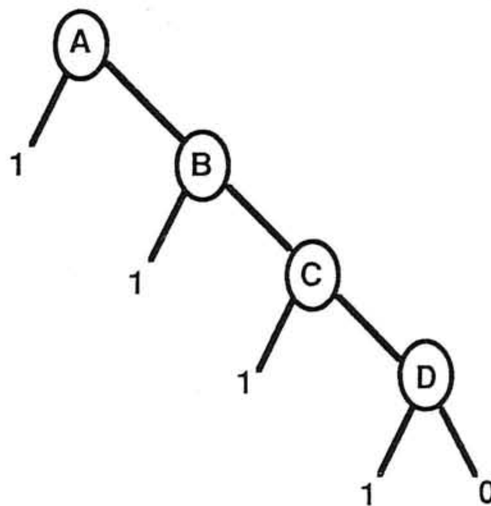


Fig. 5.5 - Representação da função NAND de 4 variáveis.

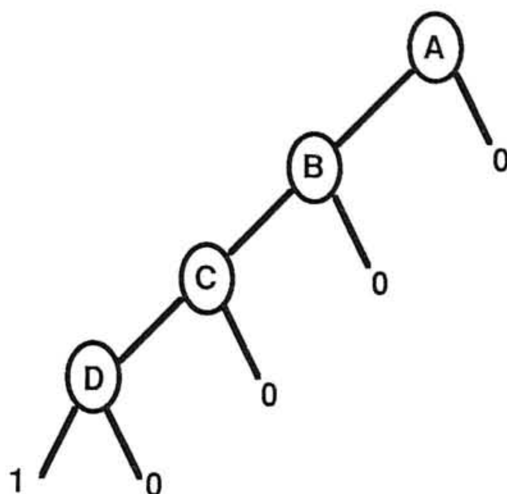


Fig. 5.6 - Representação da função NOR de 4 variáveis.

Em todos estes exemplos, o último nó poderia ter sido suprimido, considerando-se a variável **D** aplicada directamente ao ramo esquerdo ou direito do nó controlado por **C**. Isto porque se podem considerar equivalentes as duas representações ilustradas na Fig. 5.7 que exemplifica como **complementar o valor de um nó**. No primeiro caso a saída do nó vale **C** e no segundo vale o seu complementar.

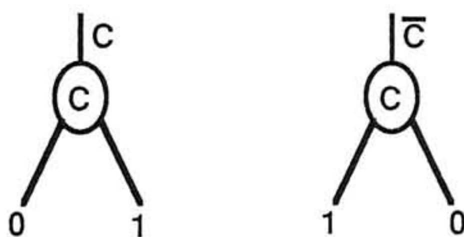


Fig. 5.7 - Complementação do valor de um nó.

Assim, nas Fig. 5.8 e 5.9 estão desenhados diagramas equivalentes aos das Fig. 5.3 e 5.5, respectivamente.

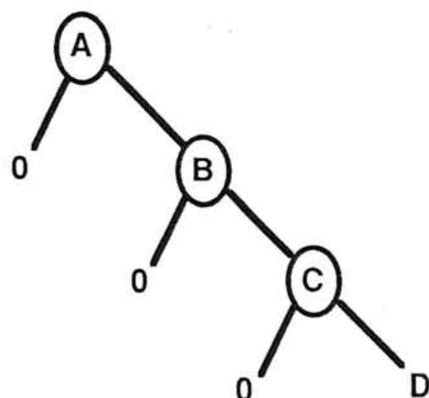


Fig. 5.8 - Representação da função AND de 4 variáveis.

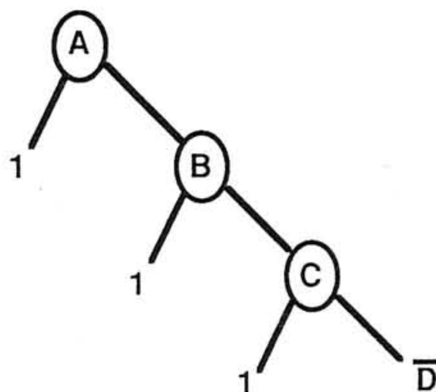


Fig. 5.9 - Representação da função NAND de 4 variáveis.

E generalizando este conceito, dir-se-á que o **complementar de uma função** pode ser obtido directamente do BDD, complementando os valores de todas as folhas da árvore binária (folhas são os filhos dos ramos terminais, isto é, que já não são alimentados pela saída de nenhum outro nó). Nos exemplos apresentados é facilmente reconhecível esta regra, ao passar-se da Fig. 5.3 para a Fig. 5.5 e da Fig. 5.4 para a Fig. 5.6.

Outro conceito importante, que permite a obtenção de reduções nas representações das funções lógicas por BDD's, é o da **inversão lógica do valor de um ramo.** Ao avaliar-se um diagrama, isto é, ao calcular-se o valor do nó raiz, se existe inversão de algum ramo, que será representada por um ponto (•) no

ramo em questão, o complementar do valor desse ramo deve ser considerado.

Igualmente útil do ponto de vista da manipulação dos BDD's é a noção de **complementar da variável de controlo** de um nó, que se obtém simplesmente complementando a variável de controlo e trocando entre si os ramos esquerdo e direito desse nó.

5.3 Construção de um BDD

A obtenção de um BDD pode ser conseguida a partir de uma tabela de verdade, de uma lista de mintermos ou ainda das equações lógicas que representam a função. Note-se que para uma função com **n** variáveis de entrada podem definir-se $\mathbf{2^n}$ diferentes percursos desde a saída do nó raiz do diagrama até cada uma das folhas da árvore binária. A cada um dos $\mathbf{2^n}$ percursos definíveis corresponde uma das $\mathbf{2^n}$ possíveis combinações das **n** variáveis de entrada [AKE77, AKE78].

Para que uma dada combinação de variáveis de entrada seja um mintermo de função, é necessário que seja **f** = 1 para esse conjunto de variáveis de entrada. Relembre-se que um **mintermo** é dado por um produto de todas as variáveis de entrada ou seus complementares a que corresponde no BDD um percurso terminado em **1**. Então, para se construir o BDD que represente uma dada função definida pelos seus mintermos, basta aplicar um **1** às entradas dos nós terminais a que correspondem os mintermos dados; todas as outras entradas tomarão o valor **0**.

Suponha-se a função definida pela seguinte lista de mintermos

$$f(A, B, C, D) = \sum m(1, 3, 7, 12, 13, 14, 15)$$

e representada na Fig. 5.10. Após simplificação resulta a função equivalente representada na Fig. 5.11. Note-se que duas funções booleanas são equivalentes se lhes corresponder a mesma tabela de verdade.

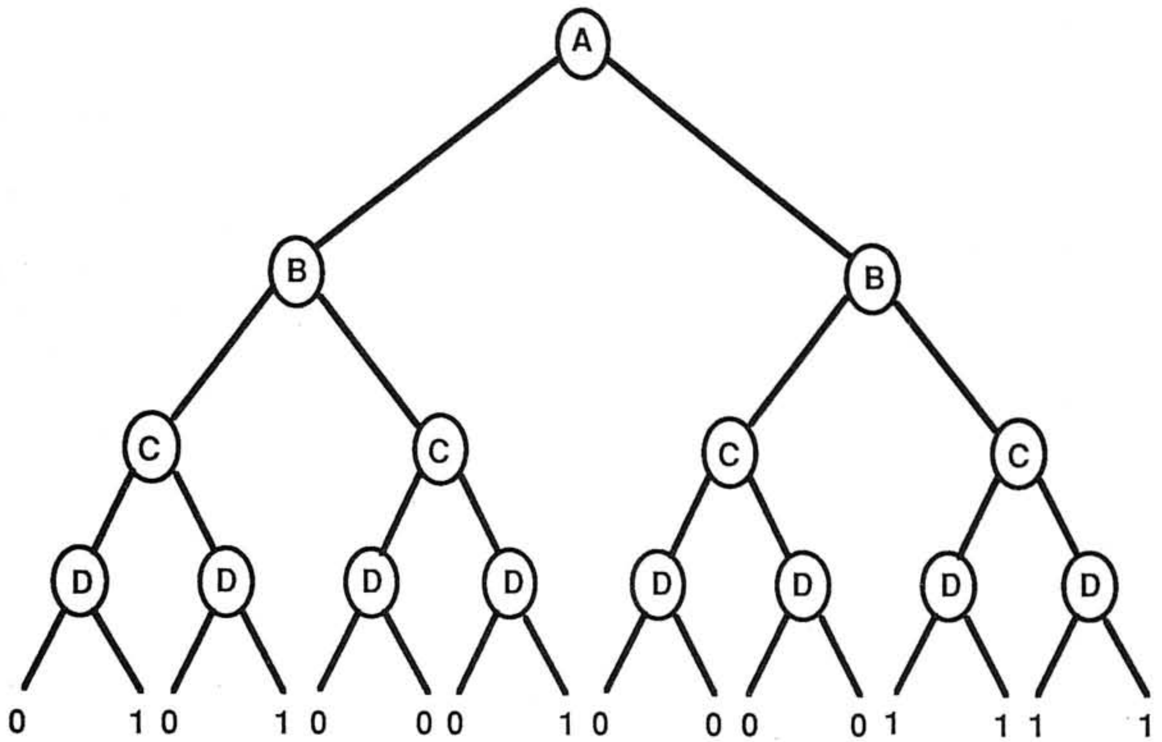


Fig. 5.10 - Representação de uma função lógica de 4 variáveis.

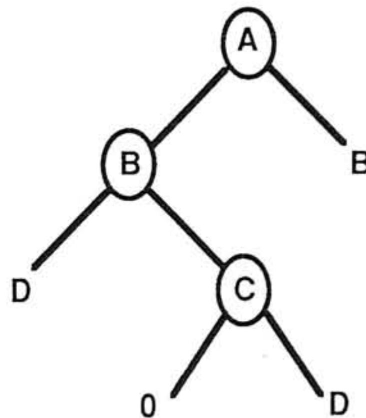


Fig. 5.11 - Representação da função após simplificação.

A simplificação é obtida por eliminação de nós redundantes (nós redundantes são aqueles que têm o mesmo valor nos dois ramos),

como adiante se explicará em pormenor. Após simplificação, a função será definida por $f = \bar{A}\,\bar{B}\,D + \bar{A}\,B\,C\,D + A\,B$.

Repare-se que os termos produto que se obtêm a partir dos BDD's não são necessariamente primos implicantes (isto é, podem não ser formados pelo menor número possível de literais), mas como cada novo percurso que se considere contém pelo menos um ramo diferente dos ramos que formam os percursos anteriores, isso garante que os termos produto considerados são disjuntos, isto é, não se intersectam, e portanto pode dizer-se que são essenciais no sentido de que cada mintermo é coberto por um e só um implicante essencial.

No exemplo representados na Fig. 5.10 os mintermos $\mathbf{m_1}$ e $\mathbf{m_3}$ são cobertos por $\bar{A}\,\bar{B}\,D$, enquanto que $\mathbf{m_7}$ é coberto por $\bar{A}\,B\,C\,D$ e $\mathbf{m_{12}}$, $\mathbf{m_{13}}$, $\mathbf{m_{14}}$ e $\mathbf{m_{15}}$ são cobertos por AB.

Como facilmente se vê existe uma analogia nítida entre esta configuração do BDD e a soma de produtos da lógica booleana. Um termo produto corresponde a um caminho percorrido desde a raiz até um **1** terminal ou uma variável de entrada.

Este modo de configurar o BDD fornece um bom ponto de partida para se proceder à geração e simplificação da função respectiva. Repare-se que a largura do diagrama cresce exponencialmente, isto é, duplica sempre que se acrescente uma nova variável de entrada. Isto faz com que a simplificação do BDD seja um factor imprescindível; para se atingir esse fim é necessário eliminar todas as redundâncias que facilmente aparecem neste tipo de estrutura. Deste modo, geralmente, obtêm-se grandes reduções no números de nós necessários para definir a função.

O ideal será ir construindo a árvore dinamicamente e simultaneamente fazendo simplificações possíveis, ainda que não completas, de modo a minimizar as necessidades totais de armazenamento. Assim a dimensão da árvore será mantida dentro de limites razoáveis. Finalmente assinale-se que parece ser indiscutível que um diagrama binário tem o tipo de estrutura facilmente armazenável e processável em computadores, nomeadamente se for utilizado uma linguagem de alto nível tipo Pascal ou C.

Então, avaliar um BDD, ou seja determinar o valor lógico associado à saída do nó raiz do diagrama, para um dado conjunto de combinações das variáveis de entrada é uma tarefa que pode ser definida de forma recursiva. Começa por determinar-se o valor da variável de controlo. Se essa variável de controlo for a saída de um outro nó é necessário calcular primeiro o valor desse outro nó. Depois toma-se a decisão sobre qual ramo (esquerdo ou direito) seleccionar. Se o ramo escolhido apontar para uma constante (**0** ou **1**) ou para uma variável de entrada, determina-se o valor de saída do nó e o processo termina. Caso contrário, isto é, se o ramo escolhido apontar para outro nó, calcula-se primeiro o valor desse outro nó pelo processo descrito.

É importante notar aqui que, para um BDD construido da forma descrita, o número de operações envolvidas na sua avaliação, para uma dada combinação de variáveis de entrada, é limitado superiormente pelo número de variáveis da função que ele representa [MOR82].

5.4 Identificação ou descrição de um BDD

Um BDD fica completamente definido se sobre cada nó se conhecer um determinado número de elementos seus identificadores:

- **número do nó**: representa um inteiro que identifica univocamente o nó;

- **variável de controlo**: representa uma referência que indica qual a variável de entrada, ou qual a saída de um outro nó, que é usada como variável de controlo;

- **ponteiro esquerdo**: representa uma referência para o ramo esquerdo; pode ser uma constante, uma variável de entrada ou a saída de outro nó;

- **ponteiro direito**: representa uma referência para o ramo direito; pode ser uma constante, uma variável de entrada ou a saída de outro nó.

Com base nestes elementos a descrição de um BDD pode ser apresentada sob a forma de uma tabela, onde as variáveis de

entrada e os nós são numerados segundo as convenções que a seguir se indicam.

Seja **ni** o número de variáveis de entrada da função a ser representada.

As constantes lógicas **0** e **1** são sempre representadas pelos valores inteiros **0** e **1**.

Se existirem valores indeterminados ou don't care, estes serão representados pelo valor inteiro **2**.

Os casos particulares dos nós com don't care serão distinguidos conforme os valores presentes nos seus ramos esquerdo e direito, usando-se as seguintes regras:

ao nó (20) corresponde o valor inteiro (-3)

ao nó (02) corresponde o valor inteiro (-4)

ao nó (21) corresponde o valor inteiro (3)

ao nó (12) corresponde o valor inteiro (4)

ao nó (22) corresponde o valor inteiro (2).

As variáveis de entrada são representadas por um valor inteiro pertencente ao intervalo [5, ni+4]; o complementar de uma variável de entrada é representado por um valor inteiro negativo pertencente ao intervalo [-ni-4, -5].

Os nós são numerados consecutivamente começando no valor inteiro (ni+5); isto é, à raiz da árvore será associado a variável de controlo identificada por (ni+5).

Portanto, sempre que se fizer referência a **n**, sendo **abs(n)** ≥ **(ni+5)**, estamos a associar **n** ao número de um nó; mas se a associação for tal que **abs(n)** < **(ni+5)** então **n** refere-se a uma variável de entrada.

Como já foi anteriormente referido, para se construir um BDD pode partir-se da informação contida numa tabela de verdade ou na correspondente lista de mintermos e de combinações don't care, ou ainda da informação de uma expressão lógica ou de um esquema lógico representativo da função.

À medida que o número de variáveis de entrada da função cresce, a tabela de verdade vai-se tornando um modo pouco prático de obter informação. É então que a lista de mintermos e de combinações don't care se torna muito útil e mais facilmente manipulável.

Assim para gerar o BDD a partir dessas listas, basta associar a cada uma das 2^n entradas dos nós do último nível da árvore o valor **1** se a essa entrada corresponder um mintermo, o valor **2** se lhe está associada uma combinação don't care e o valor **0** em todos os outros casos. Posteriormente será associado a cada valor don't care um **0** ou um **1** lógico de modo a conseguir-se a maior simplificação possível.

Por outro lado, para se obter um BDD a partir da descrição da função em termos de uma equação lógica dever-se-á proceder do seguinte modo: para cada ramo de entrada de um nó do último nível, calcular a expressão, usando a combinação de entradas que activa o caminho que termina nesse ramo de entrada.

É evidente que este processo tem o grande inconveniente de, para uma função com **n** entradas, requerer 2^n cálculos da mesma expressão cada um deles para uma das 2^n possíveis diferentes combinações das variáveis de entrada. Além disso, sempre que se considerar uma nova variável de entrada, o número de cálculos duplica.

Para este caso, uma abordagem mais eficiente pode ser conseguida mediante a utilização da denominada expansão de Shannon, referida no Apêndice A

$$f(x_1, x_2, ..., x_n) = x_1\ f(1, x_2, ..., x_n) + \bar{x}_1\ f(0, x_2, ..., x_n)$$

Facilmente se vê que a aplicação sucessiva, mediante a utilização de uma técnica recursiva, da expansão de Shannon à função **f** pode ser equivalente à construção de uma árvore binária em que cada próximo nó será o cofactor do nó anterior (no caso presente **f**), relativamente à variável x_1 ou ao seu complementar, portanto $f(0, x_2, ..., x_n)$ ou $f(1, x_2, ..., x_n)$

Esta técnica aplica-se sucessivamente até se obterem cofactores monótonos ou valores constantes ao atingir-se os nós terminais [FAB90].

Os **nós terminais** são caracterizados por terem ambos os ponteiros a **nil** e por o seu número identificador ser em valor absoluto menor do que **(ni+5)**.

Convém aqui fazer notar que, como a geração de um BDD pelo processo inicialmente descrito conduz a um crescimento exponencial do mesmo com o número de variáveis de entrada, poderia parecer existir, inerente ao processo, uma limitação quanto ao número de variáveis de entrada da função a representar.

Contudo, esse problema pode ser contornado na medida em que não é necessário começar por se construir o diagrama completo, isto é, o conjunto de $\mathbf{2^n-1}$ nós e dos respectivos apontadores [MAT83].

O diagrama é construido por um processo dinâmico e então logo que, para cada nível, se conhece o valor de um dado nó e dos respectivos ramos de entrada é feita uma tentativa de simplificação. Assim, estas primeiras simplificações são feitas ao mesmo tempo que se vai construindo o diagrama, por forma a manter a quantidade de informação a armazenar, dentro de limites aceitáveis [AKE78].

Sendo assim, facilmente se conclui que linguagens de programação como o Pascal ou o C, que permitem a atribuição e desatribuição dinâmica de espaço de memória são especialmente convenientes para a implementação destas técnicas.

5.5 Simplificação de um BDD

A eficiência da representação que permite a construção e simplificação dos BDD's depende da ordem pela qual as variáveis de entrada são consideradas na respectiva construção. Existem processos de se determinar a ordenação mais conveniente das variáveis de entrada de modo a conduzir à máxima simplificação da árvore binária [BRY86, FRI87, MOR82]. Contudo, convém notar que um critério de ordenação que produza uma solução óptima relativamente a uma determinada característica do problema, pode não conduzir à melhor solução quando se pretende optimizar uma outra característica do mesmo.

Um possível algoritmo de ordenação das variáveis de entrada que produziria vantajosas simplificações poderia ser enunciado do seguinte modo. Na tabela da função a estudar começa-se por procurar qual a linha que contém mais don't care. Então, qualquer uma das variávies dessa linha, que não seja don't care, poderá ser tomada como variável de topo do diagrama. Veja-se, por exemplo, o caso de uma função em que, sempre que a variável de entrada **E** fosse igual a **0**, a saída da função fosse igual a **0** independentemente do valor das outras variáveis de entrada. Assim, conviria que **E** fosse a variável de topo, pois o seu ramo esquerdo seria sempre **0**.

Com o objectivo de construir e minimizar BDD's foi desenvolvido um programa designado por SIMPBDD. Neste programa não se teve a preocupação de obter a ordenação óptima das variáveis de entrada, apenas se pretendeu partir de uma ordenação aceitável e optimizar a simplificação daí decorrente.

As operações mais significativas do modo como essa minimização é efectuada serão descritas no capítulo seguinte, onde serão simultaneamente referidas as heurísticas utilizadas em cada passo da simplificação.

5.6 Construção e simplificação de BDD's correspondentes a funções de saídas múltiplas

Após se ter estudado, na secção 5.3, como construir um BDD no caso de a função ser de saída única, será agora analisado o caso em que a função em estudo é de saídas múltiplas. Então o equivalente BDD pode considerar-se como sendo formado pelo conjunto dos BDD's que se obtêm a partir de cada uma das funções de saída única em que se pode decompor a função inicial, após uma determinada associação.

Essa associação deve permitir simplificações que não seriam possíveis se as funções de saída única obtidas a partir da função inicial não fossem agrupadas. A ideia base será começar por colocar lado a lado os BDD's parcelares e transformá-los num único "**super-BDD**".

Para se obter tal resultado recorrer-se-á à mesma noção que foi usada para se proceder à simplificação dos diagramas, mas agora em sentido inverso; isto é, de modo a que se produza uma expansão da árvore e não uma redução da mesma [MAT83].

Seja, por exemplo, uma função de saídas múltiplas (quatro saídas **f_0**, **f_1**, **f_2** e **f_3**) de três variáveis de entrada (**A**, **B** e **C**); então para se poderem associar os quatro BDD's correspondentes, de modo a obter-se um único BDD, é necessário expandir a árvore agrupando esses BDD's.

Assim, começa-se por determinar o número de variáveis fictícias necessárias à construção do "super-BDD". Sendo **m** o número de saídas da função de saídas múltiplas, verifica-se que o número de variáveis fictícias é igual ao menor inteiro superior ou igual a $\log_2 m$, portanto, neste caso, 2 (**X** e **Y**).

Então pode construir-se um único BDD cujo número total de variáveis seja igual ao número de variáveis fictícias mais o número de variáveis de entrada. A Fig. 5.12 ilustra tal processo de construção.

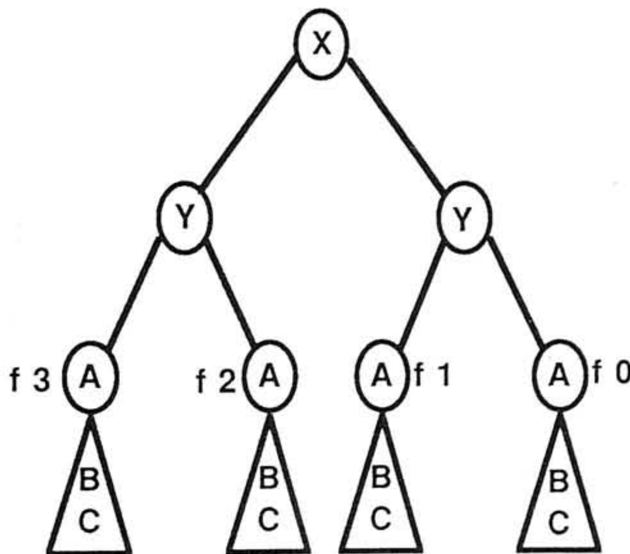


Fig. 5.12 - Construção de um "super-BDD".

Após a obtenção desse "**super-BDD**", ele será simplificado, basicamente pelos mesmos algoritmos que os BDD's correspondentes às funções de saída única. Neste caso concreto, como resultado da simplificação, encontra-se um BDD ao qual é preciso retirar os dois primeiros níveis (correspondentes às variáveis **X** e **Y**) para se poderem obter as quatro saídas $\mathbf{f_0}$, $\mathbf{f_1}$, $\mathbf{f_2}$ e $\mathbf{f_3}$ da função de saídas múltiplas considerada.

Note-se que se se pensasse em encontrar uma ordenação óptima para as variáveis de entrada, as variáveis de topo **X** e **Y** nunca deveriam ser incluidas nessa eventual ordenação.

6. O Programa de Minimização Booleana

6.1 Introdução

Neste capítulo é feita a descrição de um conjunto de algoritmos que constituem o programa de minimização booleana SIMPBDD, que utiliza BDD's para representar as funções lógicas a simplificar. Para o caso de funções de saídas múltiplas existem no programa algoritmos específicos para o seu tratamento, que serão referidos oportunamente.

O programa foi implementado em Pascal e corre num computador HP9000/300, com sistema operativo UNIX e permite a realização de testes com funções de, no máximo, 20 variáveis de entrada; no caso de funções de saídas múltiplas é possível obterem-se 16 saídas simultâneas.

As entradas para o programa poderão ser fornecidas sob a forma de uma lista de mintermos e de combinações don't care, ou sob a forma de uma expressão lógica escrita como uma soma de produtos ou ainda como uma tabela de verdade.

A estrutura de dados usada para armazenar a função a simplificar é, como já se referiu, uma árvore binária, em que cada nó contém, além de dois apontadores (esquerdo e direito) necessários à implementação da árvore em memória dinâmica, o número do nó e uma referência à variável de controlo do mesmo (que neste caso é o nível), constituindo portanto um record com quatro campos. Quando se fala na variável de entrada de um nó, pode estar-se a referenciar uma variável de entrada externa, uma constante lógica **0**, **1** ou **2**, ou ainda a saída de um outro nó.

As saídas do programa são as funções simplificadas escritas sob a forma de somas de produtos.

6.2 Avaliação de expressões booleanas

A determinação do valor lógico (**0** ou **1**) a atribuir a cada parcela da soma de produtos que define a função a simplificar, quando os

dados são fornecidos ao programa sob a forma de uma expressão booleana, é uma operação que tem de ser executada repetidas vezes até se avaliar toda a expressão dada.

Para tal, quando se chega a um nó terminal, usa-se o procedimento **AVALIAR** que tem como argumento a referência ao nó em estudo e retorna o valor lógico associado à saída desse nó. Assim, **AVALIAR** calcula, para cada possível combinação de valores das variáveis de entrada, o valor lógico a atribuir a cada parcela da soma de produtos que define a função lógica e associa-o ao correspondente nó terminal.

Contudo, se os dados forem fornecidos ao programa sob a forma de uma lista de mintermos e don't care, basta procurar, ordenadamente, para todos os possíveis índices, quais os que correspondem a mintermos, don't care ou zeros da função.

6.3 Simplificação de um BDD correspondente a uma função de saída única

Conforme já se referiu, a partir da informação relativa às variáveis de entrada, a árvore binária que representa a função em estudo é construida e simultaneamente minimizada.

A construção da árvore binária é feita de cima para baixo (isto é, partindo da raiz) e começando sempre pelo ramo esquerdo de cada nó. Para tal é usado o procedimento **CONSTRUIR** que começa por criar dinamicamente a primeira célula do record já referido, atribuindo-lhe os valores correspondentes à raiz da árvore.

Assim, ao campo número do nó é atribuido o valor dado por (**novo nó** + **1**); é de notar que, no programa principal, **novo nó** é feito igual a (**ni**+**4**), para que o número do nó seja sempre, em valor absoluto, superior ou igual a (**ni**+**5**), como já foi referido. Ao campo nível do nó é atribuido o valor dado por (**j**+**1**); no programa principal, **j** é inicializado a zero, de modo que à raiz da árvore possa corresponder o nível **1**.

E enquanto não se atingir o último nível e portanto não se definirem os nós terminais, vai-se recursivamente usando **CONSTRUIR** de modo a criar dinamicamente todos os nós do diagrama pelo processo descrito, começando sempre pelo ramo

esquerdo da árvore. Os nós terminais são caracterizados por terem ambos os ponteiros a **nil** e por o seu número identificador ser em valor absoluto menor do que (**ni+5**).

Construidos os primeiros nós terminais, imediatamente é feita uma tentativa de simplificação do diagrama, com o objectivo de não se ter de armazenar informação redundante. Deste modo o procedimento **PRSIMPL** (primeiras simplificações) é aplicado sucessivamente a cada nó terminal logo após a sua construção, obtendo-se assim as primeiras possíveis simplificações do diagrama. Como já foi referido, só após a aplicação de **PRSIMPL** é armazenada a árvore simplificada. Para se obterem essas primeiras simplificações foram estabelecidas as seguintes regras aplicáveis aos filhos (nós terminais) de um dado nó. Sendo $\mathbf{p_1}$ o apontador para o filho esquerdo do nó apontado por **p** e $\mathbf{p_2}$ o apontador para o seu filho direito, então fazendo

p1^.número nó = número do nó apontado por p1

p2^.número nó = número do nó apontado por p2

p^.número nó = número do nó apontado por p

teremos

* se (p1^.número nó)=0 e (p2^.número nó)=0
  então (p^.número nó)=0;

* se (p1^.número nó)=1 e (p2^.número nó)=1
  então (p^.número nó)=1;

* se (p1^.número nó)=2 e (p2^.número nó)=2
  então (p^.número nó)=2;

* se (p1^.número nó)=0 e (p2^.número nó)=1
  então (p^.número nó)=(p^.nível+4), isto é, (p^.número nó) é igual ao número identificador da variável de controlo desse nó;

* se (p1^.número nó)=1 e (p2^.número nó)=0
  então (p^.número nó)=(-p^.nível-4), isto é, (p^.número nó) é igual ao número identificador do complementar da variável de controlo desse nó;

* se (p1^.número nó)=0 e (p2^.número nó)=2
  então (p^.número nó)=-4;

* se (p1^.número nó)=2 e (p2^.número nó)=0
  então (p^.número nó)=-3;

* se (p1^.número nó)=1 e (p2^.número nó)=2
  então (p^.número nó)=4;

* se (p1^.número nó)=2 e (p2^.número nó)=1
  então (p^.número nó)=3.

Os nós apontados por **p**, assim obtidos, são agora nós terminais, portanto os seus apontadores são postos a **nil** e o seu número de nó é em valor absoluto menor do que (**ni+5**). Verifica-se então que dependendo do valor dos filhos direito e esquerdo de um dado nó, a esse nó será associado um valor dado pela regra acima estabelecida.

Por exemplo, considere-se o diagrama binário correspondente à função lógica definida pelos mintermos (3 e 5) e pelos don't care (4 e 7) e representada na Fig. 6.1. Após a primeira simplificação resulta o diagrama da Fig. 6.2, que representa a função equivalente $f = \overline{A} B C + A \overline{B}$.

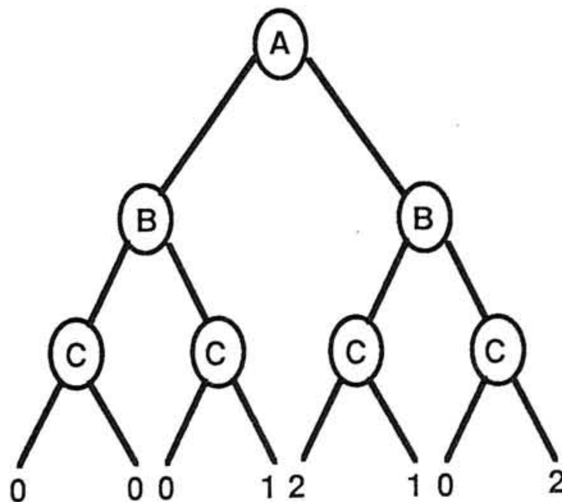


Fig. 6.1 - Representação da função a simplificar.

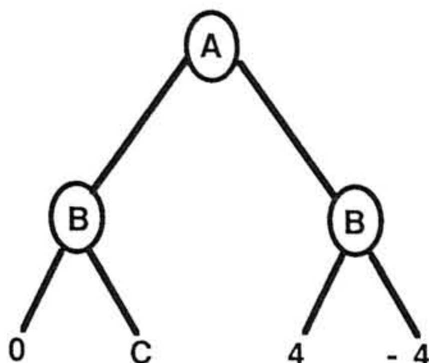


Fig. 6.2 - Aplicação de PRSIMPL ao BDD da Fig. 6.1.

Convém aqui evidenciar que durante o processo recursivo de construção da árvore, o procedimento **PRSIMPL** pode também ser aplicado aos nós apontados por **p** (atrás definidos) que passaram entretanto a ser nós terminais, pesquisando-se aí novas possíveis simplificações.

Após estas primeiras simplificações feitas por **PRSIMPL**, que são executadas partindo do último nível em direcção à raiz da árvore, será tentada uma outra simplificação em sentido inverso, isto é, iniciando-se na raiz da árvore já parcialmente simplificada e procurando começar por eliminar a variável de topo desse diagrama, ou seja, a raiz. Caso não seja possível fazer-se a simplificação da raiz, tentar-se-á proceder de modo idêntico relativamente aos ramos esquerdo e direito da raiz, isto é, procurando eliminar a variável de topo de cada um desses dois ramos.

Para tal, o procedimento **ELIMINAR** efectuará a minimização referida, caso ela seja possível. Assim, **ELIMINAR** procede à supressão de um nó cujas subárvores direita e esquerda são iguais, assim como à eliminação de uma dessas subárvores. Ou seja, retira da árvore um nó redundante e uma das subárvores que foi detectado serem iguais.

Para decidir da possibilidade ou não de tal simplificação é utilizado o procedimento **COMPARE**. Caso não seja possível eliminar-se a variável de topo do diagrama, deve então tentar aplicar-se essa mesma simplificação a cada um dos ramos desse

nó raiz; sempre na tentativa de eliminar as variáveis o mais cedo possível, isto é, nos níveis o mais próximo possível da raiz.

O procedimento **COMPARE** analisa a árvore de cima para baixo e compara o ramo esquerdo com o direito de cada nó em estudo, na tentativa de encontrar igualdade. Deste modo, partindo da raiz da árvore e sendo p_1 o apontador para o seu ramo esquerdo e p_2 o apontador para o seu ramo direito, poderemos ter os seguintes casos:

* se p_1 se refere a um nó terminal, isto é, se o número (**n**) do nó apontado por p_1 é tal que **abs(n)** < (**ni+5**), em que **ni** é o número de variáveis de entrada da função, verifica-se se p_2 se refere ou não a um nó terminal;

* se p_1 se refere a um nó normal, verifica-se se p_2 se refere ou não a um nó terminal.

No texto que se segue usar-se-á p_1 para designar o apontador para o ramo esquerdo de um dado nó e p_2 para designar o apontador para o ramo direito do mesmo nó e ainda p_1^ e p_2^ para representar os nós apontados por p_1 e p_2, respectivamente.

Assim sendo teremos quatro hipóteses em análise:

se (p_1^ é um nó terminal e p_2^ é um nó terminal), utiliza-se **COMPAREA**;

se (p_1^ é um nó terminal e p_2^ é um nó normal), utiliza-se **COMPAREB**;

se (p_1^ é um nó normal e p_2^ é um nó terminal), utiliza-se **COMPAREC**;

se (p_1^ é um nó normal e p_2^ é um nó normal), utiliza-se **COMPARED**.

Um nó terminal é caracterizado por ter ambos os ponteiros a **nil** e por o seu número identificador ser em valor absoluto menor do que (**ni+5**), enquanto que um nó normal tem os ponteiros a indicar os seus filhos esquerdo e direito e o seu número identificador é em valor absoluto maior do que (**ni+5**).

Em cada um destes quatro subprocedimentos é decidido se se encontrou igualdade entre $\mathbf{p_1}$ e $\mathbf{p_2}$ ou se é possível fazer-se $\mathbf{p_1}$ igual a $\mathbf{p_2}$. Esta última operação envolverá a atribuição de valores lógicos definidos em substituição dos don't care.

COMPARE analisa os nós utilizando regras onde é estabelecido o modo de fazer a atribuição de valores aos don't care por forma a se obterem as maiores simplificações possíveis. Assim impôs-se que:

- A comparação seja sempre feita entre os elementos do último nível do ramo esquerdo do nó que se quer tentar eliminar e os elementos do último nível do ramo direito desse mesmo nó. Por outras palavras, se na Fig. 6.3 se quer, por exemplo, eliminar o nó **A**, devem comparar-se os nós terminais correspondentes às subárvores definidas pelos filhos esquerdo e direito de **A**; ou seja, comparar $\mathbf{x_1}$ com $\mathbf{x_1}^*$, $\mathbf{x_2}$ com $\mathbf{x_2}^*$, $\mathbf{x_3}$ com $\mathbf{x_3}^*$ e $\mathbf{x_4}$ com $\mathbf{x_4}^*$.

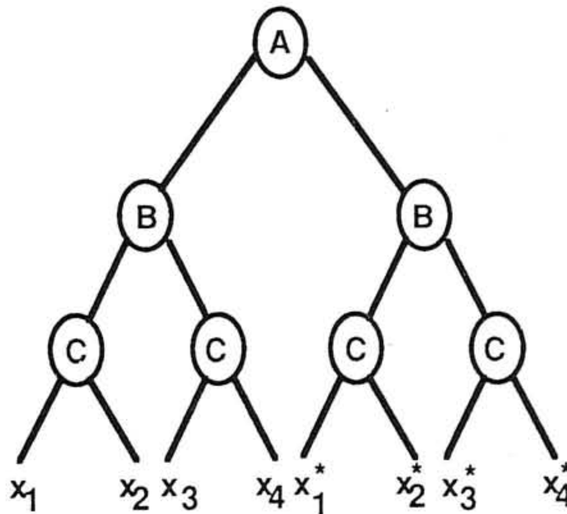


Fig. 6.3 - Exemplificação do modo de execução de COMPARE.

Ao proceder-se deste modo permitem-se simplificações que seriam impossíveis se a comparação fosse feita entre nós sucessivos, como por exemplo se se comparasse $\mathbf{x_1}$ com $\mathbf{x_2}$, $\mathbf{x_3}$ com $\mathbf{x_4}$, etc.

Utilizando o exemplo anterior, facilmente se pode provar a afirmação feita. Admita-se que $x_1, x_2, x_3, x_4, x_1^*, x_2^*, x_3^*, x_4^*$ valem respectivamente, 2, 0, 0, 1, 1, 0, 0, 1. Então, comparando x_1 com x_1^*, x_2 com x_2^*, x_3 com x_3^* e x_4 com x_4^*, facilmente se vê que fazendo o don't care (x_1) igual a **1** (x_1^*), a subárvore esquerda do nó **A** é igual à subárvore direita; logo pode eliminar-se o nó **A** e uma das subárvores, ficando o BDD reduzido à outra subárvore. Contudo, se se tivesse comparado x_1 com x_2, x_3 com x_4, etc., ter-se-ia feito o don't care x_1 igual a **0** (x_2) e portanto não se poderia ter eliminado o nó **A**.

- Uma vez que simplificar uma função booleana significa minimizar o número de parcelas da soma de produtos que a define; e como cada parcela corresponde a um termo produto (mintermo); para se minimizar a função, pode pensar-se em minimizar o número de mintermos. Logo, **sempre que possível**, os don't care serão passados a **zeros** e não a **uns**. Assim, um nó cujos filhos sejam (2, 0) ou (0, 2) ou (2, 2), será considerado igual a **0**, a menos que outra atribuição possível conduza a maiores simplificações da árvore binária.

- Sempre que um dos nós em análise **p1^** ou **p2^** corresponda a um don't care, isto é, se **p1^.número nó = 2** ou **p2^.número nó = 2**, então será sempre possível fazer esse nó comparável ao outro. Isto significa que, por exemplo, qualquer que seja o valor de **p2^**, se **p1^.número nó = 2**, então pode fazer-se **p1^ = p2^** e portanto eliminar-se o nó pai de **p1^** e **p2^** assim como **p1^** ou **p2^**; então, a árvore ficará reduzida, por exemplo, a **p1^**.

No final de **COMPARE** é sempre retornado um resultado, indicando se se conseguiu ou não obter igualdade, que permita ao procedimento **ELIMINAR** proceder ou não à eliminação do nó em estudo. Se a eliminação do nó que se está a avaliar não for possível, a atribuição dos valores **0** ou **1** aos don't care não é feita; só posteriormente e em face de outras possíveis simplificações do diagrama tal atribuição será executada.

Em **COMPAREA** analisa-se o caso de **p1^** e **p2^** serem nós terminais, começando por estudar-se o caso de **p1^** e **p2^** poderem ser iguais. Então impõe-se que se faça a eliminação correspondente usando **ELIMINAR** do modo já descrito e no caso de existirem don't care fazem-se as seguintes atribuições:

* se (p1^.número nó=p2^.número nó)=2
  então (p1^.número nó=p2^.número nó)=0;

* se (p1^.número nó=p2^.número nó)=-4
  então (p1^.número nó=p2^.número nó)=0;

* se (p1^.número nó=p2^.número nó)=-3
  então (p1^.número nó=p2^.número nó)=0;

* se (p1^.número nó=p2^.número nó)=3
  então (p1^.número nó=p2^.número nó)=1;

* se (p1^.número nó=p2^.número nó)=4
  então (p1^.número nó=p2^.número nó)=1.

Os records que representam **p_1^** e **p_2^**, no campo número do nó, poderão ter um dos seguintes valores:

0=(0 0), isto é, o número do nó será 0, se o número dos nós dos filhos esquerdo e direito desse nó forem 0;

p^.nível+4=(0 1), ou seja, o número do nó valerá (p^.nível+4), se o número do nó do seu filho esquerdo for 0 e o do direito for 1;

e analogamente para os casos restantes:

-4=(0 2);

1=(1 1);

-p^.nível-4=(1 0);

4=(1 2);

2=(2 2);

-3=(2 0);

3=(2 1).

Note-se que temos aqui nove possíveis hipóteses de nós terminais.

No caso de **p_1^** e **p_2^** serem diferentes são impostas as seguintes regras para atribuição de valores aos don't care:

- se entre **p_1^** e **p_2^** existirem incompatibilidades, então simplifica-se **p_1^** e **p_2^** independentemente um do outro; por exemplo, se **p_1^.número nó = -4** e **p_2^.número nó = 4**, como a comparação de **0** com **1** não é possível, faz-se **p_1^.número nó = 0** e **p_2^.número nó = 1** e é retornado para o procedimento **ELIMINAR** a informação de que a comparação entre **p_1^** e **p_2^** falhou;

- se **p_1^** e **p_2^** são compatíveis, então fazem-se as simplificações correspondentes, e retorna-se para **ELIMINAR** a informação de que a comparação foi possível, por exemplo:

* se (p1^.número nó=-3) e (p2^.número nó=-4)
  então (p1^.número nó=0) e (p2^.número nó=0)

* se (p1^.número nó=-3) e (p2^.número nó=4)
  então (p1^.número nó=-p2^.nível-4) e (p2^.número nó=-p2^.nível-4)

* se (p1^.número nó=-4) e (p2^.número nó=p2^.nível+4)
  então (p1^.número nó=p2^.nível+4) e (p2^.número nó=p2^.nível+4)

* se (p1^.número nó=3) e (p2^.número nó=1)
  então (p1^.número nó=1) e (p2^.número nó=1)

* se (p1^.número nó=4) e (p2^.número nó=3)
  então (p1^.número nó=1) e (p2^.número nó=1)

e assim sucessivamente usando a mesma filosofia de redução.

Repare-se que existem agora 81 diferentes possíveis combinações de (**p_1^.número nó**) com (**p_2^.número nó**), decorrentes das nove hipóteses referidas para os nós terminais. Podem portanto formar-se nove tabelas, como se mostra na Fig. 6.4:

Se p_1^.nº nó	e p_2^.nº nó	então p_1^.nº nó	e p_2^.nº nó	Comparação
-3	-3	0	0	possível
-3	-4	0	0	possível
-3	3	0	1	falhou
-3	4	-nível-4	-nível-4	possível
-3	-nível-4	-nível-4	-nível-4	possível
-3	nível+4	0	nível+4	falhou
-3	0	0	0	possível
-3	1	0	1	falhou
-3	2	0	0	possível

Fig. 6.4 - Comparação de nós terminais.

O subprocedimento **COMPAREB** estuda o caso de **p_1^** ser um nó terminal e **p_2^** ser um nó normal. Então, como **p_1^** poderá tomar um dos possíveis valores já referidos em **COMPAREA**, para que **p_2^** possa ser feito igual a **p_1^**, **p_2^** terá de ser comparável a **p_1^**. Por outras palavras, se, por exemplo, for **p_1^.número nó = 0**, **p_2^.número nó** só será comparável a **0** (e portanto **p_2^ = p_1^**) se os ramos da esquerda e direita de **p_2^** terminarem em **0's** ou **2's**, isto é, se não houver nenhum **1** entre eles.

Análogo raciocínio se poderá fazer para qualquer um dos outros casos. De qualquer modo **COMPAREB** retornará informação sobre a obtenção ou não de igualdade entre **p_1^** e **p_2^**. Em tudo semelhante a este é o subprocedimento **COMPAREC**, em que agora **p_1^** é um nó normal e **p_2^** um nó terminal.

Quanto a **COMPARED** é um subprocedimento em que se chama recursivamente **COMPARE** até se atingir um dos casos terminais, analisáveis então por **COMPAREA**, **COMPAREB** ou **COMPAREC**.

Realce-se que a ideia do procedimento **ELIMINAR** é reduzir o número de nós do diagrama; logo, sempre que este procedimento é executado com sucesso produz uma simplificação máxima em todos os sentidos - reduz o número de nós, diminui a largura do diagrama e, eventualmente, reduz um nível. Como tal, deve ser executado de cima para baixo, para que as variáveis sejam eliminadas ao mais alto nível e portanto conduzam a maiores simplificações. Mais uma vez se faz aqui notar a vantagem da ordenação das variáveis de entrada relativamente às possíveis simplificações obtidas.

Uma outra simplificação pode agora ser tentada sobre a árvore minimizada, executando-se o procedimento **FUNDIR**. Basicamente, este procedimento compara os ramos da árvore e produz, se possível, fusões entre eles. Quando se fundem dois ramos surge uma vantajosa redução do número de nós do diagrama.

Considere-se, por exemplo, o diagrama da Fig. 6.5 e aplique-se **FUNDIR**.

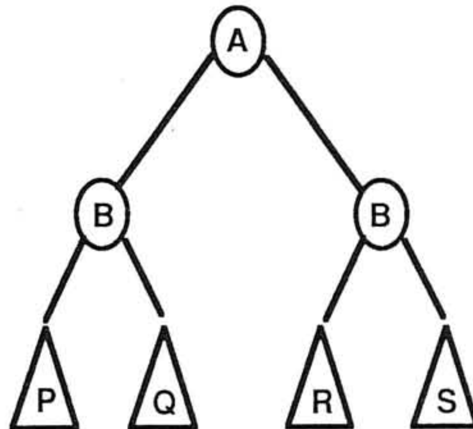


Fig. 6.5 - BDD a FUNDIR.

Este procedimento pesquisa a árvore de cima para baixo na tentativa de descobrir possíveis igualdades entre os quatro ramos **P**, **Q**, **R** e **S**. Recorde-se que durante a execução do procedimento **ELIMINAR** são procuradas possíveis igualdades entre (**P** e **R**) e (**Q** e **S**) simultaneamente; caso não se verifiquem essas igualdades, pesquisa-se o caso de ser (**P** = **Q**) ou (**R** = **S**). Logo, quando termina

ELIMINAR, é porque (**P** ≠ **R**) ou (sendo **P** = **R** é **Q** ≠ **S**) ou (**P** ≠ **Q**) ou (**R** ≠ **S**).

Assim, usando **FUNDIR** poder-se-á encontrar um dos seguintes casos:

* **Q** = **R** e **P** ≠ **S**, isto é, se **Q** e **R** puderem ser feitos iguais, então faz-se a fusão, isto é, o apontador esquerdo do nó direito **B** será feito igual ao apontador direito do nó esquerdo **B**; assim ambos apontarão para o mesmo ramo, como se mostra na Fig. 6.6.

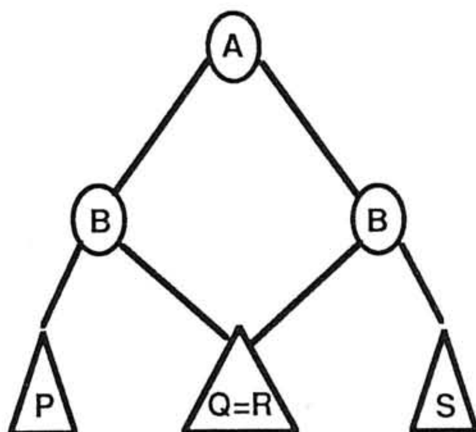


Fig. 6.6 - Fusão de Q e R.

* **P** = **R** e **Q** ≠ **S**, ou seja, se **P** e **R** puderem considerar-se iguais, então começa por se trocar os ramos **P** e **Q** da árvore, considerando portanto o complementar da variável de controlo do nó **B**, pai de **P** e de **Q** e depois executa-se **FUNDIR** (**P** e **R**) do modo indicado, resultando o diagrama da Fig. 6.7.

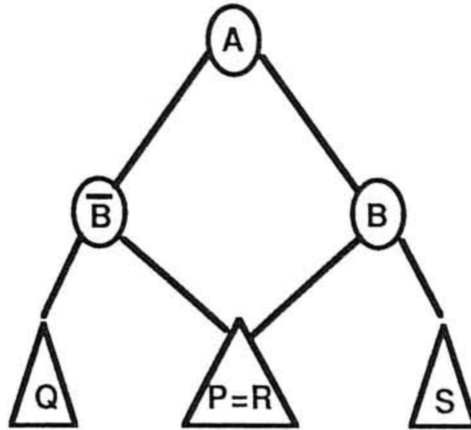


Fig. 6.7 - Fusão de P e R.

* **Q** = **S** e **P** ≠ **R**, este caso é análogo ao anterior; logo inicialmente trocam-se os ramos **R** e **S** e complementa-se a variável de controlo do nó seu pai, o nó **B**, e em seguida procede-se à fusão de **Q** e **S**, resultando o diagrama da Fig. 6.8.

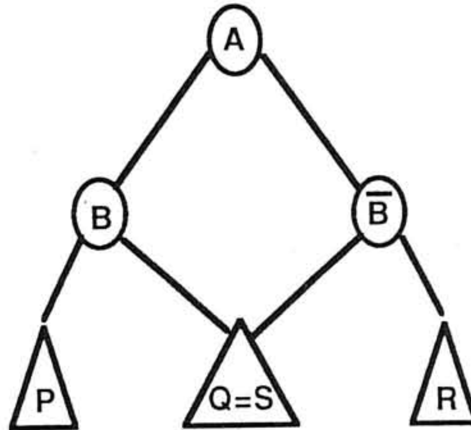


Fig. 6.8 - Fusão de Q e S.

* **P** = **S** e **Q** ≠ **R**, esta situação necessita de uma complementação das variáveis de controlo de ambos os nós **B**, portanto dos nós cujos filhos são (**P** e **Q**) e (**R** e **S**); seguidamente poder-se-á executar a fusão de **P** e **S**, pelo processo já referido.

As igualdades (**Q** = **S**) e (**P** = **R**) nunca aqui ocorrerão simultaneamente, pois este caso teria sido detectado em **ELIMINAR**.

Então, após a execução de qualquer um dos casos anteriores, se for possível encontrar igualdade entre o par de ramos que ainda não foi fundido, a realização dessa nova fusão conduzirá inevitavelmente a cruzamentos dos ramos do diagrama simplificado.

Assim, **FUNDIR** pode produzir uma redução do número de nós do diagrama, com todas as vantagens daí decorrentes.

Após a execução de **PRSIMPL** poder-se-ia ter utilizado outra estratégia, continuando a simplificar a função de baixo para cima. Para tal, o procedimento **ELIMINAR** seria substituido por **SIMPLTOTAL** que se descreve em seguida.

SIMPLTOTAL é um procedimento que utiliza a mesma estratégia de simplificação que **PRSIMPL**, portanto faz a análise de dois sucessivos nós terminais, e decide logo qual a possível simplificação, mediante a utilização de tabelas idênticas às utilizadas em **COMPARE**. Enquanto os nós obtidos deste modo puderem ser sucessivamente simplificados, o processo prossegue até, por fim, se chegar à raiz do diagrama.

Para se proceder à escrita da função simplificada sob a forma de soma de produtos executa-se uma sequência de procedimentos que, partindo da raiz e à medida que se desce na árvore, primeiro à esquerda e depois à direita, começa por guardar numa pilha os valores das variáveis correspondentes a cada percurso possível ao longo do BDD.

Ao atingir-se um nó terminal, se se verificar que esse percurso corresponde a uma parcela da função simplificada, então e só nesse caso, escreve-se o termo produto correspondente. Para que um dado percurso represente uma parcela da soma de produtos que define a função simplificada é necessário que o número do nó terminal correspondente a esse percurso tenha um dos seguintes valores: **1**, **3**, **4**, (**nível do nó** + **4**) ou (-**nível do nó** - **4**). Quando se tiver percorrido todo o diagrama, ter-se-ão escrito todas as parcelas da soma de produtos que representa a função em estudo.

A apresentação e comparação dos resultados obtidos é feita no final do capítulo.

6.4 Simplificação de um BDD correspondente a uma função de saídas múltiplas

Como já foi referido no capítulo anterior, quando se trata de funções de saídas múltiplas, é necessário começar por calcular o número de variáveis fictícias que são precisas para associar os BDD's correspondentes a cada saída da função e obter um único super-BDD, como descrito na Fig. 6.9.

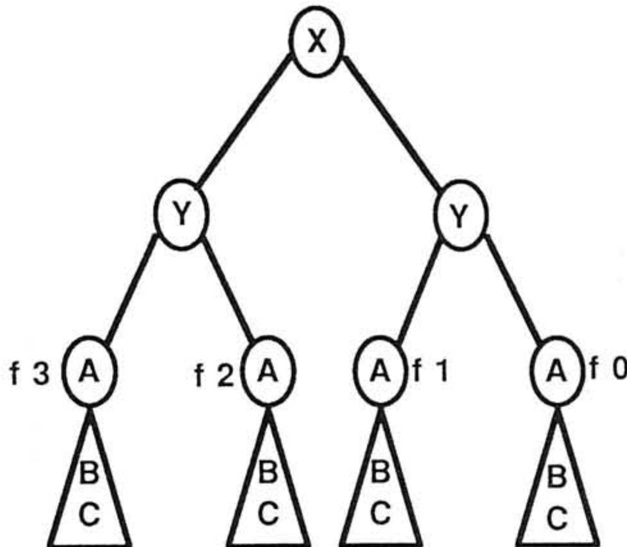


Fig. 6.9 - Construção de um "super-BDD".

Assim, para uma função com **m** saídas, o número de variáveis fictícias é igual ao menor inteiro superior ou igual a ($\log_2 m$). Sendo **ni** o número de variáveis de entrada de cada uma das **m** funções de saída única, obtidas a partir da função de **m** saídas múltiplas, pode construir-se o super-BDD do modo descrito a seguir.

Começa por construir-se um BDD pelo processo descrito para funções de saída única, mas em que o número total de variáveis de entrada seja a soma de **ni** com o menor inteiro superior ou igual a ($\log_2 m$), isto é, a soma de **ni** com o número de variáveis fictícias.

Esse BDD assim construido será simplificado normalmente pelos procedimentos atrás descritos. Referindo-nos à Fig. 6.9, para se obterem agora as saídas f_0, f_1, f_2, e f_3 é suficiente eliminar, no

super-BDD, os dois primeiros níveis, o que é equivalente a dizer-se suprimir as variáveis fictícias **X** e **Y**.

Para tal basta que, partindo da raiz do super-BDD, e enquanto o nível em estudo for menor ou igual ao número de variáveis fictícias, se continue a descer na árvore, usando um processo recursivo; atingido esse nível devem começar a escrever-se as expressões que representam as funções simplificadas. Assim, no exemplo referido, em que o número de variáveis fictícias é igual a dois, quando se chega ao nível 3 é chamado o procedimento que executa a escrita das funções simplificadas (f_0, f_1, f_2, e f_3).

Quando ($\log_2 m$) não é um valor inteiro, é necessário construir o super-BDD recorrendo a subárvores don't care. Se, por exemplo, no caso referido anteriormente fosse $m = 3$, a saída f_3 seria uma árvore don't care; isto é, todos os seus nós terminais seriam don't care e portanto o valor da saída f_3 seria qualquer, desde que conduzisse às maiores simplificações da árvore. Então, o número de subárvores don't care será em cada caso igual a ($2^M - m$), em que **M** é o menor inteiro superior ou igual a ($\log_2 m$).

6.5 Resultados obtidos

Para se verificar a eficiência do programa desenvolvido realizaram-se vários testes, não só no sentido de se quantificarem os tempos de execução, mas também para se avaliar a qualidade das somas de produtos que formam as funções simplificadas.

Um dos testes realizado utilizou um tipo de funções que se sabia à priori ir obrigar à execução total do programa, isto é, em que nunca surgissem hipóteses simplificativas que conduzissem à interrupção da execução dos procedimentos antes do final dos mesmos. Deste modo, os tempos obtidos para a execução do programa seriam os máximos. Testou-se uma função de **16** variáveis de entrada (**ni=16**), que era definida apenas pelo mintermo m_{65535} ($2^{16} - 1 = 65535$); assim, toda a árvore foi construida e simplificada até se atingir o mintermo m_{65535}; a função simplificada obtida foi, conforme se esperava, **f = ABCDEFGHIJKLMNOP**.

Utilizaram-se neste teste duas versões do programa, baseadas em duas estratégias diferentes; primeiro, recorreu-se a **PRSIMPL** seguido de **ELIMINAR**, isto é, inicialmente a simplificação do BDD foi feita de baixo para cima e em seguida partindo da raiz tentaram eliminar-se os nós redundantes (VERSÃO 1); depois, utilizou-se **PRSIMPL** e **SIMPLTOTAL**, ou seja, simplificou-se todo o BDD de baixo para cima, sem se tentar eliminar nenhum nó (VERSÃO 2).

Dependendo do modo como os dados foram fornecidos ao programa, por exemplo, lista de mintermos e don't care ou expressão lógica que define a função, os tempos de execução resultantes (obtidos a partir da função Time do UNIX) foram substancialmente diferentes. Assim obteve-se:

VERSÃO 1	mintermos	13.0 s
VERSÃO 1	exp. lógica	42.5 s

* função simplificada:
  f = ABCDEFGHIJKLMNOP

VERSÃO 2	mintermos	14.2 s
VERSÃO 2	exp. lógica	44.4 s

* função simplificada:
  f = ABCDEFGHIJKLMNOP

Em seguida testou-se uma função idêntica definida por dois mintermos, $\mathbf{m_{32767}}$ e $\mathbf{m_{65535}}$, também escolhidos de modo a possibilitarem toda a execução dos procedimentos, com **16** variáveis de entrada (**ni=16**) e obteve-se:

VERSÃO 1	mintermos	13.1 s

* função simplificada:
  f = BCDEFGHIJKLMNOP

VERSÃO 2	mintermos	14.2 s

* função simplificada:
  $f = \bar{A}BCDEFGHIJKLMNOP + ABCDEFGHIJKLMNOP$

Então, o que sugerem as estratégias de simplificação escolhidas?

Primeiro, as expressões que definem as funções simplificadas obtidas são as esperadas. A VERSÃO 2 do programa não produz eliminação de nós redundantes; logo, no segundo exemplo, era de prever a existência do nó A, o que conduz ao imediato aparecimento de duas parcelas na expressão simplificada.

Segundo, os tempos de execução são idênticos para ambas as versões, sendo contudo ligeiramente superiores na VERSÃO 2, dado que não se eliminaram variáveis, mas em vez disso se procurou fazer simplificações utilizando uma técnica semelhante à de **COMPARE**.

Terceiro, quando os dados são fornecidos ao programa através de uma expressão lógica, os tempos de execução são muito maiores do que quando é dada a lista de mintermos e don't care, isto porque é necessário avaliar uma expressão lógica, que é uma operação complicada, e repeti-la várias vezes.

Contudo, se a função testada for apenas de, por exemplo, 4 ou 5 variáveis de entrada, independentemente da forma como os dados são fornecidos ao programa, ou da versão utilizada, os tempos de execução são idênticos e muito baixos. Seja a função de 5 variáveis de entrada definida por

$$\sum m(2, 3, 4, 7, 9, 10, 12, 13, 18, 19, 20, 23, 25, 27, 28, 29)$$

e testada nas condições referidas; então, em qualquer dos casos, os tempos de execução são menores que 0.1 s.

Como era de prever, quando se passa de **ni=15** para **ni=16**, os tempos de execução, para o mesmo tipo de função e para a mesma versão do programa, duplicam aproximadamente.

CASO 1

VERSÃO 1	$\sum$ m(32767)	ni=15	6.7 s
VERSÃO 1	$\sum$ m(65535)	ni=16	13.0 s
VERSÃO 2	$\sum$ m(32767)	ni=15	7.0 s
VERSÃO 2	$\sum$ m(65535)	ni=16	14.2 s

CASO 2

VERSÃO 1	Σ m(16383, 32767)	ni=15	6.5 s
VERSÃO 1	Σ m(32767, 65535)	ni=16	13.1 s
VERSÃO 2	Σ m(16383, 32767)	ni=15	7.0 s
VERSÃO 2	Σ m(32767, 65535)	ni=16	14.2 s

Comparando o CASO 1 com o CASO 2, quando a função é definida por uma lista de mintermos, os tempos de execução são idênticos; enquanto que se a forma de entrada de dados for uma expressão lógica os tempos são maiores no segundo caso, o que também é razoável uma vez que a expressão que define a função tem mais parcelas que terão de ser avaliadas sucessivamente.

Um outro teste foi realizado utilizando um multiplicador, em que os dois factores foram definidos com 4 bits cada, resultando portanto o produto com 8 bits. Um exemplo realizado foi a multiplicação 11*9=99 sendo $11_{(10)}=1011_{(2)}$, $9_{(10)}=1001_{(2)}$ e $99_{(10)}=01100011_{(2)}$, usando a versão do programa que permite a simplificação de funções de saídas múltiplas (VERSÃO 3) com 8 variáveis de entrada e 8 saídas simultâneas. Obtiveram-se os resultados esperados que se apresentam a seguir:

$f_0 = \bar{B}DE\bar{F}\bar{G}H + B\bar{C}DE\bar{F}\bar{G}H$

$f_1 = \bar{A}CE\bar{F}\bar{G}H + A\bar{B}CE\bar{F}\bar{G}H$

$f_2 = \bar{A}BE\bar{F}\bar{G}H + AB\bar{C}E\bar{F}\bar{G}H$

$f_3 = \bar{A}DE\bar{F}\bar{G}H + A\bar{B}\bar{D}E\bar{F}\bar{G}H + AB\bar{C}\bar{D}E\bar{F}\bar{G}H$

$f_4 = \bar{A}CE\bar{F}\bar{G}H + A\bar{B}\bar{C}DE\bar{F}\bar{G}H + A\bar{B}C\bar{D}E\bar{F}\bar{G}H + AB\bar{C}DE\bar{F}\bar{G}H$

$f_5 = \bar{A}BE\bar{F}\bar{G}H + A\bar{B}CDE\bar{F}\bar{G}H + AB\bar{C}E\bar{F}\bar{G}H$

$f_6 = A\bar{B}E\bar{F}\bar{G}H + AB\bar{C}E\bar{F}\bar{G}H$

$f_7 = 0.$

O tempo de execução da minimização, isto é, o tempo que decorre desde que foram encontrados os mintermos que definem a função até se obterem as oito funções simplificadas foi de 0.6 s.

Foram ainda minimizadas várias funções de 4 e 5 variáveis de entrada, apenas com o intuito de comparar a qualidade das expressões lógicas resultantes das simplificações feitas usando BDD's com as obtidas por outros métodos, nomeadamente com os métodos de Karnaugh e Quine-McCluskey. Em qualquer dos exemplos testados encontraram-se tempos de execução inferiores a 0.1s.

Por exemplo, considerou-se a função de 4 variáveis de entrada e 3 saídas múltiplas definida do seguinte modo:

$$f_0 = \sum m(2, 4, 10, 11, 12, 13)$$

$$f_1 = \sum m(4, 5, 10, 11, 13)$$

$$f_2 = \sum m(1, 2, 3, 10, 11, 12)$$

Obtiveram-se as seguintes funções simplificadas:

$$f_0 = \bar{A}\bar{B}C\bar{D} + \bar{A}B\bar{C}\bar{D} + A\bar{B}C + AB\bar{C}$$

$$f_1 = \bar{A}B\bar{C} + A\bar{B}C + AB\bar{C}D$$

$$f_2 = \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}C + A\bar{B}C + AB\bar{C}\bar{D}$$

Para o caso de uma função de 5 variáveis de entrada e saída única definida por

$$f = \sum m(0, 6, 8, 10, 12, 14, 17, 19, 20, 22, 25, 27, 28, 30)$$

obteve-se

$$f = \bar{A}B\bar{E} + \bar{A}\bar{B}CD\bar{E} + A\bar{C}E + AC\bar{E} + \bar{A}\bar{B}\bar{C}\bar{D}\bar{E}$$

e se a função considerada for

$$f = \sum m(1, 4, 7, 14, 17, 20, 21, 22, 23) + \sum d(0, 3, 6, 19, 30)$$

a função simplificada é:

$$f = \bar{A}\bar{B}\bar{C}E + \bar{A}\bar{B}C\bar{D}\bar{E} + \bar{A}\bar{B}CD + \bar{A}BCD\bar{E} + A\bar{B}\bar{C}E + A\bar{B}C$$

Embora a partir das expressões minimizadas, escritas sob a forma de somas de produtos, se verifique que o número de literais que formam as diversas parcelas obtidas por este método pode ser, em alguns casos, maior do que o número de literais encontrado por outros métodos, os baixos tempos de execução do programa permitem-nos admitir a eficiência do mesmo.

Os diversos testes realizados sugerem então que a qualidade dos resultados obtidos, embora não seja óptima, possa ser considerada boa, não só relativamente às somas de produtos que definem as funções minimizadas, mas fundamentalmente no que se refere aos tempos de execução das simplificações realizadas.

7. Conclusões

Este trabalho pretendeu constituir um contributo para o estudo e resolução de problemas decorrentes da necessidade de minimizar funções lógicas.

Os algoritmos de minimização de funções lógicas podem facilmente atingir, como já se referiu, elevados graus de complexidade, impossibilitando na prática a obtenção de soluções exactas para casos reais. Assim, a pesquisa e utilização de métodos heurísticos que conduzam à redução dos tempos de execução e do espaço de memória utilizado durante a minimização são áreas de investigação de grande importância e actualidade.

As funções lógicas devem ser simplificadas com o duplo objectivo de se obter uma expressão lógica mínima a que corresponda um circuito com bom desempenho e também um tempo de execução aceitável.

Desde o início dos anos 50, quando se iniciou o estudo do projecto digital, têm sido propostos vários métodos de minimização de funções lógicas, alguns dos quais foram referenciados e analisados ao longo do texto, nomeadamente os métodos clássicos de Karnaugh e Quine-McCluskey, os métodos heurísticos que tentam resolver alguns dos problemas levantados pela execução exaustiva inerente aos métodos clássicos e o método baseado na representação das funções booleanas por BDD's, proposto por Akers em finais da década de 70.

Nesta tese foi desenvolvido um programa com o intuito de ser utilizado como "banco de testes" de algumas das estratégias de simplificação propostas. Nesse sentido interligaram-se, por diversas formas, os procedimentos anteriormente descritos, de modo a poderem comparar-se entre si os méritos relativos das heurísticas utilizadas.

Fundamentalmente e resumindo, estudaram-se as seguintes situações:

a) Aplicação inicial de uma simplificação de baixo para cima, feita apenas enquanto são encontrados nós terminais, executada

por **PRSIMPL**, seguida de uma redução do diagrama efectuada de cima para baixo e realizada por **ELIMINAR** e **FUNDIR** - VERSÃO 1.

b) Execução apenas de simplificações realizadas de baixo para cima, utilizando os procedimentos **PRSIMPL** e **SIMPLTOTAL**. Verifica-se neste caso que, embora os tempos de execução sejam idênticos aos obtidos a partir da VERSÃO 1, as expressões lógicas que representam as funções simplificadas podem ter mais parcelas, já que não há eliminação dos nós redundantes do diagrama - VERSÃO 2.

c) Simplificação de funções de saídas múltiplas; as estratégias aqui utilizadas são idênticas às realizadas na VERSÃO 1, pois verificou-se serem estas as mais eficientes. Considerando as funções a simplificar como funções de saídas múltiplas, podem conseguir-se fusões de ramos que doutro modo não seriam possíveis - VERSÃO 3.

O programa foi ensaiado com diversos exemplos e obtiveram-se resultados que confirmaram as previsões iniciais relativamente aos méritos do método investigado, nomeadamente no que se refere à sua eficiência.

Pode considerar-se que os objectivos inicialmente propostos para esta tese foram atingidos. Naturalmente que ao longo do trabalho e, em particular, após a análise dos resultados obtidos, se evidenciaram alguns tópicos que poderão justificar novas linhas de trabalho e investigação a aprofundar no futuro, quer no que respeita à melhoria de aspectos de funcionamento do programa, quer quanto à análise de estratégias de minimização alternativas e possíveis aplicações do programa.

A eficiência do programa na obtenção de uma soma de produtos mínima, não só do ponto de vista do número de parcelas mas também do número de literais de cada parcela, depende consideravelmente da ordenação das variáveis de entrada. Daí as vantagens decorrentes da pesquisa e utilização de algoritmos que conduzam à ordenação óptima das variáveis de entrada da função a minimizar, e que não foram considerados no âmbito desta tese.

Numa outra perspectiva poder-se-ia pensar na possível integração deste programa como parte de um gerador de PLA's e de síntese lógica.

Referências

[AKE77] S. B. Akers, "On the Specification and Analysis of Large Digital Functions", Proc. 7th Int. Symp. Fault-Tolerant Comput., June 1977, pp. 88-93.

[AKE78] S. B. Akers, "Binary Decision Diagrams", IEEE Trans. Comput., vol. C-27, nº 6, June 1978.

[BRA82] R. K. Brayton, G. D. Hachtel, L. Hemachandra, A. R. Newton and A. L. Sangiovanni-Vincentelli, "A Comparision of Logic Minimization Strategies using EXPRESSO. An APL Program Package for Partitioned Logic Minimization", Proc. Int. Symp. on Circ. and Syst., Rome, May 1982, pp. 43-49.

[BRA84] R. K. Brayton, G. D. Hachtel, C. T. McMullen and A. L. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis", Kluwer Publ., 1984.

[BRO81] D. W. Brown, "A State-machine Synthesizer - SMS", Proc. 18th Design Automation Conference, Nashville, June 1981, pp. 301-304.

[BRY86] R. E. Bryant, "Graph-based Algoritms for Boolean Function Manipulation", IEEE Trans. Comput., vol. C-35, nº 8, August 1986, pp. 677-691.

[DEM84] G. DeMicheli, M. Hofmann, R. Newton and A. L. Sangiovanni-Vincentelli, "A System for the Automatic Synthesis of Programmable Logic Arrays", in Advances in Computer-Aided Engineering, Sangiovanni-Vincentelli editor, Jay Press, 1984.

[FRI87] S. Friedman and K. Supowit, "Finding the Optimal Variable Ordering for Binary Decision Diagrams", 24th ACM/IEEE Design Automation Conference, 1987, paper 21.2, pp. 348-356.

[FAB90] E. D. Fabricius, "Introduction to VLSI Design", McGraw-Hill, 1990.

[HIL81] F. J. Hill and G. R. Peterson, "Introduction to the Switching Theory and Logical Design", Wiley, 1981.

[HON74] S. J. Hong, R. G. Cain and D. L. Ostapko, "MINI: a Heuristic Approach for Logic Minimization", IBM Journal of Research and Development, vol. 18, September 1974, pp. 443-458.

[KAN81] S. Kang and W. M. VanCleemput, "Automatic PLA Synthesis from a DDL-P Description", Proceedings of the 18th Design Automation Conference, June 1981, pp. 391-397.

[MAR67] M. P. Marcus, "Switching Circuits for Engineers", 2nd edition, Prentice-Hall, 1967.

[MAT83] J. S. Matos, "The Binary Decision Diagram: a Tool for Logic Design and Implementation", Ph.D. dissertation, Syracuse University, December 1983.

[MCC65] E. J. McCluskey Jr., "Introdution to the Theory of Switching Circuits", McGraw-Hill, 1965.

[MOR82] B. M. E. Moret, "Decision Trees and Diagrams", Computing Surveys, vol. 14, nº 4, December 1982.

[PET59] S. R. Petrick, "On the Minimization of Boolean Functions", Proc. Symp. on Switching Theory, ICIP, Paris, June 1959.

[QUI52] W. V. Quine, "The Problem of Simplifying Truth Functions", Am. Math. Monthly, vol. 59, nº 8, October1952, pp. 521-531.

Apêndice A. Decomposição de Funções

A.1 Cofactores e expansão de Shannon

Dado um conjunto de cubos **G** = $\{c^1, c^2, ..., c^h\}$ e um cubo **p**, todos com **n** entradas e **m** saídas, o cofactor de **G** relativamente a **p** é um conjunto de cubos (eventualmente vazio) que se obtém determinando o cofactor de cada um dos cubos da cobertura **G**.

O cofactor de $\mathbf{c^i}$ (i=1, 2, ..., h) relativamente a **p** é um cubo cujas componentes são:

$$(c^i_p)_k = \begin{cases} \emptyset & \text{se } c^i \cap p = \emptyset \\ 2 & \text{se } p_k=0 \text{ ou } p_k = 1 \\ 4 & \text{se } p_k = 3 \\ c^i_k & \text{nos outros casos} \end{cases}$$

k - é o número de elementos do cubo **p**

$\mathbf{c^i_k}$ - são os elementos dos cubos $\mathbf{c^i}$

$\mathbf{G_p}$ - é o cofactor da matriz **G** relativamente ao cubo **p**

$\mathbf{c^i_p}$ - é o cofactor do cubo $\mathbf{c^i}$ relativamente ao cubo **p**

Por exemplo, o cofactor de

$$G = \begin{vmatrix} 1 & 1 & 0 & 2 & 4 & 4 \\ 0 & 1 & 2 & 0 & 4 & 4 \\ 1 & 1 & 1 & 1 & 4 & 3 \end{vmatrix}$$

relativamente ao cubo $x^4 = [\ 2\ \ 2\ \ 2\ \ 1\ \ 4\ \ 4\ ]$ é

$$G_{x^4} = \begin{vmatrix} 1 & 1 & 0 & 2 & 4 & 4 \\ 1 & 1 & 1 & 2 & 4 & 3 \end{vmatrix}$$

e relativamente ao cubo $\overline{x^4}$ = [2 2 2 0 4 4] é

$$G_{\bar{x}^4} = \begin{vmatrix} 1 & 1 & 0 & 2 & 4 & 4 \\ 0 & 1 & 2 & 2 & 4 & 4 \end{vmatrix}$$

Seja **g** uma representação algébrica da função lógica correspondente à matriz **G**, e $\mathbf{g}_{x_j}$ e $\mathbf{g}_{\bar{x}_j}$ representações algébricas das funções lógicas correspondentes às matrizes $\mathbf{G}_{x^j}$ e $\mathbf{G}_{\bar{x}^j}$; $\mathbf{G}_{x^j}$ é o cofactor de **G** relativamente ao cubo $\mathbf{x}^j$ e $\mathbf{g}_{x_j}$ o cofactor de **g** relativamente à variável $\mathbf{x}_j$. Então, a expressão que traduz a expansão de Shannon de **g** é [AKE78]

$$g = x_j \cdot g_{x_j} + \overline{x}_j \cdot g_{\bar{x}_j}$$

Facilmente se vê que a aplicação sucessiva, mediante a utilização de uma técnica recursiva, da expansão de Shannon à função **g** pode ser equivalente à construção de uma árvore binária em que cada próximo nó será o cofactor do nó anterior (**g**) relativamente, neste caso, à variável $\mathbf{x}_j$ e ao seu complementar (logo $\mathbf{g}_{x_j}$ e $\mathbf{g}_{\bar{x}_j}$).

Esta técnica aplica-se sucessivamente até se obterem cofactores monótonos ou valores constantes ao atingirem-se os nós terminais da árvore [FAB90].

Se agora calcularmos

$$\tilde{G} = x^j \, G_{x^j} + \overline{x^j} \, G_{\bar{x}^j}$$

obtemos um conjunto de cubos que formam uma outra representação da função **g** diferente de **G** mas equivalente a ela. Note-se que $\mathbf{x}^j \, \mathbf{G}_{x^j}$ representa a intersecção de $\mathbf{x}^j$ com $\mathbf{G}_{x^j}$ feita linha a linha. Dizer-se que **G** e $\tilde{\mathbf{G}}$ são **logicamente equivalentes**, isto é têm a mesma tabela de verdade, é dizer-se que $\tilde{\mathbf{G}}$ cobre os mesmos vértices que **G** e portanto ambos correspondem à mesma função lógica **g**.

Relativamente ao exemplo anterior, obtém-se

$$x^4\, G_{x^4} = \begin{vmatrix} 1 & 1 & 0 & 1 & 4 & 4 \\ 1 & 1 & 1 & 1 & 4 & 3 \end{vmatrix}$$

e

$$\overline{x^4}\, G_{\bar{x}^4} = \begin{vmatrix} 1 & 1 & 0 & 0 & 4 & 4 \\ 0 & 1 & 2 & 0 & 4 & 4 \end{vmatrix}$$

logo

$$\tilde{\mathbf{G}} = x^4\, G_{x^4} + \overline{x^4}\, G_{\bar{x}^4} = \begin{vmatrix} 1 & 1 & 0 & 1 & 4 & 4 \\ 1 & 1 & 0 & 0 & 4 & 4 \\ 1 & 1 & 1 & 1 & 4 & 3 \\ 0 & 1 & 2 & 0 & 4 & 4 \end{vmatrix}$$

Partindo da matriz **G**, podemos escrever a representação algébricas de **g**

$$\mathbf{g} = \begin{vmatrix} g_1 \\ g_2 \end{vmatrix} = \begin{vmatrix} \bar{x}_1 x_2 \bar{x}_4 + x_1 x_2 x_3 x_4 + x_1 x_2 \bar{x}_3 \\ x_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 \bar{x}_4 \end{vmatrix}$$

e a expansão de Shannon produz

$$\mathbf{g} = x_4\, g_{x_4} + \overline{x_4}\, g_{\bar{x}_4} = x_4 \begin{vmatrix} x_1 x_2 x_3 + x_1 x_2 \bar{x}_3 \\ x_1 x_2 \bar{x}_3 \end{vmatrix} + \overline{x_4} \begin{vmatrix} \bar{x}_1 x_2 + x_1 x_2 \bar{x}_3 \\ x_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 \end{vmatrix}$$

Quando se diz que **g** é igual a $(\mathbf{x_4\, g_{x_4} + \overline{x_4}\, g_{\bar{x}_4}})$, o sinal igual significa que $(\mathbf{x_4\, g_{x_4} + \overline{x_4}\, g_{\bar{x}_4}})$ tem a mesma tabela de verdade que **g**, isto é, $(\mathbf{x_4\, g_{x_4} + \overline{x_4}\, g_{\bar{x}_4}})$ é logicamente equivalente a **g**.

Referir-se-ão, em seguida, algumas proposições fundamentais usadas no estudo das funções booleanas.

Proposição 1 - Sejam **f** e **g** as representações algébricas de duas funções booleanas completamente especificadas; então

a) $(f \cap g)_{x_j} = (f_{x_j} \cap g_{x_j})$, isto é, o cofactor da intersecção é igual à intersecção dos cofactores;

b) $(\bar{f})_{x_j} = (\overline{f_{x_j}})$, isto é, o cofactor do complemento é o complemento do cofactor.

Proposição 2 - Seja $G = \{ c^i \}$ uma cobertura de cubos e seja **p** um cubo; então

$$p \cap G = p \cap G_p.$$

ou seja

$$(p \cap c^i)_k = (p \cap c^i{}_p)_k \qquad \text{qualquer que seja } k,$$

em que G_p é o cofactor de **G** relativamente a **p**, $c^i{}_p$ é o cofactor do cubo c^i relativamente a **p**, **k** é o número de elementos dos cubos **p** e c^i e **i** é o número de elementos da cobertura **G**.

Proposição 3 - Um conjunto de cubos **C** cobre um cubo **c**, se e só se o cofactor de **C** relativamente a **c**, C_c, for uma tautologia, isto é, se $C_c = 1$.

Recorde-se que o OFF-set de uma tautologia é o conjunto vazio, isto é, qualquer que seja a entrada, a saída da função é igual a 1.

Note-se também que **C** cobre **c**, se e só se $(C \cap c) = c$.

Proposição 4 - Seja $f = x_j . \overline{f_{x_j}} + \overline{x_j} . \overline{f}_{\bar{x}_j}$ a expansão de Shannon de uma função lógica completamente especificada **f**, então **f** será uma tautologia, isto é, $f = 1$, se e só se $f_{x_j} = 1$ e $f_{\bar{x}_j} = 1$.

A.2 Fusão

Os resultados anteriormente obtidos relativamente às operações de intersecção, complementação e tautologia servem de base para a resolução do paradigma geral que pode ser expresso resumidamente do seguinte modo:

1) aplicar a operação em questão aos cofactores;

2) fundir os resultados.

Genericamente diremos que utilizando este paradigma uma função lógica é recursivamente dividida até que cada parte seja monótona.

Seja, por exemplo, **h** = (**f** ∩ **g**); se se aplicar a expansão de Shannon obtém-se

$$h = x_j . h_{x_j} + \overline{x}_j . h_{\bar{x}_j}$$

mas como

$$h_{x_j} = (f \cap g)_{x_j} = f_{x_j} \cap g_{x_j}$$

obtém-se

$$h = (f \cap g) = x_j\, (f_{x_j} \cap g_{x_j}) + \overline{x}_j\, (f_{\bar{x}_j} \cap g_{\bar{x}_j}).$$

Aqui trata-se de se obter a intersecção de **f** e **g** à custa da intersecção dos cofactores respectivos, o que sugere facilmente um processo recursivo.

Analogamente se trataria o problema da complementação de uma função lógica.

Note-se que uma estratégia recursiva aponta para o aparecimento de uma árvore binária, em que cada nó será obtido a partir da fusão dos resultados encontrados para as subárvores correspondentes.

Por exemplo, sejam h_0 e h_1 as funções lógicas correspondentes às subárvores de um dado nó; então o processo de fusão dá-nos como resultado a função total

$$h = x_j\, h_1 + \overline{x}_j\, h_0 \qquad \text{(em que } x_j \text{ e } \overline{x}_j \text{ são variáveis).}$$

Considerando H, H_0 e H_1 coberturas de h, h_0 e h_1, respectivamente, então por aplicação da expansão de Shannon, obtém-se

$$H = x^j\, H_1 + \overline{x^j}\, H_0 \qquad \text{(em que } x^j \text{ e } \overline{x^j} \text{ são cubos).}$$

O principal objectivo será agora manter o resultado na forma o mais compacta possível. Se, por exemplo, $\mathbf{H_0}$ e $\mathbf{H_1}$ forem coberturas primas e mínimas de $\mathbf{h_0}$ e $\mathbf{h_1}$, o que se deseja é obter uma cobertura **H** de **h** que seja também prima e mínima.

Para se conseguir atingir tal objectivo, primeiro transformam-se todos os cubos de **H** em cubos primos, obtendo-se assim uma cobertura prima; depois para se fazer a cobertura ser mínima, removem-se de **H** todos os cubos redundantes.

A obtenção de uma cobertura prima a partir da cobertura dada **H**, é feita por transformação de cada cubo de **H** no cubo o maior possível, mas de modo a que o cubo resultante não fique a conter vértices que não pertenciam a **H**. Assim teremos de mudar os 0's ou 1's da parte de entradas do cubo para 2's e mudar os 3's para 4's na parte de saídas. O cubo assim formado conterá o inicial.

Como $\mathbf{H_0}$ e $\mathbf{H_1}$ são coberturas primas, só na posição **j** poderemos mudar os 0's e 1's em 2's e além disso os 3's não mudam para 4's. Logo, a única possibilidade de se aumentarem os cubos de **H** é mudar para 2's os valores dos elementos da coluna **j** das matrizes $\mathbf{x^j H_1}$ e $\mathbf{\overline{x^j} H_0}$.

Sendo $\mathbf{H_1}$ uma cobertura prima, os seus cubos já são o maior possível; mas ao fazer-se a intersecção (produto) $\mathbf{x^j H_1}$ novamente se colocam 1's na coluna **j** da matriz; logo, só se consegue aumentar a matriz $\mathbf{x^j H_1}$, isto é, transformá-la numa matriz prima, colocando 2's na sua coluna **j**.

Logo, ao estudarmos um cubo de $\mathbf{x^j H_1}$, por exemplo $[c_1, ..., 1, ..., c_{n+m}]$, para que este cubo pertença a **H** e portanto interesse transformá-lo num cubo primo $[c_1, ..., 2, ..., c_{n+m}]$ é necessário que $[c_1, ..., 0, ..., c_{n+m}]$ pertença a $\mathbf{\overline{x^j} H_0}$. E, analogamente, o caso simétrico, ou seja, ao considerar-se um cubo $[c_1, ..., 0, ..., c_{n+m}]$ de $\mathbf{\overline{x^j} H_0}$, para que esse cubo pertença a **H** é necessário que o correspondente cubo $[c_1, ..., 1, ..., c_{n+m}]$ pertença a $\mathbf{x^j H_1}$. Notar que como $\mathbf{H_1}$ e $\mathbf{H_0}$ são coberturas primas, só na coluna **j** de $\mathbf{x^j H_1}$ ou de $\mathbf{\overline{x^j} H_0}$ existem 1's ou 0's respectivamente.

Infelizmente, as operações de aumento da dimensão dos cubos, assim como a transformação da cobertura em cobertura mínima, são muito dispendiosas. Pode contudo optar-se por um processo aproximado mas mais rápido, em desfavor da obtenção de uma

cobertura prima e mínima, encontrando-se assim apenas uma boa cobertura.

Note-se que este processo de fusão não garante que **H** seja um conjunto de primos.

Um melhoramento heurístico que pode ser utilizado consiste em fazer uma prévia ordenação dos cubos de $\mathbf{H_0}$ e $\mathbf{H_1}$ de acordo com a dimensão de cada um, na medida em que um cubo menor não pode conter um maior.

O procedimento EXPAND utiliza estes conceitos quando faz a transformação de uma dada cobertura numa cobertura prima e mínima.

Apêndice B. Funções Monótonas

B.1 Definições e conceitos básicos

Na resolução do paradigma referido no Apêndice A, durante a fase correspondente à aplicação da estratégia recursiva, é necessário seleccionar uma variável x_j, variável de partição, relativamente à qual se calculam os cofactores da função lógica. Essa variável deve ser tal que minimize a complexidade da solução. Então, é possível obterem-se métodos de cálculo eficientes se restringirmos o nosso estudo ao caso particular das funções monótonas e se aplicarmos algumas heurísticas simplificativas.

A escolha da variável de partição deve ser feita de modo a que os cofactores da função lógica completamente especificada **f**, sejam o mais possível aproximados a funções monótonas.

Uma **função lógica f** é **monótona** crescente (decrescente) na variável x_j, se quando x_j variar de 0 para 1, todas as saídas de **f** que mudam, variarem de 0 para 1 (de 1 para 0). Ou, por outras palavras, a função é monótona crescente (decrescente) em x_j, se na expressão lógica que define a função a variável x_j não é (é) complementada. Então, uma função lógica **f** diz-se monótona em x_j se for ou monótona crescente ou monótona decrescente em x_j. Uma função lógica **f** é monótona se for monótona em todas as variáveis. Por exemplo, $f = x_1\bar{x}_2 + \bar{x}_2x_3$ é monótona, porque é monótona crescente em x_1 e x_3 e monótona decrescente em x_2.

Uma cobertura **C** é monótona crescente (decrescente) na variável x_j se todos os cubos de **C** têm um 1 (um 0) ou um 2 na posição **j**. Para uma cobertura ser monótona em x_j, na coluna na coluna **j** só pode ter 1's e 2's (ou 0's e 2's). Uma cobertura é monótona se for monótona em todas as variáveis de entrada.

Verifica-se que se uma cobertura é monótona em x_j, então a função correspondente também é monótona em x_j; contudo, uma função monótona pode ter uma cobertura que não seja monótona. Mas, uma cobertura prima de uma função monótona, é monótona.

Uma função lógica **f** é monótona crescente (decrescente) em x_j se e só se nenhum primo implicante de **f** tiver um 0 (1) na posição **j**,

BIBLIOTECA F.E. PORTO

pois para uma cobertura de **f** ser monótona crescente em x_j, na coluna **j** só podem existir 1's e 2's.

Uma cobertura monótona é uma tautologia se e só se contiver uma linha de 2's.

O complementar de uma função monótona é monótona.

Os cofactores de uma função monótona relativamente a x^j e $\overline{x^j}$ são monótonos.

B.2 Escolha da variável de partição

A obtenção de um algoritmo eficiente para a realização de, por exemplo, intersecção, complementação ou tautologia, a partir do paradigma geral atrás referido, tem como passo mais importante a escolha da **variável de partição**.

A heurística utilizada nessa escolha consiste em fazer com que as coberturas C_x e $C_{\bar{x}}$ das funções lógicas f_x e $f_{\bar{x}}$, sejam monótonas após um número mínimo de partições. Daí o ser necessário escolher para variável de partição, a variável menos monótona, pois elimina a coluna com maior número de 0's e 1's, como veremos em seguida.

Como podemos não ter cubos primos em todos os passos, as funções lógicas f_x e $f_{\bar{x}}$ podem ser monótonas mesmo que as correspondentes coberturas C_x e $C_{\bar{x}}$ não o sejam.

Se uma cobertura é monótona relativamente à variável x_j, então diz-se que essa variável x_j é monótona. Uma variável não monótona, diz-se BINATE. A variável de partição é escolhida entre as variáveis binates; de facto, escolhe-se a variável mais binate para variável de partição.

Notar que uma cobertura, ou uma função lógica, é monótona relativamente à variável x_j, se na coluna **j** só existirem 1's e 2's (ou 0's e 2's), isto é, na coluna **j** não podem existir 0's e 1's simultaneamente. Logo, se na coluna **j** existirem 0's e 1's (e eventualmente 2's) a cobertura ou a função lógica não é monótona relativamente à variável x_j. Então, a variável mais binate, x_j, será

aquela a que corresponde o maior número de 0's e 1's por coluna, ou seja, o menor número de 2's por coluna.

B.3 Complementar de uma cobertura monótona

O complementar de uma cobertura monótona **F** é

$\overline{F} = \overline{x^j}\,\overline{F_{\overline{x}j}} + \overline{F_{xj}}$ se **F** é monótona crescente na variável $\mathbf{x_j}$ (todos os cubos de **F** têm um 1 ou um 2 na posição **j**)

$\overline{F} = x^j\,\overline{F_{xj}} + \overline{F_{\overline{x}j}}$ se **F** é monótona decrescente na variável $\mathbf{x_j}$ (todos os cubos de **F** têm um 0 ou um 2 na posição **j**)

Podemos então decompor o problema da complementação de uma cobertura monótona em dois subproblemas:

1º) calcular $\overline{x^j}\,\overline{F_{\overline{x}j}}$, e isto envolve complementar um reduzido número de cubos, pois | $\mathbf{F_{\overline{x}j}}$ | < | **F** |; e além disso, em cada cubo não é necessário complementar a variável $\mathbf{x_j}$, uma vez que $\mathbf{F_{\overline{x}j}}$ contém uma coluna de 2's na posição **j**;

2º) calcular $\overline{F_{xj}}$, e isto envolve complementar a cobertura inicial com excepção (em cada cubo) da variável $\mathbf{x_j}$, pois | $\mathbf{F_{xj}}$ | = | **F** | uma vez que **F** é monótona crescente em $\mathbf{x_j}$ e portanto na coluna **j** de **F** só existem 1's ou 2's; logo, na coluna **j** de $\mathbf{F_{xj}}$ só vão existir 2´s.

Então o processo de complementação é simples:

a) escolher a variável $\mathbf{x_j}$;

b) recursivamente complementar uma cobertura menor formada por cubos que na coluna **j** só contêm 2's;

c) recursivamente complementar a cobertura inicial mas com a coluna **j** preenchida por 2's;

d) associar a variável $\mathbf{x_j}$ aos cubos do resultado da alínea b);

e) concatenar os resultados c) e d).

A recursividade usada tem um final natural, pois o complemento de uma cobertura sem cubos é o universo e o complemento de uma cobertura sem variáveis é o vazio.

Verifica-se que para o caso de funções de saída múltipla é mais fácil e mais rápido complementar uma saída de cada vez do que complementar a função de saída múltipla.

A chave do sucesso na realização dos procedimentos de complementação consiste na boa selecção da variável de partição $\mathbf{x}_j$. Assim, começa por identificar-se as possíveis variáveis de partição, escolhendo-as no maior cubo de **F**, isto é, o que tem maior número de 2's. Poderão ser variáveis de partição todas as variáveis que pertencendo ao maior cubo de **F** sejam diferentes de 2. Depois de as identificar, escolhe-se a variável (ou variáveis) que aparece mais frequentemente nos outros cubos de **F**, ou seja, a variável correspondente à coluna com menor número de 2's. Essa é a variável mais binate e é a variável de partição.

BIBLIOTECA F.E. PORTO