

# Lógica Computacional

**José Manuel E. Valença \***  
<jmvalenca@di.uminho.pt>

17 de Novembro de 2015

# Conteúdo

<b>1 Introdução às Funções Booleanas</b>	<b>4</b>
1.1 Notação . . . . .	4
1.2 Números Algébricos e Raízes da Unidade . . . . .	5
1.3 Corpos Primos . . . . .	6
1.4 Forma das Funções Booleanas . . . . .	8
1.5 Espetro de Exponentes e Suporte . . . . .	12
1.6 Dualidade Booleana e seu Algoritmo . . . . .	14
1.A Implementação da D.B. . . . .	16
<b>2 Satisfação Proposicional (SAT)</b>	<b>17</b>
2.1 O Problema SAT . . . . .	17
2.1.1 Definições e algoritmos de conversão . . . . .	17
2.1.2 Resolução . . . . .	22
2.1.3 Segurança e Completude de Algoritmos . . . . .	25
2.1.4 Algoritmos de conversão com memória global . . . . .	28
2.1.5 Resolvendo o problema <b>SAT</b> . . . . .	32
2.2 Aplicações . . . . .	37
2.2.1 Inversão da chave numa <i>S-Box</i> . . . . .	38
2.2.2 Outra vez as funções booleanas . . . . .	40
2.2.3 Inversão da chave no corpo binário $\mathbb{F}_{2^n}$ . . . . .	42
2.3 Variantes do problema SAT . . . . .	45
2.3.1 <i>k</i> -SAT . . . . .	46
2.3.2 MAX-SAT . . . . .	48
2.3.3 ALL-SAT e #SAT . . . . .	50
<b>3 Teorias com Satisfação (SMT's)</b>	<b>52</b>
3.1 Introdução . . . . .	52
3.1.1 Objetivo . . . . .	52
3.1.2 Teorias de Base . . . . .	53

<b>CONTEÚDO</b>	<b>2</b>
3.1.3 Satisfação em Teorias de Base . . . . .	54
3.2 Algoritmos SMT . . . . .	55
3.3 Aplicações . . . . .	57
3.3.1 Autómatos Híbridos . . . . .	58
3.3.2 Verificação de Programas Imperativos . . . . .	63
3.4 Sistemas de Transição de 1 <sup>a</sup> Ordem . . . . .	69
3.A Codificações SMT . . . . .	76
<b>4 Verificação de Modelos Finitos</b>	<b>78</b>
4.1 Lógica Temporal . . . . .	78
4.1.1 Estruturas de Kripke . . . . .	80
4.1.2 Sintaxe e semântica de LTL e CTL . . . . .	81
4.2 Máquinas de Estado Finitas . . . . .	84
4.2.1 FSM's determinísticas e probabilísticas . . . . .	85
4.2.2 FSM's não determinísticas . . . . .	87
4.3 Verificação Algoritmica . . . . .	88
4.3.1 Minimização por Bissimulação . . . . .	88
4.3.2 Diagramas de Decisão Binária . . . . .	89
4.3.3 O $\mu$ -Calculus . . . . .	92
4.3.4 Verificação orientada aos BDD's . . . . .	95
4.4 “Bounded Model Checking” (BMC) . . . . .	99
4.4.1 Traços e Laços . . . . .	100
4.4.2 Codificação LTL . . . . .	101
4.4.3 Problemas BMC . . . . .	102
4.5 Exemplos e Aplicações . . . . .	104
4.5.1 Verificação orientada às BDD's . . . . .	104
4.5.2 Verificação em BMC . . . . .	109
<b>5 Programação com Restrições</b>	<b>112</b>
5.1 Introdução . . . . .	112
5.2 Programação Inteira . . . . .	115

5.2.1	Definições . . . . .	115
5.2.2	Algoritmos de MIP . . . . .	117
5.3	Alguns Problemas em Programação Inteira . . . . .	121
5.3.1	O problema da mochila (“knapsack problem”) . . . . .	121
5.3.2	Problemas do empacotamento, cobertura e partição . . . . .	123
5.3.3	Problemas de Grafos . . . . .	125
5.4	Programação Inteira com Restrições (CIP) . . . . .	130
5.4.1	A restrição ”todos diferentes”. . . . .	131
5.4.2	Restrições booleanas . . . . .	132
5.4.3	“Mixed-Integer Quadratic Constrained Programming” . . . . .	133

# 1

## Introdução às Funções Booleanas

### 1.1 Notação

Neste curso vamos mencionar algumas estruturas algébricas que são relevantes aos problemas que iremos analisar:

(i) Vamos mencionar *anéis* tais como o anel dos inteiros  $\mathbb{Z}$ , o anel dos polinómios a uma variável de coeficientes inteiros ou racionais ( $\mathbb{Z}[x]$  e  $\mathbb{Q}[x]$ ) e o anel dos polinómios a  $n$  variáveis de coeficientes inteiros ou racionais ( $\mathbb{Z}[x_1, \dots, x_n]$  e  $\mathbb{Q}[x_1, \dots, x_n]$ ).

(ii) Anéis comutativos onde todos os elementos exceto o zero têm inversa multiplicativa, designam-se por *corpos*. Neste curso usa-se o corpo dos números racionais  $\mathbb{Q}$  e, menos frequentemente, as suas extensões números reais  $\mathbb{R}$ , números complexos  $\mathbb{C}$  e números algébricos  $\mathbb{Q}(\beta)$ <sup>†</sup>. Todos estes corpos têm *característica zero*; isto é,  $x + x = 0$  se e só se  $x = 0$ .

Os restantes corpos são *corpos de característica prima*. Um corpo tem característica  $q$ , sendo  $q$  um inteiro primo, se  $q \times x = 0$  para todo  $x$  no corpo.

(iii) Vamos também mencionar duas formas de *grupos aditivos*:

(a) *Espaços vetoriais*  $V \equiv K^d$ , sendo  $K$  um corpo e  $d$  a *dimensão* do espaço.

Note-se que  $d$  pode ser infinito e neste caso é costume representar o espaço vetorial como  $K^*$  ou como  $K^\omega$ .

(b) *Módulos Inteiros* (eq. *grupos Abelianos*) são grupos aditivos comutativos.

Num tal grupo  $M$ , o fato da adição ser comutativa, permite definir a *multiplicação escalar*  $k x$  como a operação binária  $\mathbb{Z} \times M \rightarrow M$  que verifica:  $0x = 0$ ,  $1x = x$ ,  $(k+l)x = kx + lx$ ,  $(kl)x = k(lx)$ ,  $k0 = 0$  e  $k(x+y) = kx + ky$ .

Representa-se por  $\mathbb{F}_2$  o corpo de característica 2 com elementos  $\{0, 1\}$  equipado com a adição  $\oplus$  e a multiplicação tal que  $y \times x = 1$  se e só se  $x = 1$  e  $y = 1$ . Identificando 1 com o valor lógico **true** e 0 com o valor lógico **false**, a adição  $\oplus$  identifica-se com o operador **xor** e multiplicação com o operador **and**.

Genericamente, em qualquer anel ou corpo, usamos o símbolo  $+$  para representar a adição e o símbolo  $\times$  (por vezes não representado mas assumido implicitamente)

---

<sup>†</sup>

a multiplicação. As exceções são  $\mathbb{F}_2$  ou os espaços vetoriais  $(\mathbb{F}_2)^d$ ; nestes casos a adição é representada por  $\oplus$ .

Somas múltiplas num anel, espaço vetorial ou num corpo de característica  $\neq 2$  são representadas por  $\sum_i$ ; somas múltiplas num corpo de característica 2 (ou nos espaços vetoriais que geram) são representadas por  $\bigoplus_i$ .

Se  $K$  é um corpo, dados vetores  $a, b \in K^d$  ( $d$  não necessariamente finito), define-se produto interno  $a \cdot b$  como o resultado (se existir\*) da soma  $\sum_{i=1}^d a_i \times b_i$ . Se  $K$  tem característica 2 o produto interno escreve-se  $a \cdot b$  e é o resultado de  $\bigoplus_{i=1}^d a_i b_i$ .

O espaço vetorial  $(\mathbb{F}_2)^d$  é também representado como  $\{0, 1\}^d$  e, neste caso, os seus elementos designam-se por “palavras de bits de comprimento  $d$ ” ou, simplesmente,  $d$ -palavras. Uma  $d$ -palavra cujas componentes são sempre 1 representa-se por  $1^d$  ou, simplesmente,  $\mathbf{1}$  se a dimensão está implícita; igualmente  $0^d$  e  $\mathbf{0}$  denotam a palavra com todas as componentes iguais a zero.

Frequentemente é necessário, numa mesma computação, interpretar os objetos como elementos de estruturas algébricas distintas. Tipicamente os elementos 0 e 1 podem aparecer na mesma computação uma vez como valores lógicos, como inteiros ou mesmo como números racionais.

Para ser possível englobar todas estas visões numa mesma teoria lógica coerente é normal introduzir a aritmética baseada nos seguintes princípios:

- (i) 0 e 1 são o mesmo objeto como elemento de  $\mathbb{Z}_2$ , de  $\mathbb{Z}$  ou de  $\mathbb{Q}$
- (ii) Os operadores binários  $(a \oplus b)$  e multiplicação  $(a \cdot b)$  definem-se

$$a \oplus b \equiv (a + b) \bmod 2 \quad , \quad a \cdot b \equiv (a \times b) \bmod 2$$

- (iii) Define-se, nos inteiros, o operador ternário *if-then-else*, **ite** como

$$\text{ite}(c, a, b) \equiv \begin{cases} a & \text{se } c = 1 \bmod 2 \\ b & \text{se } c = 0 \bmod 2 \end{cases}$$

A notação  $c \rightarrow a ; b$  é equivalente a  $\text{ite}(c, a, b)$ .

- (iv) As conjetivas lógicas  $\wedge, \vee, \rightarrow$  e  $\neg$  definem-se como

$$\begin{aligned} a \wedge b &\equiv \text{ite}(a, b, 0) \quad , \quad a \vee b \equiv \text{ite}(a, 1, b) \\ a \rightarrow b &\equiv \text{ite}(a, b, 1) \quad , \quad \neg a \equiv \text{ite}(a, 0, 1) \end{aligned}$$

## 1.2 Números Algébricos e Raízes da Unidade

Os *números algébricos* são as raízes complexas de polinómios com coeficientes racionais. O corpo de *números algébricos*  $\mathbb{Q}(\beta)$  é definida do seguinte modo:

Seja  $\beta$  uma qualquer raiz complexa de um polinómio irreduzível  $p \in \mathbb{Q}[x]$  de grau  $d + 1$ ; então  $\mathbb{Q}(\beta)$  é o sub-corpo de  $\mathbb{C}$  cujos elementos têm a forma  $\sum_{i=1}^d a_i \beta^{i-1}$  com todos  $a_i$ 's números racionais.

---

\*Se a dimensão  $d$  não for finita, não existe garantia que a soma tenha um resultado bem definido.

Equivalentemente pode-se definir os elementos de  $\mathbb{Q}(\beta)$  como polinómios de coeficientes racionais identificados módulo  $p(x)$ ; ou seja, identifica-se  $\mathbb{Q}(\beta)$  com o corpo quociente  $\mathbb{Q}[x]/p(x)$ . Note-se que, se  $p(x)$  for redutível, então  $\mathbb{Q}[x]/p(x)$  é um anel mas não é um corpo.

**EXEMPLO 1.1:** Se for  $p(x) \equiv (x^2 - 1)$ , tem-se  $(x - 1) \times (x + 1) = 0 \pmod{p(x)}$ ; portanto, em  $\mathbb{Q}[x]/p(x)$ , existem elementos não-zero (i.e.  $x + 1$  e  $x - 1$ ) cuja multiplicação é zero.  $\square$

Quando  $\beta$  é uma raiz complexa de um polinómio arbitrário  $p(x) \in \mathbb{Q}[x]$ , as somas  $\sum_{i=1}^d a_i \beta^{i-1}$ , com  $a_i \in \mathbb{Q}$ , formam um sub-anel de  $\mathbb{C}$  representado por  $\mathbb{Q}[\beta]$ .

São particularmente importantes os polinómios da forma  $p_n(x) \equiv (x^n - 1)$ . As raízes  $\zeta$  de  $p_n(x)$  verificam a equação  $\zeta^n = 1$ ; por isso são chamadas *n-raízes da unidade*. A *n*-raiz da unidade  $\zeta$  é *n-primitiva* quando  $\zeta^k \neq 1$  para todo  $k < n$ .

O polinómio  $p_n(x) \equiv (x^n - 1)$  fatoriza sempre em

$$p_n(x) = (x - 1) \times (1 + x + x^2 + \cdots + x^{n-1}) \quad (1.1a)$$

Desta forma as *n*-raízes da unidade são 1 (raiz de  $x - 1$ ) ou então são as raízes de

$$q_n(x) \equiv (x^n - 1)/(x - 1) = (1 + x + \cdots + x^{n-1}) \quad (1.1b)$$

Prova-se que  $q_n(x)$  é irreductível se e só se  $n$  é primo e isto ocorre se e só se todas as raízes de  $q_n(x)$  são distintas e *n*-primitivas. Como consequência, sendo  $\zeta$  uma qualquer raiz *n*-primitiva, verificam-se algumas propriedades importantes:

$$(i) \quad 1 + \zeta + \zeta^2 + \cdots + \zeta^{n-1} = 0$$

Isto resulta do fato de toda a *n*-primitiva  $\zeta$  ser raiz de  $q_n(x)$ .

$$(ii) \quad O \text{ conjunto } \{\}$$

$$(iii) \quad \beta \text{ é uma } n\text{-raiz primitiva se e só se } \beta = \zeta^k \text{ para algum } 0 < k < n.$$

Tem-se  $\zeta^k \neq \zeta^j$  se e só se  $k \neq j \pmod{n}$ ; portanto existem  $n - 1$

### 1.3 Corpos Primos

Um anel  $A$  diz-se *booleano* se, para todo  $a \in A$ , se verifica  $a + a = 0$  e  $a \times a = a$ .

Vamos mencionar *corpos primos*  $\mathbb{F}_p$  (com  $p$  primo) gerados a partir deste anéis; nomeadamente:

(i) O corpo dos booleanos  $\mathbb{F}_2$  definido sobre o conjunto de dois valores  $\{0, 1\}$  com a soma  $+$  definida por `xor` de bits e a multiplicação  $\times$  definida pelo `and` de bits.

Desta forma  $x + x = 0$  e  $x \times x = x$  para todo  $x$ , e  $\mathbb{F}_2$  é um anel booleano.

(ii) Genericamente o corpo  $\mathbb{F}_p$  é um corpo com  $p$  de elementos  $\{0, 1, \dots, p - 1\}$  em que as operações são as equivalentes operações inteiras efetuadas módulo  $p$ .

**EXEMPLO 1.2:** A “tabuada” da soma e multiplicação em  $\mathbb{F}_7$  é

$+$	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

$\times$	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

□

Propriedades de  $\mathbb{F}_p$

Para todo  $a, b \in \mathbb{F}_p$  verifica-se

$$(a + b)^p = a + b \quad , \quad a^p = a$$

O anel  $\mathbb{F}_2[\alpha]$  é definido pelos polinómios numa só variável  $\alpha$  de coeficientes em  $\mathbb{F}_2$ . Um tal polinómio  $p(\alpha)$  diz-se *irreduzível* se não tem nenhum fator distinto de 1 e dele próprio. No anel  $\mathbb{F}_2[\alpha]$  o polinómio  $p(\alpha)$  desempenha um papel análogo ao dos inteiros primos no anel  $\mathbb{Z}$ .

Um corpo  $\mathbb{F}_{2^n}$  é formado por todos os  $2^n$  polinómios em  $\mathbb{F}_2[\alpha]$  de grau inferior a  $n$ , e em que as operações  $+$  e  $\times$  são equivalentes às operações no anel dos polinómios mas efetuadas módulo um dado polinómio irreduzível  $p(\alpha)$ . Este polinómio é *característico* do corpo.

**EXEMPLO 1.3:** O corpo de  $16 = 2^4$  elementos é definido pelo conjunto dos polinómios de coeficientes booleanos e grau inferior a 4

$$\begin{aligned} 0, 1, \alpha, \alpha^2, \alpha^3, 1+\alpha, 1+\alpha^2, 1+\alpha^3, \alpha+\alpha^2, \alpha+\alpha^3, \\ \alpha^2+\alpha^3, 1+\alpha+\alpha^2, 1+\alpha+\alpha^3, 1+\alpha^2+\alpha^3, 1+\alpha+\alpha^2+\alpha^3 \end{aligned}$$

Como polinómio característico vamos usar  $p(\alpha) = 1 + \alpha + \alpha^4$ . Isto significa que todos os múltiplos de  $p(\alpha)$  são iguais a zero; donde  $1 + \alpha + \alpha^4 \equiv 0 \pmod{p(\alpha)}$  o que implica

$$\alpha^4 \equiv 1 + \alpha \pmod{p(\alpha)}$$

Portanto as multiplicações em  $\mathbb{F}_{16}$  são as usuais multiplicações de polinómios seguidas da substituição de  $\alpha^4 \sim 1 + \alpha$ . Por exemplo,

$$\begin{aligned} (1 + \alpha + \alpha^3) \times (\alpha + \alpha^2) &= (\alpha + \alpha^2 + \alpha^4 + \alpha^2 + \alpha^3 + \alpha^5) = \\ &= (\alpha + \alpha^2 + 1 + \alpha + \alpha^2 + \alpha^3 + \alpha + \alpha^2) = 1 + \alpha^2 + \alpha^3 \end{aligned} \quad (1.2)$$

Note-se que, como neste corpo  $a+a$  é sempre 0, na soma pares de elementos iguais anulam-se mutuamente. □

Propriedades de  $\mathbb{F}_q$  com  $q = 2^n$

Para todo  $a, b \in \mathbb{F}_q$ , verifica-se

$$(i) \quad a^q = a \quad \text{e} \quad (a + b)^2 = a^2 + b^2.$$

(ii)  $\sum_{i=0}^{n-1} a^{2^i}$  é um elemento de  $\mathbb{F}_2$ .

Este elemento designa-se por *traço* de  $a$  e representado por  $\text{tr}(a)$ .

(iii)  $\text{tr}(a + b) = \text{tr}(a) + \text{tr}(b)$

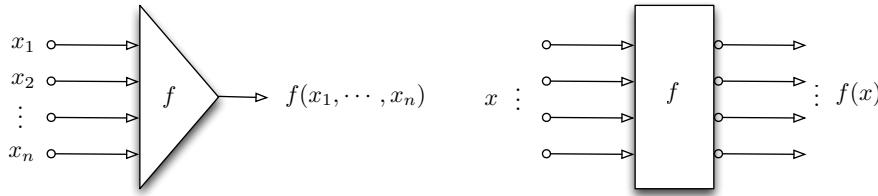
Se  $K$  é um qualquer corpo dos que referimos anteriormente, o *espaço vetorial*  $K^n$  é um grupo dos  $n$ -tuplos  $\langle a_1, \dots, a_n \rangle$ , com  $a_i \in K$ , em que a operação de grupo é a soma efetuada componente a componente.

Se  $A$  é qualquer um anéis anteriormente referidos, o anel  $A^n$  tem como componentes os  $n$ -tuplos  $\langle a_1, \dots, a_n \rangle$ , com  $a_i \in A$ , e as operações do anel são a soma e multiplicação de  $A$  efetuadas componente a componente.

## 1.4 Forma das Funções Booleanas

As *funções booleanas* descrevem-se de duas formas básicas

$$f : (\mathbb{F}_2)^n \rightarrow \mathbb{F}_2 \quad \text{ou} \quad f : \mathbb{F}_{2^n} \rightarrow \mathbb{F}_{2^n} \quad (1.3)$$



Na primeira forma, dita *lógica*, o domínio é grupo dos vetores de  $n$  bits. Na segunda forma, dita *algébrica*, o domínio é o corpo finito de  $2^n$  elementos. A imagem, na forma lógica, é um bit enquanto que na forma algébrica é um elemento do corpo  $\mathbb{F}_{2^n}$  (equivalente a um vetor de  $n$  bits)\*.

Como, em qualquer das formas, o domínio é finito, toda a função booleana é sempre descrita por um polinómio. Assim, na representação lógica, como  $\mathbb{F}_2$  é um anel booleano, a função  $f$  é descrita por um polinómio booleano a  $n$  variáveis. Desta forma, em cada monómio, cada variável só pode ter o expoente 0 ou 1

**EXEMPLO 1.4:** Exemplos de funções booleanas a 4 variáveis,  $F(a_1, a_2, a_3, a_4)$ , em representação lógica

$$0, 1, x_1 + x_2 + x_3 + x_4, \quad x_1 x_2 + x_1 x_3 + x_1 x_4 + x_2 x_3 + x_2 x_4 + x_3 x_4 \\ x_1 x_2 x_3 + x_1 x_2 x_4 + x_1 x_3 x_4 + x_2 x_3 x_4, \quad x_1 x_2 x_3 x_4$$

Todas estas funções gozam da propriedade de serem invariantes face a qualquer permutação das variáveis. Tais funções designam-se por *simétricas*.  $\square$

\*Uma forma híbrida, considera funções  $f : (\mathbb{F}_{2^n})^m \rightarrow \mathbb{F}_{2^n}$  com  $n \times m$  bits de “input” e  $n$  bits de “output”. Uma tal função é representada por um polinómio  $f(x_1, \dots, x_m) \in \mathbb{F}_{2^n}[x_1, \dots, x_m]$ .

Na representação algébrica, porque  $\mathbb{F}_{2^n}$  não é um anel booleano (exceto se  $n = 1$ ), já  $f$  não é um polinómio booleano; é ao invés, um polinómio numa só variável  $x$ , não booleano, de grau inferior a  $2^n$  (porque  $x^{2^n} = x$ ).

**EXEMPLO 1.5:** Seja  $n = 4$ ; o corpo resultante é  $\mathbb{F}_{16}$  que foi apresentado no exemplo 3. Exemplos de funções definidas por duas constantes  $a, b \in \mathbb{F}_{16}$ .

$$f(x) = a \times x^2 + b \times x^4 \quad , \quad f(x) = a \times x + b \times x^8$$

Ambas estas funções são *lineares*: i.e. verificam a relação

$$f(x+y) = f(x) + f(y)$$

Como  $x^{16} = x$  então, para  $x \neq 0$ , tem-se  $x^{15} = 1$ . Como  $x^{15} = x \times x^{14} = 1$ , conclui-se que  $x^{14} = x^{-1}$ . Assim a função inversa  $x \mapsto x^{-1}$  define-se, para  $x \neq 0$ , como  $f(x) = x^{14}$ .

Genericamente, para todo  $n$ , o polinómio  $f(x) = x^{2^n-2}$  representa a função pseudo-inversa

$$f(x) = \begin{cases} x^{-1} & \text{se } x \neq 0 \\ 0 & \text{se } x = 0 \end{cases} \quad (1.4)$$

Este tipo de função é muito usado em aplicações criptográficas; por exemplo, a cifra **AES** que é atualmente a cifra standard, usa um corpo finito  $\mathbb{F}_{256}$  ( $n = 8$ ) e a função  $x \mapsto x^{254}$  como a base para garantir não-linearidade. Mais precisamente, a função não-linear **AES**, tem a forma  $f(x) = (a + b \times x)^{254}$ , sendo  $a, b \in \mathbb{F}_{256}$  constantes definidas no standard.  $\square$

Dado que os espaços  $(\mathbb{F}_2)^n$  e  $\mathbb{F}_{2^n}$  são isomórficos, é importante criar um método para converter uma função booleana, da forma algébrica para a forma lógica e vice-versa.

Um elemento  $a \in \mathbb{F}_{2^n}$  é representado por um polinómio em  $\alpha$  de grau menor do que  $n$ ,  $a = a_1 + a_2 \alpha + \cdots + a_n \alpha^{n-1}$ ; o vetor de coeficientes  $\bar{a} = \langle a_1, \dots, a_n \rangle$  é a representação de  $a$  em  $(\mathbb{F}_2)^n$ . Inversamente, um qualquer vetor  $x \in (\mathbb{F}_2)^n$  constrói o elemento  $\underline{x} = \sum_{i=1}^n x_i \alpha^{i-1}$  que representa  $x$  em  $\mathbb{F}_{2^n}$ . A computação que permite determinar as conversões  $a \mapsto \bar{a}$  e  $x \mapsto \underline{x}$  são definidas pelos seguintes algoritmos.

---

**§ 1.1 Conversão  $(\mathbb{F}_2)^n \rightleftarrows \mathbb{F}_{2^n}$ .**


---

**- Conversão  $(\mathbb{F}_2)^n \rightarrow \mathbb{F}_{2^n}$** 

- (a) Defina-se o vetor  $\beta = \langle 1, \alpha, \dots, \alpha^{n-1} \rangle$ ; genericamente  $\beta_i = \alpha^{i-1}$ .  
(b) Dado  $x \in (\mathbb{F}_2)^n$  determina-se  $\underline{x} \leftarrow x \beta^\top$

**- Conversão  $\mathbb{F}_{2^n} \rightarrow (\mathbb{F}_2)^n$** 

- (a) Define-se  $\lambda: \mathbb{F}_q \rightarrow (\mathbb{F}_q)^n$ , com  $q = 2^n$ , por  $\lambda(a)_j = a^{2^{j-1}}$  para  $j = 1..n$

$$\lambda(a) = \langle a, a^2, a^4, \dots, a^{2^{n-1}} \rangle$$

+  $\lambda(a)$  é o vetor cuja primeira componente é  $a$  e as restantes obtêm-se elevando ao quadrado a componente anterior; note-se que  $a^{2^n} = a$  e, portanto, calculando o quadrado da última componente regressa-se à primeira.  $\square$

- (b) Calcular as matrizes  $G = \lambda(\beta^\top)$  e  $T = G G^\top$ .

+ A matriz  $G$  tem o elemento genérico  $G_{i,j} = (\beta_i)^{2^{j-1}}$ ; i.e. a  $i$ -ésima linha da matriz  $G$  coincide com  $\lambda(\beta_i)$ . Dado que  $T = G G^\top$ , o elemento  $T_{i,j}$  é o produto interno  $\lambda(\beta_i) \lambda(\beta_j)^\top$ ; pela definição de  $\lambda$  será  $T_{i,j} = \text{tr}(\beta_i \beta_j)$ .  $\square$

- (c) Calcular a *matriz de conversão*  $\mathcal{B} \leftarrow G^\top T^{-1}$ .

+ Note-se que a matriz  $B$ , tal como as matrizes  $G$  e  $T$ , depende apenas da “base”  $\beta$  e não do elemento a converter.  $\square$

- (d) Para converter  $a \in \mathbb{F}_{2^n}$  calcula-se

$$\bar{a} \leftarrow \lambda(a) \mathcal{B} \quad (1.5)$$


---

**Justificação** Note-se que, sendo  $a = \sum_{i=1}^n \bar{a}_i \beta_i$  e atendendo que os  $\bar{a}_i$  são bits, pode-se construir as seguintes igualdades calculando quadadros sucessivos desta igualdade.

$$\begin{aligned} a &= \bar{a}_1 \beta_1 + \dots + \bar{a}_n \beta_n \\ a^2 &= \bar{a}_1 (\beta_1)^2 + \dots + \bar{a}_n (\beta_n)^2 \\ a^4 &= \bar{a}_1 (\beta_1)^4 + \dots + \bar{a}_n (\beta_n)^4 \\ \dots &\dots \\ a^{2^{n-1}} &= \bar{a}_1 (\beta_1)^{2^{n-1}} + \dots + \bar{a}_n (\beta_n)^{2^{n-1}} \end{aligned}$$

Tem-se  $a^{2^{j-1}} = \sum_{i=1}^n \bar{a}_i (\beta_i)^{2^{j-1}}$ ; sendo  $G_{i,j} \stackrel{\text{def}}{=} (\beta_i)^{2^{j-1}}$ , conclui-se que  $\lambda(a) = \bar{a} G$ .

Da relação  $\lambda(a) = \bar{a} G$ , tem-se  $\lambda(a) G^\top = \bar{a} G G^\top$ ; como  $T = G G^\top$  tem-se  $\lambda(a) G^\top = \bar{a} T$  e, portanto,  $\lambda(a) G^\top T^{-1} = \bar{a}$ .  $\square$

**EXEMPLO 1.6:** O sistema **SAGE**, que vai ser usado neste exemplo e outros exemplos futuros, obtém-se em [www.sagemath.org](http://www.sagemath.org). É um calculador baseado em **python** que implementa um enorme número de estruturas algébricas essenciais em Ciências da Computação. No-meadamente permite calcular sobre corpos finitos.

Considere-se  $n = 4$  e vamos procurar calcular a conversão  $\mathbb{F}_{16} \rightarrow (\mathbb{F}_2)^4$  com o polinómio característico  $\alpha^4 + \alpha + 1$ . Seguindo o algoritmo anterior e usando **SAGE** calculam-se a matrizes  $G$  e  $T$

$$G = \begin{pmatrix} 1 & 1 & 1 & 1 \\ \alpha & \alpha^2 & \alpha+1 & \alpha^2+1 \\ \alpha^2 & \alpha+1 & \alpha^2+1 & \alpha \\ \alpha^3 & \alpha^3+\alpha^2 & \alpha^3+\alpha^2+\alpha+1 & \alpha^3+\alpha \end{pmatrix}, \quad T = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

A matriz  $\mathcal{B} \stackrel{\text{def}}{=} G^\top T^{-1}$  é

$$\mathcal{B} = \begin{pmatrix} \alpha^3 + 1 & \alpha^2 & \alpha & 1 \\ \alpha^3 + \alpha^2 + 1 & \alpha + 1 & \alpha^2 & 1 \\ \alpha^3 + \alpha^2 + \alpha & \alpha^2 + 1 & \alpha + 1 & 1 \\ \alpha^3 + \alpha + 1 & \alpha & \alpha^2 + 1 & 1 \end{pmatrix}$$

Tome-se, por exemplo,  $a = 1 + \alpha$ ; então  $\lambda(a) = (\alpha + 1 \ \alpha^2 + 1 \ \alpha \ \alpha^2)$ ; pode-se verificar que  $\lambda(a)\mathcal{B} = (1 \ 1 \ 0 \ 0)$ .  $\square$

Uma vez obtida a conversão para valores, a conversão para funções é simples. Note-se porém que uma função booleana na forma lógica tem imagem em  $\mathbb{F}_2$  enquanto que na forma algébrica tem imagem em  $\mathbb{F}_{2^n}$ . Por isso o espaço das funções booleanas algébricas  $\mathbb{F}_{2^n} \rightarrow \mathbb{F}_{2^n}$  não é isomórfico com o espaço das funções booleanas lógicas mas antes com o espaço das funções  $f: (\mathbb{F}_2)^n \rightarrow (\mathbb{F}_2)^n$ .

(i) *Conversão de  $f: \mathbb{F}_{2^n} \rightarrow \mathbb{F}_{2^n}$  para  $\bar{f}: (\mathbb{F}_2)^n \rightarrow (\mathbb{F}_2)^n$ .*

$$y \leftarrow \bar{f}(x) \equiv a \leftarrow x\beta^\top, b \leftarrow f(a), y \leftarrow \lambda(b)\mathcal{B}$$

(ii) *Conversão de  $f: (\mathbb{F}_2)^n \rightarrow (\mathbb{F}_2)^n$  para  $\underline{f}: \mathbb{F}_{2^n} \rightarrow \mathbb{F}_{2^n}$ .*

$$b \leftarrow \underline{f}(a) \equiv x \leftarrow \lambda(a)\mathcal{B}, y \leftarrow f(x), b \leftarrow y\beta^\top$$

**EXEMPLO 1.7:** A segurança da cifra AES é assegurada por uma função booleana  $f: \mathbb{F}_{256} \rightarrow \mathbb{F}_{256}$  definida do seguinte modo

$$f(x) = a(\alpha) + b(\alpha) \times x^{-1}(\alpha) \pmod{\alpha^8 + 1}$$

sendo  $x, a, b$  polinómios em  $\alpha$  de grau inferior a 8 (note-se que  $256 = 2^8$ ) em que as constantes  $a$  e  $b$  são representados pelos vetores de bits  $\bar{a} = \mathbf{c6}$  e  $\bar{b} = \mathbf{f1}$ .

A transformação pode ser implementada eficientemente numa S-Box  $8 \times 8$  invertível. Na tabela 7 é apresentada essa S-Box. A transformação inversa implementa-se usando esta mesma S-Box mas em sentido oposto.

A constante  $a$  é escolhida de modo que a S-Box não tenha pontos fixos (pontos  $x$  onde  $S(x) = x$ ) nem anti-pontos fixos (pontos  $x$  onde  $S(x) = 1 + x$ ).

*S-Box da cifra Rijndael*

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	c6	37	87	47	df	46	6	ac	f3	e0	86	42	1f	8d	4a	97
1	5c	d8	6c	27	5f	65	84	ff	2a	bd	da	a	39	ba	d7	fc
2	8b	2f	c9	92	93	3	8f	3c	b3	aa	ae	ef	e7	7d	e3	a1
3	b0	8c	c2	cc	71	99	a0	59	80	d1	f8	de	4e	82	db	a7
4	60	c8	32	51	41	16	55	fa	d5	43	9d	cb	62	ce	2	b8
5	c5	ed	f0	2e	f2	3f	eb	45	56	4c	1b	63	54	34	75	c
6	fd	e	5a	4f	c4	24	c3	a8	a4	6f	d0	7	f5	33	9	7a
7	e5	ca	f4	8	d9	29	73	af	3b	9b	5d	e2	f1	f	cf	dd
8	2c	30	c1	3e	5	89	b4	81	bc	8a	17	23	b6	25	61	c7
9	f6	e8	4	3d	d2	52	f9	78	94	1e	7b	b1	1d	15	40	4d
a	fe	d3	53	50	64	90	b2	35	dc	cd	3a	d6	e9	a9	be	67
b	8e	7c	83	26	28	ad	14	6a	36	95	bf	5e	a6	57	1a	70
c	5b	77	a2	12	31	9a	bb	9c	7e	2d	b7	1	44	2b	48	58
d	f7	13	ab	96	74	c0	9f	10	e6	a3	85	6b	98	ec	21	19
e	ee	7f	79	e1	66	6d	18	b9	49	11	88	6e	1c	a5	72	d
f	38	ea	68	20	b	9e	d4	76	e4	69	22	0	fb	b5	4b	91

□

## 1.5 Espetro de Exponentes e Suporte

Neste curso interessam-nos essencialmente as formas lógicas  $f: (\mathbb{F}_2)^n \rightarrow \mathbb{F}_2$  das funções booleanas. Nesta secção vamos estudar a representação destas funções.

Começamos por representar por  $\mathcal{B}_n$  o anel formado pelo conjunto de todas as funções  $(\mathbb{F}_2)^n \rightarrow \mathbb{F}_2$  equipado com a soma e multiplicação de funções definidas por

$$(f + g)(x) = f(x) + g(x) \quad , \quad (f \times g)(x) = f(x) \times g(x) \quad (1.6)$$

Vimos que uma função booleana com  $n$  “inputs” booleanos, é sempre descrita por polinómio booleano nas variáveis  $x_1, \dots, x_n$ . Portanto, a representação mais óbvia, está relacionada com a “melhor” forma de representar tais polinómios. Tem-se

- i) Um *polinómio* é uma soma de *monómios*; como a ordem das parcelas é irrelevante pode-se representar um *polinómio* só pelo *conjunto dos seus monómios*. Por exemplo,  $f(x_1, \dots, x_4) = 1 + x_1 x_3 + x_2 x_2 x_4$  é representado pelo conjunto  $\{1, x_1 x_3, x_2 x_2 x_4\}$ . O conjunto vazio  $\{\}$  representa a função 0.
- ii) Um *monómio* é um produto  $\prod_{i=1}^n x_i^{e_i}$  de termos  $x^e$ , sendo  $x$  uma variável e  $e$  um *exponente*. Como o polinómio é booleano, tem-se  $x^e = x$  para todo  $e > 0$ ; portanto só é relevante saber se  $e = 0$  ou  $e \neq 0$ ; neste caso, assume-se  $e = 1$ .

Por exemplo, na função anterior, tem-se

$$1 = x_1^0 x_2^0 x_3^0 x_4^0 \quad , \quad x_1 x_3 = x_1^1 x_2^0 x_3^1 x_4^0 \quad , \quad x_2 x_2 x_4 = x_1^0 x_2^1 x_3^1 x_4^1$$

Este exemplo ilustra como cada monómio  $x_1^{e_1} \cdots x_n^{e_n}$  é completamente determinado pelo vetor dos exponentes  $e = \langle e_1, \dots, e_n \rangle$ . Assim

$$1 \rightsquigarrow 0000 \quad , \quad x_1 x_3 \rightsquigarrow 1010 \quad , \quad x_2 x_2 x_4 \rightsquigarrow 0111$$

A função  $f = 1 + x_1 x_3 + x_2 x_2 x_4$  é representada pelo conjunto dos seus monómios e, por sua vez, cada monómio é representado pelo vetor dos seus exponentes. Por isso, a representação de  $f$  é o conjunto

$$E(f) = \{0000, 1010, 0111\} \quad (1.7)$$

que se designa por *espetro de exponentes* de  $f$ .

Para formalizar estes conceitos vamos introduzir alguma notação.

- a)  $\mathcal{L}_n$  é espaço das linguagens das *strings* de bits de comprimento  $n$ .

Note-se que uma tal linguagem é um subconjunto  $L \subseteq \{0, 1\}^n$ ; portanto  $\mathcal{L}_n$  é o *power set*  $\wp(\{0, 1\}^n)$  também representado por

$$2^{\{0,1\}^n}$$

Como  $\{0, 1\}$  tem  $2^n$  elementos distintos, o *power set*  $\mathcal{L}_n$  tem  $2^{2^n}$  elementos.

- b) Para qualquer string  $e \in \{0, 1\}^n$ , representamos por  $x^e$  o monómio  $\prod_{i=1}^n x_i^{e_i}$ .
- c) Uma função  $f \in \mathcal{B}_n$  é completamente descrita pelo seu *espetro de expoentes*  $E(f) \in \mathcal{L}_n$ . O operador *espetro*  $E: \mathcal{B}_n \rightarrow \mathcal{L}_n$  é o isomorfismo que verifica

$$f = \sum_{e \in E(f)} x^e \quad (1.8)$$

Como o número de espetros distintos é igual à cardinalidade de  $\mathcal{L}_n$  (i.e.  $2^{2^n}$ ) concluímos que existem  $2^{2^n}$  funções booleanas distintas com  $n$  “inputs”.

Dado que as funções booleanas definem um anel  $\mathcal{B}_n$  é importante analisar se o espaço de linguagens  $\mathcal{L}_n$  também pode ser equipado com a estrutura de um anel. De facto  $\mathcal{L}_n$  pode ser equipado com a estrutura de anel através de um par de operações soma e multiplicação; note-se que os elementos de  $\mathcal{L}_n$  são conjuntos de vetores e portanto soma e multiplicação têm de ser funções que recebem dois conjuntos como operandos e devolvem um outro conjunto como resultado.

### 1.1 DEFINIÇÃO Dados $L, K \in \mathcal{L}_n$

+ a soma  $L \oplus K$  define-se como a diferença simétrica destes conjuntos;

$$L \oplus K \stackrel{\text{def}}{=} (L \setminus K) \cup (K \setminus L)$$

+ a multiplicação  $L \otimes K$  define-se por

$$L \otimes K \stackrel{\text{def}}{=} \bigoplus_{x \in L, y \in K} \{x \vee y\}$$

sendo  $x \vee y$  o vetor que se obtém por disjunção componente a componente de  $x$  e  $y$ ; ou seja,  $(x \vee y)_i = \max(x_i, y_i)$ .

Com esta estrutura de anéis é fácil verificar que o operador  $E: \mathcal{B}_n \rightarrow \mathcal{L}_n$  é um isomorfismo de anéis. De fato verifica-se

$$E(f + g) = E(f) \oplus E(g) \quad , \quad E(f \times g) = E(f) \otimes E(g) \quad (1.9)$$

**EXEMPLO 1.8:** Considere-se  $n = 4$  e duas funções  $f, g \in \mathcal{B}_4$ .

$$f = x_1 x_3 + x_2 x_4 + x_1 x_2 \quad , \quad g = x_1 x_2 + x_3 x_4$$

cujos espetros de expoentes são

$$E(f) = \{1010, 0101, 1100\} \quad , \quad E(g) = \{1100, 0011\}$$

Tem-se  $E(f) \oplus E(g) = \{1010, 0101, 0011\}$ ; donde  $f + g = x_1 x_3 + x_2 x_4 + x_3 x_4$ . A multiplicação produz  $E(f) \otimes E(g) = \{1110, 1011, 1101, 0111, 1100, 1111\}$ .  $\square$

O *suporte* de uma função booleana  $f \in \mathcal{B}_n$ , representado por  $\text{sup}(f)$ , é o conjunto dos vetores  $z \in (\mathbb{F}_2)^n$  onde  $f(z)$  tem o valor 1. Formalmente

$$\text{sup}(f) \stackrel{\text{def}}{=} \{z \in \{0, 1\}^n \mid f(z) \neq 0\} \quad (1.10)$$

Vemos que  $\text{sup}(f)$  é, tal como  $E(f)$ , um elemento de  $\mathcal{L}_n$ ; portanto pode-se ver  $\text{sup}$ , tal como  $E$ , como um operador  $\mathcal{B}_n \rightarrow \mathcal{L}_n$ .

Duas funções distintas têm obviamente suportes distintos. Por outro lado, dado um qualquer  $S \in \mathcal{L}_n$ , é simples reconstruir a função  $f \in \mathcal{B}_n$  que tem  $S$  como suporte: basta fazer  $f(z) = 1$ , se  $z \in S$ , e  $f(z) = 0$  em caso contrário. Desta forma  $\text{sup}$  determina igualmente um isomorfismo entre  $\mathcal{B}_n$  e  $\mathcal{L}_n$ .

$$\begin{array}{ccc} & \mathcal{B}_n & \\ E \swarrow & & \searrow \text{sup} \\ \mathcal{L}_n & \xrightarrow{\pi_n} & \mathcal{L}_n \\ & \xleftarrow{\pi_n^{-1}} & \end{array}$$

Dado que tanto  $E$  como  $\text{sup}$  são isomorfismos, existe endomorfismo  $\pi_n: \mathcal{L}_n \rightarrow \mathcal{L}_n$  que faz comutar este diagrama. O operador  $\pi$  é uma aplicação que toma como argumento o espetro de expoentes de uma função  $f$  e calcula o seu suporte; obviamente  $\pi^{-1}$  faz a transformação inversa: toma o suporte como argumento e calcula o espetro; isto é,  $\pi(E(f)) = \text{sup}(f)$  e  $\pi_n^{-1}(\text{sup}(f)) = E(f)$ .

Vamos verificar que  $\pi$  coincide com a sua própria inversa: i.e. é uma *involução*. Por isso, para todo  $L \in \mathcal{L}_n$  tem-se  $\pi(\pi(L)) = L$ ; isto significa que, se  $L$  for o suporte de uma função, então  $\pi(L)$  é o espetro dessa função; inversamente, se  $L$  for o espetro de uma função, então  $\pi(L)$  é o suporte da função.

Desta forma, todas as asserções que sejam feitas em relação ao espetro de uma função, refletem-se (por aplicação de  $\pi$ ) no seu suporte, e vice-versa. Por isso  $\pi$  pode ser visto como uma *dualidade*. Por exemplo, dada uma qualquer função  $f \in \mathcal{B}_n$ , a sua *função dual*  $f'$  é a função cujo espetro coincide com o suporte de  $f$ .

## 1.6 Dualidade Booleana e seu Algoritmo

Computar a transformação  $\pi_n: \mathcal{L}_n \rightarrow \mathcal{L}_n$ , que mapeia o espetro de uma função no seu suporte e vice-versa o suporte no espetro, é uma operação fundamental no estudo das funções booleanas. Esta secção é dedicada às propriedades de  $\pi$ , nomeadamente à prova de que é uma involução, e aos algoritmos que permitem computar tal transformação.

Como exemplo considere-se a função  $f \in \mathcal{B}_3$  dada pelo polinómio

$$f = 1 + x_1 x_2 + x_1 x_3 + x_2 x_3$$

Vamos determinar o seu espetro em  $\mathcal{L}_3$  seguindo os seguintes passos:

- i) “*Split*”: em  $f$  isolamos os monómos que contêm  $x_3$  dos restantes; isto é, escreve-se

$$f = (1 + x_1 x_2) + (x_1 + x_2) x_3$$

Constrói-se assim duas funções  $g, h \in \mathcal{B}_2$  tais que

$$f = g(x_1, x_2) + h(x_1, x_2) x_3$$

- ii) “*Group*” somando duas vezes  $(1 + x_1 x_2) x_3$  reescreve-se como

$$f = (1 + x_1 x_2) (1 + x_3) + ((1 + x_1 x_2) + (x_1 + x_2)) x_3$$

Constrói-se assim duas funções  $g$  e  $g + h$  que não contêm  $x_3$  e verificam

$$f = g(1 + x_3) + (g + h)x_3 \quad (1.11)$$

- iii) “*Solve*”: recursivamente calcula-se  $G = \text{sup}(g)$  e  $H = \text{sup}(g + h)$  em  $\mathcal{L}_2$
- iv) “*Join*”

Vemos em 1.11 que, quando  $x_3 = 0$ , o suporte de  $f$  é determinado por  $G$ ; i.e. todos os  $x = \langle x_1, x_2, 0 \rangle$  que pertencem a  $\text{sup}(f)$  são determinados pelos pares  $\langle x_1, x_2 \rangle \in G$ .

Do mesmo modo, quando  $x_3 = 1$ , o suporte de  $f$  é determinado por  $H$ ; i.e. todo  $\langle x_1, x_2, 1 \rangle \in \text{sup}(f)$  é determinado pelos pares  $\langle x_1, x_2 \rangle \in H$ .

Portanto

$$\begin{aligned} \text{sup}(f) = & \{ \langle x_1, x_2, 0 \rangle \mid \langle x_1, x_2 \rangle \in \text{sup}(g) \} \cup \\ & \{ \langle x_1, x_2, 1 \rangle \mid \langle x_1, x_2 \rangle \in \text{sup}(g + h) \} \end{aligned} \quad (1.12)$$

Para formalizar este método e construir um algoritmo que o implemente, é necessário introduzir dois novos operadores nos anéis  $\mathcal{L}_n$ .

- + O operador *join* mapeia  $L, K \in \mathcal{L}_n$  na linguagem em  $\mathcal{L}_{n+1}$  definida por

$$\text{join}(L, K) \stackrel{\text{def}}{=} \{x0 \mid x \in L\} \cup \{y1 \mid y \in K\} \quad (1.13)$$

Este operador é também conhecido como *união disjunta* de linguagens.

- + O operador *split* é inverso do *join*; decompõe uma linguagem  $L \in \mathcal{L}_n$ , com  $n > 0$ , em duas linguagens  $K, J \in \mathcal{L}_{n-1}$  tais que  $L = \text{join}(K, J)$ .

Tem-se  $\langle K, J \rangle = \text{split}(L)$ , com  $L \in \mathcal{L}_n$  e  $n > 0$ , se e só se

$$K = \{x \mid x0 \in L\}, \quad J = \{x \mid x1 \in L\} \quad (1.14)$$

```
-- constrói a linguagem dual do parâmetro L
fun dual (L) is
    if L = {} or L = {ε} then return L
    -- linguagem vazia denota 0, a linguagem {ε} denota 1

    ⟨K, J⟩ ← split(L), J ← K ⊕ J -- split+group
    G ← dual(K), H ← dual(J) -- chamada duplamente recursiva

    return join(G, H)
```

**Algorithm 1.1:** Algoritmo da Dualidade Booleana - versão básica

Para demonstrar que  $\text{dual}(\text{dual}(L)) = L$  procedemos por indução; o algoritmo torna claro que esta relação se verifica quando  $L$  é vazio ou quando  $L = \{\varepsilon\}$  porque, nestes casos,  $\text{dual}(L) = L$ ; se  $L$  é não-vazia e os seus elementos têm comprimento  $> 0$ , então assume-se a hipótese de indução de que a relação se verifica para  $J$  e  $K$  e, desta forma, é simples provar que a relação também é válida para  $L$ .

Note-se que o algoritmo elimina uma variável em cada chamada; no entanto cada chamada dá origem a outras duas chamadas recursivas; por isso, no pior caso, existem  $2^n$  chamadas até que o algoritmo pare.

## Apêndice 1.A Implementação Python do Algoritmo da Dualidade Booleana

```
class LingError(Exception): ...  
  
class Ling(object):  
    def __init__(self,value=None,N=None): ....  
  
    def __str__(self): ...  
  
    def add(self,L):  
        self.value ^= L.value  
        return self  
  
    def split(self,s):  
        if (s != '0') and (s != '1'):  
            raise LingError("illegal value: must be 0 or 1")  
        else:  
            v = filter(lambda l: l[0] == s, self.value)  
            return Ling(value=[x[1:self.N] for x in v])  
  
    def join(self,L):  
        v = ["0" + l for l in self.value] + ["1" + l for l in L.value]  
        return Ling(v)  
  
    def dual(self):  
        if (self.value == set()) or ("" in self.value):  
            return Ling(self.value)  
        else:  
            E = self.split('0'); D = self.split('1'); D.add(E)  
            EE = E.dual() ; DD = D.dual()  
            return EE.join(DD)
```

# 2

## Satisfação Proposicional (SAT)

### 2.1 O Problema SAT

#### 2.1.1 Definições e algoritmos de conversão

Qualquer fórmula proposicional com  $n$  variáveis  $P$  pode ser descrita por uma função booleana  $f : (\mathbb{F}_2)^n \rightarrow \mathbb{F}_2$  que verifica

$$\langle x_1, \dots, x_n \rangle \models P \Leftrightarrow f(x_1, \dots, x_n) = 1 \quad (2.1)$$

para todo  $\langle x_1, \dots, x_n \rangle \in (\mathbb{F}_2)^n$ .

---

#### § 2.1 Conversão entre conetivas proposicionais e operadores algébricos em $\mathbb{F}_q$ .

Assume-se que  $a, b, c$  são fórmulas proposicionais genéricas representadas, respetivamente, pelas funções booleanas  $a', b', c'$ .

- (i) A função  $f$  relacionada com  $P$  via (2.1), obtém-se de  $P$  aplicando recursivamente as seguintes regras de substituição.

$$\begin{aligned} \text{true} &\rightsquigarrow 1 & \text{false} &\rightsquigarrow 0 \\ \neg a &\rightsquigarrow 1 + a' & a \rightarrow b &\rightsquigarrow 1 + a' + a' \times b' \\ a \vee b &\rightsquigarrow a' + b' + a' \times b' & a \wedge b &\rightsquigarrow a' \times b' \end{aligned} \quad (2.2)$$

- (ii) Inversamente  $P$  obtém-se de  $f$  usando as regras

$$\begin{aligned} a' \times b' &\rightsquigarrow a \wedge b & 1 &\rightsquigarrow \text{true} & 0 &\rightsquigarrow \text{false} \\ a' + b' &\rightsquigarrow (a \vee b) \wedge (\neg a \vee \neg b) & \text{ou} & & a' + b' &\rightsquigarrow (\neg a \wedge b) \vee (a \wedge \neg b) \end{aligned} \quad (2.3)$$

- (iii) A *conetiva* de Shannon  $c \rightarrow a ; b \stackrel{\text{def}}{=} (c \rightarrow a) \wedge (\neg c \rightarrow b)$  é uma conetiva ternária muito usada em SAT. Formas equivalentes desta conetiva são

$$c \rightarrow a ; b \equiv (\neg c \vee a) \wedge (c \vee b) \equiv (c \wedge a) \vee (\neg c \wedge b)$$

Em termos de representação como função booleana tem-se

$$\begin{aligned} c \rightarrow a ; b &\rightsquigarrow a' \times c' + b' \times (1 + c') & \text{ou} \\ c \rightarrow a ; b &\rightsquigarrow b' + (a' + b') \times c' \end{aligned} \quad (2.4)$$


---

**EXEMPLO 2.1:** Exemplos de conversão entre fórmulas proposicionais e funções booleanas:

- $(x_1 \vee x_2 \rightarrow x_3) \wedge (x_2 \rightarrow x_4) \equiv (x_1 \rightarrow x_3) \wedge (x_2 \rightarrow x_3) \wedge (x_2 \rightarrow x_4)$  converte para  $(1 + x_1 + x_1 x_3)(1 + x_2 + x_2 x_3)(1 + x_2 + x_2 x_4)$
- A função  $f = 1 + x_1 x_2 x_3 + x_2 x_4$  reescreve-se em

$$\begin{aligned} & \neg(x_1 \wedge x_2 \wedge x_3 + x_2 \wedge x_4) \rightsquigarrow \\ & \neg((x_1 \wedge x_2 \wedge x_3 \vee x_2 \wedge x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_2 \vee \neg x_4)) \\ & \rightsquigarrow \neg(x_1 \wedge x_2 \wedge x_3) \wedge \neg(x_2 \wedge x_4) \vee x_1 \wedge x_2 \wedge x_3 \wedge x_4 \end{aligned}$$

□

Normalmente os algoritmos que manipulam as fórmulas proposicionais usam determinadas estruturas algébricas que pressupõem uma determinada representação para as fórmulas. Do mesmo modo que as funções booleanas são representadas por conjuntos de *strings* de bits que representam tanto o espetro como o suporte, também as fórmulas proposicionais assumem formatos particulares.

Para analisar e classificar tais formatos vamos introduzir os elementos constituintes básicos das proposições:

**Variável**, ou *símbolo*, é o elemento atómico das fórmulas proposicionais.

**Modelo** um vetor  $m \in \{0, 1\}^n$ , tal que  $m \models x_i$  sse  $m_i = 1$  para a variável  $x_i$ .

**Literal** uma variável  $l = x$  ou a sua negação  $l = \neg x$ .

**Cláusula disjuntiva**, disjunção de literais,  $C = l_1 \vee \dots \vee l_m$

**Cláusula conjuntiva**, conjunção de literais,  $C = l_1 \wedge \dots \wedge l_m$

**Forma Normal Conjuntiva (CNF)**, conjunção de cláusulas disjuntivas.

**Forma Normal Disjuntiva (DNF)**, disjunção de cláusulas conjuntivas.

**Grafo de Shannon (SG)** as constantes `true` ou `false` ou então uma fórmula  $x \rightarrow S ; R$  em que  $x$  é uma variável e  $R$  e  $S$  são grafos de Shannon que não contêm  $x$ .

**2.1 PROPOSIÇÃO** Toda a proposição  $P$  tem uma representação única como CNF e uma representação única como DNF que lhe são semanticamente equivalentes.

**2.2 PROPOSIÇÃO** Toda a fórmula proposicional  $P$  tem uma única representação como GS que lhe é semanticamente equivalente.

A demonstração destas proposições realiza-se construtivamente apresentando algoritmos que efetuam as conversões em causa usando representações apropriadas para as três formas normalizadas referidas.

Em termos gerais, cláusulas são representadas por conjuntos de literais e as formas normais conjuntivas e disjuntivas são representadas por conjuntos de cláusulas. Grafos de Shannon são, obviamente, representados por grafos.

---

### § 2.2 Operações básicas em formas normais.

Formalmente, os domínios CNF e DNF têm como representação  $2^{2^L}$  sendo  $L$  o domínio dos literais. Em ambos os domínios define-se três operações binárias:

- (i) Se  $A$  e  $B$  são formas normais (conjuntivas ou disjuntivas) a união de conjuntos  $A \cup B$  é também uma forma normal na mesma classe.

Em CNF a união de conjuntos  $A \cup B$  representa a conjunção  $A \wedge B$ ; em DNF a união  $A \cup B$  representa a disjunção  $A \vee B$ .

- (ii) O produto  $A \times B$  define-se como

$$A \times B \stackrel{\text{def}}{=} \bigcup_{a \in A, b \in B} a \cup b$$

Em CNF o produto  $A \times B$  representa a disjunção  $A \vee B$ ; em DNF o produto  $A \times B$  representa a conjunção  $A \wedge B$ .

- (iii) A CNF  $\bar{A}$  representa a negação da forma normal  $A$ .

Para cada cláusula  $a \in A$ , sendo  $a = (l_1 \vee \dots \vee l_k)$ , tem-se  $\neg a = (\neg l_1) \wedge \dots \wedge (\neg l_k)$ ; seja  $\bar{a}$  a CNF que se obtém convertendo cada literal  $l$  numa cláusula singular  $\{\neg l\}$  e agregando estas cláusulas singulares na única CNF  $\bar{a}$ ; isto é

$$\bar{a} = \{ \{\neg l\} \mid l \in a \}$$

Sendo  $A = \bigwedge_{a \in A} a$ , será  $\neg A = \bigvee_{a \in A} \bar{a}$ ; portanto

$$\bar{A} = \times_{a \in A} \bar{a}$$


---

Com estas operações básicas é possível construir as formas normais equivalentes a uma proposição arbitrária.

---

### § 2.3 Conversão de uma forma proposicional para CNF

A conversão  $\text{Prop} \rightarrow \text{CNF}$  usa recursivamente as seguintes regras de substituição. A hipótese de indução é que as proposições  $P$  e  $Q$  são representadas, respetivamente, pelas CNF's  $A$  e  $B$ ;  $l$  designa um literal arbitrário.

$$\begin{array}{rclcrclcrcl} \text{true} & \rightsquigarrow & \{ \} & \text{false} & \rightsquigarrow & \{ \{ \} \} & l & \rightsquigarrow & \{ \{ l \} \} \\ P \wedge Q & \rightsquigarrow & A \cup B & P \vee Q & \rightsquigarrow & A \times B & \neg P & \rightsquigarrow & \overline{A} \end{array}$$


---

---

### § 2.4 Conversão de uma forma proposicional para DNF

A conversão  $\text{Prop} \rightarrow \text{DNF}$  usa recursivamente as seguintes regras de substituição. A hipótese de indução é que as proposições  $P$  e  $Q$  são representadas, respetivamente, pelas DNF's  $A$  e  $B$ ;  $l$  designa um literal arbitrário.

$$\begin{array}{rclcrclcrcl} \text{true} & \rightsquigarrow & \{ \{ \} \} & \text{false} & \rightsquigarrow & \{ \} & l & \rightsquigarrow & \{ \{ l \} \} \\ P \wedge Q & \rightsquigarrow & A \times B & P \vee Q & \rightsquigarrow & A \cup B & \neg P & \rightsquigarrow & \overline{A} \end{array}$$


---

As conversões  $\text{CNF} \rightarrow \text{GS}$  (a conversão  $\text{DNF} \rightarrow \text{GS}$  é análoga) é constituída, para proposição que não se reduzam às constantes **false** e **true**, pelos passos básicos de

---

### § 2.5 Estrutura geral de um algoritmo CNF → GS

---

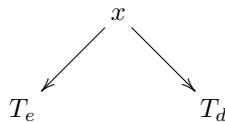
1. *Caso atómico:* Converter  $\{\}$  no nodo **true** e  $\{\{\}\}$  no nodo **false**.
  2. *Split:* dada a NF  $A$  (distinta dos dois casos anteriores) e uma variável  $x$  em  $A$ , construir duas NF's  $B$  e  $C$  tais que  $A \equiv x \rightarrow B ; C$  e  $x$  não ocorre em  $B$  nem em  $C$ .
  3. *Solve:* recursivamente construir os grafos  $B'$  e  $C'$  que representam  $B$  e  $C$ .
  4. *Join:* combinar os grafos  $B'$  e  $C'$  com a variável  $x$  para formar o grafo que representa  $x \rightarrow B ; C$ .
- 

uma algoritmo *split-and-join*.

A implementação das operações *split* e *join*, assim como a escolha da variável  $x$  em cada *split*, depende do fato da forma normal ser conjuntiva ou disjuntiva e depende também da estrutura de dados escolhida para representar o grafo.

Os grafos de Shannon são grafos orientados acíclicos que têm nodos marcados com as variáveis  $x$  ou com as constantes **true** e **false**. Nodos marcados com as constantes **true** e **false** não têm quaisquer descendentes; nodos marcados com variáveis têm um ou dois descendentes. Adicionalmente, em nenhum caminho do grafo, a mesma variável ocorre duas vezes.

Na variante mais simples cada proposição  $x \rightarrow B ; C$  é representado por uma árvore binária; as proposições **true** e **false** são representadas pelos nodos com o mesmo nome e são sempre folhas da árvore; a proposição  $x \rightarrow B ; C$  é representada pela árvore binária  $\langle x, T_e, T_d \rangle$



que tem  $x$  com nodo raíz e as árvores  $T_e$  e  $T_d$ , que representam  $B$  e  $C$  respectivamente, como sub-árvores esquerda e direita. Esta construção determina a realização da operação *join* neste algoritmo  $\text{NF} \rightarrow \text{GS}$ .

Resta analisar a implementação da operação *split*; esta implementação vai depender do fato de a forma normal ser conjuntiva ou disjuntiva e das heurísticas usadas na escolha da variável  $x$ . Para definir essas heurísticas assume-se que as variáveis estão ordenadas e a ação “escolher uma variável  $x$  em  $A$ ” é sempre realizada determinando a variável de menor ordem contida em  $A$ .

Vamos também assumir que as CNF's são representadas como conjuntos de cláusulas e as cláusulas como conjuntos de literais; deste modo, a estrutura tanto do domínio das cláusulas como do domínio das CNF's é sempre a de uma álgebra de conjuntos.

Em § 2.6 é apresentado o algoritmo básico de *split* em CNF's. A partir de uma CNF arbitrária  $A$ , o algoritmo produz duas CNF's  $B$  e  $C$  que não contêm uma variável  $x$  e verificam

$$x \wedge B \equiv x \wedge A , \quad (\neg x) \wedge C \equiv (\neg x) \wedge A$$

Deste modo será  $A \equiv x \rightarrow B ; C$  uma vez que

$$A \equiv x \wedge A \vee (\neg x) \wedge A \equiv x \wedge B \vee (\neg x) \wedge C$$

---

**§ 2.6** Operação de *split* na conversão CNF → GS.

É dada a CNF  $A$  distinta de  $\{\}$  – **true** – e de  $\{\{\}\}$  – **false**.

1. Escolhe-se a variável  $x$  de menor ordem em  $A$ .
2. Determina-se a CNF  $B$ , que verifica  $x \wedge B \equiv x \wedge A$  e não contém  $x$ ; para isso,
  - (i) remove-se de  $A$  todas as cláusulas que contêm o literal  $x$ ,
  - (ii) remove-se o literal  $\neg x$  de todas as cláusulas restantes; se neste processo se obtém uma cláusula vazia, então todas as restantes cláusulas são removidas,

$B$  é a CNF que resulta depois destas remoções.
3. Determina-se a  $C$ , que verifica  $\neg x \wedge C \equiv \neg x \wedge A$  e não contém  $x$ ; para isso,
  - (i) remove-se de  $A$  todas as cláusulas que contêm o literal  $\neg x$ ,
  - (ii) remove-se a variável  $x$  de todas as cláusulas restantes; se neste processo se obtém uma cláusula vazia, então todas as restantes cláusulas são removidas,

$C$  é a CNF que resulta depois destas remoções.

**Justificação**

Uma cláusula disjuntiva em  $A$ , que contenha um literal  $l$ , tem a forma  $(l \vee C)$ , sendo  $C$  também uma cláusula disjuntiva. Dado que  $x \wedge (x \vee C) \equiv x$  e  $x \wedge (\neg x \vee C) \equiv x \wedge C$ , em  $x \wedge A$  a cláusula  $(x \vee C)$  pode ser removida e a cláusula  $(\neg x \vee C)$  reduz-se a  $C$ .  $\square$

---

Uma vez obtida uma forma normal é necessário associar-lhe semântica. O seguinte resultado é a justificação do uso das formas normalizadas.

2.3 PROPOSIÇÃO Seja  $\mathcal{S}$  um conjunto de cláusulas,

- (i) Se  $\mathcal{S}$  é uma CNF que contém a cláusula vazia  $\{\}$ , então é incoerente. Se  $\mathcal{S}$  é uma DNF que contém a cláusula vazia  $\{\}$ , então é uma tautologia.
- (ii) Uma CNF vazia é uma tautologia; uma DNF vazia é incoerente.

**Justificação**

- (i) Um cláusula disjuntiva vazia é incoerente; uma conjunção de cláusulas que contenha uma cláusula incoerente é incoerente. De forma análoga, uma cláusula conjuntiva vazia é uma tautologia; uma disjunção de cláusulas que contenha uma tautologia é uma tautologia.
- (ii) Uma conjunção de zero cláusulas é tautologia e a disjunção de zero cláusulas é incoerente.

 $\square$ 

Para manipular os conjuntos de conjuntos de literais que representam CNF's e DNF's é necessário verificar se estes conjuntos são “mínimos”: i.e. não contêm cláusulas ou literais supérfluos. Para isso precisamos de algumas noções adicionais.

Uma cláusula diz-se *fechada* se contém simultaneamente um literal  $l$  e a sua negação  $\neg l$ . Uma cláusula diz-se *unitária* se contém um único literal.

2.4 PROPOSIÇÃO Seja  $\mathcal{S}$  um conjunto de cláusulas

- (i) Se  $\mathcal{S}$  contém uma cláusula fechada  $C$ , então é equivalente ao conjunto que se obtém removendo  $C$  de  $\mathcal{S}$ .

- (ii) Se  $\mathcal{S}$  contém uma cláusula unitária  $\{l\}$ , então  $\mathcal{S}$  é equivalente ao conjunto que se obtém removendo todas as restantes cláusulas que contêm  $l$  e removendo das restantes cláusulas todos os literais  $\neg l$ .

#### Justificação

- (i) Uma cláusula fechada disjuntiva tem a forma  $C = (l \vee \neg l \vee C')$ ; por isso é uma tautologia. Numa conjunção de cláusulas  $\mathcal{S} = C \wedge \mathcal{S}'$ , sendo  $C$  uma tautologia, é sempre equivalente a  $\mathcal{S}'$ . Dualmente uma cláusula conjuntiva fechada tem a forma  $C = (\neg l \wedge l \wedge C')$ ; portanto é incoerente. Uma disjunção de cláusulas,  $\mathcal{S} = C \vee \mathcal{S}'$ , com  $C$  incoerente, é equivalente a  $\mathcal{S}'$ .
- (ii) Tem-se  $l \wedge (l \vee C) \equiv l$  e  $l \wedge (\neg l \vee C) \equiv l \wedge C$ . Portanto, numa CNF da forma  $\mathcal{S} = l \wedge (l \vee C) \wedge \dots \wedge (\neg l \vee C') \wedge \dots$ , pode-se remover a cláusula  $(l \vee C)$  e substituir  $(\neg l \vee C')$  por  $C'$  preservando a equivalência. Em DNF's justifica-se de forma análoga.

□

O processo de simplificação determinado por (ii) na proposição 2.4 designa-se por *propagação da cláusula unitária* ou, simplesmente, *propagação unitária*.

Como consequência da proposição 2.4 é possível *simplificar* um conjunto de cláusulas; isto é, obter o menor conjunto de cláusulas equivalente ao conjunto inicial, sem conter cláusulas fechadas e propagando sucessivamente as cláusulas unitárias. O seguinte exemplo ilustra o processo de simplificação.

**EXEMPLO 2.2:** Considere-se a seguinte CNF

$$\mathcal{S} = \{\{x_1\}, \{\neg x_2, x_2\}, \{x_1, \neg x_2\}, \{\neg x_1, x_2\}, \{\neg x_2, x_3\}\}$$

e o problema de decidir se  $\mathcal{S}$  é incoerente. Vamos procurar simplificar  $\mathcal{S}$  e construir uma forma  $\mathcal{S}_s$  que é mais simples (não contém cláusulas ou literais supérfluos) e cujo teste de incoerência seja equivalente ao teste de incoerência da forma original:  $\mathcal{S}$  será incoerente se e só se  $\mathcal{S}_s$  for incoerente.

O primeiro passo é a eliminação de cláusulas fechadas; em  $\mathcal{S}$  existe a cláusula fechada  $\{\neg x_2, x_2\}$ . Eliminando-a fica

$$\{\{x_1\}, \{x_1, \neg x_2\}, \{\neg x_1, x_2\}, \{\neg x_2, x_3\}\}$$

Como a cláusula eliminada é uma tautologia, esta forma é equivalente a  $\mathcal{S}$ .

O passo seguinte é a propagação da cláusula unitária  $x_1$ ; seguindo as regras de simplificação em (ii) na proposição 2.4 obtém-se

$$\mathcal{S} \equiv \{\{x_1\}, \{x_2\}, \{\neg x_2, x_3\}\}$$

Propagando a nova cláusula unitária  $x_2$ , tem-se

$$\mathcal{S} \equiv \{\{x_1\}, \{x_2\}, \{x_3\}\}$$

Conclui-se que  $\mathcal{S}$  é satisfazível; de fato existe um único modelo  $m = 111$  que a valida. □

### 2.1.2 Resolução

A noção de resolução, introduzida por Robinson em 1965, e os algoritmos a ela associados determinam um conjunto de ferramentas muito importantes na construção de sistemas computacionais de *prova por refutação*.

O método da prova por refutação, demonstra a validade de uma inferência  $P \vdash Q$  mostrando que  $P \wedge \neg Q$  é incoerente.

A terminologia básica deste método (usando resolução) é introduzido em seguida.

### Resolução e Resolvente

Seja  $\mathcal{S}$  um conjunto de cláusulas sem cláusulas fechadas nem cláusulas singulares.

Duas cláusulas disjuntivas  $C$  e  $D$  em  $\mathcal{S}$  dizem-se *resolúveis* se existe um literal  $l$  tal que  $C = (l \vee C')$  e  $D = (\neg l \vee D')$ , sendo  $C'$  e  $D'$  cláusulas disjuntivas que não contêm  $l$  ou a sua negação  $\neg l$ .

Analogamente são resolúveis pares de cláusulas  $C = (l \wedge C')$  e  $D = (\neg l \wedge D')$  em que  $C'$  e  $D'$  são cláusulas conjuntivas que não contêm  $l$  ou  $\neg l$ .

O *resolvente* de  $(l \vee C)$  com  $(\neg l \vee D)$  é a cláusula  $(C \vee D)$ ;

O *resolvente* de  $(l \wedge C)$  e  $(\neg l \wedge D)$  é a cláusula  $(C \wedge D)$ ;

O conjunto  $\mathcal{S}$  é *fechado por resolução* quando contém a resolução de qualquer par de cláusulas resolúveis  $C, D \in \mathcal{S}$ .

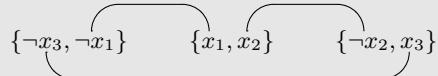
O *fecho por resolução* de  $\mathcal{S}$  é o menor conjunto de cláusulas  $\mathcal{S}^*$  que contém  $\mathcal{S}$  e é fechado por resolução.

O *grafo de resolução* de  $\mathcal{S}$  tem as cláusulas  $C \in \mathcal{S}$  como nodos e tem um arco ligando cada par de literais que define um par de cláusulas resolúveis.

**EXEMPLO 2.3:** Identificando cada cláusula pelo conjunto dos seus literais, considere-se

$$\mathcal{S} = \{ \{\neg x_3, \neg x_1\}, \{x_1, x_2\}, \{\neg x_2, x_3\} \} \quad (2.5)$$

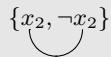
A figura seguinte apresenta o grafo de resolução deste conjunto de cláusulas.



Cada arco representa o resolvente construído removendo o arco juntamente com os literais a que está ligado. Por exemplo, removendo o arco associado a  $x_3$ , obtém-se



Note-se que os literais  $\neg x_1$  e  $\neg x_2$  movem-se para a nova cláusula arrastando os arcos a que estão ligados. Removendo o arco  $x_1$ , obtém-se um conjunto com uma única cláusula

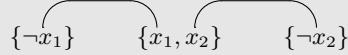


Dado que esta é uma cláusula fechada, ela pode ser removida do conjunto que se reduz ao conjunto vazio. Um tal conjunto não é incoerente.

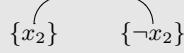
Vamos repetir este processo com um novo conjunto de cláusulas

$$\mathcal{S}' = \{ \{\neg x_1\}, \{x_1, x_2\}, \{\neg x_2\} \} \quad (2.6)$$

Construindo o grafo de resolução de  $\mathcal{S}'$  tem-se



Removendo o arco  $x_1$ , tem-se



O resolvente identificado pelo arco  $x_2$  é a cláusula vazia; por isso o conjunto reduz-se a esta cláusula vazia. Obviamente o mesmo resulta obtinha-se diretamente (sem usar resolução) por simples simplificação de  $\mathcal{S}'$ .  $\square$

Este exemplo ilustra as componentes essenciais do chamado *algoritmo de Robinson* para decidir sobre a satisfação ou incoerência de uma proposição representada por uma forma normal conjuntiva ou disjuntiva. Antes de apresentarmos o algoritmo convém apresentar a relação semântica entre um par de cláusulas resolúveis e o respectivo resolvente.

**2.5 PROPOSIÇÃO** Seja  $\mathcal{S} = \{A, B\}$  um conjunto formado por duas cláusulas resolúveis, seja  $R$  o resolvente de  $A$  e  $B$  e seja  $\mathcal{S}' = \{A, B, R\}$ . Então  $\mathcal{S}$  e  $\mathcal{S}'$  determinam proposições equivalentes.

#### Justificação

Se  $\mathcal{S}$  representa uma CNF, as cláusulas  $A$  e  $B$  são disjuntivas e, como são resolúveis, têm a forma  $A = l \vee A'$  e  $B = \neg l \vee B'$  para algum literal  $l$  e cláusulas  $A'$  e  $B'$ . O resolvente é  $R = A' \vee B'$ . Prova-se, usando dedução natural, a inferência  $A, B \vdash A' \vee B'$ .

$$\frac{\begin{array}{c} \frac{\begin{array}{c} A \\ l \vee A' \end{array}}{l \vee \neg l} \quad \frac{\begin{array}{c} B \\ \neg l \vee B' \end{array}}{\neg l} \\ \hline \frac{A'}{A' \vee B'} \quad \frac{B'}{A' \vee B'} \end{array}}{A' \vee B'}$$

Se  $\mathcal{S}$  representa uma DNF, as cláusulas  $A$  e  $B$  são conjuntivas e, como são resolúveis, têm a forma  $A = l \wedge A'$  e  $B = \neg l \wedge B'$  para algum literal  $l$  e cláusulas  $A'$  e  $B'$ . O resolvente é  $R = A' \wedge B'$ . Prova-se, usando dedução natural, a inferência  $A', B' \vdash A \vee B$ .

$$\frac{\begin{array}{c} \frac{\begin{array}{c} l \quad A' \\ \hline A \end{array}}{A \vee B} \quad \frac{\begin{array}{c} \neg l \quad B' \\ \hline B \end{array}}{A \vee B} \\ \hline A \vee B \end{array}}{A \vee B}$$

Numa CNF temos  $\mathcal{S} \vdash R$ ; portanto  $\mathcal{S} \vdash R \wedge \mathcal{S}$  o que implica  $\mathcal{S} \equiv \mathcal{S}'$ . Numa DNF provámos que  $R \vdash \mathcal{S}$ ; portanto  $R \vee \mathcal{S} \vdash R$  o que implica  $\mathcal{S} \equiv \mathcal{S}'$ .  $\square$

Esta proposição pode ser expressa de outro modo:

a proposição representada por um conjunto de cláusulas  $\mathcal{S}$  é logicamente equivalente equivalente à proposição representada pelo seu fecho por resolução  $\mathcal{S}^*$

A proposição 2.3 diz-nos que, contendo  $\mathcal{S}^*$  a cláusula vazia, se  $\mathcal{S}$  representar uma CNF ela é incoerente e, se representar uma DNF, ela é uma tautologia. A implicação inversa é muito mais difícil de provar. Vamos apresentar aqui o resultado final

**2.1 TEOREMA (COMPLETUDE DA RESOLUÇÃO)** Se  $\mathcal{S}$  representar uma CNF então essa proposição é incoerente se e só se o fecho por resolução  $\mathcal{S}^*$  contém a cláusula vazia. Inversamente, se  $\mathcal{S}$  representar uma DNF, ela será uma tautologia se e só se o fecho por resolução  $\mathcal{S}^*$  contém a cláusula vazia.

Este teorema justifica o *algoritmo de Robinson* que, essencialmente, procura obter a cláusula vazia a partir de  $\mathcal{S}$  sem ter de calcular explicitamente todo o fecho  $\mathcal{S}^*$ . Em § 2.7 apresenta-se o esquema geral do algoritmo.

---

### § 2.7 Algoritmo de Robinson

Dado um conjunto de cláusulas  $S$  procura-se encontrar a cláusula vazia no fecho por resolução de  $S$ .

1. Construir o grafo de resolução de  $S$  da seguinte forma
    - (i) Para cada cláusula  $C \in S$  e para cada literal  $l \in C$  que não esteja ligado, procurar todas as cláusulas  $C' \neq C$  onde exista o literal  $\neg l$ .
    - (ii) Estabelecer uma ligação entre o literal  $l$  em  $C$  e o literal  $\neg l$  em  $C'$
  2. Para cada arco  $a$  no grafo:
    - (i) Substituir as duas cláusulas ligadas por  $a$  pelo respetivo resolvente ligando cada um dos literais do resolvente aos arcos que tinham nas duas cláusulas iniciais.
    - (ii) Se for construída uma cláusula fechada remover essa cláusula de  $S$  juntamente com todos os arcos ligados aos seus literais.
    - (iii) Se for construída a cláusula vazia, terminar o algoritmo.
- 

A aplicação principal do algoritmo de Robinson é a *prova por refutação* de uma inferência  $A \vdash B$ . Em termos gerais o método consiste em escrever  $A$  como uma CNF e  $B$  como uma DNF; desta forma  $\neg B$  tem uma simples representação como CNF. Em seguida, usando resolução, verifica-se se  $A \wedge \neg B$  é ou não satisfazível.

### 2.1.3 Segurança e Completude de Algoritmos

Outra aplicação da resolução de Robinson é a prova da equivalência  $A \equiv B$  em que  $A$  e  $B$  são duas CNF's. Formalmente pretende-se construir um algoritmo que *aceite* uma prova da inferência  $\vdash A \equiv B$ .

Essencialmente tem de se aceitar ambas as inferências  $A \vdash B$  e  $B \vdash A$  tendo em atenção que, sendo  $A$  e  $B$  CNF's, é “hard” calcular  $\neg A$  e  $\neg B$  como CNF's.

No limite é esse o processo que tem de ser seguido: usar prova por refutação de  $A \vdash B$  e  $B \vdash A$ . Porém ocorre com frequência que, como conjuntos de cláusulas,  $A$  e  $B$  têm muitos elementos comuns e só diferem num pequeno conjunto de cláusulas.

Suponhamos que  $A$  e  $B$  são CNF's que têm um conjunto de cláusulas comuns  $C$ ; pode-se escrever  $A = C \wedge A'$  e  $B = C \wedge B'$  sendo  $A'$  e  $B'$  conjuntos com menos cláusulas do que  $C$ . O teste de equivalência limita-se a provar as inferências  $C \wedge A' \vdash B'$  e  $C \wedge B' \vdash A'$ ; para isso basta provar que são incoerentes as formas  $A \wedge \neg b$  e  $B \wedge \neg a$ , para todo  $b \in B'$  e todo  $a \in A'$ .

Vamos analisar como provar a incoerência de uma destas formas  $A \wedge b$ .

Atendendo que  $b$  é uma cláusula disjuntiva, será  $b = l_1 \vee \dots \vee l_k$  para determinados literais  $l_i$ ; portanto  $A \wedge \neg b = A \wedge \neg l_1 \wedge \dots \wedge \neg l_k$ . Esta CNF tem, pelo menos,  $k$  cláusulas unárias  $\{l_i\}$ ; portanto pode ser simplificada por propagação unária.

Da simplificação de  $A$  por propagação dos diversos  $l_i$  resulta uma nova CNF que vamos representar por  $A_0$ ; nesta CNF já não ocorre nenhuma das variáveis nos  $l_i$ ; e, por isso,  $A_0$  pode ter muito menos variáveis do que  $A$ . Como  $A \wedge \neg b$  é equivalente a  $A_0 \wedge \neg b$ , e  $A_0$  não contém variáveis em  $b$ , temos de concluir que  $A \wedge \neg b$  é incoerente se e só se  $A_0$  é incoerente.

Pode-se sumariar estas considerações no seguinte esquema de algoritmo

---

### § 2.8 Aceitar a contradição $A \wedge \neg b$ .

$A$  é uma CNF e  $b = l_1 \vee \dots \vee l_k$  é uma cláusula disjuntiva.

**accept**( $A \wedge \neg b \equiv \perp$ )

1. Simplificar  $A$  com propagação unária dos literais  $\neg l_1, \dots, \neg l_k$ . Seja  $A_0$  a cláusula simplificada resultante.
2. Se  $A_0$  contém a cláusula vazia terminar com “output” **true**; senão ...
3. Usar um algoritmo alternativo **accept**( $A_0 \equiv \perp$ ) e terminar com o respetivo “output”.

Para o “algoritmo alternativo” mencionado no passo 3, temos duas hipóteses básicas:

- (a) Usar a resolução completa que é dada pelo algoritmo de Robinson.
- (b) Numa abordagem *model checking* usar uma de duas estratégias:
  - (i) Verificar se, de entre todos os modelos  $m$ , nenhum valida  $A_0$ .
  - (ii) Verificar se, de entre alguns dos modelos  $m$  escolhidos criteriosamente, nenhum valida  $A_0$ .

Genericamente, para verificar a validade de uma qualquer asserção  $\vdash P$ , usa-se o tipo de algoritmos probabilísticos, designados por **accept/reject**, que recebem como argumento uma proposição  $P$  e terminam com um “output” booleano.

Um processo **accept** é  $\rho$ -correto, ou *seguro*, quando, se  $\vdash \neg P$  é válido, termina em **false** com probabilidade não inferior a  $\rho$ . Se  $\vdash P$  for válido a correcção não impõe qualquer “output”; à falta de melhor informação pode-se assumir que **true** e **false** são, neste caso, “outputs” com igual probabilidade.

O algoritmo **accept** é  $\rho$ -completo quando, se  $\vdash P$  válido, termina com “output” **true** com probabilidade não inferior a  $\rho$ . Se  $\vdash \neg P$  for válido, à falta de melhor informação, assume-se que o algoritmo terminar em **true** ou **false** com igual probabilidade.

Quando  $\rho = 1$  diz-se simplesmente que o algoritmo é *correto (seguro)* ou *completo* consoante o caso.

Quando  $\text{accept}(P)$  é um algoritmo *seguro* e termina com “output” **true**, um observador tem confiança absoluta na existência de uma inferência  $\vdash P$ . Se o “output” for **false**,  $P$  tanto pode ser uma tautologia como não. Se o algoritmo for  $\rho$ -*seguro* e terminar em **true**, o observador tem confiança não inferior a  $\rho$  na validade de  $\vdash P$ .

Inversamente, se o algoritmo for  $\rho$ -*completo* e terminar em **false**, o observador tem um grau de confiança não inferior a  $\rho$  na validade de  $\vdash \neg P$ . Se o “output” for **true** o observador nada pode garantir:  $P$  pode ser válido ou não.

$\text{reject}(P)$  é definido como  $\text{accept}(\neg P)$ .

Desta forma  $\text{reject}(P)$  é seguro se e só se  $\text{accept}(P)$  é completo e, inversamente,  $\text{reject}(P)$  é completo se e só se  $\text{accept}(P)$  é seguro.

**EXEMPLO 2.4:** Considere-se uma algoritmo  $\text{accept}(P)$  que, independentemente de  $P$ , termina imediatamente com a mensagem **false**. Um tal algoritmo é trivialmente correto; obviamente está longe de ser completo.

Da mesma forma um algoritmo  $\text{accept}(P)$  que se limita a terminar com a mensagem **true** é completo mas está longe de ser correto.  $\square$

Obviamente nenhum dos algoritmos no exemplo 2.4 é aceitável porque ambos têm uma tendência muito marcada: têm sempre o mesmo “output” independentemente do “input”  $P$ . Essencialmente esta tendência diz-nos que o algoritmo não dá qualquer informação sobre a validade de um  $P$  específico. Para construir algoritmos probabilísticos sem tendência temos de recorrer a outro tipo de abordagem.

A completude da lógica proposicional diz-nos que  $A$  implica  $\perp$  (i.e.  $A$  é uma contradição) se e só se nenhum modelo a valida:  $A$  é não satisfazível.

$$\vdash (A \rightarrow \perp) \quad \text{sse} \quad \forall m \in \{0, 1\}^\kappa \cdot m \not\models A \quad (2.7)$$

equivalentemente

$$\vdash (A \rightarrow \perp) \quad \text{sse} \quad A^{\{0, 1\}^\kappa} = 0$$

A forma mais direta de implementar esta relação é a do algoritmo 2.1.

```
fun accept (A → ⊥) is
    for every m ∈ {0, 1}κ do
        if m ⊨ A then return false
    return true
```

**Algorithm 2.1:** Validação exaustiva de  $A \rightarrow \perp$ .

O espaço de modelos  $\{0, 1\}^\kappa$  tem cardinalidade  $2^\kappa$ ; se o número de variáveis for razoavelmente pequeno, é viável percorrer todo este espaço e testar, para cada modelo, a validade de  $A$ ; obviamente, se o número de variáveis for superior a algumas (poucas!) dezenas, já não é viável esta forma de procura exaustiva.

Tendo a desvantagem da complexidade, o algoritmo 2.1 tem a vantagem de ser 100% seguro e 100% completo. É seguro porque, sempre que  $A$  é satisfazível, o algoritmo encontra um modelo  $m$  que a satisfaz e dá a resposta `false`. É completo porque, sempre que  $A$  não é satisfazível, o ciclo **for every** percorre todos os modelos  $m$  sem que  $m \models A$  se verifique e, portanto, o algoritmo termina com a resposta `true`.

Para melhorar no aspetto da complexidade pode-se considerar um algoritmo onde a procura não seja exaustiva: em vez de percorrer todos os modelos, o algoritmo percorre apenas alguns modelos gerados aleatoriamente.

```
fun accept ( $A \rightarrow \perp$ ) is
    do  $N$  times
         $m \leftarrow \{0, 1\}^\kappa$  --  $m$  é gerado aleatoriamente no espaço  $\{0, 1\}^\kappa$ 
        if  $m \models A$  then return false
    return true
```

**Algorithm 2.2:** Validação probabilística de  $A \rightarrow \perp$ .

A complexidade é melhorada porque, independentemente do número de variáveis  $\kappa$ , o número de testes  $m \models A$  está limitado pelo parâmetro  $N$ . Em contrapartida o algoritmo já não é 100% seguro uma vez que, se  $A$  for satisfazível, ele pode não encontrar um modelo que a satisfaça. É porém, 100% completo uma vez que se  $A$  não for satisfazível ele termina sempre com a mensagem `true`.

Considere-se de novo a questão da segurança. Vamos supor que  $A$  é satisfazível e que se conhece um limite inferior à probabilidade de se verificar  $m \models A$  quando  $m$  é gerado aleatoriamente; isto é conhecido um  $\varepsilon$  que verifique

$$\Pr[m \models A | m \leftarrow \{0, 1\}^\kappa] \geq \varepsilon \quad (2.8)$$

Equivalentemente tem-se  $\Pr[m \not\models A | m \leftarrow \{0, 1\}^\kappa] \leq 1 - \varepsilon$ . Como a probabilidade de o algoritmo terminar em `true` é igual à probabilidade de, em  $N$  tentativas, se verificar sempre  $m \not\models A$ , será  $\Pr[\text{accep}(A \rightarrow \perp) = \text{true}] \leq (1 - \varepsilon)^N$ ; portanto

$$\Pr[\text{accep}(A \rightarrow \perp) = \text{false}] \geq 1 - (1 - \varepsilon)^N \quad (2.9)$$

O parâmetro  $\rho = 1 - (1 - \varepsilon)^N$  é, por isso, a segurança do algoritmo. Quando  $\varepsilon$  é muito pequeno tem-se  $\rho \sim \varepsilon N$ ; neste caso é preciso um  $N$  bastante grande para se ter algum nível de segurança. Quando  $\varepsilon \sim 1/2$  então  $\rho \sim 1 - 2^{-N}$  e basta um  $N$  razoavelmente pequeno para garantir um elevado nível de segurança.

#### 2.1.4 Algoritmos de conversão com memória global

Considere-se de novo o algoritmo **CNF → GS** apresentado em § 2.5 com a implementação dos grafos através de árvores binárias. Após o passo *split*, do qual resultam duas CNF's  $B$  e  $C$ , o passo *solve* constrói duas árvores que representam estas duas proposições; no passo *join* estas árvores são combinadas com a variável  $x$  para construir uma nova árvore.

Esta estrutura força a que sejam necessárias duas chamadas recursivas em cada *solve*; daí resulta que o número total de chamadas é, no pior caso, exponencial com o número de variáveis. Porém, se em *split* chegarmos à conclusão que  $B$  e  $C$  são equivalentes, basta uma única chamada recursiva no passo *solve*. Daí resulta

uma melhoria na eficiência do algoritmo já que, sempre que o *split* produz fórmulas iguais e em relação ao pior caso, o número de chamadas pode ser reduzida a metade.

Também, se for  $A = x \rightarrow B ; B$ , tem-se  $A = B$  uma vez que  $x \wedge B \vee (\neg x) \wedge B \equiv B$ . Por isso se, após o *split*, for possível concluir que  $B \equiv C$ , então o grafo que representa  $B$  é o mesmo que representa  $A$ .

Para considerar estas otimizações já as árvores binárias não são uma boa representação dos grafos de Shannon. Neste caso é conveniente um grafo que permita ter construções da seguinte forma



Uma boa estratégia, para implementação estes grafos, usa um *dicionário persistente*.

Um “*dicionário*” é uma associação funcional *chaves*  $\rightarrow$  *valores*; as chaves são pequenas “strings” de bits de tamanho fixo; os “valores” são os itens de informação que se pretende gerir; cada chave está associada, quanto muito, a um valor.

São duas as ações básicas sobre dicionários.

- **get** recebe uma chave como argumento e devolve o valor associado à chave, caso a associação exista; em caso contrário devolve um “erro”,
- **add** recebe como argumento um par (*chave, valor*) e acrescenta-a ao dicionário; caso já exista outra associação com a mesma chave, ela é previamente removida do dicionário.

Neste algoritmo as chaves obtêm-se aplicando uma função de *hash* à representação das CNF’s. Uma função de *hash* recebe como argumento uma “string” de bits de tamanho arbitrário e devolve uma “string” de bits de tamanho fixo\*.

A característica essencial destas funções é a muito baixa probabilidade de produzir o mesmo “output” para dois “inputs” distintos; desta forma o *hash* de uma proposição pode ser interpretado como uma “impressão digital” de poucos bits que, com elevada probabilidade, representa unicamente essa fórmula.

A representação de um *split*  $A \equiv x \rightarrow B ; C$ , quando  $B \not\equiv C$ , vai ser uma associação que tem, como chave, a “string”  $\text{hash}(A)$  e, como valor, o triplô  $\langle x, \text{hash}(B), \text{hash}(C) \rangle$  que representa um grafo de Shannon associado a  $A$ . Quando  $B \equiv C$ , a chave  $\text{hash}(A)$  vai ser associada ao mesmo valor que está associada a chave  $\text{hash}(B)$ .

Em qualquer dos casos o número de bits necessários à representação do valor associado a  $A$  é sempre limitado; essencialmente é duas vezes o tamanho dos *hash*’s mais o número de bits necessários à representação das variáveis  $x$  (mesmo um milhão de variáveis necessitam, quanto muito, 20 bits).

Seguindo esta representação, pode-se construir um algoritmo recursivo que usa um dicionário como “memória persistente”; isto é, memória que persiste ao longo das várias invocações do algoritmo e, durante cada invocação, pode ser escrita e lida.

No passo 2 aparece referência a “variáveis livres”; estas variáveis são determinadas pelas eventuais cláusulas unitárias de  $A$ : uma variável diz-se *livre em A* se ocorre em  $A$  mas não ocorre em nenhuma das suas eventuais cláusulas unárias.

\*Exemplos de funções de *hash* são o `md5`, para outputs de 128 bits, o `sha1`, para “outputs” de 160 bits e o `sha3` para “outputs” de 256 bits.

---

**§ 2.9** Conversão CNF → GS com dicionário auxiliar persistente.
 

---

O dicionário é inicializado com as representação das constantes `true` e `false` e é atualizado pelas várias chamadas ao algoritmo. Cada chamada recebe uma CNF  $A$  como parâmetro e atualiza o dicionário com uma associação entre  $A$  e um grafo de Shannon que lhe seja equivalente.

**CNFtoGS ( $A$ ) is**

1. Calcular `hash( $A$ )` e verificar se esta chave já existe no dicionário; se tal ocorrer terminar; senão, continuar no passo seguinte.
  2. Selecionar  $x$  como a menor de entre as *variáveis livres* de  $A$ ,
  3. Calcular, usando § 2.6, o *split* ( $x \rightarrow B ; C$ ) de  $A$ ; apresentar  $B$  e  $C$  na forma simplificada que resulta da propagação das suas cláusulas unárias.
  4. Invocar **CNFtoGS** com argumento  $B$ .
  5. Se  $B \equiv C$ , executar `add(hash( $A$ ), get(hash( $B$ )))` e terminar; senão, continuar para o passo seguinte.
  6. Invocar **CNFtoGS** com argumento  $C$ .
  7. Executar `add(hash( $A$ ), ⟨ $x$ , hash( $B$ ), hash( $C$ )⟩)`.
- 

Assume-se que  $A$  já foi simplificado por propagação unária e, portanto, as variáveis que aparecem em cláusulas unárias não ocorrem em nenhuma outra cláusula de  $A$ . Desta forma, para que  $A$  seja satisfazível o valor de uma variável numa cláusula unária está completamente determinado:  $A$  contendo a cláusula for  $\{x_i\}$  só pode ser satisfazível se for  $x_i = 1$ , e contendo  $\{\neg x_j\}$  só é satisfazível se for  $x_j = 0$ .

Assim as variáveis que ocorrem em cláusulas unárias estão *presa*s a um valor bem determinado; por oposição, as restantes variáveis de  $A$  estão *livres*.

Essenciais à complexidade deste algoritmo é a comparação de `hash( $A$ )` com as chaves do dicionário no 1º passo, e o teste  $B \equiv C$  executado no passo 5. Ambos estão relacionados com a probabilidade de se verificar

$$B \equiv C \implies \text{hash}(B) = \text{hash}(C) \quad (2.10)$$

No 1º passo, procura-se verificar se a proposição  $A$  já existe no dicionário uma vez que, se tal acontecer, o algoritmo simplesmente termina. Porém  $A$  pode estar escrita de uma forma tal que `hash( $A$ )` não coincida com nenhuma chave do dicionário. Basta, por exemplo, que as cláusulas estejam apresentadas de forma diferente. No passo 5 temos algo semelhante: as formas  $B$  e  $C$  podem ser logicamente equivalentes mas terem *hash's* muito diferentes.

**□1:** Suponhamos, por exemplo, que se tem

$$B = (\neg l_1 \vee l_2) \wedge (\neg l_2 \vee l_3) \wedge (\neg l_1 \vee l_3) \quad , \quad C = (\neg l_1 \vee l_2) \wedge (\neg l_2 \vee l_3)$$

com  $l_1, l_2, l_3$  literais. Neste caso  $B \equiv C$  é válido mas é muito provável (dependendo da função de *hash*) que não se verifique `hash( $B$ ) = hash( $C$ )`. □

Desta forma é necessário estabelecer uma forma *standard* de escrever conjuntos de cláusulas na forma de sequências de sequências. Por exemplo pode-se escrever  $A$  pondo primeiro a sequência das suas cláusulas unárias pela ordem dos índices das

variáveis, seguem-se as cláusulas com 2 literais, depois as com 3 literais, etc. Dentro das cláusulas com o mesmo número de literais usa-se a ordem das variáveis para decidir a ordem das cláusulas.

O método de ordenação de cláusulas não é crítico mas, o que é crítico, é que, para um dado um conjunto de cláusulas, exista uma forma standard única de representar este conjunto como uma sequência de sequências de literais à qual se possa aplicar, sem ambiguidades, a função de `hash`. Assim pode-se garantir que, independentemente da forma como representamos um conjunto de cláusulas  $A$ ,  $\text{hash}(A)$  é bem determinado. Nomeadamente, se tivermos  $\text{hash}(A) \neq \text{hash}(B)$ , pode-se afirmar que, com grande probabilidade,  $A$  e  $B$  são conjuntos de cláusulas distintos.

O passo 5 deve conter um algoritmo que teste a validade da asserção  $B \equiv C$ . Este algoritmo não deve ter complexidade computacional elevada porque, se tal ocorrer, anulam-se os os ganhos de eficiência que se obtêm pelo fato de existir apenas uma chamada recursiva e não duas.

No algoritmo de conversão em § 2.9, a verificação de que  $B$  e  $C$  são equivalentes é realizado por um teste probabilístico da forma `accept`( $B \equiv C$ ). Um algoritmo `accept` que, neste caso, seja determinístico, correto e completo seria uma implementação ideal do teste  $B \stackrel{?}{\equiv} C$ . Sabe-se que tal algoritmo existe: basta usar prova por refutação usando o algoritmo de resolução.

O uso da resolução implica, normalmente, uma complexidade computacional que anula a vantagem de se ter que fazer apenas uma chamada recursiva quando o teste `accept`( $B \equiv C$ ) tem sucesso.

Assim é razoável pensar-se em usar um algoritmo `accept` que seja correto e tão completo quanto possível; formalmente `accept` é  $\rho$ -completo com  $\rho < 1$  e tão grande quanto possível; naturalmente, quanto maior for  $\rho$ , mais complexo será `accept`.

**EXEMPLO 2.5:** Para testar  $B \stackrel{?}{\equiv} C$  pode-se escrever um algoritmo que responde com o valor lógico do teste  $\text{hash}(A) \stackrel{?}{=} \text{hash}(B)$ . Tal algoritmo será muito provavelmente correto mas não completo.

Pode-se melhorar o algoritmo calculando, quando  $\text{hash}(A) \neq \text{hash}(B)$ , provas por refutação parciais de  $B \vdash C$  e  $C \vdash B$ .  $\square$

Usando um `accept` correto no passo 5 o algoritmo **CNFtoGS** estará sempre correto. Porém se `accept` não for completo, o algoritmo não é tão eficiente quanto poderia ser; isto porque, em situações onde ocorra  $B \equiv C$ , `accept` rejeita esta situação o que implica que em vez de existir uma única chamada recursiva ao argumento  $B$  existe uma chamada supérflua com argumento  $C$ .

Adicionalmente a simplificação  $(x \rightarrow B ; C) \rightsquigarrow B$  não é registada no dicionário: mesmo sendo  $B$  e  $C$  equivalentes o valor associado a  $\text{hash}(A)$  continua a ser  $\langle x, \text{hash}(B), \text{hash}(C) \rangle$ , e não o simplificado `get(hash(B))`.

Usando um algoritmo `accept`( $B \equiv C$ ) que não seja completamente seguro a conversão resultante também não é completamente segura. É o que ocorre se se usar um algoritmo de validação probabilística da forma do algoritmo 2.2 como forma de testar a incoerência de um conjunto de cláusulas.

### 2.1.5 Resolvendo o problema SAT

Considere-se a abordagem *model checking* referida na nota § 2.8 para construir um algoritmo de UNSAT,  $\text{reject}(A)$ , supondo que  $A$  é uma CNF com  $\kappa$  variáveis.

Neste problema os *modelos* identificam-se com elementos de  $\{0, 1\}^\kappa$ . O algoritmo  $\text{reject}(A)$  implementa a asserção que estabelece que  $A$  não é satisfazível.

$$\forall m \in \{0, 1\}^\kappa \cdot m \not\models A \quad (2.11)$$

Representamos por  $\llbracket A \rrbracket$  o conjunto de todos os modelos que validam  $A$ .

$$\llbracket A \rrbracket \stackrel{\text{def}}{=} \{m \in \{0, 1\}^\kappa \mid m \models A\} \quad (2.12)$$

Assim a CNF  $A$  é *insatisfazível* se e só se  $\llbracket A \rrbracket = \{\}$ ; a completude da lógica proposicional diz-nos que esta igualdade é equivalente à asserção “ $A$  é uma contradição”.

É importante ver como se pode construir  $\llbracket A \rrbracket$  para diferentes formas (literal, cláusula e CNF) de  $A$ . Por exemplo, para literais tem-se

$$\llbracket x_i \rrbracket = \{m \mid m_i = 1\}, \quad \llbracket \neg x_i \rrbracket = \{m \mid m_i = 0\} \quad (2.13)$$

No contexto das CNF's, uma cláusula conjuntiva  $v = l_1 \wedge \dots \wedge l_n$  em que os literais  $l_i$  têm variáveis distintas, designa-se por *atribuição*; obviamente tem-se  $\llbracket v \rrbracket = \bigcap_i \llbracket l_i \rrbracket$  e o facto de as variáveis serem distintas implica  $\llbracket v \rrbracket \neq \{\}$ . A atribuição com  $n = 0$  literais é representada por  $\epsilon$  e é equivalente a *true*; será  $\llbracket \epsilon \rrbracket \equiv \{0, 1\}^\kappa$ .

As variáveis contidas em  $v$  dizem-se *atribuídas*; as restantes dizem-se *livres*. A atribuição  $v$  é *completa* quando todas as variáveis estão atribuídas: i.e.  $v$  contém  $n = \kappa$  literais; neste caso  $\llbracket v \rrbracket$  é um conjunto singular  $\{m\}$  com o modelo  $m$  definido por:  $m_i = 1$  se e só se  $v \wedge \neg x_i$  é uma contradição.

Para completar a família de possíveis atribuições, vamos introduzir um *literal trivial* ou *conflito*  $\tau$  semanticamente equivalente a *false*\*; i.e.  $\llbracket \tau \rrbracket = \{\}$ . Toda a atribuição  $v$  que contenha  $\tau$  diz-se *trivial* e verifica  $\llbracket v \rrbracket = \{\}$ .

Se  $\mu$  e  $v$  são atribuições que não têm variáveis comuns (i.e. todas as variáveis atribuídas em  $\mu$  estão livres em  $v$  e vice-versa) então  $\mu \wedge v$  é outra atribuição com modelos  $\llbracket \mu \wedge v \rrbracket = \llbracket \mu \rrbracket \cap \llbracket v \rrbracket$ .

Uma atribuição  $v$  é uma *testemunha* (“witness”) de  $A$  quando se verifica  $\llbracket v \rrbracket \subseteq \llbracket A \rrbracket$ ; o conjunto das testemunhas de  $A$  é representado por  $[A]$ .

**FACTO 2.1** Uma CNF  $A$  é satisfazível se e só se tem pelo menos uma testemunha. Adicionalmente

$$\llbracket A \rrbracket \equiv \bigcup_{v \in [A]} \llbracket v \rrbracket \quad (2.14)$$

Um dos resultados mais importantes da abordagem “*model cheking*” é o designado *teorema de interpolação de Craig* que pode ser enunciado da seguinte forma.

**2.2 TEOREMA** Sejam  $A(x, y)$  e  $B(x, z)$  duas CNF's que partilham o vetor de variáveis  $x$ ; os vetores de variáveis  $y$  e  $z$  são específicos de  $A$  e  $B$  respetivamente. Então existe uma função booleana  $x \mapsto f(x)$ , designada por “*interpolante*” tal que

$$\llbracket A(x, y) \wedge B(x, z) \rrbracket = \{\} \quad \text{sse} \quad \begin{cases} \llbracket A(x, y) \rrbracket = \{\} & \text{se } f(x) = 1 \\ \llbracket B(x, z) \rrbracket = \{\} & \text{se } f(x) = 0 \end{cases}$$

\*Pode-se ver  $\tau$  como um símbolo ao qual não é possível atribuir qualquer valor lógico.

Uma forma alternativa para descrever validações e modelos usa a noção de *função de valoração* associada a qualquer fórmula  $A$  quer ela seja um literal, uma cláusula disjuntiva ou um conjunto de cláusulas disjuntivas (i.e. uma CNF).

Cada  $A$  determina uma função booleana, a sua *função valoração*,  $m \mapsto A^{\{m\}}$  definida pelas asserções

$$A^{\{m\}} = 1 \quad \text{sse} \quad m \models A \quad \text{sse} \quad m \in \llbracket A \rrbracket \quad (2.15)$$

Da definição conclui-se imediatamente

$$\begin{cases} l^{\{m\}} = m_i & \text{se } l \text{ é o literal } x_i \\ l^{\{m\}} = 1 - m_i & \text{se } l \text{ é o literal } \neg x_i \\ a^{\{m\}} = \max_{l \in a} l^{\{m\}} & \text{se } a \text{ é uma cláusula} \\ A^{\{m\}} = \min_{a \in A} a^{\{m\}} & \text{se } A \text{ é uma CNF} \end{cases} \quad (2.16)$$

A noção de valoração estende-se para conjuntos de modelos

$$A^{\{m_1, \dots, m_k\}} \stackrel{\text{def}}{=} \max_i A^{\{m_i\}} \quad (2.17)$$

Vamos supor que um conjunto de cláusulas  $A$  é escrito sob a forma de uma sequência ordenada de cláusulas onde ocorrem, em primeiro lugar, as cláusulas singulares ordenadas por um critério determinado; e.g. pela ordem das variáveis que contêm.

Pode-se sempre escrever  $A = v \wedge B$  em que  $v = l_1 \wedge \dots \wedge l_n$  é uma atribuição e  $B$  não contém cláusulas não-singulares. As variáveis de  $A$  que ocorrem em  $v$  dizem-se *atribuídas*; as restantes dizem-se *livres*. Se  $B$  só contém variáveis livres em  $A$ , a forma  $A = v \wedge B$  está *reduzida por propagações unitárias* (UPR “unit propagation reduced”).

**FACTO 2.2** Se  $A = v \wedge B$  está reduzida por propagações unárias então

- (i)  $\llbracket A \rrbracket = \{ \}$  se e só se  $\llbracket B \rrbracket = \{ \}$ .
- (ii)  $[A] \equiv \{ v \wedge \mu \mid \mu \in [B] \}$ .

Vamos analisar uma técnica muito eficaz de acelerar os algoritmos de **sat** designado por *aprendizagem de cláusulas* (“clause learning” ou CL).

Considere-se uma atribuição  $v = l_1 \wedge \dots \wedge l_n$  e vamos supor que se consegue concluir que  $v \wedge A$  é uma contradição; nomeadamente, vamos assumir que, por propagação unária, se atinge uma CNF que contém a cláusula vazia.

Como a atribuição  $v$  é uma cláusula conjuntiva, a sua negação  $\neg v = \neg l_1 \vee \dots \vee \neg l_n$  é uma cláusula disjuntiva; deste modo  $A \wedge \neg v$  continua a ser uma CNF. Por outro lado, como  $A \wedge v \vdash \perp$ , tem-se  $A \vdash \neg v$ . Em conclusão,  $A$  e  $(A \wedge \neg v)$  são CNF’s equivalentes.

Esta abordagem serve para “aprender cláusulas através de conflitos”. O princípio é simples e usa o seguinte esquema abstrato para algoritmos que “tentam” uma atribuição  $v$ , no contexto de um conjunto de cláusulas  $A$ .

1. Calcula-se a propagação unitária de  $v \wedge A$  de modo a obter uma forma reduzida  $\mu \wedge A'$ .

2. Se  $\mu$  é uma atribuição completa ou  $B$  é o conjunto vazio de cláusulas, então o algoritmo termina com a mensagem **sat** e com a indicação de que  $\mu$  é uma atribuição que valida  $A$ .
3. Se  $B$  contém a cláusula vazia então obtém-se um conflito. Como consequência
  - (i)  $\neg v$  é a nova cláusula, “aprendida com o conflito”, que é adicionada a  $A$ ,
  - (ii) a atribuição  $v$  é completamente abandonada (*backtrack*) e o algoritmo regressa ao início mantendo o novo conjunto de cláusulas;
  - (iii) se não existir um novo  $v$  que seja possível tentar o algoritmo termina com a mensagem **unsat**; se existir tal  $v$ , ele é escolhido e regressa-se ao princípio.
4. Se  $B$  não verifica nenhuma das situações anteriores nenhuma dos casos anteriores, então  $\mu$  é aumentado com um novo literal e regressa-se ao início.

Resta definir uma estratégia para lidar com o “backtrack” e, genericamente, com o mecanismo de escolha de atribuições.

Em primeiro lugar, o algoritmo pode ser iniciado com a atribuição vazia  $\epsilon$ . Para lidar com estes aspetos de gestão vamos manipular as atribuições como um “stack”.

No início é escolhida uma qualquer variável livre  $x$  e faz-se o **push** das atribuições  $x$  e  $\neg x$ . O passo 1 e seguintes recolhe  $v$  do topo do “stack”.

A operação de “backtrack” é efetuada com um **pop** no “stack”; testar se existe um novo  $v$  disponível é equivalente a testar se o “stack” está vazio.

No passo 4 escolhe-se uma nova variável livre  $x$ , faz-se o **pop** do “stack” (elimina-se  $v$ ) e faz-se o **push** de  $\mu \wedge x$  e de  $\mu \wedge \neg x$ .

O algoritmo 2.3 ilustra a estrutura de um algoritmo construído segundo estes princípios.

□

Para visualizar o funcionamento deste esquema é conveniente usar a noção de grafo de implicação.

**2.1 DEFINIÇÃO** *O grafo de implicação para a CNF  $A$  é um grafo acíclico que tem nodos marcados com atribuições e ramos  $v \rightarrow \mu$  quando:*

- (i)  $\mu = v \wedge x$  ou  $\mu = v \wedge \neg x$ , sendo  $x$  uma variável que não ocorre em  $v$ .
- (ii)  $\mu \equiv \tau$  e  $v \wedge A$  simplifica, por propagação unária, numa CNF  $B$  que contém a cláusula vazia.
- (iii)  $v \wedge A$  simplifica, por propagação unária, numa forma  $\mu \wedge B$  reduzida por propagações unárias.

**EXEMPLO 2.6:** Considere-se a seguinte CNF a 8 variáveis

$$\begin{aligned} A = & (x_1 \vee \neg x_2 \vee x_8) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge \dots \\ & \dots \wedge (\neg x_4 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_6 \vee x_7) \wedge (x_5 \vee x_6) \end{aligned}$$

O algoritmo usa duas estruturas globais:

- stack** – é um “stack” que gera as várias atribuições
- prob** – é a estrutura de dados que armazena a CNF do problema

```

fun reject (A) is
    -- inicialização
    1   prob  $\leftarrow A$  ,
    2   escolher uma variável x livre em prob
    3   stack  $\leftarrow \varepsilon$  , push(x) , push( $\neg x$ )
        -- fazer enquanto existirem atribuições no stack
    4   while stack  $\neq \varepsilon$ 
        5     v  $\leftarrow$  pop
        6     Reduzir por propagação unária v  $\wedge A \rightsquigarrow \mu \wedge B$ 
        7     if B é o conjunto vazio de cláusulas then
            8       return false e, eventualmente,  $\mu$  como testemunha
        9     elif B contém a cláusula vazia then
            -- conflito
        10    adicional a cláusula aprendida  $\neg v$  a prob
        11    else
        12      escolher uma variável z livre em B
        13      push( $\mu \wedge z$ ) , push( $\mu \wedge \neg z$ )
    14  return true
```

**Algorithm 2.3:** “Conflict Drivem Clause Learning”

Para representar uma atribuição vamos usar a seguinte convenção: o literal  $x_i$  representa-se por *i* e o literal  $\neg x_i$  representa-se por  $\bar{i}$ . Por exemplo, a atribuição  $x_1 \wedge \neg x_2 \wedge x_5 \wedge \neg x_3$  é representada por  $1\bar{2}5\bar{3}$ .

Vamos representar um grafo de implicação iniciado com a atribuição  $\bar{1}\bar{8}$ . Por propagação unária obtém-se a seguinte cadeia de simplificações

$$\begin{aligned}
 & A \wedge \neg x_1 \neg x_8 \rightsquigarrow \dots \\
 & \neg x_1 \wedge \neg x_8 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4 \wedge (\neg x_4 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_6 \vee x_7) \wedge (x_5 \vee x_6) \rightsquigarrow \dots \\
 & \dots \rightsquigarrow \neg x_1 \wedge \neg x_8 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4 \wedge \neg x_5 \wedge (\neg x_6 \vee x_7) \wedge (x_5 \vee x_6) \rightsquigarrow \dots \\
 & \dots \rightsquigarrow \neg x_1 \wedge \neg x_8 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4 \wedge \neg x_5 \wedge x_6 \wedge x_7
 \end{aligned}$$

O grafo de implicação reduz-se a



Como  $\bar{1}\bar{8}\bar{2}\bar{3}\bar{4}\bar{5}67$  é uma atribuição completa, conclui-se que *A* é satisfazível e que é verificada, pelo menos, pelo modelo 00010110.

Obviamente a solução tão simples resulta de se ter feito uma boa escolha da atribuição inicial. Vamos supor que se fazia o mesmo tipo de simplificações a partir de uma outra atribuição inicial; por exemplo,  $\bar{1}\bar{8}\bar{7}$ .

$$\begin{aligned}
 & \neg x_1 \wedge \neg x_7 \wedge A \rightsquigarrow \dots \\
 & \neg x_1 \wedge \neg x_7 \wedge \neg x_3 \wedge (\neg x_2 \vee x_8) \wedge (x_2 \vee x_4) \wedge (\neg x_4 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_6) \wedge (x_5 \vee x_6)
 \end{aligned}$$

Esta redução é representada pelo ramo



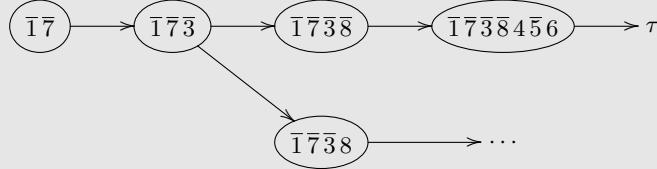
Aumentando a atribuição  $\bar{1}\bar{7}\bar{3}$  com o literal  $\neg x_8$ , constrói-se mais um nodo do grafo



Por propagação unária obtém-se a redução

$$A \wedge \neg x_1 \wedge \neg x_7 \wedge \neg x_8 \rightsquigarrow \neg x_1 \wedge \neg x_7 \wedge \neg x_8 \wedge x_4 \wedge \neg x_5 \wedge x_6 \wedge \neg x_6$$

No grafo tem-se



Este grafo indica-nos que a atribuição mais geral que conduz a um conflito é  $\bar{1}\bar{7}\bar{3}\bar{8}$ ; a sua negação ( $x_1 \vee x_7 \vee x_3 \vee x_8$ ) é a cláusula aprendida.  $\square$

Em alternativa a uma abordagem completa e segura para resolver o problema do SAT, cujo paradigma é o algoritmo 2.3 (página 35), pode-se tentar um tipo de algoritmo probabilístico como o ilustrado no algoritmo 2.2 (página 28).

Para isso vamos modificar este algoritmo de forma a que, quando  $m \models A$  falha, não se vai imediatamente tentar um novo modelo gerado aleatoriamente mas tenta-se ainda “aproveitar”  $m$  mudando algum dos seus bits em função do número de cláusulas que são validadas. O algoritmo 2.4 incorpora estas mudanças.

#### Observações

A operação básica é o “flip” de uma posição  $i$  num modelo  $m$ ; assim  $\text{flip}(i, m)$  é tal tal que o modelo  $m \oplus \text{flip}(i, m)$  tem todas as componentes a 0 exceto a  $i$ -ésima.

Para cada cláusula  $C$ ,  $\text{flip}(i, m, C)$  é um inteiro que toma um de três valores  $\{-1, 0, 1\}$ . Tem-se  $\text{flip}(i, m, C) = 0$  quando  $\text{flip}(i, m)$  não altera o fato de o modelo satisfazer ou não a cláusula  $C$ :  $m$  satisfaz  $C$  se e só se  $\text{flip}(i, m)$  satisfaz  $C$ ; tem-se  $\text{flip}(i, m, C) = -1$  quando existe uma perda de satisfação:  $m$  satisfaz  $C$  mas  $\text{flip}(i, m)$  não satisfaz  $C$ ; finalmente tem-se  $\text{flip}(i, m, C) = +1$  quando existe um ganho de satisfação:  $m$  não satisfaz  $C$  mas  $\text{flip}(i, m)$  satisfaz  $C$ .

Para uma CNF  $A = \bigwedge C$  define-se

$$\text{flip}(i, m, A) \stackrel{\text{def}}{=} \sum_{C \in A} \text{flip}(i, m, C)$$

Essencialmente  $\text{flip}(i, m, A)$  mede o ganho total em cláusulas de  $A$  que alteram o seu grau de satisfação quando o  $i$ -ésimo bit de  $m$  é alterado.

O algoritmo WALK-SAT usa ainda a distribuição de Bernoulli  $\{0, 1\}_\eta$ ; uma amostragem nesta distribuição gera o bit 0 com probabilidade  $\eta$  e o bit 1 com probabilidade  $1 - \eta$ .  $\square$

No algoritmo 2.4,  $A = \bigwedge_i C_i$  é uma CNF com cláusulas disjuntivas  $C_i$ . O parâmetro  $N$  limita o número de “restarts”; isto é o número de vezes que o algoritmo de procura

é reiniciado com um novo ponto de partida  $m$  gerado aleatoriamente. O parâmetro  $M$  limita o número de “flips” que são efetuados, na procura iniciada em  $m$ . Cada procura segue uma estratégia em que os bits são modificados de uma de duas formas: ou por escolha aleatória de uma posição no intervalo  $\{1..κ\}$  ou então por seleção da posição que maximiza o “ganho em satisfação” de determinadas variáveis; o parâmetro  $η$  é uma probabilidade que determina a forma como é escolhido tal bit.

```

fun reject ( $A$ ) is
    do  $N$  times
         $m \leftarrow \{0, 1\}^κ$ 
        do  $M$  times
            if para todo  $C \in A$ ,  $m \models C$  then
                return false e testemunha  $m$ 
            else
                 $b \leftarrow \{0, 1\}_η$ 
                if  $b = 0$  then  $k \leftarrow \{1..κ\}$ 
                else
                    escolhe aleatoriamente uma cláusula  $C$  tal que  $m \not\models C$ 
                    de entre as variáveis  $x_i \in C$ 
                    seleciona  $k$  que maximiza  $\text{flip}(i, m, A)$ 
                 $m \leftarrow \text{flip}(k, m)$ 
        return true

```

**Algorithm 2.4:** Algoritmo WALK-SAT.

## 2.2 Aplicações

O resto deste curso é essencialmente dedicado às aplicações de SAT e das suas variantes. Nesta secção pretende-se focar apenas, a título de exemplo, um problema central da criptoanálise: computar a chave  $k$  da cifra representada na figura 2.1 a partir de um número finito de pares *input/output*  $(x, y)$  conhecidos.

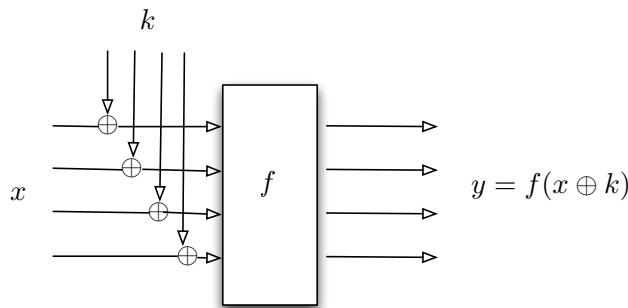


Figura 2.1: Cifra básica

Designamos este problema por *inversão da chave* e vamos analisa-lo sob duas perspetivas: a de funções booleanas  $f: (\mathbb{F}_2)^n \rightarrow (\mathbb{F}_2)^n$  e a de polinómios no corpo  $\mathbb{F}_{2^n}$ .

### 2.2.1 Inversão da chave numa *S-Box*

Pode-se representar a função de cifra  $f$  é uma função booleana com  $n$  entradas e  $n$  saídas, usualmente designada por uma “substitution box”  $n \times n$ , e 3 vetores de variáveis booleanas: o *input*  $x$ , a *chave*  $k$  e o *output*  $y$ . A relação entre estas variáveis é

$$y = f(x \oplus k) \quad (2.18)$$

O problema pode agora enunciar-se da seguinte forma.

#### § 2.10 Problema da inversão da chave numa *S-Box*.

1. É dada uma *S-Box* através de uma função booleana  $f: (\mathbb{F}_2)^n \rightarrow (\mathbb{F}_2)^n$ .
2. São dados  $N$  pares de vetores *input/output*  $(\bar{x}_i, \bar{y}_i)$ , com  $\bar{x}_i, \bar{y}_i \in \{0, 1\}^n$ .
3. Pretende-se determinar uma chave  $\bar{k} \in \{0, 1\}^n$  tal que, para todo  $i = 1..N$  é satisfeita a equação

$$\bar{y}_i = f(\bar{x}_i \oplus \bar{k}) \quad (2.19)$$

**Variante 1** A *S-Box* é definida no corpo finito  $\mathbb{F}_{2^n}$ . Os  $x_i$ ,  $y_i$  e  $k$  são elementos desse corpo e  $f$  é um polinómio a uma variável com coeficientes nesse corpo.  $\square$

**Variante 2** Se não existe nenhum  $\bar{k}$  que verifique todas as equações (2.19), procura-se determinar o  $\bar{k}$  que verifica o maior número de equações.

Para tal escolhem-se  $N$  pesos racionais  $w_i \geq 0$  que verificam  $\sum_i w_i = 1$  e procura-se encontrar  $\bar{k}$  que minimiza o “custo”  $\sum_i w_i (\bar{y}_i \oplus f(\bar{x}_i \oplus \bar{k}))$ .  $\square$

Vamos tentar converter este problema num problema de SAT. Para isso, vamos ilustrar este método de conversão com um exemplo muito simples.

Vamos supor que temos uma *S-Box*  $4 \times 4$ , descrita por 4 funções booleanas representadas por proposições

$$\begin{aligned} y_1 &= x_1 \wedge (x_2 \vee x_4) \\ y_2 &= (x_1 \vee \neg x_2) \wedge x_4 \\ y_3 &= (x_3 \vee \neg x_4) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee x_3) \\ y_4 &= x_2 \wedge x_4 \end{aligned} \quad (2.20)$$

Estas equações representam a relação  $\bar{y} = f(\bar{x})$ . Para representar  $\bar{y} = f(\bar{x} + \bar{k})$  temos de substituir  $\bar{x}$  por  $\bar{x} \oplus \bar{k}$  em (2.20).

Considere-se apenas um par *input/output*:  $(\bar{x}, \bar{y}) = (0101, 1010)$ . Substituindo, em (2.20), cada componente  $x_i$  por  $x_i + k_i$  e, em seguida, os  $x_i$  e  $y_i$  pelas constantes definidas neste par, obtém-se

$$\begin{aligned} 1 &= k_1 \wedge ((1 + k_2) \vee (1 + k_4)) \\ 0 &= (k_1 \vee \neg(1 + k_2)) \wedge (1 + k_4) \\ 1 &= (k_3 \vee \neg(1 + k_4)) \wedge (k_1 \vee k_3) \wedge (\neg(1 + k_2) \vee k_3) \\ 0 &= (1 + k_2) \wedge (1 + k_4) \end{aligned} \quad (2.21)$$

O passo seguinte consiste em converter todos  $(1 + k)$  em  $\neg k$  e os  $\neg(1 + k)$  em  $k$ .

$$\begin{aligned} 1 &= k_1 \wedge (\neg k_2 \vee \neg k_4) \\ 0 &= (k_1 \vee k_2) \wedge \neg k_4 \\ 1 &= (k_3 \vee k_4) \wedge (k_1 \vee k_3) \wedge (k_2 \vee k_3) \\ 0 &= \neg k_2 \wedge \neg k_4 \end{aligned} \tag{2.22}$$

Finalmente vamos converter as equações que têm 0 no lado esquerdo, as equações negativas, em equações positivas passando o lado direito para a forma negada. Neste exemplo temos duas equações negativas:  $0 = (k_1 \vee k_2) \wedge \neg k_4$  converte-se em  $1 = (\neg k_1 \wedge \neg k_2) \vee k_4 = (\neg k_1 \vee k_4) \wedge (\neg k_2 \vee k_4)$  e a equação  $0 = \neg k_2 \wedge \neg k_4$  passa a  $1 = (k_2 \vee k_4)$ .

Portanto, o par  $(\bar{x}, \bar{y}) = (0101, 1010)$  determina as seguintes 4 equações

$$\begin{aligned} 1 &= k_1 \wedge (\neg k_2 \vee \neg k_4) \\ 1 &= (\neg k_1 \vee k_4) \wedge (\neg k_2 \vee k_4) \\ 1 &= (k_3 \vee k_4) \wedge (k_1 \vee k_3) \wedge (k_2 \vee k_3) \\ 1 &= (k_2 \vee k_4) \end{aligned} \tag{2.23}$$

Note-se que o lado direito de cada equação é uma CNF nas 4 variáveis  $k_1, k_2, k_3, k_4$ . O problema da inversão de chaves consiste em encontrar atribuições a estas variáveis que validem todas estas CNF's. Para isso basta construir uma CNF  $F$  que é a conjunção destas CNF's individuais

$$\begin{aligned} F \equiv & k_1 \wedge (\neg k_2 \vee \neg k_4) \wedge (\neg k_1 \vee k_4) \wedge (\neg k_2 \vee k_4) \wedge \\ & \wedge (k_3 \vee k_4) \wedge (k_1 \vee k_3) \wedge (k_2 \vee k_3) \wedge (k_2 \vee k_4) \end{aligned} \tag{2.24}$$

e usar um algoritmo de SAT para determinar a atribuição de  $k$  que satisfaz  $F$ .

Neste exemplo é simples verificar se existe tal atribuição; usando propagação unária

$$\begin{aligned} F \rightsquigarrow & k_1 \wedge (\neg k_2 \vee \neg k_4) \wedge k_4 \wedge (\neg k_2 \vee k_4) \wedge (k_3 \vee k_4) \wedge (k_2 \vee k_3) \wedge (k_2 \vee k_4) \\ \rightsquigarrow & k_1 \wedge k_4 \wedge \neg k_2 \wedge (k_2 \vee k_3) \rightsquigarrow k_1 \wedge k_4 \wedge \neg k_2 \wedge k_3 \end{aligned}$$

Portanto só existe uma única atribuição,  $\bar{k} = 1011$ , que satisfaz  $F$ .

Este exemplo sumaria o método de resolução do problema da inversão da chave numa S-Box, usando SAT, quando esta é definida por um vetor de funções booleanas.

A essência deste método pode-se sumariar no algoritmo em § 2.11.

---

**§ 2.11** Conversão num problema SAT do problema em § 2.10.
 

---

Assume-se que a  $S$ -Box  $n \times n f$  é determinada por um vetor de  $n$  funções booleanas  $f = \langle f_1, \dots, f_n \rangle$ .

1. Converte-se cada função  $f_i$  numa CNF; procede-se do mesmo modo para as funções  $1 + f_i$ ; todas as CNF's são expressas nas variáveis  $x_1, x_2, \dots, x_n$ .
2. Para cada par *input/output*  $(\bar{x}_i, \bar{y}_i)$  constrói-se a CNF  $F_i$  do seguinte modo:

Sejam  $\bar{x}_{i,j}$  e  $\bar{y}_{i,j}$  as componentes de ordem  $j$  dos vetores  $\bar{x}_i$  e  $\bar{y}_i$

Para cada  $j = 1..n$

- seleciona-se a CNF  $f_j$ , se  $\bar{y}_{i,j} = 1$ , ou a CNF  $1 + f_j$  quando  $\bar{y}_{i,j} = 0$ .
  - na CNF selecionada, para todo  $k = 1..n$ , substitui-se a variável  $x_k$  por  $\neg x_k$  sempre que for  $\bar{x}_{i,k} = 1$ .
- Seja  $U_{i,j}$  a CNF resultante.

Faz-se  $F_i = \bigwedge_j U_{i,j}$

3. Constrói-se  $F = \bigwedge_i F_i$ .

Usa-se um algoritmo de SAT para construir uma atribuição que valide  $F$ . Essa atribuição determina possíveis soluções de § 2.10.

---

## 2.2.2 Outra vez as funções booleanas

Antes de procurarmos resolver o problema da inversão de chaves no corpo  $\mathbb{F}_{2^n}$ , vamos estender a análise que fizemos na secção 1.4 (página 8) e procurar regras de conversão de representação para algumas classes de funções booleanas.

Recordemos de § 1.1, na página 10, as relações entre vetores  $\bar{a} \in (\mathbb{F}_2)^n$  e elementos  $a \in \mathbb{F}_{2^n}$ , relações essas determinadas por uma base  $\beta = \langle \beta_1, \dots, \beta_n \rangle$  de  $\mathbb{F}_{2^n}$ .

$$G = \lambda(\beta^\top) \quad , \quad T = G G^\top \quad , \quad G^{-1} = G^\top T^{-1} \quad (2.25)$$

$$\lambda(a) = \bar{a} G \quad , \quad \bar{a} = \lambda(a) G^{-1} \quad (2.26)$$

Adicionalmente vamos introduzir o *elemento dual* de  $a$ , representado por  $a'$ , tal que

$$\bar{a}' = \bar{a} T \quad , \quad \lambda(a') = \bar{a} (G^\top)^{-1} \quad , \quad \bar{a}' = \lambda(a) G^\top \quad (2.27)$$

Note-se que  $(G^\top)^{-1} = T^{-1} G$ .

Com estas entidades e relações entre entidades podemos analisar as possíveis representações de vários tipos de funções de domínio  $(\mathbb{F}_2)^n$  ou  $\mathbb{F}_{2^n}$ . Formalmente,

Dada uma base  $\beta$  de  $\mathbb{F}_{2^n}$ , procura-se relacionar  $f: (\mathbb{F}_2)^n \rightarrow (\mathbb{F}_2)^n$ , definida por um polinómio multivariável de coeficiente em  $\mathbb{F}_2$ , com  $F: \mathbb{F}_{2^n} \rightarrow \mathbb{F}_{2^n}$ , definida por um polinómio a uma variável com coeficientes em  $\mathbb{F}_{2^n}$ , tal que

$$\overline{F(a)} = f(\bar{a}) \quad \text{para todo } a \in \mathbb{F}_{2^n} \quad (2.28)$$

Neste contexto colocam-se dois problemas

- *problema direto*: dado  $f$ , determinar  $F$ , e
- *problema inverso*: dado  $F$ , determinar  $f$

### Testes de igualdade

A função booleana que testa a igualdade  $x \stackrel{?}{=} 0$  tem o nome de *função de Kronecker*\* e representa-se por  $\delta(x)$ . Tem-se sempre  $\delta(0) = 1$  e  $\delta(x) = 0$  para todo  $x \neq 0$ .

Com assinatura  $(\mathbb{F}_2)^n \rightarrow \mathbb{F}_2$  a função  $\delta(\bar{x})$ , com  $\bar{x} = \langle x_1, \dots, x_n \rangle$ , implementa-se

$$\delta(\bar{x}) = (1 + x_1)(1 + x_2) \cdots (1 + x_n)$$

Com a assinatura  $\mathbb{F}_{2^n} \rightarrow \mathbb{F}_2$  a função  $\delta(x)$  implementa-se por

$$\delta(x) = 1 + x^{2^n - 1}$$

Esta implementação justifica-se por uma das propriedades do corpo  $\mathbb{F}_{2^n}$ : para todo  $x$  verifica-se  $x^{2^n} = x$  e, por isso,  $x^{2^n - 1} = 1$  para todo  $x \neq 0$ .

O teste de igualdade a 0 generaliza à igualdade entre quaisquer  $x, y \in \mathbb{F}_{2^n}$  ou  $\bar{x}, \bar{y} \in (\mathbb{F}_2)^n$ . Note-se que  $x = y$  verifica-se se e só se  $x + y = 0$ ; portanto

$$x \stackrel{?}{=} y \equiv \delta(x + y) \quad (2.29)$$

### Funções lineares $(\mathbb{F}_2)^n \rightarrow \mathbb{F}_2$

Uma função booleana linear  $f: (\mathbb{F}_2)^n \rightarrow \mathbb{F}_2$  tem a forma  $f(a_1, \dots, a_n) = \sum_{i=1}^n c_i a_i$  com os  $c_i$ 's bits. Sendo  $\bar{c}$  o vetor dos  $c_i$ 's, tem-se  $f(\bar{a}) = \bar{a} \bar{c}^\top$ .

Seja  $a \in \mathbb{F}_{2^n}$  o elemento de coordenadas  $\bar{c}$ ; então  $F(a) = \text{tr}(a c')$ .

**Justificação** Como o contradomínio  $\mathbb{F}_2$  tem apenas uma componente, tem-se  $\overline{F(\bar{a})} \equiv F(a)$ . Como  $\bar{a} \bar{c}^\top = \bar{a} G G^{-1} \bar{c}^\top = \bar{a} G (\bar{c} T^{-1} G)^\top = \lambda(a) \lambda(c')^\top$ , conclui-se que

$$F(a) = f(\bar{a}) = \text{tr}(a c')$$

□

### Funções lineares $(\mathbb{F}_2)^n \rightarrow (\mathbb{F}_2)^n$

A função linear  $f: (\mathbb{F}_2)^n \rightarrow (\mathbb{F}_2)^n$  pode ser definida por uma matriz  $H \in (\mathbb{F}_2)^{n \times n}$  através da relação  $f(\bar{a}) = \bar{a} H^\top$ .

Se pensarmos em  $H$  organizada em linhas  $\langle \bar{c}_1, \dots, \bar{c}_n \rangle$ , a  $i$ -ésima componente de  $f(\bar{a})$  é o produto interno  $\bar{a} \bar{c}_i^\top$  que, como vimos no caso anterior, é igual a  $\lambda(a) \lambda(c'_i)^\top$ . Portanto

$$F(a) = \sum_i \lambda(a) \lambda(c'_i)^\top \beta_i$$

Para construir formalmente a função  $F$  vamos construir primeiro uma mudança de base definindo  $\eta = \beta H$ ; então

$$\begin{aligned} F(a) &= f(\bar{a}) \beta^\top = \bar{a} H^\top \beta^\top = \bar{a} \eta^\top = \lambda(a) G^{-1} \eta^\top \\ &= \lambda(a) h^\top \end{aligned}$$

sendo  $h \in (\mathbb{F}_{2^n})^n$  definido como  $h \stackrel{\text{def}}{=} \eta (G^\top)^{-1} = \eta (T^{-1} G)$ .

---

\*LEOPOLD KRONECKER 1823-1891 foi um importante matemático do século XIX essencialmente interessado na axiomatização da Matemática. A sua mais famosa frase “Deus criou os inteiros, tudo o resto é trabalho dos homens”, tem sido reescrita no século XXI como “Deus criou as bit-strings, tudo o resto é trabalho de programadores”!

A transformação  $H \Rightarrow h$  permite resolver o problema direto; a solução do problema inverso  $h \Rightarrow H$  parte das igualdades  $h G^\top = \eta = \beta H$ ; donde  $\lambda(h G^\top) = G^\top H$  o que permite concluir  $H = (G^\top)^{-1} \lambda(h G^\top)$ .

**Funções bilineares**  $(\mathbb{F}_2)^n \times (\mathbb{F}_2)^n \rightarrow (\mathbb{F}_2)^n$  e  $\mathbb{F}_{2^n} \times \mathbb{F}_{2^n} \rightarrow \mathbb{F}_{2^n}$

Uma função  $f(\bar{x}, \bar{y})$  é descrita por  $n$  funções booleanas  $f_i: (\mathbb{F}_2)^n \times (\mathbb{F}_2)^n \rightarrow \mathbb{F}_2$ ; cada  $f_i$  determina a  $i$ -ésima componente de  $f$  e, sendo bilinear, é completamente determinada por uma matriz de bits  $H_i \in (\mathbb{F}_2)^{n \times n}$  e pela relação

$$f_i(\bar{x}, \bar{y}) = \bar{x} H_i \bar{y}^\top$$

Por outro lado, com domínio em  $\mathbb{F}_{2^n}$  toda a função bilinear tem a forma

$$F(x, y) = \lambda(x) \mathbf{A} \lambda(y)^\top$$

sendo  $\mathbf{A}$  uma matriz no espaço  $(\mathbb{F}_{2^n})^{n \times n}$ .

Como  $\lambda(x) = \bar{x} G$  e  $\lambda(y)^\top = G^\top \bar{y}^\top$ , fazendo

$$\mathbf{H} \stackrel{\text{def}}{=} G^{-1} \mathbf{A} (G^\top)^{-1}$$

temos  $F(x, y) = \bar{x} \mathbf{H} \bar{y}^\top$ . Decompondo cada elemento de  $\mathbf{H} = \sum_i H_i \beta_i$  nas suas componentes constroem-se matrizes de bits  $H_i$  que verificam as igualdades

$$f_i(\bar{x}, \bar{y}) = \overline{F(x, y)}_i = \bar{x} H_i \bar{y}^\top$$

Estas relações determinam a solução do problema inverso; para resolver o problema direto  $H_1, \dots, H_n \Rightarrow \mathbf{A}$  constrói-se  $\mathbf{H} \leftarrow \sum_i H_i \beta_i$  e depois  $\mathbf{A} \leftarrow G \mathbf{H} G^\top$ .

### Funções quadráticas

Uma *função quadrática*, de domínio num espaço vetorial qualquer  $X$ , é uma função da forma  $f(x, x)$  sendo  $f$  uma função bilinear de domínio  $X^2$ .

Desta forma

- uma função quadrática com domínio  $\mathbb{F}_{2^n}$  tem a forma  $F(x) = \lambda(x) \mathbf{A} \lambda(x)^\top$  com  $\mathbf{A} \in (\mathbb{F}_{2^n})^{n \times n}$ .
- uma função booleana quadrática tem a forma  $f(\bar{x}) = \bar{x} H \bar{x}^\top$ , sendo  $H$  uma matriz  $n \times n$  de bits.

Portanto a conversão de uma forma para a outra faz-se como nas funções bilineares.

### 2.2.3 Inversão da chave no corpo binário $\mathbb{F}_{2^n}$

Para resolver este problema quando a cifra é definida por uma função  $f: \mathbb{F}_{2^n} \rightarrow \mathbb{F}_{2^n}$  pode-se proceder de forma análoga ao que foi feito na secção 2.2.1.

Para isso toma-se a igualdade básica  $y = f(x)$  e tenta-se escrever o lado direito como CNF's. Isto passa por converter a função de domínio  $\mathbb{F}_{2^n}$  em  $n$  funções (uma para cada componente dos  $y$ 's) de domínio  $(\mathbb{F}_2)^n$ . Como vimos na secção anterior, a menos que  $f$  seja uma função simples, linear ou bilinear, não é conhecido um modo sistemático de efetuar tal conversão.

Em alternativa, a igualdade  $y = f(x)$  é escrita de forma implícita como

$$F(x, y) = 1 \quad \text{sse} \quad y = f(x) \quad (2.30)$$

sendo  $F: \mathbb{F}_{2^n} \times \mathbb{F}_{2^n} \rightarrow \mathbb{F}_2$  uma função booleana com dois argumentos no corpo  $\mathbb{F}_{2^n}$ .

Agora o problema da inversão de chaves (ver figura 2.2) assume que a chave se divide em duas componentes: uma componente  $k$  altera a entrada  $x$  e uma componente  $w$  modifica a saída  $y$ . O problema consiste em encontrar  $k$  e  $w$  tal que se verifique

$$f(x_i + k) + w = y_i$$

conhecidos  $N$  pares  $(x_i, y_i)$ . Equivalentemente, usando a representação implícita, o problema procura determinar as chaves  $(k, w)$  que, para todos os  $(x_i, y_i)$ , verificam

$$F(x_i + k, y_i + w) = 1 \quad (2.31)$$

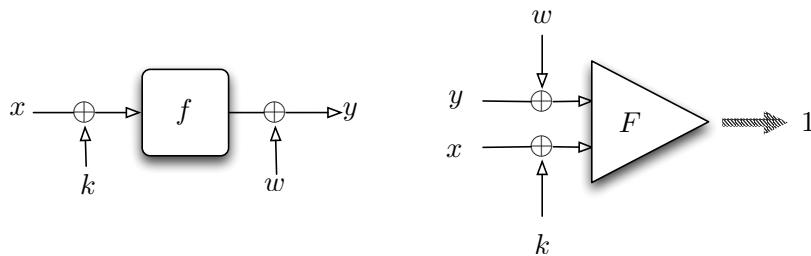


Figura 2.2: Inversão de chave dupla.

**EXEMPLO 2.7:** Na cifra AES a função não-linear que garante a sua segurança é a “pseudo-inversa”  $f(x) = x^{254}$  definida num corpo finito  $\mathbb{F}_{256}$ . Esta função verifica

$$x f(x) = \begin{cases} 1 & \text{se } x \neq 0 \\ 0 & \text{se } x = 0 \end{cases}$$

Faz sentido definir uma função bilinear

$$g(x, y) = xy$$

que vai verificar  $g(x, f(x)) = 1$ , se  $x \neq 0$ , e  $g(0, 0) = 0$ .  $\square$

Uma vez determinada a função booleana implícita  $F(x, y)$  é preciso convertê-la numa CNF susceptível de ser usada em SAT “solvers”.

**EXEMPLO 2.8:** Neste exemplo vamos tomar a função bilinear definida no exemplo anterior e definir uma função semelhante mas em  $\mathbb{F}_{16}$ . Para tal vamos recuperar os elementos, incluindo a matriz de conversão  $G$ , que calculamos no exemplo 1.6, página 10. Em  $\mathbb{F}_{16}$  a pseudo-inversa é dada pelo polinómio  $f(x) = x^{14}$ .

A função bilinear  $g(x, y)$ , escrita como  $\lambda(x) \mathbf{A} \lambda(y)^\top$ , é determinada pela matriz

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

como se pode confirmar atendendo que  $\lambda(x) = (x, x^2, x^4, x^8)$  e  $\lambda(y) = (y, y^2, y^4, y^8)$ .

Para transformar igualdade  $g(x, y) = 1$  numa CNF usamos a estratégia já utilizada em § 2.11: convertemos a função na sua representação vetorial  $\bar{x} \mathbf{H} \bar{y}$  e igualamos componente a componente com a representação vetorial do elemento 1.

A matriz  $\mathbf{H} = G^{-1} \mathbf{A} (G^\top)^{-1}$  é, neste caso,

$$\mathbf{H} = \begin{pmatrix} \alpha^3 + \alpha^2 + 1 & \alpha^3 + \alpha^2 + \alpha + 1 & \alpha^2 + \alpha + 1 & \alpha^3 + \alpha^2 \\ \alpha^3 + \alpha^2 + \alpha + 1 & \alpha^3 + \alpha^2 + \alpha & \alpha^3 + \alpha & \alpha^2 + \alpha \\ \alpha^2 + \alpha + 1 & \alpha^3 + \alpha & \alpha^3 + \alpha + 1 & \alpha^3 \\ \alpha^3 + \alpha^2 & \alpha^2 + \alpha & \alpha^3 & \alpha^3 + 1 \end{pmatrix}$$

Decompondo  $\mathbf{H}$  obtém-se 4 matrizes de bits  $H_1, H_2, H_3, H_4$ , respectivamente

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

Cada uma destas matrizes determina uma função booleana  $f_i(\bar{x}, \bar{y}) = \bar{x} H_i \bar{y}$  nas 8 variáveis  $x_1, \dots, x_4, y_1, \dots, y_4$ . Neste caso as funções são

$$\begin{aligned} f_1 &= x_1 y_1 + x_1 y_2 + x_1 y_3 + x_2 y_1 + x_3 y_1 + x_3 y_3 + x_4 y_4 \\ f_2 &= x_1 y_2 + x_1 y_3 + x_2 y_1 + x_2 y_2 + x_2 y_3 + x_2 y_4 + x_3 y_1 + x_3 y_2 + x_3 y_3 + x_4 y_2 \\ f_3 &= x_1 y_1 + x_1 y_2 + x_1 y_3 + x_1 y_4 + x_2 y_1 + x_2 y_2 + x_2 y_4 + x_3 y_1 + x_4 y_1 + x_4 y_2 \\ f_4 &= x_1 y_1 + x_1 y_2 + x_1 y_4 + x_2 y_1 + x_2 y_2 + x_2 y_3 + x_3 y_2 + x_2 y_3 + x_3 y_4 + x_4 y_1 + x_4 y_3 + x_4 y_4 \end{aligned}$$

O elemento 1 de  $\mathbb{F}_{2^n}$  tem, com a base aqui usada, a representação vetorial  $\langle 1, 0, 0, 0 \rangle$ . Ficamos assim com 4 equações

$$\{f_1 = 1, f_2 = 0, f_3 = 0, f_4 = 0\}$$

que se podem converter numa CNF  $P \equiv f_1 \wedge (1 + f_2) \wedge (1 + f_3) \wedge (1 + f_4)$ .  $\square$

A partir do momento que se constrói uma CNF  $P(\bar{x}, \bar{y})$  que represente o predicado  $F(x, y)$ , pode-se usar uma metodologia análoga à usada na secção 2.2.1 para converter num problema SAT a determinação de pares de chaves  $(k, w)$  que verificam (2.31).

Essencialmente

- (i) Para cada um par *input/output*  $(\bar{x}_i, \bar{y}_i)$ , constrói-se a CNF  $P_i$ , aplicando a  $P$  a substituição  $x_j \rightsquigarrow \neg x_j$  sempre que  $\bar{x}_{i,j} = 1$  e aplicando a substituição  $y_j \rightsquigarrow \neg y_j$  sempre que  $\bar{y}_{i,j} = 1$ ,
- (ii) Constrói-se a CNF global  $P' = \bigwedge_i P_i$  e usa-se um algoritmo de SAT para construir uma testemunha da satisfação desta fórmula.

Portanto a questão essencial é, por um lado, converter uma equação  $y = f(x)$  num predicado  $F(x, y)$  e depois converter esse predicado numa CNF  $P(\bar{x}, \bar{y})$ .

Estas conversões dependem, obviamente, da forma da função inicial  $f$ ; como temos um reportório limitado de regras de conversão que permitam transformar o predicado  $F(x, y)$  em funções vetoriais (basicamente só dispomos de regras de conversão

para funções lineares e bilineares), a função inicial  $f$  deve possuir uma forma que permite uma representação implícita que possa ser descrita neste reportório.

Algumas funções que não são lineares nem bilineares podem ser transportadas para uma destas classes através de um simples artifício: a constante 1 é interpretada como uma nova variável.

Representando por  $\bar{x}.1$  o vetor de variáveis aumentado com a “pseudo-variável” 1, quando uma CNF  $P(\bar{x}.1)$  é usada num algoritmo de SAT, são rejeitados os modelos  $m$  que validam  $P$  mas que têm a “pseudo-variável” 1 associada a **falso**; i.e. o algoritmo de SAT deve rejeitar qualquer atribuição que contenha o literal  $\neg 1$ .

#### EXEMPLO 2.9:

Vamos considerar algumas funções booleanas de domínio  $(\mathbb{F}_2)^n$ ; em todas estas funções  $\bar{x}$  representa o vetor  $\langle x_1, \dots, x_n \rangle$ ; portanto  $\bar{x}.1 = \langle x_1, \dots, x_n, 1 \rangle$ .

$$(i) \quad f(\bar{x}) = 1 + x_1 + \dots + x_n$$

Esta função verifica  $f(\bar{x} + \bar{y}) \neq f(\bar{x}) + f(\bar{y})$ ; por isso não é linear.

Porém, vendo 1 como uma variável,  $f(\bar{x}.1) = 1 + x_1 + \dots + x_n$  já é linear.

$$(ii) \quad f(\bar{x}, \bar{y}) = 1 + x_1 + x_1 \times y_1$$

A função  $f(\bar{x}, \bar{y})$  não é bilinear porque não é linear quer fixando  $\bar{x}$  quer fixando  $\bar{y}$ . No entanto escrevendo-a na forma

$$1 \times 1 + x_1 \times 1 + x_1 \times y_1$$

vemos que, adicionando a pseudo-variável 1 tanto a  $\bar{x}$  como a  $\bar{y}$ , se obtém uma função  $f(\bar{x}.1, \bar{y}.1)$  que é bilinear. De fato

$$f(\bar{x}.1, \bar{y}.1) = \langle x_1, 1 \rangle \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \langle y_1, 1 \rangle^\top$$

□

A construção de funções bilineares pode ser estendida a funções “trilineares”, i.e. fixando um dos argumentos a função é bilinear nos outros dois. No domínio  $\mathbb{F}_{2^n}$ , dadas duas funções bilineares

$$f(x, y) = \lambda(x) \mathbf{A} \lambda(y)^\top \quad , \quad g(x, y) = \lambda(x) \mathbf{B} \lambda(y)^\top$$

com  $\mathbf{A}, \mathbf{B} \in (\mathbb{F}_{2^n})^{n \times n}$ , constrói-se

$$h(x, y, z) = g(f(x, y), z)$$

Na representação vetorial tais funções são representadas por polinómios em que todos os monómios têm grau 3.

## 2.3 Variantes do problema SAT

O problema da satisfação de uma proposição  $A = \bigwedge_i C_i$ , representada na forma normal conjuntiva, e os algoritmos que lhe estão associados, podem ser adaptados a formas particulares das cláusulas ou a um objetivo que não seja só provar que  $A$  não é satisfazível.

Algumas das mais importantes variações são os problemas  $k$ -SAT, onde o número de literais em cada cláusula é exatamente  $k$ , o problema MAX-SAT onde se procura maximizar o número de cláusulas válidas, e os problemas ALL-SAT e #SAT que procuram determinar o conjunto  $\llbracket A \rrbracket$  dos modelos de  $A$  ou, simplesmente, a cardinalidade deste conjunto.

### 2.3.1 $k$ -SAT

No problema  $k$ -SAT exige-se que todas as cláusulas tenham exatamente  $k$  literais.

Alguns problemas  $k$ -SAT verificam propriedades específicas que convém examinar com algum detalhe. Nomeadamente

- (i) Existe uma algoritmo polinomial que resolve o problema 2-SAT.
- (ii) Existe um algoritmo polinomial que converte qualquer problema de SAT num problema 3-SAT à custa de um aumento, eventualmente exponencial, do número de cláusulas.

A ideia básica para uma algoritmo 2-SAT assenta na noção de *relação de implicaçāo* definida nos literais de  $A$ . Assume-se que  $A$  não contém cláusulas fechadas (i.e. da forma  $(\neg l \vee l)$ ). Então

- (a) A relação de implicaçāo  $l \rightsquigarrow l'$  verifica-se se existe em  $A$  um par de cláusulas  $(\neg l \vee x) \wedge (\neg x \vee l')$ .
- (b) A relação  $\rightsquigarrow^*$  é o fecho transitivo de  $\rightsquigarrow$ .
- (c) A relação de equivalência  $l \leftrightarrow l'$ , designada *ligação forte*, verifica-se quando ambas as relações  $l \rightsquigarrow^* l'$  e  $l' \rightsquigarrow^* l$  se verificam.

Prova-se facilmente, usando resolução e a completude da resolução (teorema 2.1) os seguintes resultados.

## 2.6 PROPOSIÇÃO

- (i) Verifica-se  $l \rightsquigarrow^* l'$  se e só o fecho por resolução de  $A$  contém  $(\neg l \vee l')$ .
- (ii) Se se verifica  $l \rightsquigarrow^* \neg l$ , então toda atribuição que valide  $A$  contém  $\neg l$ .
- (iii)  $A$  não é satisfazível se e só se existe um literal  $l$  que verifica  $l \leftrightarrow \neg l$ .

Usando este resultado pode-se construir um algoritmo que procura os literais fortemente ligados em  $A$ ; este algoritmo está esboçado em § 2.12.

O número de variáveis pode ser usado como referência na avaliação da complexidade dos algoritmos. Por exemplo, é simples verificar que o algoritmo em § 2.12 tem complexidade polinomial com o número de variáveis.

Pode-se também caracterizar a complexidade dos algoritmos de conversão entre proposições e CNF's. Como se sabe, a forma normal conjuntiva é uma representação universal das fórmulas proposicionais; isto é, toda a fórmula proposicional é equivalente a uma CNF. De facto prova-se algo mais forte,

---

**§ 2.12** Algoritmo 2SAT.

Dado a CNF  $A$  sem cláusulas fechadas e em que todas as cláusulas têm 2 literais distintos, pretende-se verificar se  $A$  não é satisfazível ou, sendo satisfazível, apresentar uma atribuição que valide  $A$

1. Construir o grafo de resolução de  $A$ .
  2. Inicializar uma atribuição  $v$  com o conjunto vazio de literais.
  3. Para cada literal  $l$  fazer,
    - Se  $l$  pertence a um caminho iniciado em  $l$ , adicionar  $l$  a  $v$ ; senão continuar com novo  $l$
    - Se  $v$  contém também  $\neg l$  terminar com a mensagem UNSAT; em caso contrário continuar com novo  $l$
  4. Terminar com a mensagem SAT e a atribuição  $v$ .
- 

**2.3 TEOREMA** Existe um algoritmo de complexidade exponencial com o número de variáveis que, sob “input” de uma fórmula proposicional  $\phi$  com  $\ell$  variáveis, produz uma CNF  $A$  que lhe é equivalente e tem o mesmo número de variáveis.

Adicionalmente, existe um algoritmo de complexidade linear com o número de variáveis que, sob “input” de uma fórmula proposicional  $\phi$  com  $\ell$  variáveis, produz uma CNF  $A$ , com um número de variáveis  $k \geq \ell$  que cresce exponencialmente com  $\ell$ , tal que todo o eventual modelo\* de  $\phi$  está contido num modelo de  $A$ .

Essencialmente os dois algoritmos referidos no teorema 2.3 têm a mesma funcionalidade: convertem uma fórmula proposicional arbitrária numa CNF que lhe é equivalente. As duas variantes diferem na complexidade e no número de variáveis do resultado: o primeiro algoritmo mantém o número de variáveis mas tem complexidade exponencial com o número de variáveis; o segundo algoritmo tem complexidade linear com o número de variáveis mas, em contrapartida, aumenta o número de variáveis exponencialmente.

Restringindo as CNF’s às classes  $k$ -CNF’s é simples provar uma redução do problema SAT genérico ao problema 3-SAT

**2.7 PROPOSIÇÃO** Existe um algoritmo linear no número de variáveis que, sob “input” de uma CNF  $P$  com  $\ell$  variáveis, constrói uma 3-CNF  $A$  com  $\ell + O(\ell)$  variáveis, tal que todo o eventual modelo de  $P$  é satisfazível está contido num modelo de  $A$ .

O mecanismo para converter uma cláusula com  $k > 3$  literais, numa CNF onde cada cláusula não contém mais de 3 literais, ilustra-se no exemplo 2.10.

**EXEMPLO 2.10:** Seja  $C = \{u_1 \vee u_2 \vee \dots \vee u_k\}$  uma cláusula não-fechada com  $k > 3$  literais. Como  $C$  não é fechada nenhum dos  $u_i$  é a negação de um outro literal em  $C$ .

Considere-se o par de cláusulas  $C_1 = (u_1 \vee \dots \vee u_{k-2} \vee x)$  e  $C_2 = (u_{k-1} \vee u_k \vee \bar{x})$ , sendo  $x$  uma variável que não ocorre em nenhum dos  $u_i$ . Note-se que  $C_1$  tem  $k - 1$  literais e  $C_2$  tem 3 literais.

---

\*Entende-se, nesta relação, que um modelo é um conjunto de variáveis às quais está associado o valor de verdade **true**.

Como  $C$  é a resolução de  $C_1$  com  $C_2$ , substituindo  $C$  pelo par de cláusulas  $C_1, C_2$  obtém-se uma CNF equivalente com mais cláusulas mas com cláusulas menores.  $\square$

### 2.3.2 MAX-SAT

MAX-SAT é uma versão optimizada do SAT cujo objetivo é encontrar o modelo que maximiza o número de cláusulas verificadas.

Formalmente, dada a CNF  $A \equiv \bigwedge_{i=1}^{\ell} C_i$  e um conjunto de *pesos* racionais  $w_i > 0$  tais que  $\sum_i w_i = 1$ , define-se a função custo

$$\phi(m) = \sum_{i=1}^{\ell} w_i C_i^{\{m\}} \quad (2.32)$$

e MAX-SAT determina  $m$  que maximiza  $\phi(m)$ .

Este problema pode ser formulado em termos de programação inteira.

Um problema de *programação inteira* (IP) tem, como constituinte básico, os chamados *cortes racionais* escritos como inequações

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \geq b \quad (2.33)$$

em que  $x_1, \dots, x_n$  são variáveis que tomam valores inteiros e  $a_1, \dots, a_n$  e  $b$  são constantes inteiras.

A cláusula disjuntiva  $C_i = l_1 \vee \cdots \vee l_k$  é convertida num corte racional usando as seguintes regras:

- (i) Para cada símbolo proposicional  $x_i$ , é criada uma variável inteira com o mesmo nome; a estas variáveis só podem ser atribuídos valores  $\{0, 1\}$ ; é importante ter em conta que 0 e 1 são agora inteiros e não valores de verdade.
- (ii) O literal  $x_j \in C_i$  é substituído pela variável inteira  $x_j$ ; o literal  $\neg x_j \in C_j$  é substituído pela expressão inteira  $(1 - x_j)$ ; as disjunções  $\vee$  são substituídas por adições inteiras. Seja  $C_i$  a expressão inteira assim formada.
- (iii) É criada uma nova variável inteira  $z_i$  associada à cláusula  $C_i$  e também limitada ao par de valores  $\{0, 1\}$ ; a asserção “a cláusula  $C_i$  é válida” é transformada na asserção “verifica-se  $C_i \geq z_i$ ”.

**EXEMPLO 2.11:** Considere-se a CNF  $A = C_1 \wedge C_2 \wedge C_3$  em que

$$C_1 = x_1 \vee \neg x_2 \vee \neg x_3, \quad C_2 = x_2 \vee x_3 \vee \neg x_4, \quad C_3 = \neg x_1 \vee x_4$$

Estas cláusulas são convertidas nas expressões

$$C_1 = x_1 + (1 - x_2) + (1 - x_3), \quad C_2 = x_2 + x_3 + (1 - x_4), \quad C_3 = (1 - x_1) + x_4$$

Criam-se novas variáveis  $z_1, z_2, z_3$  e cortes racionais  $C_i - z_i \geq 0$ ; neste caso obtêm-se os seguintes cortes racionais

$$x_1 - x_2 - x_3 - z_1 \geq -2, \quad x_2 + x_3 - x_4 - z_2 \geq -1, \quad -x_1 + x_4 - z_3 \geq -1$$

$\square$

Às variáveis  $z_i$  é atribuído o valor 1 ou o valor 0 consoante a cláusula  $C_i$  é satisfeita ou não; por isso faz sentido definir a função custo como  $\sum_{i=1}^{\ell} w_i z_i$ . Consequentemente o problema MAX-SAT reescreve-se com o seguinte problema IP:

---

### § 2.13

---

*Maximizar a função custo  $\phi = \sum_{i=1}^{\ell} w_i z_i$  sujeito às restrições:*

- $C_i \geq z_i$  para todo  $i$
  - $z_i \in \{0, 1\}$  para todas as cláusulas e  $x_j \in \{0, 1\}$  para todas as variáveis.
- 

**EXEMPLO 2.12:** Na continuação do exemplo 2.11, admitindo igual peso para todas as cláusulas ( $w_1 = w_2 = w_3$ ) a função custo é equivalente a  $\phi = z_1 + z_2 + z_3$ .

O problema MAX-SAT pode-se resolver achando uma solução do problema IP: maximizar esta função custo sujeito às restrições definidas pelos cortes racionais calculados no exemplo anterior e as restrições  $0 \leq z_1, z_2, z_3, x_1, x_2, x_3, x_4 \leq 1$ .  $\square$

O problema IP em § 2.13 pode ser aproximado usando a seguinte estratégia:

1. Convenciona-se um número  $N$ , que seja um limite inferior de cláusulas a satisfazer, e substitui-se a função custo  $\phi$  por uma restrição  $\sum_{i=1}^{\ell} w_i z_i \geq N$ .
2. Procura-se encontrar uma solução do sistema de restrições
  - $\sum_{i=1}^{\ell} z_i \geq N$
  - $C_i \geq z_i$  para todo  $i$
  - $z_i, x_j \in \{0, 1\}$  para todas as variáveis  $z_i$  e  $x_j$ .

Por simplicidade assume-se que todos os pesos  $w_i$  são iguais.

3. Se for encontrada uma solução, sob a forma de uma atribuição  $v$ , aumenta-se o valor de  $N$  e usa-se a atribuição como ponto de partida para uma procura de uma nova solução.

Para isso, faz-se uma “procura local” a partir de  $v$ : faz-se uma propagação unária por cada um dos literais em  $v$ , nas cláusulas  $C_i$ ; as cláusulas verificadas por esta atribuição são substituídas por `true` e, por isso, desaparecem do conjunto de cláusulas; no conjunto simplificado resolve-se o problema com um novo valor de  $N$  igual ao antigo  $N$  subtraído do número de cláusulas verificadas.

4. Se a solução não for melhorada o processo procura encontrar uma nova atribuição com o  $N$  antigo; para isso acrescenta  $\neg v$  como nova cláusula em  $A$ , evitando assim que o algoritmo encontre de novo o mesmo  $v$ , e regressa-se ao passo 2.

Note-se que uma atribuição  $v$  é uma cláusula conjuntiva: tem a forma  $l_1 \wedge \dots \wedge l_k$ ; por isso a sua negação  $\neg v = (\neg l_1 \vee \dots \vee \neg l_k)$  é uma cláusula disjuntiva que pode ser diretamente adicionada ao conjunto de cláusulas  $\{C_i\}$ .

Regressando a uma formulação do problema como CNF's, a construção em § 2.13 sugere que o problema MAX-SAT pode ser reformulado como SAT desde que:

1. Sejam introduzidas  $\ell$  novas variáveis  $z_i$
2. Cada cláusula  $C_i$  de  $A$  seja substituída por uma nova cláusula  $C'_i = C_i \vee \neg z_i$ ,
3. Seja possível definir uma CNF,  $\phi$  que representa a inequação  $\sum_{i=1}^{\ell} z_i \geq N$

Nestas circunstâncias define-se uma nova CNF

$$A' = \left( \bigwedge_{i=1}^{\ell} C'_i \right) \wedge \phi$$

e usa-se uma algoritmo de SAT para encontrar uma solução.

Falta apenas ver como se representa a inequação  $\sum_{i=1}^{\ell} z_i \geq N$  como uma CNF. Para ilustrar o método de representação considere-se  $\ell = 4$  e  $N = 3$ .

Essencialmente exige-se que, dentro do conjunto de 4 variáveis  $\mathcal{Z} = \{z_1, \dots, z_4\}$ , pelo menos um dos subconjunto com 3 variáveis  $v \subseteq \mathcal{Z}$ , com  $\#v = 3$ , tem todas as variáveis atribuídas a **true**. Aqui existem 4 conjuntos com 3 variáveis:  $\{x_1, x_2, x_3\}$ ,  $\{x_1, x_2, x_4\}$ ,  $\{x_1, x_3, x_4\}$  e  $\{x_2, x_3, x_4\}$ ; assim  $z_1 + z_2 + z_3 + z_4 \geq 3$  é

$$(x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_4) \vee (x_1 \wedge x_3 \wedge x_4) \vee (x_2 \wedge x_3 \wedge x_4) \quad (2.34)$$

Esta proposição porém está na forma normal disjuntiva e, por isso, não pode ser adicionada (pelo menos diretamente) à CNF  $A$ . Alternativamente pode-se exprimir o fato de não mais do que  $\ell - N$  variáveis  $z_i$  poderem ter atribuído o valor **false**; isto significa que cada subconjunto  $\mu \subseteq \mathcal{Z}$ , de tamanho  $\ell - N + 1$ , tem de ter pelo menos uma variável **true**.

Neste exemplo  $\ell - N + 1 = 2$ ; os conjuntos com 2 variáveis são  $\{x_1, x_2\}$ ,  $\{x_1, x_3\}$ ,  $\{x_1, x_4\}$ ,  $\{x_2, x_3\}$ ,  $\{x_2, x_4\}$  e  $\{x_3, x_4\}$ ; cada um destes conjuntos produz uma cláusula disjuntiva e tem-se

$$(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (x_1 \vee x_4) \wedge (x_2 \vee x_3) \wedge (x_2 \vee x_4) \wedge (x_3 \vee x_4) \quad (2.35)$$

No caso geral, para representar  $\sum_{i=1}^{\ell} z_i \geq N$  como uma CNF, geram-se todos os subconjuntos de  $\{z_1, \dots, z_\ell\}$  com  $(\ell - N + 1)$  elementos; cada um destes subconjuntos define uma cláusula disjuntiva e a conjunção destas cláusulas é a CNF pretendida.

### 2.3.3 ALL-SAT e #SAT

ALL-SAT é uma versão específica de SAT para determinar o conjunto de modelos que verificam uma CNF. Formalmente, dada a CNF  $A$  o problema ALL-SAT determina  $\llbracket A \rrbracket$ .

Uma solução do problema ALL-SAT usa uma técnica já esboçada no problema MAX-SAT: sempre que é encontrada uma solução do problema SAT sob a forma de uma atribuição  $v$ , acrescenta-se  $\neg v$  à CNF  $A$  e repete-se o processo até que a CNF resultante não seja satisfazível.

O problema #SAT, também designado por *model counting*, procura determinar quantos modelos verificam a CNF  $A$ ; i.e. a cardinalidade de  $\llbracket A \rrbracket$ .

```
fun ALL-SAT ( $A$ ) is
    results  $\leftarrow \{ \}$ 
    do forever :
         $(b, v) \leftarrow \text{SAT}(A)$ 
        if  $\neg b$  then return results
        results  $\leftarrow$  results  $\cup \{v\}$ 
         $A \leftarrow A \wedge \neg v$ 
```

**Algorithm 2.5:** Algoritmo ALL-SAT.

# 3

## Teorias com Satisfação (SMT's)

### 3.1 Introdução

#### 3.1.1 Objetivo

“Satisfiability Modulo Theories” (SMT’s) é um capítulo da Lógica onde se pretende combinar a abordagem de SAT com teorias da Lógica de Primeira Ordem (FOL) onde seja possível estabelecer *procedimentos de decisão* para a satisfação, pelo menos em algumas classes de fórmulas<sup>†</sup>.

No contexto de uma aplicação das SMT’s, cada uma destas teorias designa-se por *teoria de base* ou *background theory*. Nestas se incluem teorias que lidam com a aritmética dos inteiros e dos reais e ainda algumas das estruturas usuais das ciências da computação: *strings*, *arrays*, árvores, conjuntos, etc.

Dado que por axiomatização, em qualquer destas teorias, não é possível decidir se é ou não satisfazível uma fórmula LPO arbitrária, é necessário restringir a classe de fórmulas de tal modo que seja possível definir um *procedimento de decisão*, específico dessa classe, que seja computável.

O exemplo paradigmático de classe fórmulas que admite um procedimento de decisão computável, é a classe fórmulas não-fechadas, livres de quantificadores.

Recordemos que, na LPO, uma fórmula é *fechada* se todas as suas variáveis ocorrem no contexto de um quantificador. Por exemplo, em  $\forall x \cdot \exists y \cdot (x < y)$  ambas as variáveis estão *ligadas* a quantificadores; uma variável que não está ligada a qualquer quantificador diz-se *livre*. Numa fórmula não fechada existem variáveis livre e, se ela for ainda “livre de quantificadores”, de fato todas as variáveis são livres.

Nas teorias de base que misturem variáveis inteiros  $x, y, \dots$  com variáveis lógicas  $a, b, \dots$ , usando a interpretação *standard* dos símbolos da aritmética inteira, vai ser possível resolver problemas SAT que incluem, por exemplo, algo da forma

$$(b \wedge x \leq y) \vee (\neg a \wedge y \geq x + 2)$$

Este capítulo lida com este tipo de problemas: decidir sobre a satisfação ou não de fórmulas onde se misturam variáveis lógicas com variáveis inteiros/reais e onde, para além das conjetivas proposicionais, ocorrem funções e relações da aritmética.

<sup>†</sup>Para mais detalhes ver **Satisfiability Module Theories**, C. Barret, R. Sebastiani, S. Seshia e C. Tinelli, Capt.26 de *Handbook of Satisfiability*, IOS Press, 2009.

Os problemas da satisfação usam algoritmos que seguem uma de duas abordagens:

- Na abordagem *eager*, procura codificar em fórmulas proposicionais as fórmulas na teoria base, usando para isso informação relevante sobre essa teoria. Em princípio qualquer SAT solver pode analisar a fórmula convertida mas a viabilidade desta método depende da capacidade do “solver” para pre-processar as fórmulas proposicionais muito grandes que codificam a teoria.
- Em alternativa, na abordagem *lazy*, constroem-se procedimentos específicos para implementar cada teoria. Normalmente constroem-se procedimentos específicos apenas para fragmentos da teoria base, e esses “solvers” são incorporados como módulos de um SAT clássico.

Neste capítulo vamos procurar introduzir algumas teorias base e procedimentos de decisão. Antes porém

### 3.1.2 Teorias de Base

A definição de uma teoria de base começa pela definição da sua *assinatura*  $\Sigma$ ; nomeadamente, a identificação dos operadores e a definição da aridade de cada. No estudo das SMT's não são admissíveis quantificadores.

Um exemplo simples é a assinatura determinada pelos operadores

$$0, 1, +, -, \leq, \text{ite} \quad (3.1)$$

em que 0 e 1 são *constantes* (i.e. operadores de aridade 0), os operadores  $+$ ,  $-$  e  $\leq$  são *binários* (i.e. aridade 2) e **ite**, abreviatura de *if then else*, é um operador *ternário* (i.e. aridade 3).

Sintaticamente uma teoria  $\Sigma$  é aumentada com um conjunto  $\mathcal{V}$  de novos símbolos designados por *variáveis*. Os *termos de base*  $\mathcal{T}_{\Sigma \cup \mathcal{V}}$  são as árvores finitas cujas folhas são constantes de  $\Sigma$  ou variáveis em  $\mathcal{V}$  e, se não forem folhas, têm como raíz um operador de  $\Sigma$  com aridade  $n > 0$ , e como sub-árvores  $n$  termos em  $\mathcal{T}_{\Sigma \cup \mathcal{V}}$ .

Exemplos de termos gerados pela assinatura (3.1) e pelas variáveis  $\{x, y, z\}$ ,

$$(x \leq y) + (y \leq z), \quad \text{ite}(x \leq y, x, y), \quad \text{ite}(x, x, y), \quad \text{ite}(x - y, z, 0) \quad (3.2)$$

A definição da teoria completa-se com a sua *semântica*: o estudo dos *modelos* que dão significado aos seus símbolos.

Um *modelo* para a assinatura  $\Sigma$  é um *domínio*  $A$  e, para cada  $n$ -ário operador  $f$ , uma função  $f^A: A^n \rightarrow A$ ;  $f^A$  designa-se por *interpretação* de  $f$ . A noção de interpretação estende-se a termos sem variáveis: a interpretação da árvore  $f(t_1, \dots, t_n)$  em  $\mathcal{T}_{\Sigma}$  é o resultado da aplicação da interpretação de  $f$  à interpretação das sub-árvores  $t_1, \dots, t_n$ .

Apesar da LPO admitir modelos arbitrários, o estudo das SMT's impõe os chamados *modelos standard*. Em todos os modelos assume-se que o domínio  $A$  estende  $\mathbb{Z}$  e que 0 e 1 fazem parte da assinatura. As constantes 0 e 1 são interpretadas pelos elementos  $0, 1 \in A$ , respetivamente; finalmente, cada termo  $t$ , interpretado pelo

inteiro  $t^A$ , é associado ao valor de verdade `false` se  $t^A = 0$  e ao valor de verdade `true` quando  $t^A \neq 0$ .

Nomeadamente, nos *modelos standard da aritmética* o domínio  $A$  é  $\mathbb{Z}$  e os operadores aritméticos  $+$  e  $-$  são interpretados como a adição e subtração em inteiros. A interpretação dos operadores  $=, \neq, \leq, \not\leq, \geq, \not\geq$  é definida pela relação de ordem nos inteiros; é associado o inteiro 1 à verificação da relação ordem e o inteiro 0 à não verificação dessa relação. Por exemplo  $(1 < 0)$  e  $(1 = 0)$  são termos ambos interpretados como o inteiro 0, enquanto que  $(1 \geq 0)$  é interpretado como 1.

Quando à interpretação de `ite` é definida, para todos os termos  $t, r, s$ , por

$$\text{ite}(t, r, s)^A = \begin{cases} r^A & \text{se } t^A \neq 0 \\ s^A & \text{se } t^A = 0 \end{cases}$$

Com esta interpretação, `ite` permite implementar outros operadores; por exemplo o símbolo de Kronecker  $\delta$  é

$$\delta(x) \stackrel{\text{def}}{=} \text{ite}(x, 0, 1)$$

Os operadores igualdade e desigualdade são implementados a partir de  $-$  como

$$x = y \stackrel{\text{def}}{=} \text{ite}(x - y, 0, 1) \quad , \quad x \neq y \stackrel{\text{def}}{=} \text{ite}(x - y, 1, 0)$$

A partir de  $\leq$  e `ite` implementa-se os restantes operadores relacionais

$$x > y \stackrel{\text{def}}{=} \text{ite}(x \leq y, 0, 1) \quad , \quad x \geq y \stackrel{\text{def}}{=} y \leq x \quad , \quad x < y \stackrel{\text{def}}{=} y > x$$

O monómio  $a x$  é muito importante na representação de cortes racionais; tem-se

$$a x \stackrel{\text{def}}{=} \text{ite}(x, a, 0)$$

Finalmente pode-se também implementar as conetivas proposicionais usuais

$$\begin{aligned} x \vee y &\stackrel{\text{def}}{=} \text{ite}(x, 1, y) \quad , \quad x \rightarrow y \stackrel{\text{def}}{=} \text{ite}(x, 1, y, 1) \\ x \wedge y &\stackrel{\text{def}}{=} \text{ite}(x, 1, y, 0) \quad , \quad \neg x \stackrel{\text{def}}{=} \delta(x) \quad , \quad \perp \stackrel{\text{def}}{=} 1 \leq 0 \end{aligned}$$

Podemos concluir que, dentro de um modelo standard da aritmético, a assinatura em (3.1) descreve completamente a aritmética inteira sem quantificadores.

### 3.1.3 Satisfação em Teorias de Base

Vimos como interpretar termos sem variáveis gerados por uma assinatura  $\Sigma$ . Vimos também que, em modelos ditos *standard*, essa interpretação é fixa.

Assim, para tornar possível dar várias interpretações a termos, são necessárias as variáveis. Aumentando a assinatura  $\Sigma$  com um conjunto de variáveis  $\mathcal{V}$  estas passam a comportar-se como constantes na assinatura  $\Sigma \cup \mathcal{V}$ . Os termos gerados por esta nova assinatura  $\Sigma \cup \mathcal{V}$  designam-se, como vimos, por *termos de base*. A sua interpretação vai depender crucialmente da interpretação dada às várias constantes na assinatura aumentada.

Semanticamente existe uma diferença essencial entre as constantes originais em  $\Sigma$  e as novas “constantes” em  $\mathcal{V}$ : as primeiras têm uma interpretação fixa pelo modelo

standard, enquanto que as segundas têm uma interpretação completamente livre dentro do domínio  $A$  do modelo de  $\Sigma$ .

Uma interpretação dos símbolos em  $\mathcal{V}$ , uma vez que têm todos aridade 0, é uma *atribuição*  $v : \mathcal{V} \rightarrow A$ , que associa cada variável  $x$  a um elemento  $x^v \in A$ . Aumentando o modelo standard com a atribuição  $v$  constrói-se a interpretação  $t^v \in A$ , de cada um dos termo de base  $t$ . Assim, escreve-se

$$v \models t \text{ sse } t^v \neq 0 \quad (3.3)$$

Um *contexto de base*  $C$  é um conjunto finito de termos de base; define-se

$$v \models C \text{ sse } t^v \neq 0 \text{ para todo termo } t \in C \quad (3.4)$$

Diz-se que o termo de base  $t$  é *satisfazível* quando existe alguma atribuição  $v$  que valide  $v \models t$ . Do mesmo modo um contexto de base  $C$  é satisfazível quando existe alguma atribuição  $v$  que valide  $v \models C$ .

Uma vez caracterizado o valor de verdade de termos de base é possível definir outras fórmulas e a sua semântica. No contexto das SMT's usamos um fragmento da LPO sem quantificadores, composto pelas seguintes fórmulas.

- *Fórmulas atómicas*: são contextos de base  $C$  ou igualdades  $t_1 = t_2$ ; tem-se

$$v \models (t_1 = t_2) \text{ sse } t_1^v = t_2^v \quad (3.5)$$

- *Literais*: fórmula atómica ou a sua negação; tem-se

$$v \models \neg l \text{ sse } v \not\models l \quad (3.6)$$

- *Cláusulas*: disjunção de literais; tem-se

$$v \models (l_1 \vee \dots \vee l_k) \text{ sse } v \models l_i \text{ para algum } l_i \quad (3.7)$$

- *CNF's*: conjunções de cláusulas.

O problema da satisfação, em SMT's, pode colocar a diversos níveis: na versão mais simples temos o *problema de satisfação de base* que consiste em determinar se um contexto de base é ou não satisfazível e, se o for, apresentar um modelo; na versão SAT procura-se determinar a satisfação de uma CNF.

Normalmente, em SMT “solvers” que sigam a abordagem “lazy”, procura-se definir para cada teoria um mecanismo de decisão específico que conduza a soluções do problema de satisfação de base. O problema SAT de CNF's será resolvido por um SAT “solver” genérico modificado de forma a integrar esse mecanismo de decisão como módulo.

## 3.2 Algoritmos SMT

Neste capítulo só é possível apresentar uma descrição muito esquemática dos algoritmos SMT. Como já referimos estes algoritmos agrupam-se em duas classes: os que procuram uma representação direta da teoria em CNF's que possam ser apresentadas como “input” de SAT's, ou aqueles que procuram desenvolver um

algoritmo específico de cada teoria de base que resolva o problema da satisfação de base e combinam este procedimentos específicos com SAT's genéricos. Aqui vamos limitar-nos a uma breve introdução aos algoritmos do primeiro tipo.

O primeiro procedimento de conversão lida a noção de igualdade. Essencialmente dado um conjunto de literais  $C$  pretende-se construir o seu *fecho por congruência* que resulta do fato de a igualdade de termos de base ser uma relação de congruência.

---

### § 3.1 Regras da congruência.

---

$$\frac{}{t = t} \text{ (rf1)} \quad , \quad \frac{t = r \quad r = s}{t = s} \text{ (trans)} \quad , \quad \frac{t = r}{r = s} \text{ (sim)}$$

$$\frac{t = r}{f(t) = f(r)} \text{ (cong)}$$

Adicionalmente, numa teoria com quantificação sobre termos, tem-se

$$\frac{\bigwedge t \cdot (f(t) = g(t))}{f = g} \text{ (univ1)} \quad , \quad \frac{\bigwedge f \cdot (f(t) = f(r))}{t = r} \text{ (univ2)}$$


---

O objetivo da construção de fechos por congruência é detetar incongruências mesmo sem conhecer a interpretação dos termos.

Suponhamos, por exemplo, que se tem o conjunto de literais  $\{a = b, f(a) \neq f(b)\}$ ; mesmo sem conhecer a interpretação de  $a, b$  ou  $f$ , detetamos imediatamente que este conjunto é incongruente por simples aplicação das regras de congruência.

$$\frac{\frac{a = b}{f(a) = f(b)} \quad f(a) \neq f(b)}{\perp}$$

Explorando esta propriedade o algoritmo de conversão tem de definir heurísticas para escolher as igualdades que deve acrescentar ao conjunto de cláusulas inicial.

O segundo passo de conversão é a substituição de símbolos de função por constantes.

Vamos supor a CNF original contém  $f$  em  $n$  termos:  $f(a_1), f(a_2), \dots, f(a_n)$ . Pode-se eliminar  $f$  substituindo cada termo  $f(a_i)$  por uma constante  $f^i$  e adicionando, para cada par  $i \neq j$ , a seguinte cláusula

$$(a_i \neq a_j) \vee (f^i = f^j)$$

Uma vez eliminados os símbolos de função, o passo seguinte é lidar com a aritmética. Existem várias abordagens mas, neste curso, vamos apenas referir que uma codificação dos inteiros ou reais como na assinatura 3.1 pode ser sempre convertido em contextos formados por termos da forma de cortes racionais

$$a_1 x_1 + a_2 x_2 + \dots \geq b$$

com  $a_i, b$  inteiros e  $x_i$  variáveis inteiiras.

**É frequente usar SMT's dentro do ciclo de desenvolvimento de um sistema  $\mathcal{S}$ .** Para isso modela-se o sistema como uma fórmula  $P$  e usa-se um SMT “solver” para

verificar de  $P$  é satisfazível; se não for satisfazível, isto indica que o sistema  $\mathcal{S}$  tem falhas e deve ser corrigido; uma vez corrigido  $\mathcal{S}$ , constrói-se um novo  $P$  e o processo repete-se até que seja encontradas modelos que validem  $P$ .

Genericamente um SMT “solver”, quando atua sobre a fórmula  $P$ , devolve como resultado uma atribuição  $v$  que constitui a componente não-standard de um modelo que valida esta fórmula; em alternativa devolve o resultado `unsat` que anuncia a não existência de tal modelo.

Neste último caso, e na perspetiva do ciclo de desenvolvimento do sistema  $\mathcal{S}$ , é importante que o “solver” devolva alguma informação que seja útil para a correção de  $P$  ou, idealmente, para a correção de  $\mathcal{S}$ .

Dois tipos de informação lógica podem ser fornecidos: *interpolantes* (ver teorema 2.2, página 32) e “*unsatisfiable cores*”.

Recorde-se que a noção de *interpolante* é relevante quando é possível decompor o vetor de variáveis de  $P$  em três componentes  $x, y, z$  e se pode escrever

$$P(x, y, z) = A(x, y) \wedge B(x, z)$$

Nestas circunstâncias existe uma função booleana  $x \mapsto f(x)$ , o *interpolante*, que depende apenas das variáveis  $x$  e satisfaz a propriedade

$$P \models \perp \text{ sse } \bigwedge_{x, f(x)=1} A(x, \cdot) \models \perp \text{ e } \bigwedge_{x, f(x)=0} B(x, \cdot) \models \perp \quad (3.8)$$

Essencialmente o interpolante faz-nos interpretar uma fórmula  $A(x, y)$  ou  $B(x, z)$  como um conjunto de fórmulas, uma para cada valor de  $x$ , só nas variáveis  $y$  e  $z$ , respetivamente. A satisfação de cada uma destas fórmulas é analisada: as fórmulas  $A(x, \cdot)$  só são analisadas quando  $x \in f^{-1}(1)$  enquanto que as fórmulas  $B(x, \cdot)$  nos restantes casos (i.e. quando  $x \in f^{-1}(0)$ ).

Num sistema formado por várias componentes “quase-independentes”, onde a comunicação entre partes é feita pelas variáveis  $x$ , o interpolante permite inferir qual a componente que deve ser corrigida.

Um *unsatisfiable core* é um subconjunto  $C$  das cláusulas de  $P$  que não é satisfazível sempre que  $P$  não for satisfazível. Dito de outro modo, sempre que  $C$  é satisfazível também  $P$  é satisfazível.

Essencialmente tem-se  $P \equiv C \wedge Q$  com  $C$  e  $Q$  tais que

$$C, Q \models \perp \Rightarrow C \models \perp \quad (3.9)$$

Detetando um “*unsatisfiable core*” de  $P$  pode permitir detetar qual é a componente de  $P$  que deve ser corrigida em caso de falha.

### 3.3 Aplicações

Os SMT “solvers” têm inúmeras aplicações na verificação de sistemas complexos; nomeadamente sistemas de software e sistemas de comando, comunicação e controlo (CCC's). Nesta secção veremos, a título ilustrativo, algumas destas aplicações.

### 3.3.1 Autómatos Híbridos

A *Teoria dos Autómatos Híbridos (AH)*\* é um formalismo para modelar sistemas complexos onde se misturam decisões baseadas em *grandezas físicas* (temperatura, posição, velocidade, concentração, etc) com decisões baseadas em *modos*; as primeiras são modeladas por valores reais enquanto que as segundas são modeladas por valores booleanos.

A aplicação típica destes modelos está na área do controlo digital de sistema de controlo analógico. Os *modos* modelam o estado discreto do controlador enquanto as decisões discretas, designada por *switches*, representam as transições de estado. O estado analógico do sistema é representado por pontos num espaço  $\mathbb{R}^n$  e a sua dinâmica é modelada por “equações de fluxo” que são, quase sempre, equações diferenciais.

O comportamento do sistema analógico depende do estado do controlador; cada modo define condições de fluxo próprias e cada *switch* determina uma transição de modo. Igualmente o comportamento do controlador depende do estado do sistema através de um *invariante*; isto é, uma condição que caso seja violada causa a ativação de um determinado *switch*.

Sucintamente

---

#### § 3.2 Autómato Híbrido.

Um Autómato Híbrido é caracterizado pelas seguintes componentes:

1. Um conjunto finito  $\Sigma$  de *eventos*.
  2. Um conjunto de variáveis reais  $X = \{x_1, x_2, \dots, x_n\}$ . Para cada variável  $x$ , denota-se por  $\dot{x}$  a sua derivada e denota-se por  $x'$  o valor de  $x$  depois de uma transição discreta.  
e.g. A atribuição  $x \leftarrow x + 1$ , representa-se pelo predicado  $x' = x + 1$ . □
  3. Um conjunto finito  $\mathcal{M}$  de *modos*. Cada *modo*  $m$  é caracterizado por três predicados:
    - $\text{init}_m(x)$  e  $\text{inv}_m(x)$  usam só as variáveis  $x$  e estabelecem, respetivamente, as condições iniciais do modo  $m$  e a sua condição invariante cuja eventual violação conduz à ativação de um *switch*.
    - $\text{flow}_m(x, \dot{x})$  um predicado envolvendo variáveis  $x$  e  $\dot{x}$  que determina as condições de fluxo do modo.
  4. Um conjunto de *switches*; cada *switch* é determinado por um *evento*  $e$  e por um predicado  $\text{jump}_e(x, x')$  envolvendo variáveis  $x$  e  $x'$ .
  5. Modos e “switches” determinam um grafo; os modos são os nodos do grafo e os “switches” são os arcos.
- 

**EXEMPLO 3.1:** Considere-se um sistema de controlo de temperatura assente num termóstato que representamos na figura 3.1.

---

\* *Theory of Hybrid Automaton*, Thomas A. Henzinger, 1996, IEEE Computer Society Press.

Essencialmente o sistema três modos, **Start**, **Off** e **On**, e três *switches* associados aos eventos **start**, **turnOn** e **turnOff**.

O *switch start* é ativado com  $x = 20$  e coloca o sistema no modo **Off**.

No modo **Off** a temperatura desce segundo a equação  $\dot{x} = 0.1x$  e um invariante  $x \geq 18$ ; no modo **On** a temperatura sobe de acordo com a condição  $\dot{x} = 5 - 0.1x$  e o invariante é  $x \leq 22$ . Assim que um destes invariantes é violado, um *switch* é ativado.

Os *switches turnOn* e *turnOff* podem ser ativados, sem esperar que os invariantes sejam violados, logo que se verifique  $x < 19$  ou  $x > 21$ .  $\square$

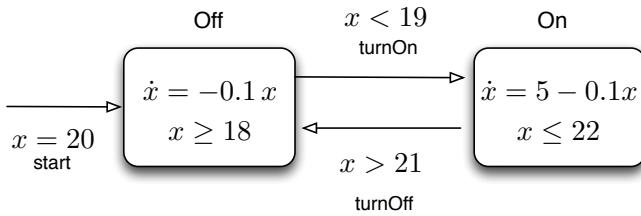


Figura 3.1: Autómato do termostato no exemplo 3.1

O nosso objetivo é a codificação de um AH numa SMT\*; em § 3.3 e § 3.4 apresentam-se um conjunto de regras usadas para codificar um sistema que seja modelado por um único autómato híbrido.

### § 3.3 Codificação de um autómato AH numa SMT – variáveis.

Um autómato AH codifica-se numa SMT declara as seguintes VARIÁVEIS:

1. A variável discreta **ev** toma valores no conjunto de eventos  $\Sigma$ . Predicados ( $\text{ev} = e$ ), com  $e \in \Sigma$ , seleciona  $e$  como o último evento ocorrido.  
O conjunto  $\Sigma$  é aumentado com dois eventos especiais **alive** e **stoped**; o primeiro determina que o SMT executa uma evolução contínua; **dead** que determina que o SMT passa a estar parado.
2. As variáveis discretas **md** e **md'** tomam valores no conjunto dos modos  $\mathcal{M}$ . O predicado ( $\text{md} = m$ ) indica que  $m$  é a localização corrente. O predicado ( $\text{md}' = m$ ) indica que  $m$  é a próxima localização.
3. Cada  $x \in X$  é codificado em duas variáveis reais:  $x$  e  $x'$ . Adicionalmente acrescenta-se uma variável real  $t$ , que representa o tempo, e a sua variante  $t'$  que representa o “próximo tempo”.
4. Sendo *estado* do autómato é um tuplo  $S = \langle x_1, \dots, x_n, t, m \rangle$ , **init**, **inv** e **trans** são representados por fórmulas  $\text{init}(S)$ ,  $\text{inv}(S)$  e  $\text{trans}(S, S', \text{ev})$ .

Seguindo as regras em § 3.3, para codificar numa SMT o autómato híbrido representado na figura 3.1, é necessário começar por declarar tipos e variáveis.

\*O conteúdo desta secção segue *A quantifier free SMT encoding of non-linear hybrid automata*, publicado por A. Cimatti, D. Mover e S. Tonetta em *Proceedings of the 12th Conference on Formal Methods in Computer-Aided Design (FMCAD 2012)*.

- (i) Tipos de dados que representam os domínios dos modos  $\mathcal{M} = \{\text{Start}, \text{Off}, \text{On}\}$  e de eventos  $\Sigma = \{\text{start}, \text{turnOn}, \text{turnOff}, \text{alive}, \text{dead}\}$ .
- (ii) A variável  $\text{ev}$  do tipo  $\Sigma$ , e as variáveis  $\text{md}$  e  $\text{md}'$  do tipo  $\mathcal{M}$ .
- (iii) As variáveis reais  $x$  e  $x'$  e ainda a representação da grandeza “tempo” nas variáveis reais  $t$  e  $t'$ .

As fórmulas que codificam num SMT agrupam-se em três “grandes fórmulas”: **init**, que representa a inicialização, **inv**, que representa os invariantes dos diversos modos, e **trans** que representa o fluxo. Neste caso tem-se

$$(i) \text{ init} \equiv (t = 0) \wedge (\text{md} = \text{Start}) \wedge (x = 20).$$

Note-se que os modos **On** e **Off** não especificam qualquer inicialização.

$$(ii) \text{ inv} \equiv (\text{md} = \text{Off} \rightarrow x \geq 18) \wedge (\text{md} = \text{On} \rightarrow x \leq 22)$$

Numa situação onde o tempo e a variável  $x$  variam, esta condição apenas garante que o invariante se verifica no início da evolução de cada modo. Não garante que o invariante se verifica; essa responsabilidade cai sobre **trans**.

- (iii) A fórmula **trans**, que codifica as transições, tem ela própria três componentes: uma representa a possibilidade de o autómato estar parado, outra representa a possibilidade de existir uma transição discreta de estado e a terceira representa a situação onde há uma evolução contínua.

$$\text{trans} \equiv \text{blocked} \vee \text{untimed} \vee \text{timed}$$

(a) **blocked**  $\equiv (t' = t) \wedge (\text{md}' = \text{md}) \wedge (x' = x)$  denota a situação em que não há evolução de qualquer variável

(b) **untimed** tem uma descrição distinta em cada um dos dois estados; temos

$$\begin{aligned} \text{untimed} \equiv & (t' = t) \wedge \\ & (\text{md} = \text{Off} \rightarrow (\text{md}' = \text{On}) \wedge (\text{ev} = \text{turnOn}) \wedge (x' < 19)) \wedge \\ & (\text{md} = \text{On} \rightarrow (\text{md}' = \text{Off}) \wedge (\text{ev} = \text{turnOff}) \wedge (x' > 21)) \end{aligned}$$

(c) **timed** representa a situação onde a evolução de  $x$  é regido por uma equação diferencial de uma das formas

$$a \dot{x} \bowtie b \quad \text{ou} \quad a \dot{x} + x \bowtie b$$

sendo  $a$  e  $b$  constantes e  $\bowtie$  um dos operadores relacionais  $\{=, \geq, >, \leq, <\}$ .

A codificação deste tipo de equação diferencial numa SMT é a componente mais complexa do processo de codificação e é detalhada em seguida. De momento vamos assumir que temos uma codificação  $\text{dif}_m(x, x', t, t')$  para a equação diferencial do modo  $m$ . Então será

$$\begin{aligned} \text{timed} \equiv & (\text{ev} = \text{alive}) \wedge (\text{md}' = \text{md}) \wedge (t' > t) \wedge \\ & (\text{md} = \text{On} \rightarrow \text{dif}_{\text{On}}) \wedge (\text{md} = \text{Off} \rightarrow \text{dif}_{\text{Off}}) \end{aligned}$$

Para completar este processo de codificação é só necessário analisar o modo como se codifica uma equação diferencial  $a \dot{x} \bowtie b$  ou  $a \dot{x} + x \bowtie b$ .

A forma  $a \dot{x} \bowtie b$  é codificada representando a derivada  $\dot{x}$  por

$$\dot{x} = \frac{\Delta x}{\Delta t} \quad \text{com} \quad \Delta x = x' - x, \Delta t = t' - t$$

Dado que se tem sempre  $t' > t$ , a equação diferencial  $a\dot{x} \leq b$  codifica-se relação

$$a(x' - x) \leq b(t' - t)$$

Note-se que esta relação é “linear”; isto é, se  $a$  e  $b$  forem racionais, a relação tem a forma de um corte racional nas variáveis  $x, x', t$  e  $t'$ .

Se se aplicar exatamente a mesma estratégia à equação  $a\dot{x} + x \leq b$  obtém-se

$$a(x' - x) + x(t' - t) \leq b(t' - t)$$

que já não é uma relação linear: o termo  $x(t' - t)$  contém multiplicações de variáveis, o que impede que a relação possa ser vista como um “corte racional”.

Uma forma de resolver esta dificuldade recorre à combinação entre a equação de fluxo e o invariante; do fato se no estado onde o fluxo é regido por uma equação diferencial  $a\dot{x} + x = b$ , o invariante for da forma  $x \geq c$ , então verifica-se também  $a\dot{x} = b - x \leq b - c$ . Como  $(b - c)$  é uma constante, pode-se usar a situação anterior e codificar o fluxo por

$$\Delta x \leq (b - c)\Delta t$$

Da mesma forma, o mesmo fluxo com um invariante  $x \leq c$  produz uma codificação

$$\Delta x \geq (b - c)\Delta t$$

No nosso exemplo, no modo **Off**, a equação de fluxo é  $10\dot{x} + x = 0$  e o invariante é  $x \geq 18$ ; daqui resulta  $10\dot{x} \leq -18$ ; a codificação será

$$\text{dif}_{\text{Off}} \equiv (x' - x) + 1.8(t' - t) \leq 0$$

No modo **On**, a equação de fluxo é  $10\dot{x} = 50 - x$  e o invariante é  $x \leq 22$ ; portanto tem-se  $10\dot{x} \geq 28$  cuja codificação será

$$\text{dif}_{\text{On}} \equiv (x' - x) - 2.8(t' - t) \geq 0$$

Genericamente a codificação SMT de um autómato híbrido genérico é efectuado por três fórmulas, **init**, **inv** e **trans** como se indica esquematicamente em § 3.3.

### Verificação de Autómatos Híbridos: BMC

Após codificado um autómato híbrido no tuplo de fórmulas,  $\Phi = \langle \text{init}, \text{inv}, \text{trans} \rangle$ , é preciso codificar a forma como estas três fórmulas são usadas para verificar propriedades de um tal sistema.

Representa-se por  $S$  o conjunto de variáveis de estado,  $X \cup \{\text{md}, t\}$ , e por  $S' = X' \cup \{\text{md}', t'\}$  a sua versão após evolução. Vimos que **init** e **inv** têm livres as variáveis  $S$  e que **trans** têm livres as variáveis  $S$  e  $S'$ ; para realçar esse fato vamos passar a escrever  $\text{init}(S)$ ,  $\text{inv}(S)$  e  $\text{trans}(S, S')$ .

A primeira forma de verificação usa a abordagem do *bounded model checking* ou *BMC* (que veremos em mais detalhe no próximo capítulo) para detetar se, após um qualquer número de transições de estado, uma “má propriedade”  $\phi(S)$  ocorre.

Sem perda de generalidade pode-se considerar que o invariante já contém a informação sobre essa má propriedade; assim assume-se que  $\phi$  e **inv** estão relacionados por uma das condições

$$\text{inv}, \phi \models \perp \quad \text{ou} \quad \models \text{inv}, \phi \tag{3.10}$$

---

§ 3.4 Codificação de um autómato AH numa SMT – fórmulas.

Um autómato AH codificado numa SMT define as seguintes fórmulas:

1. **init**: a CNF que representa a inicialização dos diversos modos

$$\text{init} \equiv (t = 0) \wedge \bigwedge_{m \in \mathcal{M}} (\text{md} = m \rightarrow \text{init}_m(x))$$

2. **inv**: a CNF que denota os invariantes dos vários modos

$$\text{inv} \equiv \bigwedge_{m \in \mathcal{M}} (\text{md} = m \rightarrow \text{inv}_m(x))$$

3. **trans**: a fórmula **trans** vai representar o fluxo e a transição e pode ser definido por várias componentes;

$$\begin{aligned} \text{trans} \equiv & \text{parado} \vee \\ & \bigwedge_{m \in \mathcal{M}} (\text{md} = m \rightarrow \text{timed}_m \vee \bigvee_{m',e} \text{untimed}_{m,m',e}) \end{aligned}$$

sendo

- (i) **parado** indica que nenhuma variável muda

$$\text{parado} \equiv (\text{md}' = \text{md}) \wedge \bigwedge_{x \in X} (x' = x)$$

- (ii) **untimed** <sub>$m,m',e$</sub>  representa as transições discretas de localização  $m \rightarrow m'$  por ocorrência de um evento  $e$ .

$$\text{untimed}_{m,m',e} \equiv (\text{ev} = e) \wedge (\text{md}' = m') \wedge (t' = t) \wedge \text{jump}_e(x, x')$$

- (iii) **timed** <sub>$m$</sub>  representa a transição de fluxo dentro do modo  $m$

$$\text{timed}_m \equiv (\text{ev} = \text{alive}) \wedge (\text{md}' = \text{md}) \wedge (t' > t) \wedge \text{FLOW}_m(x, x')$$

sendo  $\text{FLOW}_m(x, x')$  a codificação SMT da equação de fluxo  $\text{flow}_m(x, \dot{x})$ .

---

Seja  $P = \langle S_0, S_1, \dots, S_n \rangle$  a sequêncioa formada por  $n + 1$  cópias das variáveis de estado  $S$ ; represente-se por  $\Phi(P)$  a fórmula

$$\Phi(P) \equiv \text{init}(S_0) \wedge \left( \bigwedge_{k=1}^n \text{inv}(S_{k-1}) \wedge \text{trans}(S_{k-1}, S_k) \right) \quad (3.11)$$

Nestas circunstâncias:

- 3.1 PROPOSIÇÃO Se  $\Phi(P) \wedge \text{inv}(S_n)$  for satisfazível e se for  $\phi, \text{inv} \models \perp$  então existe um caminho com  $N$  transições de estado tal que, em qualquer dos seus estados,  $\phi$  não se verifica.

Se  $\Phi(P) \wedge \neg \text{inv}(S_n)$  for satisfazível e se for  $\models \phi, \text{inv}$  então existe um caminho com  $N$  transições de estado que chega a um estado onde se verifica  $\phi$ .

Usando uma sequêncioa crescente de valores de  $n$  pode-se procurar, por tentativas sucessivas, se a “má propriedade” se verifica em algum estado. Obviamente não é

possível, por este processo, ter a certeza que  $\phi$  nunca se verifica. Para obter essa certeza é necessário recorrer a outras abordagens.

### 3.3.2 Verificação de Programas Imperativos

Uma das aplicações mais importantes dos SMT é a verificação da correção parcial ou total de programas\*.

No âmbito dos programas imperativos a metodologia aqui usada é a de Floyd-Hoare que, essencialmente, vê um programa  $P$  no contexto de condições lógicas: a *pré-condição*  $\phi$  e a *pós-condição*  $\varphi$ . Este triplo, representado como

$$\phi \{P\} \varphi \quad (3.12)$$

designa-se por *triplo de Hoare*; no contexto de um tal triplo,  $P$  é *parcialmente correto* quando sempre que  $P$  termina e  $\phi$  é válido antes da execução do programa, a condição  $\varphi$  é válida após a terminação do programa.

É importante notar que a noção de correção parcial assume que o programa termina. Se adicionalmente for possível garantir que sempre que  $\phi$  é válido antes da execução de  $P$ , este programa termina então o triplo de Hoare diz-se *totalmente correto*. Nesta secção vamos concentrar-nos no problema da correção parcial e adiar o estudo da terminação de programas para a próxima secção.

Representa-se por  $[P]\varphi$  a *pré-condição mais fraca* (**wpc** ou “weakest pre-condition”) que, se for válida num estado imediatamente anterior à execução de  $P$ , garante a validade de  $\varphi$  no estado imediatamente posterior a essa execução. Pode-se interpretar  $[P]$  como um *operador modal* que transforma o argumento  $\varphi$  na mais fraca condição *a priori* que garante a validade de  $\varphi$  *a posteriori* de  $P$ .

Neste contexto, a correção parcial do triplo  $\phi \{P\} \varphi$  descreve-se pela asserção

$$\phi \Rightarrow [P]\varphi$$

ou, equivalentemente, pelas asserções  $\{\phi, \neg[P]\varphi\} \Rightarrow$  ou  $\Rightarrow \{\neg\phi, [P]\varphi\}$ .

#### Nota

No que se segue  $\Rightarrow$  é distinto da conetiva  $\rightarrow$  e denota a implicação semântica: i.e.  $\phi \Rightarrow \varphi$  afirma que todo o modelo que valida  $\phi$  também valida  $\varphi$ . Nomeadamente  $\varphi \Rightarrow \perp$ , ou simplesmente  $\varphi \Rightarrow$ , afirmam que  $\varphi$  não é satisfazível. Da mesma forma  $\Rightarrow \varphi$  é a asserção que afirma ser  $\varphi$  uma tautologia. Estendendo a conjuntos de predicados,  $\Gamma \Rightarrow$  afirma que nenhum modelo satisfaz todos os predicados  $\varphi \in \Gamma$ , enquanto que  $\Rightarrow \Gamma$  afirma que qualquer modelo satisfaz algum desses predicados.  $\square$

A formalização da noção de correção parcial exige uma teoria de base, necessária para definir a pré-condição, a pós-condição e os invariantes, que inclua, no mínimo, a aritmética linear nos inteiros e reais e algumas estruturas de dados como *arrays*, *lists* e *sets*. Nessa teoria é necessário definir um *cálculo de pré-condições* que, para cada programa  $P$  e pós-condição  $\varphi$ , permita calcular  $[P]\varphi$ .

Para os programas vamos definir uma linguagem de *programas anotados* que seja suficientemente completa para incluir modelos apropriados de uma gama vasta de

\*Ver detalhes em *Applications of SMT Solvers to Program Verification*, Nikolaj Bjørner e Leonardo de Moura, SSFT 2014, Maio 2014.

programas nas linguagens imperativas comuns (C<sup>++</sup>, JAVA, etc) mas suficientemente abstrata para permitir um cálculo de pré-condições que seja simples.

Por *programas anotados* entendemos programas que, na sua estrutura de comandos, incluem *anotações* representando *asserções*, *pré-condições*, *pós-condições* e *invariáveis de ciclos*. É importante realçar que as anotações não são meta-informação externa ao programa mas, pelo contrário, fazem parte do próprio programa.

---

### § 3.5 Núcleo da *linguagem de programas anotados* (LPA).

No que se segue  $x$ ,  $t$  e  $\varphi$  denotam uma *variável*, um *termo* e uma *fórmula* genéricos na teoria de base;  $P$  denota um programa anotados genérico;  $S$  e  $R$  denotam comandos genéricos.

(a) Constituem *comandos atómicos*

$$x \leftarrow t \quad | \quad \text{havoc } x \quad | \quad \text{assert } \varphi \quad | \quad \text{assume } \varphi \quad | \quad P \quad (3.13)$$

(b) Constituem-se *comandos compostos*

$$S ; R \quad | \quad S \parallel R \quad (3.14)$$

(c) Um programa anotado tem a forma  $\{ S \}$ , sendo  $S$  um comando.

A semântica destes comandos obedece às seguintes regras:

1. O cálculo de termos  $t$  e fórmulas  $\varphi$  não tem quaisquer efeitos laterais.
2.  $x \leftarrow t$  e **havoc**  $x$  denotam ambas atribuições de valores à variável  $x$ ; a atribuição “tradicional”  $x \leftarrow t$  atribui a  $x$  o valor do termo  $t$ ; a atribuição **havoc**  $x$  associa a  $x$  um valor arbitrário não determinado.

**Nota:** Após a execução do programa  $\{ \text{havoc } x ; y \leftarrow x ; \text{havoc } x \}$  não se pode garantir a igualdade  $(x = y)$ . □

3. Os comandos **assert**  $\varphi$  e **assume**  $\varphi$ , inseridos num programa  $P$ , têm o mesmo significado quando  $\varphi$  tem o valor **true**: em ambos casos o programa  $P$  prossegue sem alteração do estado.

Quando  $\varphi$  tem o valor **false** ambos os comandos abortam  $P$ ; o comando **assert**  $\varphi$  aborta  $P$  num estado de *erro*; ao invés o comando **assume**  $\varphi$  aborta  $P$  num estado de *sucesso*.

4. Cada programa  $P$ , identificado por um nome ou referência, constitui um comando atómico; esta construção pretende representar a noção de procedimento numa linguagem de programação.
5. O comando  $S ; R$  denota a usual *sequência* de comandos: o comando  $R$  continua a ação de  $S$ .
6. O comando  $S \parallel R$  denota *escolha não determinística*: se algum dos candidatos  $S$  ou  $R$  aborta num estado de sucesso, a escolha  $S \parallel R$  é equivalente ao outro comando. Se nenhum dos candidatos aborta, a escolha é não-deterministicamente equivalente a qualquer deles.

---

A correção parcial dos programas é determinada pela pré-condição mais fraca associada a cada um destes comandos.

**Nota:** Em rigor a noção de *weakest precondition* aplica-se a programas; porém, dado que o comando  $S$  determina o programa  $P = \{S\}$ , também se associa  $S$  a  $[P]$ .  $\square$

---

### § 3.6 Pré-condição mais fraca do núcleo da LPA.

---

$$\begin{array}{lll} [x \leftarrow t] \varphi & \equiv & \varphi[t/x] \\ [\text{assert } \phi] \varphi & \equiv & \phi \wedge \varphi \\ [S ; R] \varphi & \equiv & [S][R] \varphi \end{array} \quad \parallel \quad \begin{array}{lll} [\text{havoc } x] \varphi & \equiv & \forall x \cdot \varphi \\ [\text{assume } \phi] \varphi & \equiv & \phi \rightarrow \varphi \\ [S \parallel R] \varphi & \equiv & [S] \varphi \wedge [R] \varphi \end{array} \quad (3.15)$$

#### Justificação

- i) Após a atribuição  $x \leftarrow t$ , a fórmula  $\varphi$  é válida se, antes deste comando, for válida a mesma fórmula com  $x$  substituído por  $t$ .
  - ii) Após a atribuição **havoc**  $x$ , dado que a  $x$  é atribuído um valor arbitário, só se pode garantir a validade de  $\varphi$  se, antes da atribuição, for válido  $\varphi$  com  $x$  substituído por um qualquer valor; isto é, quando for válido  $\forall x \cdot \varphi$ .
  - iii) Após o comando **assert**  $\phi$ , a fórmula  $\varphi$  só é garantidamente válida se  $\phi$  é válido antes e, dado que neste caso não há alteração de estado,  $\varphi$  também for válido antes.
  - iv) Após o comando **assume**  $\phi$ , sendo  $\phi$  válido,  $\varphi$  é válido se e só se for válido antes do comando. Não sendo  $\phi$  válido, não faz sentido falar do valor lógico de  $\varphi$  após a execução do comando; por isso a pré-condição é trivialmente **true**.
- Nomeadamente **assume false** é o elemento neutro de uma escolha não-determinística: nunca é escolhido mas deixa executar a opção alternativa.
- v) O operador  $[S; R]$  transforma  $\varphi$  por ordem inversa dos comandos: primeiro  $[R]$  transforma  $\varphi$  obtendo a pré-condição de  $R$  e depois esta pré-condição é transformada por  $[S]$  obtendo-se finalmente a pré-condição do comando composto.
  - vi) Em  $S \parallel R$  tanto  $S$  como  $R$  podem ser ativados; assim, para poder garantir que  $\varphi$  é válido após a execução do comando composto, é necessário impor uma pré-condição que garanta essa validade qualquer que seja o comando escolhido para executar.

$\square$

---

Como exemplo vamos construir um programa que implemente o algoritmo de Euclides estendido. Para isso

- (i) Assume-se que  $a$  e  $b$  são inteiros positivos não nulos; define-se

$$\text{init}_{a,b} \equiv (a > 0) \wedge (b > 0) \quad (3.16)$$

- (ii) Um triplo de Bézout para o par  $(a, b)$  é um tuplo de inteiros  $\langle x, y, z \rangle$  tais que  $z > 0$  e  $ax + by = z$ . Por conveniência define-se

$$\text{bezout}_{a,b}(x, y, z) \equiv (ax + by = z) \wedge (z > 0) \quad (3.17)$$

- (iii) O algoritmo percorre os triplos de Bézout do par  $(a, b)$  até encontrar aquele que minimiza  $z$ ; esse triplo  $\langle x, y, z \rangle$  é o resultado do algoritmo e tem-se  $z = \text{gcd}(a, b)$ .
- (iv) A correção do algoritmo baseia-se no asserção de que, sendo  $\langle x, y, z \rangle$  e  $\langle x', y', z' \rangle$  triplos de Bézout para  $(a, b)$  e existir um inteiro  $q$  que verifica  $z - z'q > 0$ , então  $\langle x - x'q, y - y'q, z - z'q \rangle$  é um triplo de Bézout para o mesmo par.

$$\begin{aligned} \text{bezout}_{a,b}(x, y, z) \wedge \text{bezout}_{a,b}(x', y', z') \wedge (z - z'q > 0) \Rightarrow \\ \text{bezout}_{a,b}(x - x'q, y - y'q, z - z'q) \end{aligned} \quad (3.18)$$

Baseia-se também em duas asserções referentes a dois triplos particulares

$$\Rightarrow \text{bezout}_{a,b}(1, 0, a) \quad , \quad \Rightarrow \text{bezout}_{a,b}(0, 1, b) \quad (3.19)$$

- (v) Os modelos na teoria de base deste algoritmo, nomeadamente nas asserções (3.17) e (3.18), são atribuições de valores inteiros às 9 variáveis

$$\langle x, y, z, x', y', z', a, b, q \rangle$$

Uma implementação deste algoritmo é apresentada em **algoritmo 3.1**.

```

1: begin
2:    $\langle x, y, z \rangle, \langle x', y', z' \rangle \leftarrow \langle 1, 0, a \rangle, \langle 0, 1, b \rangle$ 
3:   while  $z \neq z'$  do
4:     if  $z > z'$  then
5:        $q \leftarrow z/z'; \langle x, y, z \rangle \leftarrow \langle x - q x', y - q y', z - z' q \rangle$ 
6:     else
7:        $q \leftarrow z'/z; \langle x', y', z' \rangle \leftarrow \langle x' - q x, y' - q y, z' - z q \rangle$ 
8:   exit
```

*Algoritmo 3.1: Euclides.*

#### Observação

É importante notar que este algoritmo pode alcançar um estado em que o ciclo nunca termina; nomeadamente num estado onde  $z$  seja múltiplo de  $z'$  (ou  $z'$  seja múltiplo de  $z$ ) tem-se  $z - q z' = 0$  (ou, respetivamente,  $z' - q z = 0$ ), após a atribuição tem-se  $z = 0$  (ou  $z' = 0$ ) o que faz com que os valores de  $z$  e  $z'$  nunca sejam alterados em ciclos futuros.  $\square$

Para além de comandos não-ativos **begin** e **exit**, usados apenas como indicadores dos estados inicial e final do algoritmo, o resto da implementação usa uma linguagem não anotada *standard*. Esta linguagem está próxima das linguagens de programação usuais, é compacta e reflete completamente a progressão do estado; nomeadamente permite analisar as questões de terminação e complexidade do algoritmo.

Em contrapartida não permite analisar facilmente a noção de correção, muito dificultada neste exemplo pela presença do ciclo **while**. Para análise da correção é necessário incluir pré e pós condições e, nomeadamente, libertar o programa da sua estrutura cíclica. Para isso necessitamos da linguagem anotada.

Como ilustração vamos construir um algoritmo em LPA equivalente ao algoritmo anterior e vamos estruturar a prova da sua correção parcial. Por simplicidade abstrámos a construção “sequência de comandos” e vamos considerar implícito, entre cada duas linhas, o operador “;”.

O algoritmo é livre de ciclos e usa o seguinte invariante

$$\text{inv}_{a,b} \equiv \text{bezout}_{a,b}(x, y, z) \wedge \text{bezout}_{a,b}(x', y', z') \quad (3.20)$$

Adicionalmente tem definida uma pós-condição arbitrária  $\text{post}_{a,b}$ .

O programa anotado 3.2 está organizado em sub-programas cada um dos quais é identificado com uma marca  $\ell i$ . Este programa  $P$  pode ser escrito como

$$P = \ell 1 ; \ell 2 ; \ell 3 ; \ell 7 \quad \text{com} \quad \ell 3 = \{\ell 4 \parallel \ell 5 \parallel \ell 6\}$$

A função do ciclo **while**, essencial ao algoritmo 3.1, é aqui exercida por

$\ell2 : \{\text{assert } \text{inv}_{a,b} ; \text{havoc } x, y, z, x', y', z', q ; \text{assume } \text{inv}_{a,b}\}$

Este comando gera valores arbitrários para as variáveis do programa, desde que validem o invariante do algoritmo; com estes valores o corpo do ciclo é executado; como os valores são arbitrários o corpo do ciclo é testado com qualquer valor possível.

```

ℓ1:  {
      assume inita,b
       $x, y, z, x', y', z' \leftarrow 1, 0, a, 0, 1, b$ 
    }

ℓ2:  {
      assert inva,b
      havoc x, y, z, x', y', z', q
      assume inva,b
    }

ℓ3:  {
    }

ℓ4:  {
      assume  $(z > q z') \wedge (q > 0)$ 
       $\langle x, y, z \rangle \leftarrow \langle x, y, z \rangle - q \langle x', y', z' \rangle$ 
      assert inva,b
      assume false
    }

ℓ5:  {
      assume  $(z' > q z) \wedge (q > 0)$ 
       $\langle x', y', z' \rangle \leftarrow \langle x', y', z' \rangle - q \langle x, y, z \rangle$ 
      assert inva,b
      assume false
    }

ℓ6:  assume  $z = z'$ 

ℓ7:  {
      assert posta,b
      assume false
    }
  
```

Programa Anotado 3.2: Algoritmo de Euclides.

A correção parcial prova-se começando por associar, a cada um dos subprogramas  $\ell_i$ , um transformador de predicados  $[\ell_i]$  tal que, para toda pós-condição  $\varphi$ ,  $[\ell_i]\varphi$  é a pré-condição mais fraca correspondente.

1.  $\ell1 : \{\text{assume init}_{a,b} ; \langle x, y, z, x', y', z' \rangle \leftarrow \langle 1, 0, a, 0, 1, b \rangle\}$

O comando  $\ell1$  inicializa o programa; atendendo a § 3.6 tem-se, para todo  $\varphi$ ,

$$[\ell1]\varphi \equiv \text{init}_{a,b} \rightarrow \varphi[1, 0, a, 0, 1, b/x, y, z, x', y', z'] \quad (3.21)$$

2.  $\ell2 : \{\text{assert } \text{inv}_{a,b} ; \text{havoc } x, y, z, x', y', z', q ; \text{assume } \text{inv}_{a,b}\}$

Tem-se

$$[\ell2]\varphi \equiv \text{inv}_{a,b} \wedge (\forall x, y, z, x', y', z', q \cdot \text{inv}_{a,b} \rightarrow \varphi) \quad (3.22)$$

É necessário ter em atenção que as variáveis com a mesma designação nas duas instâncias de  $\text{inv}_{a,b}$  são interpretadas de forma diferente: na primeira instância as variáveis são livres enquanto que na segunda instância estão ligadas aos quantificadores  $\forall$ . Assim as substituições de variáveis por constantes só é feita na primeira destas instâncias.

Nomeadamente o transformador  $[\ell_1; \ell_2]$  produz

$$[\ell_1; \ell_2] \varphi \equiv (\text{init}_{a,b} \rightarrow \text{inv}_{a,b}[1, 0, a, 0, 1, b/x, y, z, x', y', z']) \wedge \\ \wedge (\text{init}_{a,b} \rightarrow (\forall x, y, z, x', y', z', q \cdot \text{inv}_{a,b} \rightarrow \varphi)) \quad (3.23)$$

Note-se que esta fórmula não contém qualquer uma de  $x, y, z, x', y', z', q$  como variáveis livres; as únicas variáveis livres são  $a$  e  $b$ . Obviamente admite-se a possibilidade de  $\varphi$  conter as variáveis  $a$  e  $b$

Usando SMT prova-se que  $(\text{init}_{a,b} \rightarrow \text{inv}_{a,b}[1, 0, a, 0, 1, b/x, y, z, x', y', z'])$  é uma tautologia. Por isso, tomando modelos apenas nos conjuntos de variáveis  $x, y, z, x', y', z', q$  pode-se simplificar (3.23) e escrever

$$[\ell_1; \ell_2] \varphi \equiv \text{init}_{a,b} \rightarrow (\text{inv}_{a,b} \Rightarrow \varphi) \quad (3.24)$$

3. Tem-se  $\ell_3 = \{\ell_4 \parallel \ell_5 \parallel \ell_6\}$ ; portanto  $[\ell_3] \varphi = [\ell_4] \varphi \wedge [\ell_5] \varphi \wedge [\ell_6] \varphi$ .

Combinando com (3.24) pode-se escrever

$$[\ell_1; \ell_2; \ell_3] \varphi \equiv \text{init}_{a,b} \rightarrow (\text{inv}_{a,b} \Rightarrow [\ell_4] \varphi \wedge [\ell_5] \varphi \wedge [\ell_6] \varphi) \quad (3.25)$$

4. Antes de desenvolver  $[\ell_4], [\ell_5]$  e  $[\ell_6]$ , vamos considerar o comando

$$\ell_7 : \{ \text{assert post}_{a,b}; \text{assume false} \}$$

Temos

$$[\ell_7] \varphi = \text{post}_{a,b} \wedge (\text{false} \rightarrow \varphi) \equiv \text{post}_{a,b}$$

que é independente de  $\varphi$ .

Genericamente qualquer comando  $\{ \text{assert } \phi; \text{assume false} \}$  verifica, independentemente de  $\varphi$ ,

$$[\text{assert } \phi; \text{assume false}] \varphi \equiv \phi \quad (3.26)$$

Pode-se pois concluir que, independentemente de  $\varphi$ ,

$$[\ell_1; \ell_2; \ell_3; \ell_7] \varphi \equiv \text{init}_{a,b} \rightarrow \\ (\text{inv}_{a,b} \Rightarrow [\ell_4] \text{post}_{a,b} \wedge [\ell_5] \text{post}_{a,b} \wedge [\ell_6] \text{post}_{a,b}) \quad (3.27)$$

5. Em  $\ell_4$  e  $\ell_5$  tem-se sempre, como em (3.26),

$$[\text{assert inv}_{a,b}; \text{assume false}] \text{post}_{a,b} \equiv \text{inv}_{a,b}$$

Portanto

$$[\ell_4] \text{post}_{a,b} \equiv (z > qz') \wedge (q > 0) \rightarrow \\ \text{inv}_{a,b}[x - qx', y - qy', z - qz'/x, y, z] \quad (3.28)$$

$$[\ell_5] \text{post}_{a,b} \equiv (z' > qz) \wedge (q > 0) \rightarrow \\ \text{inv}_{a,b}[x' - qx, y' - qy, z' - qz/x', y', z'] \quad (3.29)$$

6. Em  $\ell 6$  tem-se simplesmente

$$[\ell 6] \text{post}_{a,b} \equiv (z = z') \rightarrow \text{post}_{a,b} \quad (3.30)$$

7. Finalmente, substituindo (3.28),(3.29) e (3.30) em (3.27), obtém-se a fórmula que determina a correção parcial do programa.

$$\begin{aligned} & \text{init}_{a,b} \rightarrow \text{inv}_{a,b} \Rightarrow \\ & \left\{ \begin{array}{l} ((z > qz') \wedge (q > 0)) \rightarrow \text{inv}_{a,b}[x - qx', y - qy', z - qz'/x, y, z] \\ ((z' > qz) \wedge (q > 0)) \rightarrow \text{inv}_{a,b}[x' - qx, y' - qy, z' - qz/x', y', z'] \\ ((z = z') \rightarrow \text{post}_{a,b}) \end{array} \right. \wedge \end{aligned} \quad (3.31)$$

Pode-se agora usar um sistema de verificação de SMT's para testar se esta fórmula é uma tautologia.

### 3.4 Sistemas de Transição de 1ª Ordem

Pode-se interpretar “First Order Transition Systems” (FOTS) como sistemas de transição de estados com um número infinito de estados que são descritos usando fórmulas em Lógica de 1ª Ordem.

Formalmente um FOTS  $\Sigma = \langle \mathcal{X}, \mathcal{X}', \text{init}, \text{inv}, \text{trans}, \text{exit} \rangle$  é caracterizado pelos seguintes elementos:

- (i) Um conjunto de variáveis  $\mathcal{X}$  que tomam valores nos domínios de teorias base SMT\*; uma réplica  $\mathcal{X}'$  das variáveis “após a transição”. Os *estados*  $v$  são atribuições parciais de valores a variáveis.
- (ii) Duas fórmulas  $\text{init}(x)$  e  $\text{inv}(x)$ , em LPO, que descrevem o estado inicial e o invariante de estado. Sem perda de generalidade pode-se considerar sempre que  $\text{init} \Rightarrow \text{inv}$  ocorre.
- (iii) Uma fórmula  $\text{trans}(x, x')$  em LPO que descreve a relação entre o estado antes e depois da transição.
- (iv) Uma fórmula  $\text{exit}(x)$  que caracteriza os *estados terminais* do sistema de transição; isto é, os estados onde um caminho termina.

Ao contrário dos Autómatos Híbridos, onde as fórmulas  $\text{init}$ ,  $\text{inv}$  e  $\text{trans}$  estão distribuídas pelos vários modos, aqui existe apenas um exemplar de cada uma destas fórmulas. Essencialmente a codificação de um autómato híbrido em SMT que descrevemos na secção 3.3.1, foi um processo de codificação num FOTS.

A partir deste modelo é possível usar SMT para tentar provar propriedades que ocorrem ao longo de sequências de estados. As transições legítimas (i.e. que preservam o invariante e respeitam a iniciaização) num caminho de comprimento  $n$  são

---

\*Obviamente que seria possível definir FOTS sem exigir que o domínio das variáveis fosse um domínio de uma teoria base SMT. No entanto, como se pretende sempre codificar um FOTS num SMT, não faz sentido considerar essa hipótese.

representadas pela seguinte fórmula nas variáveis  $x_0, x_1, \dots, x_n$ .

$$\Phi(x_0, \dots, x_n) \equiv \left( \bigwedge_{k=0}^{n-1} \text{trans}(x_k, x_{k+1}) \wedge \text{inv}(x_k) \wedge \neg\text{exit}(x_k) \right) \wedge \text{inv}(x_n) \quad (3.32)$$

Neste contexto pode-se considerar alguns problemas paradigmáticos

- (i) *Existe de um n-caminho legítimo que valida sempre a propriedade P ?*

Quere-se provar que existe um caminho, com  $N$  transições de estado, ao longo do qual é sempre válida uma propriedade  $P$ . Para isso prova-se a validade de

$$\text{inv} \Rightarrow P \quad \text{ou} \quad \forall x \cdot \text{inv}(x) \rightarrow P(x) \quad (3.33)$$

Essencialmente, nos estados onde  $\text{inv}$  é válido,  $P$  também é válido. Em seguida temos de encontrar estados  $s_0, s_1, \dots, s_n$  que validem a fórmula de transições  $\Phi$ . Isto é, é necessário provar a validade de

$$\exists x_0, \dots, x_n \cdot \Phi(x_0, \dots, x_n) \quad (3.34)$$

Recorrendo a um SMT, prova-se (3.33) mostrando que  $\text{inv} \rightarrow P$  é uma tautologia; alternativamente prova-se que a sua negação  $\neg\text{inv} \vee \neg P$  é insatisfazível.

Analogamente encontram-se os estados  $\langle s_0, \dots, s_N \rangle$  verificando que  $\Phi$  é satisfazível e tomando, como solução do problema, um qualquer modelo de  $\Phi$ .

- (ii) *Todos os n-caminhos legítimos terminam num estado que não valida uma propriedade de erro E ?*

Este problema representa-se pela validade de

$$\forall x_0, \dots, x_n \cdot \Phi(x_0, \dots, x_n) \rightarrow \neg E(x_n) \quad (3.35)$$

ou, equivalentemente, pela não validade de

$$\exists x_0, \dots, x_n \cdot \Phi(x_0, \dots, x_n) \wedge E(x_n)$$

Desta forma pode-se usar um SMT para responder a esta questão: a resposta é SIM se e só se  $\Phi(x_0, \dots, x_n) \wedge E(x_n)$  não é satisfazível.

- (iii) *Nenhum estado de um n-caminho legítimo valida sempre a propriedade S ?*

Esta é uma variante da questão (i). Pretende-se provar a validade de

$$\forall x_0, \dots, x_n \cdot \Phi(x_0, \dots, x_n) \rightarrow \neg \bigwedge_{k=0}^n S(x_k)$$

ou, de forma equivalente, provar à não validade de

$$\exists x_0, \dots, x_n \cdot \Phi(x_0, \dots, x_n) \wedge \bigwedge_{k=0}^n S(x_k) \quad (3.36)$$

Note-se que, devido à forma de  $\Phi$ , se considerarmos  $\text{inv}$  como uma variável de “ordem superior” (uma variável que toma valores nas fórmulas de 1ª ordem), então (3.37) é equivalente a  $\Phi$  substituindo a fórmula  $\text{inv}$  pela fórmula  $\text{inv} \wedge S$ . Ou seja, a questão tem resposta afirmativa se e só se não é válida a fórmula

$$\exists x_0, \dots, x_n \cdot \Phi[(\text{inv} \wedge S)/\text{inv}] \quad (3.37)$$

ou, equivalentemente, se e só se  $\Phi[(\text{inv} \wedge S)/\text{inv}]$  não é satisfazível.

Vamos supor que os caminhos se vão tornando cada vez maiores até se tornarem numa sequência infinita de estados. Vamos considerar *paths*  $s = \langle s_0, s_1, \dots, s_k, \dots \rangle$  e enunciar questões do tipo: *verificar se em todos os “paths”  $s = \langle s_0, s_1, \dots, s_k, \dots \rangle$  cujo estado inicial  $s_0$  valida a fórmula  $I$ , verifica-se a propriedade de segurança  $S$  em todos os estados.*

Muitas vezes, quando se lida com “paths”, engloba-se o invariante com a relação de transição e vê-se a conjunção  $\text{inv}(x) \wedge \text{trans}(x, x')$  como uma nova relação de transição  $\text{trans}'(x, x')$ . Assim, no contexto de um FOTS  $\Sigma = \langle V, \text{init}, \text{trans} \rangle$ , um “path” é uma sequência infinita de estados  $s = \langle s_0, s_1, \dots, s_k, \dots \rangle$  que verifica

$$\text{init}(s_0) \wedge (\forall k > 0 \cdot \text{trans}(s_{k-1}, s_k)) \quad (3.38)$$

Representamos pelo mesmo símbolo  $\Sigma$  o conjunto dos “paths” no FOTS  $\Sigma$ . Para formalizar o conceito de “segurança” faz sentido definir uma fórmula de “ordem superior” que tem como variável  $P$ , instanciada no domínio das fórmulas de 1ª ordem e é quantificada por “paths” em  $\Sigma$ .

$$\text{safe}_\Sigma(P) \doteq \forall s \in \Sigma \cdot P(s_0) \rightarrow (\forall n \cdot P(s_n)) \quad (3.39)$$

A definição de segurança diz-nos que  $\Sigma$  é seguro em relação a  $P$  quando todo o traço em que estado inicial valida  $P$ , todos os restantes estados também validam  $P$ . Esta noção pode ser generalizada para  $k$ -segurança em que a pré-condição propõe que a propriedade seja válida nos primeiros  $k$  estados. Formalmente, para  $k > 0$

$$k\text{-safe}_\Sigma(P) \doteq \forall s \in \Sigma \cdot (\forall n < k \cdot P(s_n)) \rightarrow (\forall n \cdot P(s_n)) \quad (3.40)$$

O problema que impede usar as definições de segurança (3.39) e (3.40) diretamente está na quantificação  $\forall s \in \Sigma$  que, interpretada à letra, significaria que todos os “paths” em  $\Sigma$  deveriam ser testados; dado que o número e “paths” é infinito este tipo de teste exaustivo é impossível. A solução está no uso da indução sobre os inteiros e, mais abstratamente, sobre os “paths”.

### Indução sobre “paths” e propriedades de segurança

Recorde-se que, quando se quer provar uma propriedade do tipo  $\forall n \cdot P(n)$  a prova por indução prova primeiro que  $P$  é válido quando  $n = 0$  e depois assume um qualquer  $n$  e que  $P$  é válido em  $n$ ; dessas assunções prova que  $P$  é válida em  $n + 1$ . Essencialmente a prova por indução usa como axioma a fórmula

$$P(0) \wedge (\forall n \cdot P(n) \rightarrow P(n + 1)) \rightarrow (\forall n \cdot P(n))$$

Usando este princípio nos estados do “path”  $s \in \Sigma$  e reescrevendo o axioma

$$(\forall n \cdot P(s_n) \rightarrow P(s_{n+1})) \rightarrow (P(s_0) \rightarrow \forall n \cdot P(s_n)) \quad (3.41)$$

O estado  $s_n$  relaciona-se com  $s_{n+1}$  verificando  $\text{trans}(s_n, s_{n+1})$ ; assim pode-se substituir o antecedente  $(\forall n \cdot P(s_n) \rightarrow P(s_{n+1}))$  pela condição mais forte

$$\forall x, x' \cdot \text{trans}(x, x') \wedge P(x) \rightarrow P(x')$$

Agora a quantificação é feita sobre todos os  $x, x'$  e não apenas sobre os valores dos  $s_n$ . Como é independente do “path”  $s$  pode-se escrever

$$\begin{aligned} (\forall x, x' \cdot \text{trans}(x, x') \wedge P(x) \rightarrow P(x')) &\Rightarrow \\ &\Rightarrow (\forall s \in \Sigma \cdot P(s_0) \rightarrow \forall n \cdot P(s_n)) \equiv \text{safe}_\Sigma(P) \end{aligned} \quad (3.42)$$

Acabámos de provar

**3.2 PROPOSIÇÃO (PRINCÍPIO DA INDUÇÃO)** A fórmula  $\text{safe}_\Sigma(P)$  é válida se verificar  $(\forall x, x' \cdot \text{trans}(x, x') \wedge P(x) \rightarrow P(x'))$ .

Vamos supor que  $\text{safe}_\Sigma(P) \equiv \forall s \in \Sigma \cdot P(s_0) \rightarrow (\forall n \cdot P(s_n))$  é válida. Pretende-se agora assegurar que  $P$  é válida em todos os estados de um “path” arbitrário  $s \in \Sigma$ ; i.e.

$$\forall s \in \Sigma \cdot \forall n \cdot P(s_n) \quad (3.43)$$

Atendendo à definição de  $\text{safe}_\Sigma(P)$ , basta provar que  $P$  é válido em todos os estados iniciais  $s_0$  de todos os “paths”  $s \in \Sigma$ . Como os estados iniciais são aqueles que validam a condição `init`, pode-se concluir

**3.1 COROLÁRIO** A propriedade  $P$  é **universal** em  $\Sigma$ , no sentido que valida (3.43), se  $\text{init}(x) \rightarrow P(x)$  e  $\text{trans}(x, x') \wedge P(x) \rightarrow P(x')$  forem ambas tautologias.

Para usar este resultado numa codificação num SMT é necessário converter tautologias em fórmulas não satisfazíveis; basta mostrar que não são satisfazíveis

$$\begin{cases} \text{init}(x) \wedge \neg P(x) \\ \text{trans}(x, x') \wedge P(x) \wedge \neg P(x') \end{cases} \quad (3.44)$$

Considere-se o seguinte procedimento, onde  $a > 0$  denota uma constante inteira não determinada.

```

1:   begin
      x ← a
2:   while x > 0 do
      x ← x - 1
3:   exit

```

Algoritmo 3.3: Um simples ciclo.

Pretende-se construir um FOTS que descreva a semântica operacional deste algoritmo de forma a ser possível exprimir as suas condições de terminação.

(i) *Variáveis e Estado*

Vamos usar um conjunto  $X = \{\ell, x\}$  com duas variáveis que tomam valores sobre os inteiros. A variável  $\ell$  toma valores na gama  $\{1..3\}$  e o seu valor, num determinado estado, seleciona a linha de código que é executada em seguida. Nesse estado, o valor da variável  $x$  é o conteúdo de  $x$  no algoritmo. Assim os estados neste sistema são os pares  $(\ell, x) \in \{1..3\} \times \mathbb{Z}$ .

- (ii) Para inicialização consideramos o `init`  $\equiv (\ell = 1)$ ; como invariante vamos considerar simplesmente `inv`  $\equiv (\ell \in \{1, 2, 3\})$ . A relação de transição pode-se escrever como uma disjunção de cláusulas conjuntivas.

$$\text{trans}(\ell, x, \ell', x') \equiv \begin{cases} (\ell = 1) \wedge (\ell' = 2) \wedge (x' > 0) & \vee \\ (\ell = 2) \wedge (\ell' = 2) \wedge (x > 0) \wedge (x > x') & \vee \\ (\ell = 2) \wedge (\ell' = 3) \wedge (x \leq 0) \wedge (x' = x) & \end{cases} \quad (3.45)$$

As técnicas de verificação da correção parcial permitem aferir se (3.45) é uma relação de transição correta para o algoritmo 3.3. Para analisar as condições de terminação do algoritmo pode-se começar por provar que o invariante  $\text{inv}(\ell) \equiv (\ell \in \{1, 2, 3\})$  é universal para esta relação de transição.

Para tal prova-se que  $\text{init}(\ell) \rightarrow \text{inv}(\ell)$  e  $\text{trans}(\ell, x, \ell', x') \wedge \text{inv}(\ell) \rightarrow \text{inv}(\ell')$  são tautologias. Usando um verificador SMT é simples mostrar que a negação destas fórmulas, respectivamente  $\text{init}(\ell) \wedge \neg \text{inv}(\ell)$  e  $\text{trans}(\ell, x, \ell', x') \wedge \text{inv}(\ell) \wedge \neg \text{inv}(\ell')$ , não são satisfazíveis.

Figura 3.2: FOTS para algoritmo 3.3

```

Labels, (var, place) = EnumSort('Labels', ['var', 'place'])
Estado = ArraySort(Labels, IntSort())

s = Const('s', Estado)
w = Const('w', Estado)

def inv(s):
    l = s[place]
    return Or(l == 1, l == 2, l == 3)

def init(s):
    return (s[place] == 1)

def trans(u,v):
    x = u[var] ; l = u[place] ; x_ = v[var] ; l_ = v[place]
    return Or(
        And(l == 1, l_ == 2, x_ > 0),
        And(l == 2, l_ == 2, x > 0, x_ < x),
        And(l == 2, l_ == 3, x <= 0, x_ == x)
    )

def non_safe(P):
    return And(trans(s,w), P(s), Not(P(w)))

def non_rooted(P):
    return And(init(s), Not(P(s)))

def non_universal(P):
    return Or(non_rooted(P), non_safe(P))

def test(P):
    solver = Solver()
    solver.add(P)
    try:
        print(solver.check())
        print(solver.model())
    except Exception:
        pass

test(non_universal(inv))

```

### Propriedades de animação

Essencialmente *uma propriedade de animação* pode ser caracterizada por um conjunto de estados terminais, definido por um predicado **exit**, e verificar tal propriedade é equivalente à afirmação de que um caminho atinge um destes estados terminais. Formalmente **exit** é tal que é válido

$$\forall s \in \Sigma \cdot \exists n \cdot \mathbf{exit}(s_n) \quad (3.46)$$

Equivalentemente pode-se formalizar a terminação, afirmando que

$$\forall n \cdot \neg \mathbf{exit}(s_n) \quad (3.47)$$

não é satisfazível por nenhum caminho  $s \in \Sigma$ .

Uma estratégia usual de implementar este tipo de condições recorre ao axioma que determina a ordem total nos inteiros. Nomeadamente,

$$(\forall n \cdot f(n+1) < f(n)) \wedge (\forall n \cdot f(n) > 0) \quad (3.48)$$

não é satisfazível para nenhuma função  $f : \mathbb{N} \rightarrow \mathbb{Z}$ .

Para aplicar este axioma à terminação de um programa ou, mais genericamente, à terminação de um FOTS vamos definir uma função do estado  $F(s)$  com valores inteiros e, para todo  $s \in \Sigma$ , a função inteira  $f(n) \stackrel{\text{def}}{=} F(s_n)$ .

Vamos supor que  $F(s)$  é escolhido de forma a que sejam tautologias

$$(\mathbf{exit}(s) \vee F(s) > 0) \quad , \quad \mathbf{trans}(s, s') \rightarrow (\mathbf{exit}(s) \vee F(s) > F(s')) \quad (3.49)$$

Então é tautologia

$$(s \in \Sigma) \rightarrow \forall n \cdot \mathbf{exit}(s_n) \vee (F(s_n) > 0)$$

e também é tautologia

$$(s \in \Sigma) \rightarrow \forall n \cdot \mathbf{exit}(s_n) \vee (F(s_n) > F(s_{n+1}))$$

Se eventualmente (3.47) fosse satisfazível para algum  $s \in \Sigma$ , então, fazendo

$$f(n) = F(s_n)$$

a validade das duas fórmulas anteriores faz com que sejam válidos simultaneamente  $(\forall n \cdot f(n) > 0)$  e  $(\forall n \cdot f(n) > f(n+1))$  e, consequentemente, (3.48). Como a ordem dos inteiros nos diz que (3.48) nunca é satisfazível, por redução ao absurdo temos de concluir que (3.47) não é satisfazível e, por isso, todo o caminho termina.

Em conclusão,

**3.3 PROPOSIÇÃO** *Dado um FOTS  $\Sigma$ , dada uma função inteira do estado  $F(s)$  e um predicado **exit**( $s$ ) que caracteriza os estados terminais, se forem tautologias*

$$(F(s) > 0 \vee \mathbf{exit}(s)) \quad , \quad \mathbf{trans}(s, s') \rightarrow (F(s) > F(s') \vee \mathbf{exit}(s))$$

*então verifica-se  $\forall s \in \Sigma \cdot \exists n \cdot \mathbf{exit}(s_n)$ .*

Por vezes não é possível provar que  $\text{trans}(s, s') \rightarrow (F(s) > F(s') \vee \text{exit}(s))$  é uma tautologia. Por exemplo, aplicando a metodologia expressa na proposição 3.3 com os elementos

$$\text{exit}(\ell, x) \equiv (x \leq 0) \quad , \quad F(\ell, x) = x \quad (3.50)$$

vemos esta fórmula não é uma tautologia porque, no estado onde  $\ell = 1$ , transita-se para o estado onde  $\ell = 2$  sem que haja uma alteração do valor da variável  $x$  e, consequentemente, de  $F(s)$ .

Nestas circunstâncias a metodologia pode ser generalizada relaxando a pré-condição na segunda tautologia. Para tal estende-se a relação  $\text{trans}(s, s')$  de forma a poder representar, em vez de uma só transição, exatamente  $k$  transições (com  $k \geq 1$ ). Assim define-se

$$\begin{aligned} k\text{-}\text{trans}(s, s') \equiv \\ \exists x_0, \dots, x_k \cdot (s = x_0) \wedge \left( \bigwedge_{i=0}^{k-1} \text{trans}(x_i, x_{i+1}) \right) \wedge (s' = x_k) \end{aligned} \quad (3.51)$$

Com esta definição é simples generalizar a proposição 3.3 e obter

3.4 PROPOSIÇÃO *Dado um FOTS  $\Sigma$ , dada uma função inteira  $F(s)$  e um predicado  $\text{exit}(s)$ , se for tautologia  $(F(s) > 0 \vee \text{exit}(s))$  e, para algum  $k > 0$ , for tautologia*

$$k\text{-}\text{trans}(s, s') \rightarrow (F(s) > F(s') \vee \text{exit}(s))$$

*então verifica-se  $\forall s \in \Sigma \cdot \exists n \cdot \text{exit}(s_n)$ .*

Voltando ao exemplo do algoritmo 3.3, usando os elementos definidos em (3.50), usamos a seguinte codificação com todas as fórmula apresentadas na forma negativa.

Figura 3.3: FOTS do algoritmo 3.3 – continuação

```
def next(u):
    return (u[var] > 0)

def f(u):
    return u[var]

def trans2(u,v):
    x = Const('_', Estado)
    return And(trans(u,x) , trans(x,v))

def not_anim0(F):
    return And( F(s) <= 0 , next(s))

def not_anim1(F):
    return And(trans(s,w) , F(s) <= F(w) , next(s))

def not_anim2(F):
    return And(trans2(s,w) , F(s) <= F(w) , next(s))
```

Invocando `test(not_anim0(f))` ou `test(not_anim2(f))`, a resposta é `unsat` em ambos os casos; isto indica que a negação de cada uma destas fórmulas é uma tautologia. Porém se invocarmos `test(not_anim1(f))`, a resposta é `sat` e o modelo resultante indica precisamente a transição da localização  $\ell = 1$  para  $\ell = 2$ .

## Apêndice 3.A Codificações SMT

Codificação no sistema Z3 do autómato híbrido na Fig. 3.1.

```

from z3 import *

# enumerated sorts

Modes, (Start, On, Off) = EnumSort('Modes', ['Start', 'On', 'Off'])
Events, (start, turnOn, turnOff, alive, dead) =
    EnumSort('Events', ['start', 'turnOn', 'turnOff', 'alive', 'dead'])

# variables

N=10

temp   = RealVector('x', N+1)
time   = RealVector('t', N+1)

modes   = AstVector() ; modes.resize(N+1)
events  = AstVector() ; events.resize(N+1)
for k in range(N+1):
    modes[k] = Const('loc_'+str(k), Modes)
    events[k] = Const('ev_'+str(k), Events)

# init

def init(S):
    (X,T,M) = S
    return And(T == 0, M == Start)

# invariant

def inv(S):
    (X,T,M) = S
    return If(M == On, X <= 22, X >= 18)

# trans

def timed(S,S_,E):
    (X,T,M) = S ; (X_,T_,M_) = S_
    return And(
        T_ - T > 0, M_ == M, E == alive,
        Implies( M == On, (X_ - X) - 2.8 * (T_ - T) >= 0),
        Implies( M == Off, (X_ - X) + 1.8 * (T_ - T) <= 0),
        Implies( M == Start, False)
    )

def untimed(S,S_,E):
    (X,T,M) = S ; (X_,T_,M_) = S_
    return And(
        T_ == T,
        Implies( M == On, And( M_ == Off, E == turnOff, X_ > 21)),
        Implies( M == Off, And( M_ == On, E == turnOn, X_ < 19)),
        Implies( M == Start, And( M_ == Off, E == start, X_ == 20))
    )

```

```
def trans(S,S_,E):
    return Or(untimed(S,S_,E) , timed(S,S_,E))

# state

def S(k):
    return (temp[k],time[k],modes[k])

def E(k):
    return events[k]

# model

s = Solver()

s.add(init(S(0)))

for k in range(N):
    s.add(inv(S(k)))
    s.add(trans(S(k),S(k+1),E(k)))

s.add(inv(S(N)))

# check

print(s.check())

try:
    print(s.model())
except Z3Exception as ex:
    print("failed: %s" % ex)
```

# 4

## Verificação de Modelos Finitos

A *verificação de modelos finitos*, em inglês “*model checking*” (MC), é provavelmente a técnica formal mais usada para verificação das propriedades temporais do *hardware*. Adicionalmente, a sua popularidade neste campo, fez com que em vários outros contextos tenha também sido usada na verificação ou falsificação de *propriedades lógicas* de sistemas modelados por uma *máquina de estados finita* (FSM).

O denominador comum é a representação das propriedades lógicas e da especificação das FSM’s em fórmulas de lógicas apropriadas e transformar o problema *a propriedade  $\phi$  é válida no modelo  $M$*  num problema de verificação da validade de uma fórmula nessas lógicas.

Neste capítulo vamos analisar as duas estratégias MC mais importantes. A chamada *verificação algorítmica* codifica propriedades e modelos por entidades designadas por *binary decision diagrams* (BDD’s) e usa algoritmos específicos das BDD’s para verificar a sua validade. A segunda abordagem, designada por *bounded model checking* ou BMC, codifica propriedades e modelos por fórmulas numa teoria SMT (ou SAT) e usa um SAT ou SMT *solver* para verificar a sua validade.

Em qualquer dos casos as propriedades são formalmente descritas numa *lógica temporal* como as que veremos na secção 4.1. As *finite state machines* são formalmente descritas por uma *estrutura de Kripke* como veremos na secção 4.2.

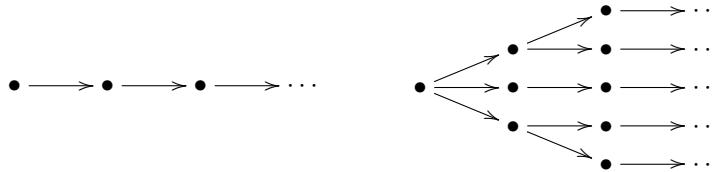
### 4.1 Lógica Temporal

A *lógica temporal* (TL) é um formalismo que foi desenvolvido para descrever as propriedades de sistemas concorrentes e reativos onde não bastam as asserções simples da lógica proposicional, ou de primeira ordem, porque o valor de verdade dessas asserções está ligado ao estado (ou instante de tempo) onde é avaliado.

Existem dois formalismos distintos<sup>†</sup> em lógica temporal que se distinguem na forma como o *tempo* é interpretado: ou como *tempo linear* ou como *tempo ramificado*. Formalmente o tempo é visto, no primeiro caso, como uma *ordem total* enquanto que em tempo ramificado os instantes de tempo (ou estados) estão organizados

<sup>†</sup>Esta secção baseia-se no artigo *Characterizing Finite Kripke Structures in Propositional Temporal Logic*, M.C.Browne, E.M. Clarke e O.Grumberg, Theoretical Computer Science 59 (1988).

como árvores.



No *tempo linear* todos os instantes estão linearmente ordenados desde o passado mais longínquo até ao futuro mais remoto, e em cada instante só existe um único futuro imediato. Este ponto de vista é conveniente para descrever algoritmos determinísticos; se o algoritmo for não determinístico a sua descrição contém o conjunto de todas as possíveis execuções, ou traços; individualmente os traços são vistos na perspetiva de tempo linear: em cada traço, cada tempo tem um único futuro.

Em contraste, em *tempo ramificado*, o futuro não é determinado e cada instante de tempo pode ter vários futuros imediatos. Este formalismo é mais adequado à descrição de sistemas não-determinísticos mas, geralmente, os formalismos lineares são mais simples, mais fáceis de entender e de mais fáceis de formalizar.

Um outro aspecto da LT que é crucial ao sucesso do “model checking” é o fato de conseguir descrever comportamentos infinitos usando estruturas finitas. Comportamentos infinitos ou, mais exatamente, “*não-limitados*” permitem descrever sistemas complexos e tecnologicamente relevantes que “não terminam”\*; a finitude das estruturas tem como consequência que os problemas de “model checking” são decidíveis e frequentemente têm algoritmos eficientes.

Neste curso a noção de “estrutura” é interpretada de duas formas diferentes:

- A *estrutura computacional* é um modelo de um algoritmo ou de um sistema. Tipicamente assume a forma de um ou vários sistemas de transição de estados, numa classe apropriada, que interagem entre si; pode também assumir a forma de um circuito formal ou real (*hardware*).  
Estas estruturas podem ser finitas ou infinitas mas, mesmo no primeiro caso, são capazes de representar um número infinito de comportamentos.
- A *estrutura comportamental* é um modelo do comportamento do sistema modelado pela estrutura computacional. Normalmente é um modelo infinito que se obtém do modelo computacional através de uma noção de *travessia* ou “*unfolding*”.

Uma lógica temporal define asserções, e estabelece a respetiva validade, sobre a estrutura comportamental; por isso a *estrutura lógica* de um sistema é realmente a sua *estrutura comportamental*.

A estrutura computacional é um modelo de transição de estados a partir do qual se gera, por regras apropriadas de semântica operacional, a estrutura comportamental. A lógica não descreve nada da estrutura computacional a não ser o comportamento que gera; nomeadamente não consegue distinguir estruturas computacionais diferentes que gerem o mesmo comportamento ou comportamentos equivalentes.

Desta forma pode-se sumariar a essência do problema do “model checking” definindo-o como “*a verificação algorítmica da validade de asserções lógicas, ou propriedades*

---

\*e.g. cpu's, sistemas operativos, sistemas de monitorização e controlo, etc.

de um modelo comportamental, no contexto de um dado **modelo computacional**”. Obviamente, quando o modelo computacional é finito, estamos em presença da “verificação em modelos finitos”.

### 4.1.1 Estruturas de Kripke

No contexto do “model checking”, a **estrutura computacional** tem, quase sempre, a forma de *estruturas de Kripke* cuja definição apresentamos em § 4.1.

---

#### § 4.1 Estrutura de Kripke.

Uma *estrutura de Kripke* é um tuplo  $\mathcal{K} = \langle \mathcal{P}, Q, \rightarrow, \ell \rangle$  em que:

1.  $\mathcal{P}$  é um conjunto enumerado de *asserções atómicas*.
  2.  $Q$  é um conjunto finito de *estados*; os elementos de  $Q$  também são designados por *configurações* ou *mundos*.
  3.  $\rightarrow \subseteq Q \times Q$  é uma *relação total* escrita como  $w \rightarrow w'$ ;
 

□1: Note-se  $\rightarrow$  é total quando, para todo  $w \in Q$ , existe  $w' \in Q$  tal que  $w \rightarrow w'$ . □
  4.  $\ell: Q \rightarrow 2^{\mathcal{P}}$  associa cada mundo  $w$  ao conjunto de proposições atómicas que são válidas nesse mundo; escreve-se  $w \models P$  se e só se  $P \in \ell(w)$ .
- 

A **estrutura comportamental** baseia-se na noção de *caminho* (“path”) e de *traço* (“run”) e está expressa em § 4.2.

---

#### § 4.2 Comportamento linear gerado por uma estrutura de Kripke.

Tomando como referência a estrutura de Kripke definida em § 4.1 define-se

1. Um *caminho* (*path*) é uma sequência  $\pi = \langle w_0, w_1, \dots, w_k, \dots \rangle$  finita ou infinita de mundos tal que  $w_{k-1} \rightarrow w_k$  para todo  $k > 0$ .
    - Escreve-se  $\pi_k$  para designar o  $(k+1)$ -ésimo mundo no caminho  $\pi$ .
    - O caminho  $\pi$  é *completo* se não é prefixo de nenhum outro caminho.

□2: Como  $\rightarrow$  é uma relação total, todos os caminhos completos são infinitos. □
  2. Dado um qualquer subconjunto de estados  $I \subseteq Q$ , um *traço* (*run*) de  $I$  é um caminho completo  $\pi$  que verifica  $\pi_0 \in I$ . Representamos por  $\mathcal{K}(I)$  os traços de  $I$  gerados pela estrutura de Kripke  $\mathcal{K}$ . O conjunto  $\mathcal{K}(Q)$  de todos os caminhos completos é representado simplesmente por  $\mathcal{K}$ .
  3. Um *comportamento linear* (“linear behaviour”) na estrutura de Kripke é um qualquer subconjunto  $\mathcal{B} \subseteq \mathcal{K}$  fechado por sufixos. Representamos por  $\mathcal{B}(I)$  o conjunto dos traços de  $I$  contidos em  $\mathcal{B}$ : isto é,  $\mathcal{B} \cap \mathcal{K}(I)$ .
- 

As noções de comportamento estendem-se para outro tipo de *relações de acessibilidade* como as que estão definidas em § 4.3.

Uma mesma estrutura de Kripke pode gerar vários comportamentos lineares; por isso interessa comparar esses comportamentos e aferir se dois deles são, de alguma

---

### § 4.3 Comportamento (continuação).

---

1. Escreve-se  $w \rightarrow^* w'$  quando existe algum  $n \geq 0$  tal que  $w \rightarrow^n w'$ . Escreve-se  $w \rightarrow^+ w'$  quando existe algum  $n > 0$  tal que  $w \rightarrow^n w'$ .
  2. Generaliza-se 4. para qualquer comportamento  $\mathcal{B}$ . Assim escreve-se  $w \rightarrow_{\mathcal{B}}^n w'$  quando existe  $\pi \in \mathcal{B}(w)$  tal que  $w' = \pi_n$ ;  $w \rightarrow_{\mathcal{B}} w'$  é equivalente a  $w \rightarrow_{\mathcal{B}}^1 w'$ ; escreve-se  $w \rightarrow_{\mathcal{B}}^* w'$  quando existe  $n \geq 0$  tal que  $w \rightarrow_{\mathcal{B}}^n w'$ ; escreve-se  $w \rightarrow_{\mathcal{B}}^+ w'$  quando existe  $n > 0$  tal que  $w \rightarrow_{\mathcal{B}}^n w'$ .
  3.  $[w : w']_\pi$ , “ $w$  upto  $w'$  on  $\pi$ ”, é o conjunto dos estados que no caminho  $\pi$  estão entre  $w$  e  $w'$ ; formalmente  $[w : w']_\pi = \{v \mid w \rightarrow_\pi^* v \text{ e } v \rightarrow_\pi^* w'\}$ . Define-se também,  $[w : w']_\pi = \{v \mid w \rightarrow_\pi^* v \text{ e } v \rightarrow_\pi^+ w'\}$ , “ $w$  until  $w'$  on  $\pi$ ”.
- 

forma, “equivalentes”. As relações de *bissimulação* são mecanismo formais que normalmente são usados na comparação de comportamentos; em § 4.4 vamos apresentar um exemplo de tal mecanismo.

---

### § 4.4 Bissimulação de comportamentos lineares.

---

No contexto da estrutura de Kripke definida em § 4.1 e nas noções de comportamento definidas em § 4.2 consideram-se definidos dois comportamentos lineares  $\mathcal{B}, \mathcal{B}'$ . Então:

1. Uma relação de equivalência  $\sim \subseteq Q \times Q$  é uma  $(\mathcal{B}, \mathcal{B}')$ -*bissimulação* nos estados da estrutura de Kripke  $\mathcal{K}$ , quando  $w \sim w'$  implica:
    - (i)  $\ell(w) = \ell(w')$ ,
    - (ii) Se  $w \rightarrow_{\mathcal{B}}^* q$ , então existe  $q'$  tal que  $q \sim q'$  e  $w' \rightarrow_{\mathcal{B}'}^* q'$ .
    - (iii) Se  $w' \rightarrow_{\mathcal{B}'}^* q'$ , então existe  $q$  tal que  $q \sim q'$  e  $w \rightarrow_{\mathcal{B}}^* q$ .
  2. Os comportamentos  $\mathcal{B}, \mathcal{B}'$  são *bissimilares* se existe uma relação de  $(\mathcal{B}, \mathcal{B}')$ -*bissimulação* nos estados  $Q$ .
- 

**EXEMPLO 4.1:** Vamos considerar o caso em que  $\ell$  é injectiva; isto é, dois estados que validem extamente as mesmas proposições atómicas são iguais. Neste caso,  $w \sim w'$  implica  $w = w'$ .

Portanto, a definição de bissimulação toma aqui uma forma particularmente simples: os comportamentos  $\mathcal{B}$  e  $\mathcal{B}'$  são bissimilares quando, para todo  $w, w'$ , verifica-se  $w \rightarrow_{\mathcal{B}}^* w'$  se e só se verifica-se  $w \rightarrow_{\mathcal{B}'}^* w'$ .  $\square$

#### 4.1.2 Sintaxe e semântica de LTL e CTL

A lógica temporal mais simples, designada por LTL (“linear temporal logic”), estende da lógica proposicional com dois *operadores modais*: o operador unário “next”  $X$ , e o operador binário “until”  $U$ .

---

**§ 4.5** Sintaxe da lógica LTL.

Dada o conjunto de *fórmulas atómicas* (ou *símbolos proposicionais*)  $\mathcal{P}$ , a lógica LTL tem as fórmulas

$$\phi, \varphi ::= p \in \mathcal{P} \mid \phi \wedge \varphi \mid \neg\phi \mid X\phi \mid \phi U \varphi \quad (4.1)$$

- As três primeiras regras geram as fórmulas puramente proposicionais; as restantes conectivas definem-se através de equivalências:

$$T \equiv \neg(p \wedge \neg p) , \quad \phi \vee \varphi \equiv \neg(\neg\phi \wedge \neg\varphi) , \quad \phi \rightarrow \varphi \equiv \neg(\phi \wedge \neg\varphi) \quad (4.2)$$

- A fórmula  $X\phi$  é válida “agora”, se  $\phi$  for válida “já a seguir”.
- A fórmula  $\phi U \varphi$  é válida “agora”, se  $\varphi$  for válida “algures no futuro” e  $\phi$  é válida “até lá”.
- Dois outros operadores são comuns em lógicas temporais: o operador  $F$ , “algures no futuro”, e o operador  $G$ , “sempre no futuro”. Definem-se por

$$F\varphi \equiv T U \varphi , \quad G\varphi \equiv \neg F \neg \varphi \quad (4.3)$$


---

As lógicas CTL (“computation tree logic”) e a sua versão estendida CTL\* introduzem o tempo ramificada e, por isso, redefinem a noção de “futuro”.

Essencialmente introduzem dois quantificadores sobre os estados do estado presente: o quantificador universal  $A$  e o quantificador existencial  $E$ . Informalmente,  $A\phi$  é válida “agora” se, para todos os traços do estado presente,  $\phi$  é válido;  $E\phi$  é válido “agora” se existe um traço do estado presente que valida  $\phi$ .

---

**§ 4.6** Sintaxe das lógicas CTL e CTL\* e sua interpretação .

A lógica CTL estende a lógica LTL mas restringe o uso dos seus operadores básicos  $X$  e  $U$  de forma a poderem ser utilizados apenas no âmbito imediato dos quantificadores  $A$  ou  $E$ . A sua sintaxe é

$$\phi, \varphi ::= p \in \mathcal{P} \mid \phi \wedge \varphi \mid \neg\phi \mid A(X\phi) \mid A(\phi U \varphi) \mid E(X\phi) \mid E(\phi U \varphi) \quad (4.4)$$

Pode-se ver CTL como uma versão estendida de LTL, substituindo o par  $(X, U)$  por dois pares de operadores compostos,  $(AX, AU)$  e  $(EX, EU)$ , e modificando o conceito de “futuro”.

A noção de “futuro” é interpretada de duas formas distintas: para  $(AX, AU)$ , o futuro é visto na perspetiva de “todos os traços” ou “todas as consequências”; para  $(EX, EU)$  o futuro é visto na perspetiva de “algum traço” ou “alguma consequência”.

A lógica CTL\* aumenta LTL introduzindo o quantificador existencial  $E$ , sem impor qualquer restrição ao seu uso. A sintaxe mínima é

$$\phi, \varphi ::= p \in \mathcal{P} \mid \phi \wedge \varphi \mid \neg\phi \mid X\phi \mid \phi U \varphi \mid E\phi \quad (4.5)$$

Neste caso  $A$  não é primitivo e define-se como  $A\phi \equiv \neg E \neg \phi$ .

---

A semântica da lógica temporal CTL\* é definida como uma relação entre as fórmulas da lógica e os comportamentos lineares de uma determinada estrutura de Kripke.

Como CTL e LTL são fragmentos de CTL\*, definindo a semântica de CTL\* fica automaticamente definida a semântica tanto de CTL como de LTL.

---

#### § 4.7 Semântica de LTL, CTL e CTL\*.

Dada uma estrutura de Kripke sobre um conjunto de símbolos proposicionais  $\mathcal{P}$ , e dado um comportamento linear  $\mathcal{B}$  em  $\mathcal{K}$ , define-se

1. Para  $p \in \mathcal{P}$  e  $\mathcal{B}(w) \neq \emptyset$  define-se  $\mathcal{B}, w \models p$  como  $p \in \ell(w)$ .
  - 3:** Se  $\mathcal{B}(w) = \emptyset$ , é indefinido se  $\mathcal{B}, w \models p$  ocorre ou não. □
  2.  $\mathcal{B}, w \models \neg\phi$  sse  $\mathcal{B}, w \not\models \phi$ , e  $\mathcal{B}, w \models \phi \wedge \varphi$  sse  $\mathcal{B}, w \models \phi$  e  $\mathcal{B}, w \models \varphi$ .
  3. Para todo  $\pi \in \mathcal{B}$  é definido
    - i)  $\pi, w \models X\phi$  se e só se  $w \rightarrow_\pi w'$  e  $\pi, w' \models \phi$ .
    - ii)  $\pi, w \models \phi U \varphi$  sse existe  $w'$  que verifica  $w \rightarrow_\pi^* w'$  e tal que
      - $\pi, v \models \phi$  se  $v \in [w : w')_\pi$ ,
      - $\pi, v \models \varphi$  se  $v = w'$
  4.  $\mathcal{B}, w \models E\phi$  sse existe  $\pi \in \mathcal{B}(w)$  tal que  $\pi, w \models \phi$ .
- 

Note-se que as três primeiras regras definem completamente a semântica de LTL. Esta semântica define-se exclusivamente sobre caminhos  $\pi$ . Nomeadamente tem-se

$$\pi, w \models F\varphi \text{ sse existe } w' \text{ tal que } w \rightarrow_\pi^* w' \text{ e } \pi, w' \models \varphi \quad (4.6)$$

$$\pi, w \models G\varphi \text{ sse } w \rightarrow_\pi^* w' \text{ implica sempre } \pi, w' \models \varphi \quad (4.7)$$

A semântica dos restantes operadores temporais em CTL e CTL\* (incluindo os operadores compostos EX, EU, AX, AU) define-se essencialmente combinando a regra 4 com as restantes regras.

**EXEMPLO 4.2:** A semântica dos operadores temporais em CTL é formada pelas regras 1-3 de § 4.7 e ainda

1.  $\mathcal{B}, w \models EX\phi$  sse existe  $\pi \in \mathcal{B}(w)$  tal que  $\pi, w \models X\phi$ .
2.  $\mathcal{B}, w \models AX\phi$  sse para todo  $\pi \in \mathcal{B}(w)$  tem-se  $\pi, w \models X\phi$ .
3.  $\mathcal{B}, w \models E(\phi U \varphi)$  sse existe  $\pi \in \mathcal{B}(w)$  tal que  $\pi, w \models \phi U \varphi$ .
4.  $\mathcal{B}, w \models A(\phi U \varphi)$  sse para todo  $\pi \in \mathcal{B}(w)$  tem-se  $\pi, w \models \phi U \varphi$ .

Destas regras deduzem-se outras regras que são usadas frequentemente.

- a)  $\mathcal{B}, w \models EX\phi$  sse  $w \rightarrow_B w'$  e  $\mathcal{B}, w' \models \phi$ , para algum  $w'$ .
- b)  $\mathcal{B}, w \models AX\phi$  sse  $w \rightarrow_B w'$  implica  $\mathcal{B}, w' \models \phi$ , para todo  $w'$ .

Os operadores EX e AX são duais: o primeiro quantifica existencialmente em relação a todos os estados acessíveis em um só passo a partir do estado presente. O segundo quantifica universalmente sobre os mesmos estados.

- c)  $\mathcal{B}, w \models EF\phi$  sse  $w \rightarrow_B^* w'$  e  $\mathcal{B}, w' \models \phi$ , para algum  $w'$ .
- d)  $\mathcal{B}, w \models AG\phi$  sse  $w \rightarrow_B^* w'$  implica  $\mathcal{B}, w' \models \phi$ , para todo  $w'$ .

Os operadores  $\text{EF}$  e  $\text{AG}$  são também duais: o primeiro quantifica existencialmente em relação aos estados acessíveis do estado presente, enquanto que o segundo faz uma quantificação universal em relação aos mesmos estados. Pode-se dizer que os operadores compostos  $\text{EX}$ ,  $\text{AX}$ ,  $\text{EF}$ ,  $\text{AG}$  “actuam sobre estados acessíveis”. Ao invés o operador  $\text{EG}$  e o seu dual  $\text{AF}$  misturam dois quantificadores de natureza distinta, um sobre os traços e outro sobre os estados de cada traço.

- e)  $\mathcal{B}, w \models \text{EG} \phi$  sse existe  $\pi \in \mathcal{B}$  tal que  $w \rightarrow_{\pi}^* w'$  implica  $\mathcal{B}, w' \models \phi$  para todo  $w'$ .
- f)  $\mathcal{B}, w \models \text{AF} \phi$  sse para todo  $\pi \in \mathcal{B}$  existe  $w'$  tal que  $w \rightarrow_{\pi}^* w'$  e  $\mathcal{B}, w' \models \phi$ .

□

As noções de tautologia e de fórmula não satisfazível têm uma formulação própria em cada uma das lógicas temporais.

---

#### § 4.8 Satisfação em LTL, CTL e CTL\*.

---

Na continuação de § 4.7, tem-se

1. Em LTL, tem-se  $\mathcal{B} \models \phi$  sse  $\pi, \pi_0 \models \phi$  para todo  $\pi \in \mathcal{B}$ .
  2. Em CTL e CTL\*, tem-se  $\mathcal{B} \models \phi$  sse  $\mathcal{B}, \pi_0 \models \phi$  para todo  $\pi \in \mathcal{B}$ .
- 

As noções de comportamentos bissimilares e de validade de fórmulas CTL estão relacionadas. De facto prova-se

- 4.1 TEOREMA (BISSIMULAÇÃO) *Os comportamentos  $\mathcal{B}$  e  $\mathcal{B}'$  são bissimilares se e só se, para todo  $\phi \in \text{CTL}$ ,  $\mathcal{B} \models \phi$  sse  $\mathcal{B}' \models \phi$ .*

Este resultado diz-nos que, a menos da relação de bissimulação, as fórmulas CTL conseguem distinguir comportamentos: dados dois comportamentos que não sejam bissimilares, existe sempre uma fórmula CTL que um valida e o outro não.

Na lógica CTL define-se também uma relação de *equivalência semântica*:  $\phi \equiv_{\kappa} \varphi$  ocorre quando, para todo o comportamento  $\mathcal{B} \subseteq \mathcal{K}$ , se verifica  $\mathcal{B} \models \phi$  se e só se  $\mathcal{B} \models \varphi$ . Neste contexto pode-se afirmar

- 4.1 COROLÁRIO *As classes de equivalência definidas pela relação de bissimulação na família dos comportamentos  $\mathcal{B} \subseteq \mathcal{K}$  estão em correspondência biunívoca com as classes de equivalência definidas em CTL pela equivalência semântica  $\equiv_{\kappa}$ .*

## 4.2 Máquinas de Estado Finitas

Neste curso vamos considerar as FSM's envolvidas no problema de “model checking” como FOTS (“first order transition systems”), tal como foram apresentadas na secção 3.4, mas sujeitos a uma restrição essencial: o espaço de estados é finito.

Como as propriedades das FSM's são expressas numa das lógicas LTL ou CTL, cuja semântica é definida em estruturas de Kripke, é essencial para a verificação dessas propriedades que as FSM's possam ser representadas por estruturas de Kripke.

Nesta secção pretende-se, precisamente, estudar a forma como se deve ser feita essa representação.

Tomando por referência a definição na secção 3.4, as FSM's vão ser descritas por FOTS's  $\Sigma = \langle \mathcal{X}, \mathcal{X}', \text{init}, \text{trans} \rangle$ . Estas FOTS's têm uma forma simplificada (sem condição de terminação nem invariante) porque se assume que os sistemas que queremos descrever não terminam, nomeadamente os traços são infinitos, e o invariante é uma propriedade que deve ser verificada como qualquer outra.

Adicionalmente, dado que o espaço de estados tem de ser finito, os conjuntos de variáveis  $\mathcal{X}$  e  $\mathcal{X}'$  são necessariamente finitos assim como os domínios associados às diferentes variáveis.

Para simplificar a formalização vamos concentrar todas as variáveis  $x_i \in \mathcal{X}$ , numa única variável vetorial  $x = \langle x_1, \dots, x_n \rangle$ ; se  $D_i$  for o domínio associado a  $x_i$ , então  $x$  toma valores no domínio  $\mathcal{D} = D_1 \times \dots \times D_n$ .

Vamos supor que existe uma teoria de 1ª ordem que tem os elementos de  $\mathcal{D}$  como objetos; nessa teoria os predicados `init`( $x$ ) e `trans`( $x, x'$ ) são fórmulas válidas. Como o domínio é finito a teoria é decidível mesmo que recorra a quantificadores.

Do lado das estruturas de Kripke  $\mathcal{K} = \langle \mathcal{P}, Q, \rightarrow, \ell \rangle$ , ver § 4.1, temos de definir, em primeiro lugar, o conjunto enumerado das *asserções atómicas*  $\mathcal{P}$ .

O conjunto  $\mathcal{P}$  na estrutura de Kripke é determinado pelo conjunto de asserções atómicas usadas na descrição das propriedades da FSM; nomeadamente na lógica temporal que serve de *linguagem de especificação* da FSM. Por outro lado as frases da linguagem de especificação estão sempre relacionadas com os valores que  $x$  pode tomar; desta forma pode-se sempre assumir que:

- A *linguagem de especificação* determina o conjunto de asserções atómicas  $\mathcal{P}$ .
- Cada  $p \in \mathcal{P}$  está associado a um predíco  $\mathbf{p}(x)$  na teoria do FOTS.

Os restantes elementos da estrutura de Kripke (o espaço de estados  $Q$ , a relação de acessibilidade  $\rightarrow \subseteq Q \times Q$  e a função  $\ell : Q \rightarrow 2^{\mathcal{Q}}$ ) dependem do tipo de máquina de estados que estamos a definir: *determinística*, *probabilística* ou *não-determinística*.

Uma vez definida a estrutura de Kripke, é preciso ter em atenção que nem todos os traços da estrutura de Kripke  $\mathcal{K}$  são execuções válidas na máquina de estados  $\Sigma$ . Isto porque, em  $\Sigma$  só são válidas as execuções que se iniciam num estado que valida `init`. Desta forma é necessário especificar o comportamento sobre o qual se define o significado das fórmulas na linguagem de especificação e essa especificação vai também depender do tipo de máquina de estado.

#### 4.2.1 FSM's determinísticas e probabilísticas

No contexto de uma teoria de primeira ordem finita com domínio  $\mathcal{D}$ , um FOTS  $\Sigma = \langle \mathcal{X}, \mathcal{X}', \text{init}, \text{trans} \rangle$  é **determinística** quando:

1. O predíco `init` é determinado pela igualdade a uma *constante*; isto é, para alguma constante  $c$  na teoria  $T_{\Sigma}$ ,

$$\text{init}(x) \equiv (x = c) \quad (4.8)$$

2. O predicado **trans** é *funcional*: i.e., existe uma função  $f$  na teoria  $T_\Sigma$  tal que

$$\mathbf{trans}(x, x') \equiv (x' = f(x)) \quad (4.9)$$

Uma tal FSM é descrita por um modelo  $\mathcal{K} = \langle \mathcal{P}, Q, \rightarrow, \ell \rangle$  que verifica:

- O espaço de estados  $Q$  é um subdomínio de  $\mathcal{D}$  tal que são tautologias

$$\mathbf{init}(x) \rightarrow (x \in Q) , \quad \mathbf{trans}(x, x') \wedge (x \in Q) \rightarrow (x' \in Q) \quad (4.10)$$

Essencialmente o predicado característico ( $x \in Q$ ) é um invariante do FOTS.

- Cada estado  $w \in Q$  identifica-se com o predicado

$$\mathbf{w}(x) \equiv (x = w) \quad (4.11)$$

Genericamente cada  $W \subseteq Q$  identifica-se com o predicado característico

$$(x \in W) \equiv \bigvee_{w \in W} (x = w) \quad (4.12)$$

Uma vez definido o espaço de estados, as restantes componentes da estrutura de Kripke definem-se através de tautologias na teoria  $T_\Sigma$ .

Assim, para todo  $w, w' \in Q$  e  $p \in \mathcal{P}$ , a relação  $\rightarrow \subseteq Q \times Q$  define-se

$$w \rightarrow w' \text{ sse é tautologia } \mathbf{w}(x) \wedge \mathbf{trans}(x, x') \rightarrow \mathbf{w}'(x') \quad (4.13)$$

e a função  $\ell : Q \rightarrow 2^\mathcal{P}$  define-se por

$$p \in \ell(w) \text{ sse é tautologia } \mathbf{w}(x) \rightarrow \mathbf{p}(x) \quad (4.14)$$

Para comportamento  $\mathcal{B}$  escolhe-se o conjunto dos traços de  $\mathcal{K}$  cujo estado inicial satisfaz **init**.

$$\mathcal{B} = \{ \pi \in \mathcal{K} \mid \mathbf{init}(\pi_0) \} \quad (4.15)$$

Tomando por referência a definição de FSM's determinísticas, as FSM **probabilísticas** divergem destas na assunção de que as constantes e as funções são computadas usando um *oráculo aleatório*.

Concretamente considera-se um domínio  $R \equiv \{0, 1\}^r$ , para algum  $r > 0$ , e diz-se que o FOTS  $\Sigma$  é *R-probabilístico* quando

1. O predicado **init** é determinado pela igualdade com a uma *constante aleatória*; isto é, existe  $c^R$  na teoria  $T_\Sigma$  tal que

$$\mathbf{init}(x) \equiv \exists z \in R \cdot (x = c(z)) \quad (4.16)$$

2. O predicado **trans** é *funcional aleatório*: i.e., existe uma função  $f^R$  na teoria  $T_\Sigma$  tal que

$$\mathbf{trans}(x, x') \equiv \exists z \in R \cdot (x' = f(z, x)) \quad (4.17)$$

Tendo em atenção estas restrições a estrutura de Kripke define-se do mesmo modo.

### 4.2.2 FSM's não determinísticas

Numa FSM não determinística não se impõe nenhuma restrição adicional ao FOTS  $\Sigma$  para além de se exigir que a teoria  $T_\Sigma$  seja finita.

Na estrutura de Kripke  $\mathcal{K} = \langle \mathcal{P}, Q, \rightarrow, \ell \rangle$ , vamos associar a noção de *estado* aos subconjuntos do domínio  $\mathcal{D}$ ; em princípio cada  $q \subseteq \mathcal{D}$  pode ser um estado. Equivalentemente cada predicado unário  $\mathbf{q}(x)$  pode ser um estado: cada  $q \in Q$  é também determinado, a menos da equivalência proposicional, por um predicado

$$\mathbf{q}(x) = \bigvee_{w \in q} (x = w) \quad (4.18)$$

Temos  $Q \subseteq 2^{\mathcal{D}}$ ; no entanto, nem todas as coleções de conjuntos  $q \subseteq \mathcal{D}$  definem um espaço de estados numa estrutura de Kripke que modele  $\Sigma$ ; de fato existem algumas condições que os espaços de estados devem verificar.

Em primeiro lugar o predicado  $\text{init}(x)$  define um estado, o *estado inicial*; portanto  $Q$  tem de conter  $\text{init}$ .

Em segundo lugar só pode ser espaço de estado um dois cuja união coincide com  $\mathcal{D}$ .

$$\bigwedge_{q \neq q'} (q \cap q' = \emptyset) \quad , \quad \bigcup_{q \in Q} q = \mathcal{D} \quad (4.19)$$

Em termos de predicados,  $Q$  define as tautologias:  $(\bigvee_{q \in Q} \mathbf{q}(x))$  e  $(\neg \mathbf{q}(x) \vee \neg \mathbf{q}'(x))$  para todos  $q \neq q' \in Q$ .

Um predicado  $P(x)$  é *representável* por  $Q$  se existe  $S \subseteq Q$  tal que é tautologia

$$P(x) \leftrightarrow \bigvee_{q \in S} \mathbf{q}(x)$$

O conjunto  $S$  é a *representação* de  $P(x)$  em  $Q$ .

Para um qualquer  $q \in Q$ , defina-se o predicado

$$\vec{\mathbf{q}}(x) \equiv \exists z \cdot \mathbf{q}(z) \wedge \text{trans}(z, x) \quad (4.20)$$

Como terceira e final restrição em  $Q$  vamos impor que, para todo  $q \in Q$ , o predicado  $\vec{\mathbf{q}}(x)$  seja representável em  $Q$ .

Designemos por  $\sigma(q)$  a representação de  $\vec{\mathbf{q}}(x)$ . A relação de acessibilidade  $\rightarrow$  da estrutura de Kripke pode-se definir como

$$q \rightarrow q' \quad \text{sse } q' \in \sigma(q) \quad (4.21)$$

ou, equivalentemente, por

$$q \rightarrow q' \quad \text{sse é tautologia } \neg \mathbf{q}'(x) \vee \vec{\mathbf{q}}(x) \quad (4.22)$$

A função  $\ell$  define-se facilmente por

$$p \in \ell(q) \quad \text{sse é tautologia } \neg \mathbf{q}(x) \vee \mathbf{p}(x) \quad (4.23)$$

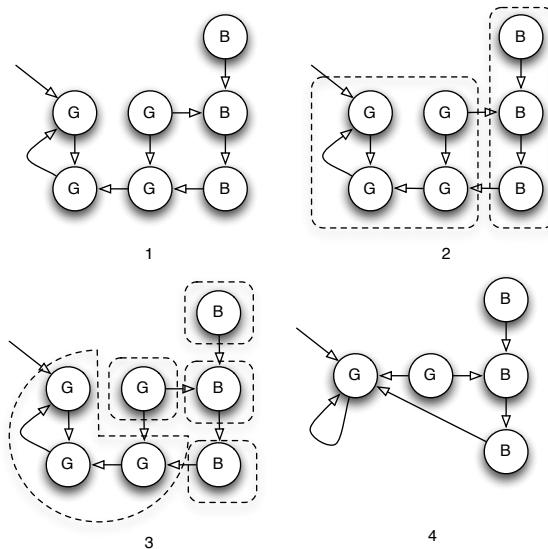
Finalmente o comportamento  $\mathcal{B}$  é o conjunto de todos os traços que têm  $\text{init}(x)$  como estado inicial.

## 4.3 Verificação Algoritmica

### 4.3.1 Minimização por Bissimulação

Frequentemente a verificação de fórmulas temporais num determinado comportamento  $\mathcal{B}$  envolve apenas algumas (poucas) proposições atómicas. Nestas circunstâncias se for possível reduzir a dimensão do espaço de estados e construir um comportamento  $\mathcal{B}'$  que, nesse espaço de estados reduzido, seja bissimilar com  $\mathcal{B}$ , então, para toda a fórmula  $\phi$  que só use esse conjunto limitado de proposições atómicas, avaliar  $\mathcal{B} \models \phi$  pode ser realizado avaliando  $\mathcal{B}' \models \phi$ ; como normalmente  $\mathcal{B}'$  é mais simples do que  $\mathcal{B}$  pode-se, desta forma, simplificar substancialmente o processo de “model checking”.

Figura 4.1: *Minimização por bissimulação* em relação a um só símbolo.



A figura 4.1 ilustra a estrutura essencial de um algoritmo para reduzir o espaço de estados numa situação onde é relevante apenas uma proposição atómica  $p$ .

No passo 1 separa os estados em dois conjuntos: o conjunto dos estados, ditos “good” e marcados com  $G$ , onde a proposição atómica é válida e o conjunto dos estados, ditos “bad” e marcados com  $B$ , onde a proposição atómica não é válida.

No passo 2 forma-se um multigraffo com dois grandes estados designados por *blocos*: um bloco é o conjunto dos estados  $B$  e o outro bloco é o conjunto dos estados  $B$ .

No passo 3 cada bloco é fracionado em função da relação de acessibilidade dos seus estados. No bloco  $G$  vemos que 3 dos estados só accedem a outros estados dentro do mesmo conjunto; como consequência estes três estados formam um novo bloco que se distingue do quarto estado  $G$  porque este accede a estados fora do bloco original; por isso o bloco  $G$  separa-se em dois. O bloco  $B$  fraciona-se em três blocos singulares porque as relações de acessibilidade são distintas para cada um destes estados: um dos estados accede só a estados  $B$ , outro estado é accedido por estados  $G$  e o terceiro estado accede ele próprio a estados  $G$ .

No final, no passo 4, constrói-se um novo conjunto de estados, um para cada um dos blocos construídos em 3, e uma nova relação de acessibilidade formada pelas transições herdadas das transições originais nestes blocos.

É simples provar, por indução, que o comportamento gerado por esta subestrutura é bissimilar ao comportamento gerado pela estrutura inicial. Por isso, para provar na estrutura original a validade de uma fórmula que contenha  $p$  como única proposição atómica, basta provar a validade desta fórmula na estrutura simplificada. Existem vários algoritmos de minimização por bissimulação baseados nesta estrutura\*.

### 4.3.2 Diagramas de Decisão Binária

Os algoritmos diretos de *model checking* assentam na representação quer da estrutura de Kripke quer ainda das fórmulas a verificar por estruturas de dados que, neste contexto, são conhecidas por *Binary Decision Diagrams* (BDD's).

Neste curso já analisámos estas estruturas na secção 2.1.4 (página 28); essencialmente BDD's representam grafos de Shannon ( $x \rightarrow B ; C$ ) em memória global e, por isso, o algoritmo de conversão de CNF em GS apresentado em § 2.9 (página 30) vai servir de referência aos diversos algoritmos que agem sobre BDD's.

Recordemos que no grafo ( $x \rightarrow B ; C$ ), as fórmulas  $B$  e  $C$  são também GS's que não contêm a variável  $x$  ou então são uma das constantes 1 ou 0.

Vamos considerar o conjunto  $\text{GS}(\mathcal{V})$  das BDD's definidas sobre o conjunto  $\mathcal{V} = \{v_1, \dots, v_n\}$  de *variáveis proposicionais* e a sua representação por BDD's†.

Em primeiro lugar os diagramas de decisão representam GS's *ordenados*; isto é, em  $(v_j \rightarrow A ; B)$  tanto  $A$  como  $B$  são ordenados e, se  $v_k$  ocorre em  $A$  ou  $B$ , então  $k > j$ . Em segundo lugar só vamos representar GS's *ordenados e reduzidos*: em  $\phi = (v_j \rightarrow A ; B)$ ,  $A$  e  $B$  são ordenados e reduzidos e nunca se verifica  $A \leftrightarrow B$ .

**4.1 PROPOSIÇÃO** Se  $\phi$  e  $\varphi$  são GS's ordenados e reduzidos, verifica-se  $\phi \leftrightarrow \varphi$  se e só se  $\phi = \varphi$ . Todo o GS ordenado é equivalente a um GS ordenado e reduzido.

**Justificação** Se  $\phi, \varphi$  são GS's ordenados e reduzidos, então  $\phi \leftrightarrow \varphi \Rightarrow \phi = \varphi$ . Este resultado prova-se por indução no número de variáveis; se  $\phi$  e  $\varphi$  têm zero variáveis, então só são equivalentes que coincidirem com a mesma constante 0 ou 1; se  $\phi = x \rightarrow A ; B$  e  $\varphi = x \rightarrow A' ; B'$  são equivalentes, então  $A \leftrightarrow A'$  e  $B \leftrightarrow B'$  e, pela hipótese de indução, tem-se  $A = A'$  e  $B = B'$  o que implica  $\phi = \varphi$ ; finalmente não é possível ter  $\phi = x \rightarrow A ; B$  e  $\varphi = y \rightarrow A' ; B'$  com  $x \neq y$  e  $\phi \leftrightarrow \varphi$  porque um dos GS  $\phi$  ou  $\varphi$  não seria reduzido.

A segunda parte da proposição prova-se com o algoritmo

$$\begin{aligned} \phi \leftarrow \text{RED}(x \rightarrow A ; B) &\equiv \\ A' \leftarrow \text{RED}(A) , B' \leftarrow \text{RED}(B) , \phi \leftarrow \begin{cases} A' & \text{se } A' = B' \\ (x \rightarrow A' ; B') & \text{se } A' \neq B' \end{cases} & \end{aligned} \quad (4.24)$$

Também por indução no número de variáveis de  $(x \rightarrow A ; B)$  é simples provar que  $\phi$  é ordenado, reduzido e equivalente a  $(x \rightarrow A ; B)$ .  $\square$

\*Ver *Bissimulation and Model Checking*, Fisler K. e Vardi M.Y., 1999.

†Existe um longo percurso na evolução das implementações eficientes de BDD's e suas

---

§ 4.9 Operações básicas sobre BDD's.

Dados GS's ordenados e reduzidos  $\phi \equiv (x \rightarrow A ; B)$  e  $\varphi \equiv (x' \rightarrow A' ; B')$ , existem implementações eficientes para as seguintes operações

1. Verificar a equivalência proposicional  $\phi \leftrightarrow \varphi$ . Nomeadamente verificar se  $\phi \leftrightarrow 1$  ( $\phi$  é tautologia) e  $\phi \leftrightarrow 0$  ( $\phi$  não é satisfazível).

□4: Atendendo à proposição 4.1, tem-se  $\phi \leftrightarrow \varphi$  se e só se  $\phi = \varphi$ . □

2. Calcular as GS's que representam  $\neg\phi$ ,  $\phi \vee \varphi$ ,  $\phi \wedge \varphi$  e  $\phi \rightarrow \varphi$ .

□5: Tem-se  $\neg\phi = x \rightarrow \neg A ; \neg B$  e, dado que  $\neg A \not\leftrightarrow \neg B$ , este GS é reduzido. Existe porém uma forma mais eficiente de determinar  $\neg\phi$  como veremos em § ??.

O cálculo da disjunção efetua-se por casos:

- $\phi \vee 1$  e  $\phi \vee 0$  coincidem com **1** e  $\phi$  respetivamente; ambos estão reduzidos.
- Se  $x = x'$ , tem-se  $\phi \vee \varphi \leftrightarrow \text{RED}(x \rightarrow A \vee A' ; B \vee B')$ . Note-se que o fato de ser  $A \not\leftrightarrow A'$  e  $B \not\leftrightarrow B'$  não impede que se possa ser  $(A \vee A') \leftrightarrow (B \vee B')$ ; e.g. se for  $A' = B$  e  $B' = A$ . Portanto é indispensável efetuar a redução.
- Se  $x > x'$  tem-se  $\phi \vee \varphi \leftrightarrow \text{RED}(x \rightarrow A \vee \varphi ; B \vee \varphi)$ ; não há razão para assumir que  $A \vee \varphi$  e  $B \vee \varphi$  não são equivalentes e a redução é, por isso, indispensável.

Os GS's  $\phi \wedge \varphi$  e  $\phi \rightarrow \varphi$  podem ser construídos como  $\neg(\neg\phi \vee \neg\varphi)$  e  $\neg\phi \vee \varphi$ ; dada a forma eficiente de determinar a negação, esta construção é preferível a uma representação direta da conjunção e da implicação. □

3. Dada a variável  $z \in \mathcal{V}$ , calcular GS's reduzidos  $\phi|_{z=0}$ ,  $\phi|_{z=1}$ , que resultam da substituição em  $\phi$  de  $z$  por 0 ou 1, e  $\exists z \cdot \phi$  definida como  $\phi|_{z=0} \vee \phi|_{z=1}$ .

□6: Tem-se  $\phi|_{x=1} = A$ ,  $\phi|_{x=0} = B$ ,  $\phi|_{z=\ell} = \phi$  se  $z > x$ . Se  $z < x$  tem-se  $\phi|_{z=\ell} = \text{RED}(x \rightarrow A|_{z=\ell} ; B|_{z=\ell})$ , para  $\ell \in \{0, 1\}$ ; neste caso, sendo  $\phi$  reduzido, não existe garantia que  $A|_{z=\ell} \not\leftrightarrow B|_{z=\ell}$  (se  $z < x$ ); logo  $(x \rightarrow A|_{z=\ell} ; B|_{z=\ell})$  não está reduzido e é indispensável usar o algoritmo de redução.

O GS  $\exists z \cdot \phi$  calcula-se como  $\phi|_{z=0} \vee \phi|_{z=1}$  usando a implementação da disjunção que referimos em 2. □

4. Calcular a BDD que representa a substituição genérica,  $\phi|_{z=\varphi}$ .

□7: Calcula-se  $\phi|_{z=\varphi}$  como  $\exists z \cdot (z \leftrightarrow \varphi) \wedge \phi \leftrightarrow \exists z \cdot (z \rightarrow \varphi ; \neg\varphi) \wedge \phi$ . □

---

A implementação das construções referidas em § 4.9 vai se basear na representação num dicionário das várias entidades sintáticas definidas na seguinte gramática.

$$\begin{aligned}
 \text{exp} &::= \text{name} \mid \neg\text{exp} \mid \text{exp} \vee \text{exp} \mid \exists \text{var} \cdot \text{exp} \\
 \text{name} &::= \text{key} \mid \neg\text{key} \\
 \text{key} &::= \text{one} \mid \text{hash(gs)} \\
 \text{gs} &::= \langle \text{var} , \text{name} , \text{name} \rangle \\
 \text{var} &::= v \in \mathcal{V}
 \end{aligned} \tag{4.25}$$

Para que cada  $\text{name}$  represente um e um só  $gs$  é necessário que o  $gs$  esteja *normalizado*; para isso em cada  $\langle x, u, w \rangle$  no dicionário,  $u$  é sempre do tipo *key*.

operações. No entanto pode-se dizer que todos estes trabalhos derivam do artigo seminal Brace et al. (1990) onde se introduz a noção de *diagramas de decisão ordenados* (OBDD's) e *diagramas de decisão reduzidos* (ROBDD's).

Vamos construir as funções seguintes:

$$\begin{array}{lll} \mathbf{bdd} & : \exp & \rightarrow \text{name} \\ \mathbf{bdd-ex} & : \text{var} \times \text{name} & \rightarrow \text{name} \\ \mathbf{bdd-or} & : \text{name} \times \text{name} & \rightarrow \text{name} \end{array} \quad (4.26)$$

**bdd** é a função principal; toma uma expressão  $\exp$  como argumento e determina a sua forma reduzida  $\text{RED}(\exp)$ ; como efeito lateral, atualiza o dicionário com  $\text{RED}(\exp)$  e com todos as expressões reduzidas intermédias; devolve o  $\text{name}$  que identifica a entrada no dicionário associada a  $\text{RED}(\exp)$ . As funções **bdd-or** e **bdd-ex** implementam **bdd** para argumentos da forma  $\exp \vee \exp$  ou da forma  $\exists \text{var} \cdot \exp$ .

Para implementar estas funções usa-se o algoritmo 4.2, assumindo que existem funções **exists**, **get** e **add** que implementam as operações básicas sobre dicionários: o teste de existência de uma chave no dicionário, a seleção de um valor associado a uma chave e a atualização do dicionário com uma nova associação  $(\text{chave}, \text{gs})$ . Usam-se também funções auxiliares

$$\begin{array}{lll} \mathbf{s-get} & : \text{name} & \rightarrow \text{gs} \\ \mathbf{s-add} & : \text{var} \times \text{name} \times \text{name} & \rightarrow \text{name} \end{array} \quad (4.27)$$

**s-get** é uma função auxiliar que, para toda  $key k$  com  $g = \mathbf{get}(k)$ , mapeia  $k$  em  $g$  e mapeia  $-k$  em  $\neg g$ . Analogamente **s-add** armazena no dicionários, se necessário for, a normalização do triplo  $\langle \text{var}, \text{name}, \text{name} \rangle$  e devolve o nome a ele associado. O algoritmo 4.1 implementa estas funções auxiliares.

```

fun S-GET ( $n$ ) is
  if  $n$  is key then return get( $n$ )
   $\langle v, u, w \rangle \leftarrow \mathbf{get}(-n)$ 
  return  $\langle v, -u, -w \rangle$ 

fun S-ADD ( $v, u, w$ ) is
  if  $u = w$  then return  $u$ 
  if  $u$  is key then
     $g \leftarrow \langle v, u, w \rangle$  ,  $s \leftarrow 1$ 
  else
     $g \leftarrow \langle v, -u, -w \rangle$  ,  $s \leftarrow -1$ 
     $k \leftarrow \text{hash}(g)$ 
    if not exists( $k$ ) then add( $k, g$ )
  return  $(s * k)$ 
```

**Algorithm 4.1:** Funções auxiliares do algoritmo 4.2.

```

fun BDD-OR ( $n, n'$ ) is
  if  $n = -n'$  or  $n = \text{one}$  or  $n' = \text{one}$  then return  $\text{one}$ 
  if  $n = n'$  or  $n' = -\text{one}$  then return  $n$ 
  if  $n = -\text{one}$  then return  $n'$ 
   $\langle x, a, b \rangle \leftarrow \text{S-GET}(n)$  ,  $\langle x', a', b' \rangle \leftarrow \text{S-GET}(n')$ 
  if  $x = x'$  then
     $u \leftarrow \text{BDD-OR}(a, a')$  ,  $w \leftarrow \text{BDD-OR}(b, b')$  ,  $v \leftarrow x$ 
  elif  $x > x'$  then
     $u \leftarrow \text{BDD-OR}(a, n')$  ,  $w \leftarrow \text{BDD-OR}(b, n')$  ,  $v \leftarrow x$ 
  else
     $u \leftarrow \text{BDD-OR}(n, a')$  ,  $w \leftarrow \text{BDD-OR}(n, b')$  ,  $v \leftarrow x'$ 
  return  $\text{S-ADD}(v, u, w)$ 

fun BDD-EX ( $z, n$ ) is
  fun SUB ( $n, z, \ell$ ) is
    if  $n = \text{one}$  or  $n = -\text{one}$  then return  $n$ 
     $\langle v, a, b \rangle \leftarrow \text{S-GET}(n)$ 
    if  $v < z$  then return  $n$ 
    if  $v = z$  then
      if  $\ell = 1$  then return  $a$ 
      else return  $b$ 
    else
       $u \leftarrow \text{SUB}(a, z, \ell)$  ,  $w \leftarrow \text{SUB}(b, z, \ell)$ 
    return  $\text{S-ADD}(v, u, w)$ 
  return BDD-OR(SUB( $n, z, 1$ ), SUB( $n, z, 0$ ))

fun BDD ( $e$ ) is
  if  $e$  is name then return  $e$ 
  case  $e$  is
     $(-\bar{f})$  then
      return  $(-\text{BDD}(f))$ 
     $(f \vee g)$  then
      return BDD-OR (BDD( $f$ ), BDD( $g$ ))
     $(\exists z \cdot f)$  then
      return BDD-EX ( $z, \text{BDD}(f)$ )
  
```

**Algorithm 4.2:** Implementação eficiente de GS ordenados e reduzidos.

### 4.3.3 O $\mu$ -Calculus

As fórmulas  $\text{GS}(\mathcal{V})$ , e a sua representação por BDD's, são usadas para codificar as entidades de uma lógica mais geral: o  $\mu$ -Calculus. Esta lógica é usada para descrever não só as fórmulas da lógica temporal CTL mas também para descrever os modelos de Kripke que estabelecem significado a essas fórmulas.

Vamos representar por  $v$  o vetor das variáveis  $\langle v_1, \dots, v_n \rangle$ .

---

### § 4.10 $\mu$ -Calculus.

Um  $\mu$ -Calculus sobre  $\mathcal{V}$  é um fragmento da Lógica de 2ª Ordem gerado por um conjunto de *símbolos proposicionais*  $\mathcal{F}$  e um conjunto de *símbolos relacionais*  $\mathcal{R}$  e é formado por *fórmulas*  $\phi$  e *relações*  $R$  geradas pela seguinte sintaxe.

$$\begin{aligned}\phi, \varphi &::= f \in \mathcal{F} \mid S(v) \mid \exists z \cdot \phi \mid \neg\phi \mid \phi \wedge \varphi \\ S &::= R \in \mathcal{R} \mid \lambda v \cdot \phi \mid \mu X \cdot T\end{aligned}$$

sendo  $z$  uma variável proposicional,  $X$  uma variável relacional e  $T$  uma expressão relacional que contém  $X$  como variável livre.

- Toda a relação tem associado uma aridade  $k$ ; a aridade dos símbolos  $R \in \mathcal{R}$  é definida na sintaxe; de resto a aridade é definida na construção sintática.
- $S(v)$  é a fórmula que se obtém por *aplicação* da relação unária  $S$  ao vetor de variáveis proposicionais  $v$ .
- A relação  $S \equiv \lambda v \cdot \phi$  é a *abstração* da fórmula  $\phi$ ; por definição é a relação unária que verifica  $S(v) \equiv \phi$ .
- Na relação  $\mu X \cdot T$ , a expressão  $T$  determina uma aplicação  $S \mapsto T|_{X=S}$ ; a relação  $\mu X \cdot T$  é o *menor ponto fixo* dessa aplicação.

Isto significa que  $S \equiv \mu X \cdot T$  é a menor relação que verifica  $S = T|_{X=S}$ . Se  $T$  tem aridade  $k > 0$ , então  $\mu X \cdot T$  tem aridade  $k - 1$ .

---

Sem perda de generalidade as relações vão ser interpretadas sobre um domínio finito  $Q$  com cardinalidade  $\leq 2^n$ , de tal forma que uma relação de aridade  $k$  é interpretada como um subconjunto de  $Q^k$ .

Os elementos de  $Q$  designam-se por *estados* e o limite na cardinalidade deste domínio, significa que cada  $w \in Q$  pode ser codificado com  $n$  bits. Para isso vamos considerar uma função de descodificação  $\rho: \{0,1\}^n \rightarrow Q$  que é sobrejetiva mas não, necessariamente, injetiva. Os *códigos* de  $w$  são os elementos de  $\rho^{-1}(w)$ .

A interpretação de fórmulas  $f \in \text{GS}(\mathcal{V})$  é feita como em qualquer lógica proposicional; nomeadamente os *modelos* são vetores  $x \in \{0,1\}^n$ . Como  $f$  é representável por uma BDD pode-se calcular  $f|_{v=x}$  como BDD e verificar  $x \models f$  usando

$$x \models f \quad \text{sse} \quad f|_{v=x} \leftrightarrow 1 \tag{4.28}$$

Pode-se também representar um modelo  $x$  por uma BDD  $v(x)$  definida pela cláusula conjuntiva  $l_1 \wedge \dots \wedge l_n$ , com  $l_i = v_i$  se  $x_i = 1$  e  $l_i = \neg v_i$  se  $x_i = 0$ . Neste caso

$$x \models f \quad \text{sse} \quad v(x) \rightarrow f \text{ é tautologia}$$

A interpretação de relações  $\Phi$  e de fórmulas  $\phi$  envolvendo relações envolve também modelos para os símbolos relacionais  $R \in \mathcal{R}$ .

Uma relação unária  $S$  é interpretada como um subconjunto de  $Q$ ; sem ambiguidade vamos usar o mesmo símbolo para representar tanto a expressão relacional como o conjunto que a interpreta. Deste modo a fórmula  $S(v)$  é válida em todos os modelos que codificam um estado na interpretação de  $S$ ; isto é

$$x \models S(v) \quad \text{sse} \quad \rho(x) \in S \tag{4.29}$$

A interpretação das restantes fórmulas é feita da forma usual

$$\begin{aligned} x \models \exists z \cdot \phi &\quad \text{sse } x \models \phi|_{z=0} \text{ ou } x \models \phi|_{z=1} \\ x \models \neg\phi &\quad \text{sse } x \not\models \phi \\ x \models \phi \wedge \varphi &\quad \text{sse } x \models \phi \text{ e } x \models \varphi \end{aligned} \tag{4.30}$$

A interpretação das expressões relacionais associa a cada uma destas expressões com aridade  $k$ , um subconjunto de  $Q^k$ .

Nestas circunstâncias, os símbolos relacionais  $R \in \mathcal{R}$  têm uma interpretação que é definida por um *modelo*  $\mathcal{I}$ . Pode-se ver  $\mathcal{I}$  como uma função que mapeia cada  $R \in \mathcal{R}$  numa relação com a aridade adequada. As restantes expressões relacionais têm a forma  $(\lambda v \cdot \phi)$  ou  $(\mu X \cdot T)$ .

A construção mais simples é a relação unária  $(\lambda v \cdot \phi)$ ; tal relação é interpretada

$$(\lambda v \cdot \phi) \equiv \{\rho(x) \mid x \in \{0, 1\}^n, x \models \phi\} \tag{4.31}$$

Para interpretar  $\mu X \cdot T$  temos de assumir que a expressão  $T$  é monótona não-decrescente em  $X$ ; isto é, tem de se verificar para todas as relações  $N$  e  $M$

$$N \subseteq M \Rightarrow T|_{X=N} \subseteq T|_{X=M} \quad \text{e} \quad N \subseteq T|_{X=N}$$

Desta forma  $S \equiv \mu X \cdot T$  é interpretada pela relação construída como o *limite* da sequência crescente  $\emptyset \equiv S_0 \subseteq \dots \subseteq S_i \subseteq \dots$ , com  $S_{i+1} \equiv T|_{X=S_i}$ . O limite  $S$  existe porque todo  $S_i$  está contido num conjunto finito; por isso,  $S$  coincide com o primeiro  $S_i$  que verifica  $T|_{X=S_i} = S_i$ .

$$\begin{aligned} (\mu X \cdot T) &\equiv \lim_{\uparrow} \{S_i\} \\ \text{com } S_0 &\equiv \emptyset \quad \text{e} \quad S_{i+1} \equiv T|_{X=S_i} \end{aligned} \tag{4.32}$$

A partir da definição de *menor ponto fixo* ( $\mu X \cdot T$ ) é possível definir o seu dual, o *maior ponto fixo*, representado por  $\nu X \cdot T$ . Esta construção aplica-se a expressões  $T$  que sejam monótonas não-crescentes; isto é, para todas relações  $N, M$  verifica-se

$$N \supseteq M \Rightarrow T|_{X=N} \supseteq T|_{X=M} \quad \text{e} \quad N \supseteq T|_{X=N}$$

Então  $S \equiv \nu X \cdot T$  é interpretado como a maior relação tal que  $S_0 \supseteq \dots \supseteq S_i \supseteq \dots \supseteq S$ , com  $S_0 \equiv \{0, 1\}^n$  e  $S_{i+1} \equiv T|_{X=S_i}$ ; i.e.  $S$  é o primeiro  $S_i$  que verifica  $T|_{X=S_i} = S_i$ .

$$\begin{aligned} (\nu X \cdot T) &\equiv \lim_{\downarrow} \{S_i\} \\ \text{com } S_0 &\equiv \{0, 1\}^n \quad \text{e} \quad S_{i+1} \equiv T|_{X=S_i} \end{aligned} \tag{4.33}$$

É simples verificar que  $\nu X \cdot T \equiv \neg(\mu Y \cdot \neg(T|_{X=-Y}))$  e portanto o maior ponto fixo não é uma construção primitiva do  $\mu$ -Calculus. No entanto as implementações do  $\mu$ -Calculus apresentam normalmente dois algoritmos distintos para realizar o menor ponto fixo e o maior ponto fixo.

Uma outra construção não primitiva, mas com grande relevância no problema de “model checking”, é a aplicação da relação inversa. Seja  $N$  uma relação binária e seja  $N(v, v')$  a fórmula que a representa; para toda a relação unária  $S$  define-se

$$N^{-1}(S) \equiv \lambda v \cdot \exists v' \cdot N(v, v') \wedge S(v') \tag{4.34}$$

#### 4.3.4 Verificação orientada aos BDD's

Os algoritmos de “model checking” usam o  $\mu$ -Calculus como uma linguagem intermédia: o modelo (estrutura de Kripke) define um determinado  $\mu$ -Calculus, as fórmulas do CTL são convertidas em expressões relacionais nesse  $\mu$ -Calculus e finalmente fórmulas e relações do  $\mu$ -Calculus são representadas por BDD's.

A figura 4.2 representa a sequência de codificações: das estruturas e das fórmulas na linguagem intermédia e, finalmente, da linguagem intermédia em BDD's.

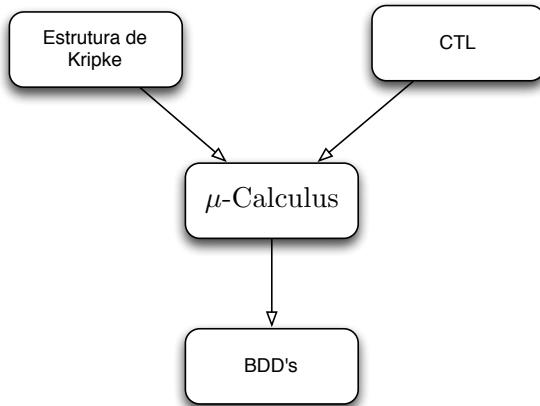


Figura 4.2: Codificações usadas na verificação algorítmica

Como primeiro passo vamos começar precisamente pela representação do  $\mu$ -Calculus em BDD's. Vamos definir um algoritmo BDD que recursivamente constrói representações das entidades do  $\mu$ -Calculus, fórmulas e relações, por BDD's. Formalmente cada BDD é um grafo de Shannon e a construção e representação desses GS's segue a abordagem do armazenamento em memória global adotado no algoritmo § 2.9.

---

§ 4.11 “Model Checking” - BDD’s para as fórmulas e relações do  $\mu$ -Calculus.

No que se segue vamos assumir um  $\mu$ -Calculus gerado por um conjunto  $\mathcal{F}$  de símbolos proposicionais e um conjunto  $\mathcal{R}$  de símbolos relacionais unários ou binários. Facilmente se estende os algoritmos que vamos apresentar a símbolos relacionais de aridade arbitrária.

1. Um modelo  $\mathcal{I}$  associa a cada  $R \in \mathcal{R}$  unário um elemento de  $\text{GS}(\mathcal{V})$  representando a fórmula  $R(v)$ , e a cada  $N \in \mathcal{R}$  binário um elemento de  $\text{GS}(\mathcal{V} \cup \mathcal{V}')$  representando a fórmula  $N(v, v')$ .

Identicamente o modelo  $\mathcal{I}$  associa a cada símbolo proposicional  $f \in \mathcal{F}$  um elemento de  $\text{GS}(\mathcal{V})$ .

O dicionário é inicializado armazenando cada uma das representações  $g$  (tanto de relações como de proposições) através da associação  $\text{hash}(g) \rightarrow g$ .

2. A fórmula  $S(v)$  é representada pela mesmo GS que representa  $S$ .
  3. As fórmulas  $\neg\phi$  e  $\phi \wedge \varphi$  são representadas por GS's que se calculam eficientemente a partir das representações de  $\phi$  e  $\varphi$ .
  4. A fórmula  $\exists z \cdot \phi$  é representada pelo GS que representa  $\phi|_{z=0} \vee \phi|_{z=1}$ . A partir do GS que representa  $\phi$ , é eficiente computar os GS que representam  $\phi|_{z=0}$  e  $\phi|_{z=1}$  e depois calcular o GS que representa a sua disjunção.
  - $\square$ 8: Uma fórmula  $\exists v' \cdot S(v, v')$  é equivalente a  $\exists v'_1 \cdot \exists v'_2 \cdots \exists v'_n \cdot S(v, v')$ . Isto significa que a sua representação por um GS pode ser calculada usando esta regra  $n$  vezes: uma para cada uma das variáveis  $v'_i$ .  $\square$
  5. A relação  $\lambda v. \phi$  é representada pela mesma BDD que representa  $\phi$ . As expressões relacionais  $\neg S$ ,  $S \wedge R$  e  $S \vee R$  são abreviaturas de  $\lambda v \cdot \neg S(v)$ ,  $\lambda v \cdot S(v) \wedge R(v)$  e  $\lambda v \cdot S(v) \vee R(v)$ , respectivamente, e são representadas pelas BDD's construídas por estas expressões.
  6. As relações  $\mu X \cdot T$  e  $\nu X \cdot T$  têm representações por GS calculados pelo algoritmo 4.3.
- 

```
fun ponto_fixo (T, X, s, ℓ) is
    if ℓ = 0 then return s
    s' ← BDD(T|X=λv·s)
    if s = s' then return s
    return ponto_fixo(T, X, s', ℓ - 1)
```

**Algorithm 4.3:** Calculo do ponto fixo.

[Notas](#)

- O algoritmo é recursivo; para além dos parâmetros  $T$  e  $X$  é invocado com dois parâmetros extra:  $s$  é um GS (essencialmente uma chave do dicionário) que representa uma aproximação à solução;  $\ell$  é um limite no número de iterações.
- No cálculo de um *menor ponto fixo*  $\mu X \cdot T$ , a aproximação  $s$  deve ser uma relação menor ou igual do que que a solução. No cálculo de um *maior ponto fixo*  $\nu X \cdot T$  a aproximação é um limite superior da solução.
- A função `BDD`, invocada em `ponto_fixo(s, ℓ)`, é a conversão de uma relação em GS definida precisamente neste conjunto de regras em § 4.11.

- Normalmente  $T$  é representado por uma GS que tem  $X$  como variável livre. Neste caso  $T|_{X=\lambda v.s}$  tem uma implementação simples: basta substituir no GS  $T$  a variável  $X$  por  $s$  e reduzir o GS resultante.

□

O segundo passo do algoritmo de “model checking” é a representação de modelos de Kripke por  $\mu$ -Calculus e a sua interpretação por BDD’s.

---

#### § 4.12 Representação de uma estrutura de Kripke no $\mu$ -Calculus e sua interpretação.

---

1.  $\mathcal{K} = \langle \mathcal{P}, Q, \rightarrow, \ell \rangle$  é uma estrutura de Kripke:  $\mathcal{P}$  é o conjunto das asserções atómicas,  $Q$  é o conjunto dos estados,  $\rightarrow \subseteq Q \times Q$  é a relação de transição e  $\ell : Q \rightarrow 2^{\mathcal{P}}$  é a função de “labelling”. Tanto  $Q$  como  $\mathcal{P}$  são finitos.
  2.  $\mathcal{K}$  representa-se por um  $\mu$ -Calculus  $\mathcal{M} = \langle \mathcal{V}, \mathcal{F}, \mathcal{R} \rangle$  definido por:
    - (i) Seja  $n \geq \lceil \log_2(\#Q) \rceil$  um número de bits suficiente para codificar sem ambiguidade cada estado  $q \in Q$ . O conjunto das variáveis proposicionais é  $\mathcal{V} \equiv \{v_1, \dots, v_n\}$ .
    - (ii) Cada estado  $q \in Q$  determina um símbolo proposicional; isto é,  $\mathcal{F} \equiv Q$ .
    - (iii) Cada  $p \in \mathcal{P}$  determina um símbolo relacional unário  $\mathbf{p}$ . A relação de transição  $\rightarrow$  é identificada com um símbolo relacional binário  $\mathbf{N}$  (“next”). Assim  $\mathcal{R} \equiv \{ \mathbf{p} \mid p \in \mathcal{P} \} \cup \{ \mathbf{N} \}$ .
  3. A interpretação  $\mathcal{I}$  de  $\mathcal{M}$  é gerada do modo seguinte:
    - (i) Escolhe-se uma descodificação  $\rho : \{0,1\}^n \rightarrow Q$  sobrejectiva. Para cada  $q$  seja  $\rho^{-1}(q) = \{x_1, \dots, x_m\}$  o conjunto dos distintos códigos de  $q$ ; então, a interpretação  $\mathcal{I}$  associa  $q$  à BDD  $\hat{q}$  que representa  $v(x_1) \vee \dots \vee v(x_m)$ . Isto é,  $\hat{q} \equiv \text{BDD}(\bigvee_{x \in \rho^{-1}(q)} v(x))$ .
    - (ii) O conjunto das relações unárias  $\mathbf{p}$  representa a função de “labelling”  $\ell$ , através da identidade  $q \in \mathbf{p}$  sse  $p \in \ell(q)$ . Define-se  $\hat{\mathbf{p}} \equiv \text{BDD}(\bigvee_{q \in \mathbf{p}} \hat{q})$ . Assim tem-se  $\models \hat{q} \rightarrow \hat{\mathbf{p}}$  sse  $q \in \mathbf{p}$  sse  $p \in \ell(q)$ .
    - (iii) A BDD  $\hat{\mathbf{N}}$ , associada ao símbolo relacional  $\mathbf{N}$ , obtém-se simplesmente como  $\text{BDD}(\rightarrow)$  ou  $\text{BDD}(\rightarrow_B)$ , caso se esteja restrito a um comportamento particular  $B$  da estrutura  $\mathcal{K}$ .
- 

O passo final no algoritmo “model checking” é a codificação das lógicas CTL no  $\mu$ -Calculus. Vamos codificar a lógica  $\text{CTL}(\mathcal{P})$ , gerada pelas asserções atómicas  $\mathcal{P}$ , tal como é definida em (4.4). Essa codificação é feita no contexto da codificação da estrutura de Kripke  $\mathcal{K}$  que vimos em § 4.12 e é apresentado em § 4.13.

Uma codificação *correta* (“sound”) mapeia cada  $\phi \in \text{CTL}(\mathcal{P})$  numa relação unária  $[\phi]$  do  $\mu$ -Calculus de tal modo que, para qualquer estado  $q \in Q$ , se tem

$$\mathcal{K}, q \models \phi \quad \text{sse} \quad q \in [\phi] \quad \text{sse} \quad \hat{q} \rightarrow \text{BDD}([\phi]) \text{ é tautologia} \quad (4.35)$$

O objectivo de § 4.13 é definir uma codificação correta.

---

§ 4.13 Codificação de CTL em expressões relacionais  $\mu$ -Calculus.

---

1. A fórmula  $p$ , com  $p \in \mathcal{P}$ , é codificada na expressão relacional  $\llbracket p \rrbracket = \mathbf{p}$ .

□9: Em § 4.12 vimos que o símbolo relacional  $\mathbf{p}$  denota a relação  $\{ q \in Q \mid p \in \ell(q) \}$  e, por isso, a sua interpretação é  $\hat{\mathbf{p}} \equiv \text{BDD}(\bigvee_{q \in \mathbf{p}} \hat{q})$ , sendo  $\hat{q}$  a BDD que interpreta o estado  $q$ . □

2. As fórmulas  $\neg\phi$  e  $\phi \wedge \varphi$  são codificadas nas expressões relacionais

$$\llbracket \neg\phi \rrbracket = \lambda v \cdot \neg\llbracket \phi \rrbracket(v) \quad , \quad \llbracket \phi \wedge \varphi \rrbracket = \lambda v \cdot \llbracket \phi \rrbracket(v) \wedge \llbracket \varphi \rrbracket(v)$$

3. A codificação de  $\text{EX } \phi$  é a expressão

$$\llbracket \text{EX } \phi \rrbracket = \mathbf{N}^{-1}(\llbracket \phi \rrbracket)$$

Ver (4.34) para a definição da inversa de uma relação binária. A fórmula  $\text{AX } \phi$  é uma abreviatura de  $\neg\text{EX}(\neg\phi)$ .

4. A codificação de  $\text{EF } \phi$  é a expressão relacional

$$\llbracket \text{EF } \phi \rrbracket = \mu Y \cdot \llbracket \phi \rrbracket \vee \mathbf{N}^{-1}(Y)$$

A fórmula  $\text{AG } \phi$  é uma abreviatura de  $\neg\text{EF}(\neg\phi)$ .

5. A codificação de  $\text{EG } \phi$  é a expressão relacional

$$\llbracket \text{EG } \phi \rrbracket = \nu Y \cdot \llbracket \phi \rrbracket \wedge \mathbf{N}^{-1}(Y)$$

A fórmula  $\text{AF } \phi$  é uma abreviatura de  $\neg\text{EG}(\neg\phi)$ .

6. A codificação de  $\text{E}(\phi \cup \varphi)$  é a expressão relacional

$$\llbracket \text{E}(\phi \cup \varphi) \rrbracket = \mu Y \cdot \llbracket \varphi \rrbracket \vee (\llbracket \phi \rrbracket \wedge \mathbf{N}^{-1}(Y))$$

7. A codificação de  $\text{A}(\phi \cup \varphi)$  pode ser feita de duas formas equivalentes que exploram a dualidade entre  $\text{EX}$  e  $\text{AX}$ . Pode-se escrever

$$\llbracket \text{A}(\phi \cup \varphi) \rrbracket = \mu Y \cdot \llbracket \varphi \rrbracket \vee (\llbracket \phi \rrbracket \wedge \text{AX } Y) \quad \text{ou}$$

$$\llbracket \text{A}(\phi \cup \varphi) \rrbracket = \neg(\nu Y \cdot \neg\llbracket \varphi \rrbracket \wedge (\neg\llbracket \phi \rrbracket \vee \mathbf{N}^{-1}(Y)))$$


---

É simples – apesar de trabalhoso – provar por indução percorrendo cada um dos casos, que a codificação em § 4.13 é correta. A título de exemplo pode-se analisar a codificação de  $\text{EX } \phi$  e de  $\text{E}(\phi \cup \varphi)$ .

- Verifica-se  $\mathcal{K}, q \models \text{EX } \phi$  quando existe  $q'$  tal que  $q \rightarrow q'$  e  $\mathcal{K}, q' \models \phi$ . Aprendendo que, na representação de  $\mathcal{K}$ , a expressão  $\mathbf{N}$  representa a relação  $\rightarrow$ , tem-se  $q \in \llbracket \text{EX } \phi \rrbracket$  sse  $\exists q' \cdot (q, q') \in \mathbf{N} \wedge q' \in \llbracket \phi \rrbracket$ ; isto é equivalente a afirmar que  $q \in \mathbf{N}^{-1}(\llbracket \phi \rrbracket)$ .
- Verifica-se  $\mathcal{K}, q \models \text{E}(\phi \cup \varphi)$  se no estado  $q$  a fórmula  $\varphi$  é válida ou então a fórmula  $\phi$  é válida e, em algum estado  $q'$  imediatamente acessível de  $q$ , é também válido  $\mathcal{K}, q' \models \text{E}(\phi \cup \varphi)$ . Isto significa que a expressão  $\text{E}(\phi \cup \varphi)$  deve

verificar em qualquer estrutura de Kripke

$$\mathbf{E}(\phi \mathbf{U} \varphi) \equiv \varphi \vee (\phi \wedge \mathbf{EX}(\mathbf{E}(\phi \mathbf{U} \varphi)))$$

Atendendo à representação em  $\mathcal{K}$  do operador  $\mathbf{EX}$  pela inversa de  $\mathbf{N}$ , obtém-se imediatamente a representação na regra 6.

Finalmente estamos em condições de descrever o algoritmo que implementa “model checking” em CTL.

---

#### § 4.14 “Model Checking” em CTL via BDD’s.

---

É dado uma estrutura de Kripke  $\mathcal{K}$ . Pretende-se construir um algoritmo que dado um qualquer estado  $q$  deste modelo e uma qualquer fórmula temporal  $\phi \in \text{CTL}$  determine o valor lógico de  $\mathcal{K}, q \models \phi$ .

1. Usando § 4.12 interpreta-se  $\mathcal{K}$  num  $\mu$ -Calculus e calcula-se a BDD  $\hat{q}$  que interpreta o estado  $q$ .
  2. Usando § 4.13 codifica-se  $\phi$  numa expressão relacional  $[\![\phi]\!]$ .
  3. Usando § 4.11 calcula-se a interpretação por BDD’s das relações atómicas do  $\mu$ -Calculus – os símbolos de relações  $\mathbf{p}$  e  $\mathbf{N}$  – e, através delas, obtém-se BDD ( $[\![\phi]\!]$ ).
  4. Calcula-se a BDD  $\hat{q} \rightarrow \text{BDD}([\![\phi]\!])$  e verifica-se se é uma tautologia testando a sua equivalência com 1.
- 

## 4.4 “Bounded Model Checking” (BMC)

Na lógica LTL é mais complexo usar um algoritmo de “model checking” do tipo do algoritmo em § 4.14. Isto porque, ao contrário do que ocorre em CTL, a verificação de uma fórmula não depende apenas de um estado  $q$  mas sim depende de um “estado  $q$  dentro de um caminho  $\pi$ ”. Por esse motivo a noção de verificação tem de envolver algum modo de codificar o caminho  $\pi$ .

Por simplicidade vamos considerar a verificação símbólica apenas em LTL. Recorremos a semântica de LTL definida em § 4.7 (página 83); dada uma estrutura de Kripke  $\mathcal{K}$ , para um qualquer caminho completo  $\pi \in \mathcal{K}$  e qualquer fórmula  $\phi \in \text{LTL}$ , vamos usar a notação  $\pi \models \phi$  como abreviatura de  $\pi, \pi_0 \models \phi$ . Adicionalmente, dado  $\pi \in \mathcal{K}$ , o seu sufixo  $\langle \pi_i, \pi_{i+1}, \dots \rangle$  é um elemento de  $\mathcal{K}$  representado por  $\pi^i$ .

Com esta notação a semântica de LTL pode ser escrita de forma alternativa:

$$\left[ \begin{array}{ll} \pi \models p & \text{sse } p \in \ell(\pi_0) \\ \pi \models \neg \phi & \text{sse } \pi \not\models \phi \\ \pi \models \phi \wedge \varphi & \text{sse } \pi \models \phi \text{ e } \pi \models \varphi \\ \pi \models X \phi & \text{sse } \pi^1 \models \phi \\ \pi \models \phi \mathbf{U} \varphi & \text{sse } \exists j \cdot (\pi^j \models \varphi) \wedge (\forall i < j \cdot \pi^i \models \phi) \end{array} \right] \quad (4.36)$$

#### 4.4.1 Traços e Laços

“Bounded Model Checking” estuda a validade, numa máquina de estados finita, de fórmulas temporais válidas sobre *traços* descritos por um número finito de estados.

O qualificativo “bounded” refere-se ao número de estados necessários para representar o traço; dado um traço é sempre uma sequência não limitada de estados, para ser possível representa-lo com um número finito de estados, ele terá de ter “ciclos”; i.e. os estados  $\pi_i$  repetem-se dentro de um conjunto finito de possibilidades. Esta situação está ilustrada na figura 4.3; este tipo de caminhos designam-se por *laços*.

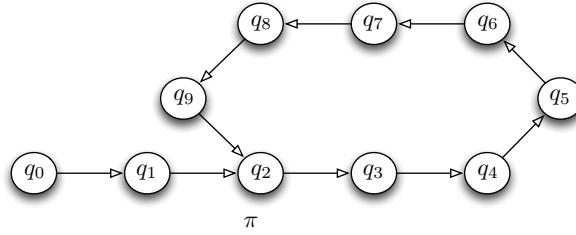
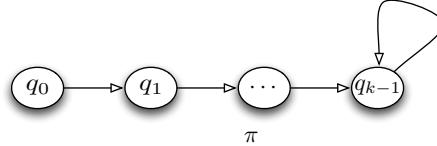


Figura 4.3: Caminho completo finito – um *laço*.

O laço representado na figura 4.3 tem 10 estados diferentes e o 11-ésimo estado coincide com o 3º; por isso designa-se por um *laço*-(10, 2). Genericamente, um *laço*-( $k, l$ ) é um caminho completo  $\pi$  que verifica  $\pi_{k+j} = \pi_{l+j}$  para todo  $j \geq 0$ .

Em particular um laço  $(k, k - 1)$  é um traço  $\pi$  em que, a partir de  $\pi_{k-1}$  todos os estados seguintes têm este mesmo valor; i.e.  $\pi$  “bloqueia” no  $k$ -ésimo estado.



Se  $\pi$  é um *laço*-( $k, l$ ) e  $j \geq l$ , então  $\pi^j$  é um laço com parâmetros  $(k - l, 0)$ . Se  $j < l$ ,  $\pi^j$  é um laço de parâmetros  $(k - j, l - j)$ .

Neste modelo vamos usar a convenção dos SMT’s e usar variáveis que tomam valores no domínio dos estados  $Q$ . Para tentar encontrar um *laço*-( $k, l$ )  $\pi$  pode-se representar cada um dos seus  $k$  primeiros estados por variáveis  $w_i$ . Se forem determinados os valores  $q_0, \dots, q_{k-1}$  que instanciam as variáveis  $w_0, \dots, w_{k-1}$ , então os estados  $\pi_j$  são determinados por\*

$$\pi_j = q_{\sigma(j)} \quad \text{sendo} \quad \sigma(j) = \begin{cases} j & \text{se } j < k \\ l + (j - k) \bmod (k - l) & \text{se } j \geq k \end{cases} \quad (4.37)$$

Genericamente tem-se

$$\pi_j^i = \pi_{j+i} = q_{\sigma(j+i)} \quad (4.38)$$

\*Note-se que os  $q_i$  são estados concretos, enquanto que os  $w_i$  designam variáveis a que estão atribuídas esses estados.

Dentro desta notação vamos procurar representar a propriedade “ $\pi$  é um traço em laço  $(k, l)$ ” por uma proposição  $\hat{\pi}$  nas variáveis  $\bar{w} \equiv \langle w_0, \dots, w_{k-1} \rangle$ .

Para isso vamos representar a relação  $\rightarrow$  na estrutura de Kripke  $\mathcal{K}$  por uma proposição  $\mathbf{N}(w, w')$  em duas variáveis  $w$  e  $w'$ .

Assim  $\mathbf{N}$  determina  $\hat{\pi}(w_0, w_1, \dots, w_{k-1})$  por

$$\hat{\pi} \equiv \mathbf{N}(w_0, w_1) \wedge \mathbf{N}(w_1, w_2) \wedge \dots \wedge \mathbf{N}(w_{k-2}, w_{k-1}) \wedge \mathbf{N}(w_{k-1}, w_l) \quad (4.39)$$

#### 4.4.2 Codificação LTL

A abordagem da verificação simbólica baseia-se na codificação de cada  $\phi \in \text{LTL}$  numa proposição  $\hat{\phi}(w_0, \dots, w_{k-1})$  de tal forma que se verifique

$$\exists \pi \cdot \pi \models \phi \quad \text{sse} \quad \hat{\pi} \wedge \hat{\phi} \text{ é satisfazível} \quad (4.40)$$

Para codificar as fórmulas de LTL em proposições sobre as variáveis  $w_0, \dots, w_{k-1}$ , vamos definir um operador *shift*  $\mathbf{S}_l$  que aplicado à proposição  $f$  substitui as variáveis segundo a progressão  $w_0 \rightsquigarrow w_1 \rightsquigarrow \dots \rightsquigarrow w_{k-1} \rightsquigarrow w_l$ . Isto é

$$(\mathbf{S}_l f)(w_0, w_1, \dots, w_{k-1}) = f(w_1, \dots, w_{k-1}, w_l) \quad (4.41)$$

com esta notação podemos reescrever a definição de  $\hat{\pi}$  em 4.39, num laço- $(k, l)$  como

$$\hat{\pi}_l \equiv \mathbf{N} \wedge \mathbf{S}_l \mathbf{N} \wedge (\mathbf{S}_l)^2 \mathbf{N} \wedge \dots \wedge (\mathbf{S}_l)^{k-1} \mathbf{N} \quad (4.42)$$

#### § 4.15 Codificação das fórmulas LTL em proposições.

Considere-se a lógica LTL definida em § 4.5 sobre um conjunto de asserções atómicas no contexto de uma estrutura de Kripke  $\langle Q, \rightarrow, \ell \rangle$ .

1. Para  $p \in \mathcal{P}$  tem-se  $\pi \models p$  sse  $p \in \ell(\pi_0) \equiv \pi_0 \in \ell^{-1}(p)$ ; então a codificação de  $p$  é a proposição

$$\hat{p}(w_0, \dots, w_{k-1}) \equiv \bigvee_{q \in \ell^{-1}(p)} (w_0 = q)$$

2. As fórmulas temporais  $\neg\phi$ ,  $\phi \wedge \varphi$  e  $\phi \vee \varphi$  são codificadas, respetivamente, por  $\neg\hat{\phi}$ ,  $\hat{\phi} \wedge \hat{\varphi}$  e  $\hat{\phi} \vee \hat{\varphi}$ .
3. A fórmula temporal  $\mathbf{X}\phi$  é codificada pela proposição  $\mathbf{S}_l \hat{\phi}$ .
4. A fórmula temporal  $\phi \mathbf{U} \varphi$  é codificada na proposição

$$\bigvee_{j < k} (\mathbf{S}_l)^j \hat{\varphi} \wedge \left( \bigwedge_{i < j} (\mathbf{S}_l)^i \hat{\phi} \right)$$

**Justificação** Estas regras de codificação são uma tradução direta da semântica definida em (4.36) e da relação  $\pi^j \models \phi$  sse é válido  $(\mathbf{S}_l)^j \hat{\phi}$  □

A partir destas regras básicas pode-se determinar a codificação de outros operadores temporais derivados, nomeadamente os operadores  $\mathbf{F}$  e  $\mathbf{G}$ .

5. A fórmula temporal  $\mathbf{F}\varphi$  codifica-se na proposição  $\bigvee_{j < k} (\mathbf{S}_l)^j \hat{\varphi}$ .
6. A fórmula temporal  $\mathbf{G}\phi$  é uma abreviatura de  $\neg\mathbf{F}\neg\phi$  e codifica-se na proposição  $\bigwedge_{j < k} (\mathbf{S}_l)^j \hat{\phi}$ .

A definição do operador  $\mathbf{S}_l$  em 4.41 está fixa a uma determinada estrutura de laço; isto é,  $\mathbf{S}_l$  aplica-se a traços  $\pi$  que são *laços-(k,l)*.

Como a codificação de  $\phi \in \text{LTL}$ , apresentada em § 4.15, depende do operador  $\mathbf{S}_l$ ,  $\hat{\phi}$  só é a codificação de  $\phi$  se realmente o traço tem esta estrutura. Quando não existe certeza sobre qual é a estrutura do traço temos de assumir que a fórmula  $\phi$  tem várias codificações possíveis  $\hat{\phi}_l$ , uma para cada valor possível de  $l \in \{0..k-1\}$ .

A mesma observação se aplica à codificação do traço  $\pi$  apresentada em (4.42): cada  $l \in \{0..k-1\}$  determina uma codificação  $\hat{\pi}_l$ .

Assim, para codificar a fórmula a validar  $\phi$ , aumenta-se a teoria SMT com uma nova variável inteira  $\lambda$ , que toma valores no intervalo  $\{0..k-1\}$ , codifica-se  $\phi$  como

$$\hat{\phi} \equiv \bigvee_{l=0}^{k-1} (\lambda = l) \wedge \hat{\phi}_l \quad (4.43)$$

Do mesmo modo codifica-se  $\pi$  como

$$\hat{\pi} \equiv \bigvee_{l=0}^{k-1} (\lambda = l) \wedge \hat{\pi}_l \quad (4.44)$$

Desta forma, resolver problemas BMC para a especificação LTL  $\phi$ , envolve verificar se é satisfazível

$$\hat{\phi} \wedge \hat{\pi} \equiv \bigvee_{l=0}^{k-1} (\lambda = l) \wedge \hat{\phi}_l \wedge \hat{\pi}_l \quad (4.45)$$

Esta estratégia de codificação, traduzindo diretamente a semântica das fórmulas LTL, pode não ser a mais eficiente. Em alternativa pode-se considerar uma formulação via pontos fixos análoga ao que fizemos em § 4.13 para CTL\*.

Em traços gerais,

- (i) Codifica-se  $\mathbf{X}\phi$  como  $\bigvee_{l=0}^{k-1} (\lambda = l) \wedge \mathbf{S}_l \hat{\phi}$ .
- (ii) Codifica-se  $\mathbf{G}\phi$  como o maior ponto fixo da transformação  $Y \mapsto \hat{\phi} \wedge \mathbf{X}Y$ .

#### 4.4.3 Problemas BMC

Uma vez definida a forma de codificação pode-se enumerar vários tipos de problemas em “bounded model checking”

---

**§ 4.16** Procurar um traço  $\pi$ , limitado a  $k$  estados distintos, que verifique  $\pi \models \phi$ .

SOLUÇÃO

Uma vez obtidas as codificações  $\hat{\pi}$  e  $\hat{\phi}$ , usando (4.44) e (4.43), verifica-se se é satisfazível a proposição  $\hat{\pi} \wedge \hat{\phi}$ .

Esta solução é implementada invocando um SMT “solver” com “input”  $\hat{\pi} \wedge \hat{\phi}$ . Um modelo para esta proposição, obtido pelo “solver”, determina não só a estrutura do laço  $(k,l)$  mas também a sequência de estados  $\langle \pi_0, \dots, \pi_{k-1} \rangle$  - a *testemunha*.

---

\*Ver detalhes em Biere et al. (2003).

---

**§ 4.17** Verificar se todo o traço  $\pi$ , limitado a  $k$  estados distintos, verifica  $\pi \models \phi$ .

SOLUÇÃO

Usar § 4.16 para procurar uma testemunha de  $\pi \models \neg\phi$ .

Se não existir a referida testemunha, então  $\pi \models \phi$  verifica-se para todos os traços  $\pi$  que sejam limitados a  $k$  estados distintos. Se existir a referida testemunha ela designa-se por *contra-exemplo* e serve de refutação da propriedade  $\phi$ .

---

Note-se que nesta formulação dos problemas BMC, os traços  $\pi$  não são sujeitos a nenhuma outra restrição que não seja o fato de estarem limitados a  $k$  estados distintos. No entanto, na maioria das FSM's existe pelo menos uma restrição extra: impõe-se que o estado inicial  $\pi_0$  satisfaça uma dada condição de inicialização `init`. Isto é exemplo de uma *restrição de comportamento*.

Em termos gerais os traços  $\pi$  podem ser restritos a um comportamento  $\mathcal{B}$  genérico. Para isso a restrição  $\pi \in \mathcal{B}$  é formalizada por um predicado  $\hat{\mathcal{B}}(w_0, \dots, w_{k-1})$ . Agora o problema da procura de testemunhas reescreve-se

$$\exists \pi \in \mathcal{B} \cdot \pi \models \phi \quad \text{sse} \quad \hat{\mathcal{B}} \wedge \hat{\pi} \wedge \hat{\phi} \quad \text{é satisfazível} \quad (4.46)$$

e o problema da verificação reescreve-se

$$\forall \pi \in \mathcal{B} \cdot \pi \models \phi \quad \text{sse} \quad \hat{\mathcal{B}} \wedge \hat{\pi} \wedge \neg\hat{\phi} \quad \text{não é satisfazível} \quad (4.47)$$

Um outro problema particularmente importante é a *verificação de invariantes*. Dada a estrutura de Kripke  $\mathcal{K}$ , uma fórmula  $\theta$  é um *invariante* do comportamento  $\mathcal{B} \subseteq \mathcal{K}$  (ou um  $\mathcal{B}$ -*invariante*) quando ocorre

$$\forall \pi \in \mathcal{B} \cdot \pi \models G\theta \quad (4.48)$$

Para verificar se  $\theta$  é um invariante, usa-se o princípio da indução nos sistemas de transição de 1ª ordem – ver 3.41, pag. 71. Vamos começar definir uma noção mais fraca: diz-se que  $\theta \in \text{LTL}$  é *segura* (“safe”) em  $\mathcal{K}$  quando ocorre

$$\forall \pi \in \mathcal{K} \cdot \pi \models \theta \rightarrow X\theta \quad (4.49)$$

É simples verificar, pela semântica de LTL e pelo princípio da indução, o seguinte resultado

4.2 PROPOSIÇÃO (INDUÇÃO) *Se  $\theta$  é segura em  $\mathcal{K}$  e ocorre*

$$\forall \pi \in \mathcal{B} \cdot \pi \models \theta \quad (4.50)$$

*então  $\theta$  é um  $\mathcal{B}$ -invariante.*

A condição 4.50 prova-se, uma vez obtidas as codificações  $\hat{\mathcal{B}}$  e  $\hat{\theta}$ , verificando que  $\hat{\mathcal{B}} \rightarrow \hat{\theta}$  é uma tautologia. A maior dificuldade reside, normalmente, na verificação de (4.49).

Usando § 4.17 consegue-se verificar que  $\theta$  é segura mas apenas para traços que sejam  $k$ -limitados. A verificação para traços arbitrários pode ser feita por aproximações sucessivas fazendo crescer o limite  $k$ ; se um contra-exemplo ocorrer, tem-se a certeza que o candidato  $\theta$  não é seguro; porém, se ao fim de um número suficientemente elevado de iterações, não aparecer nenhum contra-exemplo pode-se afirmar, com pouca margem de erro, que  $\theta$  é segura mas não se pode ter a certeza disso.

## 4.5 Exemplos e Aplicações

Vamos basear os exemplos deste capítulo nos sistemas NuSMV e nuXmv\* e usar alguns dos exemplos que são referidos na documentação destes sistemas.

### 4.5.1 Verificação orientada às BDD's

O nosso primeiro exemplo, apresentado em 4.3, ilustra a descrição na linguagem NuSMV de um problema “model checking” num sistema digital muito simples: essencialmente o sistema é formado por três contadores de um só bit, interligados por um bit de “carry”.

A formulação do problema em NuSMV é feita em duas partes: uma primeira parte onde se descreve o *modelo* e uma segunda parte onde se descrevem as *propriedades* a validar. Neste caso o modelo é uma FSM determinística e a especificação é um conjunto de fórmulas CTL.

#### EXEMPLO 4.3:

```
-- descrição do modelo via uma FSM determinística

MODULE counter(input)
VAR
    value : boolean;
ASSIGN
    init(value) := FALSE;
    next(value) := value xor input;
DEFINE
    output := value & input;

MODULE main
VAR
    bit0 : counter(TRUE);
    bit1 : counter(bit0.output);
    bit2 : counter(bit1.output);

-- especificação em CTL

SPEC
    AG AF bit2.output
SPEC
    AG AF (!bit2.output)
SPEC
    AF AG (!bit2.output)
```

□

O exemplo 4.3 começa por uma descrição da FSM determinística.

\*Ver <http://{nusmv,nuxmv}.fbk.eu>; ver nomeadamente a documentação *NuSMV Tutorial* e *nuXmv User Manual* e os exemplos nela contidos.

1. O modelo é formado por três instâncias (`bit0`, `bit1` e `bit2`) de um mesmo módulo `counter`.
2. O estado de cada instância é determinado por dois tipos de variáveis: os parâmetros de entrada e as variáveis locais declaradas na secção `VAR`; neste exemplo temos um parâmetro `input` e uma variável local `value`.
3. A secção `ASSIGN` determina não só o estado inicial, em `init(value) := FALSE`, mas também a relação de transição, em `next(value) := value xor input`. Numa FSM determinística a especificação do estado inicial e da relação de transição é sempre feita na forma de atribuições.
4. A secção `DEFINE` determina as asserções atómicas  $p \in \mathcal{P}$  que vão ser usadas na formulação das propriedades da FSM.  
Cada instância determina uma asserção atómica que se identifica associando o nome `output` ao nome da instância. Assim são declaradas três asserções atómicas `bit0.output`, `bit1.output` e `bit2.output`.

A especificação contém três propriedades usando a asserção atómica  $p \equiv \text{bit2.output}$

1. A propriedade `AG(AF p)` especifica que para todos os traços  $\pi$  e todos os estados  $\pi_j$  nesses traços (o operador composto `AG`) e todos os traços  $\sigma$  de raiz  $\pi_j$  (i.e.  $\sigma_0 = \pi_j$ ) existe um estado  $\sigma_i$  (o operador `AF`) onde  $p$  é válido.  
A fórmula `AG(AF p)` é interpretada como “infinitely often”  $p$ ; isto é, qualquer que seja o estado futuro (em qualquer caminho), todo o caminho seguinte contém um estado onde  $p$  ocorre.
2. A propriedade `AG(AF  $\neg p$ )` é análoga à anterior mas referente à negação de  $p$ : denota “infinitely often”  $\neg p$ .
3. A propriedade `AF(AG  $\neg p$ )` especifica que para todos os traços  $\pi$  existe um estado  $\pi_j$  (o operador `AF`) a partir do qual em todos os caminhos e todos os estados (o operador `AG`)  $p$  não ocorre.  
Essencialmente `AF(AG  $\neg p$ )` diz que em todos os traços existe um estado a partir do qual, em qualquer caminho e em qualquer estado,  $p$  não ocorre nunca mais. Obviamente esta propriedade contradiz 1.

**EXEMPLO 4.4:** Assume-se que a descrição do exemplo 4.3 consta num ficheiro `counter.smv`. Então a execução e a resposta do “model checker” `NUXMV`, em qualquer OS UNIX, é

```
$ nuxmv counter.smv
*** This is nuXmv 1.0.1 (compiled on Mon Nov 17 17:49:50 2014)
*** Copyright (c) 2014, Fondazione Bruno Kessler

-- specification AG (AF bit2.output)  is true
-- specification AG (AF !bit2.output)  is true
-- specification AF (AG !bit2.output)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
-> State: 1.1 <-
  bit0.value = FALSE
```

```

bit1.value = FALSE
bit2.value = FALSE
bit0.output = FALSE
bit1.output = FALSE
bit2.output = FALSE
-> State: 1.2 <-
  bit0.value = TRUE
  bit0.output = TRUE
-> State: 1.3 <-
  bit0.value = FALSE
  bit1.value = TRUE
  bit0.output = FALSE
-> State: 1.4 <-
  bit0.value = TRUE
  bit0.output = TRUE
  bit1.output = TRUE
-> State: 1.5 <-
  bit0.value = FALSE
  bit1.value = FALSE
  bit2.value = TRUE
  bit0.output = FALSE
  bit1.output = FALSE
-> State: 1.6 <-
  bit0.value = TRUE
  bit0.output = TRUE
-> State: 1.7 <-
  bit0.value = FALSE
  bit1.value = TRUE
  bit0.output = FALSE
-> State: 1.8 <-
  bit0.value = TRUE
  bit0.output = TRUE
  bit1.output = TRUE
  bit2.output = TRUE
-> State: 1.9 <-
  bit0.value = FALSE
  bit1.value = FALSE
  bit2.value = FALSE
  bit0.output = FALSE
  bit1.output = FALSE
  bit2.output = FALSE

```

□

Analizando a resposta do “model checker” vemos que as duas primeiras propriedades são válidas, mas a terceira não é. O sistema `nuxmv` prova a não validade de `AF (AG !bit2.output)` apresentando, como contra-exemplo, os primeiros 9 estados dum traço cíclico (ou “loop”). Os estados são identificados por 1.1 até 1.9; isto deve ser lido como “os estados 1..9 do traço 1”.

Este segmento de traço é um contra-exemplo porque `bit2.output = TRUE` ocorre no estado 1.8 e o estado 1.9 repete o estado 1.1. Isto indica que o traço é um “loop”, repetindo sucessivamente uma sequência de 8 estados num dos quais se tem `bit2.output = TRUE`.

## EXEMPLO 4.5:

```

MODULE inverter(input)
VAR
    output : boolean;
INIT
    output = TRUE
TRANS
    next(output) = !input | next(output) = output

MODULE main
VAR
    gate1 : inverter(gate3.output);
    gate2 : inverter(gate1.output);
    gate3 : inverter(gate2.output);

SPEC
    AG AF gate1.output

```

□

O exemplo 4.6 ilustra uma FSM não-determinística.

Uma vez mais o sistema descrito é formado por três contadores (“gates”) que aqui estão ligadas “em anel”: o *output* de cada “gate”, que também funciona como estado, é apresentado como *input* da “gate” seguinte.

A FSM é não-determinística uma vez que a transição não é definida por atribuições mas sim por um predicado genérico envolvendo o estado atual e o estado imediatamente seguinte. Note-se que a relação de transição descreve duas evoluções possíveis do estado: ou o próximo estado é a negação do *input* ou então o estado não é alterado.

A especificação exige que o *output* da *gate1* seja válido um número infinito de vezes. Claramente, porque existe a possibilidade de a *gate1* estar bloqueada no valor FALSE, a especificação não será válida. É o que mostra a execução do “model checker”\*.

```

$ nuxmv ring.smv
*** This is nuXmv 1.0.1 (compiled on Mon Nov 17 17:49:50 2014)
*** Copyright (c) 2014, Fondazione Bruno Kessler

-- specification AG (AF gate1.output) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
    gate1.output = TRUE
    gate2.output = TRUE
    gate3.output = TRUE
-- Loop starts here
-> State: 1.2 <-
    gate1.output = FALSE

```

\*O ficheiro *ring.smv* contém a descrição no exemplo 4.6.

```
gate2.output = FALSE
gate3.output = FALSE
-> State: 1.3 <-
```

Os sistemas NUSMV e nuXmv têm a possibilidade de restringir o comportamento aos chamados *traços justos* de tal forma que os quantificadores sobre traços A e E se restrinjam apenas a esses traços.

Genericamente um traço  $\pi$  é justo para uma condição  $C$  se, ao longo desse traço,  $C$  ocorre um número infinito de vezes. Formalmente, usando LTL,  $\pi$  é *justo* para  $C$  quando é válido  $\pi \models \text{GF } C$ .

Em CTL, uma especificação que contenha a condição

```
SPEC A...
FAIRNESS C
```

limita o domínio do quantificador A (e do quantificador E) ao conjunto dos traços que validam  $\text{GF } C$ .

**EXEMPLO 4.6:** Suponhamos que, no exemplo 4.6, se substitui a especificação por

```
SPEC
  AG AF gate1.output
FAIRNESS
  gate1.output
```

Agora a execução do “model checker” é

```
$ nuxmv ring.smv
*** This is nuXmv 1.0.1 (compiled on Mon Nov 17 17:49:50 2014)
*** Copyright (c) 2014, Fondazione Bruno Kessler

-- specification AG (AF gate1.output)  is true
```

Pode-se também ver a influência que tem na especificação a alteração da condição de “fairness” para a saída e um outra **gate**. Por exemplo, alterando a especificação para

```
SPEC
  AG AF gate1.output
FAIRNESS
  gate3.output
```

obtém-se a execução

```
$ nuxmv ring.smv
*** This is nuXmv 1.0.1 (compiled on Mon Nov 17 17:49:50 2014)
*** Copyright (c) 2014, Fondazione Bruno Kessler

-- specification AG (AF gate1.output)  is false
-- as demonstrated by the following execution sequence
```

```

Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
    gate1.output = TRUE
    gate2.output = TRUE
    gate3.output = TRUE
-> State: 1.2 <-
    gate1.output = FALSE
    gate2.output = FALSE
    gate3.output = FALSE
-- Loop starts here
-> State: 1.3 <-
    gate3.output = TRUE
-- Loop starts here
-> State: 1.4 <-
-> State: 1.5 <-

```

□

#### 4.5.2 Verificação em BMC

Num problema BMC pode-se definir à partida um limite no número  $k$  de estados, ou deixar que o sistema tente vários desses limites.

Considere-se o exemplo 4.7 onde se define uma máquina determinística numa teoria de inteiros.

##### [EXEMPLO 4.7:](#)

```

MODULE main
VAR
    y : 0..15;
INIT
    y = 0
TRANS
    (y = 7 -> next(y) = 0) & ( y = 7 | next(y) = ((y + 1) mod 16))
LTLSPEC
    G (y = 4 -> X y=6)

```

□

Note-se que a especificação é uma fórmula LTL A execução do “model checker” usa uma opção adicional `-bmc` para invocar o algoritmo de verificação BMC.

```

$ nuxmv -bmc bmc-couter.smv
*** This is nuXmv 1.0.1 (compiled on Mon Nov 17 17:49:50 2014)
*** Copyright (c) 2014, Fondazione Bruno Kessler

-- no counterexample found with bound 0
-- no counterexample found with bound 1

```

```
-- no counterexample found with bound 2
-- no counterexample found with bound 3
-- no counterexample found with bound 4
-- specification G (y = 4 -> X y = 6)    is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  y = 0
-> State: 1.2 <-
  y = 1
-> State: 1.3 <-
  y = 2
-> State: 1.4 <-
  y = 3
-> State: 1.5 <-
  y = 4
-> State: 1.6 <-
  y = 5
```

Neste caso não se deteta qualquer contra-exemplo com limites  $\leq 4$  no número de estados do traço. Porém, quando se consideram os 5 primeiros estados já se deteta uma situação onde a especificação deixa de ser válida.

**EXEMPLO 4.8:** Suponhamos que a especificação passa a ser

```
LTLSPEC !G F (y=2)
```

□

Neste caso temos a seguinte execução que ilustra um *loop* (8,0).

```
$ nuxmv -bmc bmc-couter.smv
*** This is nuXmv 1.0.1 (compiled on Mon Nov 17 17:49:50 2014)
*** Copyright (c) 2014, Fondazione Bruno Kessler

-- no counterexample found with bound 0
-- no counterexample found with bound 1
-- no counterexample found with bound 2
-- no counterexample found with bound 3
-- no counterexample found with bound 4
-- no counterexample found with bound 5
-- no counterexample found with bound 6
-- no counterexample found with bound 7
-- specification !( G ( F y = 2))    is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  y = 0
-> State: 1.2 <-
  y = 1
-> State: 1.3 <-
```

```

y = 2
-> State: 1.4 <-
y = 3
-> State: 1.5 <-
y = 4
-> State: 1.6 <-
y = 5
-> State: 1.7 <-
y = 6
-> State: 1.8 <-
y = 7
-> State: 1.9 <-
y = 0

```

Pode-se usar este exemplo para ilustrar a verificação de invariantes.

**EXEMPLO 4.9:** Suponhamos que se altera a especificação para

```

INVARSPEC y <= 7
INVARSPEC y in (0..12)

```

□

Então a execução produz

```

$ nuxmv -bmc bmc-counter.smv
*** This is nuXmv 1.0.1 (compiled on Mon Nov 17 17:49:50 2014)
*** Copyright (c) 2014, Fondazione Bruno Kessler

-- invariant y <= 7   is true
-- cannot prove the invariant y in (0 .. 12)  is true or false :
-- the induction fails as demonstrated by the following execution sequence
Trace Description: BMC Failed Induction
Trace Type: Counterexample
-> State: 1.1 <-
y = 12
-> State: 1.2 <-
y = 13

```

# 5

## Programação com Restrições

### 5.1 Introdução

A *programação com restrições* (“Constraint Programming” – CP) é uma importante área da optimização combinatória que cobre uma vasta classe de problemas relevantes às Ciências da Computação. Neste capítulo iremos falar apenas de uma forma particular da programação com restrições designada *programação inteira com restrições* (“Constraint Integer Programming” – CIP) que, como o nome indica, lida com restrições expressas no domínio dos inteiros; esta abordagem estabelece uma ligação directa outras áreas relevantes às Ciências da Computação e que também envolvem inteiros: a *Programação Inteira* (“Integer Programming” – IP) e a *Programação Inteira Híbrida* (“Mixed Integer Programming” – MIP).

As ferramentas de suporte são o sistema SCIP (ver <http://scip.zib.de/>), que é talvez o melhor “solver” CIP no domínio público, e o “wrapper” NUMBERJACK (ver <http://numberjack.ucc.ie/>) que, para além de outros “solvers”, suporta o SCIP como *backend*.

Ilustrativo de um problema CIP típico, e da sua resolução com as ferramentas sugeridas, temos a *alocação temporal* nomeadamente o problema do *horário escolar*. Antes de definir este problema específico, convém referir a um problema mais geral: a *alocação com restrições* (CAP ou “constrained allocation problem”). O CAP é um problema de otimização que se pode definir por

- (i) Um domínio finito  $D$ ; subconjuntos  $X \subseteq D$  designam-se por *alocações*;  $d \in X$  significa que o objecto  $d$  é *alocado em*  $X$ ; o domínio das alocações identifica-se por isso com o *power-set*  $2^D$  ou  $\wp(D)$ . Equivalentemente,  $X$  pode ser visto como uma função inteira  $X: D \rightarrow \{0, 1\}$  (i.e.  $d$  é alocado em  $X$  quando  $X(d) = 1$ ) ou então como um vector  $X \in \{0, 1\}^D$  ( $d$  é alocado quando  $X_d = 1$ ).
- (ii) Um conjunto de *restrições*  $\mathcal{C} \equiv \{\varphi_1, \varphi_2, \dots, \varphi_N\}$ . Cada  $\varphi_i$  é um predicado com alocações como argumentos;  $X$  *satisfaz* a restrição  $\varphi_i$  quando  $\varphi_i(X) = 1$ .
- (iii) Uma “função objetivo”  $\phi: 2^D \rightarrow \mathbb{Q}$ : i.e. uma função racional com alocações como argumentos.

$\phi(X)$  pode representar um “benefício” da alocação  $X$  (eficiência no uso dos recursos) ou, em alternativa, um “custo” de  $X$  (encargos dos clientes). Uma solução do CAP definido pelo triplo  $\langle D, \phi, \mathcal{C} \rangle$  é uma alocação  $X$  que optimiza (maximiza benefícios ou minimiza custos)  $\phi(X)$  e que satisfaz todos os  $\varphi_i \in \mathcal{C}$ .

A forma das restrições e da função objetivo caracteriza o tipo de problema. Um problema da classe *constrained integer programming*, na versão de maximização do objetivo, usa restrições da forma

$$\varphi_i(X) \equiv \#(X \cap C_i) \leq b_i \quad i = 1, \dots, N \quad (5.1)$$

sendo os  $C_i \subseteq D$  alocações constantes e  $b_i$  constantes inteiras positivas. A função objetivo tem forma semelhante: é determinada por uma alocação constante  $C_0$  e

$$\phi(X) \equiv \#(X \cap C_0) \quad (5.2)$$

Vendo as alocações como vectores, o mesmo problema pode ser descrito por

$$\text{maximizar} \quad \sum_{k \in C_0} X_k \quad (5.3a)$$

sujeito às restrições

$$X_k \in \{0, 1\} \quad , \quad \sum_{k \in C_i} X_k \leq b_i \quad i = 1, \dots, N \quad (5.3b)$$

O problema, como está formulado em (5.3), é uma forma particular da *programação inteira* designada por *0/1-programming*.

Na descrição do horário vamos usar quatro domínios finitos  $[1..H]$ ,  $[1..D]$ ,  $[1..S]$  e  $[1..P]$  representando **horas**, **dias**, **salas** e **professores**; o domínio do “constrained allocation problem” será o produto cartesiano  $[1..H] \times [1..D] \times [1..S] \times [1..P]$ ; o tuplo  $d = (h, d, s, p)$  indica uma alocação na hora  $h$ , no dia  $d$ , da sala  $s$  ao professor  $p$ .

Pretende-se maximizar é o número total de alocações sujeito às seguintes restrições definidas em § 5.1.

Note-se que cada uma das alíneas (i) a (v) introduz várias restrições; de facto os quantificadores  $\forall_h, \forall_d, \forall_s \forall_p$  introduzem uma instância da fórmula quantificada para cada uma das combinações de índices  $(h, d, s, p)$  abrangidas pela quantificação.

Todas as restrições têm a forma definida em (5.3) com exceção das definidas em (iii). Estas restrições um quantificador existencial  $\exists T \dots$  para exprimir o fato de existir um  $T$  comum que é igual ao número de tempos letivos de qualquer um dos docentes. Desta forma fixa-se igual carga horária de todos os professores.

Uma implementação usando **Numberjack** e **SCIP** é apresentada no exemplo 5.1. Usou-se parâmetros com dimensão realista (40 tempos semanais, 10 salas e 20 professores) que definem a dimensão do espaço de alocações: vão ser necessárias 8000 ( $= 8 \times 5 \times 10 \times 20$ ) variáveis booleanas para representar todas as alocações possíveis.

Estas variáveis são declaradas num vetor de variáveis booleanas **alloc** com esta dimensão. Para aceder a este vetor usando índices  $h, d, s, p$  definiu-se uma função auxiliar **X**. Adicionalmente é declarada uma variável inteira **T** com valores no intervalo  $[0..número\ máx\ de\ slots]$ .

Note-se que, no sistema **Numberjack** o objetivo a maximizar ou minimizar é escrito como uma forma particular de restrição.

---

### § 5.1 Problema do Horário.

---

**Obs:** nas expressões seguintes, todos os índices são inicializados no valor 1.

- i) Em cada **tempo** – i.e. par **(hora,dia)** – cada **sala** tem alocado quanto muito um **professor**.

$$\forall_{h \leq H} \forall_{d \leq D} \forall_{s \leq S} \cdot \left( \sum_{p \leq P} X_{h,d,s,p} \leq 1 \right)$$

- ii) Em cada **tempo**, cada **professor** é alocado quanto muito a uma **sala**.

$$\forall_{h \leq H} \forall_{d \leq D} \forall_{p \leq P} \cdot \left( \sum_{s \leq S} X_{h,d,s,p} \leq 1 \right)$$

- iii) Todos os professores têm exatamente a mesma carga total de tempos de aula.

$$\exists T \cdot \forall_{p \leq P} \cdot \left( \sum_{h \leq H, d \leq D, s \leq S} X_{h,d,s,p} = T \right)$$

- iv) Em cada dia, o número de horas alocadas a cada professor tem um limite  $C$ .

$$\forall_{d \leq D} \forall_{p \leq P} \cdot \left( \sum_{h \leq H, s \leq S} X_{h,d,s,p} \leq C \right)$$

- v) Em cada dia, alocações do mesmo professor em horas contíguas, são na mesma sala.

Para formalizar esta restrição temos de escrever que, para todo  $d, p$  e todo  $s \neq s'$ , não podem existir duas alocações em tempos contíguos: i.e. se  $(h, d, s, p)$  for alocação, não podem ser alocações  $(h - 1, d, s', p)$  e  $(h + 1, d, s', p)$ . Tendo em atenção os casos especiais em que  $h$  é a primeira ou a última hora do dia, esta restrição escreve-se

$$\forall_{d \leq D} \forall_{p \leq P} \forall_{h < H} \forall_{s < S} \forall_{s' \leq S} \cdot \\ (X_{h,d,s,p} + X_{h+1,d,s',p} \leq 1) \wedge (X_{h,d,s',p} + X_{h+1,d,s,p} \leq 1)$$

- vi) A função objectivo é simplesmente o valor de  $T$ .

Note-se que o número total de alocações é  $T \times P$ ; assim, dado que  $P$  é constante, maximizando o número total de alocações de cada professor maximiza-se o número total de alocações.

---

#### EXEMPLO 5.1:

```
from Numberjack import *

# constantes: número de horas diárias, número de dias, número de salas e número de professores
H = 8 ; D = 5 ; S = 10 ; P = 20
# número total de alocações binárias
N = H*D*S*P

# Variáveis
# 'alloc' é declarada com domínio [0,1]; como tem só 1 dimensão e o horário tem 4,
# a função X aceude a 'alloc' com 4 índices
alloc = VarArray(N,2,'alloc_')
def X(h,d,s,p):
    return alloc[h+H*(d + D*(s + S*p))]
```

```

# carga horária total de cada professor -- declarada como variável
T = Variable(0,H*D,'T')
# carga horaria diaria máxima -- declarada como constante
C = 4

## Inicialização do modelo
horario = Model()

## Restrições.

# Em cada dia e hora cada sala é ocupada 0 ou 1 vezes
for h in range(H):
    for d in range(D):
        for s in range(S):
            horario.add(Sum([X(h,d,s,p) for p in range(P)]) <= 1)

# Em cada dia e hora, cada professor é alocado 0 ou 1 vezes
for h in range(H):
    for d in range(D):
        for p in range(P):
            horario.add(Sum([X(h,d,s,p) for s in range(S)]) <= 1)

# Cada professor tem um número total de horas alocadas T
for p in range(P):
    horario.add(Sum([X(h,d,s,p) for d in range(D) for h in range(H) for s in range(S)]) == T)

# Em cada dia, cada professor tem um limite constante C ao número de horas alocadas.
for d in range(D):
    for p in range(P):
        horario.add(Sum([X(h,d,s,p) for h in range(H) for s in range(S)]) <= C)

# No mesmo dia, alocações contíguas do mesmo professor são na mesma sala.
for d in range(D):
    for p in range(P):
        for h in range(H-1):
            for s in range(S-1):
                for s_ in range(s+1,S):
                    horario.add(X(h,d,s,p) + X(h+1,d,s_,p) <= 1)
                    horario.add(X(h+1,d,s,p) + X(h,d,s_,p) <= 1)

# O objectivo: maximizar o número de horas lecionadas
horario.add(Maximize(T))

## Finalmente, resolver usando 'SCIP'
horario.load('SCIP').solve()

# Query ao modelo
print T.get_value()

```

□

## 5.2 Programação Inteira

### 5.2.1 Definições

A *Programação Inteira* direta estuda alguns dos problemas mais importantes da otimização combinatória; a importância vem essencialmente do fato de muitos dos chamados “problemas difíceis” desta área poderem ser formalizados em programação inteira. Nesta secção apresentamos uma breve introdução à programação inteira, aos problemas com ela relacionados (nomeadamente a “programação inteira híbrida” e a “programação linear”), aos algoritmos para a sua resolução e às suas aplicações. O material aqui descrito é baseado no livro de texto de Conforti et al. (2014) que pode também ser consultado para um estudo mais aprofundado deste tema.

Na sua versão “pura” a programação inteira define-se

---

**§ 5.2** Programação inteira (IP) e programação inteira híbrida (MIP).*Problema da Programação Inteira*

Dada uma matriz de inteiros  $\mathbf{A} \in \mathbb{Z}^{n \times m}$ , o vetor linha  $\mathbf{b} \in \mathbb{Z}^m$  e o vetor coluna  $\mathbf{c} \in \mathbb{Z}^n$  determinar um vetor de racionais  $x \in \mathbb{Q}^n$  que maximiza  $\phi \equiv x \mathbf{c}$  sujeito a

- (i) Verifica-se  $\sum_{i=1}^n x_i \mathbf{A}_{ij} \leq \mathbf{b}_j$  para todo  $j = 1..m$ ; i.e.  $x \mathbf{A} \leq \mathbf{b}$ .
- (ii) Todo o  $x_i$  é não-negativo; i.e.  $x \geq 0$ .
- (iii) Todo o  $x_i$  é inteiro; i.e.  $x \in \mathbb{Z}^n$ .

Representamos por **IP(A, b)** o conjunto de todos os  $x \in \mathbb{Q}^n$  que verificam estas restrições (i) a (iii). Cada  $x \in \mathbf{IP}(\mathbf{A}, \mathbf{b})$  designa-se por *solução viável* do problema.

*Problema da Programação Inteira Híbrida*

Os parâmetros são os mesmos que em IP: a matriz  $\mathbf{A}$  e os vetores  $\mathbf{b}$  e  $\mathbf{c}$ ; é ainda parâmetro um subconjunto de índices  $I \subseteq [1..n]$ . Pretende-se determinar um vetor de racionais  $x \in \mathbb{Q}^n$  que maximiza  $\phi \equiv x \mathbf{c}$  sujeito a

- (i) Verifica-se  $\sum_{i=1}^n x_i \mathbf{A}_{ij} \leq \mathbf{b}_j$  para todo  $j = 1..m$ ; i.e.  $x \mathbf{A} \leq \mathbf{b}$ .
- (ii) Todo o  $x_i$  é não-negativo; i.e.  $x \geq 0$ .
- (iii) Para todo  $i \in I$ ,  $x_i$  é inteiro.

Representamos por **MIP(A, b, I)** o conjunto de todos os  $x \in \mathbb{Q}^n$  que verificam estas restrições (i) a (iii). Analogamente, os elementos deste conjunto designam-se por *soluções viáveis* do problema MIP.

---

**□1: Notas**

Os operadores  $\leq$ ,  $=$  e  $\geq$  aplicadas a vetores, agem componente a componente; por exemplo, verifica-se  $x \leq y$  quando se verifica  $x_i \leq y_i$  para todo  $i$ .

A matriz  $\mathbf{A}$  e os vetores  $\mathbf{b}$  e  $\mathbf{c}$  são os parâmetros do problema; na resolução do problema IP são vistos como constantes. A incógnita do problema IP é o vetor  $x$ ; na sua resolução  $x$  é visto como uma sequência de  $n$  variáveis inteiros.

O MIP tem uma formalização muito semelhante à do IP; a única diferença substancial está na restrição (iii). No MIP só algumas variáveis  $x_i$ , aquelas com índice  $i \in I$ , têm de ser instanciadas com um valor que satisfaça a restrição “ser inteiro”; as restantes podem ser instanciadas com qualquer valor racional. □

É óbvio que IP é o caso particular de MIP correspondente à situação em que  $I$  coincide com a totalidade do intervalo  $[1..n]$ ; isto é, todos os  $x_i$  têm de ser inteiros. Várias outras variantes do MIP têm importância própria. Nomeadamente

1) *Programação Linear*

Forma particular do MIP em que  $I = \emptyset$ ; isto é, todos os  $x_i$  podem ser racionais.

2) *Programação 0/1 Híbrida*

Forma particular do MIP que se obtém juntando, para cada  $i$ , a restrição  $x_i \leq 1$ .

Como consequência, para todo  $i \in I$ ,  $x_i$  está limitada aos valores 0 e 1.

3) *Programação 0/1*

Programação inteira com a restrição de todo  $x_i$  estar limitado aos valores 0 e 1.

Todos estes problemas, com exceção da “programação linear” (LP) são considerados “difíceis”; de fato verifica-se que todos eles estão na classe de complexidade NP.

Ao invés o problema LP pode ser resolvido de forma muito eficiente; existem algoritmos\* que mostram, em teoria e na prática corrente, que o problema está na classe de complexidade P.

Este fato leva-nos a pensar em que medida é possível aproximar uma solução do MIP variando o conjunto de índices  $I$  permitindo que mais valores  $x_i$  não sejam necessariamente inteiros.

### 5.2.2 Algoritmos de MIP

Dados problemas MIP definidos por tuplos de parâmetros  $P' = \langle \mathbf{A}', \mathbf{b}', \mathbf{c}', I' \rangle$  e  $P = \langle \mathbf{A}, \mathbf{b}, \mathbf{c}, I \rangle$ , diz-se que  $P'$  é uma *relaxação* de  $P$ , e escreve-se  $P' \supseteq P$ , quando toda a solução viável de  $P'$  é uma solução viável de  $P$ . Isto é,

$$\mathbf{MIP}(\mathbf{A}', \mathbf{b}', I') \supseteq \mathbf{MIP}(\mathbf{A}, \mathbf{b}, I)$$

A forma mais simples de relaxação, designada por *relaxação natural*, consiste em eliminar completamente a restrição “ser inteiro” nas várias componentes  $x_i$ , mantendo o resto dos parâmetros. Formalmente é a relaxação

$$\langle \mathbf{A}, \mathbf{b}, \mathbf{c}, \emptyset \rangle \supseteq \langle \mathbf{A}, \mathbf{b}, \mathbf{c}, I \rangle \quad (5.4)$$

O problema  $\langle \mathbf{A}, \mathbf{b}, \mathbf{c}, \emptyset \rangle$  é um problema de programação linear que, por isso, tem uma implementação eficiente. Assim uma estratégia para resolver o problema MIP pode passar por tentar resolver a sua relaxação natural e tentar descobrir uma forma de poder usar a solução do problema relaxado como uma aproximação da solução do problema inicial. Por aproximações sucessivas poderemos, eventualmente, construir a solução desejada.

#### O método “branch-and-bound”

O método “branch-and-bound” (também conhecido por “split-and-join”) aplica-se a uma grande variedade de problemas de otimização combinatória. Essencialmente e em termos gerais, a estratégia do algoritmo consiste, em cada passo, dividir o espaço de soluções em dois e aplicar o mesmo processo a cada um destes sub-espacos. Existe depois um simples mecanismo de escolha para construir a solução do problema inicial a partir das soluções dos dois sub-problemas. As especificidades do problema decidem em que circunstâncias este “branch/split” é executado e dizem também em que circunstâncias um dos sub-problemas tem uma solução óbvia.

Sem perda de generalidade, vamos ilustrar o método com um problema de programação 0/1 híbrida  $P \equiv \langle \mathbf{A}, \mathbf{b}, \mathbf{c}, I \rangle$ . O algoritmo aqui apresentado é trivialmente estendido a qualquer problema MIP.

---

\*O primeiro destes algoritmos foi publicado em 1947 por G.B.Dantzig e designa-se por “método SIMPLEX” ou “método de Dantzig”.

---

**§ 5.3** “Branch-and-Bound”: programação 0/1 híbrida  $P \equiv \langle \mathbf{A}, \mathbf{b}, \mathbf{c}, I \rangle$ .
 

---

Como “inputs” do algoritmo são dados os parâmetros do problema  $\langle \mathbf{A}, \mathbf{b}, \mathbf{c}, I \rangle$  e um limite inferior para um valor ótimo da função objetivo  $\ell$ .

- (1) Constrói-se a relaxação natural  $P' \equiv \langle \mathbf{A}, \mathbf{b}, \mathbf{c}, \emptyset \rangle$  e resolve-se este problema LP. Se não existe uma solução ou se existe solução mas o valor ótimo  $z$  verifica  $z < \ell$ , termina com a mensagem **unsat**.
  - (2) Em caso contrário, seja  $\bar{x}$  a solução do problema LP. Escolhe-se  $i \in I$  tal que a componente  $\bar{x}_i$ , é fracionária (i.e.  $\neq 0, 1$ ). Se tal  $i$  não existir, é porque a solução do problema relaxado coincide com a solução do problema  $P$ ; assim o algoritmo termina com a mensagem **sat**, a solução  $\bar{x}$  e valor ótimo  $z$ .
  - (3) Se  $i$  existe, constrói-se problemas  $P_0 = P \cap \{x_i = 0\}$  e  $P_1 = P \cap \{x_i = 1\}$  que se obtêm juntando a  $P$ , respetivamente, a restrição  $(x_i = 0)$  e a restrição  $(x_i = 1)$ . Os espaços de soluções viáveis de  $P_0$  e  $P_1$  são disjuntos (devido ao valor de  $x_i$ ); porém a sua união forma o espaço de soluções viáveis de  $P$ . Assim,
  - (4) Recorre-se a este algoritmo para resolver recursivamente um dos sub-problemas  $P_j$  ( $j = 0, 1$ ) com o “bound”  $\ell$ .
  - (5) Se a invocação anterior termina com a mensagem **unsat**, usar este algoritmo no outro sub-problema  $P_{1-j}$ , também com o “bound”  $\ell$ .
  - (6) Se a invocação em (4) termina com a mensagem **sat**, e o valor ótimo  $z^*$ , invocar o algoritmo para resolver o sub-problema  $P_{1-j}$  com o “bound”  $z^*$ .
  - (7) Se a invocação (6) termina com **unsat**, devolver como resultado o resultado de (4). Senão devolver como resultado, o resultado de (6).
- 

No pior caso este algoritmo terá complexidade exponencial porque, em cada invocação podem ocorrer duas invocações recursivas. No entanto existe uma probabilidade elevada de algumas destas invocações recursivas terminar rapidamente logo nos passos (1) ou (2).

Um outro aspeto importante diz respeito à paralelização. Dado que as duas invocações nos passos (4) e (6) (a menos do “bound”) não partilham qualquer estado nem têm relações de dependência temporal, elas podem correr em máquinas distintas. Existe por isso um elevado potencial de paralelização neste algoritmo.

Várias heurísticas são possíveis na escolha do índice  $i$  em (2), ou na escolha do sub-problema  $j$  em (4). Escolhas apropriadas podem ainda melhorar a eficiência do algoritmo.

### O método do plano de corte

Tal como o método “branch-and-bound”, o método do plano de corte aplicado a um problema  $P \equiv \langle \mathbf{A}, \mathbf{b}, \mathbf{c}, I \rangle$  também começa por construir uma solução LP  $(\bar{x}, \bar{z})$  da relaxação natural  $P' \equiv \langle \mathbf{A}, \mathbf{b}, \mathbf{c}, \emptyset \rangle$ . Ambos os métodos verificam, em seguida, se todos  $\bar{x}_i$ , com  $i \in I$ , são inteiros e, caso sejam, devolvem  $(\bar{x}, \bar{z})$  como solução de  $P$ .

Os dois métodos divergem quando esta verificação falha. O método do plano do

corte constrói um plano  $\{x \mid \sum_{i=1}^n \alpha_i x_i = \alpha_0\}$  (com todos  $\alpha_i$  não necessariamente inteiros) que “separa” o conjunto de soluções viáveis de  $P$  da solução “aproximada”  $\bar{x}$ . Isto é, as soluções viáveis do problema pertencem a um dos sub-espacos definido pelo plano de corte (por exemplo,  $\{x \mid x\alpha < \alpha_0\}$ ) enquanto que a solução aproximada  $\bar{x}$  pertence ao seu complemento  $\{x \mid x\alpha \geq \alpha_0\}$

Formalmente as soluções viáveis de  $P$  formam o subconjunto  $S \subseteq \mathbb{Q}^n$  dado por

$$S \equiv \{x \mid (x\mathbf{A} \leq \mathbf{b}) \wedge (x \geq 0) \wedge (\forall i \in I \cdot x_i \text{ é inteiro})\}$$

Então o plano de corte  $x\alpha = \alpha_0$  tem de verificar ambas as condições

$$x \in S \Rightarrow (x\alpha < \alpha_0) \quad \text{e} \quad \bar{x}\alpha \geq \alpha_0 \quad (5.5)$$

Existem várias estratégias possíveis para definir os planos de corte. A mais óbvia consiste em tomar a solução ótima  $z$  do problema relaxado, que verifica a igualdade  $\bar{x}\mathbf{c} = z$ , e definir o plano  $x\mathbf{c} \leq [z]$ . No entanto pode acontecer que a primeira condição  $x \in S \Rightarrow x\mathbf{c} \leq [z]$  não se verifique; por isso pode ser necessário outras formas de plano de corte.

Independentemente da estratégia para selecionar o plano de corte, a estrutura global do algoritmo é sempre a mesma e está ilustrada em § 5.4.

---

#### § 5.4 Método do plano de corte.

---

- (1) Construir uma solução  $\bar{x}$  do problema relaxado. Senão existir tal solução terminar com a mensagem **unsat**.
  - (2) Verificar se todos os valores  $\bar{x}_i$ , com  $i \in I$ , são inteiros. Se forem, terminar com  $\bar{x}$  como solução e a mensagem **sat**.
  - (3) Senão, gerar um plano de corte  $x\alpha = \alpha_0$  que verifique (5.5). Adicionar  $x\alpha \leq \alpha_0$  às restrições do problema.
  - (4) Repetir o processo a partir de (1).
- 

#### Equações e Inequações Lineares

Um problema MIP com um sistema de inequações  $x\mathbf{A} \leq \mathbf{b}$  e uma função objetivo  $x\mathbf{c}$  pode ser sempre reescrito só com equações. Em primeiro lugar introduz-se uma variável  $x_0$ , que representa o valor da função objetivo, e a equação

$$-x_0 + x\mathbf{c} = 0$$

Em sequida introduzem-se novas variáveis  $s_1, \dots, s_m$  (uma para cada inequação) e substitui-se as inequações  $x\mathbf{A} \leq \mathbf{b}$  pelas equações

$$s + x\mathbf{A} = \mathbf{b}$$

Globalmente o novo sistema de equações fica

$$\begin{bmatrix} s & x_0 & x \end{bmatrix} \begin{bmatrix} 0 & \mathbf{I} \\ -1 & 0 \\ \mathbf{c} & \mathbf{A} \end{bmatrix} = \begin{bmatrix} 0 & \mathbf{b} \end{bmatrix}$$

Para todas as variáveis  $v \in \{s_1, \dots, s_m, x_0, x_1, \dots, x_n\}$  temos a restrição  $v \geq 0$ . Finalmente, para toda a variável  $u \in \{x_i \mid i \in I\} \cup \{s_j \mid j = 1..m\}$ , temos a restrição “ser inteiro”; formalmente  $u = \lfloor u \rfloor$ .

Pode-se reformular o problema MIP de forma a se ter só equações e ter variáveis como objetivo. A forma genérica é agora

- É dada uma matriz  $\mathbf{A} \in \mathbb{Z}^{n \times m}$  e um vetor  $\mathbf{b} \in \mathbb{Z}^m$  ambos de componentes inteiras. São definidos conjuntos de índices:  $O, I \subseteq [1..n]$  com  $O \cap I = \emptyset$ .
- São *soluções viáveis* os vetores  $x \in \mathbb{Q}^n$  que verificam as restrições
  - $x \mathbf{A} = \mathbf{b}$
  - $x \geq 0$
  - $x_i = \lfloor x_i \rfloor$  para todo  $i \in I$ .
- A solução ótima  $\bar{x}$  é a solução viável que maximiza o objetivo  $\max_{j \in O} x_j$ .

Sopunhamos que, por eliminação gaussiana, se transforma o sistema de equações lineares num outro sistema equivalente da forma

$$x' + x'' \mathbf{A}' = \mathbf{b}'$$

em que  $(x', x'')$  é uma partição do vetor de variáveis  $x$ , e em que  $x'$  tem  $m$  variáveis e  $x''$  tem as restantes  $n - m$  variáveis. Vamos também supor que a eliminação gaussiana foi feita de modo a se verificar  $b' \geq 0$ ; como resultado as componentes de  $\mathbf{A}'$  e de  $\mathbf{b}'$  podem não ser inteiros. Se tomarmos  $x' = b'$  e  $x'' = 0$ , temos uma solução viável do problema LP que resulta da relaxação do problema MIP original.

Se esta solução não for uma solução viável MIP, é necessário encontrar, por exemplo, um plano de corte que a separe do conjunto de soluções MIP. Para isso procura-se, no conjunto de equações  $x' + x'' \mathbf{A}' = b'$  uma equação

$$\sum_i x_i a_i = a_0 \tag{5.6a}$$

onde  $a_i = 0$  para todo  $i \notin I$ ,  $a_0 \neq \lfloor a_0 \rfloor$  e

$$x_i \in b'' \wedge i \in I \Rightarrow a_i \neq \lfloor a_i \rfloor \tag{5.6b}$$

É trivial verificar que uma solução viável MIP que satisfaça (5.6a) também verifica a restrição

$$\sum_i x_i (a_i - \lfloor a_i \rfloor) \geq a_0 - \lfloor a_0 \rfloor \tag{5.6c}$$

Por outro lado (5.6b) implica que a solução  $(x', x'')$  não verifica esta nova restrição. Portanto pode-se usar (5.6c) como plano de corte.

**EXEMPLO 5.2:** Considere-se o problema definido pelas restrições

$$\begin{cases} x_1 + 2x_2 & \leq 5 \\ 2x_1 - x_2 & \leq 3 \end{cases} \quad x_1, x_2 \geq 0 \quad x_1, x_2 \text{ inteiros}$$

e pela função objetivo  $\phi \equiv 3x_1 + 2x_2$ .

Para construir uma solução LP do problema relaxado, começamos por introduzir variáveis que transformem o objetivo numa equação  $x_0 - 3x_1 - 2x_2 = 0$ , e “slack variables”  $x_3, x_4$  que transformem as inequações em equações; o objetivo é agora maximizar  $x_0$ .

$$\begin{cases} x_0 - 3x_1 - 2x_2 = 0 \\ x_1 + 2x_2 + x_3 = 5 \\ 2x_1 - x_2 + x_4 = 3 \end{cases}$$

Por eliminação gaussiana obtém-se o sistema equivalente

$$\begin{cases} x_0 + 1.4x_3 + 0.8x_4 = 9.4 \\ x_1 + 0.2x_3 + 0.4x_4 = 2.2 \\ x_2 + 0.4x_3 - 0.2x_4 = 1.4 \end{cases}$$

que fornece a solução  $\{x_0 = 9.4, x_1 = 2.2, x_2 = 1.4, x_3 = x_4 = 0\}$ .

Como  $x_1$  e  $x_2$  não são inteiros, esta não é uma solução viável do problema MIP. Tomemos uma qualquer das equações onde só ocorram variáveis inteiras; por exemplo

$$x_1 + 0.2x_3 + 0.4x_4 = 2.2 \quad (5.7a)$$

Aplicando a transformação (5.6c) obtém-se  $0.2x_3 + 0.4x_4 \geq 0.2$  que é equivalente a

$$x_3 + 2x_4 \geq 1 \quad (5.7b)$$

A solução do problema relaxado não verifica esta inequação (porque  $x_3 = x_4 = 0$ ); no entanto qualquer solução inteira de (5.7a) verifica (5.7b). Portanto esta inequação define um plano de corte que separa a solução LP das soluções viáveis do problema MIP.

O algoritmo do plano de corte adiciona esta inequação às restrições; para a transformar numa equação tem de introduzir uma nova variável inteira positiva  $x_5$  e escrevê-la como

$$x_3 + 2x_4 - x_5 = 1, \quad x_5 \geq 0, \quad x_5 \text{ inteiro}$$

Obtém-se um novo sistema

$$\begin{cases} 5x_0 + 7x_3 + 4x_4 = 47 \\ 5x_1 + x_3 + 2x_4 = 11 \\ 5x_2 + 2x_3 - x_4 = 7 \\ x_3 + 2x_4 - x_5 = 1 \end{cases}$$

e o processo repete-se até que a solução do problema relaxado tenha valores inteiros para todos  $x_i$ , com  $i \neq 0$ .  $\square$

## 5.3 Alguns Problemas em Programação Inteira

Nesta secção vamos ver alguns problemas clássicos que se exprimem em programação inteira ou em programação inteira híbrida.

### 5.3.1 O problema da mochila (“knapsack problem”)

Dada uma coleção de  $N$  tipos de objetos com “pesos”  $w_1, w_2, \dots, w_N$  em que os  $w_i$  são racionais positivos e não nulos. Sem perda de generalidade pode-se assumir que estes pesos estão normalizados: i.e.  $\sum_{i=1}^N w_i = 1$ .

Cada tipo de objetos tem um valor  $c_i \geq 0$ . Assumimos também que estes valores estão normalizados:  $\sum_{i=1}^N c_i = 1$ . Pretende-se maximizar o valor total dos objetos transportados numa mochila (“knapsack”) sabendo que a mochila tem um limite  $W$  no peso total que pode suportar.

As variáveis  $x_1, \dots, x_N$  denotam o número de objetos de cada tipo. A formulação do problema está ilustrada em § 5.5.

---

### § 5.5 “Knapsack Problem”.

---

Maximizar  $\sum_{i=1}^N c_i x_i$  sujeito às restrições

- (i)  $\sum_{i=1}^N w_i x_i \leq W$
  - (ii)  $x_i \geq 0$  e  $x_i$  é inteiro para todo  $i = 1..N$ .
- 

Uma variante deste problema, designado por “bin packing” ou 0/1-“knapsack”, impõe que o número de itens  $x_i$  seja sempre 0 ou 1; adicionalmente os valores  $c_i$  são todos iguais. O conjunto das soluções viáveis do problema “bin packing”  $K \equiv \langle w_1, \dots, w_n; W \rangle$  é

$$\hat{K} \equiv \{x \in \{0,1\}^N \mid \sum_{i=1}^N x_i w_i \leq W\} \quad (5.8)$$

Assim o problema resume-se a enumerar  $\hat{K}$  ou, equivalentemente, todos os conjuntos  $S \subseteq [1..N]$  tais que  $\sum_{i \in S} w_i \leq W$ .

O conjunto  $S \subseteq [1..N]$  é uma *cobertura* do problema  $K$  quando  $W < \sum_{i \in S} w_i$ . Se adicionalmente nenhum  $S \setminus \{i\}$ , com  $i \in S$ , é cobertura de  $K$  então  $S$  é uma *cobertura mínima*. Isto é, para todo  $j \in S$

$$W < \sum_{i \in S} w_i \leq W + w_j \quad (5.9)$$

Cada cobertura mínima  $S$  define uma restrição  $\hat{S}$  definida como

$$\hat{S} \equiv \{x \in \{0,1\}^N \mid \sum_{i \in S} x_i \leq \#S - 1\} \quad (5.10)$$

É simples provar o seguinte resultado:

**5.1 PROPOSIÇÃO** *O conjunto de soluções viáveis do problema  $K$ , i.e.  $\hat{K}$ , coincide com a intersecção de todos os  $\hat{S}$  quando  $S$  percorre as coberturas mínimas de  $K$ .*

Este resultado diz-nos que o mesmo problema “bin packing” pode ser formulado de duas formas distintas.

**EXEMPLO 5.3:** Considere-se o problema  $K \equiv \langle 1, 1, 2, 3; 6 \rangle$ . A sua formulação direta é

$$x_1 + x_2 + 2x_3 + 3x_4 \leq 6$$

O problema tem uma única cobertura mínima  $S \equiv [1..4]$  de cardinalidade 4; este  $S$  dá origem a uma restrição

$$x_1 + x_2 + x_3 + x_4 \leq 3$$

A proposição 5.1 diz-nos que as restrições têm o mesmo conjunto de soluções viáveis.  $\square$

### 5.3.2 Problemas do empacotamento, cobertura e partição

Seja  $A \in \{0,1\}^{n \times m}$  uma matriz de elementos 0, 1 com  $n$  linhas e  $m$  colunas. Geneticamente tais matrizes chamam-se *matrizes de incidência*. Sejam  $E \equiv \{1, \dots, n\}$  e  $F \equiv \{1, \dots, m\}$  os conjuntos de  $n$  e de  $m$  índices.

Define-se, para cada  $j \in F$ , o conjunto  $E_j \equiv \{i \in E \mid A_{ij} = 1\}$ ; dualmente, para cada  $i \in E$ , define-se  $F_i \equiv \{j \in F \mid A_{ij} = 1\}$ .

Um subconjunto  $S \subseteq E$  é uma *empacotamento* de  $A$  quando se verifica

$$\forall j \in F \cdot \#(S \cap E_j) \leq 1$$

Isto é:  $S$  intersecta cada  $E_j$  não mais do que uma vez. Diz-se que  $S$  é uma *cobertura* de  $A$  quando  $S$  intersecta cada  $E_j$  uma ou mais vezes; finalmente quando  $S$  intersecta cada  $E_j$  exatamente uma vez, diz-se que  $S$  é uma *partição* de  $A$ .

Em termos de restrições temos, para todo  $j \in F$ ,

$$\begin{cases} \text{empacotamento} & \sum_{i \in S} A_{ij} \leq 1 \\ \text{cobertura} & \sum_{i \in S} A_{ij} \geq 1 \\ \text{partição} & \sum_{i \in S} A_{ij} = 1 \end{cases} \quad (5.11a)$$

Dualmente temos as noções de *co-empacotamento*, *co-cobertura* e *co-partição*. Diz-se que  $C \subseteq F$  é, em relação a  $A$ , um

$$\begin{cases} \text{co-empacotamento} & \sum_{j \in C} A_{ij} \leq 1 \\ \text{co-cobertura} & \sum_{j \in C} A_{ij} \geq 1 \\ \text{co-partição} & \sum_{j \in C} A_{ij} = 1 \end{cases} \quad (5.11b)$$

para todo  $i \in E$ . Obviamente um co-empacotamento de  $A$  é um empacotamento da matriz transposta de  $A$ ; o mesmo se passa em relação às outras duas noções.

Se representarmos  $S \subseteq E$  e  $C \subseteq F$  pelas suas funções características  $x \in \{0,1\}^n$  e  $y \in \{0,1\}^m$  com  $x_i = 1$  sse  $i \in S$  e  $y_j = 1$  sse  $j \in C$ , então tem-se

$$\begin{cases} \text{empacotamento / co-empacotamento} & x A \leq 1 \quad / \quad y A^\top \leq 1 \\ \text{cobertura / co-cobertura} & x A \geq 1 \quad / \quad y A^\top \geq 1 \\ \text{partição / co-partição} & x A = 1 \quad / \quad y A^\top = 1 \end{cases} \quad (5.11c)$$

Muitos problemas práticos e fundamentais da optimização combinatória podem ser expressos como empacotamentos, coberturas ou partições. Por exemplo,

---

#### § 5.6 “Point/Set Cover Problems”.

Dado um universo  $U \equiv \{u_1, \dots, u_n\}$ , uma família  $\mathcal{S} \equiv \{S_1, \dots, S_m\}$  de subconjuntos de  $U$  e uma função custo  $c: \mathbb{N}_+ \rightarrow \mathbb{Q}_+$ , então

**“point cover problem”** Determinar coleção de pontos  $Q \subseteq U$ , de menor custo total, que cobre  $\mathcal{S}$ ; isto é, cada  $S \in \mathcal{S}$  contém um dos elementos de  $Q$ .

**“set cover problem”** Determinar a coleção de subconjuntos  $Q \subseteq \mathcal{S}$ , de menor custo total, que cobre  $U$ ; isto é, cada  $u \in U$  está contido num dos  $S \in Q$ .

**Nota:** A função custo  $c$  pode também ser vista como uma sequência infinita de racionais positivos  $c \equiv c_1, c_2, \dots, c_k, \dots$ . No “point cover” o custo de cada  $u_i$  é dado por  $c_i$ ; no “set cover” o custo de cada  $S_j$  é  $c_j$ ; o custo total da coleção  $Q$  (pontos ou subconjuntos) é a soma destes custos individuais estendida aos elementos de  $Q$ . □

Estes problemas exprimem-se facilmente na formalização que apresentamos em (5.11). Nomeadamente

A matriz incidência  $A \in \{0, 1\}^{n \times m}$  define-se naturalmente como

$$A_{ij} = 1 \Leftrightarrow u_i \in P_j$$

Representamos cada coleção  $Q$  (de pontos ou de conjuntos) por uma sequência infinita  $x: \mathbb{N}_+ \rightarrow \{0, 1\}$  de tal forma que

$$x_i = 1 \Leftrightarrow u_i \in Q \quad (\text{pontos}) \quad , \quad x_j = 1 \Leftrightarrow S_j \in Q \quad (\text{conjuntos})$$

Em qualquer dos casos o custo total da coleção é  $\sum_{i>0} x_i c_i$ .

Uma coleção de pontos  $Q$  cobre  $\mathcal{S}$  quando cada  $S_j \in Q$  contém pelo menos um ponto em  $Q$ ; isto é, quando  $\forall j \cdot \exists u_i \in Q \cdot u_i \in S_j$ . Equivalentemente

$$\forall j \cdot \sum_i x_i A_{ij} \geq 1 \quad \text{ou} \quad x A \geq 1$$

Portanto o problema “point cover” é um problema de cobertura da matriz  $A$  tal como está definido em (5.11c).

No problema “set cover”, a coleção  $Q \subseteq \mathcal{P}$  é tal que todo o ponto  $u_i$  tem de estar contido em algum  $S_j \in Q$ . Isto é, verifica-se  $\forall i \cdot \exists S_j \in Q \cdot u_i \in S_j$ . Equivalentemente  $\forall i \cdot \sum_{S_j \in Q} A_{ij} \geq 1$ , ou seja

$$\forall i \cdot \sum_j x_j A_{ij} \geq 1 \quad \text{ou} \quad x A^\top \geq 1$$

Assim o problema “set cover” é equivalente ao problema co-cobertura da matriz  $A$ .

De forma análoga os problemas de empacotamento co-empacotamente de uma matriz de incidência são equivalentes a dois problemas clássicos: o “point packing problem” e o “set packing problem”.

### § 5.7 Problemas “Point Packing” e “Set Packing”.

Dado um universo  $U \equiv \{u_1, \dots, u_n\}$ , uma família  $\mathcal{S} \equiv \{S_1, \dots, S_m\}$  de subconjuntos de  $U$  e uma função custo  $c: \mathbb{N}_+ \rightarrow \mathbb{Q}_+$ , então

#### “point pack”

Determinar a coleção de pontos  $Q$  que maximiza o benefício  $\sum_i x_i c_i$  e é tal que qualquer  $S_j \in \mathcal{S}$  contém quanto muito um ponto  $u_i \in Q$ .

#### “set pack”

Determinar a coleção de subconjuntos  $Q \subseteq \mathcal{S}$  que maximiza o benefício  $\sum_i x_i c_i$  e é tal que qualquer  $u_i \in U$  está contido quanto muito num  $S_j \in Q$ .

Tal como na situação anterior prova-se que o problema “point packing” é um problema de empacotamento da matriz  $A$  (i.e. maximizar  $x \cdot c$  sujeito a  $x A \leq 1$ ) enquanto que “set packing” é um problema de co-empacotamentos de  $A$  (i.e. maximizar  $x \cdot c$  sujeito a  $x A^\top \leq 1$ ).

**EXEMPLO 5.4:** Exemplo de um problema de *point packing*.

- (i) Um comboio é usado para transportar *cargas* dos tipos  $u_1, \dots, u_N$ .
- (ii) O comboio usa *vagões* diferentes  $s_1, \dots, s_M$  que estão habilitados para transportar tipos de cargas distintos;  $S_j$  é o conjunto de tipos de cargas para as quais o vagão  $v_j$  está habilitado.
- (iii) Em cada *viagem* cada vagão só pode transportar carga de um dos tipos que está habilitado a transportar.
- (iv) Pretende-se determinar o maior número de tipos de carga distintos que pode transportar numa única viagem.

O domínio dos tipos de carga é representado pelo intervalo inteiro  $[1..N]$ ; o domínio dos vagões é representado pelo intervalo  $[1..M]$ . A matrix de incidência é representado pelo array constante  $A$  segundo a regra “ $A[j][i] == 1$  se e só se tipo  $i$  pode ser transportado pelo vagão  $j$ ”; i.e. o array  $A[j]$  representa o conjunto de todos os tipos que podem ser transportados pelo  $j$ -ésimo vagão. O array de variáveis  $X$  será instanciado com o conjunto de tipos que vai ser transportado numa viagem.

Como cada vagão não pode transportar, numa viagem, mais do que um tipo de carga temos a restrição  $\sum_i A_{ji} X_i \leq 1$  para cada  $j$ ; equivalente à restrição  $\#(A_j \cap X) \leq 1$ . A função objetivo é a cardinalidade do conjunto  $X$ ,  $\#X$ , implementado como  $\sum_i X_i$ .

Na implementação seguinte a matriz de incidência  $A$  é construída gerando os seus elementos aleatoriamente com a distribuição de Bernoulli  $\mathbb{B}_\varepsilon$ ,  $\varepsilon = 0.2$ . Isto significa que, em média, cada conjuntos  $A_j$  tem  $\varepsilon \times N$  elementos.

```
from Numberjack import *
import random as rn
# Número de tipos de carga
N = 40
# Número de vagões
M = 20

# Gerador de Bernoulli
bias = 0.2
def bernoulli():
    return 0 if rn.random() > bias else 1

# Matrix de incidência -- aleatória
A = [[bernoulli() for i in range(N)] for j in range(M)]

# Varáveis, Modelo e Restrições
X = VarArray(N, 2, 'X')

tipos = Model()
for j in range(M):
    tipos.add(Sum(X, A[j]) <= 1)
    tipos.add(Maximize(Sum(X)))

OK = tipos.load('SCIP').solve()
if OK:
    print [i for i in range(N) if X[i].get_value() == 1]
```

□

### 5.3.3 Problemas de Grafos

Alguns problemas importantes da teoria de grafos podem-se exprimir em problemas de empacotamento de pontos ou conjuntos.

Recordemos os elementos base da formulação em § 5.6. Temos um universo de “pontos”  $U = \{u_1, \dots, u_n\}$  e uma família de sub-conjuntos de  $U$ ,  $\mathcal{S} = \{S_1, \dots, S_m\}$ . Temos custos  $c_i > 0$  associados a pontos  $u_i$  mas também custos  $c_j$  associados aos conjuntos  $S_j$ . A “cobertura de pontos” procura a família de pontos  $Q \subseteq U$  de menor custo que cobre  $\mathcal{S}$  (i.e. cada  $S \in \mathcal{S}$  contém pelo menos um elemento de  $Q$ ). A “cobertura de conjuntos” procura a sub-família de conjuntos  $Q \subseteq \mathcal{S}$  de menor custo que cobre  $U$  (i.e. cada  $u \in U$  está contido num dos elementos de  $Q$ ).

Nos grafos estas ideias aplicam-se a dois universos: *vértices* e *arestas*. Recorde-se que um *grafo*  $G \equiv (V, A)$  (nesta secção considera-se só grafos finitos) é definido por um conjunto finito  $V$  de *vértices* (ou *nodos*) e um conjunto finito  $A$  de *arestas* (ou *ramos*). Cada aresta é identificada por um par de vértices: a origem  $s$  (“source”) e o destino  $t$  (“target”); por isso  $A$  identifica-se com um subconjunto de  $V \times V$ .

Um *caminho* no grafo  $G = (V, A)$  com origem  $s$  e destino  $t$ , é um conjunto de arestas  $P \subseteq A$  tal que

$$\overrightarrow{ab} \in P \rightarrow \left( a = s \vee \exists a' . \overrightarrow{a'a} \in P \right) \wedge \left( b = t \vee \exists b' . \overrightarrow{bb'} \in P \right) \quad (5.12)$$

Um caminho é *mínimo* quando não contém, como sub-conjunto próprio, um outro caminho com a mesma origem e destino.

A figura 5.1 representa um grafo com 8 vértices e 10 arestas; o conjunto de vértices é  $V \equiv \{v_0, \dots, v_7\}$ . A aresta genérica de origem  $v_i$  e destino  $v_j$  designa-se por  $\overrightarrow{v_i v_j}$ . O grafo é definido por  $V$  e pelo conjunto  $A$  formado por tais arestas.

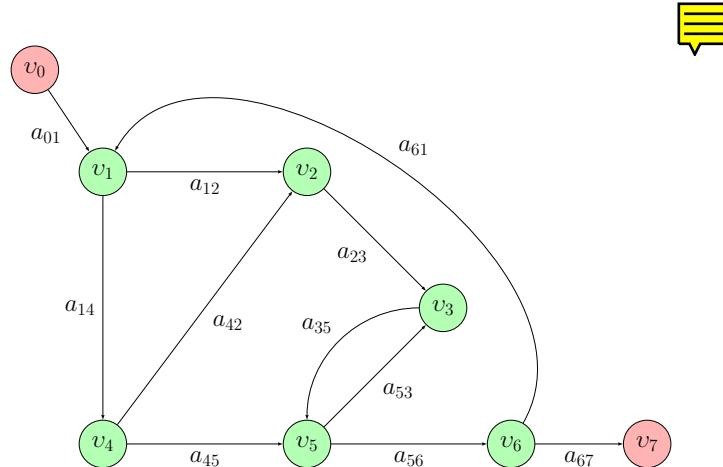


Figura 5.1: Exemplo de grafo de fluxos.

## Grafos de Fluxos

Este grafo em particular tem associado a cada aresta um *fluxo*, i.e. um número racional estritamente positivo; designa-se por  $a_{ij}$  o fluxo que está associado à aresta  $\overrightarrow{v_i v_j}$ ; os fluxos definem uma *matriz de fluxos*  $\mathcal{F}$  de elemento genérico  $a_{ij}$ ; por convenção, neste tipo de grafos, se não existe aresta com origem  $v_i$  e destino  $v_j$ , então faz-se  $a_{ij} = 0$ . Os vértices  $v_0$  e  $v_7$  têm papéis especiais; o primeiro é origem de arestas mas não é destino de nenhuma aresta: designa-se por “*source*”; ao invés o segundo é destino de arestas mas não é origem de nenhuma: designa-se por “*sink*”.

---

### § 5.8 Problema de Fluxos

**Interpretação do grafo:** o grafo da figura 5.1 tem a seguinte interpretação:

1. As arestas são interpretados como condutas/canais de fluidos, energia ou informação; as constantes  $a_{ij}$  são interpretadas como *capacidades* destas condutas/canais.
2. Cada conduta/canal está equipada com uma válvula/interruptor; na aresta  $\overrightarrow{v_i v_j}$ , a posição aberto/fechado deste dispositivo é representado pelo valor de uma variável binária  $x_{ij}$ ; tem-se  $x_{ij} = 1$  se e só se a válvula na aresta  $\overrightarrow{v_i v_j}$  está aberta.
3. O fluxo que flui na aresta  $\overrightarrow{v_i v_j}$  é representada por uma variável racional  $y_{ij}$ .
4. Os vértices distintos da “*source*”  $v_0$  são interpretados como *consumos*; a quantidade de fluido ou informação consumida em  $v_i$  é representado por uma variável racional  $z_i$ .

## Restrições

1. As variáveis  $x_{ij}$  tomam valores em  $\{0, 1\}$ ;  $y_{ij}$  e  $z_i$  tomam valores racionais positivos; i.e.  $\forall i, j \cdot x_{ij} \in \{0, 1\} \wedge y_{ij} \geq 0$  e ainda  $\forall i \neq 0 \cdot z_i \geq 0$ .
2. O fluxo  $y_{ij}$  é zero se a válvula  $x_{ij}$  está fechada e é limitado pela capacidade da conduta se a válvula está aberta; i.e. para todo  $i, j$

$$y_{ij} \leq a_{ij} x_{ij}$$

3. Em cada nodo  $v_i$  distinto da “*source*” ( $i \neq 0$ ) o fluxo total de “*entrada*” (soma dos fluxos nas arestas que têm este nodo como destino) menos o fluxo total de “*saída*” (soma dos fluxos nas arestas que têm este nodo como origem) tem de ser igual ao valor consumido neste nodo. Ou seja, para todo  $i \neq 0$ ,

$$\sum_j y_{ji} - \sum_j y_{ij} = z_i$$

**□2:** As soluções viáveis deste problema são determinadas com programação 0/1 híbrida. Normalmente o problema é complementado com outras restrições; por exemplo com limites mínimos nos consumos  $z_i$  nos nodos “normais” (distintos da “*source*” e do “*sink*”) e um objetivo de minimização do número total de “switches” abertos  $\sum_{i,j} x_{ij}$ . □

**EXEMPLO 5.5:** O seguinte exemplo implementa o problema de fluxos para o grafo na figura 5.1. Assumiu-se que todas as capacidades  $a_{ij}$  são normalizadas no valor 1.

Assumiu-se que os consumos na “source” e no “sink” do grafo são nulos e que nos “nodos normais” é no mínimo 0.1. Pretende-se minimizar o número total de “switches” abertos.

```
from Numberjack import *
N=8
a = [[0 for j in range(N)] for i in range(N)]
a[0][1] = 1.0
a[1][2] = a[1][4] = a[4][2] = a[4][5] = a[5][6] = 1.0
a[6][7] = a[6][1] = a[2][3] = a[3][5] = a[5][3] = 1.0

x = Matrix(N,N,2)
y = Matrix(N,N,0.0,1.0)
z = VarArray(N,0.0,1.0)

m = Model()

for i in range(N):
    for j in range(N):
        m += y[i][j] <= a[i][j]*x[i][j]

for i in range(1,N):
    m += Sum([y[j][i] - y[i][j] for j in range(N)]) == z[i]

m += (z[0] == 0) & (z[N-1] == 0)
for i in range(1,N-1):
    m += z[i] >= 0.1
m += Minimize(Sum(x.flat))

if m.load('SCIP').solve():
    print z
    print x
    print y
```

□

## Cortes em Grafos

A noção de “corte” é fundamental à teoria de grafos. Para introduzir alguma terminologia vamos considerar um grafo genérico  $G = (V, A)$ , sem fluxos e como exemplo ilustrativo pode-se tomar o grafo na figura 5.2.

Seja  $S \subseteq V \neq \emptyset$  um conjunto não vazio de vértices de  $G$ . O *corte*  $C(S)$  no grafo  $G = (V, A)$  é o conjunto de arestas  $\{ \overrightarrow{v_i v_j} \in A \mid v_i \in S \wedge v_j \notin S \}$ .

Considere-se, no exemplo da figura 5.2, o conjunto  $S = \{v_0, v_1, v_4\}$ . O corte respetivo  $C(S)$  é o conjunto de 3 arestas marcadas a vermelho: todas estas arestas têm origem num vértice em  $S$  e destino num vértice que não pertence a  $S$ . A intuição por detrás deste conceito é que estas três arestas “cortam” o fluxo entre os vértices de  $S$  e os restantes vértices do grafo.

Um dos problemas associados a cortes em grafos é o *problema do corte de custo mínimo* que isola dois nodos  $s, t$  de um grafo. Isto é o corte  $C(S)$  de menor custo tal que  $s \in S$  e  $t \notin S$ .

---

### § 5.9 $s, t$ -minimum cut.

Dado um grafo  $G = (V, A)$ , dados pesos racionais positivos  $w_e > 0$  associados às arestas  $e \in A$ , e dados dois vértices distintos  $s, t \in V$  determinar o corte de menor peso que isola  $s$  de  $t$ .

---

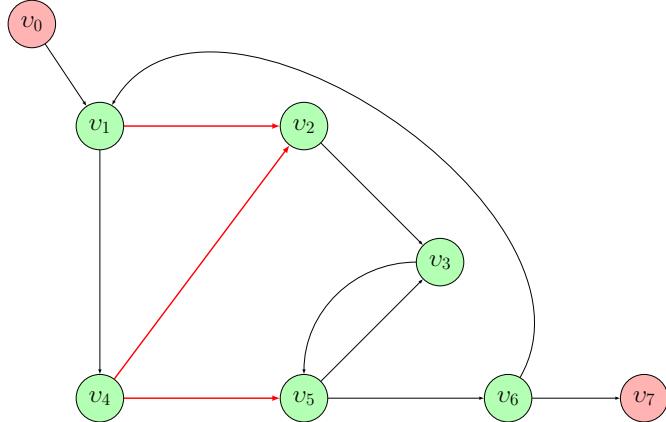


Figura 5.2: Corte num grafo.

Prova-se que para determinar tal corte basta percorrer todos os caminhos  $P$  de origem  $s$  e destino  $t$  e, de cada um destes caminhos, retirar no mínimo uma aresta.

Vamos supor que existem  $m$  caminhos  $P_1, P_2, \dots, P_m$  com origem  $s$  e destino  $t$ ; sem perda de generalidade pode-se considerar apenas caminho mínimo. O problema do corte mínimo é portanto equivalente ao de encontrar o conjunto de arestas  $E$  de peso total mínimo em que cada um dos  $e \in E$  pertence pelo menos a um dos  $P_j$ .

Dito de outro modo, é um problema de “point cover” em que os “pontos” são as arestas  $V$  e a coleção  $\mathcal{S}$  é definido pelos caminhos  $P_j$  entre  $s$  e  $t$ . A formulação do problema segue a estrutura de qualquer “point cover”; seja  $x_e \in \{0, 1\}$  a função característica do corte  $E$ ; então o problema formula-se como o seguinte problema de programação 0/1.

$$\begin{array}{ll} \min & \sum_{e \in A} w_e x_e \\ & \sum_{e \in P_j} x_e \geq 1 \quad \text{para cada um dos caminhos } P_j \\ x_e & \in \{0, 1\} \end{array} \quad (5.13)$$

Obviamente esta formulação requer que todos os caminhos  $P$  entre  $s$  e  $t$  sejam previamente calculados. Consideremos de novo o grafo na figura 5.2 e, nesse grafo, vamos enumerar todos os caminhos mínimos de origem  $v_1$  e destino  $v_3$ . Existem 3 caminhos mínimos

$$\left\{ \begin{array}{l} \{\overrightarrow{v_1 v_2}, \overrightarrow{v_2 v_3}\} \\ \{\overrightarrow{v_1 v_4}, \overrightarrow{v_4 v_2}, \overrightarrow{v_2 v_3}\} \\ \{\overrightarrow{v_1 v_4}, \overrightarrow{v_4 v_5}, \overrightarrow{v_5 v_3}\} \end{array} \right. \quad (5.14)$$

Existem outros caminhos que não são mínimos; por exemplo, os definidos pelas sequências de vértices  $\langle 1, 4, 5, 6, 1, 2, 3 \rangle$ ,  $\langle 1, 4, 5, 3, 5, 3 \rangle$ , etc.. Este caminhos têm

como sub-conjunto próprio um dos caminhos mínimos; por isso não introduzem informação que já não esteja representada no conjunto inicial.

O conjunto de arestas é

$$A \equiv \{\overrightarrow{v_0 v_1}, \overrightarrow{v_1 v_2}, \overrightarrow{v_1 v_4}, \overrightarrow{v_2 v_4}, \overrightarrow{v_2 v_3}, \overrightarrow{v_4 v_5}, \overrightarrow{v_5 v_6}, \overrightarrow{v_6 v_1}, \overrightarrow{v_6 v_7}, \overrightarrow{v_5 v_3}, \overrightarrow{v_3 v_5}\}$$

Associado a cada aresta  $\overrightarrow{v_i v_j}$  associamos um peso racional  $w_{ij} > 0$  e uma variável  $x_{ij}$  com valores em  $\{0, 1\}$ . A formulação do problema será

$$\begin{aligned} \min \quad & w_{01} x_{01} + w_{12} x_{12} + w_{14} x_{14} + w_{24} x_{24} + w_{23} x_{23} + w_{45} x_{45} + \\ & + w_{56} x_{56} + w_{61} x_{61} + w_{67} x_{67} + w_{53} x_{53} + w_{35} x_{35} \end{aligned}$$

sujeito às restrições

$$\begin{aligned} x_{12} + x_{13} &\geq 1 & , & x_{14} + x_{42} + x_{23} \geq 1 & , & x_{14} + x_{45} + x_{53} \geq 1 \\ x_{ij} &\in \{0, 1\} \end{aligned}$$

## 5.4 Programação Inteira com Restrições (CIP)

Frequentemente a descrição de um problema exige restrições mais gerais do que aquelas que estão associadas aos problemas de programação inteira ou programação inteira híbrida. Tomando por base o formalismos da programação inteira híbrida, é possível expandir este modelo com outro tipo de restrições. Um modelo geral da programação com restrições envolvendo variáveis inteiros  $x$ , variáveis racionais  $y$  e  $m$  restrições binárias genéricas  $\mathcal{C}_1, \dots, \mathcal{C}_m$ , seria

$$\begin{aligned} & \text{maximizar} \quad \phi(x, y) \\ & \text{sujeito a} \quad \mathcal{C}_j(x, y) = 1 \quad \text{para todo } j = 1..m \\ & \quad y_i \in \mathbb{Q}_+ \quad , \quad x_i \in \mathbb{Z} \end{aligned} \tag{5.15}$$

O objetivo da CIP é o de tentar definir uma subclasse destes problemas que ainda possa ser resolvido usando as estratégias de *branch-and-bound*, *cut-plane*, apresentadas anteriormente, ou ainda combinações destas duas como a *branch-and-cut*. Para isso cada um dos  $\mathcal{C}_j$  tem de satisfazer as condições que permitem usar estas estratégias; nomeadamente,

---

### § 5.10 Restrições CIP

Seja  $\mathcal{C}(x, y)$  uma restrição em CIP, com  $x$  representando o vetor das variáveis inteiros e  $y$  o vetor das variáveis racionais. Então deve-se verificar,

1. Cada uma das variáveis em  $x$  ou  $y$  está associado um domínio limitado  $D$  no sentido em que qualquer solução viável do problema atribui à variável um valor nesse domínio.
2. Toda a instanciação  $x \leftarrow v$  das variáveis inteiros por constantes, origina uma restrição  $\mathcal{C}(v, y)$  e uma função de custo  $\phi(v, y)$  que têm sempre a forma da programação linear. Isto é

$$\mathcal{C}(v, y) \equiv y \cdot a(v) \leq b(v) \quad , \quad \phi(v, y) \equiv y \cdot c(v)$$

$a(v)$ ,  $b(v)$  e  $c(v)$  são vetores de componentes racionais e dimensão apropriada; “ $\cdot$ ” denota o produto interno de vetores.

---

Em termos gerais a verificação destas condições implica que as soluções viáveis do problema CIP pode ser sempre encontradas enumerando todas as variáveis inteiras nos seus domínios finitos e resolvendo o problema LP resultante. Para a enumeração das variáveis inteiras recorremos precisamente aos algoritmos *branch-and-bound* e *branch-and-cut*.

Note-se que a formalização de um problema em SMT's, pode ser visto como “programação com restrições no sentido lato” mas não verifica a condição 1; de fato um tal modelo SMT pode lidar com inteiros ou reais não limitados.

Em seguida apresentamos exemplos de restrições CIP que ocorrem com frequência.

#### 5.4.1 A restrição ”todos diferentes”.

Frequentemente ocorrem situações onde se exige que duas variáveis  $x$  e  $y$  só pode ser instanciado com valores distintos. Esta restrição exige que  $x$  e  $y$  estejam ambas restritas a um mesmo universo  $U \equiv \{u_1, \dots, u_m\}$ , com  $m \geq 2$  elementos.

Em princípio a restrição poderia ser escrita como  $x \neq y$ . No entanto as estratégias *branch-and-cut* exigem que todas as restrições assumam a forma de equações; de fato mesmo as inequações  $x \leq b$  transformam-se em equações com a introdução de variáveis adicionais  $z + x = b$  com  $z \geq 0$ . Por isso  $x \neq y$  tem de ser escrito sob a forma de equações; uma vez que  $x$  e  $y$  estarem restritos a  $U$ , tem-se

$$(x \neq y) \equiv \bigvee_{a \in U} \bigvee_{b \in U \setminus a} (x = a) \wedge (y = b) \quad (5.16a)$$

Quando o universo  $U$  é um intervalo finito no domínio dos inteiros,  $x \neq y$  pode ser formulado de outro modo mais simples

$$(x \neq y) \equiv (x + 1 \leq y) \vee (y + 1 \leq x) \quad (5.16b)$$

Normalmente exige-se mais: dado um conjunto de variáveis  $X \equiv \{x_1, \dots, x_n\}$ , exige-se que todos os  $x_i$  sejam distintos. A restrição **all-different** aplica-se a tais conjuntos e define-se como

$$\text{all-different}(X) \equiv \bigwedge_{i \neq j} (x_i \neq x_j) \quad (5.16c)$$

sendo  $x_i \neq x_j$  expandido como em (5.16a) ou como em (5.16b).

Note-se que tanto (5.16a) como (5.16b) contêm disjunções de igualdades e tais disjunções não estão contempladas na formulação da programação inteira. Portanto a restrição (5.16c) não pode ser formulada em IP. No entanto as estratégias de *branch-and-cut* conseguem lidar com estas restrições usando procedimentos específicos, como veremos em seguida.

Considere-se, por exemplo, a construção de uma matriz SUDOKU de dimensão  $N$  como está apresentada no exemplo 5.6. Pretende-se construir uma matriz quadrada  $X$ , de dimensão  $N$ , preenchida com os inteiros  $1..N^2$  de tal modo que todas as linhas, todas as colunas, a diagonal principal e a diagonal secundária somem sempre uma mesma quantidade  $T$ .

As restrições têm todas a forma de um problema de programação inteira exceto a última **AllDiff(X.flat)**. Nesta restrição a construção **X.flat** transforma a matriz variável **X** numa lista de variáveis; a construção **AllDiff** agora força a que os valores que vão instanciar esta lista de variáveis sejam todos diferentes.

## EXEMPLO 5.6:

```

from Numberjack import *
N=5

X = Matrix(N,N,1,N*N)
T = N*(N*N+1)/2

m = Model()
for i in range(N):
    m += Sum([X[i][j] for j in range(N)]) <= T # soma de cada linha
    m += Sum([X[j][i] for j in range(N)]) <= T # soma de cada coluna

    m += Sum([X[j][j] for j in range(N)]) <= T # soma da diagonal principal
    m += Sum([X[j][N-j-1] for j in range(N)]) <= T # soma da diagonal secundária

    m += AllDiff(X.flat) # todos diferentes

if m.load('SCIP').solve():
    print X

```

□

## 5.4.2 Restrições booleanas

Frequentemente, num problema com variáveis inteiras  $x$  e racionais  $y$  definido por um conjunto de restrições  $\mathcal{S}$ , uma dessas restrições  $\mathcal{C}$  tem a forma

$$\mathcal{C}(x, y) = \mathcal{C}_1(x, y) \bowtie \mathcal{C}_2(x, y)$$

em que  $\mathcal{C}_1(x, y)$  e  $\mathcal{C}_2(x, y)$  são restrições CIP e  $\bowtie$  é uma das conetivas binárias da lógica proposicional: *conjunção*  $\wedge$ , *disjunção*  $\vee$  ou *implicação*  $\rightarrow$ .

No caso mais geral  $\mathcal{C}$  pode não ser uma restrição CIP. Por isso é necessário ver como é possível converter um tal conjunto  $\mathcal{S}$  num conjunto onde só existam restrições CIP.

No caso em que  $\bowtie$  denota a conjunção a solução é simples: basta substituir  $\mathcal{C}_1 \wedge \mathcal{C}_2$  pelo conjunto de restrições  $\{\mathcal{C}_1, \mathcal{C}_2\}$  e juntar este conjunto às restantes restrições do problema. Isto é, faz-se a transformação

$$\mathcal{S} \leftarrow \mathcal{S} - \{\mathcal{C}_1 \wedge \mathcal{C}_2\} + \{\mathcal{C}_1, \mathcal{C}_2\}$$

Quando  $\bowtie$  denota a disjunção, vamos assumir (sem perda de generalidade) que  $\mathcal{C}_1(x, y)$  e  $\mathcal{C}_2(x, y)$  são *mutuamente inconsistentes*; isto é, não existe qualquer solução viável de uma das restrições que seja também solução viável da outra.

Isto significa que  $\mathcal{C}_1$  e  $\mathcal{C}_2$  determinam uma partição do espaço de procura. De fato, gere-se a partir de  $\mathcal{S}$  os dois conjuntos de restrições CIP

$$\mathcal{S}_1 \equiv \mathcal{S} - \{\mathcal{C}_1 \vee \mathcal{C}_2\} + \{\mathcal{C}_1\} \quad , \quad \mathcal{S}_2 \equiv \mathcal{S} - \{\mathcal{C}_1 \vee \mathcal{C}_2\} + \{\mathcal{C}_2\}$$

Os conjuntos  $\mathcal{S}_1$  e  $\mathcal{S}_2$  também são mutuamente inconsistentes porque os respetivos conjuntos de soluções viáveis estão contidos, respetivamente, nas soluções viáveis de  $\mathcal{C}_1$  e  $\mathcal{C}_2$ . Por outro lado, como  $\mathcal{C}_1 \vee \mathcal{C}_2$  faz parte de  $\mathcal{S}$ , as soluções viáveis de  $\mathcal{S}$  são a união das soluções viáveis de  $\mathcal{S}_1$  com as soluções viáveis de  $\mathcal{S}_2$ .

Note-se que este processo de partição do espaço de procura é precisamente o mesmo tipo de mecanismo que foi usado nos algoritmos *branch-and-bound*. Por isso é muito

simples estender o algoritmo para lidar com este tipo de partições. A conclusão que podemos assumir é que não existe complexidade adicional que resulte de restrições que usem a disjunção: temos apenas uma forma particular de partição do espaço de procura que o algoritmo *branch-and-bound* consegue processar.

Quando  $\times \equiv \rightarrow$  usamos a tautologia  $\mathcal{C}_1 \rightarrow \mathcal{C}_2 \equiv \neg \mathcal{C}_1 \vee \mathcal{C}_2$  e notamos que a negação de uma restrição CIP é também uma restrição CIP.

De fato, se após a instanciação das variáveis inteiros, tivermos uma restrição  $y A \leq b$  a sua negação é  $-y A < -b$ .

### 5.4.3 “Mixed-Integer Quadratic Constrained Programming”

Uma extensão “simples” da programação linear (LP) formula-se como

$$\begin{aligned} &\text{maximizar} && c \cdot y + y Q \cdot y \\ &\text{sujeito a} && y A \leq b \\ &&& y \in \mathbb{Q}_+^n \end{aligned} \tag{5.17}$$

e é determinado por matrizes  $A \in \mathbb{Q}^{n \times m}$ ,  $Q \in \mathbb{Q}^{n \times n}$  (simétrica) e vetores  $b \in \mathbb{Q}^m$  e  $c \in \mathbb{Q}^n$ . O símbolo “ $\cdot$ ” representa o produto interno de vetores. Esta formulação determina o chamado *problema de programação quadrática* (QP).

É sabido que, ao contrário de LP, o problema QP é NP-completo. Portanto um problema CP, cuja relaxação natural tenha a forma QP, não pode usar soluções da relaxação para construir a solução do problema não-relaxado; isto porque a relaxação é tão difícil de resolver quanto o problema original.

**EXEMPLO 5.7:** O exemplo paradigmático da programação quadrática é o designado *problema do “portfolio”* uma versão do qual se pode enunciar da seguinte forma.

Um investidor monitoriza  $N$  investimentos associados a retornos  $R_1, \dots, R_N$ ; cada  $R_j$  é modelado por uma variável aleatória com valor expectável  $r_j$  e variância  $s_j$ ; i.e.

$$r_j = \mathbb{E}[R_j] \quad , \quad s_j = \mathbb{E}[(R_j - r_j)^2]$$

Adicionalmente, como primeira hipótese simplificadora, assume-se que os  $R_j$ ’s são mutuamente independentes; i.e.  $\mathbb{E}[(R_i - r_i)(R_j - r_j)] = 0$  para todo  $i \neq j$ .

O investidor distribui o total do seu investimento, aqui normalizado ao valor 1, pelos vários  $R_j$ . Seja  $x_j \in [0..1]$  a fração do investimento que o investidor aloca ao  $j$ -ésimo investimento. Estas frações têm de verificar a restrição

$$\sum_{j=1}^N x_j \leq 1 \tag{5.18}$$

O retorno total é determinado pela variável aleatória

$$R \equiv \sum_{j=1}^N x_j R_j$$

O investidor quer maximizar o valor expectável de  $R$  e minimizar o risco do seu investimento; aqui o risco é determinado pela variância  $\mathbb{E}[(R - \mathbb{E}[R])^2]$ . Assim a função custo deste problema tem a forma

$$\phi(x) \equiv \mu \mathbb{E}[R] - \mathbb{E}[(R - \mathbb{E}[R])^2] \tag{5.19}$$

sendo  $\mu > 0$  um parâmetro que afere a importância relativa dos dois termos desta função.

É fácil o determinar tanto o valor expectável como a variância de  $R$ ; tem-se

$$\mathbb{E}[R] = \sum_j x_j r_j \quad (5.20)$$

Para calcular a variância  $\mathbb{E}[(R - \mathbb{E}[R])^2]$  usa-se

$$\begin{aligned} \mathbb{E}[(R - \mathbb{E}[R])^2] &= \mathbb{E}\left[\left(\sum_j x_j (R_j - r_j)\right)^2\right] = \\ &= \sum_{i,j} x_i x_j \mathbb{E}[(R_i - r_i)(R_j - r_j)] = \sum_j x_j^2 s_j \end{aligned} \quad (5.21)$$

uma vez que  $\mathbb{E}[(R_i - r_i)(R_j - r_j)] = 0$  para todo  $i \neq j$ . Portanto a função custo será

$$\phi(x) \equiv \mu \sum_j x_j r_j - \sum_j x_j^2 s_j \quad (5.22)$$

Se a hipótese de investimentos independentes se não verificar então define-se as correlações

$$\alpha_{ij} \equiv \mathbb{E}[(R_i - r_i)(R_j - r_j)]$$

e tem-se

$$\phi(x) \equiv \mu \sum_j x_j r_j - \sum_i \sum_j x_i x_j \alpha_{ij} \quad (5.23)$$

Resumindo, o problema da otimização do “portfolio” consiste em

$$\begin{array}{ll} \text{maximizar} & \phi(x) \\ \text{sujeito a} & \sum_i x_i \leq 1 \\ & x \in \mathbb{Q}_+^N, x_i \geq 0, x_i \leq 1 \end{array} \quad (5.24)$$

sendo  $\phi(x)$  o custo em (5.22) ou em (5.23) consoante se assume investimentos independentes ou dependentes.  $\square$

Embora o problema do “portfolio” seja formulado como um problema de programação quadrática, e portanto não possa ser descrito diretamente em “constraint integer programming”, é relativamente simples inseri-lo neste formalismo. Para isso basta considerar que o investimento total como um inteiro  $T$  e assim como as parcelas  $x_j$ . A formulação será

$$\begin{array}{ll} \text{maximizar} & \mu \sum_j r_j x_j - \sum_{i,j} \alpha_{ij} x_i x_j \\ \text{sujeito a} & \sum_j x_j \leq T \\ & x \in \mathbb{Z}^N, x_j \geq 0, x_j \leq T \end{array} \quad (5.25)$$

## Bibliografia

- Biere, A., Cimatti, A., Clarke, E., Strichman, O. & Zhu, Y. (2003), ‘Bounded Model Checking’, *Advances in Computers* .  
**URL:** <http://www.sciencedirect.com/science/article/pii/S0065245803580032>
- Brace, K. S., Rudell, R. L. & Bryant, R. E. (1990), Efficient Implementation of a BDD Package, in ‘Proceedings of the 27th ACM/IEEE Design Automation Conference’, DAC ’90, ACM, New York, NY, USA, pp. 40–45.  
**URL:** <http://doi.acm.org/10.1145/123186.123222>
- Conforti, M., Cornuéjols, G. & Zambelli, G. (2014), *Integer Programming*, Vol. 271 of *Graduate Texts in Mathematics*, Springer.