Capítulo 1: Breve Revisão da Lógica

Domínios e Computações

N denota o domínio dos números naturais e $\mathrm{B} \equiv \{0,1\}$ (ou $\mathbf{2}$) denota o domínio dos *bits*. O domínio $\mathrm{B}^* \equiv \{0,1\}^*$ denota o domínio das *strings* de bits finitas. Os domínios N e B^* são isomórficos; na maioria dos problema é indiferente usar um ou o outro.

Neste contexto $|x|$ denota o *comprimento* da sequência de bits $x \in \{0,1\}^*$. Para um inteiro n, o seu *tamanho* é o comprimento da menor string $\widetilde{n}$ que codifica n; isto é $|n| \equiv \lceil \log_2 n \rceil = |\widetilde{n}|$.

Uma função que estabelece o isomorfismo $\{0,1\}^* \to \mathrm{N}$ é, usualmente,

$$x \mapsto 2^{|x|} - 1 + \sum_{i=0}^{|x|-1} x_i\, 2^i$$

mas outras podem ser utilizadas.

Esta função é conhecida por **enumeração lexicográfica** das strings, uma vez que as enumera com se indica sem seguida $\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111 \cdots$,

> **Nota**
> Em muitas situações o isomorfismo entre N e $\{0,1\}^*$ está sempre presente e por isso qualquer definição que envolva um destes domínios gera uma definição análoga envolvendo o outro domínio.
>
> Por isso no que se segue usamos $\mathbf{2}$ como abreviatura de qualquer domínio isomórfico com $\{0,1\}$ e usamos ϖ para representar qualquer domínio contável isomórfico com $\{0,1\}$.
> O símbolo ϖ denota o **ordinal dos contáveis**.

A cardinalidade de domínios cujos elementos são sequência de bits ou números naturais permite uma classificação desses domínios. Por exemplo

- **domínios finitos** têm como exemplo paradigmático o domínio $\mathrm{B}^n \equiv \{0,1\}^n$ das *palavras* ("words") ou *vetores* ("bit-vectors") com exatamente n componentes. São também exemplos de domínios finitos os domínios da forma

$$[t] \equiv \{i \in \mathrm{N} \mid i < t\}$$

- **domínios contáveis** são domínios isomórficos com um subconjunto dos números naturais; i.e. domínios D para os quais existe uma função injetiva $\mathrm{inx} : D \to \mathrm{N}$.
- **domínios de Cantor** são classes cujos elementos são domínios contáveis.

Na terminologia da computabilidade,

- **objectos** são elementos de domínios contáveis,
- **conjuntos** identificam-se com coleções de números naturais e são elementos dos domínios de Cantor
- **classes** são coleções de conjuntos ou coleções de classes

O paradigma de um domínio de Cantor é a classe $\{0,1\}^\infty$ (ou $\mathbf{2}^\infty$) de todas as sequências infinitas de bits (também designadas por *streams)*

$$\{w_n\}_{n\in\mathrm{N}}$$

As *streams* $w \in \{0,1\}^{\infty}$ podem-se identificar ainda com

- Funções $w : \varpi \to \mathbf{2}$
  O domínio de tais funções representa-se pelo ordinal $\mathbf{2}^{\mathrm{N}}$ ou $\mathbf{2}^{\varpi}$.

- Conjuntos $\overline{w} \equiv \{n \in \mathrm{N} \mid w_n = 1\}$.
  Genericamente tem-se conjuntos $\overline{w} \equiv \{x \in \omega \mid w_{\mathrm{inx}(x)} = 1\}$.

- séries infinitas
$$0 \cdot w \equiv \sum_{n\in\overline{w}} 2^{-n-1}$$

- números reais vistos como limites de sequências crescentes e limitadas de números racionais da forma $\{2^{-n}\}$.

Computações

No modelo de computação que se segue considera-se que N (ou $\{0,1\}^*$) tem a ordem dos inteiros e está equipado com um conjunto de **operações primitivas:**

i. as constantes $0, 1$,
ii. relações binárias $<, \neq, \cdots$
iii. os operadores binários soma $+$ e concatenação de strings xy ;
   o par (x,y) é equivalente à concatenação $0^{|x|}\,1\,xy$.
iv. o operador ternário ite (*if-then-else).*

A concatenação xy pode também ser representada com um operador específico: $x \parallel y$.

Funções Booleanas

A título de exemplo vamos considerar os problemas paradigmáticos das funções booleanas representadas como polinómios booleanos ou *polinómios de Zhegalkin*.
Estes problemas têm variantes que ocorrem também em outras estruturas: nomeadamente nas fórmulas proposicionais e nas relações lineares inteiras.

Uma função booleana total $f : \{0,1\}^n \to \{0,1\}$ é completamente determinada por um polinómio booleano a n variáveis e de grau menor ou igual a n.

Todo o polinómio é definido por operações de soma e multiplicação numa estrutura algébrica apropriada (um *corpo*) munido de somas e multiplicações.
Neste caso a soma $+$ é a operação **xor** de bits e a multiplicação é o **and** de bits. O corpo assim definido representa-se por F_2 (*corpo finito com 2 elementos*).

Exemplo 1
Exemplos de polinómios com 4 variáveis serão

$$1 + x_0 x_2 + x_1 x_3 \quad , \quad x_0 x_1 + x_2 + x_1 x_2 x_3 \quad , \quad x_0 + x_1 + x_2 + x_3$$

Um polinómio $f(x_0, \cdots, x_{n-1})$ que verifica $f^2 = f$ diz-se **booleano**.
f é binário quando $f + f = 0$.

As funções booleanas são sempre descritas por polinómios booleanos e binários.

Um polinómio diferente de zero que usa apenas multiplicações designa-se por ***monómio***.
Cada monómio escreve-se como

$$x_0^{e_0} \times x_1^{e_1} \times \cdots \times x_{n-1}^{e_{n-1}}$$

e, por isso, é completamente determinado pelo vetor dos seus expoentes

$$(e_0, e_1, \cdots, e_{n-1})$$

Nos polinómios booleanos os expoentes são sempre 0 ou 1 uma vez que $x_i^2 = x_i$, para todo i.
Por isso cada monómio é completamente determinada por um vetor de n bits.

Exemplo 2
Nos monómios com 4 variáveis tem-se

- $x_0 x_2 x_3 = x_0^1 x_1^0 x_2^1 x_3^1$ é representado pela palavra 1011
- $x_1 x_2 = x_0^0 x_1^1 x_2^1 x_3^0$ é representado pela palavra 0110.
- $1 = x_0^0 x_1^0 x_2^0 x_3^0$ é representado pela palavra 0000

Um polinómio é uma soma de monómios e é completamente determinado pelo conjunto dos vetores de expoentes. Por isso um polinómio booleano de **grau** n é completamente descrito por um conjunto de palavras de bits de tamanho n; isto é um subconjunto de B^n.

Exemplo 3
Os três polinómios no exemplo 1 acima são descritos pelos seguintes conjuntos de "strings"

$$\{0000, 1010, 0101\} \quad , \quad \{1100, 0010, 0111\} \quad , \quad \{1000, 0100, 0010, 0001\}$$

Note-se que com a soma **xor** verifica-se $x + x = 0$ para todo x; por isso na construção do polinómio, dois monómios iguais anulam-se mútuamente.

Considere-se uma linguagem (conjunto de strings) $\mathrm{L} \subseteq \{0,1\}^n$ como descrição de um polinómio booleano f com n variáveis .
O **tamanho de** f, representado por $|f|$, é o número de elementos de L; isto é, o número de monómios em f.

Vamos esboçar algoritmos para resolver os dois problemas essenciais nas aplicações deste tipo de funções:

- **valoração** :
  dado um vetor $x \in \{0,1\}^n$ *calcular o valor de* $f(x)$.
- **satisfação** ou **SAT**:
  determinar se existe algum vetor $x \in \{0,1\}^n$ *que "valide"* f.
  $$\exists\, x \in \{0,1\}^n \cdot f(x) = 1\,.$$

O **algoritmo de valoração** percorre as n componentes de x e, para cada uma percorre todos os elementos de L. É um exemplo paradigmático de algoritmo determinístico.

$\mathrm{EVAL}(n, \mathrm{L}, x) \equiv$

1. Testa o tipo dos argumentos: $n > 0 \ \wedge\ |x| = n \ \wedge\ \mathrm{L} \subseteq \{0,1\}^n$. Se falhar devolve 0.
2. Para $i = 0, 1, \cdots, n-1$
   Se $x_i = 0$ remover de L todos os expoentes $e \in \mathrm{L}$ tais que $e_i = 1$
3. Se no final a cardinalidade de L é um número par, então o resultado é 0 se L tem o número ímpar de elementos, então o resultado será 1.

Os monómios da função booleana que têm o valor 0 após as suas variáveis serem substituidas pelos valores x_i são aqueles onde pelo menos um desses x_i é zero.
No final do passo 2. foram eliminados de L todos os monómios onde uma das suas variáveis tem o valor 0. Restam só os monómios que têm a valoração 1 ; a soma final de todos estes 1's será 0 se o número desses monómios for par e será 1 se for ímpar.

O algoritmo limita-se a comparar vetores de bits e verificar o valor de componentes individuais de vetores; não executa qualquer operação de soma ou multiplicação.

A sua complexidade computacional depende crucialmente do número de monómios em f: isto é, a cardinalidade de L. No pior caso, o número de monómios é exponencial com n : $|f| = O(2^n)$ e a complexidade será exponencial com n.

Porém, muitas vezes o número de monómios é polinomialmente limitado com a dimensão n; nesse caso já a complexidade deste algoritmo é polinomial.

O **algoritmo de satisfação** é mais complexo: mesmo quando o número de monómios é polinomialmente limitado, os melhores algoritmos de SAT têm complexidade exponencial no pior caso. Isto não significa que, no caso médio, não existam algoritmos viáveis.

A computação $\mathrm{SAT}(n, f)$ produz $\perp$ se f não é satisfazível e produz não-deterministicamente algum x tal que $f(x) = 1$ quando for satisfazível.

Um polinómio f em n variáveis pode ser decomposto numa soma
$$f = x_0\, g + h$$
em que x_0 é a primeira variável e g, h são polinómios em $n-1$ variáveis que não contém x_0.

Seja $v \equiv (x_1, \cdots, x_{n-1})$ o vetor das restantes variáveis x_i.

Somando $x_0\, h$ duas vezes a decomposição passa a ser

$$f = x_0\,(g + h) + (1 + x_0)\,h$$

Isto é

$$f(x_0\, v) = \begin{cases} (g+h)(v) & \text{se} \quad x_0 = 1 \\ h(v) & \text{se} \quad x_0 = 0 \end{cases}$$

Agora, se $y \in \{0,1\}^{n-1}$ é uma solução de SAT para $(g + h)$ então $\langle 1, y\rangle$ é uma solução do SAT para f; ao invés, se y é uma solução do SAT para h, então $\langle 0, y\rangle$ é uma solução do SAT para f.

Esta decomposição é a base de todos os algoritmos de SAT; o algoritmo é um exemplo paradigmático de algoritmo não determinístico que se define recursivamente

$\mathrm{SAT}(n,f) \equiv$

1. Se $n = 0$ e $f = 0$ devolve $\perp$; se $n = 0$ e $f = 1$ devolve a string vazia ε.
2. Se $n > 0$ obtém de f os polinómios $(g + h)$ e h, de grau $n - 1$, como referido atrás.
3. Calcula $y \leftarrow \mathrm{SAT}(n-1, g+h)$ e $z \leftarrow \mathrm{SAT}(n-1, h)$.
4. Se $y = \perp$ e $z = \perp$ devolve $\perp$
5. Se $y \neq \perp$ então a string $1\,y$ é resultado possível; se $z \neq \perp$ então $0\,z$ é resultado possível. Escolhe não-deterministicamente um dos resultados possíveis e devolve-o.

Nota
Se f é descrito por um conjunto de palavras de n bits, então dividindo f em palavras cujo 0-ésimo bit é 1 ou 0 constrói-se

$$g = \{v \mid 1\, v \in f\} \quad \text{e} \quad h = \{v \mid 0\, v \in f\}$$

Para calcular a descrição de $(g + h)$, faz-se a construção $g \setminus h \cup h \setminus g$. Isto é, faz-se a união de g e h como multiconjuntos e remove-se os pares de elementos iguais.

Neste algoritmo existem n níveis de recursividade (um por variável) e, como a recursividade é dupla, a complexidade no pior caso será $O(2^n)$.

A computação $\mathrm{SAT}(n,f)$ é não determinístico mas tem não-determinismo finito; de facto o número de resultados possíveis está limitado a 2^n: em cada nível de recursividade, uma execução particular do algoritmo tem um número "escolhas" ≤ 2 (passo 5).

Como qualquer algoritmo com não-determinismo finito, é sempre possível construir um algoritmo determinístico SAT^* tal que $\mathrm{SAT}^*(n,f)$ devolve, como resultado, o conjunto todos os resultados possíveis de $\mathrm{SAT}(n,f)$.

$\mathrm{SAT}^*(n,f) \equiv$

1. Se $n = 0$ e $f = 0$ devolve o conjunto vazio $\{\}$; se $n = 0$ e $f = 1$ devolve $\{\varepsilon\}$.
2. Se $n > 0$ decompõe f e obtém os polinómios $(g + h)$ e h de grau $n - 1$ como referido anteriormente.

3. Calcula $y \leftarrow \mathrm{SAT}^*(n-1, g+h)$ e $z \leftarrow \mathrm{SAT}^*(n-1, h)$.
4. Calcula $y' \leftarrow \{1\,a \mid a \in y\}$ e $z' \leftarrow \{0\,b \mid b \in z\}$
5. Devolve $y' \cup z'$

Circuitos Booleanos e Aritméticos

Circuitos são um modelo importante para os problemas que vamos encontrar ao longo deste curso.

A função principal de um circuito é a descrição de uma computação funcionalmente complexa através de **operações elementares** numa determinada álgebra. Por isso, uma parte essencial da descrição do circuito é a escolha do domínio onde se vai realizar a computação e os operadores básicos desse domínio.

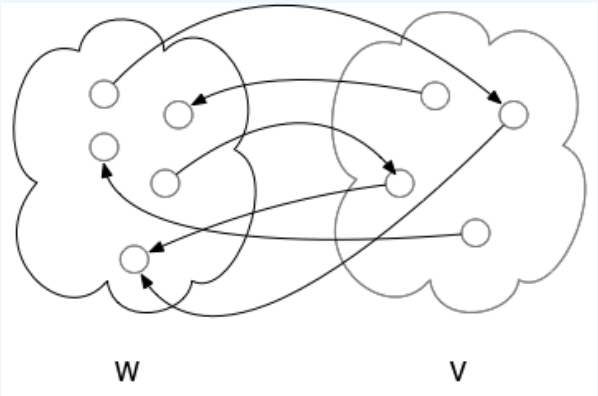
Outra componente do circuito é a definição da sua **topologia**; isto é, a forma como as operações básicas se combinam para construir a computação principal.

É possível descrever a topologia de várias formas mas a mais simples é através de **diagramas de operações ("gates") e conexões ("wires").**

Diagramas e Grafos

Genericamente um **diagrama** é um bi-grafo orientado, acíclico e com pelo menos uma raíz.

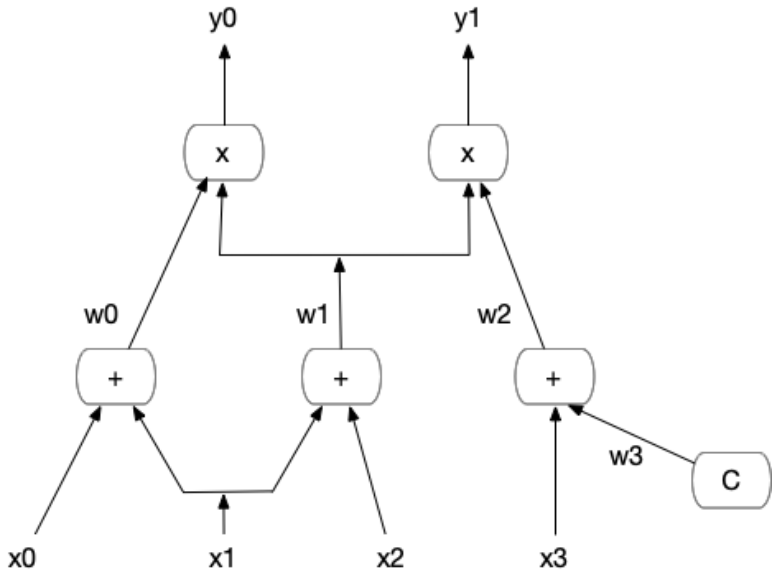
Um bi-grafo orientado é um grafo onde os nodos estão agrupados em dois conjuntos W, V disjuntos e onde cada ramo tem origem em um nodo de um dos conjuntos e destino num nodo do outro conjunto.



Num diagrama os nodos que não são destino de qualquer ramo designam-se por **sources** e os nodos que não são origem de qualquer ramo designam-se por **sinks**. Uma **raíz** do diagrama D é um "sink" r tal que, para qualquer "source" s existe um caminho com origem em s e destino em r.

Num circuito a sua **topologia** está organizada num diagrama onde os nodos estão agrupados em dois conjuntos, **gates** e **wires,** e onde todos os "sinks" são "wires". As raízes do circuito designam-se por

outputs e os "wires" que são "sources" do circuito designam-se por **inputs**.



Um circuito 4×2

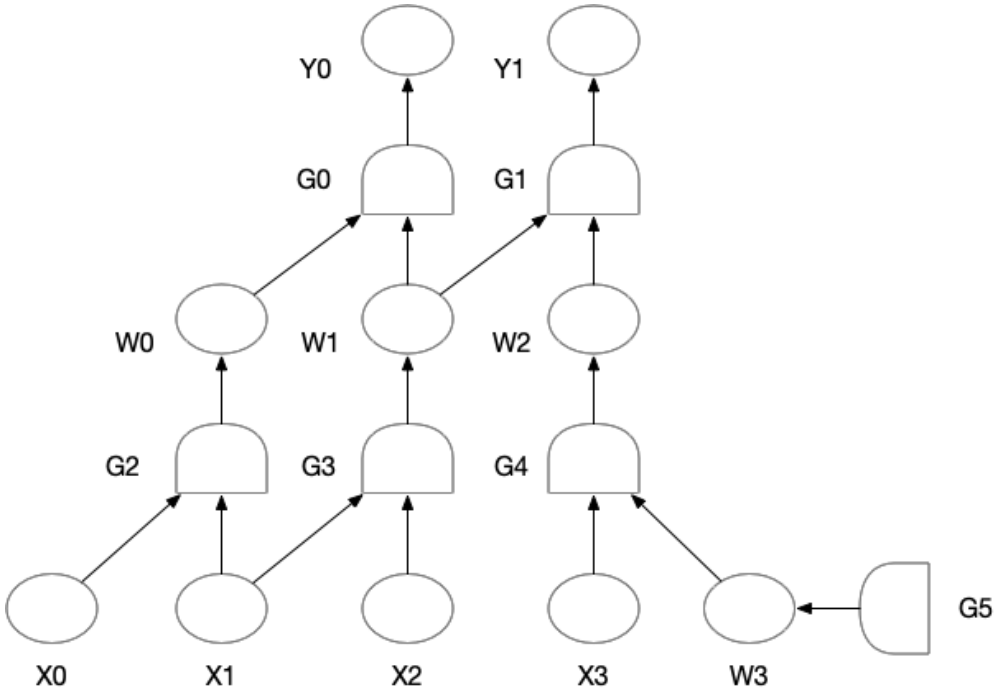
O diagrama acima representa um circuito com 6 operadores (três somas, duas multiplicações e uma constante) e 10 conexões.
Destas conexões, duas são "outputs do circuito", quatro são "inputs do circuito" e quatro são conexões intermédias ligando "outputs" a "inputs" de operadores.
Nestes diagramas não existem ciclos: a informação flui sempre dos inputs para os outputs.

Genericamente, um diagrama representa apenas a *topologia do circuito;* isto é, o arranjo das "gates" e de "wires" a que estão ligados. Para completar a definição do circuito é necessária também uma descrição algébrica como veremos em seguida.

O diagrama anterior ilustra como, cada "gate" tem vários "wires" de onde flui informação (os "inputs") e um só "wire" para onde flui informação (o seu "output"). A informação flui, sempre, na forma de elementos do domínio algébrico do circuito.

Em cada wire, a informação pode fluir de uma só origem: o exterior do circuito ou uma só "gate".
Analogamente, cada "wire" envia informação para o exterior do circuito ou então para uma ou mais "gates".

O bi-grafo que descreve o diagrama anterior é representado em seguida. Note-se os dois tipos de nodos, "gates" e "wires", que aqui são representados por símbolos distintos.



Descrição algébrica

A descrição algébrica do circuito parte de uma álgebra (um domínio e um conjunto de operadores) e associa a cada "wire" uma variável que toma valores no domínio do circuito e a cada "gate" um operador da álgebra. A descrição é formada por um conjunto de equações, uma por "gate", da forma

$$output = operador(input_1, \cdots, input_\ell)$$

Em que $operador$ é o operador da "gate" e $output$ e os $input_i$ denotam as variáveis associadas, respetivamente, ao "output" e aos "inputs" da "gate".

No exemplo anterior associámos multiplicações às "gates" g_0 e g_1, somas às "gates" g_2, g_3 e g_4e uma constante c à "gate" g_5. As relações algébricas definidas são, neste caso,

$y_0 = w_0 \times w_1$, $y_1 = w_1 \times w_2$, $w_0 = x_0 + x_1$, $w_1 = x_1 + x_2$, $w_2 = x_3 + w_3$, $w_3 = c$

Como no diagrama e no bi-grafo não existem ciclos, e a informação flui sempre dos "inputs" para os "outputs" , é sempre possível manipular as relações eliminando todas as variáveis intermédias.
Neste exemplo teríamos

$$y_0 = (x_0 + x_1) \times (x_1 + x_2) \quad , \quad y_1 = (x_1 + x_2) \times (x_3 + c)$$

Após simplificação, a descrição tem uma equação por cada variável de "output" , o lado esquerdo da equação, e no lado direito tem uma expressão exclusivamente formada por operadores das "gates" e variáveis de "input".

Num circuito genérico de dimensão $n \times m$, em que as variáveis de "input" são $x_0, \cdots, x_{n-1}$ e as variáveis de "output" são $y_0, \cdots, y_{m-1}$, a descrição é definida por um sistema de equações

$$y_i = f_i(x_0, \cdots, x_{n-1}) \qquad \text{para} \quad i = 0, \cdots, m-1$$

sendo f_i são expressões compostas pelos operadores das "gates" e pela variáveis dos "wires".

Tipos de Circuitos

Essencialmente descrevemos um processo que, a partir da topologia do circuito $n \times m$ sobre o domínio X e dos operadores elementares associados, permite construir uma função computável $f : X^n \to X^m$.

O processo inverso designa-se por **síntese** do circuito: partindo de uma função computável do tipo $f : X^n \to X^m$ pretende-se construir a descrição topológica e a descrição algébrica que permitem definir essa função.

Obviamente podem existir vários circuitos C que construam a mesma função. Parece intuitivo afirmar que, se a descrição algébrica é simples, então a descrição topológica será complexa; e vice-versa. Por isso a síntese está associada a uma qualquer forma de optimização: por exemplo, minimizando o número de "gates".

Existem algumas formas frequentes de circuitos:

- **circuitos booleanos**
  O domínio é formado pelos valores lógicos $X \equiv \{0, 1\}$ que também determinam as constantes. Os operadores binários elementares são as conectivas **and**, **or** e **xor**, representadas por $\wedge$, $\vee$, $\oplus$.
  Quando o número de "outputs" é 1, o circuito booleano é completamente determinado por uma única fórmula proposicional.

- **circuitos booleanos vetoriais**
  A informação esta organizada em vetores de bits de um tamanho ℓ fixo. Assim o domínio é $X \equiv \{0, 1\}^{\ell}$ e cada elemento desse domínio define uma constante. Os operadores binários são $\wedge$, $\vee$ e $\oplus$ executadas posição a posição. Cada posição $0 \leq i < n$, define um operador unário que seleciona, em cada $x \in X$, a sua componente x_i .

- **circuitos aritméticos**
  A informação está organizada em inteiros positivos representáveis com ℓ bits. Formalmente o domínio é $\mathrm{Z}_N \equiv \{k \in \mathrm{Z} \mid 0 \leq k < N\}$, sendo $N \equiv 2^{\ell}$.
  Neste domínio define-se a estrutura algébrica de um anel com soma e multiplicação modular: $x + y \bmod N$ e $x \times y \bmod N$.
  Associando inteiros à sua descrição binárias estão ainda disponíveis todas as operações elementares nos circuitos booleanos vetoriais: $\wedge$, $\vee$, $\oplus$ e seleção de componente.

Descrição de circuitos e suas complexidades

O *tamanho de um circuito* C, representado por $|\mathrm{C}|$, é o seu número de "gates". Nesta secção vamos analisar alguns dos principais problemas definíveis através de circuitos, analisar a sua complexidade computacional e relacioná-la com o tamanho do circuito.

A maioria destes problemas podem ser estudados no contexto simplificado dos circuitos booleanos com um único "output". Nesse caso a dimensão do circuito coincide com o número dos seus "input wires".

Em primeiro lugar um circuito é uma entidade finita composto por um número finito de conexões e gates. Por isso, pode ser descrita, numa linguagem apropriada, por uma 'string' de bits.

A descrição algébrica de um circuito permite definir recursivamente uma função que calcula os valores dos "outputs" do circuito através de uma ordenação das "gates" designada por **ordenação tipológica**.

1. Ordenamos todas as "gates" g_i do circuito com um índice i de tal forma que todos os os "wires" que são inputs de g_i têm indices distintos e inferiores a i. O "wire" output de g_i é w_i identificado pelo mesmo símbolo i
2. As primeiras n "gates" são "inputs" do circuito; por isso valcula-se
$$w_i \leftarrow x_i \quad \text{para todo} \quad 0 \leq i < n$$
3. As restantes m gates g_i com $n \leq i < m + n$ são tuplos
$$g_i = \langle \mathrm{op}_i , *args_i \rangle$$
em que op_i é o identificador do operador que interpreta algebricamente essa "gate" e $*args_i$ é a lista dos índices das wires que são inputs dessa "gate".
Portanto o circuito introduz uma computação singular (uma por "gate") dada por
$$w_i \leftarrow f_{\mathrm{op}_i}(\{w_a \mid a \in args_i\}) \quad \text{para} \quad i = n \,..\, m-1$$

Assim o circuito é completamente descrito pela sua ordenação tipológica
$$\mathrm{C} = \{\langle \mathrm{op}_i , *args_i \rangle\}_{n \leq i < m}$$

O algoritmo $\mathrm{EVAL}(\mathrm{C}, x)$ calcula o "output" w_{m-1} percorrendo todas as "gates" do circuito de forma ordenada

$$\begin{array}{l} \text{for} \quad i < n \\ \quad w_i \leftarrow x_i \\ \text{for} \quad n \leq i < m \\ \quad w_i \leftarrow f_{\mathrm{op}_i}(\{w_a \mid a \in args_i\}) \end{array}$$

Por outro lado um circuito identifica outras entidades também descritas por uma string finita. Nomeadamente

1. *Linguagem de suporte*
O circuito C determina um conjunto de 'strings'
$$\mathrm{L_C} \equiv \{x \mid \mathrm{EVAL}(\mathrm{C}, x) = 1\}$$

Os elementos de $\mathrm{L_C}$ dizem-se **aceites** pelo circuito C e L designa-se por *linguagem de suporte* de C.

A linguagem $\mathrm{L_C}$ é de ordem n; isto é todos os seus elementos têm comprimento n. É também uma entidade finita e, por isso, também pode ser descrita por uma string de bits.

2. *Função booleana*

   Como vimos anteriormente, um circuito booleano C de dimensão n com um só "output" implementa uma *função booleana*

   $$f_{\mathrm{C}}(x_1, \cdots, x_n)$$

   com exatamente n variáveis.

   Recorde-se que f_C, como toda a função booleana, é descrita por um conjunto de monómios é representado pelo vetor $e \in \{0,1\}^n$ dos expoentes das n variáveis.

   Assim, f_{C} identifica-se como um subconjunto de $\{0,1\}^n$; isto é, f_{C} é também uma linguagem de ordem n e, portanto, pode ser descrita por uma string finita.

É importante frisar que o circuito, a função que ele implementa e a linguagem de suporte são entidades distintas. A mesma função pode ser implementada por circuitos diferentes e, igualmente, uma linguagem de ordem n é suporte de circuitos distintos.

No entanto para cada função booleana f de ordem n existe uma única linguagem S_f formada por todos os $x \in \{0,1\}^n$ que satisfazem f. Essa linguagem é determinada, a partir de f, pelo algoritmo SAT^* que descrevemos anteriormente.

É interessante comparar a cardinalidade dessas linguagens em alguns casos simples de funções de ordem n:

1. A função booleana constante $f = 1$ tem um único monómio; de facto $f \equiv \{0^n\}$. No entanto qualquer $x \in \{0,1\}^n$ satisfaz f; por isso o suporte $\mathrm{S}_f \equiv \{0,1\}^n$.
   Neste caso temos uma função em que a função é descrita por um conjunto singular enquanto que o seu suport é a linguagem máxima de ordem n.
2. A *função de Kronecker* define-se por $f(x) = 1 \text{ sse } x = 0$. O seu suporte é o conjunto singular $\mathrm{S}_f \equiv \{0^n\}$.
   Esta função pode-se escrever também como
   $$f(x_1, \cdots, x_n) = (1 + x_1)(1 + x_2) \cdots (1 + x_n)$$
   que, por indução em n, se verifica que contém todos os monómios de ordem n. Portanto a representação de f é a linguagem máxima de ordem n.

Terminologia da Lógica

Uma lógica é uma linguagem L, cujos elementos se designam por **frases**. Quando as frases formam um conjunto enumerável e finitamente gerado, a lógica diz-se **formal** e as frases designam-se por **fórmulas**.

Quando frases (mesmo "informais") se agrupam num conjunto a que se associa uma identidade própria, esse conjunto designa-se por **conjunção**. Frases e conjunções têm o nome genérico de **asserções**.

A lógica tem por objetivo descrever duas noções fundamentais que associamos às asserções, nomeadamente a conjunções: a noção de **validade** e a noção de **inconsistência.**

Estas noções nunca são independentes e existem uma relações estruturais que as ligam. Nomeadamente

1. A conjunção vazia é sempre válida,
2. Se a linguagem for **não trivial** então a conjunção de todas as suas frases é inconsistente.
3. **Se** a linguagem contiver **negação então** a o juízo

> *"se* $\{A_1, \cdots, A_\kappa\}$ *é uma conjunção válida então* $\{B_1, \cdots, B_n\}$ *é uma conjunção válida"*
> é equivalente ao juízo
> *"para todo* $i = 1..n$ *a conjunção* $\{A_1, \cdots, A_k, \neg B_i\}$ *é inconsistente"*

Este ou outro conjunto de regras lidam com **juízos** sobre o significado das asserções. Assim são juízos construções como

- <conjunção> **é válida**
- <conjunção> **é inconsistente**
- **se** <juízo> **então** <juízo>
- **absurdo**

A construção **se** J **então** C pode ser interpretada de várias formas ; por exemplo

- sempre que J ocorre também ocorre C $\rightarrow$ juízos interpretados como eventos observáveis
- J é causa suficiente para C $\rightarrow$ juízos como causa-efeito
- J é evidência que assegura C $\rightarrow$ visão deontológica dos juízos

Ambas as noções (validade e inconsistência) podem ser definidas de forma sintática e de forma semântica.

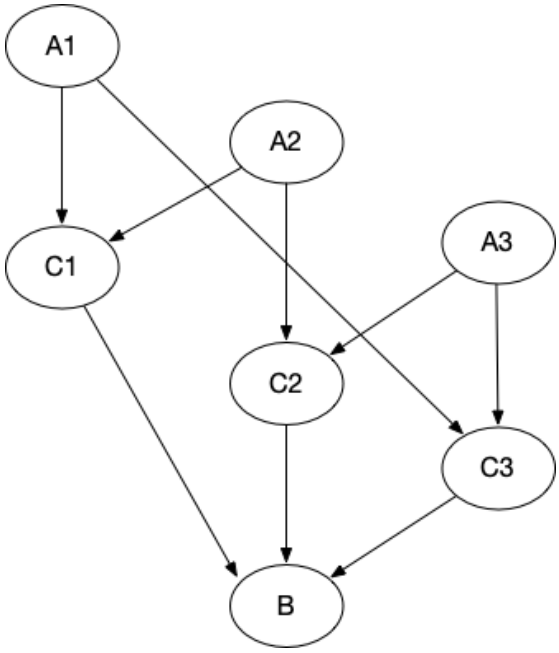
Contexto Sintático (Teoria da Prova)

No **contexto sintático** o objetivo de uma lógica formal L concretiza-se através de uma **relação de inferência** $\vdash$ determinada por

- um conjunto de **axiomas** que são fórmulas válidas por definição, e
- por um conjunto de **regras de inferência** que constroem uma nova asserção válida, a **conclusão** da regra, a partir de outras asserções, ditas as **hipóteses** da regra.

As *hipóteses* também se designam por **assunções**. *No contexto sintático a noção de validade é determinado por* ***provas.***

Sintaticamente uma prova é um grafo orientado, acíclico, com raíz ("rooted acyclic diagram") que tem fórmulas como nodos. A raíz do diagrama designa-se por **conclusão** da prova.



Prova com asserções/axiomas A_1, A_2, A_3 e conclusão B

A relação de inferência ocorre numa prova bem construída de duas formas:

- As "sources" do diagrama (nodos sem antecedentes) são **axiomas** da lógica, e podem ocorrer em qualquer prova, ou então são **específicos** de uma determinada prova e designam-se por **assunções** dessa prova.
- Os restantes nodos da prova são construídos por aplicação das **regras de inferência**: um nodo pertence à prova se é conclusão de uma regra de inferência cujas hipóteses pertencem à prova.

Se existe uma prova bem construída com assunções $A_1, \cdots, A_\kappa$ e conclusão B, escreve-se

$$A_1, \cdots, A_\kappa \vdash B$$

No contexto sintático temos

- *Uma fórmula é **válida** (ou **teorema)** se e só se é um axioma ou então é a conclusão de uma prova cujas assunções são válidas.*
- Uma conjunção Γ é **válida** se todas as fórmulas $A \in \Gamma$ são válidas
- Uma conjunção Γ é **inconsistente** se e só se , para toda a fórmula $C \in \mathrm{L}$, existe uma prova bem construída $\Gamma \vdash C$.

Note-se que, sendo Γ inconsistente, consegue-se provar ambas as inferências $\Gamma \vdash C$ e $\Gamma \vdash \neg C$; isto é, prova-se a partir das assunções em Γ que qualquer fórmula C e a sua negação $\neg C$ são conclusões bem construídas.

No contexto sintático uma **teoria** é um sub-conjunto da lógica $\mathrm{T} \subseteq \mathrm{L}$ que é fechado pelo relação de inferência; isto é, se as assunções $A_1, \cdots, A_\kappa$ são elementos da teoria T e se inferência anterior se

verifica então a conclusão B também é elemento dessa teoria T .

Exemplo
As fórmulas da Lógica Proposicional são geradas por um conjunto de símbolos (ou variáveis) proposicionais P , uma constante $\perp$ (absurdo ou Falso) e a conetiva binária $\rightarrow$ (implicação).

Esta sintaxe mínima é complementada com um conjunto de abreviaturas; nomeadamente

$$\begin{aligned} \neg A &\equiv A \rightarrow \perp \\ A \vee B &\equiv \neg A \rightarrow B \\ A \wedge B &\equiv \neg(A \rightarrow \neg B) \end{aligned}$$

Com estas fórmulas podem ser definidas várias Lógicas Proposicionais dependendo da forma como se define a relação de inferência $\vdash$; nomeadamente do conjunto de axiomas e regras de inferência.

Por exemplo, chamada *Lógica Proposicional Intuicionista* tem uma relação de inferência definida por três axiomas:

- $\vdash A \rightarrow (B \rightarrow A)$
- $\vdash (A \rightarrow B) \rightarrow (A \rightarrow (B \rightarrow C)) \rightarrow (A \rightarrow C)$
- $\vdash \perp \rightarrow A$ *Ex Falso Quodlibet*

e uma regra de inferência

- $A , A \rightarrow B \vdash B$ *Modus Ponens*

para quaisquer fórmulas A, B, C.

Um outro exemplo é a *Lógica Proposicional Clássica* cuja relação de inferência acrescenta, aos da Lógica Intuicionista, um quarto axioma.

- $\vdash (\neg A \rightarrow A) \rightarrow A$ *Reductio Ad Absurdum*

Um terceiro exemplo é a *Lógica Proposicional Mínima* que se obtém da intuicionista removendo o axioma "ex falso quodlibet".

Contexto Semântico (Teoria dos Modelos)

Provas determinam validade das fórmulas de uma forma meramente sintática. A **semântica** da lógica, porém, determina a validade das fórmulas associando-as estruturas matemáticas concretas: o "mundo real".

Formalmente a semântica é definida por uma estrutura algébrica dada por um domínio W e um conjunto de relações sobre esse domínio. Sendo n o número de variáveis usadas numa determinada lógica L , um elemento $w \in W^n$ designa-se por **mundo** para a interpretação de L .

Uma **interpretação** de L é uma função π que associa cada fórmula $A(x_1, \cdots, x_n)$, com n variáveis livres, uma relação $\pi_A \subseteq W^n$.

Para um qualquer $w \in W^n$, escreve-se $w \models A$ para indicar que a fórmula A é **verdadeira** no mundo w. Note-se que $w \models A$ é apenas uma outra forma de escrever

$$w \in \pi_A$$

interpretando π_A como um conjunto de mundos. Os mundos w nos quais A é verdadeiro, designam-se por **modelos** de A.

Alguma notação adicional relativo a conjuntos de fórmulas $\Gamma \equiv \{A_1, \cdots, A_\kappa\}$

- Escreve-se $w \models \Gamma$ quando w valida toda a fórmula $A \in \Gamma$; ou seja
  $$\forall\, A \in \Gamma \;\cdot\; w \models A.$$
  Os *modelos* de Γ, i.e. o conjuntos dos mundos onde todos os $A \in \Gamma$ são verdadeiros, coincide com a intersecção dos conjuntos dos modelos dos vários A's em Γ.
  $$\pi_\Gamma \;\equiv\; \bigcap_{A \in \Gamma} \pi_A$$

- Escreve-se $\Gamma \models B$ quando, para todo ω, se $\omega \models \Gamma$ então também $\omega \models B$.

- Γ é **inconsistente** ou **falso** se nenhum mundo ω valida Γ. Ao invés, Γ é **satisfazível** se existe um mundo ω que valida Γ.

- Γ é **tautológico** ou **verdadeiro** se todo o mundo ω valida Γ. Ao invés, Γ é **refutável** se existe um mundo ω que não valida Γ.

Exemplo
A interpretação das Lógicas Proposicionais usa como "mundo real" o conjunto dos valores booleanos $\{0, 1\}$ equipado como a soma $\oplus$ (**xor**) e a multiplicação $\cdot$ (**and**).
A estrutura algébrica assim definida é um *corpo ("field")*; mais concretamente é um *corpo finito* representado por F_2.
Exemplos

- A interpretação do símbolo $\bot$ é a relação vazia $\{\}$.
  i.e, $a \models \bot$ se e só se $a \in \{\}$
- A interpretação da conetiva implicação $\rightarrow$ é a relação binária
  $$I \;\equiv\; \{(a, b) \in \mathrm{F}_2 \times \mathrm{F}_2 \mid a = a \cdot b\}$$
  i.e. se $a \models A$ e $b \models B$ então $(a, b) \models A \rightarrow B$ se e só se $(a, b) \in I$.

Correção, Completude e Decidibilidade

Uma vez que numa Lógica Formal surgem duas abordagens distintas às noções de validade e inconsistência (a sintática e a semântica) faz sentido relacionar estes dois conceitos.
Recorde-se que a validade sintática é apenas uma relação entre conjuntos de fórmulas, mas a validade semântica é uma relação entre fórmulas e estruturas matemáticas num "mundo real".

A **correção** de uma lógica L resume-se na frase: *"tudo o que é inferível é verdadeiro"*. Formalmente, para todo o conjunto de hipóteses Γ e conclusão A, tem-se

$$\Gamma \vdash A \quad \text{implica} \quad \Gamma \models A$$

Informalmente a correção exprime a confiança que se tem no mecanismo de inferência: se for possível construir uma prova então aquilo que se provou é garantidamente verdadeiro.

A **completude** é a propriedade que se resume na afirmação: *"tudo o que é verdadeiro pode ser provado"*. Formalmente

$$\Gamma \models A \quad \text{implica} \quad \Gamma \vdash A$$

Informalmente esta propriedade garante que o mecanismo de prova é suficientemente poderoso para conseguir provar como verdadeiro tudo que realmente é verdadeiro.

Para provar que uma lógica é correta é necessário provar:

- que a interpretação de qualquer axioma é a relação total
- que, em cada regra de inferência, com a assunção da validade semântica das hipóteses conclui-se a validade semântica da conclusão.

Nas Lógicas Proposicionais é simples usar este princípio para concluir a correção.

Já a completude é algo bastante mais difícil de verificar e mesmo a análise de lógicas simples como a proposicional têm provas de completude que saem fora do âmbito introdutório deste capítulo.

A verificação da completude e da correção assenta na capacidade de o verificador construir provas $\Gamma \vdash A$ ou modelos $\Gamma \models A$.
Dado que tanto provas como modelos são, em princípio, objectos de domínios enumeráveis, é necessário saber se, a partir de descrições do par (Γ, A), é possível computar a prova, o modelo ou ambos.
Mesmo que a computação directa não seja possível é indispensável saber decidir, computavelmente, se os objetos prova e modelo existem.

Estas questões, computáveis por natureza, estão na base da construção de ferramentas computacionais de apoio à lógica. Por isso estão na base dos objectivos desta disciplina.
A noção fundamental que domina todas estas questões (nomeadamente a viabilidade dos objectivos da disciplina) é a de **decidibilidade**; por isso vamos em seguida falar um pouco dela.

Só para lógicas particularmente simples, como a Lógica Proposicional (PROP), é possível computar diretamente provas ou modelos a partir do par assunções-conclusão (Γ, A). Por isso em lógicas de 1ª Ordem (FOL) ou em Aritmética é necessário usar uma forma indirecta de obter informação computacional sobre essas entidades. Surge assim a noção de **decidibilidade.**

Numa lógica L uma **teoria** é uma sub-linguagem $\mathrm{T} \subseteq \mathrm{L}$ fechada por inferência. Isto é, se tivermos $\Gamma \vdash A$, para algum conjunto de hipóteses $\Gamma \subseteq \mathrm{T}$, então também se verifica $A \in \mathrm{T}$.

Uma teoria $\mathrm{T} \subseteq \mathrm{L}$ diz-se **decidível** (aka **computável)** se existe um algoritmo específico de T (dito de decisão) que, sob input de uma fórmula arbitrária $A \in \mathrm{L}$, termina sempre e termina com a mensagem *sucesso quando* $A \in \mathrm{T}$ e com a mensagem *falha* quando $A \notin \mathrm{T}$.

A teoria T é **semi-decidível** (aka **recursivamente enumerável**) quando, sob um input arbitrário $A \in \mathrm{L}$, caso seja $A \in \mathrm{T}$ então o algoritmo de decisão termina com mensagem *sucesso;* porém, quando $A \notin \mathrm{T}$, o algoritmo de decisão pode ou não terminar; se terminar então termina com a mensagem *falha*.

Para a construção computável de provas ou modelos coloca-se ainda a questão de saber se uma teoria T é **finitamente axiomatisável.**
Uma teoria é definida com uma relação de inferência $\vdash$ que está nela implícita. Aliás as noções de correção e completude exigem que exista essa relação. Se a relação puder ser definida com um número finito de axiomas e de regras de inferência não-constantes (i.e. contendo símbolos de variáveis) a teoria diz-se finitamente axiomatisável.

No caso de uma teoria $\mathrm{T} \subseteq \mathrm{L}$ ser decidível e a lógica L ser finitamente axiomatisável, a relação de inferência pode ser implementada por uma algoritmo que, sob input de um conjunto finito de assunções $A_1, \cdots, A_\kappa \in \mathrm{T}$ e de uma eventual conclusão $B \in \mathrm{T}$, termina com a mensagem *sucesso* caso se verifique $A_1, \cdots A_\kappa \vdash B$ ou com a mensagem *falha* caso essa inferência não se verifique.

Note-se que em qualquer lógica formal L cada axioma e cada regra de inferência, a menos que sejam constantes, contêm um número finito de símbolos que podem ser instanciados com qualquer fórmula em L.
Dado que existe um número contável de fórmulas em L, cada axioma ou regra tem um número contável de instâncias diferentes.
Se o número de axiomas e regras não fosse finito (mesmo que fosse contável) o número total de instâncias não seria contável; isto porque N^{N} não é contável já que tem a cardinalidade do contínuo. Isto entraria em contradição com a definição de lógica formal que exige um número total de fórmulas que seja contável. Portanto o número de axiomas e regras não-constantes (contendo símbolos) tem de ser finito.

Por estes motivos ferramentas do tipo "model checker" ou "solver" só funcionam corretamente em teorias decidíveis e finitamente axiomatisáveis. No entanto porque existem teorias que, não sendo decidíveis ou finitamente axiomatisáveis, são importantes como descrições de problemas reais (por exemplo, a Aritmética), está-se constantemente à procura de **fragmentos decidíveis** dessas teorias. Em grande medida este é um dos objectivos da Teoria dos Modelos.

Ao longo deste curso as propriedades de correção, completude, decidibilidade e axiomatisação finita são cruciais para o desenvolvimento e aplicação apropriada das ferramentas computacionais que permitem resolver os "grandes" problemas modelados em lógica.

As Lógicas Proposicionais (PROP) são lógicas completas e corretas e em que qualquer teoria é decidível e finitamente axiomatisável. Isto traz imensas vantagens na construção de ferramentas e é por isso que estas lógicas são a base deste curso.

Quando se passa para a Lógica de 1ª Ordem (FOL) nomeadamente para a Aritmética, já não é possível fazer a mesma afirmação. Godel provou, nos anos 30, a indecidibilidade da Aritmética e esse facto teve consequências quase catastróficas para a Lógica da altura. De certa forma pode-se afirmar que toda a Ciência da Computação nasceu da forma como Turing interpretou este resultado.

Algoritmos de Satisfação Proposicional

A Lógica Proposicional tem uma descrição muito simples mas os seus algoritmos principais, nomeadamente o algoritmo de SAT, exigem representações específicas para poderem ser razoavelmente eficientes.

Nesses algoritmos são usadas, essencialmente, dois tipos de representações ou estruturas de dados : **formas normais** e **diagramas de decisão binária (bdd's)**.

Os algorimos Eval e SAT usados nas funções booleanas exigiam que cada função fosse apresentadas com um conjunto de bit-strings de tamanho igual ao seu número de variáveis. Cada bit-string nesse conjunto descreve um dos monómio representando os expoentes das várias variáveis; a soma dos monómios está implícita na representação do conjunto de strings.
Os algoritmos ignoram o significado lógico desses elementos e limitam-se a manipular as componentes dessa estrutura de dados.

É esta mesma abordagem que se usa nas fórmulas proposicionais: tanto as formas normais com as bdd's são estruturas de dados e os diversos algoritmos ignoram a interpretação da estrutura e apenas se limitam a manipular as suas componentes.

Formas Normais

Estruturalmente as formas normais são simples conjuntos de conjuntos de **literais**.

Um **literal** é um símbolo proposicional x ou a sua negação $\neg x$. Os conjuntos de literais designam-se por **cláusulas** e os conjuntos de cláusulas são as **formas normais**.

Exemplo
A seguinte estrutura

$$\{\ \{\neg x_1, x_2, \neg x_3, x_4\}\ ,\ \{x_1, \neg x_2, \neg x_3, \neg x_4\}\ \}$$

é uma forma formal formada por duas cláusulas com 4 literais cada.

Por si só uma forma normal não tem qualquer significado lógico: é apenas a estrutura de dados que os vários algoritmos manipulam. Para estabelecer uma ligaçãoà Lógica Proposicional é preciso incluir as conetivas lógicas que ligam os vários literais numa clásula e as várias cláusulas entre si.

Nas **Formas Normais Conjuntivas** ("conjunctive normal forms" ou **CNF**'s), as clásulas são interpretadas como disjunções de literais e a CNF é interpretada como a conjunção das interpretações das cláusulas.

Como CNF, o exemplo acima é interpretado como a fórmula

$$(\neg x_1 \vee x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4)$$

As **Formas Normais Disjuntivas (DNF**'s**)** têm uma interpretação dual da anterior: as clásulas são interpretadas como conjunções de literais e a DNF é interpretada como a disjunções das interpretações das clásulas.

Como DNF, o exemplo anterior é interpretado como a fórmula

$$(\neg x_1 \wedge x_2 \wedge \neg x_3 \wedge x_4) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4)$$

Toda a fórmula proposicional pode ser representada tanto como CNF como DNF. O algorimo de construção da forma normal é relativamente fácil de descrever: percorre recursivamente a árvore sintática da fórmula. No entanto é fácil verificar que a sua complexidade no pior caso é exponencial. Nomeadamente tem complexidade exponencial no pior caso o algoritmo que a partir da CNF que descreve um fórmula φ constrói a DNF que descreve a mesma fórmula.

O algorimo **SAT**, conhecido por **algoritmo de Davis-Putman**, é o algoritmo base a partir do qual todas as melhorias de eficiência têm sido desenvolvidas. O algoritmo segue uma estratégia conhecida por **split-and-bound,** que já foi usada nas funções booleanas e vai ser usada noutras situações como veremos em seguida.

SAT

OBJETIVO dada uma forma normal conjuntiva F , pretende-se verificar se é insatisfazível (**unsat**).

ALGORITMO

1. Simplifica-se F, eliminando cláusulas redundantes e substituindo as restantes por cláusulas equivalentes com menos literais.
2. Deteta-se na forma simplificada uma eventual solução trivial do problema. Se existir o algoritmo termina e devolve essa solução. Se não existir, prossegue com os restantes passos.
3. Escolhe-se uma variável x segundo um critério adequado e procede-se ao **split** de F em duas outras formas $F_0 \equiv \neg x \wedge F$ e $F_1 \equiv x \wedge F$. Recursivamente aplica-se a F_0 e F_1 este mesmo algoritmo.
4. Combina-se o resultado das duas chamadas recursivas (**bound**) no resultado de F.

São essenciais as formas como se concretiza cada um destes passos e como essa concretização se relaciona com a estrutura das formas normais. O passo da simplificação é onde se concentra grande parte do esforço: é necessário detetar cláusulas redundantes e remover ou simplificar cláusulas cuja informação seja nula ou já esteja contida na informação de outras cláusulas.

Neste momento do nosso curso não é muito relevante discutir estes detalhes que por si só poderiam ocupar toda a disciplina.

Diagramas de Decisão Binária

Os Diagramas de Decisão Binária ("binary decision diagrams" ou **BDD**'s) são uma alternativa representação de fórmulas proposicionais. As BDD's são estruturas de dados compactas com capacidade para gerar algoritmos que manipulam fórmulas proposicionais de grandes dimensões de forma muito eficiente. De facto BDD's manipulam eficientemente fórmulas com centenas de milhar de variáveis.

No que se segue vamos considerar sempre fórmulas com n variáveis livres enumeradas como $x_1, x_2, \cdots, x_n$. No conjunto das variáveis está definida uma ordem crescente com a enumeração: i.e. $x_i < x_j$ se e só se $i < j$. Na terminologia das BDD's diz-se que os nossos diagramas são sempre **ordenados**.

Qualquer fórmula proposicional f, com a variável livre x, pode-se sempre escrever como

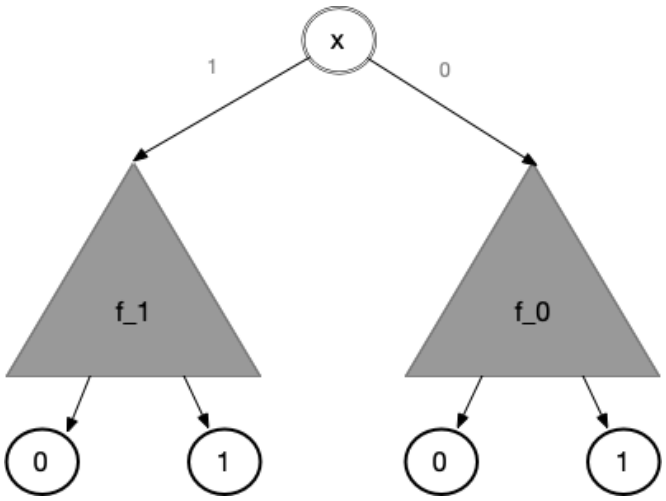
$$f \equiv (x \wedge f_{x\leftarrow 1}) \vee (\neg x \wedge f_{x\leftarrow 0})$$

Obviamente, f é **independe de** x se e só se $f_{x\leftarrow 0} \equiv f_{x\leftarrow 1} \equiv f$.

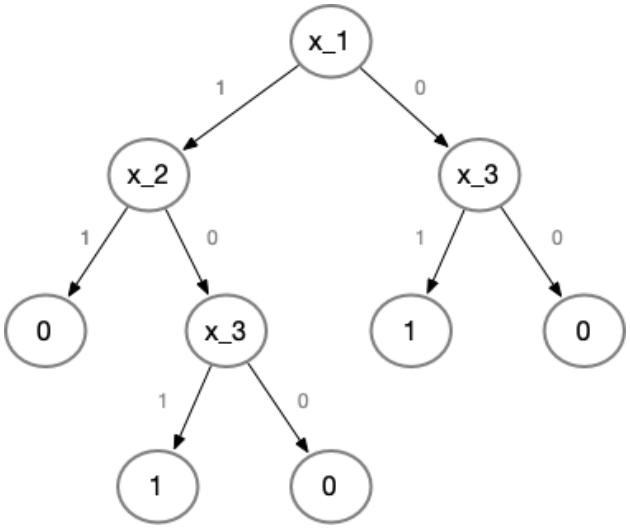
Esta é a origem da descrição da fórmula f como uma **árvore de decisão** ("decision tree" ou **DT**).

Os nodos de uma árvore de decisão são sempre marcados com variáveis ou então com as constantes 1 e 0. De forma a sistematizar as regras de construção de árvore vamos considerar 1 como um símbolo proposicional que, ao contrário das outras variáveis tem um valor lógico fixo, e vamos considerar 0 como a negação de 1. Vamos também considerar que 1 é supremum dos símbolos proposcionais na ordem que definimos nas variáveis.

A árvore que descreve f é uma árvore binária que tem, como folhas, nodos marcados com símbolos 1 e 0 (como abreviatura de $\neg 1$); a sub-árvore esquerda é a árvore que representa $f_{x\leftarrow 1}$ e a sub-árvore direita é a árvore que descreve $f_{x\leftarrow 0}$.



Por exemplo a árvore de decisão na figura seguinte



descreve a seguinte fórmula

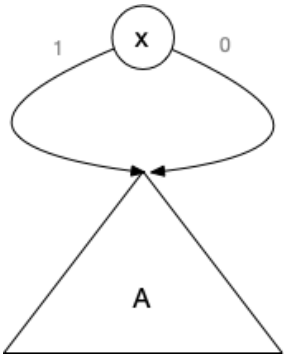
$$(x_1 \wedge x_0 \wedge 0) \vee (x_1 \wedge \neg x_2 \wedge x_3 \wedge 1) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge 0) \vee (\neg x_1 \wedge x_3 \wedge 1) \vee (\neg x_1 \wedge \neg x_3 \wedge 0)$$

Tomando 1 como um símbolo proposicional e 0 como a sua negação, esta fórmula é uma DNF.
Note-se que cada cláusula da DNF representa um caminho ordenado na árvore de decisão (1 é o supremum na ordem dos símbolos proposicionais) sem repetição de símbolos.

Redução de árvores de decisão

Uma árvore assim construída não é a forma mais eficiente de descrever fórmulas preposicionais. De fato é simples verificar que a mesma fórmula pode ser descrita por diferentes árvores. Para ter uma árvore única para cada fórmula preposicional é necessário usar uma simplificação designada por **redução das árvores de decisão**.

Suponhamos que uma fórmula f tem uma árvore de decisão em que a sub-árvore esquerda é equivalente à sub-árvore direita.



Isto é equivalente a afirmar que $A \equiv f_{x\leftarrow 1} \equiv f_{x\leftarrow 0}$ e

$$f \equiv x \wedge f_{x\leftarrow 1} \vee \neg x \wedge f_{x\leftarrow 0} \equiv A$$

Portanto, sempre que aparece uma árvore de decisão em que a sub-árvore esquerda equivale logicamente à sub-árvore direita, pode-se substituir toda a árvore pela sub-árvore.

Esta é a essência do algoritmo de redução de árvores. Recursivamente percorre-se a árvore do topo para as folhas reduzindo as sub-árvores e, depois de reduzidas, verificando se são logicamente equivalentes e, se forem, substituindo toda a árvore pela sub-árvore.
O resultado é uma RBDT ("reduced binary decision tree").

O teorema fundamental é

> ***Teorema:*** *Cada fórmula proposicional é descrita por uma única árvore de decisão ordenada e reduzida.*

Como consequências temos vários corolários que nos ajudam a perceber a importância das BDD's para o manuseamento de fórmulas proposicionais.

> ***Corolário 1:*** *Duas fórmulas proposicionais são logicamente equivalentes se e só se são descritas pela mesma árvore de decisão ordenada e reduzida.*

Como caso particular deste corolário temos as situações em que uma fórmula é inconsistente ou é uma tautologia. Como fórmulas inconsistentes são logicamente equivalentes a **Falso** (ou 0) e fórmulas tautológicas são logicamente equivalentes a **Verdadeiro** (ou 1) então

> ***Corolário 2:*** *Uma fórmula f é* ***inconsistente*** *se e só se a RBDT que a descreve se reduz à folha 0 e é tautológica se e só se essa árvore se reduz à folha 1.*

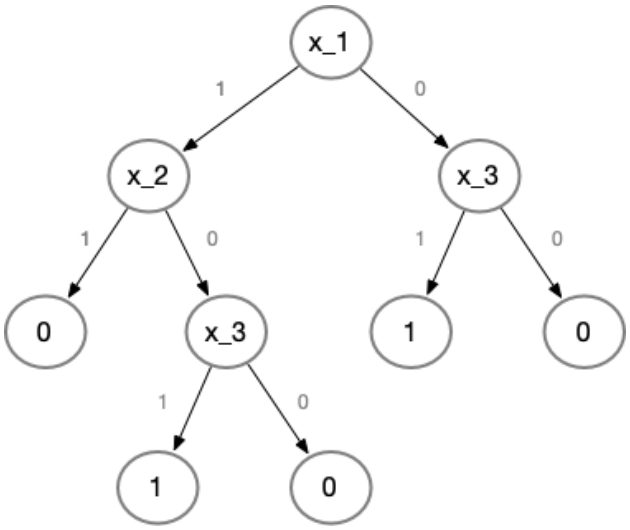
Este corolário mostra como o algoritmo de redução implementa um algoritmo de SAT.

Uma observação que é possível fazer é que a estruturas de dados usadas para descrever árvore binárias não são as mais adequadas para descrever RBDT´s.

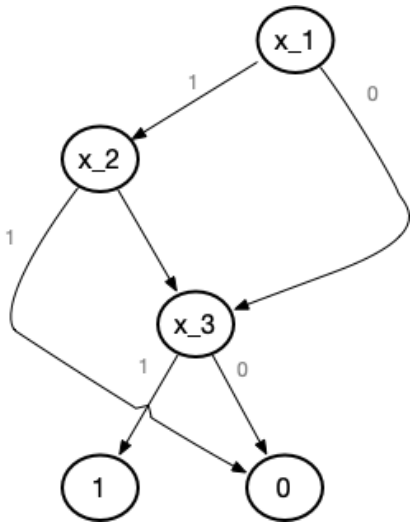
Isto porque, frequentemente, uma RBDT tem sub-árvores repetidas. A repetição de sub-árvores não só ocupa espaço desnecessário como, e principalmente, força os algoritmos que percorrem as árvores a

computações repetidas e desnecessárias.

Considere-se de novo a árvore que nos serviu de exemplo inicial.



Nesta árvore repete-se a sub-árvore de raíz x_3 e também as folhas 0 e 1. Para economizar espaço e tempo pode-se transformar a árvore num diagrama substituindo as entidades repetidas por referências à cópia original.
Desta forma obtém-se o diagrama, que a seguir se mostra, onde se passa de 9 para 5 nodos e 8 para 6 referências



O resultado designa-se por **diagrama de decisão binária** ("binary decision diagram" ou BDD).
Existem muitas formas de eficientes de implementar BDD's tanto em "software" como em "hardware" mas quase todas se baseiam na noção de *dicionário* que nos é familiar da linguagem Python.
O dicionário guarda todos os diagramas do problema. Cada diagrama é representado por uma chave (por exemplo, o hash da própria estrutura) a que está associado uma folha ou então um triplo definido por

uma variável e as duas chaves que identificam as sub-árvores já guardadas no dicionário.

Com uma estrutura deste tipo é muito eficiente construir diagramas que representem conjunções $f \wedge g$ e disjunções $f \vee g$ de fórmulas cujos diagramas estão já guardados no dicionário.

Calcular o diagrama de $\neg f$ a partir do diagrama de f é bastante mais difícil. Porém a maioria dos algoritmos armazenam um bit extra na raíz para indicar se o diagrama é "positivo" , i.e. representa a fórmula f, ou é "negativo" representando a fórmula $\neg f$.

Como este não é o objetivo do curso não vamos aqui fornecer detalhes desses algoritmos, que estão disponíveis na documentação complementar desta disciplina.

É preciso ter em conta que apesar dos algoritmos mencionados serem eficientes, um algoritmo genérico que receba como "input" uma fórmula proposicional f (por exemplo, como CNF) e construa a respetiva RBDT tem complexidade exponencial, no pior caso.

Por isso tentar usar BDD's para implementar um algoritmo de que tenha uma CNF como "input" , aproveitando o facto do algoritmo de SAT nas BDD's ter complexidade polinomial, continua a ter complexidade exponencial (no pior caso) porque a conversão $\mathrm{CNF} \rightarrow \mathrm{RBDT}$ tem complexidade exponencial.

Normalmente as BDD's são usadas quando é possível criá-las a partir de outras BDD's mais simples ou então diretamente a partir da descrição do problema que se pretende resolver. Esta é a estratégia que veremos quando se estuda "Model Checking" em sistemas dinâmicos.

A estratégia usada para modelar, usando refinamento, um problema qualquer P usando formas normais difere substancialmente da estratégia usada para modelar o mesmo problema usando BDD's.

Suponhamos que o problema P decompõe-se em sub-problemas $\langle \mathrm{P}_1, \cdots, \mathrm{P}_n \rangle$ e se pretende construir o modelo de P , na forma de uma fórmula $\phi(\mathrm{P})$ que se submete como "input" a um algoritmo SAT.

Usando formas normais a estratégia seria essencialmente formada pelos seguinte passos:

a. Usando esta estratégia do refinamento ou partindo de soluções atómicos, constrói-se os modelos $\phi(\mathrm{P}_i)$, dos diferentes sub-problemas, sob a forma de proposições.
b. A forma como os sub-problemas compõem para construir P vai permitir construir a fórmula $\phi(\mathrm{P})$. Isto é teremos um algoritmo $\phi(\mathrm{P}_1), \cdots, \phi(\mathrm{P}_n) \rightsquigarrow \phi(\mathrm{P})$.
   A fórmula $\phi(\mathrm{P})$ tem uma forma genérica que depende deste algoritmo. Normalmente <u>não</u> é uma forma normal.
c. Converte-se $\phi(\mathrm{P})$ para uma forma normal (CNF ou DNF) e submete-se o resultado ao algoritmo SAT.

A dificuldade desta abordagem está no ultimo passo: o algoritmo de conversão de uma proposição genérica numa forma normal tem, no pior caso, complexidade equivalente à do próprio algoritmo SAT: isto é, potencialmente tem tempo exponencial.

Em alternativa, a estratégia usando BDD's tem os seguintes passos:

a. Usando esta estratégia do refinamento ou partindo de sub-problemas atómicos, constrói-se modelos $\mathrm{bdd}(\mathrm{P}_i)$ dos diferentes sub-problemas sob a forma de diagramas de decisão binária orientados e reduzidos.
b. A forma como os sub-problemas compõem para construir P pode permitir construir $\mathrm{bdd}(\mathrm{P})$. Ou seja, pode existir um algoritmo $\mathrm{bdd}(\mathrm{P}_1), \cdots, \mathrm{bdd}(\mathrm{P}_n) \rightsquigarrow \mathrm{bdd}(\mathrm{P})$ que construa eficientemente o diagrama orientado e reduzido que modela o problema P a partir de modelos similares dos sub-problemas P_i.
c. Como o modelo $\mathrm{bdd}(\mathrm{P})$ já é um diagrama reduzido, a simples análise do diagrama permite conconcluir se é insatisfazíviel ou tautológico em tempo constante.

A dificuldade desta segunda abordagem está no 2º passo: pode ser que exista o algoritmo aì referido mas também pode ser que não exista tal algoritmo. Ao invés nas formas normais o 2º passo é quase sempre mais simples de realizar.

Aritmética Linear Inteira

Para além das Lógicas Proposicionais este curso necessita de lógicas que lidam com inteiros. Necessitamos da Aritmética que permite usar as operações básicas sobre inteiros, constantes 0 e 1, somas $+$, multiplicações $\times$, as relações de igualdade $=$ e de ordem $<$ e funções com elas construídas.

Nomeadamente a **Aritmética de Peano** (**AP**) é uma lógica capaz de descrever todas estas funções e relações e analisar as suas propriedades.

A **AP** é uma linguagem poderosa que usa quantificadores $\exists$ e $\forall$ sobre os objetos da lógica (os inteiros) mas também, como no *princípio da indução*, sobre o próprios predicados. É o princípio da indução que permite definir funções recursivas sem as quais não seria possível definir as funções básicas da aritmética (somas e multiplicações inclusive).

Por isso a AP é uma linguagem de 2ª ordem sobre um domínio infinito (os inteiros) e, portanto, não é nem completa nem decidível. Também, devido à presença do princípio da indução, não é finitamente axiomatizável.

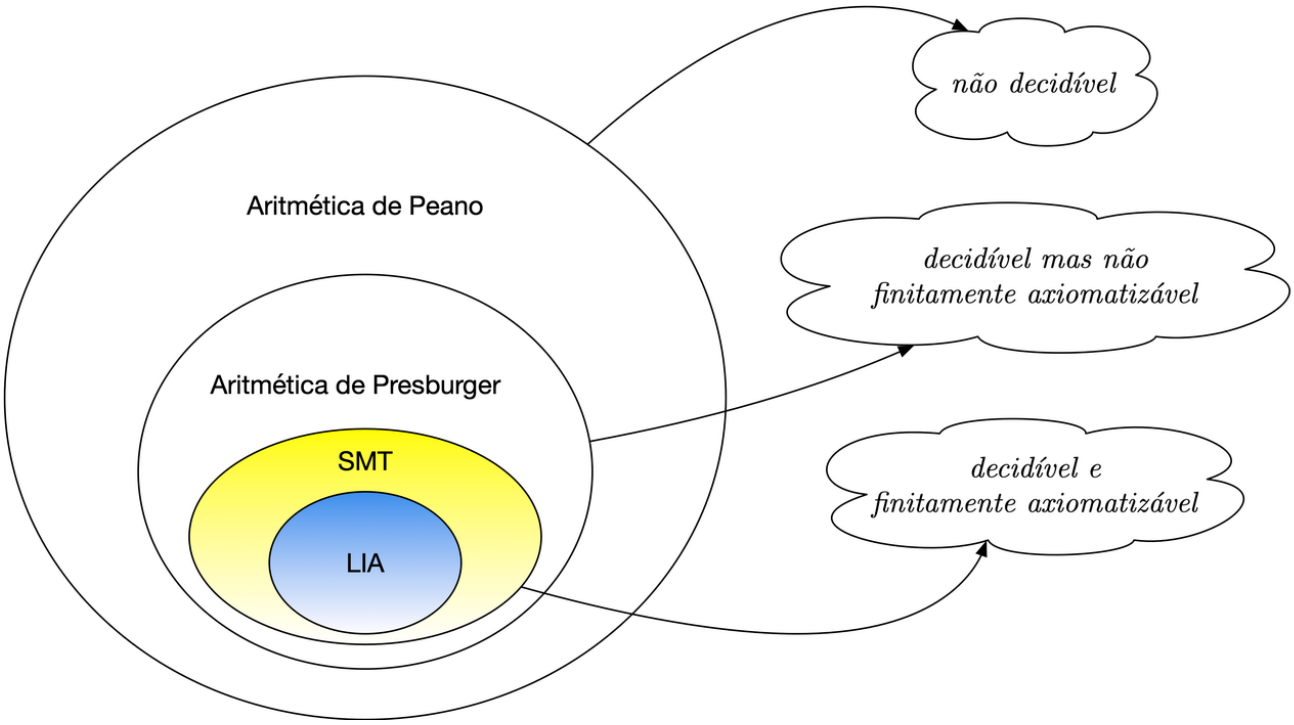
Assim é impossível construir um algoritmo que implemente completamente a inferência da **AP** .

O uso de ferramentas computacionais na Aritmética implica que elas só sejam válidas em sub-teorias decidíveis. Tais teorias designam-se por **fragmentos decidíveis** da Aritmética.

A **LIA** ("linear integer arithmetic") é um tal fragmento relevante não só aos problemas de Programação Inteira mas também às aplicações dos SMT's que são o principal objeto deste curso.

Esta lógica deriva de um exemplo importante de um fragmento decidível da **AP** designado por **aritmética de Presburger**. Esta é uma lógica com igualdade $=$, as constantes 0 e 1 e somas $+$ mas não multiplicações genéricas. Na axiomatização inclui o princípio da indução para qualquer predicado de 1ª ordem.

A aritmética de Presburger é correcta, completa e decidível. No entanto, devido à presença do princípio da indução, não pode ser definida por um conjunto finito de axiomas de 1ª ordem e, por isso, não existe nenhum algoritmo finito que implemente a relação de inferência.



A **LIA** assim como as **SMT** contornam esta questão não usando o princípio da indução mas introduzindo uma axiomatização finita do que é uma relação, ou genericamente um predicado, em cada teoria. Desta forma já é possível definir algoritmos da SAT e de inferência específicos de cada teoria.

A **LIA** é uma lógica com igualdade cujos objetos são as constante 0 e 1, as somas $a + b$, o simétrico aditivo $-a$, a operador binário $<$ (uma ordem total) e o operador ternário **ite**. O objeto $\mathbf{ite}(a, b, c)$ interpreta-se como

$$\text{if } a \text{ then } b \text{ else } c$$

Nesta lógica todo $a \neq 0$ é associado ao valor lógico True e 0 é associado ao valor lógico False.

Note-se que esta aritmética tem somas e subtrações mas não tem multiplicações arbitrárias $a \times b$.
Também não tem a relação de divisibilidade $a \mid b$ (a divide b).
Tem porém uma forma de multiplicação $a \cdot c$, designada por *multiplicação escalar*, cujos argumentos são um objeto arbitrário a e uma constante c. A multiplicação escalar é apenas uma abreviaturas de uma soma

$$a \cdot c \equiv \begin{cases} a + a + \cdots + a \quad (c \text{ vezes}) & \text{se } c > 0 \\ -(a \cdot (-c)) & \text{se } c < 0 \end{cases}$$

Sintaxe

Os *predicados,* designadas por "inequações lineares inteiras" com n variáveis livres $x_1, \cdots, x_n$, têm a forma.

$$\phi \equiv x_1 \cdot c_1 + x_2 \cdot c_2 + \cdots + x_n \cdot c_n \leq c_0$$

sendo $c_0, c_1, c_2, \cdots, c_n$ constantes inteiras. O vetor $\overline{c} = (c_0, c_1, \cdots, c_n)$ formado por estas $n+1$ constantes determina completamente o predicado.

Os predicados substituem na **LIA** os símbolos proposicionais da Lógica Proposicional. Assim

- *literais* são predicados ou negações de predicados
- *cláusulas* são conjuntos de literais interpretados como conjunções ou, alternativamente, como disjunções.
- *fórmulas* são conjuntos de cláusulas interpretados como conjunções ou como disjunções tal como acontece nas formas normais da Lógica Proposicional.

Semântica

Dependendo do tipo de problemas que se pretende resolver (veremos adiante quais) a semântica desta lógica vai ser definida, neste curso, em três domínios distintos:

- O domínio dos inteiros positivos N.
- Os inteiros $\{0, 1\}$.
- Os racionais positivos Q_+.

Para cada um destes domínios D, um predicado φ, com n variáveis livres, vai ser interpretado num mundo $w \in D^n$ de forma a que

> $w \models \varphi$ *tem como valor lógico do objecto* $\varphi(w)$ *que se obtém substituindo as variáveis livres pelas componentes de* w.

No caso do dominio Q_+ a substituição é um pouco mais técnica porque a LIA só lida com inteiros. Por isso é necessário escrever os racionais r como pares de inteiros $r \equiv a/b$.

A interpretação das negações, conjunções e disjunções é a usual:

- $w \models \neg\varphi$ sse $w \not\models \varphi$
- $w \models \phi \wedge \varphi$ sse $w \models \phi$ e $w \models \varphi$
- $w \models \phi \vee \varphi$ sse $w \models \phi$ ou $w \models \varphi$

Problemas de Programação

A palavra "programação" não se restringe à interpretação usual de "programação de computadores". Tradicionalmente a palavra significa algo mais lato: a noção de **planeamento.** É nesse sentido que iremos usar esta palavra aqui.

O problema fundamental na lógica LIA, tal como na lógica proposicional, é a verificação de uma fórmula é inconsistente e, se não for, determinar uma **testemunha** de consistência. No contexto desta lógica, porém, o problema obedece a restrições específicas e tem uma terminologia própria:

Em primeiro lugar, o "input" do problema é sempre uma fórmula dada pela **conjunção** de predicados todos com as mesmas n variáveis livres.

$$\varphi \equiv \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_\kappa$$

Cada predicado é determinado por um vetor de $n+1$ inteiros $\overline{c}_j$, com $j = 1..\kappa$. O inteiro $\overline{c}_{j,i}$ é a i-ésima componente do j-ésimo vetor; estes inteiros definem, por isso, uma matriz

$$C \in \mathrm{Z}^{\kappa\times(n+1)}$$

que descreve completamente a fórmula φ.

Consoante a semântica usada a verificação da inconsistência de uma tal fórmula φ, pode-se definir problemas distintos

Programação Inteira (PI)

A semântica é interpretada no domínio N; para uma fórmula conjuntiva com n variáveis livres os mundos w são vetores de n componentes inteiras positivas; isto é $w \in \mathrm{N}^n$. São estas componentes que podem substituir variáveis em φ.

Verificar a inconsistência de φ é mostrar que, para todo o mundo $w \in \mathrm{N}^n$ se tem

$$w \nvDash \varphi$$

Isto é equivalente a afirmar que, para todo $w \in \mathrm{N}^n$, existe algum $j \le \kappa$ tal que $\phi_j(w) = 0$.
Um contra-exemplo da inconsistência é, por isso, um mundo w tal que $w \vDash \varphi$.

Neste modelo semântico é simples verificar que a negação $\neg\phi$, de um predicado ϕ definido pelo vetor $(c_0, c_1, \cdots, c_n)$, é o predicado definito pelo vetor $(-c_0 - 1, -c1, \cdots, -c_n)$.

Programação Inteira Binária (PIB)

A semântica é aqui interpretada no domínio $\{0,1\} \subseteq \mathrm{N}$. Por isso, para uma fórmula φ com n variáveis livres, os mundo são vetores de bits $w \in \{0,1\}^n$.

Por isso todos os mundos $w \in \{0,1\}^n$ que validam φ nesta semântica binária também validam a mesma fórmula na semântica inteira; logo um fórmula que seja consistente no problema PIB também será consistente no problema PI.

Porém o inverso não é verdadeiro; pode existir uma fórmula φ que seja inconsistente em termos de mundos $w \in \{0,1\}^n$ mas não seja inconsistente em mundos $w \in \mathrm{N}^n$; basta que nenhum mundo $w \in \mathrm{N}^n$ que verifica $w \vDash \varphi$ esteja contido em $\{0,1\}^n$.

Nesta semântica existe uma forma simples de simplificar predicados. Suponhamos que um predicado é determinado por um vetor $(c_0, c_1, \cdots, c_n)$; então, para um mundo arbitrário $w \in \{0,1\}^n$, tem-se

$$\phi(w) \equiv c_0 \geq w_1 \cdot c_1 + \cdots + w_n \cdot c_n$$

Considere-se apenas a soma do lado direito da comparação $\sigma(w) \equiv \sum_{i=1}^{n} c_i \cdot w_i$. Note-se que as constantes c_i podem ser positivas ou negativas.

O valor máximo de $\sigma(w)$ ocorre quando $w_i = 0$, se $c_i < 0$, e $w_i = 1$ se $c_i > 0$; portanto o valor máximo é $\sigma_{\max} = \sum_{c_i>0} c_i$. Do mesmo modo o valor mínimo de $\sigma(w)$ será $\sigma_{\min} \equiv \sum_{c_i<0} c_i$. Portanto

- Se $c_0 \geq \sigma_{\max}$ então $\phi(w)$ é verdadeiro para todo $w \in \{0,1\}^n$.
- Se $c_0 < \sigma_{\min}$ então $\phi(w)$ é falso para todo $w \in \{0,1\}^n$.

Numa conjunção $\varphi \equiv \phi_1 \wedge \cdots \wedge \phi_\kappa$ um predicado que é sempre verdadeiro pode ser removido da conjunção. Se existe algum predicado que é sempre falso a conjunção também é sempre falsa.

Programação Linear (PL)

A semântica da LIA é aqui descrita no domínio dos números racionais positivos Q_+. O problema PL procura determinar se existe um mundo $w \in \mathrm{Q}_+^n$ que satisfaça $\varphi(w) \neq 0$.

Note-se que, neste domínio, $\varphi(w)$ se pode escrever como um sistema de inequações lineares.

$$\begin{cases} \sum_{i=1}^{n} C_{j,i}\, w_i \leq C_{j,0} & \text{para } j = 1..\kappa \\ w_i \geq 0 & \text{para } i = 1..n \end{cases}$$

Como os racionais têm a estrutura algébrica de um corpo, nomeadamente porque neles existe a possibilidade de se efectuar divisões, é possível usar as técnicas clássicas (e.g. eliminação gaussiana) para resolver sistemas de inequações lineares. Algoritmos como Simplex, que surgiram nos finais dos anos 1940's e estão na origem dos métodos de optimização linear criados desde essa altura, usam precisamente estas técnicas.

Como consequência o problema PL tem uma solução muito eficiente: a sua complexidade é , no pior caso, polinomial com grau menor do que 3. Pelo contrário tanto PI como PIB têm, no pior caso, complexidade exponencial. De fato ambos os problemas são **NP** completos.

Algoritmos para a Programação Inteira Binária

Na programação inteira binária a semântica da LIA é definida no domínio inteiro $\{0,1\}$; por isso, uma fórmula φ genérica e com n variáveis livres tem um conjunto de modelos

$$\hat{\varphi} \equiv \{w \mid w \vDash \varphi\}$$

que é um subconjunto de $\{0,1\}^n$.

O problema PIB pretende determinar se $\hat{\phi}$ é vazio e, se não for, apresentar como "testemunha" um elemento desse conjunto.

Porque, aqui, $\hat{\phi}$ é um subconjunto de um domínito finito, ele próprio é sempre finito. É esta a característica principal que distingue a PIB dentro da PI genérica onde um $\hat{\phi}$ genérico pode eventualmente ser infinito.

Uma outra característica é a ligação à Lógica Proposicional. De fato uma fórmula propositional escrita como uma CNF pode ser sempre representada por uma fórmula conjuntiva na LIA através das seguintes regras de conversão.

1. Os valores booleanos $\{0,1\}$ são interpretados como inteiros $\{0,1\}$. Uma variável x da lógica proposicional é convertida numa variável inteira x. Um literal $\neg x$ é convertido na expressão inteira $1-x$.

2. Uma cláusula disjuntiva $\mathrm{C} \equiv \{\ell_1 \vee \ell_2 \vee \cdots \vee \ell_\kappa\}$ é convertida no predicado
$$\phi \equiv \ell_1 + \ell_2 + \cdots + \ell_\kappa \geq 1$$

3. Uma CNF dada por $F \equiv \{\mathrm{C}_1, \cdots, \mathrm{C}_r\}$ é transformada na fórmula conjuntiva
$$\varphi \equiv \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_r$$
em que ϕ_i é a representação da cláusula C_i.

Exemplo
Considere-se a CNF
$$F \equiv (x \vee \neg y \vee \neg z) \wedge (\neg x \vee z \vee w)$$
A conversão de cada uma das suas cláusulas é
- $x \vee \neg y \vee \neg z \quad \rightsquigarrow \quad x + (1-y) + (1-z) \geq 1$
- $\neg x \vee z \vee w \quad \rightsquigarrow \quad (1-x) + z + w \geq 1$

A fórmula φ que codifica F será então (após simplificações)

$$\varphi \equiv (-x + y + z \leq 1) \wedge (x - z - w \leq 0)$$

Algoritmo "split-and-bound"

O algoritmo "split-and-bound", apresentado para para o SAT de fórmulas proposicionais, pode ser adaptado diretamente ao PIB tendo em atenção apenas que, como as estruturas de dados são distintas das usadas para representar CNF's, a concretização de cada um dos 4 passos é específica da LIA. Os passoa são, no entanto, essencialmente os mesmos:

1. Simplificar a fórmula φ e detetar se a fórmula simplificada é um dos casos especiais que terminam imediatamente o algoritmo.

2. Escolhe-se uma das variáveis livres x, seguindo um critério pré-definido, e procede-se ao "split" construindo $\varphi' \equiv \varphi_{x\leftarrow 1}$ e $\varphi'' \equiv \varphi_{x\leftarrow 0}$.
3. Aplica-se recursivamente este mesmo algoritmo a φ' e a φ'' e recolhe-se os dois resultados.
4. Combina-se ("bound") os dois resultados anteriores na construção do resultado de φ.

Também não vamos aqui elaborar sobre os detalhes da concretização de cada um destes passos que, como dissemos, dependem da estrutura de dados usada para representar as fórmulas.
No entanto é importante notar que o algortimo é bi-recursivo e, por isso, a sua complexidade é sempre exponencial no pior caso.
Uma outra observação é a que este algoritmo, por muito complexo que seja, termina sempre. De facto cada "split" reduz o número de variáveis livres pelo menos uma unidade; eventualmente o argumento do algoritmo não tem variveis livres; o passo 2 não pode ser executado e termina.

Algoritmo do "plano-de-corte"

O algoritmo "split-and-bound" é recursivo e termina; o algoritmo do plano de corte é iterativo e pode não-terminar.

O algoritmo do plano-de-corte constrói iterativamente a solução de um problema da "programação linear" (PL) que aproxima o problema PIB; verifica se essa solução está no domínio das soluções PIB e, se não estiver, modifica o problema PL e repete o processo.

Note-se que o problema de programação linear resolve-se com um algoritmo muito eficiente e por isso este passo pode ser repetido muitas vezes,

Em concreto, dada uma fórmula $\varphi \equiv \phi_1 \wedge \cdots \wedge \phi_\kappa$ com n variáveis livres a aplicação da técnica do "plano-de-corte" segue os seguintes passos:

1. **inicialização**: constrói-se um problema LP definido por uma fórmula φ^* que coincide com φ.
2. **enquanto não termina**
   a. Resolve o problema LP definido por φ^* e obtém uma solução $x^* \in [0,1]^n$. Se não existir tal solução termina com a mensagem unsat.
   b. Se x^* está contido em $\{0,1\}^n$ (isto é, se todas as componentes x_i^* são inteiros $0, 1$) termina com a mensagem sat e a solução x^*.
   c. Se x^* tem componentes que não são inteiros, escolhe-se um predicado ϕ_j da fórmula φ^* tal que c^*, definido como $\sum_i c_i x_i^*$, não é inteiro. Constrói-se ϕ_j' substituindo a constante inteira c_0 por $c_0' \equiv \lfloor c^* \rfloor$.
   d. Substitui-se em φ^* o predicado ϕ_j pelo predicado ϕ_j' e repete-se o ciclo desde o início.

Note-se que, ao substituir c_0 por $c_0' \equiv \lfloor c^* \rfloor$ em ϕ_j, o vetor x^* já não é solução do novo LP. No entanto eventuais soluções do PIB inicial não sao afetadas porque, para elas, a quantidade c^* é um inteiro.